Optimierung datenintensiver Workflows: Konzepte und Realisierung eines heuristischen, regelbasierten Optimierers

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Marko Vrhovnik

aus Stuttgart

Hauptberichter:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Mitberichter:	Prof. Dr.-Ing. Wolfgang Lehner
Tag der mündlichen Prüfung:	25.07.2011

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart

2011

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter in der Abteilung Anwendersoftware des Instituts für Parallele und Verteilte Systeme der Universität Stuttgart. Die Motivation zu der behandelten Thematik entstammt meiner wissenschaftlichen Arbeit im Kooperationsprojekt II4BPEL, das im Rahmen des IBM-Programms Center for Advanced Studies zusammen mit dem IBM-Entwicklungslabor in Böblingen durchgeführt wurde.

Mein besonderer Dank gilt Herrn Professor Bernhard Mitschang für seine Unterstützung und Betreuung dieser Arbeit. Die zahlreichen Diskussionen und seine wertvollen Anregungen haben wesentlich zum Gelingen dieser Arbeit beigetragen.

Ebenso danke ich Herrn Professor Wolfgang Lehner für sein Interesse am Thema meiner Arbeit, die kritische Durchsicht und die Bereitschaft zur Übernahme des Mitberichts.

Darüber hinaus möchte ich mich beim IBM-Entwicklungslabor in Böblingen für die Unterstützung und bei allen, die am Kooperationsprojekt beteiligt waren, für die angenehme und konstruktive Zusammenarbeit bedanken, insbesondere bei Herrn Professor Bernhard Mitschang, Holger Schwarz, Sylvia Radeschütz und Tobias Kraft aus der Abteilung Anwendersoftware, Albert Maier und Oliver Suhre aus dem IBM-Entwicklungslabor Böblingen, Herrn Professor Volker Markl von der Technischen Universität Berlin sowie den Studierenden Stephan Ewen, Daniel Fischer, Sarah Hofmann, Hendrik Holzhausen, Friedrich Münsch und Peter Reimann.

Allen Mitarbeiterinnen und Mitarbeitern der Abteilung Anwendersoftware danke ich für die kollegiale und freundschaftliche Zusammenarbeit. Insbesondere gilt mein Dank Holger Schwarz, Peter Reimann und Sylvia Radeschütz für fachliche Ratschläge und Diskussionen im Verlauf der Arbeit sowie Florian Niedermann für die Geduld beim Korrekturlesen der formalen Definitionen im Anhang dieser Arbeit.

Ein herzliches Dankeschön gebührt letztlich auch meiner Familie, ohne deren Geduld, Verständnis und Unterstützung diese Arbeit sicherlich nicht zustande gekommen wäre.

Stuttgart, im Juli 2011 *Marko Vrhovnik*

Zusammenfassung

Heutzutage spielen Workflow- und Datenbankmanagementsysteme bei der Automatisierung von Geschäftsprozessen eine wichtige Rolle. Geschäftsprozesse werden in Workflows überführt, die zahlreiche Anwendungssysteme und Datenquellen integrieren und von Workflowmanagementsystemen rechnergestützt ausgeführt werden. Datenbankmanagementsysteme verwalten die von den Geschäftsprozessen referenzierten Daten. Die Effektivität des Datenzugriffs, der Datenverarbeitung und der Datenverwaltung ist entscheidend für die Leistungsfähigkeit eines Geschäftsprozesses.

Um die Modellierung datenintensiver Workflows, die große relationale Datenmengen verarbeiten, zu vereinfachen, wurden Workflowbeschreibungssprachen, wie BPEL, von führenden Herstellern von Workflow- und Datenbankmanagementsystemen um SQL-Funktionalität erweitert. Dadurch müssen Datenverarbeitungsoperationen, wie SQL-Anweisungen oder Aufrufe benutzerdefinierter Prozeduren, nicht mehr in Web-Services gekapselt werden, sondern können direkt auf der Workflowebene definiert werden. Daraus resultiert eine neue Möglichkeit der Anfrageoptimierung, die existierende Optimierungsansätze in Datenbanksystemen ergänzt: Suboptimal modellierte Datenverarbeitungsoperationen lassen sich in einer Workflowbeschreibung unter Verwendung von Restrukturierungsregeln derart transformieren, dass sie von einem Workflow- bzw. Datenbankmanagementsystem wesentlich effizienter ausgeführt werden können.

In dieser Doktorarbeit werden Konzepte zur Realisierung eines heuristischen, regelbasierten Optimierers für datenintensive Workflows vorgestellt. Der Optimierer wendet eine Regelbasis gemäß einer wohldefinierten Kontrollstrategie auf eine interne Repräsentation für datenintensive Workflows, dem sogenannten Prozessgraphenmodell (PGM), an, um die Datenverarbeitung eines datenintensiven Workflows zu optimieren.

PGM erlaubt eine effiziente und sprachunabhängige Definition und Anwendung der Restrukturierungsregeln und unterstützt somit eine Optimierung von Datenverarbeitungsoperationen, die in unterschiedlichen Beschreibungssprachen definiert sein können.

Die Regelbasis enthält Restrukturierungsregeln, die auf existierenden und neuen Optimierungsstrategien beruhen. Insbesondere nutzen die Restrukturierungsregeln das Wissen über Abhängigkeiten in einer Workflowbeschreibung aus, um die darin eingebetteten Datenverarbeitungsoperationen unter Beibehaltung der ursprünglichen Ausführungssemantik eines datenintensiven Workflows zu optimieren.

Die Kontrollstrategie bestimmt, welche Restrukturierungsregeln in welcher Reihenfolge auf welche Teile einer Workflowbeschreibung angewendet werden, um zum einen das Op-

timierungspotential eines datenintensiven Workflows umfassend zu nutzen und zum anderen die Korrektheit der Regelanwendungen sicherzustellen.

Die ausführliche Beschreibung des Prozessgraphenmodells, der Regelbasis und der Kontrollstrategie stehen im Mittelpunkt dieser wissenschaftlichen Abhandlung. Des Weiteren wird eine prototypische Implementierung des Optimierungsansatzes vorgestellt, welche dessen praktische Einsatzfähigkeit unterstreicht. Schließlich wird die Effektivität der einzelnen Restrukturierungsregeln mithilfe verschiedener Messszenarien untersucht. Dabei wird gezeigt, dass durch Anwendung der Restrukturierungsregeln Leistungssteigerungen in mehreren Größenordnungen erreicht werden können.

Der vorgestellte Optimierungsansatz für datenintensive Workflows basiert auf Ergebnissen, die im Rahmen des Kooperationsprojektes II4BPEL zusammen mit dem IBM-Entwicklungsprojekt in Böblingen entstanden sind. Dazu zählen neben der Entwicklung des Sprachkonzepts von BPEL/SQL, das eine direkte Einbettung von SQL-Anweisungen in einer BPEL-Workflowbeschreibung ermöglicht, auch die Definition einführender heuristischer Optimierungsregeln für BPEL/SQL-Workflows sowie eine erste prototypische Implementierung eines heuristischen, regelbasierten Optimierers für datenintensive Workflows. Ebenso wurde ein erster Optimierungsansatz für Stored-Procedures betrachtet.

Die in diesem Projekt gewonnenen Erkenntnisse zur Optimierung datenintensiver Workflows dienten als Grundlage für weiterführende Forschungsarbeiten auf diesem Gebiet. Deren Ergebnisse werden in dieser Doktorarbeit vorgestellt. Hierbei sollen insbesondere die folgenden eigenen Beiträge hervorgehoben werden: (i) Die Entwicklung eines generischen Optimierungsframeworks für datenintensive Workflows, das an die Bedürfnisse verschiedener Beschreibungssprachen flexibel angepasst werden kann und darüber hinaus eine Optimierung datenintensiver Workflows über Sprachgrenzen hinweg ermöglicht. (ii) Die (formale, grafische und syntaktische) Definition des Prozessgraphenmodells als Grundlage für eine sprachunabhängige Darstellung und Optimierung von Workflow- und Stored-Procedure-Beschreibungen. (iii) Die systematische Erweiterung der initialen Regelbasis um neue Restrukturierungsregeln einschließlich der Betrachtung existierender Optimierungskonzepte und deren Übertragung auf den Kontext datenintensiver Workflows. (iv) Die formale Definition der Regelbasis auf Grundlage der formalen Definition von PGM. (v) Die Entwicklung einer Kontrollstrategie zur effektiven Anwendung der gegebenen Regelbasis. (vi) Die prototypische Implementierung des Optimierungsframeworks sowie (vii) eine fundierte Evaluierung des Optimierungsansatzes basierend auf Messungen.

Abstract

Nowadays, workflow- and database technology is crucial for automating business processes. Workflow management systems execute business processes by means of workflows that integrate numerous application systems and data sources. Typically, database management systems store business data that is referenced by business processes. Obviously, the performance of a business process is directly dependent on the efficiency of data access, data processing, and data management.

Leading vendors of workflow- and database management systems extended workflow description languages like BPEL by SQL-functionality. This simplifies the modeling of data-intensive workflows that process huge amount of relational data. Thereby, data processing operations like SQL statements or stored procedure calls may be defined directly on the workflow level, instead of encapsulating them into Web services. Thus, a workflow description discloses the whole optimization potential concerning its data management. The outcome of this is a new query optimization approach that completes existing approaches in database systems: Rewrite rules transform inefficient data processing operations in a workflow description into equivalent operations that perform better on a workflow- and database management system respectively. This results in an improved performance of data-intensive workflows with respect to their data management.

This doctoral thesis introduces a heuristic, rule-based optimizer for data-intensive workflows. In order to improve the data management of a data-intensive workflow, the optimizer applies an appropriate rule set according to a well-defined control strategy on an internal representation for data-intensive workflows, the so-called process graph modell (PGM).

PGM is adjusted to the optimization of data-intensive workflows and is the appropriate basis for rule-based transformations. It allows for an exact and language independent definition of rewrite rules. Consequently, it supports various optimization scenarios covering a multitude of data processing operations defined in different description languages.

The rule set contains rewrite rules that are based on existing as well as on new optimization techniques. Each rewrite rule consists of two parts: a condition and an action part. The condition part defines what conditions have to hold for a rule application in order to preserve the original workflow semantics. It refers to control flow dependencies as well as to data and communication dependencies. Additionally, it considers detailed information of activities. The action part of a rewrite rule defines the transformations applied to a workflow description provided that the corresponding condition part is fulfilled.

The purpose of the control strategy is to use efficiently the optimization potential of a data-intensive workflow and to ensure the correctness of a rule's application. Therefore, it identifies so-called optimization spheres, i.e., parts of a workflow, for which applicable rewrite rules should be identified. Determining such spheres is necessary because if one applies rewrite rules across spheres, the semantics of a workflow may change. Another function of the control strategy is to define the order, in which rule conditions are checked for applicability and the order, in which rules are finally applied.

This scientific paper focuses in detail on the process graph modell, the rule set and the control strategy. Furthermore, a prototype of the optimizer demonstrates its practical utilizability. Finally, several experiments emphasize the effectiveness of the optimization approach. The experimental results show that performance gains of orders of magnitude are achievable when applying the given set of rewrite rules to data-intensive workflows.

The introduced optimization approach for data-intensive workflows is based on results that have been developed together with the IBM lab in Böblingen in the cooperation project II4BPEL. These results include all the following: The development of BPEL/SQL, an extension for BPEL that allows to embed SQL statements directly within a BPEL workflow description. The definition of introductive heuristic rewrite rules for BPEL/SQL workflows as well as a first prototype implementation of a heuristic, rule-based optimizer for data-intensive workflows. In addition, first steps were considered towards the optimization of stored procedures.

All these findings served as foundation for continuative research on the area of optimizing data-intensive workflows. This doctoral thesis presents the research results. In particular, all the following is a contribution of the author concerning this research area: (i) The development of a generic optimization framework for data-intensive workflows that can be adapted to the needs of different description languages and that allows to apply rewrite rules on a workflow description accross language boarders. (ii) The formal, visual and syntactical definition of the process graph modell that enables a language independent representation and optimization of workflow and stored procedure descriptions. (iii) The systematic extension of the initial rule set through new rewrite rules. This includes the analysis of existing optimization techniques and their transfer to the context of data-intensive workflows. (iv) The formal definition of the rule set by means of the formal definition of PGM. (v) The development of an efficient control strategy for applying the given rule set to data-intensive workflows. (vi) The implemenation of a prototyp of the optimization framework, as well as (vii) a sound evaluation of the optimization approach based on experiments.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ACID	Atomicity-Consistency-Isolation-Durability
AXML	Active-XML
A2A	Application-To-Application
BPEL	Business-Process-Execution-Language
BPELJ	BPEL Extension for Java-Technology
BPEL-SPE	BPEL Extension for Subprocesses
BPEL/SQL	BPEL Extension for SQL
BPEL4People	BPEL Extension for People
BPMN	Business-Process-Modeling-Notation
CGO	Coarse-Grained-Optimization
CPU	Central-Processing-Unit
DDL	Data-Definition-Language
DML	Data-Manipulation-Language
EPK	Ereignisgesteuerte-Prozessketten
ETL	Extract-Transform-Load
ICS	Input-Controlflow-Slot
IO	Input-Output
HTTP	Hypertext-Transfer-Protocol
LOS	Loop-Optimization-Sphere
MQO	Multi-Query-Optimierung
OCS	Output-Controlflow-Slot
OS	Optimization-Sphere
PGM	Prozessgraphenmodell
PGM/F	Prozessgraphenmodell/Framework
PSM	Persistent-Stored-Modules
P2A	Person-To-Application
P2P	Person-To-Person

QGM	Query-Graph-Model
SOA	Service-Oriented-Architecture
SPJ	Select-Project-Join
SQL	Structured-Query-Language
UDTF	User-Defined-Table-Functions
UML	Unified-Modeling-Language
WSDL	Web-Services-Description-Language
XML	Extensible-Markup-Language
XQSE	XQuery-Scripting-Extension
XSLT	Extensible-Stylesheet-Language-Transformations

1 Einleitung

Dieses Kapitel führt in das Optimierungsproblem ein, das im Rahmen dieser Doktorarbeit behandelt wird, und gibt einen Überblick über deren Inhalt.

1.1. Motivation

Unternehmen sind heutzutage zunehmend gezwungen, ihre Geschäfte zu optimieren und zu automatisieren, um an den globalen Märkten erfolgreich bestehen zu können. Dabei spielen Workflow- und Datenbankmanagementsysteme (im Folgenden Workflow- und Datenbanksysteme genannt) eine zentrale Rolle. Damit können Geschäftsprozesse, die zahlreiche heterogene Anwendungssysteme und umfangreiche Datenbestände eines Unternehmens integrieren, automatisiert werden [Dvt05].

Abbildung 1.1 zeigt eine typische Architektur zur Automatisierung von Geschäftsprozessen. Zunächst wird auf der *Prozessmodellierungsebene* ein abstraktes Prozessmodell, das die Struktur eines zu automatisierenden Geschäftsprozesses anschaulich und leicht verständlich beschreibt, erstellt [LR00]. Hierfür stehen eine Reihe unterschiedlicher Beschreibungssprachen zur Verfügung, wie z.B. die Business-Process-Modeling-Notation (BPMN) [OMGa], Ereignisgesteuerte-Prozessketten (EPK) [HKS92, KNS92] oder die Unified-Modeling-Language (UML) [OMGd].

Ein Prozessmodell kann aus Teilen bestehen, die von Menschen ausgeführt werden, und aus Teilen, die rechnergestützt ablaufen. Die rechnergestützten Teile werden auf der *Workflowmodellierungsebene* in ein Workflowmodell überführt, das anschließend auf der *Workflowausführungsebene* von einem Workflowsystem zur Ausführung gebracht wird. Im Gegensatz zu einem abstrakten Prozessmodell wird ein Workflowmodell präzise definiert, und es enthält alle technischen Details, die zu seiner Ausführung notwendig sind.

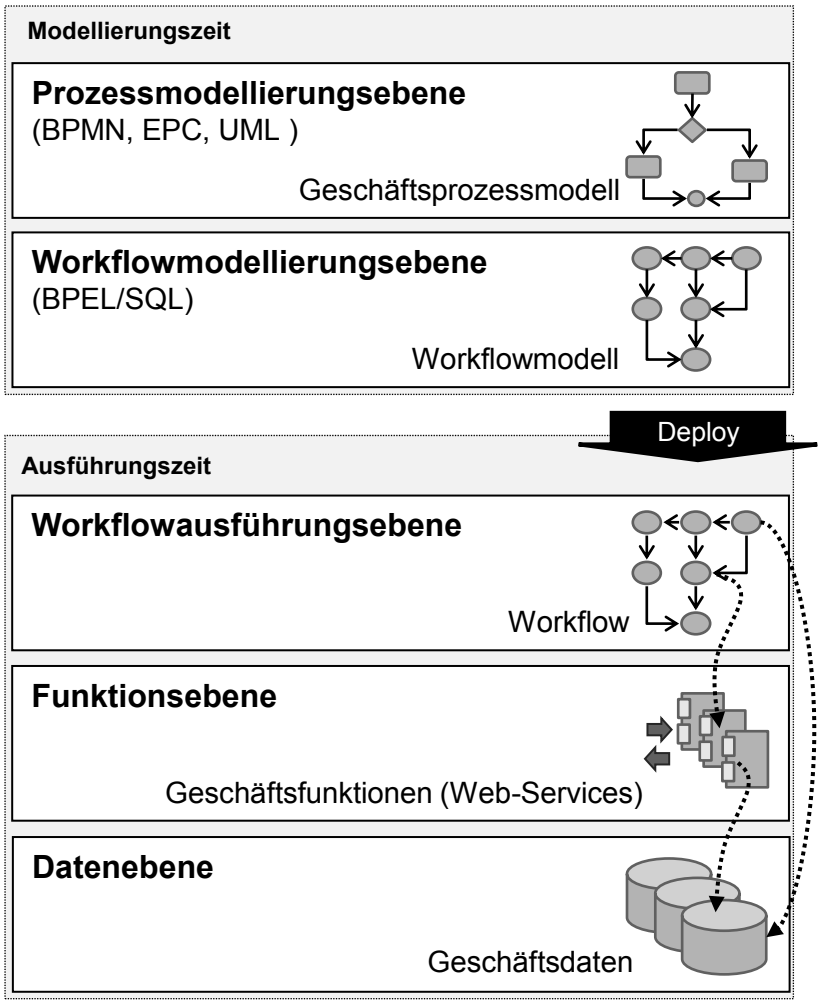


Abbildung 1.1.: Architektur zur Automatisierung von Geschäftsprozessen

Zur Beschreibung eines Workflowmodells wird die Standardbeschreibungssprache BPEL (Business-Process-Execution-Language) [OAS07] verwendet. Die Implementierung eines BPEL-Workflows erfolgt auf zwei Ebenen (Prinzip der „Zwei-Ebenen-Programmierung“) [LR00]: Auf der Workflowmodellierungsebene wird das Workflowmodell spezifiziert, das den Kontroll- und Datenfluss zwischen Aktivitäten, welche die einzelnen Arbeitsschritte eines Geschäftsprozesses repräsentieren, definiert („Programmierung-im-Großen“). Auf der *Funktionsebene* erfolgt die Implementierung dieser Aktivitäten in Form von Web-Services, welche die gewünschten Geschäftsfunktionen realisieren und unabhängig vom Workflowmodell entwickelt werden können („Programmierung-im-Kleinen“). Die Geschäftsfunktionen greifen auf Geschäftsdaten zu, die meistens von Datenbanksystemen auf der *Datenebene* verwaltet werden.

Führende Hersteller von Workflowsystemen, wie IBM, Oracle und Microsoft, haben BPEL um SQL-spezifische Funktionalitäten erweitert, um komplexe Datenverarbeitungsoperationen mit BPEL beschreiben zu können. Diese Technologie wird im Folgenden *BPEL/SQL* genannt. Workflows, die in BPEL/SQL definiert sind, werden als *datenin-*

tensive Workflows bezeichnet. Merkmal eines datenintensiven Workflows ist die Verarbeitung großer relationaler Datenmengen, welche die zu Grunde liegenden Geschäftsdaten repräsentieren.

Mit BPEL/SQL kann direkt aus einem Workflow heraus auf die Datenebene zugegriffen werden, ohne eine entsprechende Implementierung auf der Funktionsebene entwickeln zu müssen. Dies wird beispielsweise durch entsprechende vordefinierte Aktivitätstypen erreicht, denen SQL-Anweisungen[1], die von einem Workflow- direkt auf einem Datenbanksystem ausgeführt werden können, als Parameter übergeben werden.

Ein Vorteil einer direkten Spracheinbettung ist, dass alle SQL-Anweisungen Teil einer Workflowbeschreibung sind. Dadurch wird das gesamte Optimierungspotential eines Workflowmodells bezüglich seiner ausgeführten Datenverarbeitungsoperationen sichtbar. Dies eröffnet eine neue Möglichkeit der SQL-Optimierung oberhalb der Datenebene. SQL-Anweisungen können unter Berücksichtigung der Ausführungslogik eines Workflows direkt auf der Workflowmodellierungsebene optimiert werden mit dem Ziel, die Ausführungszeit eines datenintensiven Workflows zu verkürzen. Eine solche Art der SQL-Optimierung ist dagegen nicht möglich, wenn Datenverarbeitungsoperationen außerhalb einer Workflowbeschreibung in Web-Services definiert werden. In diesem Fall bleibt das Optimierungspotential auf der Funktionsebene verborgen und muss erst durch aufwändige Analysen der Implementierungen der einzelnen Web-Services aufgedeckt werden. Zudem ist zu beachten, dass die Implementierung eines Web-Services bei der Modellierung eines Workflows in der Regel nicht verfügbar ist.

Ein weiterer Vorzug der direkten Spracheinbettung besteht darin, dass die Optimierungsschritte direkt auf der Workflowmodellierungsebene vollzogen werden können. Hierzu muss lediglich die Beschreibung eines Workflowmodells angepasst werden. Bei Web-Services müssen dagegen deren Implementierungen auf der Funktionsebene geändert werden. Dies verursacht einen vielfach höheren Aufwand. Zugleich wird eine unabhängige Entwicklung eines Workflowmodells und seiner zugehörigen Geschäftsfunktionen (bzw. Web-Services) auf der Funktionsebene verhindert. Dies steht im Gegensatz zum Prinzip der Zwei-Ebenen-Programmierung eines Workflows.

Bislang gibt es noch keinen Ansatz, der eine solche Art der SQL-Optimierung in einem datenintensiven Workflow realisiert: Auf der Prozessmodellierungsebene werden Geschäftsprozesse primär aus betriebswirtschaftlicher Sicht optimiert [Rei03, Dvt05, LR00]. Dabei werden durch eine systematische Untersuchung Schwachstellen in dem zu Grunde gelegten Geschäftsprozessmodell aufgedeckt, und es wird Verbesserungspotential erkannt. Da erst nach dieser Phase die Umsetzung des optimierten Geschäftsprozesses in einen entsprechenden datenintensiven Workflow erfolgt, bleiben technische Details, wie die später auszuführenden SQL-Anweisungen, unberücksichtigt. Zur Ausführungszeit werden die in einem datenintensiven Workflow eingebetteten SQL-Anweisungen gemäß ihrer Ausführungsreihenfolgen einzeln an ein zu Grunde liegendes Datenbanksystem gesendet. Folglich kann ein Datenbanksystem alle SQL-Anweisungen ausschließlich einzeln entgegennehmen, optimieren und ausführen. Dadurch können bei der Anfrageoptimierung we-

[1] Der Begriff der SQL-Anweisung wird in dieser Arbeit als Sammelbegriff für Retrieval-Anfragen, Datenmanipulations- und Datendefinitionsanweisungen verwendet.

der Kontroll- noch Datenflussabhängigkeiten zwischen den in der Ausführungslogik eines Workflows eingebetteten SQL-Anweisungen berücksichtigt werden. Hierdurch bleibt ein erhebliches Optimierungspotential ungenutzt.

An dieser Stelle setzt das in dieser Arbeit vorgestellte PGM/F-System zur Optimierung datenintensiver Workflows an. Dabei handelt es sich um einen heuristischen, regelbasierten Optimierungsansatz, der die klassische SQL-Anfrageoptimierung von der Daten- auf die Workflowmodellierungsebene überträgt. Unter Verwendung von Restrukturierungsregeln werden die Datenverarbeitungsoperationen eines datenintensiven Workflowmodells unter Beibehaltung seiner ursprünglichen Ausführungssemantik derart transformiert, dass eine Laufzeitverbesserung zu erwarten ist. Dieser Ansatz berücksichtigt für seine Optimierungsentscheidung alle SQL-Anweisungen sowie die zwischen ihnen existierenden Kontroll- und Datenflussabhängigkeiten, um das Optimierungspotential eines datenintensiven Workflows effektiv zu nutzen.

1.2. Überblick

Die vorliegende Arbeit ist wie folgt gegliedert: Das zweite Kapitel befasst sich mit Grundlagen und verwandten Arbeiten. Dabei werden die wichtigsten Konzepte von BPEL/SQL vorgestellt, und darauf aufbauend wird das Optimierungspotential eines BPEL/SQL-Workflows erörtert. Außerdem werden existierende Optimierungsansätze für SQL-Anweisungen in Datenbanksystemen behandelt und mit dem Optimierungsansatz von PGM/F verglichen. Anschließend gibt Kapitel drei einen Überblick über die grundlegenden Konzepte des PGM/F-Systems. Der heuristische, regelbasierte Optimierungsansatz wird vorgestellt, und verschiedene Optimierungsszenarien, die von PGM/F unterstützt werden, werden erläutert. Das Prozessgraphenmodell (PGM) steht im Mittelpunkt der Betrachtungen in Kapitel vier. Dabei handelt es sich um eine interne Repräsentation von PGM/F, die als Grundlage für die Definition und Anwendung der Restrukturierungsregeln auf ein Workflowmodell dient. Das fünfte Kapitel stellt die Regelbasis von PGM/F im Detail vor und diskutiert ihre Effekte auf einen datenintensiven Workflow. Dabei wird gezeigt, wie PGM/F das Wissen über Abhängigkeiten in einer Workflowbeschreibung nutzt, um darin eingebettete Datenverarbeitungsoperationen bzw. Datenverarbeitungsmuster unter Beibehaltung der ursprünglichen Ausführungssemantik eines datenintensiven Workflows zu optimieren. Teil dieser Regelmenge sind sowohl existierende Optimierungsansätze, die auf den Kontext datenintensiver Workflows übertragen werden können, als auch neue Optimierungstechniken, die in einem solchen Workflow gewinnbringend angewendet werden können. Des Weiteren werden die Grenzen eines heuristischen Optimierungsansatzes aufgezeigt, und es wird erörtert, welche Optimierungsschritte mit einer kostenbasierten Optimierungsstrategie realisiert werden können. Kapitel sechs führt eine Kontrollstrategie ein, die bestimmt, auf welchen Teilen und in welcher Reihenfolge die Regelbasis auf eine PGM-Repräsentation angewendet wird, um die ursprüngliche Ausführungssemantik einer Workflowbeschreibung zu erhalten. Schließlich werden im siebten Kapitel die durch PGM/F erzielbaren Leistungsverbesserungen durch Messungen belegt und bewertet. Das abschließende Kapitel fasst die Ergebnisse dieser Arbeit zusammen und gibt einen Ausblick auf weiterführende Arbeiten.

2
Grundlagen

Dieses Kapitel behandelt die für das Verständnis dieser Arbeit notwendigen Grundlagen. Dazu wird zunächst in Teilkapitel 2.1 die Workflowbeschreibungssprache BPEL/SQL vorgestellt, mit der datenintensive Workflows modelliert werden, die im Mittelpunkt dieser Arbeit stehen. Zu diesem Teilkapitel gehört ebenfalls eine Klassifikation datenintensiver Workflows und eine Diskussion über das Optimierungspotential, das in datenintensiven Workflows gewinnbringend genutzt werden kann. Abschließend gibt Teilkapitel 2.2 einen Überblick über existierende Optimierungstechniken in Datenbanksystemen und grenzt diese zum Optimierungsansatz von PGM/F ab.

2.1. Die Workflowbeschreibungssprache BPEL/SQL

Die Workflowbeschreibungssprache BPEL bietet zahlreiche vordefinierte Aktivitätstypen an, mit denen die Ausführungslogik eines Workflowmodells definiert werden kann. Dabei wird zwischen elementaren und strukturierten Aktivitätstypen unterschieden. Elementare Aktivitätstypen sind atomar, d.h. sie sind nicht aus anderen Aktivitäten aufgebaut und können diese auch nicht enthalten. Die strukturierten Aktivitätstypen dienen dagegen in einem Workflow zur Modellierung von sequentiellen (<`sequence`>), parallelen (<`flow`>), alternativen (<`if`>) und iterativen (<`forEach`>, <`while`>, <`repeatUntil`>) Abläufen. Dabei kann der Kontrollfluss entweder blockbasiert und prozedural definiert werden, d.h. durch Verschachtelung der strukturierten Aktivitätstypen, oder graphenbasiert, d.h. durch Verwendung sogenannter <`links`> (gerichtete Kontrollflusskanten), welche die Aktivitäten in einer <`flow`> Aktivität miteinander verbinden und somit deren Kontrollfluss definieren. Außerdem können beide Modellierungsstile kombiniert werden.

Dagegen wird der Datenfluss in BPEL nicht explizit modelliert. Stattdessen werden Daten, die von Aktivitäten referenziert werden, in Variablen zur Verfügung gestellt. Die Bearbeitung der Daten erfolgt in BPEL mittels einer <assign> Aktivität. Neben einem selektiven Datenzugriff unterstützt dieser Aktivitätstyp zur einfachen Verarbeitung und Transformation der Daten auch die Verwendung von XPath-Ausdrücken [W3Cb] bzw. XSLT-Funktionen [W3Cd].

Zur Modellierung von Kommunikationsmustern stehen verschiedene Aktivitätstypen zur Verfügung: <receive> und <pick> Aktivitäten können Nachrichten von Web-Services empfangen, eine <reply> Aktivität kann Nachrichten an Web-Services versenden, und eine <invoke> Aktivität dient zum Aufruf von Web-Service-Operationen.

Eine <scope> Aktivität definiert einen speziellen Ausführungskontext für eine Gruppe von Aktivitäten. Dieser Aktivitätstyp unterstützt die Definition lokaler Variablen, eine Fehlerbehandlung (<faultHandlers>) und eine auf Kompensation basierende Recovery-Semantik (<compensationHandler>).

Aufgaben der Verarbeitung von Geschäftsdaten auf der Datenebene können in BPEL nur in Web-Services, die in einem BPEL-Workflow mittels <invoke> Aktivitäten aufgerufen werden, gekapselt werden. Detailinformationen zu ausgeführten Datenverarbeitungsoperationen werden folglich nicht direkt im Workflowmodell erfasst. Insbesondere ist die im Web-Service gekapselte SQL-Anweisung nicht Bestandteil einer Workflowbeschreibung. Hierdurch werden die Beschreibungen von Datenverarbeitungsoperationen inhaltlich getrennt, was gerade bei datenintensiven Workflows nicht sinnvoll ist.

Aus diesem Grund bieten führende Hersteller von Workflowsystemen, wie IBM, Oracle oder Microsoft, Erweiterungen zu BPEL an, die eine inhaltliche Trennung der Datenverwaltung vermeiden. Dabei wird die SQL-Funktionalität nicht in einem Web-Service gekapselt, sondern in die Workflowbeschreibung eingebettet. In diesem Fall gehört die auszuführende SQL-Anweisung direkt zur Beschreibung der Aktivität und ist somit integraler Bestandteil eines Workflowmodells. Deshalb wird in dieser Arbeit eine solche Erweiterung auch als *SQL-Inline-Support* für BPEL bezeichnet [VSRM08].

Für diesen SQL-Inline-Support gibt es keinen Standard. Es werden hierfür proprietäre Spracherweiterungen für BPEL, die in dieser Arbeit unter der Bezeichnung *BPEL/SQL* zusammengefasst werden, angeboten.

Beispielsweise erweitert IBMs WebSphere-Process-Server [IBMf] die Menge an vordefinierten BPEL-Aktivitäten um SQL-spezifische Aktivitätstypen. Diese erlauben eine Definition von SQL-Anweisungen direkt in einem Workflowmodell. Dadurch umfasst eine Workflowbeschreibung alle für einen Workflow relevanten Datenverwaltungsaufgaben.

Im Gegensatz dazu bietet Microsofts Workflow-Foundation [Mica] keinen proprietären SQL-Inline Support an. Stattdessen können Aktivitätstypen auf Grundlage einer .NET-basierten Programmiersprache, wie z.B. C# oder Visual Basic, implementiert werden. Auf diese Weise lässt sich ein SQL-Inline-Support anwenderspezifisch realisieren.

Der Ansatz von Oracles SOA-Suite [Ora] basiert auf proprietären SQL-spezifischen XPath-Extension-Functions, die von einer <assign> Aktivität ausgeführt werden. Diese Funktionen erhalten die SQL-Anweisung und die Datenbank, auf der die SQL-Anweisung ausgeführt werden soll, als Parameter. Die Funktionen unterstützen sowohl SQL-Anfragen,

Datenmanipulations- (DML) und Datendefinitionsanweisungen (DDL) als auch Aufrufe benutzerdefinierter Prozeduren (im Folgenden Stored-Procedures genannt). Anfrageergebnisse werden in mengenbasierten Datenstrukturen im Workflow zur Verfügung gestellt.

Im Folgenden wird die IBM-spezifische Spracherweiterung als eine mögliche Realisierungsvariante von BPEL/SQL näher vorgestellt. Weitere Details zu den anderen beiden Ansätzen sind in [VSRM08] zu finden.

An dieser Stelle sei noch erwähnt, dass neben BPEL/SQL weitere Ergänzungen für BPEL angeboten werden, um die Modellierungsmächtigkeit des BPEL-Standards zu erhöhen. Beispielsweise unterstützt BPEL4People [KKL+05a] die Modellierung von Aktivitäten, die von Menschen ausgeführt werden. BPEL-SPE [KKL+05b] erlaubt die Einbindung von Unterprozessen in einen BPEL-Workflow. Dagegen ermöglicht BPELJ [BGK+04], Java-Code direkt in eine BPEL-Workflowbeschreibung einzubetten.

2.1.1. Grundlagen

In diesem Teilkapitel werden die grundlegenden Konzepte der IBM-spezifischen Variante von BPEL/SQL vorgestellt. Dazu gehören neben SQL-spezifischen Aktivitätstypen auch mengenbasierte Datenstrukturen zur Verarbeitung relationaler Datenmengen.

SQL-spezifische Aktivitätstypen

BPEL/SQL bietet folgende SQL-spezifische Aktivitätstypen zur Realisierung einer mengenbasierten Datenverarbeitung auf der Workflowebene:

- Eine <sql> Aktivität führt SQL-Anweisungen auf einem relationalen Datenbanksystem aus. Es werden sowohl Anfragen, DML- und DDL-Anweisungen als auch der Aufruf von Stored-Procedures unterstützt. Als Eingabeparamter einer SQL-Anweisung können die im Workflow definierten BPEL-Variablen verwendet werden. Die Beschreibung einer <sql> Aktivität enthält eine SQL-Anweisung und das Datenbanksystem, auf dem die SQL-Anweisung ausgeführt werden soll. Dabei kann das Datenbanksystem auch logisch über einen qualifizierenden Bezeichner referenziert werden. In diesem Fall wird die logische Referenz zum Deploymentzeitpunkt eines Workflowmodells an ein konkretes physisches Datenbanksystem gebunden.

  Zur Laufzeit wird die SQL-Anweisung einer <sql> Aktivität an das zu Grunde liegende Datenbanksystem gesendet und dort ausgeführt. Bei einer SQL-Anfrage wird die resultierende Ergebnismenge nicht an die <sql> Aktivität zurückgesendet, sondern im Datenbanksystem materialisiert. Stattdessen wird als Anfrageergebnis eine Referenz auf die materialisierte Ergebnismenge geliefert.

  Im Vergleich zum Aufruf eines Web-Services ist die Ausführung einer <sql> Aktivität performanter: Führt ein Workflowsystem eine <sql> Aktivität aus, wird die in der Aktivität definierte SQL-Anweisung direkt an das mit der <sql> Aktivität assoziierte Datenbanksystem übermittelt. Die Interaktion mit einem Web-Service

ist dagegen aus Sicht eines Workflowsystems mit höheren Ausführungskosten, die insbesondere durch SOAP [W3Ca] verursacht werden, verbunden.

- Eine <retrieveSet> Aktivität lädt eine Anfrageergebnismenge in einen Workflow. Bei diesem Materialisierungsschritt werden die relationalen Daten in einer mengenbasierten XML-RowSet-Datenstruktur (siehe nächster Abschnitt) bereitgestellt.
- In einem kurzlaufenden BPEL/SQL-Workflow werden alle <sql> Aktivitäten in einer Transaktion ausgeführt. In einem langlaufenden BPEL/SQL-Workflow ist dagegen die flexible Definition von Transaktionsgrenzen notwendig. Hierfür steht eine <atomicSQLSequence> Aktivität zur Verfügung, die eine Liste von <sql> und <retrieveSet> Aktivitäten enthält, die in einer Transaktion ausgeführt werden.

Mengenbasierte Datenstrukturen

- Relationale Tabellen können in einem BPEL/SQL-Workflow an sogenannte *Set-Referenz-Variablen* gebunden werden. Set-Referenz-Variablen sind als Zeiger auf eine relationale Tabelle definiert. Je nach Verwendung unterscheidet man zwischen zwei Arten von Set-Referenzen: Eine *Input-Set-Referenz* verweist auf eine Tabelle in einer SQL-Anweisung. Sie kann anstelle eines statischen Tabellennamens verwendet werden. Eine *Result-Set-Referenz* verweist dagegen auf eine relationale Tabelle, die das Ergebnis einer Anfrage bzw. eines Stored-Procedure-Aufrufes beinhaltet. Eine Result-Set-Referenz kann in einer nachfolgenden <sql> Aktivität als eine Input-Set-Referenz umdefiniert werden. Auf diese Weise können relationale Daten über Aktivitäts- und Workflowgrenzen hinweg referenziert (by-reference) anstatt materialisiert (by-value) übergeben werden. Dadurch kann ein hoher Leistungsgewinn erzielt werden.

  Typischerweise verweisen Result-Set-Referenzen auf temporäre Tabellen. Um zu verhindern, dass temporäre Tabellen vor der Ausführung einer Workflowinstanz manuell erzeugt bzw. nach der Ausführung einer Workflowinstanz manuell entfernt werden müssen, gibt es in BPEL/SQL die Möglichkeit, sogenannte *Preparation-* und *Cleanup-Anweisungen*, die mit einer Result-Set-Referenz assoziiert sein können, zu definieren. Diese Anweisungen enthalten Create- und Drop-Anweisungen, die zu Beginn bzw. nach Beendigung der Ausführung einer Workflowinstanz vom Workflowsystem ausgeführt werden. Auf diese Weise kann der Lebenszyklus einer temporären Tabelle direkt auf der Workflowmodellierungsebene definiert werden.
- Bei einer *Set-Variablen* handelt es sich um eine XML-RowSet-Datenstruktur, in der die Materialisierung einer relationalen Tabelle zur Verfügung gestellt wird. Ein Materialisierungsschritt erfolgt in einer <retrieveSet> Aktivität, die relationale Daten lokal im Workflow in Set-Variablen verfügbar macht. Abbildung 2.1 veranschaulicht die Struktur einer XML-RowSet-Datenstruktur.

  Auf der rechten Seite der Abbildung ist eine relationale Tabelle *ApprovedOrders* mit den Spalten *ItemID*, *ItemQuantity*, *SupID* und *Price* abgebildet. Die linke Seite zeigt die Materialisierung der Tabelle in Form einer XML-RowSet-Datenstruktur, wie sie in BPEL/SQL in einer Set-Variablen zur Verfügung gestellt wird.

XML-RowSet

```
<ApprovedOrders>
 <Row>
  <ItemID> 3245 </ItemID>
  <ItemQuantity> 34 </ItemQuantity>
  <SupID> 32 </SupID>
  <Price> 10.000 </Price>
 </Row>
 ...
</ApprovedOrders>
```

Relationale Tabelle

ApprovedOrders

ItemID	ItemQuantity	SupID	Price
3245	34	32	10.000
...			

Abbildung 2.1.: XML-RowSet-Datenstruktur der Tabelle *ApprovedOrders*

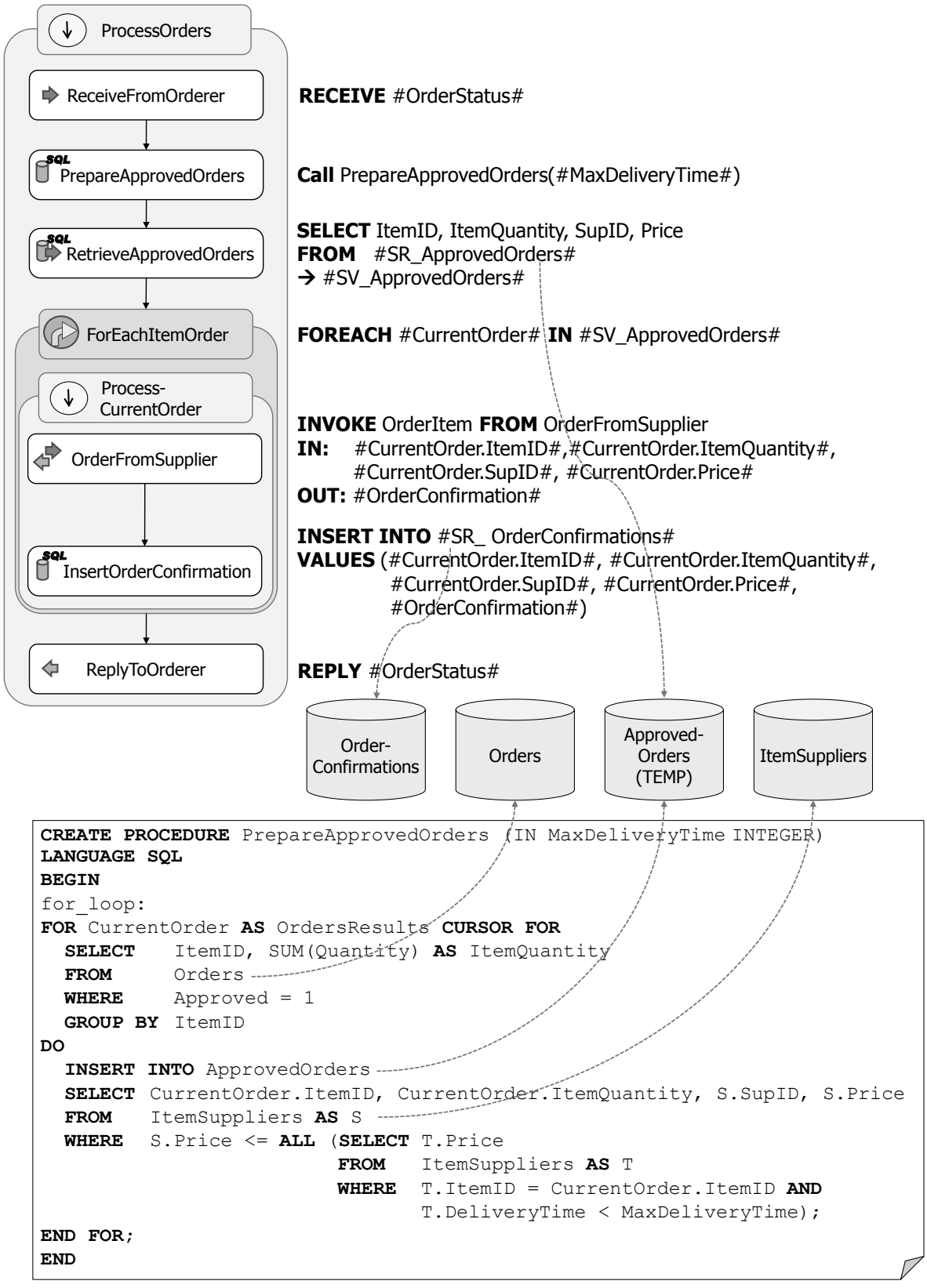
Die Tupel der Tabelle *ApprovedOrders* werden in dieser XML-RowSet-Datenstruktur durch XML-Elemente repräsentiert. Dabei stellt <ApprovedOrders> das Wurzelelement der Datenstruktur dar. Jedes Tupel der Tabelle *ApprovedOrders* wird durch ein <Row> Element definiert. Darin enthalten sind jeweils XML-Elemente für die einzelnen Attribute, deren Inhalte den einzelnen Attributwerten eines Tupels entsprechen.

Beispielszenario: Automatisierung eines Bestellprozesses mit BPEL/SQL

Die Konzepte von BPEL/SQL sollen anhand des in Abbildung 2.2 dargestellten Beispielszenarios veranschaulicht werden. Das Beispielszenario enthält gängige Datenverarbeitungsmuster, die in einem datenintensiven Workflow typischerweise auftreten können. Der Beispiel-Workflow basiert auf einem einfachen Bestellprozess, der eine Menge vorliegender Bestelldaten automatisch verarbeitet. Bei dem Beispiel-Workflow handelt es sich um einen prozedural modellierten, kurzlaufenden BPEL/SQL-Workflow, dessen <sql> Aktivitäten alle auf demselben Datenbanksystem bzw. innerhalb derselben Transaktion ausgeführt werden. Die Fehlerbehandlung des Beispiel-Workflows wird aus Gründen der Übersichtlichkeit nicht dargestellt. Das Datenbankschema, das dem Beispiel-Workflow zu Grunde liegt, ist in Abbildung E.2 im Anhang veranschaulicht.

Im oberen Teil der Abbildung sind die Aktivitäten des Beispiel-Workflows, die in der <sequence> Aktivität *ProcessOrders* nacheinander ausgeführt werden, dargestellt. Die <receive> Aktivität *ReceiveFromOrderer* empfängt eine Nachricht, die den Bestellprozess startet. Am Ende des Bestellprozesses wird die Nachricht mithilfe der <reply> Aktivität *ReplyToOrderer* wieder direkt an den Aufrufer des Bestellprozesses zurückgesendet.

Die <sql> Aktivität *PrepareApprovedOrders* liest die vorliegenden Informationen zu bestätigten Bestellungen aus einer Datenbank aus. Hierfür ruft sie die Stored-Procedure *PrepareApprovedOrders* auf, welche die Bestelldaten zur weiteren Verarbeitung im Workflow aufbereitet. Mit der folgenden <retrieveSet> Aktivität *RetrieveApprovedOrders* werden die zur weiteren Verarbeitung notwendigen Bestelldaten auf der Workflowebene materialisiert. Mit der <forEach> Aktivität *ForEachItemOrder* wird die materialisierte Bestelldatenmenge innerhalb der <sequence> Aktivität *ProcessCurrentOrder* schrittweise durchlaufen. Im Schleifenrumpf ruft die <invoke> Aktivität zunächst die Operation *OrderItem* des Web-Services *OrderFromSupplier* auf, der eine Bestellung an einen jeweils passenden Zulieferer weiterleitet. Die vom Web-Service gelieferte Bestätigung wird schließlich mit der <sql> Aktivität *InsertOrderConfirmation* persistent gespeichert.




```
CREATE PROCEDURE PrepareApprovedOrders (IN MaxDeliveryTime INTEGER)
LANGUAGE SQL
BEGIN
for_loop:
FOR CurrentOrder AS OrdersResults CURSOR FOR
  SELECT    ItemID, SUM(Quantity) AS ItemQuantity
  FROM      Orders
  WHERE     Approved = 1
  GROUP BY  ItemID
DO
  INSERT INTO ApprovedOrders
  SELECT CurrentOrder.ItemID, CurrentOrder.ItemQuantity, S.SupID, S.Price
  FROM   ItemSuppliers AS S
  WHERE  S.Price <= ALL (SELECT T.Price
                         FROM   ItemSuppliers AS T
                         WHERE  T.ItemID = CurrentOrder.ItemID AND
                                T.DeliveryTime < MaxDeliveryTime);
END FOR;
END
```

Abbildung 2.2.: Automatisierter Bestellprozess in BPEL/SQL

Abbildung 2.2 zeigt weitere Details zu den im Workflowmodell definierten Aktivitäten. Die von einer Aktivität ausgeführte Funktionalität wird neben der grafischen Repräsentation der Aktivität in sprachneutralem Pseudocode dargestellt. Der Pseudocode umfasst unter anderem die von den <sql> Aktivitäten ausgeführten SQL-Anweisungen und die von einer Aktivität referenzierten Variablen, die in dieser Darstellung mit # markiert sind. Außerdem werden Variablen vom Typ *Set* bzw. *Set-Referenz* mit einem Präfix *'SV'* bzw. *'SR'* gekennzeichnet. Referenzen zu externen Daten werden durch Pfeile illustriert.

Beispielsweise referenziert die Set-Referenzvariable *SR_ApprovedOrders* die temporäre Tabelle *ApprovedOrders*, die von der Stored-Procedure *PrepareApprovedOrders* mit Bestelldaten gefüllt wird. Aus Gründen der Übersichtlichkeit wurden die zugehörigen Preparation- und Cleanup-Anweisungen für die temporäre Tabelle *ApprovedOrders*, die mit der Set-Referenzvariablen *SR_ApprovedOrders* assoziiert sind, nicht dargestellt. Die <retrieveSet> Aktivität *RetrieveApprovedOrders* schreibt die in der Tabelle *ApprovedOrders* enthaltenen Bestelldaten in die Set-Variable *SV_ApprovedOrders*, deren Datenstruktur bereits in Abbildung 2.1 veranschaulicht wurde.

Die folgende Aktivität *ForEachItemOrder* iteriert über *SV_ApprovedOrders* und bindet dabei in jeder Iteration ein Tupel der Menge an die Variable *CurrentOrder*. Die Attribute *ItemID*, *ItemQuantity*, *SupID* und *Price* von *CurrentOrder* dienen als Eingabe für den Web-Service-Aufruf der Aktivität *OrderFromSupplier* und für die folgende Insert-Anweisung, welche die Ausgabe des Web-Service-Aufrufes, die in Variable *OrderConfirmation* gespeichert wird, zusätzlich in die Tabelle *OrderConfirmations* schreibt.

Im unteren Teil von Abbildung 2.2 wird die Beschreibung der Stored-Procedure *PrepareApprovedOrders* dargestellt, welche die Bestelldaten, die im Workflow anschließend lokal weiterverarbeitet werden, aufbereitet. Hierfür wird ein Cursor *OrdersResults* definiert, der alle bestätigten Bestellungen aus einer Tabelle *Orders* ausliest und schrittweise für jede bestätigte Bestellung einen geeigneten Lieferanten bestimmt. Die so ermittelten Bestellinformationen werden danach mit einer Insert-Anweisung, die im Schleifenrumpf des Cursors definiert ist, in die Tabelle *ApprovedOrders* hineingeschrieben.

2.1.2. Klassifikation

Die Tabelle 2.1 gibt einen Überblick über wichtige Merkmale von BPEL/SQL zur Klassifizierung datenintensiver Workflows, die im Folgenden näher beschrieben werden.

Grad der Automatisierung

Workflows lassen sich unter anderem nach dem Grad ihrer Automatisierung kategorisieren [GHS95]: Man unterscheidet zwischen Person-to-Person (P2P), Person-to-Application (P2A) und Application-to-Application (A2A). P2P-Workflows beinhalten Aktivitäten, die ausschließlich von Menschen ausgeführt werden. Im Gegensatz dazu werden A2A-Workflows automatisch, d.h. vollständig rechnergestützt, ausgeführt. P2A-Workflows vereinigen entsprechend die Eigenschaften der beiden anderen Workflowtypen.

Der BPEL-Standard unterstützt lediglich die Modellierung von A2A-Workflows, die vollautomatisch ausgeführt werden. Zur Definition von P2P- oder P2A-Workflows wird in BPEL eine entsprechende Erweiterung, wie z.B. BPEL4People, benötigt.

Klassifikationsmerkmal	Realisierungsvarianten
Grad der Automatisierung	P2P ⇔ P2A ⇔ A2A
Ausführungsdauer	kurzlaufend ⇔ langlaufend
Modellierungsstil	prozedural ⇔ graphenbasiert
Transaktionsmodell	ACID ⇔ Kompensation
Definition einer <sql> Aktivität	vollständig ⇔ partiell
Organisation der Datenebene	zentralisiert ⇔ verteilt
	homogen ⇔ heterogen

Tabelle 2.1.: Klassifikation von BPEL/SQL-Workflows

Ausführungsdauer

Ebenso können Workflows nach ihrer Ausführungsdauer klassifiziert werden. Man unterscheidet dabei zwischen lang- und kurzlaufenden Workflows.

Die Ausführungszeit langlaufender Workflows reicht von einigen Minuten bis zu mehreren Monaten. Typischerweise findet dabei eine Benutzerinteraktion statt. Bei der Ausführung eines langlaufenden Workflows navigiert das Workflowsystem durch das Workflowmodell und ruft die mit einer Aktivität verknüpfte Geschäftsfunktion auf.

Die Ausführungszeit kurzlaufender Workflows beträgt dagegen wenige Sekunden bzw. Minuten. Für gewöhnlich laufen die Workflows vollautomatisch ohne Benutzerinteraktion ab. Infolge der kurzen Ausführungszeit führt in diesem Fall eine navigierende Ausführung durch das Workflowsystem zu Leistungseinbußen. Deshalb bietet ein Workflowsystem hier die Möglichkeit an, kurzlaufende Workflows in Programmcode, der zur Laufzeit direkt ausgeführt werden kann, zu übersetzen. Auf diese Weise ist eine effiziente Ausführung kurzlaufender Workflows gewährleistet.

Modellierungsstil

Des Weiteren lassen sich datenintensive Workflows anhand ihres Modellierungsstiles einteilen. Wie bereits zu Beginn von Teilkapitel 2.1 diskutiert, kann ein BPEL/SQL-Workflow sowohl prozedural als auch graphenbasiert modelliert werden. Für eine prozedurale Modellierung stehen beispielsweise die Aktivitätstypen <sequence>, <if>, <forEach>, <while> und <repeatUntil> zur Verfügung. Eine graphenbasierte Modellierung wird dagegen durch eine <flow> Aktivität unterstützt.

Transaktionsmodell

Ein weiteres Klassifikationsmerkmal liefern die beiden Transaktionsmodelle für Workflows [LR00, Ley97]:

Mit einer *atomaren Sphäre* können eine Menge von Aktivitäten bzw. ihre assoziierten Implementierungen innerhalb einer verteilten ACID-Transaktion atomar ausgeführt werden. Dabei werden die Effekte einer erfolgreich ausgeführten Aktivität nur sichtbar, wenn alle Aktivitäten der atomaren Sphäre erfolgreich ausgeführt worden sind. Dagegen werden im Fehlerfall die Effekte bereits erfolgreich ausgeführter Aktivitäten automatisch wieder rückgängig gemacht. Dieses Konzept lässt sich beispielsweise mithilfe eines verteilten Zwei-Phasen-Commit Protokolls [GR93] realisieren. Voraussetzung hierfür ist eine kurze Ausführungsdauer einer atomaren Sphäre, da sonst eine verteilte Transaktion durch Sperren, die auf den beteiligten Ressourcen bis zum Transaktionsende gehalten werden müssen, nicht mehr effizient realisiert werden kann. Aus diesem Grund ist eine Optimierung der Datenverarbeitung innerhalb einer atomaren Sphäre besonders gewinnbringend, da hierdurch die Ausführungsdauer der Sphäre entscheidend reduziert werden kann.

In Geschäftsprozessen laufen auch langlaufende Transaktionen ab, deren Ausführungszeit sich im Bereich von Minuten bis zu Monaten bewegt. Atomare Sphären sind hierfür ungeeignet. Zur Modellierung langlaufender Transaktionen stehen sogenannte *Kompensationssphären* zur Verfügung. Aktivitäten bzw. ihre assoziierten Implementierungen können in einer Kompensationssphäre in individuellen Transaktionen ausgeführt werden. Im Fehlerfall können die Effekte bereits erfolgreich abgeschlossener Transaktionen durch kompensatorische Transaktionen wieder rückgängig gemacht werden. Die kompensatorischen Transaktionen werden mit kompensatorischen Aktivitäten, die mit einer Aktivität in einer Kompensationssphäre assoziiert sind, modelliert. Beim Zurücksetzen einer Kompensationssphäre erfolgt die Ausführung der kompensatorischen Aktivitäten aller bisher ausgeführten Aktivitäten einer Kompensationssphäre in umgekehrter Reihenfolge. Alternativ kann eine einzelne Kompensationsaktion ausgeführt werden, die der Kompensationssphäre direkt zugeordnet ist.

Definition einer `<sql>` Aktivität

Die Definition einer `<sql>` Aktivität bietet ein weiteres Klassifikationsmerkmal für BPEL/SQL-Workflows. Es ist möglich, dass wichtige Ausführungsparameter einer `<sql>` Aktivität zur Modellierungszeit noch nicht vollständig vorliegen. Dies betrifft zum einen Datenbanksysteme, auf denen eine SQL-Anweisung ausgeführt wird, und zum anderen Parameter, die innerhalb einer SQL-Anweisung referenziert werden. Bei einer logischen Referenzierung eines Datenbanksystems erfolgt erst zum Deploymentzeitpunkt des Workflows die Bindung der logischen Referenz an ein physisches Datenbanksystem. Da überdies die Werte von BPEL-Variablen erst zur Laufzeit eines Workflows vorliegen, können folglich BPEL-Variablen, die als Eingabeparameter für SQL-Anweisungen genutzt werden, erst zu diesem Zeitpunkt an die SQL-Anweisung gebunden werden.

Organisation der Datenebene

Die Organisation der Datenebene, die einem BPEL/SQL-Workflow zu Grunde liegt, liefert ein weiteres Klassifikationsmerkmal: Die aus einem Workflow heraus referenzierten Daten können sowohl von einem einzelnen als auch von mehreren Datenbanksystemen verwaltet werden. Im ersten Fall greifen alle <sql> Aktivitäten auf dasselbe Datenbanksystem zu. Dabei kann es sich um ein zentralisiertes oder um ein föderiertes Datenbanksystem, das unterschiedliche Datenquellen virtualisiert, handeln. Dadurch ist sichergestellt, dass alle SQL-Anweisungen in den <sql> Aktivitäten immer gegen ein einzelnes Datenbankschema und im selben SQL-Dialekt definiert werden. Im zweiten Fall können die <sql> Aktivitäten auf mehreren autonomen Datenbanksystemen, die vollständig unabhängig voneinander arbeiten, ausgeführt werden. Bei den Datenbanken kann es sich sowohl um homogene als auch um heterogene Systeme handeln. Eine Heterogenität spiegelt sich in den <sql> Aktivitäten, deren SQL-Anweisungen in unterschiedlichen SQL-Dialekten bzw. gegen unterschiedliche Datenbankschemata definiert werden, wider.

Kurz- und langlaufende BPEL/SQL-Workflows

Basierend auf diesen allgemeinen Klassifikationsmerkmalen lassen sich zwei Arten von BPEL/SQL-Workflows unterscheiden [IBMf]: Bei einem *kurzlaufenden BPEL/SQL-Workflow* handelt es sich um einen A2A-Workflowtypen, der vollständig rechnergestützt abläuft, mit dem Ziel, große relationale Datenmengen in kürzester Zeit höchsteffizient zu verarbeiten. Folglich darf hier die Workflowausführung nicht durch eine Benutzerinteraktion bzw. durch die Ausführung asynchroner Kommunikationsmuster unterbrochen werden. Des Weiteren kann ein kurzlaufender BPEL/SQL-Workflow als atomare Sphäre interpretiert werden, in der alle definierten <sql> Aktivitäten in einer kurzlaufenden ACID-Transaktion ausgeführt werden.

Diese Eigenschaften haben auch unmittelbare Auswirkungen auf die Aktivitätstypen und Konstrukte, die in kurzlaufenden BPEL/SQL-Workflows verwendet werden dürfen: Da keine asynchrone Kommunikation erlaubt ist, können weder <EventHandler> zur Behandlung externer, asynchroner Ereignisse noch asynchrone <invoke> Aktivitäten eingesetzt werden. Ebenso kann eine <receive> bzw. eine <reply> Aktivität nur am Anfang bzw. am Ende der Ausführungslogik definiert werden. Ferner darf die Workflowausführung durch keine <wait> Aktivität, welche die Ausführungszeit eines Workflows um eine beliebige Zeitdauer bzw. bis zu einem beliebigen Zeitpunkt verzögern kann, unterbrochen werden. Da keine Benutzerinteraktion stattfinden darf, ist die Verwendung einer BPEL-Erweiterung wie BPEL4People nicht zulässig. Wegen der transienten Eigenschaft eines kurzlaufenden Workflows können darüber hinaus keine <CompensationHandler> eingesetzt werden, weil die kompensatorische Recovery-Funktionalität für <scope> Aktivitäten hier nicht zur Verfügung steht.

In einem *langlaufenden BPEL/SQL-Workflow* gelten dagegen diese Restriktionen nicht. Insbesondere ist hier eine Benutzerinteraktion möglich. Außerdem können die im BPEL-Standard definierten Aktivitätstypen und Konzepte uneingeschränkt genutzt werden. Im Gegensatz zu einem kurzlaufenden BPEL/SQL-Workflow übernimmt eine <scope> Aktivität in einem langlaufenden BPEL/SQL-Workflow die Rolle einer Kompensationss-

phäre, in der alle definierten Aktivitäten in eigenen Transaktionen ausgeführt werden. Vor allem ist hier eine Aktivität mit einer entsprechenden kompensatorischen Aktivität zu assoziieren, die im Fehlerfall die Effekte der zuvor erfolgreich ausgeführten Aktivität wieder rückgängig macht. Zur Bündelung mehrerer <sql> Aktivitäten zu einer einzelnen Transaktion muss eine <atomicSQLSequence> Aktivität, mit deren Hilfe die Transaktionsgrenzen von <sql> Aktivitäten innerhalb langlaufender Transaktionen benutzerspezifisch definiert werden können, verwendet werden.

2.1.3. Optimierungspotential

Durch die Integration mengenorientierter Verarbeitungsstrukturen in BPEL vereinfacht sich die Modellierung datenintensiver Workflows erheblich. SQL-Anweisungen können direkt in ein Workflowmodell eingebettet werden, ohne dass entsprechende Funktionalität auf der Funktionsebene implementiert werden muss. Die Spezifizierung der auszuführenden SQL-Anweisung in einer <sql> Aktivität ist hierfür ausreichend.

Jede Ausführung einer <sql> Aktivität verursacht Ausführungskosten auf der Workflow- und Datenebene. Die in einer <sql> Aktivität gekapselte SQL-Anweisung muss vom Workflow- an das Datenbanksystem, wo sie optimiert und ausgeführt wird, übertragen werden. Im Falle einer SQL-Anfrage wird eine Ergebnismenge an das Workflowsystem zurückgesendet, wo diese Ergebnismenge materialisiert und als XML-RowSet-Datenstruktur in einer Set-Variablen zur weiteren lokalen Verarbeitung zur Verfügung gestellt wird.

Um die Laufzeit eines BPEL/SQL-Workflows zu reduzieren, müssen folglich insbesondere die Ausführungskosten der darin eingebetteten <sql> Aktivitäten minimiert werden. Dies lässt sich erreichen, indem suboptimal modellierte Datenverarbeitungoperationen bzw. Datenverarbeitungsmuster, die auf <sql> Aktivitäten basieren, identifiziert und durch effizientere Strukturen ersetzt werden. Suboptimale Datenverarbeitungsmuster können von einem zu Grunde liegenden Datenbanksystem nicht so effizient ausgeführt werden, wie dies eigentlich möglich wäre.

Die Modellierung suboptimaler Datenverarbeitungsstrukturen kann verschiedene Ursachen haben. Langjährige Erfahrungen im Bereich der Datenbanksprachen haben gezeigt, dass Datenbank- und Programmiersprachenkonzepte im Allgemeinen nicht gut zusammenpassen [Mit95]. Diese Diskrepanz (impedance mismatch) drückt sich in einem mengenbasierten Datenzugriff mittels einer Datenbanksprache und einer tupelbasierten Verarbeitung in einer Programmiersprache aus. Sie wird in den meisten Fällen durch eine Fehlanpassung der Datentypen von Datenbank- und Programmiersprachen weiter verstärkt.

Eine solche Diskrepanz lässt sich ebenfalls in BPEL/SQL feststellen. BPEL bietet prozedurale Verarbeitungsstrukturen, die mit den mengenbasierten Verarbeitungsstrukturen der SQL-Spracherweiterung kombiniert werden. Darüber hinaus werden auf der Workflowebene XML- mit SQL-basierten Datenkonstrukten vermischt. Folge der Diskrepanz sind ineffiziente, prozedurale Datenverarbeitungsmuster auf der Workflowebene, die eine effiziente Datenverarbeitung auf der Datenebene verhindern.

Ebenso entsteht ein Optimierungspotential durch die Definition suboptimaler SQL-Anweisungen in einer Workflowbeschreibung. Dies liegt häufig daran, dass Entwickler nicht die komplette Mächtigkeit einer Anfragesprache ausnutzen bzw. auf einfache und vielfach erprobte Anfragen zurückgreifen, um die Robustheit einer Anwendung zu erhöhen.

Ein schlechter Datenbankentwurf kann ebenfalls zu erheblichen Leistungseinbußen führen, da sich hierdurch die Anzahl der zur Datenverarbeitung notwendigen SQL-Anweisungen bzw. <sql> Aktivitäten erhöhen kann. Insbesondere können aufgrund unnötiger Verbundoperationen komplexere und damit für ein Datenbanksystem ineffizienter ausführbare SQL-Anweisungen entstehen. Dadurch erhöht sich die Anzahl der Datenbanksystemaufrufe, um die SQL-Anweisungen abzusetzen, und steigt der Verarbeitungsaufwand sowohl auf Seiten eines Workflow- als auch auf Seiten eines Datenbanksystems. Eine höhere Anzahl an Datenbanksystemaufrufen hat einen höheren Kommunikationsaufwand und ein höheres Datenvolumen, das zwischen Workflow- und Datenebene auszutauschen ist, zur Folge. Somit ist ein guter Datenbankentwurf Voraussetzung für eine effiziente Modellierung der Datenverarbeitung in einem BPEL/SQL-Workflow.

Das PGM/F-System wendet verschiedene Optimierungstechniken an, um suboptimale Datenverarbeitungsstrukturen in einem datenintensiven Workflowmodell zu beseitigen. Dabei wird insbesondere das Wissen über die Ausführungslogik eines Workflowmodells, das einem einzelnen Datenbanksystem nicht zur Verfügung steht, genutzt.

Zu diesen Optimierungstechniken zählen beispielsweise die Kombination bzw. Parallelisierung von SQL-Anweisungen sowie das Zusammenfassen von SQL-Anweisungen, Web-Service-Aufrufen und Zuweisungsoperationen. Darüber hinaus können Kontrollflussmuster in effizientere Strukturen umgewandelt werden. Ebenso kann durch eine geeignete Optimierung der SQL-Anfragen eine Minimierung der von einem Datenbank- an ein Workflowsystem zu übertragenden Ergebnismengen erreicht werden. Des Weiteren kann es gewinnbringend sein, SQL-Anweisungen mithilfe von Stored-Procedures von der Workflow- auf die Datenebene zu verlagern. Dazu werden geeignete Datenverarbeitungsoperationen in einer Workflowbeschreibung durch entsprechende Stored-Procedures ersetzt, welche die zuvor auf der Workflowebene definierten Datenverarbeitungsoperationen direkt auf der Datenebene effizient ausführen können.

Mit diesen Optimierungstechniken können Laufzeitgewinne auf der Workflow- und Datenebene erzielt werden. Beispielsweise lassen sich durch eine Verringerung der Anzahl von <sql> Aktivitäten Ausführungskosten auf der Workflowebene einsparen. Damit verbunden ist eine Reduzierung der Anzahl von Datenbanksystemaufrufen und von Nachrichten, die zwischen einem Workflow- und einem Datenbanksystem ausgetauscht werden müssen. Durch eine Kombination von SQL-Anweisungen auf der Workflowebene kann sich auf der Datenebene die Ausführung einer kombinierten SQL-Anweisung im Vergleich zu den Einzelausführungen der SQL-Anweisungen beschleunigen. Dies insbesondere vor dem Hintergrund, dass eine kombinierte SQL-Anweisung typischerweise ein höheres Optimierungspotential aufweist, das von einem Anfrageoptimierer effektiver genutzt werden kann. Können zugleich die Größen von Anfrageergebnismengen verringert werden, reduzieren sich die Kosten zur Übertragung und zur Materialisierung dieser Datenmengen. Dadurch können weitere positive Laufzeitgewinne erreicht werden.

Wie effektiv die Optimierungstechniken auf einen BPEL/SQL-Workflow angewendet werden können, hängt entscheidend von den charakteristischen Eigenschaften eines datenintensiven Workflows ab.

Bei PGM/F erfolgt die Optimierung zum Modellierungs- bzw. Deploymentzeitpunkt eines datenintensiven Workflows (siehe Teilkapitel 3.3). Deshalb können die Optimierungstechniken nur dann korrekt angewendet werden, wenn die Kontrollflussabhängigkeiten in einem BPEL/SQL-Workflow statisch ermittelt werden können. Dies ist lediglich bei prozedural modellierten Workflowmodellen möglich, bei denen Kontrollflussabhängigkeiten zwischen den Aktivitäten unmittelbar aus der Semantik einer Kontrollflussstruktur abgeleitet werden können. Beispielsweise definiert eine Sequenz eine „wird ausgeführt vor“ Semantik zwischen sequentiell ausgeführten Aktivitäten. Das gleiche gilt für Schleifen oder parallel bzw. alternativ ausgeführte Kontrollflussstrukturen.

Dagegen liegen im Falle eines graphenbasierten Workflowmodells die Kontrollflussabhängigkeiten zwischen den Aktivitäten erst zur Laufzeit vor. Dies liegt an dem Ausführungsmodell, das einem graphenbasierten Workflow zu Grunde liegt [LR00]. In einem solchen Workflowmodell wird die Ausführungsreihenfolge der Aktivitäten mit Kontrollflusskanten explizit definiert. Eine Kontrollflusskante ist mit einem Prädikat, der sogenannten Transitioncondition, assoziiert. Deren Wahrheitswert bestimmt, ob der Kontrollfluss an eine folgende Aktivität weitergeleitet wird oder nicht.

Transitionconditions können erst zur Laufzeit evaluiert werden, weil sie typischerweise von Laufzeitdaten abhängen. Somit ist zum Modellierungs- bzw. Deploymentzeitpunkt eines Workflows nicht feststellbar, welche der modellierten Kontrollflusspfade zur Laufzeit ausgeführt werden. Insbesondere können die zur Laufzeit im Rahmen einer Dead-Path-Elimination deaktivierten Aktivitäten statisch nicht ermittelt werden. Folglich können die oben ausgeführten Optimierungsmaßnahmen nicht auf graphenbasierte Workflowmodelle angewendet werden, ohne die ursprüngliche Ausführungssemantik eines solchen Workflowmodells zu ändern. Ausnahmen bestehen bei Workflowmodellen, deren Teile direkt in entsprechende prozedurale Konstrukte umgewandelt werden können bzw. in denen triviale Transitionconditions (mit den Konstanten `True` oder `False`) definiert werden.

Des Weiteren muss beachtet werden, dass sich die Optimierungsmaßnahmen auf einen kurz- bzw. langlaufenden BPEL/SQL-Workflow unterschiedlich auswirken können. Beispielsweise kann durch das Verschmelzen zweier SQL-Anweisungen die ursprüngliche Transaktionssemantik eines langlaufenden BPEL/SQL-Workflows geändert werden. Hierdurch kann nicht mehr gewährleistet werden, dass die Ausführung einer kombinierten SQL-Anweisung zum selben Ergebnis führt, wie die Ausführung der beiden ursprünglichen SQL-Anweisungen. Ebenso sind bei einer Verschmelzung von SQL-Anweisungen die zugehörigen Kompensationsmaßnahmen anzupassen, was infolge der komplizierten Ausnahmebehandlung von BPEL/SQL sehr schwierig und deshalb nicht immer möglich ist. Erschwerend kommt hinzu, dass alle Konzepte und Aktivitätstypen des BPEL-Standards bei einer Optimierungsentscheidung zu berücksichtigen sind. Dadurch können die Optimierungsmaßnahmen im Allgemeinen nur unter einem sehr hohen Aufwand auf einen langlaufenden BPEL/SQL-Workflow angewendet werden. Hierdurch reduziert sich gleichzeitig das Optimierungspotential, das in einem solchen Workflow gewinnbringend genutzt

werden kann. Im Falle eines kurzlaufenden BPEL/SQL-Workflows können dagegen diese Optimierungsmaßnahmen wesentlich einfacher angewendet werden. Dies liegt zum einen an der eingeschränkten Nutzung bestimmter Konzepte und Aktivitätstypen des BPEL-Standards und zum anderen an der geänderten Transaktionssemantik. Da alle Aktivitäten in einer Transaktion ausgeführt werden, müssen bei der Optimierung wegen der transienten Eigenschaft einer kurzlaufenden Transaktion keine Kompensationsmaßnahmen berücksichtigt werden. Damit bleibt die ursprüngliche Transaktionssemantik auch nach Anwendung der Optimierungsmaßnahmen erhalten.

Die Optimierungsgewinne, welche durch die Anwendung der Optimierungstechniken von PGM/F erzielt werden können, liegen im Bereich weniger Millisekunden bzw. mehrerer Minuten (siehe Teilkapitel 7.3.1). Dies kann bei einem kurzlaufenden BPEL/SQL-Workflow, der große relationale Datenmengen verarbeitet, zu erheblichen Laufzeitgewinnen führen. Auch verkürzen sich die Sperrzeiten auf den Tabellen, die bis zum Ende einer Workflowausführung gehalten werden müssen. Dagegen können die Optimierungsgewinne bei langlaufenden BPEL/SQL-Workflows ihre Wirkung verlieren, insbesondere dann, wenn die Ausführung eines solchen Workflows für eine gewisse Zeit unterbrochen bzw. wegen einer Benutzerinteraktion beliebig verlängert werden kann. Folglich können die Optimierungsmaßnahmen auf einen kurzlaufenden BPEL/SQL-Workflow nicht nur wesentlich effektiver, sondern auch wesentlich gewinnbringender angewendet werden als bei langlaufenden BPEL/SQL-Workflows.

Aufgrund des höheren Optimierungspotentials, dem geringeren Optimierungsaufwand und dem höheren Optimierungsgewinn steht in dieser Arbeit die Optimierung prozedural modellierter, kurzlaufender BPEL/SQL-Workflows im Mittelpunkt. Langlaufende BPEL/SQL-Workflows eignen sich weniger für den in dieser Arbeit vorgestellten Optimierungsansatz und werden folglich nicht weiter berücksichtigt.

2.2. Optimierungsansätze in Datenbanksystemen

Abbildung 2.3 illustriert eine Klassifikation existierender Ansätze zur Anfrageoptimierung, die in der Arbeitsgruppe der Abteilung Anwendersoftware am Institut für Parallele und Verteilte Systeme an der Universität Stuttgart erarbeitet und in [Sch10] bereits diskutiert wurde. Das vorliegende Teilkapitel basiert zum Teil auf dieser Publikation.

Bei dieser Klassifikation können zwei Dimensionen unterschieden werden: Einerseits lassen sich Optimierungsansätze nach der Komplexität der Optimierung einteilen. Andererseits ist die Komplexität des zu berücksichtigenden Kontrollflusses zu beachten.

Daraus ergeben sich die in Abbildung 2.3 gezeigten vier Optimierungsklassen, die in heuristische und kostenbasierte Optimierungsvarianten aufgeteilt werden können. Die vier Optimierungsklassen erstrecken sich von der Optimierung von Einzelanfragen bzw. Mengen von Einzelanfragen bis hin zur Optimierung von SQL-Anweisungssequenzen bzw. von SQL-Anweisungen, die in datenintensiven Workflows eingebettet sind.

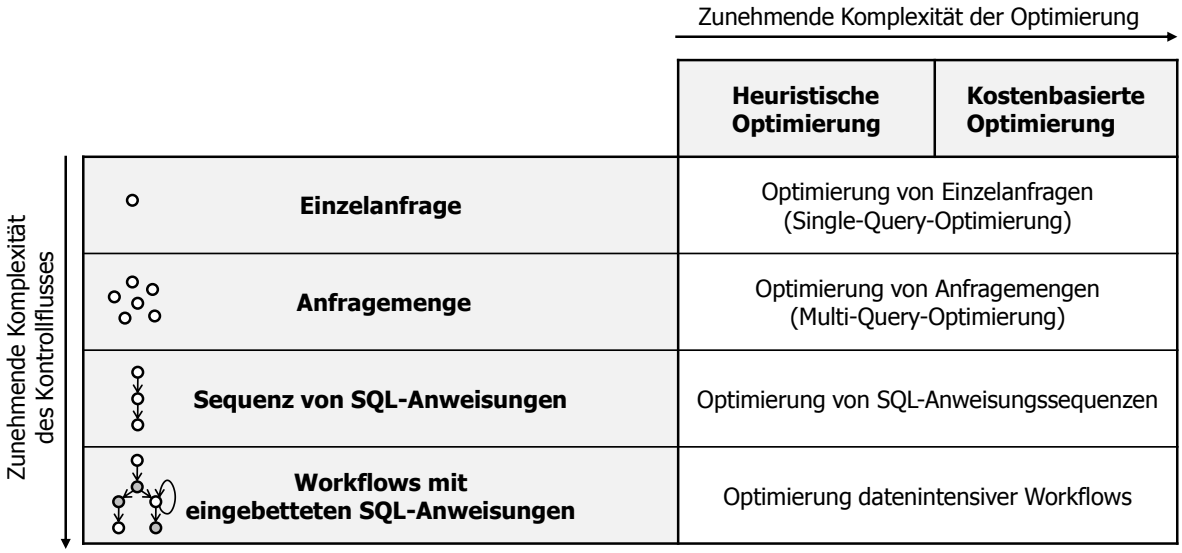


Abbildung 2.3.: Spektrum der Anfrageoptimierung

Eine Optimierungsklasse erbt sowohl die Probleme als auch die zugehörigen Optimierungstechniken ihrer vorausgehenden Optimierungsklasse und besitzt ein eigenes Optimierungspotential. Dadurch eröffnen sich neue Möglichkeiten der Optimierung.

In den folgenden Teilkapiteln werden die Optimierungsklassen kurz vorgestellt.

2.2.1. Optimierung von Einzelanfragen

Klassische Anfrageoptimierung

Bereits seit den Anfängen relationaler Datenbanksysteme in den frühen 1970er Jahren ist die Anfrageoptimierung ein zentrales Thema [Cha98, Mit95, Ioa96]. Die Anfrageoptimierung bezweckt, einen möglichst guten Ausführungsplan für eine gegebene SQL-Anfrage zu finden. Ein Ausführungsplan liegt in der Regel in Form eines Operatorbaumes vor. Dessen Knoten repräsentieren physische Operatoren, die eine Realisierung der logischen Operatoren der relationalen Algebra darstellen. Ein durch einen Anfrageoptimierer erstellter Ausführungsplan wird zur Ausführungszeit interpretiert, oder es wird daraus Code, der zur Ausführung gebracht wird, generiert. Wegen der deskriptiven Natur der Anfragesprache SQL gibt es zu einer einzigen Anfrage für gewöhnlich mehrere alternative Ausführungspläne, die sehr unterschiedliche Laufzeiten aufweisen können. Diese Pläne können sich insbesondere in der Reihenfolge der Verbundoperationen und in der gewählten Verbundimplementierung unterscheiden.

Anfrageoptimierer lassen sich in zwei Klassen unterteilen: Heuristisch arbeitende und kostenbasierte Optimierer. Bei den heuristischen Optimierern wird die SQL-Anfrage zunächst in einen initialen Operatorbaum übersetzt. Basierend auf heuristischen Regeln wird dieser restrukturiert und in einen Ausführungsplan transformiert. Zentrale Komponenten

eines heuristischen Optimierers sind die Sammlung der Restrukturierungsregeln und eine Kontrollstrategie, die über deren Anwendungsreihenfolge entscheidet. Häufig ist der kostenbasierten Optimierung auch eine heuristische Optimierung vorgeschaltet, oder beim Aufbau des Suchraums werden Heuristiken zu dessen Einschränkung angewendet.

Bei kostenbasierten Anfrageoptimierern wird ein Suchraum möglicher Operatorbäume aufgebaut. Für jeden dieser alternativen Bäume wird ein Kostenwert berechnet und der Operatorbaum mit den geringsten Kosten ausgewählt. Die kostenbasierten Optimierer werden infolge ihrer Vorgehensweise bei der Optimierung in zwei Gruppen unterteilt: Bottom-Up- und Top-Down-Optimierer.

Bottom-Up-Optimierer beruhen auf dem Prinzip der Dynamischen Programmierung [Bel03], bei der die Lösung von Teilproblemen zur Lösung eines Gesamtproblems genutzt werden kann. Hierbei werden zunächst die günstigsten Pläne für jede Verbundkombination aus zwei Tabellen berechnet. Diese werden dazu verwendet, den günstigsten Plan für alle Verbundkombinationen aus drei Tabellen zu berechnen, wobei die suboptimalen Pläne für die Verbundkombinationen aus zwei Tabellen gar nicht mehr berücksichtigt werden müssen. Aus den bisher berechneten Plänen werden die günstigsten Pläne für Verbundkombinationen aus vier Tabellen ermittelt usw.. System R und dessen Nachfolgesysteme sowie kommerzielle Datenbankprodukte von IBM sind typische Vertreter dieser Vorgehensweise [SAC$^+$79].

Bei den Top-Down-Optimierern wird eine SQL-Anfrage in einen initialen Operatorbaum mit den logischen Operatoren der relationalen Algebra als Knoten umgewandelt. Mithilfe der Tiefensuche wird der initiale Operatorbaum durchlaufen. Dabei werden durch Umformung alternative Pläne erzeugt. Man unterscheidet hierbei zwischen logischer und physischer Optimierung.

In der logischen Optimierung werden auf der relationalen Algebra beruhende Restrukturierungsregeln auf den logischen Operatorbaum angewendet. Dadurch entstehen neue, alternative, logische Operatorbäume. Die mit den Restrukturierungsregeln verfolgten Ziele sind vielfältig. Wichtige Beispiele sind das Modifizieren der Verbundreihenfolge, das Vertauschen von Verbund- und Gruppierungsoperationen, das frühzeitige Ausführen von Selektionen und Projektionen sowie das Reduzieren der Anfrageblöcke [RGL90, CS94, MP94, CS95, YL95, CS96].

In der physischen Optimierung werden die logischen Operatoren in physische Operatoren transformiert. Dabei gibt es für einen logischen Operator meist mehrere physische Operatoren, die unterschiedliche Implementierungen des logischen Operators darstellen. Der mit physischen Operatoren versehene Operatorbaum stellt die Grundlage für die Kostenbewertung dar. In diese fließen statistische Informationen über die zu Grunde liegenden Daten ein. Hierzu gehören die Anzahl der Tupel in einer Tabelle ebenso wie die Werteverteilung für die einzelnen Attribute [PIHS96]. Die Angaben zur Werteverteilung werden dazu verwendet, die Selektivitäten der Selektionen und Verbünde im Operatorbaum zu berechnen [IC93, IP95a, IP95b, PI97, Cha98, Ioa03]. Die Selektivitäten werden in Kombination mit der Kardinalität der Tabellen dazu genutzt, die Ausgabekardinalität der einzelnen Operatoren des Operatorbaums abzuschätzen. Mithilfe der Eingabekardinalitäten

und weiterer Statistiken werden die Kosten (IO- und CPU-Kosten) für die physischen Operatoren berechnet und im Operatorbaum zu einem Gesamtkostenwert akkumuliert.

Die Top-Down-Optimierer unterscheiden sich insbesondere in der Art und Weise, Ausführungspläne zu generieren. In einigen wird zuerst die logische Optimierung für den kompletten Operatorbaum durchgeführt und erst wenn diese abgeschlossen ist, mit der physischen Optimierung begonnen [HFLP89]. Bei anderen Top-Down-Optimierern wird während der Traversierung innerhalb des aktuellen Teilbaums zunächst die logische und anschließend die physische Optimierung durchgeführt bevor zum nächst höheren Operator aufgestiegen wird [GD87, GM93, Gra95].

Anfrageoptimierung über Web-Services

Aktuellere Forschungsarbeiten konzentrieren sich auf das Problem der Anfrageoptimierung über Web-Services, die einzelne Anfragen kapseln bzw. relationale Datenmengen als Ergebnis liefern können.

Beispielsweise erlaubt Active-XML (AXML) [ABC$^+$04, ABM08] eine direkte Einbettung von Web-Service-Aufrufen in ein XML-Dokument. Um eine (XPath/XQuery) Anfrage über einem AXML-Dokument effizient ausführen zu können, muss zunächst bestimmt werden, welche der eingebetteten Web-Service-Aufrufe Ergebnisse zur Beantwortung einer Anfrage liefern. Insbesondere sind irrelevante Web-Service-Aufrufe zu vermeiden, um eine Verlängerung der Ausführungszeit einer Anfrage zu verhindern. In einem nächsten Optimierungsschritt wird eine geeignete Aufrufsequenz für die relevanten Web-Services ermittelt. Dabei müssen Datenabhängigkeiten zwischen den einzelnen Web-Services berücksichtigt werden. Die Web-Services werden entsprechend der Ausführungsreihenfolge aufgerufen, und ihre Ergebnisse werden materialisiert in das AXML-Dokument eingefügt. Anschließend erfolgt die Ausführung der Anfrage über dem zuvor erzeugten AXML-Dokument.

In [SMWM06] wird die Optimierung von Select-Project-Join (SPJ) Anfragen über einer Menge von Web-Services betrachtet. Jeder dieser Web-Services kapselt einzelne Anfragen. Eine SPJ-Anfrage wird zunächst in eine Sequenz von Web-Services, mit denen die SPJ-Anfrage beantwortet werden kann, übersetzt. Jeder Web-Service in der Sequenz produziert ein Zwischenergebnis, das von seinem nachfolgenden Web-Service als Eingabe genutzt wird. Auf diese Weise wird das Ergebnis der SPJ-Anfrage berechnet. Die Optimierung besteht darin, die Sequenz derart umzubauen, dass die Antwortzeit der SPJ-Anfrage minimiert werden kann. Hierfür werden verschiedene Ausführungspläne erstellt, die sich in der Reihenfolge der Web-Service-Aufrufe und dem Grad der Parallelisierung zwischen den einzelnen Web-Service-Aufrufen unterscheiden. Dabei müssen Datenabhängigkeiten zwischen den Web-Services berücksichtigt werden. Mithilfe eines Kostenmodells wird der für die SPJ-Anfrage optimale Ausführungsplan bestimmt. Das Kostenmodell unterstellt, dass die Selektivitäten bzw. die Antwortzeiten der in den einzelnen Web-Services gekapselten Anfragen bekannt und konstant sind.

2.2.2. Optimierung von Anfragemengen

Als Multi-Query-Optimierung wird eine Erweiterung der klassischen Anfrageoptimierung bezeichnet. Bei diesem Ansatz wird eine Anfragemenge, die mehrere meist komplexe Anfragen umfasst, gemeinsam optimiert. Das Optimierungsziel besteht darin, die Ausführungskosten für die Anfragemenge insgesamt zu optimieren. Dies kann insbesondere auch bedeuten, dass einzelne Anfragen einer Anfragemenge weniger effizient ausgeführt werden, als dies bei isolierter Anfrageoptimierung der Fall wäre. Eine der grundlegenden Arbeiten zur Architektur eines solchen Optimierers findet sich in [RC88].

Die Vorteile der Multi-Query-Optimierung gegenüber der klassischen Anfrageoptimierung resultieren aus der Identifizierung gemeinsamer Teilausdrücke, die nur einmalig ausgeführt werden. Das berechnete Ergebnis wird materialisiert und bei der Ausführung der einzelnen Anfragen einer Anfragemenge verwendet [Fin82, Sel86, Sel88, PS88, RSSB00]. Werden in einer Anfragemenge gemeinsame Teilausdrücke identifiziert, profitiert die einzelne Anfrage nicht notwendigerweise davon. Für die Anfragemenge insgesamt ist allerdings eine wesentlich effizientere Ausführung zu erwarten.

Eine vergleichbare Technik zur Multi-Query-Optimierung wird bei der Bereitstellung materialisierter Sichten genutzt (siehe z.B. [LY85, CKPS95, LMS95, LSPC00, ZCL+00, GL01, MRSR01, FGW+05]). Materialisierte Sichten stellen partiell vorausberechnete Anfrageergebnisse bereit, auf die zukünftige Anfragen unmittelbar zugreifen können. Somit kann ein Teil des Aufwandes zur Verarbeitung der Anfragen vermieden werden.

Neben der Materialisierung von Daten ist die intelligente Auswahl der geeigneten Zugriffspfade auf Tabellen ein entscheidendes Mittel, um Ausführungszeiten von Anfragen zu verkürzen. Es wurde gezeigt, dass man die Entscheidungen darüber, welche Ergebnisse man materialisieren sollte und welche Zugriffspfade auf den Daten angelegt werden sollten, nicht isoliert betrachten darf [GL01].

Ein weiterer wichtiger Einsatzbereich für die Techniken der Multi-Query-Optimierung ist die Verarbeitung von Datenströmen. Hier werden Anfragen an einen oder mehrere Datenströme formuliert. Die Anfragen verbleiben typischerweise längerfristig im System, weshalb in der Regel von Continuous Queries gesprochen wird [AH00, TGNO92, CDTW00, BW01, MSHR02]. Diese stellen immer dann ein Anfrageergebnis zur Verfügung, wenn einer der Datenströme Daten, die sich für die Ergebnismenge der Anfrage qualifizieren, liefert. In diesem Fall soll das aktuelle Anfrageergebnis ausgegeben werden. Eine Herausforderung in diesem Bereich besteht in der effizienten Verwaltung der Anfragen, weil in einem System zur Verarbeitung von Datenströmen typischerweise viele Anfragen gleichzeitig aktiv sind. Bei der Verarbeitung der Anfragen muss für alle neuen Daten eines Datenstroms überprüft werden, welchen Beitrag sie zur Ergebnismenge der einzelnen Anfragen leisten. Dieser ressourcenintensive Verarbeitungsschritt kann mit Techniken der Multi-Query-Optimierung effizienter gestaltet werden. Hierzu werden gemeinsame Teilausdrücke in den Anfragen identifiziert, und durch den Optimierer wird dafür gesorgt, dass diese nicht mehrfach verarbeitet, sondern einmal gewonnene Teilergebnisse für unterschiedliche Anfragen genutzt werden.

2.2.3. Optimierung von Anweisungssequenzen

Die Ausführung einer Sequenz von SQL-Anweisungen erfolgt in einzelnen Schritten. In der Reihenfolge, in der die einzelnen Anweisungen in der Sequenz definiert sind, werden sie an ein zu Grunde liegendes Datenbanksystem zur Ausführung übergeben. Die Ausführung erfolgt vollständig sequentiell, d.h. zu jedem Zeitpunkt führt das Datenbanksystem genau eine der Anweisungen einer Sequenz aus. Für das Datenbanksystem besteht damit jeweils nur die Möglichkeit, eine Anweisung zu optimieren. Bei der Optimierung können somit weder die Abhängigkeiten zwischen den Anweisungen einer Sequenz noch mehrfach auftretende Anfrageteile berücksichtigt werden. An dieser Stelle setzen Ansätze zur Optimierung von Anweisungssequenzen an.

Der QUEL*-Optimierer [SS91] ist einer der ersten Ansätze, bei dem eine Sequenz von Anweisungen optimiert wird. QUEL* ist eine relationale Datenbanksprache, vergleichbar mit SQL, die im Rahmen des Ingres-Projekts entwickelt wurde. Beim QUEL*-Optimierer kommen sowohl Techniken aus dem Bereich der Anfrageoptimierung als auch aus dem Bereich des Compilerbaus zum Einsatz. Hierzu zählen: (i) Das Herausziehen von Invarianten in iterativ ausgeführten Anweisungen, um redundante Berechnungen zu vermeiden; (ii) die Identifizierung gemeinsamer Ausdrücke in Anweisungen, um Zwischenergebnisse nicht mehrfach berechnen zu müssen; (iii) und das Verschmelzen von Aktualisierungsoperationen mit gemeinsamen Prädikaten.

Aktuelle Arbeiten im Bereich der automatischen Auswahl eines physischen Datenbankentwurfs betrachten ebenfalls Sequenzen von Anweisungen [ACN06]. Diese Ansätze beschleunigen die Ausführungsgeschwindigkeit einer Sequenz, indem neue Anweisungen der Sequenz, die Indexstrukturen erzeugen bzw. löschen, hinzugefügt werden. Typischerweise werden Indexstrukturen erzeugt, bevor Anfragen eintreffen, und sie werden wieder entfernt, bevor Änderungsoperationen ausgeführt werden. Auf diese Weise werden Kosten, die für die Aktualisierung einer Indexstruktur bei der Ausführung einer Änderungsoperation anfallen, eingespart.

Einen weiteren Ansatz stellt die Optimierung von Anweisungssequenzen mit der sogenannten Coarse-Grained-Optimierung (CGO) [KSRM03, KS04, Kra07, KM07, KSM07] dar. Hierbei werden Sequenzen von SQL-Anweisungen betrachtet, in denen eine Anweisung ein Zwischenergebnis erzeugt, das von nachfolgenden Anweisungen direkt genutzt werden kann. Die Zwischenergebnisse werden in temporären Tabellen, die durch entsprechende Create- und Drop-Anweisungen innerhalb der Sequenz erzeugt bzw. wieder entfernt werden, gespeichert. Diese werden von einer Folge von Insert-Anweisungen, welche die temporären Tabellen lesend bzw. schreibend referenzieren, berechnet. CGO schreibt eine solche Sequenz von SQL-Anweisungen mithilfe von Restrukturierungsregeln derart um, dass eine neue Sequenz entsteht, die von einem Datenbanksystem effizienter ausgeführt werden kann. Beispielsweise werden Regeln, die durch Verschmelzen von Anweisungen die explizite Erzeugung von temporären Tabellen in dieser Sequenz verhindern, verwendet. CGO unterstützt einen heuristischen und kostenbasierten Optimierungsansatz.

2.2.4. Optimierung datenintensiver Workflows

Die Forschung im Bereich der Optimierung datenintensiver Workflows steht erst am Anfang. Deshalb kann an dieser Stelle neben der hier vorgestellten Arbeit nur auf wenige weitere Forschungsergebnisse verwiesen werden.

Es existieren bereits Arbeiten im Bereich der ETL-Workflows, die auch in die Gruppe der datenintensiven Workflows eingeordnet werden können. Zu betonen ist, dass ETL-Workflows im Bereich des Data-Warehousing angesiedelt sind und somit keine Geschäftsprozesse realisieren, deren Optimierung im Fokus dieser Arbeit steht.

ETL-Werkzeuge (Extraction-Transformation-Loading) erlauben es, Daten aus verschiedenen Datenquellen zu extrahieren, zu bereinigen und in ein Data-Warehouse zu integrieren [KC04]. Die hierfür notwendigen Prozesse werden häufig durch ETL-Workflows definiert und ausgeführt. In einem ETL-Workflow entsprechen die Aktivitäten typischerweise logischen Operatoren, wie sie aus der relationalen Algebra [Cod72] bekannt sind.

In diesem Bereich existieren Optimierungsansätze ([SVS05a, SVS05b, SS07, DHW+08]), bei denen Techniken, die aus der logischen Anfrageoptimierung bekannt sind, auf den Kontext von ETL-Workflows übertragen werden. Bei der Durchführung einer logischen Optimierung von ETL-Workflows müssen insbesondere Daten- und Kontrollflussabhängigkeiten zwischen den einzelnen Aktivitäten berücksichtigt werden, um die Korrektheit der während der Optimierung durchgeführten Transformationen auf dem ETL-Workflow garantieren zu können.

In [Böh11] wird ein kostenbasierter Optimierungsansatz für ETL-Workflows vorgestellt, der auf der Workflowausführungsebene angewendet wird. Dabei wird der Ausführungsplan eines Workflows kontinuierlich an sich ändernde Arbeitslasten angepasst, mit dem Ziel, die Ausführungskosten eines Workflows über dessen gesamten Ausführungszeitraum hinweg zu minimieren. Dazu werden während der Workflowausführung Statistiken gesammelt, auf deren Grundlage für die aktuell vorliegenden Arbeitslasten optimale Ausführungspläne berechnet werden. Hierbei kommen beispielsweise Optimierungstechniken zum Einsatz, die zum einen den Parallelisierungsgrad des Kontrollflusses eines Ausführungsplans maximieren, und die zum anderen die Anzahl notwendiger Aufrufe externer Systeme durch eine geeignete Gruppierung der zu versendenden Nachrichten minimieren.

In [BS04] wird ein Optimierungssystem beschrieben, mit dem ein Datenstrom relationaler Anfragen optimiert werden kann, der von einer Anwendung an ein zugrunde liegendes Datenbanksystem gesendet wird. Dazu analysiert das Optimierungssystem den gesendeten Datenstrom, mit dem Ziel, wiederkehrende Muster in dem Datenstrom zu identifizieren und zu optimieren. Die Optimierung besteht darin, die in einem solchen Muster auftretenden SQL-Anfragen zu einer einzelnen Anfrage zu kombinieren. Taucht anschließend die erste SQL-Anfrage in diesem Muster im Datenstrom erneut auf, sendet das Optimierungssystem transparent für die Anwendung direkt die kombinierte SQL-Anfrage an das Datenbanksystem und speichert die resultierende Ergebnismenge lokal im Datencache ab. Auf diese Weise können alle SQL-Anweisungen in diesem Muster unmittelbar vom Optimierungssystem mithilfe des Datencaches beantwortet werden, wodurch die Ausführungszeit der SQL-Anweisungen verkürzt werden kann. Eine solche Prefetching-Strategie lässt sich ebenso auf einfache Weise auf den Kontext datenintensiver Workflows übertragen.

Heutzutage spielen Workflowsysteme auch bei der Automatisierung wissenschaftlicher Anwendungen eine zentrale Rolle. Durch die Verfügbarkeit immer leistungsfähigerer Rechner und rasant wachsender Speicherkapazitäten wird es möglich, immer komplexere Simulationen und Analysen unter Berücksichtigung riesiger Datenmengen auf einer IT-Infrastruktur auszuführen. Dabei werden solche rechen- und datenintensiven Anwendungen typischerweise mithilfe sogenannter Scientific-Workflows modelliert und auf einem Workflowsystem zur Ausführung gebracht.

Allerdings werden hierfür keinerlei Standards berücksichtigt. Vielmehr existieren viele proprietäre, domänenspezifische Lösungen zur Modellierung und Ausführung dieser speziellen Art datenintensiver Workflows. Discovery-Net [RKO+03], Taverna [OAF+04] oder Kepler [LAB+06] sind prominente Vertreter hiervon. Die Bemühungen gehen aber in die Richtung, die in der Welt der Geschäftsprozesse bewährten Technologien, wie z.B. die Workflowbeschreibungssprache BPEL, auf das Gebiet der Scientific-Workflows zu übertragen (siehe z.B. [AMA06, TDG06]).

Die Optimierung von Scientific-Workflows wurde bislang nur am Rande betrachtet. Existierende Ansätze beschränken sich insbesondere darauf, die Bereitstellung und Verarbeitung großer Datenmengen in einem Workflowsystem auf der Workflowausführungsebene zu optimieren. Optimierungsansätze auf der Workflowmodellierungsebene, die ein Modell eines Scientific-Workflows berücksichtigen, sind dagegen nicht bekannt.

2.2.5. Abgrenzung der Optimierungsansätze zu PGM/F

Mit dem in dieser Arbeit vorgestellten PGM/F-System lässt sich das in einem datenintensiven Workflow vorliegende Optimierungspotential bezüglich der darin definierten Datenverarbeitungsoperationen gewinnbringend nutzen. Dabei werden sowohl Konzepte aus der Optimierung von Einzelanfragen und Anfragemengen bzw. -sequenzen an die Problemstellung der Optimierung datenintensiver Workflows angepasst als auch neue Konzepte betrachtet. Somit verbindet PGM/F wohlbekannte Techniken der Anfrageoptimierung auf der Datenebene mit der Optimierung der in einem datenintensiven Workflow eingebetteten SQL-Anweisungen.

Optimierung von Einzelanfragen und Anfragemengen

PGM/F basiert auf wohlbekannten Konzepten der regelbasierten, heuristischen Optimierung von Einzelanfragen und Anfragemengen. Zudem erweitert PGM/F diese Ansätze und überträgt sie auf den Kontext datenintensiver Workflows. In Analogie zu einem föderierten Datenbanksystem ergänzt PGM/F die Anfrageoptimierer der Datenbanksysteme, welche für die Ausführung der SQL-Anweisungen eines datenintensiven Workflows verantwortlich sind. Insbesondere kann durch eine Optimierung der SQL-Anweisungen auf der Workflowebene das von einem zu Grunde gelegten Anfrageoptimierer auf der Datenebene nutzbare Optimierungspotential erhöht werden.

Optimierung von Anweisungssequenzen

Der Optimierungsansatz von CGO berücksichtigt Sequenzen von SQL-Anweisungen bestehend aus Insert-, Create- und Drop-Operationen. Da solche Anweisungssequenzen auch in einem datenintensiven Workflow modelliert werden können, ist es sinnvoll, die Regelmenge des CGO-Optimierers in die Regelmenge des PGM/F-Optimierers aufzunehmen (siehe Teilkapitel 5.2). Darüber hinaus ergeben sich für den PGM/F-Optimierer weitere Optimierungsmöglichkeiten, weil bei einem datenintensiven Workflow beliebige Kontrollflussstrukturen bzw. SQL-Anweisungen und zusätzliche Operationen, wie z.B. Web-Service-Aufrufe oder Variablenzuweisungen, zu berücksichtigen sind. Wegen der gesteigerten Komplexität weisen datenintensive Workflows im Vergleich zu Anweisungssequenzen ein weit höheres Optimierungspotential auf, das durch einen CGO-Optimierer alleine nicht abgedeckt werden kann. Um dieses Optimierungspotential ausschöpfen zu können, sind zusätzliche, neue Optimierungsschritte notwendig, die in dieser Arbeit vorgestellt werden.

Optimierung datenintensiver Workflows

ETL-Workflows weisen Ähnlichkeiten zu den in dieser Arbeit betrachteten datenintensiven Workflows auf, denen Geschäftsprozesse zu Grunde liegen. In beiden Fällen spielt die Einbettung von Datenverarbeitungsoperationen in komplexe Ablaufstrukturen eine wichtige Rolle. Allerdings steht kein Standard zur Modellierung von ETL-Workflows zur Verfügung. Vielmehr werden hierfür verschiedene proprietäre Lösungen angeboten.

Im Bereich der Optimierung sind weder Arbeiten bekannt, die ETL-Workflows betrachten, deren Einzeloperationen die Mächtigkeit von komplexen SQL-Anfragen haben, noch Arbeiten, welche die speziellen Charakteristika datenintensiver Workflows berücksichtigen. Stattdessen erfolgt hier die Optimierung von ETL-Workflows auf einer feingranularen Sprachebene. Dabei entsprechen die Aktivitäten eines ETL-Workflows logischen Operatoren der relationalen Algebra. Dementsprechend werden bei der Optimierung von ETL-Workflows Techniken, die von der algebraischen Anfrageoptimierung bekannt sind, eingesetzt. Dagegen spielt sich die PGM/F-Optimierung auf einer gröberen, deklarativen Sprachebene von SQL-Anweisungen ab. Aufgrund der verschiedenen sprachlichen Modellierungs- und Optimierungsebenen ergeben sich Unterschiede in den Optimierungsregeln, die in feingranularen ETL-Workflows bzw. in den von PGM/F berücksichtigten grobgranularen, datenintensiven Workflows verwendet werden.

Es ist aber zu beachten, dass neben einer feingranularen auch eine grobgranulare Modellierung von ETL-Workflows, beispielsweise mit BPEL/SQL, möglich ist. In diesem Fall kann der Optimierungsansatz von PGM/F ebenso auf ETL-Workflows übertragen werden.

Scientific-Workflows stellen eine weitere Gruppe datenintensiver Workflows dar, die komplexe Datenverarbeitungsoperationen auf großen Datenmengen ausführen. Es sind keine Arbeiten bekannt, welche dieses Optimierungspotential auf der Workflowmodellierungsebene gewinnbringend nutzen können. Bisherige Optimierungsansätze beschränken sich auf die Laufzeitumgebung eines Scientific-Workflows. Das in dieser Arbeit vorgestellte PGM/F-System kann einen Beitrag zur Optimierung von Scientific-Workflows liefern, falls diese mit einer Workflowbeschreibungssprache, wie BPEL/SQL, beschrieben werden.

3
Überblick über PGM/F

Dieses Kapitel gibt einen Überblick über das *Prozessgraphenmodell/Framework (PGM/F)*, das im Rahmen dieser Doktorarbeit zur Optimierung der in Teilkapitel 2.1 vorgestellten datenintensiven Workflows entwickelt wurde. Es stellt die Architektur, den Optimierungsansatz und die charakteristischen Eigenschaften des PGM/F-Systems vor.

In diesem System spielt das Prozessgraphenmodell eine entscheidende Rolle, weil es die Grundlage für die Definition und Anwendung der Restrukturierungsregeln auf einen datenintensiven Workflow bildet. Der Framework-Gedanke soll dabei die Flexibilität und Erweiterbarkeit des Optimierungsansatzes von PGM/F unterstreichen.

3.1. Architektur

Ziel des PGM/F-Systems ist die Optimierung der Datenverarbeitung in datenintensiven Workflows. Hierzu werden die zu einem Workflowmodell gehörenden Datenverarbeitungsoperationen und -muster unter Verwendung von Restrukturierungsregeln derart modifiziert, dass deren effizientere Verarbeitung durch ein Workflow- bzw. Datenbanksystem ermöglicht wird. Grundvoraussetzung für die Zulässigkeit solcher Modifikationen ist die Semantikerhaltung, d.h. die modifizierten Anweisungen müssen dasselbe Ergebnis liefern wie die ursprünglichen Varianten.

Die Architektur des PGM/F-Systems ist in Abbildung 3.1 dargestellt.

Die Restrukturierungsregeln werden auf einer internen Repräsentation eines Workflowmodells, dem *Prozessgraphenmodell (PGM)*, angewendet. PGM ist die Basis für die Definition und Anwendung der Restrukturierungsregeln und wird in Kapitel 4 eingeführt.

Der *PGM-Optimierer* stellt die Kernkomponente dar, der für die heuristische Benutzung der Restrukturierungsregeln auf eine PGM-Repräsentation verantwortlich ist. Diese Komponente verwendet die verfügbare *Regelbasis* und eine *Kontrollstrategie*, mit deren Hilfe die Nutzung der Regelbasis gesteuert wird.

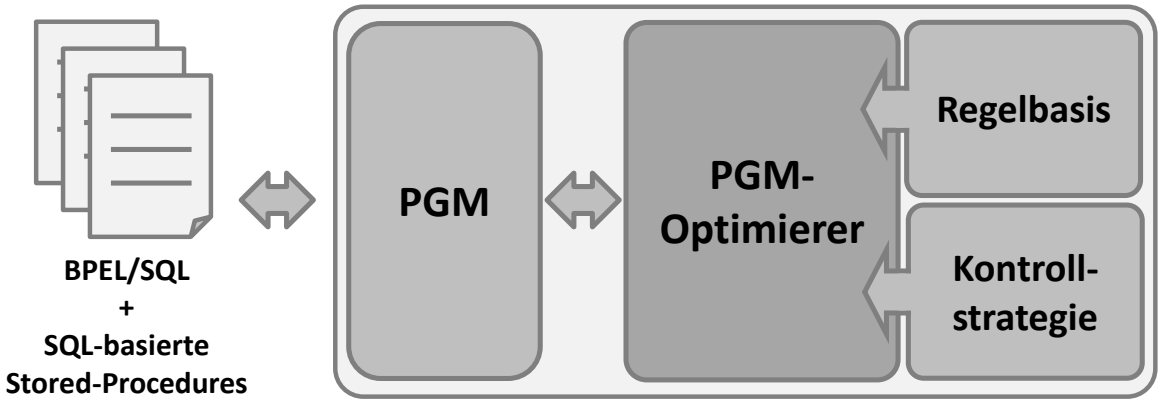


Abbildung 3.1.: Architektur von PGM/F

Jede Restrukturierungsregel besteht aus zwei Teilen: Einem *Bedingungs-* und einem *Aktionsteil*. Der Bedingungsteil definiert die Bedingungen bezüglich Daten-, Kontrollfluss- und Kommunikationsabhängigkeiten, die gelten müssen, damit eine Regel auf ein Workflowmodell korrekt angewendet werden kann. Ferner stellen diese Bedingungen sicher, dass die Ausführungssemantik des ursprünglichen Workflowmodells erhalten bleibt. Der Aktionsteil einer Regel definiert die Transformationsschritte, die auf jene Teile des Workflowmodells, welche den Bedingungsteil einer Regel erfüllen, durchgeführt werden. In Kapitel 5 wird die Regelbasis von PGM/F im Detail beschrieben.

Des Weiteren muss der PGM-Optimierer entscheiden, auf welche Teile eines Workflowmodells und in welcher Reihenfolge die Restrukturierungsregeln angewendet werden sollen. Hierfür greift er auf eine wohldefinierte Kontrollstrategie zurück. Eine Funktion der Kontrollstrategie besteht darin, sogenannte Optimierungssphären zu identifizieren. Solche Optimierungssphären entsprechen genau den Teilen eines Workflowmodells, auf welche die Regelbasis ohne Änderung der Ausführungssemantik des ursprünglichen Workflowmodells Anwendung finden kann. Eine weitere Funktion der Kontrollstrategie ist die Festlegung der Reihenfolge, in der die Regelbasis auf eine Optimierungssphäre ausgeführt werden soll. In Kapitel 6 wird die Kontrollstrategie von PGM/F vorgestellt.

3.2. Optimierungsansatz

Auf Grundlage der in Abbildung 3.1 gezeigten Architektur sieht der Optimierungsablauf von PGM/F wie folgt aus: Ein Workflowmodell wird zunächst in die interne PGM-Repräsentation transformiert. Darauf wendet der PGM-Optimierer die Restrukturierungsregeln gemäß der gewählten Kontrollstrategie an. Hierfür müssen Muster innerhalb einer PGM-Repräsentation, welche den Bedingungsteil einer Regel erfüllen, gefunden werden.

Zur Identifizierung der Muster müssen die Aktivitäten einer PGM-Repräsentation mit ihren Daten-, Kontrollfluss- und Kommunikationsabhängigkeiten überprüft werden. Dies ist sehr einfach, weil diese Informationen in einer PGM-Repräsentation explizit dargestellt werden. Muster, welche die Bedingung einer Regel erfüllen, werden gemäß der Aktion einer Regel transformiert. Zuletzt wird die sich aus dem Optimierungsprozess ergebende PGM-Repräsentation in ein entsprechendes Workflowmodell zurücktransformiert.

Die Überführung eines Workflowmodells in eine PGM-Repräsentation wird mit Transformationsregeln durchgeführt, welche die Abbildung zwischen den Konstrukten eines Workflowmodells und den Konstrukten von PGM definieren. Als Ergebnis dieses Transformationsschrittes erhält man eine PGM-Repräsentation, die alle für die PGM-Optimierung relevanten Informationen aus dem zu Grunde gelegten Workflowmodell explizit darstellt bzw. die für die PGM-Optimierung nicht relevanten Informationen verbirgt. Dieses Abstraktionsprinzip führt zu einer erheblichen Vereinfachung der Definition und Anwendung der Restrukturierungsregeln. Insbesondere können die Restrukturierungsregeln hierdurch sprachneutral definiert werden. Dadurch kann die Anwendbarkeit des Optimierungsansatzes von PGM/F auf andere SQL-spezifische Beschreibungssprachen erweitert werden. Des Weiteren vereinfacht sich die Abbildung zwischen einem Workflowmodell und der zugehörigen PGM-Repräsentation, da nicht alle Details eines Workflowmodells in PGM repräsentiert werden müssen. Allerdings gehen durch diesen Abstraktionsschritt gleichzeitig alle sprachspezifischen Informationen eines Workflowmodells, die in PGM nicht explizit dargestellt werden, verloren. Somit kann aus einer PGM-Repräsentation nicht direkt ein syntaktisch korrektes Workflowmodell abgeleitet werden. Aus diesem Grund ist es zwingend erforderlich, für jedes Element der PGM-Repräsentation die ursprüngliche Beschreibung des zugehörigen Elements des Workflowmodells zu speichern. Wenn beispielsweise BPEL/SQL als Workflowbeschreibungssprache verwendet wird, enthält jedes PGM-Element auch die BPEL/SQL-Beschreibung, die ihr zu Grunde liegt. Damit ist sichergestellt, dass das ursprüngliche Workflowmodell aus der zugehörigen PGM-Repräsentation direkt wieder rekonstruiert werden kann.

Die Erzeugung konsistenter PGM-Repräsentationen setzt korrekte Transformationsregeln voraus. Die Definitionen der Restrukturierungsregeln gewährleisten die Konsistenz einer PGM-Repräsentation nach Anwendung der Restrukturierungsregeln.

Nach dem Gebrauch der Restrukturierungsregeln wird die ursprüngliche Beschreibung des Workflowmodells aus den Elementen einer PGM-Repräsentation ausgelesen, um die neue Beschreibung des optimierten Workflowmodells zu erzeugen. Für Aktivitäten bzw. Kontrollflussstrukturen, die nicht optimiert wurden, können die Beschreibungen des ursprünglichen Workflowmodells direkt wiederverwendet werden. Da der Schwerpunkt der Restrukturierungsregeln auf der Optimierung von <sql> Aktivitäten liegt, müssen ausschließlich neue Beschreibungen für <sql> Aktivitäten bzw. für die darin definierten SQL-Anweisungen erzeugt werden. Die Beschreibungen lassen sich unmittelbar aus den vorliegenden SQL-Anweisungen in einer PGM-Repräsentation ableiten. Werden dagegen Elemente infolge einer Regelanwendung aus einer PGM-Repräsentation entfernt, verschwinden gleichzeitig die in diesen Elementen gekapselten, zugehörigen ursprünglichen Beschreibungen, die somit bei der Erzeugung des neuen Workflowmodells nicht mehr berücksichtigt werden. Auf diese Weise können alle Transformationseffekte von der PGM-Ebene wie-

der korrekt auf die Workflowmodellierungsebene übertragen werden. Dabei wird durch entsprechende Validierungsregeln sichergestellt, dass als Ergebnis der PGM-Optimierung immer ein syntaktisch, korrektes Workflowmodell geliefert wird.

Wie Abbildung 3.1 zeigt, stellen auch die Beschreibungen von SQL-basierten Stored-Procedures, die in SQL/PSM (ISO/IEC 9075-4:2003 – Persistent Stored Modules (SQL/PSM)) definiert werden, mögliche Eingaben für den PGM-Optimierer dar. SQL/PSM erweitert SQL um prozedurale Kontrollflusskonstrukte, mit denen SQL-Anweisungen verbunden werden können. Der beschriebene Optimierungsansatz kann auch auf die Definition SQL-basierter Stored-Procedures (im Folgenden nur Stored-Procedures genannt) angewendet werden, da die interne PGM-Repräsentation hinreichend generisch definiert ist. Darüber hinaus ergibt sich eine weitere Option: Stored-Procedures können aus einem datenintensiven Workflow heraus aufgerufen werden. Wenn neben der Beschreibung eines Workflowmodells die Beschreibung einer aufgerufenen Stored-Procedure vorliegt, können beide Beschreibungen in die interne PGM-Repräsentation überführt werden. Die PGM-Repräsentationen können isoliert, aber auch kombiniert optimiert werden. Im letzteren Fall wird dazu eine kombinierte PGM-Repräsentation erstellt, die sowohl das Workflowmodell als auch die einzelnen Stored-Procedures repräsentiert.

Somit können in PGM/F bei der Optimierung datenintensiver Workflows drei verschiedene Optimierungsszenarien unterschieden werden:

- Bei einer *homogenen Optimierung* wird ein Workflowmodell in die interne PGM-Repräsentation überführt und mittels der Regelbasis von PGM/F optimiert. Die Optimierung betrifft ausschließlich alle Datenverarbeitungsoperationen, die direkter Bestandteil eines Workflowmodells sind.
- In einem *heterogenen, isolierten Optimierungsszenario* berücksichtigt der PGM-Optimierer neben dem Workflowmodell die Beschreibungen der Stored-Procedures, die vom Workflow aufgerufen werden, deren Definition aber nicht Teil des Workflowmodells sind. Die PGM-Repräsentation erlaubt die Darstellung datenintensiver Workflows und Stored-Procedures. Hierdurch ergeben sich weitere Optimierungsmöglichkeiten im Vergleich zur homogenen Optimierung. Die Restrukturierungsregeln können sowohl auf die PGM-Repräsentation des Workflowmodells als auch auf die PGM-Repräsentationen der Stored-Procedures angewendet werden. Dieser Ansatz wird heterogene, isolierte Optimierung genannt, weil die Beschreibungen eines Workflowmodells und einer Stored-Procedure getrennt voneinander optimiert werden.
- Im Falle einer *heterogenen, kombinierten Optimierung* wird eine einzelne PGM-Repräsentation erstellt, die das Workflowmodell und die Definitionen der Stored-Procedures, die vom Workflow aufgerufen werden, umfasst. Eine solche integrierte PGM-Repräsentation ermöglicht die Anwendung der Restrukturierungsregeln über Sprachgrenzen hinweg. D.h. die Effekte einer Regelanwendung können sowohl ein Workflowmodell als auch die Beschreibung einer Stored-Procedure betreffen.

Im Folgenden werden die genannten Varianten, ausgehend von dem in Abbildung 2.2 gezeigten Beispiel-Workflow, veranschaulicht:

- Bei einer homogenen Optimierung werden die Restrukturierungsregeln ausschließlich auf die Beschreibung des Beispiel-Workflows angewendet. Die Beschreibung der Stored-Procedure bleibt unberücksichtigt. Abbildung 3.2 zeigt im oberen Teil die optimierte Variante des Beispiel-Workflows. Diese resultiert aus der Anwendung dreier Restrukturierungsregeln: In einem ersten Schritt wird der Aufruf der Web-Service-Operation *OrderItem* in eine entsprechende SQL-Anfrage gewandelt (*Web-Service-Pushdown*-Regel, siehe Teilkapitel 5.3). Dazu wird der Aufruf der Web-Service-Operation in Form der benutzerdefinierten Funktion *OrderItem* bereitgestellt. Anschließend wird die SQL-Anfrage in die With-Klausel der Insert-Anweisung der Aktivität *InsertOrderConfirmation* verschoben, wo das Ergebnis der SQL-Anfrage direkt referenziert werden kann (*Select-Into-Merging*-Regel, siehe Teilkapitel 5.4.6). Dieser Transformationsschritt ermöglicht, die gesamte ForEach-Schleife aufzulösen und durch eine modifizierte <sql> Aktivität *InsertOrderConfirmation* zu ersetzen (*Insert-Tuple-To-Set*-Regel, siehe Teilkapitel 5.5.3). Die hierbei angewendete Restrukturierungsregel sorgt dafür, dass die ursprünglich über eine Schleifeniteration ausgedrückte mengenorientierte Datenverarbeitung direkt in SQL definiert werden kann. Dies wird erreicht, indem die zuvor tupelbasierte Insert-Anweisung in eine entsprechende mengenbasierte Insert-Anweisung umgeschrieben wird.
- Im Falle einer heterogenen, isolierten Optimierung werden die Restrukturierungsregeln jeweils getrennt voneinander auf die Beschreibungen des Beispiel-Workflows und der Stored-Procedure *PrepareApprovedOrders* angewendet. Abbildung 3.2 zeigt den resultierenden Workflow bzw. die resultierende Stored-Procedure nach der voneinander getrennten Optimierung beider Teile. Die Optimierung des Beispiel-Workflows läuft analog zur bereits beschriebenen homogenen Optimierung ab. In der Stored-Procedure wurde die ursprüngliche Realisierung des Cursors *CurrentOrder* durch eine einzelne SQL-Anweisung ersetzt, welche die kombinierten Bestellinformationen aus den Tabellen *Orders* und *ItemSuppliers* in der Tabelle *ApprovedOrders* zur Verfügung stellt (*Insert-Tuple-To-Set*-Regel, siehe Teilkapitel 5.5.3). Die sich hierdurch ergebende neue Beschreibung der Stored-Procedure ist im unteren Teil von Abbildung 3.2 dargestellt.
- Die dritte Optimierungsvariante kombiniert das vorausgegangene Szenario. Um das gesamte Potential des Optimierungsansatzes von PGM/F ausnutzen zu können, werden die Restrukturierungsregeln in einem heterogenen, kombinierten Optimierungsszenario auf eine einzelne, integrierte PGM-Repräsentation, welche die Beschreibung des Beispiel-Workflows und der Stored-Procedure *PrepareApprovedOrders* umfasst, angewendet. Im vorliegenden Beispiel erhält man eine solche integrierte PGM-Repräsentation, wenn die PGM-Repräsentation der SQL-Aktivität *PrepareApprovedOrders* durch die PGM-Repräsentation der Stored-Procedure *PrepareApprovedOrders* ersetzt wird. Dieser Integrationsschritt führt zu einer weiteren Optimierungsmöglichkeit: Die nach der heterogenen, isolierten Optimierung verbliebenen Insert-Anweisungen in der Stored-Procedure- bzw. Workflowbeschreibung können aufgrund der Datenabhängigkeiten, die auf der gemeinsam referenzierten Tabelle *ApprovedOrders* basieren, verschmolzen werden (*Eliminate-Temporary-Table*-Regel, siehe Teilkapitel 5.4.7). Als Ergebnis dieses Optimierungsschrittes kann die Beschrei-

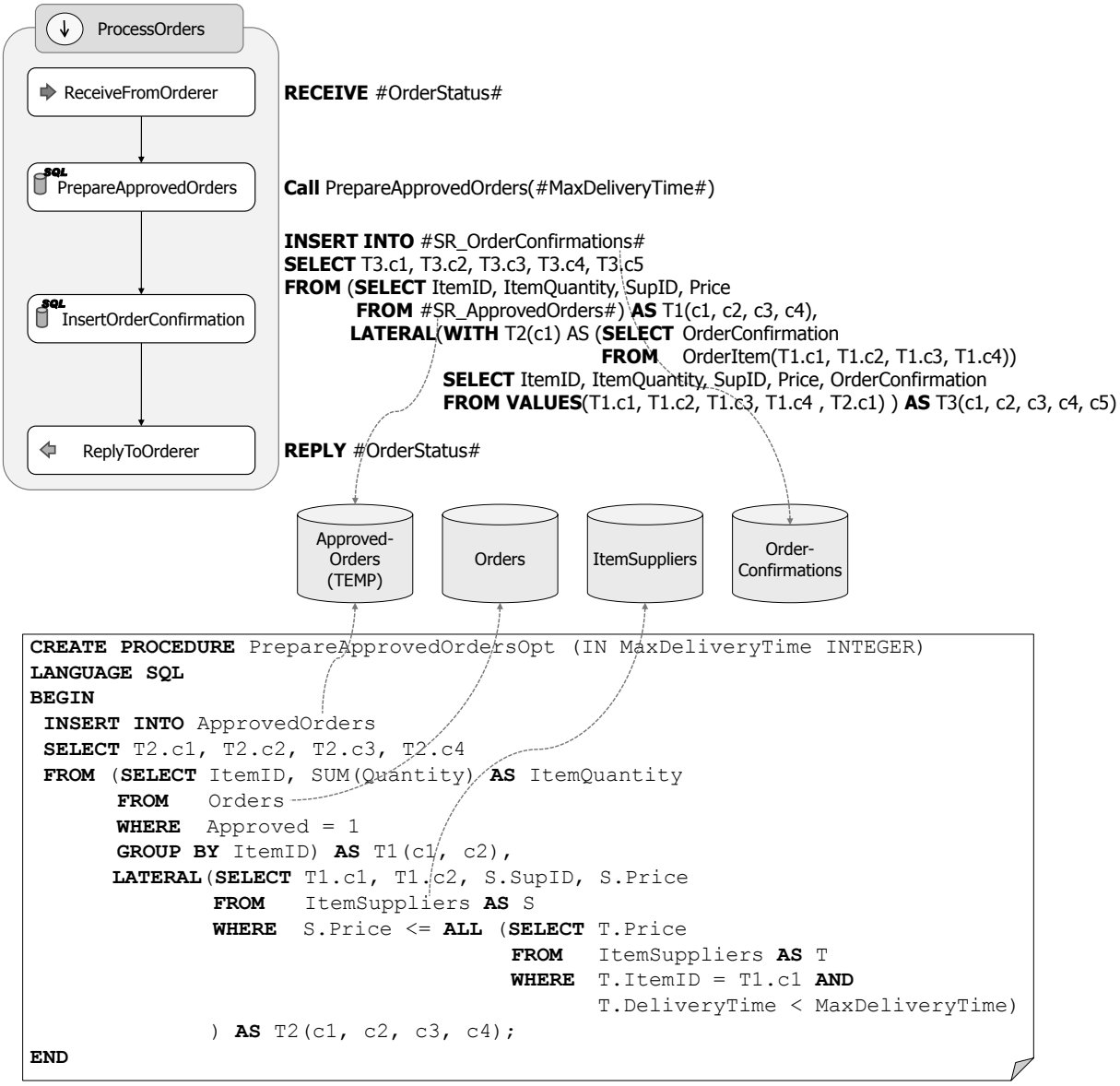


Abbildung 3.2.: Beispiel-Workflow nach homogener/heterogener, isolierter Optimierung

bung der gesamten Stored-Procedure aufgelöst werden. Dies führt zu der in Abbildung 3.3 gezeigten finalen Workflowbeschreibung. Ein solches Optimierungsergebnis wäre nicht möglich, wenn die Workflowbeschreibung von der Stored-Procedure getrennt optimiert worden wäre. Aus Sicht der Datenverarbeitung liefern der optimierte und der in Abbildung 2.2 gezeigte unoptimierte Beispiel-Workflow jeweils dasselbe Ergebnis. In der optimierten Form kann auf den Aufruf der Stored-Procedure bzw. des Web-Services auf der Workflowebene verzichtet werden. Damit kann der Mehraufwand für die Ausführung einzelner Aktivitäten und SQL-Anweisungen reduziert werden. Außerdem ergeben sich für einen Anfrageoptimierer zusätzliche Optionen zur Optimierung der verbliebenen Insert-Anweisung.

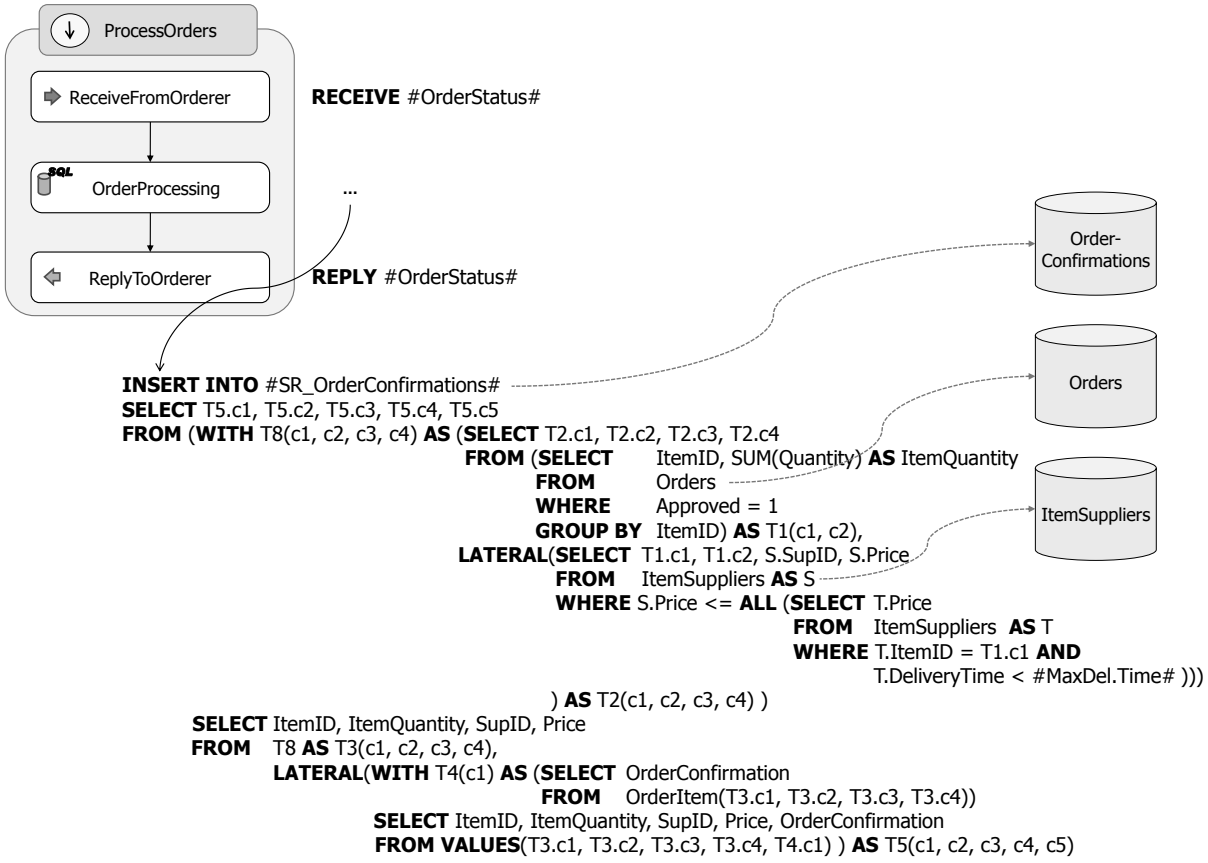


Abbildung 3.3.: Beispiel-Workflow nach heterogener, kombinierter Optimierung

Zusammenfassend kann festgehalten werden, dass sich durch den Gebrauch der Restrukturierungsregeln auf den Beispiel-Workflow folgende Verbesserungen ergeben haben:

- Im Vergleich zu dem ursprünglichen Workflow gibt es mehr Optimierungsmöglichkeiten für ein Datenbanksystem. Die kombinierte Insert-Anweisung weist ein höheres Optimierungspotential als die einzelnen SQL-Anweisungen auf.
- Die Optimierung führt zu einer Reduzierung der Datenbanksystemaufrufe und somit zu einer Reduzierung des Datenvolumens, das zwischen Datenbank- und Workflowebene transferiert werden muss. Nach der Optimierung muss lediglich eine SQL-Anweisung an das zugehörige Datenbanksystem übertragen werden. Insbesondere entfallen die kostenintensive Übertragung und Materialisierung der Ergebnismenge der SQL-Anfrage, die von der <`retrieveSet`> Aktivität *RetrieveApprovedOrders* ausgeführt wird. Des Weiteren reduziert sich die Anzahl der auszuführenden Aktivitäten im Workflowmodell, da nach der Optimierung lediglich eine <`sql`> Aktivität verbleibt. Somit verringert sich auch der Verarbeitungsaufwand für die Ausführung von Aktivitäten bzw. SQL-Anweisungen auf der Workflow- bzw. Datenebene.

Obwohl die Optimierung des Beispiel-Workflows unkompliziert erscheint, müssen verschiedene Bedingungen bezüglich des Kontrollflusses und der Datenabhängigkeiten gelten, damit die Restrukturierungsregeln angewendet werden können. Dies soll an folgenden Modifikationen des Beispiel-Workflows aus Abbildung 2.2 veranschaulicht werden.

Greift beispielsweise die <sql> Aktivität *InsertOrderConfirmation* nicht auf das Ergebnis des Web-Services zu, welcher von der Aktivität *OrderFromSupplier* aufgerufen wird, existiert zwischen beiden Aktivitäten keine Datenabhängigkeit. Folglich kann der Web-Service-Aufruf nicht in die Insert-Anweisung integriert werden. Dadurch sind weitere Optimierungsschritte, wie z.B. die Auflösung der ForEach-Schleife, nicht mehr möglich.

Existiert etwa eine zusätzliche <sql> Aktivität SQL_1 zwischen den Aktivitäten *PrepareApprovedOrders* und *ForEachItemOrder*, die weder auf die Tabelle *Orders* noch auf die Tabelle *OrderConfirmation* zugreift, bestehen keine Datenabhängigkeiten zwischen SQL_1 und den restlichen <sql> Aktivitäten. Somit können Restrukturierungsregeln auf SQL_1 nicht angewendet werden. Der resultierende Workflow würde, wie in Abbildung 3.3 veranschaulicht, die <sql> Aktivitäten *OrderProcessing* aber auch SQL_1 enthalten.

Ist stattdessen die <sql> Aktivität SQL_1 innerhalb der ForEach-Schleife vor der Aktivität *OrderFromSupplier* platziert, kann der Web-Service-Aufruf von der Aktivität *OrderFromSupplier* mit der <sql> Aktivität *InsertOrderConfirmation* weiterhin verschmolzen werden. Eine weitere Verschmelzung mit <sql> Aktivität SQL_1 ist dagegen aufgrund fehlender Datenabhängigkeiten nicht möglich. Folglich ist die Auflösung der ForEach-Schleife wegen der zwei verbliebenen <sql> Aktivitäten im Schleifenrumpf nicht durchführbar.

Um effizient entscheiden zu können, ob eine Restrukturierungsregel angewendet werden kann, müssen Kontrollfluss-, Daten- und Kommunikationsabhängigkeiten zwischen den einzelnen Aktivitäten bzw. SQL-Anweisungen explizit repräsentiert werden. Hierfür wurde das Prozessgraphenmodell, das genau diese Anforderungen erfüllt, entwickelt.

3.3. Eigenschaften

Die Tabelle 3.1 gibt einen Überblick über die charakteristischen Eigenschaften von PGM/F, die im Folgenden näher beschrieben werden.

Unabhängigkeit

Der Optimierungsansatz von PGM/F ist unabhängig von der Syntax von BPEL/SQL. Damit kann der Ansatz auch auf andere ähnliche Beschreibungssprachen übertragen werden. Die Unabhängigkeit von PGM/F wird dabei durch die interne PGM-Repräsentation garantiert, die von den syntaktischen Details einer zu Grunde gelegten Beschreibungssprache abstrahiert. Auf der internen Repräsentation können die Restrukturierungsregeln sprachunabhängig definiert und angewendet werden. Dieser Ansatz setzt voraus, dass entsprechende Transformationskomponenten für eine gegebene Beschreibungssprache zur Verfügung stehen, mit denen deren sprachspezifische Konstrukte auf die entsprechenden abstrakten Elemente der internen Repräsentation abgebildet werden können und umgekehrt. Auf diese Weise kann der PGM/F-Optimierungsansatz unter anderem sowohl für BPEL/SQL als auch für SQL/PSM genutzt werden.

Eigenschaften	Beschreibung
Unabhängigkeit	Bezüglich Workflowbeschreibungssprache
Erweiterbarkeit	Bezüglich PGM, Regelmenge, Kontrollstrategie
Optimierungsart	Heuristisch, regelbasiert
Optimierungsziel	Verschmelzung/Restrukturierung von SQL-Anweisungen Transformation von Datenverarbeitungsmustern
Optimierungsvarianten	Homogen, heterogen isoliert/kombiniert
Optimierungsebene	Workflowmodellierungsebene
Optimierungszeitpunkt	Modellierungs-, Deploymentzeitpunkt
Eigenschaften unterstützter datenintensiver Workflowmodelle	Grad der Automatisierung: A2A Ausführungsdauer: kurzlaufend Modellierungsstil: prozedural Transaktionsmodell: ACID Definition einer <sql> Aktivität: vollständig/partiell Organisation der Datenebene: beliebig

Tabelle 3.1.: Charakteristische Eigenschaften von PGM/F

Erweiterbarkeit

Die Erweiterbarkeit ist eine weitere wichtige charakteristische Eigenschaft von PGM/F. Neue Restrukturierungsregeln können bei Bedarf in die bestehende Regelbasis integriert, und die aktuelle Kontrollstrategie kann entsprechend angepasst werden. Ebenso kann die interne PGM-Repräsentation durch neue Elemente erweitert werden, um die Verwendung neuer Regeln zu ermöglichen. Diese Erweiterungen am PGM/F-System können durchgeführt werden, ohne dass der existierende Optimierungsansatz beeinflusst wird.

Optimierungsart/-ziel

PGM/F ist ein System zur heuristischen, regelbasierten Optimierung datenintensiver Workflows, das bestehende Optimierungsverfahren auf der Prozessmodellierungs- und Datenebene ergänzt. Die Restrukturierungsregeln von PGM/F erzielen Effizienzsteigerungen durch Verschmelzung bzw. Restrukturierung suboptimaler SQL-Anweisungen sowie durch Transformation ineffizienter Datenverarbeitungsmuster unter Berücksichtigung gegebener Kontrollfluss-, Daten- und Kommunikationsabhängigkeiten.

Optimierungsvarianten

Drei verschiedene Optimierungsvarianten werden von PGM/F unterstützt: Bei einer homogenen Optimierung werden nur Datenverarbeitungsoperationen, die integraler Bestandteil eines Workflowmodells sind, berücksichtigt. Bei einer heterogenen Optimierung werden zusätzlich die Beschreibungen der vom Workflow heraus aufgerufenen Stored-Procedures beachtet. Hier können die Restrukturierungsregeln jeweils getrennt voneinander auf die Beschreibungen eines Workflows bzw. den Stored-Procedures (heterogene, isolierte Optimierung) oder aber kombiniert auf diese Beschreibungen angewendet werden (heterogene, kombinierte Optimierung).

Optimierungsebene/-zeitpunkt

Da der Optimierungsansatz von PGM/F auf der Workflowmodellierungsebene angewendet wird, kann er sowohl zum Modellierungs- als auch zum Deploymentzeitpunkt eines Workflowmodells durchgeführt werden. Somit ist die Optimierungsdauer, anders als bei der klassischen Anfrageoptimierung, keine kritische Größe, weil durch die Optimierung die Ausführung eines Workflows nicht beeinträchtigt wird.

Unterstützte Workflowmodelle

PGM/F unterstützt Workflowmodelle, die (i) vollständig automatisch (A2A) und (ii) kurzlaufend ausgeführt werden sowie ausschließlich auf (iii) prozeduralen Kontrollflussstrukturen bzw. (iv) auf einer ACID-Transaktionssemantik beruhen. Prozedural modellierte, kurzlaufende BPEL/SQL-Workflows und SQL-basierte Stored-Procedures sind Beispiele für Beschreibungen, welche diese Eigenschaften erfüllen.

Die Organisation der Datenebene spielt für eine korrekte Nutzung der Restrukturierungsregeln keine Rolle. Die Daten können dabei von zentralisierten bzw. föderierten Datenbanksystemen oder von mehreren autonomen Datenbanksystemen verwaltet werden. Die Definition einer Restrukturierungsregel stellt sicher, dass die Regel nur dann angewendet werden kann, wenn dies von der Organisation der zu Grunde liegenden Datenebene unterstützt wird. Der Bedingungsteil einer Restrukturierungsregel garantiert, dass die SQL-Anweisungen in einem datenintensiven Workflow auch nach der Verwendung einer Restrukturierungsregel weiterhin auf der Datenebene ausgeführt werden können.

Bei der Optimierung eines datenintensiven Workflows muss immer sichergestellt sein, dass zum Optimierungszeitpunkt alle Informationen vorliegen, die für eine korrekte Benutzung einer Restrukturierungsregel notwendig sind. Vor allem bei partiell definierten <sql> Aktivitäten kann dies zum Modellierungszeitpunkt nicht immer garantiert werden. Deshalb müssen in einem solchen Fall die fehlenden Informationen entweder explizit von einem Benutzer zur Verfügung gestellt oder aus dem Deploymentdeskriptor eines Workflowmodells ausgelesen werden. Im letzteren Fall muss die Regelanwendung auf den Deploymentzeitpunkt eines Workflowmodells verschoben werden. Folglich kann eine Regelanwendung nicht immer automatisch durchgeführt werden, sondern erfordert unter Umständen eine Interaktion mit dem Benutzer.

3.4. Zusammenfassung

Das PGM/F-System ist ein heuristischer, regelbasierter SQL-Optimierer für datenintensive Workflows, die auf prozeduralen Kontrollflussstrukturen basieren, automatisch ausgeführt werden und sowohl eine kurze Ausführungsdauer als auch eine ACID-Transaktionssemantik besitzen. Dies entspricht genau den Eigenschaften eines kurzlaufenden BPEL/SQL-Workflows, dessen Optimierung im Mittelpunkt dieser Arbeit steht. Das Optimierungsziel von PGM/F besteht darin, die in einem datenintensiven Workflow definierten suboptimalen Datenverarbeitungsstrukturen zu identifizieren und durch Strukturen zu ersetzen, die auf der Workflow- bzw. Datenebene effizienter ausgeführt werden können. Dieser Optimierungsansatz kann sowohl zum Modellierungs- als auch zum Deploymentzeitpunkt eines Workflowmodells angewendet werden. Dabei wird eine verfügbare Regelbasis mittels einer wohldefinierten Kontrollstrategie auf eine interne PGM-Repräsentation angewendet, die von den syntaktischen Details eines Workflowmodells abstrahiert. Auf diese Weise können die Restrukturierungsregeln sprachunabhängig definiert und angewendet werden. Ein weiterer Vorteil von PGM besteht darin, dass der Optimierungsansatz ebenfalls für Stored-Procedures, die aus einem datenintensiven Workflow heraus aufgerufen werden können, verwendet werden kann. Ebenso wird hierdurch eine kombinierte Optimierung eines Workflowmodells und der darin aufgerufenen Stored-Procedures ermöglicht.

In den folgenden drei Kapiteln werden die PGM-Repräsentation, die Restrukturierungsregeln und die Kontrollstrategie des PGM/F-Systems im Detail beschrieben.

4

Das Prozessgraphenmodell (PGM)

Für die Optimierung datenintensiver Workflows wird in PGM/F eine generische interne Repräsentation, das sogenannte *Prozessgraphenmodell (PGM)*, genutzt. In diesem Kapitel wird erläutert, weshalb eine speziell zugeschnittene Repräsentation notwendig ist, und welches deren zentrale Merkmale sind. Dazu werden in Teilkapitel 4.1 Anforderungen, die eine interne Repräsentation erfüllen muss, beschrieben. Anschließend werden in Teilkapitel 4.2 existierende Repräsentationen für Prozess- und Workflowmodelle sowie für Datenverarbeitungsanweisungen vorgestellt. Dabei wird erörtert, warum diese Repräsentationen für die Optimierung datenintensiver Workflows ungeeignet sind. In Teilkapitel 4.3 werden die Konzepte von PGM im Detail betrachtet. Abschließend werden in Teilkapitel 4.4 die wesentlichen Eigenschaften des Prozessgraphenmodells hervorgehoben, und es wird gezeigt, dass PGM die in Teilkapitel 4.1 beschriebenen Anforderungen an eine interne Repräsentation zur Optimierung datenintensiver Workflows erfüllt.

4.1. Anforderungen

PGM bildet die Basis für die regelbasierte Optimierung datenintensiver Workflows. Dabei handelt es sich um eine generische Repräsentation, die ein Minimum notwendiger Ausdruckskraft besitzt, um prozedural modellierte Workflowmodelle abstrakt und sprachneutral repräsentieren zu können. Dabei werden alle für die Definition und Anwendung der Restrukturierungsregeln notwendigen Informationen explizit zur Verfügung gestellt.

Die Anforderungen an eine interne Repräsentation betreffen zum einen die Optimierung und Repräsentation datenintensiver Workflows [VSS+07, VSRM08] und zum anderen die Anfrageoptimierung im Allgemeinen [PHH92, Cha98, SH05]. Diese wesentlichen Anforderungen sind:

- *Heterogenität bezüglich Aktivitätstypen*: Die interne Repräsentation sollte generisch aufgebaut sein, sodass eine Vielzahl unterschiedlicher Aktivitätstypen unterstützt werden kann. Hierzu gehören sowohl Aktivitätstypen aus Workflowbeschreibungssprachen, wie BPEL, zur prozeduralen Modellierung kurzlaufender Workflows, als auch Aktivitätstypen zur Ausführung von SQL-Anweisungen in datenintensiven Workflows. Weiter sollte die interne Repräsentation flexibel sein, sodass auch Stored-Procedures dargestellt werden können. Diese Flexibilität ermöglicht eine Optimierung der Datenverarbeitung über Aktivitäts- und Workflowgrenzen hinweg.
- *Kontrollflussmuster*: In Workflowbeschreibungen werden unterschiedliche Kontrollflussmuster unterstützt [AHKB03, RHAM06]. Der SQL-Standard definiert ebenfalls Kontrollflussmuster zur Implementierung von Stored-Procedures. Alle relevanten Kontrollflussmuster sollen mit der internen Repräsentation abgebildet werden können. Dabei soll von sprachspezifischen Eigenschaften abstrahiert werden, um eine sprachunabhängige Modellierung der Kontrollflussmuster zu ermöglichen.
- *Explizite Abhängigkeiten*: Zur korrekten Verwendung der Restrukturierungsregeln müssen Daten- und Kontrollflussabhängigkeiten sowie Kommunikationsbeziehungen in einem datenintensiven Workflow analysiert werden. Typischerweise werden die Abhängigkeiten in einem Workflowmodell nicht explizit dargestellt. Dies erschwert eine Definition und Anwendung der Restrukturierungsregeln direkt auf einem solchen Workflowmodell erheblich. Deshalb muss eine interne Repräsentation alle für den hier vorgestellten heuristischen, regelbasierten Optimierungsansatz relevanten Abhängigkeiten explizit darstellen. Dies erlaubt eine einfache Definition und Anwendung der Restrukturierungsregeln.
- *Kompakte Repräsentation*: Die interne Repräsentation eines Workflowmodells sollte möglichst kompakt erfolgen, damit die Restrukturierungsregeln exakt und gleichzeitig einfach definiert und angewendet werden können. Dabei sollten alle für die Restrukturierungsregeln unbedeutenden Details ausgeblendet werden. Beispielsweise können Detailinformationen zu Aktivitätstypen, für die keine Restrukturierungsregeln verfügbar sind, entfallen. Hierdurch kann eine wesentliche Vereinfachung der Optimierung erreicht werden.
- *Erweiterbarkeit*: Die interne Repräsentation sollte generisch sein, damit sie auf einfache Weise erweitert werden kann. Damit können bei Bedarf zusätzliche Aktivitätstypen bei der Optimierung berücksichtigt werden.

4.2. Abgrenzung zu existierenden Repräsentationen

Da bereits eine Vielzahl von Repräsentationen für Workflows und Datenverarbeitungsanweisungen existiert, soll im Folgenden erörtert werden, warum diese für die Optimierung datenintensiver Workflows nicht geeignet sind.

4.2.1. Repräsentationen für Datenverarbeitungsanweisungen

In Datenbanksystemen werden insbesondere graphenbasierte Repräsentationen zur Darstellung von Datenverarbeitungsanweisungen verwendet (siehe z.B. [RH86, PHH92, CKS+00, BG05, HH07]). Das Query-Graph-Model (QGM) [PHH92] ist ein prominenter Vertreter hiervon. Es erlaubt eine strukturelle, prozedurale Darstellung einer deklarativen SQL-Anfrage und bildet die Grundlage für deren Optimierung und Ausführung. Da sich eine Repräsentation auf die strukturelle Darstellung von Anfragen beschränkt, können insbesondere die Ausführungslogik eines Workflows und die darin enthaltenen Daten-, Kontrollfluss- und Kommunikationsbeziehungen zwischen den einzelnen Aktivitäten nicht repräsentiert werden. Folglich ist eine solche Repräsentation für die Darstellung datenintensiver Workflows nicht geeignet.

Ebenso ist eine feingranulare Darstellung für die Repräsentation einer SQL-Anweisung auf der PGM-Ebene nicht erforderlich, weil es sich bei der PGM-Optimierung um syntaktische Transformationen bzw. Substitutionen von SQL-Anweisungen handelt.

Auch die in Teilkapitel 2.2.3 vorgestellten Ansätze zur Optimierung von Anweisungssequenzen lassen sich nicht in den Bereich der PGM-Optimierung übertragen. Einerseits gibt es erhebliche Einschränkungen hinsichtlich der unterstützten Kontrollflusskonstrukte, da lediglich Produzent-Konsument-Beziehungen zwischen den Anweisungen darzustellen sind. Für die PGM-Optimierung sind darüber hinaus neben sequentiellen auch iterative, alternative und parallele Kontrollflussstrukturen sowie Operationen, wie Web-Service-Aufrufe oder Variablenzuweisungen, von Bedeutung. Andererseits umfasst die PGM-Optimierung einen wesentlich größeren Optimierungsbereich. Begründet ist dies darin, dass neben einzelnen SQL-Anweisungen auch deren Zusammenspiel in einem Workflow berücksichtigt werden.

Ebenso lassen sich Repräsentationen für ETL-Workflows wegen der unterschiedlichen Sprachebenen nicht direkt auf eine PGM-Optimierung übertragen. Dies liegt vor allem an den diversen Aktivitätstypen, die zur Modellierung der jeweiligen Workflowtypen zur Verfügung stehen. Hieraus ergeben sich verschiedene Optimierungsebenen: Während bei ETL-Workflows eine logische, feingranulare Optimierung algebraischer Operatoren im Vordergrund steht, liegt der Schwerpunkt der PGM-Optimierung auf einer logischen, grobkörnigeren Optimierung von SQL-Anweisungen.

4.2.2. Repräsentationen für Prozess- und Workflowmodelle

Bei der Repräsentation und Optimierung von Geschäftsprozessen kann zwischen mehreren Abstraktionsebenen unterschieden werden. Dementsprechend existieren sowohl grafische als auch technische bzw. theoretische Repräsentationen für Geschäftsprozesse.

Grafische Repräsentationen, wie BPMN, EPK oder UML (siehe Teilkapitel 1.1), werden insbesondere zur Erstellung einfacher, abstrakter Prozessmodelle verwendet, um alle wichtigen Aspekte eines Geschäftsprozesses in einer allgemein verständlichen Form zu beschreiben bzw. zu visualisieren. Technische Details, die zur Ausführung eines Geschäftsprozesses auf einer konkreten IT-Infrastruktur notwendig sind, werden vollständig ausgeblendet.

Insbesondere fehlen auch Detailinformationen, die für die Optimierung datenintensiver Workflows relevant sind. Deshalb sind grafische Repräsentationen für den hier betrachteten Optimierungsansatz ungeeignet.

Auf der technischen Ebene erfolgt die Überführung eines abstrakten Prozessmodells in ein Workflowmodell. Für die Beschreibung von Workflowmodellen hat sich heute BPEL als Standard durchgesetzt. Die Beschreibung eines Workflowmodells umfasst sämtliche Detailinformationen, die für die rechnergestützte Ausführung des Workflows notwendig sind. Allerdings sind nicht alle dieser Details für die Optimierung relevant. Umgekehrt fehlen für die Optimierung erforderliche Detailinformationen, wie beispielsweise Daten- und Kommunikationsabhängigkeiten, die typischerweise nur teilweise oder nicht explizit repräsentiert werden. Dies würde im Rahmen der PGM-Optimierung dazu führen, dass erst zum Optimierungszeitpunkt die entsprechenden Informationen durch umfassende Analysen aus dem Workflowmodell gewonnen werden müssten. Wegen des damit verbundenen erhöhten Optimierungsaufwandes ist eine technische Repräsentation für Workflowmodelle für die PGM-Optimierung nicht geeignet.

Ferner gibt es theoretische Modelle zur Beschreibung von Prozessen, wie beispielsweise die Petri-Netze [Rei92] oder das Pi-Kalkül [Mil93]. Mit diesen Modellen kann das Verhalten von Prozessen formal definiert werden. Somit bilden sie die Grundlage für die formale Analyse und Verifizierung von Prozessmodellen [OS96, Aal97, Aal99, Aal03, AV04]. Ein Argument gegen ihre Verwendung im Rahmen der PGM-Optimierung ist deren Komplexität, infolge derer der für den hier betrachteten Optimierungsansatz notwendige, höhere Abstraktionsgrad nicht erreicht werden kann.

Jede der hier diskutierten Repräsentationen weist erhebliche Schwächen für die Verwendung im Rahmen der PGM-Optimierung auf. Aus diesem Grund wurde das Prozessgraphenmodell als interne Repräsentation entwickelt. Dabei wurde es auf die Anforderungen der PGM-Optimierung zugeschnitten. Nichtsdestotrotz besitzt es Ähnlichkeiten zu den hier diskutierten Repräsentationen. Dabei orientieren sich die verfügbaren Modellierungskonstrukte an den Anforderungen zur Repräsentation von prozeduralen, kurzlaufenden Workflowmodellen. Deshalb gibt es auch Gemeinsamkeiten mit Modellen, die auf technischer Ebene Geschäftsprozesse beschreiben. Allerdings handelt es sich bei dem Prozessgraphenmodell, im Gegensatz zu den technischen Repräsentationen, um keine vollständige und damit ausführbare Repräsentation eines Workflowmodells. Explizit repräsentiert werden lediglich die für die Optimierung notwendigen Details eines Workflowmodells. Darüber hinaus weist das Prozessgraphenmodell Gemeinsamkeiten mit grafischen Repräsentationen für abstrakte Prozessmodelle auf. Ergänzend zur formalen Definition existiert eine anschauliche grafische Repräsentation für die mit dem Prozessgraphenmodell repräsentierten Workflowmodelle.

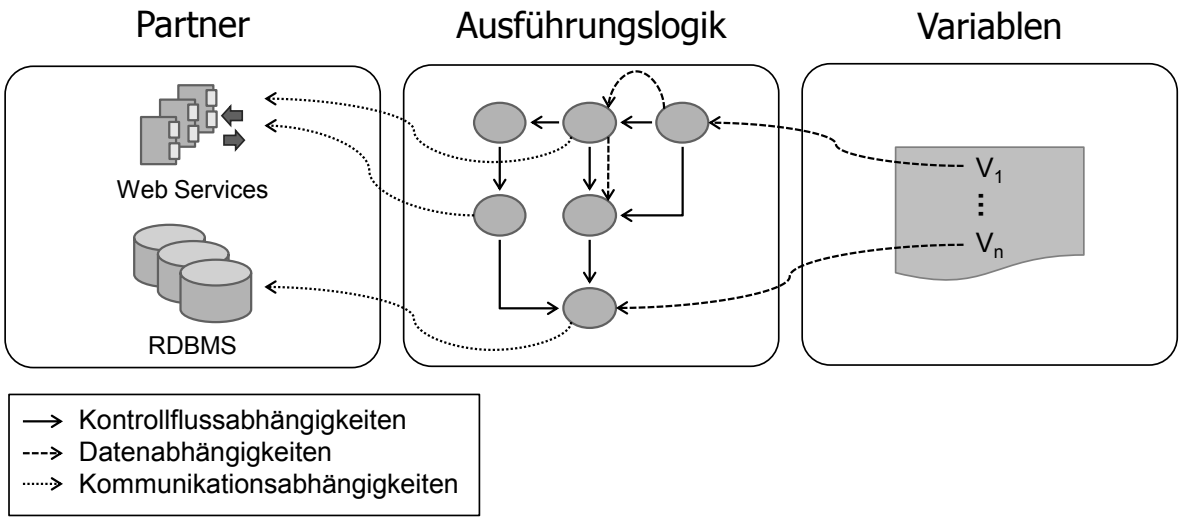


Abbildung 4.1.: Grundlegendes Konzept von PGM

4.3. Einführung in PGM

PGM ist eine generische Repräsentation für prozedural modellierte, kurzlaufende Workflowmodelle, das die in Teilkapitel 4.1 beschriebenen Anforderungen erfüllt. Das Ziel von PGM besteht nicht darin, alle syntaktischen Details eines Workflowmodells darzustellen. Vielmehr repräsentiert PGM eine Art Sicht, die von einem Workflowmodell abgeleitet wird, und die nur jene Teile eines Workflowmodells explizit sichtbar macht, die für eine Optimierung datenintensiver Workflows notwendig sind. Dagegen werden alle Teile eines Workflowmodells, die nicht vom Optimierungsansatz betroffen sind, entweder in PGM nicht berücksichtigt oder aber durch generische Konstrukte gekapselt.

Abbildung 4.1 zeigt das grundlegende Konzept des Prozessgraphenmodells, das auf einer Dreiteilung in Partner, Variablen und Ausführungslogik beruht. Partner sind externe Systeme, wie beispielsweise relationale Datenbanksysteme, mit denen ein Workflow während seiner Ausführung interagiert. Variablen speichern Daten, die innerhalb eines Workflows während dessen Ausführung verarbeitet werden. Die Ausführungslogik umfasst sowohl einzelne Aktivitäten als auch den Kontrollfluss zwischen den Aktivitäten. In PGM werden Kommunikationsbeziehungen zwischen Aktivitäten und Partnern sowie Datenabhängigkeiten zwischen den Aktivitäten explizit modelliert. Dabei beschreiben Datenabhängigkeiten den Datenfluss zwischen Aktivitäten basierend auf den Variablen, die von den Aktivitäten verarbeitet werden.

In den folgenden Teilkapiteln werden die Bestandteile von PGM im Detail beschrieben. Dabei wird auf eine formale Definition von PGM zugunsten der Übersichtlichkeit verzichtet. Stattdessen werden alle wesentlichen Aspekte anhand der grafischen Darstellung von PGM erläutert. Die formale Definition von PGM findet man in Anhang A dieser

Arbeit, auf die an der entsprechenden Stelle verwiesen wird. Ebenso ist in Anhang B eine XML-basierte Syntax für PGM zu finden, welche auf der formalen Definition beruht.

4.3.1. PGM-Graph

Eine PGM-Repräsentation ist ein gerichteter Graph, der aus Knoten und Kanten besteht (siehe Definition 1 in Anhang A). Knoten repräsentieren Aktivitäten; Kanten stellen Daten-, Kontrollfluss- und Kommunikationsabhängigkeiten dar. Außerdem werden in einem solchen Graphen Variablen und Partner, welche die Basis für die Berechnung von Daten- und Kommunikationsabhängigkeiten bilden, explizit repräsentiert.

4.3.2. Partner

Partner (siehe Definition 3) sind externe Systeme, die mit Aktivitäten eines Workflows interagieren. In PGM werden folgende Partnertypen explizit unterstützt:

- *Web-Services* (Partner vom Typ `WS`) sind die Bausteine einer Web-Service-Orchestrierung, die in einer Workflowbeschreibungssprache, wie BPEL/SQL, zur Realisierung von Geschäftsfunktionen verwendet werden.
- Ein *relationales Datenbanksystem* (Partner vom Typ `RDBMS`) ist eine Datenquelle, die relationale Daten verwaltet und SQL als deklarative Anfragesprache unterstützt. Solche relationalen Datenbanksysteme sind zentraler Bestandteil datenintensiver Workflows und liegen somit im Fokus der PGM-Optimierung. Im Vergleich zu einem Partner vom Typ `WS` kann ein Partner vom Typ `RDBMS` zusätzliche Eigenschaften, die für einige Restrukturierungsregeln von Bedeutung sind, besitzen (siehe Definition 4): Unterstützt beispielsweise dieser Partnertyp den Aufruf von Web-Services durch spezifische User-Defined-Functions bzw. die Ausführung von Stored-Procedures, zeigt dies die Eigenschaft `WS-UDTF` bzw. `SP` an.
- Alle verbleibenden möglichen Partnertypen eines Workflowmodells, die nicht auf diese beiden Partnertypen abgebildet werden können, werden als *generischer* Partnertyp (Partner vom Typ `GENERIC`) dargestellt. Der generische Partnertyp erlaubt zum einen, Partnertypen, die für die PGM-Optimierung nicht relevant sind, auszublenden, und zum anderen, neue Partnertypen abzuleiten, wenn dies für die Definition neuer Restrukturierungsregeln notwendig ist.

4.3.3. Variablen

Variablen (siehe Definition 6) speichern Daten, die von Aktivitäten während einer Workflowausführung bearbeitet werden. Bei diesen Daten handelt es sich beispielsweise um Tabellen, die in SQL-Anweisungen referenziert werden, oder um Eingabe- und Ausgabe-Parameter von Stored-Procedure- bzw. Web-Service-Aufrufen.

Für die Definition der Restrukturierungsregeln genügt die Kenntnis, ob einer Variablen eine mengenorientierte Daten- oder Basisdatenstruktur zu Grunde gelegt ist. Aus diesem Grund ist für PGM ein einfaches, abstraktes Datentypsystem bestehend aus drei Typen ausreichend (siehe Definition 5):

- Darin enthalten ist ein `SET`-Datentyp, der relationale Tabellen repräsentiert, die sowohl von Partnern vom Typ `RDBMS` verwaltet werden als auch materialisiert in einem Workflow vorliegen können.
- Ein `SIMPLE`-Datentyp deckt alle wohlbekannten Basisdatentypen, wie z.B. String, Integer usw. ab, die auch von relationalen Datenbanksystemen unterstützt werden.
- Schließlich werden alle verbleibenden Datentypen einer Workflowbeschreibungssprache, die nicht auf einen `SET`- bzw. `SIMPLE`-Datentyp abgebildet werden können, durch einen allgemeinen, abstrakten `GENERIC`-Datentypen dargestellt. Der generische Datentyp blendet in einer PGM-Repräsentation alle Datentypen eines Workflowmodells, welche für die Definition der Restrukturierungsregeln nicht relevant sind, aus. Außerdem erlaubt er die Ableitung neuer Datentypen, wenn dies für neue Restrukturierungsregeln erforderlich ist.

Im Folgenden wird für Variablen vom Typ `SET`, `SIMPLE` und `GENERIC` die abkürzende Schreibweise `SET`-, `SIMPLE`- und `GENERIC`-Variable verwendet.

4.3.4. Aktivitäten

Aktivitäten führen in PGM abstrakte Operationen aus, die von spezifischen syntaktischen Merkmalen eines Workflowmodells abstrahieren. Zusätzlich enthält eine Aktivität Komponenten zur Modellierung von Daten- und Kontrollflussabhängigkeiten zu anderen Aktivitäten sowie zur Modellierung von Kommunikationsabhängigkeiten zu Partnern.

Eine *Generic*-Aktivität (siehe Definition 12) erlaubt es, die Menge vordefinierter Aktivitätstypen zu erweitern, um PGM an die Bedürfnisse neuer Restrukturierungsregeln anpassen zu können. Außerdem wird dieser generische Aktivitätstyp verwendet, um Aktivitäten eines Workflowmodells, die für Restrukturierungsregeln nicht relevant sind, zu repräsentieren. Auf diese Weise kann PGM ein breites Spektrum von Workflowmodellen unabhängig von deren zu Grunde gelegten Syntax repräsentieren.

Da alle Aktivitätstypen in PGM von der *Generic*-Aktivität abgeleitet werden, besitzen alle Aktivitätstypen in PGM eine gemeinsame Basisstruktur, die durch den Aufbau der *Generic*-Aktivität vorgegeben ist. Abbildung 4.2 zeigt diesen in PGM allgemeingültigen Aufbau, der Komponenten zur expliziten Repräsentation von Daten-, Kontrollfluss- und Kommunikationsabhängigkeiten beinhaltet.

Kontrollflussabhängigkeiten werden durch Kontrollflusskanten modelliert, die sogenannte *Input-* (ICS) und *Output-Controlflowslots* (OCS) miteinander verbinden (siehe Definitionen 8 und 9). Man unterscheidet jeweils zwischen einer einfachen und einer komplexen Variante dieser Slots. Ein einfacher Slot (vom Typ `SIMPLE`) beschreibt die Übernahme eines eingehenden Kontrollflusses von jeweils genau einer Aktivität bzw. die Weitergabe

eines ausgehenden Kontrollflusses an genau eine Aktivität. Die komplexeren Varianten der Slots für den eingehenden Kontrollfluss (vom Typ `AND` und `XOR`) dienen zur Zusammenführung mehrerer alternativer bzw. parallel ausgeführter Kontrollflusspfade. Entsprechend kann durch die komplexe Variante des Slots für den ausgehenden Kontrollfluss (vom Typ `SPLIT`) die Verarbeitung parallel fortgesetzt werden.

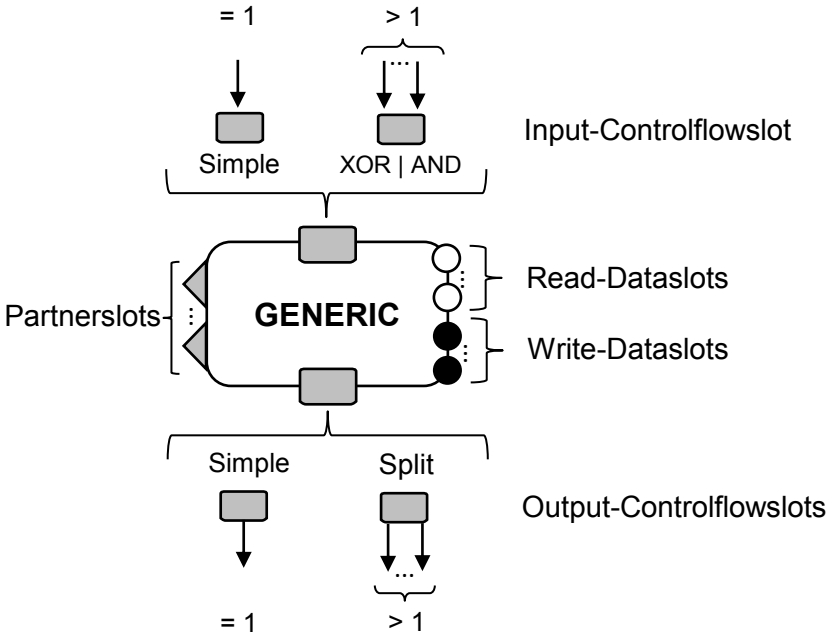


Abbildung 4.2.: Aufbau einer *Generic*-Aktivität

Mit diesen unterschiedlichen Slot-Typen lassen sich Aktivitäten in PGM definieren, mit denen Basiskontrollflussmuster, wie Sequenzen, Schleifen sowie alternative und parallele Kontrollflüsse, repräsentiert werden können. In Teilkapitel 4.3.4 werden die vordefinierten Aktivitätstypen zur Modellierung solcher Kontrollflussmuster näher beschrieben.

Ein ICS bzw. OCS vom Typ `SIMPLE` ist immer der Start- bzw. Endpunkt genau einer Kontrollflusskante. Komplexe ICS- bzw. OCS-Typen repräsentieren dagegen einen Punkt in der Ausführungslogik, bei dem Kontrollflusspfade aufgespalten bzw. vereinigt werden. Deshalb stellen sie immer einen Start- bzw. Endpunkt von mindestens zwei ein- bzw. ausgehenden Kontrollflusskanten dar.

Neben einem ICS und einem oder mehreren OCSs enthält eine Aktivität eine endliche Menge von *Partnerslots*. Ein Partnerslot ermöglicht die Modellierung von Kommunikationsabhängigkeiten. Dazu wird ein Partnerslot einer Aktivität mit dem Partner, der während der Workflowausführung mit dieser Aktivität interagiert, verbunden. Ein solcher Partnerslot wird auf der linken Seite einer Aktivität als Pfeilspitze dargestellt.

Auf der rechten Seite einer generischen Aktivität befinden sich *Dataslots*, die eine explizite Beschreibung von Datenabhängigkeiten ermöglichen (siehe Definition 11). Sie repräsentieren Variablen, die von einer Aktivität während ihrer Ausführung lesend bzw. schreibend referenziert werden. Entsprechend unterscheidet man zwischen Slots für lesende und schreibende Variablenzugriffe, sogenannte `Read`- und `Write`-Dataslots. Mit

diesen Informationen lässt sich die Lese- bzw. Schreibmenge einer Aktivität unmittelbar bestimmen. Außerdem können damit Datenabhängigkeiten zwischen Aktivitäten auf Grundlage gemeinsamer Lese- und Schreibmengen berechnet werden. Existiert eine Datenabhängigkeit zwischen zwei Aktivitäten, werden die entsprechenden Dataslots mit einer gerichteten Datenflusskante verbunden. Darüber hinaus kann die Art einer Datenabhängigkeit direkt über die beiden Typen der durch die Datenflusskante verbundenen Dataslots bestimmt werden. Dies vereinfacht die Überprüfung von Datenabhängigkeiten bei der Anwendung von Restrukturierungsregeln.

Hierarchie der Aktivitätstypen

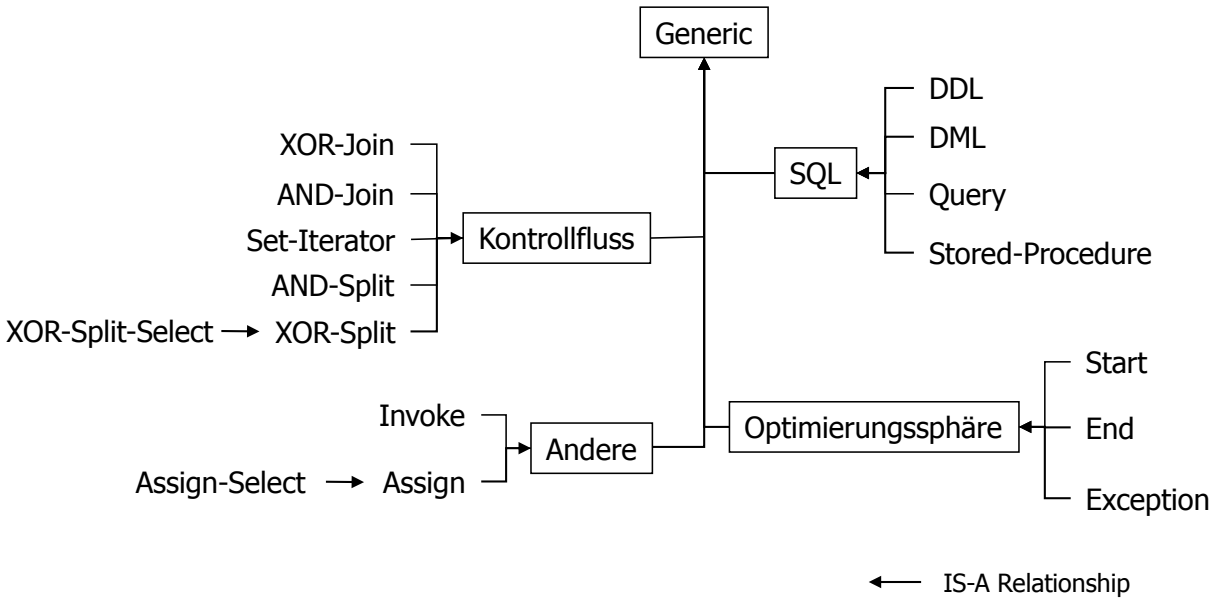


Abbildung 4.3.: Hierarchie der Aktivitätstypen in PGM

PGM unterstützt eine heterogene Menge von Aktivitätstypen, um eine Optimierung über verschiedene Aktivitätstypen hinweg zu ermöglichen. Dabei handelt es sich um für die Restrukturierungsregeln relevante Aktivitätstypen. Diese Menge von Aktivitätstypen reicht von SQL-Aktivitäten, welche die Ausführung von SQL-Anweisungen erlauben, bis hin zu Aktivitätstypen, die Kontrollflussmuster und den Aufruf von Web-Services bzw. Variablenzuweisungen repräsentieren.

Abbildung 4.3 zeigt die Hierarchie vordefinierter Aktivitäten in PGM, an dessen Spitze der generische Aktivitätstyp steht, von dem alle restlichen Aktivitätstypen abgeleitet werden. Somit stellen alle Aktivitätstypen Spezialisierungen des generischen Aktivitätstypen dar. Unterschiede können in der Art und Anzahl verfügbarer Slots bestehen sowie in typspezifischen Zusatzinformationen. Beispielsweise wird bei einer SQL-Aktivität die assoziierte SQL-Anweisung gespeichert. Diese Information ist notwendig, um die Definition und Anwendung der Restrukturierungsregeln zu vereinfachen. Abbildung 4.4 gibt einen Überblick über den Aufbau der in PGM vordefinierten Aktivitätstypen, die im Folgenden näher beschrieben werden. In Anhang A.4 findet man die zugehörige formale Definition dieser Aktivitätstypen.

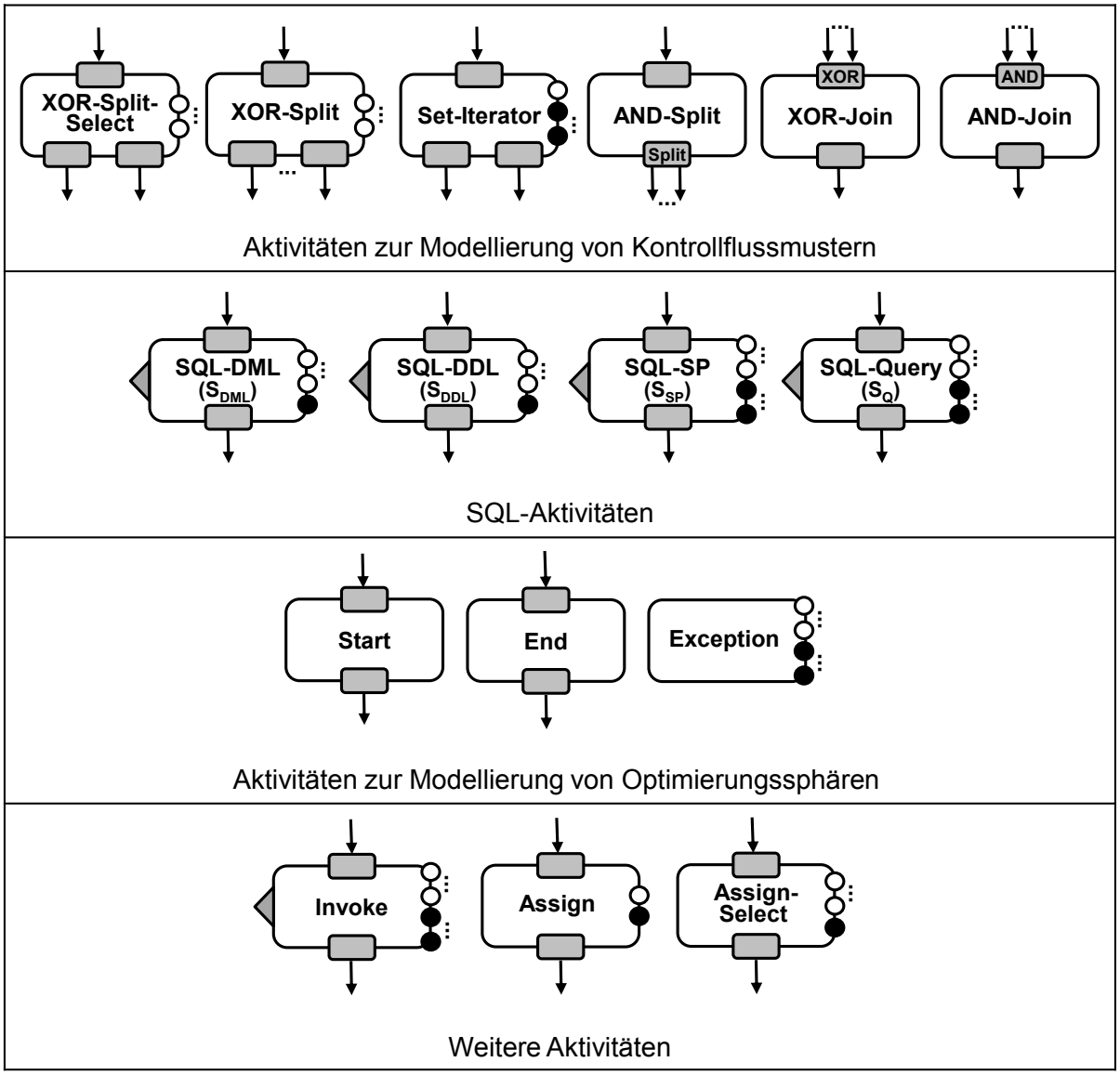


Abbildung 4.4.: Aufbau der vordefinierten Aktivitätstypen in PGM

Aktivitäten zur Modellierung von Kontrollflussmustern

Die Aktivitäten zur Modellierung von Kontrollflussmustern besitzen verschiedene Slots für den ein- und ausgehenden Kontrollfluss. Mit Partnern sind sie nicht verbunden. Ihre Aufgabe besteht darin, entweder einen ausgehenden Kontrollfluss an eine oder mehrere Folgeaktivitäten weiterzuleiten (Split-Typ), oder einen über mehrere unterschiedliche Pfade eingehenden Kontrollfluss wieder zu vereinigen (Join-Typ). Es ist wichtig, dass alle Kontrollflussabhängigkeiten zwischen den Aktivitäten explizit modelliert werden, um die Definition der Restrukturierungsregeln zu vereinfachen.

Eine *XOR-Split*-Aktivität (siehe Definition 14) spaltet einen Kontrollfluss in mehrere alternativ ausgeführte Kontrollflusspfade auf. Dazu besitzt der Aktivitätstyp einen einfa-

chen ICS und mindestens zwei einfache OCSs. Der ICS ist Endpunkt einer eingehenden, und jeder OCS ist Startpunkt einer ausgehenden Kontrollflusskante. Typischerweise ist die Auswahl eines dieser Kontrollflusspfade mit Prädikaten, die zur Laufzeit evaluiert werden, verknüpft. Da die Prädikate Eingabeparameter enthalten können, besitzt eine *XOR-Split*-Aktivität `Read`-Dataslots, die mit Variablen assoziiert sind. Deren Werte werden eingelesen, um die Prädikate zu evaluieren. Da die Prädikate für die PGM-Optimierung nicht relevant sind, werden sie in PGM nicht explizit dargestellt.

Im Gegensatz dazu spielt das Prädikat einer *XOR-Split-Select*-Aktivität (siehe Definition 15) bei der PGM-Optimierung eine wichtige Rolle. Deshalb stellt es einen integralen Bestandteil der Definition dieses Aktivitätstypen dar. Eine *XOR-Split-Select*-Aktivität besitzt genau zwei OCSs, die Startpunkt zweier alternativer Kontrollflusspfade sind. Welche der beiden Alternativen ausgewählt wird, hängt von einem Selektionsprädikat σ ab, das folgende Struktur besitzt (siehe Definition 13): $\sigma = A_1$ `AND/OR` ... `AND/OR` A_m, wobei A_i die Form c_i *op* $k_i \mid v_i$ besitzt. Dabei entspricht c_i in einem Teilausdruck A_i einem Attribut einer Tabelle, $op \in \{=, \neq, <, >, \leq, \geq\}$ stellt einen Vergleichsoperator dar, und k_i entspricht einer Konstanten (Zahl oder Text) bzw. v_i entspricht einer Variablen, deren Werte zur Evaluierung eines Teilausdrucks herangezogen werden. Einem Selektionsprädikat σ kann beispielsweise ein XPath-Ausdruck zu Grunde gelegt sein, der in einem BPEL/SQL-Workflow in einer <`if`> Aktivität in einem <`condition`> Element auf einer materialisierten Tabelle (XML-RowSet-Datenstruktur) definiert sein kann. Der XPath-Ausdruck entspricht dabei der Form des Selektionsprädikates σ, mit dem Unterschied, dass das Attribut c_i über einen Pfadausdruck der Form $RowSet/Row/c_i$ definiert ist. Ein solcher XPath-Ausdruck lässt sich dann unmittelbar in die entsprechende PGM-Repräsentation, in dem lediglich ein Pfadausdruck auf das zugehörige Attribut der zu Grunde gelegten materialisierten Tabelle abgebildet wird, überführen.

Eine *XOR-Join*-Aktivität (siehe Definition 16) vereinigt alternativ ausgeführte Kontrollflusspfade. Dazu besitzt dieser Aktivitätstyp einen ICS vom Typ `XOR`, der die verschiedenen Kontrollflusspfade wieder auf einen einzelnen Kontrollflusspfad unter Verwendung eines einfachen OCS umlegt.

Zur Modellierung paralleler Kontrollflusspfade stellt PGM *AND-Split-* und *AND-Join*-Aktivitäten zur Verfügung (siehe Definitionen 17 und 18). Eine *AND-Split*-Aktivität legt einen einzelnen Kontrollflusspfad auf mindestens zwei parallel ausgeführte Kontrollflusspfade um. Hierzu besitzt eine *AND-Split*-Aktivität einen einfachen ICS und einen OCS vom Typ `Split`. Zur Synchronisation parallel ausgeführter Kontrollflusspfade wird die *AND-Join*-Aktivität verwendet, die mit ihrem ICS vom Typ `AND` die parallel ausgeführten Kontrollflusspfade zum einen synchronisiert und zum anderen wieder auf einen einzelnen Kontrollflusspfad mithilfe eines einfachen OCS umleitet.

Schleifen werden oft eingesetzt, um über Elemente einer Datenmenge zu iterieren. Eine Schleife wird in PGM mittels einer *Set-Iterator*-Aktivität modelliert (siehe Definition 19). In jedem Durchlauf liefert die *Set-Iterator*-Aktivität das nächste Tupel einer `Set`-Variablen. Außerdem definiert dieser Aktivitätstyp zwei alternative Pfade: Einen, der ausgeführt wird, solange weitere Tupel in der Menge verfügbar sind, und einen zweiten, der gewählt wird, sobald alle Tupel der Datenmenge geliefert worden sind. Zur Repräsentation

dieses Kontrollflussverhaltens besitzt der Aktivitätstyp einen einfachen ICS und zwei einfache OCSs. Der einzige `Read`-Dataslot der Aktivität ist mit einer `SET`-Variablen assoziiert, welche die Datenmenge, über die iteriert wird, repräsentiert. Die Attributwerte eines Tupels der Datenmenge werden in Variablen, die mit `Write`-Dataslots verbunden sind, geschrieben. Für jeden Attributwert eines Tupels existiert somit eine Variable, deren Name und Datentyp dem zugehörigen Namen bzw. Datentypen des Attributs eines Tupels entspricht. Hieraus folgt, dass eine *Set-Iterator*-Aktivität genauso viele `Write`-Dataslots besitzt wie Attributwerte in einem Tupel vorhanden sind. Jedes Mal, wenn die Aktivität ausgeführt wird, bestimmt sie das nächste Tupel der Datenmenge und weist die einzelnen Attributwerte des Tupels den entsprechenden Variablen zu, die an die `Write`-Dataslots gebunden sind. Nach dem Schreiben der Tupelwerte in die Variablen wird der Kontrollfluss an die folgende Aktivität, die typischerweise die gelieferten Tupelwerte verarbeitet, übergeben. Kommt der Kontrollfluss wieder zur *Set-Iterator*-Aktivität zurück, bestimmt sie das nächste Tupel der Datenmenge. Dieser Vorgang wiederholt sich, solange bis keine Tupel mehr in der Datenmenge verfügbar sind. In diesem Fall wird der Kontrollfluss an den alternativen Kontrollflusspfad übergeben und damit die Schleife wieder verlassen.

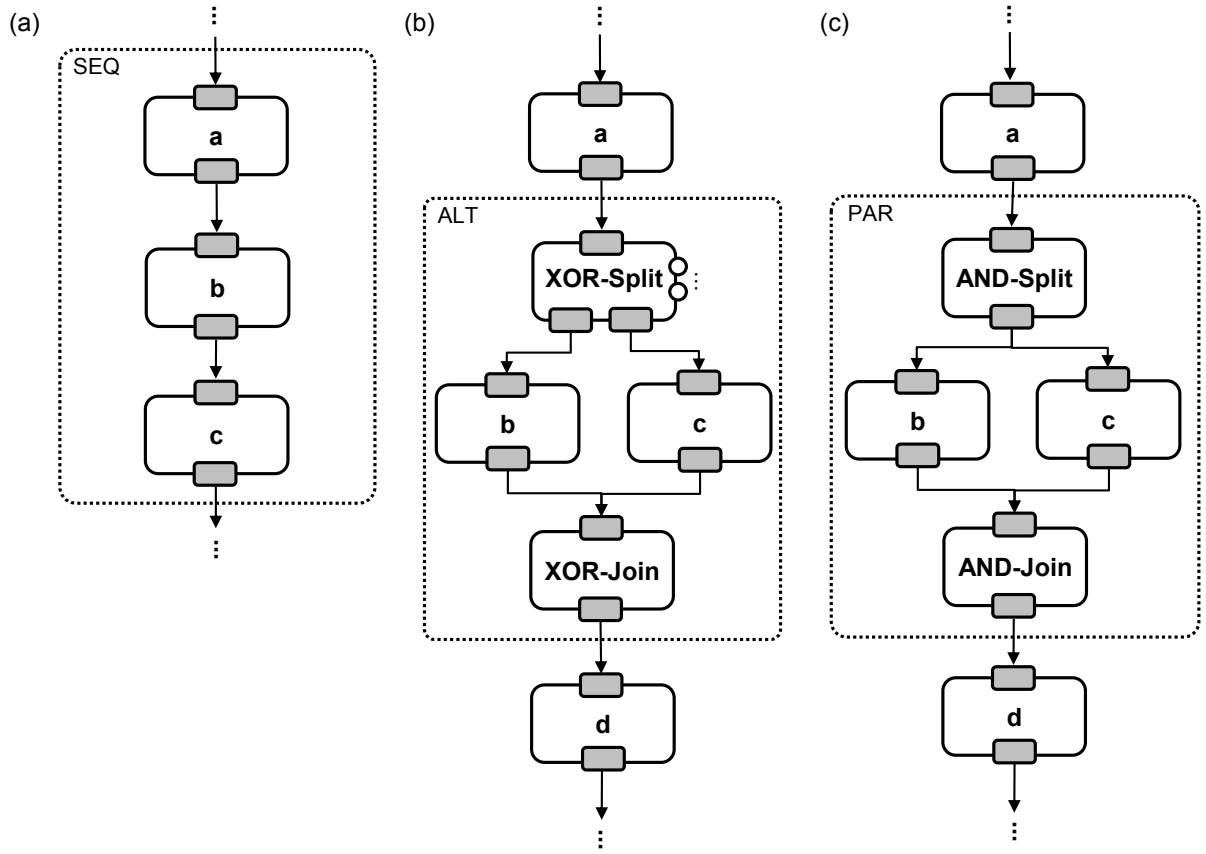


Abbildung 4.5.: Modellierung (a) sequentieller, (b) alternativer und (c) paralleler Kontrollflussmuster in PGM

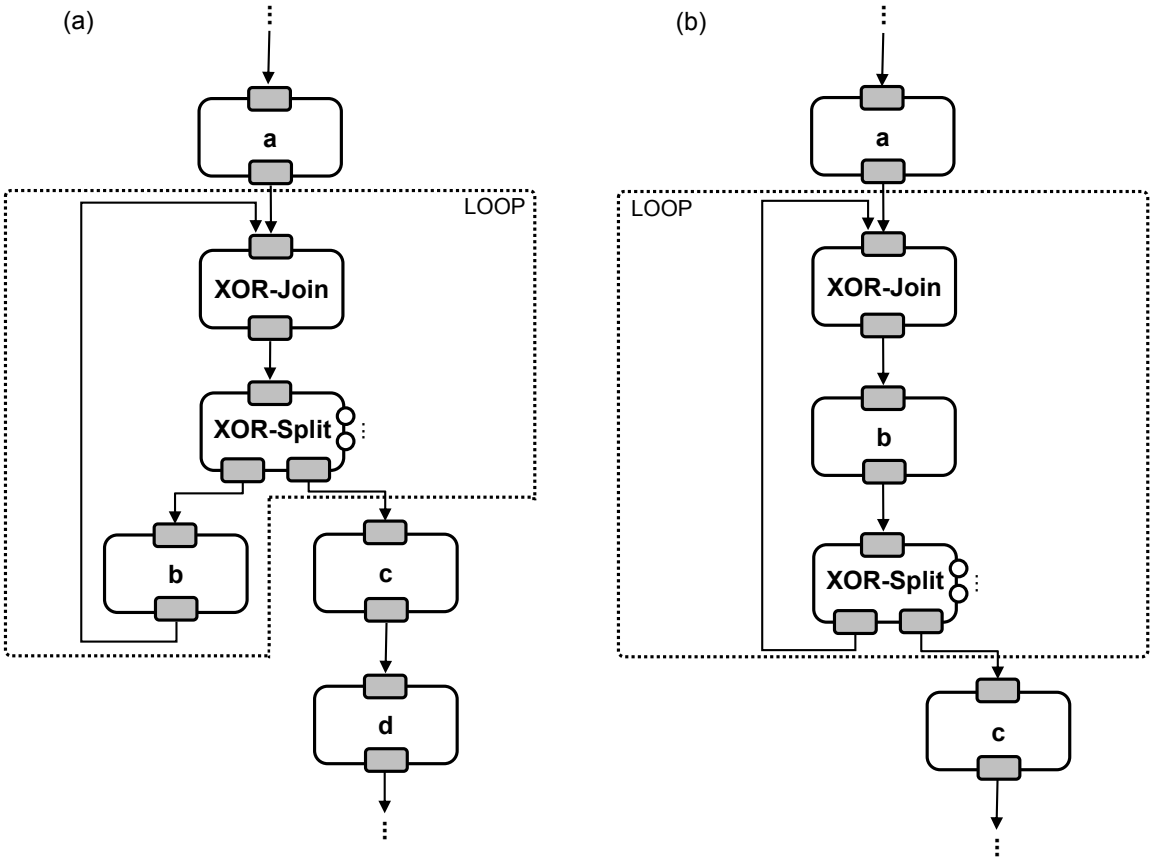


Abbildung 4.6.: Modellierung schleifenbasierter Kontrollflussmuster in PGM

Durch Kombination der oben vorgestellten Split- und Join-Aktivitätstypen lassen sich in PGM vier grundlegende Kontrollflussmuster definieren. Deren formale Definition ist in Anhang A.5 zu finden: Sequenz (`SEQ`), alternative Verzweigung (`ALT`), parallele Verzweigung (`PAR`) und Schleife (`LOOP`).

Es gibt für die Modellierung eines `SEQ`-Kontrollflussmusters keinen spezifischen Aktivitätstypen (siehe Definition 32), weil Kontrollflussabhängigkeiten in PGM explizit dargestellt werden. Stattdessen werden Kontrollflusskanten benutzt, um sequentiell ausgeführte Aktivitäten bzw. Kontrollflussmuster miteinander zu verbinden. Wie in Abbildung 4.5 (a) gezeigt, existiert eine Kontrollflusskante zwischen Aktivität a und Aktivität b. Somit wird a vor b ausgeführt (abkürzende Schreibweise: $a < b$). Folglich ist b ein *direkter Nachfolger* von a bzw. a ist ein *direkter Vorgänger* von b (siehe Definition 37). Ebenso ist a ein Vorgänger von c, da b direkter Vorgänger von c ist.

Abbildung 4.5 (b) und (c) veranschaulicht die Kontrollflussmuster `ALT` und `PAR` für die Modellierung alternativer und paralleler Kontrollflusspfade in PGM (siehe Definitionen 33 und 34). Die Kontrollflüsse werden jeweils durch *XOR-Split*- bzw. *AND-Split*-Aktivitäten aufgespalten, wobei die Aktivitäten b und c anschließend in (b) alternativ (abkürzende Schreibweise: $b \otimes c$) und in (c) parallel (abkürzende Schreibweise: $b \,||\, c$) ausgeführt werden.

Über eine *XOR-Join*- bzw. *AND-Join*-Aktivität werden die Kontrollflusspfade wieder verschmolzen bzw. synchronisiert.

Das `LOOP`-Kontrollflussmuster führt abhängig von einem gegebenen Schleifenprädikat eine einzelne Aktivität bzw. ein einzelnes Kontrollflussmuster genau $n \geq 0$ mal aus. Dieses Kontrollflussmuster lässt sich mit den Aktivitäten *XOR-Split*, *XOR-Join* und *Set-Iterator* in verschiedenen Varianten darstellen.

Abbildung 4.6 (a) zeigt eine Schleifenvariante mit einem *XOR-Join*/*XOR-Split*-Aktivitätspaar (siehe Definition 35). Der *XOR-Split* definiert zwei alternative Pfade, die abhängig von Prädikaten ausgewählt werden können. Solange die Bedingung `true` ist, wird Aktivität *b* ausgeführt, und der Kontrollfluss wird an die *XOR-Join*-Aktivität zurückgeführt. Ist dagegen die Bedingung `false`, wird der Kontrollfluss an Aktivität *c* weitergeleitet. Diese Schleifenvariante lässt sich auch realisieren, wenn die *XOR-Split*- durch eine *Set-Iterator*-Aktivität ersetzt wird. In diesem Fall iteriert der *Set-Iterator* über eine Datenmenge und liefert in jeder Schleifeniteration das nächste Tupel der Datenmenge. Ist kein Tupel mehr vorhanden, wird die Schleife wieder verlassen.

Abbildung 4.6 (b) zeigt eine alternative Variante einer Schleife, bei der eine Bedingung nicht am Schleifenanfang, sondern am Schleifenende evaluiert wird (siehe Definition 36). Entsprechend wird die *XOR-Split*-Aktivität am Ende der Schleife positioniert.

Die hier vorgestellten PGM-Kontrollflussmuster reichen zur Darstellung der Ausführungslogik von prozedural modellierten, kurzlaufenden BPEL/SQL-Workflows bzw. mit SQL/PSM definierten Stored-Procedures aus (siehe Transformationstabellen in Anhang C). Nichtsdestotrotz gibt es weitere Kontrollflussmuster in Workflowbeschreibungssprachen, die mit diesen grundlegenden Kontrollflussmustern nicht dargestellt werden können [AHKB03, RHAM06]. Wegen der Erweiterbarkeit von PGM können bei Bedarf aus den generischen Typen der einzelnen Komponenten von PGM neue Aktivitätstypen abgeleitet werden, mit denen die gewünschten Kontrollflussmuster modelliert werden können.

SQL-Aktivitätstypen

PGM definiert SQL-Aktivitäten (siehe Abbildung 4.4), welche die Ausführung von SQL-Anweisungen oder den Aufruf von Stored-Procedures repräsentieren.

Eine SQL-Aktivität (siehe Definition 21) interagiert mit einem Partner vom Typ `RDBMS` und besitzt einen einfachen ICS bzw. OCS. Ein weiterer Bestandteil einer SQL-Aktivität ist die Definition der SQL-Anweisung, die von der SQL-Aktivität an den assoziierten Partner zur Ausführung gesendet wird. Die explizite Repräsentation dieser SQL-Anweisung ist erforderlich, weil eine SQL-Anweisung Gegenstand der Optimierung ist und somit explizit in einer PGM-Repräsentation verfügbar sein muss.

Da die Restrukturierung von SQL-Anweisungen im Mittelpunkt der PGM-Optimierung steht, müssen alle in einer SQL-Anweisung referenzierten Tabellen und Parameter in PGM explizit dargestellt werden, um darauf aufbauend Datenabhängigkeiten zwischen SQL-Anweisungen berechnen zu können. Daraus resultieren folgende Schritte zur Ableitung der PGM-Repräsentation einer SQL-Anweisung (siehe Definition 20):

1. Zunächst werden alle in einer SQL-Anweisung referenzierten Tabellen und Parameter bestimmt und durch PGM-Variablen ersetzt. Da der SQL-Standard nur skalare Werte als Eingabeparameter zulässt, werden die Parameter auf `SIMPLE`-Variablen abgebildet. Entsprechend werden Tabellen mit `SET`-Variablen repräsentiert. Da bei Ausgabeparametern, die beispielsweise bei Stored-Procedure-Aufrufen auftreten können, sowohl skalare Werte als auch Tabellen erlaubt sind, können hier Variablen beider Datentypen auftreten.
2. Nach diesem Substitutionsschritt wird die Lese- bzw. Schreibmenge einer SQL-Anweisung bestimmt. Es wird überprüft, ob ein lesender oder schreibender Zugriff auf die generierten Variablen erfolgt. Eine Variable, der ein Eingabeparameter zu Grunde gelegt ist, wird in einer SQL-Anweisung immer lesend referenziert. Im Gegensatz dazu erfolgt ein schreibender Zugriff immer auf eine Variable, die auf einem Ausgabeparameter basiert. DDL-Anweisungen, wie Create und Drop, bzw. DML-Anweisungen, wie Insert, Update, Delete oder Merge, definieren Schreiboperationen auf einer zu aktualisierenden Tabelle und somit auf der zugehörigen `SET`-Variablen. Dagegen handelt es sich immer um einen lesenden Zugriff auf eine `SET`-Variable, die von einer Tabelle, die in einer Anfrage referenziert wird, abgeleitet wurde.
3. Nach Ermittlung der Lese- und Schreibmengen einer SQL-Anweisung werden die darin enthaltenen Variablenmengen mit Dataslots assoziiert. Variablen, die zur Lesemenge einer Anweisung gehören, werden mit `Read`-Dataslots und Variablen, die Teil der Schreibmenge einer Anweisung sind, mit `Write`-Dataslots verknüpft. Auf diese Weise werden die innerhalb einer SQL-Anweisung referenzierten Tabellen und Parameter „nach außen hin“ sichtbar gemacht. Erst durch diesen Schritt wird im Folgenden eine effiziente Berechnung der Datenabhängigkeiten zwischen den einzelnen SQL-Anweisungen bzw. SQL-Aktivitäten möglich.

Gemäß der in diesem Optimierungsansatz betrachteten vier SQL-Anweisungstypen (DDL, DML, Query und Stored-Procedure) existieren in PGM entsprechend vier SQL-Aktivitätstypen: *SQL-DDL*, *SQL-DML*, *SQL-Query* und *SQL-SP*.

Diese Spezifizierung ist sinnvoll, da hierdurch die Definition der Restrukturierungsregeln vereinfacht wird. Insbesondere kann vom SQL-Aktivitätstypen direkt auf die ausgeführte SQL-Anweisung geschlossen werden. Dadurch wird die Suche nach „geeigneten“ SQL-Aktivitäten bei einer Regelanwendung erheblich erleichtert.

Alle SQL-Aktivitätstypen weisen dieselbe Struktur auf. Sie unterscheiden sich nur in der Anzahl und Art der verfügbaren Dataslots, die sich aus der Art einer SQL-Anweisung bzw. aus den darin referenzierten Tabellen und Parametern ergeben.

Eine *SQL-Query*-Aktivität (siehe Definition 22) definiert beispielsweise eine SQL-Anfrage, deren Lesemenge alle Variablen enthält, denen die in der Anfrage referenzierten Tabellen und Eingabeparameter zu Grunde gelegt sind. Die Variablen sind mit entsprechenden `Read`-Dataslots verknüpft. Zusätzlich zu diesen `Read`-Dataslots besitzt eine *SQL-Query*-Aktivität einen oder mehrere `Write`-Dataslots. Im Falle einer Select-Anfrage existiert genau ein `Write`-Dataslot, der mit einer `SET`-Variablen, welche die Ergebnismenge der Select-Anfrage speichert, verknüpft ist. Liegt dagegen eine Select-Into-Anfrage vor, exis-

tiert ein entsprechender `Write`-Dataslot für jede Variable in der Into-Klausel, in die ein Attributwert des von der Select-Into-Anfrage selektierten Tupels gespeichert wird.

Eine *SQL-SP*-Aktivität (siehe Definition 23) ruft eine Stored-Procedure auf, die Eingabe- und Ausgabeparameter besitzen kann. Eingabeparameter sind immer skalare Werte, wohingegen als Ausgabeparameter sowohl einzelne skalare Werte als auch Ergebnismengen geliefert werden können. Die Eingabeparameter werden der Lesemenge und die Ausgabeparameter entsprechend der Schreibmenge des Stored-Procedure-Aufrufs zugeordnet. Dies wird in den `Read`- und `Write`-Dataslots der *SQL-SP*-Aktivität reflektiert.

Eine DML-Anweisung wird von einer *SQL-DML*-Aktivität ausgeführt (siehe Definition 24). Diese Aktivität besitzt genau einen `Write`-Dataslot, der mit der `SET`-Variablen, welche die Tabelle repräsentiert, auf die eine Insert-, Update-, Delete- oder Merge-Operation ausgeführt wird, verknüpft ist. Alle restlichen Variablen der DML-Anweisung gehören zu deren Lesemenge und werden folglich auf `Read`-Dataslots abgebildet.

Eine *SQL-DDL*-Aktivität (siehe Definition 25) führt eine DDL-Anweisung aus und besitzt eine ähnliche Struktur wie eine *SQL-DML*-Aktivität. Der einzige `Write`-Dataslot ist mit der `SET`-Variablen verknüpft, welche die Tabelle repräsentiert, auf die eine Create- oder Drop-Anweisung ausgeführt wird. Außerdem kann eine *SQL-DDL*-Aktivität im Falle einer Create-Anweisung `Read`-Dataslots enthalten. Dies ist möglich, weil eine Tabelle von einer existierenden Tabelle abgeleitet werden kann. Dies kann durch eine entsprechende Unteranfrage innerhalb einer Create-Anweisung umgesetzt werden. Die in dieser Unteranfrage enthaltenen Variablen sind, wie bei einer SQL-Anfrage bereits erläutert, Teil der Lesemenge und werden somit mit `Read`-Dataslots verknüpft.

Aktivitäten zur Modellierung von Optimierungssphären

Aufgrund der Struktur eines zu optimierenden datenintensiven Workflows müssen Optimierungsgrenzen, die bei der Benutzung der Regelbasis von PGM/F nicht überschritten werden dürfen, berücksichtigt werden. Dies ist notwendig, um Änderungen an der ursprünglichen Ausführungssemantik eines Workflowmodells durch die Regelanwendungen zu vermeiden. Die Optimierungsgrenzen definieren in PGM einen geschlossenen Optimierungsbereich, eine sogenannte *Optimierungssphäre* (siehe Definition 31). Alle Effekte einer Regel bleiben immer lokal auf eine Optimierungssphäre begrenzt. In PGM werden Optimierungssphären durch ein *Start*/*End*-Aktivitätspaar repräsentiert (siehe Abbildung 4.4). Da beide Aktivitätstypen lediglich als Markierung dienen, besitzen sie neben einem einfachen ICS und OCS keine weiteren Komponenten.

Die Verwendung von Optimierungssphären in PGM ermöglicht eine einheitliche Behandlung verschiedenster Optimierungsgrenzen, die sowohl in einem datenintensiven Workflowmodell als auch in einer SQL-basierten Stored-Procedure definiert sein können.

Beispielsweise definiert eine <scope> Aktivität in einem kurzlaufenden BPEL/SQL-Workflow eine solche sprachspezifische Optimierungsgrenze. In einer <scope> Aktivität kann ein <faultHandler> beliebiger Komplexität, der das Verhalten einer <scope> Aktivität im Fehlerfall definiert, deklariert werden. Werden Restrukturierungsregeln angewendet, die Aktivitäten sowohl innerhalb als auch außerhalb einer <scope> Aktivität betreffen, wird die Anpassung der zugehörigen Fehlerbehandlung notwendig, um die ursprüngliche Ausführungssemantik einer <scope> Aktivität zu erhalten. Eine solche Anpassung ist aber eine schwierige Aufgabe. Es müssen die durch die Regelanwendung betroffenen Teile der Fehlerbehandlung identifiziert und entsprechend angepasst werden, was im Allgemeinen nicht möglich ist. Außerdem würde eine Anpassung zu komplexeren Regelbeschreibungen führen. Um sicherzustellen, dass die Effekte einer Regelanwendung die Grenzen einer <scope> Aktivität nicht überschreiten, wird eine <scope> Aktivität in PGM durch eine Optimierungssphäre repräsentiert.

Die Fehlerbehandlung einer <scope> Aktivität darf in einer PGM-Repräsentation nicht vollständig ausgeblendet werden. Vielmehr müssen Datenabhängigkeiten zwischen der Fehlerbehandlung und den restlichen Aktivitäten einer Optimierungssphäre explizit dargestellt werden. Dies ist für eine korrekte Regelanwendung innerhalb einer Optimierungssphäre zwingend erforderlich, um zu verhindern, dass Variablen, die in der Fehlerbehandlung referenziert werden, durch Anwendung einer Restrukturierungsregel entfernt werden. Zu diesem Zweck steht eine *Exception*-Aktivität, die mit einer Optimierungssphäre assoziiert ist, zur Verfügung. Die *Exception*-Aktivität besitzt lediglich Dataslots, die mit Variablen, die innerhalb der Fehlerbehandlung referenziert werden, verknüpft sind.

Auch eine BPEL <invoke> Aktivität kann mit einer lokalen Fehlerbehandlung verknüpft sein (<catch> Element). Eine korrekte Regelanwendung unter Einbeziehung einer <invoke> Aktivität ist nur unter Anpassung der lokalen Fehlerbehandlung möglich. Da eine solche Anpassung bei der PGM-Optimierung nicht durchgeführt wird, wird eine solche <invoke> Aktivität auf eine *Generic*-Aktivität in PGM abgebildet. Auch hier müssen die Variablen, die in der lokalen Fehlerbehandlung referenziert werden, durch entsprechende Dataslots der *Generic*-Aktivität nach außen sichtbar gemacht werden, um eine korrekte Darstellung aller Datenabhängigkeiten in einem Workflowmodell zu gewährleisten.

Transaktionsgrenzen sind eine weitere sprachspezifische Optimierungsgrenze, die insbesondere bei der isolierten Optimierung von Stored-Procedures zu berücksichtigen sind. Solche Grenzen müssen in PGM ebenfalls als Optimierungssphäre repräsentiert werden, um zu verhindern, dass die Effekte einer Regelanwendung Transaktionsgrenzen überqueren können. Dadurch kann die ursprüngliche Ausführungssemantik einer Stored-Procedure geändert werden. Dagegen sind Transaktionsgrenzen bei der Optimierung eines kurzlaufenden BPEL/SQL-Workflows nicht von Bedeutung, da in diesem Fall alle Aktivitäten, inklusive aller Stored-Procedure-Aufrufe, in derselben Transaktion ausgeführt werden.

In dem in Abbildung 2.2 dargestellten Beispiel-Workflow existiert genau eine Optimierungssphäre. Der BPEL/SQL-Workflow definiert implizit eine globale <scope> Aktivität (nicht dargestellt), die mit einer Fehlerbehandlung verknüpft sein kann. Gleichzeitig definiert die <scope> Aktivität die Grenze der kurzlaufenden Transaktion innerhalb derer alle Aktivitäten inklusive der Stored-Procedure *PrepareApprovedOrders*

atomar ausgeführt werden. Abbildung 4.7 zeigt die resultierende PGM-Repräsentation des Beispiel-Workflows mit der umschließenden Optimierungssphäre, die durch das *Start*/*End*-Aktivitätspaar repräsentiert wird.

Abbildung 4.8 veranschaulicht die PGM-Repräsentation der Stored-Procedure *PrepareApprovedOrders* in einem isolierten Optimierungsszenario. Die Logik der PGM-Repräsentation wird von einer Optimierungssphäre umschlossen. Diese repräsentiert den Transaktionskontext, innerhalb dessen die Stored-Procedure bei einer Einzelausführung abläuft.

In einem heterogenen, kombinierten Optimierungsszenario entfällt diese Optimierungssphäre, da die PGM-Repräsentation der Stored-Procedure mit der PGM-Repräsentation des Beispiel-Workflows verschmolzen wird, und somit Teil der kurzlaufenden Transaktion ist, die durch die globale <scope> Aktivität des Beispiel-Workflows definiert wird. In Abbildung E.15 im Anhang ist die kombinierte PGM-Repräsentation dargestellt.

Eine weitere spezielle Optimierungssphäre bildet die sogenannte *Loop-Optimierungssphäre* (LOS), die alle Aktivitäten eines `LOOP`-Kontrollflussmusters umfasst. In Analogie zu den allgemeinen Optimierungssphären bleiben alle Effekte einer Regelanwendung auf eine LOS beschränkt. Dies ist notwendig, weil Aktivitäten nicht aus einem Schleifenrumpf entfernt bzw. hinzugefügt werden können, ohne dass die ursprüngliche Ausführungssemantik einer Aktivität verletzt wird (siehe Teilkapitel 5.4.1).

Weitere Aktivitätstypen

Ein synchroner Web-Service-Aufruf wird in PGM durch eine *Invoke*-Aktivität repräsentiert (siehe Definition 27). Dieser Aktivitätstyp besitzt einen Partnerslot, der mit dem Web-Service assoziiert ist, dessen Operation aufgerufen wird. Außerdem steht ein einfacher ICS bzw. OCS zur Einbettung einer *Invoke*-Aktivität in die PGM-Ausführungslogik zur Verfügung. `Read`- und `Write`-Dataslots repräsentieren Zugriffe auf Variablen, deren Werte entweder als Eingabeparameter gelesen oder als Ausgabeparameter des Web-Service-Aufrufs geschrieben werden. Als zusätzliches Element wird der Name der aufgerufenen Web-Service-Operation gespeichert, um später die *Web-Service-Pushdown*-Regel korrekt anwenden zu können (siehe Teilkapitel 5.3).

Die Tabelle 4.1 zeigt eine Klassifikation verschiedener Typen von Web-Service-Aufrufen, die für die PGM-Optimierung von Bedeutung sind. Die Klassifikation basiert zum einen auf der Kardinalität und zum anderen auf den PGM-Datentypen der Ein- und Ausgabeparameter einer aufgerufenen Web-Service-Operation. Diese Informationen lassen sich aus der WSDL-Beschreibung eines Web-Services direkt ableiten. Die Web-Service-Aufrufstypen 1-4 lesen Eingabe- bzw. liefern Ausgabewerte, die in PGM durch Variablen vom Typ `SIMPLE` bzw. `SET` dargestellt werden können. Ein Web-Service-Aufruf vom Typ 5 repräsentiert dagegen alle Aufrufe von Web-Service-Operationen, die gemäß dieser Klassifikation nicht den ersten vier Gruppen zugeordnet werden können. Da ein solcher Web-Service-Aufruf für die PGM-Optimierung nicht relevant ist, wird er in einer PGM-Repräsentation durch eine *Generic*-Aktivität gekapselt. Dagegen werden die für die PGM-Optimierung relevanten Web-Service-Aufrufstypen 1-4 auf eine *Invoke*-Aktivität abgebildet und somit explizit für den PGM-Optimierer sichtbar gemacht. Voraussetzung hierfür

Klasse	Eingabeparameter		Ausgabeparameter	
	Kardinalität	PGM-Datentyp	Kardinalität	PGM-Datentyp
Typ 1	$n \geq 1$	`SIMPLE`	$m \geq 1$	`SIMPLE`
Typ 2	$n \geq 1$	`SIMPLE`	$m = 1$	`SET`
Typ 3	$n \geq 1$	`SET`	$m \geq 1$	`SIMPLE`
Typ 4	$n \geq 1$	`SET`	$m = 1$	`SET`
Typ 5	sonst			

Tabelle 4.1.: Klassifikation von Web-Service-Aufrufen

ist, dass ein Web-Service-Aufruf über SOAP/HTTP durchgeführt werden kann. Dies lässt sich aus der WSDL-Beschreibung eines Web-Services ermitteln (<`binding`> Element). Anderenfalls ist ein Web-Service-Aufrufstyp 1-4 nicht für eine *Web-Service-Pushdown*-Regel, die in Teilkapitel 5.3 vorgestellt wird, geeignet. In einem solchen Fall ist auch ein Web-Service-Aufruf vom Typ 1-4 durch eine *Generic*-Aktivität zu repräsentieren.

Folglich wird eine BPEL <`invoke`> Aktivität nur dann auf eine *Invoke*-Aktivität in PGM abgebildet, wenn sie eine Web-Service-Operation vom Typ 1-4 über SOAP/HTTP aufruft und mit keiner lokalen Fehlerbehandlung assoziiert ist.

Auch einfache Variablenzuweisungen der Form $v_i := v_j$ spielen bei der PGM-Optimierung eine wichtige Rolle. Eine *Assign*-Aktivität (siehe Definition 28) repräsentiert eine solche Variablenzuweisung auf der PGM-Ebene. Sie besitzt einen einfachen ICS bzw. OCS. Zusätzlich gibt es einen `Read`- bzw. `Write`-Dataslot. Der `Write`-Dataslot ist mit der Variablen, der ein neuer Wert zugewiesen wird, verknüpft. Der neue Wert ist in einer Variablen gespeichert, die mit dem `Read`-Dataslot der Aktivität assoziiert ist.

Bei einer *Assign-Select*-Aktivität (siehe Definition 29) handelt es sich um eine Spezialisierung einer *Assign*-Aktivität. Dieser Aktivitätstyp repräsentiert eine Variablenzuweisung zwischen `SET`-Variablen. Dabei wird das bereits bei der *XOR-Split-Select*-Aktivität vorgestellte Selektionsprädikat σ auf einer `SET`-Variablen angewendet, um daraus bestimmte Tupel, die in einer zweiten `SET`-Variablen zur Verfügung gestellt werden, zu selektieren. Die `Read`-Dataslots der Aktivität sind mit den Eingabeparametern des Selektionsprädikats σ bzw. mit der `SET`-Variablen, auf die σ angewendet wird, verknüpft. Der einzige `Write`-Dataslot ist entsprechend mit der `SET`-Variablen, in die das Ergebnis der Selektionsoperation geschrieben wird, assoziiert.

Somit werden bei der Abbildung eines BPEL/SQL-Workflows nach PGM nur solche <`assign`> Aktivitäten in einem BPEL/SQL-Workflow auf eine PGM *Assign*- oder *Assign-Select*-Aktivität abgebildet, die entweder eine einfache Variablenzuweisung oder einen XPath-Ausdruck, der unmittelbar auf das Selektionsprädikat σ abgebildet werden kann, definieren. Andere Arten von Variablenzuweisungen sind für die PGM-Optimierung nicht von Bedeutung und werden deshalb in einer *Generic*-Aktivität gekapselt.

4.3.5. Modellierung von Abhängigkeiten

Daten-, Kontrollfluss- und Kommunikationsabhängigkeiten werden in PGM zur Vereinfachung der Definition der Restrukturierungsregeln explizit repräsentiert. Kontrollflussabhängigkeiten werden direkt von den Kontrollflussstrukturen eines Workflowmodells abgeleitet und auf die entsprechenden PGM-Kontrollflussmuster `SEQ`, `ALT`, `PAR` und `LOOP` abgebildet (siehe Definitionen 32 und 36). Ebenso ergeben sich Kommunikationsabhängigkeiten (siehe Definition 42) unmittelbar aus der Verbindung einer Aktivität zu seinem Partner, die in einem Workflowmodell typischerweise explizit beschrieben ist. Die Datenabhängigkeiten zwischen den Aktivitäten müssen dagegen berechnet werden. Hierzu werden in einem ersten Schritt für jede Aktivität die Variablenmengen, die von einer Aktivität lesend bzw. schreibend referenziert werden, bestimmt. Daraus werden die `Read`- und `Write`-Dataslots einer Aktivität abgeleitet. Auf Grundlage dieser Informationen und den gegebenen Kontrollflussabhängigkeiten können die Datenabhängigkeiten zwischen den Aktivitäten berechnet werden. Zur Berechnung der Datenabhängigkeiten werden Standard-Algorithmen aus dem Bereich des Compilerbaus, die in dieser Arbeit nicht weiter vertieft werden, verwendet. Details hierzu findet man z.B. in [ALSU06].

Die berechneten Datenabhängigkeiten werden durch Datenflusskanten repräsentiert, die Dataslots verschiedener Aktivitäten, welche dieselben Variablen referenzieren, miteinander verbinden (siehe Definition 40). Über die beiden Typen der miteinander verbundenen Dataslots lässt sich die Art der vorliegenden Datenabhängigkeit unmittelbar ableiten. Auf diese Weise können Schreib-Schreib-, Schreib-Lese-, Lese-Schreib- und Lese-Lese-Datenabhängigkeiten in einer PGM-Repräsentation explizit dargestellt werden. Allerdings wird in nachfolgenden PGM-Repräsentationen auf die Darstellung von Lese-Lese-Datenabhängigkeiten verzichtet, da sie für die Definition und Anwendung der in dieser Arbeit vorgestellten Restrukturierungsregeln nicht relevant sind.

Eine sequentielle Ausführung zweier Aktivitäten ist Voraussetzung für die Existenz einer solchen Datenabhängigkeit. Dagegen existieren bei einer alternativen Ausführung beider Aktivitäten keine Datenabhängigkeiten, weil zur Laufzeit immer genau eine der beiden Aktivitäten exklusiv ausgeführt wird. Bei einer parallelen Ausführung können Datenabhängigkeiten nur existieren, wenn beide Aktivitäten auf dieselbe Variable schreibend bzw. lesend zugreifen. Durch den parallelen Zugriff ist die Referenzierungsreihenfolge und damit die Wertebelegung dieser Variablen nach der Ausführung beider Aktivitäten statisch nicht zu bestimmen. Da die Modellierung einer solchen Anomalie nicht sinnvoll ist, wird in dieser Arbeit der parallele Zugriff auf eine Variable nicht weiter berücksichtigt. Mit anderen Worten, es wird davon ausgegangen, dass keine Anomalien und somit keine Datenabhängigkeiten zwischen parallel ausgeführten Aktivitäten existieren.

4.3.6. Erzeugung einer PGM-Repräsentation

Die Überführung eines Workflowmodells in eine PGM-Repräsentation erfolgt mittels Transformationsregeln, welche die Abbildung zwischen den Konstrukten eines Workflowmodells und denen von PGM definieren. Bei der Transformation werden alle Informationen eines Workflowmodells, die für die Regelanwendung relevant sind, in PGM explizit

dargestellt. Dazu gehören (i) die Partner, mit denen die Aktivitäten eines Workflows kommunizieren, (ii) die Aktivitäten und Kontrollflussstrukturen, die in der Ausführungslogik definiert sind, sowie (iii) die Variablen, die von den Aktivitäten des Workflowmodells referenziert werden. Nachfolgend wird die Überführung eines Workflowmodells in eine PGM-Repräsentation am Beispiel des in Abbildung 2.2 gezeigten BPEL/SQL-Workflows veranschaulicht. Die resultierende PGM-Repräsentation ist in Abbildung 4.7 dargestellt. In Anhang C ist eine Transformationstabelle enthalten, die zeigt, wie die Konstrukte von BPEL/SQL bzw. SQL/PSM nach PGM abgebildet werden.

Die Informationen über Partner und Variablen können unmittelbar aus der Beschreibung des Beispiel-Workflows ausgelesen werden (<`partnerLinks`>, <`variables`>). In dem Beispiel werden alle SQL-Anweisungen gegen ein einzelnes Datenbanksystem *OrderDB* ausgeführt. Ferner ruft die <`invoke`> Aktivität den Web-Service *OrderFromSupplier* auf. Folglich existieren in der PGM-Repräsentation hierfür zwei Partner: *OrderDB* vom Typ `RDBMS` (p_2) und *OrderFromSupplier* vom Typ `WS` (p_3). Auch der Aufrufer des Beispiel-Workflows wird als Partner *Orderer* vom Typ `GENERIC` (p_1) repräsentiert. Da es sich bei *OrderDB* um ein DB2-Datenbanksystem handelt, besitzt der Partner die Eigenschaften `WS-UDTF` und `SP`, die anzeigen, dass Aufrufe von Web-Services durch benutzerdefinierte Funktionen und von Stored-Procedures unterstützt werden.

Die im Beispiel-Workflow in Abbildung 2.2 mit # gekennzeichneten Variablen werden auf entsprechende PGM-Variablen abgebildet (v_1-v_{10}). Dabei leiten sich die Datentypen der PGM-Variablen direkt aus den Datentypen der zu Grunde liegenden BPEL/SQL-Variablen ab. In diesem Beispiel werden skalare Datentypen auf einen `SIMPLE`- und Tabellen bzw. Materialisierungen auf einen `SET`-Datentypen abgebildet. Zusätzlich wird `SET`-Variable *SR_ApprovedOrders* (v_3) als temporäre Tabelle ([T]) markiert. Diese Information kann im Beispiel-Workflow unmittelbar aus den mit der zugehörigen Set-Referenz-Variablen assoziierten Create- und Drop-Anweisungen abgeleitet werden.

Im nächsten Schritt wird die Ausführungslogik der PGM-Repräsentation erzeugt, die von einer *Source*- und *Sink*-Markierung, welche den Start- und den Endpunkt des Kontrollflusses darstellen, umschlossen wird. Diese Markierungen werden, wie aus Abbildung 4.7 ersichtlich, als Kreise dargestellt. Die in der Ausführungslogik des Beispiel-Workflows definierten Aktivitätstypen werden schrittweise auf entsprechende PGM-Kontrollflussmuster bzw. PGM-Aktivitätstypen abgebildet (a_1-a_{10}). Da es sich in diesem Beispiel um einen kurzlaufenden BPEL/SQL-Workflow handelt, wird die Ausführungslogik der PGM-Repräsentation von einem *Start*/*End*-Aktivitätspaar (a_1, a_{10}) umschlossen. Das Aktivitätspaar repräsentiert die Grenzen der zugehörigen Optimierungssphäre, welche durch die globale <`scope`> Aktivität des Beispiel-Workflows definiert wird. Die Aktivität markiert gleichzeitig die Transaktionsgrenzen, innerhalb derer alle Aktivitäten des Workflows ausgeführt werden. Die <`sequence`> Aktivitäten *ProcessOrders* bzw. *ProcessCurrentOrder* werden jeweils auf ein `SEQ`-Kontrollflussmuster, in dem alle darin eingebetteten Aktivitäten über Kontrollflusskanten direkt miteinander verbunden sind, abgebildet. Die <`receive`> und <`reply`> Aktivitäten *ReceiveFromOrderer* und *ReplyToOrderer* spielen für die PGM-Optimierung keine Rolle und werden deshalb jeweils mittels einer generischen Aktivität repräsentiert (a_2, a_9). Die SQL-Aktivitäten des Beispiel-Workflows werden in PGM abhängig von den von ihnen ausgeführten SQL-Anweisungen auf entsprechen-

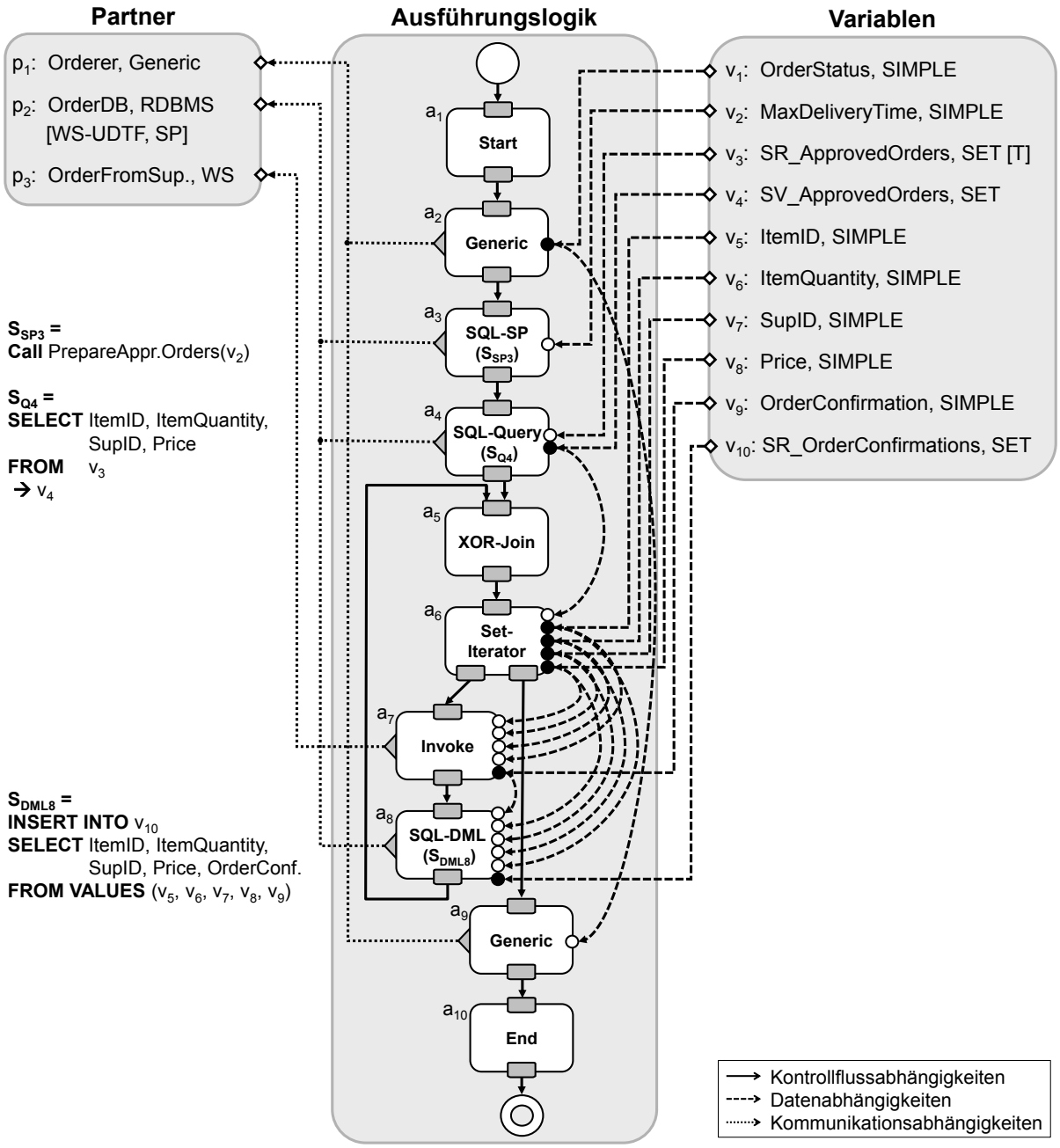


Abbildung 4.7.: PGM-Repräsentation des Beispiel-Workflows aus Abbildung 2.2

de SQL-Aktivitätstypen abgebildet: *PrepareApprovedOrders* auf eine *SQL-SP*-Aktivität (a_3), *RetrieveApprovedOrders* auf eine *SQL-Query*-Aktivität (a_4) und *InsertOrderConfirmation* auf eine *SQL-DML*-Aktivität (a_8). Die Schleife, die durch Aktivität *ForEachItemOrder* modelliert ist, wird in PGM auf ein `LOOP`-Kontrollflussmuster abgebildet. Das Kontrollflussmuster besteht aus einer *XOR-Join*- und einer *Set-Iterator*-Aktivität (a_5, a_6), die zusammen das iterative Auslesen der Datenmenge in *SV_ApprovedOrders* (v_4)

repräsentieren. Schließlich wird der synchrone Web-Service-Aufruf der Aktivität *OrderFromSupplier* durch eine *Invoke*-Aktivität (a_7) dargestellt.

Die Kontrollflussabhängigkeiten in der BPEL/SQL-Workflowbeschreibung können unmittelbar auf die PGM-Repräsentation übertragen werden. Die sequentielle Ausführung der Aktivitäten wird direkt über Kontrollflusskanten ausgedrückt; die Schleife wird über das `LOOP`-Kontrollflussmuster repräsentiert.

Ebenso können die Kommunikationsabhängigkeiten umgehend aus der Beschreibung einer BPEL/SQL-Aktivität abgeleitet werden. Dazu wird ein Partner mit dem Partnerslot einer Aktivität verbunden.

Zur Berechnung der Datenabhängigkeiten müssen zunächst die Lese- und Schreibmengen der zu Grunde liegenden BPEL/SQL-Aktivitäten bestimmt werden. Beispielsweise schreibt Aktivität *ReceiveFromOrderer* die vom Aufrufer empfangenen Daten in Variable *OrderStatus* (v_1). Umgekehrt liest Aktivität *ReplyToOrderer* die Variable *OrderStatus*, um eine Antwortnachricht an den Aufrufer zurückzusenden. Beim Parameter *MaxDeliveryTime* (v_2) handelt es sich um einen Eingabeparameter der Stored-Procedure *PrepareApprovedOrders*. Folglich wird der Parameter der Lesemenge der Aktivität zugeordnet. Die ForEach-Schleife liest die Datenmenge *SV_ApprovedOrders* (v_4) und schreibt in jeder Iteration das aktuelle Tupel in Variable *CurrentOrder*. Aktivität *OrderFromSupplier* liest die aktuellen Tupelwerte *ItemID* (v_5), *ItemQuantity* (v_6), *SupID* (v_7) und *Price* (v_8) von *CurrentOrder* als Eingabe und schreibt das Ergebnis des Web-Service-Aufrufs in die Variable *OrderConfirmation* (v_9).

Bei den <`sql`> Aktivitäten *RetrieveApprovedOrders* und *InsertOrderConfirmation* müssen die zugehörigen SQL-Anweisungen analysiert werden, um daraus die Lese- und Schreibmengen der beiden Aktivitäten bestimmen zu können. Aus Gründen der Übersichtlichkeit werden in Abbildung 4.7 die PGM-Repräsentationen der SQL-Anweisungen neben den zugehörigen SQL-Aktivitäten dargestellt.

Die Aktivität *RetrieveApprovedOrders* führt eine SQL-Anfrage auf der temporären Tabelle *ApprovedOrders* aus, die durch die Set-Referenz *SR_ApprovedOrders* (v_3) referenziert wird. Folglich gehört *SR_ApprovedOrders* zur Lesemenge und die `SET`-Variable *SV_ApprovedOrders* (v_4), in der die Materialisierung der Ergebnismenge gespeichert wird, zur Schreibmenge dieser Aktivität.

Da es sich bei der Insert-Anweisung um eine Schreiboperation auf der Tabelle *OrderConfirmations* handelt, ist die zugehörige `SET`-Variable *SR_OrderConfirmations* (v_{10}) Teil der Schreibmenge der *SQL*-Aktivität *InsertOrderConfirmation*. Die Variable *OrderConfirmation* (v_9) dient zusammen mit den aktuellen Tupelwerten von *CurrentOrder* (v_4-v_7) als Eingabeparameter für die Insert-Anweisung. Folglich werden diese Variablen der Lesemenge der Aktivität *InsertOrderConfirmation* zugeordnet. Zu beachten ist, dass bei der Abbildung der Insert-Anweisung nach PGM ein Normalisierungsschritt durchgeführt wird, bei dem die Insert-Values-Anweisung in eine entsprechende Insert-Select-Anweisung überführt wird. Ein solcher Normalisierungsschritt ist sinnvoll, damit während der Optimierung einer PGM-Regeldefinition nicht Insert-Select- und Insert-Values-Anweisungen getrennt voneinander betrachtet werden müssen.

Die Dataslots einer PGM-Aktivität können direkt aus den Lese- und Schreibmengen einer zugehörigen BPEL/SQL-Aktivität abgeleitet werden. Darauf aufbauend werden die Datenabhängigkeiten berechnet. Die erste Datenabhängigkeitskante beginnt immer bei der Definition einer Variablen, die mit dem Dataslot der ersten Aktivität, welche die Variable referenziert, verbunden ist. Die restlichen Datenabhängigkeiten werden durch Verbindungen zwischen den Dataslots in der PGM-Repräsentation dargestellt. Beispielsweise besteht eine Schreib-Lese-Datenabhängigkeit auf der Variablen *SV_ApprovedOrders* zwischen der *SQL-Query*- und der *Set-Iterator*-Aktivität, die von der *SQL-Query*-Aktivität geschrieben und von der *Set-Iterator*-Aktivität gelesen wird. Dies wird jeweils durch einen `Write`- bzw. `Read`-Dataslot angezeigt.

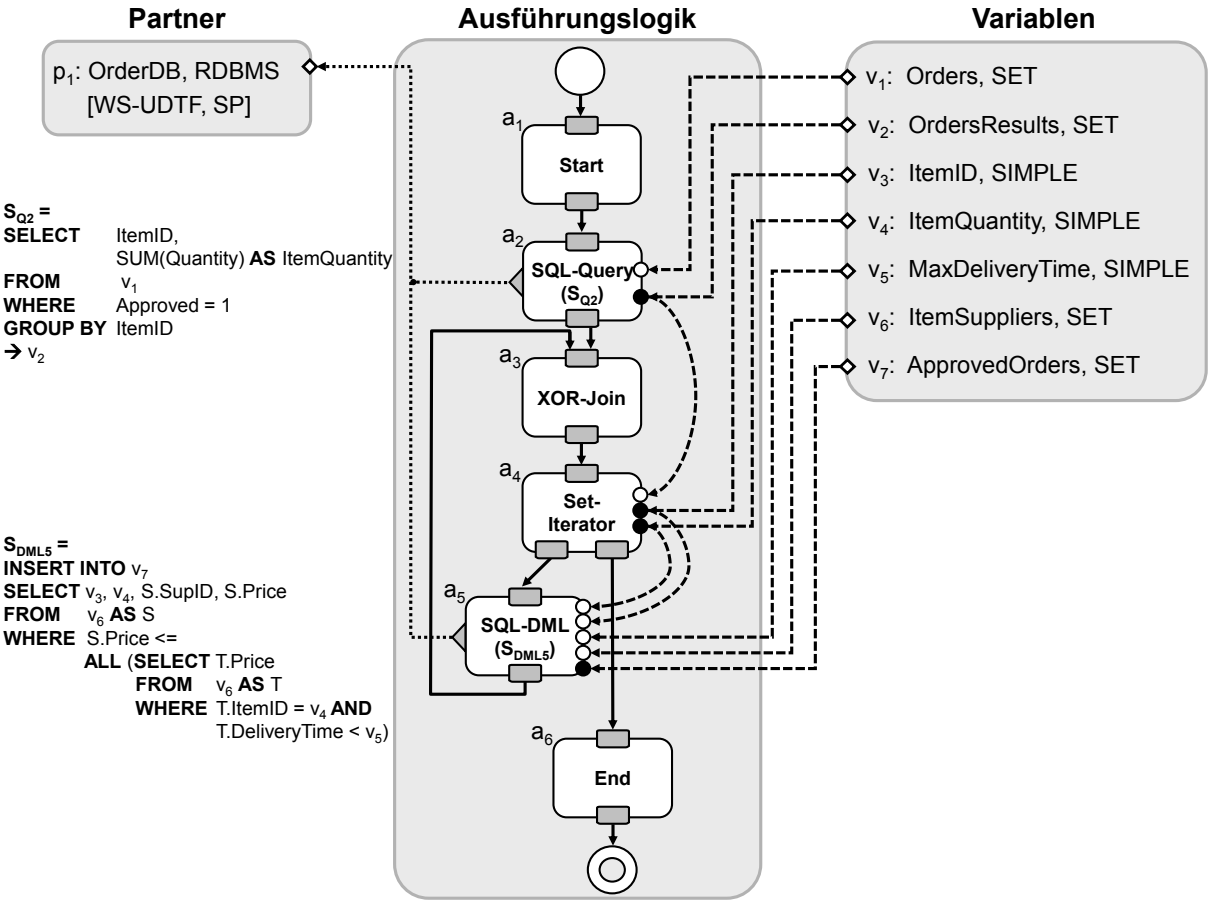


Abbildung 4.8.: PGM-Repräsentation der Stored-Procedure *PrepareApprovedOrders* aus Abbildung 2.2

Abbildung 4.8 stellt die PGM-Repräsentation der Stored-Procedure-Beschreibung dar, die analog zur PGM-Repräsentation des Beispiel-Workflows abgeleitet wird. *OrderDB* (p_1) ist der einzige Partner in der PGM-Repräsentation, weil die Stored-Procedure und somit alle darin definierten SQL-Anweisungen auf dieser Datenbank ausgeführt werden. Die *Start*- und *End*-Aktivitäten (a_1, a_6) repräsentieren die Optimierungssphäre, die durch die Transaktionsgrenzen, die bei einer isolierten Ausführung der Stored-Procedure implizit durch das Datenbanksystem gesetzt werden, gegeben sind. Der Cursor wird mit einem `LOOP`-Kontrollflussmuster bestehend aus einer *XOR-Join*- bzw. *Set-Iterator*-Aktivität (a_3,

a_4) definiert. Die Anfrage, an deren Ergebnismenge der Cursor *CurrentOrder* gebunden ist, wird durch eine *SQL-Query*-Aktivität (a_2) repräsentiert. Die Insert-Anweisung im Schleifenrumpf wird entsprechend durch eine *SQL-DML*-Aktivität (a_5) dargestellt. Die Variablen (v_1 - v_7) leiten sich aus dem Eingabeparameter der Stored-Procedure bzw. den in den Anweisungen eingelesenen Parametern und Tabellen ab. Der Eingabeparameter *MaxDeliveryTime* kann direkt auf eine PGM-Variable vom Typ `SIMPLE` abgebildet werden (v_5). Die in der Anfrage bzw. in der Insert-Anweisung referenzierten Tabellen *Orders*, *ApprovedOrders* und *ItemSuppliers* werden jeweils auf eine entsprechende `SET`-Variable (v_1, v_6, v_7) abgebildet. Ebenso existieren für die Attribute *ItemID* und *ItemQuantity* eines Tupels, das vom Cursor *CurrentOrder* in jeder Iteration geliefert wird, entsprechende Variablen vom Typ `SIMPLE` (v_3, v_4).

4.3.7. Semantik von PGM

Die Ausführungssemantik von PGM kann durch einen Tokenfluss beschrieben werden: Die Tokens wandern entlang der Kontrollflusskanten durch einen PGM-Graphen und werden in den ICSs und OCSs der Aktivitäten abgelegt. Dabei konsumiert eine Aktivität die in ihrem ICS vorhandenen Tokens und führt ihre zu Grunde gelegte Operation aus. Dazu werden Daten, die in den `Read`- bzw. `Write`-Dataslots einer Aktivität repräsentiert werden, gelesen bzw. geschrieben. Ebenso kann hierfür eine Interaktion mit einem Partner erforderlich sein. Nach Ausführung der Operation wird der Kontrollfluss an die direkten Nachfolger der Aktivität weitergeleitet. Hierzu stellt die Aktivität eine bestimmte Anzahl von Tokens in ihrem OCS zur Verfügung. Die Tokens werden aus dem OCS entfernt und über die angeschlossenen Kontrollflusskanten an die ICSs der direkten Nachfolger einer Aktivität übergeben. Auf diese Weise werden Tokens in einem PGM-Graphen beginnend bei der Source-Markierung von Aktivität zu Aktivität weitergereicht bis der Kontrollfluss die Sink-Markierung erreicht.

Eine Aktivität kann erst ausgeführt werden, wenn in ihrem ICS eine entsprechende Anzahl Tokens zur Verfügung steht. Die Anzahl der notwendigen Tokens hängt vom Typen eines ICS ab: Ein ICS vom Typ `SIMPLE` ist Ziel einer eingehenden Kontrollflusskante. Deshalb aktiviert bereits ein einzelnes Token die Ausführung der zugehörigen Aktivität. Ein ICS vom Typ `AND` ist dagegen Endpunkt mehrerer parallel ausgeführter Kontrollflusspfade. Da es sich bei diesem ICS-Typen um einen Synchronisationspunkt handelt, ist die Ausführung einer Aktivität erst möglich, wenn über jede eingehende Kontrollflusskante ein Token in den ICS abgelegt worden ist. Dagegen vereinigt ein ICS vom Typ `XOR` alternativ ausgeführte Kontrollflusspfade. Da immer nur einer dieser Kontrollflusspfade zur Laufzeit aktiv ist, kann auch nur über diesen einen Kontrollflusspfad ein Token in den ICS abgelegt werden. Aus diesem Grund kann die Ausführung einer Aktivität mit einem solchen ICS-Typen beginnen, wenn in ihrem ICS ein Token vorhanden ist.

Sobald die in einem ICS einer Aktivität erforderliche Anzahl an Tokens zur Verfügung steht, werden die Tokens aus dem ICS entfernt, und die Aktivität wird ausgeführt. Nach Ausführung der Aktivität muss der Kontrollfluss entsprechend der Ausführungssemantik einer Aktivität an die nachfolgenden Aktivitäten weitergeleitet werden. Dazu muss für

jede ausgehende Kontrollflusskante, auf welcher der Kontrollfluss übermittelt werden soll, ein entsprechendes Token erzeugt werden. Die Tokens werden im OCS der Aktivität zur Verfügung gestellt, daraus entfernt und gleichzeitig über die entsprechenden angeschlossenen Kontrollflusskanten an die ICSs der nachfolgenden Aktivitäten übertragen.

Die Anzahl der zu erzeugenden Tokens hängt zum einen vom OCS-Typen und zum anderen von der Ausführungssemantik einer Aktivität ab. Ein OCS vom Typ `SIMPLE` ist der Startpunkt einer ausgehenden Kontrollflusskante. Deshalb muss für diesen OCS-Typen lediglich ein einzelnes Token erzeugt werden. Ein OCS vom Typ `SPLIT` ist dagegen der Ausgangspunkt mehrerer ausgehender Kontrollflusskanten, die parallel ausgeführt werden. Daher ist hier für jede der ausgehenden Kontrollflusskanten jeweils ein Token zu generieren. Ist eine Aktivität der Ausgangspunkt alternativ ausgeführter Kontrollflusspfade, wie z.B. eine *XOR-Split-* oder *Set-Iterator*-Aktivität, besitzt sie mehrere OCSs vom Typ `SIMPLE`. In diesem Fall wird nur für eine der ausgehenden Kontrollflusskanten ein Token in ihrem zugehörigen OCS erzeugt, und somit der Kontrollfluss auf einem der alternativen Kontrollflusspfade exklusiv weitergeleitet.

Zu Beginn der Ausführung einer PGM-Repräsentation liegt initial genau ein Token in der Source-Markierung, und es existieren keine weiteren Tokens in der PGM-Repräsentation. Am Ende der Ausführung einer PGM-Repräsentation ist dagegen genau ein Token in der Sink-Markierung vorhanden. Dies kann aus folgenden Gründen für die definierte Menge von PGM-Aktivitäten und -Kontrollflussmuster garantiert werden:

Alle PGM-Aktivitäten mit einem einfachen ICS bzw. OCS (*SQL, Start, End, Invoke, Assign, Generic*) konsumieren genau ein Token und geben exakt ein Token weiter.

Alternative Kontrollflussmuster können in PGM ausschließlich über *XOR-Split*/*XOR-Join-* bzw. *Set-Iterator*/*XOR-Join*-Aktivitätspaare modelliert werden (siehe Abbildung 4.5 (b) und 4.6). Die Aktivitäten *XOR-Split* bzw. *Set-Iterator* konsumieren und erzeugen jeweils ein Token, das auf genau einem von mehreren verfügbaren Kontrollflusspfaden weitergeleitet wird. Zum Verschmelzen der alternativen Kontrollflusspfade wird eine *XOR-Join*-Aktivität, die ein Token auf einem dieser Kontrollflusspfade entgegennimmt und ein einzelnes Token weiterleitet, benötigt. Das gleiche Argument gilt für Schleifenkonstrukte, die nur aus den obigen Aktivitätspaaren gebildet werden können (siehe Abbildung 4.6).

Parallele Kontrollflussmuster lassen sich in PGM nur über ein *AND-Split*/*AND-Join*-Aktivitätspaar ausdrücken (siehe Abbildung 4.5 (c)). Die *AND-Split*-Aktivität konsumiert ein einzelnes Token und erzeugt jeweils für jede ihrer ausgehenden Kontrollflusskanten ein neues Token. Die Tokens werden anschließend mit einer *AND-Join*-Aktivität, die ein einzelnes Token an die nachfolgende Aktivität weiterleitet, eingesammelt. Damit ist sichergestellt, dass jedes von einer *AND-Split*-Aktivität erzeugte Token von einer nachfolgenden *AND-Join*-Aktivität wieder eliminiert wird.

4.4. Eigenschaften von PGM

Dieser Abschnitt fasst die wesentlichen Eigenschaften des Prozessgraphenmodells zusammen. Aus der Diskussion geht hervor, dass PGM allen Anforderungen an eine interne Repräsentation für die Optimierung datenintensiver Workflows, wie sie in Teilkapitel 4.1 beschrieben wurden, entspricht.

PGM abstrahiert von der Syntax eines Workflowmodells und ist somit in der Lage, mit der Vielfalt prozedural modellierter, kurzlaufender Workflowbeschreibungen einschließlich der Beschreibung SQL-basierter Stored-Procedures umzugehen. Die in Teilkapitel 4.3.4 vorgestellten Aktivitätstypen sind so allgemein definiert, dass damit solche Beschreibungen sprachunabhängig repräsentiert werden können. Auf diese Weise ist eine einheitliche Darstellung von Beschreibungen sowohl von Workflowmodellen als auch von Stored-Procedures in PGM möglich. Dies lässt eine Optimierung der Datenverarbeitung über Sprachgrenzen hinweg zu. Die Kontrollfluss-Aktivitäten von PGM sind so gewählt, dass alle relevanten prozeduralen Basiskontrollflussmuster repräsentiert werden können. Mit SQL-Aktivitäten können beliebige SQL-Anweisungen und benutzerdefinierte Prozeduren ausgeführt werden. Ebenso werden Web-Service-Aufrufe und Variablenzuweisungen unterstützt. Außerdem wird in PGM für jeden der Aktivitätstypen nur die für die Optimierung tatsächlich relevante Information erfasst, wodurch eine kompakte PGM-Repräsentation entsteht.

In PGM können Aktivitäten mit Slot-Typen für Daten, Partner und den Kontrollfluss verknüpft sein. Kanten, welche diese Slots verbinden, stellen die jeweiligen Abhängigkeiten dar. Auf diese Weise können alle für die Optimierung erforderlichen Abhängigkeiten in PGM explizit repräsentiert werden. Hierdurch wird eine direkte und einfache Überprüfung aller relevanter Abhängigkeiten zwischen den Aktivitäten ermöglicht. Insbesondere führen diese expliziten Abhängigkeiten zu einer exakten und kompakten Formulierung der einzelnen Restrukturierungsregeln.

Die generischen Aktivitäts-, Partner- und Datentypen sind wesentliche Merkmale des Prozessgraphenmodells. Sie schaffen die Voraussetzung dafür, Workflowmodelle kompakt zu repräsentieren und PGM zu erweitern.

Der generische Typ ermöglicht es, die für die Optimierung nicht relevanten Teile eines Workflowmodells in PGM auszublenden. Die genaue Beschreibung der Teile wird verborgen, weil die Details für die Optimierung der Datenverarbeitung nicht von Bedeutung sind. Dadurch definiert PGM eine spezielle Sicht auf ein Workflowmodell, welche die für die Optimierung der darin definierten Datenverarbeitung notwendigen Informationen hervorhebt, bzw. umgekehrt die für die Optimierung nicht relevanten Informationen verbirgt.

Durch dieses Abstraktionsprinzip wird eine PGM-Repräsentation erheblich vereinfacht. Dadurch wird die Komplexität der Abbildung eines Workflowmodells auf eine PGM-Repräsentation reduziert. Ebenso führt eine solche Vereinfachung zu einer effizienteren Definition, Implementierung und Anwendung der Restrukturierungsregeln.

Des Weiteren sind die generischen Aktivitäts-, Partner- und Datentypen für die Erweiterbarkeit von PGM von Bedeutung. Diese Typen verbergen alle Detailinformationen eines Workflowmodells, die für die Optimierung nicht relevant sind. Durch Definition neuer, geeigneter Aktivitäts-, Partner- und Datentypen können diese Detailinformationen später bei Bedarf in PGM sichtbar gemacht werden. Damit können weitere Restrukturierungsregeln, welche die Detailinformationen benötigen, unterstützt werden. Es ist zu beachten, dass alle für die zusätzlichen Restrukturierungsregeln erforderlichen Informationen in PGM mittels dieser neuen Typen explizit darzustellen sind. Außerdem müssen die Transformationsregeln zur Abbildung eines Workflowmodells nach PGM und umgekehrt angepasst werden. Auf diese Weise können der Regelbasis neue Restrukturierungsregeln einfach hinzugefügt werden.

Auf diese Weise kann die Optimierung einer PGM-Repräsentation erweitert werden, um beispielsweise neben in SQL formulierten Anfragen XQuery-Ausdrücke [W3Cc] zu berücksichtigen. Um in diesem Fall eine neue Restrukturierungsregel, die eine Optimierung von XQuery-Anweisungen erlaubt, auf PGM definieren zu können, sind die folgenden Schritte notwendig: Zunächst ist ein neuer Partnertyp vom generischen Partnertyp abzuleiten, der eine Datenquelle repräsentiert, welche die Verwaltung von XML-Daten sowie die Abfrage dieser Daten mit einer deklarativen Anfragesprache, wie XQuery, unterstützt. Danach ist die Ableitung eines *XQuery*-Aktivitätstypen, der die Ausführung einer XQuery-Anweisung auf einer entsprechenden Datenquelle repräsentiert, von der *Generic*-Aktivität erforderlich. Bei Bedarf können noch entsprechende Datentypen, die bei einer XQuery-spezifischen Restrukturierungsregel berücksichtigt werden müssen, vom `GENERIC`-Datentyp abgeleitet werden. Wird ein Workflowmodell, das XQuery-Anweisungen einbettet, auf PGM abgebildet, können die neuen XQuery-spezifischen Aktivitäts-, Partner- und Datentypen genutzt werden, um die für eine XQuery-spezifische Restrukturierungsregel notwendigen Teile des Workflowmodells hervorzuheben.

Für die Definition neuer Restrukturierungsregeln kann es notwendig sein, neue prozedurale Kontrollflussmuster, die in PGM bislang nicht berücksichtigt worden sind, zu definieren. Hierfür sind in PGM geeignete Aktivitäten abzuleiten, um eine abstrakte und sprachneutrale Definition neuer Kontrollflussmuster zu ermöglichen.

Auf der PGM-Ebene werden die in einer SQL-Anweisung referenzierten Tabellen und Parameter durch entsprechende PGM-Variablen repräsentiert bzw. durch entsprechende Dataslots einer SQL-Aktivität „nach außen“ sichtbar gemacht. Erst durch die explizite Repräsentation ist eine effiziente Berechnung und Darstellung der Datenabhängigkeiten zwischen den Aktivitäten eines Workflowmodells und somit zwischen den zu optimierenden SQL-Anweisungen möglich.

Schließlich können alle wichtigen Operationen des PGM-Optimierers auf dem Prozessgraphenmodell effizient ausgeführt werden. Die Graphenstruktur kombiniert mit der expliziten Repräsentation wichtiger Abhängigkeiten erlaubt ein einfaches Überprüfen der Bedingungen, welche für die Nutzung der einzelnen Restrukturierungsregeln gelten müssen. Ebenso kann ein Workflowmodell auf das Prozessgraphenmodell einfach abgebildet werden: Die für die Optimierung relevanten Aktivitäten im Workflowmodell werden unmittelbar in eine passende PGM-Aktivität überführt, während nicht relevante Aktivitäten im

Workflowmodell auf eine *Generic*-Aktivität in PGM abgebildet werden. Schließlich werden Kontrollflussmuster des Workflowmodells durch entsprechende Kontrollflussmuster in PGM dargestellt. Dieses Abbildungsverfahren gewährleistet, dass für ein Workflowmodell ein vollständiges und konsistentes Prozessgraphenmodell erstellt werden kann.

4.5. Zusammenfassung

PGM bildet die Basis für die regelbasierte Optimierung datenintensiver Workflows. Diese generische Repräsentation erlaubt eine abstrakte und sprachneutrale Darstellung prozedural modellierter Workflowmodelle. Neben der Repräsentation aller relevanten Basiskontrollflussmuster ist die Darstellung verschiedenster Datenverarbeitungsoperationen, Variablenzuweisungen und Web-Service-Aufrufe möglich.

Zudem erlaubt PGM eine sprachübergreifende Optimierung von Datenverarbeitungsstrukturen, die in unterschiedlichen Beschreibungssprachen definiert sein können. So ermöglicht eine integrierte PGM-Repräsentation eine kombinierte Optimierung eines BPEL/SQL-Workflows zusammen mit den darin aufgerufenen Stored-Procedures.

Des Weiteren stellt PGM alle für die Definition und Anwendung der Restrukturierungsregeln notwendigen Informationen explizit zur Verfügung. Die Informationen umfassen neben der Ausführungslogik eines Workflowmodells die darin definierten Variablen und die Partner, die mit einem Workflow interagieren. Zusätzlich werden Kontrollfluss- und Datenabhängigkeiten zwischen Aktivitäten sowie Kommunikationsabhängigkeiten zwischen Aktivitäten und ihren Partnern explizit durch entsprechende Abhängigkeitskanten dargestellt. Eine solche explizite Repräsentation ist eine wesentliche Voraussetzung für eine effiziente Definition, Implementierung und Anwendung der Restrukturierungsregeln.

Schließlich kann PGM leicht an geänderte Anforderungen, die beispielsweise durch eine Erweiterung der Regelbasis erforderlich werden, angepasst werden. Hierfür stehen generische Typen zur Verfügung, aus denen bei Bedarf neue Aktivitäts-, Partner- und Datentypen abgeleitet werden können. Mit diesen generischen Typen können die für eine Verwendung neuer Restrukturierungsregeln notwendigen Informationen in einer PGM-Repräsentation explizit dargestellt werden.

Somit erfüllt PGM alle Eigenschaften, die für eine effiziente Definition und Anwendung von Restrukturierungsregeln auf einen datenintensiven Workflow notwendig sind.

Für PGM wurde im Rahmen dieser Arbeit neben einer formalen Definition auch eine visuelle Repräsentation entwickelt. Ferner existiert für PGM eine XML-basierte Syntax und eine auf Petri-Netzen basierende Semantikbeschreibung.

5

Regelbasis von PGM/F

In diesem Kapitel wird die heuristische Regelbasis von PGM/F und die ihr zu Grunde gelegten Konzepte im Detail beschrieben. Entsprechend dem Optimierungsmodell von PGM/F werden die Regeln auf PGM-Repräsentationen definiert und angewendet. Bei den Restrukturierungsregeln handelt es sich sowohl um neue als auch um bereits existierende Ansätze zur Optimierung von Datenverarbeitungsoperationen, die auf den Kontext datenintensiver Workflows übertragen werden können.

Auf eine formale Definition der Restrukturierungsregeln wird in diesem Kapitel zugunsten der Übersichtlichkeit verzichtet. Stattdessen werden die Restrukturierungsregeln mithilfe von Datenverarbeitungsmustern, die jeweils den relevanten Ausschnitt einer PGM-Repräsentation darstellen, veranschaulicht. Die Datenverarbeitungsmuster erfüllen alle Voraussetzungen für die Anwendung einer Restrukturierungsregel. Damit lassen sich die zentralen Bedingungen für eine Regelanwendung erklären. Den Datenverarbeitungsmustern ist das in Abbildung E.2 im Anhang gezeigte Datenbankschema zu Grunde gelegt. Restrukturierungsregeln, die für die Optimierung des Beispiel-Workflows aus Abbildung 2.2 relevant sind, werden dagegen mithilfe einer PGM-Repräsentation des Beispiel-Workflows, die in Anhang E zu finden ist, veranschaulicht - In Anhang E werden die PGM-Repräsentationen im Einzelnen dargestellt, die während der Optimierung des Beispiel-Workflows vom PGM/F-System generiert werden. Die formalen Definitionen der wichtigsten heuristischen Restrukturierungsregeln befinden sich in Anhang D, auf die an der entsprechenden Stelle einer Regelbeschreibung verwiesen wird. Die verwendete Syntax zur Definition der Restrukturierungsregeln wird auf den SQL-Standard begrenzt. Dadurch wird vermieden, dass für jedes Datenbanksystem alle Spezifika der Syntax bei der Definition von Restrukturierungsregeln berücksichtigt und aktuell gehalten werden müssen. Allerdings können die Leistungssteigerungen unabhängig von einem zu Grunde gelegten Datenbanksystem erzielt werden (siehe auch Kapitel 7).

Die Optimierungseffekte, die durch Anwendung einer Restrukturierungsregel zu erwarten sind, werden bei deren Beschreibung ebenfalls behandelt. Dabei können Optimierungseffekte auf der Workflow- und der Datenebene erzielt werden. Auch die Reduzierung des Datenvolumens, das zwischen Workflow- und Datenebene ausgetauscht werden muss, spielt hierbei eine entscheidende Rolle. Im Mittelpunkt steht unter anderem die Frage, welchen Einfluss die Organisation der Datenebene auf die Optimierungseffekte hat. Dabei werden sowohl zentralisierte und föderierte als auch mehrere unabhängig voneinander operierende Datenbanksysteme betrachtet.

Neben den heuristischen Restrukturierungsregeln, welche den Schwerpunkt dieser Arbeit bilden, werden auch Optimierungsstrategien vorgestellt, die nur im Rahmen einer kostenbasierten PGM-Optimierung angewendet werden können. Dadurch soll die Diskussion über geeignete Optimierungstechniken für datenintensive Workflows abgerundet, und das weitere Entwicklungspotential des PGM/F-Systems aufgezeigt werden.

Das vorliegende Kapitel ist wie folgt gegliedert: Zunächst werden in Teilkapitel 5.1 wichtige Eigenschaften der Regelbasis von PGM/F erläutert. Danach stellt Teilkapitel 5.2 eine Klassifikation der Regelbasis vor und gibt einen Überblick über die betrachteten Optimierungsstrategien. Anschließend werden in den Teilkapiteln 5.3 bis 5.5 die wichtigsten heuristischen Restrukturierungsregeln von PGM/F, die im Fokus dieser Arbeit stehen, im Detail beschrieben. Einen Überblick über weitere heuristische Optimierungsstrategien gibt Teilkapitel 5.6. In Teilkapitel 5.7 werden dann Optimierungsstrategien, die nur im Rahmen einer kostenbasierten PGM-Optimierung angewendet werden können, diskutiert. Abschließend fasst Teilkapitel 5.8 die wichtigsten Ergebnisse zusammen.

5.1. Eigenschaften

Die Regelbasis von PGM/F besitzt folgende charakteristische Eigenschaften:

- In PGM/F lassen sich Regeln beliebiger Komplexität definieren. Dies wird durch die Struktur einer Regel, die einen *Bedingungs-* und *Aktionsteil* enthält, sichergestellt. Der Bedingungsteil besteht aus Prädikaten, die einen bestimmten Zustand in einem PGM-Graphen beschreiben, und damit festlegen, in welchen Situationen eine Regel anwendbar ist. Der Aktionsteil enthält die Vorschrift, wie der von einer Regel betroffene Teil eines PGM-Graphen umzustrukturieren ist. Die Vorschrift besteht aus einzelnen Transformationsschritten, die auf einem PGM-Graphen definiert sind.
- Durch die Gruppierung der Regeln in Klassen lässt sich eine Strukturierung und Modularisierung der gesamten Regelbasis erzielen. Insgesamt kann damit die Komplexität verringert und die Effizienz der Regelverarbeitung verbessert werden. Ebenso kann eine einfache *Erweiterbarkeit* der Regelbasis gewährleistet werden.
- Zudem überführt die Regelmenge von PGM einen korrekten PGM-Graphen in einen äquivalenten, korrekten PGM-Graphen. Allerdings kann während eines Optimierungslaufes auch eine inkonsistente PGM-Repräsentation, die aber immer nur einen Zwischenzustand eines Optimierungslaufes reflektiert, entstehen. Die *Korrektheit* einer PGM-Repräsentation am Ende eines Optimierungslaufes ist durch eine ent-

sprechende Aufrufreihenfolge der Restrukturierungsregeln und durch die Anwendung entsprechender konsistenzerhaltender Regeln immer gewährleistet. Darauf aufbauend kann die Korrektheit der gesamten Regelmenge hinsichtlich ihrer PGM-Graphentransformationen gefolgert werden.

- Da alle Restrukturierungsregeln die Ausführungskosten eines datenintensiven Workflows reduzieren, führt keine Anwendung einer Restrukturierungsregel zu einer *Leistungsverschlechterung*.

5.2. Klassifikation

Abbildung 5.1 zeigt eine Klassifikation der in diesem Kapitel diskutierten Restrukturierungsregeln. In der Abbildung werden Regelklassen als Rechtecke dargestellt, ihre Instanzen sind im Klartext gegeben. Die Regelklassen lassen sich in heuristische und kostenbasierte Regeln weiter unterteilen.

- Ziel der *Web-Service-Pushdown*-Regelklasse (siehe Teilkapitel 5.3) ist die Verlagerung eines Web-Service-Aufrufs von der Workflow- auf die Datenebene, indem ein Web-Service-Aufruf direkt in eine SQL-Anfrage eingebettet wird. Damit verbunden ist eine Wandlung einer *Invoke*- in eine *SQL-Query*-Aktivität. Dadurch können SQL-spezifische Regeln auf Web-Service-Operationen, die ursprünglich nicht in SQL definiert wurden, angewendet werden, sodass das Optimierungspotential einer PGM-Repräsentation erheblich gesteigert werden kann. Die Regeln sind nur anwendbar, wenn ein zu Grunde liegendes Datenbanksystem Web-Service-Aufrufe unterstützt, z.B. in Form benutzerdefinierter Funktionen.
- Die *Activity-Merging*-Regelklasse zielt auf die Verschmelzung datenabhängiger Aktivitäten ab. Eine Datenabhängigkeit existiert, wenn beispielsweise eine Quellaktivität Daten, die anschließend von einer Zielaktivität gelesen werden, produziert. Die Datenabhängigkeiten lassen sich auflösen, indem die Operation der Quellaktivität mit der Operation der Zielaktivität verschmolzen wird. Bei der Zielaktivität handelt es sich immer um eine SQL-Aktivität, wohingegen die Quellaktivität eine *Assign*- oder eine SQL-Aktivität sein kann. Je nach Typ der Quellaktivität ergeben sich unterschiedliche Regeln, die im Folgenden näher beschrieben werden.

  Die Regeln *Assign-Merging* und *Eliminate-Temporary-Table* gehören zur Gruppe der heuristischen Restrukturierungsregeln. Durch die *Assign-Merging*-Regel wird eine *Assign*- mit einer SQL-Aktivität verschmolzen. Voraussetzung ist, dass die *Assign*-Aktivität einer Variablen einen neuen Wert zuweist, der in der folgenden SQL-Aktivität als Parameter eingelesen wird. In diesem Fall kann die durch die *Assign*-Aktivität definierte Wertzuweisung auch direkt in der SQL-Aktivität erfolgen. Hierdurch können neue Datenabhängigkeiten, die zu weiteren Regelanwendungen führen können, entstehen. Die *Eliminate-Temporary-Table*-Regel entfernt dagegen in einer SQL-Anweisung Referenzen auf temporäre Tabellen, indem eine Referenz direkt durch ihre definierende SQL-Anfrage ersetzt wird. Damit können Kosten zur

Verwaltung einer temporären Tabelle auf der Workflow- und Datenebene eingespart werden.

Die *Update-Merging*-Regeln enthalten heuristische und kostenbasierte Regeln. Ziel dieser Regelgruppe ist es, zwei *SQL-DML*-Aktivitäten, die Änderungen auf derselben Tabelle durchführen, zu kombinieren. Je nach Kombination der betrachteten Änderungsoperation ist hierfür jeweils eine eigene Regel definiert. Als ein Beispiel betrachte man zwei aufeinanderfolgende Insert-Anweisungen, die beide dieselbe Datenbanktabelle aktualisieren. Mithilfe der `UNION`-Operation verschmilzt die *Insert-Insert-Merging*-Regel beide Anweisungen. Dies führt zu einer Reduzierung der Anzahl der SQL-Aktivitäten in einem PGM-Graphen. Ferner können hierdurch Intra-Query-Parallelitäten [Rah94] entstehen, die bei einer parallelen bzw. föderierten Architektur eines zu Grunde liegenden Datenbanksystems zu einer beschleunigten Ausführung einer solchen kombinierten Anweisung führen können.

Wie in Abbildung 5.1 zu sehen, gibt es ähnliche heuristische Restrukturierungsregeln für Update- bzw. Delete-Anweisungen. Darüber hinaus existieren andere Kombinationen von Aktualisierungsoperationen, die jedoch für einen heuristischen Optimierungsansatz nicht geeignet sind. Dies liegt daran, dass eine solche Kombination nur mithilfe einer Merge-Anweisung realisiert werden kann, die aufgrund des höheren Berechnungsaufwandes in einigen Fällen zu Laufzeitverschlechterungen führen kann. In Abbildung 5.1 sind zudem nur Kombinationen von Aktualisierungsoperationen aufgelistet, die alle auf dem SQL:2003-Standard (ISO/IEC 9075(1-4,9-11,13,14):2003) basieren. Wegen der Vielzahl von SQL-Dialekten gibt es weitere dialektspezifische *Update-Merging*-Regeln, die in Abbildung 5.1 nicht berücksichtigt worden sind. Beispielsweise bietet der SQL-Dialekt der DB2 von IBM [IBMa] eine Merge-Anweisung, welche die Kombination von Update-, Insert- und Delete-Operationen in einer einzelnen SQL-Anweisung erlaubt. Im Gegensatz dazu definiert der SQL:2003-Standard eine Merge-Anweisung als Kombination von lediglich einer Update- und einer Insert-Operation. Deshalb tauchen in Abbildung 5.1 nicht alle Kombinationen zwischen den Aktualisierungsoperationen auf. Die Kombinationen sind dennoch wichtig für eine kostenbasierte Optimierung DB2-spezifischer SQL-Anweisungen, da sie das Optimierungspotential in einer PGM-Repräsentation wesentlich erhöhen.

Das primäre Ziel der *Query-Merging*-Regeln ist es, das zwischen Daten- und Workflowebene ausgetauschte Datenvolumen durch geeignete Kombination abhängiger SQL-Anfragen zu reduzieren. Die *Predicate-Pushdown*-Regel verschiebt z.B. Selektionsoperationen, die in einer PGM-Repräsentation in *Assign-Select*-Aktivitäten auf materialisierten Tabellen definiert sind, in *SQL-Query*-Aktivitäten. Dort können sie frühzeitig ausgeführt werden, damit die Größe einer an das Workflowsystem zu übertragenden Ergebnismenge reduziert wird. Die *Select-* bzw. *Select-Into-Merging*-Regel löst dagegen Datenabhängigkeiten zwischen SQL-Anfragen auf, indem diese mithilfe einer With-Klausel verschmolzen werden. Dadurch kann auf die Übertragung einer Anfrageergebnismenge an ein Workflowsystem vollständig verzichtet werden, da durch die Verschmelzung der SQL-Anfragen die Parameterübergabe direkt im Hauptspeicher eines Datenbanksystems erfolgen kann.

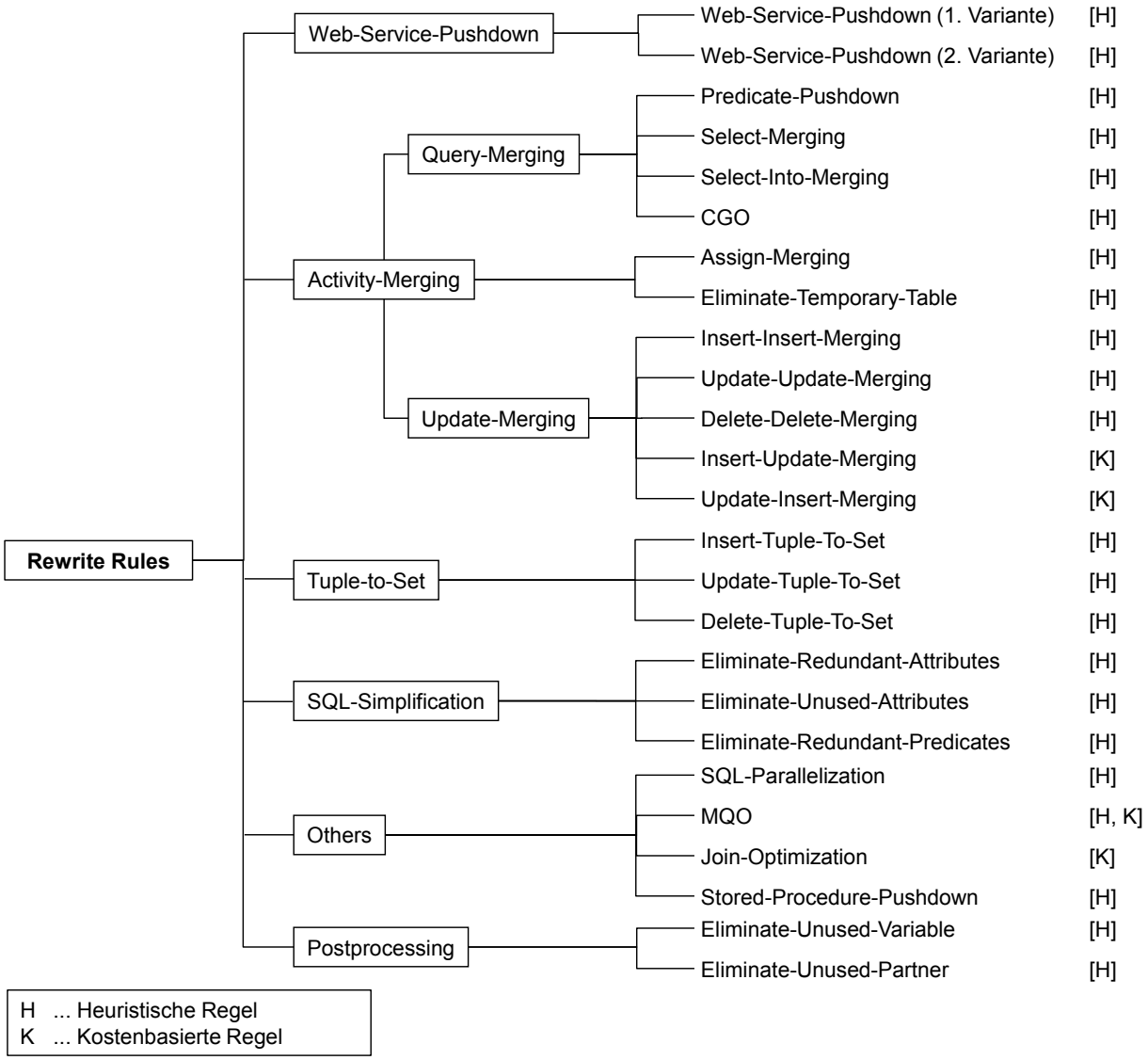


Abbildung 5.1.: Klassifikation der PGM-Regelmenge

In die Regelklasse *Query-Merging* können auch die heuristischen *CGO*-Regeln (siehe Teilkapitel 2.2.3) integriert werden, die eine Sequenz ähnlich strukturierter Insert-Select-Anweisungen verschmelzen. Allerdings ist zu beachten, dass diese Regeln lediglich im Kontext von Anweisungssequenzen definiert wurden und deshalb erst auf PGM übertragen werden müssen, um sie für die Optimierung datenintensiver Workflows nutzbar zu machen.

Durch Anwendung einer Restrukturierungsregel aus der *Activity-Merging*-Regelklasse reduziert sich die Anzahl der SQL-Aktivitäten in einem datenintensiven Workflow. Gleichzeitig verringert sich die Anzahl der Datenbanksystemaufrufe und somit die Datenübertragungskosten bzw. das zwischen Daten- und Workflowebene auszutauschende Datenvolumen. In einigen Fällen kann durch die Kombination von SQL-Anweisungen ein zusätzliches Optimierungspotential auf der Datenebene entstehen, das von einem Anfrageoptimierer zwecks Beschleunigung der Ausführung einer kom-

binierten SQL-Anweisung genutzt werden kann. Ebenso können durch Anwendung einer *Activity-Merging*-Regel redundante Berechnungen bei der Ausführung einer kombinierten SQL-Anweisung vermieden werden. In Teilkapitel 5.4 wird die *Activity-Merging*-Regelklasse im Detail beschrieben.

- Die *Tuple-to-Set*-Regelklasse (siehe Teilkapitel 5.5) adressiert Schleifen, die über einer Datenmenge iterieren, die im Schleifenrumpf tupelorientiert verarbeitet werden. Folglich wird die Änderungsoperation im Schleifenrumpf für jedes einzelne Element der gegebenen Datenmenge ausgeführt. Mit der Anwendung einer der Regeln aus der Klasse *Tuple-to-Set* lässt sich ein solches Datenverarbeitungsmuster direkt in SQL ausdrücken. Dabei wird die gesamte Schleife in einer einzelnen SQL-Anweisung zusammengefasst, wodurch sich sehr große Laufzeitvorteile ergeben können. Insbesondere entfällt hierdurch der Aufwand für wiederholte Ausführungen der Änderungsoperation im Schleifenrumpf.
- Die Regeln der Klasse *SQL-Simplification* (siehe Teilkapitel 5.6.3) erzielen durch kleine Änderungen an der Struktur einer SQL-Anweisung Optimierungsgewinne. Beispielsweise lässt sich durch Veränderungen in einer Select-Klausel einer SQL-Anfrage die Größe einer Ergebnismenge und damit das an ein Workflowsystem zu übertragende Datenvolumen reduzieren. Insbesondere ist nach Anwendung einer solchen Regel sichergestellt, dass keine Daten mehrfach bzw. ausschließlich nur jene Daten an ein Workflowsystem übertragen werden, die auf der Workflowebene tatsächlich weiterverarbeitet werden. Dadurch können insbesondere redundante Datenübertragungen vermieden werden, was zu reduzierten Übertragungskosten und somit zu Laufzeitgewinnen führt.
- Die Regelmenge von PGM/F umfasst weitere heuristische Regeln, deren Anwendung zu einer verkürzten Ausführungszeit eines datenintensiven Workflows führen kann: Beispielsweise wird bei der Parallelisierung von SQL-Anweisungen (siehe Teilkapitel 5.6.1) eine Sequenz datenunabhängiger SQL-Anweisungen in eine Ablauffolge gewandelt, welche die maximal mögliche Parallelität zwischen diesen Datenverarbeitungsoperationen ausnutzt. Abhängig von dem hierbei erzielten Grad der Parallelisierung lässt sich die Laufzeit eines datenintensiven Workflows reduzieren.

  Die MQO-Optimierungstechnik (siehe Teilkapitel 5.7.3) bietet eine weitere Möglichkeit zur Verkürzung der Laufzeit eines datenintensiven Workflows. Diese bewährte Optimierungstechnik kann dazu genutzt werden, Ergebnisse- bzw. Teilergebnisse der in einem datenintensiven Workflow definierten SQL-Anfragen vorab berechnen zu lassen, um die Ausführung der Anfragemenge zu beschleunigen. Allerdings kann die Optimierungstechnik nur dann für einen heuristischen Optimierungsansatz verwendet werden, wenn ein zu Grunde gelegtes Datenbanksystem eine geeignete Unterstützung anbietet. Als Alternative können MQO-Optimierungstechniken auch direkt auf der PGM-Ebene angewendet werden, in diesem Fall allerdings nur im Rahmen eines kostenbasierten Optimierungsansatzes.

  Die Optimierung von Verbundberechnungen weist ebenfalls ein hohes Optimierungspotential auf. In einem datenintensiven Workflow ist es möglich, solche Verbundoperationen sowohl auf der Daten- als auch auf der Workflowebene zu definieren. Zum

einen können Verbundoperationen direkt in SQL-Anfragen deklariert werden; zum anderen können die Operationen auch durch entsprechende Datenverarbeitungsmuster auf der Workflowebene prozedural dargestellt werden. Welche der beiden Varianten effizienter ist, hängt unter anderem von der Größe der an einer Verbundoperation beteiligten Relationen bzw. von dem Selektivitätsfaktor einer Verbundoperation ab. Deshalb kommt hier nur eine kostenbasierte Optimierung in Frage. Das Problem der Verbundoptimierung wird in Teilkapitel 5.7.2 näher beschrieben.

Eine andere Möglichkeit der Leistungssteigerung besteht darin, auf der Workflowebene modellierte Datenverarbeitungsoperationen direkt auf der Datenebene mit Stored-Procedures zu definieren. Hierdurch können die Datenverarbeitungsoperationen zur Laufzeit auf dem zugehörigen Datenbanksystem ohne Involvierung des Workflowsystems direkt ausgeführt werden. Dadurch kann eine effizientere Datenverarbeitung erzielt werden. Die Identifizierung geeigneter Datenverarbeitungsoperationen und deren Transformation in äquivalente Stored-Procedure-Definitionen ist Aufgabe der *Stored-Procedure-Pushdown*-Regelklasse (siehe Teilkapitel 5.6.2).

- Die *Postprocessing*-Regeln stellen die Konsistenz einer PGM-Repräsentation sicher. Beispielsweise kann die Anwendung einiger Restrukturierungsregeln dazu führen, dass Partner bzw. Variablen, die ursprünglich in einer PGM-Repräsentation definiert wurden, nicht mehr benötigt werden. Das Entfernen redundanter Partner- bzw. Variablendefinitionen ist Aufgabe der Regeln *Eliminate-Unused-Partner* bzw. *Eliminate-Unused-Variable*, die einen Optimierungslauf abschließen.

5.3. Die Regelklasse Web-Service-Pushdown

Eine *Web-Service-Pushdown*-Regel verlagert einen Web-Service-Aufruf von der Workflow- auf die Datenebene, indem der Web-Service-Aufruf in eine SQL-Anweisung eingebettet und von einem Datenbanksystem im Rahmen der Verarbeitung der SQL-Anweisung ausgeführt wird. Damit verbunden ist eine Transformation einer *Invoke*- in eine *SQL-Query*-Aktivität. Ein solcher Transformationsschritt kann das Optimierungspotential einer PGM-Repräsentation erheblich erweitern, da auf diese Weise Datenabhängigkeiten zwischen SQL-Anweisungen und Web-Service-Aufrufen aufgelöst werden können. Dadurch kann sich die Anwendung weiterer Restrukturierungsregeln, wie z.B. einer *Query-Merging*- bzw. *Tuple-To-Set*-Regel, ergeben (siehe Teilkapitel 5.4.6 bzw. 5.5). Somit können SQL-spezifische Regeln auf Web-Service-Operationen, die ursprünglich nicht in SQL definiert wurden, in einer PGM-Repräsentation angewendet werden.

In den folgenden Abschnitten wird zunächst diskutiert, welche Möglichkeiten zur Durchführung von Web-Service-Aufrufen in einem Datenbanksystem zur Verfügung stehen. Anschließend wird eine Variante der *Web-Service-Pushdown*-Regel im Detail beschrieben.

5.3.1. Verlagerung von Web-Service-Aufrufen auf die Datenebene

Auf Seiten eines Datenbanksystems existiert kein Standard für den Aufruf von Web-Services. Vielmehr bieten führende Datenbankhersteller, wie IBM, Oracle und Microsoft, ihre eigenen proprietären Lösungen an [IBMc, Micb, Ora06]. Typischerweise werden hierfür benutzerdefinierte Funktionen, sogenannte User-Defined-(Table-)Functions (im Folgenden WS-UDTFs genannt) angeboten, mit denen über SOAP/HTTP Web-Services aufgerufen werden können. Dabei wird eine SOAP-Anfragenachricht aus den Eingabeparametern einer WS-UDTF und aus weiteren Informationen aus der WSDL-Beschreibung des aufzurufenden Web-Services, wie z.B. dem Service-Endpoint oder dem Namen der aufzurufenden Operation, konstruiert und mittels HTTP an den Web-Service übermittelt. Das in einer SOAP-Antwortnachricht gekapselte Anfrageergebnis des Web-Services wird anschließend mit spezifischen Funktionen extrahiert und auf relationale Datenstrukturen, wie Tupel oder Tabellen, die als Ergebnis eines WS-UDTF-Aufrufs geliefert werden, abgebildet.

Die *Web-Service-Pushdown*-Regel kann sowohl auf zentralisierten als auch auf föderierten Datenbanksystemen ausgeführt werden. Einzige Voraussetzung ist die Unterstützung von WS-UDTFs zur Ausführung von Web-Service-Aufrufen. Ob ein Datenbanksystem zur Ausführung einer solchen WS-UDTF in der Lage ist, wird in PGM mithilfe der Eigenschaft `WS-UDTF`, die mit einem Partner vom Typ `RDBMS` assoziiert sein kann, angezeigt (siehe Definition 4 in Anhang A). Diese Eigenschaft lässt sich aus dem Datenbanksystemtypen ableiten, auf dem die WS-UDTF ausgeführt werden soll. Entweder können die Informationen direkt aus der Beschreibung eines Workflowmodells ausgelesen werden, oder sie müssen dem PGM/F-Optimierer explizit durch den Benutzer zur Verfügung gestellt werden. Alternativ können die Informationen auch aus dem Deploymentdeskriptor eines Workflowmodells gewonnen werden. In diesem Fall ist die Anwendung einer *Web-Service-Pushdown*-Regel erst zum Deploymentzeitpunkt eines Workflows möglich. In PGM/F muss für jeden Datenbanksystemtypen mit der Eigenschaft `WS-UDTF` eine entsprechende Transformationskomponente, mit der eine WS-UDTF-Implementierung bei Bedarf erzeugt werden kann, realisiert werden. Des Weiteren ist zu berücksichtigen, dass eine generierte WS-UDTF auf dem entsprechenden Datenbanksystem installiert werden muss, um die Ausführbarkeit aller SQL-Anweisungen nach dem Optimierungslauf garantieren zu können. Hierzu müssen entsprechende Zugriffsrechte für PGM/F bzw. für einen Benutzer auf einem Datenbanksystem vorhanden sein.

Aufgrund der implementierungsspezifischen Einschränkungen einer WS-UDTF muss ein Web-Service zum einen eine Kommunikation über SOAP/HTTP anbieten. Zum anderen müssen seine Ein- und Ausgabeparameter auf Datenstrukturen, wie Tupel oder Tabellen, die von einem relationalen Datenbanksystem unterstützt werden, abgebildet werden können. Ob ein Web-Service diese Voraussetzungen erfüllt, lässt sich mithilfe seiner WSDL-Beschreibung ermitteln. Dort enthalten sind sowohl Informationen über das zu verwendende Transportprotokoll bzw. Nachrichtenaustauschformat als auch die Definitionen der Datenstrukturen der Ein- und Ausgabeparameter der angebotenen Operationen. Die Informationen müssen PGM/F bei der Konstruktion einer PGM-Repräsentation aus einem Workflowmodell vorliegen und entsprechend analysiert werden können. Nur wenn ein Web-Service-Aufruf alle diese Voraussetzungen erfüllt, wird er bei der Transformation

eines Workflowmodells nach PGM auf eine *Invoke*-Aktivität abgebildet. Dadurch wird dem PGM-Optimierer explizit angezeigt, dass eine *Invoke*-Aktivität für die Anwendung der *Web-Service-Pushdown*-Regel geeignet ist. Web-Service-Aufrufe, welche diese Voraussetzungen nicht erfüllen, werden dagegen in PGM mittels *Generic*-Aktivitäten gekapselt, da sie auf die Datenebene nicht korrekt verlagert werden können.

Des Weiteren ist zu beachten, dass durch Anwendung einer *Web-Service-Pushdown*-Regel die Kontrolle über einen Web-Service-Aufruf von einem Workflow- auf ein Datenbanksystem übertragen wird. Dies bedeutet, dass ein solcher Web-Service-Aufruf auf der Workflowebene nicht mehr sichtbar ist. Folglich wird die Ausführung der *Invoke*-Aktivität bzw. der Web-Service-Aufruf auf der Workflowebene nicht mehr protokolliert, was bei einer späteren Analyse des Workflows berücksichtigt werden muss.

Zudem müssen nach der Regelanwendung alle Eingabeparameter des Web-Service-Aufrufs von der Workflow- auf die Datenebene übertragen werden, um dort den Web-Service korrekt aufrufen zu können. Hierdurch entstehen zusätzliche Übertragungskosten. Ebenso wird das Ergebnis eines Web-Service-Aufrufs nicht mehr direkt an das Workflowsystem, sondern zuerst an das Datenbanksystem gesendet, wodurch die Übertragungskosten insgesamt steigen können. Wie groß diese Steigerungen im Einzelnen sind, hängt von den an einen Web-Service zu übertragenden bzw. von ihm zu empfangenden Datenmengen ab.

Wie bereits in 4.3.4 diskutiert, werden in PGM vier unterschiedliche Typen von Web-Service-Aufrufen unterschieden, die durch eine *Invoke*-Aktivität repräsentiert werden und somit für eine *Web-Service-Pushdown*-Regel von Bedeutung sind (siehe Tabelle 4.1).

Bei einem Web-Service, der ausschließlich skalare Werte einliest bzw. als Ergebnis liefert (Typ 1), können die durch die Regelanwendung entstehenden zusätzlichen Übertragungskosten vernachlässigt werden.

Dagegen können bei einem Web-Service-Aufruf vom Typ 2 die zusätzlichen Übertragungskosten für die als Ergebnis gelieferte materialisierte Ergebnismenge nicht unberücksichtigt bleiben. Deshalb sollte ein Aufruf eines solchen Typs nur dann auf die Datenebene verlagert werden, wenn sichergestellt ist, dass die vom Web-Service gelieferte Ergebnismenge auf der Datenebene direkt weiterverarbeitet werden kann, und somit nicht mehr an das Workflowsystem übermittelt werden muss. Dies ist genau dann der Fall, wenn im Anschluss an eine *Web-Service-Pushdown-* eine *Tuple-To-Set*-Regel angewendet werden kann (siehe Teilkapitel 5.5).

Im Falle eines Web-Service-Aufrufes vom Typ 3 fallen die erhöhten Übertragungskosten für die Eingabeparameter (materialisierte Tabellen) ins Gewicht. Hier sollte eine Verlagerung dieses Aufrufes nur dann erfolgen, wenn auf die Übertragung dieser Eingabeparameter verzichtet werden kann. Dies ist dann gegeben, wenn nach einer *Web-Service-Pushdown-* eine *Select-Merging*-Regel (siehe Teilkapitel 5.4.6) angewendet werden kann.

Da ein Web-Service-Aufruf vom Typ 4 die Probleme der Typen 2 und 3 vereinigt, sollte ein solcher Aufruf nur dann auf die Datenebene verlagert werden, wenn anschließend eine *Tuple-To-Set-* und *Select-Merging*-Regel angewendet werden können.

Die Kontrollstrategie von PGM/F garantiert, dass eine Verlagerung eines Web-Service-Aufrufs von der Workflow- auf die Datenebene nur erfolgt, wenn keine zusätzlichen Übertragungskosten, die sich auf die Laufzeit eines datenintensiven Workflows negativ

auswirken können, entstehen. So wird beispielsweise ein Web-Service-Aufruf vom Typ 3 lediglich dann auf die Datenebene verlagert, wenn anschließend die Anwendung einer *Select-Merging*-Regel möglich ist. Dabei wird in einem ersten Schritt die zugehörige *Invoke*-Aktivität durch Anwendung der *Web-Service-Pushdown*-Regel in eine äquivalente *SQL-Query*-Aktivität transformiert. Kann danach auf dieser *SQL-Query*-Aktivität auch eine *Select-Merging*-Regel angewendet werden, sind durch die Verlagerung des Web-Service-Aufrufes auf die Datenebene keine negativen Laufzeiteffekte zu erwarten. Anderenfalls müssen die Effekte der *Web-Service-Pushdown*-Regel wieder rückgängig gemacht werden, d.h. die *SQL-Query*- muss in die ursprüngliche *Invoke*-Aktivität zurück gewandelt werden. Die Kontrollstrategie von PGM/F wird in Kapitel 6 noch ausführlich beschrieben.

Die *Web-Service-Pushdown*-Regel ist in zwei Varianten verfügbar: Die erste Variante definiert die Transformation von Web-Service-Aufrufen, die als Ergebnis ausschließlich skalare Werte liefern (Typ 1 und 3); die zweite Variante betrachtet Web-Service-Aufrufe, die genau eine materialisierte Ergebnismenge liefern (Typ 2 und 4). Im ersten Fall wird der Web-Service-Aufruf in eine Select-Into-Anweisung eingebettet, welche die vom Web-Service gelieferten skalaren Werte direkt in entsprechende `SIMPLE`-Variablen in der Into-Klausel abspeichert. Im zweiten Fall wird der Web-Service-Aufruf in einer Select-Anweisung ausgeführt, welche die Ergebnismenge des Web-Service-Aufrufs in einer entsprechenden `SET`-Variablen zur Verfügung stellt. Die formalen Definitionen beider Regelvarianten findet man in den Anhängen D.3 und D.4. Im Folgenden wird die erste Variante der *Web-Service-Pushdown*-Regel näher betrachtet.

Die Implementierung einer WS-UDTF ist datenbanksystemspezifisch und wird hier nicht weiter vertieft. Es sei an dieser Stelle auf die Dokumentation der entsprechenden Datenbankhersteller verwiesen. Als Beispiel für eine solche Implementierung findet man in Abbildung E.6 die Definition der DB2-spezifischen WS-UDTF *OrderItem*, die bei Anwendung der *Web-Service-Pushdown*-Regel auf den Beispiel-Workflow in Abbildung 2.2 generiert wurde.

5.3.2. Regelbeschreibung

Regelbedingung

Um eine *Web-Service-Pushdown*-Regel korrekt anwenden zu können, müssen in einer PGM-Repräsentation folgende Bedingungen erfüllt sein:

- Für die Anwendung der ersten Variante der *Web-Service-Pushdown*-Regel muss in einer PGM-Repräsentation eine *Invoke*-Aktivität existieren, die eine Web-Service-Operation vom Typ 1 oder 3 aufruft.
- Die *Invoke*-Aktivität besitzt n `Read`-Dataslots, die mit Variablen vom Typ `SIMPLE` oder `SET` assoziiert sind, deren Werte als Eingabeparameter für die aufzurufende Web-Service-Operation dienen.
- Ebenso besitzt die *Invoke*-Aktivität m `Write`-Dataslots, die alle mit `SIMPLE`-Variablen verknüpft sind, in welche die skalaren Ergebnisparameter der aufgerufenen Web-Service-Operation geschrieben werden.

Regelaktion

Sind alle Voraussetzungen erfüllt, können folgende Transformationsschritte in einer PGM-Repräsentation durchgeführt werden:

- Durch Anwendung der *Web-Service-Pushdown*-Regel wird eine *Invoke*-Aktivität durch eine *SQL-Query*-Aktivität ersetzt.
- Dabei bleiben alle ursprünglichen Datenabhängigkeiten zwischen der *Invoke*-Aktivität und anderen Aktivitäten in der PGM-Repräsentation und Kontrollflussabhängigkeiten erhalten.
- Lediglich die Kommunikationsabhängigkeit zum aufgerufenen Web-Service-Partner wird entfernt, da dieser nach der Regelanwendung direkt auf der Datenebene aufgerufen wird. Stattdessen wird die *SQL-Query*-Aktivität mit dem Datenbanksystem verknüpft, auf dem der Web-Service-Aufruf ausgeführt werden soll.
- Die *SQL-Query*-Aktivität definiert eine Select-Into-Anweisung, in deren From-Klausel eine WS-UDTF definiert ist, welche die Web-Service-Operation aufruft. Der Name der WS-UDTF entspricht dem Namen der aufgerufenen Web-Service-Operation. Ferner liest die WS-UDTF die n Variablen als Eingabe, die an die n `Read`-Dataslots der *SQL-Query*-Aktivität gebunden sind.
- Die m skalaren Ausgabeparameter der Web-Service-Operation werden in m `SIMPLE`-Variablen geschrieben, die in der Into-Klausel der Anfrage definiert und mit den m `Write`-Dataslots der *SQL-Query*-Aktivität assoziiert sind.

5.3.3. Anwendungsbeispiel

Die PGM-Repräsentation des Beispiel-Workflows aus Abbildung E.4 erfüllt alle Voraussetzungen für die Anwendung der ersten Variante der *Web-Service-Pushdown*-Regel. Die *Invoke*-Aktivität (a_7) repräsentiert den Aufruf des Web-Services *OrderFromSupplier*, einem Web-Service-Aufruf vom Typ 1. Dabei entsprechen die Lese- und Schreibmengen von a_7 den Ein- und Ausgabeparametern der aufgerufenen Web-Service-Operation *OrderItem*, der die skalaren Werte der Variablen *ItemID* (v_5), *ItemQuantity* (v_6), *SupID* (v_7) und *Price* (v_8) als Eingabe einliest und einen skalaren Wert, der in der Variablen *OrderConfirmation* (v_9) gespeichert wird, als Ergebnis liefert. Da in diesem Beispiel *OrderFromSupplier* über SOAP/HTTP aufgerufen werden kann (siehe WSDL-Beschreibung in Abbildung E.3), sind alle Voraussetzungen erfüllt, den Web-Service-Aufruf auf einen äquivalenten WS-UDTF-Aufruf korrekt abbilden zu können.

In Abbildung E.5 ist die PGM-Repräsentation des Beispiel-Workflows nach Anwendung der *Web-Service-Pushdown*-Regel zu sehen. Die *Invoke*-Aktivität a_7 wurde durch eine *SQL-Query*-Aktivität, die eine Select-Into-Anfrage definiert, ersetzt. Die Select-Into-Anfrage ruft in ihrer From-Klausel den Web-Service *OrderFromSupplier* über eine generierte WS-UDTF auf. Dabei entspricht der Name der WS-UDTF dem Namen der aufgerufenen Web-Service-Operation *OrderItem*. Die Werte der Variablen $v_5 - v_8$ dienen als Eingabeparameter für *OrderItem*. Die WS-UDTF liefert als Ergebnis ein Tupel bestehend aus dem skalaren Ausgabewert des Web-Service-Aufrufs, der in der Variablen v_9 zur Verfügung gestellt wird.

Alle ursprünglichen Kontrollflussabhängigkeiten bleiben erhalten, weil die *SQL-Query*-Aktivität a_7 die *Invoke*-Aktivität in der PGM-Repräsentation ersetzt. Die ursprünglichen Datenabhängigkeiten in der PGM-Repräsentation ändern sich auch nicht, da die *SQL-Query*-Aktivität dieselben Lese- und Schreibmengen wie die *Invoke*-Aktivität besitzt. Jedoch entfällt durch die Regelanwendung die ursprüngliche Kommunikationsbeziehung zwischen der *Invoke*-Aktivität und dem Partner *OrderFromSupplier*.

Da dieser Partner von keiner weiteren Aktivität mehr aufgerufen wird, kann er am Ende des Optimierungslaufes durch Anwendung der Regel *Eliminate-Unused-Partner* aus der PGM-Repräsentation entfernt werden. Dadurch kann die Konsistenz der optimierten PGM-Repräsentation wiederhergestellt werden (siehe Abbildung E.9).

5.3.4. Optimierungseffekt

Die *Web-Service-Pushdown*-Regel führt zu keiner unmittelbaren Laufzeitverbesserung eines datenintensiven Workflows (siehe Kapitel 7), wenn auf der Workflow- und Datenebene identische physische Bedingungen zum Austausch von Nachrichten mit einem Web-Service zur Verfügung stehen. Dies ist nachvollziehbar, da ein Web-Service-Aufruf in einem Workflowsystem ähnlich implementiert ist wie in einem Datenbanksystem.

Die *Web-Service-Pushdown*-Regel ist dennoch sinnvoll, da sie die Anwendung weiterer SQL-spezifischer Restrukturierungsregeln vorbereiten kann. Dies wurde bereits in Teilkapitel 3.2 anhand des Beispiel-Workflows demonstriert, bei dem erst durch die *Web-Service-Pushdown*-Regel die Anwendung der gewinnbringenden *Insert-Tuple-to-Set*-Regel möglich wurde. Zudem stellt die Kontrollstrategie durch eine geeignete Aufrufreihenfolge der Restrukturierungsregeln sicher, dass durch die *Web-Service-Pushdown*-Regel keine zusätzlichen Übertragungskosten entstehen, die sich auf die Laufzeit eines datenintensiven Workflows negativ auswirken können (siehe Kapitel 6).

5.3.5. Korrektheit

In einer PGM-Repräsentation werden nur solche Web-Service-Aufrufe durch eine *Invoke*-Aktivität dargestellt, die auf die Datenebene korrekt verlagert werden können. Dabei handelt es sich um die in Kapitel 4.3.4 vorgestellten Web-Service-Aufrufe der Typen 1 bis 4, die eine Kommunikation über SOAP/HTTP unterstützen. Dagegen werden alle übrigen Web-Service-Aufrufe, die diese Voraussetzungen nicht erfüllen, durch eine *Generic*-Aktivität gekapselt, auf die eine *Web-Service-Pushdown*-Regel nicht anwendbar ist.

Die explizite Repräsentation der Eigenschaft `WS-UDTF` eines Partners vom Typ `RDBMS` garantiert zudem, dass ein *Web-Service-Pushdown* nur dann erfolgt, wenn das zu Grunde gelegte Datenbanksystem die Ausführung von Web-Services unterstützt.

Aufgrund der Ersetzung einer *Invoke*- durch eine *SQL-Query*-Aktivität bleiben die ursprünglichen Kontrollflussabhängigkeiten in einer PGM-Repräsentation erhalten. Dies gilt auch für alle Datenabhängigkeiten, weil eine äquivalente SQL-Anweisung genau die in der ursprünglichen *Invoke*-Aktivität definierten Ein- und Ausgabeparameter besitzt.

5.4. Die Regelklasse Activity-Merging

Ziel der Regelklasse *Activity-Merging* ist es, eine Datenverarbeitungsoperation, die in einer *Assign*- oder SQL-Aktivität definiert ist, mit einer folgenden datenabhängigen SQL-Anweisung zu verschmelzen. Dabei wird die datenabhängige SQL-Anweisung derart transformiert, dass die zuvor getrennt ausgeführten Datenverarbeitungsoperationen zu einer einzelnen Datenverarbeitungsoperation kombiniert werden.

Entsprechend der unterschiedlichen Aktivitäts- bzw. SQL-Anweisungstypen, die miteinander kombiniert werden können, setzt sich die Regelklasse *Activity-Merging* aus verschiedenen Regeln bzw. Regelunterklassen zusammen: Zum einen aus den Regeln *Assign-Merging* und *Eliminate-Temporary-Table* und zum anderen aus den Regelunterklassen *Query-Merging* und *Update-Merging*.

In dem folgenden Abschnitt wird zunächst ausgeführt, welche Voraussetzungen erfüllt sein müssen, um eine *Activity-Merging*-Regel korrekt anwenden zu können. Anschließend werden die Bedingungen, Aktionen und Optimierungseffekte beschrieben, die für alle Regeln dieser Klasse gelten. In den darauf folgenden Abschnitten wird die Diskussion für die einzelnen Regeln bzw. Regelunterklassen fortgeführt.

5.4.1. Verschmelzen von Datenverarbeitungsoperationen

Eine *Activity-Merging*-Regel setzt die Existenz zweier Aktivitäten a_i und a_j in einer PGM-Repräsentation voraus, die jeweils eine Datenverarbeitungsoperation o_i bzw. o_j ausführen. Bei der Datenverarbeitungsoperation o_i kann es sich entweder um die Ausführung einer SQL-Anweisung handeln oder um eine einfache Variablenzuweisung, die durch eine *Assign*-Aktivität repräsentiert wird. Im Falle der Datenverarbeitungsoperation o_j handelt es sich dagegen immer um eine SQL-Anweisung. Zudem muss zwischen den beiden Aktivitäten eine Datenabhängigkeit existieren, auf deren Grundlage die beiden Datenverarbeitungsoperationen verschmolzen werden können. Bei der Verschmelzung wird o_i in die Operation o_j integriert. Die daraus resultierende Datenverarbeitungsoperation o_{j+i} wird von Aktivität a_{j+i} ausgeführt, die aus der Aktivität a_j abgeleitet wird.

Die Organisation der Datenebene spielt für die Anwendbarkeit einer *Activity-Merging*-Regel eine entscheidende Rolle. SQL-Anweisungen können nämlich nur dann verschmolzen werden, wenn beide auf demselben Datenbanksystem ausgeführt werden. Dabei kann es sich sowohl um ein zentralisiertes als auch um ein föderiertes Datenbanksystem handeln. Dagegen ist eine Verschmelzung nicht möglich, wenn beide SQL-Anweisungen auf unterschiedlichen Datenbanksystemen ausgeführt werden.

Das Verschmelzen von o_i und o_j kann weitreichende Auswirkungen auf die Ausführungssemantik von o_i bzw. auf die Datenabhängigkeiten in einer PGM-Repräsentation haben. Deshalb kann eine Verschmelzung nur unter bestimmten Voraussetzungen korrekt durchgeführt werden, die im Folgenden näher beschrieben werden.

Erhalt der ursprünglichen Ausführungssemantik einer Datenverarbeitungsoperation

Beim Verschmelzen der beiden Datenverarbeitungsoperationen o_i und o_j wird o_i aus dem Aktivität a_i umschließenden Kontrollflussmuster m_i entfernt und in das a_j umschließende Kontrollflussmuster m_j eingefügt. Wurde vor dem Verschmelzen die Ausführungssemantik von o_i durch m_i bestimmt, legt nach dem Verschmelzen das Kontrollflussmuster m_j die Ausführungssemantik von o_i fest, die als Teil der kombinierten Datenverarbeitungsoperation o_{j+i} ausgeführt wird. Eine Verschmelzung ist folglich nur dann korrekt, wenn die ursprüngliche Ausführungssemantik von o_i nicht geändert wird. Mit anderen Worten, die Datenverarbeitungsoperation o_i muss im Kontrollflussmuster m_j genauso oft und unter denselben Bedingungen ausgeführt werden wie zuvor in m_i. Dieser Sachverhalt wird in den weiteren Ausführungen wie folgt bezeichnet: Die beiden Kontrollflussmuster m_i und m_j sind zueinander „kompatibel“.

Wird o_i etwa aus einem `SEQ`-Kontrollflussmuster entfernt, kann die Operation in ein anderes `SEQ`- oder `PAR`-Kontrollflussmuster ohne Änderung der ursprünglichen Ausführungssemantik von o_i eingefügt werden. Bei einem `SEQ`-Kontrollflussmuster kann es sich um dasselbe Kontrollflussmuster handeln. Diese beiden Kontrollflussmuster garantieren, dass o_i auch nach dem Verschmelzen genau einmal ausgeführt wird. Umgekehrt kann o_i nur dann in ein `SEQ`-Kontrollflussmuster eingefügt werden, wenn es zuvor in einem `SEQ`- oder `PAR`-Kontrollflussmuster definiert war. Dieselbe Argumentation gilt auch, wenn o_i aus einem `PAR`-Kontrollflussmuster entfernt bzw. in ein `PAR`-Kontrollflussmuster eingefügt wird.

Das Entfernen von o_i aus einem alternativen Kontrollflusspfad eines `ALT`-Kontrollflussmusters ist dagegen nicht möglich, da die Ausführung von o_i von der Ausführung des alternativen Kontrollflusspfades, auf dem Aktivität a_i definiert ist, abhängt. Jedes Eliminieren von o_i aus dem alternativen Kontrollflusspfad zerstört die Abhängigkeit und führt zwangsläufig zu einer geänderten Ausführungssemantik von o_i. Umgekehrt führt das Einfügen von o_i in einen alternativen Kontrollflusspfad zu einer Abhängigkeit der Ausführung von o_i von der Ausführung des alternativen Kontrollflusspfades. Dies ist mit einer Änderung der Ausführungssemantik von o_i verbunden. Eine solche Änderung lässt sich vermeiden, wenn o_i bzw. die zugehörige Aktivität a_i in alle existierenden alternativen Kontrollflusspfade eingefügt wird. In diesem Fall ist garantiert, dass o_i zur Laufzeit weiterhin genau einmal ausgeführt wird, unabhängig davon, welcher der alternativen Kontrollflusspfade tatsächlich ausgeführt wird. Somit ist es möglich, eine Datenverarbeitungsoperation in ein `ALT`-Kontrollflussmuster einzufügen, wenn die zugehörige Aktivität a_i zuvor in einem `SEQ`- bzw. `PAR`-Kontrollflussmuster definiert war. Ebenso bleiben die ursprünglichen Datenflussabhängigkeiten zwischen a_i und den Aktivitäten auf den alternativen Pfaden erhalten. Eine Änderung der Ausführungssemantik von o_i kann in Kauf genommen werden, wenn ausschließlich die Datenverarbeitungsoperation o_j von den Daten, die von o_i geliefert werden, abhängt (Produzent-Konsument-Beziehung). Dann kann sogar eine redundante Ausführung von o_i durch eine Verschmelzung mit o_j verhindert werden, was sich positiv auf die Laufzeit eines datenintensiven Workflows auswirkt.

Ein `LOOP`-Kontrollflussmuster ist dagegen nicht mit den übrigen Kontrollflussmustern kompatibel. Die Ausführung von o_i im Schleifenrumpf eines `LOOP`-Kontrollflussmusters hängt vom Schleifenprädikat ab, das bestimmt, wie oft der Schleifenrumpf und so-

	SEQ	ALT	PAR	LOOP
SEQ	x	x	x	-
ALT	-	-	-	-
PAR	x	x	x	-
LOOP	-	-	-	x

Tabelle 5.1.: Kompatibilitätsmatrix der Kontrollflussmuster

mit o_i ausgeführt wird. Das Entfernen von o_i aus dem Schleifenrumpf löst gleichzeitig die Abhängigkeit von o_i zum Schleifenprädikat auf und führt außerhalb des LOOP-Kontrollflussmusters zu einer Änderung der Ausführungssemantik von o_i. Ebenso ist das Einfügen von o_i in ein LOOP-Kontrollflussmuster nicht ohne Änderung der Ausführungssemantik von o_i möglich, da durch das Einfügen die Ausführung von o_i von dem Schleifenprädikat abhängig gemacht wird. Dadurch wird die Ausführungssemantik von o_i zwangsläufig geändert. Die einzige Möglichkeit zur Entfernung einer Datenverarbeitungsoperation aus einem LOOP-Kontrollflussmuster besteht darin, die Operation in ein anderes LOOP-Kontrollflussmuster mit identischem Schleifenprädikat einzufügen. Dieser Fall wird von den Restrukturierungsregeln nicht weiter berücksichtigt. Da weder das Einfügen einer Datenverarbeitungsoperation in noch deren Entfernen aus einem LOOP-Kontrollflussmuster ohne Änderung ihrer Ausführungssemantik möglich ist, kann eine Optimierung nur innerhalb eines Schleifenrumpfes durchgeführt werden. Deshalb wird ein LOOP-Kontrollflussmuster als eigene Optimierungssphäre in PGM dargestellt, innerhalb derer alle Restrukturierungsregeln lokal angewendet werden (siehe Teilkapitel 4.3.4).

Die Tabelle 5.1 fasst die Ergebnisse dieser Diskussion mithilfe einer Kompatibilitätsmatrix zusammen. Die erste Spalte der Tabelle repräsentiert den Typ des Kontrollflussmusters m_i, in das die Operation o_i vor dem Verschmelzen ursprünglich eingebettet ist. Die obere Zeile stellt den Typ des Kontrollflussmusters m_j dar, in das o_i eingefügt werden muss, damit o_i mit o_j zu o_{j+i} verschmolzen werden kann. Ein Verschmelzen von o_i mit o_j ist somit nur möglich, wenn o_i in einem SEQ- oder PAR-Kontrollflussmuster und o_j in einem SEQ-, PAR- oder ALT-Kontrollflussmuster definiert ist. Für den Fall, dass o_i in einem LOOP-Kontrollflussmuster ausgeführt wird, ist ein Verschmelzen nur dann möglich, wenn o_j in einem LOOP-Kontrollflussmuster mit identischem Schleifenprädikat ausgeführt wird.

Wird eine Datenverarbeitungsoperation aus einem SEQ- bzw. PAR-Kontrollflussmuster entfernt, ist weiter zu berücksichtigen, dass dies unmittelbare Auswirkungen auf die Struktur des betroffenen Kontrollflussmusters haben kann. Entsteht dabei ein leerer sequentieller bzw. ein leerer paralleler Kontrollflusspfad, ist die Korrektheit einer PGM-Repräsentation verletzt. Um dies zu korrigieren, müssen die leeren Kontrollflusspfade aus der PGM-Repräsentation entfernt werden. Entsteht danach ein PAR-Kontrollflussmuster mit lediglich einem Kontrollflusspfad, muss das PAR- in ein SEQ-Kontrollflussmuster zwecks Wiederherstellung der Korrektheit der PGM-Repräsentation gewandelt werden. Deshalb wird unmittelbar nach Anwendung einer *Activity-Merging*-Regel die Konsistenz einer resultierenden PGM-Repräsentation überprüft und im Fehlerfall wiederhergestellt.

Ferner ist beim Verschmelzen zu beachten, dass die Ausführungssemantik eines Kontrollflussmusters selbst von einem anderen Kontrollflussmuster abhängen kann. Dies ist möglich, da Kontrollflussmuster in PGM rekursiv definiert werden können (siehe Anhang A.5). Ist beispielsweise ein `SEQ`-Kontrollflussmuster Teil eines `LOOP`-Kontrollflussmusters, werden das `SEQ`-Kontrollflussmuster und die darin eingebetteten Aktivitäten abhängig vom Schleifenprädikat des `LOOP`-Kontrollflussmusters $n \geq 0$ mal ausgeführt.

Deshalb muss beim Verschmelzen zweier Datenverarbeitungsoperationen o_i und o_j darauf geachtet werden, dass das Kontrollflussmuster m_i nicht nur mit m_j, sondern auch mit allen Kontrollflussmustern kompatibel ist, die m_j unmittelbar umschließen. Da die m_j umschließenden Kontrollflussmuster auf dem Kontrollflusspfad von a_i nach a_j liegen, der in PGM explizit über Kontrollflusskanten definiert wird, können die umschließenden Kontrollflussmuster in einer PGM-Repräsentation leicht bestimmt werden.

Schließlich ist zu berücksichtigen, dass in einem Workflowmodell Ausführungsreihenfolgen zwischen Aktivitäten fest vorgeschrieben sein können. Beispielsweise legen Kommunikationsmuster den Ablauf eines Nachrichtenaustausches zwischen einem Workflow und seinen Partnern genau fest, um eine gewünschte Geschäftsfunktion zu realisieren. Solche Kommunikationsmuster werden in einem Workflowmodell durch eine Menge von Aktivitäten umgesetzt, deren Ausführungsreihenfolgen durch die in den Kommunikationsmustern festgelegten Abläufe bestimmt werden. Durch das Verschmelzen zweier Datenverarbeitungsoperationen können aber diese festgelegten Ausführungsreihenfolgen verloren gehen. Da die notwendigen Informationen über vorgeschriebene Ausführungsreihenfolgen zwischen Aktivitäten im Allgemeinen nicht Teil eines Workflowmodells sind, müssen diese dem PGM/F-System vorab zur Verfügung gestellt werden, um zu verhindern, dass die ursprünglichen Kommunikationsmuster durch das Verschmelzen zweier Datenverarbeitungsoperationen geändert werden. Deshalb kann eine korrekte Anwendung einer *Activity-Merging*-Regel letztendlich nur über eine entsprechende Benutzerinteraktion durchgeführt werden.

Aus der obigen Diskussion lassen sich zwei Bedingungen, die beim Verschmelzen zweier Datenverarbeitungsoperationen o_i und o_j gelten müssen, ableiten, damit die ursprüngliche Ausführungssemantik von o_i erhalten bleibt:

1. Die Kontrollflussmuster m_i und m_j müssen es erlauben, o_i aus m_i zu entfernen bzw. o_i in m_j einzufügen, ohne dass die ursprüngliche Ausführungssemantik von o_i geändert wird. Dies gilt auch für alle Kontrollflussmuster auf dem Kontrollflusspfad zwischen a_i und a_j, in die m_j unmittelbar eingebettet ist.
2. Durch das Verschmelzen von o_i mit o_j dürfen zudem keine Ausführungsreihenfolgen von Aktivitäten geändert werden, die aufgrund der Ausführungssemantik eines Workflowmodells zwingend einzuhalten sind. Zur Erfüllung dieser Bedingung ist eine Benutzerinteraktion unerlässlich, um die hierfür notwendigen Informationen dem PGM-Optimierer zur Verfügung zu stellen.

Erhalt existierender Datenabhängigkeiten

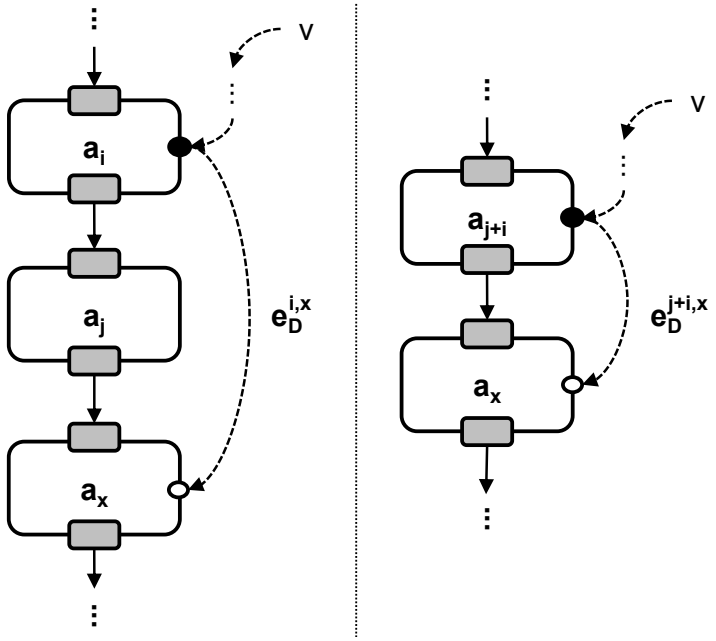


Abbildung 5.2.: Fall $a_i < a_x$ `AND` $a_j < a_x$

Die *Activity-Merging*-Regel setzt eine Datenabhängigkeit zwischen zwei Aktivitäten a_i und a_j voraus, die durch das Verschmelzen der Datenverarbeitungsoperationen o_i und o_j aufgelöst wird. Zu beachten ist, dass sich in der Ausführungslogik durch das Verschieben von o_i weitere Datenabhängigkeiten in einer PGM-Repräsentation ändern können, die zwischen a_i und einer weiteren Aktivität a_x ($\neq a_j$) definiert sein können. Dann muss die Anwendung einer *Activity-Merging*-Regel verhindert werden, um die ursprünglich modellierten Datenabhängigkeiten in einer PGM-Repräsentation zu erhalten.

Werden beispielsweise die beiden Aktivitäten a_i und a_x sequentiell ausgeführt, wobei a_i eine Variable v schreibt, deren Wert anschließend von a_x gelesen wird, liegt eine Schreib-Lese-Datenabhängigkeit zwischen a_i und a_x basierend auf v vor. Wird o_i mit o_j verschmolzen, kann die ursprüngliche Ausführungsreihenfolge zwischen o_i und der von a_x ausgeführten Operation o_x geändert werden. Wurde zuvor o_i vor o_x ausgeführt, ist nach dem Verschmelzen eine vertauschte Reihenfolge von o_i (bzw. o_{j+i}) und o_x möglich. Dies hat zur Folge, dass sich die Referenzierungsreihenfolge auf die Variable v und somit die Art der ursprünglichen Datenabhängigkeit zwischen a_i und a_x ändert. Auf diese Weise wandelt sich die Schreib-Lese- in eine Lese-Schreib-Datenabhängigkeit, wodurch die ursprünglich modellierte Datenabhängigkeit zwischen a_i und a_x verloren geht. Eine ähnliche Problematik liegt für Lese-Schreib- bzw. Schreib-Schreib-Datenabhängigkeiten vor.

Aus diesem Grund muss beim Verschmelzen von o_i mit o_j gewährleistet sein, dass existierende Datenabhängigkeiten zwischen einer Aktivität a_i und einer Aktivität a_x ($\neq a_j$) erhalten bleiben. Hierzu müssen die Kontrollflussabhängigkeiten zwischen den Aktivitäten a_i, a_j und a_x überprüft werden, welche die Referenzierungsreihenfolge auf die gemeinsam referenzierten Variablen und somit die Art der Datenabhängigkeiten bestimmen. Nur

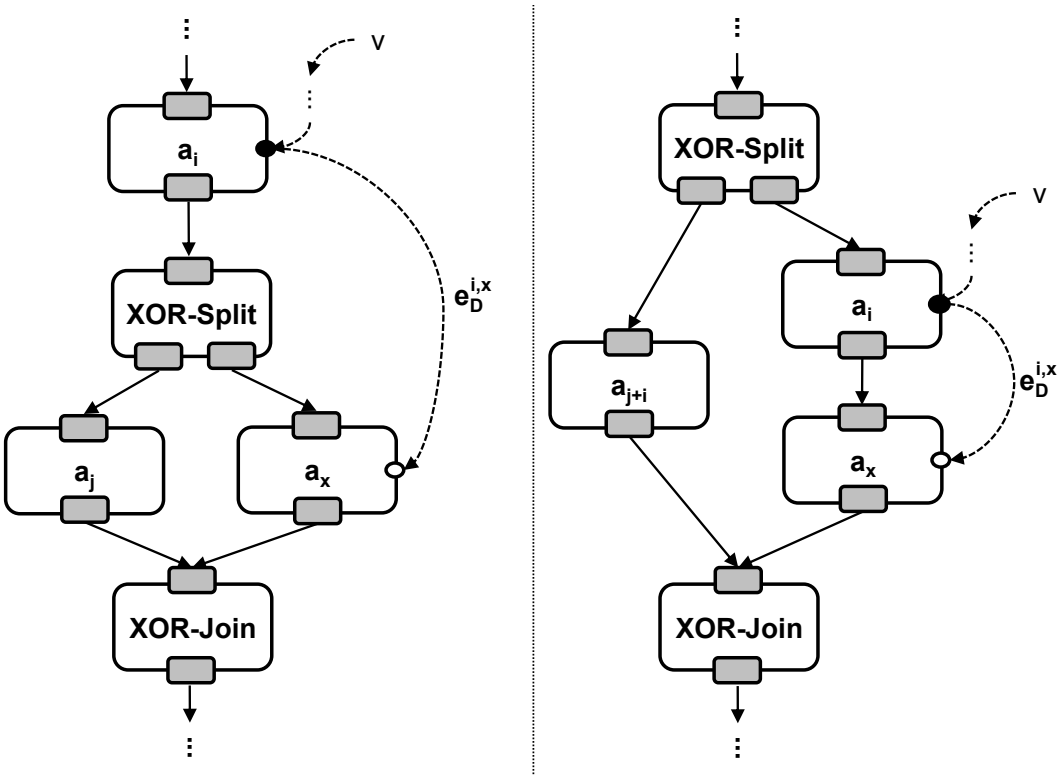


Abbildung 5.3.: Fall $a_i < a_x$ AND $a_x \otimes a_j$

wenn durch diese Kontrollflussabhängigkeiten die ursprüngliche Referenzierungsreihenfolge auf eine Variable nach dem Verschmelzen erhalten bleibt, ist eine Regelanwendung ohne Änderung der ursprünglichen Datenabhängigkeiten gewährleistet.

Hierfür muss folgende Bedingung erfüllt sein: Sei $e_D^{i,x}$ eine Datenabhängigkeitskante zwischen zwei sequentiell ausgeführten Aktivitäten a_i und a_x ($a_i < a_x$), die vor dem Verschmelzen von o_i mit o_j in einer PGM-Repräsentation existiert. Und sei $e_D^{j+i,x}$ die Datenabhängigkeitskante zwischen a_{j+i} und a_x, die sich aus $e_D^{i,x}$ nach dem Verschmelzen von o_i mit o_j ergibt. Dann bleibt nach dem Verschmelzen von o_i und o_j die ursprüngliche Datenabhängigkeit zwischen a_i und a_x genau dann erhalten, wenn a_{j+i} vor a_x ausgeführt wird. Folglich muss Aktivität a_x die Zielaktivität beider Datenabhängigkeitskanten $e_D^{i,x}$ und $e_D^{j+i,x}$ sein.

Beim Verschmelzen von o_i und o_j können folgende Fälle von Kontrollflussabhängigkeiten zwischen a_i, a_j und a_x unterschieden werden:

- Fall 1: $a_i < a_x$ AND $a_j < a_x$ (siehe Abbildung 5.2)
  Werden a_i und a_j vor a_x ausgeführt, bleiben die Kontrollflussabhängigkeiten und somit die ursprünglichen Datenabhängigkeiten zwischen den Aktivitäten nach dem Verschmelzen bestehen. Damit kann o_i mit o_j ohne Änderung der ursprünglichen Datenabhängigkeiten zwischen a_i bzw. a_{j+i} und a_x verschmolzen werden.
- Fall 2: $a_i < a_x$ AND $a_x \otimes a_j$ (siehe Abbildung 5.3)
  Dieser Fall ist für das Verschmelzen von o_i mit o_j ebenso unproblematisch, unter der

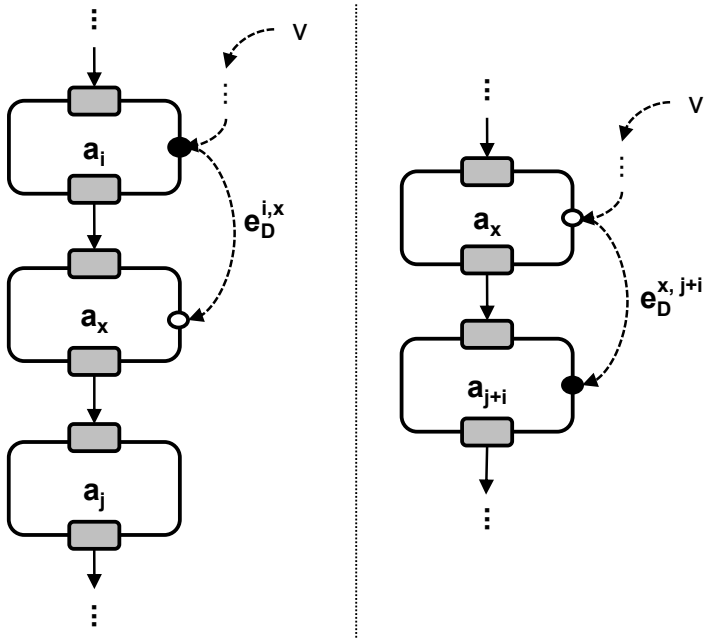


Abbildung 5.4.: Fall $a_i < a_x$ `AND` $a_x < a_j$

Voraussetzung, dass o_i bzw. Aktivität a_i auch in den alternativen Kontrollflusspfad von a_x eingefügt wird (aber nur bei Schreib-Lese-Datenabhängigkeiten notwendig). Hierdurch kann sichergestellt werden, dass nach dem Verschmelzen die ursprüngliche sequentielle Kontrollflussabhänigkeit zwischen a_i und a_x bzw. zwischen a_{j+i} und a_x bestehen bleibt.

- Fall 3: $a_i < a_x$ `AND` $a_x < a_j$ (siehe Abbildung 5.4)
  In diesem Fall kommt ein Verschmelzen von o_i mit o_j ohne Änderung der ursprünglichen Datenabhängigkeit $e_D^{i,x}$ nicht in Betracht, weil sich durch das Verschmelzen die Ausführungsreihenfolge von a_i und a_x und somit die Referenzierungsreihenfolge auf die gemeinsam bearbeitete Variable ändert. Folglich stimmen die Zielaktivitäten der beiden Datenabhängigkeitskanten $e_D^{i,x}$ und $e_D^{j+i,x}$ nicht mehr überein ($a_x \neq a_{j+i}$). Somit ändert sich die Referenzierungsreihenfolge auf die der Datenabhängigkeit zu Grunde gelegten Variable und damit auch die Art der Datenabhängigkeit. Handelt es sich bei $e_D^{i,x}$ noch um eine Schreib-Lese-Datenabhängigkeit zwischen a_i und a_x (erkennbar an dem `Write`- bzw. `Read`-Dataslot von a_i und a_j), liegt im Falle $e_D^{x,j+i}$ eine Lese-Schreib-Datenabhängigkeit zwischen a_x und a_{j+i} vor.
- Fall 4: $a_i < a_x$ `AND` $a_x \mid\mid a_j$ (siehe Abbildung 5.5)
  Auch in diesem Fall ist das Verschmelzen von o_i mit o_j ausgeschlossen. Vor dem Verschmelzen wird a_i vor a_x, danach werden die Aktivitäten a_x und a_{j+i} parallel ausgeführt. Somit greifen beide Aktivitäten nach dem Verschmelzen parallel lesend bzw. schreibend auf dieselbe Variable zu. Dadurch wird die ursprüngliche sequentielle Referenzierungsreihenfolge auf die Variable geändert. Dieses Problem lässt sich an den Datenabhängigkeitskanten $e_D^{i,x}$ und $e_D^{j+i,x}$ veranschaulichen: Ist die Ziel- und Quellaktivität von $e_D^{i,x}$ vor dem Verschmelzen eindeutig bestimmbar, ist das für $e_D^{j+i,x}$ wegen des parallelen Datenzugriffs auf dieselbe Variable nicht mehr möglich.

Als Ergebnis dieser Diskussion kann festgehalten werden, dass eine Verschmelzung von o_i mit o_j nur in den Fällen 1 und 2 zulässig ist. In den Fällen 3 und 4 muss dagegen die Anwendung einer *Activity-Merging*-Regel verhindert werden, damit der Erhalt der ursprünglichen Datenabhängigkeiten in einer PGM-Repräsentation sichergestellt ist.

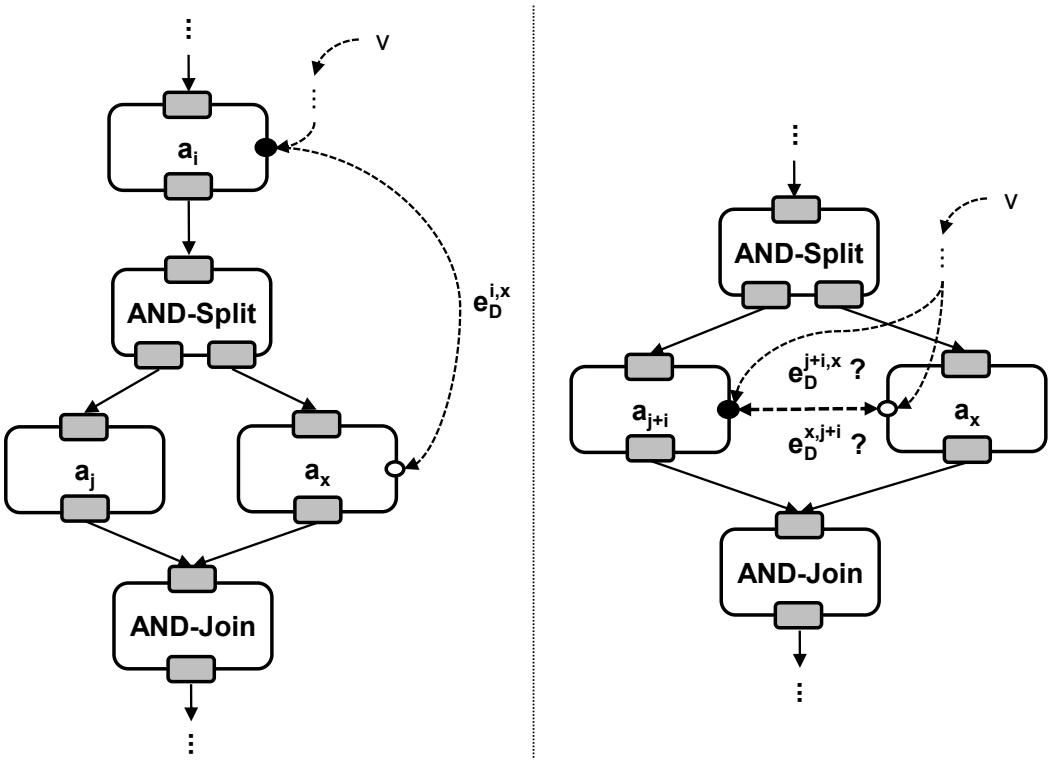


Abbildung 5.5.: Fall $a_i < a_x$ `AND` $a_x \mid\mid a_j$

Exklusive Schreib-Lese-Datenabhängigkeit

Die Existenz einer Datenabhängigkeit $e_D^{i,j}$ zwischen den Aktivitäten a_i und a_j basierend auf einer gemeinsam referenzierten Variablen v ist Voraussetzung für das Verschmelzen der zugehörigen Datenverarbeitungsoperationen o_i und o_j. Dabei verschmelzen einige der *Activity-Merging*-Regeln o_i mit o_j miteinander, indem ein Vorkommen von v in o_j direkt durch die Datenverarbeitungsoperation o_i ersetzt wird. Auf einer solchen Substitutionsstrategie basieren die folgenden Regeln: *Assign-Merging*, *Predicate-Pushdown*, *Select-(Into)-Merging* und *Eliminate-Temporary-Table*.

Aus einem solchen Substitutionsschritt folgt, dass der von o_i zuvor geschriebene Wert in v nicht mehr länger verfügbar ist. Dies stellt ein Problem dar, wenn neben $e_D^{i,j}$ noch eine weitere Schreib-Lese-Datenabhängigkeit $e_D^{i,x}$ basierend auf Variable v zwischen a_i und einer weiteren Aktivität a_x ($\neq a_j$) existiert (siehe Abbildung 5.6). In einem solchen Fall wird durch das Verschmelzen von o_i mit o_j neben $e_D^{i,j}$ gleichzeitig die ursprünglich definierte Datenabhängigkeit $e_D^{i,x}$ aufgelöst.

Um dies zu verhindern, kann o_i nur dann mit o_j verschmolzen werden, wenn sichergestellt ist, dass alle Schreib-Lese-Datenabhängigkeiten von a_i basierend auf der Variablen v ausschließlich zwischen a_i und a_j existieren. In diesem Fall ist a_j die einzige Aktivität, die von dem Wert von v, der von a_i geschrieben wurde, abhängt. Mit anderen Worten: Zwischen a_i und a_j existiert eine *exklusive Schreib-Lese-Datenabhängigkeit* basierend auf Variable v (siehe Definition 41 in Anhang A.6).

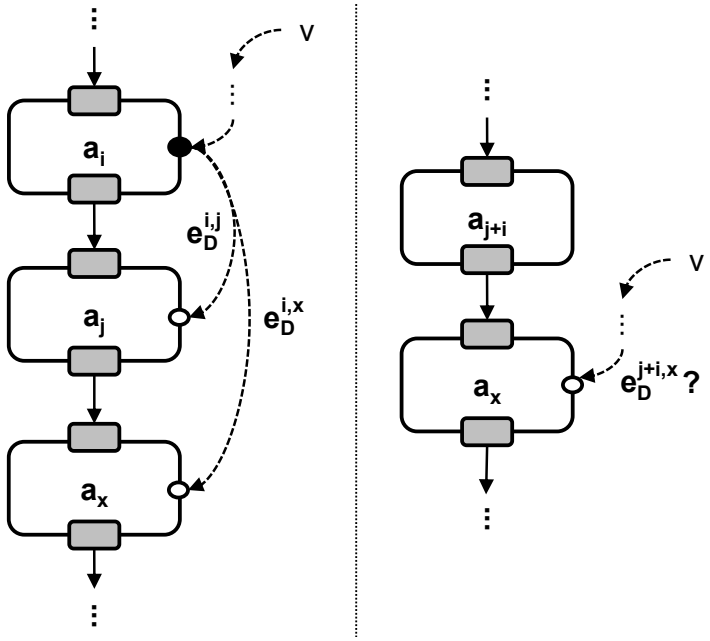


Abbildung 5.6.: Ungewolltes Auflösen von Datenabhängigkeiten

Die Existenz einer solchen exklusiven Schreib-Lese-Datenabhängigkeit stellt sicher, dass durch das Verschmelzen von o_i mit o_j neben der exklusiven Schreib-Lese-Datenabhängigkeit keine weiteren Datenabhängigkeiten zwischen a_i und einer anderen Aktivität a_x ($\neq a_j$) aufgelöst werden können. Dadurch bleibt die Konsistenz einer PGM-Repräsentation nach dem Verschmelzen von o_i mit o_j erhalten.

5.4.2. Klassenspezifische Regelbeschreibung

Regelbedingung

Um eine *Activity-Merging*-Regel korrekt anwenden zu können, müssen in einer PGM-Repräsentation folgende, klassenspezifische Bedingungen erfüllt sein:

- Es müssen zwei Aktivitäten a_i und a_j existieren, die jeweils eine Datenverarbeitungsoperation o_i bzw. o_j ausführen.
- Bei a_i kann es sich entweder um eine *Assign*-Aktivität oder um eine SQL-Aktivität handeln, wohingegen a_j immer eine SQL-Aktivität sein muss.

- Im Falle zweier SQL-Aktivitäten müssen beide mit demselben Datenbankpartner verknüpft sein, um die Ausführbarkeit einer resultierenden, kombinierten SQL-Anweisung sicherzustellen.
- Ebenso darf sich die Ausführungssemantik der Datenverarbeitungsoperation o_i durch das Verschmelzen mit o_j nicht ändern. Dies gilt auch für Datenabhängigkeiten, die zwischen a_i und anderen Aktivitäten existieren können. Die zu überprüfenden Bedingungen wurden bereits in Teilkapitel 5.4.1 diskutiert.
- Ferner müssen neben den klassenspezifischen Bedingungen unter Umständen weitere regelspezifische Bedingungen erfüllt sein, auf welche bei der Diskussion der einzelnen Regeln noch genauer eingegangen wird.

Regelaktion

Sind alle Voraussetzungen erfüllt, können die Datenverarbeitungsoperationen o_i und o_j zu o_{j+i} verschmolzen werden. Dabei werden folgende Transformationsschritte in einer PGM-Repräsentation durchgeführt:

- Durch die Verschmelzung wird die Datenverarbeitungsoperation o_j in o_{j+i} überführt.
- Dabei kann sich durch die Integration von o_i die von o_j ursprünglich gelesene bzw. geschriebene Variablenmenge ändern. Eine Änderung der Lese- und Schreibmengen hat Auswirkungen auf die Dataslots der Aktivität a_j, die durch die Kombination beider Datenverarbeitungsoperationen in die Aktivität a_{j+i} überführt wird. Dataslots sind zu entfernen, wenn die zugehörigen Variablen in o_{j+i} (aufgrund eines Substitutionsschrittes) nicht mehr länger referenziert werden. Umgekehrt sind Dataslots zu erzeugen, wenn Variablen in o_{j+i} durch das Einfügen von o_i neu referenziert werden.
- Eine geänderte Menge an Dataslots wirkt sich auf die Datenabhängigkeiten zwischen den Aktivitäten aus. Deshalb müssen die Datenabhängigkeiten für jeden Dataslot bzw. die mit ihm assoziierte Variable neu berechnet werden.
- Die ursprünglichen Kommunikations- und Kontrollflussabhängigkeiten von a_j bleiben dagegen nach der Restrukturierung von o_j erhalten.
- Anschließend kann a_i aus der Ausführungslogik der PGM-Repräsentation entfernt werden. Dabei müssen folgende Anpassungen bezüglich der Kontrollfluss-, Daten- und Kommunikationsabhängigkeiten vollzogen werden:

  – Um die „Lücke“ in der Ausführungslogik zu schließen, die durch das Entfernen von a_i entsteht, müssen direkter Vorgänger und Nachfolger von a_i verbunden werden. Hierfür muss eine entsprechende Kontrollflusskante in die PGM-Repräsentation eingefügt werden.

  – Durch den Wegfall der Dataslots von a_i müssen zudem alle Datenabhängigkeiten neu berechnet werden, die auf Variablen basieren, die mit diesen Dataslots assoziiert waren.

  - Des Weiteren entfällt im Falle einer SQL-Aktivität die Kommunikationsabhängigkeit zwischen a_i und dem ausführenden Datenbanksystem.

- Variablen, die nach der Regelanwendung weder von a_{j+i} noch von einer anderen Aktivität in der PGM-Repräsentation referenziert werden, werden am Ende des Optimierungslaufes durch die *Eliminate-Unused-Variable*-Regel entfernt. Dadurch wird die Konsistenz einer PGM-Repräsentation wiederhergestellt.

5.4.3. Klassenspezifische Optimierungseffekte

Durch die Verschmelzung von Datenverarbeitungsoperationen kann die Anzahl der auszuführenden Aktivitäten in einer PGM-Repräsentation verringert und können Ausführungskosten eingespart werden. Im Falle einer SQL-Aktivität reduziert sich die Anzahl der Datenbanksystemaufrufe und das zwischen Workflow- und Datenebene auszutauschende Datenvolumen. Ebenso können neue Datenabhängigkeiten in einer PGM-Repräsentation entstehen, wodurch die Anwendung weiterer gewinnbringender Restrukturierungsregeln ermöglicht wird. Schließlich können durch eine *Activity-Merging*-Regel weitere Optimierungseffekte auf der Datenebene erzielt werden. Dies kann eine beschleunigte Ausführung einer kombinierten SQL-Anweisung bzw. eine Reduzierung der Größe von Anfrageergebnismengen zur Folge haben. Da diese Optimierungseffekte regelspezifisch sind, werden sie in den folgenden Teilkapiteln jeweils bei der Diskussion der einzelnen *Activity-Merging*-Regeln näher beschrieben.

5.4.4. Assign-Merging

Die *Assign-Merging*-Regel (siehe Regeldefinition D.5) eliminiert eine Variablenzuweisung, die in einer *Assign*-Aktivität definiert ist, indem die Zuweisung direkt in einer abhängigen SQL-Aktivität durchgeführt wird.

Anwendungsbeispiel Abbildung 5.7 zeigt die Bedingungen zur korrekten Anwendung einer *Assign-Merging*-Regel in einer PGM-Repräsentation: Eine *Assign*-Aktivität a_i führt eine Variablenzuweisung durch, bei welcher der Wert einer Variablen v_i einer anderen Variablen v_j gleichen Datentyps zugewiesen wird. Anschließend wird v_j von einer SQL-Aktivität a_j exklusiv gelesen. In diesem Beispiel handelt es sich um eine *SQL-Query*-Aktivität, welche die SQL-Anfrage S_{Q_j} ausführt, die Bestelldaten aus der Tabelle *ApprovedOrders* ermittelt. Der Preis der Bestelldaten liegt dabei über einem bestimmten Schwellwert, der aus der zuvor von a_i geschriebenen Variablen v_j ausgelesen wird.

Aufgrund der exklusiven Schreib-Lese-Datenabhängigkeit zwischen a_i und a_j basierend auf v_j kann die *Assign-Merging*-Regel korrekt angewendet werden. Dazu wird das Vorkommen der Variablen v_j in S_{Q_j} direkt durch Variable v_i ersetzt. Hierdurch wird die SQL-Anfrage S_{Q_i} in die SQL-Anfrage $S_{Q_{j+i}}$ überführt. Die geänderte Lesemenge von $S_{Q_{j+i}}$ spiegelt sich in der Menge der `Read`-Dataslots von SQL-Aktivität a_{j+i} wider. Der `Read`-Dataslot von a_j, der vor der Regelanwendung an Variable v_j gebunden war, wurde in a_{j+i} durch einen

mit der Variablen v_i verknüpften `Read`-Dataslot ersetzt. Des Weiteren wurde die *Assign*-Aktivität a_i aus der PGM-Repräsentation entfernt. Am Ende des Optimierungslaufes wird Variable v_j durch die *Eliminate-Unused-Variable*-Regel gelöscht, falls sie nicht mehr in der PGM-Repräsentation referenziert wird.

Optimierungseffekte Der Gewinn dieser einfachen Restrukturierungsregel ergibt sich alleine dadurch, dass zum einen auf die Ausführung der *Assign*-Aktivität verzichtet werden kann, und dass zum anderen im weiteren Optimierungsverlauf aufgrund des Substitutionsschrittes und der sich hieraus ergebenden neuen Datenabhängigkeiten weitere Restrukturierungsregeln angewendet werden können. Auf der Datenebene führt die Regelanwendung dagegen zu keiner Laufzeitverbesserung.

Korrektheit Da es sich bei der Regelaktion nur um eine Substitution von Variablennamen handelt, bleibt die Korrektheit der resultierenden PGM-Repräsentation erhalten.

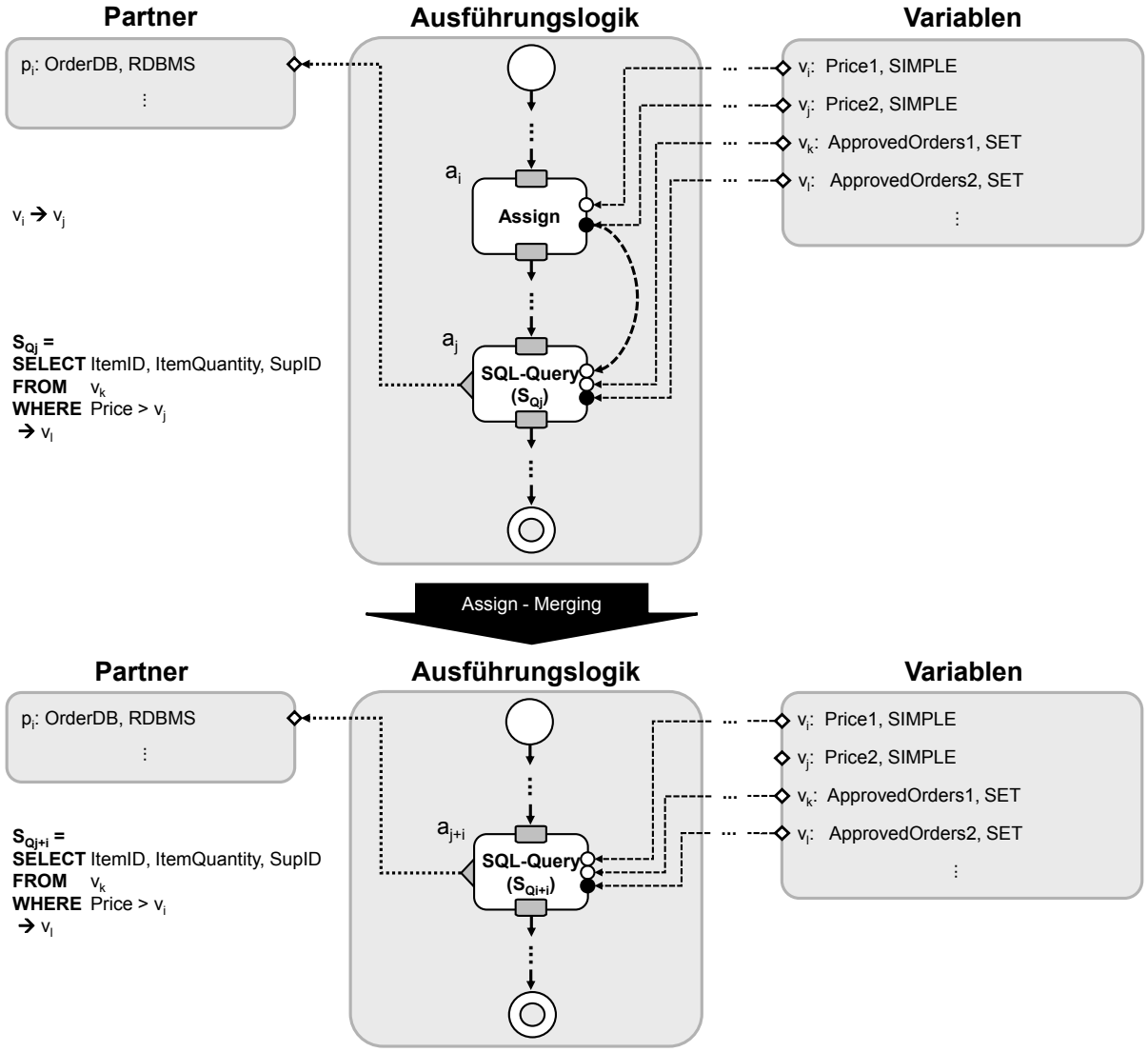


Abbildung 5.7.: Anwendung der *Assign-Merging*-Regel

5.4.5. Update-Merging

Gleichartige Typen von DML-Anweisungen, die auf derselben Tabelle ausgeführt werden, lassen sich durch geeignete Kombinationen zu einer einzelnen Aktualisierungsoperation verschmelzen. Dies kann durch eine entsprechende Anpassung von Klauseln bzw. durch Verwendung geeigneter Mengenoperationen realisiert werden. Dadurch lassen sich positive Optimierungseffekte auf der Datenebene erzielen, die darauf beruhen, dass vor der Verschmelzung wiederholt ausgeführte Berechnungsschritte auf einer zu aktualisierenden Tabelle nach der Verschmelzung vermieden werden können. Ebenso können durch eine solche Kombination Intra-Query-Parallelitäten entstehen, die bei einer geeigneten Unterstützung auf der Datenebene zu einer parallelen Anfrageverarbeitung führen können.

Bei einer solchen Verschmelzung muss sichergestellt sein, dass durch ein kombiniertes, zeitgleiches Einbringen der Aktualisierungsoperationen das gleiche Ergebnis erzielt wird wie durch ein schrittweises, sequentielles Einbringen der Operationen. Deshalb ist eine Kombination von Aktualisierungsoperationen nicht beliebig möglich.

Da in den von PGM/F betrachteten Workflowmodellen alle Datenverarbeitungsoperationen innerhalb derselben Transaktion ausgeführt werden, bleibt die ursprüngliche Transaktionssemantik auch nach Anwendung einer *Update-Merging*-Regel erhalten. Wird eine solche Transaktion in einer Konsistenzstufe 3 nach [GLPT76] ausgeführt, ist das Serialisierbarkeitskriterium für nebenläufige Transaktionen erfüllt. Somit ist sichergestellt, dass der Zustand einer zu aktualisierenden Tabelle nicht durch nebenläufige Transaktionen geändert werden kann. Dadurch unterscheidet sich das Ergebnis eines zeitgleichen Einbringens der Änderungen in eine Tabelle nicht von dem Ergebnis eines schrittweisen Einbringens der Änderung in diese Tabelle. Daraus kann gefolgert werden, dass die Anwendung einer *Update-Merging*-Regel zu denselben Effekten führt, wie sie vor der Regelanwendung erzielt worden sind. Bei einer geringeren Konsistenzstufe kann dies aufgrund möglicher Anomalien im Mehrbenutzerbetrieb dagegen nicht mehr garantiert werden.

Entsprechend der drei unterschiedlichen DML-Anweisungstypen Insert, Delete und Update kann zwischen einer *Insert-Insert-*, *Delete-Delete-* und *Update-Update-Merging*-Regel unterschieden werden. Zudem existieren weitere Regeln zur Kombination von Insert- und Update-Anweisungen mithilfe von Merge-Anweisungen. Allerdings können diese Regeln nur im Rahmen einer kostenbasierten Optimierungsstrategie angewendet werden, weshalb sie erst später in Teilkapitel 5.7.1 betrachtet werden. Im Folgenden werden die heuristischen *Update-Merging*-Regeln vorgestellt.

Insert-Insert-Merging

Die *Insert-Insert-Merging*-Regel (siehe Regeldefinition D.6) kombiniert zwei Insert-Anweisungen, die auf derselben Tabelle ausgeführt werden, mit einer `UNION-ALL`-Operation zu einer einzelnen Aktualisierungsoperation. Dies kann auf der Datenebene zu einer beschleunigten Verarbeitung der kombinierten DML-Anweisung führen, wenn ein zu Grunde gelegtes Datenbanksystem eine entsprechende parallele Anfrageverarbeitung unterstützt.

Anwendungsbeispiel Voraussetzung für die Anwendung der *Insert-Insert-Merging*-Regel ist die Existenz zweier *SQL-DML*-Aktivitäten, wie sie in Abbildung 5.8 gezeigt

werden. In diesem Datenverarbeitungsmuster führen die beiden *SQL-DML*-Aktivitäten a_i und a_j die Insert-Select-Anweisungen S_{DML_i} und S_{DML_j} auf dem Datenbanksystem $OrderDB$ aus. Da die beiden Insert-Anweisungen auf der Tabelle $PremiumCustomer$ (v_i) definiert sind, existiert zwischen a_i und a_j eine Schreib-Schreib-Datenabhängigkeit basierend auf v_i. Dabei bestimmt S_{DML_i} alle Premiumkunden in der Tabelle $Orders$, die einen vorgegebenen Bestellwert überschreiten und speichert diese Datenmenge in der Tabelle $PremiumCustomers$ ab. Anschließend fügt S_{DML_j} ein weiteres Tupel der Tabelle $PremiumCustomers$ hinzu. Dabei handelt es sich bei S_{DML_j} ursprünglich um eine Insert-Values-Anweisung, die bei der Transformation nach PGM durch einen Normalisierungsschritt in eine entsprechende Insert-Select-Anweisung überführt wurde. Ein solcher Normalisierungsschritt ist sinnvoll, da hierdurch die Regeldefinitionen auf Insert-Select-Anweisungen beschränkt werden können.

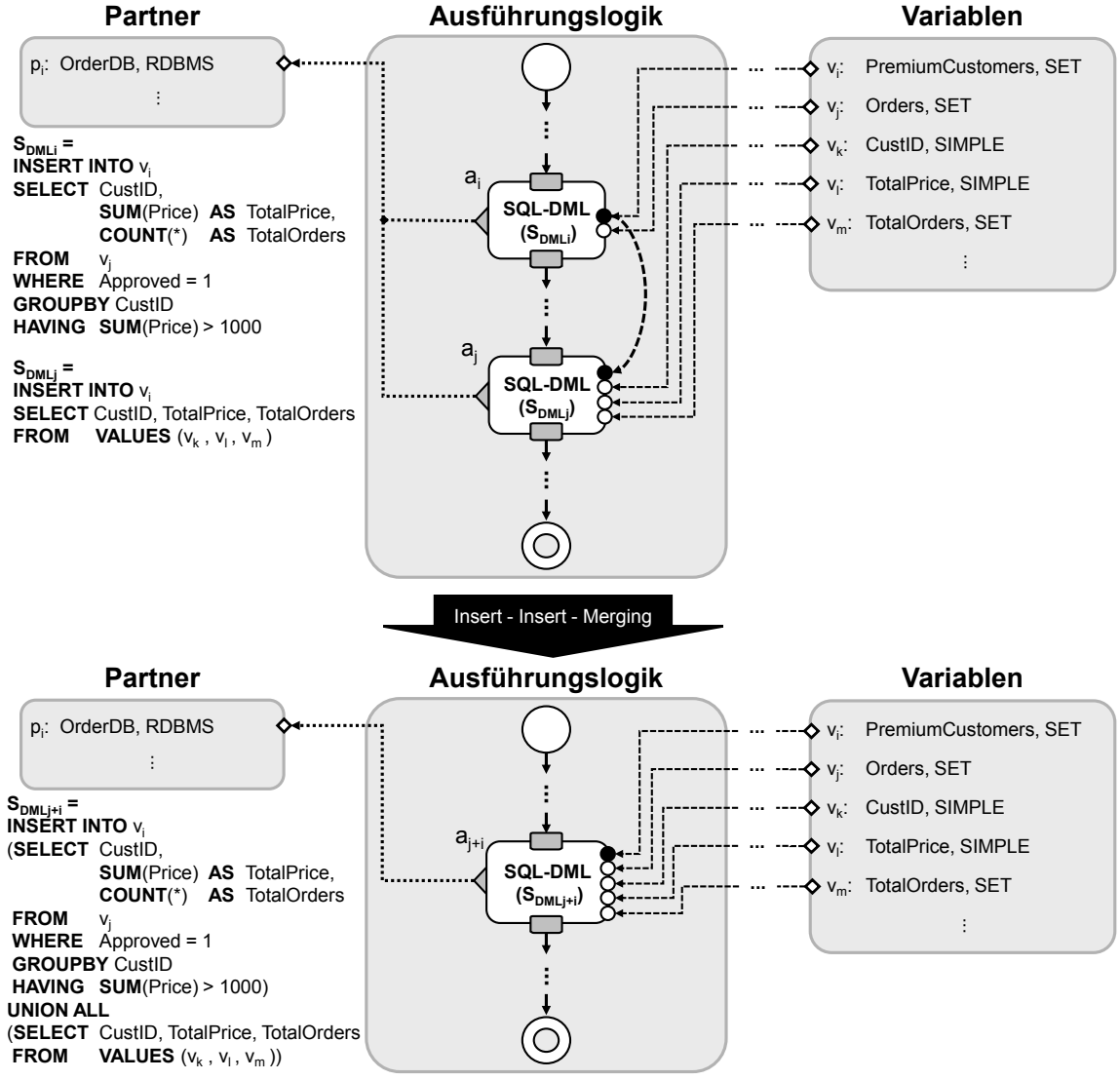


Abbildung 5.8.: Anwendung der *Insert-Insert-Merging*-Regel

Durch die *Insert-Insert-Merging*-Regel lassen sich S_{DML_i} und S_{DML_j} zu $S_{DML_{j+i}}$ kombinieren. In $S_{DML_{j+i}}$ werden die Select-Anfragen von S_{DML_i} und S_{DML_j} mittels einer `UNION-ALL`-Operation verbunden. $S_{DML_{j+i}}$ wird von a_{j+i}, das aus der *SQL-DML*-Aktivität a_j hervorgeht, ausgeführt. Die Menge der Dataslots von a_{j+i} spiegelt die Lese- und Schreibmenge von $S_{DML_{j+i}}$ wider. Neben dem `Write`- und den `Read`-Dataslots, die von a_j direkt übernommen werden können, kommt ein weiterer `Read`-Dataslot hinzu, der mit der `SET`-Variablen v_j, die ursprünglich in S_{DML_j} referenziert worden ist, assoziiert ist. Aufgrund der Verschmelzungsoperation ist die Ausführung von a_i nicht mehr erforderlich, sodass a_i aus der transformierten PGM-Repräsentation entfernt werden kann.

Optimierungseffekte Mit der *Insert-Insert-Merging*-Regel lassen sich auf der Datenebene positive Optimierungseffekte erzielen, indem die beiden einzufügenden Datenmengen nach der Vereinigung in einer, und nicht, wie ursprünglich definiert, in zwei getrennten Operationen in die zu aktualisierende Tabelle eingebracht werden. Dadurch können CPU- und I/O-Kosten eingespart werden.

Des Weiteren können in der kombinierten DML-Anweisung Intra-Query-Parallelitäten entstehen, die durch eine entsprechende Unterstützung auf der Datenebene zu Parallelisierungseffekten führen können. Liegt beispielsweise ein Mehrrechner-Datenbanksystem zu Grunde, das über mehrere Prozessoren- und Festplatteneinheiten verfügt, können die beiden durch `UNION-All` verknüpften Select-Anfragen von S_{DML_i} und S_{DML_j} im besten Fall unabhängig voneinander und damit parallel verarbeitet werden. Dadurch kann sich die Ausführungszeit der kombinierten DML-Anweisung $S_{DML_{j+i}}$ im Vergleich zur Einzelausführung von S_{DML_i} und S_{DML_j} erheblich reduzieren. Solche Parallelisierungseffekte sind auch für ein föderiertes Datenbanksystem möglich: Bei geeigneter Partitionierung der Daten können die beiden Select-Anfragen auf unterschiedlichen Datenknoten des föderierten Systems verteilt und dort unabhängig voneinander ausgeführt werden. Ob eine solche Intra-Query-Parallelität tatsächlich genutzt werden kann, hängt allerdings immer vom Anfrageoptimierer des ausführenden Datenbanksystems ab, der über die Auswertungsstrategie einer vorliegenden SQL-Anweisung entscheidet.

Im besten Fall können die in der DML-Anweisung kombinierten Select-Anfragen unabhängig voneinander auf verschiedenen Ressourcen ausgeführt werden. Vor dem Verschmelzen wurden beide Anfragen strikt sequentiell bearbeitet. Im schlechtesten Fall ist eine parallele Verarbeitung beider Select-Anfragen ausgeschlossen. Laufzeitnachteile entstehen im Vergleich zu der Hintereinanderausführung der beiden einzelnen Select-Anfragen nicht.

Schließlich müssen noch die positiven Optimierungseffekte auf der Workflowebene berücksichtigt werden: Zum einen kann auf die Ausführung einer *SQL-DML*-Aktivität verzichtet werden; zum anderen kann durch die Anwendung der *Insert-Insert-Merging*-Regel auf eine temporäre Tabelle die gewinnbringende *Eliminate-Temporary-Table*-Regel vorbereitet werden (siehe Teilkapitel 5.4.7).

Korrektheit Durch die `UNION-ALL`-Operation werden die beiden in die Tabelle *PremiumCustomers* einzufügenden Datenmengen lediglich vereinigt. Somit wird vor und nach der Regelanwendung dieselbe Datenmenge in die zu aktualisierende Tabelle eingefügt.

Delete-Delete-Merging

Die *Delete-Delete-Merging*-Regel (siehe Regeldefinition D.7) kombiniert zwei Delete-Anweisungen, die nacheinander auf derselben Tabelle ausgeführt werden, zu einer einzelnen Delete-Anweisung, indem die Selektionsprädikate in der Where-Klausel beider Anweisungen mittels einer `OR`-Verknüpfung verschmolzen werden.

Anwendungsbeispiel Das in Abbildung 5.9 dargestellte Datenverarbeitungsmuster erfüllt alle Voraussetzungen zur Anwendung einer *Delete-Delete-Merging*-Regel.

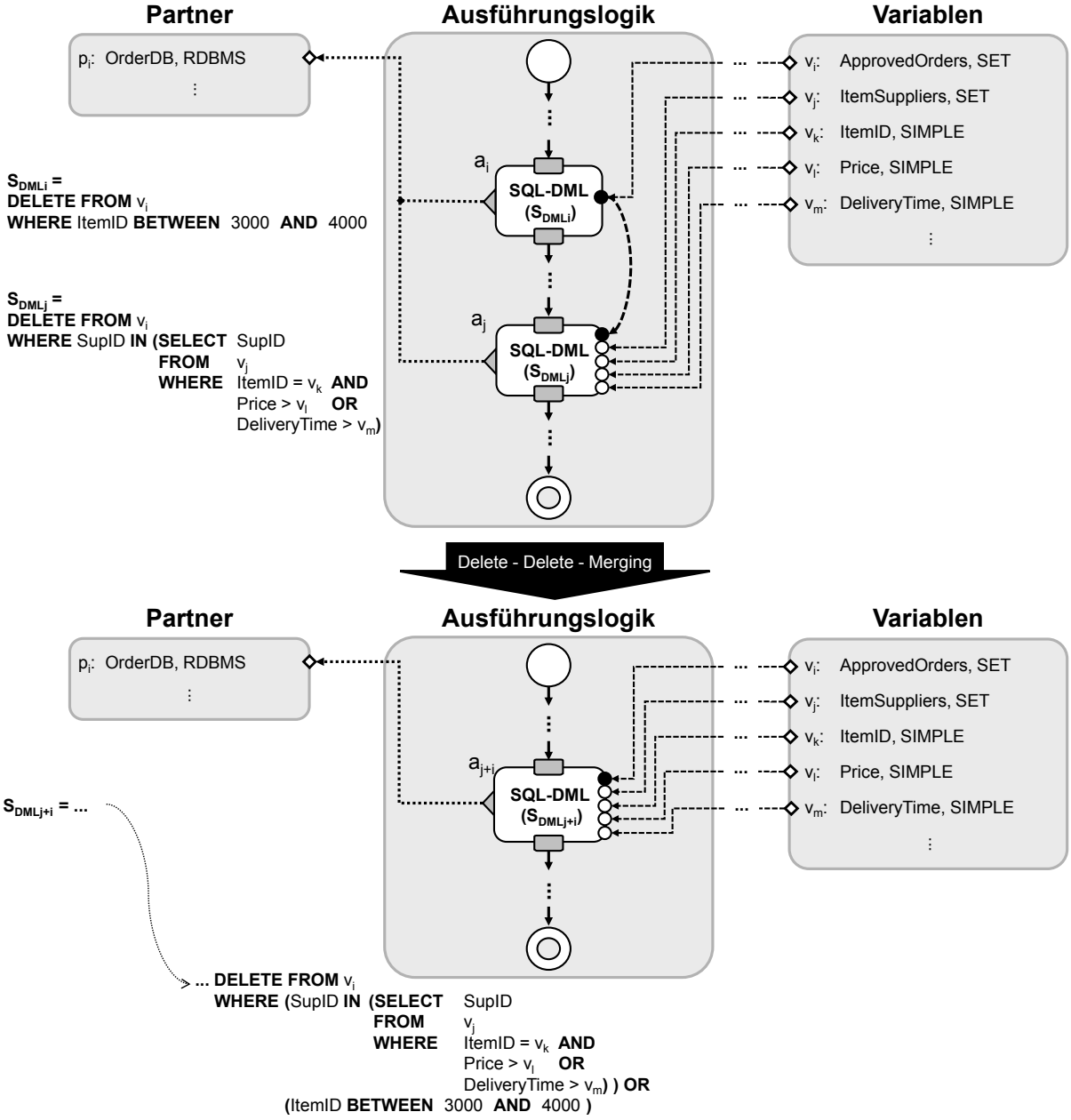


Abbildung 5.9.: Anwendung der *Delete-Delete-Merging*-Regel

In diesem Beispielszenario führen die beiden *SQL-DML*-Aktivitäten a_i und a_j jeweils zwei Delete-Anweisungen S_{DML_i} und S_{DML_j} nacheinander auf der Tabelle *ApprovedOrders* aus, die in der zugehörigen PGM-Repräsentation durch die `SET`-Variable v_i dargestellt wird. Die erste Operation löscht eine bestimmte Gruppe von Bestellungen aus *ApprovedOrders*. Die zweite Operation entfernt dagegen alle Bestellungen von Lieferanten, deren Preis bzw. Lieferzeit einen gegebenen Schwellwert überschreiten. Hierfür enthält S_{DML_j} drei Eingabeparameter *ItemID*, *Price* und *DeliveryTime*, die jeweils durch die `SIMPLE`-Variablen v_k, v_l und v_m repräsentiert werden.

Aufgrund der Schreib-Schreib-Datenabhängigkeit basierend auf v_i lassen sich S_{DML_i} bzw. S_{DML_j} zu einer einzelnen DML-Anweisung kombinieren, indem ihre beiden Where-Klauseln mittels einer `OR`-Verknüpfung verbunden werden. Die hieraus resultierende DML-Anweisung $S_{DML_{j+i}}$ ist in der unteren Hälfte von Abbildung 5.9 dargestellt.

Optimierungseffekt Vor dem Verschmelzen müssen für beide Delete-Anweisungen jeweils die Tupel, die sich für eine Delete-Operation qualifizieren, in der Tabelle *ApprovedOrders* bestimmt werden. Im schlechtesten Fall muss jeweils ein kompletter Scan auf *ApprovedOrders* durchgeführt werden. Nach dem Verschmelzen ist durch die Kombination beider Selektionsprädikate in der Where-Klausel lediglich ein solcher Scan auf *ApprovedOrders* notwendig. Im besten Fall lassen sich die anfallenden CPU-Kosten halbieren. Werden zudem in der Where-Klausel der kombinierten Delete-Anweisung zwei voneinander unabhängige Unteranfragen miteinander verknüpft, können diese analog zur *Insert-Insert-Merging*-Regel bei einer geeigneten Unterstützung auf der Datenebene parallel ausgeführt werden, wodurch sich weitere Laufzeitgewinne ergeben können.

Korrektheit Durch die `OR`-Verknüpfung werden die Selektionsprädikate in den Where-Klauseln beider Delete-Anweisungen verbunden. Somit ist sichergestellt, dass im kombinierten Fall genau jene Tupel aus der zu aktualisierenden Tabelle entfernt werden, die bei der Einzelausführung der beiden Delete-Operationen ebenfalls gelöscht werden.

Update-Update-Merging

Die *Update-Update-Merging*-Regel (siehe Regeldefinition D.8) verschmilzt zwei Update-Anweisungen, die auf derselben Tabelle nacheinander ausgeführt werden, zu einer Update-Anweisung durch Kombination der Set-Klauseln beider DML-Anweisungen.

Anwendungsbeispiel Die *Update-Update-Merging*-Regel kann auf das in Abbildung 5.10 dargestellte Datenverarbeitungsmuster angewendet werden. In diesem Beispielszenario führen die DML-Aktivitäten a_i und a_j jeweils zwei Update-Anweisungen S_{DML_i} und S_{DML_j} aus, die beide auf der Tabelle *ApprovedOrders* (v_i) ausgeführt werden. Beide Anweisungen aktualisieren Bestellungen aus *ApprovedOrders*, deren Lieferzeit einen bestimmten Schwellwert überschreiten, welcher in der Tabelle *ItemSuppliers* (v_k) für eine Bestellung hinterlegt ist. Dabei unterscheiden sich beide Anweisungen lediglich in ihren Set-Klauseln. S_{DML_i} aktualisiert das Attribut *ItemQuantity* der Tabelle *ApprovedOrders*, wohingegen S_{DML_j} sich auf die Aktualisierung des Attributs *Price* beschränkt.

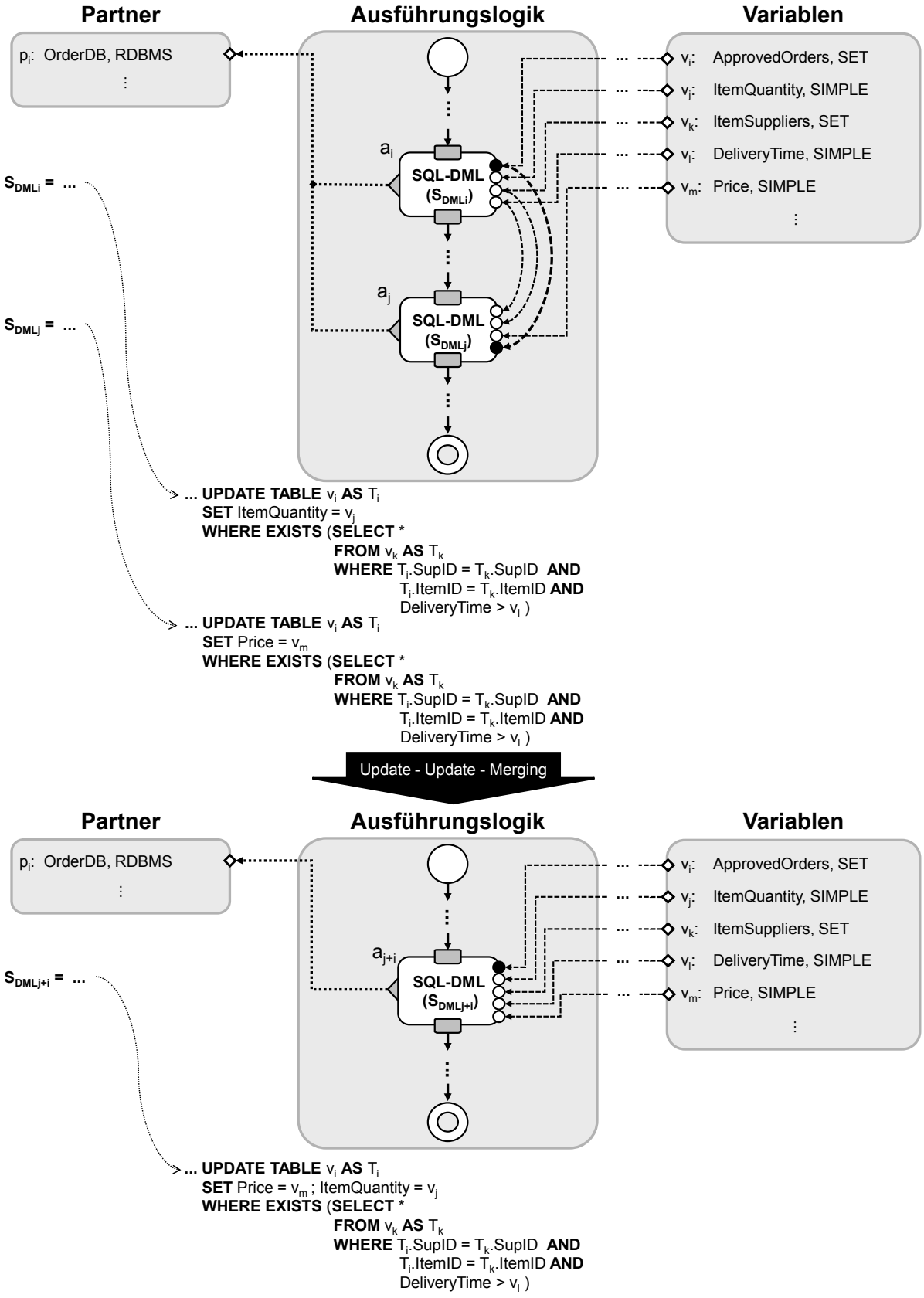


Abbildung 5.10.: Anwendung der *Update-Update-Merging*-Regel

Somit ist gewährleistet, dass S_{DML_i} und S_{DML_j} auf derselben Tupelmenge von *Approved-Orders* ausgeführt werden. Folglich können beide Update-Anweisungen miteinander kombiniert werden. Das Ergebnis der Regelanwendung ist in der unteren Hälfte von Abbildung 5.10 illustriert. Die kombinierte DML-Anweisung $S_{DML_{j+i}}$ vereinigt die zuvor nacheinander ausgeführten Update-Operationen, indem die Set-Klauseln von S_{DML_i} und S_{DML_j} direkt hintereinander geschrieben werden.

Optimierungseffekt Die Tupel, die sich für eine Update-Operation qualifizieren, müssen vor dem Verschmelzen für jede Update-Anweisung jeweils wiederholt in *ApprovedOrders* bestimmt werden. Folglich muss ein zu aktualisierendes Tupel zweimal referenziert werden, um beide Update-Operationen darauf anwenden zu können. Im schlechtesten Fall muss jeweils ein kompletter Scan auf der Tabelle *ApprovedOrders* durchgeführt werden. Nach dem Verschmelzen ist in $S_{DML_{j+i}}$ durch Kombination beider Set-Klauseln lediglich ein solcher Scan auf der Tabelle notwendig, um die beiden Aktualisierungsoperationen auf einem Tupel ausführen zu können. Insbesondere muss ein zu aktualisierendes Tupel nur noch einmal referenziert werden. Hierdurch lassen sich die CPU- und I/O-Kosten um die Hälfte reduzieren. Wie in diesem Beispiel gezeigt, halbiert sich durch die Regelanwendung auch die Anzahl der Ausführungen von Unteranfragen in einer Update-Operation.

Korrektheit Unterschiedliche Attribute eines Tupels aus der Tabelle *ApprovedOrders* werden vor dem Verschmelzen in zwei getrennten Update-Operationen aktualisiert. Nach dem Verschmelzen erfolgt die Aktualisierung dieser Attribute des Tupels in einem Schritt. Dies ist korrekt, da sich beide Update-Anweisungen lediglich in ihren Set-Klauseln unterscheiden. Zudem werden beide Update-Anweisungen auf derselben Tupelmenge einer zu aktualisierenden Tabelle ausgeführt. Dies kann aus folgenden Gründen garantiert werden: Zum einen durch die Gleichheit der Selektionsprädikate in der Where-Klausel. Zum anderen durch die Tatsache, dass kein Attribut eines zu aktualisierenden Tupels in der Set-Klausel der ersten Update-Anweisung, das auch in der Where-Klausel dieser Anweisung zur Selektion der zu aktualisierenden Tupelmenge referenziert wird, geändert wird. Schließlich ist eine korrekte Ausführung der kombinierten Update-Anweisung garantiert, da die beiden Set-Klauseln unterschiedliche Attribute eines Tupels referenzieren und somit gemäß des SQL-Standards hintereinander ausgeführt werden können.

5.4.6. Query-Merging

Das zwischen einem Workflow- und einem Datenbanksystem zu übertragende Datenvolumen kann für die Laufzeit eines datenintensiven Workflows entscheidend sein. Durch eine Reduzierung der Größe von Anfrageergebnismengen, die von der Daten- auf die Workflowebene übertragen werden müssen, lassen sich Übertragungs- und Materialisierungskosten senken. Deshalb sollte auf unnötige Übertragungen von Anfrageergebnismengen verzichtet werden. Dies kann durch die Regeln *Predicate-Pushdown*, *Select-Into-Merging* und *Select-Merging*, die in den folgenden Abschnitten vorgestellt werden, erreicht werden.

Predicate-Pushdown

Die frühzeitige Ausführung von Selektionsoperationen ist eine bewährte Heuristik in der SQL-Anfrageoptimierung, um die Größe von Zwischenergebnissen, die zur Ermittlung eines Anfrageergebnisses erforderlich sind, zu verkleinern. Hierdurch lassen sich hohe Laufzeitgewinne erzielen. Entsprechende Regeln findet man sowohl in der klassischen SQL-Anfrageoptimierung als auch in MQO bzw. CGO (siehe Teilkapitel 2.2).

Ebenso lässt sich diese Optimierungsidee auf eine PGM-Repräsentation zur Optimierung datenintensiver Workflows übertragen. Voraussetzung ist die Existenz einer *SQL-Query*-Aktivität, deren Anfrageergebnis in einer `SET`-Variablen materialisiert wird, auf der anschließend ein in einer *Assign-Select*-Aktivität definiertes Selektionsprädikat angewendet wird. Durch die *Predicate-Pushdown*-Regel (siehe Regeldefinition D.9) wird das Selektionsprädikat von der *Assign-Select*- in die *SQL-Query*-Aktivität verschoben. Dies kann zu einer Verkleinerung der Anfrageergebnismenge, welche an die *SQL-Query*-Aktivität zurückgesendet wird, führen. Hiermit verbunden ist eine Reduzierung der Übertragungs- und Materialisierungskosten. Zudem können sich durch die Regelanwendung neue Optimierungsmöglichkeiten für einen Anfrageoptimierer ergeben, die zu einer beschleunigten Ausführung der SQL-Anfrage auf der Datenebene führen können.

Anwendungsbeispiel Abbildung 5.11 zeigt das Datenverarbeitungsmuster, das in einer PGM-Repräsentation vorliegen muss, damit die *Predicate-Pushdown*-Regel korrekt angewendet werden kann.

Das Muster besteht aus einer *SQL-Query*-Aktivität a_i, die mittels der Anfrage S_{Q_i} eine Bestellmenge aus der Tabelle *ApprovedOrders* (v_j) bestimmt, deren Bestellwert, welcher aus der Variablen *Price* (v_j) ausgelesen wird, eine bestimmte Summe übersteigt. Das Anfrageergebnis wird in der `SET`-Variablen *ApprovedOrders2* (v_k) zur Verfügung gestellt.

Die in der Ausführungslogik folgende *Assign-Select*-Aktivität a_j wendet auf v_k ein Selektionsprädikat der Form σ an, das Bestellungen einer bestimmten Lieferantengruppe bzw. Bestellmenge aus *ApprovedOrders2* herausfiltert. Dabei dienen die Werte der Variablen *SupID* (v_l) und *ItemQuantity* (v_m) als Eingabeparameter für die Selektionsoperation. Das Anfrageergebnis wird in der `SET`-Variablen *ApprovedOrders3* (v_n) abgespeichert.

Aufgrund der exklusiven Schreib-Lese-Datenabhängigkeit zwischen a_i und a_j basierend auf der `SET`-Variablen v_k kann die *Predicate-Pushdown*-Regel korrekt angewendet werden.

Das Ergebnis der Regelanwendung ist in der unteren Hälfte von Abbildung 5.11 dargestellt. Die *SQL-Query*-Aktivität a_{i+j}, die aus a_i abgeleitet wurde, führt die SQL-Anfrage $S_{Q_{i+j}}$ aus, die durch Verschieben des Selektionsprädikats σ von a_j nach S_{Q_i} entstanden ist. Dabei wurde das Selektionsprädikat σ mittels einer logischen `AND`-Verknüpfung mit dem Selektionsprädikat in der Where-Klausel von S_{Q_i} kombiniert. Somit selektiert die SQL-Anfrage $S_{Q_{i+j}}$ genau jene Tupel aus v_j, die zuvor durch S_{Q_i} und σ in zwei getrennten Operationen auf zwei unterschiedlichen Ausführungsebenen bestimmt worden sind.

Durch das Verschieben der Selektionsoperation von der Workflow- auf die Datenebene kann die *Assign-Select*-Aktivität a_j aus der Ausführungslogik der PGM-Repräsentation

entfernt werden. Zudem wird die `SET`-Variable *ApprovedOrders2* (v_k) nicht mehr länger in a_{i+j} referenziert. Stattdessen wird der einzige `Write`-Dataslot von a_{i+j} mit der `SET`-Variablen *ApprovedOrders3* (v_n) verknüpft, die zuvor von a_j schreibend referenziert wurde. Diese Vertauschung der Slot-Belegung ist erforderlich, um die auf v_n basierenden Datenabhängigkeiten in der PGM-Repräsentation zu erhalten.

Wird Variable v_k nicht mehr in der PGM-Repräsentation referenziert, wird sie am Ende des Optimierungslaufes durch die *Eliminate-Unused-Variable*-Regel entfernt. Dadurch wird die Konsistenz der PGM-Repräsentation wiederhergestellt.

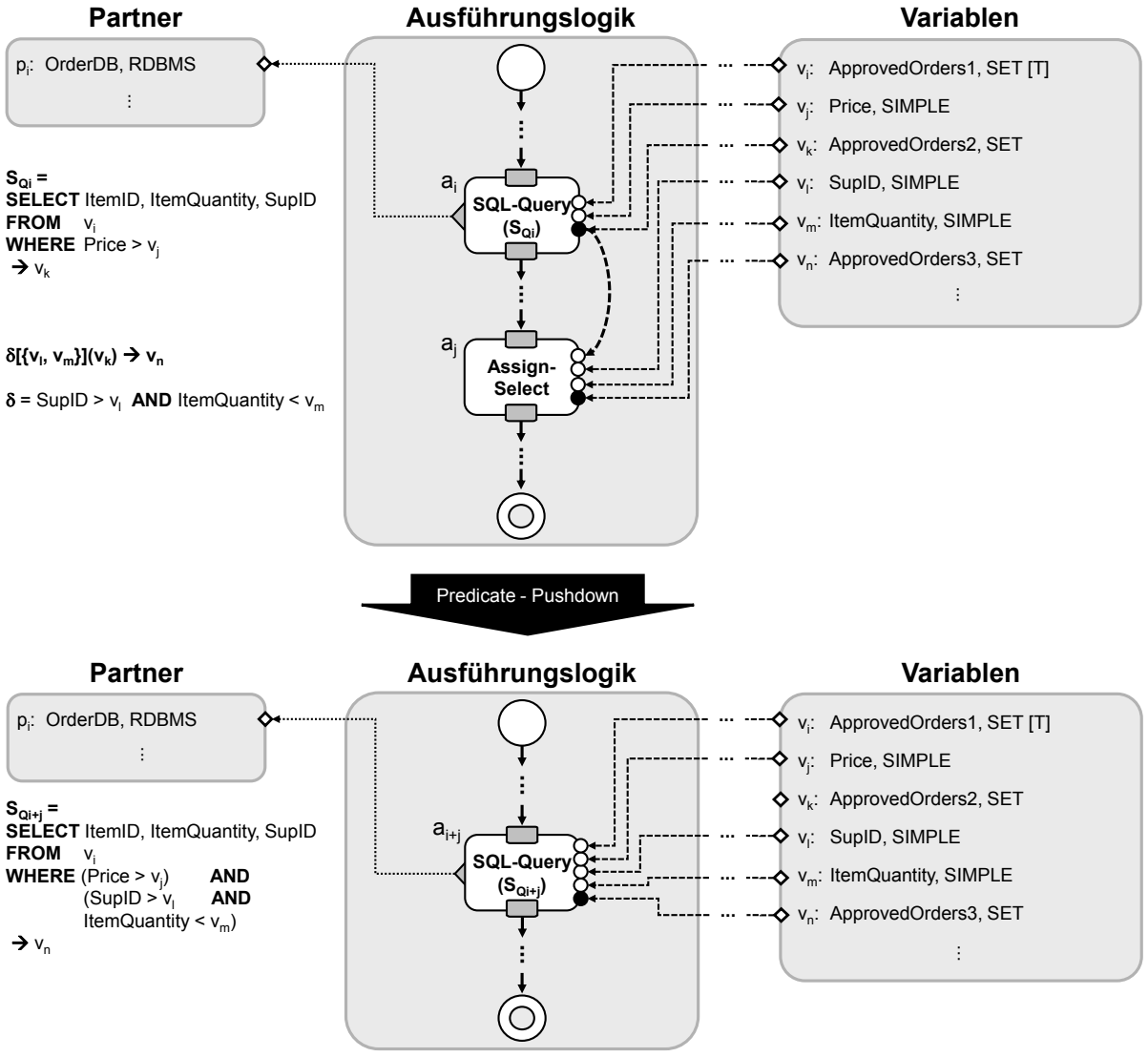


Abbildung 5.11.: Anwendung der *Predicate-Pushdown*-Regel

Optimierungseffekte Mit der *Predicate-Pushdown*-Regel können Berechnungskosten für die Ausführung einer Selektionsoperation auf einer materialisierten Datenmenge auf der Workflowebene eingespart werden. Durch das Verschieben des Selektionsprädikates in

die definierende SQL-Anfrage steigt zudem die Selektivität des Selektionsprädikates der Anfrage. Dadurch kann die Ergebnismenge und damit das an ein Workflowsystem zu übertragende Datenvolumen reduziert werden. Außerdem erweitern sich durch eine solche Verschiebung die Optimierungsmöglichkeiten für einen Anfrageoptimierer. Wird beispielsweise ein verschobenes Prädikat vor einem Verbund ausgeführt, verringert sich die Anzahl der am Verbund beteiligten Tupel. Ebenso kann durch die Regelanwendung die Verwendung von Indexstrukturen ermöglicht werden, welche die Auswertung des Prädikates unterstützen. Diese Effekte können dazu führen, dass der Anfrageoptimierer andere Verbundimplementierungen oder sogar andere Verbundreihenfolgen auswählen kann. Dies kann sich positiv auf die Laufzeit eines datenintensiven Workflows auswirken.

Korrektheit Vor dem Verschmelzen wird das Anfrageergebnis von S_{Q_i} materialisiert und in der `SET`-Variablen v_k gespeichert. Anschließend wird eine Selektionsoperation σ auf v_k ausgeführt. Das Ergebnis der beiden getrennt ausgeführten Datenverarbeitungsoperationen ist eine materialisierte Datenmenge, die dem Anfrageergebnis von S_{Q_i} abzüglich der darin enthaltenen Tupel, die das Selektionsprädikat σ nicht erfüllen, entspricht.

Nach dem Verschmelzen wird das Selektionsprädikat σ direkt auf den Tupeln, die von S_{Q_i} selektiert werden, angewendet. Die logische `AND`-Verknüpfung stellt sicher, dass das Anfrageergebnis von $S_{Q_{i+j}}$ genau jener Anfragemenge entspricht, die vor der Regelanwendung durch die Ausführung von S_{Q_i} und σ jeweils getrennt voneinander geliefert wurde.

Wie bereits in Teilkapitel 4.3.4 diskutiert, ist bei der Abbildung eines Workflowmodells nach PGM zu gewährleisten, dass nur Selektionsprädikate, die der Form σ entsprechen und somit korrekt auf einen äquivalenten SQL-Ausdruck abgebildet werden können, mithilfe einer *Assign-Select*-Aktivität repräsentiert werden.

Variante der Predicate-Pushdown-Regel

Abbildung 5.12 zeigt ein Datenverarbeitungsmuster in einer PGM-Repräsentation, auf das eine Variante der *Predicate-Pushdown*-Regel (siehe Regeldefinition D.10) angewendet werden kann.

Das Datenverarbeitungsmuster besteht aus einer *SQL-Query*-Aktivität a_i, die eine Anfrage S_{Q_i} auf der Tabelle *Orders* ausführt. Das materialisierte Anfrageergebnis wird in der `SET`-Variablen *OrdersResults* (v_j) gespeichert und vom folgenden `LOOP`-Kontrollflussmuster sequentiell durchlaufen. Dazu liefert die *Set-Iterator*-Aktivität a_k in jedem Durchlauf das nächste Tupel der materialisierten Datenmenge in v_j, das im Anschluss von einer *XOR-Split-Select*-Aktivität a_l bearbeitet wird. Aktivität a_l entscheidet anhand des Attributwertes von *ItemID*, ob sich ein vorliegendes Tupel für eine Weiterverarbeitung qualifiziert oder nicht.

Die „späte“ Filterung der Tupel der materialisierten Datenmenge in v_j durch die *XOR-Split-Select*-Aktivität ist sehr ineffizient. Das von der *SQL-Query*-Aktivität a_i gelieferte Anfrageergebnis enthält Daten, die auf der Workflowebene nicht benötigt werden. Dadurch entstehen zusätzliche Materialisierungs- und Übertragungskosten. Ferner steigen die Ausführungskosten für das *LOOP*-Kontrollflussmuster proportional zur Größe der unnötig übertragenen Datenmenge.

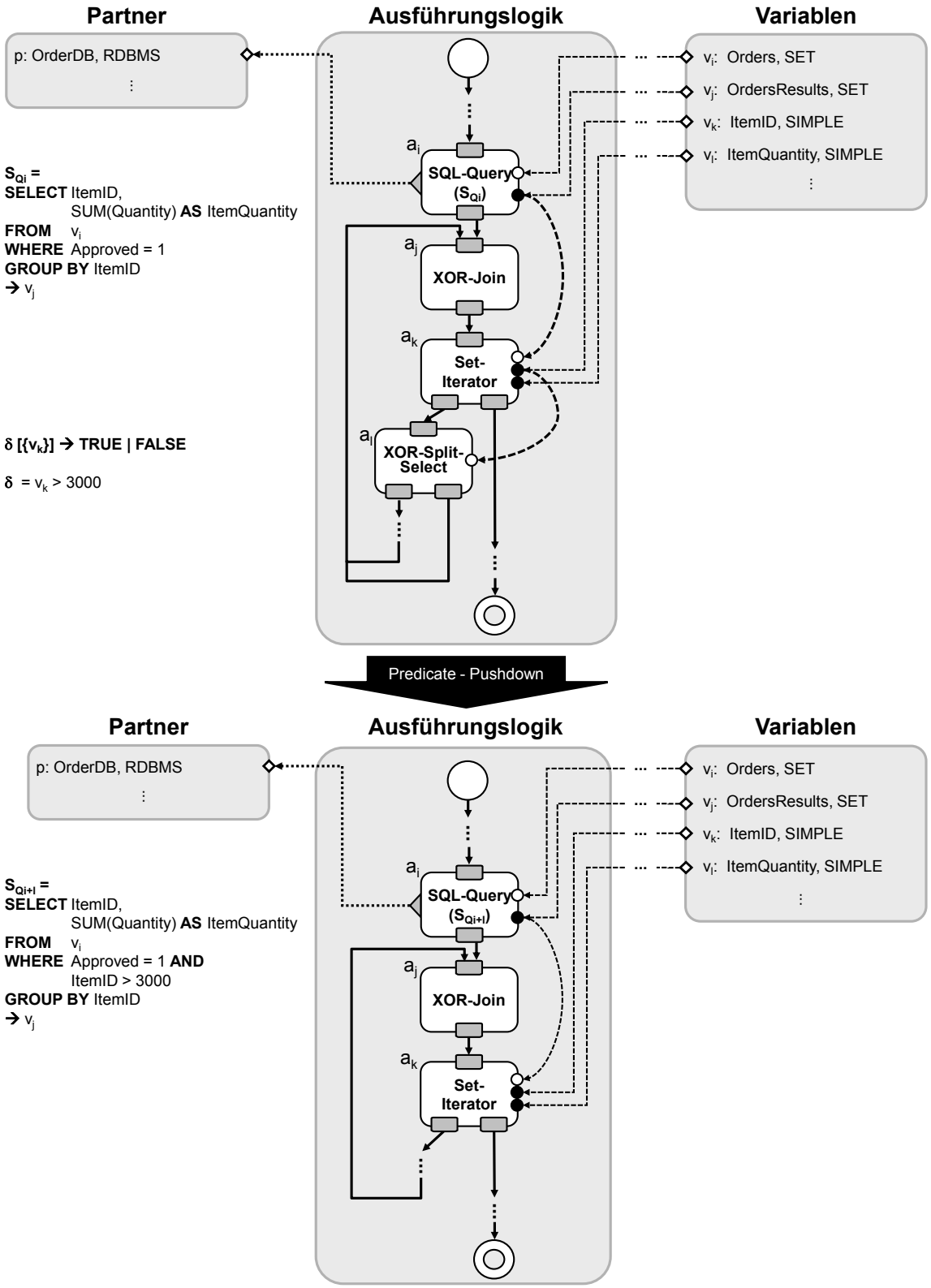


Abbildung 5.12.: Anwendung einer Variante der *Predicate-Pushdown*-Regel

Für die Reduzierung der Laufzeit eines solchen Datenverarbeitungsmusters ist entscheidend, überflüssige Daten aus einer zu iterierenden, materialisierten Datenmenge frühzeitig zu eliminieren. Dies kann durch Kombination des Selektionsprädikats σ der *XOR-Split-Select*-Aktivität mit dem Selektionsprädikat der Where-Klausel der SQL-Anfrage S_{Q_i} erreicht werden. Dieser Transformationsschritt stellt sicher, dass das resultierende Anfrageergebnis genau jene Daten enthält, die für eine Weiterverarbeitung auf der Workflowebene relevant sind. Hierdurch beschleunigt sich die Ausführung des *LOOP*-Kontrollflussmusters, da zum einen über eine potentiell kleinere Datenmenge iteriert werden muss, und zum anderen auf die Ausführung der *XOR-Split-Select*-Aktivität verzichtet werden kann. Zudem kann ein Datenverarbeitungsmuster entstehen, das eine *Tuple-To-Set*-Regel ermöglicht (siehe Teilkapitel 5.5). Das Ergebnis der Anwendung dieser Variante der *Predicate-Pushdown*-Regel ist in der unteren Hälfte von Abbildung 5.12 dargestellt.

Select-Into-Merging

Eine Select-Into-Anfrage selektiert ein einzelnes Tupel aus einer Datenmenge und stellt dessen Attributwerte in einzelnen Variablen zur Verfügung. Werden die Attributwerte anschließend von einer SQL-Anweisung referenziert, lassen sich die auf diesen Variablen basierenden Datenabhängigkeiten auflösen, indem die Select-Into-Anfrage (genauer: die Select-Into-Anfrage ohne die Insert-Klausel) direkt mit der abhängigen SQL-Anweisung verschmolzen wird. Hierdurch können die Ergebnisparameter der Select-Into-Anfrage unmittelbar auf der Datenebene von der abhängigen SQL-Anweisung eingelesen werden.

Die *Select-Into-Merging*-Regel (siehe Regeldefinition D.11) ist eine wichtige Regel für die Verschmelzung von Web-Service-Aufrufen, die zuvor durch Anwendung der *Web-Service-Pushdown*-Regel in eine Select-Into-Anweisung gewandelt wurden, mit anderen abhängigen SQL-Anweisungen.

Ersetzung skalarer Eingabeparameter in einer SQL-Anweisung Eine Select-Into-Anfrage liefert als Ergebnis genau n skalare Werte, die als Eingabeparameter für eine folgende, abhängige SQL-Anweisung dienen können. Wird jeder der n Parameter in der abhängigen Anweisung durch die definierende Select-Into-Anfrage ersetzt, entsteht unter Umständen eine Einzelanfrage mit redundanten Unteranfragen. Wurde die Select-Into-Anfrage ursprünglich genau einmal vom Datenbanksystem ausgeführt, kann sie nach dem Verschmelzen sogar bis zu n-Mal ausgeführt werden. Eine solche redundante Ausführung von Unteranfragen kann, wie bereits in [Kra09] diskutiert wurde, zu erheblichen Laufzeitverschlechterungen führen.

Selbst wenn genau ein Parameter durch die definierende Select-Into-Anfrage ersetzt wird, ist nicht sichergestellt, dass die Select-Into-Anfrage, wie ursprünglich definiert, genau einmal vom Datenbanksystem ausgeführt wird. Dies hängt allein von der Auswertungsstrategie eines ausführenden Anfrageoptimierers ab, auf die im Allgemeinen von außen kein Einfluss genommen werden kann.

Neben der möglichen Laufzeitverschlechterung können sich durch eine redundante Ausführung der ursprünglichen Select-Into-Anfrage weitere negative Effekte ergeben:

- Aufgrund von Phantomen kann bei einer wiederholten Ausführung einer Select-Into-Anfrage jedes Mal ein anderes Ergebnis geliefert werden, wenn im ausführenden Datenbanksystem nicht die Phantome verhindernde Konsistenzstufe 3 nach [GLPT76] eingestellt ist. In diesem Fall kann sich das Anfrageergebnis der abhängigen Anfrage vor und nach der Regelanwendung unterscheiden.
- Die Problematik der Substitution verschärft sich, wenn innerhalb der Select-Into-Anfrage eine WS-UDTF, welche durch die Anwendung der *Web-Service-Pushdown*-Regel generiert worden ist, aufgerufen wird. Wird die aufgerufene Web-Service-Operation nicht, wie ursprünglich definiert, genau einmal, sondern entsprechend öfter von einer WS-UDTF ausgeführt, kann die Ausführungssemantik eines Workflows geändert werden: Liegt eine nichtdeterministische Web-Service-Operation vor, liefert diese bei gleichbleibender Eingabe unterschiedliche Ergebnisse. Dadurch kann sich das ursprüngliche Anfrageergebnis der abhängigen Anfrage ändern.
- Ebenso kann sich ein redundanter Web-Service-Aufruf auf den Zustand einer vom Web-Service referenzierten Ressource auswirken, wenn diese nicht, wie ursprünglich definiert, genau einmal, sondern öfter referenziert wird. Schließlich ist zu berücksichtigen, dass sich durch einen wiederholten Web-Service-Aufruf das ursprüngliche Kommunikationsmuster zwischen dem Workflow und dem aufgerufenen Web-Service sowie die ursprüngliche Ausführungssemantik des Workflows ändern.
- Weitere Probleme treten auf, wenn die *Select-Into-Merging*-Regel mehrfach mit unterschiedlichen Select-Into-Anfragen auf eine einzelne, abhängige SQL-Anweisung angewendet wird. Da die Ausführungsreihenfolge dieser Select-Into-Anfragen nach dem Verschmelzen vom Anfrageoptimierer bestimmt wird, ist nicht mehr sichergestellt, dass nach der Regelanwendung die ursprüngliche Ausführungsreihenfolge zwischen den einzelnen Select-Into-Anfragen erhalten werden kann.

Bei Verwendung der Substitutionsstrategie hängt es somit ausschließlich von der Auswertungsstrategie eines Anfrageoptimierers ab, ob die ursprüngliche Ausführungssemantik eines Workflows bei der Verschmelzung von SQL-Anweisungen erhalten werden kann. Deshalb sollte die Substitutionsstrategie nur dann angewendet werden, wenn ein Anfrageoptimierer eine redundante Ausführung von Unteranfragen vermeidet. Dies wird aber nur sehr eingeschränkt möglich sein, weil ein solches Verhalten eines Anfrageoptimierers in der Regel von außen nicht festgelegt werden kann. Deshalb wird die Substitutionsstrategie bei der Definition der *Select-Into-Merging*-Regel nicht angewendet.

Stattdessen wird zur Lösung der oben diskutierten Probleme eine With-Klausel verwendet, in welche eine Select-Into-Anfrage verschoben wird, um die gewünschte Verschmelzung durchzuführen. Die With-Klausel stellt sicher, dass eine Anfrage innerhalb einer With-Klausel zum einen genau einmal und zum anderen vor der eigentlich abhängigen Anweisung ausgeführt wird. Damit bleiben die ursprünglichen Daten- und Kontrollflussabhängigkeiten zwischen einer Select-Into-Anfrage und einer abhängigen Anweisung gewahrt. Wenn mehrere Select-Into-Anfragen mit derselben abhängigen Anweisung ver-

schmolzen werden, kann die ursprüngliche Ausführungsreihenfolge zwischen den einzelnen Select-Into-Anfragen erhalten bleiben, da innerhalb der With-Klausel die dort definierten Anweisungen entsprechend ihrer Reihenfolge hintereinander ausgewertet werden. Folglich können auch in diesem Fall die ursprünglichen Kontrollfluss- und Datenabhängigkeiten in der kombinierten Anfrage korrekt reflektiert werden.

Da eine With-Klausel Teil einer Select-Anfrage ist, kann die *Select-Into-Merging*-Regel nur auf abhängige Select-Anfragen bzw. Insert-Select-Anweisungen angewendet werden. Hierbei kann es sich auch um eine Unteranfrage handeln, die innerhalb einer Where-Klausel einer Anfrage oder einer DML-Anweisung ausgeführt wird. In diesem Fall kann die Unteranfrage nur innerhalb eines mengenbasierten SQL-Prädikates, wie z.B. `IN`, `EXISTS`, `ALL`, `ANY` oder `SOME`, definiert sein, innerhalb dessen die abhängige Unteranfrage genau einmal ausgeführt und deren Ergebnis anschließend zur Evaluierung des Prädikates herangezogen wird. Es ist aber zu beachten, dass eine solche Evaluierungsstrategie von dem SQL-Standard nicht zwingend vorgegeben ist. Allerdings ist davon auszugehen, dass die führenden Datenbanksysteme aus Effizienzgründen keine andere Evaluierungsstrategie zulassen. Somit kann unterstellt werden, dass nach der Verschmelzung der abhängigen Unteranfrage mit der Select-Into-Anfrage die abhängige Unteranfrage innerhalb der Where-Klausel weiterhin genau einmal ausgeführt wird, und Laufzeitverschlechterungen aufgrund redundanter Ausführungen der ursprünglichen Select-Into-Anfrage ausscheiden. Sonst muss die Regel entsprechend restriktiver formuliert und die Verschmelzung von Select-Into-Anfragen mit einer Unteranfrage innerhalb einer Where-Klausel einer SQL-Anweisung ausgeschlossen werden.

Wird die Select-Into-Anfrage mit einer Unteranfrage, die innerhalb einer Where-Klausel einer DML-Anweisung ausgeführt wird, kombiniert, muss zusätzlich beachtet werden, dass die Select-Into-Anfrage bzw. DML-Anweisung auf verschiedenen Tabellen ausgeführt werden. Sonst entsteht durch die *Select-Into-Merging*-Regel gemäß des SQL:2003-Standards eine nicht ausführbare DML-Anweisung.

Anwendungsbeispiel Die in Abbildung E.5 dargestellte PGM-Repräsentation des Beispiel-Workflows erfüllt alle Voraussetzungen für die *Select-Into-Merging*-Regel. Die SQL-Anfrage S_{Q_7}, die durch vorherige Anwendung der *Web-Service-Pushdown*-Regel entstanden ist, führt in ihrer From-Klausel die WS-UDTF *OrderItem* aus, welche die gleichnamige Operation des Web-Services $OrderFromSupplier$ aufruft. Der skalare Ausgabeparameter des Web-Service-Aufrufs wird in der Variablen $OrderConfirmation$ (v_9) abgespeichert. Diese Variable dient zusammen mit den Variablen $v_5 - v_8$ als Eingabeparameter für die folgende Insert-Anweisung S_{DML_8}, das ein einzelnes Tupel in die Tabelle $OrderConfirmations$ einfügt.

Aufgrund der exklusiven Schreib-Lese-Datenabhängigkeit basierend auf dem Ausgabeparameter von S_{Q_7}, können beide SQL-Anweisungen miteinander kombiniert werden, indem die Select-Into-Anfrage in die With-Klausel der Select-Anfrage von S_{DML_8} verschoben wird. Auf diese Weise kann das Ergebnis der Select-Into-Anfrage in der Insert-Anweisung direkt referenziert werden. Infolgedessen ist eine separate Ausführung von S_{Q_7} nach der Regelanwendung nicht mehr erforderlich.

Die resultierende SQL-Anweisung $S_{DML_{8+7}}$ ist in Abbildung E.7 dargestellt. Darin wurde S_{Q_7} in die With-Klausel der Select-Anfrage der Insert-Anweisung S_{DML_8} verschoben. Dadurch wird die Variable v_9 in S_{DML_8} direkt durch den Korrelationsnamen $T_1.c_1$ ersetzt, welcher den in der With-Klausel berechneten skalaren Ausgabeparameter des Web-Service-Aufrufs nach der Regelanwendung referenziert. Am Ende des Optimierungslaufes wird die Variable v_9 durch die *Eliminate-Unused-Variable*-Regel entfernt, weil sie in der resultierenden PGM-Repräsentation nicht mehr referenziert wird (siehe Abbildung E.9).

Optimierungseffekt auf der Datenebene Durch die kombinierte Ausführung von S_{Q_7} und S_{DML_8} können im Vergleich zur Hintereinanderausführung beider Anweisungen Berechnungskosten eingespart werden. Vor der Regelanwendung muss das von der Select-Into-Anfrage ermittelte Tupel an das Workflowsystem übermittelt und materialisiert werden. Anschließend wird es als Eingabeparameter zusammen mit der abhängigen SQL-Anweisung wieder an das ausführende Datenbanksystem zurückgesendet. Die ineffiziente Parameterübergabe über Systemgrenzen hinweg kann durch die kombinierte Ausführung beider Anweisungen vermieden werden. Das von der Select-Into-Anfrage ermittelte Tupel kann nach der Regelanwendung unmittelbar im Hauptspeicher des Datenbanksystems an die abhängige Anweisung übergeben werden. Auf diese Weise lassen sich CPU-, Übertragungs- und Materialisierungskosten einsparen.

Korrektheit Wie bereits diskutiert, ist die Korrektheit der Regelanwendung durch die Verwendung einer With-Klausel sichergestellt. Zum einen bleiben die ursprünglich sequentiell modellierten Kontrollflussabhängigkeiten zwischen den kombinierten SQL-Anweisungen erhalten, d.h. die Select-Into-Anfrage wird auch nach der Regelanwendung vor der abhängigen Anweisung ausgeführt. Zum anderen ist garantiert, dass die Select-Into-Anfrage auch nach der Regelanwendung genau einmal ausgeführt wird, und es somit zu keinen Änderungen in der Ausführungssemantik der Select-Into-Anfrage kommen kann.

Select-Merging

Die *Select-Merging*-Regel (siehe Regeldefinition D.12) bietet eine weitere Möglichkeit zur Auflösung von Datenabhängigkeiten zwischen SQL-Anfragen und Web-Service-Aufrufen. Voraussetzung ist die Existenz einer SQL-Anfrage, die ein Anfrageergebnis liefert, deren Materialisierung als Eingabeparameter für einen folgenden Web-Service-Aufruf vom Typ 3 oder 4 genutzt wird. Es wird gefordert, dass der Web-Service-Aufruf bereits durch Anwendung der *Web-Service-Pushdown*-Regel in eine entsprechende SQL-Anfrage (im Folgenden WS-Anfrage genannt) gewandelt worden ist. Diese Datenabhängigkeit lässt sich durch Verschiebung der SQL-Anfrage in die With-Klausel der abhängigen WS-Anfrage auflösen. Dabei muss das Anfrageergebnis der SQL-Anfrage materialisiert werden, damit es vom Web-Service korrekt verarbeitet werden kann. Hierbei werden SQL/XML-Funktionen eingesetzt, die ein relationales Anfrageergebnis in eine XML-Datenstruktur überführen können. SQL/XML ist Teil des SQL:2003-Standards und wird von allen führenden Datenbankherstellern unterstützt. Durch einen solchen Transformationsschritt entfallen die

Kosten zur Übertragung und Materialisierung der Ergebnismenge der SQL-Anfrage an das Workflowsystem, wodurch positive Laufzeiteffekte erzielt werden können. Zudem ist ein solcher Transformationsschritt Voraussetzung für einen effizienten Web-Service-Aufruf vom Typ 3 oder 4 auf der Datenebene (siehe Teilkapitel 5.3.1).

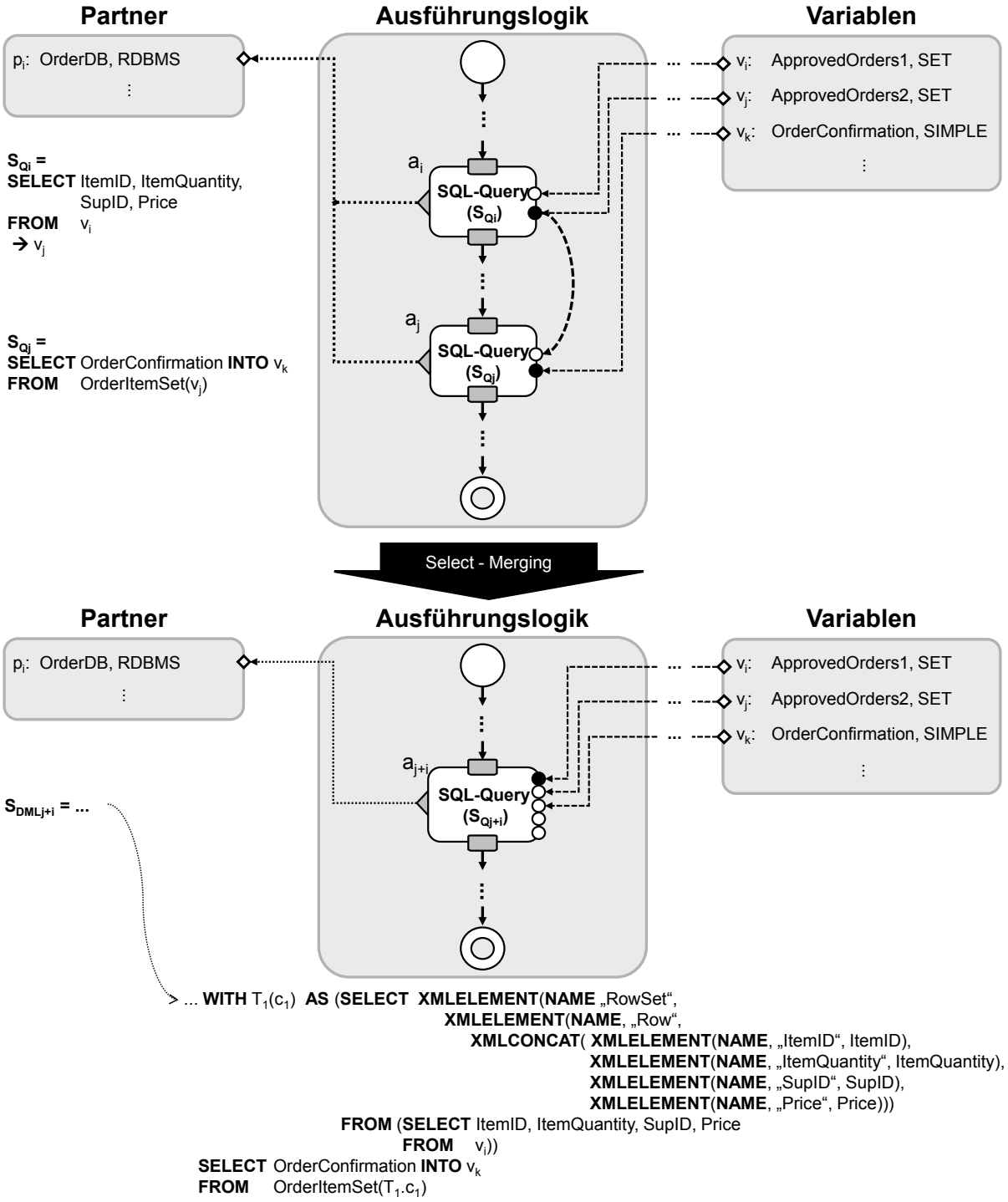


Abbildung 5.13.: Anwendung der *Select-Merging*-Regel

Anwendungsbeispiel Abbildung 5.13 stellt ein Datenverarbeitungsmuster dar, das alle Voraussetzungen für eine *Select-Merging*-Regel erfüllt. Die beiden *SQL-Query*-Aktivitäten a_i und a_j führen nacheinander zwei SQL-Anfragen S_{Q_i} und S_{Q_j} aus, bei denen S_{Q_i} eine Ergebnismenge liefert, die anschließend von S_{Q_j} exklusiv als Eingabeparameter gelesen wird. Eine solche Schreib-Lese-Datenabhängigkeit zwischen beiden Select-Anfragen ist nur möglich, wenn S_{Q_j} eine WS-UDTF ausführt, die durch vorherige Anwendung einer *Web-Service-Pushdown*-Regel auf einen Web-Service-Aufruf vom Typ 3 entstanden ist.

In dem Beispiel liest die SQL-Anfrage S_{Q_i} die Tabelle *ApprovedOrders* (v_i) aus und stellt das Anfrageergebnis in der `SET`-Variablen *ApprovedOrders2* (v_j) materialisiert zur Verfügung, die von der folgenden SQL-Anfrage S_{Q_j} als Eingabeparameter konsumiert wird. Die SQL-Anfrage S_{Q_j} führt eine WS-UDTF aus, welche die Operation *OrderItemSet* vom Typ 3 des Web-Services *OrderFromSupplier* aufruft (siehe auch WSDL-Beschreibung in E.3). Die Web-Service-Operation nimmt als Eingabe die materialisierte Datenmenge v_j entgegen und liefert einen skalaren Ausgabewert, der in der Variablen $OrderConfirmation$ (v_k) gespeichert wird.

Die zwischen beiden SQL-Anweisungen existierende exklusive Schreib-Lese-Datenabhängigkeit basierend auf v_j kann wie folgt aufgelöst werden: Zunächst wird die Anfrage S_{Q_i} in die With-Klausel der abhängigen Anfrage S_{Q_j} verschoben. Um das Anfrageergebnis von S_{Q_i} in eine äquivalente XML-RowSet-Datenstruktur zu transformieren, werden SQL/-XML-Funktionen, die auf das Anfrageergebnis von S_{Q_i} angewendet werden, eingesetzt. Die resultierende Anfrage $S_{Q_{j+i}}$ ist in der unteren Hälfte von Abbildung 5.13 dargestellt.

Die Wandlung des Anfrageergebnisses von S_{Q_i} in eine entsprechende XML-RowSet-Datenstruktur wird in $S_{Q_{j+i}}$ wie folgt durchgeführt: Zuerst werden die Attributwerte von $ItemID$, $ItemQuantity$, $SupID$ und $Price$ für jedes Tupel von *ApprovedOrders* mithilfe der SQL/XML-Funktion `XMLELEMENT` in entsprechende XML-Elemente transformiert. Als Ergebnis entsteht für jeden Attributwert eines Tupels ein zugehöriges XML-Element, dessen Name dem Attributnamen und dessen Inhalt dem Attributwert eines Tupels entspricht. Die XML-Elemente werden für jedes Tupel mittels der SQL/XML-Funktion `XMLCONCAT` hintereinandergereiht und dienen als Inhalt für ein XML-Element namens „Row“. Auf diese Weise entsteht für jedes Tupel der Anfrageergebnismenge ein entsprechendes Row-Element in der XML-Datenstruktur. Zuletzt wird die erzeugte Menge an Row-Elementen in ein umschließendes RowSet-Element eingebettet. Die resultierende XML-RowSet-Datenstruktur wurde bereits in Abbildung 2.1 veranschaulicht.

Optimierungseffekt auf der Datenebene Das Anfrageergebnis von S_{Q_i} muss vor der Regelanwendung an die Workflowebene transferiert und dort materialisiert werden. Anschließend wird die Materialisierung als Eingabeparameter an den Web-Service übertragen. Die mit dieser ineffizienten Parameterübergabe verbundenen Kosten steigen proportional mit der Größe einer Anfrageergebnismenge.

Durch die *Select-Merging*-Regel lässt sich dieser Mehraufwand reduzieren. Das Anfrageergebnis von S_{Q_i} kann direkt auf der Datenebene materialisiert und an den Web-Service übermittelt werden. Hierdurch können Übertragungs- und Materialisierungskosten in beträchtlichem Umfang eingespart werden. Durch die Verlagerung der Materialisierungs-

funktion auf die Datenebene entstehen zudem keine nennenswerten Veränderungen für das Laufzeitverhalten des Workflows, da eine Materialisierung auf der Datenebene in ähnlicher Weise wie auf der Workflowebene implementiert wird.

Zudem ist zu beachten, dass erst die *Select-Merging*-Regel eine gewinnbringende Verlagerung eines Web-Service-Aufrufes vom Typ 3 bzw. 4 ermöglicht. Denn dadurch kann verhindert werden, dass nach dem *Web-Service-Pushdown* die Ergebnismenge der Select-Anfrage zuerst an das Workflowsystem und danach wieder an das Datenbanksystem, von wo aus der Web-Service-Aufruf durchgeführt wird, zurück übermittelt wird. Ein solches Ausführungsverhalten kann, wie bereits in Teilkapitel 5.3.1 diskutiert, abhängig von der Größe der zu übertragenden Ergebnismenge auch zu Laufzeitverschlechterungen führen.

Korrektheit Die Verwendung der With-Klausel garantiert, dass bei der Anwendung der *Select-Merging*-Regel zum einen die ursprüngliche sequentielle Kontrollflussabhängigkeit zwischen der SQL- und WS-Anfrage erhalten bleibt, und dass zum anderen die SQL-Anfrage weiterhin genau einmal ausgeführt wird.

Die Materialisierungsfunktion auf der Datenebene muss die Korrektheit des Materialisierungsschrittes sicherstellen. SQL/XML bietet alle Funktionen an, um aus einer relationalen Datenstruktur eine entsprechende XML-Datenstruktur zu generieren. Die genaue Form der XML-Datenstruktur hängt von der zu Grunde gelegten Workflowbeschreibungssprache bzw. von dem zugehörigen XML-Schema ab, das in der WSDL-Beschreibung des aufzurufenden Web-Services für eine solche mengenbasierte Datenstruktur definiert wird. Im Falle von BPEL/SQL entspricht die XML-Datenstruktur einem XML-RowSet.

Eliminierung redundanter SQL-Anfragen mittels heuristischer CGO-Regeln

Der heuristische CGO-Ansatz (siehe Teilkapitel 2.2.3) wendet eine Regelmenge auf eine Sequenz von Insert-Select-Anweisungen an, um die Ablauffolge zu optimieren. Dabei werden ähnlich strukturierte Insert-Select-Anweisungen geeignet kombiniert.

Die Optimierungsidee von CGO besteht darin, ähnlich strukturierte Unteranfragen von Insert-Select-Anweisungen, die sich etwa nur in einer Klausel unterscheiden, zu verschmelzen. So existieren entsprechende Regeln zur Verschmelzung von Select- (*MergeSelect*-Regel), Where- (*MergeWhere*-Regel), Having- (*MergeHaving*-Regel) und GroupBy-Klauseln (*WhereToGroup*-Regel). Durch das Verschmelzen der Klauseln wird eine der beiden Anfragen redundant und kann somit aus der Anweisungssequenz entfernt werden. Die I/O- und CPU-Kosten können auf der Datenebene um die Hälfte reduziert werden, da auf die in der Anfrage referenzierten Tabellen nur noch einmal und nicht wie zuvor zweimal zugegriffen werden muss. Im Falle der *MergeWhere*-Regel bzw. *MergeHaving*-Regel lässt sich aufgrund der Kombination von Selektionsprädikaten zudem die Selektivität der kombinierten Anfrage steigern. Dadurch kann die Größe einer Ergebnismenge reduziert werden. Bei der *WhereToGroup*-Regel können weitere Laufzeitgewinne auf der Datenebene erzielt werden, da auf die Berechnung einer teuren Gruppierungsfunktion verzichtet werden kann. Eine ausführlichere Beschreibung der CGO-Regeln ist in [Kra09] zu finden.

Da es sich beim heuristischen CGO-Ansatz um ein Teilproblem der PGM-Optimierung handelt, kann dieser Optimierungsansatz auch auf den Kontext von PGM übertragen werden. Dabei lassen sich die Restrukturierungsregeln von CGO in einer PGM-Repräsentation nicht nur auf Insert-Select-, sondern auch auf SQL-Anfragen gewinnbringend anwenden: Durch die Kombination ähnlich strukturierter SQL-Anfragen können auf der Datenebene die Ausführungskosten kombinierter SQL-Anfragen im Vergleich zu deren Einzelausführungen gesenkt werden. Ebenso kann die Größe von Anfrageergebnismengen verkleinert werden. Dadurch muss ein geringeres Datenvolumen an die Workflowebene übertragen und dort materialisiert werden. Ferner lässt sich die Anzahl der *SQL-Query*-Aktivitäten in einer PGM-Repräsentation reduzieren, wodurch sich weitere Laufzeitgewinne ergeben können.

Es ist zu beachten, dass die heuristischen CGO-Regeln erst auf den Kontext von PGM übertragen werden müssen, um sie für die Optimierung datenintensiver Workflows nutzbar zu machen. Dies bedeutet insbesondere, dass neben Optimierungssphären auch Kontrollfluss-, Daten- und Kommunikationsabhängigkeiten zwischen den in einer PGM-Repräsentation definierten Aktivitäten berücksichtigt werden müssen, um eine korrekte Anwendung der heuristischen CGO-Regeln im Rahmen der PGM/F-Optimierung zu gewährleisten.

5.4.7. Eliminate-Temporary-Table

Temporäre Tabellen werden typischerweise dazu verwendet, die für weiterführende Berechnungsschritte benötigten Zwischenergebnisse zu speichern. Solche Tabellen verursachen aber sowohl auf der Workflow- als auch auf der Datenebene einen nicht zu vernachlässigenden Mehraufwand, der sich auf die Laufzeit eines datenintensiven Workflows negativ auswirken kann. Zum einen muss für jede temporäre Tabelle eine entsprechende Create- und Drop-Anweisung an ein zu Grunde gelegtes Datenbanksystem gesendet werden. Dort sind entsprechende Katalogdaten für eine temporäre Tabelle zu erzeugen bzw. wieder zu entfernen. Zum anderen wird mindestens eine Insert-Anweisung benötigt, mit der ein Zwischenergebnis in die temporäre Tabelle geladen wird. Es ist zu beachten, dass dieser Mehraufwand für jede Workflowinstanz zu leisten ist, wodurch sich ein entsprechend hohes Optimierungspotential ergibt.

Die *Eliminate-Temporary-Table*-Regel (siehe Regeldefinition D.13) entfernt redundante, temporäre Tabellen aus einer PGM-Repräsentation, deren gespeichertes Zwischenergebnis von genau einer abhängigen SQL-Anweisung referenziert wird. Dabei wird eine solche Referenz innerhalb einer SQL-Anweisung direkt durch die SQL-Anfrage ersetzt, die das Zwischenergebnis berechnet. Hierdurch kann der Mehraufwand zur Verwaltung einer temporären Tabelle sowohl auf der Workflow- als auch auf der Datenebene reduziert werden.

Anwendungsbeispiel Die in Abbildung E.16 dargestellte kombinierte PGM-Repräsentation des Beispiel-Workflows und der Beispiel-Stored-Procedure aus Abbildung 2.2 erfüllen alle Voraussetzungen zur Anwendung der *Eliminate-Temporary-Table*-Regel. Diese PGM-Repräsentation spiegelt den kombinierten finalen Zustand der beiden PGM-Repräsentationen des Beispiel-Workflows und der Beispiel-Stored-Procedure im heteroge-

nen, isolierten Optimierungsszenario wider (siehe Abbildungen E.9 und E.13). Dabei ist Aktivität a_{6+3} durch Anwendung einer *Insert-Tuple-To-Set*-Regel entstanden. Entsprechend ist Aktivität $a_{11+10+7}$ das Resultat der Regelanwendungen *Web-Service-Pushdown*, *Select-Into-Merging* und *Insert-Tuple-To-Set*.

In dieser kombinierten PGM-Repräsentation führt die *SQL-DML*-Aktivität a_{6+3} die Insert-Select-Anweisung $S_{DML_{6+3}}$ aus, welche die leere temporäre Tabelle *Approved-Orders* mit Bestelldaten befüllt. Dabei wird *ApprovedOrders* durch die `SET`-Variable *SR_ApprovedOrders* (v_8) repräsentiert. Dass es sich um eine leere temporäre Tabelle handelt, wird zum einen durch die Eigenschaft [`T`] (siehe Definition 7 in Anhang A.3) und zum anderen durch die direkte Verbindung von v_8 zum `Write`-Dataslot von Aktivität a_{6+3}, der mit v_8 assoziiert ist, explizit angezeigt.

Die von $S_{DML_{6+3}}$ befüllte temporäre Tabelle *ApprovedOrders* wird anschließend von der Insert-Anweisung $S_{DML_{11+10+7}}$ exklusiv lesend referenziert. Somit existiert zwischen a_{6+3} und $a_{11+10+7}$ eine exklusive Schreib-Lese-Datenabhängigkeit basierend auf `SET`-Variable v_8. Die Datenabhängigkeit lässt sich auflösen, indem das Vorkommen von v_8 in $S_{DML_{11+10+7}}$ direkt durch die definierende Select-Anfrage von $S_{DML_{6+3}}$ ersetzt wird. Als Ergebnis erhält man die in Abbildung E.17 dargestellte DML-Anweisung $S_{DML_{11+10+7+6+3}}$, die von *SQL-DML*-Aktivität $a_{11+10+7+6+3}$ ausgeführt wird.

Durch diesen Substitutionsschritt wird die Referenzierung der temporären Tabelle *ApprovedOrders* überflüssig. Dies spiegelt sich auch in den Dataslots der Aktivität $a_{11+10+7+6+3}$ wider, bei welcher der `Read`-Dataslot, der ursprünglich in $a_{11+10+7}$ mit der `SET`-Variablen v_8 assoziiert war, entfernt wurde. Somit kann nach Anwendung der *Eliminate-Temporary-Table*-Regel nicht nur auf die Ausführung der Aktivität a_{6+3}, sondern auch auf die `SET`-Variable v_8 und damit auf die Erzeugung und Verwaltung der temporären Tabelle *ApprovedOrders* verzichtet werden. Die Variable v_8 wird anschließend durch Anwendung der *Eliminate-Redundant-Variable*-Regel entfernt, um die Konsistenz der optimierten PGM-Repräsentation wiederherzustellen (siehe Abbildung E.18).

Optimierungseffekt auf der Datenebene Durch Anwendung der *Eliminate-Temporary-Table*-Regel kann auf die explizite Materialisierung einer Ergebnismenge durch eine temporäre Tabelle verzichtet werden. Damit verbunden sind Einsparungen von Ausführungskosten für die zur Definition einer temporären Tabelle notwendigen Create-, Insert- und Drop-Anweisungen. Diese Optimierungseffekte können somit zu erheblichen Laufzeitgewinnen sowohl auf der Workflow- als auch auf der Datenebene führen.

Korrektheit Durch die Regelanwendung wird die Referenz auf die temporäre Tabelle *ApprovedOrders* in $S_{DML_{11+10+7}}$, die durch `SET`-Variable v_8 repräsentiert wird, direkt durch die definierende Select-Anweisung aus $S_{DML_{6+3}}$ ersetzt. Ein solcher Substitutionsschritt ist gemäß des SQL-Standards zulässig. Somit wird nach der Regelanwendung die zuvor in die temporäre Tabelle *ApprovedOrders* explizit eingefügte Datenmenge direkt in der abhängigen DML-Anweisung $S_{DML_{11+10+7}}$ berechnet. Folglich wird vor und nach der Regelanwendung dieselbe Datenmenge in die Tabelle $OrderConfirmations$ eingefügt.

Des Weiteren ist zu beachten, dass eine temporäre Tabelle auch in einer Unteranfrage, die innerhalb einer Where-Klausel in einer Anfrage oder in einer DML-Anweisung ausgeführt wird, referenziert werden kann. Wie bereits bei der *Select-Into*-Regel diskutiert, ist eine Regelanwendung hier nur gewinnbringend, wenn sichergestellt ist, dass die Unteranfrage genau einmal ausgeführt wird. Sonst kann es nach dem Substitutionsschritt aufgrund redundanter Ausführungen der definierenden Anfrage sogar zu Laufzeitverschlechterungen kommen. In einem solchen Fall darf die *Eliminate-Temporary-Table*-Regel nicht auf Unteranfragen angewendet werden.

Die hier vorgestellte Variante der *Eliminate-Temporary-Table*-Regel ist nur dann effektiv anwendbar, wenn genau eine Referenz auf eine temporäre Tabelle durch die definierende SQL-Anfrage ersetzt wird. Existieren dagegen mehrere solcher Referenzen in einer abhängigen SQL-Anweisung, kann ein mehrfacher Substitutionsschritt zu einer redundanten Ausführung einer definierenden Anfrage und somit zu einer Laufzeitverschlechterung führen. Wie bereits bei der *Select-Into-Merging*-Regel diskutiert, kann dies verhindert werden, indem eine definierende SQL-Anfrage in die With-Klausel einer abhängigen SQL-Anweisung verschoben wird. Bei der abhängigen SQL-Anweisung kann es sich dabei um eine Select- oder Insert-Select-Anweisung handeln. Dadurch ist sichergestellt, dass die in die With-Klausel verschobene SQL-Anfrage genau einmal berechnet wird, auch wenn ihr Ergebnis anschließend öfter in der abhängigen Anweisung referenziert wird. Somit können Laufzeitverschlechterungen auch bei einer Mehrfachreferenzierung einer temporären Tabelle verhindert werden.

5.5. Die Regelklasse Tuple-To-Set

Eine Datenverarbeitungsoperation, die für jedes Tupel einer materialisierten Datenmenge aufgerufen wird, besitzt ein hohes Optimierungspotential. Typischerweise entsteht eine solche ineffiziente Ausführung, wenn eine gegebene materialisierte Datenmenge sequentiell durchlaufen wird und dabei die einzelnen Tupel der Menge von einer folgenden Datenverarbeitungsoperation als Eingabeparameter eingelesen werden. Folglich steigt die Anzahl der Ausführungen einer tupelbasierten Datenverarbeitungsoperation proportional zur Größe einer zu iterierenden Datenmenge.

Ziel der *Tuple-To-Set*-Regelklasse ist es, solche ineffizienten Datenverarbeitungsmuster aufzulösen, indem eine tupelbasierte in eine äquivalente mengenbasierte Ausführung einer DML-Anweisung überführt wird. Dazu wird die Anfrage, welche die zu iterierende materialisierte Datenmenge als Ergebnis liefert, mit der DML-Anweisung verschmolzen. Somit kann das ineffiziente Datenverarbeitungsmuster direkt durch die transformierte, mengenbasierte DML-Anweisung ersetzt werden.

Mit diesem Transformationsschritt werden erhebliche Laufzeitgewinne erzielt, da eine mengenbasierte DML-Anweisung auf der Datenebene effizienter ausgeführt werden kann als das äquivalente, prozedurale Datenverarbeitungsmuster auf der Workflowebene.

Die genauen Transformationsschritte hängen von der Art einer DML-Anweisung ab. Folglich kann zwischen einer *Insert-*, *Update-* und *Delete-Tuple-To-Set*-Regel, die in den folgen-

den Teilkapiteln beschrieben werden, unterschieden werden. Davor werden in den beiden folgenden Teilkapiteln die Bedingungen, Aktionen und Optimierungseffekte beschrieben, die für alle Regeln dieser Klasse gelten.

5.5.1. Klassenspezifische Regelbeschreibung

Regelbedingung

Um eine *Tuple-To-Set*-Regel korrekt anwenden zu können, müssen in einer PGM-Repräsentation folgende klassenspezifische Bedingungen erfüllt sein:

- Das Datenverarbeitungsmuster, auf das eine *Tuple-To-Set*-Regel angewendet werden kann, beginnt mit einer *SQL-Query*-Aktivität a_i, die eine Select-Anfrage S_{Q_i} beinhaltet. Das Ergebnis dieser Anfrage wird in einer `SET`-Variablen v_i bereitgestellt. Zu beachten ist, dass an dieser Stelle auch eine SQL-Anfrage, die durch vorherige Anwendung der *Web-Service-Pushdown*-Regel auf einen Web-Service-Aufruf vom Typ 4 hervorgegangen sein kann, stehen kann.
- Es folgt ein `LOOP`-Kontrollflussmuster, das aus einer *XOR-Join*-Aktivität a_j und einer Set-Iterator-Aktivität a_k, die eine iterative Verarbeitung der in v_i gespeicherten Daten ermöglicht, gebildet wird. Innerhalb des Schleifenrumpfes wird eine *SQL-DML*-Aktivität a_l, welche die DML-Anweisung S_{DML_l} definiert, ausgeführt.
- Bei jedem Durchlaufen stellt die *Set-Iterator*-Aktivität a_k die Attributwerte des nächsten Tupels aus v_i in entsprechenden `SIMPLE`-Variablen, die als Eingabeparameter für S_{DML_l} genutzt werden, zur Verfügung.
- Sofern Tupel aus v_i gelesen werden können, geht der Kontrollfluss zur nachfolgenden *SQL-DML*-Aktivität a_l innerhalb der Schleife über, die auf demselben Datenbanksystem ausgeführt wird wie zuvor *SQL-Query*-Aktivität a_i.
- Wurde v_i vollständig verarbeitet, wird die Schleife verlassen, und der Kontrollfluss wird an die dem `LOOP`-Kontrollflussmuster folgende Aktivität weitergeleitet.
- Innerhalb der Schleife verarbeitet die *SQL-DML*-Aktivität a_l die jeweils aktuellen Tupelwerte, die in den von a_k geschriebenen `SIMPLE`-Variablen bereitgestellt werden.
- Da a_l der einzige Leser dieser Variablenwerte ist, existiert zwischen a_k und a_l eine exklusive Schreib-Lese-Datenabhängigkeit.
- Des Weiteren besteht eine exklusive Schreib-Lese-Datenabhängigkeit zwischen der *SQL-Query*-Aktivität a_i und der *Set-Iterator*-Aktivität a_k basierend auf der `SET`-Variablen v_i, die von a_i geschrieben und von a_k gelesen wird.

Regelaktion

Sind alle Voraussetzungen erfüllt, können folgende Transformationsschritte in einer PGM-Repräsentation durchgeführt werden:

- Die tupelbasierte Ausführung von S_{DML_l} kann in eine mengenbasierte Ausführung umgewandelt werden, indem die Select-Anfrage S_{Q_i} mit S_{DML_l} kombiniert wird. Die einzelnen Transformationsschritte hängen von der Art der DML-Anweisung ab und werden deshalb bei den Beschreibungen der Regeln genauer diskutiert.
- Die resultierende DML-Anweisung $S_{DML_{l+i}}$ wird von der *SQL-DML*-Aktivität a_{l+i}, die aus a_l abgeleitet wurde, ausgeführt.
- Änderungen in den Lese- und Schreibmengen von $S_{DML_{l+i}}$ im Vergleich zur ursprünglichen DML-Anweisung S_{DML_l} müssen in den Dataslots der SQL-Aktivität a_{l+i} reflektiert werden.
- Eine geänderte Menge an Dataslots wirkt sich auf die Datenabhängigkeiten zwischen den Aktivitäten, welche diese Variablen gemeinsam bearbeiten, aus. Deshalb müssen für jeden geänderten Dataslot bzw. die mit diesem Dataslot assoziierte Variable die Datenabhängigkeiten in der PGM-Repräsentation neu berechnet werden.
- Die ursprüngliche Kommunikationsabhängigkeit der SQL-Aktivität a_{l+i} bleibt nach der Regelanwendung erhalten.
- Dagegen müssen die Kontrollflussabhängigkeiten in der PGM-Repräsentation wie folgt angepasst werden:
  - Zunächst werden die Aktivitäten a_i, a_j und a_k entfernt.
  - Anschließend wird der direkte Vorgänger von a_i bzw. direkte Nachfolger von a_k mit a_{l+i} verbunden. Hierfür müssen entsprechende Kontrollflusskanten in die PGM-Repräsentation eingefügt werden. Somit ist sichergestellt, dass Aktivität a_{l+i} an derselben Stelle in der Ausführungslogik ausgeführt wird wie zuvor das Datenverarbeitungsmuster bestehend aus den Aktivitäten a_i, a_j, a_k und a_l.
  - Des Weiteren entfällt die Kommunikationsabhängigkeit zwischen a_i und dem ausführenden Datenbanksystem.
  - Variable v_i wird durch das Entfernen von a_i aus der PGM-Repräsentation nicht mehr benötigt, weil das Anfrageergebniss von S_{Q_i} nach der Regelanwendung nicht mehr materialisiert werden muss. Dies gilt auch für die `SIMPLE`-Variablen, in die zuvor a_k die aktuellen Attributwerte eines Tupels geschrieben hat, da sie aufgrund der Kombination von S_{DML_l} und S_{Q_i} in $S_{DML_{l+i}}$ nicht mehr referenziert werden. Diese Variablen werden durch Anwendung der Regel *Eliminate-Unused-Variable* nach Abschluss des Optimierungslaufes entfernt, um die Konsistenz der transformierten PGM-Repräsentation sicherzustellen.

5.5.2. Klassenspezifische Optimierungseffekte

Die *Tuple-To-Set*-Regel ermöglicht sehr hohe Leistungssteigerungen: Da das ineffiziente, prozedurale Datenverarbeitungsmuster durch eine einzelne *SQL-DML*-Aktivität ersetzt wird, kann die Anzahl der Aktivitäten in der Ausführungslogik eines Workflowmodells reduziert werden. Insbesondere kann die Anzahl der Ausführungen der transformierten *SQL-DML*-Aktivität a_{l+i} gesenkt werden. Musste Aktivität a_l zuvor noch für jedes Tupel

der materialisierten Datenmenge in v_i ausgeführt werden, ist dies für die von a_l abgeleitete Aktivität a_{l+i} genau einmal notwendig. Der hierdurch erzielbare Optimierungsgewinn steigt somit proportional zur Größe der zu iterierenden Datenmenge. Im selben Maße reduzieren sich die Ausführungskosten auf der Datenebene. Musste zuvor noch die DML-Anweisung S_{DML_l} für jedes Tupel der materialisierten Datenmenge ausgeführt werden, ist dies für $S_{DML_{l+i}}$ genau einmal notwendig. Des Weiteren ist nach der Regelanwendung eine Übertragung und anschließende Materialisierung der Ergebnismenge der SQL-Anfrage S_{Q_i} in v_i auf der Workflowebene nicht mehr erforderlich, was zu einer weiteren Reduzierung der Ausführungskosten beiträgt.

Die Optimierungseffekte der *Tuple-To-Set*-Regeln können unabhängig davon erzielt werden, ob ein zentralisiertes oder ein föderiertes Datenbanksystem zu Grunde gelegt ist. Diese Regel ist nicht anwendbar, wenn die SQL-Anfrage S_{Q_i} und die DML-Anweisung S_{DML_l} auf zwei unterschiedlichen Datenbanksystemen ausgeführt werden, da sonst durch die Kombination beider Anweisungen eine nicht ausführbare Anweisung erzeugt wird. Dadurch wird die Konsistenz der PGM-Repräsentation verletzt.

5.5.3. Insert-Tuple-To-Set

Im Folgenden soll die Anwendung einer *Insert-Tuple-To-Set*-Regel (siehe Regeldefinition D.14) veranschaulicht werden, die eine tupelbasierte Insert-Anweisung in eine äquivalente, mengenbasierte Insert-Anweisung überführt.

Anwendungsbeispiel

Die *Insert-Tuple-To-Set*-Regel kann direkt an der PGM-Repräsentation des Beispiel-Workflows aus Abbildung E.7, die alle Voraussetzungen für die Anwendung dieser Regel erfüllt, veranschaulicht werden. Dargestellt ist der Zustand der PGM-Repräsentation nach Anwendung der *Select-Into-Merging*-Regel (siehe Teilkapitel 5.4.6).

Das dargestellte Datenverarbeitungsmuster beginnt mit einer *SQL-Query*-Aktivität (a_4), die eine Select-Anfrage (S_{Q_4}) beinhaltet. Das Ergebnis der Anfrage wird in der Variablen *SR_ApprovedOrders* (v_4) bereitgestellt. Im Folgenden `LOOP`-Kontrollflussmuster stellt die *Set-Iterator*-Aktivität (a_6) die Attributwerte des nächsten Tupels aus v_4 in den Variablen $ItemID$ (v_5), *ItemQuantity* (v_6), $SupID$ (v_7) und *Price* (v_8) zur Verfügung, die innerhalb der Schleife von der *SQL-DML*-Aktivität ($a_{8+7'}$) als Eingabeparameter ausgelesen und mittels einer Insert-Operation ($S_{DML_{8+7'}}$) in die Zieltabelle $SR\_OrderConfirmations$ (v_{10}) eingefügt werden. Somit existiert eine exklusive Schreib-Lese-Datenabhängigkeit zwischen der *SQL-Query*-Aktivität a_4 und der *Set-Iterator*-Aktivität a_6, basierend auf der `SET`-Variablen v_4, die von a_4 geschrieben und von a_6 gelesen wird. Ebenso ist eine exklusive Schreib-Lese-Datenabhängigkeit zwischen *Set-Iterator*-Aktivität a_6 und *SQL-DML*-Aktivität $a_{8+7'}$ für die Variablen v_5 - v_8 vorhanden, die von a_6 geschrieben und anschließend von $a_{8+7'}$ gelesen werden. Zudem ist *SQL-DML*-Aktivität $a_{8+7'}$ der einzige Leser der Variablen v_5 - v_8, die von der *Set-Iterator*-Aktivität a_6 geschrieben wurden. Damit sind alle Voraussetzungen für eine *Insert-Tuple-To-Set*-Regel erfüllt.

Abbildung E.8 zeigt die resultierende PGM-Repräsentation nach Anwendung der *Insert-Tuple-To-Set*-Regel. Aus der Regelanwendung geht eine einzelne *SQL-DML*-Aktivität $a_{8+7'+4}$ hervor, welche die mengenbasierte Insert-Anweisung $S_{DML_{8+7'+4}}$ ausführt. Die kombinierten Anfrageausdrücke, die auf den SQL-Anweisungen S_{Q_4} und $S_{DML_{8+7'}}$ aus a_4 und $a_{8+7'}$ basieren, liefern die Daten für die Einfügeoperation. Sie sind als korrelierte Tabellenausdrücke in der Insert-Anweisung $S_{DML_{8+7'+4}}$ definiert. Die Korrelation beider Tabellenausdrücke wird gemäß dem SQL-Standard mit dem Schlüsselwort `LATERAL` ausgedrückt. Das Ergebnis des ersten Tabellenausdruckes ist das Anfrageergebnis S_{Q_4} aus a_4, das vom zweiten Tabellenausdruck direkt weiterverarbeitet wird. Der zweite Tabellenausdruck ist von der Select-Anfrage der Insert-Anweisung $S_{DML_{8+7'}}$ abgeleitet und liefert als Ergebnis die in die Zieltabelle v_{10} einzufügende Datenmenge. Im nächsten Schritt ist die von $S_{DML_{8+7'+4}}$ referenzierte Variablenmenge zu bestimmen, und daraus sind die entsprechenden Lese- und Schreibmengen abzuleiten. Änderungen in den Lese- und Schreibmengen im Vergleich zur ursprünglichen DML-Anweisung $S_{DML_{8+7'}}$ müssen in den Dataslots der *SQL-DML*-Aktivität $a_{8+7'}$ reflektiert werden. Beispielsweise werden die Variablen v_5 - v_8 in $S_{DML_{8+7'+4}}$ nicht mehr als Eingabeparameter verwendet, weil sie durch die Korrelationsnamen $T_1.c_1$ - $T_4.c_4$ im zweiten Tabellenausdruck $S_{DML_{8+7'}}$ ersetzt wurden. Folglich besitzt Aktivität $a_{8+7'+4}$ für diese beiden Variablen keine `Read`-Dataslots mehr. Dafür wurde dieser Aktivität ein `Read`-Dataslot hinzugefügt, der mit der Variablen *SR_ApprovedOrders* (v_3) assoziiert ist, deren zu Grunde gelegte Tabelle in dem Anfrageausdruck S_{Q_4} referenziert wird. Die geänderte Menge an Dataslots wirkt sich auf die Datenabhängigkeiten zwischen den Aktivitäten aus, welche diese Variablen gemeinsam bearbeiten. Deshalb müssen für jeden geänderten Dataslot bzw. die mit diesem Dataslot assoziierte Variable die Datenabhängigkeiten neu berechnet werden. Die ursprüngliche Kommunikationsabhängigkeit der SQL-Aktivität $a_{8+7'}$ bleibt nach der Regelanwendung erhalten. Dagegen müssen die Kontrollflussabhängigkeiten in der PGM-Repräsentation wie folgt angepasst werden: Zuerst werden die Aktivitäten a_4, a_5 und a_6 aus der Ausführungslogik entfernt. Anschließend wird der direkte Vorgänger von a_4 (a_3) bzw. direkte Nachfolger von a_6 (a_9) mit $a_{8+7'+4}$ verbunden. Somit ist sichergestellt, dass Aktivität $a_{8+7'+4}$ in der Ausführungslogik an derselben Stelle ausgeführt wird wie zuvor das Datenverarbeitungsmuster bestehend aus a_4, a_5, a_6 und $a_{8+7'}$.

Durch das Entfernen von a_4 aus der PGM-Repräsentation wird Variable *SV_ApprovedOrders* (v_4) nicht mehr benötigt, da eine Materialisierung des Anfrageergebnisses von S_{Q_4} nach der Regelanwendung nicht mehr erforderlich ist. Die Variable wird zusammen mit den nicht mehr referenzierten Variablen v_5 - v_8 durch die Regel *Eliminate-Unused-Variable* nach dem Optimierungslauf aus der PGM-Repräsenation entfernt, um die Konsistenz der transformierten PGM-Repräsentation zu garantieren (siehe Abbildung E.9).

Die PGM-Repräsentation der Beispiel-Stored-Procedure in Abbildung E.11 erfüllt ebenso alle Voraussetzungen für die Anwendung einer *Insert-Tuple-To-Set*-Regel. Die resultierende mengenbasierte Insert-Anweisung ist in Abbildung E.12 dargestellt.

Korrektheit

Vor der Transformation wird für jedes Tupel aus der Ergebnismenge von S_{Q_4} die Insert-Anweisung $S_{DML_{8+7'}}$ ausgeführt. Dabei liest die Select-Anfrage der Insert-Anweisung $S_{DML_{8+7'}}$ (im Folgenden als $S_{Q_{8+7'}}$ bezeichnet) die Attributwerte eines vorliegenden Tupels als Eingabe und bestimmt damit die Ergebnismenge, die in die zu aktualisierende Tabelle einzufügen ist. Nach der Transformation wird die definierende Anfrage S_{Q_4} mit der abhängigen DML-Anweisung $S_{DML_{8+7'}}$ verschmolzen, indem S_{Q_4} und $S_{Q_{8+7'}}$ mittels einer Verbundoperation kombiniert werden. Dabei liest $S_{Q_{8+7'}}$ die Tupel der Ergebnismenge von S_{Q_4} als Eingabe. Aufgrund dieser Korrelation zwischen beiden Anweisungen muss gemäß des SQL-Standards das Schlüsselwort `LATERAL` verwendet werden. Bei der Verbundberechnung wird zunächst die Ergebnismenge der „äußeren“ Anfrage S_{Q_4} ermittelt. Anschließend wird für jedes Tupel in der Ergebnismenge die „innere“ Anfrage $S_{Q_{8+7'}}$ ausgeführt, um die entsprechenden Verbundpartner zu bestimmen. Dabei berechnet $S_{Q_{8+7'}}$ mittels der aktuellen Attributwerte eines Tupels die Ergebnismenge, welche in die zu aktualisierende Tabelle einzufügen ist. Folglich wird die innere Anfrage $S_{Q_{8+7'}}$ vor und nach der Regelanwendung mit denselben Eingabeparametern aufgerufen, was zu denselben Ergebnismengen führt, welche in die zu aktualisierende Tabelle einzufügen sind. Somit bleibt die ursprüngliche Ausführungssemantik des prozeduralen Datenverarbeitungsmusters auch nach einer *Insert-Tuple-To-Set*-Regel erhalten.

5.5.4. Update-Tuple-To-Set

Mithilfe einer *Update-Tuple-To-Set*-Regel (siehe Regeldefinition D.15) lässt sich eine tupelbasierte Update-Anweisung in eine äquivalente Merge-Anweisung transformieren. Die hierfür notwendigen Transformationsschritte werden im Folgenden beschrieben.

Anwendungsbeispiel

Abbildung 5.14 stellt ein Datenverarbeitungsmuster dar, das alle Voraussetzungen zur Anwendung einer *Update-Tuple-To-Set*-Regel erfüllt.

In diesem Beispielszenario erfolgt eine tupelbasierte Update-Operation auf der Tabelle *ApprovedOrders* (v_m), welche eine aktuelle Bestellmenge verwaltet. Dazu liest die SQL-Anfrage S_{Q_i} zunächst neue Bestellungen aus der Tabelle *Orders* (v_i) aus, die noch nicht in der Tabelle *ApprovedOrders* berücksichtigt worden sind. Dies kann über das Prädikat *OrderStatus = 0* bestimmt werden. Die daraus resultierende neue Bestellmenge wird in der `SET`-Variablen *NewOrders* (v_j) abgespeichert. Das folgende `LOOP`-Kontrollflussmuster iteriert über v_j und stellt in jeder Iteration die Attributwerte des nächsten gelieferten Tupels in den Variablen $ItemID$ (v_k) und *ItemQuantity* (v_l) zur Verfügung. Diese beiden Attributwerte dienen als Eingabeparameter für die Update-Anweisung S_{DML_l}, die im Schleifenrumpf des `LOOP`-Kontrollflussmusters ausgeführt wird. Dabei wird ein zu aktualisierendes Tupel in *ApprovedOrders* über den in v_k vorliegenden Primärschlüssel $ItemID$ identifiziert. Die dort hinterlegte Bestellmenge (*ItemQuantity*) und der Gesamtpreis der Bestellung (*TotalPrice*) werden entsprechend dem in v_l gegebenen Wert angepasst.

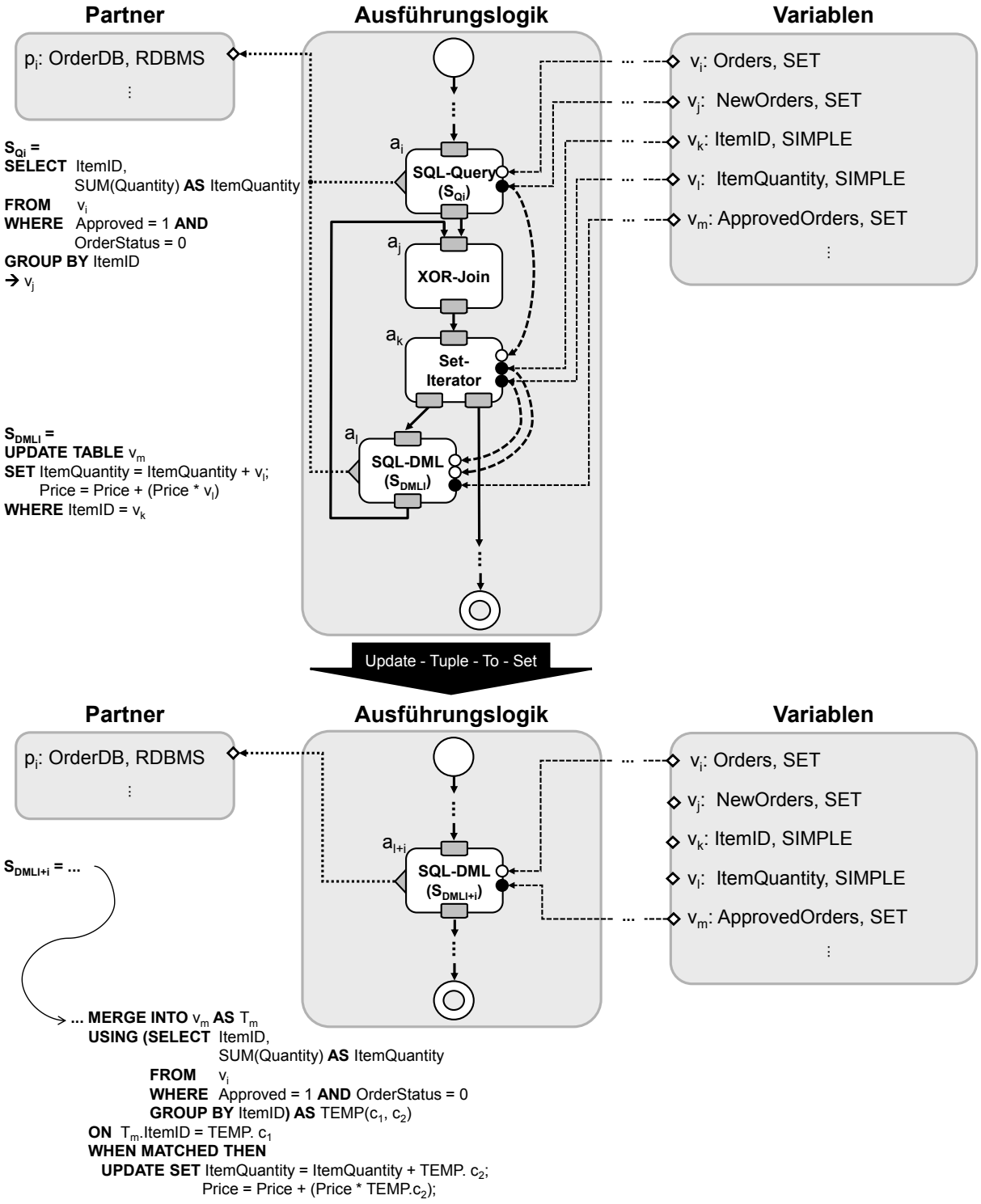


Abbildung 5.14.: Anwendung der *Update-Tuple-To-Set*-Regel

Die tupelbasierte Ausführung von S_{DML_l} lässt sich mithilfe einer Merge-Anweisung in die mengenbasierte Aktualisierungsoperation $S_{DML_{l+i}}$, die in der unteren Hälfte von Abbildung 5.14 dargestellt ist, überführen. Dazu wird die SQL-Anfrage S_{Q_i} in die Using-Klausel der Merge-Anweisung verschoben. In der On-Klausel werden für jedes Tupel aus der Ergebnismenge von S_{Q_i} mithilfe des Primärschlüssels *ItemID* passende Verbundpartner in der Tabelle *ApprovedOrders* gesucht. Existiert ein solcher Verbundpartner, wird die When-Matched-Klausel ausgeführt und das vorliegende Tupel aus *ApprovedOrders* aktualisiert. Dabei führt die When-Matched-Klausel die ursprüngliche Set-Klausel S_{DML_l} aus.

Korrektheit

Die Korrektheit des Transformationsschrittes lässt sich daraus folgern, dass die Aktualisierung der Tupel in *ApprovedOrders* in der Merge-Anweisung $S_{DML_{l+i}}$ entsprechend dem ursprünglichen prozeduralen Datenverarbeitungsmuster erfolgt. Zunächst wird die SQL-Anfrage S_{Q_i} in der Using-Klausel der Merge-Anweisung ausgeführt. Anschließend wird für jedes Tupel aus der Ergebnismenge von S_{Q_i} über den Primärschlüssel $ItemID$ ein entsprechender Verbundpartner in der Tabelle $ApprovedOrders$ gesucht. Damit wird dieselbe Tupelmenge aus *ApprovedOrders*, die schon zuvor durch das entsprechende prozedurale Datenverarbeitungsmuster bestimmt worden ist, identifiziert. Schließlich wird auf dieselbe Tupelmenge auch dieselbe Aktualisierungsoperation, die in der Set-Klausel der DML-Anweisung S_{DML_l} definiert ist, ausgeführt. Im Falle der Merge-Anweisung ist diese Set-Klausel Teil der When-Matched-Klausel, die auf die identifizierten Verbundpartner in *ApprovedOrders* angewendet wird. Somit führt die Ausführung der Merge-Anweisung $S_{DML_{l+i}}$ in der Tabelle *ApprovedOrders* zum gleichen Ergebnis wie zuvor die Ausführung des äquivalenten, prozeduralen Datenverarbeitungsmusters.

5.5.5. Delete-Tuple-To-Set

In diesem Abschnitt wird diskutiert, wie eine tupelbasierte Ausführung einer Delete-Anweisung in eine äquivalente mengenbasierte Ausführung transformiert werden kann.

Anwendungsbeispiel

Das in Abbildung 5.15 dargestellte prozedurale Datenverarbeitungsmuster erfüllt alle Voraussetzungen für eine *Delete-Tuple-To-Set*-Regel (siehe Regeldefinition D.16).

In diesem Beispielszenario wird die Tabelle *PremiumCustomers* aktualisiert, die alle Kunden, deren Bestellungen einen bestimmten Bestellwert überschreiten, auflistet. Nun sollen in dieser Tabelle nur noch Kunden berücksichtigt werden, die eine bestimmte Gruppe von Waren bestellt haben. Dagegen sollen Kunden, welche diese Anforderungen nicht erfüllen, aus *PremiumCustomers* entfernt werden. Hierzu werden mithilfe der SQL-Anfrage S_{Q_i} zunächst alle Kunden in der Tabelle *Orders* bestimmt, deren Bestellwert für die relevante Warengruppe (*ItemID* `BETWEEN` 1 `AND` 5000) einen bestimmten Schwellwert unterschreiten, der in der Variablen *TotalPrice* eingelesen wird. Die ermittelte Kundenmenge wird

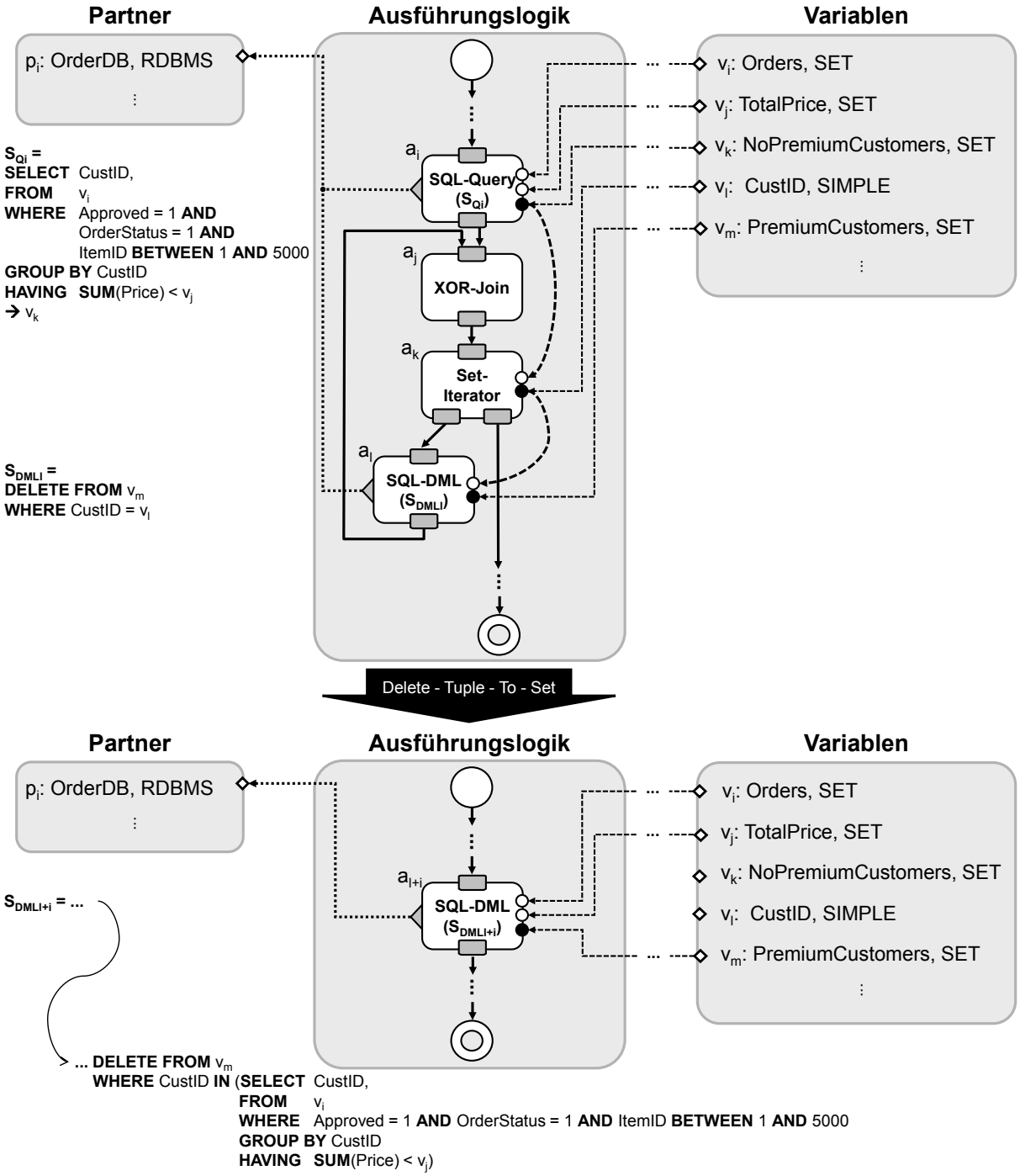


Abbildung 5.15.: Anwendung der *Delete-Tuple-To-Set*-Regel

in der SET-Variablen *NoPremiumCustomers* (v_k) abgespeichert. Anschließend wird über v_k iteriert, und die darin enthaltenen Primärschlüsselwerte werden in jeder Iteration in der Variablen *CustID* (v_l) zur Verfügung gestellt. Der in der Variablen v_l abgespeicherte Primärschlüsselwert dient als Eingabeparameter für die im Schleifenrumpf des LOOP-

Kontrollflussmusters ausgeführte Delete-Anweisung S_{DML_l}, welcher einen aus der Tabelle *PremiumCustomer* zu entfernenden Eintrag identifiziert.

Die tupelbasierte Delete-Anweisung kann mithilfe eines `IN`-Prädikates in eine mengenbasierte Delete-Anweisung $S_{DML_{l+i}}$ umgeschrieben werden, wie sie in der unteren Hälfte in Abbildung 5.15 dargestellt ist. Dazu wird der Schlüsselvergleich in der Where-Klausel von S_{DML_l}, in das die SQL-Anfrage S_{Q_i} verschoben wird, durch ein entsprechendes `IN`-Prädikat ersetzt. Dabei wird für jedes Tupel in *ApprovedOrders* überprüft, ob sich der zugehörige Primärschlüssel $CustID$ auch in der von S_{Q_i} ermittelten Ergebnismenge befindet. Im positiven Fall wird das Tupel aus *ApprovedOrders* entfernt. Sonst bleibt das betrachtete Tupel in der Tabelle erhalten.

Korrektheit

Vor der Transformation entfernt S_{DML_l} genau jene Tupel aus *PremiumCustomers*, deren Primärschlüsselwerte in der Ergebnismenge S_{Q_i} enthalten sind. Nach der Regelanwendung wird dies in $S_{DML_{l+i}}$, in das S_{Q_i} verschoben wurde, durch das `IN`-Prädikat sichergestellt. Allerdings ist zu berücksichtigen, dass sich die Auswertungsstrategie von $S_{DML_{l+i}}$ im Vergleich zum ursprünglichen Datenverarbeitungsmuster unterscheidet. Wird im ursprünglichen Datenverarbeitungsmuster noch für jeden Primärschlüsselwert in der Ergebnismenge von S_{Q_i} in S_{DML_l} ein passender Verbundpartner in der Tabelle *PremiumCustomers* gesucht, wird umgekehrt in $S_{DML_{l+i}}$ für jedes Tupel in *PremiumCustomers* ein passender Verbundpartner in der Ergebnismenge von S_{Q_i} ermittelt. Nichtsdestotrotz führen beide Auswertungsstrategien zum gleichen Ergebnis: In beiden Fällen werden genau jene Tupel aus *PremiumCustomers* entfernt, deren Primärschlüsselwerte sich in der Ergebnismenge von S_{Q_i} befinden.

5.6. Weitere heuristische Optimierungsstrategien

Das folgende Teilkapitel gibt einen Überblick über weitere heuristische Optimierungsstrategien, die auf eine PGM-Repräsentation gewinnbringend angewendet werden können.

5.6.1. Parallelisierung zwischen SQL-Anweisungen

Eine parallele Modellierung von SQL-Aktivitäten kann zu einer beschleunigten Ausführung eines datenintensiven Workflows führen, wenn sowohl auf der Workflow- als auch auf der Datenebene eine parallele Verarbeitung zwischen den SQL-Aktivitäten bzw. zwischen den darin definierten SQL-Anweisungen unterstützt wird.

Voraussetzung für eine parallele Modellierung von SQL-Aktivitäten ist die Unterstützung eines entsprechenden Kontrollflusskonstruktes in einer zu Grunde liegenden Workflowbeschreibungssprache, das in einer PGM-Repräsentation durch ein `PAR`-Kontrollflussmuster dargestellt werden kann. Beispielsweise kann in BPEL/SQL hierfür eine <`flow`> Aktivität

verwendet werden, innerhalb derer Aktivitäten (ohne Verwendung von <links>) parallel angeordnet werden können.

Bei einer parallelen Modellierung von SQL-Aktivitäten sind jedoch Präzedenzabhängigkeiten zwischen den SQL-Aktivitäten zu beachten, um sicherzustellen, dass nur unabhängige SQL-Aktivitäten parallel zueinander ausgeführt werden. Diese Präzedenzabhängigkeiten können in einer PGM-Repräsentation aufgrund der darin explizit modellierten Kontrollfluss-, Daten- und Kommunikationsabhängigkeiten leicht ermittelt werden.

Daraus ergibt sich die Möglichkeit, in einer PGM-Repräsentation zu überprüfen, ob eine sequentielle Modellierung von SQL-Aktivitäten in eine entsprechende parallele Modellierung überführt werden kann. Dabei ist es das Ziel, eine Sequenz von SQL-Aktivitäten in eine Ablauffolge zu transformieren, welche die maximal mögliche Inter-Query-Parallelität zwischen den SQL-Aktivitäten bzw. den darin definierten SQL-Anweisungen ausnutzen kann. Der hierdurch erzielbare maximale Parallelisierungsgrad (Speedup) ist durch die Anzahl der SQL-Aktivitäten in einer gegebenen Sequenz begrenzt.

Die Datenabhängigkeiten zwischen den SQL-Aktivitäten innerhalb einer Sequenz beeinflussen die Kontrollflussabhängigkeiten, die in der gesuchten Ablauffolge gelten müssen. Existiert beispielsweise in der ursprünglichen Sequenz zwischen zwei Aktivitäten a_i und a_j eine Datenabhängigkeit basierend auf einer Variablen v, ist eine Hintereinanderausführung von a_i und a_j auch in der parallelisierten Ablauffolge zwingend erforderlich, um die ursprüngliche Referenzierungsreihenfolge auf v und damit die darauf basierende Datenabhängigkeit zu erhalten. Eine Parallelisierung von a_i und a_j würde in diesem Fall aufgrund des parallelen Zugriffs auf v zu einer Anomalie führen, was eine Änderung der ursprünglichen Datenabhängigkeit zwischen beiden Aktivitäten zur Folge hätte. Deshalb ist eine parallele Ausführung von a_i und a_j nur dann korrekt möglich, wenn zwischen beiden Aktivitäten keine Datenabhängigkeit existiert.

Somit kann aus den vorliegenden Datenabhängigkeiten in einer PGM-Repräsentation die gesuchte Ablauffolge direkt abgeleitet werden: Dabei werden alle Aktivitäten, die parallel ausgeführt werden können, jeweils auf parallelen Kontrollflusspfaden in einem `PAR`-Kontrollflussmuster definiert. Sonst ist eine sequentielle Anordnung der Aktivitäten in der Ablauffolge erforderlich. Auf diese Weise entsteht im Allgemeinen ein `SEQ`-Kontrollflussmuster, in dem für alle parallelisierbaren Aktivitätsmengen ein entsprechendes `PAR`-Kontrollflussmuster vorhanden ist.

Im besten Fall können durch dieses Verfahren alle SQL-Aktivitäten einer Sequenz parallel modelliert werden, sodass das ursprüngliche `SEQ`-Kontrollflussmuster durch ein einzelnes `PAR`-Kontrollflussmuster ersetzt werden kann. Im schlechtesten Fall bleibt das ursprüngliche `SEQ`-Kontrollflussmuster unverändert erhalten.

Das Verfahren lässt sich auch auf andere sequentiell ausgeführte Aktivitätstypen bzw. Kontrollflussmuster erweitern, da die hierfür notwendigen Kontroll- und Datenflussabhängigkeiten in einer PGM-Repräsentation explizit dargestellt werden.

Durch eine parallele Modellierung von SQL-Aktivitäten bzw. der darin definierten SQL-Anweisungen entsteht eine Inter-Query-Parallelität. Diese kann auf der Datenebene besonders gewinnbringend genutzt werden, wenn die einzelnen parallel modellier-

ten SQL-Anweisungen jeweils auf verschiedenen Datenbanksystemen ausgeführt werden. Somit kann eine parallele Modellierung von SQL-Anweisungen vor allem bei mehreren Datenbanksystemen zu erheblichen Laufzeitgewinnen führen. Im Falle eines einzelnen ausführenden Datenbanksystems müssen jedoch trotz einer parallelen Modellierung auf der Workflowebene Verzögerungen auf der Datenebene in Kauf genommen werden, da wegen des gemeinsamen Transaktionskontextes die einzelnen SQL-Anweisungen typischerweise sequentiell über eine einzelne Datenbankverbindung von der Workflow- auf die Datenebene übertragen werden können. Dadurch wird eine weitere parallele Verarbeitung unabhängig von der zu Grunde gelegten Architektur eines Datenbanksystems erschwert. Diese Einschränkung gilt auch für föderierte Datenbanksysteme.

Die Diskussion zeigt, dass eine parallele Modellierung von SQL-Anweisungen in einem Workflowmodell alleine noch nicht ausreicht, um die Laufzeit eines datenintensiven Workflows zu reduzieren. Vielmehr hängt der Optimierungsgewinn auch unmittelbar von der Organisation der Datenebene und der Fähigkeit eines Workflowsystems ab, Aktivitäten parallel ausführen zu können.

Da die Anwendung der Parallelisierungsregel normalerweise nicht zu einer Laufzeitverschlechterung führt, ist sie für einen heuristischen Optimierungsansatz geeignet und sollte daher angewendet werden, um das Optimierungspotential einer PGM-Repräsentation voll ausnutzen zu können. Im besten Fall werden die SQL-Anweisungen parallel an die Datenebene übertragen und dort parallel verarbeitet. Dadurch lässt sich die Laufzeit einer Sequenz von n SQL-Anweisungen um einen Faktor n reduzieren. Im schlechtesten Fall erfolgt die Verarbeitung auf beiden Ebenen weiterhin sequentiell, und es ist kein merklicher Laufzeiteffekt erkennbar.

5.6.2. Verlagerung von SQL-Anweisungen auf die Datenebene

Das Verlagern von SQL-Anweisungen auf die Datenebene mithilfe von Stored-Procedures ist eine weitere Möglichkeit zur effektiven Reduzierung der Laufzeit eines datenintensiven Workflows. Hierdurch lässt sich die Anzahl der auszuführenden SQL-Aktivitäten in einem datenintensiven Workflow verringern. Musste zuvor für jede SQL-Anweisung eine entsprechende SQL-Aktivität in einem Workflowmodell definiert und ausgeführt werden, genügt nach der Auslagerung der SQL-Anweisungen eine einzelne *SQL-SP*-Aktivität zum Aufruf der zugehörigen Stored-Procedure. Damit lassen sich abhängig von der Anzahl der ausgelagerten SQL-Anweisungen die Übertragungskosten zwischen einem Workflow- und einem Datenbanksystem in einem hohen Maße senken. Zudem beschleunigt sich die Ausführung einer ausgelagerten SQL-Anweisung in einer Stored-Procedure im Vergleich zu ihrer ursprünglichen Einzelausführung. Bei einer Einzelausführung wird eine SQL-Anweisung an das zugehörige Datenbanksystem übertragen, dort übersetzt und ausgeführt. Dagegen findet bereits zur Übersetzungszeit einer Stored-Procedure eine Vorübersetzung aller darin enthaltenen SQL-Anweisungen statt. Der hieraus resultierende Programmcode kann somit unmittelbar beim Aufruf einer Stored-Procedure ausgeführt werden.

Alle Konstrukte eines Workflowmodells, die auch von SQL/PSM unterstützt werden, können grundsätzlich in eine Stored-Procedure ausgelagert werden. Hierzu zählen ne-

ben SQL-Anweisungen Kontrollflusskonstrukte wie Sequenzen, Schleifen oder alternative Kontrollflusspfade sowie Variablenzuweisungen. Bei einer korrekten Transformation der Konstrukte ist sichergestellt, dass die ursprünglichen Kontroll- und Datenabhängigkeiten eines Workflowmodells in eine Stored-Procedure korrekt überführt werden können. Alle hierfür notwendigen Informationen zur Erzeugung einer Stored-Procedure können aus einer PGM-Repräsentation direkt abgeleitet werden. Da es sich hierbei lediglich um eine Überführung von Sprachkonstrukten von einer Quell- in eine Zielsprache handelt, wird diese Diskussion an dieser Stelle nicht weiter vertieft.

In Kombination mit den in dieser Arbeit vorgestellten heuristischen Restrukturierungsregeln bietet sich ein *Stored-Procedure-Pushdown* insbesondere in Situationen an, in denen zuvor Restrukturierungsregeln der Klassen *Activity-Merging* bzw. *Tuple-To-Set* nicht erfolgreich angewendet werden konnten. Beispielsweise lassen sich Sequenzen von SQL-Anweisungen, die nicht weiter vereinfacht bzw. zusammengefasst werden können, in eine Stored-Procedure verlagern. Ebenso ist es sinnvoll, Schleifen über materialisierten Datenmengen, die nicht über eine *Tuple-To-Set*-Regel aufgelöst werden können, mittels einer Stored-Procedure zu definieren, um die Laufzeit eines solchen ineffizienten Datenverarbeitungsmusters zu reduzieren. Auf diese Weise kann außerdem die Übertragung und Materialisierung der zu iterierenden Datenmenge an ein Workflowsystem verhindert werden, was zu weiteren Kosteneinsparungen führt.

Um eine *Stored-Procedure-Pushdown*-Regel korrekt anwenden zu können, müssen mehrere Bedingungen erfüllt sein:

- Beispielsweise müssen alle auszulagernden SQL-Anweisungen auf demselben Datenbanksystem ausgeführt werden. Voraussetzung hierfür ist, dass ein solches Datenbanksystem die Ausführung von Stored-Procedures unterstützt, was in einer PGM-Repräsentation durch die Eigenschaft `[SP]` eines Partners vom Typ `RDBMS` explizit angezeigt wird (siehe Definition 4 in Anhang A). Dabei kann es sich sowohl um ein zentralisiertes als auch um ein föderiertes Datenbanksystem handeln.
- Zudem muss ein solches Datenbanksystem es einem Benutzer bzw. dem PGM/F-System erlauben, eine generierte Stored-Procedure zu installieren, um die Ausführbarkeit der optimierten SQL-Anweisungen sicherzustellen.
- Bei der Generierung von Stored-Procedures müssen herstellerspezifische Dialekte von SQL/PSM berücksichtigt werden, um die Ausführbarkeit einer generierten Stored-Procedure garantieren zu können.
- Können Informationen über einen Datenbanksystemtypen direkt aus einem Workflowmodell abgeleitet werden, kann ein Stored-Procedure-Pushdown automatisch zur Modellierungszeit eines Workflowmodells durchgeführt werden. Sonst müssen die Informationen dem PGM/F-System durch einen Benutzer explizit zur Verfügung gestellt werden. Alternativ können die Informationen aus dem Deploymentdeskriptor eines Workflowmodells gewonnen werden. Dann ist ein Stored-Procedure-Pushdown erst zum Deploymentzeitpunkt eines Workflowmodells möglich.
- Ebenso muss bei der Auslagerung von SQL-Anweisungen darauf geachtet werden, dass alle Datenabhängigkeiten zwischen den Anweisungen und den verbliebenen

Aktivitäten im Workflow erhalten bleiben. Dies bedeutet, dass alle Ergebnisse, die in einer Stored-Procedure berechnet und in einem Workflow referenziert werden, als Ausgabeparameter einer Stored-Procedure deklariert werden müssen.

Schließlich muss beachtet werden, dass die Ausführungskontrolle der ausgelagerten SQL-Anweisungen von einem Workflow- auf ein Datenbanksystem übertragen wird, was bei einer späteren Analyse eines Workflows zu berücksichtigen ist.

5.6.3. Vereinfachung von SQL-Anweisungen

Durch eine Reduzierung der Größe von Anfrageergebnismengen, die von der Daten- auf die Workflowebene übertragen werden, lassen sich die Übertragungs- und Materialisierungskosten in einem datenintensiven Workflow wirkungsvoll reduzieren. Deshalb sollte eine Ergebnismenge keine Daten enthalten, die bereits in einem vorausgegangenen Verarbeitungsschritt an das Workflowsystem gesendet wurden. Wie in Teilkapitel 5.4.6 diskutiert, kann dies durch Anwendung einer heuristischen *CGO*-Regel erreicht werden, bei der ähnlich strukturierte Anfragen, die sich beispielsweise nur in einer Select- oder Where-Klausel unterscheiden, verschmolzen werden. Auf diese Weise lassen sich redundante Anfragen aus der Ausführungslogik einer PGM-Repräsentation entfernen. Ebenso sollte eine Ergebnismenge keine unnötigen Daten enthalten, d.h. Duplikate bzw. Daten, die in keinem folgenden Verarbeitungsschritt mehr auf der Workflowebene referenziert werden. Dies wird durch Anwendung der heuristischen Regeln *Eliminate-Unused-Attributes* bzw. *Eliminate-Redundant-Attributes* der Regelklasse *SQL-Simplification* sichergestellt.

Dabei entfernt die *Eliminate-Unused-Attribute*-Regel Attribute aus einer Select-Klausel einer Anfrage, die in der folgenden Verarbeitung nicht mehr berücksichtigt werden. Dadurch müssen entsprechend weniger Tupelwerte in die Ergebnismenge einer Anfrage aufgenommen werden, wodurch das an die Workflowebene zu übertragende Datenvolumen reduziert werden kann. Ebenso sinkt der Berechnungsaufwand zur Ermittlung der Ergebnismenge mit der Folge eines weiteren positiven Laufzeiteffekts.

Solche ungenutzten Attribute einer Ergebnismenge lassen sich in einer PGM-Repräsentation sehr einfach ermitteln. Es muss lediglich überprüft werden, ob Datenabhängigkeiten zu anderen Aktivitäten existieren, die auf Variablen basieren, welche in PGM die Attribute einer Ergebnismenge repräsentieren.

Das Ziel der *Eliminate-Redundant-Attributes*-Regel ist es, Attribute in einer Select-Klausel einer Anfrage, die darin doppelt aufgelistet werden, zu entfernen. Eine solche Auflistung kann beispielsweise Folge der Anwendung einer *Merge-Select*-Regel sein (*CGO*-Regel), bei der die Attributlisten der Select-Klauseln zweier Anfragen miteinander verschmolzen werden. Die bei dieser Regel zu erwartenden Einsparungen hängen zum einen davon ab, wie viele Redundanzen in einer solchen Attributliste vorhanden sind, und zum anderen, welche Datenmengen von einem redundanten Attribut selektiert werden. Sonst entsprechen die Optimierungseffekte auf der Datenebene den Optimierungseffekten der *Eliminate-Unused-Attributes*-Regel.

Die *Eliminate-Redundant-Predicates*-Regel entfernt Redundanzen aus einem Selektionsprädikat, die beispielsweise durch die Anwendung der *Merge-Where*-Regel (*CGO*-Regel) entstehen können. Die Idempotenzregel für Boolesche Ausdrücke bietet ein effektives Hilfsmittel, um solche Redundanzen zu eliminieren [HR01]. Allerdings ist eine solche Vereinfachung eines Selektionsprädikates einer Anfrage bereits Teil der Anfrageoptimierung in einem Datenbanksystem. Trotzdem ist es sinnvoll, diese Vereinfachung im Rahmen der PGM-Optimierung durchzuführen. Hierdurch kann ein kleiner Teil sowohl der Übertragungskosten für die Anfrage als auch deren Ausführungskosten während der Anfrageoptimierung eingespart werden. Für den Fall, dass eine leere Ergebnismenge erkannt wurde (das Selektionsprädikat entspricht in diesem Fall der Konstanten `FALSE`), erübrigt sich zudem das Versenden der Anfrage an das Datenbanksystem, da die zugehörige SQL-Aktivität aus der PGM-Repräsentation entfernt werden kann.

5.7. Kostenbasierte Optimierungsstrategien

Dieses Kapitel gibt einen Überblick über weitere Optimierungsstrategien, die nur im Rahmen einer kostenbasierten Optimierung auf eine PGM-Repräsentation angewendet werden können. Da die kostenbasierte Optimierung nicht im Fokus dieser Arbeit steht, werden die Optimierungsstrategien hier nur kurz skizziert, um die Diskussion über geeignete Restrukturierungsregeln für datenintensive Workflows abzurunden.

5.7.1. Update-Merging mit Merge-Anweisungen

Mit einer Merge-Anweisung lassen sich die in Teilkapitel 5.4.5 betrachteten Möglichkeiten zur Verschmelzung von Aktualisierungsoperationen erweitern. Eine Merge-Anweisung erlaubt die Kombination einer Insert- mit einer Update-Operation, die beide auf derselben Tabelle ausgeführt werden. Andere Kombinationsmöglichkeiten, insbesondere unter Berücksichtigung von Delete- oder mehreren Update-Operationen, werden dagegen vom SQL:2003-Standard nicht unterstützt.

Diese Einschränkungen gelten beispielsweise nicht für eine DB2-spezifische Merge-Anweisung, welche neben der Definition von Insert- und Update- auch Delete-Operationen erlaubt. Ferner können mehrere solcher Operationen des gleichen Typs definiert werden. Mit diesen sprachlichen Mitteln erweitern sich die Möglichkeiten zur Kombination von Aktualisierungsoperationen im Vergleich zum SQL-Standard erheblich. Insbesondere lassen sich damit verschiedenste Kombinationsmöglichkeiten zwischen Insert-, Update-, Delete- und Merge-Operationen ausdrücken. Hierdurch lässt sich das Optimierungspotential einer PGM-Repräsentation wesentlich effektiver nutzen.

Nichtsdestotrotz beschränkt sich die folgende Diskussion auf die Mittel, die vom SQL-Standard zur Definition einer Merge-Anweisung zur Verfügung gestellt werden. Insbesondere ist hier zu beachten, dass Insert- und Update-Operationen nicht beliebig miteinander kombiniert werden können. Entscheidend hierfür sind die Datenmengen, die von den Selektionsprädikaten der beiden Aktualisierungsoperationen definiert werden. Hier-

bei kann es sich sowohl um disjunkte als auch um überlappende Datenmengen handeln. Die Ausführungssemantik einer Merge-Anweisung erzwingt die Disjunktheit der beiden Datenmengen, da ein Tupel nicht in eine Tabelle eingefügt und dabei gleichzeitig aktualisiert werden kann. Auf diese Weise wird sichergestellt, dass sich die Effekte beider Aktualisierungsoperationen auf einer Tabelle vor und nach ihrer Kombination zu einer Merge-Anweisung nicht unterscheiden.

In [EGLT76] wurde gezeigt, dass es im Allgemeinen unentscheidbar ist, ob zwei beliebige Prädikate disjunkte Mengen beschreiben. Eine Entscheidbarkeit ist aber mit restriktiven Beschränkungen zu erreichen. So gibt es eine Reihe von Entscheidungsverfahren [EGLT76, WE77, HR79, RHI80, LY85, GSW96], die auf einfachen Klassen von Prädikaten angewendet werden können. Das typische polynomielle Laufzeitverhalten dieser Verfahren kann bei der PGM/F-Optimierung akzeptiert werden, da die Optimierung zur Modellierungszeit und nicht zur Laufzeit eines datenintensiven Workflows durchgeführt wird.

Die Kombination einer Insert- und Update-Operation zu einer Merge-Anweisung soll an dem in Abbildung 5.16 dargestellten Datenverarbeitungsmuster veranschaulicht werden. Darin enthalten sind zwei *SQL-DML*-Aktivitäten a_i und a_j, die nacheinander zwei DML-Anweisungen S_{DML_i} und S_{DML_j} auf der Tabelle *ItemSuppliers* ausführen.

Die Insert-Anweisung S_{DML_i} fügt einen neuen Lieferanten ($SupID = 456$) eines bestimmten Produktes ($ItemID = 2546$) in die Tabelle $ItemSuppliers$ ein. Ist keine geeignete Zugriffspfadstruktur über dem Primärschlüssel definiert, ist ein Scan über $ItemSuppliers$ erforderlich, um zu überprüfen, ob sich das einzufügende Tupel bereits in dieser Tabelle befindet. Dies ist notwendig, um das Einfügen von Duplikaten zu verhindern.

Anschließend reduziert die Update-Anweisung S_{DML_j} den Preis aller Produkte eines bestimmten Lieferanten ($SupID = 346$) in $ItemSuppliers$, die nicht innerhalb von fünf Tagen geliefert werden können, um 5%. Dabei werden die zu aktualisierenden Tupel entweder mithilfe eines Scans auf der Tabelle $ItemSuppliers$ oder einer entsprechenden Zugriffspfadstruktur referenziert und anschließend aktualisiert.

Da sich in diesem Beispiel die zu aktualisierenden Tupel auf unterschiedliche Lieferanten beziehen, ist die Disjunktheit beider Aktualisierungsoperationen sichergestellt. Somit lassen sich S_{DML_i} und S_{DML_j} durch eine Merge-Anweisung zu $S_{DML_{j+i}}$ kombinieren, wie es in der unteren Hälfte von Abbildung 5.16 dargestellt ist.

In $S_{DML_{j+i}}$ wird zunächst in der Using-Klausel die zu aktualisierende Tupelmenge der Tabelle $ItemSuppliers$ basierend auf S_{DML_i} und S_{DML_j} explizit bestimmt. Dazu werden die ermittelten Insert- bzw. Update-Mengen durch eine `UNION-ALL`-Operation vereinigt. Anschließend wird in der On-Klausel über einen Primärschlüsselvergleich über $SupID$ und $ItemID$ ermittelt, ob ein in der Using-Klausel ermitteltes Tupel zur Insert- oder zur Update-Menge der Anweisung gehört. Im ersten Fall kann das Tupel direkt eingefügt werden (When-Not-Matched-Klausel), im zweiten Fall ist das Tupel in der Tabelle (erneut) zu referenzieren und anschließend zu aktualisieren (When-Matched-Klausel).

Somit ergibt sich bei der Ausführung der Merge-Anweisung ein Mehraufwand im Vergleich zu der Ausführung der beiden Einzeloperationen wegen der wiederholten Referenzierung eines zu aktualisierenden Tupels in der Tabelle *ItemSuppliers* in der Using- und When-

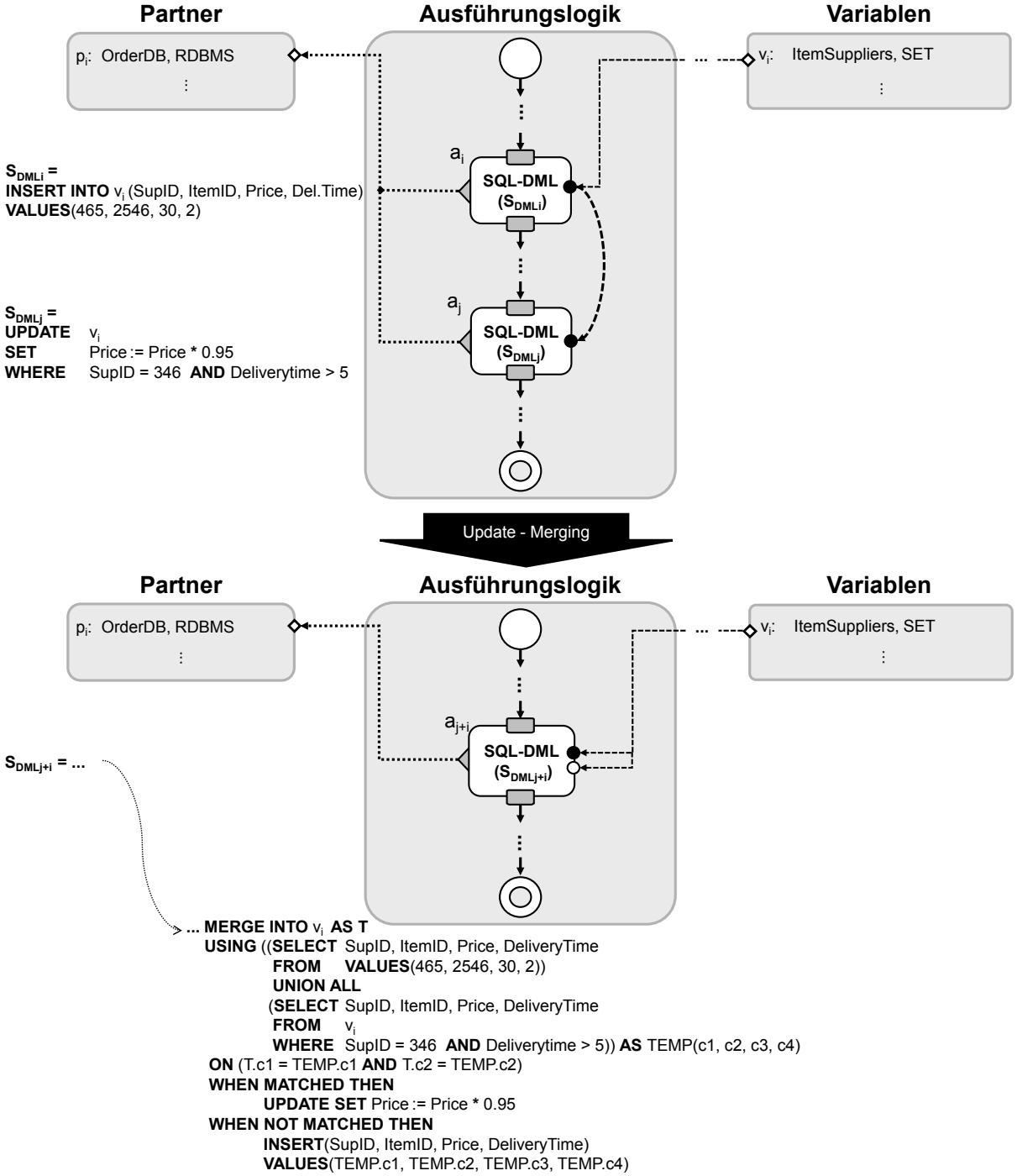


Abbildung 5.16.: Variante der *Update-Merging*-Regel mit Merge-Anweisung

Matched-Klausel. Wurden die Insert- und Update-Anweisungen S_{DML_i} und S_{DML_j} zuvor direkt auf der Tabelle *ItemSuppliers* ausgeführt, sind für die kombinierte Ausführung beider Operationen in $S_{DML_{j+i}}$ Zwischenberechnungen nötig. Zum einen muss die zu aktualisierende Datenmenge in *ItemSuppliers* vorab in der Using-Klausel der Merge-Anweisung explizit berechnet werden. Zum anderen muss die berechnete Datenmenge anschließend

tupelweise mit der aktuellen Tupelmenge in *ItemSuppliers* verglichen werden, um entscheiden zu können, ob die Insert- oder die Update-Operation darauf anzuwenden ist. Diese wiederholte Referenzierung der Tabelle *ItemSuppliers* wird auch an den beiden Dataslots der Aktivität a_{j+i} sichtbar. Aufgrund des schreibenden und lesenden Zugriffs auf *ItemSuppliers* existieren hier jeweils ein `Write`- bzw. `Read`-Dataslot. Diese sind mit der Variablen v_i, welche die Tabelle *ItemSuppliers* repräsentiert, assoziiert.

Dieser Mehraufwand fällt umso stärker ins Gewicht, je größer die zu aktualisierende Datenmenge ist. Allerdings muss der Mehraufwand auf der Datenebene immer mit der Leistungssteigerung auf der Workflowebene, die durch den Wegfall der Ausführung einer SQL-Aktivität erzielt werden kann, verrechnet werden. Deshalb kann die Kombination von Aktualisierungsoperationen mittels Merge-Anweisungen abhängig von den Charakteristika der zu Grunde gelegten Tabellen sowohl zu Laufzeitverbesserungen als auch zu Laufzeitverschlechterungen führen. Aus diesem Grund ist diese Technik für einen heuristischen Optimierungsansatz nicht geeignet. Vielmehr müssen die Kosten einer solchen Kombination mithilfe entsprechender Statistiken vorab berechnet werden, um sicherzustellen, dass die Optimierungsentscheidung zu einer Laufzeitverbesserung führt.

5.7.2. Optimierung von Verbundoperationen

Eine Verbundberechnung zwischen zwei oder mehreren Datenmengen ist eine zentrale Datenverarbeitungsoperation. Zur Definition einer Verbundberechnung stehen in einem datenintensiven Workflow verschiedene Möglichkeiten zur Verfügung:

In der ersten Variante wird ein Verbund in einer SQL-Anfrage, die von einer SQL-Aktivität an das zu Grunde gelegte Datenbanksystem gesendet wird, deklarativ definiert. Dort erfolgt im Rahmen der Anfrageoptimierung die Umsetzung der Verbunddeklaration in eine entsprechende Implementierung. Typischerweise stehen hierfür verschiedene Implementierungen, wie z.B. Nested-Loop, Sort-Merge oder Hash-Join [HR01], zur Verfügung. Zur Bestimmung einer geeigneten Verbundimplementierung verwendet ein Anfrageoptimierer eine Kostenfunktion, die auf Statistiken beruht, wie z.B. den Größen der zu kombinierenden Relationen oder der Verbundselektivität. Mithilfe dieser Kostenfunktion wird diejenige Verbundimplementierung, bei deren Ausführung die geringsten Kosten zu erwarten sind, ausgewählt. Im Anschluss an die Berechnung wird das Verbundergebnis an das Workflowsystem zurückgesendet und dort materialisiert. Die materialisierte Datenmenge kann lokal auf der Workflowebene weiterverarbeitet werden. Abbildung 5.17 stellt das zugehörige Datenverarbeitungsmuster dieser Verbundvariante dar, das in einem BPEL/SQL-Workflow mithilfe einer <`retrieveSet`> Aktivität definiert werden kann.

Der Vorteil einer Verbundberechnung auf der Datenebene liegt zum einen darin, dass lediglich eine SQL-Aktivität auf der Workflowebene und eine SQL-Anfrage auf der Datenebene ausgeführt werden müssen. Zum anderen stehen in einem Datenbanksystem effiziente Verbundimplementierungen zur Verfügung. Allerdings muss abhängig von der Größe des Verbundergebnisses unter Umständen ein sehr großes Datenvolumen an das Workflowsystem übertragen werden. Somit ist dieser Ansatz vor allem bei einer hohen Verbundselektivität

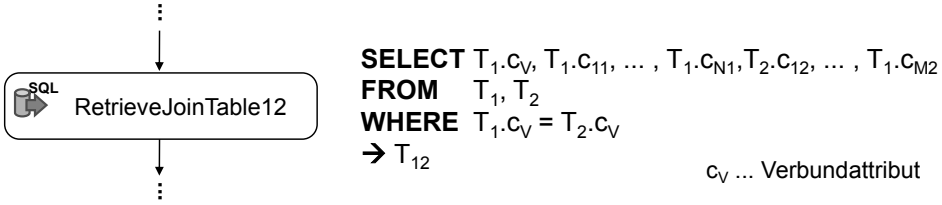


Abbildung 5.17.: Verbundberechnung in BPEL/SQL: 1. Variante

sehr effizient. Umgekehrt verursacht dieser Ansatz bei einer niedrigen Verbundselektivität und großen zu kombinierenden Relationen hohe Übertragungskosten.

Bei kleinen Relationen und niedriger Verbundselektivität kann es sinnvoll sein, eine Verbundberechnung auf der Workflowebene mithilfe entsprechender Datenverarbeitungsmuster durchzuführen. Die Datenverarbeitungsmuster können aus den Verbundimplementierungen auf der Datenebene direkt abgeleitet werden. So lässt sich etwa eine Nested-Loop-Implementierung in einem BPEL/SQL-Workflow direkt mithilfe entsprechender Schleifenkonstrukte darstellen, wie es in Abbildung 5.18 exemplarisch veranschaulicht ist. Um einen Verbund auf der Workflowebene berechnen zu können, müssen in einem ersten Schritt die zu kombinierenden Relationen vollständig an das Workflowsystem übertragen und dort materialisiert werden (*Ship-Whole-Ansatz* [Rah94]). Dieser Verarbeitungsschritt wird in den <`retrieveSet`> Aktivitäten $RetrieveTable_1$ und $RetrieveTable_2$ definiert und ausgeführt. Im Anschluss wird über die materialisierten Datenmengen T_{11} und T_{22} iteriert und dabei das gewünschte Verbundergebnis direkt auf der Workflowebene ermittelt. Dazu liefert die <`forEach`> Aktivität $ForEachTuple_1$ in jeder Iteration das nächste Tupel (t_{11}) von T_{11}. Für jedes gelieferte Tupel von T_{11} iteriert die zweite <`forEach`> Aktivität $ForEachTuple_2$ über die materialisierte Datenmenge T_{22}. Dabei wird für jedes gelieferte Tupel (t_{22}) von T_{22} überprüft, ob es sich um einen Verbundpartner von t_{11} handelt. Dazu werden in dem Beispiel in der <`if`> Aktivität $FoundMatch?$ die Verbundattribute (c_v) von t_{11} und t_{22} auf Gleichheit überprüft. Im positiven Fall können weitere Verarbeitungsschritte auf den Verbundpartnern erfolgen. Sonst wird das nächste Tupel aus T_{22} bzw. T_{11} betrachtet.

Eine optimierte Variante dieses Ansatzes besteht darin, zunächst nur die kleinere der beiden Relationen vollständig an das Workflowsystem zu übertragen. Danach werden die Verbundpartner aus der zweiten Relation angefordert, mit denen die Verbundberechnung auf der Workflowebene durchgeführt wird. Diese Vorgehensweise entspricht der Anwendung eines *Semi-Verbundes* [Rah94]. Das zugehörige Datenverarbeitungsmuster ähnelt der in Abbildung 5.18 dargestellten Variante. Allerdings erhält hier die materialisierte Datenmenge T_{22} im Vergleich zum Ship-Whole-Ansatz ausschließlich die Verbundpartner von T_{11}, die durch die <`retrieveSet`> Aktivität $RetrieveTable_2$ durch eine entsprechende SQL-Anweisung in den Workflow geladen werden. Dadurch steigen bei dieser Variante zunächst die Berechnungskosten zur Bestimmung der Verbundpartner in der zweiten Relation im Vergleich zum Ship-Whole-Ansatz. Bei einer hohen Verbundselektivität ist die Menge der angeforderten Verbundpartner wesentlich kleiner als die beim Ship-Whole-

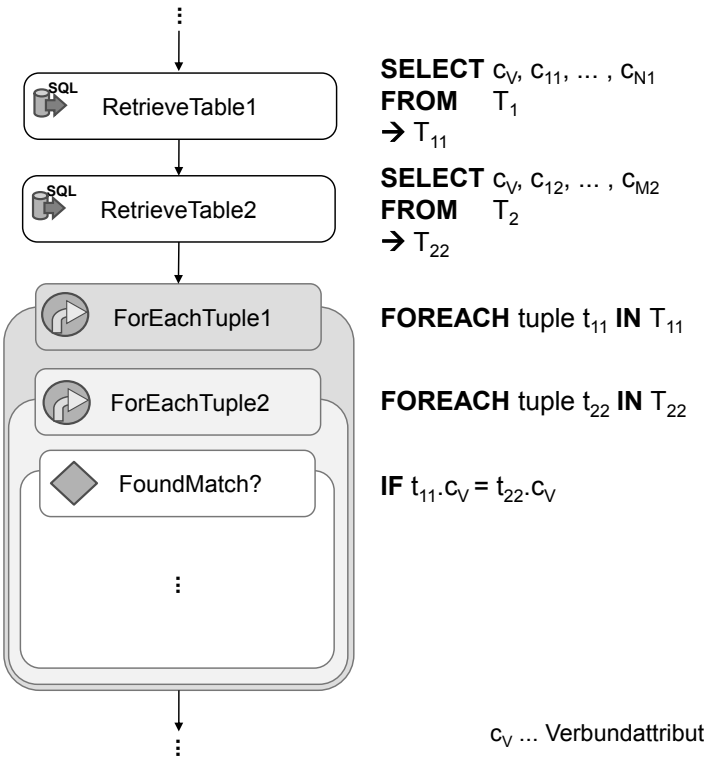


Abbildung 5.18.: Verbundberechnung in BPEL/SQL: 2. Variante

Ansatz zu übertragende ganze Relation, wodurch das zu übertragende Datenvolumen unter Umständen erheblich reduziert werden kann.

Aufgrund der alternativen Möglichkeiten für die Verbundberechnung und den damit verbundenen unterschiedlich hohen Ausführungs- und Übertragungskosten, ergibt sich ein entsprechend großes Optimierungspotential. Dies gilt bereits für einfache Verbünde zwischen zwei Relationen. Für Verbundberechnungen zwischen mehr als zwei Relationen ist zusätzlich eine günstige Ausführungsreihenfolge zu bestimmen.

Daraus folgt das Optimierungsproblem, auf welcher Ebene eine Verbundberechnung am effizientesten durchgeführt werden kann. Damit verknüpft ist eine Minimierung der Berechnungskosten für den Verbund und der Übertragungskosten für die hierfür benötigten bzw. resultierenden Datenmengen.

Lösungen für dieses Optimierungsproblem lassen sich beispielsweise aus dem Bereich der verteilten Anfrageoptimierung ableiten [Rah94]: Bei einer verteilten Verbundberechung wird mit einer geeigneten Kostenfunktion bestimmt, auf welchen Datenknoten eines verteilten Datenbanksystems in welcher Reihenfolge und mit welcher Strategie eine verteilte Verbundberechnung am kostengünstigsten durchgeführt werden kann. Diese Kostenfunktion basiert auf Statistiken, die unter anderem folgende Informationen umfassen: Kardinalität der beteiligten Relationen, Tupel- und Attributgrößen, Häufigkeitsverteilungen von Attributwerten bzw. Abschätzungen von Selektivitätsfaktoren.

Eine solche Kostenfunktion kann auch als Grundlage für die Kostenabschätzung einer Verbundberechnung in einem Workflowmodell herangezogen werden. Allerdings müssen

hier noch weitere Kosten mit einfließen, wie z.B. die Ausführungskosten für einzelne Aktivitäten und die Kosten zur Materialisierung von Datenmengen. Sollen bei einer Verbundberechnung auch Datenmengen berücksichtigt werden, die von datenintensiven Web-Services (vom Typ 2 oder 4) geliefert werden, wird eine Kostenabschätzung schwieriger, da die hierfür notwendigen Statistiken für einen Web-Service typischerweise nicht verfügbar sind. Eine mögliche Lösung dieses Problems besteht darin, das Laufzeitverhalten eines Web-Services zu beobachten und daraus die notwendigen Informationen abzuleiten.

Über eine solche Kostenfunktion können die verschiedenen Varianten einer Verbundberechnung in einem Workflowmodell verglichen und bei Bedarf ineinander überführt werden. Dabei ist eine Verbundberechnung innerhalb einer SQL-Anfrage in ein entsprechendes Datenverarbeitungsmuster auf der Workflowebene umzuwandeln und umgekehrt.

Zu beachten ist, dass es auch von der Organisation der Datenebene abhängt, welche Realisierungsvarianten für eine Verbundberechnung zur Verfügung stehen. Im Falle eines zentralisierten bzw. föderierten Datenbanksystems können alle drei Verbundvarianten berücksichtigt werden. Wird dagegen ein Verbund auf Tabellen, die von unterschiedlichen Datenbanksystemen verwaltet werden, berechnet, kann die Verbundberechnung ausschließlich auf der Workflowebene durchgeführt werden. Somit beschränkt sich hier die Auswahl auf den Ship-Whole- bzw. Semi-Join-Ansatz. Eine direkte Deklaration der Verbundberechnung in SQL ist dagegen in diesem Fall nicht möglich, um die Erzeugung einer nicht ausführbaren SQL-Anweisung zu verhindern.

Die gleichen Überlegungen gelten für die Berechnung von Mengenoperationen, die auf der Daten- und Workflowebene durchgeführt werden können. Da diese Berechnung auf der Verbundberechnung beruht [HR01], können die Ergebnisse für die Verbundberechnungen direkt auf die Berechnung von Mengenoperationen übertragen werden.

5.7.3. Optimierung gemeinsamer Teilausdrücke

In einem datenintensiven Workflow können Anfragen, die gemeinsame Teilausdrücke enthalten, definiert werden. Ohne entsprechende Optimierungsmaßnahmen werden die gemeinsamen Teilausdrücke in einem Datenbanksystem redundant ausgeführt, wodurch unter Umständen ein großes Optimierungspotential ungenutzt bleibt. Es ist effizienter, einen gemeinsamen Teilausdruck vorab in einem Datenbanksystem zu berechnen, um bei einer späteren erneuten Referenzierung dieses Teilausdruckes direkt auf das Berechnungsergebnis zugreifen zu können. Vor allem bei häufig referenzierten Teilausdrücken, die einen hohen Verarbeitungsaufwand aufweisen, sind hohe Optimierungsgewinne zu erzielen.

Zur Lösung dieses Optimierungsproblems kann auf wohlbekannte Verfahren aus dem Bereich der Multi-Query-Optimierung zurückgegriffen werden (siehe Teilkapitel 2.2.2). Bei diesen Verfahren werden zunächst die in einer Anfragemenge enthaltenen gemeinsamen Teilausdrücke identifiziert. Eine Materialisierung dieser Teilausdrücke mittels materialisierter Sichten kommt anschließend nur in Frage, wenn sichergestellt ist, dass hierdurch die Anfragemenge effizienter ausgeführt werden kann. Hierbei muss insbesondere berücksichtigt werden, dass der Inhalt einer Materialisierung bei jeder Änderungsoperation auf den zu Grunde liegenden Daten aktualisiert werden muss. Dies kann zu ei-

nem nicht vernachlässigbaren Mehraufwand führen. Somit ist die Frage einer effizienten Aktualisierung einer Materialisierung von zentraler Bedeutung, die jedoch nur mithilfe geeigneter Kostenfunktionen beantwortet werden kann.

Unterstützt ein Datenbanksystem eine solche Multi-Query-Optimierung, könnte eine heuristische Optimierungsstrategie in PGM/F wie folgt aussehen: Die in einer PGM-Repräsentation definierte Anfragemenge wird als Einheit an das zu optimierende Datenbanksystem übergeben. Hierfür muss das Datenbanksystem die notwendigen Schnittstellen bereitstellen. Der Anfrageoptimierer des Datenbanksystems identifiziert gemeinsame Teilausdrücke und definiert für diese entsprechende materialisierte Sichten. Wird die Anfragemenge anschließend ausgeführt, kann der Anfrageoptimierer prüfen, ob zu einzelnen Teilausdrücken einer Anfrage eine Materialisierung existiert oder nicht. Sofern eine solche existiert, kann für diesen Teilausdruck im Ausführungsplan ein einfacher Zugriff auf die Materialisierung vorgesehen und somit die schrittweise Berechnung des Teilausdrucks aus den Basisdaten vermieden werden.

Bietet dagegen ein Datenbanksystem keine Unterstützung für eine Multi-Query-Optimierung an, muss die Funktionalität vollständig von PGM/F realisiert werden. Insbesondere müssen dann geeignete Teilausdrücke mittels bekannter Multi-Query-Optimierungstechniken identifiziert und die zugehörigen Materialisierungen in einem Datenbanksystem definiert werden. Dies setzt einen Zugriff von PGM/F auf die hierfür notwendigen Statistiken und Datenbanksysteme voraus. Folglich ist eine solche Optimierung nur im Rahmen einer kostenbasierten Optimierung durchführbar.

5.8. Zusammenfassung

Die in diesem Kapitel vorgestellten Optimierungsstrategien können zu einer Laufzeitreduktion eines datenintensiven Workflows führen. Positive Optimierungseffekte können sowohl auf der Workflow- als auch auf der Datenebene erzielt werden. Die Tabelle 5.19 gibt einen Überblick darüber, welche Optimierungseffekte bei der Anwendung der vorgestellten Optimierungsstrategien zu erwarten sind, zu welchen Optimierungszeitpunkten die Optimierungsstrategien angewendet werden können, und welchen Einfluss die Organisation der Datenebene auf die zu erwartenden Optimierungseffekte hat.

- Die *Web-Service-Pushdown*-Regeln dienen zur Vorbereitung der Ausführung weiterer gewinnbringender Restrukturierungsregeln. Sie erhöhen das Optimierungspotential einer PGM-Repräsentation erheblich, da nach ihrer Regelanwendung Web-Service-Aufrufe bei nachfolgenden Optimierungsentscheidungen berücksichtigt werden können. Bei der Anwendung einer *Web-Service-Pushdown*-Regel ist zu beachten, dass ein ausführendes Datenbanksystem in der Lage sein muss, Web-Service-Aufrufe mittels WS-UDTFs auszuführen. Können diese Informationen direkt aus dem Workflowmodell abgeleitet werden, ist eine automatische Anwendung der *Web-Service-Pushdown*-Regel zur Modellierungszeit eines Workflowmodells möglich. Ebenso besteht die Möglichkeit, dass die Informationen dem PGM/F-System durch einen Benutzer explizit zur Verfügung gestellt werden. Alternativ können die Informa-

tionen aus dem Deploymentdeskriptor eines auszuführenden Workflowmodells gewonnen werden. In diesem Fall ist eine automatische Anwendung der *Web-Service-Pushdown*-Regel nur zum Deploymentzeitpunkt eines Workflowmodells möglich.

- Die *Activity-Merging*-Regeln kombinieren zwei Datenverarbeitungsoperationen, die beide auf demselben Datenbanksystem ausgeführt werden. Dadurch lässt sich die Anzahl der auszuführenden Aktivitäten in einer PGM-Repräsentation reduzieren. Handelt es sich bei beiden Datenverarbeitungsoperationen um SQL-Anweisungen, können zudem Übertragungskosten eingespart werden, da weniger Nachrichten und somit ein geringeres Datenvolumen zwischen Workflow- und Datenebene ausgetauscht werden müssen. Darüber hinaus lassen sich abhängig von der Art der zu kombinierenden Datenverarbeitungsoperationen weitere Optimierungseffekte auf der Datenebene erzielen: Beispielsweise verringern sich durch die Anwendung einer heuristischen *Update-Merging*-Regel die Ausführungskosten einer kombinierten Anweisung im Vergleich zu den Ausführungskosten der beiden einzelnen Anweisungen. Des Weiteren können durch einen solchen Kombinationsschritt Parallelisierungseffekte entstehen, die bei einer geeigneten Unterstützung durch ein ausführendes Datenbanksystem gewinnbringend genutzt werden können. Der Optimierungsgewinn der *Eliminate-Temporary-Table*-Regel liegt vor allem darin, dass Kosten für die Verwaltung einer temporären Tabelle sowohl auf der Workflow- als auch auf der Datenebene eingespart werden können. Hierzu zählen neben den Ausführungs- auch Übertragungskosten zur Übermittlung der hierfür notwendigen SQL-Anweisungen. Es ist zu beachten, dass diese Kosten für jede Workflowinstanz anfallen, so dass die Anwendung einer *Eliminate-Temporary-Table*-Regel insbesondere in einem Datenbanksystem zu einer erheblichen Entlastung des Normalbetriebs führen kann. Durch die Anwendung von *Query-Merging*-Regeln lässt sich das Datenvolumen, das von einer Daten- auf eine Workflowebene übertragen und dort materialisiert werden muss, reduzieren. Beispielsweise kombinieren die Regeln *Select-Into-Merging* und *Select-Merging* abhängige SQL-Anfragen, wodurch ein Anfrageergebnis direkt auf der Datenebene weiterverarbeitet werden kann. Die *Predicate-Pushdown*-Regel bzw. die heuristischen *CGO*-Regeln zielen dagegen darauf ab, die Größe einer Anfrageergebnismenge zu reduzieren, indem ähnlich strukturierte SQL-Anfragen miteinander geeignet kombiniert werden. Durch die Kombination von Selektionsprädikaten können zudem für einen Anfrageoptimierer neue Optimierungsmöglichkeiten, die zu einer beschleunigten Ausführung einer kombinierten SQL-Anfrage auf der Datenebene führen können, entstehen. Es ist zu beachten, dass sich durch Anwendung einer *Activity-Merging*-Regel die ursprüngliche Ausführungsreihenfolge zwischen Aktivitäten bzw. den von ihnen ausgeführten Operationen ändern kann. Dies kann zum Problem werden, wenn die Ausführungsreihenfolgen zwingend eingehalten werden müssen, um die Semantik eines Geschäftsprozesses korrekt umzusetzen. Deshalb ist die Anwendung einer *Activity-Merging*-Regel zwar automatisch möglich. Jedoch sollte das Ergebnis der Regelanwendung immer von einem Benutzer überprüft werden, um Änderungen an der Geschäftsprozesssemantik auszuschließen. Somit ist eine automatische Anwendung einer *Activity-Merging*-Regel zum Deploymentzeitpunkt nur dann möglich, wenn zuvor ein Benutzer dem PGM/F-System ausreichend Informa-

tionen über einzuhaltende Ausführungsreihenfolgen zwischen den Aktivitäten eines Workflowmodells zur Verfügung gestellt hat.

- Mit einer *Tuple-To-Set*-Regel lassen sich potentiell die größten Optimierungsgewinne erzielen. Durch die Wandlung einer tupelbasierten in eine äquivalente, mengenbasierte DML-Anweisung können Ausführungskosten sowohl auf der Workflow- als auch auf der Datenebene in hohem Maße eingespart werden. Muss vor der Regelanwendung noch für jedes Tupel einer materialisierten Datenmenge eine DML-Anweisung ausgeführt werden, wird nach der Regelanwendung die transformierte, mengenorientierte DML-Anweisung genau einmal ausgeführt. Je größer die materialisierte Datenmenge ist, desto höher fällt der Optimierungsgewinn aus. Dieser setzt sich zusammen aus den auf der Workflow- und Datenebene eingesparten Ausführungskosten, um eine DML-Anweisung bzw. die zugehörige *SQL-DML*-Aktivität auszuführen. Des Weiteren kann nach der Regelanwendung auf die Übertragung und Materialisierung der zu iterierenden Datenmenge an ein Workflowsystem verzichtet werden, da die Datenmenge unmittelbar in der abhängigen mengenbasierten DML-Anweisung referenziert werden kann. Alle *Tuple-To-Set*-Regeln können sowohl zur Modellierungs- als auch zur Deploymentzeit eines Workflowmodells automatisch angewendet werden.
- Die Anwendung der *SQL-Simplification*-Regeln führt dazu, dass unnötige Attribute aus der Select-Klausel einer SQL-Anfrage entfernt werden. Hierdurch beschleunigt sich die Ausführung einer SQL-Anfrage, und es verringert sich die Größe einer Anfrageergebnismenge. Dadurch können Übertragungs- und Materialisierungskosten eingespart werden. Durch die Identifizierung sich überlappender Prädikate in der Where-Klausel einer SQL-Anweisung können Selektionsprädikate vereinfacht werden, und es kann die Übersetzungszeit einer SQL-Anweisung in einem Datenbanksystem verkürzt werden.
- Eine parallele Modellierung datenunabhängiger SQL-Anweisungen ist nur dann gewinnbringend, wenn die SQL-Anweisungen sowohl auf der Workflow- als auch auf der Datenebene parallel verarbeitet werden können. Ob eine parallele Verarbeitung möglich ist, hängt zum einen von den Fähigkeiten eines Workflowsystems ab und zum anderen von der Organisation der Datenebene. Eine echte parallele Ausführung von SQL-Anweisungen auf der Datenebene ist nur möglich, wenn diese jeweils auf verschiedenen Datenbanksystemen unabhängig voneinander ausgeführt werden können. Im Falle eines zentralisierten bzw. eines föderierten Datenbanksystems können dagegen diese Parallelisierungseffekte weit weniger effizient genutzt werden. Des Weiteren ist eine Parallelisierung datenunabhängiger SQL-Anweisungen sowohl zum Modellierungs- als auch zum Deploymentzeitpunkt automatisch durchführbar, da alle hierfür notwendigen Informationen direkt aus einem Workflowmodell ausgelesen werden können.
- Durch die Verlagerung von SQL-Anweisungen in eine Stored-Procedure lässt sich die Laufzeit eines datenintensiven Workflows weiter reduzieren. Abhängig von der Anzahl der ausgelagerten SQL-Anweisungen können Ausführungskosten auf der Workflowebene für die zugehörigen auszuführenden SQL-Aktivitäten eingespart werden.

Dies verringert die Nachrichten und das Datenvolumen, die zwischen Workflow- und Datenebene ausgetauscht werden müssen. Können zudem SQL-Anfragen in eine Stored-Procedure, deren Ergebnismengen direkt auf der Datenebene weiterverarbeitet werden, ausgelagert werden, können weitere Übertragungs- und Materialisierungskosten eingespart werden. Des Weiteren können die ausgelagerten SQL-Anweisungen aufgrund der Vorübersetzung einer Stored-Procedure in einem Datenbanksystem wesentlich effizienter ausgeführt werden als dies bei den Einzelausführungen der Anweisungen möglich wäre. Zu beachten ist, dass eine Verlagerung von SQL-Anweisungen nur dann durchgeführt werden kann, wenn alle SQL-Anweisungen auf demselben Datenbanksystem ausgeführt werden. Sonst ist die Ausführbarkeit einer erzeugten Stored-Procedure nicht mehr gewährleistet. Ferner muss der Typ eines solchen Datenbanksystems bekannt sein, um die Stored-Procedure-Beschreibung an den jeweiligen herstellerspezifischen Dialekt von SQL/PSM anpassen zu können. Deshalb ist eine automatische Regelanwendung zum Modellierungszeitpunkt nur dann möglich, wenn alle relevanten Informationen über ein ausführendes Datenbanksystem direkt aus einem Workflowmodell ausgelesen werden können. Sonst müssen die Informationen explizit durch einen Benutzer zur Verfügung gestellt werden bzw. die Regelanwendung muss auf den Deploymentzeitpunkt eines Workflowmodells, bei dem die Informationen aus dem Deploymentdeskriptor eines Workflowmodells gewonnen werden können, verschoben werden.

- Bei den kostenbasierten Optimierungsstrategien können sowohl positive als auch negative Optimierungseffekte erzielt werden. Beispielsweise können sich die Ausführungskosten von Update- und Insert-Anweisungen erhöhen, wenn diese mittels einer Merge-Anweisung kombiniert werden. Der mögliche Mehraufwand muss mit den Optimierungsgewinnen, die durch den Wegfall einer *SQL-DML*-Aktivität auf der Workflowebene erzielt werden können, verrechnet werden.
- Auch die Optimierung von Verbundoperationen weist ein großes Optimierungspotential auf. Es ist abzuwägen, ob es gewinnbringender ist, eine Verbundoperation auf der Datenebene auszuführen und nur das Verbundergebnis an das Workflowsystem zu übertragen, oder die zu kombinierenden Datenmengen zuerst an das Workflowsystem zu übermitteln und anschließend die Verbundberechnung lokal auf der Workflowebenene durchzuführen. Welche der beiden Varianten zu einer größeren Reduzierung der durch eine Verbundberechnung verursachten Ausführungs-, Übertragungs- und Materialisierungskosten führt, hängt insbesondere von der Größe der zu kombinierenden Datenmengen bzw. von der Selektivität einer Verbundoperation ab. Deshalb ist hier eine Kostenabschätzung der verschiedenen Varianten unvermeidlich. Bei der Optimierung von Verbundoperationen ist zu beachten, dass die Anzahl der zur Verfügung stehenden Varianten einer Verbundberechnung von der Organisation der Datenebene abhängt. Bei einem zentralisierten bzw. föderierten Datenbanksystem kann eine Verbundoperation sowohl auf der Workflow- als auch auf der Datenebene ausgeführt werden. Dagegen kann eine Verbundoperation immer nur auf der Workflowebene durchgeführt werden, wenn die daran beteiligten SQL-Anweisungen auf unterschiedlichen Datenbanksystemen ausgeführt werden.

- Die Multi-Query-Optimierung ist eine bewährte Technik in einem Datenbanksystem zur Optimierung einer Menge von SQL-Anweisungen. Diese Technik kann auch auf den Kontext datenintensiver Workflows übertragen werden. Falls ein Datenbanksystem eine geeignete Unterstützung anbietet, kann die Optimierungstechnik sogar im Rahmen der heuristischen Optimierung einer PGM-Repräsentation angewendet werden. In diesem Fall müssen lediglich die SQL-Anweisungen, die in einem Workflowmodell definiert sind, an ein ausführendes Datenbanksystem übermittelt werden. Das Datenbanksystem entscheidet darüber, ob die Erzeugung materialisierter Sichten zu einer Laufzeitverbesserung der gegebenen Menge von SQL-Anweisungen und damit auch des datenintensiven Workflows führen kann. Ohne eine solche datenbankseitige Unterstützung muss die Multi-Query-Optimierung dagegen vom PGM/F-System realisiert werden, was nur im Rahmen einer kostenbasierten Optimierung möglich ist. Des Weiteren kann die Multi-Query-Optimierungsstrategie nur dann automatisch zum Modellierungszeitpunkt eines Workflowmodells ausgeführt werden, wenn in einem Workflowmodell ausreichend Informationen über ein ausführendes Datenbanksystem zur Verfügung stehen. Die Informationen können dem PGM/F-System auch durch einen Benutzer mitgeteilt werden. Sonst können diese Informationen erst aus dem Deploymentdeskriptor des Workflowmodells gewonnen werden. Dadurch muss die Multi-Query-Optimierung auf den Deploymentzeitpunkt eines Workflowmodells verschoben werden.
- Die Optimierungseffekte der meisten Optimierungstechniken können unabhängig davon erzielt werden, ob ein zentralisiertes oder ein föderiertes Datenbanksystem zu Grunde liegt. Nur bei den heuristischen *Update-Merging*-Regeln können die Optimierungseffekte variieren: Im Falle eines föderierten Datenbanksystems können die durch die Regelanwendung entstehenden möglichen Parallelisierungseffekte aufgrund der verteilten Architektur des Datenbanksystems immer genutzt werden. Dagegen ist dies bei einem zentralisierten Datenbanksystem nur im Falle einer parallelen Architektur möglich. Die meisten Optimierungstechniken können nicht angewendet werden, wenn die zu optimierenden Datenverarbeitungsoperationen auf verschiedenen Datenbanksystemen ausgeführt werden. Bei der Parallelisierung von SQL-Anweisungen führt diese Form der Organisation der Datenebene allerdings zu höheren Optimierungsgewinnen als dies bei einem zentralisierten bzw. föderierten Datenbanksystem möglich ist.

	Optimierungseffekt							Optimierungszeitpunkt		Organisation der Datenebene
	O1	O2	O3	O4	O5	O6	O7	MZ	DZ	
Web-Service-Pushdown	+							A, B	A	Z, F, U
Assign-Merging		+						A	A	Z, F
Update-Merging [H]		+		+	+	++		B	B	Z, F
Query-Merging		+	++	+	++	+	++	B	B	Z,F
CGO		+	++	+	++	+	++	B	B	Z, F
Eliminate-Temporary-Table		+		+	+	++		B	B	Z, F
Tuple-To-Set		++	++	++	++	++		A	A	Z, F
SQL-Simplification			+		+		+	A	A	Z, F
SQL-Parallelization						++		A	A	(Z, F), U
Stored-Procedure-Pushdown		+	+	+	+	++		A, B	A	Z, F
Update-Merging [K]		+		+	+	+/-		B	B	Z, F
Join-Optimization [K]		+/-	+/-	+/-	+/-	+/-	+/-	A	A	Z, F, (U)
MQO [K]						+/-		A, B	A	Z, F

Optimierungseffekt:
O1: Regelvorbereitung
O2: Reduzierung der # Aktivitäten
O3: Reduzierung der Materialisierungskosten
O4: Reduzierung der # Nachrichten
O5: Reduzierung des zu übertragenden Datenvolumens
O6: Reduzierung der SQL-Ausführungskosten
O7: Reduzierung der Anfrageergebnismenge

Optimierungszeitpunkt:
MZ: Modellierungszeitpunkt eines Workflows
DZ: Deploymentzeitpunkt eine Workflows
A: Automatische Regelanwendung
B: Regelanwendung mit Benutzerinteraktion

Organisation der Datenebene:
Z: Zentralisiertes Datenbanksystem
F: Föderiertes Datenbanksystem
U: *n* unabhängige Datenbanksysteme

Abbildung 5.19.: Charakteristische Eigenschaften der Regelbasis von PGM/F

6

Kontrollstrategie von PGM/F

Die Kontrollstrategie von PGM/F bestimmt die Reihenfolge, in der die in Kapitel 5 vorgestellte, heuristische Regelbasis auf eine PGM-Repräsentation angewendet wird. Dabei werden Abhängigkeiten zwischen den Regeln ausgenutzt, um die Reihenfolge der Regelausführungen zu bestimmen. Dadurch können insbesondere unnötige Optimierungsläufe vermieden werden, da gezielt jene Regeln angewendet werden, die potentiell zu einer Restrukturierung einer PGM-Repräsentation führen. Des Weiteren stellt die Kontrollstrategie sicher, dass alle Restrukturierungsmöglichkeiten in einer PGM-Repräsentation untersucht werden und die Regelverarbeitung in einer endlichen Zeit terminiert.

Somit unterscheidet sich die Kontrollstrategie von PGM/F wesentlich von der Kontrollstrategie eines typischen heuristischen, regelbasierten Anfrageoptimierers, wie er beispielsweise in Starburst [PLH97] entwickelt wurde. Bei diesem werden Regeln prioritätsgesteuert, sequentiell oder zufällig auf eine Anfrage angewendet. Ebenso ist in PGM/F eine Budgetierung der Regelausführung nicht vorgesehen, weil die PGM-Optimierung zum Modellierungs- bzw. Deploymentzeitpunkt eines Workflowmodells durchgeführt wird. In Starburst findet dagegen die Optimierung auch zur Ausführungszeit einer Anfrage statt, weshalb hier eine zeitliche Begrenzung der Anfrageoptimierung aus Effizienzgründen zwingend erforderlich ist.

In den folgenden Abschnitten wird die Kontrollstrategie von PGM/F vorgestellt. Zunächst werden in Teilkapitel 6.1 die Abhängigkeiten zwischen den einzelnen Regeln, welche die Ausführungsreihenfolge der Regeln bestimmen, beschrieben. Der genaue Ablauf der Regelanwendungen auf eine PGM-Repräsentation wird anschließend in Teilkapitel 6.2 erläutert. In Teilkapitel 6.3 wird argumentiert, weshalb dieser Ablauf terminiert. Abschließend fasst Teilkapitel 6.4 die wichtigsten Ergebnisse dieses Kapitels zusammen.

6.1. Abhängigkeitsbeziehungen

Die Kontrollstrategie von PGM/F nutzt Abhängigkeiten zwischen den Regeln aus, bei denen die Anwendung einer Regel die Anwendung einer anderen Regel ermöglichen kann. Diese Abhängigkeitsbeziehungen sind in Abbildung 6.1 veranschaulicht.

Die erste Spalte bzw. Zeile der dargestellten Tabelle listet alle Restrukturierungsregeln auf, die im Rahmen der heuristischen Optimierung auf eine PGM-Repräsentation angewendet werden. Dabei wird für jede Restrukturierungsregel in der entsprechenden Zeile markiert, welche anderen Regeln durch die Restrukturierungsregel vorbereitet (+) bzw. verhindert (−) werden können. Daraus lassen sich die entsprechenden Abhängigkeiten zwischen den einzelnen Regeln, die im Folgenden beschrieben werden, direkt ableiten.

Eine *Web-Service-Pushdown*-Regel kann für vier unterschiedliche Typen von Web-Service-Operationen verwendet werden (siehe Teilkapitel 5.3). Hieraus resultieren unterschiedliche Abhängigkeiten zu verschiedenen Regeln. Die 1. Variante der *Web-Service-Pushdown*-Regel transformiert z.B. Web-Service-Aufrufe vom Typ 1 und 3, die einen skalaren Ausgabewert liefern, in eine Select-Into-Anweisung. Dies kann die Anwendung einer *Select-Into-Merging*-Regel ermöglichen. Die 2. Variante der *Web-Service-Pushdown*-Regel wandelt dagegen Web-Service-Aufrufe vom Typ 2 und 4 mit einem mengenbasierten Ausgabewert in eine Select-Anweisung um. Wird anschließend das Ergebnis der Select-Anweisung unter Verwendung eines prozeduralen Datenverarbeitungsmusters iterativ bearbeitet, kann die Anwendung einer *Tuple-To-Set*-Regel ermöglicht werden. Ebenso können Selektionsprädikate in eine solche Select-Anweisung verschoben werden. Dies macht die *Predicate-Pushdown*-Regel anwendbar. Da die Web-Service-Aufrufe vom Typ 3 und 4 mengenbasierte Eingabewerte besitzen, kann dies zur Anwendung einer *Select-Merging*-Regel führen. Schließlich wird durch eine *Web-Service-Pushdown*-Regel der Web-Service-Partner von der Workflow- auf die Datenebene verlagert, was die Ausführung der *Eliminate-Unused-Partner*-Regel zum Erhalt der Konsistenz einer PGM-Repräsentation erzwingt.

Die Anwendung einer *Activity-Merging*-Regel innerhalb einer Schleife kann die Anzahl der Aktivitäten im Schleifenrumpf derart reduzieren, dass die Anwendung einer *Tuple-To-Set*-Regel ermöglicht werden kann.

Die Regeln *Assign-Merging*, *Eliminate-Temporary-Table*, *Predicate-Pushdown*, *Select-Merging* und *Select-Into-Merging* ersetzen in einer SQL-Anweisung eine Variable, die das Ergebnis einer zuvor ausgeführten Datenverarbeitungsoperation zur Verfügung stellt. Da die Referenzierung dieser Variablen nach einem Substitutionsschritt nicht mehr länger erforderlich ist, kann die redundante Variablendefinition mit der *Eliminate-Unused-Variable*-Regel aus einer PGM-Repräsentation anschließend entfernt werden.

Wird der Datenfluss zwischen zwei Aktivitäten durch eine Variablenzuweisung bzw. Select-Into-Anfrage unterbrochen, können die *Assign-Merging*- bzw. *Select-Into-Merging*-Regeln zu einer direkten Datenabhängigkeit zwischen den beiden Aktivitäten führen. Dadurch können weitere *Activity-Merging*-Regeln vorbereitet werden.

Eine *Insert-Insert-Merging*-Regel kann die Anzahl der Einfügeoperationen auf einer Tabelle auf eine Anweisung reduzieren. Wird eine Anweisung zugleich auf einer temporären Tabelle ausgeführt, kann die *Eliminate-Temporary-Table*-Regel angewendet werden.

Durch die *CGO*-, *Predicate-Pushdown*- und *Delete-Delete-Merging*-Regeln können in der Where-Klausel einer kombinierten SQL-Anweisung redundante Selektionsprädikate, die sich durch die *Eliminate-Redundant-Predicates*-Regel entfernen lassen, entstehen.

Des Weiteren können durch einige *CGO*-Regeln Redundanzen in einer Select-Klausel einer kombinierten SQL-Anweisung, die anschließend durch Anwendung einer *Eliminate-Redundant-Attributes*-Regel entfernt werden können, auftreten.

Die Anwendung einer *Tuple-To-Set*-Regel erzeugt eine einzelne SQL-DML-Aktivität, die von *Activity-Merging*-Regeln weiter berücksichtigt werden kann. Hierbei handelt es sich um die Regeln *Assign-Merging*, *Select-Into-Merging* und *Eliminate-Temporary-Table*, die eine Kombination der von der *SQL-DML*-Aktivität ausgeführten DML-Anweisung mit einer weiteren Datenverarbeitungsoperation unterstützen. Handelt es sich bei dieser DML-Anweisung um eine Insert- bzw. Delete-Operation kann zusätzlich die *Insert-Insert-Merging*- bzw. *Delete-Delete-Merging*-Regel verwendet werden.

Da nach Anwendung einer *Tuple-To-Set*-Regel einige Variablen nicht mehr erforderlich sind, kann die *Eliminate-Unused-Variable*-Regel angewendet werden, um die Konsistenz einer PGM-Repräsentation wiederherzustellen.

Die *Eliminate-Unused-Attributes*-Regel entfernt nicht benutzte Attribute aus einer Select-Klausel einer SQL-Anfrage. Handelt es sich dabei um eine Select-Into-Anfrage kann unter Umständen die *Select-Into-Merging*-Regel Anwendung finden.

Durch das Verlagern von Datenverarbeitungsoperationen in eine Stored-Procedure werden die Datenverarbeitungsoperationen bzw. die damit verbundenen Aktivitäten aus einer PGM-Repräsentation entfernt und stehen somit für nachfolgende Optimierungsentscheidungen nicht mehr zur Verfügung. Folglich verhindert die *Stored-Procedure-Pushdown*-Regel die Anwendung weiterer Restrukturierungsregeln auf eine PGM-Repräsentation. Dazu gehören die Regelklassen *Activity-Merging*, *Tuple-To-Set* und *SQL-Simplification*. Auch eine Parallelisierung von SQL-Anweisungen ist nach einem Stored-Procedure-Pushdown nicht mehr möglich. Variablendefinitionen, die nach der Regelanwendung nicht mehr in der PGM-Repräsentation referenziert werden, können anschließend mit der *Eliminate-Unused-Variable*-Regel entfernt werden.

Heuristische Regelbasis von PGM/F	WSPD	AM	ETT	PPD	PPD2	SIM	SM	CGO	UUM	IIM	DDM	IT2S	UT2S	DT2S	ERA	EUA	ERP	PAR	SPP	EUP	EUV
Web-Service-Pushdown (für Typ 1) (WSPD)						+														+	
Web-Service-Pushdown (für Typ 2) (WSPD)				+								+	+	+						+	
Web-Service-Pushdown (für Typ 3) (WSPD)						+	+													+	
Web-Service-Pushdown (für Typ 4) (WSPD)				+			+					+	+	+						+	
Assign-Merging (AM)		+	+	+	+	+	+	+	+	+	+	+	+	+							+
Update-Update-Merging (UUM)												+	+	+							
Insert-Insert-Merging (IIM)			+									+	+	+							
Delete-Delete-Merging (DDM)												+	+	+			+				
Select-Into-Merging (SIM)			+	+	+	+	+	+	+	+	+	+	+	+							+
Select-Merging (SM)												+	+	+							+
Predicate-Pushdown (PPD)												+	+	+			+				+
Predicate-Pushdown (Variante) (PPD2)												+	+	+			+				
CGO (CGO)												+	+	+	+		+				
Eliminate-Temporary-Table (ETT)												+	+	+							+
Insert-Tuple-To-Set (IT2S)			+			+				+											+
Update-Tuple-To-Set (UT2S)			+			+															+
Delete-Tuple-To-Set (DTTS)			+			+					+										+
Eliminate-Redundant-Attributes (ERA)																					
Eliminate-Unused-Attributes (EUA)						+															
Eliminate-Redundant-Predicates (ERP)																					
SQL-Parallelization (PAR)																					
Stored-Procedure-Pushdown (SPP)																		-			+
Eliminate-Unused-Partner (EUP)																					
Eliminate-Unused-Variable (EUV)																					

Regelanwendung wird vorbereitet (+) / verhindert (-)

Abbildung 6.1.: Abhängigkeitsbeziehungen in der Regelbasis von PGM/F

6.2. Ablauf der Regelanwendungen

Wie bereits in Teilkapitel 4.3.4 diskutiert, dürfen bei der Regelanwendung Optimierungsgrenzen eines Workflows nicht überschritten werden, damit die ursprüngliche Ausführungssemantik des Workflows nicht geändert wird. Die Optimierungsgrenzen werden in PGM durch Optimierungssphären (OS) bzw. Loop-Optimierungssphären (LOS) repräsentiert. Eine OS definiert einen Ausführungskontext für Aktivitäten, der mit einer Fehlerbehandlung assoziiert sein kann. Da die Fehlerbehandlung bei der Anwendung der Restrukturierungsregeln auf eine PGM-Repräsentation nicht berücksichtigt wird, müssen die Effekte einer Regelanwendung auf eine OS beschränkt bleiben, um die Ausführungssemantik eines zu Grunde liegenden Workflowmodells zu erhalten. Bei einer LOS handelt es sich dagegen um eine spezielle Optimierungssphäre, die ein `LOOP`-Kontrollflussmuster umfasst, aus dem keine Aktivitäten hinein oder heraus verschoben werden können, ohne die Ausführungssemantik der zugehörigen, ausgeführten Operation zu ändern (siehe auch Teilkapitel 5.4.1). Deshalb müssen hier die Effekte einer Regelanwendung auf den Schleifenrumpf eines `LOOP`-Kontrollflussmusters beschränkt werden. Gleichzeitig kann eine LOS ein Datenverarbeitungsmuster beschreiben, das für die Anwendung einer *Tuple-To-Set*-Regel erforderlich ist.

Aus der verschachtelten Struktur von Optimierungsgrenzen in einer Workflowbeschreibung entsteht in PGM eine verschachtelte Struktur von Optimierungssphären. Diese Struktur lässt sich auch als Baumstruktur darstellen, in der sich die Hierarchie der Optimierungssphären widerspiegelt. Die Hierarchie wird während des Optimierungsprozesses per Tiefensuche durchlaufen. Damit ist gewährleistet, dass jede eingeschlossene Sphäre isoliert betrachtet optimiert wird, bevor die Optimierung der umschließenden Sphäre beginnt. Dies bedeutet insbesondere, dass eine eingeschlossene Sphäre bei der Optimierung einer umschließenden Sphäre als eine Art Black-Box betrachtet werden kann. Nichtsdestotrotz müssen dabei Datenabhängigkeiten zwischen einer umschließenden Sphäre und ihrer eingeschlossenen Sphären berücksichtigt werden.

Nach der Optimierung einer Sphäre bleibt diese im Regelfall erhalten. Eine Ausnahme bildet eine LOS, die durch Anwendung einer *Tuple-to-Set*-Regel aufgelöst werden kann.

Die Restrukturierungsregeln werden immer innerhalb einer Optimierungssphäre angewendet. Dabei bestimmen die Abhängigkeiten zwischen den einzelnen Restrukturierungsregeln die Reihenfolge ihrer Anwendung. Gleichzeitig bestimmt der Typ einer Optimierungssphäre die Menge der anwendbaren Restrukturierungsregeln. In Abbildung 6.3 sind diese beiden sphärenspezifischen Kontrollstrategien, auf die später noch genauer eingegangen wird, dargestellt.

Wie in Abbildung 3.1 gezeigt, erhält der PGM-Optimierer die PGM-Repräsentation eines datenintensiven Workflows als Eingabe. In einem ersten Schritt identifiziert er alle Optimierungssphären innerhalb dieser Repräsentation. Im Falle eines zu Grunde gelegten BPEL/SQL-Workflowmodells definiert die verschachtelte Struktur des Workflows einen Baum, der eine Hierarchie aller in diesem Workflow definierten Optimierungsgrenzen repräsentiert. Der Baum wird mittels einer Tiefensuche (depthfirst manner in post-order) durchlaufen. Dies garantiert, dass alle geschachtelten Sphären vor ihren umschließen-

den Sphären verarbeitet werden. Wird eine umschließende Sphäre optimiert, werden alle darin geschachtelten Optimierungssphären ausgeblendet. Allerdings werden ihre Datenabhängigkeiten zu Aktivitäten in der betrachteten, umschließenden Optimierungssphäre berücksichtigt, um Regeln korrekt anwenden zu können. Kann eine *Tuple-to-Set*-Regel auf eine LOS angewendet werden, wird die LOS aufgelöst, da als Ergebnis der Regelanwendung eine einzelne *SQL-DML*-Aktivität entsteht. Die *SQL-DML*-Aktivität wird in die umschließende Optimierungssphäre integriert und kann somit bei der Anwendung der Regelmenge innerhalb der umschließenden Optimierungssphäre weiter berücksichtigt werden und damit Teil weiterer Transformationschritte sein.

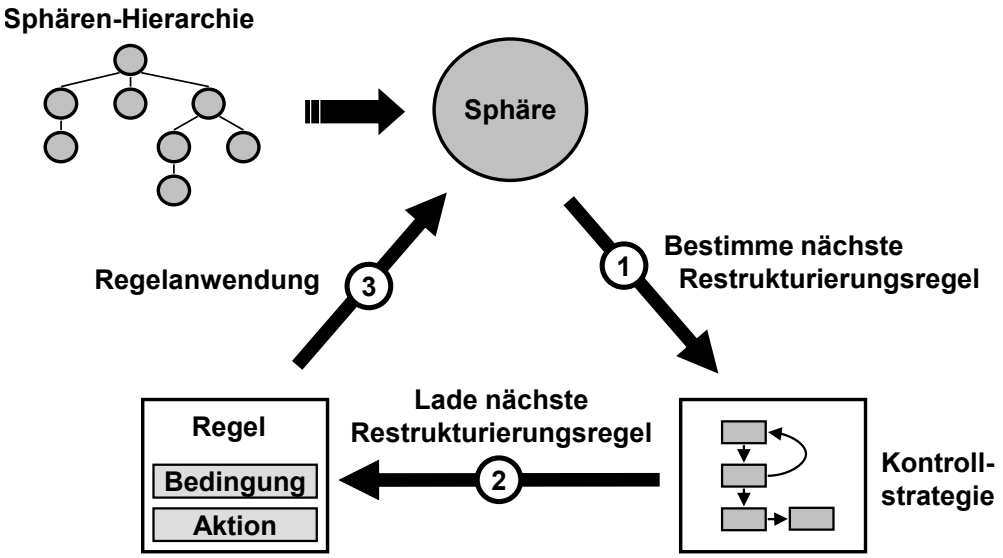


Abbildung 6.2.: Ablauf der Regelanwendungen auf eine Optimierungssphäre

Abbildung 6.2 veranschaulicht diesen Optimierungsablauf. Zuerst wird die aktuell zu betrachtende Optimierungssphäre in der Hierarchie gemäß einer Tiefensuchstrategie ermittelt. Danach wird die nächste anzuwendende Regel mithilfe der gegebenen Kontrollstrategie bestimmt. Innerhalb der Optimierungssphäre wird nach einer Gruppe von Aktivitäten gesucht, welche die Bedingung dieser Regel erfüllt, und im Folgenden als Match bezeichnet wird. Wurde ein Match gefunden, werden die in der Regel definierten Transformationsschritte angewendet. Kann die Regel in der Optimierungssphäre nicht mehr weiter angewendet werden, wird die Optimierung mit der nächsten Regel gemäß der vorliegenden Kontrollstrategie fortgesetzt. Der Optimierungsablauf setzt sich solange fort, bis die Regelmenge auf der Optimierungssphäre vollständig ausgeführt worden ist. Auf diese Weise werden alle Sphären innerhalb der Hierarchie bis hinauf zur Wurzelsphäre optimiert.

Die Algorithmen 1 und 2 implementieren diesen Ablauf. Die Methode *optimizeSphereHierarchy* (Algorithmus 1) traversiert eine gegebene Sphärenhierarchie *sh* und ruft für jede Optimierungssphäre *s* in *sh* die Methode *optimizeSphere* (Algorithmus 2) auf. Diese Methode ist für die regelbasierte Optimierung der einzelnen Sphären in *sh* verantwortlich. Die Funktion *getNextSphere* implementiert die Tiefensuche auf der Sphärenhierarchie. Die Methode *optimizeSphere* ruft zuerst die Funktion *getControlStrategy*, die entweder eine

Algorithm 1 optimizeSphereHierarchy

Require: sphere-hierarchy sh
Ensure: optimized sphere-hierarchy sh
  while sh is not fully traversed **do**
    $s \leftarrow getNextSphere(sh)$
    $optimizeSphere(s)$
  end while

Algorithm 2 optimizeSphere

Require: sphere s
Ensure: optimized sphere s
  $cs \leftarrow getControlStrategy(s)$
  while cs is not finished **do**
    $r \leftarrow getNextRule(cs)$
    while s is not fully traversed **do**
      $a \leftarrow getNextActivity(s)$
      $m \leftarrow findMatch(a, s, r)$
      if $m \neq \emptyset$ **then**
        $applyRule(m, r)$
      end if
    end while
  end while

Kontrollstrategie *cs* für eine LOS oder für eine OS abhängig von dem Typ der vorliegenden Sphäre s zurückliefert, auf. Die Funktion *getNextRule* ermittelt die nächste Regel r gemäß *cs*. In einem nächsten Schritt wird versucht, alle Matches für r in der Sphäre s zu finden. Hierfür implementiert die Funktion *getNextActivity* eine Tiefensuche auf der Sphäre s. Dabei wird jede Aktivität in s genau einmal betrachtet, wobei die Aktivitäten einer geschachtelten Optimierungssphäre ignoriert werden. Für jede gelieferte Aktivität a, prüft die Funktion *findMatch*, ob die Regel r auf den PGM-Teilgraphen beginnend bei Aktivität a anwendbar ist. Im positiven Fall wird ein Match m geliefert. Die Bedingung einer Regel stellt dabei sicher, dass es immer höchstens einen Match für eine betrachtete Aktivität a gibt. Der Treffer m wird der Methode *applyRule* übergeben, welche die Regel r auf m in s anwendet. Ist dagegen kein Match vorhanden, wird diese Methode übersprungen, und es wird die nächste Regel gemäß *cs* in der folgenden Iteration betrachtet.

Abbildung 6.3 zeigt die Kontrollstrategien für die Sphärentypen LOS und OS. Die Kontrollstrategien umfassen jeweils die Regelmenge und die Reihenfolge, in der die Regelmenge auf eine Optimierungssphäre angewendet wird. Die grundlegende Idee hierbei ist, die Regeln gemäß ihrer Abhängigkeitsbeziehungen anzuwenden.

Um die Darstellung der Kontrollstrategien zu vereinfachen, werden in Abbildung 6.3 sowohl einzelne Regeln als auch Regelklassen als Rechtecke dargestellt. Im Falle von Regelklassen können die darin enthaltenen Regeln (nicht dargestellt) in beliebiger Reihenfolge angewendet werden, weil zwischen ihnen keine Abhängigkeiten zu berücksichtigen sind.

Das Ziel einer LOS-Optimierung ist die Eliminierung der LOS durch eine *Tuple-to-Set*-Regel. Um alle Abhängigkeitsbeziehungen optimal ausnutzen zu können, werden die Regeln dabei in folgender Reihenfolge in einer LOS angewendet.

Zunächst erfolgt die Anwendung der *Eliminate-Unused-Attributes*-Regel, welche die Anwendung einer *Select-Into-Merging*-Regel ermöglichen kann. Anschließend werden die Regeln *Assign-Merging*, *Select-Into-Merging* und *Web-Service-Pushdown* abwechselnd angewendet, solange bis alle Abhängigkeitsbeziehungen zwischen den Regeln ausgenutzt werden konnten.

Dabei erfolgt die Anwendung der *Web-Service-Pushdown*-Regel auf einen Web-Service-Aufruf vom Typ 1. Da die Verlagerung dieses Web-Service-Aufrufes auf die Datenebene eine *Select-Into-Merging*-Regel ermöglichen kann, wird die *Web-Service-Pushdown*-Regel immer mit einer *Select-Into-Merging*-Regel kombiniert. Nur wenn nach der *Web-Service-Pushdown*-Regel eine *Select-Into-Merging*-Regel angewendet werden kann, wird der entsprechende Web-Service-Aufruf auf die Datenebene verlagert. Sonst wird die *Web-Service-Pushdown*-Regel mangels positiver Optimierungseffekte nicht angewendet.

Das gleiche gilt für Web-Service-Aufrufe vom Typ 3, die danach nur dann auf die Datenebene verlagert werden dürfen, wenn die Anwendung einer *Select-Merging*-Regel ermöglicht wird. Dieses Optimierungsverhalten ist durch die Kontrollstrategie gewährleistet, weil die entsprechende *Web-Service-Pushdown*-Regel nur in Kombination mit einer *Select-Merging*-Regel angewendet werden kann.

Anschließend wird die *Predicate-Pushdown*-Regel, die mit der *Eliminate-Redundant-Predicates*-Regel kombiniert wird, ausgeführt. Dies ist sinnvoll, um mögliche Redundanzen in kombinierten Selektionsprädikaten, welche durch die *Predicate-Pushdown*-Regel entstehen können, wieder unmittelbar zu entfernen. Da solche Redundanzen auch durch Anwendung der Regeln *Delete-Delete-Merging* bzw. *Predicate-Pushdown* (Variante) entstehen können, ist eine Kombination mit der *Eliminate-Redundant-Predicates*-Regel hier ebenfalls sinnvoll.

Nach den *Query-Merging*-Regeln werden die *CGO*-Regeln angewendet. Dabei werden die *CGO*-Regeln in Kombination mit den Regeln *Eliminate-Redundant-Attributes* und *Eliminate-Redundant-Predicates* ausgeführt, um mögliche, durch die Regelanwendungen entstandene Redundanzen aus den SQL-Anfragen wieder direkt zu entfernen.

Im nächsten Schritt erfolgt die Ausführung der *Update-Merging*-Regeln und der *Eliminate-Temporary-Table*-Regel.

Nach Ausführung aller *Activity-Merging*-Regeln wird versucht, eine LOS aufzulösen. Zunächst wird eine *Predicate-Pushdown*-Regel (Variante) angewendet, die anschließend eine *Tuple-To-Set*-Regel ermöglichen kann. Da auch die Verlagerung eines Web-Service-Aufrufes vom Typ 2 oder 4 eine *Tuple-To-Set*-Regel vorbereiten kann, ist eine Kombination der entsprechenden *Web-Service-Pushdown*-Regeln mit den *Tuple-To-Set*-Regeln sinnvoll. Dies bedeutet, dass eine *Web-Service-Pushdown*-Regel nur zur Vorbereitung einer *Tuple-To-Set*-Regel angewendet werden darf. Eine Verlagerung des entsprechenden Web-Service-Aufrufes auf die Datenebene ist ausgeschlossen, weil sonst zusätzliche Übertragungskosten entstehen können, die sich auf die Optimierungseffekte negativ aus-

wirken können. Bei einem Web-Service-Aufruf vom Typ 4 muss zusätzlich die Anwendung einer *Select-Merging*-Regel, mit der eine Übertragung mengenbasierter Eingabeparameter von der Workflow- auf die Datenebene verhindert werden kann, garantiert sein. Diese Kontrollstrategie gewährleistet, dass ein Web-Service-Aufruf nur dann von der Workflow- auf die Datenebene verlagert wird, wenn dies zu positiven Optimierungseffekten führt. Bei einer *Web-Service-Pushdown*-Regel wird zusätzlich überprüft, ob die Anwendung einer *Predicate-Pushdown*-Regel ermöglicht wird. Dadurch können die Abhängigkeitsbeziehungen zwischen den Regeln effektiv ausgenutzt werden.

Im besten Fall gelingt die Anwendung einer *Tuple-To-Set*-Regel, und die gesamte LOS wird durch eine einzelne *SQL-DML*-Aktivität ersetzt. Sonst bleibt die LOS erhalten.

Konnte die LOS nicht aufgelöst werden, ist eine Parallelisierung von SQL-Anweisungen im Schleifenrumpf möglich. Als finaler Transformationsschritt werden Sequenzen von SQL-Anweisungen in einer LOS bzw. die LOS selbst in eine Stored-Procedure verlagert.

Die Kontrollstrategie für eine OS leitet sich direkt aus der Kontrollstrategie für eine LOS ab. Da in einer OS eine geschachtelte LOS als separate Optimierungssphäre betrachtet wird, scheidet die Anwendung der Regelklasse *Tuple-to-Set* bei diesem Sphärentyp aus. Deshalb werden bei einer OS-Kontrollstrategie alle Regelklassen außer den *Tuple-to-Set*-Regeln angewendet. Ebenso kann in einer OS auf die Auslagerung von Schleifenkonstrukten bei einer *Stored-Procedure-Pushdown*-Regel verzichtet werden, da diese bereits bei der Optimierung einer eingeschlossenen LOS berücksichtigt worden sind.

Nach Anwendung aller Regeln auf die Wurzelsphäre einer PGM-Repräsentation werden in einem finalen Optimierungsschritt die Regeln *Eliminate-Unused-Variable* und *Eliminate-Unused-Partner* ausgeführt, um die Konsistenz einer PGM-Repräsentation, die möglicherweise zuvor verletzt worden ist, wiederherzustellen.

Wegen sprachlicher Einschränkungen von SQL/PSM werden bei der Repräsentation einer Stored-Procedure in PGM nicht alle PGM-Aktivitäten genutzt. Dies betrifft z.B. eine *Assign-Select-* oder *Invoke*-Aktivität. Daraus folgt, dass Regeln, welche diese Aktivitätstypen voraussetzen, auf die PGM-Repräsentation einer Stored-Procedure nicht angewendet werden können. Dies betrifft die Regel *Web-Service-Pushdown*, *Predicate-Pushdown* und die Regel zur Parallelisierung von SQL-Anweisungen. Auch die Anwendung einer *Stored-Procedure-Pushdown*-Regel ist in diesem Fall nicht möglich. Deshalb werden diese Regeln von einem PGM-Optimierer im Falle einer heterogenen, isolierten Optimierung nicht auf die PGM-Repräsentation einer Stored-Procedure angewendet. Auf diese Weise können redundante Optimierungsläufe vermieden werden.

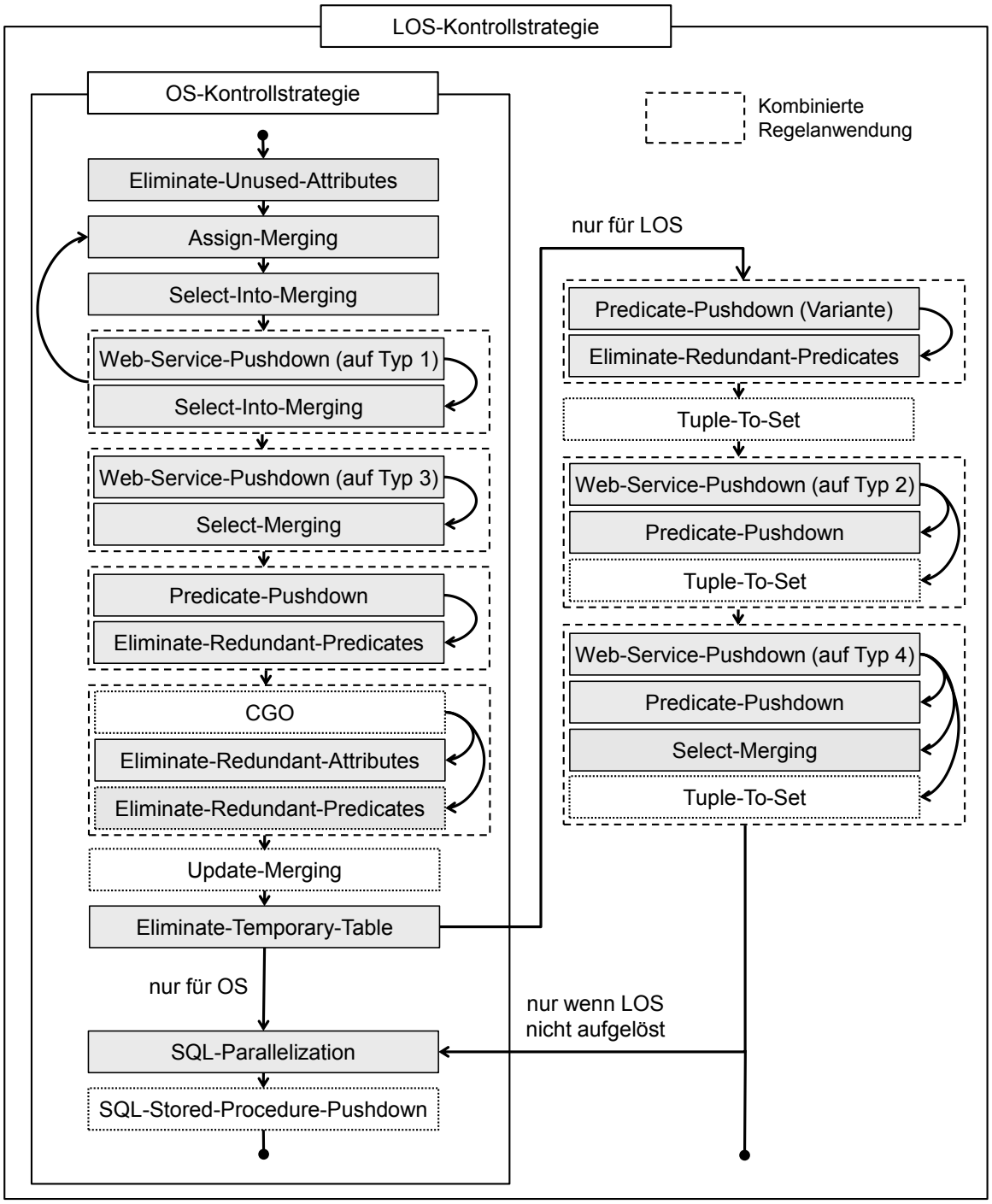


Abbildung 6.3.: Kontrollstrategie für Optimierungssphären

6.3. Terminierung

Für die im vorherigen Abschnitt vorgestellte Kontrollstrategie kann die Terminierung aus folgenden Gründen garantiert werden:

- Die Hierarchie der Optimierungssphären wird gemäß einer Tiefensuchstrategie durchlaufen. Dies stellt sicher, dass der Optimierungsprozess, der eine Optimierungssphäre verlässt und zu dessen umschließender Vatersphäre aufsteigt, nicht wieder zu dieser Optimierungssphäre hinabsteigen wird. Folglich endet der Optimierungsprozess immer an der Optimierungssphäre, welche die Wurzel der Hierarchie bildet.
- Die Traversierungsstrategie innerhalb einer Optimierungssphäre betrachtet jede Aktivität der Optimierungssphäre genau einmal, wenn überprüft wird, ob der Teil einer PGM-Repräsentation beginnend bei einer Aktivität eine Regel erfüllt oder nicht. Die endliche Anzahl von Aktivitäten gewährleistet somit, dass der Überprüfungsprozess für eine Regel nach einer endlichen Anzahl von Schritten terminiert.
- Keine Regelanwendung erhöht die Anzahl der Aktivitäten in einer PGM-Repräsentation. Außerdem existieren keine Regeln, welche Transformationsschritte anderer Regeln wieder rückgängig machen. Aus diesem Grund sind oszillierende Regelanwendungen, bei denen Regeln zyklisch angewendet werden und somit eine Terminierung der Kontrollstrategie in endlichen Schritten verhindern, ausgeschlossen.
- Wird eine Regel auf eine Aktivität angewendet, kann die Regelanwendung auf derselben Aktivität nicht wiederholt werden, weil durch die Regelanwendung das Muster, welches die Regel voraussetzt, verschwunden ist. Da es darüber hinaus nur eine endliche Anzahl an Aktivitäten gibt, kann eine Regel in einer PGM-Repräsentation folglich nicht unendlich oft angewendet werden.

6.4. Zusammenfassung

Die Kontrollstrategie von PGM/F berücksichtigt Abhängigkeiten zwischen den einzelnen Restrukturierungsregeln, um die heuristische Regelbasis auf eine PGM-Repräsentation effizient anzuwenden. Durch das Ausnutzen der Abhängigkeitsbeziehungen können insbesondere unnötige Optimierungsläufe vermieden werden, da gezielt jene Regeln angewendet werden können, die potentiell zu einer Restrukturierung einer PGM-Repräsentation führen. Ebenso ist gewährleistet, dass alle Restrukturierungsmöglichkeiten in einer PGM-Repräsentation in endlicher Zeit untersucht werden.

Darüber hinaus berücksichtigt die Kontrollstrategie von PGM/F Optimierungsgrenzen eines Workflowmodells, die bei einer Regelanwendung nicht überschritten werden dürfen, um dessen Ausführungssemantik zu erhalten. Dies wird durch Optimierungssphären gewährleistet, welche die Optimierungsgrenzen in einer PGM-Repräsentation reflektieren, und innerhalb derer alle Effekte von Regelanwendungen beschränkt bleiben.

7 Evaluierung

Im Mittelpunkt der Diskussion steht in diesem Kapitel die Analyse und Bewertung der Effektivität der heuristischen Restrukturierungsregeln von PGM/F. Dazu wird im nächsten Teilkapitel der Prototyp von PGM/F skizziert, auf dessen Grundlage die Messungen durchgeführt worden sind. Anschließend werden in Teilkapitel 7.2 die Messumgebung und die Szenarien vorgestellt, denen die Messungen zu Grunde liegen. Die Analyse der Messergebnisse erfolgt in Teilkapitel 7.3. Die Kapitel 7.4 und 7.5 beschäftigen sich mit der Fragestellung bezüglich der praktischen Anwendbarkeit und Erweiterbarkeit von PGM/F. Abschließend fasst Teilkapitel 7.6 die Ergebnisse dieses Kapitels noch einmal zusammen.

7.1. Prototyp

Auf Grundlage der in Abbildung 3.1 dargestellten Architektur und den in Kapiteln 3 bis 6 vorgestellten Konzepten zur heuristischen, regelbasierten Optimierung datenintensiver Workflows wurde als Machbarkeitsnachweis das PGM/F-System prototypisch implementiert [VSES08]. Die wichtigsten Details des Prototypen werden im Folgenden erläutert.

Die PGM-Repräsentation wird durch eine Transformationskomponente erzeugt, welche für die Abbildung einer Workflow- bzw. Stored-Procedure-Beschreibung nach PGM und umgekehrt verantwortlich ist, und dient als Eingabe für den PGM-Optimierer (siehe Abbildung 3.1). Anschließend wendet der PGM-Optimierer die gegebene Regelbasis gemäß der vorgegebenen Kontrollstrategie auf die vorliegende PGM-Repräsentation an. Danach wird die resultierende PGM-Repräsentation durch die Transformationskomponente wieder in eine Workflow- bzw. Stored-Procedure-Beschreibung gewandelt.

Dieser Optimierungslauf kann über eine Benutzerschnittstelle gesteuert werden. Es wird sowohl ein vollständiger als auch ein schrittweiser Optimierungslauf unterstützt. Im ers-

ten Fall wird eine gegebene Workflow- bzw. Stored-Procedure-Beschreibung in einem einzelnen Durchlauf optimiert. Als Ergebnis wird die optimierte Workflow- bzw. Stored-Procedure-Beschreibung direkt zurückgeliefert. Die einzelnen Optimierungsschritte werden dabei für einen Benutzer transparent auf der PGM-Ebene ausgeführt. Im Falle einer schrittweisen Optimierung können dagegen die einzelnen Optimierungsschritte vom Benutzer direkt an einer vorliegenden PGM-Repräsentation nachvollzogen werden. Hierfür wurde ein PGM-Viewer zur Visualisierung einer PGM-Repräsentation realisiert.

Zur Festlegung der Kontrollstrategie steht ein Konfigurationstool zur Verfügung, mit dem die Ausführungsreihenfolge der Restrukturierungsregeln und Regelklassen für eine Optimierungssphäre festgelegt werden kann. Dabei können sowohl sequentielle als auch iterative Ablauffolgen realisiert werden. Auf diese Weise kann die in Kapitel 6.2 vorgestellte Ablauffolge unmittelbar im Konfigurationstool definiert werden.

Die Implementierung des PGM/F-Systems basiert auf Konzepten aus dem Bereich der Model-Driven-Architecture (MDA) [OMGc], die mithilfe der Eclipse-Technologie [EF] und Java in der Version 1.4 umgesetzt wurden.

PGM kann als ein plattformunabhängiges Modell (platform-independent Model (PIM)) eines plattformspezifischen Modells (platform-specific Model (PSM)) einer BPEL/SQL- oder SQL/PSM-Beschreibung betrachtet werden. Zur Definition der Modelle für PGM, BPEL/SQL und SQL/PSM wurde das Eclipse-Modeling-Framework (EMF) eingesetzt, das auf dem Standard Meta-Object-Facility (MOF) [OMGb] beruht.

Des Weiteren wurden für die Beschreibungssprachen BPEL/SQL und SQL/PSM Transformationskomponenten realisiert, welche für die Abbildung eines EMF-Modells einer BPEL/SQL- bzw. SQL/PSM-Beschreibung auf ein entsprechendes PGM-EMF-Modell verantwortlich sind. Bei dieser Modell-zu-Modell-Transformation werden die einzelnen Komponenten des EMF-Modells einer BPEL/SQL- bzw. SQL/PSM-Beschreibung auf entsprechende Komponenten des EMF-Modells von PGM abgebildet. Dabei werden im EMF-Modell von PGM die zugehörigen Elemente des EMF-Modells einer BPEL/SQL- bzw. SQL/PSM-Beschreibung abgespeichert. Auf diese Weise ist es möglich, nach Anwendung der Regelbasis das EMF-Modell von PGM unter Berücksichtigung der Modell-zu-Modell-Transformationsregeln wieder zurück in ein äquivalentes EMF-Modell von BPEL/SQL- bzw. SQL/PSM zu überführen.

Die Restrukturierungsregeln sind als Modell-Transformationsregeln auf dem plattformunabhängigen EMF-Modell von PGM definiert. Bei deren Anwendung wird überprüft, ob in einem vorliegenden PGM-EMF-Modell Komponenten existieren, welche die Bedingung einer Restrukturierungsregel erfüllen. Ist dies der Fall, wird das EMF-Modell gemäß der in einer Restrukturierungsregel definierten Transformationsschritte umgebaut.

Die Modell-zu-Modell-Transformationsregeln und die Restrukturierungsregeln lassen sich mit dem Standard Query/View/Transformation (QVT) [OMG05] definieren. QVT bietet die Möglichkeit, deklarative Anfragen und Transformationen auf einem MOF-basierten Modell zu spezifizieren. Damit lassen sich sowohl die Abbildungsregeln zwischen dem EMF-Modell von PGM und den EMF-Modellen von BPEL/SQL und SQL/PSM als auch der Bedingungs- und Aktionsteil einer Restrukturierungsregel direkt umsetzen. Der Be-

dingungsteil einer Restrukturierungsregel kann als deklarative Anfrage auf einem MOF-basierten Modell definiert werden, die genau jene Elemente filtert, welche die Anfrage und damit den Bedingungsteil einer Restrukturierungsregel erfüllen. Der Aktionsteil einer Restrukturierungsregel ist als Folge von Transformationsschritten definiert, die auf den von der Anfrage zuvor gelieferten Elementen durchgeführt werden. Durch Verwendung von QVT kann die Komplexität der Implementierung der Modell-zu-Modell-Transformationsregeln und der Restrukturierungsregeln erheblich reduziert werden. Jedoch stand zum Zeitpunkt der Implementierung des Prototypen keine geeignete QVT-API für EMF zur Verfügung. Deshalb wurden alle Transformationsregeln direkt in Java umgesetzt, was einen erheblich höheren Implementierungsaufwand zur Folge hatte.

Der Optimierungablauf von PGM/F kann nun wie folgt konkretisiert werden: Liegt eine zu optimierende BPEL/SQL- bzw. SQL/PSM-Beschreibung vor, wird diese von der zugehörigen Tranformationskomponente deserialisiert und in ein EMF-Modell überführt. Anschließend wird das EMF-Modell gemäß der definierten Abbildungsregeln auf ein äquivalentes PMG-EMF-Modell abgebildet. Das PGM-EMF-Modell wird danach dem PGM/F-Optimierer übergeben. Der PGM/F-Optimierer wendet die vorliegenden Restrukturierungsregeln gemäß der konfigurierten Kontrollstrategie auf das PGM-EMF-Modell an. Das resultierende PGM-EMF-Modell wird der sprachspezifischen Transformationskomponente wieder übergeben, welche das PGM-EMF-Modell auf ein äquivalentes BPEL/SQL- bzw. SQL/PSM-EMF-Modell abbildet. In einem letzten Schritt erfolgt die Serialisierung des resultierenden EMF-Modells in seine ursprüngliche Form.

Das PGM/F-System wurde als Eclipse-Plug-In realisiert. Dadurch kann dem System neue Funktionalität über vordefinierte Erweiterungspunkte (extension points) hinzugefügt werden. Ein Erweiterungspunkt stellt eine Schnittstellenbeschreibung zur Verfügung, die von einer Plug-In-Erweiterung implementiert werden muss, damit die Plug-In-Erweiterung korrekt verarbeitet werden kann. Nach der Registrierung einer Erweiterung am Erweiterungspunkt kann die neue Funktionalität durch das Plug-In genutzt werden.

Das PGM/F-Plug-In besitzt zwei Erweiterungspunkte, die ein Hinzufügen neuer Restrukturierungsregeln bzw. Transformationskomponenten ermöglichen. Ebenso wäre ein Erweiterungspunkt sinnvoll, über welchen das PGM-EMF-Modell um neue Komponenten erweitert werden kann, die zur Anwendung neuer Restrukturierungsregeln erforderlich sind. Ein solcher Erweiterungspunkt wurde bei der Implementierung des Prototypen allerdings nicht berücksichtigt. Er kann dem PGM/F-Plug-In aber jederzeit hinzugefügt werden.

Der Erweiterungspunkt zur Definition neuer Restrukturierungsregeln verlangt unter anderem die Implementierung zweier Methoden zur Ausführung des Bedingungs- und Aktionsteils einer Restrukturierungsregel basierend auf dem EMF-Modell von PGM. Dazu ist es notwendig, die in einer Restrukturierungsregel definierten Bedingungen und Aktionen auf entsprechende Komponenten des EMF-Modells von PGM abzubilden.

Um eine neue Transformationskomponente zu definieren, ist zunächst ein EMF-Modell für die zu unterstützende Beschreibungssprache zu definieren. Basierend auf den Komponenten dieses EMF-Modells sind Modell-zu-Modell-Transformationsregeln zur Abbildung zwischen den EMF-Komponenten der Beschreibungssprache und PGM zu definieren.

Der WebSphere-Integration-Developer (WID) [IBMe] wurde als Modellierungswerkzeug für BPEL/SQL-Workflows verwendet. Da WID ebenfalls auf der Eclipse-Plattform basiert, konnte das PGM/F-System direkt als Plug-In in den WID integriert werden. Somit konnte das PGM/F-System in ein herstellerspezifisches Workflowmodellierungswerkzeug eingebettet und die praktische Einsatzfähigkeit des PGM/F-Systems belegt werden.

7.2. Szenarien und Messumgebung

Das für die Messungen eingesetzte System läuft unter dem Betriebssystem Windows 2003 und bestand aus einem Server mit zwei 3.2 GHz Prozessoren und 8 GB Hauptspeicher.

Als Workflowmanagementsystem wurde der WebSphere-Process-Server Version 6.0.1 [IBMf] von IBM verwendet. Die von den Workflows referenzierten Daten wurden von einer DB2 Version 9.5 und von einer Oracle 10g verwaltet. Zur Virtualisierung dieser beiden Datenbanksysteme wurde der WebSphere-Federation-Server Version 9.5 [IBMd] von IBM eingesetzt. Als Laufzeitumgebung für die in den Messszenarien aufgerufenen Web-Services wurde Apache-Tomcat Version 5.5 verwendet.

Die Effektivität der PGM-Restrukturierungsregeln wurde anhand verschiedener Szenarien untersucht:

- Die ersten Szenarien umfassen kleine Test-Workflows bzw. Stored-Procedures, die aus einer minimalen Anzahl von Aktivitäten bzw. Sprachkonstrukten bestehen, auf die jeweils eine Restrukturierungsregel angewendet werden kann. Dies ermöglicht eine isolierte Betrachtung der Effektivität der Restrukturierungsregeln.
- Im zweiten Szenario wird mit dem Beispiel-Workflow aus Abbildung 2.2 demonstriert, welche Leistungssteigerungen in einem homogenen bzw. heterogenen Optimierungsszenario durch die Verwendung der Restrukturierungsregeln möglich sind.

Allen Szenarien war ein Datenvolumen von bis zu einem GB zu Grunde gelegt. Wegen Einschränkungen in der BPEL/SQL-Implementierung des WebSphere-Process-Servers in der Version 6.0.1 war zum Zeitpunkt der Messungen eine Untersuchung mit größeren Datenvolumina nicht möglich. Des Weiteren wurden Indexstrukturen für alle Fremdschlüssel in den Tabellen erzeugt, und die Statistiken über Tabellen, Spalten und Indexstrukturen wurden vor den Testläufen aktualisiert. Vor einer Messung wurde der Inhalt des Bufferpools eines Datenbanksystems gelöscht. Jede Messung wurde jeweils 100 Mal wiederholt. Die daraus resultierenden durchschnittlichen Messwerte dienen als Grundlage für die im Folgenden präsentierten Ergebnisse.

7.3. Analyse der Messergebnisse

In diesem Abschnitt werden die Messergebnisse der einzelnen Szenarien vorgestellt. Dabei ist zu beachten, dass die Messergebnisse von den Charakteristika der in den Test-Workflows bzw. Stored-Procedure-Beschreibungen ausgeführten SQL-Anweisungen

und von den Fähigkeiten des Anfrageoptimierers eines ausführenden Datenbanksystems abhängen. Deshalb können die Messergebnisse nur einen Hinweis darauf liefern, welche Optimierungsgewinne mit den einzelnen Restrukturierungsregeln erreicht werden können. Dies bedeutet insbesondere, dass in anderen Optimierungsszenarien und Systemkonfigurationen die Anwendung der Restrukturierungsregeln zu höheren oder auch zu niedrigeren Leistungssteigerungen führen können.

7.3.1. Effektivität der einzelnen Restrukturierungsregeln

Abbildung 7.1 zeigt die Messergebnisse für die Test-Workflows, die für jede einzelne Restrukturierungsregel modelliert wurden. Die in einem Test-Workflow definierten SQL-Anweisungen kamen in diesem Testszenario dabei alle auf einem zentralisierten Datenbanksystem (DB2) zur Ausführung. Die in Abbildung 7.1 dargestellten Ergebnisse erlauben es, die Effekte der Restrukturierungsregeln isoliert voneinander zu betrachten. Hierzu werden die durchschnittlichen Laufzeiten der optimierten Varianten der Test-Workflows mit den durchschnittlichen Laufzeiten ihrer unoptimierten Varianten verglichen. Die durchschnittliche Laufzeit eines unoptimierten Test-Workflows liegt bei 100%.

Die *Web-Service-Pushdown*-Regeln (WSPD1 - WSPD4) wurden auf die vier von PGM/F unterstützten Typen von Web-Service-Operationen angewendet. Wie in Abbildung 7.1 zu sehen, hatte die Regelanwendung auf eine Web-Service-Operation vom Typ 1 (WSPD1) keinen größeren Optimierungseffekt. Dies zeigt, dass der Aufruf eines Web-Services durch ein Datenbanksystem in etwa die gleichen Kosten verursacht, wie ein entsprechender Web-Service-Aufruf durch ein Workflowsystem. Die Regelanwendungen auf die Web-Service-Operationen der Typen 2-4 (WSPD2-4) führten dagegen im Durchschnitt zu leichten Laufzeitverschlechterungen, die mit rund 7% bei einer Web-Service-Operation vom Typ 4 am größten waren. Die Laufzeitverschlechterungen können insbesondere mit den zusätzlichen Kosten begründet werden, welche durch die Übertragung der Ein- und Ausgabeparameter zwischen Workflow- und Datenebene entstehen, um einen Web-Service-Aufruf auf der Datenebene korrekt durchführen zu können. Dieses Messergebnis verdeutlicht, dass eine *Web-Service-Pushdown*-Regel nur ausgeführt werden sollte, wenn hierdurch die Ausführung weiterer Restrukturierungsregeln möglich ist. Wie in Kapitel 6 diskutiert, kann dies durch die Kontrollstrategie von PGM/F gewährleistet werden. Somit können Laufzeitverschlechterungen bei der heuristischen Optimierung eines datenintensiven Workflows durch die *Web-Service-Pushdown*-Regel ausgeschlossen werden.

Bei einer *Assign-Merging*-Regel (AM) waren nur geringe Leistungssteigerungen zu beobachten. Dies ist damit zu begründen, dass diese Regel lediglich eine Variablensubstitution durchführt. Folglich führt ein Datenbanksystem vor und nach der Ausführung dieser Regel dieselbe SQL-Anweisung aus. Die Einsparung, die durch den Wegfall der *Assign*-Aktivität auf der Workflowebene resultiert, fällt dagegen kaum ins Gewicht.

Obwohl die Regeln *Web-Service-Pushdown* und *Assign-Merging* nicht zur Leistungssteigerung beitragen, ist deren Verwendung wegen ihrer Abhängigkeitsbeziehungen (siehe Teilkapitel 6.1) für die Optimierung datenintensiver Workflows dennoch unverzichtbar.

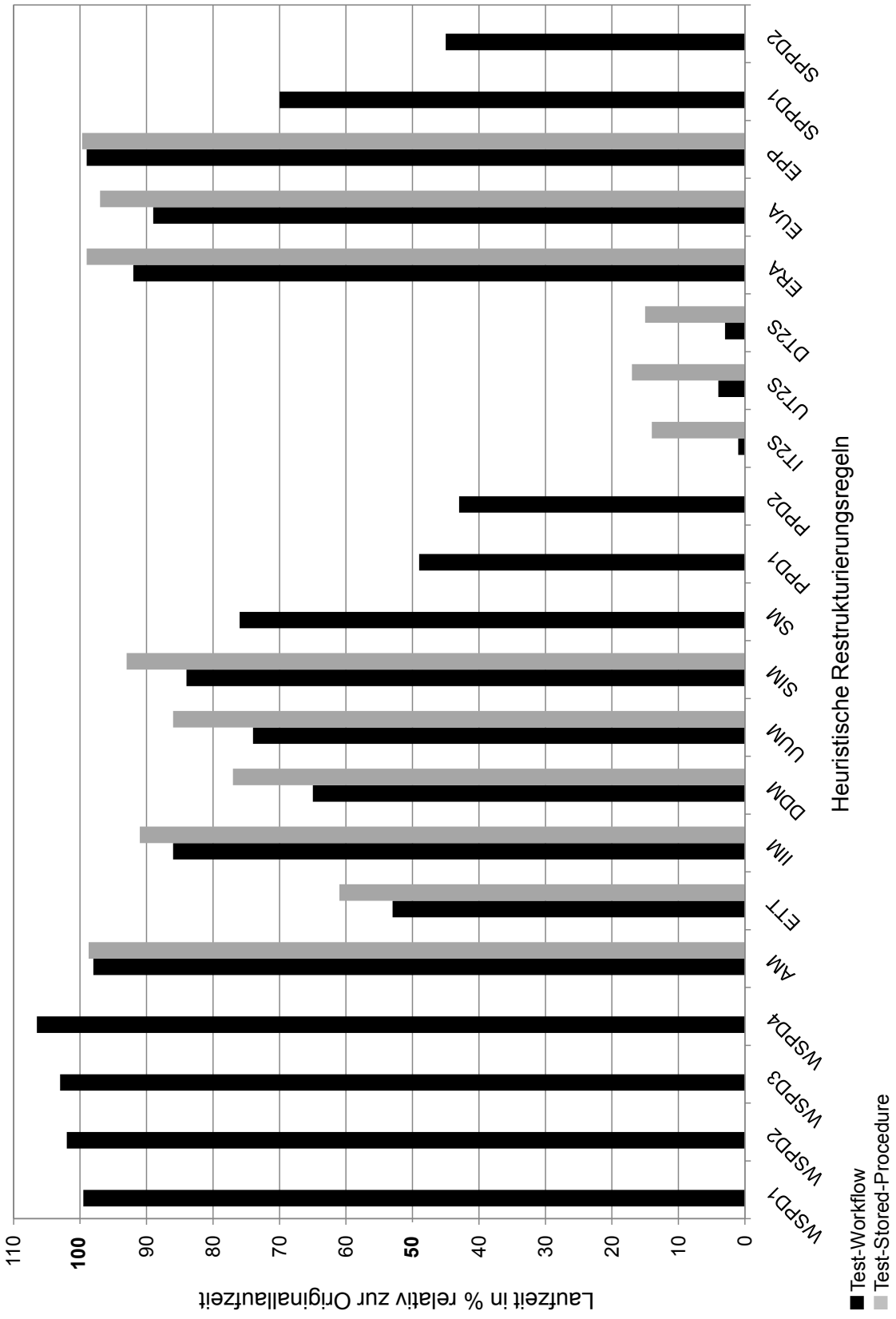


Abbildung 7.1.: Effektivität einzelner Restrukturierungsregeln

Bei den restlichen Restrukturierungsregeln waren dagegen überwiegend positive Optimierungseffekte festzustellen. Dabei wiesen die *SQL-Simplification*-Regeln geringe Optimierungsgewinne auf. Bei den Regeln *Eliminate-Redundant-Attributes* (ERA) bzw. *Eliminate-Unused-Attributes* (EUA) konnten Laufzeitsteigerungen um 10% nachgewiesen werden. Hierbei machte sich das Entfernen eines Attributs aus der Select-Klausel einer SQL-Anfrage positiv bemerkbar. Dagegen war in dem gemessenen Beispielszenario bei der *Eliminate-Redundant-Predicate*-Regel (ERP) kein Optimierungseffekt nachzuweisen.

Bei den *Update-Merging*-Regeln konnten deutlich höhere Leistungssteigerungen erreicht werden. Dies liegt vor allem daran, dass durch die Regelanwendungen auf die Ausführung jeweils einer der beiden Aktualisierungsoperationen verzichtet werden kann. In dem gemessenen Beispielszenario lag der Optimierungsgewinn bei einer *Insert-Insert-Merging*-Regel (IIM) bei rund 14%. Bei einer *Update-Update-Merging*-Regel (UUM) konnte die Laufzeit des Test-Workflows um rund 26% verbessert werden. Die größten Laufzeitgewinne von bis zu 35% wurden mit der *Delete-Delete-Merging*-Regel (DDM) erreicht.

Ähnliche Leistungssteigerungen konnten auch bei den *Query-Merging*-Regeln beobachtet werden. Hier fiel der Optimierungsgewinn bei einer *Select-Into-Merging*-Regel (SIM) mit über 15% am geringsten aus. Dies lässt sich damit begründen, dass die Ausführungskosten durch die Regelanwendung auf der Datenebene nicht wesentlich reduziert werden konnten. Der Laufzeitgewinn ergibt sich alleine daraus, dass auf der Workflowebene eine *SQL*-Aktivität eingespart und auf die Übertragung eines Ergebnistupels von der Daten- an die Workflowebene verzichtet werden kann. Bei der *Select-Merging*-Regel (SM) konnten deutlich höhere Leistungssteigerungen von über 25% erzielt werden. Dies vor dem Hintergrund, dass durch die Regelanwendung auf die Übertragung einer Ergebnismenge von der Daten- auf die Workflowebene und deren Materialisierung vollständig verzichtet werden kann. Die größten Optimierungsgewinne konnten bei den beiden *Predicate-Pushdown*-Regeln (PPD1, PPD2) festgestellt werden. Infolge eines hohen Selektivitätsfaktors der Prädikate, die in eine abhängige Anfrage verschoben wurden, konnten hier Leistungssteigerungen weit über 50% beobachtet werden.

Auch das Eliminieren einer temporären Tabelle führt zu einem Laufzeitgewinn. Zum einen entfällt der Aufwand für das Verwalten der Tabelle. Zum anderen können durch die Regelanwendung zuvor unabhängige SQL-Anweisungen vereinigt werden. Dies ermöglicht es dem Anfrageoptimierer des ausführenden Datenbanksystems, einen effizienteren Ausführungsplan zu identifizieren. Auf diese Weise war eine durchschnittliche Leistungssteigerung von knapp unter 50% möglich.

Ergebnis der Anwendung der *Tuple-to-Set*-Regeln war eine Laufzeitreduktion um mehrere Größenordnungen. Hierbei spielten folgende zwei Aspekte eine Rolle: Erstens, die Umwandlung einer tupel- in eine mengenbasierte Verarbeitung einer DML-Anweisung. Zweitens, weitergehende Optimierungsschritte durch das ausführende Datenbanksystem, die durch die Umwandlung der DML-Anweisung ermöglicht wurden. Diese Optimierungsschritte waren für die einzelnen, im ursprünglichen Test-Workflow iterativ ausgeführten SQL-Anweisungen nicht möglich. Die hohen Optimierungsgewinne konnten bei allen drei Regelvarianten beobachtet werden. Nach dem Gebrauch einer *Tuple-to-Set*-Regel betrugen die Laufzeiten der Test-Workflows im Durchschnitt nur noch knapp 2% ihrer

ursprünglichen Laufzeit. Das Ergebnis zeigt, dass der hohe Optimierungsgewinn unabhängig von der Art der DML-Anweisung erreicht werden kann. Dies ist vor allem darauf zurückzuführen, dass die Leistungssteigerungen auf der Reduzierung der Kosten zur Ausführung einer tupelbasierten DML-Anweisung innerhalb einer Schleife beruhen. Die unterschiedlichen Charakteristika der ausgeführten DML-Anweisungen fallen im Vergleich dazu kaum ins Gewicht.

Wie in Abbildung 7.1 illustriert, konnten auch für die *Stored-Procedure-Pushdown*-Regel Optimierungsgewinne nachgewiesen werden. Dabei wurden nacheinander eine Sequenz bestehend aus zwei bis zehn SQL-Anweisungen in eine Stored-Procedure ausgelagert. Damit war im besten Fall eine Leistungssteigerung von über 30% möglich (SPPD1). Die Verlagerung einer tupelbasierten DML-Anweisung innerhalb einer Schleife in eine Stored-Procedure brachte sogar einen Laufzeitgewinn von über 55% (SPPD2). Diese Messergebnisse unterstreichen, dass Datenverarbeitungsoperationen auf der Datenebene wesentlich effizienter ausgeführt werden können, als dies auf der Workflowebene möglich ist.

Aufgrund der zentralisierten Architektur der Messumgebung konnten für die Parallelisierungsregel keine aussagekräftigen Messungen durchgeführt werden. Folglich wurde diese Regel in Abbildung 7.1 nicht weiter berücksichtigt. Es kann aber davon ausgegangen werden, dass durch eine parallele Verarbeitung unabhängiger SQL-Anweisungen Optimierungsgewinne in mehreren Größenordnungen erreicht werden können.

In einer zweiten Testreihe wurden die Ausführungen der Test-Workflows und ihrer optimierten Varianten wiederholt. Dieses Mal wurden die darin eingebetteten SQL-Anweisungen auf einem föderierten Datenbanksystem (WebSphere-Federation-Server) ausgeführt. Ziel war es, zu untersuchen, wie sich eine Änderung der Organisation der Datenebene auf die Optimierungseffekte der einzelnen Restrukturierungsregeln auswirkt. Im wesentlichen konnten die in Abbildung 7.1 dargestellten Messergebnisse bestätigt werden, weshalb hier auf eine separate Abbildung verzichtet wurde. Im Durchschnitt fielen die Optimierungsgewinne etwas geringer aus als dies für das zentralisierte Datenbanksystem nachgewiesen werden konnte. Dies kann mit dem zusätzlichen Verarbeitungsaufwand auf der Föderationsschicht begründet werden, die dafür verantwortlich ist, eine SQL-Anweisung an die zu Grunde liegenden Datenbanksysteme weiterzuleiten und deren Ergebnisse zu einem Gesamtergebnis zu integrieren. Nur bei der *Insert-Insert-Merging*-Regel (IIM) wichen die Ergebnisse der beiden Testreihen vom beschriebenen Schema wesentlich ab. Hier konnte bei Verwendung des föderierten Datenbanksystems eine Leistungssteigerung von über 30% erzielt werden. Bei der Testreihe mit dem zentralisierten Datenbanksystem fielen die Leistungssteigerungen um über die Hälfte geringer aus. Dies zeigt, dass der Anfrageoptimierer des föderierten Datenbanksystems in der Lage war, einen besseren Ausführungsplan für eine kombinierte Insert-Anweisung zu finden. Insbesondere konnten hier Parallelisierungseffekte effektiver genutzt werden.

Die Test-Workflows wurden in einer dritten Testreihe in äquivalente Stored-Procedure-Beschreibungen umgeschrieben. Die darauf anwendbaren Restrukturierungsregeln führten zu den in Abbildung 7.1 dargestellten Ergebnissen. Wie zu sehen ist, sind auch in diesem Szenario Laufzeitverbesserungen in mehreren Größenordnungen möglich. Allerdings fielen die Optimierungsgewinne im Vergleich zu den Test-Workflows im Durchschnitt et-

was geringer aus. Dies kann damit begründet werden, dass bei der Optimierung einer Stored-Procedure Leistungssteigerungen alleine durch eine beschleunigte Ausführung der in einer Stored-Procedure eingebetteten SQL-Anweisungen erreicht werden können. Bei der Optimierung der Test-Workflows sind dagegen zusätzliche Laufzeitgewinne auf der Workflowebene wegen reduzierter Ausführungs- und Kommunikationskosten möglich.

7.3.2. Messergebnisse für das Beispielszenario

Im zweiten Teil der Experimente stehen die Optimierungseffekte im Mittelpunkt, die durch Anwendung der Regelbasis auf den Beispiel-Workflow in Abbildung 2.2 erzielt werden können. Hierzu wurden die Laufzeiten des Beispiel-Workflows nach einer homogenen und einer heterogenen, isolierten sowie nach einer heterogenen, kombinierten Optimierung gemessen. Zur Beobachtung von Skalierungseffekten wurden die Testläufe mit unterschiedlichen Datenvolumina zwischen 1K und 100K durchgeführt. Die Ergebnisse werden im Folgenden vorgestellt.

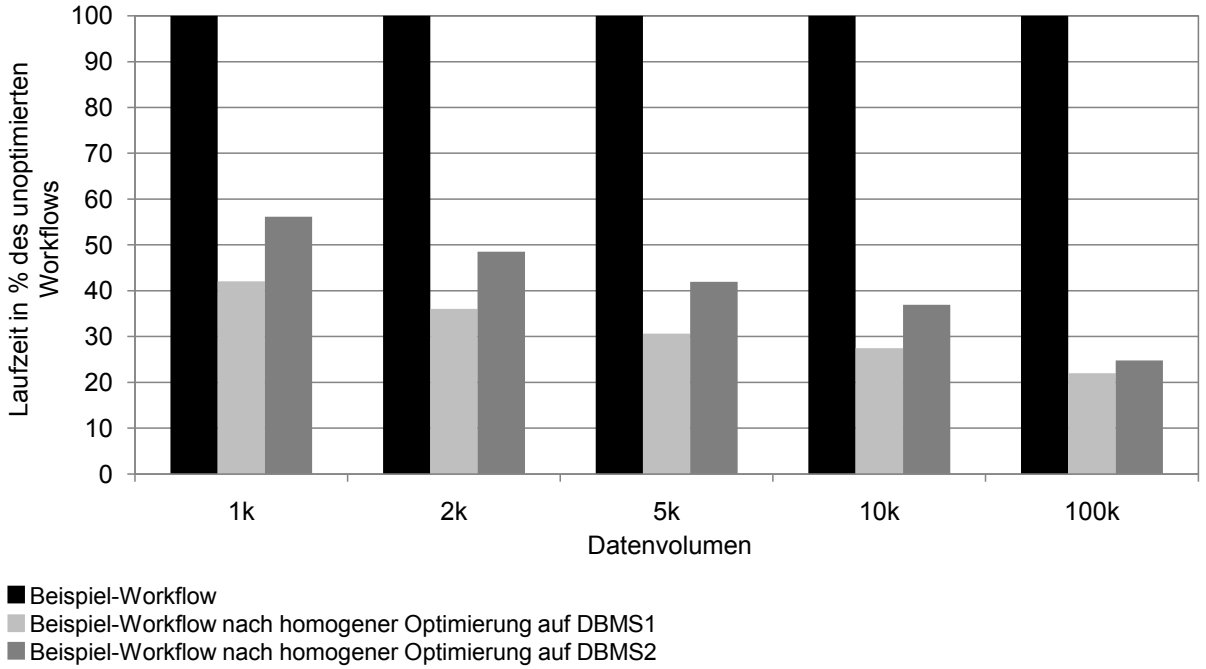


Abbildung 7.2.: Laufzeit nach homogener Optimierung

Bei der homogenen Optimierung des Beispiel-Workflows können drei Regeln auf die PGM-Repräsentation des Workflows angewendet werden (siehe auch Teilkapitel 3.2 bzw. Anhang E.1): *Web-Service-Pushdown*, *Select-Into-Merging* und *Insert-Tuple-to-Set*. In Abbildung 7.2 sind die Auswirkungen dieser Regelanwendungen auf die Laufzeit des Beispiel-Workflows veranschaulicht. Dabei wird die durchschnittliche Laufzeit des unoptimierten Beispiel-Workflows als 100% markiert. In Relation dazu steht die Laufzeit des optimierten Beispiel-Workflows, der jeweils auf einem zentralisierten Datenbanksystem (DB2 und Oracle) zweier verschiedener Hersteller ausgeführt wurde. Wie in der Abbildung gezeigt,

lässt sich die Laufzeit des Workflows durch eine homogene Optimierung auf bis zu ein Fünftel der ursprünglichen Laufzeit reduzieren. Dies zeigt, dass Leistungssteigerungen von mehreren Größenordnungen auch in komplexeren Szenarien möglich sind, und dass die Leistungssteigerungen größenteils unabhängig von der Kardinalität einer Tabelle bzw. vom zu Grunde gelegten Datenbanksystem erzielt werden können. Dies bestätigt die im vorherigen Abschnitt vorgestellten Messergebnisse. Eine genauere Analyse ergab, dass die Leistungssteigerungen hauptsächlich durch die *Insert-Tuple-to-Set*-Regel hervorgerufen werden. Hierzu war aber eine vorherige Anwendung der *Web-Service-Pushdown-* und *Select-Into-Merging*-Regel unerlässlich.

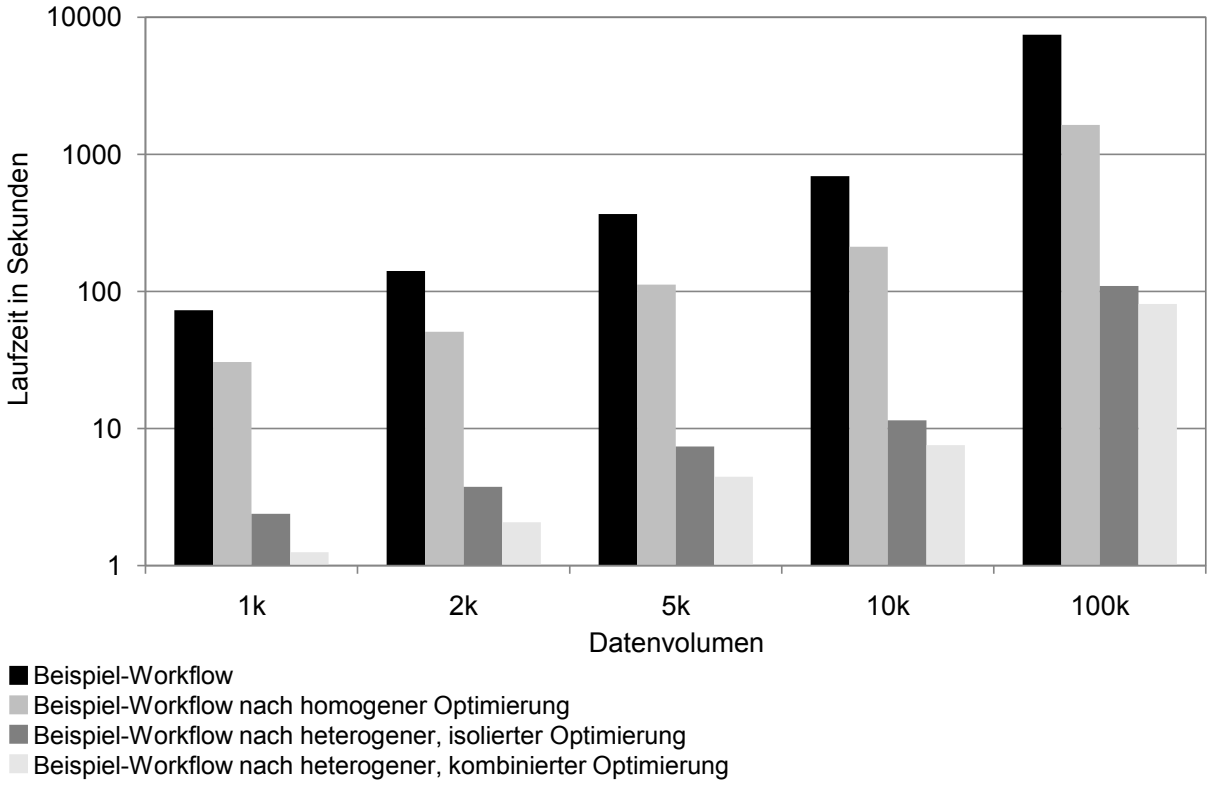


Abbildung 7.3.: Laufzeiten nach homogener und heterogener Optimierung

Wie Abbildung 7.3 zeigt, ließ sich die Laufzeit des Beispiel-Workflows weiter reduzieren, als die Regelbasis zusätzlich auf die PGM-Repräsentation der Stored-Procedure *PrepareApprovedOrders*, die aus dem Beispiel-Workflow heraus aufgerufen wird, angewendet wurde. In der Abbildung sind die Laufzeiten des Beispiel-Workflows in einer logarithmischen Skala im unoptimierten Zustand und nach einer homogenen bzw. heterogenen Optimierung dargestellt. Die zusätzlichen Laufzeitgewinne bei der heterogenen, isolierten Optimierung sind darauf zurückzuführen, dass die *Insert-Tuple-to-Set*-Regel auf die PGM-Repräsentation der Stored-Procedure *PrepareApprovedOrders* angewendet werden konnte (siehe auch Teilkapitel 3.2 bzw. Anhang E.2). Hierdurch verkürzte sich die Laufzeit des Beispiel-Workflows um mehrere Größenordnungen. Der größte Optimierungseffekt wurde bei einer heterogenen, kombinierten Optimierung erzielt. Hier war es möglich, eine *Eliminate-Temporary-Table*-Regel auf zwei SQL-Anweisungen anzuwen-

den, die ursprünglich getrennt voneinander in der Beschreibung des Beispiel-Workflows und der Stored-Procedure definiert waren (siehe auch Teilkapitel 3.2 bzw. Anhang E.3). Ein solches Optimierungsergebnis wäre ohne eine einheitliche Betrachtung des Beispiel-Workflows und der darin aufgerufenen Stored-Procedure mittels einer integrierten PGM-Repräsentation nicht möglich gewesen. Dieses Ergebnis zeigt, dass durch eine kombinierte Betrachtung aller Datenverarbeitungsoperationen über Sprachgrenzen hinweg bei der Optimierung eines datenintensiven Workflows zusätzliche Leistungssteigerungen in mehreren Größenordnungen möglich sind.

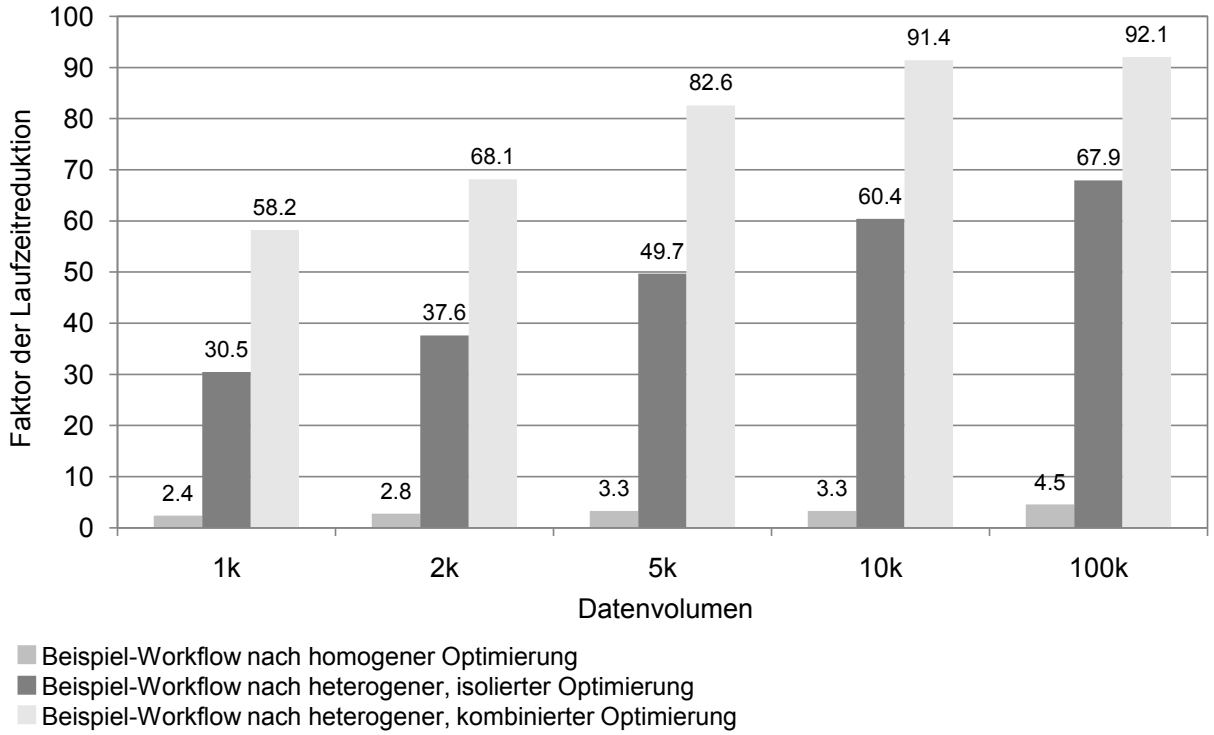


Abbildung 7.4.: Leistungssteigerungen durch homogene und heterogene Optimierung

Dieser Sachverhalt wird in Abbildung 7.4 verdeutlicht. Hervorgehoben sind noch einmal die Effekte der unterschiedlichen Optimierungsszenarien. Dargestellt sind jeweils die Faktoren, um welche die Laufzeit des ursprünglichen Beispiel-Workflows durch eine homogene bzw. heterogene Optimierung reduziert werden konnte. Wie in der Abbildung zu sehen ist, konnte die Ausführungszeit des Beispiel-Workflows durch eine homogene Optimierung um das bis zu Vierfache der ursprünglichen Laufzeit verkürzt werden. Höhere Optimierungsgewinne waren in diesem Beispielszenario trotz Verwendung einer *Insert-Tuple-To-Set*-Regel nicht möglich, da die Ausführungszeit des Beispiel-Workflows hauptsächlich durch die Laufzeit der Stored-Procedure *PrepareApprovedOrders* bestimmt wurde. Dies erklärt auch den großen Optimierungsgewinn, der bei einer heterogenen, isolierten Optimierung des Beispiel-Workflows erreicht werden konnte. Durch Auflösen des ineffizienten Schleifenkonstrukts in der Stored-Procedure konnte die Laufzeit des Beispiel-Workflows auf das bis zu 70-fache beschleunigt werden. Im Falle einer heterogenen, kombinierten Optimie-

rung war sogar eine Beschleunigung auf über das 90-fache der Laufzeit des unoptimierten Beispiel-Workflows möglich.

7.4. Praktische Anwendbarkeit von PGM/F

Im Rahmen dieser Arbeit wurden datenintensive Workflows unterschiedlicher Struktur und Komplexität aus verschiedenen Anwendungsgebieten untersucht, die Geschäftsprozesse implementieren (siehe beispielsweise [Mün06]). Dabei konnten bestimmte Datenverarbeitungsmuster identifiziert werden, die beim Zugriff und bei der Verarbeitung relationaler Geschäftsdaten eine wichtige Rolle spielen. Dazu gehören neben dem Abfragen bzw. Aktualisieren von Geschäftsdaten durch SQL-Anfragen bzw. DML-Anweisungen auch die Konfiguration relationaler Datenquellen durch DDL-Anweisungen, um beispielsweise temporäre Tabellen zu verwalten. Ebenso zählen hierzu der Aufruf von Stored-Procedures sowie die sequentielle Verarbeitung materialisierter Datenmengen direkt auf der Workflowebene. Die in Kapitel 5 vorgestellten Restrukturierungsregeln reflektieren Optimierungsstrategien, die auf Grundlage der Analyseergebnisse für die identifizierten Datenverarbeitungsmuster entwickelt wurden. Damit ist PGM/F in der Lage häufig auftretende Datenverarbeitungsmuster in datenintensiven Workflows zu unterstützen.

7.5. Erweiterbarkeit von PGM/F

Die Erweiterbarkeit von PGM/F ist eine wichtige Eigenschaft des Systems. So können PGM/F sowohl neue Beschreibungssprachen als auch neue Restrukturierungsregeln hinzugefügt werden.

Dabei spielt PGM eine entscheidende Rolle, da es eine sprachunabhängige Definition und Anwendung der Restrukturierungsregeln erlaubt. PGM wurde entwickelt, um Beschreibungssprachen abstrakt zu repräsentieren, welche die Einbettung von SQL-Aktivitäten bzw. SQL-Anweisungen in komplexe Kontrollflussstrukturen unterstützen. BPEL/SQL und SQL/PSM sind hierfür Beispiele aus zwei verschiedenen Anwendungsbereichen.

Mit dem PGM/F-System können alle Beschreibungssprachen optimiert werden, die auf PGM abgebildet werden können. Dafür ist es notwendig, das PGM/F-System um eine Transformationskomponente zu erweitern, welche die einzelnen Konstrukte einer Beschreibungssprache auf PGM abbildet und umgekehrt. Weitere Anpassungen am PGM/F-System sind nicht erforderlich.

Bei der Definition einer Transformationskomponente ist insbesondere darauf zu achten, dass alle für die Optimierung relevanten Konzepte einer Beschreibungssprache in einer PGM-Repräsentation explizit dargestellt werden. Dazu gehören insbesondere Optimierungssphären, Kontrollflussmuster sowie SQL-Anweisungen, Web-Service-Aufrufe und Variablenzuweisungen. Nur so kann das PGM/F-System das vorliegende Optimierungspotential einer Beschreibung erkennen und die Restrukturierungsregeln darauf korrekt anwenden. Je mehr Konzepte dabei von PGM von einer Beschreibungssprache unterstützt

werden, desto höher ist das vom PGM/F-System nutzbare Optimierungspotential für diese Beschreibungssprache.

Besitzt im umgekehrten Fall eine Beschreibungssprache ein Optimierungspotential, das bislang von PGM/F nicht genutzt wurde, kann das PGM/F-System um neue Restrukturierungsregeln erweitert werden. Hierzu kann es erforderlich sein, PGM um jene Aktivitäts-, Partner- oder Datentypen zu ergänzen, die das neue Optimierungspotential in einer PGM-Repräsentation offen legen. Anschließend sind die neuen Restrukturierungsregeln auf Basis von PGM zu definieren und zu implementieren. Aufgrund seiner Plug-in Architektur können dem PGM/F-System neue Restrukturierungsregeln sehr einfach hinzugefügt werden (siehe Kapitel 7.1). Im letzten Schritt muss die Ablaufreihenfolge zwischen den Restrukturierungsregeln rekonfiguriert werden, um eine neue Restrukturierungsregel in die Kontrollstrategie von PGM/F zu integrieren.

Des Weiteren vereinfacht das Konzept der Regelklassen die Definition neuer Restrukturierungsregeln erheblich. Wird eine Restrukturierungsregel einer existierenden Regelklasse neu hinzugefügt, müssen lediglich die regelspezifischen Teile der Restrukturierungsregel neu definiert werden. Eine Berücksichtigung der klassenspezifischen Teile ist dabei nicht mehr erforderlich, da sie für alle Restrukturierungsregeln einer Regelklasse gültig sind. Dies gilt auch für Aussagen bezüglich der Korrektheit neuer Restrukturierungsregeln.

Ebenso lassen sich die in dieser Arbeit entwickelten Optimierungsstrategien für SQL-Anweisungen auch auf nicht-SQL-basierten Datenbanksprachen, wie z.B. XQuery, leicht übertragen. Dazu müssen die SQL-spezifischen Teile einer Regelbeschreibung durch entsprechende Definitionen in der neuen Datenbanksprache ersetzt werden. Die Bedingungen bezüglich Kontrollfluss-, Daten- und Kommunikationsabhängigkeiten können dagegen direkt von einer SQL-spezifischen Regelbeschreibung übernommen werden.

7.6. Zusammenfassung

In diesem Kapitel wurde die Effektivität der heuristischen Restrukturierungsregeln von PGM/F untersucht. Dazu wurden unter Verwendung einer prototypischen Implementierung von PGM/F verschiedene Testläufe durchgeführt und analysiert. Die wichtigsten Ergebnisse dieser Testläufe können wie folgt zusammengefasst werden:

- Die Messergebnisse haben die in Kapitel 5 beschriebenen Optimierungseffekte für die einzelnen, heuristischen Restrukturierungsregeln größtenteils bestätigen können.
- Die größten Optimierungsgewinne konnten in den Testläufen bei der Regelklasse *Tuple-To-Set* festgestellt werden. Hohe Laufzeitsteigerungen waren ebenfalls mit den Regeln *Stored-Procedure-Pushdown*, *Predicate-Pushdown* und *Eliminate-Temporary-Table* möglich. Auch bei den *Update-Merging*- und den übrigen *Query-Merging*-Regeln konnten deutlich positive Optimierungseffekte nachgewiesen werden. Dagegen trugen die *SQL-Simplification*-Regeln nur in begrenztem Umfang zu einer Laufzeitverbesserung bei. Bei der *Assign-Merging*-Regel und den *Web-Service-Pushdown*-Regeln waren positive Optimierungseffekte kaum nachweisbar. Hier wa-

ren sogar Laufzeitverschlechterungen zu beobachten. Daraus folgt, dass die *Web-Service-Pushdown*-Regeln in einem heuristischen Optimierungsszenario nur in Kombination mit weiteren Restrukturierungsregeln zu positiven Laufzeiteffekten führen können. Diese Problematik wurde bereits in den Teilkapiteln 5.3 und 6.2 erörtert.

- Ebenso konnte durch die Testreihen gezeigt werden, dass durch die Restrukturierungsregeln Leistungssteigerungen für datenintensive Workflows und für Stored-Procedures gleichermaßen erzielt werden können. Bei Stored-Procedures fielen die Optimierungsgewinne allerdings geringer aus. Hier macht sich das höhere Optimierungspotential eines Workflowmodells bemerkbar. In einem Workflowmodell sind Laufzeitverbesserungen auf der Workflow- und Datenebene möglich. Bei einer Stored-Procedure können hingegen Laufzeitverbesserungen allein durch eine beschleunigte Ausführung der darin definierten SQL-Anweisungen erreicht werden.
- Des Weiteren spielte die Organisation der Datenebene für die Optimierungsgewinne eine untergeordnete Rolle. Ähnliche Leistungssteigerungen konnten sowohl bei einem zentralisierten als auch bei einem föderierten Datenbanksystem beobachtet werden. Allerdings fielen diese Leistungssteigerungen bei einem föderierten Datenbanksystem im Durchschnitt etwas geringer aus. Dies lässt sich mit dem zusätzlichen Verarbeitungsaufwand für eine SQL-Anweisung auf der Föderationsschicht begründen. Dagegen haben die Testläufe auch zeigen können, dass der Anfrageoptimierer eines föderierten Datenbanksystems in der Lage ist, Parallelisierungseffekte, die nach einer Regelanwendung in einer SQL-Anweisung entstehen können, effektiv zu nutzen. So fielen die Optimierungsgewinne bei einer *Insert-Insert-Merging*-Regel bei einem föderierten Datenbanksystem höher aus als bei einem zentralisierten.
- Neben der Effektivität der einzelnen Restrukturierungsregeln konnte durch die Testläufe gezeigt werden, dass Leistungssteigerungen in mehreren Größenordnungen ebenfalls in komplexeren Szenarien erzielt werden können. Die Laufzeit des Beispiel-Workflows aus Abbildung 2.2 konnte um mehrere Größenordnungen reduziert werden. Dies war unter anderem möglich, da die Regelbasis von PGM/F sowohl auf den Beispiel-Workflow als auch auf die darin aufgerufene Stored-Procedure-Beschreibung angewendet werden konnte. Dieses Ergebnis zeigt, dass zusätzliche Optimierungsgewinne in großem Umfang möglich sind, wenn neben einer homogenen auch eine heterogene Optimierung eines datenintensiven Workflows in Betracht gezogen wird.

Abschließend kann festgehalten werden, dass PGM/F die Optimierung wesentlicher Datenverarbeitungsmuster unterstützt, die in datenintensiven Workflows typischerweise auftreten können, und dass PGM/F aufgrund von PGM auf einfache Weise an neue Beschreibungssprachen und Restrukturierungsregeln angepasst werden kann.

8

Zusammenfassung und Ausblick

Dieses Kapitel fasst die Ergebnisse der vorliegenden Arbeit zusammen und gibt einen Ausblick auf weitere, ergänzende Forschungsarbeiten im Bereich der Optimierung datenintensiver Workflows.

8.1. Zusammenfassung

Der hier vorgestellte Ansatz befasst sich mit der Optimierung datenintensiver Workflows, wie sie beispielsweise mit BPEL/SQL modelliert werden können. Datenintensive Workflows verarbeiten große, relationale Datenmengen mit <sql> Aktivitäten, deren Definitionen einen integralen Bestandteil einer Workflowbeschreibung darstellen. Daraus ergibt sich eine neue Möglichkeit der Anfrageoptimierung: Suboptimale Datenverarbeitungsoperationen lassen sich in einer Workflowbeschreibung unter Berücksichtigung von Daten- und Kontrollflussabhängigkeiten derart restrukturieren, dass sie auf der Workflow- bzw. Datenebene wesentlich effizienter ausgeführt werden können. Diese Art der Anfrageoptimierung auf der Workflowebene lässt sich effektiv mit existierenden Ansätzen zur Anfrageoptimierung auf der Datenebene kombinieren. Dies kann insgesamt zu einer Verkürzung der Ausführungszeit eines datenintensiven Workflows führen.

In dieser wissenschaftlichen Abhandlung wurden Konzepte zur Realisierung eines heuristischen, regelbasierten SQL-Optimierers für datenintensive Workflows erörtert. Kern dieses Optimierers ist PGM. Dabei handelt es sich um eine generische, abstrakte Repräsentation für prozedural modellierte Workflowmodelle, auf welche eine verfügbare Regelbasis mittels einer gegebenen Kontrollstrategie zum Modellierungs- bzw. Deploymentzeitpunkt eines Workflowmodells angewendet wird. PGM erlaubt eine sprachunabhängige Definition und Anwendung der Restrukturierungsregeln. Dies lässt gleichermaßen die Anwendung des

Optimierungsansatzes sowohl auf BPEL/SQL-Workflows als auch auf Stored-Procedures zu. Auch eine kombinierte Optimierung von BPEL/SQL-Workflows zusammen mit den darin aufgerufenen Stored-Procedures mittels einer integrierten PGM-Repräsentation ist möglich. Ein wichtiges Konzept von PGM besteht darin, alle Informationen, die für eine effiziente Definition und Anwendung der Restrukturierungsregeln notwendig sind, explizit zur Verfügung zu stellen. Dies führt zu einer einfachen und kompakten Darstellung von PGM sowie zu einer effizienten Verarbeitung einer PGM-Repräsentation durch den Anfrageoptimierer. Darüber hinaus kann PGM leicht an Anforderungen neuer Restrukturierungsregeln angepasst werden. Hierfür steht ein generisches Typenkonzept zur Verfügung, das die Ableitung neuer Aktivitäts-, Partner- und Datentypen, welche für die Definition neuer Restrukturierungsregeln erforderlich sind, unterstützt.

Die Regelbasis, die auf eine PGM-Repräsentation angewendet wird, besteht aus Regeln, die auf existierenden und neuen Optimierungstechniken beruhen, die insbesondere Daten- und Kontrollflussabhängigkeiten zwischen den zu optimierenden SQL-Aktivitäten berücksichtigen. Damit lassen sich positive Optimierungseffekte sowohl auf der Workflow- als auch auf der Datenebene erzielen, die auch von der Organisation der Datenebene beeinflusst werden können.

Zu diesen Optimierungstechniken gehört unter anderem die Vereinfachung, die Kombination und die Parallelisierung von SQL-Anweisungen sowie die Verlagerung von SQL-Anweisungen in Stored-Procedures. Darüber hinaus kann durch die Einbettung von Web-Service-Aufrufen in SQL-Anfragen das Optimierungspotential einer PGM-Repräsentation erhöht werden, da hierdurch weitere SQL-spezifische Restrukturierungsregeln ausgeführt werden können. Ein weiterer Schwerpunkt der Regelbasis liegt in der Restrukturierung von DML-Anweisungen, die auf der Workflowebene unter Verwendung prozedural modellierter Datenverarbeitungsmuster tupelbasiert und somit ineffizient ausgeführt werden. Des Weiteren wurden in dieser Arbeit die Grenzen der heuristischen Anfrageoptimierung datenintensiver Workflows aufgezeigt. So existieren beispielsweise Regeln, die nur im Rahmen einer kostenbasierten Optimierungsstrategie auf eine PGM-Repräsentation angewendet werden können. Dazu zählt beispielsweise die Optimierung von Verbundoperationen, die Kombination verschiedenartiger DML-Anweisungen und der Einsatz von MQO-Techniken auf eine Menge von SQL-Anfragen.

Die Kontrollstrategie berücksichtigt Abhängigkeitsbeziehungen zwischen den einzelnen Restrukturierungsregeln, um immer genau jene Regeln anzuwenden, die potentiell zu einer Restrukturierung einer PGM-Repräsentation und somit zu positiven Optimierungseffekten führen können. Unnötige Optimierungsläufe können hierdurch vermieden werden. Ebenso ist mit dieser Kontrollstrategie gewährleistet, dass alle Restrukturierungsmöglichkeiten in einer PGM-Repräsentation gefunden werden. Des Weiteren stellt das Konzept der Optimierungssphären sicher, dass Optimierungsgrenzen eines zu Grunde liegenden Workflowmodells bei einer Regelanwendung nicht überschritten werden. Somit bleibt die ursprüngliche Ausführungssemantik eines Workflowmodells auch nach Anwendung der Regelbasis auf eine PGM-Repräsentation erhalten.

Die prototypische Implementierung des Anfrageoptimierers für datenintensive Workflows beweist die praktische Einsatzfähigkeit des hier vorgestellten Optimierungsansatzes. Zu-

dem belegen die auf diesem Prototypen basierenden Messungen, dass durch die Restrukturierungsregeln Laufzeitgewinne bis zu mehreren Größenordnungen möglich sind, und zwar unabhängig vom Herstellertypen eines Datenbanksystems.

Zusammenfassend lässt sich sagen, dass der hier vorgestellte Ansatz eine vielversprechende Grundlage für die Optimierung datenintensiver Workflows bildet. Darauf aufbauend sind eine Reihe von Möglichkeiten zur Weiterentwicklung denkbar, die weit über den hier vorgestellten Rahmen hinausreichen. Diese Weiterentwicklungen werden im nächsten Teilkapitel näher beschrieben.

8.2. Ausblick

Den in dieser Arbeit vorgestellten Optimierungsszenarien lagen die Workflowbeschreibungssprache BPEL/SQL und SQL/PSM zur Beschreibung von Stored-Procedures zu Grunde. Daraus ergibt sich unmittelbar die Fragestellung, inwieweit der vorgestellte Optimierungsansatz auf andere Workflow- bzw. Programmiersprachen, die ebenfalls eine direkte Einbettung von SQL-Funktionalität in komplexe Kontrollflussstrukturen erlauben, übertragen werden kann. Dabei ist insbesondere zu klären, welche Anpassungen und Erweiterungen am Prozessgraphenmodell vorzunehmen sind, um eine Unterstützung solcher Beschreibungssprachen zu ermöglichen.

Des Weiteren ist es sinnvoll, den Optimierungsansatz auch auf andere, nicht SQL-basierte Datenbanksprachen, wie z.B. XQuery, auszuweiten. Beispielsweise können durch eine direkte Einbettung von XQuery-Funktionalität in BPEL die Möglichkeiten der Datenverarbeitung in einem datenintensiven Workflow erweitert werden. Auch eine Berücksichtigung einer Datenbanksprache, wie XQSE [BCE+08], die es erlaubt, XQuery-Funktionalität mit komplexen Kontrollflussstrukturen zu kombinieren, ist in diesem Zusammenhang interessant. Auf diese Weise lässt sich ein heterogenes, kombiniertes Optimierungsszenario konstruieren, bei dem ein BPEL/XQuery-Workflow zusammen mit den darin aufgerufenen Datenverarbeitungsoperationen, die in XQSE definiert sind, optimiert wird.

In einem weiterführenden Optimierungsszenario können datenintensive Workflows betrachtet werden, die eine direkte Einbettung sowohl von SQL- als auch von XQuery-Funktionalität unterstützen. Mittlerweile erlauben es führende Datenbanksysteme, SQL- und XQuery-Funktionalität in einer einzelnen Datenbankanweisung zu kombinieren und auszuführen. Die pureXML-Technologie [IBMb] der DB2 Version 9.7 von IBM ist ein prominentes Beispiel hiervon. Eine solche Technologie eröffnet eine Vielzahl neuer Kombinationsmöglichkeiten, Datenverarbeitungsoperationen auf der Workflowebene über Sprach- und Technologiegrenzen hinweg zu optimieren.

Auch durch eine Veränderung der zu Grunde liegenden IT-Infrastruktur können sich neue Optimierungsmöglichkeiten auf der Workflowebene ergeben. Beispielsweise können in einem Workflowsystem Caches genutzt werden, um relationale Daten, die in einem Workflow referenziert werden, zwischenzuspeichern oder vorzuberechnen. Es bleibt zu untersuchen, wie solche Maßnahmen bei der Optimierung eines datenintensiven Workflowmodells gewinnbringend genutzt werden können.

Der Schwerpunkt des vorgestellten Ansatzes lag in der Optimierung datenintensiver Workflows, denen Geschäftsprozesse zu Grunde liegen. Eine interessante Frage ergibt sich hieraus, inwieweit dieser Optimierungsansatz auf andere Gruppen datenintensiver Workflows angewendet werden kann, und ob bei diesen Gruppen ähnliche positive Optimierungseffekte zu erwarten sind. Dies schließt sowohl die Gruppe der Scientific-Workflows als auch die Gruppe der ETL-Workflows mit ein.

Schließlich wurden in dieser Arbeit Restrukturierungsregeln skizziert, die nur im Rahmen einer kostenbasierten Optimierungsstrategie gewinnbringend angewendet werden können. Die Entwicklung einer solchen kostenbasierten Optimierungsstrategie steht noch aus. Eine wesentliche Herausforderung besteht darin, eine geeignete Kostenfunktion für die Datenverarbeitungsoperationen eines datenintensiven Workflows zu erstellen. Hierbei ist eine große Heterogenität bezüglich der Aktivitätstypen und Datenverarbeitungsoperationen zu berücksichtigen. Dieses Spektrum reicht von einfachen Zuweisungen und Web-Service-Aufrufen über SQL- und XQuery-Anweisungen bis hin zu Aufrufen benutzerdefinierter Prozeduren. Infolge dieser Heterogenität lassen sich bekannte Verfahren zur Kostenschätzung beispielsweise aus dem Bereich der föderierten Datenbanksysteme nicht einfach übertragen. Vielmehr müssen diese Verfahren entsprechend den zusätzlichen Anforderungen erweitert werden.

Neben der Definition einer geeigneten Kostenfunktion sind weitere Probleme zu lösen. Beispielsweise spielt für die Güte eines Optimierungsergebnisses auch die Aktualität der Statistiken in einem Datenbanksystem eine entscheidende Rolle, auf deren Grundlage eine Kostenabschätzung durchgeführt wird. Unter Berücksichtigung des langen Zeitraumes, über den ein Workflow hinweg insgesamt ausgeführt werden kann, ist es möglich, dass sich diese Statistiken entscheidend ändern können. Dies hat zur Folge, dass eine Optimierungsentscheidung laufend überprüft und gegebenenfalls revidiert werden muss. Die Revision einer Optimierungsentscheidung bedeutet, dass die hiermit verbundenen Änderungen unmittelbar in einem Workflowmodell vollzogen werden müssen, das bereits in einem Workflowsystem installiert ist bzw. von dem bereits Workflowinstanzen abgeleitet und ausgeführt werden. Für diese Problemstellungen müssen in der Zukunft geeignete Lösungsansätze erarbeitet werden.

A

Formale Definition von PGM

A.1. Das Prozessgraphenmodell (PGM)

Definition 1 (PGM-Graph)
Ein PGM-Graph G_{PGM} ist definiert als 6-Tupel (P, V, A, E_C, E_D, E_P) mit P als endlicher Menge von *Partnern*, V als endlicher Menge typisierter *Variablen*, A als endlicher Menge von *Aktivitäten* und E_C, E_D und E_P als endliche Mengen gerichteter *Kontrollfluss-*, *Daten-* und *Kommunikationsabhängigkeiten*.

A.2. Partner

Definition 2 (Partnertypen)
$PTypes = \{$`GENERIC, WS, RDBMS`$\}$ ist definiert als eine erweiterbare Menge unterstützter Partnertypen in PGM, die mit den Aktivitäten eines PGM-Graphen interagieren können. Dabei entspricht `GENERIC` einem generischen Partnertypen, `WS` einem Web-Service und `RDBMS` einem relationalen Datenbanksystem.

Definition 3 (Partner)
Sei P eine endliche Menge von Partnern, und sei $PTypes$ die erweiterbare Menge unterstützter Partnertypen in PGM. Dann ist ein *Partner* $p \in P$ definiert als Tupel $(pName, pType)$ mit einem eindeutigen Namen $pName$ und einem Partnertypen $pType \in PTypes$.

Definition 4 (Partner vom Typ `RDBMS`)
Sei $p \in P$ ein Partner vom Typ `RDBMS` ($pType(p) =$ `RDBMS`) und sei $PProperties =$ $\{$`WS-UDTF, SP`$\}$ eine erweiterbare Menge von Eigenschaften, welche diesem Partnertypen

zugeordnet werden können. Dann ist p definiert als Tupel $(pName, \texttt{RDBMS}, [pProperties])$ mit einem optionalen Element $pProperties \in PProperties$, das anzeigt, ob dieser Partnertyp Web-Service-Aufrufe mittels User-Defined-Functions ($pProperty = \texttt{WS-UDTF}$) bzw. die Ausführung SQL-basierter benutzerdefinierter Prozeduren (Stored-Procedures) ($pProperty = \texttt{SP}$) unterstützt. Im Folgenden steht $pProperty(p) \in PProperties$ für die von p unterstützten Eigenschaften.

A.3. Variablen

Definition 5 (Datentypen)
$DTypes = \{\texttt{SIMPLE, SET, GENERIC}\}$ ist definiert als eine erweiterbare Menge unterstützter, abstrakter Datentypen in PGM. Dabei entspricht ein `SIMPLE`-Datentyp allen Basisdatentypen, wie z.B. String, Integer usw., die auch von einem relationalen Datenbanksystem unterstützt werden. Ein `SET`-Datentyp repräsentiert eine relationale Tabelle, die auch in einer materialisierten Form vorliegen kann, und ein `GENERIC`-Datentyp ist ein generischer Datentyp, der alle restlichen Datentypen abdeckt.

Definition 6 (Variable)
Sei V eine endliche Menge von Variablen, und sei $DTypes$ eine erweiterbare Menge unterstützter, abstrakter Datentypen in PGM. Dann ist eine *Variable* $v \in V$ definiert als Tupel $(vName, dType)$ mit einem eindeutigen Namen $name$ und einem Datentypen $dType \in DTypes$. Des Weiteren gelten folgende Schreibweisen: $Name(v)$ und $DType(v)$ stehen für den Namen bzw. für den Datentypen einer Variablen $v \in V$. Eine Variable $v \in V$ mit $DType(v) = \texttt{SIMPLE} \mid \texttt{SET} \mid \texttt{GENERIC}$ wird abkürzend auch als `SIMPLE`-, `SET`- bzw. `GENERIC`-Variable bezeichnet.

Definition 7 (`SET`-Variable)
Sei $v \in V$ eine `SET`-Variable ($DType(v) = \texttt{SET}$). Dann ist v definiert als $(vName, \texttt{SET}, [T])$ mit dem Buchstaben `T` als optionalem Element, das anzeigt, ob es sich bei der von v repräsentierten Tabelle um eine temporäre Tabelle handelt. Des Weiteren gelten folgende Schreibweisen: $Columns(v)$ referenziert die Attribute einer Tabelle T, die durch v repräsentiert wird. Der Name bzw. der Datentyp eines Attributs $c \in Columns(v)$ wird durch $Name(c)$ und $DType(c)$ angezeigt. $TempTable(v) = \texttt{TRUE} \mid \texttt{FALSE}$ zeigt an, ob es sich bei v um eine temporäre Tabelle handelt oder nicht.

A.4. Aktivitäten

Definition 8 (Input Controlflowslot (ICS))
Sei $CTypes_{in} = \{\texttt{SIMPLE, AND, XOR}\}$ eine endliche Menge unterstützter ICS-Typen in PGM. Dann ist c^{in} der ICS einer Aktivität $a \in A$. Im Folgenden beschreibt $CType(c^{in}) \in CTypes_{in}$ den Typen und $CEdges(c^{in})$ die Menge der eingehenden Kontrollflusskanten

von c^{in}. Dabei gilt für die Anzahl der eingehenden Kontrollflusskanten in einen ICS c^{in} folgende Bedingung:

$$|CEdges(c^{in})| \begin{cases} = 1, \text{ für } CType(c^{in}) = \texttt{SIMPLE} \\ > 1, \text{ für } CType(c^{in}) = \texttt{AND} \mid \texttt{XOR} \end{cases} \tag{A.1}$$

Definition 9 (Output Controlflow Slot (OCS))
Sei C^{out} eine endliche Menge von OCSs einer Aktivität $a \in A$ und sei $CTypes_{out} = \{\texttt{SIMPLE, SPLIT}\}$ eine endliche Menge unterstützter OCS-Typen in PGM. Dann ist $c^{out} \in C^{out}$ ein OCS von a. Im Folgenden beschreibt $CType(c^{out}) \in CTypes_{out}$ den Typen und $CEdges(c^{out})$ die Menge der ausgehenden Kontrollflusskanten von c^{out}. Dabei gilt für die Anzahl der ausgehenden Kontrollflusskanten von c^{out} folgende Bedingung:

$$|CEdges(c^{out})| \begin{cases} = 1, \text{ für } CType(c^{out}) = \texttt{SIMPLE} \\ > 1, \text{ für } CType(c^{out}) = \texttt{SPLIT} \end{cases} \tag{A.2}$$

Definition 10 (Partnerslot)
Sei PS eine endliche Menge von Partnerslots einer Aktivität $a \in A$. Dann ist $ps \in PS$ ein Partnerslot von a, der mittels der Abbildung ν: $PS \mapsto P$ auf einen Partner $p \in P$ abgebildet wird.

Definition 11 (Dataslot)
Sei $V^x \subseteq V$ die endliche Menge aller Variablen in PGM, die von einer Aktivität $a \in A$ gelesen ($x = r$) bzw. geschrieben ($x = w$) werden. Dann ist D^x die endliche Menge von `Read-` bzw. `Write`-Dataslots von a, die mit V^x assoziiert ist. Dabei wird ein Dataslot $d^x = (v^x, counter) \in D^x$ mit einer Variablen $v^x \in V^x$ verknüpft, die von der Aktivität a gelesen ($x = r$) bzw. geschrieben ($x = w$) wird. Die Abbildung $\mu : D^x \mapsto V^x$ bildet einen Dataslot $d^x \in D^x$ auf seine assoziierte Variable $v^x \in V^x$ ab. Der Wert von *counter* zeigt an, wie oft v^x von der Aktivität a referenziert wird. Der Wert dieses Zählers wird im Folgenden mit der Funktion $Counter(d^x)$ bestimmt. Wird eine Variable von einer Aktivität sowohl gelesen als auch geschrieben, existiert für diese Variable sowohl ein `Read-` als auch ein `Write`-Dataslot.

Definition 12 (*Generic*-Aktivität)
Eine *Generic*-Aktivität $a \in A$ ist definiert als Tupel (`GENERIC`, PS, c^{in}, C^{out}, D^r, D^w), wobei `GENERIC` den Aktivitätstypen von a anzeigt, PS steht für die Menge von Partnerslots, c^{in} ist der ICS und C^{out} repräsentiert die Menge von OCSs von a. D^r und D^w stellen die endlichen Mengen von `Read-` und `Write`-Dataslots dar, die von den Variablenmengen V^r und V^w, die von a referenziert werden, abgeleitet werden.

Definition 13 (Selektionsprädikat σ)
Gegeben sei eine relationale Tabelle T mit den Spalten c_1, ..., c_n. Dann definiert der Ausdruck A_1 `AND/OR` A_2 ... `AND/OR` A_m mit $A_i = c_i \; op \; k_i \mid v_i$ ein Selektionsprädikat σ auf T. Dabei entspricht c_i einem Attribut von T, $op \in \{=, \neq, <, >, \leq, \geq\}$ stellt einen Vergleichsoperator dar und k_i entspricht einer Konstanten (Zahl oder Text) bzw. $v_i \in V$ entspricht einer Variablen, deren Werte zur Evaluierung eines Teilausdrucks A_i herangezogen werden.

Im Folgenden steht der Ausdruck $\sigma[V^r] \rightarrow$`BOOLEAN` für ein Selektionsprädikat σ, das Attributwerte eines Tupels einer Tabelle als Eingabe einliest und als Ergebnis den Wahrheitswert `TRUE` oder `FALSE` liefert. Dabei repräsentiert V^r die Variablenmenge (alle Variablen vom Typ `SIMPLE`), welche die Attributwerte eines Tupels zur Verfügung stellt.

Dagegen wendet der Ausdruck $\sigma[V^r](v') \rightarrow v''$ das Selektionsprädikat σ auf einer Tabelle an, welche durch die `SET`-Variable v' repräsentiert wird, und liefert als Ergebnis die `SET`-Variable v'', welche die selektierten Tupel aus v' zur Verfügung stellt. Dabei repräsentiert V^r alle Variablen, deren Werte in σ als Eingabeparameter gelesen werden.

Definition 14 (*XOR-Split*-Aktivität)
Eine *XOR-Split*-Aktivität ist definiert als Tupel (`XOR-Split`, $\emptyset$, c^{in}, C^{out}, D^r, $\emptyset$) mit folgenden Eigenschaften:
(1) $CType(c^{in}) = \text{SIMPLE}$
(2) $\forall c^{out} \in C^{out}$: $CType(c^{out}) = \text{SIMPLE}$
(3) $|C^{out}| > 1$

Definition 15 (*XOR-Split-Select*-Aktivität)
Eine *XOR-Split-Select*-Aktivität ist definiert als Tupel (`XOR-Split-Select`, $\emptyset$, c^{in}, $\{c^{out,1}, c^{out,2}\}$, D^r, $\emptyset$, $\sigma[V^r] \rightarrow$`BOOLEAN`) mit folgenden Eigenschaften:
(1) $CType(c^{in}) = CType(c^{out,1\,|\,2}) = \text{SIMPLE}$
(2) $v \in V^r \subseteq V$: $dType(v) =$ `SIMPLE` $\wedge\ \exists d^r \in D^r$: $DType(\mu(d^r)) = v$

Definition 16 (*XOR-Join*-Aktivität)
Eine *XOR-Join*-Aktivität ist definiert als Tupel (`XOR-Join`, $\emptyset$, c^{in}, $\{c^{out}\}$, $\emptyset$, $\emptyset$) mit folgenden Eigenschaften:
(1) $CType(c^{in}) = \text{XOR}$
(2) $CType(c^{out}) = \text{SIMPLE}$

Definition 17 (*AND-Split*-Aktivität)
Eine *AND-Split*-Aktivität ist definiert als Tupel (`AND-Split`, $\emptyset$, c^{in}, $\{c^{out}\}$, $\emptyset$, $\emptyset$) mit folgenden Eigenschaften:
(1) $CType(c^{in}) = \text{SIMPLE}$
(2) $CType(c^{out}) = \text{SPLIT}$

Definition 18 (*AND-Join*-Aktivität)
Eine *AND-Join*-Aktivität ist definiert als Tupel (`AND-Join`, $\emptyset$, c^{in}, $\{c^{out}\}$, $\emptyset$, $\emptyset$) mit folgenden Eigenschaften:
(1) $CType(c^{in}) = \text{AND}$
(2) $CType(c^{out}) = \text{SIMPLE}$

Definition 19 (*Set-Iterator*-Aktivität)
Eine *Set-Iterator*-Aktivität ist definiert als Tupel (`Set-Iterator`, $\emptyset$, c^{in}, $\{c^{out,1}, c^{out,2}\}$, $\{d^r\}$, D^w) mit folgenden Eigenschaften:
(1) $CType(c^{in}) = CType(c^{out,1\,|\,2}) = \text{SIMPLE}$
(2) $DType(\mu(d^r)) = \text{SET}$
(3) $\forall c \in Columns(\mu(d^r))$: $\exists v \in V \wedge \exists d^w \in D^W$:

$DType(v) = DType(c) \wedge Name(v) = Name(c) \wedge \mu(d^w) = v$
(4) $|D^w| = |Columns(\mu(d^r))|$.

Definition 20 (PGM-Repräsentation von SQL-Anweisungen)
Die PGM-Repräsentation einer SQL-Anweisung wird aus der ursprünglichen SQL-Anweisung durch Substitution aller darin referenzierter Tabellen sowie Ein- und Ausgabeparameter durch PGM-Variablen abgeleitet. Für diesen Substitutionsschritt gilt: Eine referenzierte Tabelle T wird auf eine `SET`-Variable abgebildet. Bei Eingabeparametern handelt es sich ausschließlich um skalare Werte. Darum wird ein Eingabeparameter durch eine `SIMPLE`-Variable substituiert. Bei Ausgabeparametern kann es sich um Tabellen und skalare Werte handeln. Deshalb können Ausgabeparameter sowohl von `SET`- als auch von `SIMPLE`-Variablen ersetzt werden. Der Name einer PGM-Variablen entspricht dabei dem Namen der zu ersetzenden Tabelle bzw. des zu ersetzenden Parameters. Für die PGM-Repräsentation einer SQL-Anweisung werden folgende abkürzende Schreibweisen verwendet, bei denen die Art einer SQL-Anweisung sowie deren Lese- und Schreibmenge explizit repräsentiert wird:

$$
\begin{array}{lcl}
S_Q & = & \texttt{SELECT}[\{v^w\}, V^r] \mid \texttt{SELECT-INTO}[V^w, V^r] \\
S_{SP} & = & \texttt{SPCALL}[V^w, V^r] \\
S_{DML} & = & \texttt{INSERT-SELECT}[\{v^w\}, V^r] \mid \texttt{UPDATE}[\{v^w\}, V^r] \mid \texttt{DELETE}[\{v^w\}, V^r] \mid \\
 & & \texttt{MERGE}[\{v^w\}, V^r] \\
S_{DDL} & = & \texttt{CREATE}[\{v^w\}, V^r] \mid \texttt{DROP}[\{v^w\}] \\
S_{SQL} & = & \texttt{SQL}[V^w, V^r]
\end{array}
$$

Die Schlüsselworte `SELECT`, `SELECT-INTO`, `SPCALL`, `INSERT-SELECT`, `UPDATE`, `DELETE`, `MERGE`, `CREATE` und `DROP` repräsentieren die Art der vorliegenden SQL-Anweisung. Die Lese- (V^r) bzw. Schreibmengen (V^w) einer SQL-Anweisung werden in eckiger Klammer angegeben. Diese beiden Mengen bestehen aus PGM-Variablen, welche die in einer SQL-Anweisung referenzierten Tabellen und Parameter ersetzt haben. Die Lesemenge einer SQL-Anfrage S_Q enthält alle Variablen, denen die in einer SQL-Anfrage referenzierten Tabellen und Eingabeparameter zu Grunde gelegt sind. Im Falle einer Select-Anweisung enthält die Schreibmenge genau eine `SET`-Variable, welche die Ergebnismenge der Select-Anweisung repräsentiert. Liegt dagegen eine Select-Into-Anweisung vor, besteht die Schreibmenge aus `SIMPLE`-Variablen, die in der Into-Klausel der Anweisung definiert sind, und in die ein Attributwert des von einer Select-Into-Anweisung selektierten Tupels gespeichert wird. Eine SQL Stored-Procedure S_{SP} kann sowohl Eingabe- als auch Ausgabeparameter, die durch entsprechende PGM-Variablen ersetzt werden, besitzen. PGM-Variablen, die auf Eingabeparametern basieren, werden der Lesemenge und PGM-Variablen, die auf Ausgabeparametern basieren, der Schreibmenge der Stored-Procedure zugeordnet. Die Schreibmenge einer DML-Anweisung S_{DML} besteht aus genau einer `SET`-Variablen, welche die Tabelle repräsentiert, auf die eine Insert-, Update-, Delete- oder Merge-Operation ausgeführt wird. Alle restlichen in einer DML-Anweisung referenzierten PGM-Variablen gehören zu deren Lesemenge. Analog zu einer DML-Anweisung besitzt die Schreibmenge einer DDL-Anweisung S_{DDL} genau eine `SET`-Variable, welche die Tabelle repräsentiert, auf die eine Create- oder Drop-Anweisung ausgeführt wird. Eine Create-Anweisung kann zudem eine Lesemenge enthalten. Dies ist möglich, da der SQL-Standard

es erlaubt, eine Tabelle von einer existierenden Tabelle abzuleiten, was durch eine entsprechende Unteranfrage innerhalb einer Create-Anweisung umgesetzt werden kann. Die in der Unteranfrage enthaltenen PGM-Variablen sind, wie im Falle einer SQL-Anfrage bereits diskutiert, Teil der Lesemenge von S_{DDL}. Schließlich verallgemeinert die Repräsentation von S_{SQL} die zuvor aufgelisteten SQL-Anweisungstypen. Sie wird in dieser Arbeit immer dann verwendet, wenn der Typ einer SQL-Anweisung keine Rolle spielt.

Für Anfragen bzw. DML-Anweisungen werden insbesondere für die Definition der Restrukturierungsregeln (siehe Abschnitt D) neben den oben genannten Kurzschreibweisen auch folgende Schreibweisen verwendet:

Für eine SQL-Anfrage S_Q:

$$\begin{aligned}\texttt{SELECT}[\{v^w\}, V^r] = [&\texttt{WITH} \quad with]\\ &\texttt{SELECT} \quad select\\ &\texttt{FROM} \quad from\\ &\texttt{WHERE} \quad where\\ &[\texttt{GROUP BY} \quad groupby]\\ &[\texttt{HAVING} \quad having]\\ &\rightarrow v^w\end{aligned}$$

$$\begin{aligned}\texttt{SELECT-INTO}[\{v_1^w, ..., v_n^w\}, V^r] = [&\texttt{WITH} \quad with]\\ &\texttt{SELECT} \quad select \quad \texttt{INTO } v_1^w, ..., v_n^w\\ &\texttt{FROM} \quad from\\ &\texttt{WHERE} \quad where\\ &[\texttt{GROUP BY} \quad groupby]\\ &[\texttt{HAVING} \quad having]\end{aligned}$$

Für eine DML-Anweisung S_{DML}:

$$\begin{aligned}\texttt{INSERT-SELECT}[\{v^w\}, V^r] = &\texttt{INSERT INTO } v^w\\ &[\texttt{WITH} \quad with]\\ &\texttt{SELECT} \quad select\\ &\texttt{FROM} \quad from\\ &\texttt{WHERE} \quad where\\ &[\texttt{GROUP BY} \quad groupby]\\ &[\texttt{HAVING} \quad having]\end{aligned}$$

$$\begin{aligned}\texttt{UPDATE}[\{v^w\}, V^r] = &\texttt{UPDATE TABLE } v^w\\ &\texttt{SET} \quad set\\ &\texttt{WHERE} \quad where\end{aligned}$$

$$\begin{aligned}\texttt{DELETE}[\{v^w\}, V^r] = &\texttt{DELETE FROM } v^w\\ &\texttt{WHERE } where\end{aligned}$$

$\texttt{MERGE}[\{v^w\}, V^r] = \texttt{MERGE INTO } v_l^w$
$\texttt{USING } using$
$\texttt{ON } on$
$\texttt{WHEN MATCHED THEN } set$
$[\texttt{WHEN NOT MATCHED THEN } insert]$

Definition 21 (*SQL*-Aktivität)
Eine *SQL*-Aktivität ist definiert als Tupel (SQL, $\{ps\}$, c^{in}, $\{c^{out}\}$, D^r,D^w, S_{SQL}) mit folgenden Eigenschaften:
(1) $PType(\nu(ps)) = RDBMS$
(2) $CType(c^{in}) = CType(c^{out}) =$ SIMPLE
(3) $\forall d^r \in D^r$: $DType(\mu(d^r)) =$ SIMPLE | SET
(4) $\forall d^w \in D^w$: $DType(\mu(d^w)) =$ SIMPLE | SET
(5) $S_{SQL} = \texttt{SQL}[V^w, V^r]$

Definition 22 (*SQL-Query*-Aktivität)
Eine *SQL-Query*-Aktivität ist definiert als Tupel (Query, $\{ps\}$, c^{in}, $\{c^{out}\}$, D^r,D^w, S_Q) mit folgenden Eigenschaften:
(1) $PType(\nu(ps)) = RDBMS$
(2) $CType(c^{in}) = CType(c^{out}) =$ SIMPLE
(3) $\forall d^r \in D^r$: $DType(\mu(d^r)) =$ SIMPLE | SET
(4) $DType(\mu(d^w)) =$ SET
(5) $S_Q = \texttt{SELECT}[\{v^w\}, V^r] \mid \texttt{SELECT-INTO}[V^w, V^r]$

Definition 23 (*SQL-SP*-Aktivität)
Eine *SQL-SP*-Aktivität ist definiert als Tupel (SP, $\{ps\}$, c^{in}, $\{c^{out}\}$, D^r, D^w, S_{SP}) mit folgenden Eigenschaften:
(1) $PType(\nu(ps)) = RDBMS$
(2) $CType(c^{in}) = CType(c^{out}) =$ SIMPLE
(3) $\forall d^{r\,|\,w} \in D^{r\,|\,w}$: $DType(\mu(d^{r\,|\,w})) =$ SIMPLE | SET
(4) $S_{SP} = \texttt{SPCALL}[V^w, V^r]$

Definition 24 (*SQL-DML*-Aktivität)
Eine *SQL-DML*-Aktivität ist definiert als Tupel (DML, $\{ps\}$, c^{in}, $\{c^{out}\}$, D^r, $\{d^w\}$, S_{DML}) mit folgenden Eigenschaften:
(1) $PType(\nu(ps)) = RDBMS$
(2) $CType(c^{in}) = CType(c^{out}) =$ SIMPLE
(3) $\forall d^r \in D^r$: $DType(\mu(d^r)) =$ SIMPLE | SET
(4) $DType(\mu(d^w)) =$ SET
(5) $S_{DML} = \texttt{INSERT-SELECT}[\{v^w\}, V^r] \mid \texttt{UPDATE}[\{v^w\}, V^r] \mid \texttt{DELETE}[\{v^w\}, V^r] \mid$
$\texttt{MERGE}[\{v^w\}, V^r]$

Definition 25 (*SQL-DDL*-Aktivität)
Eine *SQL-DDL*-Aktivität ist definiert als Tupel (DDL, $\{ps\}$, c^{in}, $\{c^{out}\}$, D^r, $\{d^w\}$, S_{DDL}) mit folgenden Eigenschaften:
(1) $PType(\nu(ps)) = RDBMS$

(2) $CType(c^{in}) = CType(c^{out}) = \text{SIMPLE}$
(3) $\forall d^r \in D^r$: $DType(\mu(d^r)) = \text{SIMPLE} \mid \text{SET}$
(4) $DType(\mu(d^w)) = \text{SET}$
(5) $S_{DDL} = \texttt{CREATE}[\{v^w\}, V^r] \mid \texttt{DROP}[\{v^w\}]$

Definition 26 (*Start-/End*-Aktivität)
Eine *Start-* bzw. *End*-Aktivität ist definiert als Tupel ($\texttt{START}|\texttt{END}$, $\emptyset$, c^{in}, $\{c^{out}\}$, $\emptyset$, $\emptyset$) mit folgender Eigenschaft: $CType(c^{in}) = CType(c^{out}) = \text{SIMPLE}$

Definition 27 (*Invoke*-Aktivität)
Eine *Invoke*-Aktivität ist definiert als Tupel ($\texttt{INVOKE}$, $\{ps\}$, c^{in}, $\{c^{out}\}$, D^r, D^w, $opName$) mit folgenden Eigenschaften:
(1) $PType(\nu(ps)) = WS$
(2) $CType(c^{in}) = CType(c^{out}) = \text{SIMPLE}$
(3) $opName$ entspricht dem Namen der aufgerufenen Web-Service-Operation

Definition 28 (*Assign*-Aktivität)
Eine *Assign*-Aktivität ist definiert als Tupel ($\texttt{ASSIGN}$, $\emptyset$, c^{in}, $\{c^{out}\}$, $\{d^r\}$, $\{d^w\}$) mit folgenden Eigenschaften:
(1) $CType(c^{in}) = CType(c^{out}) = \text{SIMPLE}$
(2) $DType(\mu(d^r)) = DType(\mu(d^w)) = \text{SIMPLE}$

Definition 29 (*Assign-Select*-Aktivität)
Eine *Assign-Select*-Aktivität ist definiert als Tupel ($\texttt{ASSIGN-SELECT}$, $\emptyset$, c^{in}, $\{c^{out}\}$, $\{d^r\} \cup D^{r'}$, $\{d^w\}$, $\sigma[V^{r'}](v') \rightarrow v'''$) mit folgenden Eigenschaften:
(1) $CType(c^{in}) = CType(c^{out}) = \text{SIMPLE}$
(2) $v', v''' \in V$: $dType(v') = dType(v''') = \texttt{SET} \wedge \mu(d^r) = v' \wedge \mu(d^w) = v'''$
(3) $\forall v'' \in V^{r'} \subset V$: $dType(v'') = \texttt{SIMPLE} \wedge \exists d^{r'} \in D^{r'}$: $\mu(d^{r'}) = v''$

A.5. Kontrollflussmuster

Definition 30 (Element eines Kontrollflussmusters)
$Elements = \{\texttt{GENERIC}, \texttt{ASSIGN}, \texttt{INVOKE}, \texttt{SQL}, \texttt{SPHERE}, \texttt{SEQ}, \texttt{ALT}, \texttt{PAR}, \texttt{LOOP}\}$ ist definiert als eine erweiterbare Menge von Aktivitäten bzw. Kontrollflussmustern, die als Element eines Kontrollflussmusters (siehe Definitionen 31 bis 36) definiert werden können. Um die Definition eines Kontrollflussmusters zu vereinfachen, gilt im Folgenden für ein Element $e_i \in Elements$ eine abstrakte Repräsentation, bei der ϵ_i als Black-Box mit lediglich einem ICS (c_i^{in}) bzw. OCS (c_i^{out}) dargestellt wird. Weitere Details werden nicht berücksichtigt. Handelt es sich bei ϵ_i um ein Kontrollflussmuster, repräsentieren c_i^{in} bzw. c_i^{out} den ICS der ersten bzw. die OCSs der letzten ausgeführten Aktivität im Kontrollflussmuster, die in der Black-Box Repräsentation von ϵ_i selbst nicht repräsentiert werden.

Definition 31 ($\texttt{SPHERE}$-Kontrollflussmuster)
Ein $\texttt{SPHERE}$-Kontrollflussmuster (auch Optimierungssphäre genannt) ist definiert als Menge $\{a_i, \epsilon_j, a_k\}$ mit $\epsilon_j \in Elements$ und $a_i, a_k \in A$: $aType(a_i) = \texttt{START}$ und $aType(a_k) = \texttt{END}$.

Ferner gelten in einem SPHERE-Kontrollflussmuster folgende Kontrollflussabhängigkeiten:
(1) $CEdges(c_i^{out}) = \{e_C^{i,j}\} = CEdges(c_j^{in})$
(2) $CEdges(c_j^{out}) = \{e_C^{j,k}\} = CEdges(c_k^{in})$

Definition 32 (SEQ-Kontrollflussmuster)
Ein SEQ-Kontrollflussmuster ist definiert als Menge $\{\epsilon_{i_1} \ldots \epsilon_{i_n}\}$ $(n > 1)$ mit $\epsilon_j \in Elements$ und folgenden Kontrollflussabhängigkeiten:
(1) $CEdges(c_{i_1}^{out}) = \{e_C^{i_1,i_2}\} = CEdges(c_{i_2}^{in})$
$\vdots$
(n-1) $CEdges(c_{i_{n-1}}^{out}) = \{e_C^{i_{n-1},i_n}\} = CEdges(c_{i_n}^{in})$

Definition 33 (ALT-Kontrollflussmuster)
Ein ALT-Kontrollflussmuster ist definiert als Menge $\{a_i\ \epsilon_{j_1} \ldots \epsilon_{j_n},\ a_k\}$ $(n > 1)$ mit $\epsilon_{j_x} \in Elements$ $(x = 1...n)$ und $a_i, a_k \in A$: $aType(a_i) =$ XOR-Split und $aType(a_k) =$ XOR-Join. In einem ALT-Kontrollflussmuster gelten folgende Kontrollflussabhängigkeiten:
(1) $\cup_{x=1..n} CEdges(c_i^{out,x}) = \{e_C^{i,j_1}, \ldots, e_C^{i,j_n}\} = CEdges(c_{j_1}^{in}) \cup \ldots \cup CEdges(c_{j_n}^{in})$
$\vdots$
(n+1) $CEdges(c_{j_1}^{out}) \cup \ldots \cup CEdges(c_{j_n}^{out}) = \{e_C^{j_1,k}, \ldots, e_C^{j_n,k}\} = CEdges(c_k^{in})$

Definition 34 (PAR-Kontrollflussmuster)
Ein PAR-Kontrollflussmuster ist definiert als Menge $\{a_i\ \epsilon_{j_1} \ldots \epsilon_{j_n},\ a_k\}$ $(n > 1)$ mit $\epsilon_{j_x} \in Elements$ $(x = 1...n)$ und $a_i, a_k \in A$: $aType(a_i) =$ AND-Split und $aType(a_k) =$ AND-Join. Ferner gelten in einem PAR-Kontrollflussmuster folgende Kontrollflussabhängigkeiten:
(1) $CEdges(c_i^{out}) = \{e_C^{i,j_1}, \ldots, e_C^{i,j_n}\} = CEdges(c_{j_1}^{in}) \cup \ldots \cup CEdges(c_{j_n}^{in})$
$\vdots$
(n+1) $CEdges(c_{j_1}^{out}) \cup \ldots \cup CEdges(c_{j_n}^{out}) = \{e_C^{j_1,k}, \ldots, e_C^{j_n,k}\} = CEdges(c_k^{in})$

Definition 35 (LOOP-Kontrollflussmuster (1. Variante))
Die erste Variante eines LOOP-Kontrollflussmusters ist definiert als Menge $\{a_i,\ a_j,\ \epsilon_k,\ \epsilon_l\}$ mit $\epsilon_k, \epsilon_l \in Elements$ und $a_i, a_j \in A$: $aType(a_i) =$ XOR-Join, $aType(a_j) =$ XOR-Split | Set-Iterator und $C_j^{out} = \{c_j^{out,1}, c_j^{out,2}\}$. Ferner gelten in der ersten Variante eines LOOP-Kontrollflussmusters folgende Kontrollflussabhängigkeiten:
(1) $CEdges(c_i^{out}) = \{e_C^{i,j}\} = CEdges(c_j^{in})$
(2) $CEdges(c_j^{out,1}) = \{e_C^{j,k}\} = CEdges(c_k^{in})$
(3) $CEdges(c_j^{out,2}) = \{e_C^{j,l}\} = CEdges(c_l^{in})$
(4) $CEdges(c_k^{out}) = \{e_C^{k,i}\} \subset CEdges(c_i^{in})$

Definition 36 (LOOP-Kontrollflussmuster (2. Variante))
Die zweite Variante eines LOOP-Kontrollflussmusters ist definiert als Menge $\{a_i,\ \epsilon_j,\ a_k,\ \epsilon_l\}$ mit $\epsilon_j, \epsilon_l \in Elements$ und $a_i, a_k \in A$: $aType(a_i) =$ XOR-Join, $aType(a_k) =$ XOR-Split und $C_k^{out} = \{c_k^{out,1}, c_k^{out,2}\}$. Ferner gelten in der zweiten Variante eines LOOP-Kontrollflussmusters folgende Kontrollflussabhängigkeiten:
(1) $CEdges(c_i^{out}) = \{e_C^{i,j}\} = CEdges(c_j^{in})$
(2) $CEdges(c_j^{out}) = \{e_C^{j,k}\} = CEdges(c_k^{in})$

(3) $CEdges(c_k^{out,1}) = \{e_C^{k,i}\} \subset CEdges(c_i^{in})$
(4) $CEdges(c_k^{out,2}) = \{e_C^{k,l}\} = CEdges(c_l^{in})$

A.6. Daten-, Kontrollfluss- und Kommunikationsabhängigkeiten

Definition 37 (Vorgänger-Relation)
Die Vorgänger-Relation $<$ definiert eine partielle Ordnung auf der Menge der Aktivitäten in einem PGM-Graphen. Dabei bedeutet $a_i < a_j$: Aktivität a_i wird vor Aktivität a_j ausgeführt bzw. die Ausführung von a_i muss abgeschlossen sein, bevor die Ausführung von a_j beginnt. Wenn die Relation $a_i < a_j$ besteht, ist a_i *Vorgänger* von a_j bzw. a_j ist *Nachfolger* von a_i. Wegen der transitiven Eigenschaft gilt zudem: $(a_i < a_j) \wedge (a_j < a_k) \Rightarrow a_i < a_k$. Außerdem ist Aktivität a_i *direkter Vorgänger* von a_j (geschrieben $a_i << a_j$), wenn gilt: $a_i < a_j$ und $\nexists a_k$: $a_i < a_k \wedge a_k < a_j$, d.h. zwischen a_i und a_j wird keine andere Aktivität ausgeführt. Entsprechend ist ein *unmittelbarer Nachfolger* einer Aktivität definiert.

Definition 38 (Kontrollflussabhängigkeitskante)
Seien $\Gamma^{in} = \bigcup_{a_i \in A} \{c_i^{in}\}$ und $\Gamma^{out} = \bigcup_{a_i \in A} C_i^{out}$ die endlichen ICS- bzw. OCS-Mengen aller Aktivitäten in A. Und sei $E_C \subseteq \Gamma^{out} \times \Gamma^{in}$ die endliche Menge gerichteter Kontrollflussabhängigkeitskanten. Dann ist eine *Kontrollflussabhängigkeitskante* $e_C^{i,j} \in E_C$ definiert als Tupel (c_i^{out}, c_j^{in}). Dabei ist $c_i^{out} \in C_i^{out}$ ein OCS einer Quellaktivität $a_i \in A$, und c_j^{in} ist ein ICS einer Zielaktivität $a_j \in A$. Eine Kontrollflusskante $e_C^{i,j} \in E_C$ existiert in einem PGM-Graphen genau dann, wenn gilt: $a_i << a_j$, d.h. Aktivität a_i ist direkter Vorgänger von Aktivität a_j.

Definition 39 (Kontrollflusspfad)
Ein *Kontrollflusspfad* in PGM ist eine geordnete Sequenz $P_{i_1,i_n} = (a_{i_1}, ..., a_{i_n})$ von n Aktivitäten mit der Startaktivität a_{i_1} und der Zielaktivität a_{i_n} mit folgender Eigenschaft: $\forall a_{i_s}, a_{i_{s+1}} \in P_{i,j}, 1 \leq s < n$: $\exists\, e_C^{i_s,i_{s+1}} \in E_C$, d.h. es existiert eine Kontrollflussabhängigkeitskante zwischen a_{i_s} und $a_{i_{s+1}}$.

Definition 40 (Datenabhängigkeitskante)
Sei $\Delta = \bigcup_{a_i \in A}(D_i^w \cup D_i^r)$ eine endliche Menge von Dataslots aller Aktivitäten in A. Und sei $E_D \subseteq \Delta \times \Delta$ die endliche Menge gerichteter Datenabhängigkeitskanten. Dann ist eine *Datenabhängigkeitskante* $e_{D_{x,y}}^{i,j} \in E_D$ definiert als Tupel (d_i^x, d_j^y). Dabei ist $d_i^x \in \Delta$ ein Dataslot der Quellaktivität $a_i \in A$, und $d_j^y \in \Delta$ ist ein Dataslot einer Zielaktivität $a_j \in A$. Eine Datenabhängigkeitskante $e_{D_{x,y}}^{i,j} \in E_D$ existiert, wenn beide Dataslots mit derselben Variablen assoziiert sind, d.h. $\mu(d_i^x) = \mu(d_j^y)$, und mindestens einer der beiden Dataslots ein `Write`-Dataslot ist. Somit können Schreib-Schreib- $(x = w, y = w)$, Schreib-Lese- $(x = w, y = r)$ und Lese-Schreib-Datenabhängigkeiten $(x = r, y = w)$ explizit repräsentiert werden. Des Weiteren wird gefordert, dass ein Kontrollflusspfad $P_{i,j}$ zwischen beiden Aktivitäten a_i und a_j existiert. Ferner gibt es keine andere Aktivität a_k auf dem Kontrollflusspfad $P_{i,j}$, die den Wert der Variablen ändert, der von a_i zuvor geschrieben wurde, d.h. $\nexists a_k \in P_{i,j}$: $\exists (d_i^x, d_k^w) \in E_D$: $\mu(d_i^x) = \mu(d_k^w)$. Die erste Datenabhängigkeit in

einem PGM-Graphen beginnt immer bei einer Variablen $v \in V$, die mit dem Dataslot der ersten Aktivität verbunden ist, welche diese Variable referenziert.

Definition 41 (Exklusive Schreib-Lese-Datenabhängigkeit)
Zwischen zwei Aktivitäten a_i, $a_j \in A$ existiert eine *exklusive* Schreib-Lese-Datenabhängigkeit $e^{i,j}_{D^X_{wr}} \in E_D$ mit $\mu(d^w_i) = \mu(d^r_j) = v \in V$ genau dann, wenn gilt: $\forall e^{i,k}_{D_{wr}} \in E_D$ mit $\mu(d^w_i) = \mu(d^r_k) = v \in V$: $k = j$. Mit anderen Worten: Neben Aktivität a_j gibt es keine weitere Aktivität $a_k \in A$, die von dem Wert einer Variablen $v \in V$ abhängt, die von Aktivität a_i geschrieben wurde.

Definition 42 (Kommunikationsabhängigkeitskante)
Sei $\Psi = \bigcup_{a_i \in A} PS_i$ die endliche Menge von Partnerslots aller Aktivitäten in A, und sei P die endliche Menge aller Partner. Ferner sei $E_P \subseteq \Psi \times P$ die endliche Menge gerichteter Kommunikationskanten. Dann ist eine *Kommunikationsabhängigkeitskante* $e^{i,j}_P \in E_P$ definiert als Tupel (ps_i, p_j). Dabei ist $ps_i \in PS_i$ ein Partnerslot einer Aktivität $a_i \in A$ und $p_j \in P$ ist ein mit a_i interagierender Partner.

B

XML-Syntax von PGM

B.1. PGM-Graph

```
<pgm>

 <partners>
  (<partner name="pName" pType="GENERIC|WS" dependencies="cdIDs" /> |
   <partner name="pName" pType="RDBMS" pProperties= ("WS-UDTF, SP")?
                                                dependencies="cdIDs" />)+
 </partners>

 <variables>?
  <variable name="vName" vType="GENERIC|SET|SIMPLE" />+
 </variables>

 <pgmLogic>
  <source />
  <optimizationSphere>
            ⋮
  </optimizationSphere>
  <sink />
 </pgmLogic>

 <communicationDependencies>?
  <communicationDependency id="cdID" partner="pName" partnerSlot="psID" />+
 </communicationDependencies>

 <controlFlowDependencies>
```

```
  <controlFlowDependency id="cfdID" ocs="ocsID" ics="icsID" />+
 </controlFlowDependencies>

 <dataDependencies>?
  <dataDependency id="ddID" ddType="D|WW|WR|RW" exclusive="True|False"
     (variable="vName" dataSlot="dsID") | (dataSlot="dsID" dataSlot="dsID") />+
 </dataDependencies>

</pgm>
```

B.2. Aktivitätstypen

```
< !-- GENERIC ------------------------------------------------------------ -- >

<generic>

 <partnerSlots>?
  <partnerSlot id="psID" dependency="cdID" />+
 </partnerSlots>

 <inputControlFlowSlot id="icsID" icsType="SIMPLE|XOR|AND" dependencies="cfdIDs" />

 <outputControlFlowSlots>?
  <outputControlFlowSlot id="ocsID" ocsType="SIMPLE|SPLIT" dependencies="cfdIDs" />+
 </outputControlFlowSlots>

 <readDataSlots>?
  <readDataSlot id="dsID" variable="vName" dependencies="ddIDs"
                                           counter="positiveInteger" />+
 </readDataSlots>

 <writeDataSlots>?
  <writeDataSlot id="dsID" variable="vName" dependencies="ddIDs"
                                           counter="positiveInteger" />+
 </writeDataSlots>

</generic>


< !-- INVOKE ---------------------------------------------------------- -- >

<invoke>

 <partnerSlots>
  <partnerSlot id="psID" dependency="cdID" />
 </partnerSlots>
```

```
 <inputControlFlowSlot id="icsID" icsType="SIMPLE" dependency="cfdID" />

 <outputControlFlowSlots>
  <outputControlFlowSlot id="ocsID" ocsType="SIMPLE" dependency="cfdID" />
 </outputControlFlowSlots>

 <readDataSlots>?
  <readDataSlot id="dsID" variable="vName" dependencies="ddIDs"
                                           counter="positiveInteger" />+
 </readDataSlots>

 <writeDataSlots>?
  <writeDataSlot id="dsID" variable="vName" dependencies="ddIDs"
                                           counter="positiveInteger" />+
 </writeDataSlots>

 <operation name="opName" />

</invoke>


< !-- ASSIGN ------------------------------------------------------------ -- >

<assign>

 <inputControlFlowSlot id="icsID" icsType="SIMPLE" dependency="cfdID" />

 <outputControlFlowSlots>
  <outputControlFlowSlot id="ocsID" ocsType="SIMPLE" dependency="cfdID" />
 </outputControlFlowSlots>

 <readDataSlots>
  <readDataSlot id="dsID" variable="vName" dependencies="ddIDs" counter="1" />
 </readDataSlots>

 <writeDataSlots>
  <writeDataSlot id="dsID" variable="vName" dependencies="ddIDs" counter="1" />
 </writeDataSlots>

</assign>


< !-- ASSIGN-SELECT ----------------------------------------------------- -- >

<assignSelect>

 <inputControlFlowSlot id="icsID" icsType="SIMPLE" dependency="cfdID" />
```

```
 <outputControlFlowSlots>
  <outputControlFlowSlot id="ocsID" ocsType="SIMPLE" dependency="cfdID" />
 </outputControlFlowSlots>

 <readDataSlots>
  <readDataSlot id="dsID" variable="vName" dependencies="ddIDs"
                                                counter="positiveInteger" />+
 </readDataSlots>

 <writeDataSlots>
  <writeDataSlot id="dsID" variable="vName" dependencies="ddIDs" counter="1" />
 </writeDataSlots>

 <sigma>?
  <predicate>
   <column name="columnName" />
   <operation type= "<" | ">" | "=" | "≠" | "≤" | "≥" />
   <value constant="constantStringOrIntegerValue" | variable="vName" />
  </predicate>
  (<conjunction type="AND"|"OR" /> <predicate> ... </predicate>)+
 </sigma>

</assignSelect>


< !-- SQL ------------------------------------------------------------- -- >

<sql type="Query|DML|DDL|SP" >

 <partnerSlots>
  <partnerSlot id="psID" dependency="cdID" />
 </partnerSlots>

 <inputControlFlowSlot id="icsID" icsType="SIMPLE" dependency="cfdID" />

 <outputControlFlowSlots>
  <outputControlFlowSlot id="ocsID" ocsType="SIMPLE" dependency="cfdID" />
 </outputControlFlowSlots>

 <readDataSlots>?
  <readDataSlot id="dsID" variable="vName" dependencies="ddIDs"
                                                counter="positiveInteger" />+
 </readDataSlots>

 <writeDataSlots>?
  <writeDataSlot id="dsID" variable="vName" dependencies="ddIDs"
                                                counter="positiveInteger" />+
```

```
 </writeDataSlots>

 <sqlStatement>
  <sqlStatementText>
   (<sqlStatementTextFragment text="sqlText" /> |
            <variable name="vName" associatedDataSlot="dsID" />)+
  </sqlStatementText>
 </sqlStatement>

</sql>


< !-- START ------------------------------------------------------------ -- >

<start>

 <inputControlFlowSlot id="icsID" icsType="SIMPLE" dependency="cfdID" />

 <outputControlFlowSlots>
  <outputControlFlowSlot id="ocsID" ocsType="SIMPLE" dependency="cfdID" />
 </outputControlFlowSlots>

</start>


< !-- END -------------------------------------------------------------- -- >

<end>

 <inputControlFlowSlot id="icsID" icsType="SIMPLE" dependency="cfdID" />

 <outputControlFlowSlots>
  <outputControlFlowSlot id="ocsID" ocsType="SIMPLE" dependency="cfdID" />
 </outputControlFlowSlots>

</end>


< !-- EXCEPTION -------------------------------------------------------- -- >

<exception>

 <readDataSlots>?
  <readDataSlot id="dsID" variable="vName" dependencies="ddIDs"
                                           counter="positiveInteger" />+
 </readDataSlots>

 <writeDataSlots>?
```

```
  <writeDataSlot id="dsID" variable="vName" dependencies="ddIDs"
                                              counter="positiveInteger" />+
 </writeDataSlots>

</exception>


< !-- AND-SPLIT ------------------------------------------------------ -- >

<andSplit>

 <inputControlFlowSlot id="icsID" icsType="SIMPLE" dependency="cfdID" />

 <outputControlFlowSlots>
  <outputControlFlowSlot id="ocsID" ocsType="SPLIT" dependencies="cfdIDs" />
 </outputControlFlowSlots>

</andSplit>


< !-- AND-JOIN ------------------------------------------------------- -- >

<andJoin>

 <inputControlFlowSlot id="icsID" icsType="AND" dependencies="cfdIDs" />

 <outputControlFlowSlots>
  <outputControlFlowSlot id="ocsID" ocsType="SIMPLE" dependency="cfdID" />
 </outputControlFlowSlots>

</andJoin>


< !-- XOR-SPLIT ------------------------------------------------------ -- >

<xorSplit>

 <inputControlFlowSlot id="icsID" icsType="SIMPLE" dependencies="cfdIDs" />

 <outputControlFlowSlots>
  <outputControlFlowSlot id="ocsID" ocsType="SIMPLE" dependency="cfdID" />+
 </outputControlFlowSlots>

 <readDataSlots>?
  <readDataSlot id="dsID" variable="vName" dependencies="ddIDs"
                                              counter="positiveInteger" />+
 </readDataSlots>
```

```
</xorSplit>

< !-- XOR-SPLIT-SELECT ------------------------------------------------ -- >

<xorSplitSelect>

 <inputControlFlowSlot id="icsID" icsType="XOR" dependencies="cfdIDs" />

 <outputControlFlowSlots>
  <outputControlFlowSlot id="ocsID" ocsType="SIMPLE" dependency="cfdID" />
  <outputControlFlowSlot id="ocsID" ocsType="SIMPLE" dependency="cfdID" />
 </outputControlFlowSlots>

 <readDataSlots>?
  <readDataSlot id="dsID" variable="vName" dependencies="ddIDs"
                                              counter="positiveInteger" />+
 </readDataSlots>

 <sigma>
     ⋮
 </sigma>

</xorSplitSelect>

< !-- XOR-JOIN ---------------------------------------------------------- -- >

<xorJoin>

 <inputControlFlowSlot id="icsID" icsType="XOR" dependencies="cfdIDs" />

 <outputControlFlowSlots>
  <outputControlFlowSlot id="ocsID" ocsType="SIMPLE" dependency="cfdID" />
 </outputControlFlowSlots>

</xorJoin>

< !-- SET-ITERATOR ------------------------------------------------------ -- >

<setIterator>

 <inputControlFlowSlot id="icsID" icsType="SIMPLE" dependencies="cfdIDs" />

 <outputControlFlowSlots>
  <outputControlFlowSlot id="ocsID" ocsType="SIMPLE" dependency="cfdID" />
```

```
  <outputControlFlowSlot id="ocsID" ocsType="SIMPLE" dependency="cfdID" />
 </outputControlFlowSlots>

 <readDataSlots>
  <readDataSlot id="dsID" variable="vName" dependencies="ddIDs" counter="1" />
 </readDataSlots>

 <writeDataSlots>
  <writeDataSlot id="dsID" variable="vName" dependencies="ddIDs" counter="1" />+
 </writeDataSlots>

</setIterator>
```

B.3. Kontrollflussmuster

```
< !-- OPTIMIZATION-SPHERE ------------------------------------------------ -- >

<optimizationSphere>

 <start> ... </start>
 <exception> ... </exception>?
 ELEMENT
 <end> ... </end>

</optimizationSphere>


< !-- SEQUENCE ------------------------------------------------------------ -- >

<sequence>

 ELEMENT+

</sequence>


< !-- PARALLEL ---------------------------------------------------------- -- >

<parallel>

 <andSplit> ... </andSplit>
 <pPath>ELEMENT</pPath>
 <pPath>ELEMENT</pPath>+
 <andJoin> ... </andJoin>
```

```
</parallel>


< !-- ALTERNATIVE ------------------------------------------------------ -- >

<alternative>

 <xorSplit> ... </xorSplit>
  <aPath>ELEMENT</aPath>
  <aPath>ELEMENT</aPath>+
 <xorJoin> ... </xorJoin>

</alternative>


< !-- LOOP ------------------------------------------------------------- -- >

<loop>

 <xorJoin> ... </xorJoin>
 <xorSplit> ... </xorSplit>
 ELEMENT
|
 <xorJoin> ... </xorJoin>
 <setIterator> ... </setIterator>
 ELEMENT
|
 <xorJoin> ... </xorJoin>
 ELEMENT
 <xorSplit> ... </xorSplit>

</loop>


< !-- ELEMENT ---------------------------------------------------------- -- >
```

ELEMENT entspricht einem der folgenden XML-Elemente:

```
 - <generic>
 - <assign>
 - <invoke>
 - <sql>
 - <optimizationSphere>
 - <sequence>
 - <alternative>
 - <parallel>
 - <loop>
```

C

Transformationstabellen

C.1. PGM - BPEL/SQL

PGM	BPEL/SQL
PARTNER (WS)	<partnerLink name="NCName" partnerRole="NCName" ... />
PARTNER (RDBMS)	<variable name="BPELVarName" type="tDataSource"/>
PARTNER (GENERIC)	<partnerLink ... />
VARIABLE (SET)	<variable name="BPELVarName" type="tSet"| "tSetRef."/>
VARIABLE (SIMPLE)	<variable name="BPELVarName" type="simpleXSDType"/>
VARIABLE (GENERIC)	<variable ... />
σ	<variable name="var" type="tSet"/> σ_{XPATH}(var) = A_1(var) AND/OR ... AND/OR A_n(var) mit A_i(var) = \$var/$RowSet/Row/c_i$ op k_i | v_i (i=1...n) mit $op \in \{=, \neq, <, >, \leq, \geq\}$, k_i = Konstante (Zahl oder Text), v_i = Variable vom Typ "simpleXSDType"
ASSIGN	<variable name="var1" type="simpleXSDType"/> <variable name="var2" type="simpleXSDType"/> ... <assign> <copy> <from variable="var1"/> <to variable="var2"/> </copy>

	</assign>
ASSIGN-SELECT	<variable name="var1" type="tSet"/> <variable name="var2" type="tSet"/> ... <assign> <copy> <from variable="var1"> <query queryLanguage="XPath">σ_{XPATH}(var)</query> </from> <to variable="var2"/> </copy> </assign>
INVOKE	<variable name="in1" type="simpleXSDType"|"tSet"/> ... <variable name="inN" type="simpleXSDType"|"tSet"/> ... <variable name="out1" type="simpleXSDType"|"tSet"/> ... <variable name="outM" type="simpleXSDType"|"tSet"/> ... <invoke partnerLink=... portType=... operation=...*> <toParts> <toPart part="NCName" fromVariable="in1"/> ... <toPart part="NCName" fromVariable="inN"/> </toParts> <fromParts> <fromPart part="NCName" toVariable="out1"/> ... <fromPart part="NCName" toVariable="outM"/> </fromParts> </invoke> * WS-Operation muss über SOAP/HTTP aufgerufen werden können
SQL-Query	<retrieveSet> <statement> *Query* </statement> </retrieveSet>
SQL-DML	<sql> <statement> *DML statement* </statement> </sql>

SQL-DDL	<sql> <statement> *DDL statement* </statement> </sql>
SQL-SP	<sql> <statement> *Stored Procedure Call* </statement> </sql>
GENERIC	<receive>, <reply>, <invoke>, <assign>, <empty>
Optimierungssphäre	<scope> ... <faultHandlers>?...</faultHandlers> *activity* </scope>
SEQ	<sequence> ... </sequence>
PAR	<flow> <sequence> ... </sequence> ... <sequence> ... </sequence> </flow>
ALT Variante: XOR-Split/XOR-Join	<if> ... </if>

ALT
Variante:
XOR-Split-Select/
XOR-Join
	<variable name="var1" type="tSet"/>
	...
	<if>
	 <condition expressionLanguage="XPath">
	  σ_{XPATH}(var1)
	 </condition>
	 activity
	 <else>
	  activity
	 </else>
	</if>

LOOP Variante: XOR-Join/ Set-Iterator	<variable name="varSet" type="tSet"/> <variable name="varC1" type="simpleXSDType"/> ... <variable name="varCN" type="simpleXSDType"/> ... <forEach counterName="k" parallel="no"> <startCounterValue>0</startCounterValue>

	<finalCounterValue> count($(varSet)/*RowSet*/*Row*)-1 </finalCounterValue> <scope> <sequence> <assign> <copy> <from variable="varSet"> <query queryLanguage="XPath"> $varSet/*RowSet*/*Row*[*k*]/c_1 </query> </from> <to variable="varC1"/> </copy> ... <copy> <from variable="varSet"> <query queryLanguage="XPath"> $varSet/*RowSet*/*Row*[*k*]/c_n </query> </from> <to variable="varCN"/> </copy> </assign> ... </sequence> </scope> </forEach>
LOOP Variante: XOR-Join/ XOR-Split	<while>, <repeatUntil>, <forEach>

Tabelle C.1.: PGM - BPEL/SQL Transformationstabelle

C.2. PGM - SQL/PSM

PGM	SQL/PSM
PARTNER	-
VARIABLE (SET)	DECLARE CURSOR statement
VARIABLE (SIMPLE)	DECLARE variable statement
GENERIC	Compound SQL statement (BEGIN ... END) mit UNDO handler, LEAVE, ITERATE, CLOSE
ASSIGN	SET variable statement
SQL-Query	Query, SET variable statement, OPEN
SQL-DML	DML statement
SQL-DDL	DDL statement
SQL-SP	Stored Procedure Call
Set-Iterator	FETCH
Optimierungssphäre	Compound SQL statement (BEGIN ... END) mit EXIT/CONTINUE handler
SEQUENCE	Compound SQL statement (BEGIN ... END) ohne handler
ALT Variante: XOR-Split/XOR-Join	CASE
LOOP Variante: XOR-Join/XOR-Split	LOOP, WHILE, REPEAT
SQL-Query + LOOP Variante: XOR-Join/Set-Iterator	FOR

Tabelle C.2.: PGM - SQL/PSM Transformationstabelle

D

Formale Definition der heuristischen Regelbasis von PGM/F

D.1. Algorithmus isMergePossible?

Durch Anwendung einer *Activity-Merging*-Regel werden zwei Datenverarbeitungsoperationen o_i und o_j zweier Aktivitäten a_i und a_j zu einer einzelnen Datenverarbeitungsoperation o_{i+j} kombiniert. Wie bereits in Teilkapitel 5.4.1 ausführlich diskutiert, ist diese Kombination nur dann korrekt, wenn die ursprüngliche Ausführungssemantik von o_i sowie existierende Datenabhängigkeiten zwischen a_i und anderen Aktivitäten ($\neq a_j$) in einer PGM-Repräsentation erhalten werden können.

Algorithm 3 *isMergePossible?*

Require: $a_i, a_j \in A \wedge \exists P_{i,j} \wedge \exists e_D^{i,j} \in E_D$
Ensure: `TRUE` iff it is possible to merge a_i with a_j, `FALSE` else
  return *keepControlflow?*$(a_i, a_j, P_{i,j}) \wedge$ *keepDataflow?*$(a_i, a_j, e_D^{i,j})$

Die hierfür notwendigen Voraussetzungen werden durch die Funktion *isMergePossible?* (siehe Algorithmus 3) algorithmisch auf einer vorliegenden PGM-Repräsentation überprüft: Diese Funktion setzt voraus, dass zwischen zwei gegebenen Aktivitäten a_i und a_j ein Kontrollflusspfad $P_{i,j}$ sowie eine Datenabhängigkeit $e_D^{i,j} \in E_D$ existiert, auf deren Grundlage o_i und o_j verschmolzen werden können. Die Funktion *isMergePossible?* liefert den Wahrheitswert `TRUE` genau dann, wenn in einer vorliegenden PGM-Repräsentation alle Voraussetzungen für eine korrekte Verschmelzung von o_i und o_j, die mit den Funktionen *keepControlflow?* und *keepDataflow?* überprüft werden, erfüllt sind.

Die Funktion *keepControlflow?* (siehe Algorithmus 4) bestimmt mithilfe des vorliegenden Kontrollflusspfades $P_{i,j}$, ob die ursprüngliche Ausführungssemantik von o_i durch eine Kombination mit o_j geändert werden kann. Dazu werden in einem ersten Schritt die a_i und a_j direkt umschließenden Kontrollflussmuster cfp_i und cfp_j mittels der Funktion *cfPattern* ermittelt. Anschließend überprüft die Funktion *isCompatible?* (siehe Algorithmus 5), ob cfp_i und cfp_j zueinander kompatibel sind. Diese Funktion setzt dabei die in Teilkapitel 5.4.1 diskutierte Kompatibilitätsmatrix direkt um (siehe Tabelle 5.1). Im positiven Fall wird anschließend durch die Funktion *getEnclosingCFPatterns* die Menge aller Kontrollflussmuster CFP_j bestimmt, die auf dem Pfad $P_{i,j}$ definiert sind und das Kontrollflussmuster cfp_j umschließen. Sind die ermittelten Kontrollflussmuster aus CFP_j zu cfp_i kompatibel, ist eine Verschmelzung von o_i und o_j möglich, ohne dass die ursprüngliche Ausführungssemantik von o_i geändert wird. In diesem Fall liefert die Funktion *keepControlflow?* den Wahrheitswert `TRUE`, sonst `FALSE`.

Algorithm 4 keepControlflow?

Require: $a_i, a_j \in A \wedge P_{i,j}$
Ensure: `TRUE` iff merge of a_i and a_j keeps the original controlflow semantics, `FALSE` else
  $cfp_i \leftarrow cfPattern(a_i)$
  $cfp_j \leftarrow cfPattern(a_j)$
  if *isCompatible?*(cfp_i, cfp_j)) **then**
    $CFP_j \leftarrow getEnclosingCFPatterns(cfp_j, P_{i,j})$
    for all $cfp_{j_x} \in CFP_j$ **do**
      if `NOT` *isCompatible?*(cfp_i, cfp_{j_x}) **then**
        return `FALSE`
      end if
    end for
    return `TRUE`
  else
    return `FALSE`
  end if

Algorithm 5 isCompatible?

Require: controlflow patterns cfp_i, cfp_j
Ensure: `TRUE` iff cfp_i and cfp_j are compatible, `FALSE` else
  if $Type(cfp_i) \in \{$`SEQ`, `PAR`$\} \wedge Type(cfp_j) \in \{$`SEQ`, `ALT`, `PAR`$\}$ **then**
    return `TRUE`
  else
    return `FALSE`
  end if

Die Funktion *keepDataFlow?* (siehe Algorithmus 6) überprüft, ob die Verschmelzung der Datenverarbeitungsoperationen o_i und o_j zu Änderungen der in einer vorliegenden PGM-Repräsentation ursprünglich modellierten Datenabhängigkeiten führen kann. Diese Funktion setzt eine Datenabhängigkeit $e_D^{i,j}$ zwischen a_i und a_j voraus, auf deren Grundlage a_i

und a_j miteinander verschmolzen werden können. Basierend auf $e_D^{i,j}$ werden für alle von a_i ausgehenden Datenabhängigkeitskanten $e_D^{i,x} \in E_D$ zu einer Aktivität a_x ($\neq a_j$) die zugehörigen Kontrollflussabhängigkeiten zwischen den Aktivitäten a_i, a_x und a_j überprüft. Liegt einer der beiden in Teilkapitel 5.4.1 diskutierten kritischen Fälle vor ($a_i \rightarrow a_x$ `AND` $a_x \rightarrow a_j$ (Fall 3) oder $a_i \rightarrow a_x$ `AND` $a_x \parallel a_j$ (Fall 4)), kann eine Verschmelzung ohne eine Änderung der ursprünglich modellierten Datenabhängigkeiten nicht duchgeführt werden. In diesem Fall liefert *keepDataFlow?* den Wahrheitswert `FALSE`, sonst `TRUE`.

Algorithm 6 keepDataflow?

Require: $a_i, a_j \in A \wedge e_D^{i,j} \in E_D$
Ensure: `TRUE` iff merge of a_i and a_j keeps the original dataflow semantics, `FALSE` else
  for all $e_D^{i,x} \in E_D$ **do**
    if $(a_i \rightarrow a_x \wedge a_x \rightarrow a_j) \vee (a_i \rightarrow a_x \wedge a_x \parallel a_j)$ **then**
      return `FALSE`
    end if
  end for
  return `TRUE`

D.2. Entfernen von Aktivitäten aus einer PGM-Repräsentation

D.2.1. Bei Anwendung einer Activity-Merging-Regel

Bei einer *Activity-Merging*-Regel (siehe Regeldefinitionen D.6 bis D.13) werden die Datenverarbeitungsoperationen zweier Aktivitäten a_i und a_j miteinander geeignet kombiniert, sodass Aktivität a_i aus der PGM-Repräsentation entfernt werden kann. Um die Konsistenz dieser PGM-Repräsentation zu erhalten, sind neben den in den Regeldefinitionen beschriebenen Aktionen zusätzlich folgende Transformationsschritte durchzuführen:

1. Sei $p_k \in P$ Partner von a_i. Dann entferne die Kommunikationsabhängigkeit zwischen a_i und p_k, d.h. $E_P \rightarrow E'_P$: $E'_P = E_P - \{e_P^{i,k}\}$.
2. Seien die Aktivitäten a_h und a_j direkter Vorgänger bzw. Nachfolger von a_i in der Ausführungslogik der PGM-Repräsentation. Dann entferne die Kontrollflussabhängigkeiten zwischen a_h, a_i und a_j, d.h. $E_C \rightarrow E'_C$: $E'_C = E_C - \{e_C^{h,i}, e_C^{i,j}\}$), und erzeuge eine neue Kontrollflussabhängigkeit zwischen a_h und a_j, d.h. $E'_C \rightarrow E''_C$: $E''_C = E'_C \cup \{e^{h,j}\}$.
3. Sei $E_{D_i} \subseteq E_D$ die Menge aller Datenabhängigkeiten, deren Start- oder Zielpunkt ein Dataslot von a_i ist. Dann entferne diese Datenabhängigkeiten aus der PGM-Repräsentation, d.h. $E_D \rightarrow E'_D$: $E'_D = E_D$ - E_{D_i}.

D.2.2. Bei Anwendung einer Tuple-To-Set-Regel

Bei einer *Tuple-To-Set*-Regel wird eine tupelbasierte DML-Anweisung, die in einem prozeduralen Datenverarbeitungsmuster definiert ist, in eine mengenbasierte DML-Anweisung überführt (siehe Regeldefinitionen D.14 bis D.16). Als Folge dieser Regelanwendung kann das prozedurale Datenverarbeitungsmuster bestehend aus den Aktivitäten *SQL-Query* (a_i), *XOR-Join* (a_j), *Set-Iterator* (a_k) und *SQL-DML* (a_l) direkt durch eine *SQL-DML*-Aktivität (a_{l+i}) ersetzt werden. Bei dieser Ersetzung sind folgende Transformationsschritte durchzuführen, damit die Konsistenz einer PGM-Repräsentation erhalten bleibt.

1. Sei $p_k \in P$ Partner von a_i. Dann entferne die Kommunikationsabhängigkeit zwischen a_i und p_k, d.h. $E_P \rightarrow E'_P$: $E'_P = E_P - \{e_P^{i,k}\}$.
2. Sei Aktivität a_h direkter Vorgänger von a_i und Aktivität a_m direkter Nachfolger von a_k. Dann entferne die Kontrollflussabhängigkeiten zwischen a_h, a_i, a_j, a_k, a_l und a_m, d.h. $E_C \rightarrow E'_C$: $E'_C = E_C - \{e_C^{h,i}, e_C^{i,j}, e_C^{j,k}, e_C^{k,l}, e_C^{k,m}, e_C^{l,j}\}$, und erzeuge neue Kontrollflussabhängigkeiten zwischen a_h, a_{l+i} und a_m, d.h. $E'_C \rightarrow E''_C$: $E''_C = E'_C \cup \{e^{h,l+i}, e_C^{l+i,m}\}$.
3. Sei $(E_{D_i} \cup E_{D_k}) \subseteq E_D$ die Menge aller Datenabhängigkeiten, deren Start- oder Zielpunkt ein Dataslot von a_i oder a_k ist. Dann entferne diese Datenabhängigkeiten aus der PGM-Repräsentation, d.h. $E_D \rightarrow E'_D$: $E'_D = E_D$ - $(E_{D_i} \cup E_{D_k})$.

D.3. Web-Service-Pushdown (1. Variante)

Regelbedingung

B1: $\exists v'_x \in V,\ x = 1...n\colon\ dType(v'_x) = \texttt{SIMPLE} \mid \texttt{SET}$

B2: $\exists v''_y \in V,\ y = 1...m\colon dType(v''_y) = \texttt{SIMPLE}$

B3: $\exists a_i \in A\colon a_i = (\texttt{INVOKE}, \{ps_i\}, c_i^{in}, \{c_i^{out}\}, \{d^r_{i_1}, ..., d^r_{i_n}\}, \{d^w_{i_1}, ..., d^w_{i_m}\}, opName_i)$
$\mu(d^r_{i_1}) = v'_1\ ...\ \mu(d^r_{i_n}) = v'_n \wedge \mu(d^w_{i_1}) = v''_1\ ...\ \mu(d^w_{i_m}) = v'_m$

B4: $\exists p_j \in P\colon pType(p_j) = \texttt{WS} \wedge \nu(ps_i) = p_j \wedge p_j$ unterstützt `SOAP/HTTP`

B6: $\exists! p_k \in P\colon pType(p_k) = \texttt{RDBMS} \wedge pProperty(p_k) = \texttt{WS-UDTF}$

B7: $\exists e_p^{i,j} \in E_P$

Regelaktion

A1: Erzeuge `WS-UDTF` $opName_i(v'_{i_1}, ..., v'_{i_n}) := v''_{i_1}, ..., v''_{i_m}$ für Partner p_k

A2: Erzeuge $S_{Q'_i} =$ `SELECT` $c_1\ \ldots\ c_m$ `INTO` $v''_{i_1}, ..., v''_{i_m}$
`FROM` $opName_i(v'_{i_1}, ..., v'_{i_n})$

A3: $a_i \rightarrow a'_i\colon\ a'_i = (\texttt{Query}, \{ps'_i\}, c_i^{in}, \{c_i^{out}\}, \{d^r_{i_1}, ..., d^r_{i_n}\}, \{d^w_{i_1}, ..., d^w_{i_m}\},$
$S_{Q'_i} = \texttt{SELECT-INTO}[\{v''_{i_1}, ..., v''_{i_m}\}, \{v'_{i_1}, ..., v'_{i_n}\}]): \nu(ps'_i) = p_k$

A4: $E_P \rightarrow E'_P\colon E'_P = (E_P - \{e_p^{i,j}\}) \cup \{e_p^{i,k}\}$

D.4. Web-Service-Pushdown (2. Variante)

Regelbedingung

B1: $\exists v'_x \in V,\ x = 1...n\text{: } dType(v'_x) =$ `SIMPLE` | `SET`

B2: $\exists v'' \in V\text{: } dType(v'') =$ `SET`

B3: $\exists a_i \in A\text{: } a_i = (\texttt{INVOKE}, \{ps_i\}, c_i^{in}, \{c_i^{out}\}, \{d_{i_1}^r, ..., d_{i_n}^r\}, \{d_i^w\}, opName_i)$
$\mu(d_{i_1}^r) = v'_1\ ...\ \mu(d_{i_n}^r) = v'_n \wedge \mu(d_i^w) = v''$

B4: $\exists p_j \in P\text{: } pType(p_j) =$ `WS` $\wedge\ \nu(ps_i) = p_j \wedge p_j$ unterstützt `SOAP/HTTP`

B5: $\exists! p_k \in P\text{: } pType(p_k) =$ `RDBMS` $\wedge\ pProperty(p_k) =$ `WS-UDTF`

B6: $\exists e_p^{i,j} \in E_P$

Regelaktion

A1: Erzeuge `WS-UDTF` $opName_i(v'_{i_1}, ..., v'_{i_n}) := v''$ für Partner p_k

A2: Erzeuge $S_{Q'_i} =$ `SELECT` $c_1\ \ldots\ c_m$
`FROM` $opName_i(v'_{i_1}, ..., v'_{i_n})$
$\rightarrow v''$

A3: $a_i \rightarrow a'_i\text{: } a'_i = (\texttt{Query}, \{ps'_i\}, c_i^{in}, \{c_i^{out}\}, \{d_{i_1}^r, ..., d_{i_n}^r\}, \{d_i^w\},$
$S_{Q'_i} =$ `SELECT`$[\{v''\}, \{v'_{i_1}, ..., v'_{i_n}\}]): \nu(ps'_i) = p_k$

A4: $E_P \rightarrow E'_P\text{: } E'_P = (E_P - \{e_p^{i,j}\}) \cup \{e_p^{i,k}\}$

D.5. Assign-Merging

Regelbedingung

B1: $\exists v', v'' \in V$: $dType(v'') = dType(v') = \texttt{SIMPLE}$

B2: $\exists a_i \in A$: $a_i = (\texttt{ASSIGN}, \emptyset, c_i^{in}, \{c_i^{out}\}, \{d_i^r\}, \{d_i^w\})$ mit $\mu(d_i^r) = v' \wedge \mu(d_i^w) = v''$

B3: $\exists a_j \in A$: $a_j = (\texttt{SQL}, \{ps_j\}, c_j^{in}, \{c_j^{out}\}, D_j^r, D_j^w, S_{SQL_j} = \texttt{SQL}[V_j^w, V_j^r])$

B4: $\exists P_{i,j} \in G_{PGM}$: $P_{i,j} = (a_i, ..., a_j)$

B5: $\exists e_{D_{wr}^x}^{i,j} \in E_D$: $\mu(d_i^w) = \mu(d_j^{r*}) = v''$, $d_j^{r*} \in D_j^r$

B6: $isMergePossible?(a_i, a_j, e_{D_{wr}^x}^{i,j}, P_{i,j}) = \texttt{True}$ (siehe Anhang D.1)

Regelaktion

A1: $S_{SQL_j} \rightarrow S_{SQL_{j+i}}$: Erzeuge $S_{SQL_{j+i}}$, indem jedes Vorkommen von v'' in S_{SQL_j} durch v' ersetzt wird.

A2: $D_j^r \rightarrow D_{j+i}^r$: $D_{j+i}^r = (D_j^r - \{d_j^{r*}\}) \cup \{d_{j+i}^r\}$ mit $\mu(d_{j+i}^r) = v'$

A3: $a_j \rightarrow a_{j+i} : a_{j+i} = (\texttt{SQL}, \{ps_j\}, c_j^{in}, \{c_j^{out}\}, D_{j+i}^r, D_j^w, S_{SQL_{j+i}} = \texttt{SQL}[V_j^w, V_{j+i}^r])$

A4: $A \rightarrow A' : A' = (A - \{a_i\})$ (siehe Anhang D.2.1)

D.6. Insert-Insert-Merging

Regelbedingung

B1: $\exists v' \in V$: $dType(v') = \texttt{SET}$

B2: $\exists a_i \in A$: $a_i = (\texttt{DML}, \{ps_i\}, c_i^{in}, \{c_i^{out}\}, D_i^r, \{d_i^w\}, S_{DML_i} = \texttt{INSERT-SELECT}[v', V_i^r])$

B3: $\exists a_j \in A, a_j \neq a_i$: $a_j = (\texttt{DML}, \{ps_j\}, c_j^{in}, \{c_j^{out}\}, D_j^r, \{d_j^w\}, S_{DML_j} = \texttt{INSERT-SELECT}[v', V_j^r])$

B4: $\exists p_k \in P$: $pType(p_k) = \texttt{RDBMS}$

B5: $\exists P_{i,j} \in G_{PGM}$: $P_{i,j} = (a_i, ..., a_j)$

B6: $\exists e_{D_{ww}}^{i,j} \in E_D$: $\mu(d_i^w) = \mu(d_j^w) = v'$

B7: $\exists e_p^{i,k}, e_p^{j,k} \in E_P$

B8: $isMergePossible?(a_i, a_j, e_{D_{ww}}^{i,j}, P_{i,j}) = \texttt{True}$ (siehe Anhang D.1)

Regelaktion

A1: $S_{DML_j} \rightarrow S_{DML_{j+i}}$: $S_{DMLj+i} = \texttt{INSERT INTO}\ v'$
$\texttt{SELECT}[V_j^r]\ \texttt{UNION ALL SELECT}[V_i^r]$

A2: $D_j^r \rightarrow D_{j+i}^r$: $D_{j+i}^r = D_j^r \cup D_i^r$

A3: $a_j \rightarrow a_{j+i}$: $a_{j+i} = (DML, \{ps_j\}, c_j^{in}, \{c_j^{out}\}, D_{j+i}^r, \{d_j^w\},$
$S_{DML_{j+i}} = \texttt{INSERT-SELECT}[\{v'\}, V_j^r \cup V_i^r])$

A4: $E_D \rightarrow E'_D$: Erzeuge E'_D durch Anpassung der Datenabhängigkeiten in E_D basierend auf D_{j+i}^r.

A5: $A \rightarrow A' : A' = (A - \{a_i\})$ (siehe Anhang D.2.1)

Anmerkung zu **A5**:
Existiert eine Schreib-Lese-Datenabhängigkeit basierend auf v' zwischen a_i und einer Aktivität $a_x \in A$, die alternativ zu a_j ausgeführt wird $(a_x \otimes a_j)$, darf a_i nicht aus A entfernt werden, sondern muss stattdessen in den alternativen Kontrollflusspfad vor a_x verschoben werden, um die existierende Datenabhängigkeit zwischen a_i und a_x zu erhalten.

D.7. Delete-Delete-Merging

Regelbedingung

B1: $\exists v' \in V$: $dType(v') = \texttt{SET}$

B2: $\exists a_i \in A$: $a_i = (\texttt{DML}, \{ps_i\}, c_i^{in}, \{c_i^{out}\}, D_i^r, \{d_i^w\}, S_{DML_i} = \texttt{DELETE}[v', V_i^r])$

B3: $\exists a_j \in A, a_j \neq a_i$: $a_i = (\texttt{DML}, \{ps_j\}, c_j^{in}, \{c_j^{out}\}, D_j^r, \{d_j^w\}, S_{DML_j} = \texttt{DELETE}[v', V_j^r])$

B4: $\exists p_k \in P$: $pType(p_k) = \texttt{RDBMS}$

B5: $\exists P_{i,j} \in G_{PGM}$: $P_{i,j} = (a_i, ..., a_j)$

B6: $\exists e_{D_{ww}}^{i,j} \in E_D$: $\mu(d_i^w) = \mu(d_j^w) = v'$

B7: $\exists e_p^{i,k}, e_p^{j,k} \in E_P$

B8: *isMergePossible?*(a_i, a_j, $e_{D_{ww}}^{i,j}$, $P_{i,j}$) = `True` (siehe Anhang D.1)

Regelaktion

A1: $S_{DML_j} \rightarrow S_{DML_{j+i}}$: $S_{DMLj+i} =$ `DELETE FROM` v' `WHERE` $where_j$ `OR` $where_i$

A2: $D_j^r \rightarrow D_{j+i}^r$: $D_{j+i}^r = D_j^r \cup D_i^r$

A3: $a_j \rightarrow a_{j+i} : a_{j+i} = (DML, \{ps_j\}, c_j^{in}, \{c_j^{out}\}, D_{j+i}^r, \{d_j^w\}, S_{DML_{j+i}} = \texttt{DELETE}[\{v'\}, V_j^r \cup V_i^r])$

A4: $E_D \rightarrow E'_D$: Erzeuge E'_D durch Anpassung der Datenabhängigkeiten in E_D basierend auf D_{j+i}^r.

A5: $A \rightarrow A' : A' = (A - \{a_i\})$ (siehe Anhang D.2.1)

Anmerkung zu **A5**:
Existiert eine Schreib-Lese-Datenabhängigkeit basierend auf v' zwischen a_i und einer Aktivität $a_x \in A$, die alternativ zu a_j ausgeführt wird ($a_x \otimes a_j$), darf a_i nicht aus A entfernt werden, sondern muss stattdessen in den alternativen Kontrollflusspfad vor a_x verschoben werden, um die existierende Datenabhängigkeit zwischen a_i und a_x zu erhalten.

D.8. Update-Update-Merging

Regelbedingung

B1: $\exists v' \in V$: $dType(v') = \texttt{SET}$

B2: $\exists a_i \in A$: $a_i = (\texttt{DML}, \{ps_i\}, c_i^{in}, \{c_i^{out}\}, D_i^r, \{d_i^w\}, S_{DML_i} = \texttt{UPDATE}[v', V_i^r])$)

B3: In der Set- (set_i) und Where-Klausel ($where_i$) von S_{DML_i} dürfen nicht dieselben Attribute referenziert werden.

B4: $\exists a_j \in A, a_j \neq a_i$: $a_j = (\texttt{DML}, \{ps_j\}, c_j^{in}, \{c_j^{out}\}, D_j^r, \{d_j^w\}, S_{DML_j} = \texttt{UPDATE}[v', V_j^r]))$

B5: S_{DML_i} und S_{DML_j} unterscheiden sich lediglich in ihren Set-Klauseln set_i und set_j.

B6: In set_i und set_j werden nicht dieselben Attribute referenziert.

B7: $\exists p_k \in P$: $pType(p_k) = \texttt{RDBMS}$

B8: $\exists P_{i,j} \in G_{PGM}$: $P_{i,j} = (a_i, ..., a_j)$

B9: $\exists e_{D_{ww}}^{i,j} \in E_D$: $\mu(d_i^w) = \mu(d_j^w) = v'$

B10: $\exists e_p^{i,k}, e_p^{j,k} \in E_P$

B11: $isMergePossible?(a_i, a_j, e_{D_{ww}}^{i,j}, P_{i,j}) = \texttt{True}$ (siehe Anhang D.1)

Regelaktion

A1: $S_{DML_j} \rightarrow S_{DML_{j+i}}$: $S_{DMLj+i} =$ $\texttt{UPDATE}$ v'
$\texttt{SET}$ set_i, set_j
$\texttt{WHERE}$ $where_i$

A2: $D_j^r \rightarrow D_{j+i}^r$: $D_{j+i}^r = D_j^r \cup D_i^r$

A3: $a_j \rightarrow a_{j+i} : a_{j+i} = (DML, \{ps_j\}, c_j^{in}, \{c_j^{out}\}, D_{j+i}^r, \{d_j^w\},$
$S_{DML_{j+i}} = \texttt{UPDATE}[\{v'\}, V_j^r \cup V_i^r])$

A4: $E_D \rightarrow E'_D$: Erzeuge E'_D durch Anpassung der Datenabhängigkeiten in E_D basierend auf D_{j+i}^r.

A5: $A \rightarrow A' : A' = (A - \{a_i\})$ (siehe Anhang D.2.1)

Anmerkung zu **A5**:
Existiert eine Schreib-Lese-Datenabhängigkeit basierend auf v' zwischen a_i und einer Aktivität $a_x \in A$, die alternativ zu a_j ausgeführt wird ($a_x \otimes a_j$), darf a_i nicht aus A entfernt werden, sondern muss stattdessen in den alternativen Kontrollflusspfad vor a_x verschoben werden, um die existierende Datenabhängigkeit zwischen a_i und a_x zu erhalten.

D.9. Predicate-Pushdown

Regelbedingung

B1: $\exists v', v''' \in V$: $dType(v') = dType(v''') = \texttt{SET}$

B2: $\exists a_i \in A$: $a_i = (\texttt{Query}, \{ps_i\}, c_i^{in}, \{c_i^{out}\}, D_i^r, \{d_i^w\}, S_{Q_i} = \texttt{SELECT}[\{v'\}, V_i^r])$ mit
$S_{Q_i} = [\texttt{WITH}\ \ with_i]$
$\texttt{SELECT}\ select_i$
$\texttt{FROM}\ \ from_i$
$\texttt{WHERE}\ \ where_i$
$\rightarrow v'$

B3: $\exists a_j \in A$: $a_j = (\texttt{ASSIGN-SELECT}, \emptyset, c_j^{in}, \{c_j^{out}\}, \{d_j^r\} \cup D_j^{r'}, \{d_j^w\}, \sigma_j[V_j^{r'}](v') \rightarrow v''')$

B3: $\exists P_{i,j} \in G_{PGM}$: $P_{i,j} = (a_i, ..., a_j)$

B4: $\exists e_{D_{wr}^x}^{i,j} \in E_D$: $\mu(d_i^w) = \mu(d_j^r) = v'$

B5: $isMergePossible?(a_j, a_i, e_{D_{wr}^x}^{i,j}, P_{i,j}) = \texttt{True}$ (siehe Anhang D.1)

Regelaktion

A1: $S_{Q_i} \rightarrow S_{Q_{i+j}}$: $S_{Q_i} = [\texttt{WITH}\ \ with_i]$ mit
$\texttt{SELECT}\ select_i$
$\texttt{FROM}\ \ from_i$
$\texttt{WHERE}\ \ where_i\ \texttt{AND}\ \sigma_j[V_j^{r'}]$
$\rightarrow v'''$

A2: $D_i^r \rightarrow D_{i+j}^r$: $D_{i+j}^r = D_i^r \cup D_j^{r'}$

A3: $D_i^w \rightarrow D_{i+j}^w$: $D_{i+j}^w = \{d_j^w\}$

A4: $a_i \rightarrow a_{i+j} : a_{i+j} = (\texttt{Query}, \{ps_i\}, c_i^{in}, \{c_i^{out}\}, D_{i+j}^r, D_{i+j}^w$
$S_{Q_{i+j}} = \texttt{SELECT}[\{v'''\}, V_i^r \cup V_j^{r'}])$

A5: $E_D \rightarrow E'_D$: Erzeuge E'_D durch Anpassung der Datenabhängigkeiten in E_D basierend auf D_{j+i}^r und D_{j+i}^w.

A6: $A \rightarrow A' : A' = (A - \{a_j\})$ (siehe Anhang D.2.1)

Regelvariante

V1: Enthält S_{Q_i} (siehe **B2**) eine GroupBy-Klausel bzw. eine Select-Klausel mit einer Aggregationsfunktion, wird $\sigma_j[V_j^{r'}]$ in die Having-Klausel von S_{Q_i} verschoben (siehe **A1**).

D.10. Predicate-Pushdown (Variante)

Regelbedingung

B1: $\exists v' \in V$: $dType(v') = \texttt{SET}$

B2: $\exists v''_x \in V, x = 1, ..., n$: $dType(v''_x) = \texttt{SIMPLE}$

B3: $\exists a_i \in A$: $a_i = (\texttt{Query}, \{ps_i\}, c_i^{in}, \{c_i^{out}\}, D_i^r, \{d_i^w\}, S_{Q_i} = \texttt{SELECT}[\{v'\}, V_i^r])$ mit
$S_{Q_i} = [\texttt{WITH}\ \ with_i]$
$\texttt{SELECT}\ select_i$
$\texttt{FROM}\ \ from_i$
$\texttt{WHERE}\ \ where_i$
$\rightarrow v'$

B4: $\exists a_j \in A$: $a_j = (\texttt{XOR-Join}, \emptyset, c_j^{in}, \{c_j^{out}\}, \emptyset, \emptyset)$

B5: $\exists a_k \in A$: $a_k = (\texttt{Set-Iterator}, \emptyset, c_k^{in}, \{c_k^{out,1}, c_k^{out,2}\}, \{d_k^r\}, \{d_{k_1}^w, ..., d_{k_n}^w\})$

B6: $\exists a_l \in A$: $a_l = (\texttt{XOR-Split-Select}, \emptyset, c_l^{in}, \{c_l^{out,1}, c_l^{out,2}\}, \{d_{l_1}^r, ..., d_{l_n}^r\}, \emptyset, \sigma_l[V_l^r] \rightarrow \textbf{Boolean})$

B7: $\exists P_{i,i} = (a_i, a_j, a_k, a_l, ..., a_i)$

B8: $\exists e_{D_{wr}^x}^{i,k} E_D$: $\mu(d_i^w) = \mu(d_k^r) = v'$

B9: $\exists e_{D_{wr}}^{k_x,l_x} E_D$: $\mu(d_{k_x}^w) = \mu(d_{l_x}^r) = v''_x$, $x = 1...n$

Regelaktion

A1: $S_{Q_i} \rightarrow S_{Q_{i+l}}$: $S_{Q_i} = [\texttt{WITH}\ \ with_i]$
$\texttt{SELECT}\ select_i$
$\texttt{FROM}\ \ from_i$
$\texttt{WHERE}\ \ where_i\ \texttt{AND}\ \sigma_l[V_l^r]$
$\rightarrow v'$

A2: $D_i^r \rightarrow D_{i+l}^r$: $D_{i+l}^r = D_i^r \cup D_l^r$

A3: $a_i \rightarrow a_{i+l}$: $a_{i+l} = (\texttt{Query}, \{ps_i\}, c_i^{in}, \{c_i^{out}\}, D_{i+l}^r, \{d_i^w\}, S_{Q_{i+l}} = \texttt{SELECT}[\{v'\}, V_i^r \cup V_l^r])$

A4: $E_D \rightarrow E'_D$: Erzeuge E'_D durch Anpassung der Datenabhängigkeiten in E_D basierend auf D_{i+l}^r.

A5: $A \rightarrow A' : A' = (A - \{a_l\})$ (siehe auch Anhang D.2.1)

Regelvariante

V1: Enthält S_{Q_i} (siehe **B3**) eine GroupBy-Klausel bzw. eine Select-Klausel mit einer Aggregationsfunktion, wird $\sigma_l[V_l^r]$ in die Having-Klausel von S_{Q_i} verschoben (siehe **A2**).

D.11. Select-Into

Regelbedingung

B1: $\exists v'_1, ..., v'_n \in V$: $dType(v'_1) = ... = dType(v'_n) = \texttt{SIMPLE}$

B2: $\exists v'' \in V$: $dType(v'') = \texttt{SET}$

B3: $\exists a_i \in A$: $a_i = (\texttt{Query}, \{ps_i\}, c_i^{in}, \{c_i^{out}\}, D_i^r, \{d_{i_1}^w, ..., d_{i_n}^w\},$
$S_{Q_i} = \texttt{SELECT-INTO}[\{v'_1 ... v'_n\}, V_i^r])$

B4: $\exists a_j \in A$: $a_j = (\texttt{Query}, \{ps_j\}, c_j^{in}, \{c_j^{out}\}, \{d_{j_1}^r, ..., d_{j_n}^r\} \subseteq D_j^r, \{d_j^w\},$
$S_{Q_j} = \texttt{SELECT}[\{v''\}, \{v'_1 ... v'_n\} \subseteq V_j^r])$

B5: $\exists p_k \in P$: $pType(p_k) = \texttt{RDBMS}$

B6: $\exists P_{i,j} \in G_{PGM}$: $P_{i,j} = (a_i, ..., a_j)$

B7: $\exists e_{D_{wr}^x}^{i_1,j_1}, ..., e_{D_{wr}^x}^{i_n,j_n} \in E_D$: $\mu(d_{i_1}^w) = \mu(d_{j_1}^r) = v'_1$ ... $\mu(d_{i_n}^w) = \mu(d_{j_n}^r) = v'_n$

B8: $\exists e_P^{i,k}, e_P^{j,k} \in E_P$

B9: $isMergePossible?(a_i, a_j, e_{D_{wr}^x}^{i_1,j_1}, ..., e_{D_{wr}^x}^{i_n,j_n}, P_{i,j}) = \texttt{True}$ (siehe Anhang D.1)

Regelaktion

A1: $S_{Q_j} \rightarrow S'_{Q_{j+i}}$: $S'_{Q_{j+i}} =$ `WITH TEMP`$(c_1, ..., c_n)$ `AS` ([`WITH` $with_i$]
`SELECT` $select_i$
`FROM` $from_i$
`WHERE` $where_i$
[`GROUP BY` $groupby_i$]
[`HAVING` $having_i$]), [$with_j$]
`SELECT` $select_j$
`FROM` $from_j$
`WHERE` $where_j$
[`GROUP BY` $groupby_j$]
[`HAVING` $having_j$]
$\rightarrow v''$

A2: $S'_{Q_{j+i}} \rightarrow S_{Q_{j+i}}$: Erzeuge $S_{Q_{j+i}}$, indem alle Vorkommen von v'_1 ... v'_n in $S'_{Q_{j+i}}$ jeweils durch die Ausdrücke `TEMP`.c_1 ... `TEMP`.c_n ersetzt werden.

A3: $D_j^r \rightarrow D_{j+i}^r$: $D_{j+i}^r = D_j^r \cup D_i^r$

A4: $a_j \rightarrow a_{j+i}$: $a_{j+i} = (\texttt{Query}, \{ps_j\}, c_j^{in}, \{c_j^{out}\}, D_{j+i}^r, \{d_j^w\}, S_{Q_{j+i}} = \texttt{SELECT}[\{v''\}, V_j^r \cup V_i^r])$

A5: $E_D \rightarrow E'_D$: Erzeuge E'_D durch Anpassung der Datenabhängigkeiten in E_D basierend auf D_{j+i}^r.

A6: $A \rightarrow A' : A' = A - \{a_i\}$ (siehe Anhang D.2.1)

Regelvarianten

V1: Bei der abhängigen SQL-Anfrage S_{Q_j} (siehe **B4**) kann es sich auch um eine Unteranfrage handeln, die in der Where-Klausel einer Query oder einer DML-Anweisung ausgeführt wird. Im Falle einer DML-Anweisung muss zudem sichergestellt sein, dass die DML-Anweisung und die in die DML-Anweisung zu verschiebende SQL-Anfrage S_{Q_i} nicht auf derselben Tabelle ausgeführt werden, da sonst gemäß des SQL:2003-Standards eine nicht ausführbare DML-Anweisung generiert wird.

V2: Bei der abhängigen SQL-Aktivität a_j (siehe **B4**) kann es sich auch um eine *SQL-DML*-Aktivität handeln, die eine Insert-Select-Anweisung ausführt.

D.12. Select-Merging

Regelbedingung

B1: $\exists v', v'' \in V$: $dType(v') = dType(v'') = \texttt{SET}$

B2: $\exists a_i \in A$: $a_i = (\texttt{Query}, \{ps_i\}, c_i^{in}, \{c_i^{out}\}, D_i^r, \{d_i^w\}, S_{Q_i} = \texttt{SELECT}[\{v'\}, V_i^r])$

B3: $\exists a_j \in A$: $a_j = (\texttt{Query}, \{ps_j\}, c_j^{in}, \{c_j^{out}\}, D_j^r, \{d_j^w\}, S_{Q_j} = \texttt{SELECT}[\{v''\}, \{v'\} \subseteq V_j^r])$

B4: S_{Q_j} ist durch Anwendung der *Web-Service-Pushdown*-Regel (2. Variante) entstanden, d.h. $S_{Q_j} =$ `SELECT` c_1 ... c_m `FROM WSUDTF`$(v'$, ... $)$ $\rightarrow v''$

B5: $\exists p_k \in P$: $pType(p_k) = \texttt{RDBMS}$

B6: $\exists P_{i,j} \in G_{PGM}$: $P_{i,j} = (a_i, ..., a_j)$

B7: $\exists e_{D_{wr}^x}^{i,j} \in E_D$: $\mu(d_i^w) = \mu(d_j^{r*}) = v'$, $d_j^{r*} \in D_j^r$

B8: $\exists e_P^{i,k} \in E_P$

B9: *isMergePossible?*$(a_i, a_j, e_{D_{wr}^x}^{i,j}, P_{i,j}) =$ `True` (siehe Anhang D.1)

Regelaktion

A1: $S_{Q_j} \rightarrow S_{Q_{j+i}}$: $S_{Q_{j+i}} =$

```
WITH TEMP(c1) AS (
    SELECT XMLELEMENT(NAME „RowSet“,
             XMLELEMENT(NAME „Row“,
              XMLCONCAT(XMLELEMENT(NAME „c1“, c1),
                                   ⋮
                        XMLELEMENT(NAME, „cn“, cn))))
    FROM (WITH     with_i
          SELECT   select_i
          FROM     from_i
          WHERE    where_i
          GROUP BY groupby_i
          HAVING   having_i))
SELECT c1 ... cm
FROM WSUDTF(TEMP.c1...)
 → v''
```

A2: $D_j^r \rightarrow D_{j+i}^r$: $D_{j+i}^r = (D_j^r - \{d_j^{r*}\}) \cup D_i^r$

A3: $a_j \rightarrow a_{j+i}$: $a_{j+i} = (\texttt{Query}, \{ps_j\}, c_j^{in}, \{c_j^{out}\}, D_{j+i}^r, \{d_j^w\}, S_{Q_{j+i}} = \texttt{SELECT}[\{v''\}, (V_j^r \text{ - } \{v'\}) \cup V_i^r])$

A4: $E_D \rightarrow E'_D$: Erzeuge E'_D durch Anpassung der Datenabhängigkeiten in E_D basierend auf D_{j+i}^r.

A5: $A \rightarrow A' : A' = A - \{a_i\}$ (siehe Anhang D.2.1)

Anmerkung zu **A1**:
Die hier gezeigte XML-Transformation gilt nur für XML-RowSet-Datenstrukturen, wie sie in BPEL/SQL für materialisierte Tabellen definiert sind.

D.13. Eliminate-Temporary-Table

Regelbedingung

B1: $\exists v \in V$: $dType(v) = \texttt{SET} \wedge TempTable(v) = \texttt{TRUE}$

B2: $\exists a_i \in A$: $a_i = (\texttt{DML}, \{ps_i\}, c_i^{in}, \{c_i^{out}\}, D_i^r, \{d_i^w\}, S_{DML_i} = \texttt{INSERT-SELECT}[\{v\}, V_i^r])$

Hinweis: Die Select-Anfrage von S_{DML_i} wird im Folgenden als S_{Q_i} bezeichnet.

B3: $\exists a_j \in A$: $a_j = (\texttt{Query}, \{ps_j\}, c_j^{in}, \{c_j^{out}\}, D_j^r, \{d_j^w\}, S_{Q_j} = \texttt{SELECT}[\{v_j^w\}, \{v\} \subseteq V_j^r])$

B4: SET-Variable v wird in S_{Q_j} nur in der From-Klausel referenziert.

B5: $\exists p_k \in P$: $pType(p_k) = \texttt{RDBMS}$

B6: $\exists P_{i,j} \in G_{PGM}$: $P_{i,j} = (a_i, ..., a_j)$

B7: $\exists e_{D_{wr}^x}^{i,j} \in E_D$: $\mu(d_i^w) = \mu(d_j^{r*}) = v'$, $d_j^{r*} \in D_j^r \wedge Counter(d_j^{r*}) = 1$

B8: $\exists e_P^{i,k}, e_P^{j,k} \in E_P$

B9: $isMergePossible?(a_i, a_j, e_{D_{wr}^x}^{i,j}, P_{i,j}) = \texttt{True}$ (siehe Anhang D.1)

Regelaktion

A1: $S_{Q_j} \rightarrow S_{Q_{j+i}}$: Erzeuge $S_{Q_{j+i}}$, indem die SET-Variable v in S_{Q_j} durch S_{Q_i} ersetzt wird.

A2: $D_j^r \rightarrow D_{j+i}^r$: $D_{j+i}^r = (D_j^r - \{d_j^{r*}\}) \cup D_i^r$

A3: $a_j \rightarrow a_{j+i} : a_{j+i} = (Query, \{ps_j\}, c_j^{in}, \{c_j^{out}\}, D_{j+i}^r, \{d_j^w\}, S_{Q_{j+i}} = \texttt{SELECT}[\{v_j^w\}, (V_j^r \text{ - } \{v\}) \cup V_i^r])$

A4: $E_D \rightarrow E'_D$: Erzeuge E'_D durch Anpassung der Datenabhängigkeiten in E_D basierend auf D_{j+i}^r.

A5: $A \rightarrow A' : A' = (A - \{a_i\})$ (siehe Anhang D.2.1)

Regelvarianten

V1: Bei der abhängigen SQL-Anfrage S_{Q_j} (siehe **B3**) kann es sich auch um eine Unteranfrage handeln, die in der Where-Klausel einer Query oder einer

DML-Anweisung ausgeführt wird. Im Falle einer DML-Anweisung muss zudem sichergestellt sein, dass die DML-Anweisung und die in die DML-Anweisung zu verschiebende SQL-Anfrage S_{Q_i} nicht auf derselben Tabelle ausgeführt werden, da sonst gemäß des SQL:2003-Standards eine nicht ausführbare DML-Anweisung generiert wird.

V2: Bei der abhängigen *SQL*-Aktivität a_j (siehe **B3**) kann es sich auch um eine *SQL-DML*-Aktivität handeln, die eine Insert-Select-Anweisung ausführt.

V3: Wird die temporäre Tabelle, d.h. die `SET`-Variable v, mehrfach in der abhängigen SQL-Anfrage S_{Q_j} referenziert ($Counter(d_j^{r*}) > 1$), muss S_{Q_i} in die With-Klausel von S_{Q_j} verschoben werden, um eine redundante Ausführung von S_{Q_i} zu verhindern.

D.14. Insert-Tuple-To-Set

Regelbedingung

B1: $\exists v' \in V$: $dType(v') = \texttt{SET}$

B2: $\exists v''_1, ..., v''_n \in V : dType(v''_1) = ... = dType(v''_n) = \texttt{SIMPLE}$

B3: $\exists a_i \in A$: $a_i = (\texttt{Query}, \{ps_i\}, c_i^{in}, \{c_i^{out}\}, D_i^r, \{d_i^w\}, S_{Q_i} = \texttt{SELECT}[\{v'\}, V_i^r])$

B4: $\exists a_j \in A$: $a_j = (\texttt{XOR-Join}, \emptyset, c_j^{in}, \{c_j^{out}\}, \emptyset, \emptyset)$

B5: $\exists a_k \in A$: $a_k = (\texttt{Set-Iterator}, \emptyset, c_k^{in}, \{c_k^{out,1}, c_k^{out,2}\}, \{d_k^r\}, \{d_{k_1}^w, ..., d_{k_n}^w\})$

B6: $\exists a_l \in A$: $a_l = (\texttt{DML}, \{ps_l\}, c_l^{in}, \{c_l^{out}\}, \{d_{l_1}^r, ..., d_{l_n}^r\} \subseteq D_l^r, \{d_l^w\},$
$S_{DML_l} = \texttt{INSERT-SELECT}[\{v_l^w\}, \{v''_1, ..., v''_n\} \subseteq V_l^r])$

Hinweis: Die Select-Anfrage von S_{DML_l} wird im Folgenden als S_{Q_l} referenziert.

B7: $\exists p_m \in P$: $pType(p_m) = \texttt{RDBMS}$

B8: $\exists P_{i,j} = \{a_i, a_j, a_k, a_l, a_j\}$

B9: $\exists e_{D_{wr}^x}^{i,k} \in E_D$: $\mu(d_i^w) = \mu(d_k^r) = v'$

B10: $\forall v''_x \in V, x = 1...n$: $\exists e_{D_{wr}^x}^{k,l} \in E_D$: $\mu(d_{k_x}^w) = \mu(d_{l_x}^r) = v''_x$

B11: $\exists e_P^{i,m}, e_P^{l,m} \in E_P$

Regelaktion

A1: $S_{DML_l} \rightarrow S'_{DML_{l+i}}$: $S'_{DML_{l+i}} =$ `INSERT INTO` v_l^w
`SELECT` $T_2.c_1$, ..., $T_2.c_m$
`FROM` (S_{Q_i}) `AS` $T_1(c_1, ..., c_n)$, `LATERAL` (S_{Q_l}) `AS` $T_2(c_1, ..., c_m)$

A2: $S'_{DML_{l+i}} \rightarrow S_{DML_{l+i}}$: Erzeuge $S_{DML_{l+i}}$, indem jedes Vorkommen von $v''_1, ..., v''_n$ in $S'_{DML_{l+i}}$ durch „$T_1.c_x$“, ..., „$T_n.c_n$“ ersetzt wird.

A3: $D_l^r \rightarrow D_{l+i}^r$: $D_{l+i}^r = (D_l^r - \{d_{l_1}^r, ..., d_{l_n}^r\}) \cup D_i^r$

A4: $a_l \rightarrow a_{l+i}$: $a_{l+i} =$(DML, $\{ps_l\}, c_l^{in}, \{c_l^{out}\}, D_{l+i}^r, \{d_l^w\},$
$S_{DML_{l+i}} = \texttt{INSERT-SELECT}[\{v_l^w\}, (V_l^r$ - $\{v''_1, ..., v''_n\}) \cup V_i^r])$

A5: $A \rightarrow A'$: $A' = (A - \{a_i, a_j, a_k\})$ (siehe Anhang D.2.2)

D.15. Update-Tuple-To-Set

Regelbedingung

B1: $\exists v' \in V$: $dType(v') = \texttt{SET}$

B2: $\exists v'' \in V$: $dType(v'') = \texttt{SIMPLE}$

B3: $\exists a_i \in A$: $a_i = (\texttt{Query}, \{ps_i\}, c_i^{in}, \{c_i^{out}\}, D_i^r, \{d_i^w\}, S_{Q_i} = \texttt{SELECT}[\{v'\}, V_i^r])$

B4: $\exists a_j \in A$: $a_j = (\texttt{XOR-Join}, \emptyset, c_j^{in}, \{c_j^{out}\}, \emptyset, \emptyset)$

B5: $\exists a_k \in A$: $a_k = (\texttt{Set-Iterator}, \emptyset, c_k^{in}, \{c_k^{out,1}, c_k^{out,2}\}, \{d_k^r\}, D_k^w)$

B6: $\exists a_l \in A$: $a_l = (\texttt{DML}, \{ps_l\}, c_l^{in}, \{c_l^{out}\}, \{d_l^{r*}\} \cup D_l^{r'}, \{d_l^w\}, S_{DML_l})$ mit
$S_{DML_l} =$ UPDATE v_l^w
SET set_l
WHERE $c_k = v''$ [AND $\sigma_l[V_l^{r'}]$] (c_k ist Primärschlüssel)

B7: $\exists p_m \in P$: $pType(p_m) = \texttt{RDBMS}$

B8: $\exists P_{i,j} = \{a_i, a_j, a_k, a_l, a_j\}$

B9: $\exists e^{i,k}_{D^x_{wr}} \in E_D$: $\mu(d_i^w) = \mu(d_k^r) = v'$

B10: $\exists e^{k,l}_{D^x_{wr}} \in E_D$: $\mu(d_k^w) = \mu(d_l^{r*}) = v''$

B11: $\exists e_P^{i,m}, e_P^{l,m} \in E_P$

Regelaktion

A1: $S_{DML_l} \rightarrow S_{DML_{l+i}}$: $S_{DML_{l+i}} =$ MERGE INTO v_l^w AS T
USING (S_{Q_i}) AS TEMP$(c_1, ..., c_m)$
ON T.c_1 = TEMP.c_1
WHEN MATCHED [AND $\sigma_l[V_l^{r'}]$] THEN
UPDATE SET set_l

A2: $D_l^r \rightarrow D_{l+i}^r$: $D_{l+i}^r = D_l^{r'} \cup D_i^r$

A3: $a_l \rightarrow a_{l+i}$: $a_{l+i} = (\texttt{DML}, \{ps_l\}, c_l^{in}, \{c_l^{out}\}, D_{l+i}^r, \{d_l^w\}, S_{DML_{l+i}} = \texttt{MERGE}[\{v_l^w\}, V_l^{r'} \cup V_i^r])$

A4: $A \rightarrow A'$: $A' = (A - \{a_i, a_j, a_k\})$ (siehe Anhang D.2.2)

D.16. Delete-Tuple-To-Set

Regelbedingung

B1: $\exists v' \in V$: $dType(v') = \texttt{SET}$

B2: $\exists v''_1, ..., v''_n \in V$: $dType(v''_1) = ... = dType(v''_n) = \texttt{SIMPLE}$

B3: $\exists a_i \in A$: $a_i = (\texttt{Query}, \{ps_i\}, c_i^{in}, \{c_i^{out}\}, D_i^r, \{d_i^w\}, S_{Q_i} = \texttt{SELECT}[\{v'\}, V_i^r])$

B4: $\exists a_j \in A$: $a_j = (\texttt{XOR-Join}, \emptyset, c_j^{in}, \{c_j^{out}\}, \emptyset, \emptyset)$

B5: $\exists a_k \in A$: $a_k = (\texttt{Set-Iterator}, \emptyset, c_k^{in}, \{c_k^{out,1}, c_k^{out,2}\}, \{d_k^r\}, \{d_{k_1}^w, ..., d_{k_n}^w\})$

B6: $\exists a_l \in A$: $a_l = (\texttt{DML}, \{ps_l\}, c_l^{in}, \{c_l^{out}\}, \{d_{l_1}^r, ..., d_{l_n}^r\} \subseteq D_l^r, \{d_l^w\}, S_{DML_l})$
$S_{DML_l} = \texttt{DELETE FROM } v_l^w$
$\texttt{WHERE } where'_l \texttt{ AND } c_1 = v''_1 \texttt{ AND } \texttt{...} \texttt{ AND } c_n = v''_n$

B7: $\exists p_m \in P$: $pType(p_m) = \texttt{RDBMS}$

B8: $\exists P_{i,j} = \{a_i, a_j, a_k, a_l, a_j\}$

B9: $\exists e^{i,k}_{D^x_{wr}} \in E_D$: $\mu(d_i^w) = \mu(d_k^r) = v'$

B10: $\forall v''_x \in V, x = 1...n$: $\exists e^{k,l}_{D^x_{wr}} \in E_D$: $\mu(d_{k_x}^w) = \mu(d_{l_x}^r) = v''_x$

B11: $\exists e_P^{i,m}, e_P^{l,m} \in E_P$

Regelaktion

A1: $S_{DML_l} \rightarrow S_{DML_{l+i}}$: $S_{DML_{l+i}} = \texttt{DELETE FROM } v_l^w$
$\texttt{WHERE } where'_l \texttt{ AND } (c_1, ..., c_n) \texttt{ IN } (S_{Q_i})$

A2: $D_l^r \rightarrow D_{l+i}^r$: $D_{l+i}^r = (D_l^r - \{d_{l_1}^r, ..., d_{l_n}^r\}) \cup D_i^r$

A3: $a_l \rightarrow a_{l+i}$: a_{l+i} =(DML, $\{ps_l\}, c_l^{in}, \{c_l^{out}\}, D_{l+i}^r, \{d_l^w\}$,
$S_{DML_{l+i}} = \texttt{DELETE}[\{v_l^w\}, (V_l^r \text{ - } \{v''_1, ..., v''_n\}) \cup V_i^r])$

A4: $A \rightarrow A'$: $A' = (A - \{a_i, a_j, a_k\})$ (siehe Anhang D.2.2)

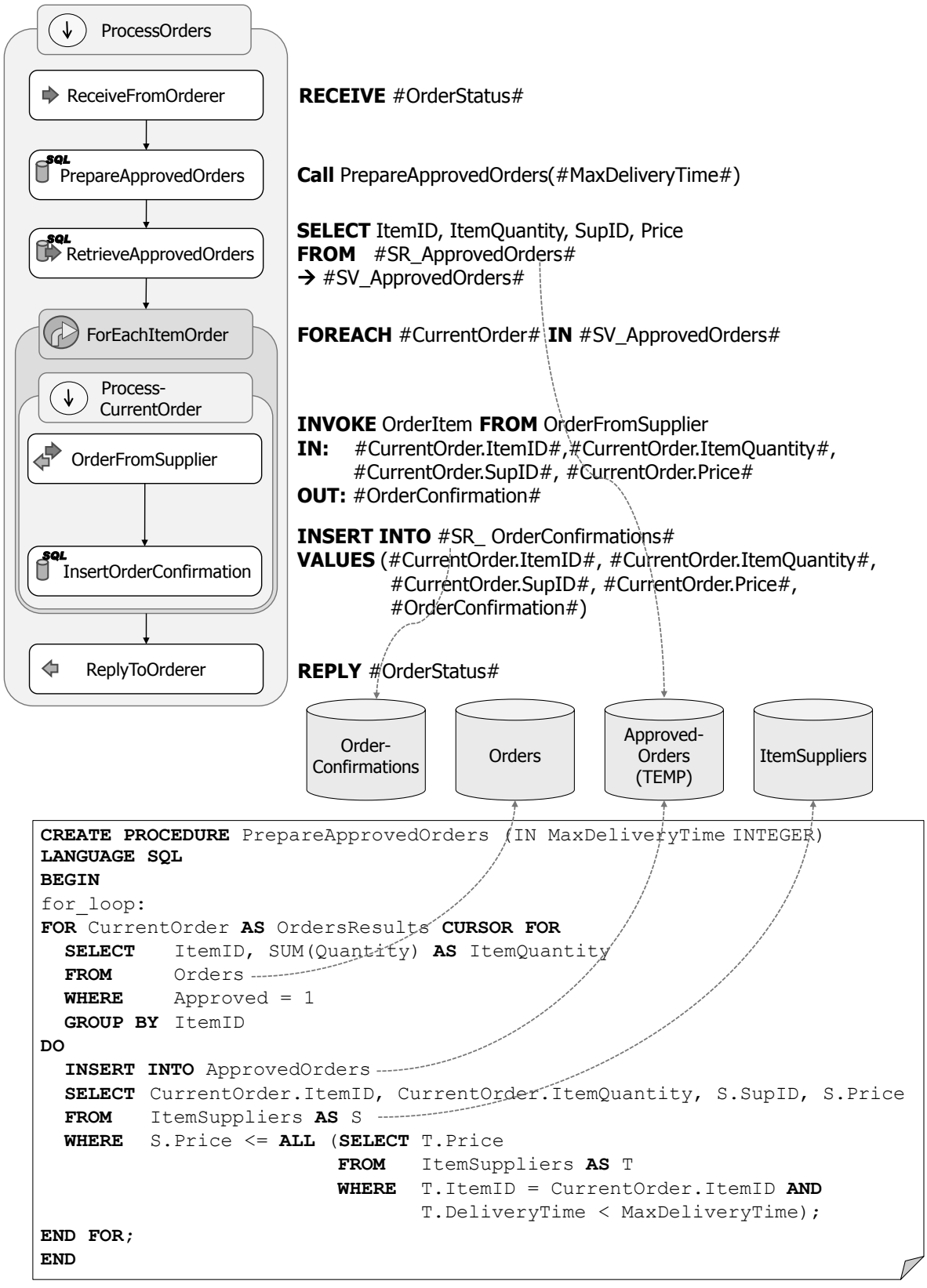
E

Optimierung des Beispielszenarios

Das PGM/F-System unterstützt drei verschiedene Optimierungsvarianten (siehe Teilkapitel 5.1): Bei einer homogenen Optimierung werden nur Datenverarbeitungsoperationen berücksichtigt, die integraler Bestandteil eines Workflowmodells sind. Bei einer heterogenen Optimierung werden zusätzlich die Beschreibungen der vom Workflow heraus aufgerufenen Stored-Procedures beachtet. Hier können die Restrukturierungsregeln jeweils getrennt voneinander auf die Beschreibungen eines Workflows bzw. den Stored-Procedures (heterogene, isolierte Optimierung) oder kombiniert auf diese Beschreibungen angewendet werden (heterogene, kombinierte Optimierung).

In diesem Abschnitt wird die Optimierung des in Teilkapitel 2.1.1 vorgestellten BPEL/-SQL-Workflows, der einen Bestellprozess automatisiert, schrittweise veranschaulicht. Abbildung E.1 zeigt nochmals den zu optimierenden BPEL/SQL-Workflow. Das Datenbankschema, das dem Workflow zu Grunde liegt, zeigt Abbildung E.2. In Abbildung E.3 ist die WSDL-Beschreibung des Web-Services zu sehen, der aus dem BPEL/SQL-Workflow heraus von der <`invoke`> Aktivität *OrderFromSupplier* aufgerufen wird.

Die folgenden Abschnitte E.1 bis E.3 stellen die einzelnen Optimierungsschritte dar, die bei einer homogenen und einer heterogenen isolierten bzw. kombinierten Optimierung des BPEL/SQL-Workflows durchlaufen werden. Dabei werden jeweils die PGM-Repräsentationen, die Software-Artefakte und die Beschreibung des resultierenden BPEL/SQL-Workflows skizziert, die während des Optimierungslaufes vom PGM/F-System generiert werden. Die auf diesen PGM-Repräsentationen angewendeten Restrukturierungsregeln werden in Kapitel 5 im Detail beschrieben.




```
CREATE PROCEDURE PrepareApprovedOrders (IN MaxDeliveryTime INTEGER)
LANGUAGE SQL
BEGIN
for_loop:
FOR CurrentOrder AS OrdersResults CURSOR FOR
  SELECT    ItemID, SUM(Quantity) AS ItemQuantity
  FROM      Orders
  WHERE     Approved = 1
  GROUP BY  ItemID
DO
  INSERT INTO ApprovedOrders
  SELECT CurrentOrder.ItemID, CurrentOrder.ItemQuantity, S.SupID, S.Price
  FROM   ItemSuppliers AS S
  WHERE  S.Price <= ALL (SELECT T.Price
                         FROM   ItemSuppliers AS T
                         WHERE  T.ItemID = CurrentOrder.ItemID AND
                                T.DeliveryTime < MaxDeliveryTime);
END FOR;
END
```

Abbildung E.1.: Automatisierter Bestellprozess mit BPEL/SQL

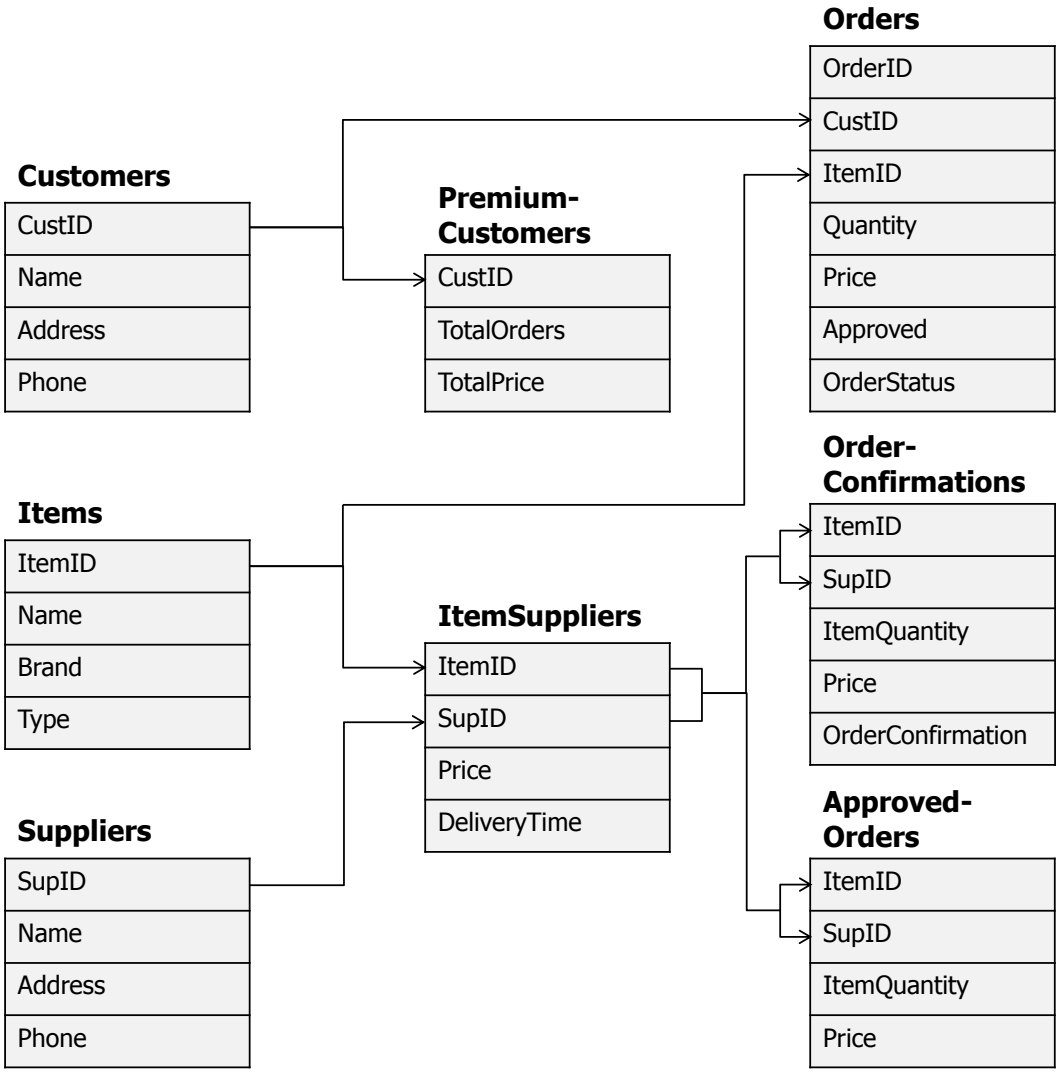


Abbildung E.2.: Datenbankschema des automatisierten Bestellprozesses

```
<?xml version="1.1" encoding="UTF-8"?>
<wsdl:definitions targetNamespace= http://SupplierServices.com ...">

...

// Messages
<wsdl:message name="orderItemRequest">
 <wsdl:part name="itemID"          type="xsd:int"/>
 <wsdl:part name="itemQuantity"    type="xsd:int"/>
 <wsdl:part name="supID"           type="xsd:int"/>
 <wsdl:part name="price"           type="xsd:int"/>
</wsdl:message>

<wsdl:message name="orderItemResponse">
 <wsdl:part name="orderConfirmation" type="xsd:string"/>
</wsdl:message>

<wsdl:message name="orderItemSetRequest">
 <wsdl:part name="itemSet" type="xsd:set"/>
</wsdl:message>

<wsdl:message name="orderItemSetResponse">
 <wsdl:part name="orderConfirmation" type="xsd:string"/>
</wsdl:message>

// PortType
<wsdl:portType name="OrderFromSupplier">
 <wsdl:operation name="orderItem">
   <wsdl:input  message="impl:orderItemRequest"  name="orderItemRequest"/>
   <wsdl:output message="impl:orderItemResponse" name="orderItemResponse"/>
 </wsdl:operation>
 <wsdl:operation name="orderItemSet">
   <wsdl:input  message="impl:orderItemSetRequest"  name="orderItemSetRequest"/>
   <wsdl:output message="impl:orderItemSetResponse" name="orderItemSetResponse"/>
 </wsdl:operation>
</wsdl:portType>

...

// Binding: SOAP over HTTP
<wsdl:binding name="OrderFromSupplierSoapBinding" type="impl:OrderFromSupplier">
 <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="orderItem">
   <wsdlsoap:operation soapAction="" style="rpc"/>
     <wsdl:input name="orderItemRequest">
      <wsdlsoap:body use="encoded" namespace=http://SupplierServices.com
                     encodingStyle="schemas.xmlsoap.org/soap/encoding/"/>
     </wsdl:input>
     <wsdl:output name="orderItemResponse">
      <wsdlsoap:body use="encoded" namespace=http://SupplierServices.com
                      encodingStyle="schemas.xmlsoap.org/soap/encoding/"/>
     </wsdl:output>
 </wsdl:operation>
 <wsdl:operation name="orderItemSet">
   <wsdlsoap:operation soapAction="" style="rpc"/>
     <wsdl:input name="orderItemSetRequest">
      <wsdlsoap:body use="encoded" namespace=http://SupplierServices.com
                      encodingStyle="schemas.xmlsoap.org/soap/encoding/"/>
     </wsdl:input>
...
```

```
...

     <wsdl:output name="orderItemSetResponse">
      <wsdlsoap:body use="encoded" namespace=http://SupplierServices.com
                          encodingStyle="schemas.xmlsoap.org/soap/encoding/"/>
     </wsdl:output>
 </wsdl:operation>
</wsdl:binding>

// Service
<wsdl:service name="OrderFromSupplierService">
 <wsdl:port binding="impl:OrderFromSupplierSoapBinding" name="OrderFromSupplier">
  <wsdlsoap:address location = "http://SupplierServices.com/OrderFromSupplier"/>
 </wsdl:port>
</wsdl:service>

</wsdl:definitions>
```

Abbildung E.3.: WSDL-Beschreibung des Web-Services *OrderFromSupplier*

E.1. Homogene Optimierung des Beispielszenarios

Bei einer homogenen Optimierung werden die Restrukturierungsregeln ausschließlich auf Datenverarbeitungsoperationen angewendet, die integraler Bestandteil des in Abbildung 2.2 dargestellten BPEL/SQL-Workflows sind. Dies bedeutet insbesondere, dass die Stored-Procedure *PrepareApprovedOrders* nicht berücksichtigt wird. Abbildung E.4 stellt die PGM-Repräsentation des BPEL/SQL-Workflows dar, der die Grundlage für die folgenden Regelanwendungen bildet. Im ersten Schritt wird darauf die *Web-Service-Pushdown*-Regel angewendet (siehe Teilkapitel 5.3). Die daraus resultierende PGM-Repräsentation und die generierte WS-UDTF zum Aufruf der Operation *OrderItem* des Web-Services *OrderFromSupplier* werden in Abbildung E.5 bzw. E.6 veranschaulicht. Im nächsten Optimierungsschritt kann die *Select-Into-Merging*-Regel angewendet werden (siehe Teilkapitel 5.4.6). Die resultierende PGM-Repräsentation ist in Abbildung E.7 zu sehen. Basierend auf dieser PGM-Repräsentation ist die Anwendung einer *Insert-Tuple-To-Set*-Regel möglich (siehe Teilkapitel 5.5.3). Daraus resultiert die in Abbildung E.8 dargestellte PGM-Repräsentation, auf die abschließend die Regeln *Eliminate-Unused-Partner* und *Eliminate-Unused-Variable* angewendet werden können. Abbildung E.9 zeigt die sich daraus ergebende finale PGM-Repräsentation, die schließlich auf die in Abbildung E.10 dargestellte optimierte Beschreibung des BPEL/SQL-Workflows abgebildet wird.

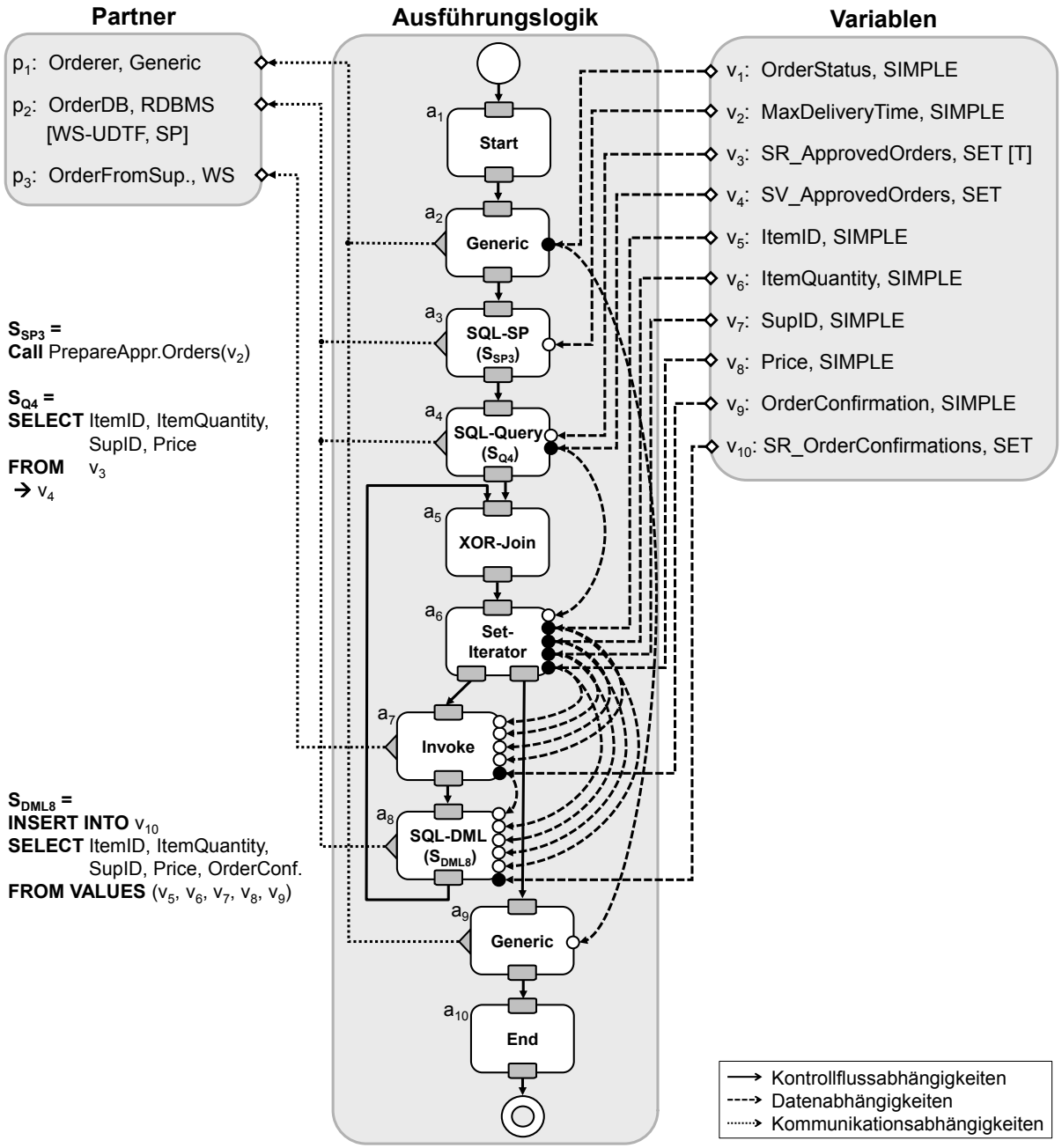


Abbildung E.4.: PGM-Repräsentation des BPEL/SQL-Workflows

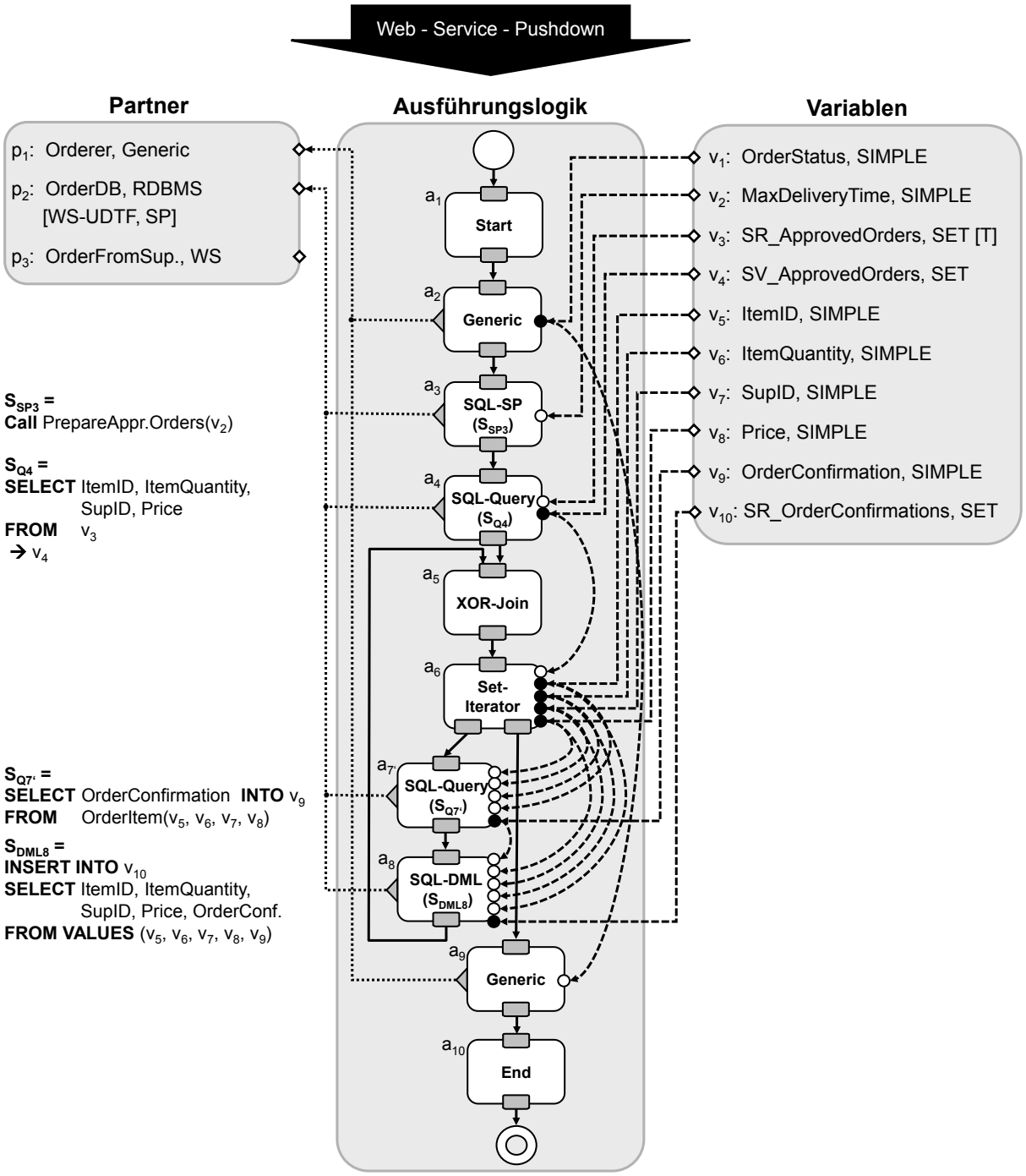


Abbildung E.5.: PGM-Repräsentation des BPEL/SQL-Workflows nach Anwendung der *Web-Service-Pushdown*-Regel

```
CREATE FUNCTION OrderItem(itemID INTEGER, itemQuantity INTEGER,
                          supID  INTEGER, price        INTEGER)
RETURNS VARCHAR(100)
LANGUAGE SQL
CONTAINS SQL
EXTERNAL ACTION
NOT DETERMINISTIC
RETURN
WITH
 -- 1. Prepare SQL input parameters for SOAP envelope
 soap_input(in) AS
 (VALUES  XMLELEMENT(NAME "ns:orderItemRequest",
                     XMLNAMESPACES('http://SupplierServices.com' AS "xmlns:ns",
                                   'http://schemas.xmlsoap.org/soap/encoding/'
                                                              AS "SOAPENV:encodingStyle"),
                     XMLELEMENT(NAME "itemID",    itemID),
                     XMLELEMENT(NAME "quantity",  quantity),
                     XMLELEMENT(NAME "supID",     supID),
                     XMLELEMENT(NAME "price",     price))

  -- 2. Submit SOAP request with input parameter and receive SOAP response
  soap_output(out) AS
  (VALUES db2xml.soaphttpc('http://SupplierServices.com/OrderFromSupplier',
                           '',
                           (SELECT in FROM soap_input))))

  -- 3. Shred SOAP response and get SQL output parameter
  SELECT substr(db2xml.extractVarchar(db2xml.xmlclob(x.out),
                                      //orderConfirmation, 1, 100))
  FROM soap_output x
```

Abbildung E.6.: Die generierte WS-UDTF zum Aufruf der Operation *OrderItem* des Web-Services *OrderFromSupplier*

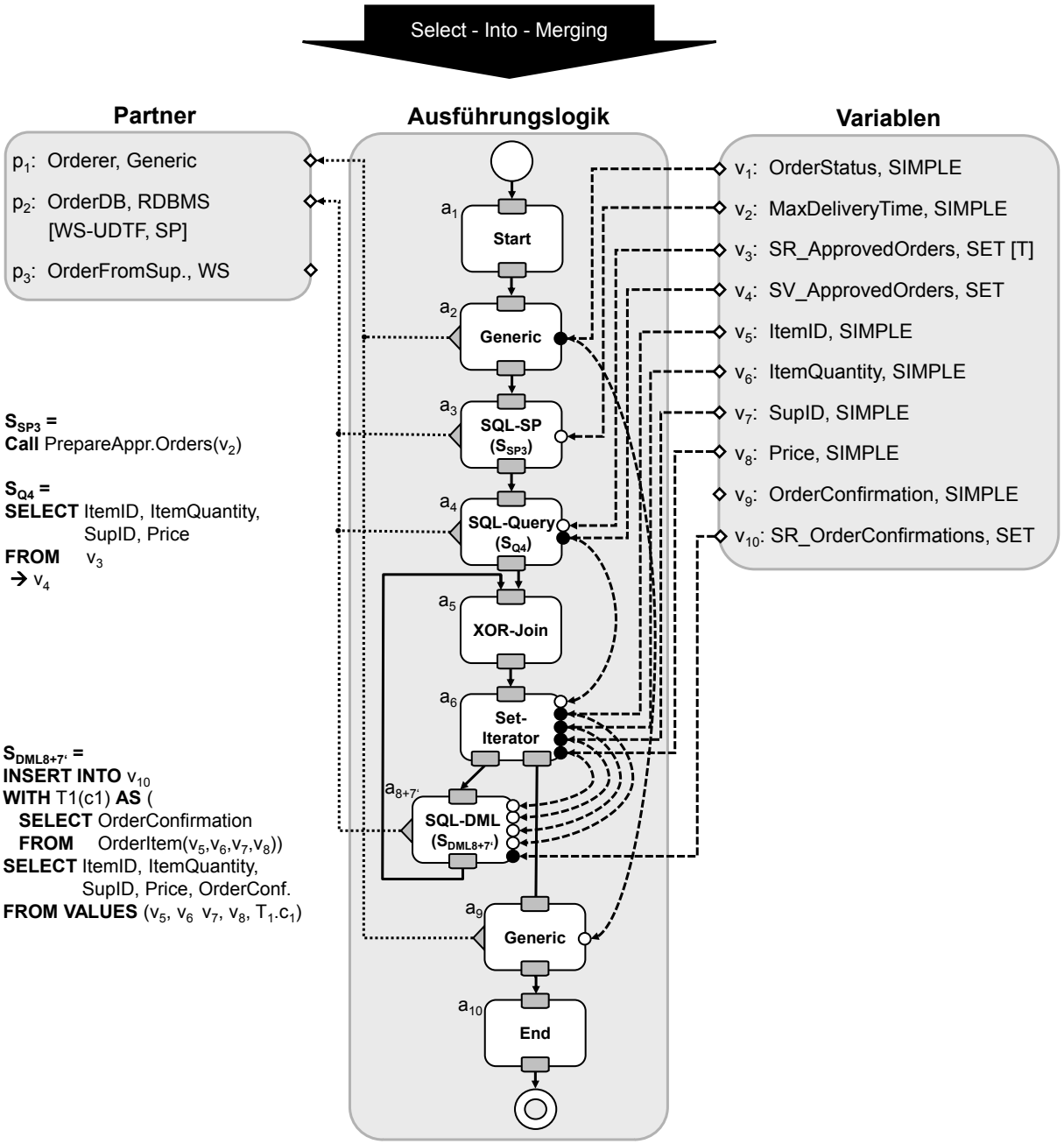


Abbildung E.7.: PGM-Repräsentation des BPEL/SQL-Workflows nach Anwendung der *Select-Into-Merging*-Regel

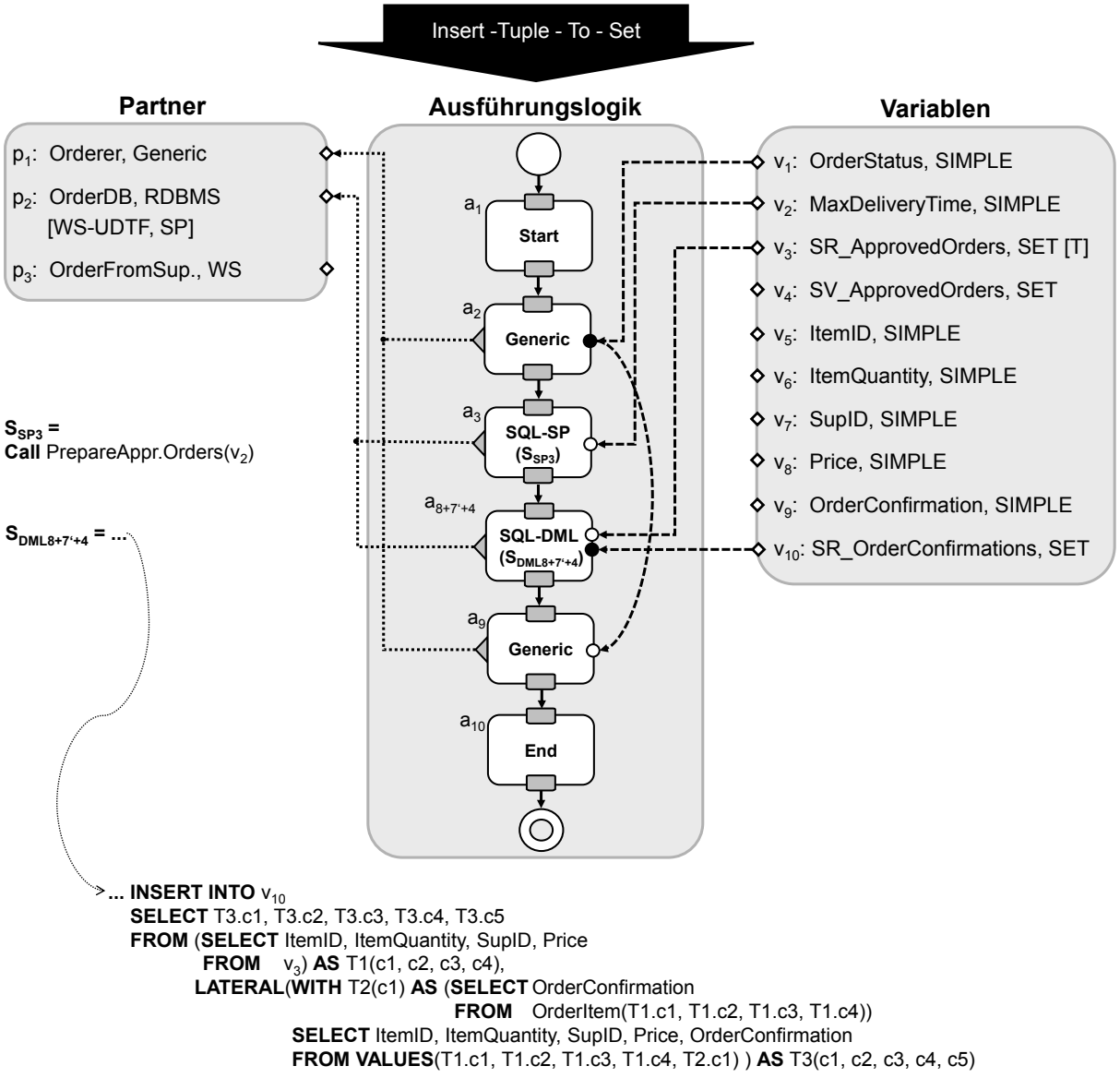


Abbildung E.8.: PGM-Repräsentation des BPEL/SQL-Workflows nach Anwendung der *Insert-Tuple-To-Set*-Regel

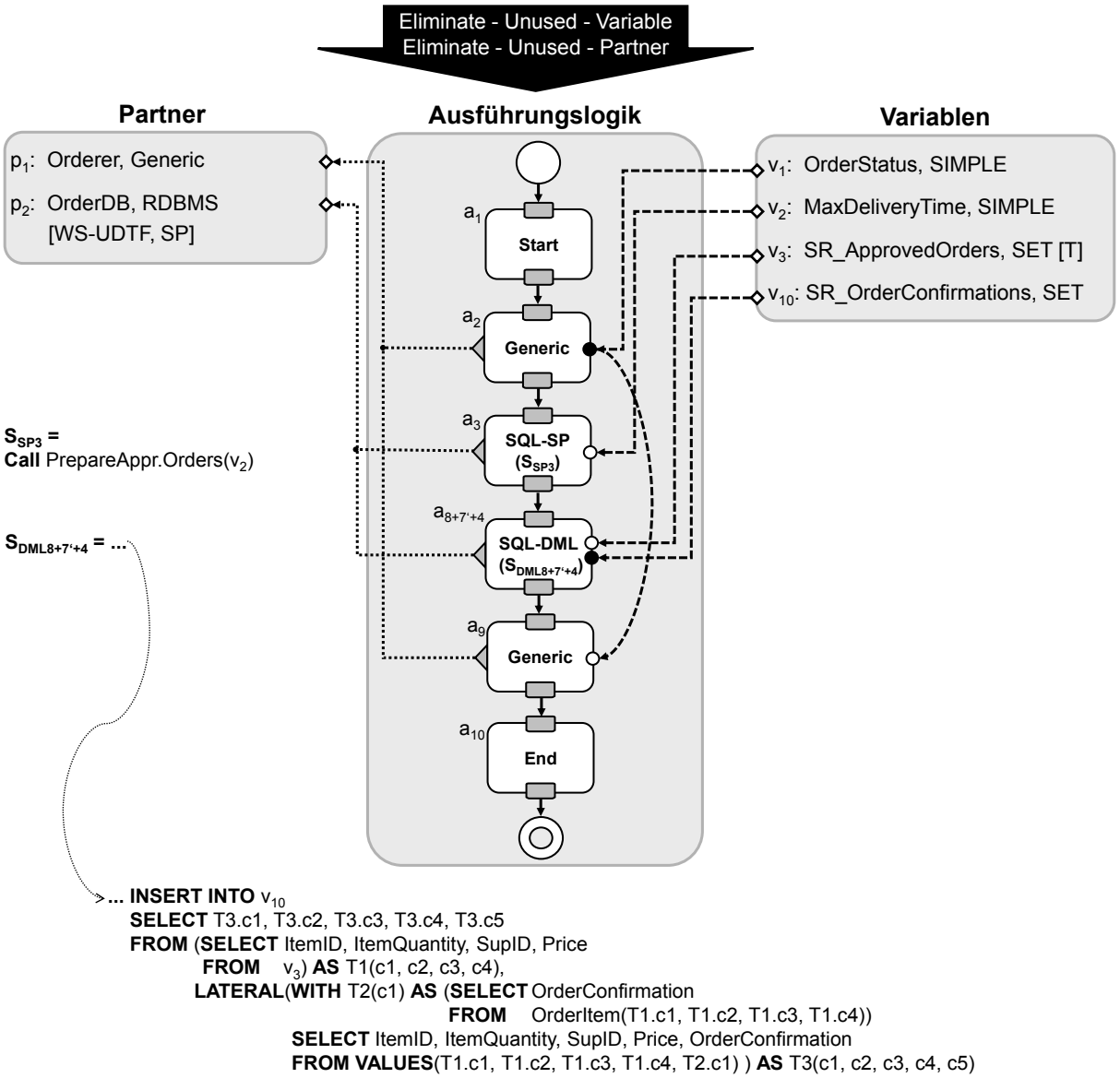


Abbildung E.9.: Finale PGM-Repräsentation des BPEL/SQL-Workflows nach Anwendung der Regeln *Eliminate-Unused-Variable* bzw. *Eliminate-Unused-Partner*

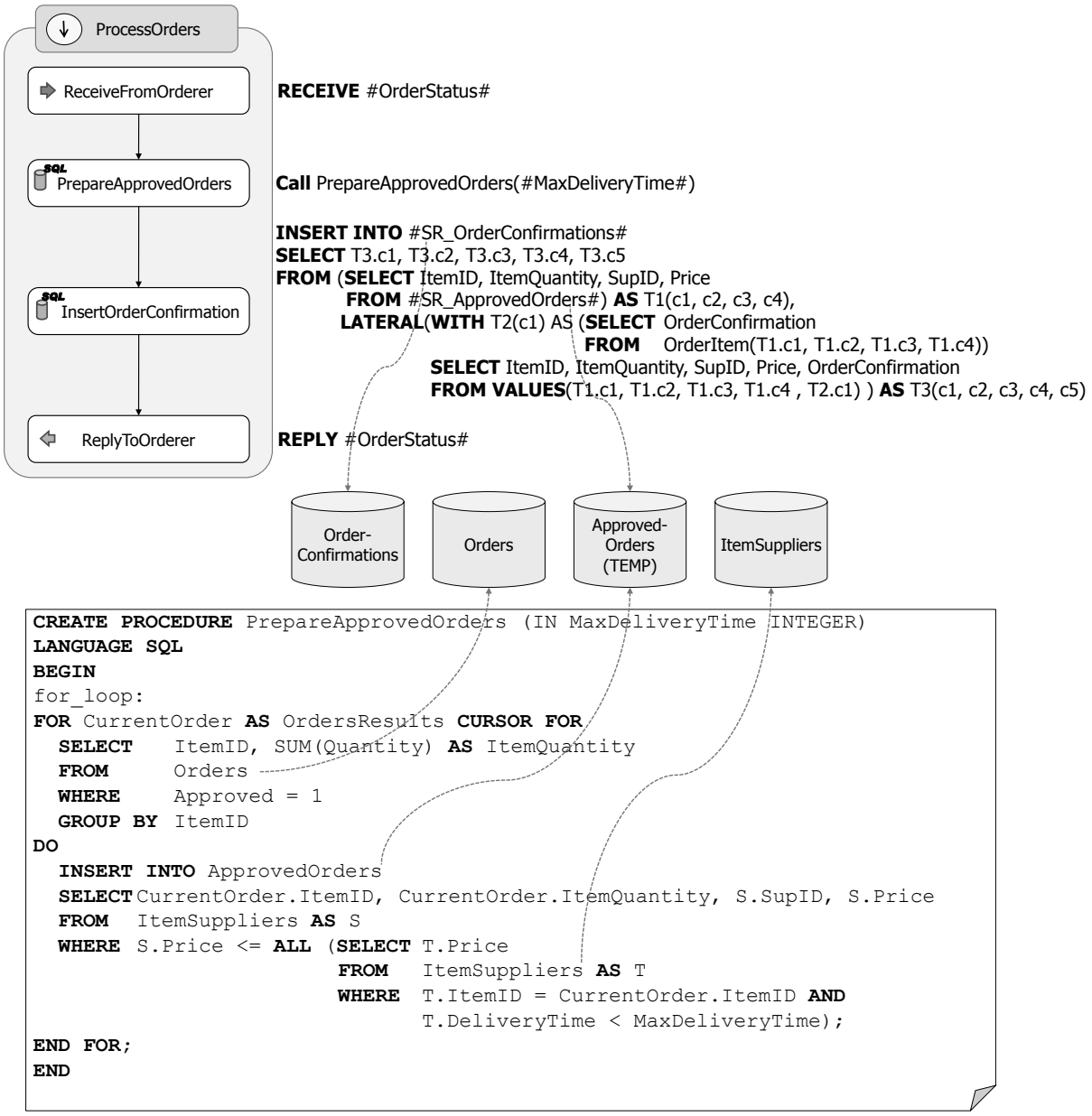


Abbildung E.10.: BPEL/SQL-Workflow nach homogener, isolierter Optimierung

E.2. Heterogene, isolierte Optimierung des Beispielszenarios

Im heterogenen, isolierten Optimierungsszenario werden die Restrukturierungsregeln auf den BPEL/SQL-Workflow und die darin aufgerufene Stored-Procedure *PrepareApproved-Orders* angewendet (siehe Abbildung E.4). Die zur Optimierung des Workflows notwendigen Transformationsschritte wurden bereits in Abschnitt E.1 veranschaulicht.

Deshalb liegt der Fokus in diesem Abschnitt auf der Darstellung der einzelnen Transformationsschritte zur Optimierung der Stored-Procedure *PrepareApprovedOrders*. Abbildung E.11 zeigt die PGM-Repräsentation dieser Stored-Procedure, auf welche die *Insert-Tuple-To-Set*-Regel angewendet werden kann (siehe Teilkapitel 5.5.3). Das Ergebnis dieser Regelanwendung wird in Abbildung E.12 veranschaulicht. Die finale PGM-Repräsentation ergibt sich nach Anwendung der *Eliminate-Unused-Variable*-Regel, die in Abbildung E.13 zu sehen ist. Die daraus abgeleitete, optimierte Stored-Procedure-Beschreibung ist zusammen mit dem optimierten BPEL/SQL-Workflow in Abbildung E.14 dargestellt.

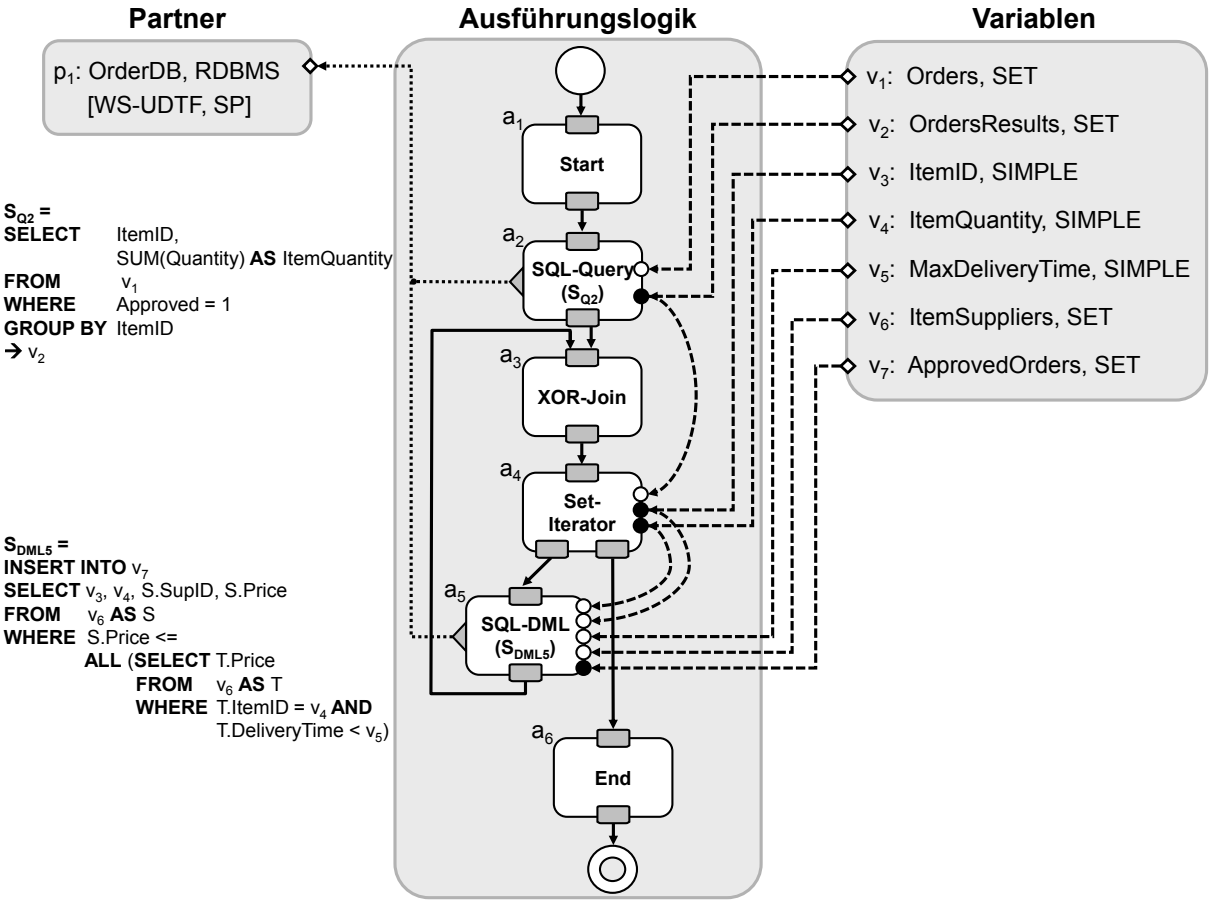


Abbildung E.11.: PGM-Repräsentation der Stored-Procedure *PrepareApprovedOrders*

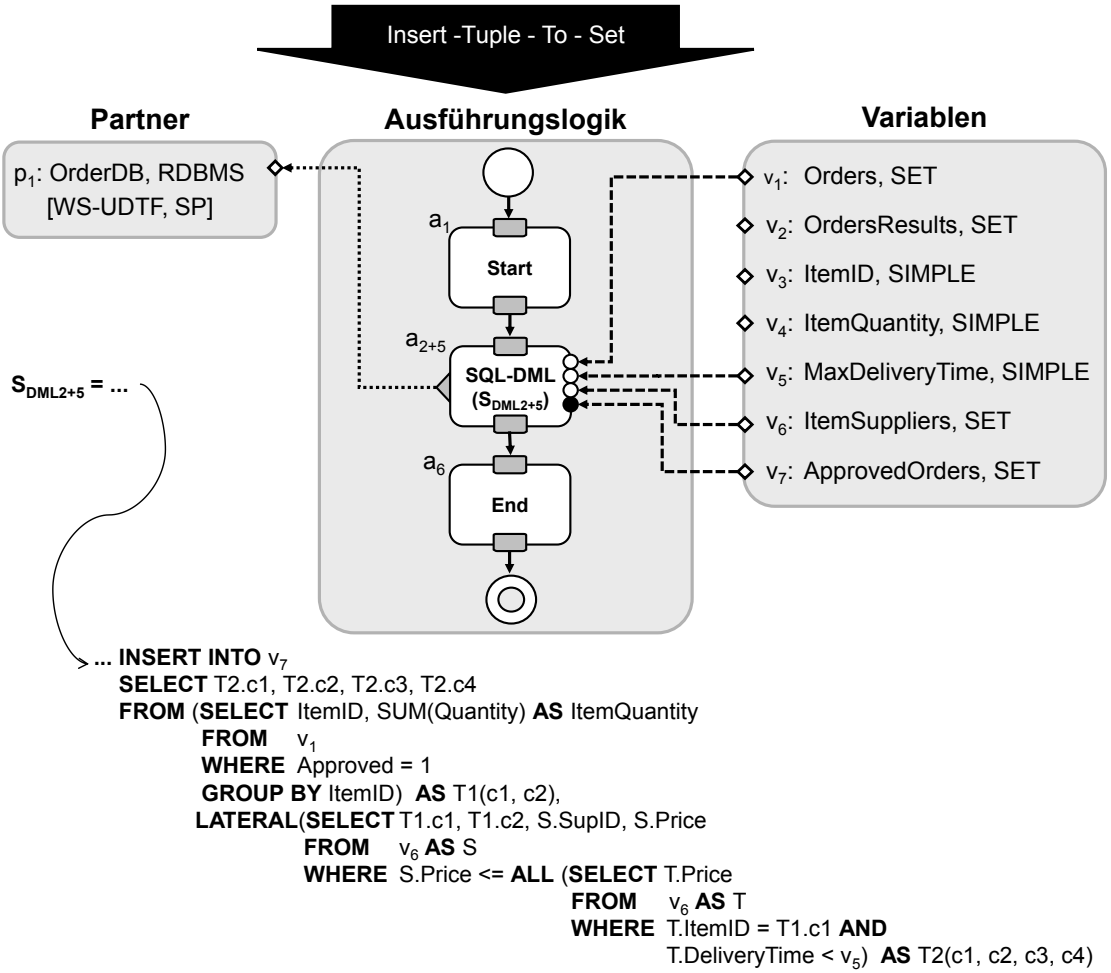


Abbildung E.12.: PGM-Repräsentation der Stored-Procedure *PrepareApprovedOrders* nach Anwendung der *Insert-Tuple-To-Set*-Regel

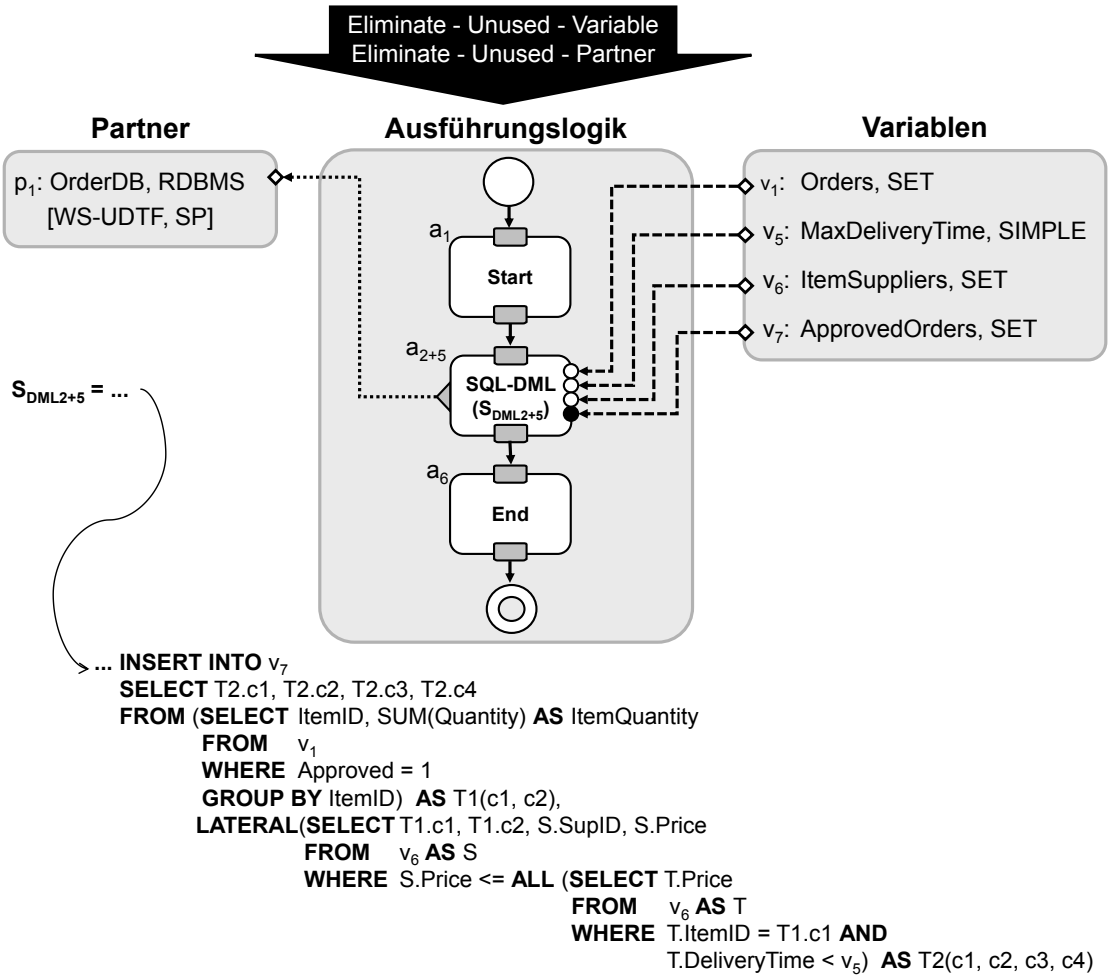
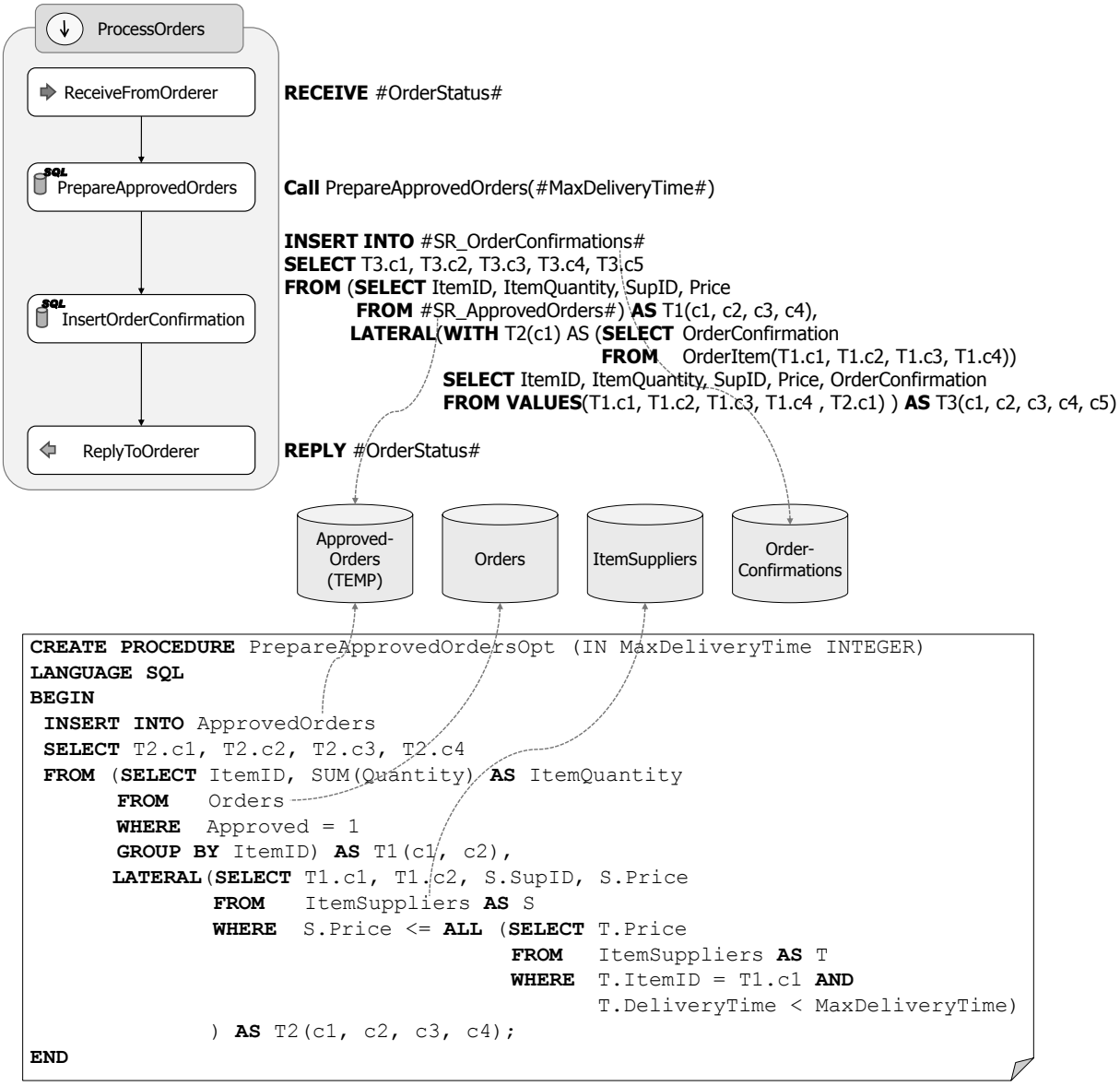


Abbildung E.13.: Finale PGM-Repräsentation der Stored-Procedure *PrepareApprovedOrders* nach Anwendung der *Eliminate-Unused-Variable*-Regel




```
CREATE PROCEDURE PrepareApprovedOrdersOpt (IN MaxDeliveryTime INTEGER)
LANGUAGE SQL
BEGIN
 INSERT INTO ApprovedOrders
 SELECT T2.c1, T2.c2, T2.c3, T2.c4
 FROM (SELECT ItemID, SUM(Quantity) AS ItemQuantity
       FROM   Orders
       WHERE  Approved = 1
       GROUP BY ItemID) AS T1(c1, c2),
      LATERAL(SELECT T1.c1, T1.c2, S.SupID, S.Price
              FROM   ItemSuppliers AS S
              WHERE  S.Price <= ALL (SELECT T.Price
                                     FROM   ItemSuppliers AS T
                                     WHERE  T.ItemID = T1.c1 AND
                                            T.DeliveryTime < MaxDeliveryTime)
              ) AS T2(c1, c2, c3, c4);
END
```

Abbildung E.14.: BPEL/SQL-Workflow nach heterogener, isolierter Optimierung

E.3. Heterogene, kombinierte Optimierung des Beispielszenarios

In diesem Abschnitt werden die Transformationsschritte zur heterogenen, kombinierten Optimierung des in Abbildung 2.2 dargestellten BPEL/SQL-Workflows veranschaulicht. Hierbei werden die Restrukturierungsregeln auf eine kombinierte PGM-Repräsentation angewendet, die sowohl den BPEL/SQL-Workflow als auch die darin aufgerufene Stored-Procedure *PrepareApprovedOrders* repräsentiert. Diese kombinierte PGM-Repräsentation ist in Abbildung E.15 veranschaulicht. Sie setzt sich aus den in Abbildung E.4 und E.11 dargestellten einzelnen PGM-Repräsentationen des BPEL/SQL-Workflows und der Stored-Procedure zusammen. Eine kombinierte PGM-Repräsentation lässt sich aus der PGM-Repräsentation des BPEL/SQL-Workflows ableiten, indem die *SQL-SP*-Aktivität (a_3), welche die Stored-Procedure *PrepareApprovedOrders* aufruft, durch die PGM-Repräsentation der Stored-Procedure ersetzt wird. Damit verbunden ist eine Kombination der Variablen- und Partnerdefinitionen beider PGM-Repräsentationen. Auf dieser kombinierten PGM-Repräsentation können die Regeln *Web-Service-Pushdown*, *Select-Into-Merging* und *Insert-Tuple-To-Set*, wie in den Abschnitten E.1 und E.2 gezeigt, angewendet werden. Abbildung E.16 zeigt die hieraus resultierende PGM-Repräsentation. Auf diese PGM-Repräsentation lässt sich anschließend die *Eliminate-Temporary-Table*-Regel anwenden (siehe Teilkapitel 5.4.7). Das Ergebnis dieser Regelanwendung ist in Abbildung E.17 illustriert. Die Anwendung dieser Regel wäre ohne eine kombinierte PGM-Repräsentation nicht möglich gewesen, da sie Datenverarbeitungsoperationen berücksichtigt, die ursprünglich sowohl im BPEL/SQL-Workflow als auch in der Stored-Procedure *PrepareApprovedOrders* definiert waren. Die finale PGM-Repräsentation in Abbildung E.18 ergibt sich schließlich durch Anwendung der Regeln *Eliminate-Unused-Variable* und *Eliminate-Unused-Partner*. Die finale Beschreibung des optimierten BPEL/SQL-Workflows ist schließlich in Abbildung E.19 zu sehen.

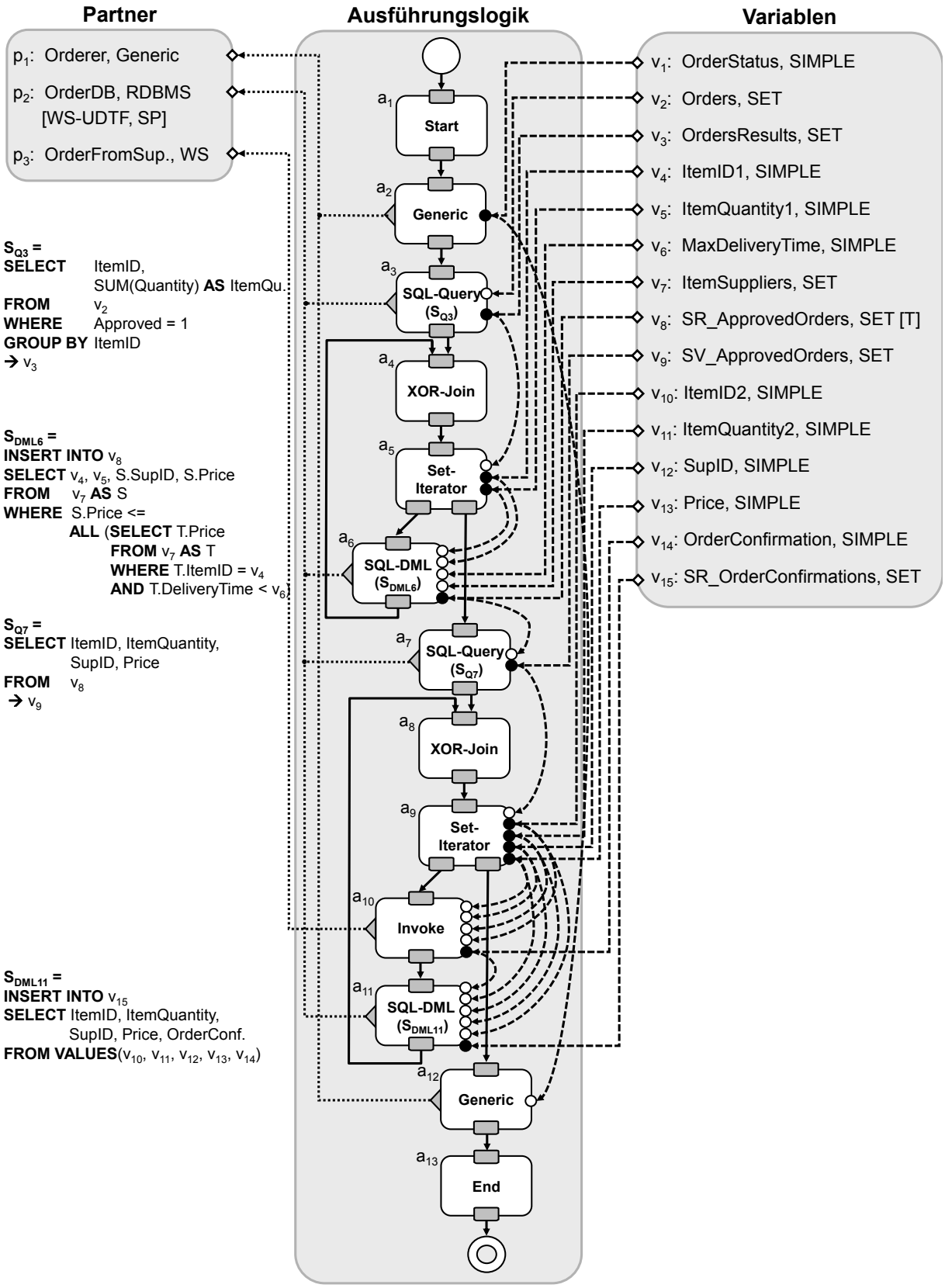


Abbildung E.15.: Kombinierte PGM-Repräsentation des BPEL/SQL-Workflows und der Stored-Procedure *PrepareApprovedOrders*

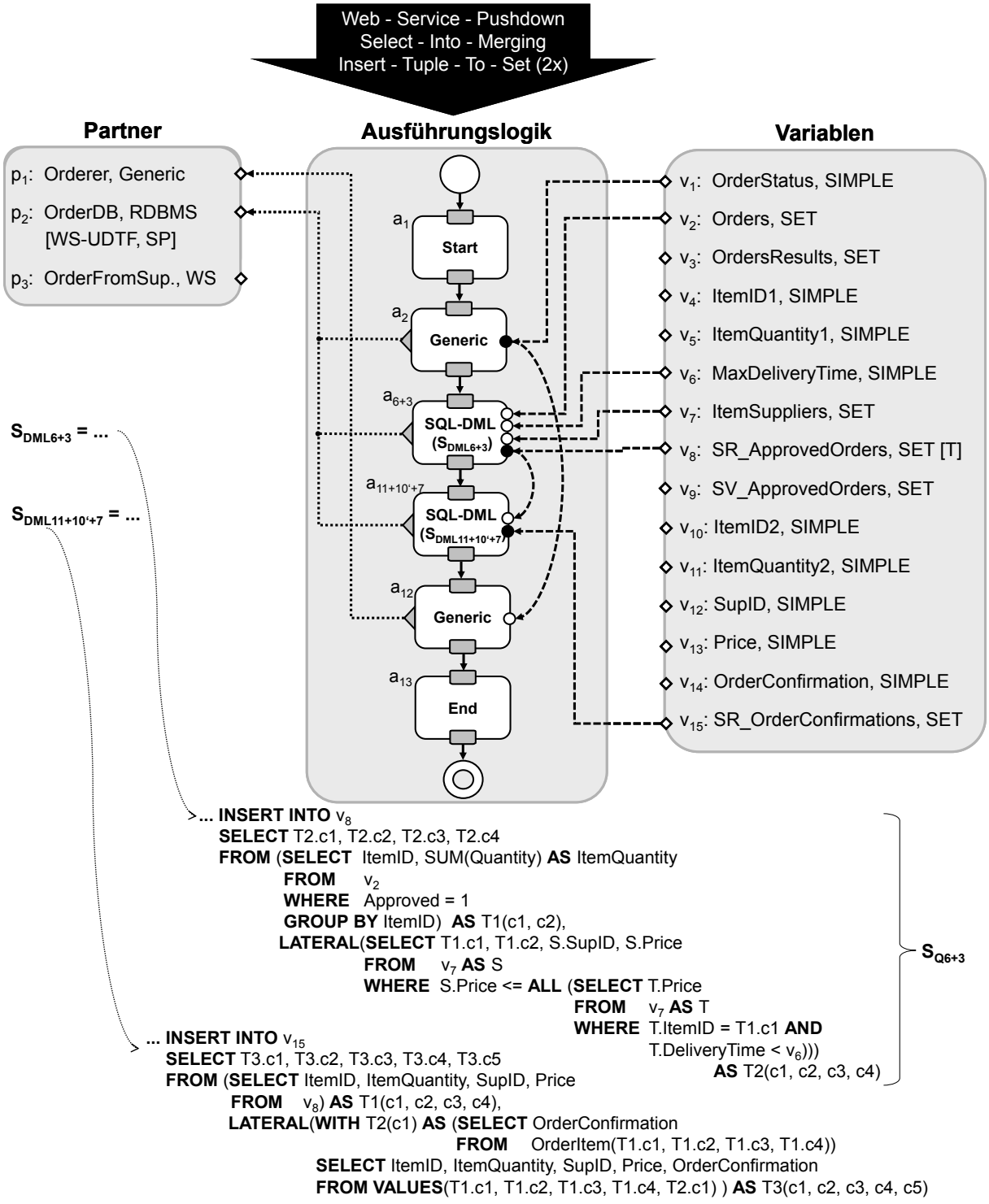


Abbildung E.16.: Kombinierte PGM-Repräsentation nach Anwendung der Regeln *Web-Service-Pushdown*, *Select-Into-Merging* und *Insert-Tuple-To-Set*

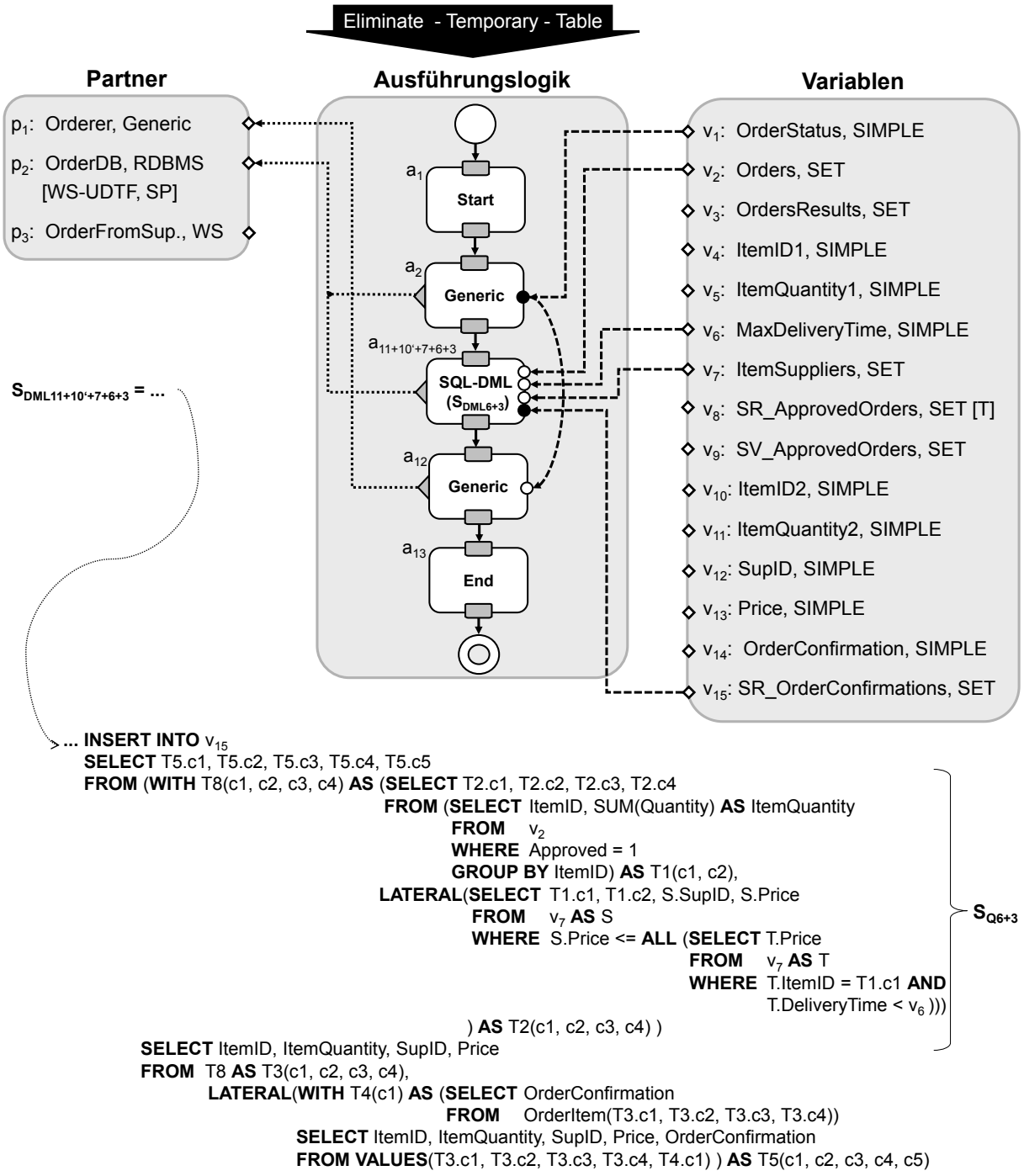
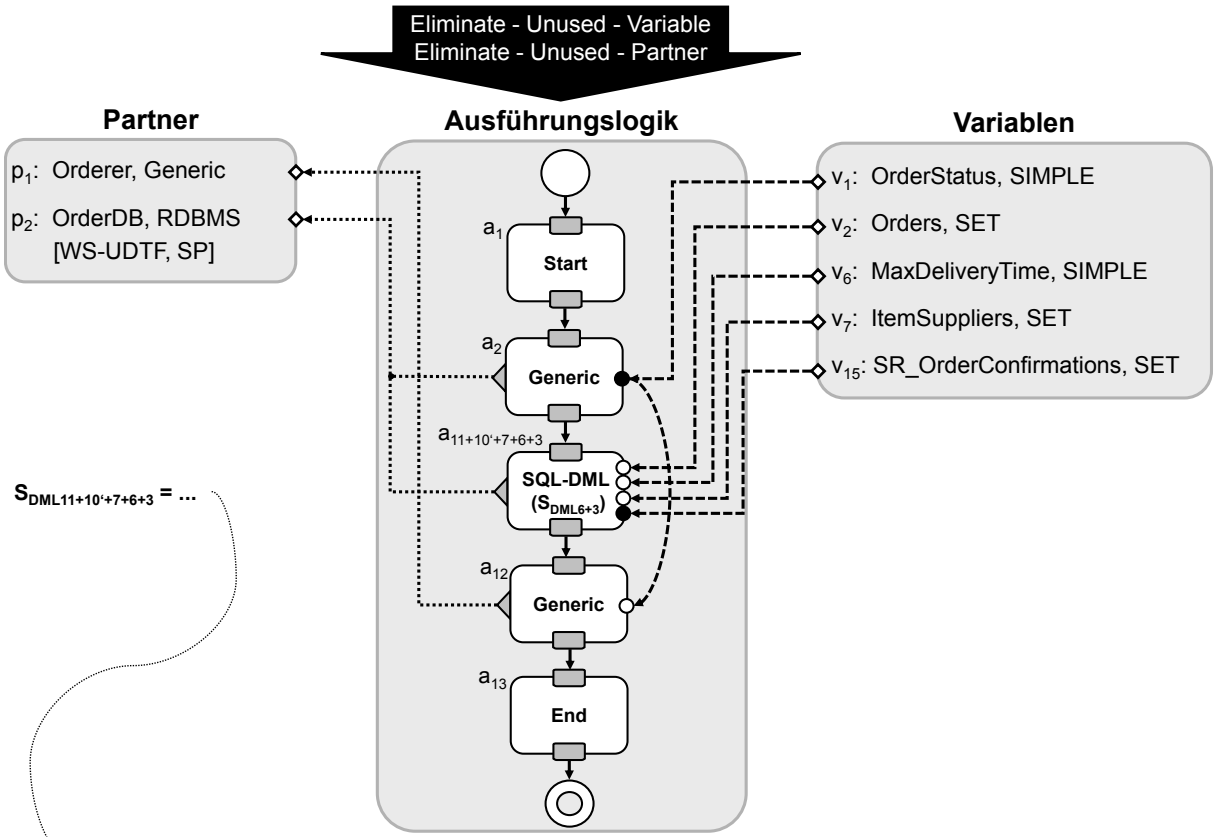


Abbildung E.17.: Kombinierte PGM-Repräsentation nach Anwendung der *Eliminate-Temporary-Table*-Regel




```
... INSERT INTO v15
    SELECT T5.c1, T5.c2, T5.c3, T5.c4, T5.c5
    FROM (WITH T8(c1, c2, c3, c4) AS (SELECT T2.c1, T2.c2, T2.c3, T2.c4
                                      FROM (SELECT ItemID, SUM(Quantity) AS ItemQuantity
                                            FROM   v2
                                            WHERE  Approved = 1
                                            GROUP BY ItemID) AS T1(c1, c2),
                                      LATERAL(SELECT T1.c1, T1.c2, S.SupID, S.Price
                                              FROM   v7 AS S
                                              WHERE  S.Price <= ALL (SELECT T.Price
                                                                     FROM   v7 AS T
                                                                     WHERE  T.ItemID = T1.c1 AND
                                                                            T.DeliveryTime < v6 )))
                                      ) AS T2(c1, c2, c3, c4) )
          SELECT ItemID, ItemQuantity, SupID, Price
          FROM  T8 AS T3(c1, c2, c3, c4),
                LATERAL(WITH T4(c1) AS (SELECT OrderConfirmation
                                        FROM   OrderItem(T3.c1, T3.c2, T3.c3, T3.c4))
                        SELECT ItemID, ItemQuantity, SupID, Price, OrderConfirmation
                        FROM VALUES(T3.c1, T3.c2, T3.c3, T3.c4, T4.c1) ) AS T5(c1, c2, c3, c4, c5)
```

SQ6+3

Abbildung E.18.: Finale, kombinierte PGM-Repräsentation nach Anwendung der Regeln *Eliminate-Unused-Variable* bzw. *Eliminate-Unused-Partner*

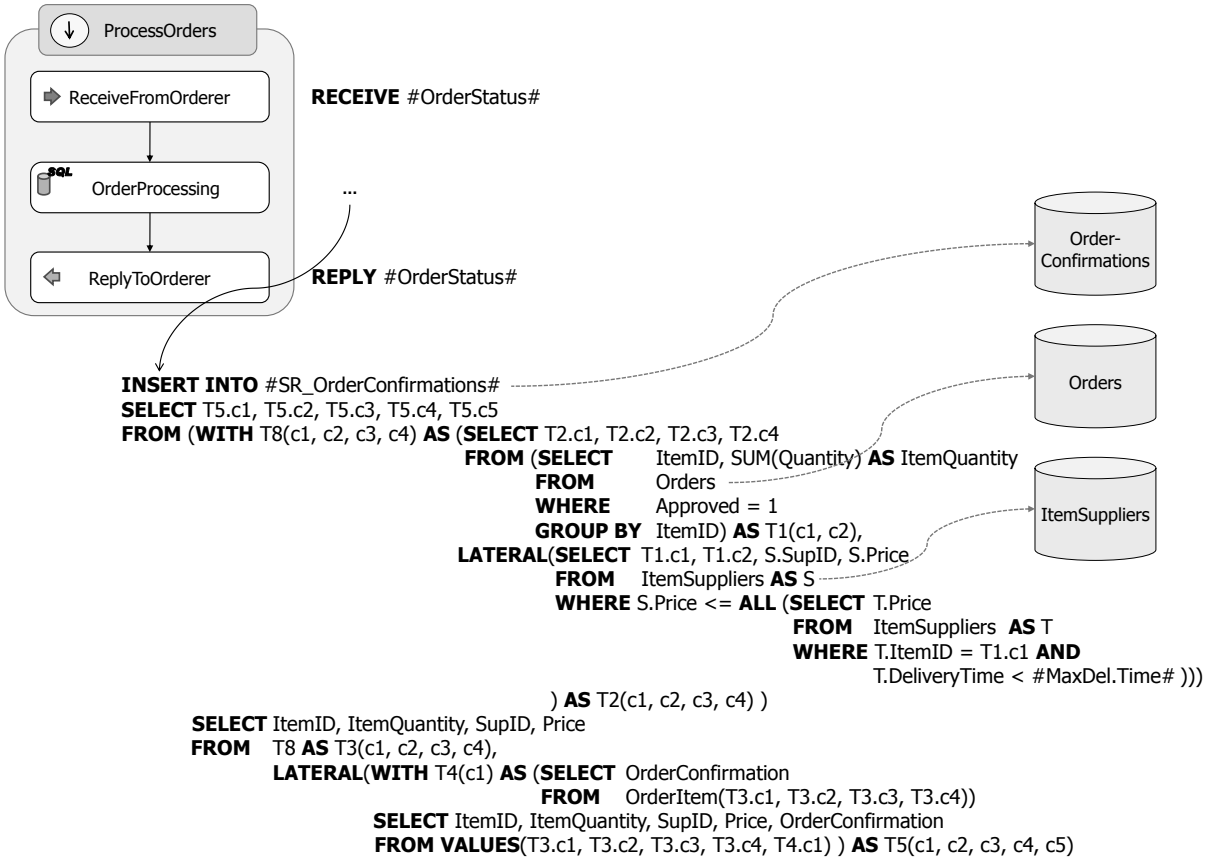


Abbildung E.19.: BPEL/SQL-Workflow nach heterogener, kombinierter Optimierung

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Aal97] Aalst, Wil M. P. d.: Verification of Workflow Nets. In: *Proceedings of the 18th International Conference on Application and Theory of Petri Nets (ICATPN 1997)*. Toulouse, France, June 1997, S. 407–426

[Aal99] Aalst, Wil M. P. d.: Formalization and Verification of Event-driven Process Chains. In: *Information and Software Technology* 41 (1999), Nr. 10, S. 639–650

[Aal03] Aalst, Wil M. P. d.: Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management. In: Desel, Jörg (Hrsg.); Reisig, Wolfgang (Hrsg.); Rozenberg, Grzegorz (Hrsg.): *Lectures on Concurrency and Petri Nets* Bd. 3098, Springer, 2003 (Lecture Notes in Computer Science), S. 1–65

[ABC+04] Abiteboul, Serge; Benjelloun, Omar; Cautis, Bogdan; Manolescu, Ioana; Milo, Tova; Preda, Nicoleta: Lazy Query Evaluation for Active XML. In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD 2004)*. Paris, France, June 2004, S. 227–238

[ABM08] Abiteboul, Serge; Benjelloun, Omar; Milo, Tova: The Active XML Project: An Overview. In: *The VLDB Journal* 17 (2008), Nr. 5, S. 1019–1040

[ACN06] Agrawal, Sanjay; Chu, Eric; Narasayya, Vivek: Automatic Physical Design Tuning: Workload as a Sequence. In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD 2006)*. Chicago, Illinois, USA, June 2006, S. 683–694

[AH00] Avnur, Ron; Hellerstein, Joseph M.: Eddies: Continuously Adaptive Query Processing. In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD 2000)*. Dallas, Texas, USA, May 2000, S. 261–272

[AHKB03] Aalst, Wil M. P. d.; Hofstede, Arthur H. M.; Kiepuszewski, Bartek; Barros, Alistair P.: Workflow Patterns. In: *Distributed and Parallel Databases* 14 (2003), Nr. 1, S. 5–51

[ALSU06] Aho, Alfred V.; Lam, Monica S.; Sethi, Ravi; Ullman, Jeffrey D.: *Compilers: Principles, Techniques and Tools*. Addison Wesley, 2006

[AMA06] Akram, Asif; Meredith, David; Allan, Rob: Evaluation of BPEL to Scientific Workflows. In: *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, 2006, S. 269–274

[AV04] Aalst, Wil van d.; Vanhee, Kees: *Workflow Management: Models, Methods, and Systems*. The MIT Press, 2004

[BCE+08] Borkar, Vinayak R.; Carey, Michael J.; Engovatov, Daniel; Lychagin, Dmitry; Westmann, Till; Wong, Warren: XQSE: An XQuery Scripting Extension for the AquaLogic Data Services Platform. In: *Proceedings of the 24th International Conference on Data Engineering (ICDE 2008)*. Cancún, México, April 2008, S. 1229–1238

[Bel03] Bellman, Richard E.: *Dynamic Programming*. Dover Publications, Incorporated, 2003

[BG05] Botev, Chavdar; Guo, Lin: Yet another Query Graph Model (YQGM) / Cornell University. 2005. – Forschungsbericht

[BGK+04] Blow, Michael; Goland, Yaron; Kloppmann, Matthias; Leymann, Frank; Pfau, Gerhard; Roller, Dieter; Rowley, Michael: BPELJ: BPEL for Java. In: *Joint white paper by BEA and IBM* (2004)

[Böh11] Böhm, Matthias: *Cost-Based Optimization of Integration Flows*, Technische Universität Dresden, Dissertation, 2011

[BS04] Bowman, Ivan T.; Salem, Kenneth: Optimization of Query Streams Using Semantic Prefetching. In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD 2004)*. Paris, France, June 2004, S. 179–190

[BW01] Babu, Shivnath; Widom, Jennifer: Continuous Queries over Data Streams. In: *SIGMOD Record* 30 (2001), Nr. 3, S. 109–120

[CDTW00] Chen, Jianjun; DeWitt, David J.; Tian, Feng; Wang, Yuan: NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD 2000)*. Dallas, Texas, USA, May 2000, S. 379–390

[Cha98] Chaudhuri, Surajit: An Overview of Query Optimization in Relational Systems. In: *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1998)*. Seattle, Washington, USA, June 1998, S. 34–43

[CKPS95] Chaudhuri, Surajit; Krishnamurthy, Ravi; Potamianos, Spyros; Shim, Kyuseok: Optimizing Queries with Materialized Views. In: *Proceedings of the 11th International Conference on Data Engineering (ICDE 1995)*. Taipei, Taiwan, March 1995, S. 190–200

[CKS+00] Carey, Michael J.; Kiernan, Jerry; Shanmugasundaram, Jayavel; Shekita, Eugene J.; Subramanian, Subbu N.: XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In: *Proceedings of*

the 26th International Conference on Very Large Data Bases (VLDB 2000). Cairo, Egypt, September 2000, S. 646–648

[Cod72] Codd, E. F.: Relational Completeness of Data Base Sublanguages. In: *In: R. Rustin (ed.): Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987, San Jose, California* (1972)

[CS94] Chaudhuri, Surajit; Shim, Kyuseok: Including Group-By in Query Optimization. In: *Proceedings of 20th International Conference on Very Large Data Bases (VLDB 1994).* Santiago de Chile, Chile, September 1994, S. 354–366

[CS95] Chaudhuri, Surajit; Shim, Kyuseok: An Overview of Cost-based Optimization of Queries with Aggregates. In: *IEEE Data Engineering Bulletin* 18 (1995), Nr. 3, S. 3–9

[CS96] Chaudhuri, Surajit; Shim, Kyuseok: Optimizing Queries with Aggregate Views. In: *Proceedings of the 5th International Conference on Extending Database Technology (EDBT 1996).* Avignon, France, March 1996, S. 167–182

[DHW+08] Dessloch, Stefan; Hernández, Mauricio A.; Wisnesky, Ryan; Radwan, Ahmed; Zhou, Jindan: Orchid: Integrating Schema Mapping and ETL. In: *Proceedings of the 24th International Conference on Data Engineering (ICDE 2008).* Cancún, México, April 2008, S. 1307–1316

[Dvt05] Dumas, Marlon (Hrsg.); van der Aalst, Wil M. (Hrsg.); ter Hofstede, Arthur H. (Hrsg.): *Process-aware Information Systems: Bridging People and Software through Process Technology.* Hoboken, NJ, 2005

[EF] Eclipse-Foundation: *Eclipse Development Platform.* `http://www.eclipse.org`, Letzter Zugriff: Oktober 2010

[EGLT76] Eswaran, K. P.; Gray, J. N.; Lorie, R. A.; Traiger, I. L.: The Notions of Consistency and Predicate Locks in a Database System. In: *Communications of the ACM* 19 (1976), Nr. 11, S. 624–633

[FGW+05] Folkert, Nathan; Gupta, Abhinav; Witkowski, Andrew; Subramanian, Sankar; Bellamkonda, Srikanth; Shankar, Shrikanth; Bozkaya, Tolga; Sheng, Lei: Optimizing Refresh of a Set of Materialized Views. In: *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB 2005).* Trondheim, Norway, August 2005, S. 1043–1054

[Fin82] Finkelstein, Sheldon: Common Expression Analysis in Database Applications. In: *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data (SIGMOD 1982).* Orlando, Florida, USA, June 1982, S. 235–245

[GD87] Greafe, G.; DeWitt, D. J.: The EXODUS Optimizer Generator. In: *Proceedings of the Association for Computing Machinery Special Interest Group on Management Data.* San Francisco, California, USA, May 1987, S. 160–172

[GHS95] Georgakopoulos, Dimitrios; Hornick, Mark F.; Sheth, Amit P.: An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. In: *Distributed and Parallel Databases* 3 (1995), Nr. 2, S. 119–153

[GL01] Goldstein, Jonathan; Larson, Per-Åke: Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. In: *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD 2001)*. Santa Barbara, California, USA, May 2001, S. 331–342

[GLPT76] Gray, Jim; Lorie, Raymond A.; Putzolu, Gianfranco R.; Traiger, Irving L.: Granularity of Locks and Degrees of Consistency in a Shared Data Base. In: *IFIP Working Conference on Modelling in Data Base Management Systems*. Freudenstadt, Germany, January 1976, S. 365–394

[GM93] Greafe, G.; McKenna, W. J.: The VOLCANO Optimizer Generator: Extensibility and Efficient Search. In: *Proceedings of the 9th International Conference on Data Engineering (ICDE 1993)*. Vienna, Austria, April 1993, S. 209–218

[GR93] Gray, Jim; Reuter, Andreas: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993

[Gra95] Graefe, Goetz: The Cascades Framework for Query Optimization. In: *IEEE Data Engineering Bulletin* 18 (1995), Nr. 3, S. 19–29

[GSW96] Guo, Sha; Sun, Wei; Weiss, Mark A.: On Satisfiability, Equivalence, and Implication Problems Involving Conjunctive Queries in Database Systems. In: *IEEE Transactions on Knowledge and Data Engineering* 8 (1996), Nr. 4, S. 604–616

[HFLP89] Haas, Laura M.; Freytag, Johann C.; Lohman, Guy M.; Pirahesh, Hamid: Extensible Query Processing in Starburst. In: *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data (SIGMOD 1989)*. Portland, Oregon, USA, May 1989, S. 377–388

[HH07] Hartig, Olaf; Heese, Ralf: The SPARQL Query Graph Model for Query Optimization. In: *Proceedings of the 4th European conference on The Semantic Web (ESWC 2007)*. Innsbruck, Austria, 2007, S. 564–578

[HKS92] Hoffmann, W.; Kirsch, J.; Scheer, A.-W.: Modellierung mit Ereignisgesteuerten Prozessketten: Methodenhandbuch. In: *Veröffentlichungen des Instituts für Wirtschaftsinformatik, Saarbrücken: Universität des Saarlandes* (1992)

[HR79] Hunt, Harry B.; Rosenkrantz, Daniel J.: The Complexity of Testing Predicate Locks. In: *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data (SIGMOD 1979)*. Boston, Massachusetts, USA, May 1979 1979, S. 127–133

[HR01] Härder, Theo; Rahm, Erhard: *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Springer, 2001

[IBMa] IBM: *DB2 Database für Linux, UNIX und Windows v.9.7*. http://www-01.ibm.com/software/data/db2/linux-unix-windows/, Letzter Zugriff: Oktober 2010

[IBMb] IBM: *DB2 pureXML-Technologie*. http://www-01.ibm.com/software/data/db2/xml/, Letzter Zugriff: Oktober 2010

[IBMc] IBM: *Web-Services-Consumer-Functions*. http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/, Letzter Zugriff: Oktober 2010

[IBMd] IBM: *WebSphere Federation Server*. http://www-306.ibm.com/software/data/integration/federation_server/, Letzter Zugriff: Oktober 2010

[IBMe] IBM: *WebSphere Integration Developer*. http://www-306.ibm.com/software/integration/wid/, Letzter Zugriff: Oktober 2010

[IBMf] IBM: *WebSphere Process Server*. http://www.ibm.com/software/integration/wps/, Letzter Zugriff: Oktober 2010

[IC93] Ioannidis, Yannis E.; Christodoulakis, Stavros: Optimal Histograms for Limiting Worst-Case Error Propagation in the Size of Join Results. In: *ACM Transactions on Database Systems* 18 (1993), Nr. 4, S. 709–748

[Ioa96] Ioannidis, Yannis E.: Query Optimization. In: *ACM Computing Surveys* 28 (1996), Nr. 1, S. 121–123

[Ioa03] Ioannidis, Yannis: The History of Histograms (abridged). In: *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB 2003)*. Berlin, Germany, September 2003, S. 19–30

[IP95a] Ioannidis, Yannis E.; Poosala, Viswanath: Balancing Histogram Optimality and Practicality for Query Result Size Estimation. In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD 1995)*. New York, NY, USA, 1995, S. 233–244

[IP95b] Ioannidis, Yannis E.; Poosala, Viswanath: Histogram-Based Solutions to Diverse Database Estimation Problems. In: *IEEE Data Engineering Bulleting* 18 (1995), Nr. 3, S. 10–18

[KC04] Kimball, Ralph; Caserta, Joe: *The Data Warehouse ETL Toolkit*. John Wiley & Sons, 2004

[KKL+05a] Kloppmann, M.; Koenig, D.; Leymann, F.; Pfau, G.; Rickayzen, A.; Riegen, C. von; Schmidt, P.; Trickovic, I.: WS-BPEL Extension for People - BPEL4People. In: *Joint white paper, IBM and SAP* (2005)

[KKL+05b] Kloppmann, M.; Koenig, D.; Leymann, F.; Pfau, G.; Rickayzen, A.; Riegen, C. von; Schmidt, P.; Trickovic, I.: WS-BPEL Extension for Sub-processes - BPEL-SPE. In: *Joint white paper, IBM and SAP* (2005)

[KM07] Kraft, Tobias; Mitschang, Bernhard: Statistics API: DBMS-Independent Access and Management of DBMS Statistics in Heterogeneous Environments. In: *Proceedings of the 9th International Conference on Enterprise Information Systems (ICEIS 2007).* Madeira, Portugal, June 2007, S. 5–12

[KNS92] Keller; Nüttgens; Scheer: Semantische Prozessmodellierung auf der Grundlage Ereignisgesteuerter Prozessketten (EPK). In: *Veröffentlichungen des Instituts für Wirtschaftsinformatik (IWi)* (1992), Januar, Nr. 89

[Kra07] Kraft, Tobias: A Cost-Estimation Component for Statement Sequences. In: *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB 2007).* Vienna, Austria, September 2007, S. 1382–1385

[Kra09] Kraft, Tobias: *Optimization of Query Sequences*, Universität Stuttgart, Dissertation, 2009

[KS04] Kraft, Tobias; Schwarz, Holger: CHICAGO: A Test and Evaluation Environment for Coarse-Grained Optimization. In: *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB 2004).* Toronto, Canada, August 2004, S. 1345–1348

[KSM07] Kraft, Tobias; Schwarz, Holger; Mitschang, Bernhard: A Statistics Propagation Approach to Enable Cost-Based Optimization of Statement Sequences. In: *Proceedings of the 11th East European Conference on Advances in Databases and Information Systems (ADBIS 2007).* Varna, Bulgaria, September 2007, S. 267–282

[KSRM03] Kraft, Tobias; Schwarz, Holger; Rantzau, Ralf; Mitschang, Bernhard: Coarse-Grained Optimization: Techniques for Rewriting SQL Statement Sequences. In: *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB 2003).* Berlin, Germany, September 2003, S. 488–499

[LAB+06] Ludäscher, Bertram; Altintas, Ilkay; Berkley, Chad; Higgins, Dan; Jaeger, Efrat; Jones, Matthew; Lee, Edward A.; Tao, Jing; Zhao, Yang: Scientific Workflow Management and the Kepler System: Research Articles. In: *Concurrency and Computation: Practice and Experience* 18 (2006), Nr. 10, S. 1039–1065

[Ley97] Leymann, Frank: Transaktionsunterstützung für Workflows. In: *Informatik, Forschung Entwicklung* 12 (1997), Nr. 2, S. 82–90

[LMS95] Levy, Alon Y.; Mendelzon, Alberto O.; Sagiv, Yehoshua: Answering Queries Using Views. In: *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1995).* San Jose, California, USA, May 1995, S. 95–104

[LR00] Leymann, Frank; Roller, Dieter: *Production Workflow: Concepts and Techniques.* Upper Saddle River, NJ, USA, 2000

[LSPC00] Lehner, Wolfgang; Sidle, Richard; Pirahesh, Hamid; Cochrane, Roberta: Maintenance of Cube Automatic Summary Tables. In: *Proceedings of*

the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD 2000). Dallas, Texas, USA, May 2000, S. 512–513

[LY85] Larson, Per-Åke; Yang, H. Z.: Computing Queries from Derived Relations. In: *Proceedings of the 11th International Conference on Very Large Data Bases (VLDB 1985)*. Stockholm, Sweden, August 1985, S. 259–269

[Mica] Microsoft: *Windows Workflow Foundation*. `http://msdn.microsoft.com/windowsvista/building/workflow/`, Letzter Zugriff: Oktober 2010

[Micb] Microsoft: *XML Web Services for Microsoft SQL Server*. `http://msdn.microsoft.com/en-us/library/ms345123%28SQL.90%29.aspx`, Letzter Zugriff: Oktober 2010

[Mil93] Milner, R.: The Polyadic Pi-Calculus: A Tutorial. In: Bauer, F. L. (Hrsg.); Brauer, W. (Hrsg.); Schwichtenberg, H. (Hrsg.): *Logic and Algebra of Specification*. Springer-Verlag, 1993, S. 203–246

[Mit95] Mitschang, Bernhard: *Anfrageverarbeitung in Datenbanksystemen - Entwurfs- und Implementierungskonzepte*. vieweg, 1995

[Mün06] Münsch, Friedrich P.: *Anwendungsszenarien für BPEL-Prozesse*, Universität Stuttgart, Diplomarbeit, 2006

[MP94] Mumick, Inderpal S.; Pirahesh, Hamid: Implementation of Magic-Sets in a Relational Database System. In: *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD 1994)*. Minneapolis, Minnesota, USA, May 1994, S. 103–114

[MRSR01] Mistry, Hoshi; Roy, Prasan; Sudarshan, S.; Ramamritham, Krithi: Materialized View Selection and Maintenance Using Multi-Query Optimization. In: *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD 2001)*. Santa Barbara, California, USA, May 2001, S. 307–318

[MSHR02] Madden, Samuel; Shah, Mehul A.; Hellerstein, Joseph M.; Raman, Vijayshankar: Continuously Adaptive Continuous Queries over Streams. In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD 2002)*. Madison, Wisconsin, USA, June 2002, S. 49–60

[OAF+04] Oinn, Tom; Addis, Matthew; Ferris, Justin; Marvin, Darren; Senger, Martin; Greenwood, Mark; Carver, Tim; Glover, Kevin; Pocock, Matthew R.; Wipat, Anil; Li, Peter: Taverna: A Tool for the Composition and Enactment of Bioinformatics Workflows. In: *Bioinformatics* 20 (2004), Nr. 17, S. 3045–3054

[OAS07] OASIS: *Web Services Business Process Execution Language (WSBPEL), Version 2.0*. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`. Version: 2007

[OMGa] OMG: *Business Process Modeling Notation (BPMN).* `http://www.omg.org/spec/BPMN/`, Letzter Zugriff: Oktober 2010

[OMGb] OMG: *Meta-Object Facility (MOF).* `http://www.omg.org/mof/`, Letzter Zugriff: Oktober 2010

[OMGc] OMG: *Model Driven Architecture.* `http://www.omg.org/mda/`, Letzter Zugriff: Oktober 2010

[OMGd] OMG: *Unified Modeling Language (UML).* `http://www.uml.org/`, Letzter Zugriff: Oktober 2010

[OMG05] OMG: *Meta Object Facility (MOF) 2.0 Query/View/Transformation Final Adopted Specification.* November 2005

[Ora] Oracle: *Oracle BPEL Process Manager.* `http://www.oracle.com/technology/products/ias/bpel/`, Letzter Zugriff: Oktober 2010

[Ora06] Oracle: *Using Oracle Database as a Web Service Consumer: In Oracle Database Java Developer's Guide 10g Release 2 (10.2).* August 2006

[OS96] Oberweis, Andreas; Sander, Peter: Information System Behavior Specification by High-Level Petri Nets. In: *ACM Transactions on Information Systems* 14 (1996), Nr. 4, S. 380–420

[PHH92] Pirahesh, Hamid; Hellerstein, Joseph M.; Hasan, Waqar: Extensible Rule Based Query Rewrite Optimization in Starburst. In: *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data (SIGMOD 1992).* San Diego, California, USA, June 1992, S. 39–48

[PI97] Poosala, Viswanath; Ioannidis, Yannis E.: Selectivity Estimation Without the Attribute Value Independence Assumption. In: *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB 1997).* Athens, Greece, August 1997, S. 486–495

[PIHS96] Poosala, Viswanath; Ioannidis, Yannis E.; Haas, Peter J.; Shekita, Eugene J.: Improved Histograms for Selectivity Estimation of Range Predicates. In: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD 1996).* Montreal, Quebec, Canada, June 1996, S. 294–305

[PLH97] Pirahesh, Hamid; Leung, T. Y. C.; Hasan, Waqar: A Rule Engine for Query Transformation in Starburst and IBM DB2 C/S DBMS. In: *ICDE '97: Proceedings of the Thirteenth International Conference on Data Engineering.* Washington, DC, USA, 1997, S. 391–400

[PS88] Park, Jooseok; Segev, Arie: Using Common Subexpressions to Optimize Multiple Queries. In: *Proceedings of the 4th International Conference on Data Engineering (ICDE 1988).* Los Angeles, California, USA, February 1988, S. 311–319

[Rah94] Rahm, Erhard: *Mehrrechner-Datenbanksysteme - Grundlagen der Verteilten und Parallelen Datenbankverarbeitung.* Addison-Wesley, 1994

[RC88] Rosenthal, Arnon; Chakravarthy, Upen S.: Anatomy of a Modular Multiple Query Optimizer. In: *Proceedings of the 14th International Conference on Very Large Data Bases (VLDB 1988)*. Los Angeles, California, USA, August 1988, S. 230–239

[Rei92] Reisig, Wolfgang; Schnupp, Peter (Hrsg.); Muchnick, S. S. (Hrsg.): *Primer in Petri Net Design*. Secaucus, NJ, USA, 1992

[Rei03] Reijers, Hajo A.: *Lecture Notes in Computer Science*. Bd. 2617: *Design and Control of Workflow Processes: Business Process Management for the Service Industry*. Springer, 2003

[RGL90] Rosenthal, Arnon; Galindo-Legaria, César A.: Query Graphs, Implementing Trees, and Freely-Reorderable Outerjoins. In: *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD 1990)*. Atlantic City, NJ, USA, May 1990, S. 291–299

[RH86] Rosenthal, Arnon; Helman, Paul: Understanding and Extending Transformation-Based Optimizers. In: *IEEE Database Engineering Bulletin* 9 (1986), Nr. 4, S. 44–51

[RHAM06] Russell, Nick; Hofstede, Arthur H.; Aalst, Wil M. P. d.; Mulyar, Natalya: Workflow Control-Flow Patterns: A Revised View. In: *BPM Center Report BPM-06-22* (2006)

[RHI80] Rosenkrantz, Daniel J.; Hunt III, Harry B.: Processing Conjunctive Predicates and Queries. In: *Proceedings of the 6th International Conference on Very Large Data Bases (VLDB 1980)*. Montreal, Quebec, Canada, October 1980, S. 64–72

[RKO+03] Rowe, Anthony; Kalaitzopolous, Dimitrios; Osmond, Michelle; Ghanem, Moustafa; Guo, Yike: The Discovery Net System for High Throughput Bioinformatics. In: *Bioinformatics* 19 (2003), S. 225–231

[RSSB00] Roy, Prasan; Seshadri, S.; Sudarshan, S.; Bhobe, Siddhesh: Efficient and Extensible Algorithms for Multi Query Optimization. In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD 2000)*. Dallas, Texas, USA, May 2000, S. 249–260

[SAC+79] Selinger, P. G.; Astrahan, M. M.; Chamberlin, D. D.; Lorie, R. A.; Price, T. G.: Access Path Selection in a Relational Database Management System. In: *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data (SIGMOD 1979)*. Boston, Massachusetts, USA, May 1979, S. 23–34

[Sch10] Schwarz, Holger: *Anfragegenerierende Systeme: Anwendungsanalyse, Implementierungs- und Optimierungskonzepte*. Vieweg+Teubner, 2010

[Sel86] Sellis, Timos K.: Global Query Optimization. In: *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data (SIGMOD 1986)* Bd. 15. Washington, D.C., USA, May 1986, S. 191–205

[Sel88] Sellis, Timos K.: Multiple-Query Optimization. In: *ACM Transactions on Database Systems (TODS)* 13 (1988), Nr. 1, S. 23–52

[SH05] Stonebraker, Michael; Hellerstein, Joseph M.: *Readings in Database Systems, 4th Edition*. Morgan Kaufmann, 2005

[SMWM06] Srivastava, Utkarsh; Munagala, Kamesh; Widom, Jennifer; Motwani, Rajeev: Query Optimization over Web Services. In: *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB 2006)*. Seoul, Korea, September 2006, S. 355–366

[SS91] Sellis, Timos K.; Shapiro, Leonard: Query Optimization for Nontraditional Database Applications. In: *IEEE Transactions on Software Engineering* 17 (1991), Nr. 1, S. 77–86

[SS07] Sellis, Timos K.; Simitsis, Alkis: ETL Workflows: From Formal Specification to Optimization. In: *Proceedings of 11th East European Conference on Advances in Databases and Information Systems (ADBIS 2007)*. Varna, Bulgaria, October 2007, S. 1–11

[SVS05a] Simitsis, Alkis; Vassiliadis, Panos; Sellis, Timos: Optimizing ETL Processes in Data Warehouses. In: *Proceedings of the 21st International Conference on Data Engineering (ICDE 2005)*. Tokyo, Japan, 2005, S. 564–575

[SVS05b] Simitsis, Alkis; Vassiliadis, Panos; Sellis, Timos: State-Space Optimization of ETL Workflows. In: *IEEE Transactions on Knowledge and Data Engineering* 17 (2005), Nr. 10, S. 1404–1419

[TDG06] Taylor, Ian J.; Deelman, Ewa; Gannon, Dennis B.: *Workflows for e-Science: Scientific Workflows for Grids*. Springer, 2006

[TGNO92] Terry, Douglas; Goldberg, David; Nichols, David; Oki, Brian: Continuous Queries over Append-only Databases. In: *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data (SIGMOD 1992)*. New York, NY, USA, 1992, S. 321–330

[VSES08] Vrhovnik, Marko; Suhre, Oliver; Ewen, Stephan; Schwarz, Holger: PGM/F: A Framework for the Optimization of Data Processing in Business Processes. In: *Proceedings of the 24th International Conference on Data Engineering (ICDE 2008)*. Cancún, México, April 2008, S. 1584–1587

[VSRM08] Vrhovnik, Marko; Schwarz, Holger; Radeschütz, Sylvia; Mitschang, Bernhard: An Overview of SQL Support in Workflow Products. In: *Proceedings of the 24th International Conference on Data Engineering (ICDE 2008)*. Cancún, México, April 2008, S. 1287–1296

[VSS+07] Vrhovnik, Marko; Schwarz, Holger; Suhre, Oliver; Mitschang, Bernhard; Markl, Volker; Maier, Albert; Kraft, Tobias: An Approach to Optimize Data Processing in Business Processes. In: *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB 2007)*. Vienna, Austria, September 2007, S. 615–626

[W3Ca] W3C: *SOAP Version 1.2*. `http://www.w3.org/TR/soap12-part1/`, Letzter Zugriff: Oktober 2010

[W3Cb] W3C: *XML Path Language (XPath), Version 1.0*. `http://www.w3.org/TR/xpath/`, Letzter Zugriff: Oktober 2010

[W3Cc] W3C: *XQuery 1.0: An XML Query Language*. `http://www.w3.org/TR/xquery/`, Letzter Zugriff: Oktober 2010

[W3Cd] W3C: *XSL Transformations (XSLT), Version 1.0*. `http://www.w3.org/TR/xslt/`, Letzter Zugriff: Oktober 2010

[WE77] Wong, Kai C.; Edelberg, Murray: Interval Hierarchies and their Application to Predicate Files. In: *ACM Transactions on Database Systems* 2 (1977), Nr. 3, S. 223–232

[YL95] Yan, Weipeng P.; Larson, Per-Åke: Eager Aggregation and Lazy Aggregation. In: *Proceedings of 21th International Conference on Very Large Data Bases (VLDB 1995)*. Zurich, Switzerland, September 1995, S. 345–357

[ZCL+00] Zaharioudakis, Markos; Cochrane, Roberta; Lapis, George; Pirahesh, Hamid; Urata, Monica: Answering Complex SQL Queries Using Automatic Summary Tables. In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD 2000)*. Dallas, Texas, USA, May 2000, S. 105–116