Optimization of Query Sequences

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
Tobias Kraft
aus Stuttgart

Hauptberichter:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Mitberichter:	Prof. Dr.-Ing. Dr. h.c. Theo Härder

Tag der mündlichen Prüfung: 03.07.2009

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

2009

This thesis is dedicated to
my parents, Helga and Hans Kraft,
with deep gratitude for their
constant support and encouragement.

Diese Dissertation widme ich
meinen Eltern, Helga und Hans Kraft,
in tiefer Dankbarkeit für ihre
stetige Ermutigung und Unterstützung.

Acknowledgments

First of all, I want to thank my doctoral advisor, Prof. Bernhard Mitschang, for giving me the opportunity to work on this challenging topic in his research group. I would also like to thank him for his guidance, his support, and many interesting discussions over all the years. Through his guidance I learned a lot about conducting scientific research and his ideas gave me new insights into my work.

Furthermore, my thanks go to the co-reviewer of my thesis, Prof. Theo Härder, for spending time reading this document and giving valuable suggestions and comments.

I also want to thank all current and some of the former colleagues in the Department *Applications of Parallel and Distributed Systems* for many interesting discussions and a lot of fun. In alphabetical order these are: Andreas Brodt, Nazario Cipriani, Carmen Constantinescu, Clemens Dorda, Yevgen Dorozhko, Matthias Großmann, Uwe Heinkel, Nicola Hönle, Mihály Jakob, Fabian Kaiser, Hyon Hee Kim, Jing Lu, Carlos Lübbe, Christoph Mangold, Marcello Mariucci, Jorge Minguez, Alexander Moosbrugger, Daniela Nicklas, Prof. Peter Peinl, Sylvia Radeschütz, Ralf Rantzau, Peter Reimann, Viera Rewucki, Annemarie Roesler, Jochen Rütschlin, Rodrigo Salvador-Monteiro, Oliver Schiller, Holger Schwarz, Lucineia Heloisa Thom, Marko Vrhovnik, Frank Wagner, Ralf Wagner, Matthias Wieland, and Mirka Zimmermann. Special thanks go to system administrator Ralf Aumüller for his technical support over the years and to Manfred Rasch from the administration department. Further thanks go to all members and students at the faculty who somehow supported me or contributed to this work.

I am grateful to IBM University Relations for selecting me as a recipient of the IBM Ph.D. Fellowship Award in 2004. This is a great honor for me.

I also want to thank the German Research Foundation (DFG) and the State of Baden-Württemberg for funding me. The results of the DFG project *CEOPS* are an important part of this thesis.

Last but not least, I want to thank all friends that were on my side over the years and I want to express my sincere thanks to my parents for their continuous support, encouragement and patience during my studies and during the work on this thesis.

Tobias Kraft
Ludwigsburg, 03.07.2009

Abstract*

Query optimization is a well-known topic in database research since the 1970s. This thesis highlights a special area of query optimization that arises from new trends in the usage of databases. Whereas in the beginning databases were primarily used for transaction-oriented processing of operative data, today databases are also used to facilitate reporting and analysis on consolidated, historic data. For the latter, the data is loaded into a large data warehouse and afterwards it is being analyzed by the use of tools. The tools used to model the flows that extract the operative data from the source systems, transform these data and load it into the data warehouse as well as the tools that process the data stored in the data warehouse often generate sequences of SQL statements that break down a complex flow or request into a sequence of computational steps. The optimization of this kind of sequences with respect to runtime is the focus of this thesis.

In this document, we propose a heuristic as well as a cost-based approach for this optimization problem. The cost-based approach is just an enhancement of the heuristic approach. It results from adding a cost estimation component to the optimizer architecture of the heuristic approach and by replacing the heuristic control strategy by a control strategy that considers cost estimates. Both approaches are rule-based approaches that rewrite a given sequence of SQL statements into a syntactically different but semantically equivalent sequence of SQL statements. Therefore, we specify a set of rewrite rules. For cost estimation, we employ the capabilities of the query optimizer of the underlying database management system (DBMS) which is responsible for the execution of the query sequences. To improve the quality of these cost estimates, we support the query optimizer of the underlying DBMS with statistics that we derive from histogram propagation. For this purpose, we need an interface that allows to access and manipulate statistics in the underlying DBMS. Since there exists no standardized interface for this purpose, we define our own DBMS-independent interface. For the heuristic approach as well as for the cost-based approach, we provide prototypic implementations in JAVA. Furthermore, we have implemented the DBMS-independent interface for the three commercial DBMSs IBM DB2, Oracle, and Microsoft SQL Server. We report on the results of experiments that we conducted with our prototypes and some sample sequences that we derived by using a commercial tool for online analytical processing (OLAP). They show the effectiveness of our optimization approach and they highlight the optimization potential that lies in rewriting sequences of SQL statements. Finally, we draw a conclusion and suggest some interesting points for future research.

* This document is written in American English. Thus, appropriate rules regarding spelling, wording and grammar are applied. Furthermore, we apply the American number format using point as decimal separator and comma as thousands separator.

Zusammenfassung

Die Anfrageoptimierung in relationalen Datenbanksystemen ist ein Forschungsgebiet in der Informatik, dessen Anfänge bis in die 70er Jahre zurückreichen. Diese Arbeit fokussiert auf ein spezielles Gebiet innerhalb der Anfrageoptimierung, das sich aus neuen Trends der Datenbanknutzung ergibt. Während Datenbanksysteme in den Anfangsjahren in erster Linie für die transaktionale Verarbeitung operationaler Daten eingesetzt wurden, werden sie heute auch immer mehr im Rahmen von Business-Intelligence-Anwendungen zur Speicherung riesiger historischer Datenbestände genutzt, welche die Basis für umfangreiche Analysen und Berichte bilden. Dazu werden die Daten aus den operativen Systemen extrahiert, transformiert und in ein Data Warehouse geladen. Die Daten werden während der Transformation konsolidiert, bereinigt und teilweise auch aggregiert.

Die Werkzeuge, die zur Modellierung der Prozesse, welche das Extrahieren, Transformieren und Laden bewerkstelligen, eingesetzt werden wie auch die Werkzeuge, welche anschließend die Daten des Data Warehouse verarbeiten, um daraus Modelle und Berichte zu erstellen, generieren oft Sequenzen von SQL-Anweisungen. Die Sequenzen von SQL-Anweisungen, die im Folgenden als Anfragesequenzen bezeichnet werden, repräsentieren die Realisierung eines solchen Prozesses oder einer komplexen Fragestellung in Form mehrerer einfacher Berechnungsschritte. Die Optimierung dieser Anfragesequenzen bezüglich ihrer Ausführungszeit ist der Schwerpunkt dieser Arbeit. Hierfür wird ein heuristischer Ansatz wie auch ein kostenbasierter Ansatz vorgestellt. Der kostenbasierte Optimierer stellt dabei eine Erweiterung des heuristischen Optimierers um eine Kostenschätzkomponente für Anfragesequenzen dar. Des Weiteren ersetzt der kostenbasierte Optimierer die heuristische Kontrollstrategie des heuristischen Optimierers durch eine kostenbasierte. Für den heuristischen Optimierer sowie für die Kostenschätzkomponente des kostenbasierten Optimierers wurde ein Prototyp in der Programmiersprache Java erstellt. Anschließend wurden mit diesen Prototypen und einer Auswahl an Anfragesequenzen Experimente durchgeführt, welche die Effektivität und das Potential dieser Optimierungsansätze belegen.

Diese Zusammenfassung gliedert sich nun wie folgt: Im nächsten Abschnitt wird gezeigt, wie sich die hier vorgestellte Arbeit in das große Themenfeld der Anfrageoptimierung einordnen lässt. Zudem werden einige verwandte Arbeiten angesprochen und gegenüber dieser Arbeit abgegrenzt. Anschließend werden in getrennten Abschnitten die beiden Optimierungsansätze vorgestellt. Die Zusammenfassung schließt mit einem kurzen Fazit und einem Ausblick auf mögliche Folgearbeiten.

Einordnung der Arbeit und verwandte Arbeiten

Die verschiedenen Ansätze in der Anfrageoptimierung lassen sich anhand der Problemstellung grob in vier Klassen unterschiedlicher Komplexität einordnen:

- Optimierung von Einzelanfragen
- Optimierung von Mengen von Einzelanfragen
- Optimierung von Sequenzen von SQL-Anweisungen
- Optimierung von Programmen und Workflows mit eingebetteten SQL-Anweisungen.

Jede Klasse lässt sich noch weiter in heuristische und kostenbasierte Ansätze unterteilen. Die Komplexität der Optimierungsprobleme sowie der zu deren Lösung notwendigen Algorithmen nimmt innerhalb dieser Auflistung von oben nach unten stetig zu. Außerdem erbt jede Klasse sowohl die Probleme wie auch die zugehörigen Optimierungstechniken der vorangehenden Klassen. Das heißt, jede Klasse bringt neue, spezifische Problemstellungen mit sich, eröffnet aber auch neue Möglichkeiten der Optimierung und birgt ein gewisses zusätzliches Optimierungspotential in sich.

Die niedrigste Komplexität liegt in der Optimierung von Einzelanfragen. Hierbei ist das Ziel, möglichst den optimalen Ausführungsplan für eine Anfrage zu finden, d.h., das Ziel besteht darin, in der Menge alternativer Ausführungspläne zu einer Anfrage genau den Ausführungsplan zu finden, der die geringste Antwortzeit besitzt. Die ersten Publikationen zu diesem Thema reichen bis in die 70er Jahre zurück. Die heuristischen Ansätze basieren dabei auf Verfahren, die immer zu einer Verbesserung führen oder die zumindest in den meisten Fällen bzw. unter normalen Bedingungen zu einer Verbesserung führen. Die kostenbasierten Ansätze hingegen basieren auf einem Kostenmodel. Mit Hilfe dieses Kostenmodells können Kosten für die alternativen Ausführungspläne berechnet werden, wodurch ein Vergleich der Ausführungspläne möglich wird. Während in den Anfangsjahren Verfahren zur Ermittlung einer möglichst optimalen Join-Reihenfolge sowie Indexstrukturen im Vordergrund standen, befasst sich die heutige Forschung auf diesem Gebiet mit Verfahren zur Realisierung von selbstoptimierenden, autonomen Datenbanksystemen, die sich dynamisch an die aktuelle Arbeitslast anpassen bzw. dynamisch auf Änderungen in der Arbeitslast reagieren.

Die nächst höhere Komplexitätsstufe wird erreicht, wenn es darum geht, nicht jede Anfrage isoliert zu betrachten, sondern eine Menge bestehend aus voneinander unabhängigen Anfragen als Gesamtes zu optimieren. Dabei ist das Ziel, einen möglichst optimalen globalen Ausführungsplan für die gesamte Anfragemenge zu finden, sodass die Gesamtausführungszeit der Anfragemenge minimal wird. Um dies zu erreichen, wird nach Überlappungen in den Anfragen bzw. in den Ausführungsplänen der Anfragen gesucht, da diese Teile dann nur einmal ausgeführt werden müssen und das Ergebnis materialisiert und von mehreren Anfragen weiterverarbeitet werden kann. Dies kann jedoch zur Folge haben, dass, obwohl die Gesamtausführungszeit der Anfragemenge sinkt, trotzdem einzelne Anfragen langsamer laufen.

Noch komplexer wird es, wenn man Sequenzen von SQL-Anweisungen betrachtet – also nicht Mengen von Anfragen, sondern eine Folge beliebiger SQL-Anweisungen. Zwischen diesen SQL-Anweisungen können Abhängigkeiten bezüglich der Ausführungsreihenfolge bestehen, d.h. beispielsweise, dass eine SQL-Anweisung auf eine Tabelle zugreift, welche von einer anderen SQL-Anweisung innerhalb derselben Sequenz erzeugt oder befüllt wurde. In diesem Bereich sind nur einzelne Arbeiten bekannt. Es gibt jedoch einige aktuelle Arbeiten, welche die Arbeitslast eines Datenbanksystems analysieren und diese dabei nicht wie früher als eine Menge von SQL-Anweisungen sehen, sondern als Sequenz von SQL-Anweisungen.

Die höchste Komplexität liegt in der Optimierung von Programmen oder Workflows mit eingebetteten SQL-Anweisungen. Programme und Workflows weisen nämlich aufgrund der Kontrollstrukturen, welche Programmier- und Workflow-Sprachen bieten, (wie Sprünge, bedingte Verzweigungen und Schleifen) teilweise recht komplexe Daten- und Kontrollflüsse auf. Somit ist eine einmalige, sequentielle Abarbeitung der SQL-Anweisungen wie etwa bei Anweisungssequenzen nicht mehr garantiert. Momentan werden SQL-Anweisungen und Programmstruktur noch vollständig getrennt und unabhängig voneinander optimiert. Es gibt zwar erste Ansätze, die versuchen, Anfrageoptimierung mit Programmiersprachenoptimierung zu kombinieren; diese sind jedoch bisher rein heuristischer Natur, da eine Kostenbewertung insbesondere von Workflow-Aktivitäten sehr schwierig ist.

Wie bereits erwähnt, befasst sich die Arbeit, welche hier vorgestellt wird, mit der Optimierung von Anfragesequenzen. Anfragesequenzen sind dabei als eine Untermenge der Sequenzen von SQL-Anweisungen zu sehen. Sie bestehen aus CREATE-TABLE-, INSERT- und DROP-TABLE-Anweisungen und folgen einem gewissen Aufbau. Anfragesequenzen unterteilen eine komplexe Anfrage in kleinere, weniger komplexe Schritte bzw. realisieren einen Prozess aus aufeinander folgenden Berechnungsschritten. Die CREATE-TABLE- und DROP-TABLE-Anweisungen dienen dazu, temporär Tabellen zu erzeugen, die zur Speicherung von Zwischenergebnissen innerhalb der Sequenz verwendet werden und nur während der Laufzeit der Sequenz existieren. Zusätzlich wird durch eine CREATE-TABLE-Anweisung eine Tabelle angelegt, die das Endergebnis der Anfragesequenz speichert und daher nicht innerhalb der Anfragesequenz gelöscht wird, sondern auch noch nach Ausführung der Anfragesequenz existiert. Die INSERT-Anweisungen realisieren die einzelnen Berechnungsschritte, welche Ihre Ergebnisse in den von der Sequenz erzeugten Tabellen ablegen. Diese Ergebnisse können sämtlichen nachfolgenden INSERT-Anweisungen innerhalb der Sequenz als Eingabe dienen. Somit besteht zwischen den INSERT-Anweisungen eine gewisse Abhängigkeit bezüglich der Ausführungsreihenfolge. Zwei Anfragesequenzen werden als semantisch äquivalent angesehen, wenn die Endergebnistabellen denselben Bezeichner sowie dieselbe Struktur besitzen und deren Inhalte nach Ausführung der Anfragesequenzen unter denselben Ausgangsbedingungen übereinstimmen. In den Optimierungsansätzen, die in dieser Arbeit vorgestellt werden, werden sowohl Konzepte aus der Optimierung von Einzelanfragen und Anfragemengen an die Problemstellung der Optimierung von Anfragesequenzen angepasst wie auch neue Konzepte betrachtet. Zwar gibt es einige Parallelen zu Optimierungsansätzen für ETL-Flows (ETL steht für Extraktion-Transformation-Laden), jedoch sind keine Ar-

beiten bekannt, die ETL-Flows betrachten, deren Einzeloperationen die Mächtigkeit von komplexen SQL-Anfragen haben, sowie keine Arbeiten, die die speziellen Charakteristiken von Anfragesequenzen berücksichtigen. Zu diesen Charakteristiken gehört zum einen die Selbstverwaltung der temporären Tabellen, d.h., temporäre Tabellen, welche innerhalb der Anfragesequenz zur Speicherung von Zwischenergebnissen verwendet werden, werden durch SQL-Anweisungen, die Teil der Anfragesequenz sind, erzeugt wie auch wieder entfernt. Zum anderen gehört die partielle Reihenfolgeabhängigkeit unter den INSERT-Anweisungen zu den speziellen Charakteristiken, d.h., INSERT-Anweisungen innerhalb einer Sequenz greifen auf Tabellen zu, welche von anderen INSERT-Anweisungen innerhalb derselben Sequenz befüllt werden und somit Zwischenergebnisse der Sequenz darstellen.

Heuristischer Optimierungsansatz

Der heuristische Optimierungsansatz basiert auf einer Menge von Restrukturierungsregeln. Diese Restrukturierungsregeln formen eine gegebene Anfragesequenz um, wodurch eine neue Anfragesequenz entsteht, d.h., sowohl Eingabe als auch Ausgabe ist eine Sequenz von SQL-Anweisungen. Die originale Sequenz und die restrukturierte Sequenz unterscheiden sich zwar syntaktisch in der Zahl und dem Aufbau der SQL-Anweisungen, sind jedoch semantisch äquivalent. Dies bedeutet, dass die CREATE-TABLE-Anweisung, welche die Tabelle für das Endergebnis der Anfragesequenz erzeugt, nicht verändert werden darf und dass die restrukturierte Anfragesequenz dasselbe Endergebnis produzieren muss wie die originale Anfragesequenz. Da sämtliche Zwischenergebnistabellen jedoch temporärer Natur sind, d.h., von der Anfragesequenz selbst erzeugt und wieder entfernt werden, können die Restrukturierungsregeln deren Struktur und Inhalt beliebig verändern sowie Zwischenergebnistabellen aus der Anfragesequenz entfernen oder neue Zwischenergebnistabellen hinzufügen, solange dadurch das Endergebnis der Anfragesequenz nicht verändert wird. Die Restrukturierungsregeln verändern somit nur die Art und Weise, wie aus den Inhalten der Basistabellen das Endergebnis berechnet wird.

Die Restrukturierungsregeln arbeiten auf einer internen Repräsentation der Anfragesequenz. Daher ist ein Parser notwendig, der die Sequenz von SQL-Anweisungen einliest und in die interne Repräsentation übersetzt, sowie ein Rückübersetzer, der die interne Repräsentation nach Abschluss des Optimierungsprozesses wieder in eine Sequenz aus SQL-Anweisungen übersetzt. Die interne Repräsentation stellt die Anfragesequenz als einen Abhängigkeitsgraphen dar. Die Knoten dieses Abhängigkeitsgraphen spiegeln eine Kombination aus Anweisungen für eine bestimmte Ergebnistabelle der Anfragesequenz wider. Für eine Zwischenergebnistabelle besteht diese Kombination aus einer CREATE-TABLE-, einer INSERT- und einer DROP-TABLE-Anweisung, für eine Endergebnistabelle besteht diese Kombination aus einer CREATE-TABLE- und einer INSERT-Anweisung. Die Anweisungen werden in der internen Struktur in Form spezieller Parse-Bäume gehalten. Die Knoten des Abhängigkeitsgraphen werden im Folgenden auch als Anfrage (engl. Query) bezeichnet – daher auch die Bezeichnung *Anfragesequenz* (bzw. *Query Sequence*) für diese spezielle Form einer Sequenz von SQL-Anweisungen. Die

gerichteten Kanten repräsentieren die Reihenfolgeabhängigkeiten zwischen den INSERT-Anweisungen der Anfragen, d.h., eine gerichtete Kante zwischen zwei Anfragen stellt die direkte Produzent-Konsument-Beziehung dar, welche über die Tabelle der Quellanfrage realisiert wird.

Jede Restrukturierungsregel besteht aus einer Regelbedingung und einer Regelaktion. Die Regelbedingung gibt an, unter welchen Gegebenheiten eine Regel anwendbar ist; die Regelaktion beschreibt, wie die Anfragesequenz umgeformt wird. Abhängig davon, ob die Regelbedingung spezifiziert, welche Eigenschaften eine einzelne Anfrage und eventuell die von ihr abhängigen Anfragen besitzen müssen, oder ob die Regel spezifiziert, in welchen Komponenten zwei Anfragen übereinstimmen bzw. sich unterscheiden müssen, unterscheidet man zwischen unären und binären Regeln.

Die Restrukturierungsregeln lassen sich anhand ihrer Charakteristiken grob in drei Klassen unterteilen:

Klasse 1: Zu dieser Klasse gehören die Regeln, die zwei Anfragen aufgrund gewisser Gemeinsamkeiten bzw. Überlappungen zu einer Anfrage verschmelzen. Manchmal werden zusätzlich auch die Anfragen, die von den verschmolzenen Anfragen abhängen, modifiziert.

Klasse 2: Zu dieser Klasse zählen die Regeln, die die Abhängigkeitsbeziehung zwischen den Anfragen zur Optimierung nutzen. Regeln dieser Klasse verschmelzen beispielsweise direkt voneinander abhängige Anfragen, verschieben Prädikate zwischen Anfragen entlang der Abhängigkeitsbeziehung oder entfernen die Attribute aus den Ergebnistabellen einer Anfrage, die in keiner nachfolgenden Anfrage mehr benötigt werden.

Klasse 3: Der Kontext der Regelbedingung einer Regel der Klasse 3 beschränkt sich auf die zu betrachtenden Anfrage. Diese Regeln entfernen beispielsweise redundant auftretende Attribute in der Ergebnistabelle einer Anfrage, welche u.a. durch Anwendung einer Regel der Klasse 1 entstehen können.

Der Optimierungseffekt der Restrukturierungsregeln basiert somit zum einen auf einer Reduzierung der Zahl der Anfragen im Abhängigkeitsgraph und zugleich auf einer Reduzierung der Zahl der Anweisungen innerhalb der Anfragesequenz, zum anderen auf einer Vereinfachung der Anfragen.

Im Rahmen dieser Arbeit werden acht Restrukturierungsregeln genauer spezifiziert. Zu jeder Regel liegt eine Beschreibung der Regelbedingung sowie der Regelaktion in Form von Pseudo-Code vor. Des Weiteren wird anhand der Auswirkungen der Regeln auf die betroffenen SQL-Anweisungen die Korrektheit der Regeln gezeigt. Anwendungsbeispiele zu den einzelnen Regeln zeigen deren Einsatzmöglichkeiten und dienen der Verständlichkeit.

Neben den Restrukturierungsregeln benötigt der Optimierer noch eine sogenannte Kontrollstrategie, welche bestimmt, wann welche Regel zur Anwendung kommt. Von verwandten Arbeiten aus dem Bereich der heuristischen Anfrageoptimierung sind bereits einige Strategien bekannt. Für die prototypische Implementierung wurde eine Prioritätenbasierte Kontrollstrategie gewählt. Das heißt, aus der Menge der Regeln, deren Regelbedingung für eine Anfrage bzw. Anfragekombination in der zu betrachtende Anfragesequenz

erfüllt ist, wird diejenige ausgewählt, die die höchste Priorität besitzt, und es wird deren Regelaktion auf genau einer Anfrage bzw. Anfragekombination ausgeführt. Dazu wird zuvor jeder Restrukturierungsregel statisch eine eindeutige Priorität zugewiesen. Beginnend mit der zu optimierenden, originalen Sequenz wird dieser Vorgang der Regelauswahl und -anwendung innerhalb einer Schleife solange wiederholt, bis keine Regel mehr feuern kann. Die Charakteristiken der in dieser Arbeit spezifizierten Restrukturierungsregeln garantieren, dass dieser Prozess auch tatsächlich nach einer endlichen Anzahl von Schritten terminiert und nicht in eine Endlosschleife gerät. Für den Fall, dass die Regelmenge einmal so erweitert wird, dass die Zusammensetzung der Regelmenge ein Terminieren des Optimierungsprozesses nicht mehr garantiert, existieren entsprechende Mechanismen, die das Terminieren sicherstellen.

Der in der Programmiersprache Java implementierte Prototyp wurde auf eine Auswahl von Anfragesequenzen unterschiedlicher Komplexität angewandt. Es wurde die Laufzeit der originalen Anfragesequenzen sowie die Laufzeit der Anfragesequenzen nach jeder Regelanwendung gemessen. Ein Vergleich der Laufzeiten zwischen den originalen Anfragesequenzen und den restrukturierten Anfragesequenzen zeigt, welches Potential in dem auf Restrukturierungsregeln basierenden Ansatz steckt. Durch die Restrukturierung konnten Performance-Gewinne von teilweise sogar mehr als 85% gemessen werden. Allerdings konnte bei einzelnen Regelanwendungen auch eine Verschlechterung der Laufzeiten bzw. keine signifikante Laufzeitverbesserung festgestellt werden. Anschließend wurden noch zwei Darstellungsvarianten betrachtet, die eine Anfragesequenz als Einzelanfrage und nicht als eine Sequenz einzelner SQL-Anweisungen repräsentieren. Eine Möglichkeit, eine Anfragesequenz in eine Einzelanfrage umzuformen, besteht darin, dass man beginnend mit der INSERT-Anweisung, die die Endergebnistabelle der Anfragesequenz befüllt, die Tabellenreferenzen in der FROM-Klausel der INSERT-Anweisungen rekursiv durch den Körper der zu dieser Tabellenreferenzen gehörenden INSERT-Anweisungen ersetzt. So entsteht eine tief geschachtelte Einzelanfrage, die eventuell aber auch viele redundante Unteranfragen enthält. Die Redundanz entsteht dadurch, dass auf das Zwischenergebnis einer Anfragesequenz von mehreren nachfolgenden INSERT-Anweisungen innerhalb derselben Anfragesequenz zugegriffen werden kann. Eine andere Möglichkeit ergibt sich aus der Verwendung der WITH-Klausel. Dabei wird der INSERT-Anweisung, welche die Endergebnistabelle der Anfragesequenz befüllt, eine WITH-Klausel hinzugefügt, welche jede Zwischenergebnistabelle der Anfragesequenz als temporäre Tabelle definiert. Die Variante der tief geschachtelten Einzelanfrage führte in einigen wenigen Fällen zwar zu einer Verbesserung, in anderen jedoch zu einer signifikanten Verschlechterung der Laufzeit. Die Umwandlung in eine Einzelanfrage mit WITH-Klausel führte zwar fast immer zu einer Verbesserung, diese lag jedoch in den meisten Fällen unter 50%. In allen durchgeführten Messungen wurde jedoch der größte Performance-Gewinn durch die Anwendung von Restrukturierungsregeln erzielt und dieser Gewinn war in den meisten Fällen deutlich größer als der durch die Umwandlung in Einzelanfragen erzielte Gewinn.

Kostenbasierter Optimierungsansatz

Durch Hinzufügen einer Kostenschätzkomponente für Anfragesequenzen sowie durch Austausch der heuristischen Kontrollstrategie gegen eine kostenbasierte Kontrollstrategie lässt sich der heuristische Optimierer zu einem kostenbasierten Optimierer erweitern. Durch die Kostenbewertung wird ein Vergleich von alternativen Anfragesequenzen ermöglicht. Dadurch sollen Regelanwendungen, die zu einer Laufzeitverschlechterung führen, vermieden werden. Zudem lässt sich basierend auf den Ergebnissen der Regelanwendungen ein Suchraum aus semantisch äquivalenten, jedoch syntaktisch unterschiedlichen Anfragesequenzen aufbauen und nach einer anschließenden Kostenbewertung die hinsichtlich des Kostenmodells günstigste Alternative ermitteln.

Da die Laufzeit einer SQL-Anweisung bzw. einer SQL-Anfrage vom dafür gewählten Ausführungsplan und somit vom physischen Datenbankdesign sowie den Operatorimplementierungen und Strategien im Optimierer des darunter liegenden Datenbanksystems, welches später die Anfragesequenz ausführt, abhängt, gestaltet sich der Entwurf eines Kostenmodells auf Ebene der Anfragesequenzen als äußerst schwierig. Dieses Kostenmodell müsste eigentlich die Kostenmodelle sämtlicher potentiell zur Ausführung von Anfragesequenzen nutzbarer Datenbankmanagementsysteme (DBMSe) nachbilden und die Strategien der zugehörigen Optimierer mit einbeziehen. Eigentlich sind die Kostenwerte aber im darunter liegenden Datenbanksystem verfügbar – vorausgesetzt, dass es sich um ein Datenbanksystem mit kostenbasiertem Anfrageoptimierer handelt. Daher greift der in dieser Arbeit verfolgte Ansatz auf die Kostenschätzung des Optimierers im darunter liegenden Datenbanksystem zurück. Die Idee besteht darin, von diesem die Kosten für die einzelnen Anweisungen einer Anfragesequenz zu erfragen, ohne jedoch die Anweisungen der Anfragesequenz auszuführen. Die Summe dieser Kostenwerte entspricht dann den Gesamtkosten der Anfragesequenz.

Dies lässt sich jedoch nicht so einfach umsetzen, sondern bringt einige Probleme mit sich. Erstens ist eine Abschätzung der Kosten für die INSERT-Anweisungen innerhalb einer Anfragesequenz nicht möglich, da die Tabellen, in die eingefügt werden soll, noch nicht existieren, sondern erst während der Ausführung der Anfragesequenz von einer in der Anfragesequenz enthaltenen CREATE-TABLE-Anweisung erzeugt werden; ähnliches gilt für DROP-TABLE-Anweisungen. Zweitens werden zur Berechnung der Kostenschätzwerte Statistiken herangezogen, welche die Verteilung der Daten innerhalb einer Tabellenspalte beschreiben, welche aber für die Tabellen, die innerhalb der Anfragesequenz erzeugt werden, ebenfalls nicht existieren. Daher basiert die Berechnung der Kostenschätzwerte für INSERT-Anweisungen, die auf zuvor berechnete Zwischenergebnissen derselben Sequenz zugreifen, auf vordefinierten Standardwerten und Annahmen bezüglich der Datenverteilung innerhalb der Tabellenspalten bzw. bezüglich der Selektivität von Prädikaten. Da diese Standardwerte und Annahmen stark vom tatsächlichen Zustand der Datenbank abweichen können, sind die Kostenschätzwerte unter Umständen sehr ungenau und wenig aussagekräftig. Um ersteres Problem zu lösen, müssen zuerst alle CREATE-TABLE-Anweisungen ausgeführt werden, bevor die Kostenschätzwerte der INSERT-Anweisungen ermittelt werden. Nach Ermittlung der Kostenschätzwerte sind diese Tabellen natürlich wieder zu entfernen, sodass der ursprüngliche Zustand der Daten-

bank wiederhergestellt wird. Um das zweite Problem zu lösen, wird wie folgt vorgegangen. Nach Ausführung der CREATE-TABLE-Anweisungen werden die Statistiken zu den Basistabellen aus dem darunter liegenden Datenbanksystem extrahiert. Dann werden diese durch die INSERT-Anweisungen der Anfragesequenz propagiert. Anschließend werden die propagierten Statistiken dann wieder als Statistiken für die Zwischenergebnistabellen der Anfragesequenz im darunter liegenden Datenbanksystem hinterlegt. Während der nachfolgenden Kostenschätzung stehen diese Statistiken dann dem Optimierer des darunter liegenden Datenbanksystems zur Verfügung. Nur so lassen sich aussagekräftige Kostenschätzwerte für eine Anfragesequenz ermitteln.

Das Erfragen der Kostenschätzwerte vom darunter liegenden Datenbanksystem sowie der lesende und schreibende Zugriff auf die Statistiken, die im darunter liegenden Datenbanksystem gespeichert sind, sind jedoch nicht trivial. Für beides gibt es keine einheitliche Schnittstelle. Sämtliche Datenbanksysteme bieten über die Systemsichten oder Systemtabellen zumindest einen lesenden Zugriff auf die Statistiken. Manche DBMSe erlauben unter gewissen Einschränkungen sogar Datenänderungen direkt an den Systemsichten oder Systemtabellen, andere stellen hierfür spezielle Funktionen und Prozeduren zur Verfügung. Doch nicht jedes DBMS erlaubt Änderung an den Statistiken durch den Benutzer. Zudem unterscheiden sich die Formate, in denen die Verteilungsstatistiken in den Datenbanksystemen gespeichert werden, von DBMS zu DBMS. Auch der Zugriff über JDBC-DatabaseMetaData auf sonstige Metadaten ist recht eingeschränkt. Im Rahmen dieser Arbeit wird daher unter dem Namen *Statistics API* eine DBMS-unabhängige Schnittstelle definiert, die einen einheitlichen Zugriff auf Metadaten, Statistiken und Optimiererschätzwerte ermöglicht. Diese Schnittstelle stellt auch DBMS-unabhängige Formate zur Verfügung, in denen die Daten gespeichert werden, welche die Methoden der Schnittstelle liefern bzw. welche als Argumente an die Methoden der Schnittstelle übergeben werden. Für jedes DBMS ist eine eigene Implementierung notwendig, die die Methoden der DBMS-unabhängigen Schnittstelle auf die spezifischen Zugriffsmöglichkeiten des jeweiligen DBMSs abbildet. Die Schnittstellenimplementierungen fungieren somit als eine Art Wrapper.

Für die Festlegung, welche Statistikwerte und Metadaten die Statistics API anbieten soll, wurden die drei kommerziellen DBMSe IBM DB2, Oracle und Microsoft SQL Server untersucht. Es wurden die Statistiken ausgewählt, welche in mindestens zwei dieser Systeme vorhanden sind. Da einfache Verteilungsstatistiken bestehend aus einem Minimalwert, einem Maximalwert, einer Kardinalität und eventuell der Zahl unterschiedlicher Werte auch als Histogramme mit einem einzigen Bucket angesehen werden können, unterscheidet die Statistics API nicht zwischen einfachen Verteilungsstatistiken und Histogrammen, sondern stellt Verteilungsstatistiken immer als Histogramme zur Verfügung. Für diese Histogramme wird im Rahmen der Statistics API ein flexibles Histogrammformat definiert, welches die speziellen Charakteristiken der unterschiedlichen Datentypen berücksichtigt und die Speicherung von Histogrammen unterschiedlicher Art bzw. unterschiedlicher Herkunft erlaubt.

Um die aus dem darunter liegenden Datenbanksystem extrahierten Histogramme nun durch die SQL-Anfragen, welche den Körper der INSERT-Anweisungen bilden, zu propagieren, werden diese in einen Baum aus logischen Operationen der Relationenalgebra

übersetzt. Im Gegensatz zur normalen Anfrageverarbeitung arbeiten diese Operationen jedoch auf Histogrammen und nicht auf Relationen. Für die Implementierung der Operationen wurden Konzepte aus der Näherungsweisen Anfragebeantwortung (engl. Approximate Query Answering) erweitert und fehlende Operationen ergänzt. Beispielsweise werden in der Näherungsweisen Anfragebeantwortung bisher keine arithmetischen Ausdrücke berücksichtigt; ebenso wenig wird die Gruppierung unterstützt. Da aber sowohl arithmetische Ausdrücke wie auch Gruppierung wichtige Bestandteile der in den INSERT-Anweisungen der Anfragesequenzen enthaltenen SQL-Anfragen sind, war eine entsprechende Erweiterung der bestehenden Konzepte um diese Funktionalität notwendig. Bei der Propagation von Histogrammen durch arithmetische Operationen kommt Intervallarithmetik zum Einsatz, um die Bucket-Grenzen im Ergebnishistogramm zu bestimmen. Für die Gruppierung wurden Heuristiken erarbeitet, womit sich die Kardinalität des Gruppierungsergebnisses sowie eine Werteverteilung für die auf den Gruppen berechneten Aggregate abschätzen lassen.

In der Programmiersprache Java wurde ein Prototyp der Kostenschätzkomponente implementiert. Dieser umfasst die Implementierung des zuvor beschriebenen Algorithmus zur Kostenermittlung, die Implementierung der Operationen der Relationenalgebra zum Zwecke der Histogrammpropagation sowie eine Komponente, die aus den INSERT-Anweisungen die Bäume für die Histogrammpropagation aufbaut. Des Weiteren wurden Implementierungen der Statistics API für die drei kommerziellen DBMSe IBM DB2, Oracle und Microsoft SQL Server erstellt.

Für die Messungen wurde eine Anfragesequenz mit einem überschaubaren Suchraum an restrukturierten Anfragesequenzen herangezogen. Mehrere INSERT-Anweisungen dieser Anfragesequenz enthalten ein Filterprädikat, welches ein Summenaggregat mit einer Konstante vergleicht und alle Tupel auswählt, deren Attributwert größer als der der Konstante ist. Über diese Konstante lassen sich folglich die Selektivität und somit auch die Größe der Zwischenergebnisse wie auch des Endergebnisses innerhalb der Anfragesequenz variieren. Für drei verschiedene Konstantenwerte – also für drei verschiedene Selektivitäten – wurde jeweils der gesamte Suchraum der Anfragesequenz aufgebaut. Anschließend wurde für jede Anfragesequenz im Suchraum die Laufzeit mit dem zugehörigen, vom Prototyp der Kostenschätzkomponente ermittelten Kostenschätzwert verglichen. Dieser Vergleich ergab, dass die ermittelten Kostenschätzwerte ein guter Indikator für die Laufzeiten der Anfragesequenzen sind, d.h., die Verteilung der Laufzeiten innerhalb des Suchraums entsprach näherungsweise der Verteilung der Kostenschätzwerte innerhalb des Suchraums. Des Weiteren hat sich die Notwendigkeit und Effektivität der Histogrammpropagation gezeigt. So gab es aufgrund dessen, dass bei der Planerstellung im darunter liegenden Datenbanksystem falsche Annahmen getroffen wurden und dadurch in der Folge ein suboptimaler Ausführungsplan gewählt wurde, bei einigen Werten für den Filterfaktor Ausreißer bezüglich der Laufzeit. Unter Berücksichtigung der im Voraus durch die gesamte Anfragesequenz propagierten Statistiken konnten diese Ausreißer jedoch vermieden werden und die Anfragesequenzen wiesen eine mit dem Kostenschätzwert konforme Laufzeit auf. Das heißt, die propagierten Histogramme sollten nicht nur für die Kostenschätzung verwendet werden, sondern lassen sich auch gewinnbringend bei der Ausführung der Anfragesequenzen einsetzen.

Fazit und Ausblick

Die Implementierung eines Prototyps für die heuristische Optimierung hat die Machbarkeit des vorgestellten Ansatzes gezeigt. Die Messungen mit diesem Prototyp haben das Optimierungspotential aufgezeigt, das in der Anwendung der Restrukturierungsregeln liegt. Die Messungen mit dem kostenbasierten Optimierer haben gezeigt, dass die Nutzung der Kostenschätzkomponente des darunter liegenden Datenbanksystems in Kombination mit der Histogrammpropagation eine brauchbare Lösung darstellt. Auch wenn die vorgestellte Regelmenge bereits sehr effektiv ist, so besteht immer noch Potential für mögliche Folgearbeiten.

Das größte Potential für zukünftige Arbeiten liegt in der Erweiterung der Regelmenge sowie in der Erweiterung der Definition von Anfragesequenzen. Zum einen können die Regelbedingungen dahingehend erweitert werden, dass sie auf noch mehr Ausprägungen von Anfragesequenzen anwendbar sind und bisher nicht berücksichtigte Sonderfälle mit einschließen. Zum anderen ist die Erweiterung der Regelmenge um weitere Regeln denkbar. Dies könnten beispielsweise Regeln sein, die eine Anfrage in mehrere Anfragen zerlegen und damit die Umkehrung einer Verschmelzungsregel darstellen. Möglichkeiten für neue Restrukturierungsregeln ergeben sich aber auch, wenn die für Anfragesequenzen definierte SQL-Untermenge erweitert wird. Beispielsweise bietet die Betrachtung von genesteten SQL-Anfragen oder Mengenoperationen innerhalb der INSERT-Anweisungen Raum für weitere Restrukturierungsmöglichkeiten.

Das Hinzufügen weiterer Regeln zur Regelmenge kann natürlich zu einer beträchtlichen Vergrößerung des Suchraums der Optimierung führen und letztlich auch zu oszillierenden Regelanwendungen, wodurch eine effiziente Durchführung sowie ein Terminieren des Optimierungsprozesses nicht mehr gewährleistet ist. Dies macht also eine Überarbeitung der Kontrollstrategien bzw. die Einbeziehung von Pruning-Techniken notwendig.

Eine Alternative zur Propagation von Histogrammen mit anschließender Kostenschätzung wäre das Ausführen der Anfragesequenzen auf einem Daten-Sample. Sampling wird nämlich auch in der Näherungsweisen Anfragebeantwortung als Alternative zur Histogrammpropagation eingesetzt. Da Sampling von immer mehr DBMSen unterstützt wird, wäre ein Vergleich zwischen dem hier vorgestellten Ansatz und einem auf Sampling basierenden Ansatz durchaus interessant.

Auch eine Kombination des in dieser Arbeit vorgestellten Ansatzes mit anderen in den verwandten Arbeiten vorgestellten Ansätzen wäre zu überprüfen. Möglicherweise könnten unterschiedliche Ansätze voneinander profitieren. Beispiel hierfür wäre eine Kombination des hier vorgestellten Ansatzes mit der dynamischen Erstellung von Indexen und Statistiken auf den Zwischenergebnistabellen. Dazu müsste die Anfragesequenz-Definition jedoch um die entsprechenden Anweisungen erweitert werden.

Auch die Anwendbarkeit der erarbeiteten Regeln auf komplexere Konstrukte wie Programme oder Workflows mit eingebetteten SQL-Anweisungen wäre zu untersuchen. Allerdings würde es hier aufgrund der potentiell komplexen Daten- und Kontrollflüsse schwierig werden, die Korrektheit der Regeln zu zeigen.

Contents

Abbreviations

API	application programming interface
CGO	coarse-grained optimization
DBMS	database management system
ETL	extract-transform-load
GUI	graphical user interface
MQO	multi-query optimization
SQO	single-query optimization

1 Introduction

This chapter highlights the optimization problem that we study in this thesis, introduces the database that we use for the examples and experiments in this document, and outlines the structure of this document.

1.1 Motivation

Today, more and more application areas make use of query generators which are embedded into the application software. Some of these generators just produce a single SQL query whereas others generate complex sequences of SQL statements. The reason of generating sequences that consist of multiple SQL statements lies in the complexity of the information requests and therefore also in the complexity of the generation process.

Our focus is on statement sequences that break down a complex information request into a sequence of computational steps where multiple steps can share and access intermediate results of previous steps in the same sequence. The intermediate results are stored in tables that are being created as part of the sequence and that exist only temporarily during the execution of the sequence. The final result of a sequence is stored in a table that is created by the sequence but not dropped by the sequence, so that the application can access the final result after the execution of the sequence. We call these sequences *query sequences* later on. Such sequences typically appear in the area of data warehousing where they are used to retrieve the data for reports, to compute data cubes, or to implement extract-transform-load (ETL) flows.

Figure 1.1 shows the typical architecture of tools and applications that embed a query-sequence generator. First, the user specifies his information request or ETL flow via a graphical user interface (GUI). Afterwards, the query-sequence generator translates this information request or ETL flow into a sequence of SQL statements. Then, the underlying database system executes the statements of this sequence. If desired by the application, the GUI reads the final-result table and prepares its content to display it to the user.

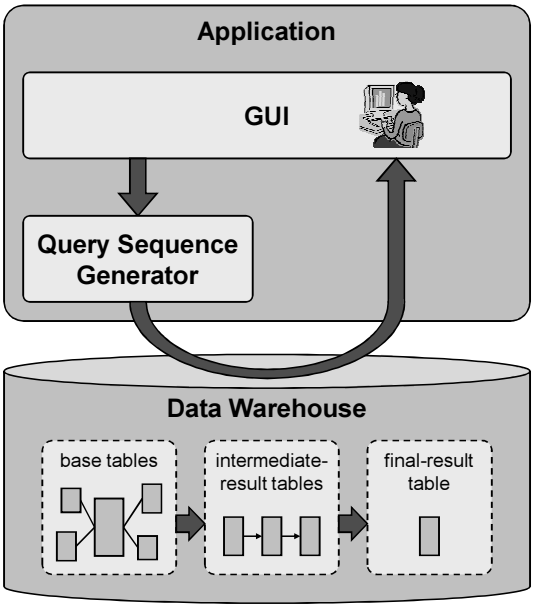


Figure 1.1: Architecture of an application that embeds a query-sequence generator.

Classic SQL query optimizers have no special support for such query sequences. They rather optimize each statement in isolation without considering the dependencies between the statements and without considering the temporary nature of the intermediate-result tables. A query sequence could also be transformed into a deeply-nested single query or into a single query that makes use of the WITH clause. However, our experiments have shown that such a transformation often does not significantly improve performance, but in some cases even leads to a performance deterioration. This is due to the complex and deeply nested structure of such a query, which results in a huge search space of very deep execution plans. These deep execution plans often cause significant cardinality estimation errors and cost estimation errors in the query optimizer of the database system that executes the sequence. Thus, the query optimizer chooses a suboptimal execution plan which leads to bad performance.

In this thesis, we propose an approach for the optimization of query sequences. The goal is to keep the query-sequence optimizer transparent for the query-sequence generator as well as for the database system. So, the optimizer serves as kind of pre-optimization step for the database's query optimizer. Our optimization approach is a rule-based approach where the rules rewrite query sequences at the SQL level, i.e., input as well as output of the optimization process is a sequence of SQL statements. The rewrite rules merge similar statements, move predicates between statements, and remove unused and redundant attributes from the tables that store the intermediate results. Furthermore, the rules guarantee that the original sequence and the rewritten sequence compute the same final result. In this document, we introduce a heuristic and a cost-based variant of the query-sequence optimizer. Both variants work with the same set of rewrite rules, but the cost-based optimizer extends the heuristic optimizer by adding a cost estimation component. The cost estimation component makes use of the cost estimates provided

by the optimizer of the underlying database system for each statement of the sequence. In order to retrieve useful cost estimates, the cost estimation component propagates histograms which are part of the base-table statistics through the INSERT statements of a query sequence and stores the resulting histograms as statistics in the underlying database system again. So the optimizer of the underlying database system can make use of these histograms during cost estimation. For both optimizer variants, we show results of experiments that we performed with prototypic implementations to demonstrate the effectiveness of our approach.

1.2 Sample Database

For the experiments in this document, we make use of the TPC-H benchmark database. The TPC-H benchmark [Tra] is a decision support benchmark that consists of a database and a suite of business-oriented ad-hoc queries and concurrent data modifications. It illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions. However, we just make use of the database because the benchmark specifies no query sequences. To obtain query sequences, we make use of the MicroStrategy DSS tools. The MicroStrategy DSS tools build a software suite that allows reporting and analysis of data stored in a relational database which is also known as relational OLAP (ROLAP). Similarly to Figure 1.1, a user specifies his report via a GUI. Afterwards, a generator creates a query sequence that computes the report data. Finally, the GUI accesses the final result of the query sequence, which is stored in the database, and displays it to the user in the specified format.

Figure 1.2 shows the schema of the TPC-H database which contains eight tables. The arrows represent one-to-many relationships between tables. They point in the direction of the one-to-many relationships. The formula below each table name represents the cardinality of the table. As TPC-H supports different database sizes, the cardinalities are factored by SF, the Scale Factor, which determines the size of the database. (The cardinality for the *LINEITEM* table is approximate.) SF proportionally scales all tables except for the tables *NATION* and *REGION*. These two tables have a constant cardinality that is independent from the size of the database. The database stores orders from customers in the *ORDERS* table. Orders consist of one or more line items which are stored in the *LINEITEM* table. Each line item refers to a part, of which a certain quantity has been ordered, and to the supplier from which this part has been ordered. As each part in the *PART* table may be offered by multiple suppliers and each supplier may offer several different parts, the *PARTSUPP* table stores valid combinations of parts and suppliers. Suppliers and customers are stored in different tables, named *SUPPLIER* and *CUSTOMER*. However, they have a common hierarchy, i.e., suppliers as well as customers are assigned to nations and nations are grouped into regions. Thus, they share the tables *NATION* and *REGION*. Each attribute name has a prefix consisting of the first letter of the name of the corresponding table followed by an underscore (see parentheses following each table name in Figure 1.2).

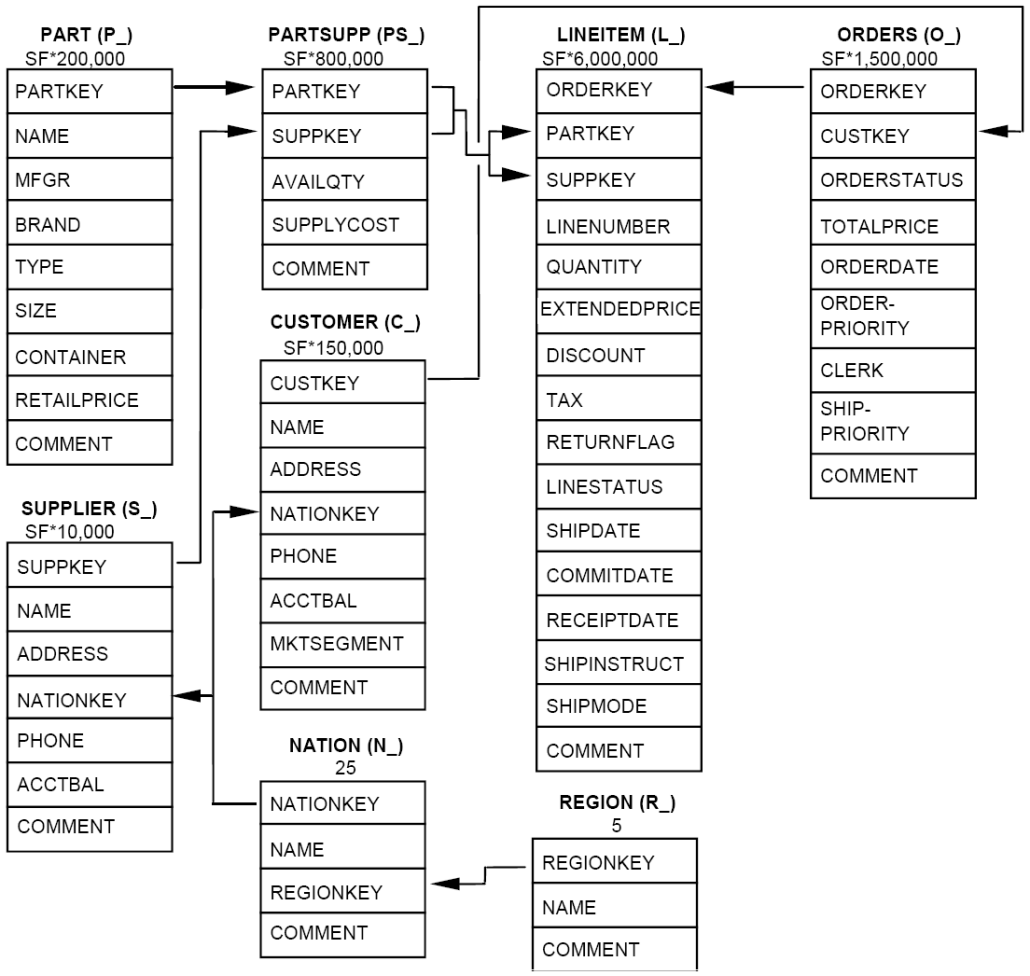


Figure 1.2: Schema of the TPC-H benchmark database [Tra].

1.3 Outline

The remainder of this document is organized as follows. Chapter 2 classifies the different approaches in the area of query optimization and discusses the relation to the optimization approach that we propose in this document. In Chapter 3, we specify how query sequences look like and we introduce the relational model and histograms. In Chapter 4, we introduce our heuristic approach for the optimization of query sequences. This comprises a detailed description of the rewrite rules, a description of possible control strategies, some implementation issues regarding our optimizer prototype, as well as some experiments that show the effectiveness of our approach. Chapter 5 presents our cost-based approach which extends the heuristic approach by a cost estimation component. This comprises a detailed description of the subcomponents of the cost estimation component, a description of several control strategies, some implementation issues regarding the prototype, as well as some experiments. Chapter 6 concludes this work and gives some outlook on future work.

2 Classification And Related Work

In this section, we classify the different approaches in the area of query optimization and discuss the relation to the optimization approach proposed within this thesis. Figure 2.1 shows a classification schema for query optimization. The vertical dimension shows the different levels of complexity starting with single queries that represent the lowest level of complexity and ending with programs and workflows that embed SQL statements and that represent the highest level of complexity. The horizontal dimension shows the different types of optimization, i.e., heuristic optimization and cost-based optimization. Each level of complexity inherits the problems as well as the corresponding optimization techniques from the levels of lower complexity. However, each level of complexity bears new problems, but also reveals new challenges, and offers additional potential for optimization. The following sections describe each level of complexity with respect to optimization problems and corresponding optimization approaches. Finally, we introduce the optimization of query sequences, which is the topic of this thesis, and contrast it with the related work presented in this chapter.

2.1 Single-Query Optimization

Single-query optimization (SQO) in relational DBMSs [Mit95] [Ioa96] [Cha98] [GMUW01] is a well-known topic in database research since the 1970s. Due to the declarative nature of SQL, there are usually multiple alternative ways to compute the result of an SQL query. Each alternative is represented by an execution plan that describes how to compute the result of a query by means of an operation tree. The nodes of such a tree are physical operations where each physical operation represents an implementation of a relational algebra operation. The alternative execution plans for a query mainly differ in the join order, in the join methods, and in the access paths used to access the tables of the database. As the alternative execution plans show varying response times, the goal of SQO is to find an execution plan for which the response time or throughput of a given

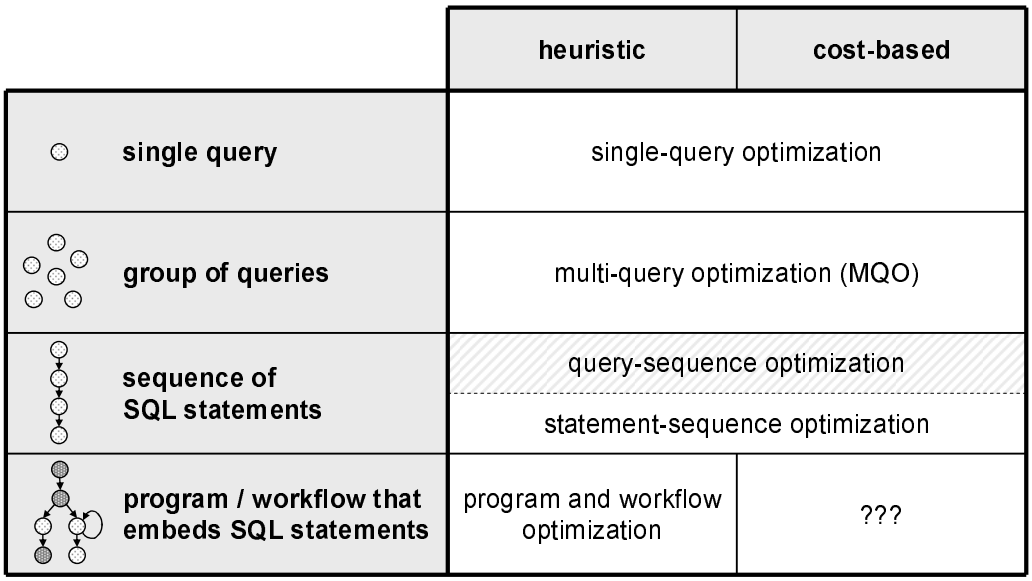


Figure 2.1: Classification of query optimization.

query is close to optimal. After optimization, there are two alternative ways to proceed with the execution plan. One alternative is to translate the execution plan into code which is being executed at runtime. The other alternative is to keep the execution plan as is and interpret it at runtime.

Figure 2.2 shows a classification schema for query optimizers. Basically, optimizers can be divided into two classes: heuristic optimizers and cost-based optimizers. The following section describes these two different classes in more detail. Subsequently, there follows a section about self-tuning optimizers. Since classic optimizers are of a static nature, current research goes towards self-tuning optimizers that supplement existing optimization approaches by adding techniques from machine learning and data mining. So, the optimizer can adapt itself to a given scenario and react dynamically to changes in the scenario.

2.1.1 Heuristic and Cost-Based Optimizers

Heuristic optimizers (like Oracle Rule Based Optimizer [Ora02]) are purely based on heuristic rules. They transform an initial operation tree for a given query in a greedy way into an operation tree that is supposed to be more efficient. A good example for heuristic rules are pushdown rules which aim to move selections and projections in the operation tree as close to the leave nodes as possible. These pushdown rules work under the assumption that these operations are zero-time operations. Another heuristic says that if there exists an index that could be used to answer a query then it has to be used.

Cost-based optimizers enumerate a whole search space of alternative operation trees for a given query. For each of these trees, the optimizer computes a cost estimate and afterwards selects the tree with the least cost estimate. However, often a heuristic optimization-step precedes the cost-based optimization or heuristics are used to prune the search space.

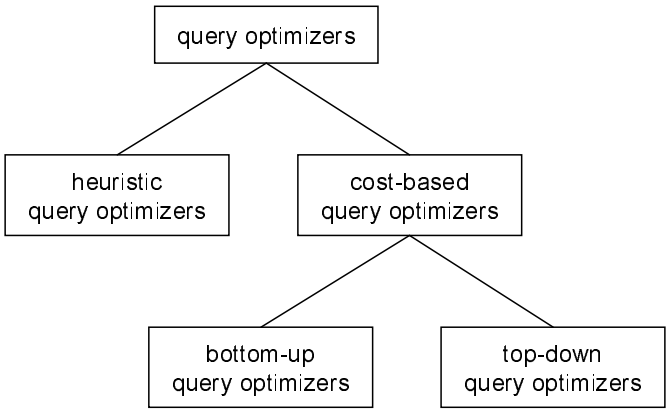


Figure 2.2: Classification of query optimizers.

The class of cost-based optimizers can be further divided into bottom-up optimizers and top-down optimizers.

Bottom-up optimizers (like IBM DB2 [HFLP89] [IBM04a]) usually use dynamic programming to enumerate execution plans. Dynamic programming [Bel57] is based on the assumption that optimal solutions of subproblems can be used to find the optimal solution of the overall problem. So, the problem of finding an efficient join order is recursively broken down into subproblems as follows. First, dynamic programming generates all 2-way join plans by using the access operations of the tables as building blocks and selects the cheapest for each combination of two tables. Thereafter, it takes the selected 2-way join plans and the access operations to build 3-way join plans and again selects the cheapest plan for each combination of three tables. After that, it generates 4-way join plans by considering all combinations of two 2-way join plans and all combinations of a 3-way join plan with an access operation. In the same way, dynamic programming continues to produce plans until it reaches the join orders that cover all tables that should be joined.

Top-down optimizers (like Microsoft SQL Server [Gra95] [Gra96]) transform a given query into an initial logical operation tree. The optimizer traverses this tree in a depth-first manner and thereby generates alternative subtrees. We have to distinguish between logical and physical optimization. Logical optimization applies equivalence rules of relational algebra to subtrees consisting of logical operations and thereby produces alternative subtrees consisting of logical operations, e.g., alternative subtrees with different join orders. Physical optimization transforms logical operations into physical operations. For most of the logical operations, there exist several physical operations that represent different implementations of the same logical operation. Some top-down optimizers completely finish logical optimization for the whole operation tree, before they start physical optimization. Others completely apply logical and physical optimization to the current subtree during traversing before they ascend to the parent node of the subtree's root node.

Cost-based optimizers gather statistics about the data stored in the tables, such as the cardinality of a table or a histogram that approximates the value distribution of an attribute [HK93] [IBM04a] [Ora03a]. The optimizer uses theses statistics to estimate the selectivity of a selection or join operation [SAC+79] [IC93] [IP95a] [IP95b] [PHIS96] [PI97]

[Cha98] [Ioa03]. The selectivity in combination with the input cardinality of an operation is used to estimate the output cardinality of this operation which is the input cardinality of the parent node in the operation tree. In turn, the input cardinality of an operation in combination with some physical statistics is used to compute a cost estimate for the corresponding operation. When we cumulate the costs of all operations, we get a total cost estimate for the whole query. Costs can be divided into I/O costs and CPU costs. The former represent the costs for accessing the I/O system, like reading from disk and writing to disk; the latter represent the costs for computations in the CPU.

Cost-based optimizers rely on several assumptions concerning selectivity estimation [Chr84]. However, these assumptions are often not valid in real-world scenarios which is reflected in errors regarding cardinality estimates and cost estimates. Moreover, these errors are propagated through the execution plan and, in the worst case, the error in the query result grows exponentially with the number of joins [IC91]. Hence, todays query optimizers still have problems with complex, deeply-nested queries that are not just made up of primary-key / foreign-key joins. GROUP BY clauses, arithmetic expressions, and UDFs bear some problems, too. The concept of statistics on query expressions [BC02], which is also known as statistical views [Che06], addresses the problem of estimation errors within an execution plan. In contrast to materialized views (see Section 2.2), statistical views materialize the statistics of the view content at creation time instead of the view content itself. However, similar to materialized views, the optimizer can make use of these materialized statistics when the view definition matches with some part of the execution plan instead of using error-prone cardinality estimates and statistics that have been propagated through several operations. Correlations between the values of multiple columns are another great problem due to the fact that selectivity estimation assumes value independence. Therefore, IBM introduced CORDS [IMH+04] which is a tool that detects correlations for pairs of columns and provides the query optimizer with this information.

2.1.2 Self-Tuning Optimizers

Self-tuning DBMSs [WMHZ02] are a hot topic in todays database research. Microsoft covers this topic in the AutoAdmin project [CN07] [Mic]. At IBM, the SMART initiative [SLMK01] [LLS+02] addresses this topic. The goal is to reduce total cost of ownership (TOC) by reducing the administration effort. On the one hand, this can be realized with tools that support the database administrator by the means of giving useful hints on physical database design and database tuning. On the other hand, this can be realized by additional components in the DBMS that dynamically react to changes in the workload and in the performance of the DBMS to avoid performance breakdowns and to tune the database. These components realize a feedback loop. Therefore, a monitoring component logs the executed SQL statements, the corresponding execution plans and some performance key values of the DBMS. Another component analyzes this monitoring data and automatically triggers the appropriate administration tasks. For the data analysis and the selection of the appropriate reactions, these tools and components employ techniques from machine learning and data mining.

Good results have been achieved in finding an appropriate set of indexes and materialized views (see Section 2.2) for a given database and workload [ACN00] [ZZL+04] [ZRL+04]. Whereas in the past database tuning was mainly triggered by the database user or by the database administrator, today the DBMSs automatically adapt their physical database design according to changes in the workload and according to changes in the characteristics of the data (like changes in the data distribution) [ACN06]. This is, they apply their techniques without user interaction. Latest approaches even allow to dynamically adapt execution plans [GW89] [CG94] or even to reoptimize execution plans during execution [KD98] [MRS+04].

Approaches that not directly concern query optimization but that also have an impact on the performance of query execution address data distribution among disks and size of buffer pools. They also react dynamically according to changes in the database and according to changes in the buffer-pool usage, respectively.

2.2 Multi-Query Optimization

In the late 1980s, the optimization of sets of queries that are independent of each other but that contain some common subexpressions [Fin82] [Sel86] [Sel88] [PS88] [RSSB00] became a hot topic. This kind of optimization is called *multi-query optimization (MQO)* [RC88]. The goal of MQO lies in finding a global execution plan where the total runtime for the whole set of queries is minimal. However, an optimal global execution plan for the whole set of queries in terms of MQO may result in suboptimal execution plans for some of the queries.

All subproblems concerning the identification of common subexpressions and the search for an optimal plan where the computation of common subexpressions can be shared between multiple queries are NP-hard. Common subexpressions can be further divided into common subexpressions within the same query (intra-query common subexpressions) and between different queries (inter-query common subexpressions). Sharing of common subexpressions can also be realized in two alternative ways, i.e., by temporarily materializing the result of the common subexpression or by sharing access paths and pipelines [DSRS01]. In contrast to the second alternative, the first alternative allows to access the materialized result several times and at different points in time, e.g., if the result of the common subexpression is accessed within the inner loop of a nested-loop join. However, in contrast to the second case, the materialization requires storage and, in the worst case, causes I/O costs when the result cannot be held in main memory. Moreover, the materialization adds a blocking operation even if the result of the common subexpression could be computed in a pipelined manner.

Todays commercial DBMSs also provide materialized views [LY85] [CKPS95] [LMS95] [ZCL+00] [GL01] which are based on a view definition like regular views, but their content is materialized like a table. The DBMS automatically updates the materialized views according to changes in the base tables of the view definition. Moreover, the query optimizer makes use of the materialized views transparently for the user when this may speed up the execution of a query. Therefore, common subexpressions that frequently

appear in the queries of a workload can permanently be provided as materialized views and the optimizer may decide for each query whether to use a materialized view. However, the problem of finding the materialized views that match to parts of an operation tree is similar to the problem of recognizing common subexpressions [MRSR01].

Heuristic approaches of MQO appear in the area of continuous queries [CDTW00] [BW01]. Continuous queries are queries defined on data streams. Thus, they are processing data as long as they are registered in the system and, theoretically, they may never terminate. The challenge is to manage and process a great number of queries efficiently. As the resource consumption (CPU and main memory) raises with the number of queries, such systems also try to merge common subexpressions of queries that are being registered in the system with queries that are already registered in the system.

2.3 Statement-Sequence Optimization

An important assumption underlying SQO and MQO is that a workload is a set of SQL statements. Hence, the corresponding optimization approaches do not consider the order of the statements and dependencies between the statements. However, these characteristics offer some optimization potential.

The QUEL* optimizer [SS91] can be referred to as one of the first optimization approaches that considers the special notion of sequences in database access languages. It applies compiler design techniques as well as query optimization techniques to sequences of QUEL* commands. QUEL* is an extension to QUEL that adds two new constructs to QUEL, namely transitive closures of some QUEL commands and a new command that allows to execute a sequence of QUEL* commands. QUEL is a relational database access language that has been developed as part of the *Ingres* project at the University of California, Berkeley. In many ways, QUEL is similar to SQL. This is also the reason why we mention the QUEL* optimizer in this context.

Latest work on automatic selection of physical database design also treats a workload as a sequence of statements and not as a set of statements [ACN06]. The goal is to add statements to the sequence that create and drop physical design structures like indexes so that the overall performance of the modified sequence is maximized. Dropping of a physical design structure makes sense when only some part of the workload benefits from this physical design structure but subsequent parts just cause update costs regarding this physical design structure. Therefore, it's recommended to create structures before queries arrive and drop such structures before updates arrive. The transient-table usage scenario that is mentioned in the introduction of the paper which introduces this approach addresses the topic of query sequences. However, the remainder of that paper does not consider this transient-table usage scenario and the corresponding kind of sequences.

2.4 Program and Workflow Optimization

In this section, we address the optimization of programs and workflows that embed SQL statements. Research in this area is at its beginning. Thus, there is not much research

work to mention, here. Workflows and programs that embed SQL statements are more general than sequences of SQL statements in that they allow for complex control and data flows due to the use of control structures. Usually, the query optimizer of the target database system optimizes the embedded SQL statements and the workflow engine / programming language compiler optimizes the workflow / program code. This means, a workflow or a program is not optimized as a whole, but SQL statements and workflow / program code are optimized in isolation.

Again, we mention the QUEL* optimizer (see Section 2.3) as one of the first optimization approaches that combines compiler design techniques and query optimization techniques. Besides the QUEL* optimizer, there is only one more approach that considers the combination of SQL statements and control flow structures for optimization. This approach [VSS+07] addresses the optimization of data processing in workflows. It is a heuristic approach based on rewrite rules. Great performance improvements have been achieved by transforming tuple-oriented data processing into set-oriented processing, i.e., by replacing a subflow that processes data iteratively into a single set-oriented SQL statement. The focus is on optimization of workflows that are modeled in the *Web Services Business Process Execution Language* (WS-BPEL) [AAA+] and that make use of some WS-BPEL extension which allows to embed SQL statements into WS-BPEL workflows. WS-BPEL is an XML-based, block structured language for modeling executable business-processes. It is an orchestration language that makes use of web services as its external communication mechanism. IBM, Oracle and Microsoft have added proprietary extensions to their WS-BPEL engines that allow to execute SQL statements within WS-BPEL workflows [VSRM08]. However, the optimization concepts proposed in [VSS+07] can also be adapted to other workflow languages or SQL Persistent Stored Modules (SQL/PSM). SQL/PSM is a procedural extension for SQL that is defined by ISO/IEC 9075-4:2003. It adds control-flow elements, condition handling, signals, cursors, local variables and variable assignment to SQL. Thus, it allows to write complex stored procedures, user-defined functions, and trigger actions directly in SQL.

2.5 Query-Sequence Optimization

Query sequences are a special kind of statement sequences that break down a complex information request into a sequence of computational steps. They consist of multiple CREATE TABLE, INSERT and DROP TABLE statements. The INSERT statements have a normal query as body and thus represent the computational steps. They temporarily store the results which are intermediate results of the query sequence in tables created and dropped within the sequence by the CREATE TABLE and DROP TABLE statements. See Section 3.1 for a definition and some examples. Since each computational step can access the results of all previous computational steps in the sequence, there is a producer-consumer dependency between the INSERT statements. The knowledge about these dependencies and the fact that the sequences manage the temporary tables on their own offer some optimization potential and allow for rewriting query sequences. Hence, we are free to rewrite the sequence as long as the final result stays the same.

The approaches introduced in the two previous sections offer some general optimization techniques for sequences of SQL statements and programs or workflows that embed SQL statements but they do not consider the special characteristics of query sequences. The QUEL* optimizer does not capture the optimization potential that lies in the producer-consumer dependencies between the INSERT statements of a query sequence and in the fact that the semantics of the statements may change as long as the semantics of the query sequence do not change. Similarly, the approach concerning automatic selection of physical database design introduced in Section 2.3 accounts for sequences and the optimization of their performance, but it doesn't modify the statements of a sequence. It just adds statements to the sequence that create and drop physical design structures. The workflow optimization approach mentioned in Section 2.4 focuses on transforming tuple-oriented data processing into set-oriented data processing. However, this is a problem which is specific to programs and workflows that embed SQL statements, but this is not a problem that occurs in query sequences. So, our work complements all these approaches with respect to the optimization of query sequences.

In the ETL area, there exists an actual approach on logical optimization of ETL processes [SVS05a] [SVS05b] [SS07] where the ETL processes are acyclic graphs that are comparable to our query sequences in a way that they consist of activities that processes record sets and that are connected via data provider relationships (see definition of query sequences in Section 3.1). However, the activities correspond to the unary and binary logical operations known from relation algebra, whereas we support complete SQL queries at that place. So, this optimization approach is comparable to the optimization at the layer of relational algebra expressions, whereas our optimization approach is settled at the declarative layer of SQL. Moreover, the rewritings which they denote as transitions do not cover merging of similar queries as we do. They only support merging of two equivalent activities that are both input of the same binary activity.

Our first approach on rewriting query sequences is a heuristic rule-based approach that extends and adapts some rules and techniques known from SQO and MQO to our needs, but that also adds new rules. For example, to the best of our knowledge, the *WhereToGroup* rule which we introduce in Section 4.1.1.4 or the transformations this rule applies to an algebraic expression have never been published before. All rules of our rule set consider the specific characteristics of query sequences such as the producer-consumer dependencies between the INSERT statements. For the cost-based approach, we additionally employ some selectivity-estimation and histogram-propagation techniques from classic query optimization and from approximate query answering. Approximate query answering or approximate query processing [GG01] is an approach to provide approximate answers using statistical summaries of the data, such as samples, histograms, and wavelets in order to get an early feedback when a precise answer is not relevant. We exploit and extend the techniques that are based on histograms [IP99] [PGI99] according to our needs. This means, we add support for grouping and aggregation as well as we add support for arithmetic expressions, because these topics are not covered by literature on histogram propagation and approximate query answering. Furthermore, we take respect for different data types and their characteristics, whereas literature is mostly restricted to the integer data type.

3 Basic Definitions

In this chapter, we introduce some terms and notations that we use later on in the remainder of this document. First, we specify how the query sequences look like whose optimization is the focus of this document. Then, we introduce the relational model including relational algebra, since we use expressions of relational algebra to show the correctness of our rewrite rules and as a base for the propagation of histograms in the cost-based optimizer. Finally, we introduce histograms and give definitions for the assumptions that our histogram-propagation algorithms are based on.

3.1 Query Sequences

The syntax diagrams in Appendix A specify the subset of SQL that we use to build the query sequences whose optimization is the focus of this thesis. This subset comprises CREATE TABLE statements, DROP TABLE statements, and INSERT statements with an SQL query as body. The extension of this subset is straightforward but each extension adds to the size and complexity of the rule descriptions and makes it more difficult to argue for correctness of the rules. Therefore, we have chosen a subset that is small enough to keep the size of the rule descriptions within acceptable limits, but huge enough to express a great variety of query sequences typically for ETL tools and reporting tools.

In the following, we introduce a graph representation for query sequences that we call *query dependency graph*. It groups the statements that target the same table to so-called *queries* which represent the vertices of the graph and it explicitly models the producer-consumer dependencies between the INSERT statements as directed edges between the corresponding queries. The rewrite rules that we specify in the next chapter work on this graph representation. Based on the term query dependency graph, we define the term *query sequence*. Furthermore, we specify what semantic equivalence means for query dependency graphs as well as for query sequences. Finally, for illustration purposes, we provide some sample sequences.

3.1.1 Definitions

Here are the formal definitions regarding query dependency graphs and query sequences:

Definition 1 (Intermediate-Result Query)
An intermediate-result query is a triple $q = \langle c, i, d \rangle$ where
c is a CREATE TABLE statement that creates a table t,
i is an INSERT statement that inserts tuples into table t and
d is a DROP TABLE statement that drops table t.

Definition 2 (Final-Result Query)
A final-result query is a pair $q = \langle c, i \rangle$ where
c is a CREATE TABLE statement that creates a table t and
i is an INSERT statement that inserts tuples into table t.

Definition 3 (Query)
A query is either an intermediate-result query or a final-result query.

We refer to the table t which is created, filled, and dropped by the statements of a query as the *target table* of that query.

Definition 4 (Direct Query Dependency)
A query q_2 directly depends on a query q_1 iff the INSERT statement of q_2 accesses the target table of q_1 within its body. This is written as $q_1 \rightarrow q_2$.

Definition 5 (Query Dependency Graph)
A query dependency graph $QDG = \langle Q, D \rangle$ is an acyclic directed graph defined by a set of queries Q representing the vertices of the graph and a set of direct query dependencies D representing the directed edges of the graph. Q consists of a single final-result query and an arbitrary number of intermediate-result queries where each intermediate-result query must have a directed path of direct query dependencies to the final-result query. Furthermore, each two queries within the same query dependency graph must not use the same table name for their target tables.

Definition 6 (Query Sequence)
An ordered list of SQL statements $S = \langle s_1, s_2, ...s_n \rangle$ is a query sequence iff there exists a query dependency graph $QDG = \langle Q, D \rangle$ that satisfies each of the following conditions ($p(S, s)$ is the position of statement s within query sequence S):

- $\forall\ q = \langle c, i, d \rangle \in Q :\ (\exists^1 s \in S :\ s = c)\ \wedge\ (\exists^1 s \in S :\ s = i)\ \wedge\ (\exists^1 s \in S :\ s = d)$
- $q = \langle c, i \rangle \in Q :\ (\exists^1 s \in S :\ s = c)\ \wedge\ (\exists^1 s \in S :\ s = i)$
- $\forall\ s \in S :\ \exists^1\ q \in Q :\ s \in q$
- $\forall\ q = \langle c, i, d \rangle \in Q :\ p(S, c) < p(S, i) < p(S, d)$,
- $q = \langle c, i \rangle \in Q :\ p(S, c) < p(S, i)$,

- $\forall\ q_1 \rightarrow q_2 \in D,\ q_1 = \langle c_1, i_1, d_1 \rangle\ ,\ q_2 = \langle c_2, i_2, d_2 \rangle :\ p(S, i_1) < p(S, i_2) < p(S, d_1),$
- $\forall\ q_1 \rightarrow q_2 \in D,\ q_1 = \langle c_1, i_1, d_1 \rangle\ ,\ q_2 = \langle c_2, i_2 \rangle :\ p(S, i_1) < p(S, i_2) < p(S, d_1).$

The first, the second, and the third condition of the query sequence definition guarantee that each statement that occurs in the queries of a query dependency graph also occurs once in the corresponding query sequence and vice versa. The fourth and the fifth condition define the order of the statements with the same target table in the query sequence, i.e., the CREATE TABLE statement is located before the INSERT statement and the INSERT statement is located before the DROP TABLE statement. The last two conditions determine the order of the statements of two queries in the query sequence when there is a direct query dependence between these queries. This is, when query q_2 depends on query q_1, the INSERT statement of query q_2 has to be located after the INSERT statement of query q_1 and the DROP TABLE statement of query q_1 has to be located after the INSERT statement of query q_2. This guarantees that the target table of query q_1 is filled and still exists when the INSERT statement of query q_2 wants to access it.

Usually, there exist multiple alternative query sequences that correspond to the same query dependency graph. These query sequences consist of the same statements, they only differ in the order of the statements within the sequence. Since the set of queries of a query dependency graph consists of a single final-result query, which is a pair of statements, and an arbitrary number of intermediate-result queries, which are triples of statements, all query sequences that correspond to a query dependency graph $QDG = \langle Q, D \rangle$ contain $(|Q| - 1) \cdot 3 + 2 = |Q| \cdot 3 - 1$ statements (where $|Q|$ is the number of queries in the query dependency graph QDG).

For the subsequent definitions regarding semantic equivalence, we presume that each query sequence is being executed within its own single ACID (atomicity, consistency, isolation and durability) transaction. So, if the execution of any statement within a query sequence fails, the complete transaction has to be rolled back. This has to be guaranteed by the database system that executes the query sequence, i.e., the database system has to drop all tables created by the query sequence. Furthermore, we presume that the set of names used for tables created by the CREATE TABLE statements of a query sequence and the set of names used for base tables and views within the database, on which this query sequence should be executed, are disjoint. Practically, this can be guaranteed by the use of a unique name prefix that is only used for tables created by the CREATE TABLE statements of a query sequence.

Definition 7 (Semantic Equivalence of Query Dependency Graphs)
Given two query dependency graphs, QDG_1 and QDG_2, we call them to be semantically equivalent iff they compute the same final result, i.e., when we execute two corresponding query sequences, S_1 and S_2, against the same DBMS and the same database state, the target table of the final-result query in QDG_1 has the same structure (table name, column names, and column data types) and contains the same tuples as the target table of the final-result query in QDG_2.[*]

[*] We assume that rounding errors do not occur or that they have no influence on the final result.

Definition 8 (Semantic Equivalence of Query Sequences)
Given two query sequences, S_1 and S_2, we call them to be semantically equivalent iff the corresponding query dependency graphs, QDG_1 and QDG_2, are semantically equivalent.

We conclude that two semantically equivalent query sequences may differ

- in the order of the statements,
- in the name and structure of the intermediate-result tables,
- in the number of tables that they create and hence in the number of statements,
- in the body of the INSERT statements, i.e., the way in which they compute the intermediate results and the final result of the sequence.

Since the query sequences that correspond to the same query dependency graph only differ in the order of the statements, they are pairwise semantically equivalent.

3.1.2 Sample Sequences

Figure 3.1 shows two simple query sequences that are semantically equivalent.

The first query sequence consists of eight statements, i.e., three CREATE TABLE statements, three INSERT statements, and two DROP TABLE statements. Thus, the corresponding query dependency graph has two intermediate-result queries, q_1 and q_2, and a single final-result query q_3. Query q_1 consists of the first CREATE TABLE statement, the first INSERT statement, and the first DROP TABLE statement in the sequence. Query q_2 consists of the second CREATE TABLE statement, the second INSERT statement, and the second DROP TABLE statement in the sequence. Finally, query q_3 consists of the third CREATE TABLE statement and the third INSERT statement in the sequence. Furthermore, there are two direct query dependencies, $q_1 \rightarrow q_3$ and $q_2 \rightarrow q_3$, between the intermediate-result queries and the final-result query. The body of the INSERT statement of query q_1 computes the turnover in the year 1990 for each customer. The body of the INSERT statement of query q_2 computes the same for the year 1991. Therefore, both INSERT statements access the base table *orders* which contains the orders of all customers and all years. Afterwards, they select a certain year and group by the customer key. The body of the INSERT statement of query q_3 retrieves those customers whose turnover is in 1991 greater than in 1990 and returns their key and their name. For this purpose, it joins the target tables of query q_1 and query q_2 and applies the predicate that realizes the comparison. Additionally, it joins the base table *customer*, because this table contains the customer names for the customer keys.

The second query sequence computes the same final result as the first query sequence but uses less statements. It only contains five statements and therefore the corresponding query dependency graph only consists of a single intermediate-result query and a single final-result query. The body of the INSERT statement of the intermediate-result query computes the turnover in the years 1990 and 1991 for each customer. This intermediate result is the union of the two intermediate results of the previous query sequence. In order to compute the final result, these two parts have to be separated again. For this

```
CREATE TABLE t1 (custkey INTEGER, turnover1990 FLOAT);

CREATE TABLE t2 (custkey INTEGER, turnover1991 FLOAT);

CREATE TABLE t3 (custkey INTEGER, name VARCHAR(25));

INSERT INTO t1
   SELECT   o.o_custkey, SUM(o.o_totalprice)
   FROM     orders o
   WHERE    year(o.o_orderdate) = 1990
   GROUP BY o.o_custkey;

INSERT INTO t2
   SELECT   o.o_custkey, SUM(o.o_totalprice)
   FROM     orders o
   WHERE    year(o.o_orderdate) = 1991
   GROUP BY o.o_custkey;

INSERT INTO t3
   SELECT   c.c_custkey, c.c_name
   FROM     t1, t2, customer c
   WHERE    t1.custkey = c.c_custkey AND t1.custkey = t2.custkey AND
            t2.turnover1991 > t1.turnover1990;

DROP TABLE t1;

DROP TABLE t2;
```

```
CREATE TABLE t1 (custkey INTEGER, orderyear INTEGER,
                 turnover FLOAT);

CREATE TABLE t3 (custkey INTEGER, name VARCHAR(25));

INSERT INTO t1
   SELECT   o.o_custkey, year(o.o_orderdate), SUM(o.o_totalprice)
   FROM     orders o
   WHERE    year(o.o_orderdate) IN (1990, 1991)
   GROUP BY o.o_custkey, year(o.o_orderdate);

INSERT INTO t3
   SELECT   c.c_custkey, c.c_name
   FROM     t1 t1, t1 t2, customer c
   WHERE    t1.custkey = c.c_custkey AND t1.custkey = t2.custkey AND
            t2.turnover > t1.turnover AND
            t1.orderyear = 1990 AND t2.orderyear = 1991;

DROP TABLE t1;
```

Figure 3.1: Two semantically equivalent query sequences.

reason, the table created by the intermediate-result query contains an additional column *year*. For the separation, the INSERT statement of the final-result query contains two additional predicates where each predicate selects a certain year. So, the second query sequence in Figure 3.1 can be obtained from the first query sequence in Figure 3.1 by pushing up the predicates that select a certain year through the GROUP BY operation and afterwards merging the intermediate-result queries.

When we would place the CREATE TABLE statements directly before the corresponding INSERT statements instead of placing them at the begin of the sequences, we would get some alternative but still semantically equivalent sequences.

3.2 Relational Model

In this section, we introduce the relational model [Cod69] [Cod70]. Since literature offers various definitions regarding the terms and operations of the relational model and its extensions and due to the fact that they often differ in some details, we provide our own definitions in the following. First, we define some basic terms regarding the relational model. Then, we introduce the operations that operate on the data structures of the relational model. Finally, we specify how an INSERT statement that is part of a query sequence can be translated into an expression of the relational algebra.

3.2.1 Definitions

Definition 9 (Domain)
A domain is a set of atomic values where atomic means that each value is indivisible with respect to the relational model.

Typical domains are basic data types like integers or strings. By adding constraints or specifying a format that the values have to satisfy, the domain can be further restricted, e.g., the set of all postal codes in Germany or the set of all strings with length 10.

Definition 10 (Attribute)
An attribute is the name of a role played by a domain in a relation.

Definition 11 (Relation Schema)
A relation schema $R\langle A_1, ..., A_n\rangle$ *defines the specific structure of a relation* R*. It consists of a relation name* R *and an ordered list of attributes* $\langle A_1, ..., A_n\rangle$ *with* $n \geq 1$*. It defines the domain of the relation as the cross product of its attribute domains, i.e.,* $dom(R) = dom(A_1) \times ... \times dom(A_n)$*.*

Definition 12 (Relation Instance)
A relation instance r *of a relation* R *is an instantiation of the relation's schema and therefore a subset of the relation's domain, i.e.,* $r(R) \subseteq dom(R)$*. Given* $R\langle A_1, ..., A_n\rangle$ *as the relation schema of* R*, the relation instance of* R *is a set of tuples* $\{t_1, ..., t_m\}$ *where each tuple is an ordered list of* n *values* $\langle v_1, ..., v_n\rangle$ *with* $v_i \in dom(A_i)$ *for* $1 \leq i \leq n$*.*

A relation instance is also denoted as the *extension of the relation*. The number of attributes in the relation schema is called *degree of the relation*, whereas the number of tuples in the relation instance is called *cardinality of the relation*. As the relation instance is a set of tuples and as there is no ordering among the elements of a set, there is also no ordering among the tuples of a relation instance.

Definition 13 (Relation)
A relation R (in the sense of the relational model) consists of a relation schema $R\langle A_1, ..., A_n\rangle$ *and an appropriate relation instance* $r(R)$.

Definition 14 (Multi-Relation)
A multi-relation is a relation where the instance is not a set but a multi-set of tuples. So a multi-relation extends a relation (as defined above) by allowing duplicates in the relation instance.

Definition 15 (NULL value)
The $NULL$ *value is a special value that represents an unknown value or a non-existing value.*

In a sense, the $NULL$ value is not an element of any data type. However, in the remainder of this paper, we presume that each domain includes the $NULL$ value.

In the subsequent chapters, especially when we refer to DBMSs, we also use the term *column* instead of *attribute* and the term *table* instead of *relation*. This is due to the fact, that the terms column and table are more common when talking about database systems whereas the terms attribute and relation are more common when talking about aspects of database theory.

3.2.2 Relational Algebra

Figure 3.2 shows a table which contains notation, name, and a short description for all operations that we need when we translate an SQL INSERT statement (see Appendix A), which is part of a query sequence, into an expression of relational algebra. However, we omit to list basic operations as well as derived operations that we do not need later on in the remainder of this document. All operations require one or two relations as input and produce a single relation as output. Hence, multiple operations can be combined in that way that the output of one operation is the input of another operation. The result is a tree of algebraic operations where the leaf nodes are operations that directly access base tables in a database (see right side of Figure 3.3). When a subtree appears twice within such a tree, we can merge these two identical subtrees into a single subtree whose result is accessed twice. In this case, the tree becomes a graph.

3.2.3 Mapping SQL to Relational Algebra

Figure 3.3 shows the mapping from an SQL INSERT statement which is part of a query sequence onto an expression of relational algebra. (See Appendix C for a description of the functions used in the formulas in the bottom left corner of Figure 3.3.) We make

Notation	Name	Description
$\pi_{ae,a}$	Projection	Unary operation that projects the elements in the arithmetic expression list ae onto the elements in the attribute list a. Hence, list a contains as much elements as list ae and each element in list a corresponds to the element at the same position in list ae.
σ_p	Selection	Unary operation that selects those tuples of the input relation that satisfy predicate p which is a logical expression.
$\gamma_{ae_1,ae_2,a}$	Grouping	Unary operation that groups the input relation according to the arithmetic expression list ae_1 and, for each group, computes the aggregate expressions given by the arithmetic expression list ae_2. It assigns the elements of the two lists to the corresponding attributes in the attribute list a. Hence, list a contains as much elements as list ae_1 and list ae_2 together and each element in list a corresponds to the element at the same position in the concatenation of list ae_1 and list ae_2.
$\rho_{t,c}$	Renaming	Unary operation that renames the table with table name t using correlation name c.
$\times$	Cartesian product	Binary operation that computes the Cartesian product of the two input tables, i.e., it combines each tuple from the first input table with each tuple from the second input table.

Figure 3.2: Table with operations of relational algebra.

use of this translation when we show the correctness of our rules and when we propagate histograms through INSERT statements.

The construction of the relational algebra tree works as follows. First, we create a rename operation for each table reference in the FROM clause. This rename operation maps the table name t_m to the corresponding correlation name c_{im} as specified in the table reference. Then, we combine the results of all rename operations by Cartesian products which results in a left-deep binary tree. Now, we apply the WHERE clause w_i to the result of this tree by a selection operation. Afterwards, we group the result by the list of arithmetic expressions in the GROUP BY clause g_i. The grouping operation also computes all aggregate expressions agg_i that appear as part of an arithmetic expression in the HAVING clause h_i or in the SELECT clause s_i and assigns some internal attributes ia_i to these aggregates. Thus, in the HAVING clause h_i and in the SELECT clause s_i, we have to replace the aggregates agg_i by the corresponding attributes ia_i. Then, we apply the modified HAVING clause h_i' to the result of the grouping operation by an additional selection operation. Afterwards, a projection operation maps the arithmetic terms listed in the modified SELECT clause s_i' onto the attributes of the result table of the INSERT

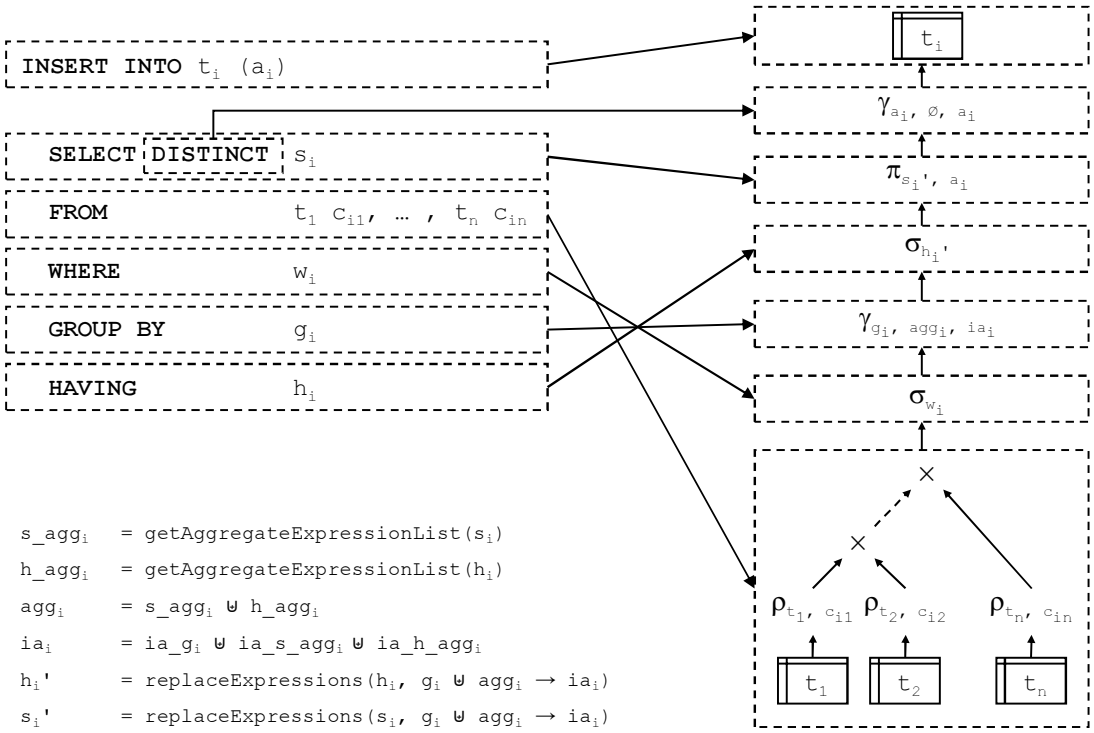


Figure 3.3: Mapping SQL to relational algebra.

statement. Finally, a grouping operation realizes the duplicate elimination caused by the DISTINCT in the SELECT clause.

When a certain clause is missing in an INSERT statement, the algebraic operation that corresponds to this clause is also not included in the tree of algebraic operations. The only exception is when an INSERT statement does not include a GROUP BY clause, but the SELECT clause contains aggregation expressions. Then, the query computes the aggregates over all tuples resulting in a table that consists of a single tuple. In this case, the tree of algebraic operations contains a grouping operation where the list of grouping expressions g_i is empty. When there is no grouping expression included in the tree of algebraic operations, the SELECT clause is not being modified and therefore the projection operation corresponding to the SELECT clause uses s_i instead of s_i'.

3.3 Histograms

Histograms [Ioa03] are a widely investigated concept of statistical summaries. They allow to summarize the content of large tables by approximating the data distribution of their attributes. Commercial DBMSs like IBM DB2 [IBM04a], Oracle [Ora03a], and Microsoft SQL Server [HK93] store unidimensional histograms in their catalog tables. Their query optimizers use them to estimate the selectivity of point queries, range queries, and queries containing equi-joins [Cha98] [Ioa03] [IC93] [IP95b] [IP95a] [PHIS96] [PI97]. Furthermore, histograms can be used in approximate query answering [IP99] [PGI99].

As we use histograms in our cost-based approach, below we define some basic terms regarding histograms and extend multi-relations to histogram-relations. We specify the characteristics of histograms, introduce a taxonomy for histograms, explain some domain-specific properties of histograms, and introduce serialization and normalization. Finally, we define some assumptions that we use later on in the remainder of this document.

3.3.1 Definitions

We adopt the following definitions from [GMP02] and [Ioa03]. The definitions presume a relation R with relation schema $R(A_1, ..., A_n)$ and relation instance $r(R)$.

Definition 16 (Value Set)
The value set V_i of attribute A_i ($1 \leq i \leq n$) is the set consisting of the attribute values for A_i that are present in $r(R)$. So, the value set is a subset of the attribute's domain, i.e., $V_i \subseteq dom(A_i)$.

Definition 17 (Frequency)
The frequency $f(v)$ of a value v in the value set V_i of attribute A_i ($1 \leq i \leq n$) is the number of tuples in $r(R)$ for whom the attribute value for A_i equals v.

Definition 18 (Data Distribution)
The data distribution D_i of attribute A_i ($1 \leq i \leq n$) is the set $D_i = \{\langle v, f(v)\rangle \mid v \in V_i\}$ that contains a value-frequency pair for each value in the value set V_i of attribute A_i.

Definition 19 (Bucket)
A bucket B represents a range in the domain of an attribute A_i ($1 \leq i \leq n$) as a 4-tuple $\langle low, high, card, dv\rangle$ with $dom(B) = dom(A_i) \times dom(A_i) \times \mathbb{Q} \times \mathbb{Q}$ where:

- low *is the lower bound of the range,*
- high *is the upper bound of the range,*
- card *is the cardinality, i.e., the sum of the frequencies of all values in the subset of the value set of attribute A_i which is bounded by* low *and* high*,*
- dv *is the number of distinct values, i.e., the number of different values in the subset of the value set of attribute A_i which is bounded by* low *and* high*.*

The NULL value is covered by a special bucket called *NULL bucket* where *low* and *high* is set to NULL and *dv* is set to 1. This eases computations because NULL buckets can be handled similar to 'normal' buckets.

We denote the size of an interval $[low, high]$ as $length(low, high)$. For a bucket B, we denote the size of the interval specified by the bounds $B.low$ and $B.high$ as $length(B)$. A bucket where *low* equals *high* is called a singleton bucket.

Definition 20 (Histogram)
A histogram H for an attribute A_i ($1 \leq i \leq n$) is a list of buckets $H = \langle B_1, B_2, ..., B_m\rangle$ with $m \geq 0$.

A histogram is constructed by partitioning the data distribution of the corresponding attribute into a set of mutually disjoint buckets approximating the frequencies and values in the range of each bucket. Thus, a histogram is an approximation of the attribute's data distribution. A taxonomy of different histogram types can be obtained by using different rules for partitioning values into buckets. See the next section for details.

Definition 21 (Histogram Relation)
A histogram relation is a relation where the instance is not a set of tuples but a single tuple that contains a histogram for each of the attributes in the corresponding relational schema.

Multi-relations can be mapped to histogram relations by approximating the data distributions of all attributes by histograms. All histograms that are part of the same instance have the same cardinality which we denote as the cardinality of the histogram relation.

3.3.2 Taxonomy

According to [PHIS96], we can build a taxonomy of different histogram types. The partitioning rule is the main characteristic when classifying histograms. It determines the buckets of a histogram and it is characterized by the following four properties that are mutually orthogonal:

- *Partition Class:* Indicates restrictions on partitioning.
- *Partition Constraint:* The mathematical constraint that uniquely identifies the histogram within its partitioning class.
- *Sort Parameter:* Indicates which property of the data distribution is being used for sorting and grouping when building the buckets.
- *Source Parameter:* Is used in conjunction with the partitioning constraint in identifying a unique partitioning.

sort parameter	source parameter			
	spread	frequency	area	cum. frequency
value	equi-sum	equi-sum v-optimal maxdiff compressed	v-optimal maxdiff compressed	spline-based v-optimal
frequency		v-optimal maxdiff		
area			v-optimal maxdiff	

Figure 3.4: Histogram taxonomy [PHIS96].

The most important partitioning class of histograms in the database area is the class of *serial* histograms. Serial histograms require that the buckets within the histogram are not overlapping with respect to the sort parameter. The class of *end-biased* histograms is a subclass of the class of serial histograms which requires at most one non-singleton bucket within the histogram. The class of *biased* histograms falls between the two mentioned classes of histograms because biased histograms have at least one singleton bucket and possibly multiple non-singleton buckets.

For the class of serial histograms, the following partition constraints have been defined for various source parameters (definitions are taken over from [PHIS96]):

- *Equi-sum:* In an equi-sum histogram with β buckets, the sum of the source values in each bucket is equal to $1/\beta$ times the sum of all the source values in the histogram.
- *V-optimal:* In a v-optimal histogram, the weighted variance of the source values is minimized. That is, the quantity $\sum_{j=1}^{\beta} n_j V_j$ is minimized where n_j is the number of entries in the jth bucket and V_j is the variance of the source values in the jth bucket.
- *Spline-based:* In a spline-based histogram, the maximum absolute difference between a source value and the average of the source values in its bucket is minimized.
- *Maxdiff:* In a maxdiff histogram with β buckets, there is a bucket boundary between two source parameter values that are adjacent (in sort parameter order) if the difference between these values is one of the $\beta - 1$ largest such differences.
- *Compressed:* In a compressed histogram, the n highest source values are stored separately in n singleton buckets; the rest is partitioned as in an equi-sum histogram.

Attribute value, frequency, and area are typical sort parameters. Area is the result of multiplying frequency and spread where spread is the distance between two adjacent values in the data distribution. Typical source parameters are spread, frequency, cumulated frequency, and area.

Figure 3.4 provides an overview of the property combinations that have been proposed in the past. Equi-sum histograms with sort parameter value and source parameter spread are also called *equi-width* histograms. Euqi-sum histograms with sort parameter value and source parameter frequency are also called *equi-depth* histograms.

Besides unidimensional histograms based upon scalar values, there also exist multidimensional histograms where the values are coordinates in a multidimensional domain. However, generating and processing a multidimensional histogram is complex and expensive. There are also new types of histograms like STHoles [BCG01]. STHoles is a type of histogram that allows bucket nesting to capture data regions with reasonably uniform tuple density. Furthermore, STHoles histograms are built without examining the data sets, but rather by just analyzing query results.

Since most commercial database systems store only unidimensional histograms and use only unidimensional histograms for query optimization, our histogram propagation approach is based on unidimensional histograms, too. As our definition of buckets and

histograms is very flexible, the corresponding implementation supports a great variety of unidimensional histograms and is not restricted to the characteristics of a certain histogram type.

Besides the four properties listed above, histograms are also characterized by the assumptions that determine how values and frequencies are approximated within a bucket. The value approximation and the frequency approximation is independent of the partitioning rule. The most common alternatives of assumptions for value approximation are the *Continuous Value Assumption* and the *Uniform Spread Assumption*. The *Continuous Value Assumption* assumes that all possible values of the corresponding domain that lie in the range of the bucket are present. The *Uniform Spread Assumption* assumes that each value within a bucket has a spread equal to the average spread of the bucket. However, in comparison to the *Continuous Value Assumption*, the *Uniform Spread Assumption* requires to store the number of distinct values in the bucket. All histograms base upon the *Uniform Frequency Assumption* for frequency approximation, i.e., they approximate all frequencies in a bucket by the average frequency of the bucket. The *Uniform Frequency Assumption* is also known under the name of the *Uniform Distribution Assumption* which we use later on in this document when we refer to this assumption.

The left side of Figure 3.5 shows a sample distribution consisting of seven distinct values with different frequencies. We group the values into two buckets where the first bucket consists of the first four values in the sample distribution and the second bucket consists of the three subsequent values. The right side of Figure 3.5 shows the resulting histogram with the values and frequencies that we approximate under the *Uniform Spread Assumption* and *Uniform Distribution Assumption*. Therefore, all values within a bucket have the same frequency and the spread between adjacent values is the same for all values within a bucket.

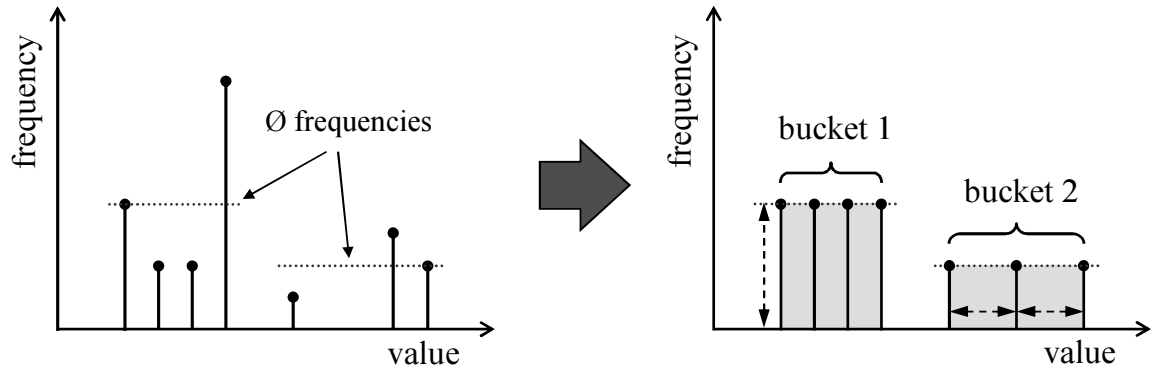


Figure 3.5: Sample data distribution and a histogram that approximates the values of this sample data distribution.

3.3.3 Domain-Specific Properties

Each domain has its own properties which is reflected in the way we treat histograms of attributes of specific domains.

For histograms defined on the domain of integer numbers, the following holds: The number of distinct values of a bucket is delimited by the maximum number of distinct values between its bucket bounds, because an integer interval contains just a finite number of values where the spread between two adjacent values is always 1. This number which we also use as length for integer buckets and intervals can be calculated as $(high - low) + 1$. Hence, the inequation $dv \leq (high - low) + 1$ holds for an integer bucket. Each open or half-open integer interval can be transformed into a closed one by replacing the bounds not included in the interval by the appropriate adjacent values, e.g., $(0, 5) = [1, 5) = (0, 4] = [1, 4]$. Accordingly, each integer bucket where one or both bounds are not included can be transformed into an integer bucket that includes both bounds.

Date values are similar to integer numbers. Therefore, they share the same properties and most of the operations defined on integer histograms.

For histograms defined on the domain of decimal numbers, the following holds: Theoretically, the number of values in a decimal interval is infinite. Hence, there is no limit for the number of distinct values. The length of a decimal interval or bucket is calculated as $high - low$. Later on, we treat open or half-open intervals as closed intervals, i.e., $(low, high) \approx [low, high]$, because the spread between adjacent decimal values is infinitely small and therefore the adjacent values of a bound cannot be determined. Accordingly, we allow that adjacent buckets in a serial decimal histogram overlap in a single point, i.e., adjacent decimal buckets may overlap in their bounds.

Histograms defined on the domain of character strings have similar properties as histograms defined on the domain of decimal numbers. For some calculations, the bounds of string buckets have to be mapped to decimal numbers. The difficulty with character strings is that comparisons with character string values may lead to differing results on various DBMSs depending on the order defined on them.

3.3.4 Serialization and Normalization

Since some of our histogram-propagation algorithms produce non-serial histograms as output, whereas others require serial histograms as input, we have to transform non-serial histograms into serial histograms. For this purpose, we split each bucket at the boundaries of the other buckets that overlap this bucket and afterwards we merge all buckets with same boundaries with respect to the *Uniform Distribution Assumption* and the *Continuous Value Assumption*.

Sometimes it could also be necessary to reduce the number of buckets in a histogram by merging adjacent buckets. We call this transformation normalization later on. Normalization is applied after serialization. We select the buckets that should be merged in that way that the error regarding frequency and number of distinct values due to merging the buckets is minimal.

Figure 3.6 shows a sample for serialization and normalization. The top of Figure 3.6 shows a sample histogram which is not a serial histogram, because the second bucket

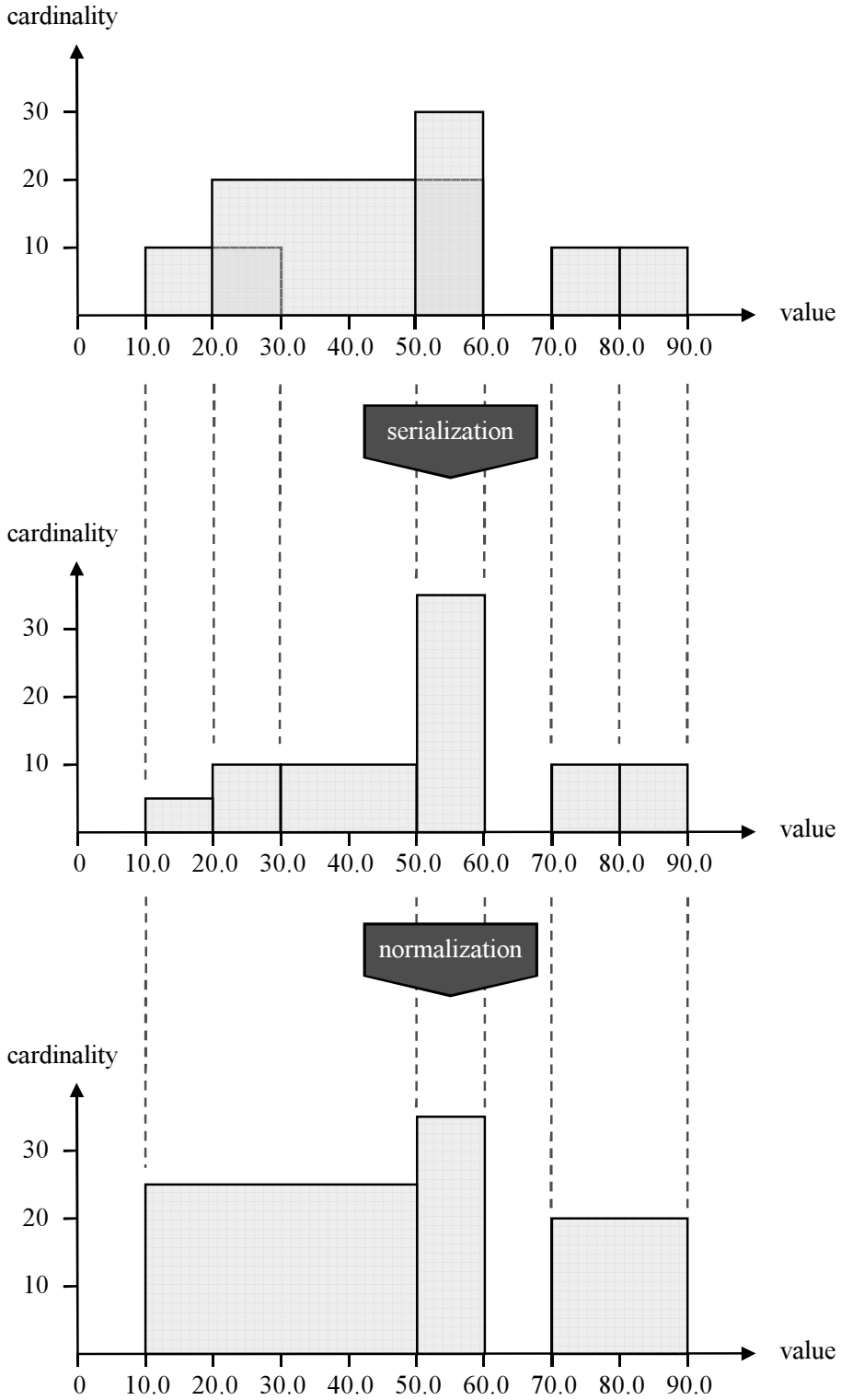


Figure 3.6: Original, serialized, and normalized sample histogram.

overlaps the first and the third bucket. (Note, that the height of a gray bar represents the total cardinality of the corresponding bucket and not the average cardinality of the distinct values that fall into this bucket). The center of Figure 3.6 shows the histogram after serialization. In this histogram, the buckets do not overlap, because the first, second, and third bucket in the original histogram have been split into a set of four new buckets in the serial histogram. Presuming that we want to reduce the number of buckets to a maximum of three buckets, the bottom of Figure 3.6 shows the serial histogram after normalization. The normalization merges the first, second, and third bucket into a single bucket as well as it merges the fifth and sixth bucket into a single bucket.

3.3.5 Assumptions

Our histogram-propagation algorithms rely on some assumptions [Ioa03] [SAC+79] [Chr84] which we define below. We already introduced some of them in Section 3.3.2. The *Uniform Distribution Assumption* and the *Continuous Value Assumption* capture how values are approximated within a bucket, the *Attribute Value Independence Assumption* and the *Inclusion Assumption* concern dependencies between the values of different attributes and value sets, respectively.

Definition 22 (Uniform Distribution Assumption)
Frequencies of all values within a bucket are the same.

Definition 23 (Continuous Value Assumption)
A bucket represents all values of the domain within the interval between its lower and upper bound.

Definition 24 (Attribute Value Independence Assumption)
All attributes are treated as if independent of each other, i.e., there are no correlations between attributes.

Definition 25 (Inclusion Assumption)
Given two value sets V_1 and V_2, we assume that each value of the smaller value set has a match in the larger value set, i.e., $V_1 \subseteq V_2$ or $V_1 \supseteq V_2$. (This is frequently true for a join between foreign key and primary key when the foreign key contains no NULL values.)

4

Heuristic Optimization of Query Sequences

In this chapter, we focus on a heuristic approach for query sequence optimization which we call *coarse-grained optimization* (CGO). (An extract of the work which we present in this chapter has been published in [KSRM03] and [KS04].) Figure 4.1 gives an architectural overview of the heuristic CGO optimizer. The optimizer consists of four major components: the SQL parser, the SQL retranslator, the CGO rule set and the heuristic control strategy. The SQL parser transforms a query sequence into an internal structure on which the rewrite rules can be applied. As internal structure, we use the query dependency graph because it groups the sequence of SQL statements into queries and explicitly models the dependencies between these queries which eases the formulation of the rewrite rules. At the end of the optimization process, the SQL retranslator transforms the optimized query dependency graph into a query sequence again. As there exist multiple query sequences that correspond to a single query dependency graph, we have to decide for one of them. As these query sequences only differ in the order of the statements, mainly in the order of the CREATE TABLE and DROP TABLE statements, this decision should have no significant influence on performance. The CGO rule set is the set of rewrite rules where each rewrite rule transforms a query dependency graph into a semantically equivalent query dependency graph with respect to the definition of semantic equivalence in Section 3.1.1. The control strategy decides which of those rules are applied to a given query dependency graph and in which order they are applied. In the heuristic version of the CGO optimizer, this decision is based on heuristics.

4.1 Rewrite Rules

In this section, we describe the rewrite rules of the CGO optimizer. Each rule consists of two parts, a condition part and an action part. If the rule condition is satisfied, the rule

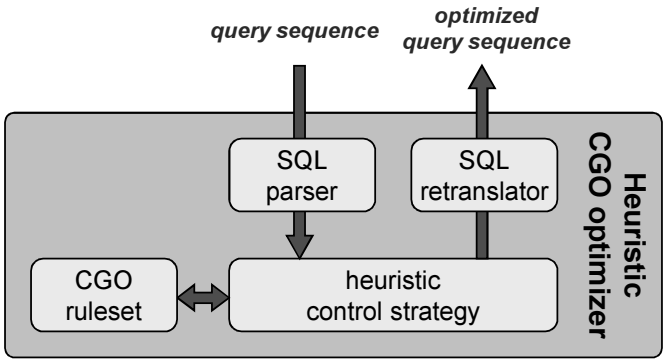


Figure 4.1: Architecture of the heuristic CGO optimizer.

action can be applied. The rule condition specifies how a subset of the queries and direct query dependencies of a query dependency graph have to look like so that the rule action can be applied to that subgraph. The rule action specifies how the query dependency graph has to be transformed so that the result of the rule application is a sequence that is equivalent to the sequence prior to the rule application with respect to the definition of semantic equivalence in Section 3.1.1. Hence, each rewrite rule transforms a consistent query sequence into another consistent query sequence.

We distinguish between unary and binary rules. The condition of a unary rule specifies the properties of a single query, whereas the condition of a binary rule specifies the properties of two queries as well as the similarities and differences between both queries. Later on, we call these queries the *target queries* of the rule condition. Furthermore, a rule condition may specify properties of queries that depend on the target queries and properties of queries the target queries depend on. The rule action modifies the target queries and possibly adapts the queries that depend on the target queries accordingly.

Based on the characteristics of the rule condition, rewrite rules for query sequences can be divided into three classes:

Class 1: Class-1 rules merge two queries that have certain similarities into a single query and therefore reduce the number of queries in the corresponding query dependency graph. Hence, these rules are binary rules. Rules that belong to class 1 typically apply MQO techniques, i.e., they search for common subexpressions within the bodies of the INSERT statements.

Class 2: These rules are unary rules where the rule condition specifies dependencies among a subset of the queries of a query dependency graph. Rules of class 2 merge queries connected by a direct query dependency, move predicates along the direct query dependencies, or eliminate unnecessary attributes from the target table of the target query.

Class 3: The condition of class-3 rules is restricted to the context of a single query in the query dependency graph. These rules eliminate redundancies which might have been caused by the application of class-1 rules.

The remainder of this section is structured as follows. There is a subsection for each rule class addressing all rules of this class. The rule descriptions in these subsections are brief non-formal descriptions of the rule conditions and rule actions. However, a more formal representation of the rule conditions and rule actions in pseudo code can be found in Appendix C. Moreover, in the descriptions of the rule actions, we refer to the corresponding lines in the pseudo code representation in the appendix. Additionally, each rule description contains a simple application sample. These application samples show fragments of sample sequences before and after rule application. The corresponding figures include those statements that are affected by the rule action. Finally, we argue for the correctness of the rules. For this purpose, we translate the INSERT statements of the original sequence that are affected by the rule condition and the rule action into trees consisting of relational algebra operations (see Section 3.2 for the definitions of the relational algebra operations as well as for a description of the mapping from SQL to relational algebra). Then, we transform these trees with equivalence rules into trees that correspond to the INSERT statements of the rewritten sequence. Mostly, we apply equivalence rules that are well-known from database-related literature such as rules that push operations downwards or upwards within the tree of algebraic operations. Furthermore, we also apply some equivalence rules known from *generalized projections* [GHQ95]. The rest of the transformations is mostly based on the cascading renaming of attributes within the trees or on the substitution of attributes, arithmetic expressions, and correlation names.

In the following, according to the specification in Appendix A, we presume that the body of each INSERT statement is always an SQL query which does not contain any subqueries. Thus, we briefly say *'the SELECT clause s of INSERT statement i'* instead of *'the SELECT clause s of the SQL query which is the body of INSERT statement i'*.

4.1.1 Class-1 Rules

All class-1 rules presented in the following require that the FROM clauses of the INSERT statements of the two target queries match. This is no trivial task because the same table can be referenced multiple times within a query using different correlation names to tell them apart and two semantically equivalent queries can use different correlation names for the same tables. Hence, there exist several valid mappings between the correlation names of the FROM clauses of two INSERT statements where valid means that the mapping is bijective and each correlation name in one FROM clause refers to the same table as the correlation name it is mapped to in the other FROM clause. Thus, the rule condition has to search for a mapping in the set of valid mappings under which the constraints specified in the rule condition evaluate to *true*.

The rule conditions specify that the INSERT statements of two target queries have to be equivalent or different in certain clauses and expressions. In this case, equivalence means that two clauses or two expressions are semantically equivalent, not syntactically. Thus, we have to consider duplicate arithmetic expressions in the expression lists and duplicate comparison expressions in the logical expressions. This is, when an arithmetic expression in one expression list matches an arithmetic expression in the other expression list, also all duplicates of those arithmetic expressions match. The same holds for comparison

expressions in logical expressions. We also have to take commutativity and associativity of certain arithmetic operations into account when matching arithmetic expressions within expression lists and we have to take symmetry of certain comparison operations into account when matching comparison expressions within logical expressions. Moreover, when a certain clause does not exist in two INSERT statements, we call these INSERT statements to be equivalent in this certain clause.

We have to determine the data type of new attributes when creating an attribute definition that should be added to a CREATE TABLE statement. The data type can be retrieved by analyzing the expression in the SELECT clause in the INSERT statement that corresponds to the attribute definition. However, this sometimes requires to know the data type of some base table attributes. This meta data has to be provided to the CGO optimizer by the user when necessary. Otherwise, similar to the cost-based optimizer (see Chapter 5), a component has to be added to the heuristic optimizer that allows the retrieval of this meta data from the underlying database system.

4.1.1.1 MergeSelect Rule

The *MergeSelect* rule merges two queries where the bodies of their INSERT statements at most differ in the SELECT clause. It adds the arithmetic expressions in the SELECT clause in the INSERT statement of the second query to the SELECT clause in the INSERT statement of the first query. Accordingly, it adds the attribute definitions in the CREATE TABLE statement of the second query to the CREATE TABLE statement of the first query. Afterwards, it removes the second query from the query dependency graph and adjusts all queries that depend on the target table of the second query.

Application Sample Figure 4.2 shows a fragment of a sample sequence before and after applying the *MergeSelect* rule. The INSERT statements in the original sequence that insert tuples into table *t1* and table *t2* both access base table *lineitem*, group the tuples by the supplier key *l_suppkey*, and compute some aggregates. Thus, they are equivalent except for the SELECT clauses. Subsequently, the INSERT statement that inserts tuples into table *t3* accesses table *t1* and the INSERT statement that inserts tuples into table *t4* accesses table *t2*. Thus, query q_1 that corresponds to table *t1* and query q_2 that corresponds to table *t2* satisfy the rule condition of the *MergeSelect* rule. The rule action merges both queries by adding the attribute definitions in the CREATE TABLE statement of query q_2 to the CREATE TABLE statement of query q_1 and by adding the expressions from the SELECT clause in the INSERT statement of query q_2 to the SELECT clause in the INSERT statement of query q_1. In order to avoid name collisions due to using the same attribute names in table *t1* and table *t2*, the *MergeSelect* rule replaces the names of the attributes that have been added to table *t1* by attribute names that are unique within table *t1* and table *t2*. After adapting the attribute references that refer to attributes in table *t2* according to these attribute renamings, the INSERT statements that previously accessed table *t2* can now access table *t1* instead. Hence, the statements that belong to query q_2 can be removed from the sequence.

```
CREATE TABLE t1 (suppkey INTEGER, extendedprice_sum FLOAT);

CREATE TABLE t2 (suppkey INTEGER, extendedprice_avg FLOAT);
...

INSERT INTO t1
   SELECT   l.l_suppkey, SUM(l.l_extendedprice)
   FROM     lineitem l
   GROUP BY l.l_suppkey;

INSERT INTO t2
   SELECT   l.l_suppkey, AVG(l.l_extendedprice)
   FROM     lineitem l
   GROUP BY l.l_suppkey;

INSERT INTO t3
   SELECT   t.suppkey, t.extendedprice_sum
   FROM     t1 t;

INSERT INTO t4
   SELECT   t.suppkey, t.extendedprice_avg
   FROM     t2 t;
...
```

```
CREATE TABLE t1 (suppkey INTEGER, extendedprice_sum FLOAT,
                 t2_suppkey INTEGER, t2_extendedprice_avg FLOAT);
...

INSERT INTO t1
   SELECT   l.l_suppkey, SUM(l.l_extendedprice),
            l.l_suppkey, AVG(l.l_extendedprice)
   FROM     lineitem l
   GROUP BY l.l_suppkey;

INSERT INTO t3
   SELECT   t.suppkey, t.extendedprice_sum
   FROM     t1 t;

INSERT INTO t4
   SELECT   t.t2_suppkey, t.t2_extendedprice_avg
   FROM     t1 t;
...
```

Figure 4.2: Fragment of a sequence before and after applying the *MergeSelect* rule.

Rule Condition The rule condition requires two queries, q_1 and q_2, as input and returns the result of the condition test and a correlation name mapping as output. The result of the condition test is a Boolean value which is *true* when the two queries satisfy the rule condition under some correlation name mapping and *false* otherwise. Both queries have to be intermediate-result queries, because the rule action removes query q_2 from the query dependency graph and adds attributes to the table created by query q_1. Due to the same reason, both queries must not eliminate duplicates in the SELECT clauses of their INSERT statements. Either the INSERT statements of both queries must have a GROUP BY clause or the INSERT statements of both queries must not have a GROUP BY clause. If they have a GROUP BY clause, the two GROUP BY clauses have to be equivalent with respect to the correlation name mapping. If they have no GROUP BY clause and the SELECT clause in the INSERT statement of one query contains aggregate expressions, the SELECT clause in the INSERT statement of the other query also has to contain aggregate expressions. This is due to the fact that, when the SELECT clause in an SQL query consists of aggregate expressions and the query has no GROUP BY clause, the query applies the aggregate functions to the whole input table and the result of the query consists of just a single tuple. Finally, all the other clauses, i.e., the WHERE clause and the HAVING clause, have to be equivalent between the INSERT statements of the two queries with respect to the correlation name mapping.

Rule Action First, the rule action retrieves the statements and the target tables of the two queries, q_1 and q_2, and replaces the correlation names in the INSERT statement of query q_2 according to the given correlation name mapping m (line 1 to 7). Then, it appends the SELECT clause in the INSERT statement of query q_2 to the SELECT clause in the INSERT statement of query q_1 (line 8 to 11). Afterwards, the rule action replaces the attribute names in the attribute definitions in the CREATE TABLE statement of query q_2 by some attribute names that are unique within the target tables of query q_1 and query q_2. It adds these modified attribute definitions to the CREATE TABLE statement of query q_1 (line 12 to 18). In the INSERT statements of the queries that depend on query q_2, the rule action replaces the attribute names in all attribute references that refer to the target table of query q_2 by the new unique attribute names. In the table references in the FROM clauses of these INSERT statements, it replaces the target table of query q_2 by the target table of query q_1 (line 19 to 24). Furthermore, the rule action replaces the sources of all direct query dependencies that have query q_2 as source by query q_1 and removes all direct query dependencies that have query q_2 as destination from the query dependency graph (line 25 to 28). Finally, it removes query q_2 from the query dependency graph (line 29).

Optimization Effect The *MergeSelect* rule identifies and materializes a common subexpression between two INSERT statements by merging these INSERT statements. This can have positive as well as negative effects on performance. A positive effect is that after merging the queries, data access and processing is done once and not twice. This may reduce I/O costs and CPU costs for the merged queries by half. However, merging

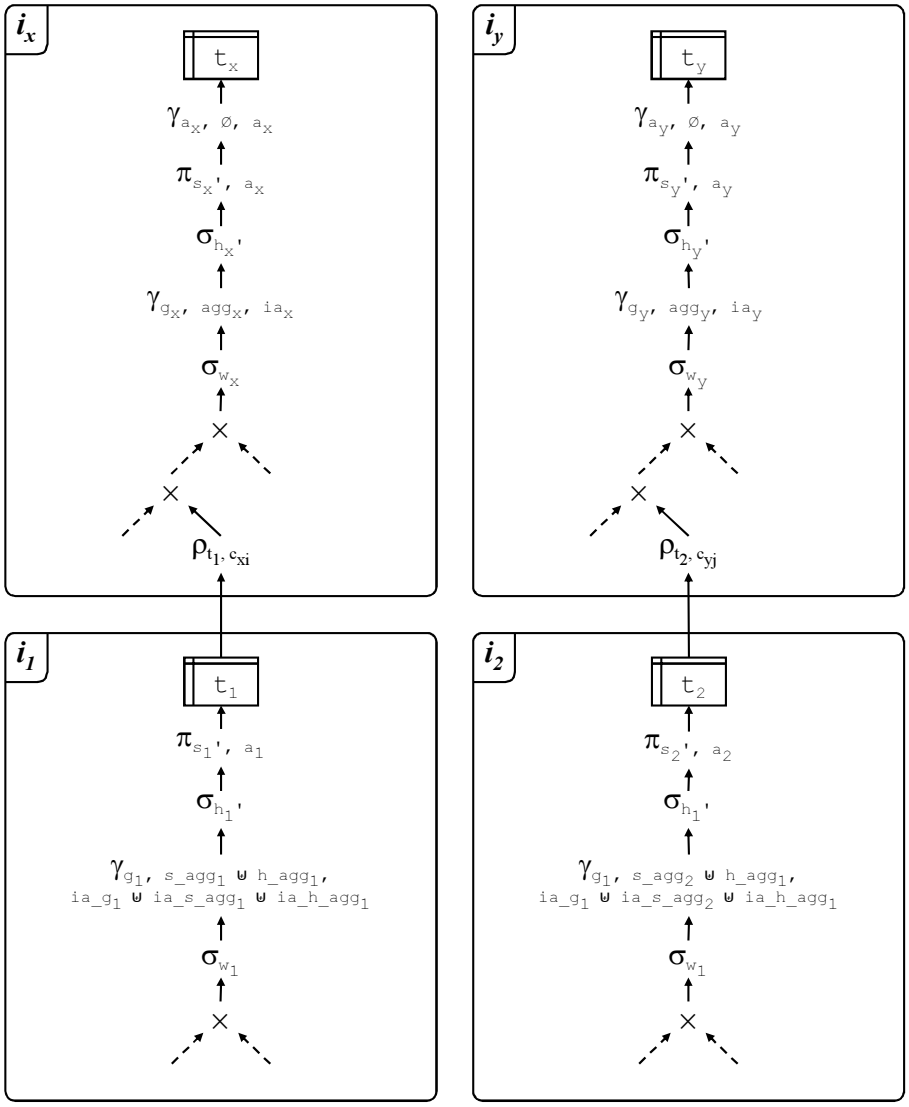


Figure 4.3: Sequence in relational algebra before applying the *MergeSelect* rule.

the INSERT statements results in a target table that has more attributes than one of the target tables of the two original INSERT statements. That is, the target table of the merged INSERT statement requires more space than one of the target tables of the two original INSERT statements due to the increased length of a tuple. This may increase the I/O costs in the INSERT statements that access these target tables. However, when there is an overlapping in the SELECT clauses of the two INSERT statements, subsequently applying the *EliminateRedundantAttributes* rule reduces the number of attributes in the target table of the merged query and therefore counteracts this negative effect. We conclude that the impact of the application of the *MergeSelect* rule is a trade-off between the costs saved by processing a single query instead of two queries and the additional costs for accessing the target table of the merged query which is larger than each of the target tables of the original two queries due to a larger tuple length.

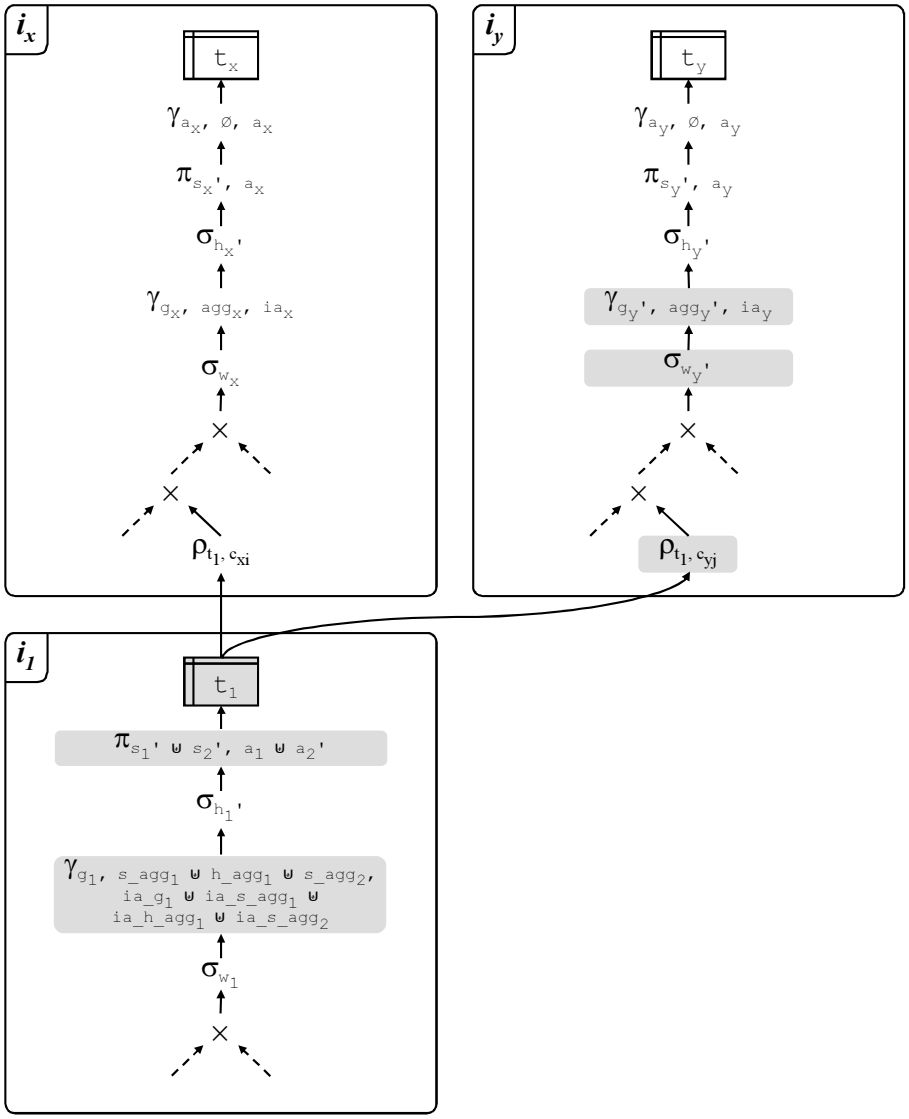


Figure 4.4: Sequence in relational algebra after applying the *MergeSelect* rule.

Correctness Figure 4.3 shows the relational algebra trees that correspond to the INSERT statements i_1 and i_2 of two target queries that satisfy the rule condition and the INSERT statement of a dependent query for each target query. We presume that the table references in INSERT statement i_2 have already been mapped to the corresponding table references in INSERT statement i_1.

Now, we first append the aggregates used in the SELECT clause in INSERT statement i_2 (s_agg_2) to the aggregate list of the grouping operation of INSERT statement i_1 using the same internal attributes ($ia\_s\_agg_2$) for the aggregation results. We also do this the other way round. Then, we insert the elements of the arithmetic expression list of the projection in INSERT statement i_2 at the end of the arithmetic expression list of the projection in INSERT statement i_1. Accordingly, we insert the elements of the arithmetic

expression list of the projection in INSERT statement i_1 at the begin of the arithmetic expression list of the projection in INSERT statement i_2. Due to possible attribute name collisions, we rename the attributes that correspond to the inserted arithmetic expressions. Since these transformations only add new attributes to the intermediate-result tables t_1 and t_2, but do not change the number of tuples in these tables, the modified sequence still computes the same final result.

After these transformations, the INSERT statements i_1 and i_2 compute the same relations. They only differ in the names of the attributes. Therefore, we can substitute all table references on table t_2 by table references on table t_1 and remove INSERT statement i_2. Hence, we also have to adapt the attribute references that originally referred to the attributes a_2 in table t_2 so that they refer to the corresponding attributes a_2' in the modified table t_1. Figure 4.4 shows the result of these transformations. The operations that were affected by modifications are marked with a gray box.

The final result of the relational algebra transformations equals the result of applying the described rule action to a sequence of SQL statements.

4.1.1.2 MergeWhere Rule

The *MergeWhere* rule merges two queries where the bodies of their INSERT statements differ in the WHERE clause and optionally in the SELECT clause. It merges the SELECT clauses of the INSERT statements and the CREATE TABLE statements in the same manner as the *MergeSelect* rule. Additionally, it appends the WHERE clause in the INSERT statement of the second query to the WHERE clause in the INSERT statement of the first query by a logical OR. So, after applying the rule, the content of the target table of the first query is the union of the content of the target tables of the first and the second query before rule application. Thus, the original WHERE clause in the INSERT statement of the first query has to be added to the WHERE clauses in the INSERT statements of all queries that depend on the first query. Accordingly, the same applies to the second query. Finally, the rule action removes the second query from the query dependency graph and adjusts all queries that depend on the target table of the second query.

Application Sample Figure 4.5 shows a fragment of a sample sequence before and after applying the *MergeWhere* rule. The INSERT statements in the original sequence that insert tuples into table *t1* and table *t2* both access base table *lineitem*, but they apply different predicates to the content of this table. Thus, they are equivalent except for the WHERE clauses. The INSERT statement that inserts tuples into table *t3* accesses table *t1* and the INSERT statement that inserts tuples into table *t4* accesses table *t2*. Thus, query q_1 that corresponds to table *t1* and query q_2 that corresponds to table *t2* satisfy the rule condition of the *MergeWhere* rule. The rule action merges both queries by adding the predicate of the WHERE clause in the INSERT statement of query q_2 to the WHERE clause in the INSERT statement of query q_1 via a logical OR. Moreover, it adds the WHERE clause from the INSERT statement of the original query q_1 to the INSERT statement that inserts tuples into table *t3* and it adds the WHERE clause from

the INSERT statement of the original query q_2 to the INSERT statement that inserts tuples into table *t4*. Similar to the *MergeSelect* rule, the rule action of the *MergeWhere* rule also merges the SELECT clauses of the INSERT statements of query q_1 and query q_2 as well as it merges the corresponding CREATE TABLE statements. Additionally, the two attribute references, $l\_returnflag$ and $l\_linestatus$, have to be added to the SELECT clause and the appropriate attribute definitions to the CREATE TABLE statement of query q_1. This is due to the fact that references to these two attributes appear in the predicates that have been added to the INSERT statements of the queries that depend on query q_1 and query q_2, respectively. Hence, the INSERT statements that previously accessed table *t2* can now access table *t1* and the statements that belong to query q_2 can be removed from the sequence.

Rule Condition The rule condition requires two queries, q_1 and q_2, as input and returns the result of the condition test and a correlation name mapping as output. The result of the condition test is a Boolean value which is *true* when the two queries satisfy the rule condition under some correlation name mapping and *false* otherwise. Both queries have to be intermediate-result queries because the rule action removes query q_2 from the query dependency graph and adds attributes to the table created by query q_1. Due to the same reason, both queries must not eliminate duplicates in the SELECT clauses of their INSERT statements. The INSERT statements of the two queries have to differ in the WHERE clauses. Moreover, the INSERT statements of both queries must not do any grouping, i.e., they must not contain a GROUP BY clause and their SELECT clauses must not contain any aggregate expressions. This is due to the fact that the rule action modifies the WHERE clause and the grouping caused by a GROUP BY clause would be applied after applying the predicates of the WHERE clause. Moreover, the expressions in the SELECT clauses of the INSERT statements of the two queries must not contain attribute references in the denominator of a fraction. This is due to the fact that the tuple sets produced by the INSERT statements of the two queries may be disjunctive. Therefore, the expressions in the SELECT clauses of the original INSERT statements may be computed on disjunctive tuple sets, too. However, after the rule application, all expressions of the two SELECT clauses are computed using the union of these tuple sets. Hence, we get into problems, when the value of an attribute in one of those tuple sets is 0 and this attribute is referenced as denominator in an arithmetic expression of the SELECT clause that was previously only applied to the other tuple set. Before rule application, 0 will never appear as attribute value in the denominator, but after rule application it will appear as denominator and it will cause an error when executing the corresponding INSERT statement. As a precaution, we exclude all queries from the application of the *MergeWhere* rule that could potentially cause a division by 0.

Rule Action First, the rule action retrieves the statements and the target tables of the two queries, q_1 and q_2, and replaces the correlation names in the INSERT statement of query q_2 according to the given correlation name mapping m (line 1 to 7). When both INSERT statements contain a WHERE clause, the rule action appends the WHERE

```
CREATE TABLE t1 (orderkey INTEGER, linenumber INTEGER);

CREATE TABLE t2 (orderkey INTEGER, linenumber INTEGER);
...

INSERT INTO t1
   SELECT   l.l_orderkey, l.l_linenumber
   FROM     lineitem l
   WHERE    l.l_returnflag = 'A';

INSERT INTO t2
   SELECT   l.l_orderkey, l.l_linenumber
   FROM     lineitem l
   WHERE    l.l_linestatus = 'O';

INSERT INTO t3
   SELECT   t.orderkey, t.linenumber
   FROM     t1 t;

INSERT INTO t4
   SELECT   t.orderkey, t.linenumber
   FROM     t2 t;
...
```

```
CREATE TABLE t1 (orderkey INTEGER, linenumber INTEGER,
                 t2_orderkey INTEGER, t2_linenumber INTEGER,
                 l_returnflag CHAR(1), l_linestatus CHAR(1));
...

INSERT INTO t1
   SELECT   l.l_orderkey, l.l_linenumber, l.l_orderkey, l.l_linenumber,
            l.l_returnflag, l.l_linestatus
   FROM     lineitem l
   WHERE    l.l_returnflag = 'A' OR l.l_linestatus = 'O';

INSERT INTO t3
   SELECT   t.orderkey, t.linenumber
   FROM     t1 t
   WHERE    t.l_returnflag = 'A';

INSERT INTO t4
   SELECT   t.t2_orderkey, t.t2_linenumber
   FROM     t1 t
   WHERE    t.l_linestatus = 'O';
...
```

Figure 4.5: Fragment of a sequence before and after applying the *MergeWhere* rule.

clause in the INSERT statement of query q_2 to the WHERE clause in the INSERT statement of query q_1 via a logical OR. Otherwise, it removes the WHERE clause from the INSERT statement of query q_1 (line 8 to 13). Then, it appends the SELECT clause in the INSERT statement of query q_2 to the SELECT clause in the INSERT statement of query q_1. Additionally, it appends all attribute references that appear in the WHERE clauses of the INSERT statements of query q_1 and query q_2 (line 14 to 19). Afterwards, the rule action replaces the attribute names in the attribute definitions in the CREATE TABLE statement of query q_2 by some attribute names that are unique within the target tables of query q_1 and query q_2. This is necessary to avoid conflicts due to using the same attribute names in the participating target tables. It adds these modified attribute definitions to the CREATE TABLE statement of query q_1. Additionally, it creates attribute definitions for the attribute references of the WHERE clauses that have been added to the SELECT clause and also appends them to the CREATE TABLE statement of query q_1 (line 20 to 32). Afterwards, the rule action takes the original WHERE clauses and replaces the attribute references in these clauses by attribute references that refer to the corresponding attributes that have been added to the target table of query q_1 (line 33 to 36). It adds the modified WHERE clause in the INSERT statement of query q_1 to the WHERE clauses in the INSERT statements of all queries that depend on query q_1 via a logical AND. This is done for each table reference in the FROM clause that references the target table of query q_1. Similarly, it adds the modified WHERE clause in the INSERT statement of query q_2 to the WHERE clauses in the INSERT statements of all queries that depend on query q_2 via a logical AND. Additionally, in the INSERT statements of the queries that depend on query q_2, the rule action replaces the attribute names in all attribute references that refer to attributes in the target table of query q_2 by the new unique attribute names. In the table references in the FROM clauses of these INSERT statements, it replaces the target table of query q_2 by the target table of query q_1 (line 37 to 49). Furthermore, the rule action replaces the sources of all direct query dependencies that have query q_2 as source by query q_1 and removes all direct query dependencies that have query q_2 as destination from the query dependency graph (line 50 to 53). Finally, it removes query q_2 from the query dependency graph (line 54).

Optimization Effect Similar to the *MergeSelect* rule, the *MergeWhere* rule identifies and materializes a common subexpression between two INSERT statements by merging these INSERT statements with nearly the same positive and negative effects on performance. The target table of the merged query contains the union of the target tables of the two original queries. In the best case, the processing of the two INSERT statements would be expensive and the selectivity of both INSERT statements would be high. A high selectivity would mean that the cardinality of their target tables is low compared to the cardinality of the tables that the two INSERT statements access. Hence, merging the INSERT statements would save costs due to the fact that the input tables have to be accessed only once by the merged query instead of twice by the two original queries. The high selectivity keeps the overhead for accessing the target table of the merged query low in comparison to these cost savings. The overhead emerges from the additional I/O costs due to the fact that the target table of the merged query contains more tuples and more

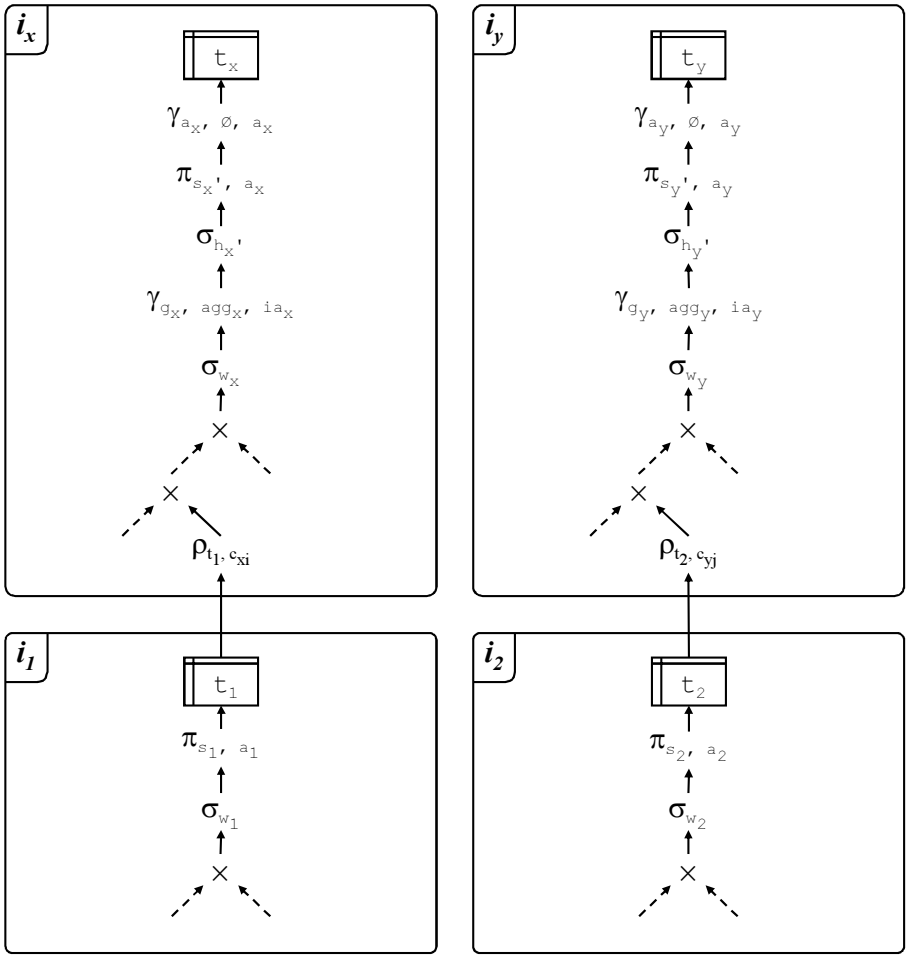


Figure 4.6: Sequence in relational algebra before applying the *MergeWhere* rule.

attributes than each of the target tables of the original queries. Further overhead emerges from the additional CPU costs for processing the predicates that have been added to the INSERT statements that access the target table of the merged query. In the worst case, one of the two original INSERT statements or both original INSERT statements could be processed by just using indexes on the base tables, whereas the merged INSERT statement could not be processed by just using indexes on the base tables (assuming that the underlying database system does not support index ORing). So, the merged INSERT statement had to fetch tuples or it even had to scan the table. (An INSERT statement can be answered by just using indexes when all attributes of the corresponding table that are referenced in the INSERT statement are part of the index key or otherwise included in the index.) Similar to the *MergeSelect* rule, the impact of applying the *MergeWhere* rule is a trade-off between the cost savings due to merging the INSERT statements and the additional overhead that arises from accessing the target table of the merged INSERT statement. Additionally, the impact depends on the access paths that can be used to process the merged INSERT statements and the access paths that can be used to process the two original INSERT statements.

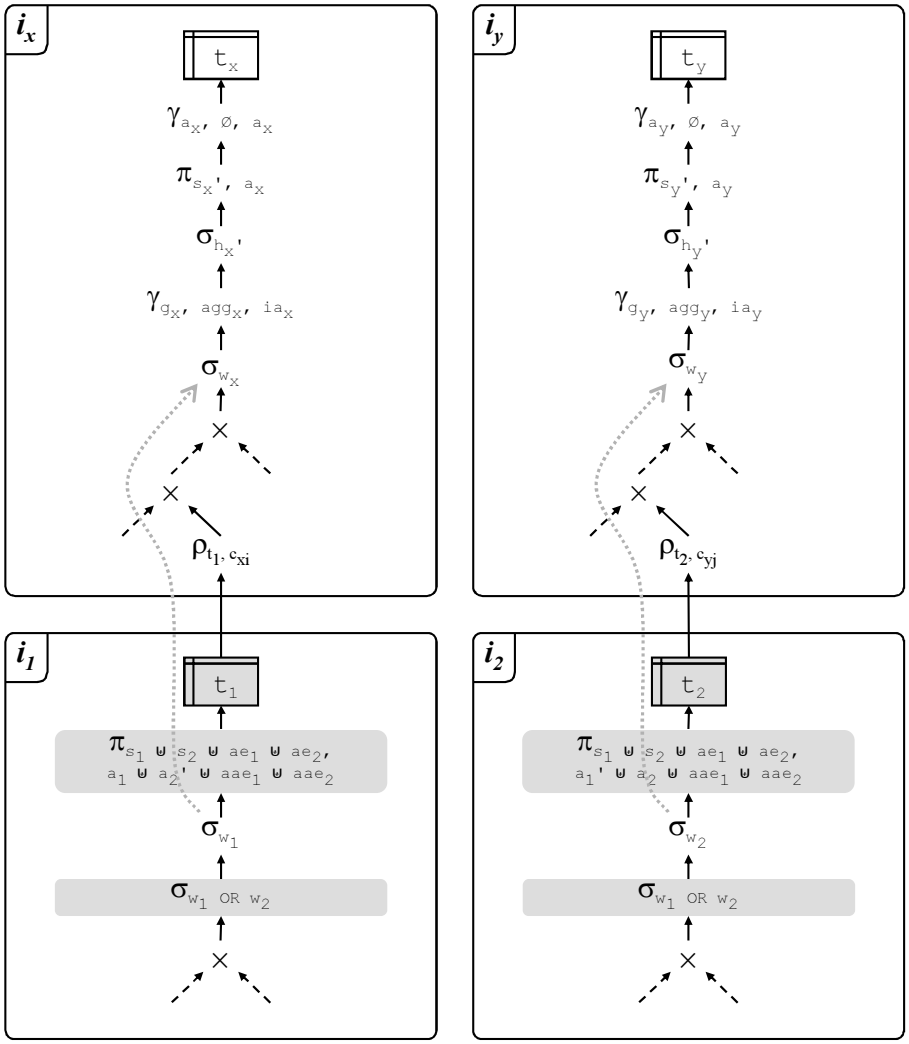


Figure 4.7: Sequence in relational algebra during application of the *MergeWhere* rule.

Correctness Figure 4.6 shows the relational algebra trees that correspond to the INSERT statements i_1 and i_2 of two target queries that satisfy the rule condition and the INSERT statement of a dependent query for each target query. We presume that the table references in INSERT statement i_2 have already been mapped to the corresponding table references in INSERT statement i_1.

We first add an additional selection operation to the INSERT statements i_1 and i_2. We call this selection the *new selection operation* later on, whereas we refer to the selection operations that already exist in the original INSERT statements i_1 and i_2 as the *existing selection operations*. The new selection operation consists of a predicate that combines the predicate of the existing selection operation in INSERT statement i_1 with the predicate of the existing selection operation in INSERT statement i_2 by a logical OR. We insert this new selection operation below the existing selection operations in both INSERT statements. Since the predicate of the new selection operation is weaker than the predicates

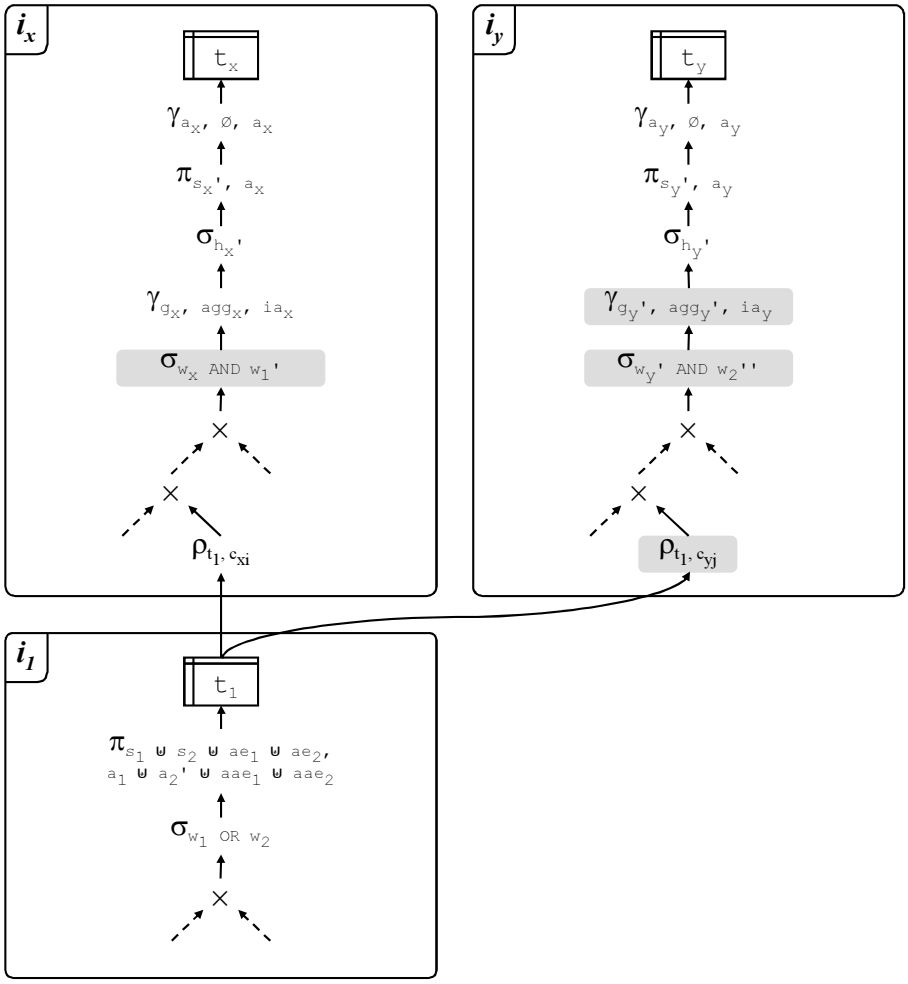


Figure 4.8: Sequence in relational algebra after applying the *MergeWhere* rule.

of the existing selection operations, this modification of the INSERT statements does not change their semantics. Then, we insert the elements of the arithmetic expression list of the projection in INSERT statement i_2 at the end of the arithmetic expression list of the projection in INSERT statement i_1. Accordingly, we insert the elements of the arithmetic expression list of the projection in INSERT statement i_1 at the begin of the arithmetic expression list of the projection in INSERT statement i_2. Due to possible attribute name collisions, we rename the attributes that correspond to the inserted arithmetic expressions. Moreover, we insert all attribute references that appear in the predicate of the new selection operation to the arithmetic expression list of the projection in INSERT statement i_1 and INSERT statement i_2. Since these transformations only add new attributes to the intermediate-result tables t_1 and t_2 but do not change the number of tuples in these tables, the modified sequence still computes the same final result. Figure 4.7 shows the sequence after these transformations where the operations that were affected by modifications are marked with a gray box.

Figure 4.7 also contains a dotted line that shows how we push the existing selection operations in the INSERT statements i_1 and i_2 upwards in the algebra trees in order to merge them with the first selection operation in the INSERT statements that depend on table t_1 and table t_2, respectively. When we push the existing selection operations through the projection operations, we have to replace the attribute references that appear in the predicates of these selection operations by the corresponding attributes in table t_1 and table t_2. However, pushing up through the renaming operation and through the Cartesian product operation is trivial. The complete pushing just changes the content of the intermediate results stored in table t_1 and table t_2, but not the content of table t_x and table t_y, i.e., this pushing does not change the semantics of the sequence.

After these transformations, the INSERT statements i_1 and i_2 compute the same relations. They only differ in the names of the attributes. Therefore, we can substitute all table references on table t_2 by table references on table t_1 and remove INSERT statement i_2. Hence, we also have to adapt the attribute references that originally referred to the attributes a_2 in table t_2 so that they refer to the corresponding attributes a_2' in table t_1. Figure 4.8 shows the result of these transformations. The operations that were affected by modifications are marked with a gray box.

The final result of the relational algebra transformations equals the result of applying the described rule action to a sequence of SQL statements.

4.1.1.3 MergeHaving Rule

The *MergeHaving* rule merges two queries where the bodies of their INSERT statements differ in the HAVING clause and optionally in the SELECT clause. It merges the SELECT clauses of the INSERT statements and the CREATE TABLE statements in the same manner as the *MergeSelect* rule. Additionally, it appends the HAVING clause in the INSERT statement of the second query to the HAVING clause in the INSERT statement of the first query by a logical OR. So, after applying the rule, the content of the target table of the first query is the union of the content of the target tables of the first and the second query before rule application. Thus, the original HAVING clause in the INSERT statement of the first query has to be added to the WHERE clauses in the INSERT statements of all queries that depend on the first query. Accordingly, the same applies to the second query. Finally, the rule action removes the second query from the query dependency graph and adjusts all queries that depend on the target table of this query.

Application Sample Figure 4.9 shows a fragment of a sample sequence before and after applying the *MergeHaving* rule. The INSERT statements in the original sequence that insert tuples into table *t1* and table *t2* both access base table *lineitem* and group by the attribute *l_orderkey*, but they apply different predicates to the groups. Thus, the bodies of the INSERT statements are equivalent except for the HAVING clauses. The INSERT statement that inserts tuples into table *t3* accesses table *t1* and the INSERT statement that inserts tuples into table *t4* accesses table *t2*. Thus, query q_1 that corresponds to table *t1* and query q_2 that corresponds to table *t2* satisfy the rule condition of the *MergeHaving* rule. The rule action merges both queries by adding the predicate of the HAVING clause

in the INSERT statement of query q_2 to the HAVING clause in the INSERT statement of query q_1 via a logical OR. Moreover, it adds the HAVING clause from the INSERT statement of the original query q_1 as WHERE clause to the INSERT statement that inserts tuples into table *t3* and it adds the HAVING clause from the INSERT statement of the original query q_2 as WHERE clause to the INSERT statement that inserts tuples into table *t4*. Similar to the *MergeSelect* rule, the rule action of the *MergeHaving* rule also merges the SELECT clauses of the INSERT statements of query q_1 and query q_2 as well as it merges the corresponding CREATE TABLE statements. Additionally, the two aggregate expressions, $COUNT(*)$ and $SUM(l.l\_extendedprice)$, have to be added to the SELECT clause and the appropriate attribute definitions to the CREATE TABLE statement of q_1. This is due to the fact that these two aggregate expressions appear in the predicates that have been added to the INSERT statements of the queries that depend on query q_1 and query q_2, respectively. Hence, the INSERT statements that previously accessed table *t2* can now access table *t1* and the statements that belong to query q_2 can be removed from the sequence.

Rule Condition The rule condition requires two queries, q_1 and q_2, as input and returns the result of the condition test and a correlation name mapping as output. The result of the condition test is a Boolean value which is *true* when the two queries satisfy the rule condition under some correlation name mapping and *false* otherwise. Both queries have to be intermediate-result queries, because the rule action removes query q_2 and adds attributes to the table created by query q_1. Due to the same reason, both queries must not eliminate duplicates in the SELECT clauses of their INSERT statements. The INSERT statements of both queries must contain a GROUP BY clause and the GROUP BY clauses of both INSERT statements have to be equivalent. Moreover, the WHERE clauses have to be equivalent. However, the INSERT statements have to differ in the HAVING clauses. The expressions in the SELECT clauses of the INSERT statements of the two queries must not contain attribute references in the denominator of a fraction. This is due to the fact that the tuple sets produced by the INSERT statements of the two queries may be disjunctive. Therefore, the expressions in the SELECT clauses of the original INSERT statements may be computed on disjunctive tuple sets, too. However, after rule application, all expressions of the two SELECT clauses are computed using the union of these tuple sets. Hence, we get into problems, when the value of an attribute in one of those tuple sets is 0 and this attribute is referenced as denominator in an arithmetic expression of the SELECT clause that was previously only applied to the other tuple set. Before rule application, 0 will never appear as attribute value in the denominator, but after rule application it will appear as denominator and it will cause an error when executing the corresponding INSERT statement. As a precaution, we exclude all queries from the application of the *MergeHaving* rule that could potentially cause a division by 0.

Rule Action First, the rule action retrieves the statements and the target tables of the two queries, q_1 and q_2, and replaces the correlation names in the INSERT statement of query q_2 according to the correlation name mapping m (line 1 to 7). When both INSERT

```
CREATE TABLE t1 (orderkey INTEGER);

CREATE TABLE t2 (orderkey INTEGER);
...

INSERT INTO t1
   SELECT          l.l_orderkey
   FROM            lineitem l
   GROUP BY        l.l_orderkey
   HAVING          COUNT(*) > 5;

INSERT INTO t2
   SELECT          l.l_orderkey
   FROM            lineitem l
   GROUP BY        l.l_orderkey
   HAVING          SUM(l.l_extendedprice) > 1000;

INSERT INTO t3
   SELECT          t.orderkey
   FROM            t1 t;

INSERT INTO t4
   SELECT          t.orderkey
   FROM            t2 t;
...
```

```
CREATE TABLE t1 (orderkey INTEGER, t2_orderkey INTEGER,
                 t1_aggregate INTEGER, t2_aggregate FLOAT);
...

INSERT INTO t1
   SELECT          l.l_orderkey, l.l_orderkey,
                   COUNT(*), SUM(l.l_extendedprice)
   FROM            lineitem l
   GROUP BY        l.l_orderkey
   HAVING          COUNT(*) > 5 OR SUM(l.l_extendedprice) > 1000;

INSERT INTO t3
   SELECT          t.orderkey
   FROM            t1 t
   WHERE           t1_aggregate > 5;

INSERT INTO t4
   SELECT          t.t2_orderkey
   FROM            t1 t
   WHERE           t2_aggregate > 1000;
...
```

Figure 4.9: Fragment of a sequence before and after applying the *MergeHaving* rule.

statements contain a HAVING clause, the rule action appends the HAVING clause in the INSERT statement of query q_2 to the HAVING clause in the INSERT statement of query q_1 via a logical OR. Otherwise, it removes the HAVING clause from the INSERT statement of query q_1 (line 8 to 13). Then, it appends the SELECT clause in the INSERT statement of query q_2 to the SELECT clause in the INSERT statement of query q_1. Additionally, it appends all aggregate expressions that appear in the HAVING clauses in the INSERT statements of query q_1 and query q_2 (line 14 to 19). Afterwards, the rule action replaces the attribute names in the attribute definitions of the CREATE TABLE statement of query q_2 by some attribute names that are unique within the target tables of query q_1 and query q_2. This is necessary to avoid conflicts due to using the same attribute names in the different target tables. It adds these modified attribute definitions to the CREATE TABLE statement of query q_1. Additionally, it creates attribute definitions for the aggregate expressions that have been added to the SELECT clause and also appends them to the CREATE TABLE statement of query q_1 (line 20 to 32). Afterwards, the rule action takes the original HAVING clauses and replaces the aggregate expressions in these clauses by references to the corresponding attributes that have been added to the target table of query q_1 (line 33 to 36). It adds the modified HAVING clause in the INSERT statement of query q_1 to the WHERE clauses in the INSERT statements of all queries that depend on query q_1 via a logical AND. This is done for each table reference in the FROM clause that references the target table of query q_1. Similarly, it adds the modified HAVING clause in the INSERT statement of query q_2 to the WHERE clauses in the INSERT statements of all queries that depend on query q_2 via a logical AND. Additionally, in the INSERT statements of the queries that depend on query q_2, the rule action replaces the attribute names in all attribute references that refer to attributes in the target table of query q_2 by the new unique attribute names. In the table references in the FROM clauses in these INSERT statements, it replaces the target table of query q_2 by the target table of query q_1 (line 37 to 49). Furthermore, the rule action replaces the sources of all direct query dependencies that have query q_2 as source by query q_1 and removes all direct query dependencies from the query dependency graph that have query q_2 as destination (line 50 to 53). Finally, it removes query q_2 from the query dependency graph (line 54).

Optimization Effect Since the rule action of the *MergeHaving* rule is very similar to the rule action of the *MergeWhere* rule, the optimization effects are also the same.

Correctness Due to the similarity between the *MergeHaving* and *MergeWhere* rule, the transformations that we apply here are similar to those that we apply in the *MergeWhere* rule. Hence, we omit some details here, i.e., the figures only show the original sequence and the sequence that results from applying the rule action (see Figure 4.10 and 4.11).

The main difference to the *MergeWhere* rule is that we push the selection operation upwards which represents the HAVING clause in the SQL query instead of the selection operation which represents the WHERE clause. Thus, we also add an additional selection operation to the INSERT statements i_1 and i_2 that combines the HAVING clauses of both

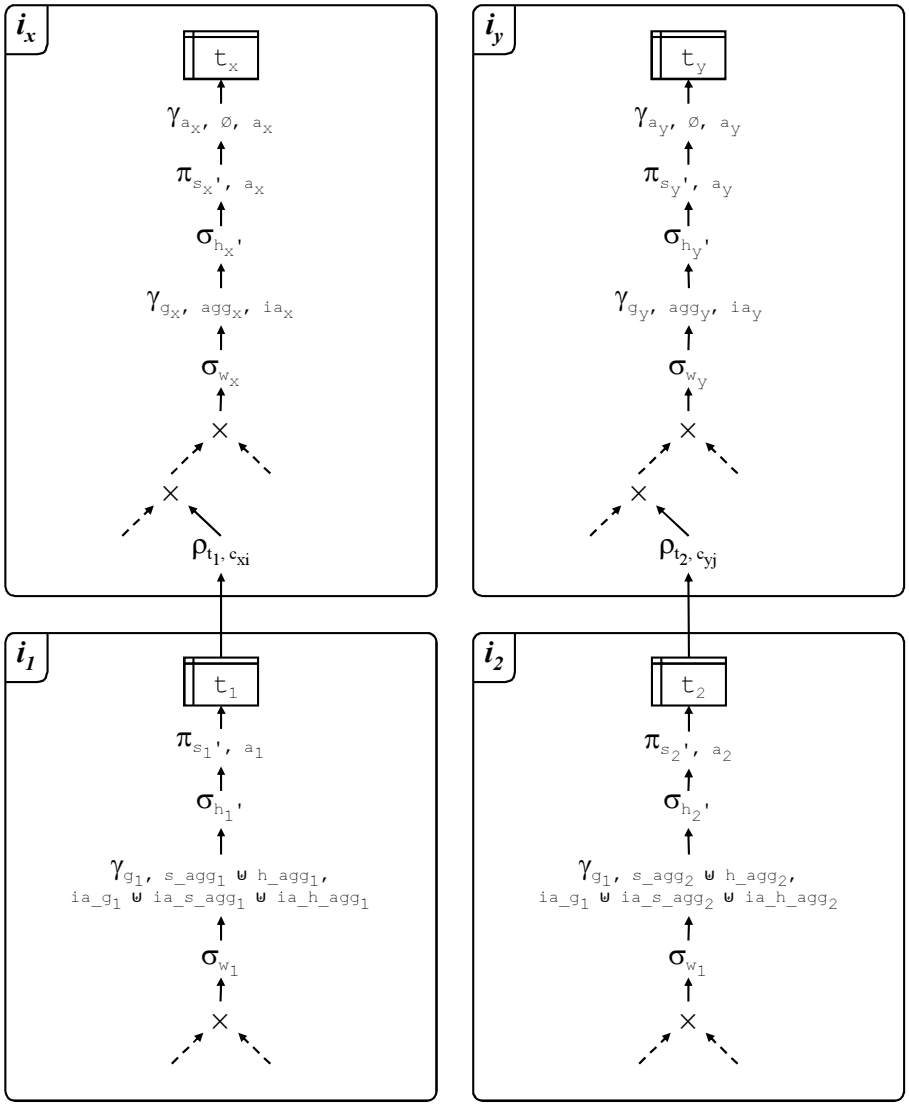


Figure 4.10: Sequence in relational algebra before applying the *MergeHaving* rule.

statements by a logical OR. Accordingly, we have to add the aggregation expressions that appear in the predicates of the new selection operation to the list of aggregation expressions in the grouping operation in INSERT statement i_1 and in INSERT statement i_2. Afterwards, we extend the projections in INSERT statement i_1 and INSERT statement i_2 and push the existing selection operations in the INSERT statements i_1 and i_2 upwards in the algebra trees. We merge them with the first selection operation in the INSERT statements that depend on table t_1 and table t_2, respectively. Finally, we substitute all table references on table t_2 by table references on table t_1, remove INSERT statement i_2 and adapt the corresponding attribute references. The argumentation for correctness of these transformation steps conforms to the argumentation of the *MergeWhere* rule.

The final result of the relational algebra transformations equals the result of applying the described rule action to a sequence of SQL statements.

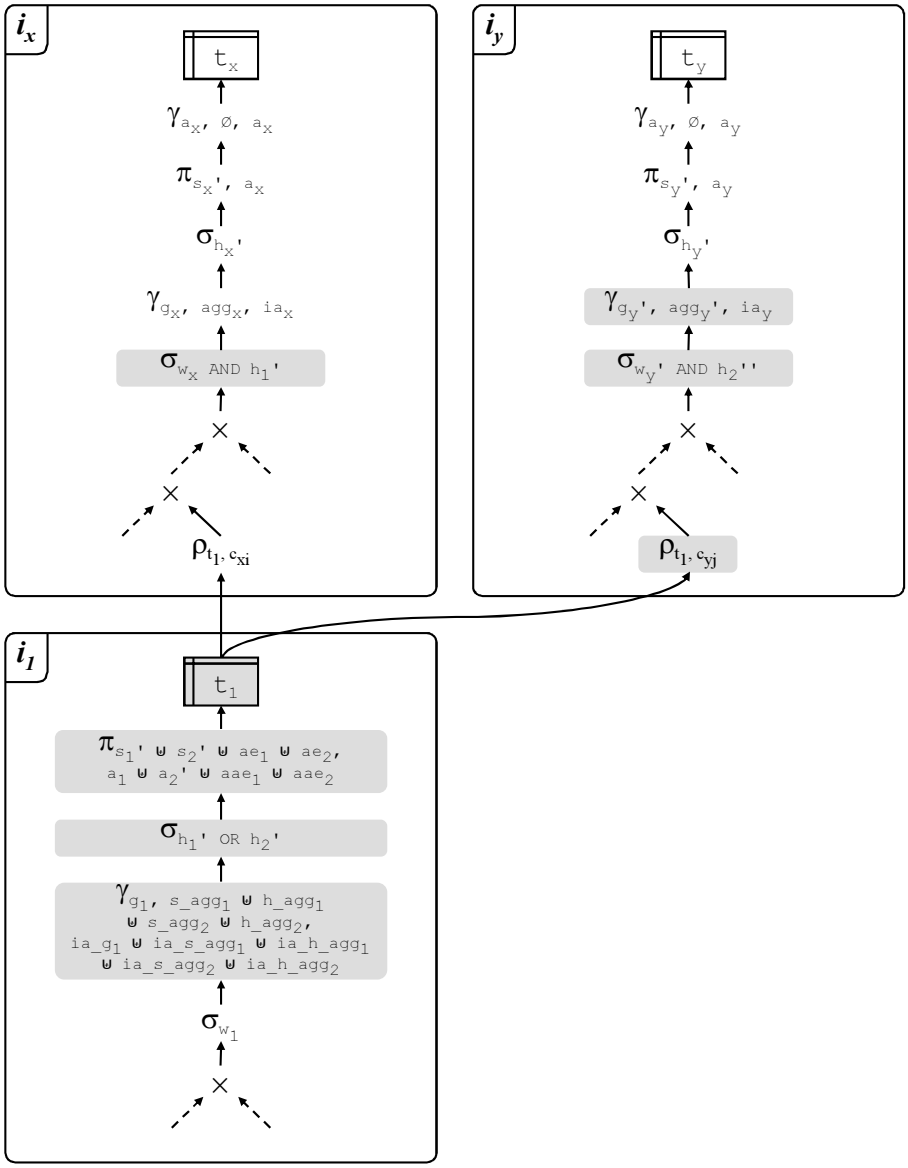


Figure 4.11: Sequence in relational algebra after applying the *MergeHaving* rule.

4.1.1.4 WhereToGroup Rule

Similar to the other *Merge* rules, the *WhereToGroup* rule merges two queries in such a way that, after applying the rule, the content of the target table of the first query is the union of the content of the target tables of the two queries before rule application. For the *WhereToGroup* rule, the INSERT statements have to differ in some special way in the WHERE clause and in the GROUP BY clause and optionally in the SELECT clause. The WHERE clauses in the INSERT statements of the two queries have to overlap except for a single predicate in each WHERE clause. Later on, we denote these two predicates as characteristic predicates. The characteristic predicates compare the same attribute (or arithmetic expression) with different sets of values. Depending on the size of the set

(a single value or several values), the GROUP BY clause in the corresponding INSERT statement has to include the attribute reference (or arithmetic expression) of the characteristic predicate or not. Thus, the rule condition differentiates four different cases where a merge is possible. It merges the SELECT clauses in the INSERT statements and the CREATE TABLE statements in the same manner as the *MergeSelect* rule. Additionally, it merges the two characteristic predicates and replaces the characteristic predicate in the WHERE clause in the first query by the merging result. Furthermore, it also adds a reference to the common attribute (or the arithmetic expression) of the characteristic predicates to the GROUP BY clause in the INSERT statement of the first query. The predicate belonging to the INSERT statement of the first query has to be added to the WHERE clauses in the INSERT statements of all queries that depend on the first query. The same applies to the second query. Finally, the rule action removes the second query from the query dependency graph and adjusts all queries that depend on the target table of the second query.

Application Sample Figure 4.12 shows a fragment of a sample sequence before and after applying the *WhereToGroup* rule. The INSERT statements in the original sequence that insert tuples into table *t1* and table *t2*, both access base table *orders*, derive the order year from the order date, select the orders of a certain year, group them by the customer key *o_custkey*, and compute the sum aggregate over the order prices *o_totalprice*. However, the two INSERT statements differ in the year used in the order selection. Thus, they are equivalent except for the characteristic predicates, $year(o.o\_orderdate) = 1990$ and $year(o.o\_orderdate) = 1991$. The INSERT statement that inserts tuples into table *t3* accesses both tables, *t1* and *t2*, and selects those customers where the total price of orders in 1991 is greater than the total price of orders in 1990. Thus, query q_1 that corresponds to table *t1* and query q_2 that corresponds to table *t2* satisfy the rule condition of the *WhereToGroup* rule. (This scenario complies with case 1 of the rule condition.) The rule action merges both queries by merging their characteristic predicates which results in the new predicate $year(o.o\_orderdate)\ IN\ (1990, 1991)$. Similar to the *MergeSelect* rule, the rule action of the *WhereToGroup* rule also merges the SELECT clauses of the INSERT statements of query q_1 and query q_2 as well as it merges the corresponding CREATE TABLE statements. Additionally, the arithmetic expression of the characteristic predicates $year(o.o\_orderdate)$ is being added to the SELECT clause and an appropriate attribute definition is being added to the CREATE TABLE statement. Furthermore, the rule action adds the characteristic predicates from the INSERT statements of the original queries, q_1 and q_2, to the WHERE clause in the INSERT statement that inserts tuples into table *t3*. Thus, the arithmetic expression in the characteristic predicates has to be replaced by the corresponding attribute that has just been added. Hence, the INSERT statement that previously accessed table *t2* can now access table *t1* and the statements that belong to query q_2 can be removed from the sequence.

Rule Condition The rule condition requires two queries, q_1 and q_2, as input and returns the result of the condition test, a correlation name mapping, and the two characteristic

```
CREATE TABLE t1 (custkey INTEGER, totalprice FLOAT);

CREATE TABLE t2 (custkey INTEGER, totalprice FLOAT);
...

INSERT INTO t1
   SELECT          o.o_custkey, SUM(o.o_totalprice)
   FROM            orders o
   WHERE           year(o.o_orderdate) = 1990
   GROUP BY        o.o_custkey;

INSERT INTO t2
   SELECT          o.o_custkey, SUM(o.o_totalprice)
   FROM            orders o
   WHERE           year(o.o_orderdate) = 1991
   GROUP BY        o.o_custkey;

INSERT INTO t3
   SELECT          t1.custkey
   FROM            t1 t1, t2 t2
   WHERE           t1.custkey = t2.custkey AND
                   t2.totalprice > t1.totalprice;
...
```

```
CREATE TABLE t1 (custkey INTEGER, totalprice FLOAT,
                 t2_custkey INTEGER, t2_totalprice FLOAT,
                 orderyear INTEGER);
...

INSERT INTO t1
   SELECT          o.o_custkey, SUM(o.o_totalprice),
                   o.o_custkey, SUM(o.o_totalprice), orderyear
   FROM            orders o
   WHERE           year(o.o_orderdate) IN (1990, 1991)
   GROUP BY        o.o_custkey, year(o.o_orderdate);

INSERT INTO t3
   SELECT          t1.custkey
   FROM            t1 t1, t1 t2
   WHERE           t1.custkey = t2.custkey AND
                   t2.t2_totalprice > t1.totalprice AND
                   t1.orderyear = 1990 AND t2.orderyear = 1991;
...
```

Figure 4.12: Fragment of a sequence before and after applying the *WhereToGroup* rule.

predicates as output. The result of the condition test is a Boolean value which is *true* when the two queries satisfy the rule condition under some correlation name mapping and *false* otherwise. Both queries have to be intermediate-result queries because the rule action removes query q_2 and adds attributes to the table created by query q_1. Due to the same reason, both queries must not eliminate duplicates in the SELECT clauses of their INSERT statements. Furthermore, the expressions in the SELECT clauses of the INSERT statements of both queries must not contain attribute references in the denominator of a fraction. This is due to the fact that the tuple sets produced by the INSERT statements of the two queries are disjunctive (see rule condition of the *MergeWhere* rule or *MergeHaving* rule for details). The INSERT statements have to be equivalent in their HAVING clauses and they both have to do some grouping, i.e., they have to contain a GROUP BY clause or their SELECT clause has to consist of aggregate expressions. The WHERE clauses in the INSERT statements of the two queries have to be equivalent except for a single predicate in each WHERE clause. Now, we can differentiate four cases:

1. The *WhereToGroup* rule has not yet been applied to any of the two queries. In this case, both characteristic predicates compare an arithmetic expression with a single constant value. The arithmetic expression is the same in both predicates but the constant value is different. The GROUP BY clauses of the INSERT statements of the queries are equivalent. Furthermore, they both do not include the arithmetic expression of the characteristic predicates.

2. The *WhereToGroup* rule has already been applied to the first query but not to the second query. In this case, the characteristic predicate that belongs to the first query compares an arithmetic expression with a list of constant values (or possibly a single constant value which equals a list that contains just a single element), whereas the characteristic predicate that belongs to the second query compares an arithmetic expression with a single constant value. The arithmetic expression is the same in both predicates. In addition to the grouping expressions in the GROUP BY clause in the INSERT statement of the second query, the GROUP BY clause in the INSERT statement of the first query also contains this arithmetic expression.

3. The *WhereToGroup* rule has already been applied to the second query but not to the first query. It's the same case as the previous one when you interchange the two queries.

4. The *WhereToGroup* rule has already been applied to both queries. In this case, both characteristic predicates compare an arithmetic expression with a list of constant values (or possibly a single constant value which equals a list that contains just a single element). The arithmetic expression is the same in both predicates but the list of constant values differs in at least an element. The GROUP BY clauses of the INSERT statements of the queries are equivalent. Furthermore, they both include the arithmetic expression of the characteristic predicates.

Rule Action First, the rule action retrieves the statements and the target tables of the two queries, q_1 and q_2, and replaces the correlation names in the INSERT statement of query q_2 according to the given correlation name mapping m (line 1 to 7). Then, the rule action extracts the arithmetic expression from the characteristic predicate that belongs to the INSERT statement of query q_1 (line 8 and 9). It appends this arithmetic expression as well as the SELECT clause in the INSERT statement of query q_2 to the SELECT clause in the INSERT statement of query q_1 (line 10 to 13). Afterwards, the rule action replaces the attribute names in the attribute definitions of the CREATE TABLE statement of query q_2 by some attribute names that are unique within the target tables of query q_1 and query q_2. This is necessary to avoid conflicts due to using the same attribute names in the various target tables. The rule action adds these modified attribute definitions to the CREATE TABLE statement of query q_1. Additionally, it creates an attribute definition for the arithmetic expression and also appends this attribute definition to the CREATE TABLE statement of query q_1 (line 14 to 23). Afterwards, the rule action adds the arithmetic expression to the GROUP BY clause in the INSERT statement of query q_1 (line 24 to 26). Then, it merges the two characteristic predicates and replaces the characteristic predicate in the WHERE clause in the INSERT statement of query q_2 by the result of this merge (line 27 to 30). In this case, merging means that the rule condition adds the elements of the list of constant values that belongs to the second characteristic predicate to the list of constant values that belongs to the first characteristic predicate. The rule action then takes the original characteristic predicates and replaces the arithmetic expression in these clauses by the corresponding attribute that has been added to the target table of query q_1 (line 31 to 33). It adds the modified characteristic predicate of the INSERT statement of query q_1 to the WHERE clauses in the INSERT statements of all queries that depend on query q_1 via a logical AND. This is done for each table reference in the FROM clause that references the target table of query q_1. Similarly, it adds the characteristic predicate of the INSERT statement of query q_2 to the WHERE clauses in the INSERT statements of all queries that depend on query q_2 via a logical AND. Additionally, in the INSERT statements of the queries that depend on query q_2, the rule action replaces the attribute names in all attribute references that refer to attributes in the target table of query q_2 by the new unique attribute names. In the table references of the FROM clauses in these INSERT statements, it replaces the target table of query q_2 by the target table of query q_1 (line 34 to 44). Accordingly, the rule action replaces the source of all direct query dependencies that have query q_2 as source by query q_1 and removes all direct query dependencies from the query dependency graph that have query q_2 as destination (line 45 to 48). Finally, it removes query q_2 from the query dependency graph (line 49).

Optimization Effect Similar to the other *Merge* rules, the *WhereToGroup* rule identifies and materializes a common subexpression between two INSERT statements by merging these INSERT statements with nearly the same positive and negative effects on performance. However, the *WhereToGroup* rule offers the greatest optimization potential of all *Merge* rules due to the fact that it saves a potentially expensive grouping operation and due to the fact that it only adds a single but quite simple predicate to the queries that depend on the target tables of the target queries.

Correctness Figure 4.13 shows the relational algebra trees that correspond to the INSERT statements i_1 and i_2 of two target queries that satisfy the rule condition and the INSERT statement of a dependent query for each target query. We presume that the table references in INSERT statement i_2 have already been mapped to the corresponding table references in INSERT statement i_1.

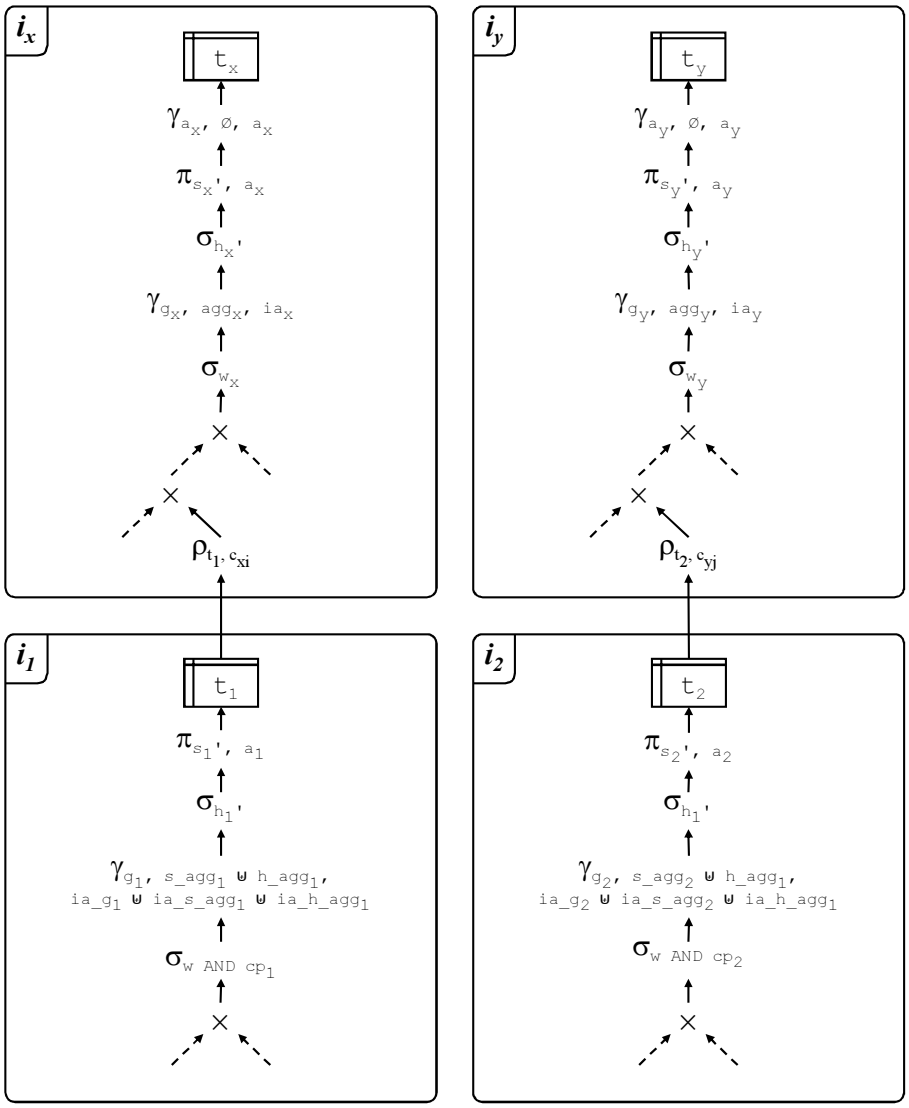


Figure 4.13: Sequence in relational algebra before applying the *WhereToGroup* rule.

Since we want to push the characteristic predicates cp_1 and cp_2 upwards in the algebra trees, we first have to split each of the two selection operations that represent the WHERE clauses into two selection operations such that one of them just contains the characteristic predicate. Then, we add an additional selection operation to the INSERT statements i_1 and i_2. The new selection operation consists of a predicate that combines the characteristic predicate of INSERT statement i_1 with the characteristic predicate of INSERT statement i_2 by a logical OR. We insert this new selection operation below the

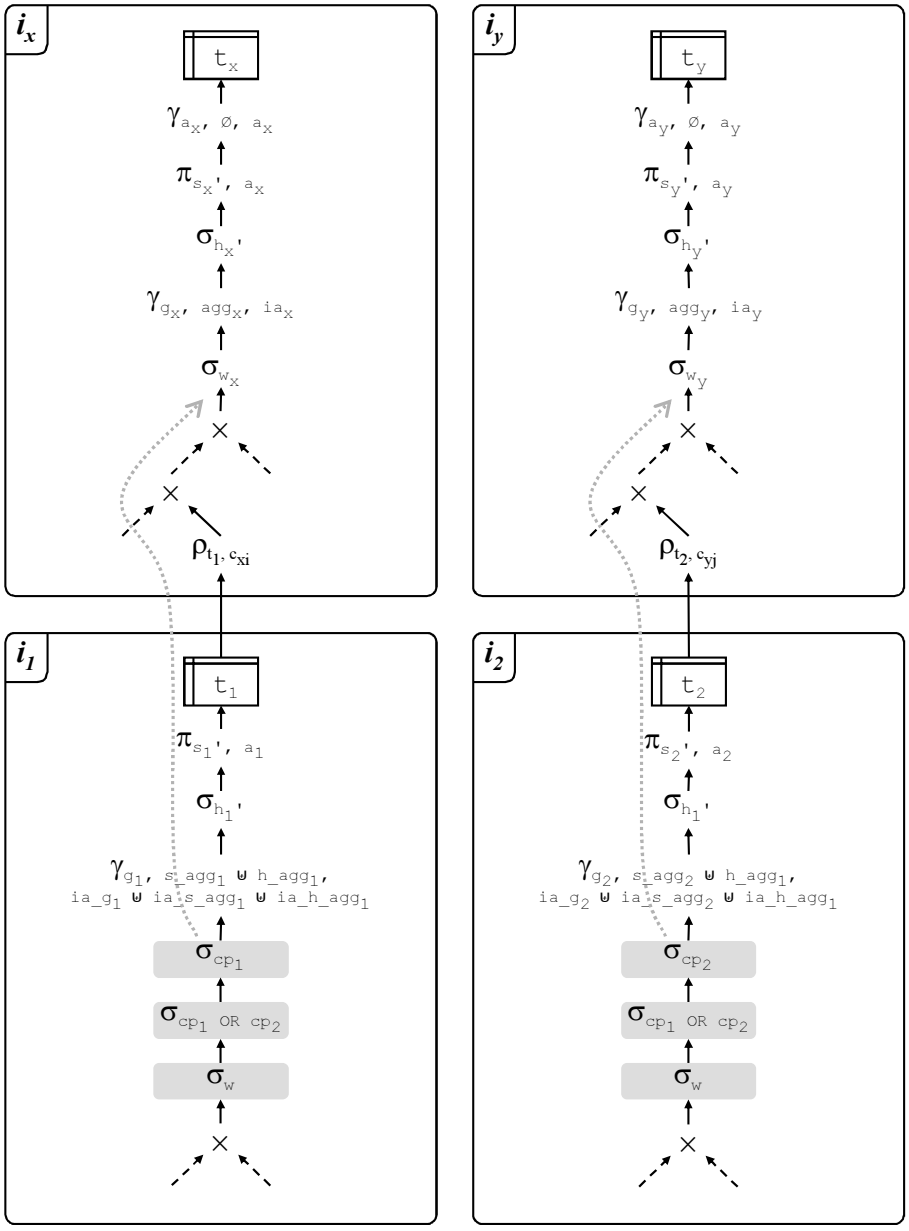


Figure 4.14: Sequence in relational algebra during application of the *WhereToGroup* rule.

existing selection operations. Since the predicate of the new selection operation is weaker than the predicates of the existing selection operations, this modification of the INSERT statements does not change their semantics. Figure 4.14 shows the algebraic operator trees after this transformations. The operations that were affected by modifications are marked with a gray box.

Now, we can push the selection operation in both INSERT statements that consist just of the characteristic predicates upwards. For this purpose, we first push the grouping operations down through these selection operations. This is, we treat the grouping operations as *generalized projections* and make use of rewrite rule *PDRule1* introduced

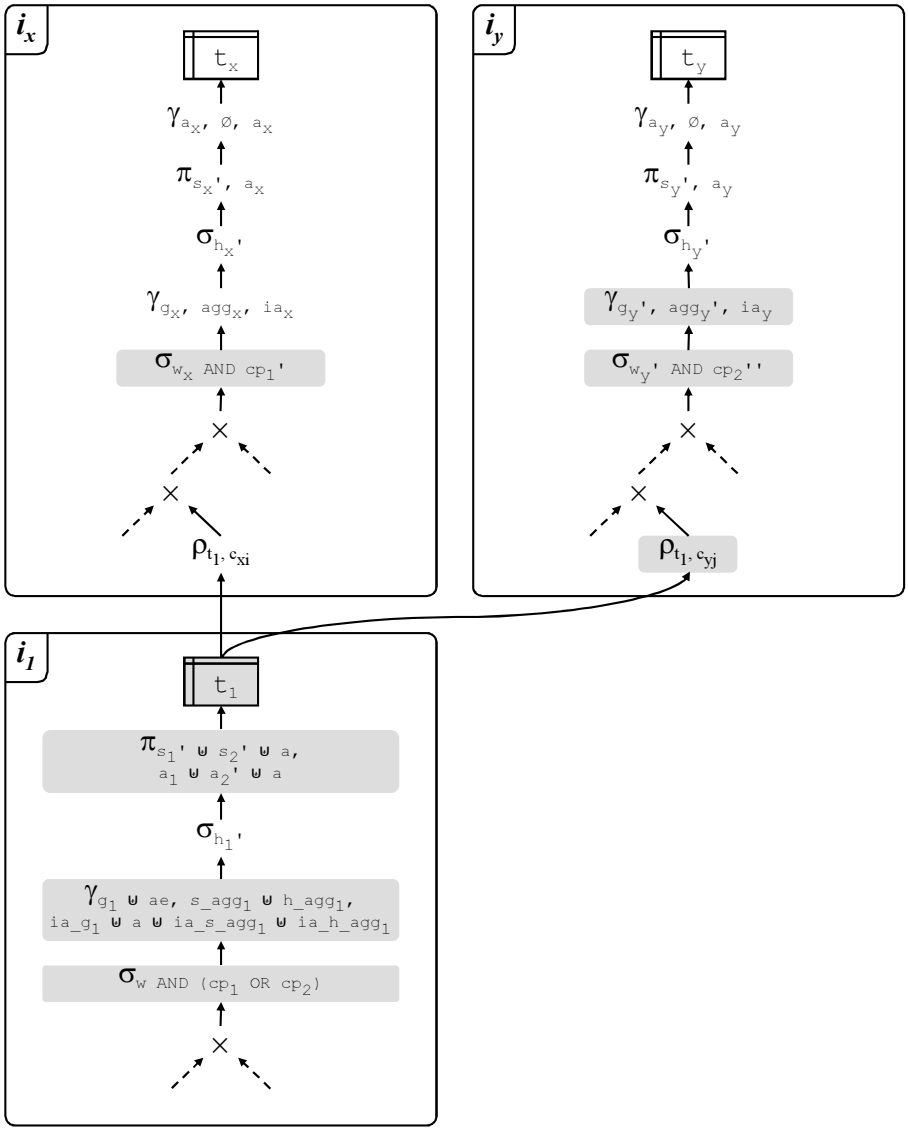


Figure 4.15: Sequence in relational algebra after applying the *WhereToGroup* rule.

in [GHQ95]. In each of the two INSERT statements, i_1 and i_2, this rule adds a new grouping operation prior to the selection operation. This new grouping operation contains all the elements of the grouping operation which follows the selection operation as grouping expressions. Additionally, the list of grouping expressions contains the arithmetic expression which is one of the two operands of the comparison operation in the characteristic predicate. However, rewrite rule *PDRule1* also keeps the original grouping operation above the selection operation. If the list of grouping expressions in the original grouping operation contains the added arithmetic expression, then the original grouping operation can be discarded according to [GHQ95]. This applies for INSERT statement i_1 in case 2 of the rule condition, for INSERT statement i_2 in case 3 of the rule condition, and for both INSERT statements, i_1 and i_2, in case 4 of the rule condition. For all

the other INSERT statements and cases, the list of grouping expressions in the original grouping operation does not contain the mentioned arithmetic expression. However, in this special case, the original grouping operation can be discarded, too, due to the following reason. After applying the selection operation, the result of the arithmetic expression of the characteristic predicate is just a fixed value. Thus, a subsequent application of the original grouping operation does not change the number of groups. However, [GHQ95] does not mention this special case. (Using the notation of [GHQ95], we get the following query equivalence for generalized projections: $\pi_{G,H}(\sigma_{A=C}) = \sigma_{A=C}(\pi_{G,A,H})$ where A is an arithmetic expression and C is a constant value).

When we push the selection operation that contains the characteristic predicate through the projection operation, we have to extend the list of arithmetic expressions of the projection operation by the attribute that we assigned in the grouping operation to the arithmetic expression of the characteristic predicate. When the selection operation that contains the characteristic predicate reaches the first selection operation in the dependent query, we merge the two selection operations into a single selection operation by combining the predicates of these two selection operations via a logical AND.

Then, we insert the elements of the arithmetic expression list of the projection in INSERT statement i_2 at the end of the arithmetic expression list of the projection in INSERT statement i_1. Accordingly, we insert the elements of the arithmetic expression list of the projection in INSERT statement i_1 at the begin of the arithmetic expression list of the projection in INSERT statement i_2. Due to possible attribute name collisions, we rename the attributes in the target table that correspond to the inserted arithmetic expressions. Since these transformations only add new attributes to the intermediate-result tables t_1 and t_2 but do not change the number of tuples in these tables, the modified sequence still computes the same final result.

After these transformations, the INSERT statements i_1 and i_2 compute the same relations. They only differ in the names of the attributes. Therefore, we can substitute all table references to table t_2 by table references to table t_1 and remove INSERT statement i_2. Hence, we also have to adapt the attribute references that originally referred to attributes a_2 in table t_2 so that they refer to the corresponding attributes a_2' in table t_1. Figure 4.15 shows the result of these transformations. The operations that were affected by modifications are marked with a gray box.

The final result of the relational algebra transformations equals the result of applying the described rule action to a sequence of SQL statements.

4.1.2 Class-2 Rules

4.1.2.1 ConcatQueries Rule

The *ConcatQueries* rule merges the target query and the query that depends on the target query. The rule condition differentiates three cases where a merge is possible. The rule action adds the content of the clauses in the INSERT statement of the target query to the clauses in the INSERT statement of the dependent query.

Application Sample Figure 4.16 shows a fragment of a sample sequence before and after applying the *ConcatQueries* rule. The INSERT statement in the original sequence that inserts tuples into table *t1* selects those line items that have been ordered in the year 1990. This intermediate result is only used by the INSERT statement that inserts tuples into table *t2*. This INSERT statement groups the line items into orders and computes the amount of line items for each order whose total (sum of *extendedprice*) is greater than 1000. The rule action merges both queries by adding the WHERE clause in the first INSERT statement to the second INSERT statement and by replacing the table reference to table *t1* in the FROM clause in the second INSERT statement by all table references in the FROM clause in the first INSERT statement. Additionally, the attribute references that refer to attributes of table *t1* have to be replaced by the corresponding arithmetic expressions in the SELECT clause in the first INSERT statement. Finally, the statements that belong to the table filled by the first INSERT statement can be removed from the sequence.

Rule Condition The rule condition requires a query q_1 as input and returns the result of the condition test and a query that depends on query q_1 as output. The result of the condition test is a Boolean value which is *true* when the query satisfies the rule condition and *false* otherwise. Query q_1 must be referenced by exactly a single query q_2 and must not be referenced more than once within the FROM clause in the INSERT statement of this query. Besides, the INSERT statement must not do any implicit type cast, i.e., query q_1 does not satisfy the rule condition if the data types assigned to the attributes in the CREATE TABLE statement cause some kind of rounding or significant changes to the values provided by the SELECT clause in the INSERT statement. Furthermore, in the rule condition, we distinguish between three different cases:

1. The INSERT statement of query q_1 must not do any grouping, i.e., it must not contain a GROUP BY clause and its SELECT clause must not contain any aggregate expressions. Furthermore, it must not eliminate duplicates in its SELECT clause. So, the body of the INSERT statement of query q_1 is a simple select-project-join query that can be merged into query q_2.

2. The INSERT statements of both queries, q_1 and q_2, must not do any grouping, i.e., they must not contain a GROUP BY clause and their SELECT clause must not contain any aggregate expressions. However, the INSERT statements of both queries have to eliminate duplicates in their SELECT clauses. So, the bodies of the INSERT statements of query q_1 and query q_2 are simple select-project-join queries that additionally perform duplicate elimination on their query results.

3. The INSERT statement of query q_2 must not do any grouping, i.e., it must not contain a GROUP BY clause and its SELECT clause must not contain any aggregate expressions. However, the INSERT statement of query q_1 has to contain a GROUP BY clause and query q_2 must not reference any other query than query q_1. So, the body of the INSERT statement of query q_2 is a simple filter query that just applies some predicates to the data set produced by the INSERT statement of query q_1.

```
CREATE TABLE t1 (orderkey INTEGER, extendedprice FLOAT);

CREATE TABLE t2 (orderkey INTEGER, lineitem_count INTEGER);
...

INSERT INTO t1
   SELECT          l.l_orderkey, l.l_extendedprice
   FROM            orders o, lineitem l
   WHERE           o.o_orderkey = l.l_orderkey AND
                   year(o.o_orderdate) = 1990;

INSERT INTO t2
   SELECT          t.orderkey, COUNT(*)
   FROM            t1 t
   GROUP BY        t.orderkey
   HAVING          SUM(t.extendedprice) > 1000;
...
```

```
CREATE TABLE t2 (orderkey INTEGER, lineitem_count INTEGER);
...

INSERT INTO t2
   SELECT          l.l_orderkey, COUNT(*)
   FROM            orders o, lineitem l
   WHERE           o.o_orderkey = l.l_orderkey AND
                   year(o.o_orderdate) = 1990
   GROUP BY        l.l_orderkey
   HAVING          SUM(l.l_extendedprice) > 1000;
...
```

Figure 4.16: Fragment of a sequence before and after applying the *ConcatQueries* rule.

Rule Action First, the rule action retrieves the statements of target query q_1 and query q_2 which depends on query q_1 as well as the clauses of their INSERT statements (line 1 to 6). Afterwards, it creates new correlation names for the table references in the FROM clause in the INSERT statement of query q_1 which are unique within the FROM clauses of the INSERT statements of both queries. It replaces the old correlation names by the new ones in all attribute references in all clauses of the INSERT statement of query q_1 (line 7 to 9). This is necessary to avoid conflicts due to using the same correlation names in both FROM clauses. The rule action replaces all attribute references that refer to attributes in the target table of query q_1 by the corresponding arithmetic expressions in the SELECT clause in the INSERT statement of query q_1. Accordingly, it adds all table references in the FROM clause in the INSERT statement of query q_1 to the FROM clause

in the INSERT statement of query q_2. Moreover, it removes the table reference that refers to the target table of query q_1 from the FROM clause in the INSERT statement of query q_2 (line 10 to 18). Now, we have to distinguish whether the INSERT statement of query q_1 does or does not contain a GROUP BY clause. If it does not contain a GROUP BY clause, then the target query satisfied case 1 or case 2 of the rule condition. In these two cases, the rule condition just has to append the WHERE clause in the INSERT statement of query q_1 to the WHERE clause in the INSERT statement of query q_2 via a logical AND (line 20). If the INSERT statement of query q_1 contains a GROUP BY clause, then the target query satisfied case 3. Thus, the GROUP BY clause in the INSERT statement of query q_1 becomes the GROUP BY clause in the INSERT statement of query q_2 (line 23). Additionally, the rule condition combines the HAVING clause in the INSERT statement of query q_1 with the WHERE clause in the INSERT statement of query q_2 via a logical AND. This combination becomes the HAVING clause in the INSERT statement of query q_2 (line 24 and 25) and the WHERE clause in the INSERT statement of query q_1 becomes the WHERE clause in the INSERT statement of query q_2 (line 25). In all three cases, the rule action removes the direct query dependency between query q_1 and query q_2 from the query dependency graph (line 28 and 29) and sets the destination of all direct query dependencies that have query q_1 as destination to query q_2 (line 30 and 31). Finally, it removes query q_1 from the query dependency graph (line 32).

Optimization Effect Merging two subsequent queries offers the query optimizer of the DBMS executing the query sequence more possibilities for optimization. The query optimizer can choose other join orders or choose other combinations of access paths. Case 2 of the *ConcatQueries* rule also eliminates the grouping operation in the execution plan which corresponds to the duplicate elimination (DISTINCT) in the SELECT clause in the INSERT statement of the target query.

Correctness Again, we distinguish the three cases known from the description of the rule condition.

In the first case, target query q_1 is a simple select-project-join query. Hence, it can be merged with the dependent query q_2 by removing the materialization of the result of INSERT statement i_1 together with the prior projection operation and the subsequent renaming operation (see Figure 4.17). When removing these operations, the attribute references in INSERT statement i_2 that refer to attributes in table t_1 have to be replaced by the corresponding arithmetic expressions. After removing the operations, we have to push the selection operation of INSERT statement i_1 upwards in the algebra tree of INSERT statement i_2 and merge it with the selection operation of INSERT statement i_2 by combining the predicates of both selection operations via a logical AND. Figure 4.18 shows the result of these transformations where the modified operations are marked with a gray box.

In the second case, both INSERT statements, i_1 and i_2, eliminate duplicates before they store their result in table t_1 and table t_2, respectively (see Figure 4.19). To remove the corresponding grouping operation from INSERT statement i_2, we treat the group-

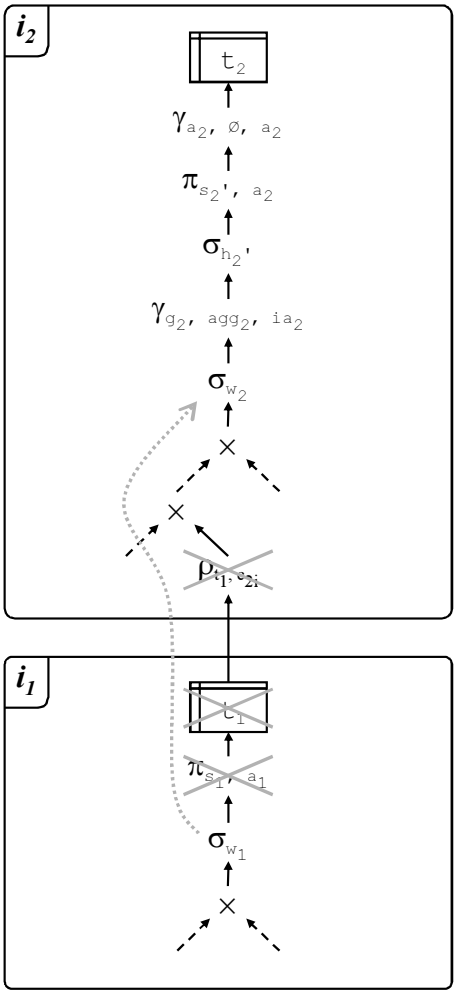


Figure 4.17: Sequence in relational algebra before applying the *ConcatQueries* rule (case 1).

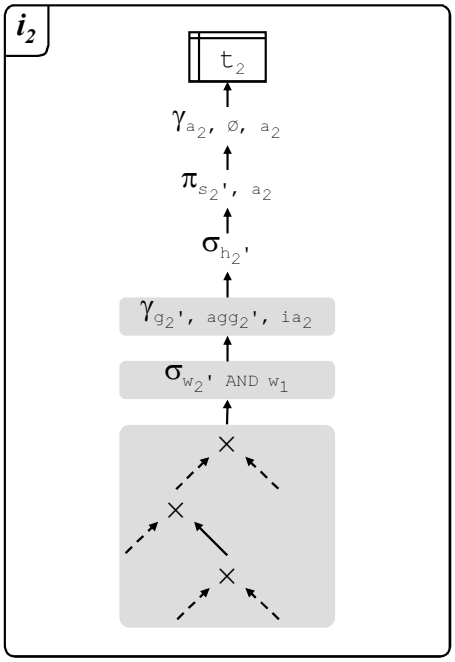


Figure 4.18: Sequence in relational algebra after applying the *ConcatQueries* rule (case 1).

ing operation in INSERT statement i_2 together with the preceding projection operation as a *generalized projection* and make use of the appropriate rewrite rules introduced in [GHQ95]. First, we split this generalized projection into two equivalent ones and push one of them down to the grouping operation in INSERT statement i_1 by the use of *PDRule1* and *PDRule3*. Then, we coalesce the generalized projection with the grouping operation according to *CRule1*, i.e., we discard the grouping operation in INSERT statement i_1. Afterwards, we push the generalized projection upwards again by applying the rules *PURule3* and *PURule1*. Then, we coalesce these two generalized projections by applying *CRule1*, i.e., we discard the generalized projection that has been pushed down and up again. So, the only reason to create this generalized projection is the elimination of the grouping operation in INSERT statement i_1. Now, after the elimination of this grouping operation, the subsequent transformations are equivalent to those of the first case. Figure 4.20 shows the result of these transformations where the modified operations are marked

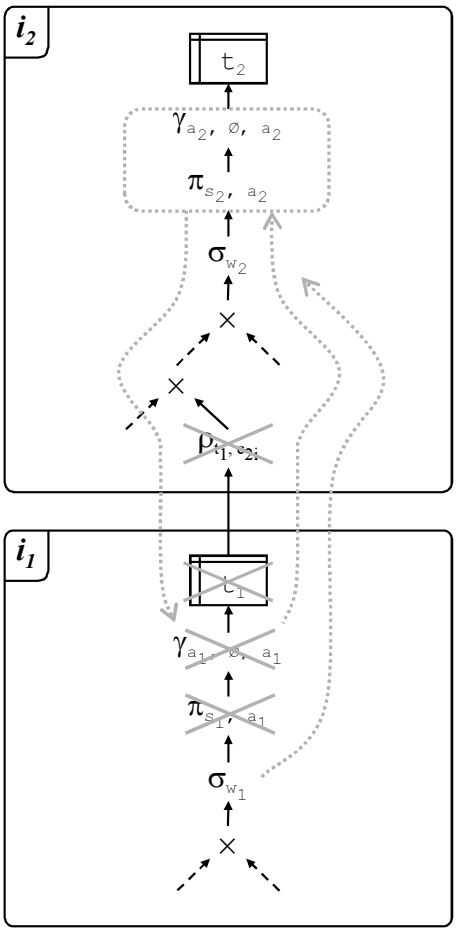


Figure 4.19: Sequence in relational algebra before applying the *ConcatQueries* rule (case 2).

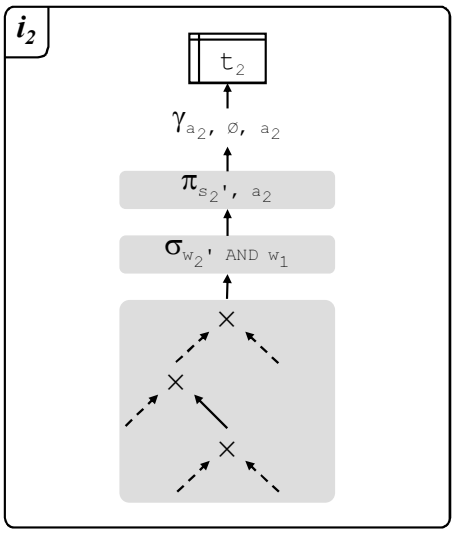


Figure 4.20: Sequence in relational algebra after applying the *ConcatQueries* rule (case 2).

with a gray box. More details about the mentioned rules can be found in [GHQ95].

The third case, is similar to the second case with respect to transformations on the relational algebra representation. Similar to the second case, we eliminate the grouping operation in INSERT statement i_1 and afterwards remove some operations concerning the materialization of the result of INSERT statement i_1. Figure 4.21 and Figure 4.22 show the sequence in relational algebra before and after the transformations, respectively. Modified operations are marked with a gray box.

4.1.2.2 PredicatePushdown Rule

The *PredicatePushdown* rule pushes the predicates that appear in the WHERE clauses in the INSERT statements of the queries that depend on the target query down into the INSERT statement of the target query. For each table reference that refers to the target table, the corresponding INSERT statement has to include the predicates that should be pushed down. If the target query does grouping, the rule action adds these predicates

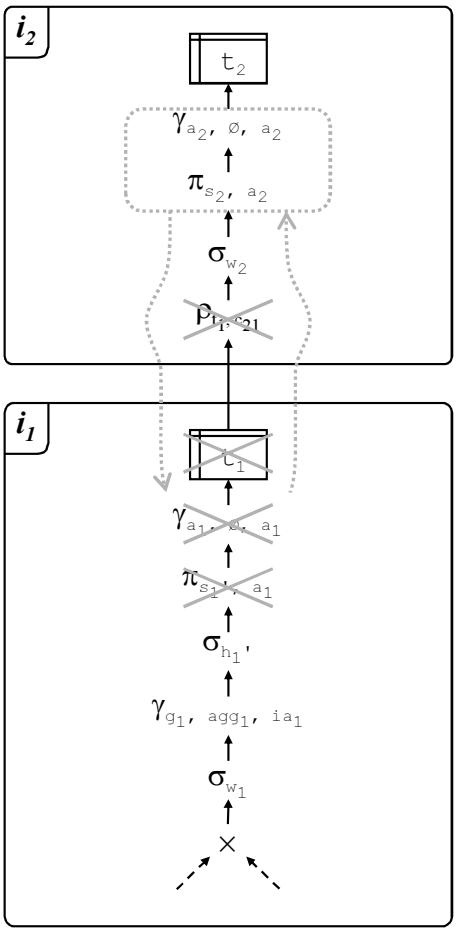


Figure 4.21: Sequence in relational algebra before applying the *ConcatQueries* rule (case 3).

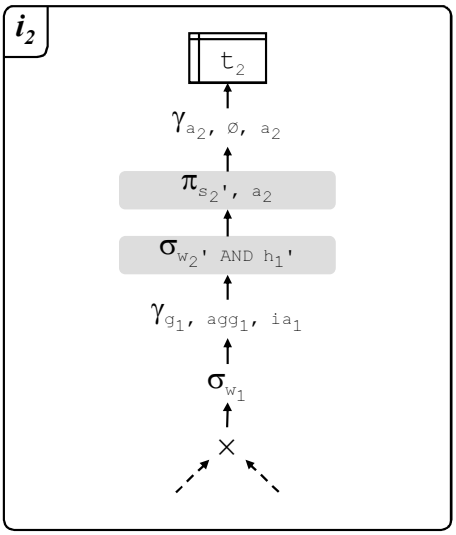


Figure 4.22: Sequence in relational algebra after applying the *ConcatQueries* rule (case 3).

to the HAVING clause; otherwise, the rule action adds these predicates to the WHERE clause.

Application Sample Figure 4.23 shows a fragment of a sample sequence before and after applying the *PredicatePushdown* rule. The INSERT statement in the original sequence that inserts tuples into table *t1* joins all line items with the corresponding orders. This intermediate result is used by the INSERT statement that inserts tuples into table *t2*. This INSERT statement selects those line items that have been ordered in the year 1990, groups them into orders, and computes the amount of line items for each order whose total (sum of *extendedprice*) is greater than 1000. The rule action moves the predicate that selects all line items that have been ordered in the year 1990 from the INSERT statement that inserts tuples into table *t2* to the INSERT statement that inserts tuples into table *t1*.

```
CREATE TABLE t1 (orderkey INTEGER, extendedprice FLOAT,
                 orderyear INTEGER);

CREATE TABLE t2 (orderkey INTEGER, lineitem_count INTEGER);
...

INSERT INTO t1
   SELECT      l.l_orderkey, l.l_extendedprice, year(o.o_orderdate)
   FROM        orders o, lineitem l
   WHERE       o.o_orderkey = l.l_orderkey;

INSERT INTO t2
   SELECT      t.orderkey, COUNT(*)
   FROM        t1 t
   WHERE       orderyear = 1990;
   GROUP BY    t.orderkey
   HAVING      SUM(t.extendedprice) > 1000;
...
```

```
CREATE TABLE t1 (orderkey INTEGER, extendedprice FLOAT,
                 orderyear INTEGER);

CREATE TABLE t2 (orderkey INTEGER, lineitem_count INTEGER);
...

INSERT INTO t1
   SELECT      l.l_orderkey, l.l_extendedprice, year(o.o_orderdate)
   FROM        orders o, lineitem l
   WHERE       o.o_orderkey = l.l_orderkey AND
               year(o.o_orderdate) = 1990;

INSERT INTO t2
   SELECT      t.orderkey, COUNT(*)
   FROM        t1 t
   GROUP BY    t.orderkey
   HAVING      SUM(t.extendedprice) > 1000;
...
```

Figure 4.23: Fragment of a sequence before and after applying the *PredicatePushdown* rule.

Rule Condition The rule condition requires a query q_1 as input and returns the result of the condition test and a list of predicates that can be pushed down into the INSERT statement of query q_1 as output. The result of the condition test is a Boolean value which is *true* when the query satisfies the rule condition and *false* otherwise. Query q_1 must be an intermediate-result query. Otherwise, there wouldn't be any queries that depend on the target query and therefore there wouldn't be any predicates that can be pushed down. For each table reference which refers to the target table of query q_1, the rule condition retrieves a list of those predicates in the WHERE clause in the corresponding query that only contain attributes corresponding to this table reference. Afterwards, the rule condition intersects the predicate lists to get those predicates that are included in all lists, because these predicates can be pushed down into the INSERT statement of query q_1.

Rule Action First, the rule action retrieves the statements and the target table of query q_1 (line 1 to 3). Then, it replaces the attribute references in the given list of pushdown predicates by the corresponding arithmetic expressions in the SELECT clause in the INSERT statement of query q_1 (line 4 to 7). If the INSERT statement of query q_1 has a GROUP BY clause or the SELECT clause in the INSERT statement of query q_1 contains aggregate expressions, the rule action adds the pushdown predicates to the HAVING clause in query q_1; otherwise, it adds the pushdown predicates to the WHERE clause in query q_1 (line 8 to 17). Finally, the rule action removes the pushdown predicates from the WHERE clauses in the INSERT statements of all queries that depend on query q_1 (line 18 to 21).

Optimization Effect A pushdown of predicates offers the query optimizer of the DBMS executing the query sequence more possibilities for optimization within the target query. For example, in some cases the query optimizer can apply such a predicate prior to a join. In the best case, this enables the use of indexes that support the predicate evaluation. Moreover, this may cause the optimizer to pick other join implementations or even another join order.

Correctness Figure 4.24 presents the relational algebra trees that correspond to the INSERT statement i_1 of a target query which satisfies the rule condition and the INSERT statements i_x and i_y of two dependent queries.

The WHERE clauses in the INSERT statements of both dependent queries contain a common predicate (p_x and p_y respectively) which only differs in correlation names used for the tables to which the attribute references refer. Now, we split the selection operations that correspond to the WHERE clauses in such a way that the mentioned predicate is in a separate selection operation. Then, we push down these selection operations. After pushing them through the renaming operations, they contain the same predicate p. Therefore, we can merge these two selection operations and push them further down into the INSERT statement i_1. This is only correct, if such a selection operation exists for each table reference that refers to t_1. When INSERT statement i_1 eliminates duplicates

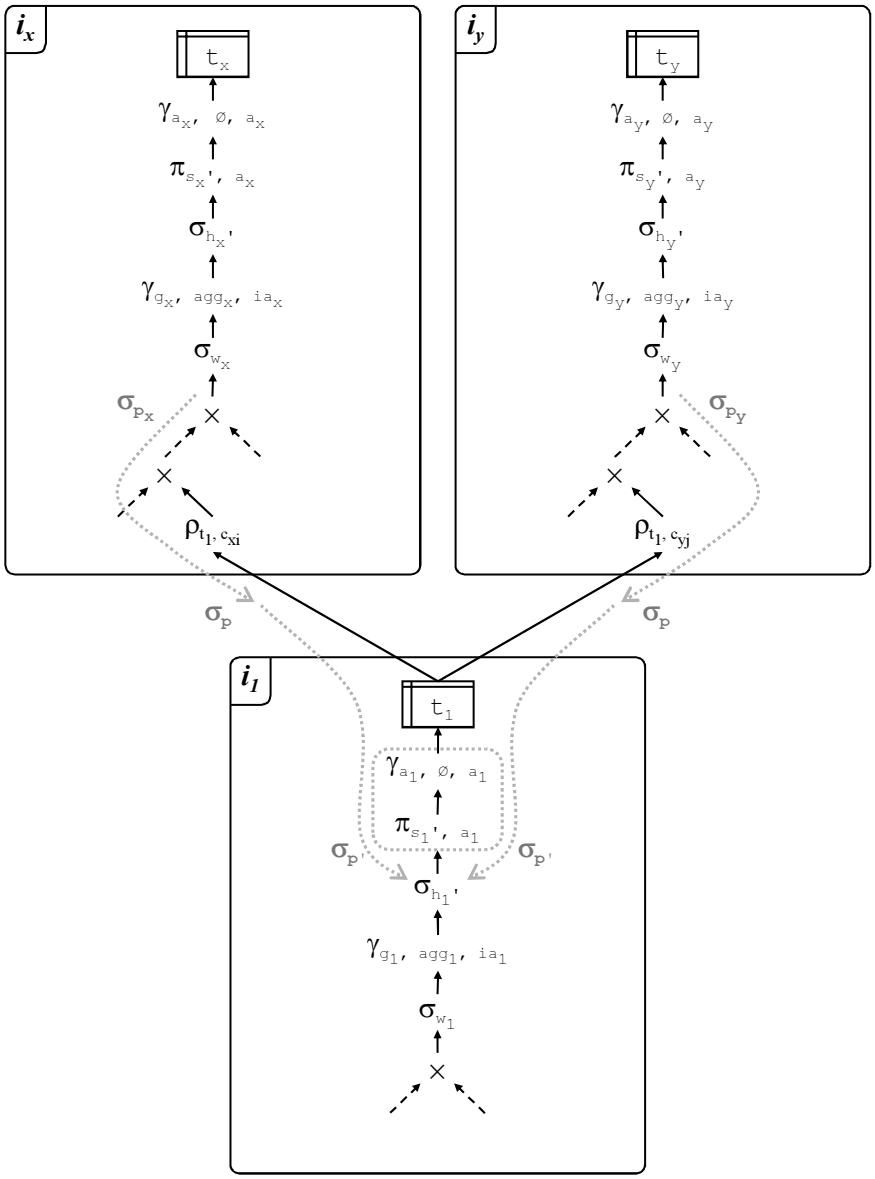


Figure 4.24: Sequence in relational algebra before applying the *PredicatePushdown* rule.

in its SELECT clause (DISTINCT), we treat the combination of grouping operation and projection operation which represents this SELECT clause as a *generalized projection*. Thus, pushing the selection operation, that applies predicate p down through this generalized projection, equals pushing up the generalized projection through the selection operation with respect to the *PURule1* introduced in [GHQ95]. When we push the selection operation through the generalized projection or through the projection operation (if the SELECT clause contains no DISTINCT), we have to replace the attribute references in predicate p by the corresponding arithmetic expressions in the SELECT clause. When the selection operation reaches the first selection operation in INSERT statement i_1, we merge these selection operations by combining their predicates via a logical AND. Figure

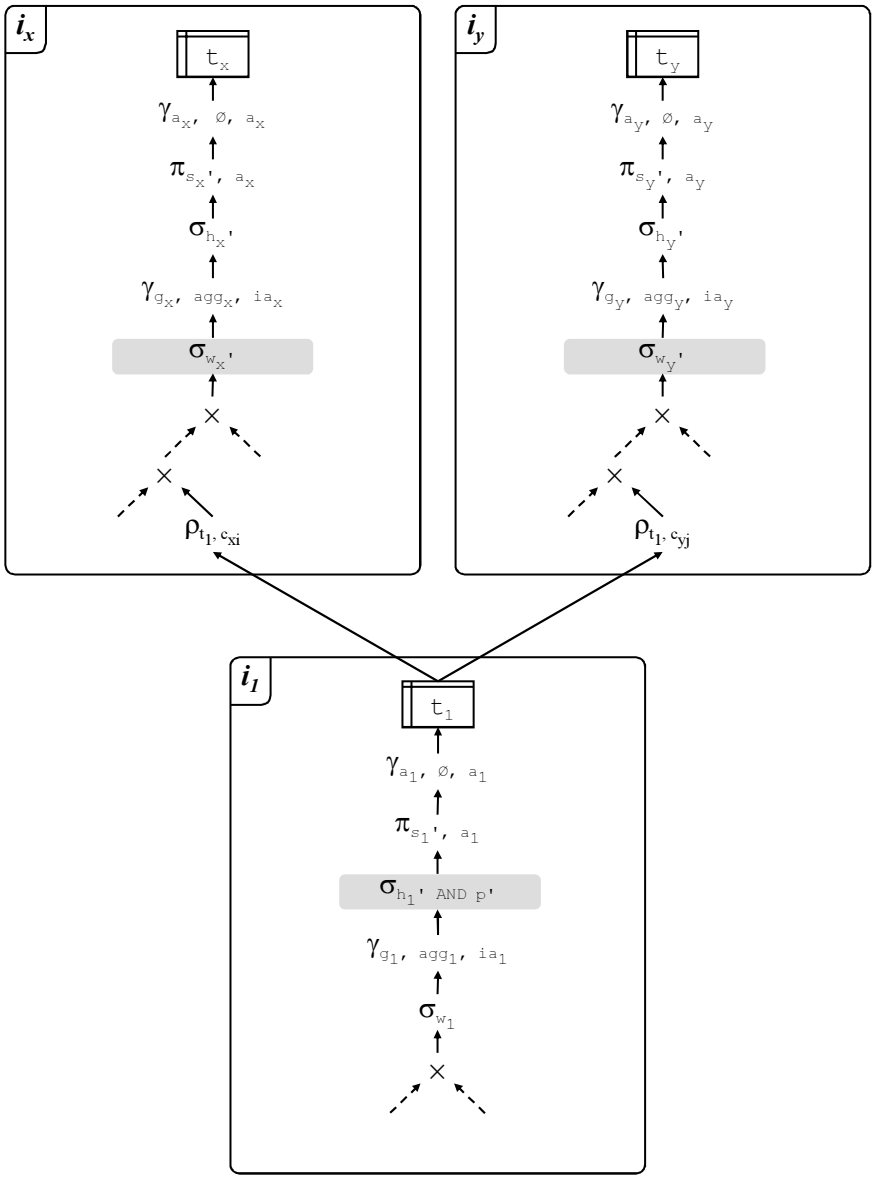


Figure 4.25: Sequence in relational algebra after applying the *PredicatePushdown* rule.

4.25 shows the result of these transformations where the modified operations are marked with a gray box.

The final result of the relational algebra transformations equals the result of applying the described rule action to a sequence of SQL statements.

4.1.2.3 EliminateUnusedAttributes Rule

The *EliminateUnusedAttributes* rule removes those attributes from the target table of the target query that are not referenced within the INSERT statement of any query that depends on the target query.

Application Sample As sample sequence, we reuse the sample sequence used in the application sample for the *PredicatePushdown* rule. However, we do not take the original version but the version after applying the *PredicatePushdown* rule. Figure 4.26 shows a fragment of this sequence before and after applying the *EliminateUnusedAttributes* rule. The first INSERT statement inserts tuples into table *t1* which is afterwards being accessed by the second INSERT statement. Table *t1* contains three attributes *orderkey*, *extendedprice* and *orderyear*. However, only two of them are required by the second INSERT statement. Therefore, the rule action removes attribute *orderyear* from table *t1*, i.e., the rule action removes the associated attribute definition as well as the associated SELECT clause element from the corresponding statements.

Rule Condition The rule condition requires a query q_1 as input and returns the result of the condition test as well as a list of names belonging to attributes that are not used by queries that depend on query q_1 as output. The result of the condition test is a Boolean value which is *true* when the query satisfies the rule condition and *false* otherwise. Query q_1 has to be an intermediate-result query because the rule action removes attributes from the table created by query q_1. Due to the same reason, query q_1 must not eliminate duplicates in the SELECT clause of its INSERT statement. Furthermore, the rule condition checks whether there are attributes in the target table of query q_1 that are not accessed by an INSERT statement of any query that depends on query q_1. Hence, for each query q that depends on query q_1, the rule condition retrieves the names of all attributes in the target table of query q_1 that are referenced by the INSERT statement of query q. Afterwards, the rule condition computes the union of all such sets of attribute names which is again a set of attribute names. Finally, it computes the difference between this set and the set of attribute names of the target table of query q_1. If the result is an non-empty set, query q_1 satisfies the condition and the result of the set difference contains the names of all attributes that can be removed from the target table of query q_1.

Rule Action First, the rule action retrieves the statements of query q_1 (line 1 and 2). Then, it retrieves the attribute definitions of the CREATE TABLE statement of query q_1 as well as the SELECT clause in the INSERT statement of query q_1 (line 3 and 4). Afterwards, it removes all attributes whose names are included in the given list of attribute names from the target table of query q_1, i.e., it removes the corresponding attribute definitions from the CREATE TABLE statement of query q_1 and it removes the corresponding arithmetic expressions from the SELECT clause in the INSERT statement of query q_1 (line 5 to 17).

Optimization Effect The *EliminateUnusedAttributes* rule removes unused attributes from an intermediate-result table. If an attribute value is the result of an arithmetic expression, removing this attribute reduces the computational effort, i.e., it reduces CPU costs of the INSERT statement that fills the intermediate-result table. Moreover, as the *EliminateUnusedAttributes* rule reduces the length of the tuples in an intermediate-result table and therefore the size of this table on disk, the I/O cost for inserting into

```
CREATE TABLE t1 (orderkey INTEGER, extendedprice FLOAT,
                 orderyear INTEGER);

CREATE TABLE t2 (orderkey INTEGER, lineitem_count INTEGER);
...

INSERT INTO t1
  SELECT      l.l_orderkey, l.l_extendedprice, year(o.o_orderdate)
  FROM        orders o, lineitem l
  WHERE       o.o_orderkey = l.l_orderkey AND
              year(o.o_orderdate) = 1990;

INSERT INTO t2
  SELECT      t.orderkey, COUNT(*)
  FROM        t1 t
  GROUP BY    t.orderkey
  HAVING      SUM(t.extendedprice) > 1000;
...
```

```
CREATE TABLE t1 (orderkey INTEGER, extendedprice FLOAT);

CREATE TABLE t2 (orderkey INTEGER, lineitem_count INTEGER);
...

INSERT INTO t1
  SELECT      l.l_orderkey, l.l_extendedprice,
  FROM        orders o, lineitem l
  WHERE       o.o_orderkey = l.l_orderkey AND
              year(o.o_orderdate) = 1990;

INSERT INTO t2
  SELECT      t.orderkey, COUNT(*)
  FROM        t1 t
  GROUP BY    t.orderkey
  HAVING      SUM(t.extendedprice) > 1000;
...
```

Figure 4.26: Fragment of a sequence before and after applying the *EliminateUnused-Attributes* rule.

this table, as well as for reading this table may decrease, too. In combination with the *PredicatePushdown* rule, this rule may lead to significant performance improvements.

4.1.3 Class-3 Rules

4.1.3.1 EliminateRedundantAttributes Rule

The *EliminateRedundantAttributes* rule removes attributes from the target table of the target query whose content equals the content of some other attribute within the target table.

Application Sample Figure 4.27 shows a fragment of a sample sequence before and after applying the *EliminateRedundantAttributes* rule. The INSERT statement in the original sequence that inserts tuples into table *t1* selects those line items that have been ordered in the year 1990. For this purpose, it joins table *orders* with table *lineitem* using the join condition $o.o\_orderkey = l.l\_orderkey$. So the two attributes of the join condition build an equivalence class. Since both attributes appear in the SELECT clause, the rule action removes one of them from the SELECT clause and the corresponding attribute definition from the CREATE TABLE statement. Additionally, the attribute *l.l_extendedprice* appears twice in the SELECT clause. Accordingly, the rule action also removes one appearance from the SELECT clause and the corresponding attribute definition from the CREATE TABLE statement. In the INSERT statements that access table *t1*, the rule action adapts the attribute references that refer to the attributes which have been removed from table *t1*.

Rule Condition The rule condition requires a query q_1 as input and returns the result of the condition test as output. The result of the condition test is a Boolean value which is *true* when the query satisfies the rule condition and *false* otherwise. Query q_1 has to be an intermediate-result query, because the rule action removes attributes from the table created by query q_1. However, query q_1 may eliminate duplicates because the *EliminateRedundantAttributes* rule only removes attributes which are duplicates of other attributes within the same table but keeps at least one of these duplicate attributes. The rule condition has to compute the equivalence classes of arithmetic expressions by analyzing the WHERE clause and the HAVING clause in query q_1. (When two arithmetic expressions are compared via an equality operator, e.g., within a join condition, they fall into the same equivalence class.) Afterwards, for each element in the SELECT clause in the INSERT statement of query q_1, the rule condition checks whether the element is in an equivalence class that contains at least one more element of this SELECT clause. If this is the case, the rule condition checks whether the data types of the corresponding attribute definitions in the CREATE TABLE statement are equivalent. If this is also true for at least one element in the SELECT clause then the rule condition evaluates to *true*, otherwise the rule condition evaluates to *false*.

```
CREATE TABLE t1 (l_orderkey INTEGER, o_orderkey INTEGER,
                 extendedprice FLOAT, lineitemprice FLOAT);
...

INSERT INTO t1
   SELECT        l.l_orderkey, o.o_orderkey,
                 l.l_extendedprice, l.l_extendedprice
   FROM          orders o, lineitem l
   WHERE         o.o_orderkey = l.l_orderkey AND
                 year(o.o_orderdate) = 1990;
...
```

```
CREATE TABLE t1 (l_orderkey INTEGER, extendedprice FLOAT);
...

INSERT INTO t1
   SELECT        l.l_orderkey, l.l_extendedprice
   FROM          orders o, lineitem l
   WHERE         o.o_orderkey = l.l_orderkey AND
                 year(o.o_orderdate) = 1990;
...
```

Figure 4.27: Fragment of a sequence before and after applying the *EliminateRedundantAttributes* rule.

Rule Action First, the rule action retrieves the statements and the target table of query q_1 as well as the attribute definitions of the CREATE TABLE statement of query q_1 and the SELECT clause in the INSERT statement of query q_1 (line 1 to 5). Afterwards, it computes the equivalence classes of arithmetic expressions by analyzing the WHERE clause and the HAVING clause in the same INSERT statement (line 6). Then, in a nested loop it iterates over all possible pairs of arithmetic expressions in the SELECT clause (line 12 and 19). When the two elements of such a pair fall into the same equivalence class and when they have the same data type, the algorithm marks the second element as duplicate of the first element (line 27). Additionally, it stores a mapping that maps the attribute name corresponding to the second element onto the attribute name corresponding to the first element (line 28 to 31). When an element has been marked as duplicate, we do not have to consider it in subsequent iterations (line 13 and 20). After the execution of the nested loop, the rule condition adapts the SELECT clause so that it only contains elements that have not been marked as duplicates (line 18 and 36). Similarly, it adapts the list of attribute definitions in the CREATE TABLE statement of query q_1 (line 17 and 35). Finally, the rule action replaces the attribute names in the attribute references that refer to the target table of query q_1 according to the mappings stored during the execution of the nested loop (line 37 to 41).

Optimization Effect The optimization effect of the *EliminateRedundantAttributes* rule equals the optimization effect of the *EliminateUnusedAttributes* rule since both rules remove attributes from an intermediate-result table. In combination with the *Merge* rules, this rule may lead to significant performance improvements.

4.2 Control Strategy

Since there are no cost estimates available, alternative query sequences cannot be compared. Hence, the control strategy in a heuristic optimizer determines a sequence of consecutive rule applications without backtracking. This means, a control strategy in a heuristic optimizer specifies in which order the query sequence is being traversed and in which order the rules are being applied to a query.

IBM's Starburst [PLH97] makes use of a rule engine that can be adapted for the heuristic CGO optimizer since query transformation in Starburst is based on rewrite rules, too. This rule engine groups the rewrite rules into rule classes and assigns a particular control strategy to each rule class. The rule set contains not just rules that realize query transformations. It also contains rules that force the execution of another rule class and rules that navigate through the internal graph representation of a query. This means, traversal of the internal graph-representation of a query is also realized via rules. The control strategies assigned to rule classes specify how rules in the rule class are selected for application. Control strategies consist of a selection strategy and a budget control strategy. The selection strategy decides which rule to fire. Three different selection algorithms exist:

1. *Prioritized*: A priority is statically associated with each rule in the rule class. The rule engine selects the rule with highest priority from the set of applicable rules in the rule class. This is being repeated in a loop, until no more rule in the rule class can be applied.

2. *Sequential*: An order is statically associated with the rules of the rule class. The rule engine cycles through the rules in the given order and applies those rules whose condition is satisfied. This is being repeated in a loop, until no more rule in the rule class can be applied. Alternatively, the number of iterations can be limited by an additional parameter.

3. *Random*: The rule engine randomly selects a rule from the set of applicable rules in the rule class.

The budget control strategy guarantees that the optimization process terminates and it limits the amount of time available for the optimization process. It assigns a budget to the rule class and each rule application reduces the budget by some amount. Therefore, every rule has some costs associated with it. A rule cannot be executed if the remaining budget is too small. When the budget of the rule class is exhausted or the costs of each rule in the rule class exceed the current budget, the rule engine finishes executing this rule class.

We conclude, that such a rule engine could be used as control strategy component in the heuristic CGO optimizer. Since the rewrite rules in Starburst are unary rules, traversing is somehow more complex for the CGO rule set because the set of CGO rewrite rules consists of unary rules and binary rules. Furthermore, the control strategies of this rule engine could be extended in that way that they also consider characteristics of query sequences, base tables and / or underlying database systems to decide whether a rule should be applied or not. Based on these characteristics, such a control strategy could avoid the application of a rule that might lead to a performance deterioration in the given scenario. Some examples of relevant characteristics of statements are the number of joins or the number and type of predicates. Moreover, each database system has different features and optimization strategies. Hence, the performance gain of some rules is different depending on the database system that executes the query sequence. For example, database systems differ in the maximum number of joins that they are able to optimize exhaustively and to process efficiently as part of a single statement. In CGO, this knowledge could be used to stop merging statements as soon as any further application of rewrite rules would lead to statements that include more than this maximum number of joins. Furthermore, the control strategy could dynamically assign rule priorities based on the same characteristics.

4.3 Prototype

This section concentrates on our prototypic implementation of a heuristic CGO optimizer. The optimizer prototype is embedded into a test and evaluation environment called *CHICAGO* [KS04]. This environment supports the whole process of generating test cases, optimizing statement sequences and running these sequences on a database system. Therefore, besides the CGO optimizer, CHICAGO includes a sequence generator and an execution engine. Additionally, CHICAGO provides the user with a GUI that allows to control the different components of the CHICAGO system. See Figure 4.28 for an architectural overview of the CHICAGO system. The GUI also offers some browser functionality that allows to view all query sequences stored in the CHICAGO system as well as the corresponding runtime statistics if available. It visualizes query sequences as a query dependency graph (see Figure 4.29 for a screenshot).

The test-case generator allows to generate query sequences and the corresponding tables to measure the effectiveness of the rewrite rules. First, via the GUI, the user has to choose one of the rewrite rules that he wants to test. Then, he specifies the rule-specific characteristics of the test case such as predicate selectivity or table cardinality. Based on these characteristics, the sequence generator produces the statements that create and fill the required tables as well as it generates a corresponding query sequence. It immediately executes the statements that create and fill the required tables via the execution engine. The query sequence is stored within the CHICAGO system and can be used as input for the CGO optimizer or can be executed via the execution engine later on.

The execution engine sends the statements of a given query sequence to the target database system by the use of JDBC and returns some runtime statistics. These statistics include the runtime for each statement of the query sequence and the cardinality of the

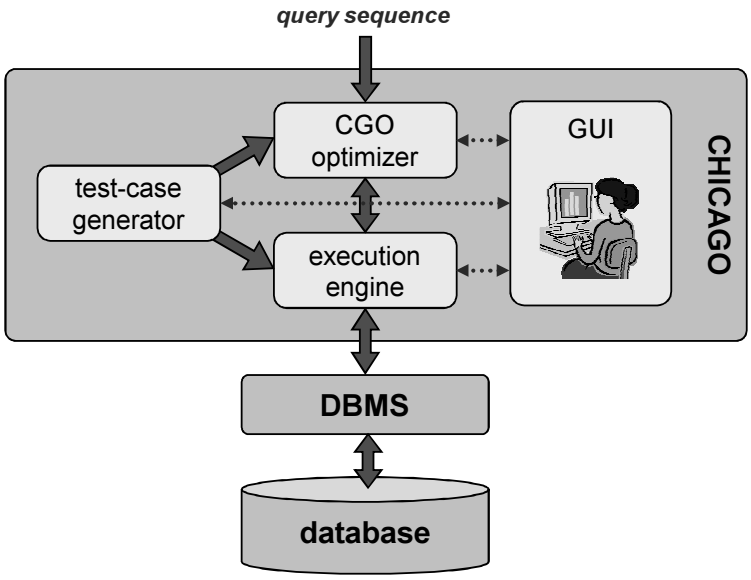


Figure 4.28: Architecture of the CHICAGO system.

tables created and filled by these statements as well as information about errors that occurred during execution. To run a sequence, the user has to specify in the GUI which query sequence should be executed and on which database it should be executed. After execution, the execution engine stores the runtime statistics and table cardinalities in the CHICAGO system where they can be accessed by the user via the GUI.

The CGO optimizer allows to optimize query sequences by applying rewrite rules. The input comes from the test-case generator or from outside the CHICAGO system. In the latter case, the originator of the query sequence is some application developer or some external query-sequence generator such as a commercial OLAP tool. Via the GUI, the user controls the optimization process. This means, the user specifies the rule set as well as the control strategy that should be used for optimization. Additionally, he can view the intermediate result of the optimization process after each rule application and, if desired, he can dynamically change the rule set as well as the control strategy. The query sequences that represent intermediate results or the final result of the optimization process can be directly executed using the execution engine or they can be stored for further usage in the CHICAGO system.

We made use of JavaCC, an open source parser generator for Java, to implement the SQL parser of the CGO optimizer. We realized the SQL retranslator as an XSLT script that transforms the internal XML representation of a query sequence into a sequence of SQL statements again that conforms to our definition of query sequences.

The following subsections concentrate on the internal representation of query sequences used within our optimizer prototype, some implementation issues concerning the rewrite rules of the CGO optimizer, and the control strategy used in the heuristic version of the CGO optimizer.

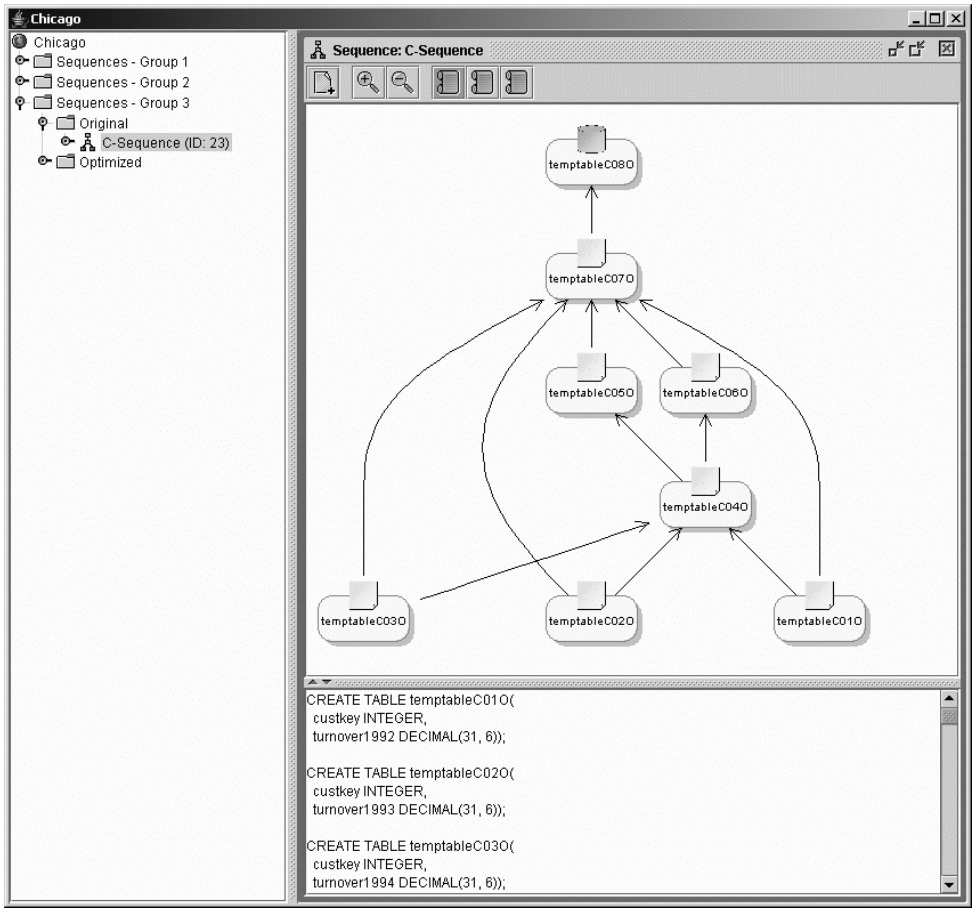


Figure 4.29: Screenshot of CHICAGO's GUI showing the query-sequence browser.

4.3.1 Internal Representation of Query Sequences

For the internal representation of query sequences, we make use of XML. This means, the SQL parser transforms a given query sequence into an XML document. Therefore, we specified our own markup language called *CGO-XML*. We have chosen XML as the basis of our internal representation due to the following reasons:

- *Extensibility*: The internal representation can be easily extended by adding new types of elements, when we want to extend the subset of SQL supported by the CGO optimizer. Moreover, the SQL retranslator can be implemented by an XSLT script, which can be extended when the internal representation is being extended.

- *Tool support*: There is a great variety of tools and libraries for XML regarding programming interfaces, query languages, and transformation languages.

- *Debugging*: The internal representation is easy to debug, because the XML document can typically be dumped to the console as ASCII text.

A DTD of CGO-XML can be found in Appendix B. CGO-XML stores the query sequences in form of a query dependency graph, i.e., it groups the statements that target the same table into queries. The root element in an CGO-XML document is the *sequence* element and its child elements represent the queries of the sequence. As an example for these child elements, Figure B.1 and Figure B.2 in Appendix B show the *query* elements that correspond to the two queries in the second query sequence of Figure 3.1. The *query* element contains a kind of parse tree of the body of the INSERT statement that belongs to the corresponding query. Thus, the children of the *query* element represent the different clauses that the body of the INSERT statement contains. Additionally, the query element contains a *referenced-by* element as first child. This element again contains an element for each query that depends on this query. It represents the set of direct query dependencies that include this query as source. The *select-clause* element not only contains the arithmetic expressions that are part of the SELECT clause in the INSERT statement of the query. It also contains the information that can be obtained from the CREATE TABLE statement of the query such as the name of the attribute assigned to an arithmetic expression within the target table of the query and the data type of this attribute. So, the CREATE TABLE statement is not represented separately by its own element within a CGO-XML document. The same holds for the DROP TABLE statement. When the query is an intermediate-result query, i.e., when the query sequence contains a DROP TABLE statement for the target table of the query, the attribute *result* in the *query* element is set to *no*. When the query is a final-result query, i.e., when the query sequence contains no DROP TABLE statement for the target table of the query, the attribute *result* in the *query* element is set to *yes*.

There are several reasons to represent the body of an INSERT statement by a parse tree instead of a relational algebra tree. One reason is that we formulated the rewrite rules for query sequences on the basis of SQL, i.e., the rewrite rules do not target the operators of relational algebra expressions but the components of SQL statements with respect to the declarative nature of SQL. The other reason is that we focus on a database-independent optimization approach where the result of the transformation process has to be translated back into SQL again. In this context, it is important that retranslation identically reproduces all parts of the original sequence that have not been affected by the rewriting process. This is due to the fact that even little modifications in the SQL statements might result in dramatically different execution plans produced by the optimizer of the target database system. With the use of an algebraic representation, it would be more difficult to achieve this goal. Moreover, for the sake of readability and understanding, it is helpful when the original sequence, which served as input to CGO and the output sequence look similar. So, a database administrator can clearly identify, which parts of the query sequence have not been affected by the application of rewrite rules.

4.3.2 Implementation of the Rewrite Rules

The rule implementations differ from the rule descriptions in Section 4.1 in that way that the rule actions of the implemented rules always eliminate redundant attributes. This means, the application of the rule action of an implemented rule complies with the

application of the rule action as described in Section 4.1 followed by the application of the rule action of the *EliminateRedundantAttributes* rule.

Each rule is implemented by a separate Java class, which is derived from a common super class *Rule*. The *Rule* class provides two abstract methods that can be invoked by the control strategy: a method checking the rule condition and a method realizing the rule action. The condition method returns a context object, if the given target queries satisfy the rule condition. Otherwise, the rule condition returns the Java *null* value. The context object is being used to pass bindings from the condition method to the action method. It stores the actual target queries as well as rule-specific data, which has been computed during the evaluation of the rule condition. Hence, there exists a separate context class for each rewrite rule and all these classes inherit from the same abstract superclass. The rule class itself is stateless, i.e., the condition method and action method do not set attribute values in the rule-class instance and the condition method does not have any side-effects on the internal representation of the given query sequence. Therefore, we have to instantiate each rule class only once and we can use it as often as required.

The implementation makes use of the document object model (DOM) and its Java language bindings to navigate in the internal representation of a query sequence and to modify this internal representation according to the rule action.

The comparison and matching functions used in the rule conditions of class-1 rules can be organized in multiple layers according to the structure of an SQL query and its components. Every layer uses the functionality of the layer below. The top layer which we denoted as the *query layer* matches two FROM clauses and compares all other clauses or a subset of the other clauses. The comparison of the clauses is implemented in the *clause layer*. For every type of clause, there exists a method that compares two clauses of this type taking into account the special characteristics of that clause type. These methods make use of the methods of the *clause element layer* that compares two clause elements. In case of a WHERE or HAVING clause, the *clause element* layer can be hierarchically further divided into the *predicate layer* and the *term layer*.

4.3.3 Control Strategy

In the prototype, we group all implemented rewrite rules into a single class and make use of the prioritized control strategy. This means, we assign a fix priority to each rewrite rule and the rule engine always selects the rule with highest priority from the set of applicable rules as follows. The rule engine cycles through the rewrite rules starting with the rule that has the highest priority. For a unary rule, it iterates over all queries within the given query sequence and checks whether a query satisfies the rule condition. For a binary rule, it enumerates all combinations of two different queries within the given query sequence and checks for each combination whether it satisfies the rule condition. When a query or query combination satisfies the rule condition, the rule engine applies the rule action and starts over again with cycling through the rules starting with the rule that has highest priority. This is being repeated until no more rule can be applied.

Since the application of a rule may disable or enable the applicability of rules on the queries that have been modified by the rule action, we assigned a high priority to those

rules that show the greatest optimization potential and to those rules that enable the application of several other rules. Otherwise, the early application of a less beneficial rule might disable any subsequent application of a more beneficial rule. Therefore, we assigned a high priority to the rules in class 1, a middle priority to the rules in class 2 and a low priority to the rules in class 3. We have chosen this order, because class-1 rules merge similar queries and therefore avoid redundant processing. This results in remarkable performance improvements as our experiments have shown. Furthermore, among the rules in class 2, we assigned the highest priority to the ConcatQueries rule, because, when we apply the ConcatQueries rule to a query, merging the subsequent queries annuls the effects of previous applications of other class-2 rules to this query. This means, applying the PredicatePushdown rule or the EliminateUnusedAttributes rule to a query is not necessary when we apply the ConcatQueries rule afterwards.

We omitted a budget control strategy because the rule set described in this document guarantees that the optimization process terminates after some finite number of rule applications. This is due to the following facts:

- Class-1 rules are binary rules that merge two similar queries into a single query. Therefore, each application of a class-1 rule reduces the number of queries in the sequence by one. Thus, the optimization process stops when there is only one query left.

- The ConcatQueries rule merges two subsequent queries into a single query. Therefore, each application of this rule reduces the number of queries in the sequence by one. Thus, the optimization process stops when there is only one query left.

- The PredicatePushdown rule removes common predicates from the INSERT statements of several queries and adds them to the INSERT statement of the query these queries depend on. This pushdown may cascade in the direction from the final-result query down to the queries that just access base tables but never in the other direction. Thus, the optimization stops when all predicates have been pushed down to the queries that just access base tables.

- The EliminateUnusedAttributes rule and the EliminateRedundantAttributes rule remove attributes from the target table of a query and therefore reduce the number of attributes in the target table. Thus, the optimization process stops when there is only one attribute left in the target table of each query.

We conclude, that there are no rules which add queries to a query sequence and each rule for itself terminates after a finite number of iterations. However, we also have to consider what happens when we combine different rules. Here, we have to consider that class-1 rules not only remove a query from the query sequence but also add predicates to the INSERT statements of the dependent queries of the target queries and attributes to the target table of the merged query. However, that doesn't violate the termination property, because an oscillating rule application is not possible due to the following reasons. The predicates that class-1 rules add to the INSERT statements of the dependent queries can not be pushed down by the PredicatePushdown rule. This is due to the fact, that a class-1 rule always

adds a different predicate to each of the dependent queries. Moreover, the attributes that class-1 rules add to the target table of the target query can possibly be removed by the EliminateUnusedAttributes rule and by the EliminateRedundantAttributes rule.

4.4 Experiments

In this section, we discuss the results of the experiments that we conducted with our optimizer prototype. The objective of these experiments is to show the optimization potential of the proposed rewrite rules. Furthermore, we want to analyze the effects when using different query optimizers in the underlying database system. Therefore, we performed all experiments on two DBMSs but with comparable database configurations.

We used the same Windows XP machine with two 1.53GHz AMD Athlon 1800+ processors and 1GB main memory for all experiments. As DBMSs, we employed DB2 V9.1 on the one hand and Oracle 10g on the other hand. The experiments were conducted on a TPC-H database (see Section 1.2) containing TPC-H data created with a scaling factor of 10 ($\approx$ 10 GB of raw data). To physically separate the TPC-H data from the data created by the query sequences, we added a tablespace that only stores the tables of the TPC-H schema and a tablespace that only stores the indexes on tables of the TPC-H schema. We created indexes on all foreign keys and forced the DBMSs once to gather statistics on all tables, columns, and indexes. Since experiments should be repeatable, automatic maintenance features and automatic self-tuning options have been turned off as far as possible. 50% of the main memory was assigned to the buffer pool. We executed each query sequence three times and took the average runtime of these three runs as basis for the diagrams. To achieve common preconditions, we flushed the buffer pool and the caches before each run of a query sequence.

The experiments were performed with five query sequences of different complexity, i.e., these query sequences differ in the number of queries and in the number of rules that can be applied to them. These sample sequences represent typical query sequences generated by OLAP tools like the MicroStrategy DSS tools. The number of queries within the sequences ranges from 7 queries to 10 queries, i.e., the number of SQL statements within the sequences ranges from 20 statements to 29 statements. First, we measured the runtime of each of the original sequences. Afterwards, we applied the rewrite rules according to the control strategy introduced in Section 4.3.3 and measured the runtime after each rewrite step. Additionally, we considered two alternative ways to transform a query sequence into a single query, i.e., into a statement sequence that consists just of a single CREATE TABLE statement and a single INSERT statement. One option is to recursively replace the table references in the FROM clauses of the INSERT statements by the bodies of the INSERT statements that correspond to the referenced tables. To end up in a single INSERT statement, we have to start this replacement process with the INSERT statement that fills the final-result table. Due to the fact that an intermediate-result table can be referenced multiple times within a query sequence, the body of the resulting INSERT statement is a deeply-nested SQL query that contains many redundant subqueries. Another option arises when using the WITH clause. This means, we add a

WITH clause to the INSERT statement that fills the final-result table. The WITH clause has to contain a definition for each intermediate-result table. Such a definition consists of the table name, the attributes listed in the corresponding CREATE TABLE statement, and the body of the corresponding INSERT statement, which specifies the content of the intermediate-result table.

Figure 4.30 and Figure 4.31 compare the original query sequences with the rewritten query sequences and the two single-query variants. Figure 4.30 refers to DB2 whereas Figure 4.31 refers to Oracle. For each original query sequence, the figures show the runtime of the original query sequence, the runtime of the nested single query, the runtime of the single query using the WITH clause, the runtime of the worst rewritten query sequence, and the runtime of the best rewritten query sequence. All runtimes are relative values that refer to the corresponding original query sequence as 100%. More details on the experimental results, i.e., a table containing the measured values of the three runs for each query sequence, can be found in Appendix D.1.

Both DBMSs show a similar behavior. For all original query sequences, best results have been achieved by rewriting the query sequence. On the DB2 database system, we achieved a performance improvement of at least 75% and, for one of the sample sequences, we even reached a performance gain of more than 85%. With a performance improvement of at least more than 50% for each sequence and 70% in the best case, the performance gain using the Oracle database system was somehow lesser than that using the DB2 database system. However, we have to take into account that the absolute runtimes of the original query sequences using the Oracle database system are just half of the absolute runtimes of the original query sequences using the DB2 database system. So, the runtimes of the sequences after rewriting are nearly the same.

Besides the best rewriting results, the figures also include the worst runtimes that appeared during the rewriting process (excluding the original sequence). Just in a single case, there was a performance deterioration due to rewriting. In all other cases, the worst results of rewriting performed as good as the original sequence or even better. Using the DB2 database system, applying just a single rule increased performance even slightly more than transforming the original query sequence into a single query that makes use of the WITH clause.

In two of five cases, the nested single query led to some performance improvements on the DB2 database system. However, the benefit was always less than 50%. In all other cases, this variant showed a bad performance, i.e., in the worst case, the runtimes increased by nearly 60% using the DB2 database system and by even 100% using the Oracle database system. This is due to the high redundancy within the body of the INSERT statement in the single-query variant.

In comparison to the nested single query, the WITH-clause variant turned out to be a good solution. Using the DB2 database system, the performance improvement mostly ranged from 10% to 30%. In a single case, we even reached a performance gain of more than 80% using the DB2 database system. Using the Oracle database system, the performance improvement mostly ranged from 30% to 60%. However, in a single case, using the WITH-clause variant caused a remarkable performance deterioration of more than 250% which is due to a bad execution plan.

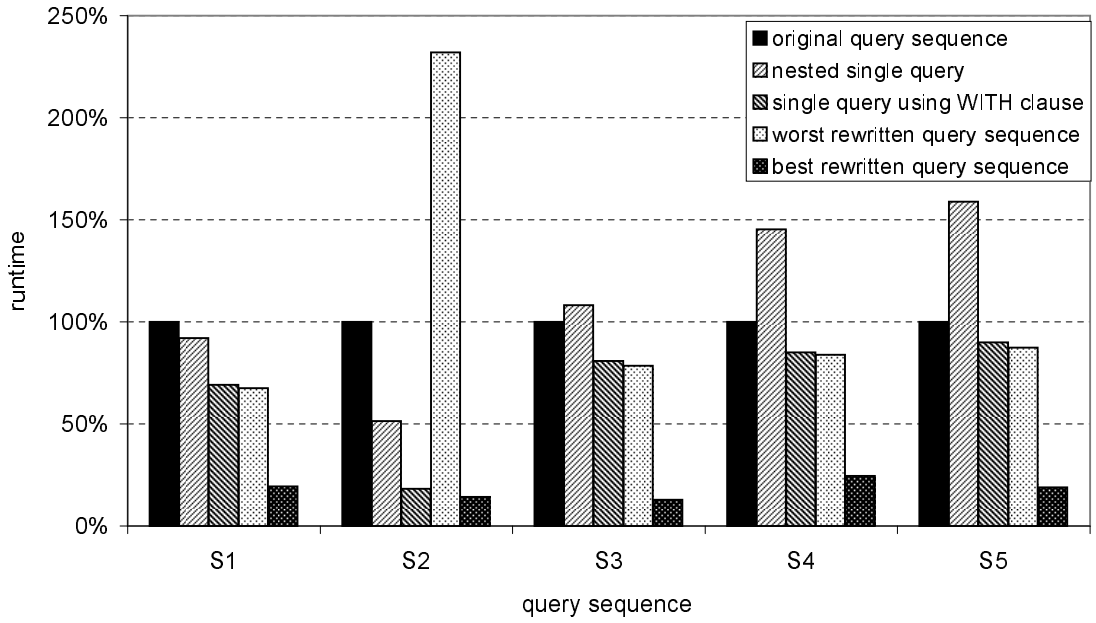


Figure 4.30: Comparison of runtimes between original query sequences, two variants of single queries, and the worst and best rewritten query sequences using the DB2 database system.

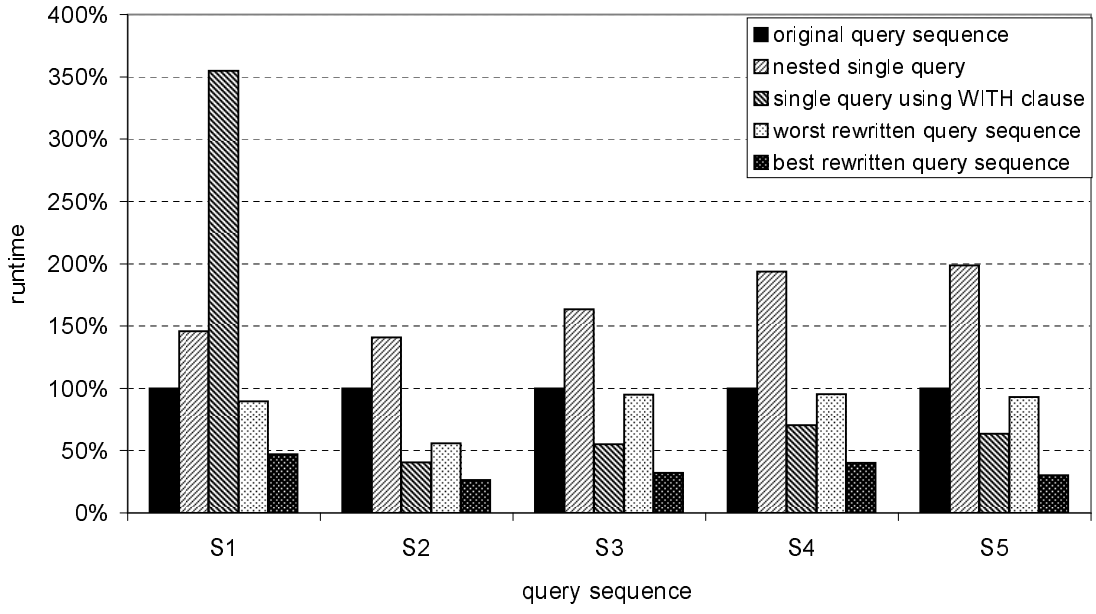


Figure 4.31: Comparison of runtimes between original query sequences, two variants of single queries, and the worst and best rewritten query sequences using the Oracle database system.

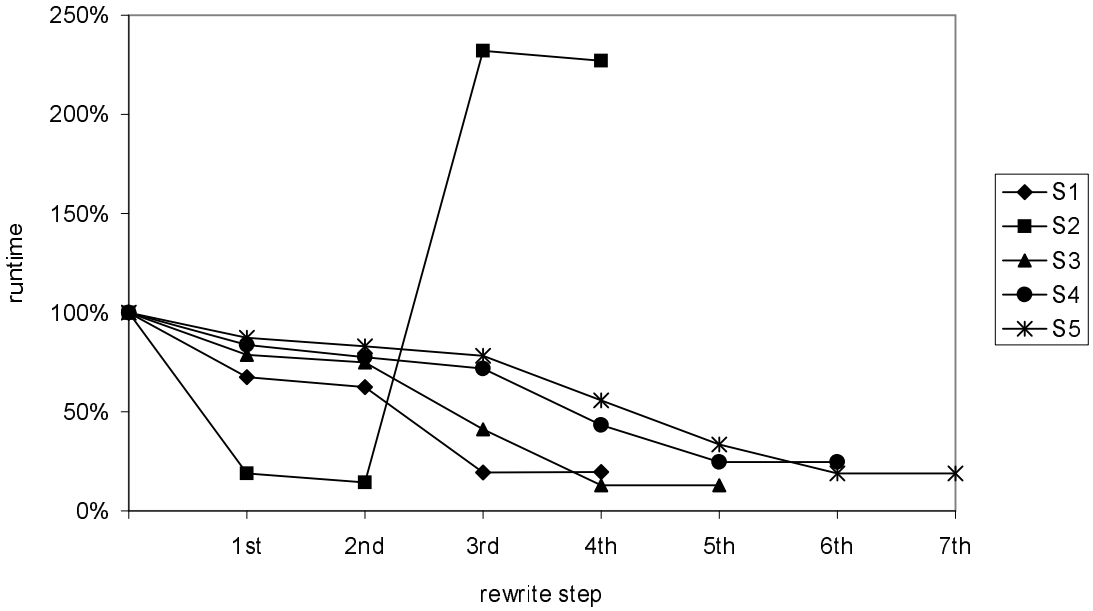


Figure 4.32: Runtimes of the query sequences after each rewrite step on the DB2 database system.

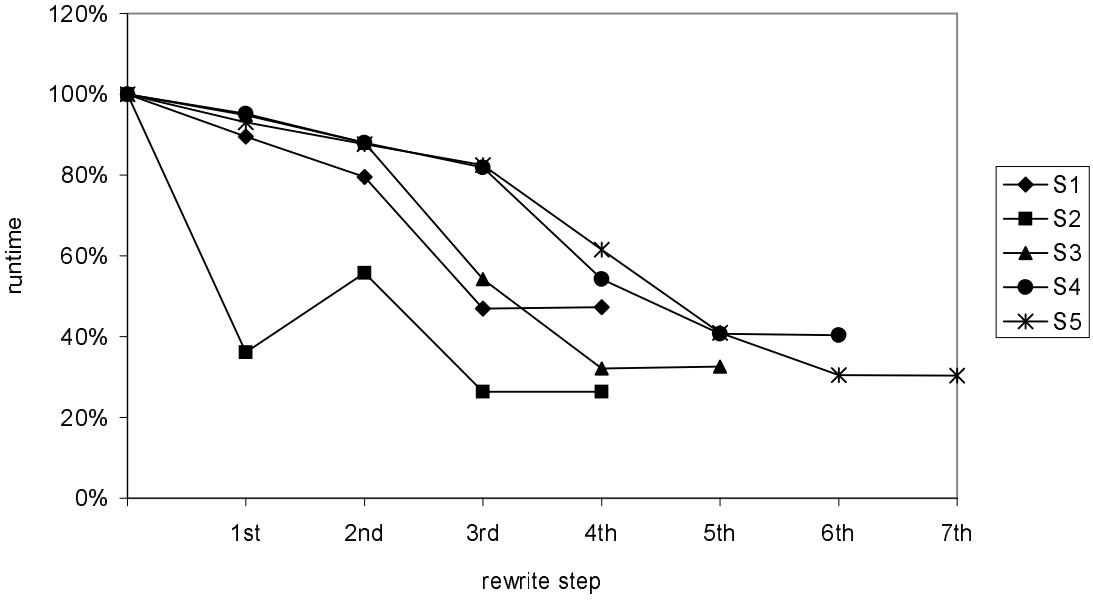


Figure 4.33: Runtimes of the query sequences after each rewrite step on the Oracle database system.

Figure 4.32 and Figure 4.33 show the runtimes of the original sequences as well as the runtimes of the query sequences after each rewrite step. These figures depict that not each rule application leads to a performance improvement and that performance improvements are not always significant. For instance, the last rule application did not significantly decrease runtimes. In some cases, the last rule application even slightly increased runtimes. Moreover, using the Oracle database system, the second rule application to query sequence *S2* caused some minor performance deterioration. However, the second rule application enabled the third rule application and after the third rule application, the sequence performed even better than the sequence after the first rule application. This means, sometimes the way to a better solution leads over a worse solution. Using the DB2 database system, the second rule application to query sequence *S2* caused no performance deterioration. However, the third rule application led to some tremendous performance deterioration followed by only a slight performance improvement due to the last rule application. Hence, *S2* is also a good example for the sometimes different behavior of the diverse DBMSs.

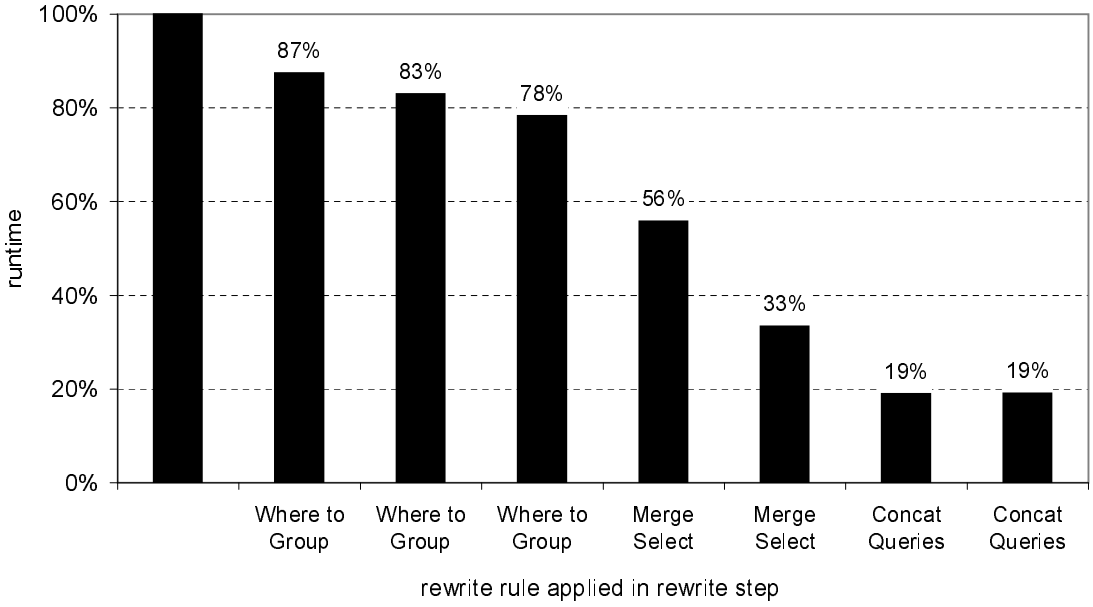


Figure 4.34: Names of the rules applied to query sequence *S5* and runtime after each rewrite step on the DB2 database system.

Figure 4.34 focuses on query sequence *S5*, which is the largest query sequence. We take this query sequence as an example to show how the rule applications affect the runtimes of a query sequence. According to the priority-based control strategy, first, we applied the class-1 rules *WhereToGroup* and *MergeSelect*. Afterwards, we applied the class-2 rule *ConcatQueries*. The *PredicatePushdown* rule would also have been applicable prior to the last application of the *ConcatQueries* rule. However, the control strategy omitted this unnecessary rule application. As the figure shows, the first application of the *WhereToGroup* rule was more beneficial than the second and third application of this rule. This is due to the fact that each application of the *WhereToGroup* rule increases

the size of the result table of the merged queries as well as the overhead for extracting a certain subset from the result table of the merged query when executing the queries which depend on the merged query. So, this depicts that the benefit of most of the *Merge* rules is always a trade-off between the cost savings due to merging and the additional overhead that arises from accessing the target table of the merged query. The greatest performance gain has been achieved by applying the *MergeSelect* rule, because, due to the merge, grouping has to be done only once, not twice. The last rule application does not have an impact on performance. This may be due to the fact that the last query just applies a predicate to the result of the previous query. Therefore, the merge does not offer much potential for optimization to the optimizer of the underlying database system.

We summarize that our approach of rewriting query sequences led to considerable performance improvements in the experiments which we conducted. In the best case, we achieved a performance gain up to more than 85%. However, the benefit depends on the DBMSs. Moreover, some rule applications within the rewriting process also caused a deterioration of performance or only less significant performance improvement. Transforming the query sequence into a single query that makes use of the WITH clause was also a good solution, but could not reach the performance gain of the rewriting approach. Transforming the query sequence into a nested single query turned out to be a bad solution, because this transformation mostly led to a performance deterioration.

5

Cost-Based Optimization of Query Sequences

In a cost-based optimizer, the decision which rules should be applied and in which order they should be applied is based on the comparison of cost estimates for the alternative sequences resulting from the rule applications. Therefore, a cost measure as well as a cost model that describes how to compute costs for a given sequence has to be defined. As CGO is a database-external approach and due to the fact that an appropriate cost estimate depends on the physical layout of the underlying database as well as on the strategies of its query optimizer, the cost estimation component would have to simulate the optimizers of all possibly underlying DBMSs. This is no feasible solution. Therefore, in this chapter, we propose a more practical approach that incorporates the cost-based optimizer of the underlying DBMS. (An extract of the work which we present in this chapter has been published in [Kra07], [KSM07], and [KM07].) Figure 5.1 gives an architectural overview of a cost-based CGO optimizer that realizes this approach. It shows the cost-based CGO optimizer and the underlying database system. Similar to the heuristic CGO optimizer, the cost-based CGO optimizer consists of the SQL parser, the SQL retranslator, the CGO rule set, and the control strategy. However, in the case of a cost-based CGO optimizer, the control strategy is cost-based instead of heuristic and the optimizer additionally contains a cost estimation component for query sequences. The cost estimation component provides a cost estimate for a given query sequence without executing it. For this purpose, it exploits the capability of the optimizer of the underlying database system to estimate costs for single SQL statements. Furthermore, it makes use of histogram propagation to improve the quality of these cost estimates in the context of query sequences. This means, it retrieves statistics data from the database catalog, modifies this data according to the statements of the query sequence, and stores the modified data in the database catalog again. The CGO optimizer communicates with the underlying database system via JDBC and Statistics API. Statistics API [KM07] is a JDBC-based

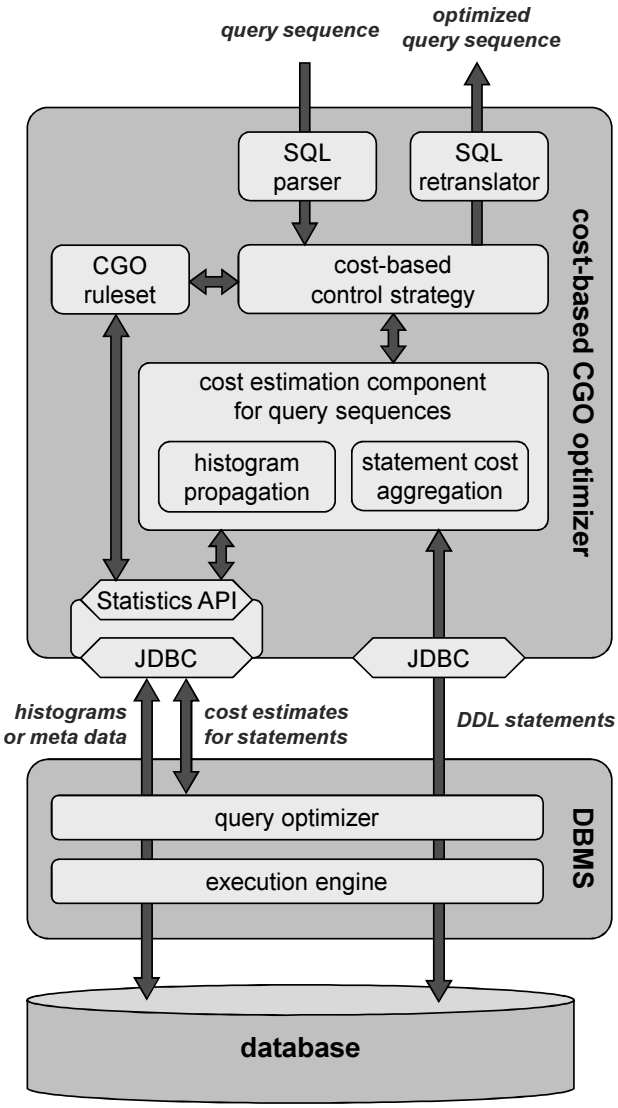


Figure 5.1: Architecture of the cost-based CGO optimizer.

application programming interface (API) that we have developed for DBMS-independent access and management of DBMS statistics, meta data, and cost estimates. It supports the retrieval and manipulation of histograms, the retrieval of meta data, and the retrieval of cost estimates for arbitrary SQL statements from different DBMSs. Thus, the CGO optimizer is largely independent from a certain vendor regarding the underlying DBMS.

The remainder of this chapter is organized as follows. In Section 5.1, we describe the main algorithm of the cost estimation component in more detail. Section 5.2 focuses on histogram propagation as it is being used within the cost estimation component. In Section 5.3, we highlight Statistics API. We look at some control strategies in Section 5.4. Afterwards, Section 5.5 discusses some implementation issues of the optimizer prototype. Finally, in Section 5.6, we present the results of some performance experiments.

5.1 Cost Estimation Algorithm

As stated before, we propose a practical approach to retrieve cost estimates for query sequences that exploits the capabilities of the optimizer of the underlying database system to estimate costs for single SQL statements. The idea is to sum up the cost estimates for all INSERT statements of a sequence to provide a cost estimate for an entire sequence. To make this work, we have to execute the CREATE TABLE statements that create the intermediate-result tables, before we force the optimizer to estimate costs for the INSERT statements. Otherwise, the optimizer would raise a failure due to the nonexistence of the intermediate-result tables used in the INSERT statements. When the CREATE TABLE statements have been executed, the optimizer can compute an execution plan for each INSERT statement and return a cost estimate for this plan. However, as there are no statistics available for the intermediate-result tables, the optimizer would use default values for the cardinality of these tables and default selectivities for predicates that include attributes of these tables. This would result in inaccurate and thus useless cost estimates. To solve this problem, we make use of histogram propagation. Since histograms are supported and used for selectivity estimation in a similar way by almost all commercial DBMSs, we can retrieve them from the database catalog of the underlying database system, propagate them through the queries that form the bodies of the INSERT statements and store the histograms for the intermediate-result tables in the database catalog again. So, the optimizer can use them to produce more accurate query plans and cost estimates for the INSERT statements that depend on intermediate-result tables.

Figure 5.2 shows the complete algorithm. The loops iterate over the statements in the order in which they appear in the query sequence. Furthermore, we presume that histograms of intermediate-result tables are being cached after their computation. Therefore, in line 8, the algorithm retrieves histograms from the underlying database system just for the base tables and not for intermediate-result tables.

As the tables created for cost estimation are being dropped after cost estimation, the associated statistics stored in the database catalog get lost. Thus, when a query sequence is being run later, the histograms derived from histogram propagation no longer exist. However, the experiments at the end of this chapter show that the propagated statistics should also be used, when executing a query sequence, and not only, when estimating the costs of a query sequence (see Section 5.6). This can be achieved by modifying the algorithm presented above so that it can also be used for executing query sequences with these histograms available. For this purpose, in line 5 the current INSERT statement i has to be executed. Furthermore, line 12 and 13 have to be modified so that only the intermediate-result tables are being dropped, i.e., the foreach-loop has to iterate over all DROP TABLE statements of the sequence and has to execute them.

5.2 Histogram Propagation

For histogram propagation, we make use of the algebraic operations introduced in Section 3.2.2 and apply them to histogram relations, accordingly. The following subsections explain how the different operations process the input histogram relations in order to return

```
Input:   A query sequence S.
Output:  A cost estimate for S.

1    TotalCosts ← 0;

2    foreach CREATE TABLE statement c in S do
3        execute c in the underlying database system;

4    foreach INSERT statement i in S do begin
5        retrieve a cost estimate Costs for i from the optimizer of the
         underlying database system by the use of Statistics API;
6        TotalCosts ← TotalCosts + Costs;
7        translate the body of i into an algebraic tree t;
8        retrieve histograms for the base tables accessed by t from the
         underlying database system by the use of Statistics API;
9        propagate the histograms through t;
10       store the histograms returned by the root node of t as statistics for
         the target table of i in the catalog of the underlying database system
         by the use of Statistics API;
11   end;

12   foreach CREATE TABLE statement c in S do
13       drop the table that has been created by c;

14   return TotalCosts;
```

Figure 5.2: Cost estimation algorithm.

an output histogram relation. However, in the following, we omit to describe histogram propagation for the renaming operation, because the renaming operation does not modify the content of the input histogram relation.

The work described in this section is based on related work that deals with the usage of histograms in query optimization [Cha98] [Ioa03] [IC93] [IP95b] [IP95a] [PHIS96] [PI97][BC02] as well as on related work that deals with the usage of histograms in approximate query answering [IP99] [PGI99]. We exploit and extend the techniques introduced in these publications for our needs. This is, we add support for grouping and aggregation as well as we add support for arithmetic expressions, because these topics are not covered by literature concerning histogram propagation and approximate query answering. Furthermore, we take respect for different data types and their characteristics in our algorithms and implementations, whereas literature contributions are mostly restricted to the integer data type.

The descriptions and formulas within this section use the following notation:

HR_1, HR_2, ...	denote histogram relations
$HR_1(A_1, A_2, A_3, ...)$, ...	denote the relation schemas of histogram relations where $A_1, A_2, ...$ are the attributes
B_1, B_2, ...	denote buckets.

In the following, we also use the term *relation* for *relation schemas* or *relation instances*, when the meaning is clear from the context in which we use this term.

The basic algorithms for algebraic operations, comparison operations, and arithmetic operations are very similar due to the fact that all calculations are based on the *Attribute Value Independence Assumption*. This is, we enumerate all bucket combinations that can be built from the histograms affected by an operation and apply the operation-specific calculations to each bucket combination to retrieve the buckets of the result histogram. Bucket combinations consist of a single bucket from each histogram that is affected, i.e., when we enumerate the bucket combinations, we actually build the Cartesian product over all affected histograms. To calculate the cardinality of the result bucket, we have to determine the probability of the bucket combination that leads to this result. Thus, we treat the data distribution as a probability distribution and calculate the proportion of the actual bucket combination at all possible bucket combinations. Details on this as well as details on the algorithms and formulas used to calculate the buckets of the result histogram can be found in the descriptions of the various operations in the remainder of this section.

Enumerating all bucket combinations implicates that, in the worst case, there are as much buckets in the result histograms as the product of the number of buckets in the histograms affected by the specific operation. Serializing the result histogram may additionally double the number of buckets. Hence, the deeper an operation tree, the larger the histograms may grow in number of buckets. To cope with all these problems, a normalization step could be added prior to the enumeration phase and / or after the result histogram has been produced (see Section 3.3.4). In this normalization step, the number of buckets could be reduced to a given upper limit by merging adjacent buckets. In addition, the histogram could be transformed into an equi-width or equi-depth histogram. For our experiments, we omitted any normalizations during the propagation process, because the number of buckets didn't increase extremely during propagation. The main reason is that selections rather reduce than increase the number of buckets.

5.2.1 Cartesian Product

Cartesian product is a binary operation that has two input histogram relations, one output histogram relation, but no further parameters. The output histogram relation contains all attributes of both input histogram relations.

In the relational algebra of multi-relations, the Cartesian product joins all tuples of one input relation with all tuples of the other input relation in every possible combination. Since all possible combinations of tuples are included in the output relation, the cardi-

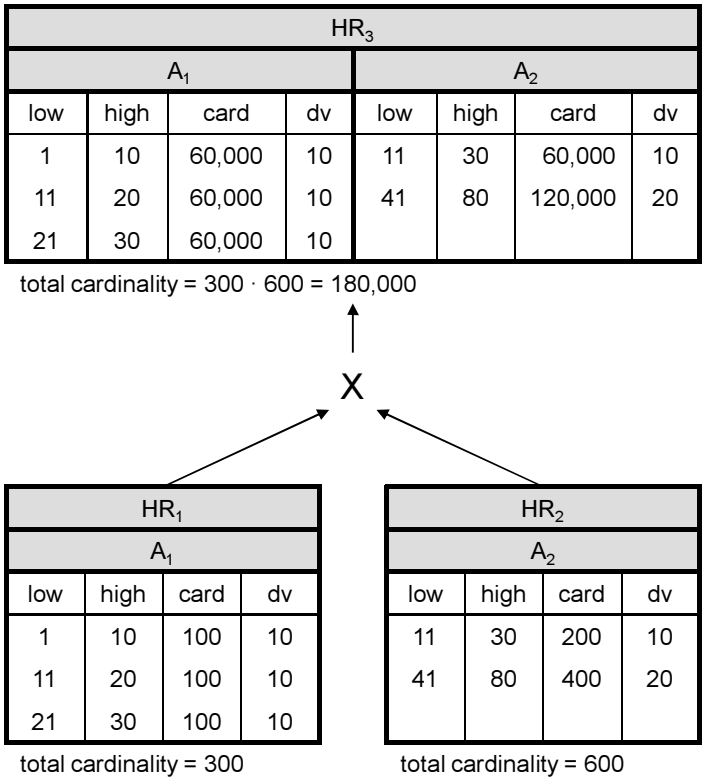


Figure 5.3: Example for the algebraic operation *Cartesian product*.

nality of the output relation is the product of the cardinalities of the two input relations. However, the relative data distribution of an attribute in the output relation equals the relative data distribution of this attribute in the corresponding input relation. This can also be applied to histogram relations. For an attribute in the output histogram relation, the histogram data concerning bucket bounds (*low* and *high*) and number of distinct values (*dv*) can be taken from the corresponding histogram in the input histogram relations. However, the cardinality (*card*) of each bucket has to be multiplied by the total cardinality of the input histogram relation, where the associated attribute does not belong to.

Figure 5.3 shows a sample scenario. Given are two input histogram relations: Histogram relation HR_1 with a single attribute A_1 and a total cardinality of 300 and histogram relation HR_2 with a single attribute A_2 and a total cardinality of 600. Hence, the output histogram relation HR_3 contains both attributes, A_1 and A_2, and has a total cardinality of 180 000, which is the product of the total cardinalities of histogram relation HR_1 and histogram relation HR_2. The histogram of attribute A_1 in histogram relation HR_3 is derived from the histogram of A_1 in histogram relation HR_1 by multiplying the *card* value of each bucket by the total cardinality of histogram relation HR_2. Accordingly, the histogram of attribute A_2 in histogram relation HR_3 is derived from the histogram of attribute A_2 in histogram relation HR_2 by multiplying the *card* value of each bucket by the total cardinality of histogram relation HR_1.

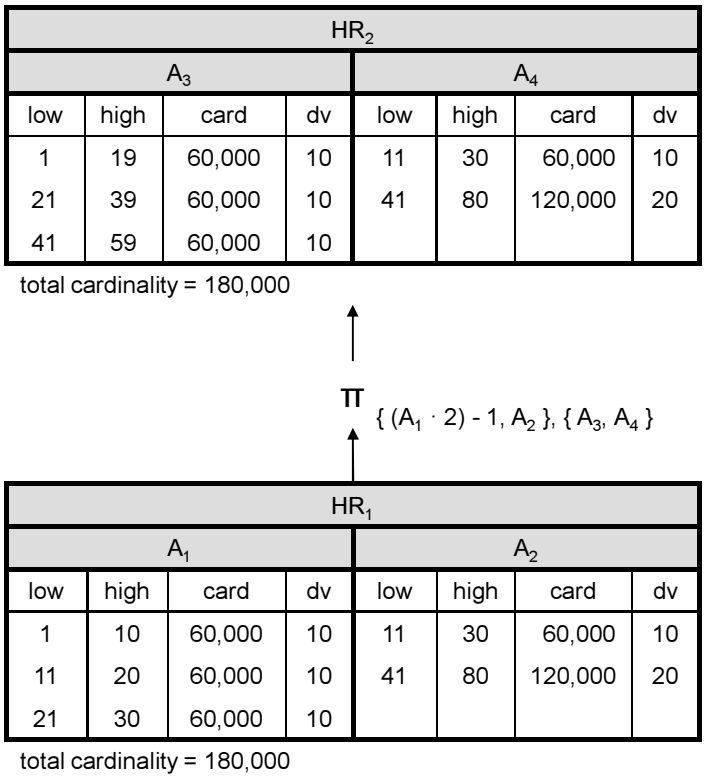


Figure 5.4: Example for the algebraic operation *projection*.

5.2.2 Projection

Projection is a unary operation that has one input histogram relation and one output histogram relation. Additionally, it has a list of arithmetic expressions and a list of attributes as parameters. Each arithmetic expression in the first list builds a pair with the attribute at the same position in the second list. This is, each attribute in the second list represents an attribute in the output histogram relation whose content is defined by the corresponding arithmetic expression in the first list. Thus, the output histogram relation contains as many attributes as there are elements in each of the two lists.

In the relational algebra of multi-relations, projections are used to eliminate attributes when they are not used anymore above this operation in a graph of algebraic operations. Furthermore, projections can be used to rename attributes. However, some definitions of the renaming operation (not the definition used in this document) also support this functionality. In these two cases, the list of arithmetic expressions is a list of attributes. In the first case, the attribute in the expression list and the corresponding attribute in the attribute list are equivalent. In addition, the projection operation supports the computation of new attributes based on arithmetic expressions. These arithmetic expressions are made up of arithmetic operations, cast operations (for data type conversion), constant values, and attributes of the input histogram relation. In all cases, the total cardinality of the output relation equals the total cardinality of the input relation. The same holds for histogram relations.

Figure 5.4 shows a sample application of the projection operation. Given is an input histogram relation HR_1 with two attributes, A_1 and A_2, and a total cardinality of 180 000. The output histogram relation HR_2 contains two attributes due to the fact that the projection has two pairs consisting of an arithmetic expression and an attribute as parameter. The first pair provides a new attribute A_3 that represents the result of the arithmetic expression $(A_1 \cdot 2) - 1$. The second pair shows an attribute renaming, i.e., it provides a new attribute A_4 which is derived from the attribute A_2 in histogram relation HR_1.

Propagating histograms through a projection means that we have to compute the result histogram for each expression in the list of arithmetic expressions. For this purpose, we look upon an arithmetic expression as a tree consisting of arithmetic operations, cast operations, constant values, and attributes. We traverse this tree in a bottom-up manner and propagate the histograms from the leaf nodes through the inner nodes to the root node. Therefore, each node in the tree has to provide an output histogram, whose computation is based on its input histograms. In the following, we explain how this works for the different kinds of nodes.

Arithmetic Operations

Input: A binary arithmetic operation Φ and two input histograms H_1 and H_2.
Output: An output histogram which is the result of applying Φ to H_1 and H_2.

```
1     create a new histogram H3;
2     optionally normalize histograms H1 and H2;

3     foreach bucket B1 in H1 do
4         foreach bucket B2 in H2 do begin
5             compute a new bucket B3 = B1 Φ B2;
6             add B3 to H3;
7         end;

8     serialize and optionally normalize histogram H3;
9     return H3;
```

Figure 5.5: Algorithm for propagating histograms through binary arithmetic operations.

Most of the arithmetic operations are binary operations. When we propagate histograms through such an operation, we have to take the *Attribute Value Independence Assumption* into account. Hence, we have to consider each possible bucket combination that can be built from the buckets of the two input histograms. The output histogram is calculated by enumerating all bucket combinations and by determining the bucket that

results from applying the arithmetic operation to each bucket combination. Thereby, the two buckets of a bucket combination serve as the operands for the arithmetic operation. See Figure 5.5 for the algorithm. To determine the bounds of an output bucket for two input buckets, we make use of interval arithmetic [PP98]. The following list shows some arithmetic operations and the corresponding formulas used to derive the lower and upper bound of the output bucket B_3 based on the lower and upper bounds of the input buckets B_1 and B_2:

- Addition ($B_3 = B_1 + B_2$):
  $B_3.low = B_1.low + B_2.low$
  $B_3.high = B_1.high + B_2.high$
- Subtraction ($B_3 = B_1 - B_2$):
  $B_3.low = B_1.low - B_2.high$
  $B_3.high = B_1.high - B_2.low$
- Multiplication ($B_3 = B_1 \cdot B_2$):
  $B_3.low = min(B_1.low \cdot B_2.low, B_1.low \cdot B_2.high, B_1.high \cdot B_2.low, B_1.high \cdot B_2.high)$
  $B_3.high = max(B_1.low \cdot B_2.low, B_1.low \cdot B_2.high, B_1.high \cdot B_2.low, B_1.high \cdot B_2.high)$
- Division ($B_3 = B_1/B_2$):
  $B_3.low = min(B_1.low/B_2.low, B_1.low/B_2.high, B_1.high/B_2.low, B_1.high/B_2.high)$
  $B_3.high = max(B_1.low/B_2.low, B_1.low/B_2.high, B_1.high/B_2.low, B_1.high/B_2.high)$
- Change of sign ($B_3 = -B_1$):
  $B_3.low = -B_1.high$
  $B_3.high = -B_1.low$

Due to the special semantics of the *null* value, the result bucket of these arithmetic operations is a *null* bucket if one of the input buckets is a *null* bucket.

As mentioned before, the total cardinality of the output histogram relation of a projection operation is the same as the total cardinality of the input histogram relation. According to this, the cardinality of a histogram that is the output of an arithmetic operation is the same as the cardinality of each of its input histograms. The proportion of the cardinality of an output bucket in relation to the total cardinality of the output histogram conforms to the proportion of the corresponding input bucket combination in relation to all possible input bucket combinations. So, for a binary arithmetic operation with a histogram cardinality *HR.card* and a bucket combination consisting of the two input buckets B_1 and B_2, the cardinality of the output bucket B_3 can be calculated as follows:

$$B_3.card = \frac{B_1.card}{HR.card} \cdot \frac{B_2.card}{HR.card} \cdot HR.card = \frac{B_1.card \cdot B_2.card}{HR.card}$$

The upper bound for the number of distinct values is the product of the *dv* values of the input buckets:

$$B_3.dv = B_1.dv \cdot B_2.dv$$

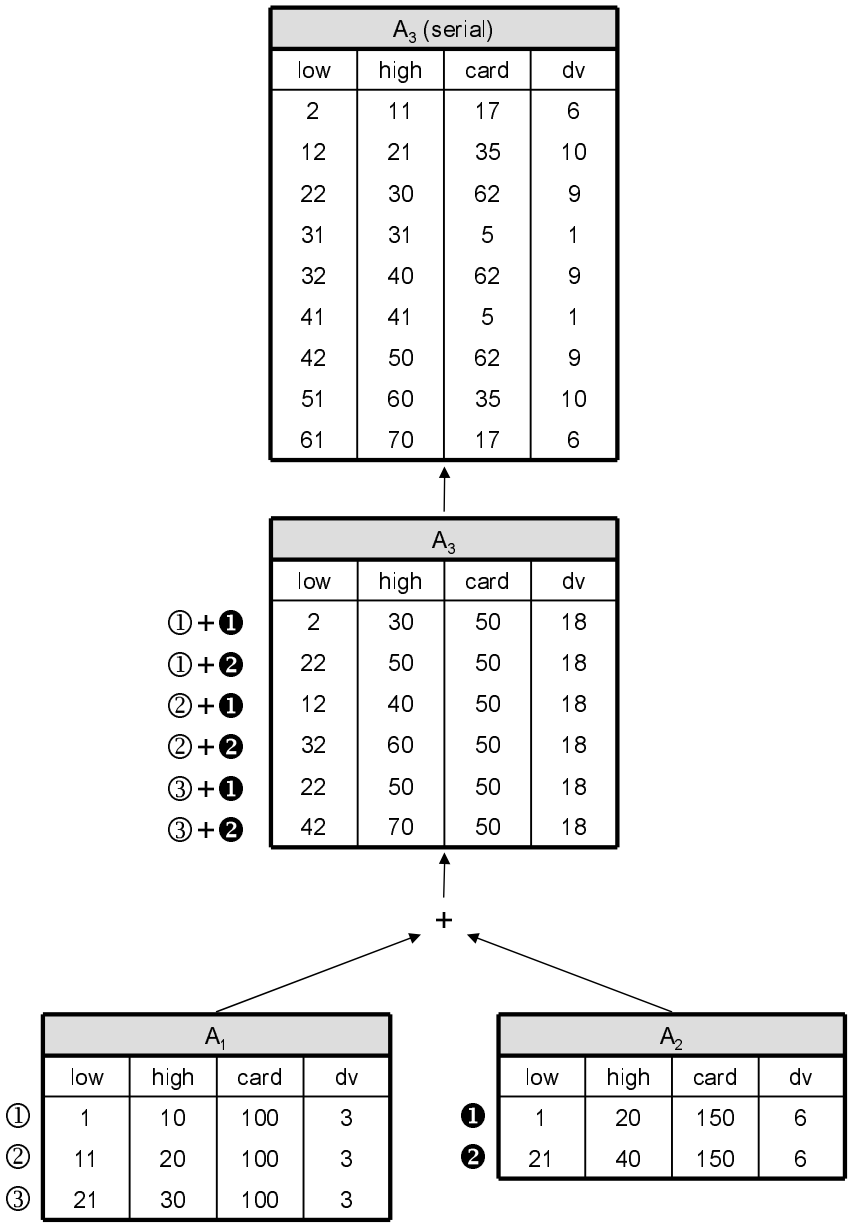


Figure 5.6: Example for the arithmetic operation *addition* ($A_3 = A_1 + A_2$).

This complies with a value distribution where each value combination leads to a unique result value when applying the arithmetic operation. For example, given are two attributes where the first one contains the distinct values $1, 2$ and the second one is built up of the distinct values $4, 5, 6$. Assuming *Attribute Value Independence*, the result of multiplying these two attributes possibly results in an attribute that contains the values $4, 5, 6, 8, 10, 12$. For addition or subtraction, the lower bound for the number of distinct values is the sum of the *dv* values of the input buckets minus 1. The same holds for multiplication and division, if none of the intervals defined by the input buckets contains the absorbing element 0. Otherwise, 1 is a safe lower bound for the number of distinct values. However, we use the upper bound in our computations.

Figure 5.6 shows the application of the arithmetic operation *addition* to two integer attributes, A_1 and A_2, from the same table. The histogram of attribute A_1 consists of three buckets, whereas the histogram of attribute A_2 consists of two buckets. Attribute A_3 stores the result of $A_1 + A_2$. Therefore, the histogram of attribute A_3 contains a bucket for each bucket combination that can be built from the buckets in the histograms of attribute A_1 and attribute A_2 (see the identifying numbers to the left of the histograms). All buckets in the histogram of attribute A_3 are pairwise overlapping with some other buckets. Since some of the arithmetic or algebraic operations (e.g. grouping) require that input histograms are serial, the histograms resulting from the application of an arithmetic operation have to be transformed into serial histograms again. For this purpose, we split each bucket at all bucket bounds that appear in the histogram and that lie between its own lower and upper bound. Afterwards, we merge the buckets with equivalent bucket bounds and get a histogram that has no overlapping buckets any more (also see Section 3.3.4). When splitting the buckets, we assume that the values are uniformly distributed between the bounds of a bucket (*Uniform Distribution Assumption / Continuous Value Assumption*). The histogram at the top of Figure 5.6 shows the serial version of the histogram of attribute A_3 (values in card and dv have been rounded).

Cast Operations

Cast operations transform histograms of a certain data type into histograms of another data type. Therefore, they traverse all buckets of the input histogram, transform their bounds and adapt their number of distinct values, if necessary. For example, when transforming a decimal histogram into an integer histogram, the cast operation rounds down the lower bound of the bucket to the next integer number, rounds up the upper bound of the bucket to the next integer number and decreases the number of distinct values to the length of the bucket's range, if it is greater.

Attributes

Attributes that appear in an arithmetic expression simply return the corresponding histogram from the input histogram relation as output histogram without any modifications.

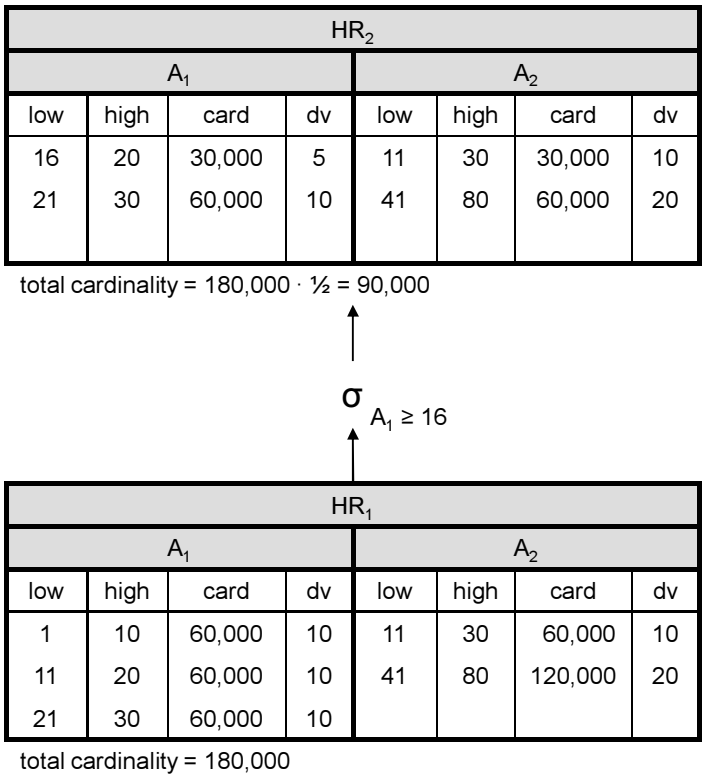


Figure 5.7: Example for the algebraic operation *selection*.

Constant Values

A constant value is a single fixed value. We represent constant values as histograms, too. Such a histogram contains a single bucket whose lower and upper bound equals the value of the constant, *dv* is set to 1 and *card* is set to the total cardinality of the input histogram relation. Moreover, the *null* value is also represented this way. Hence, arithmetic operations do not have to differ between histograms and constant values.

5.2.3 Selection

Selection is a unary operation that has one input histogram relation, one output histogram relation and a predicate as parameter. All attributes from the input histogram relation are taken to the output histogram relation.

In the relational algebra of multi-relations, selections are used to filter a relation, i.e., to eliminate tuples in a relation for which the predicate evaluates to $false$. We transfer this to histogram relations with respect to the *Attribute Value Independence Assumption*. Moreover, we distinguish between simple and complex predicates. Simple predicates have the form $A_i \, \Phi \, A_j$, $A_i \, \Phi \, C$ or $C \, \Phi \, A_i$, where A_i and A_j are attributes, C is a constant value, and Φ is one of the following comparison operations: $=, \neq, \leq, \geq, <, >$. Figure 5.7 shows a sample for a selection with a simple predicate. Complex predicates include arithmetic terms as operands of comparison operations and/or consist of several simpler predicates

combined by the logical operations AND, OR, and NOT. We differ between simple and complex predicates in histogram propagation due to the fact that we handle histograms of attributes that appear in the predicates differently for the two kinds of predicates. For simple predicates, we modify the input histograms that correspond to the attributes that appear as operands of the comparison operation in that way that we remove buckets or alter their bounds, cardinality, and number of distinct values so that the histograms of these attributes in the output histogram relation only include buckets that satisfy the predicate. Adapting histograms of attributes that are part of a complex predicate is a complicated problem, which is similar to constraint solving, because the attributes may be embedded in quite complex arithmetic and logical expressions. Sometimes it is even not possible. Therefore, we do not change the relative data distribution in the histograms of these attributes, but we treat them as histograms of attributes which are not included in the predicate. Presuming the *Attribute Value Independence Assumption*, the histograms of all attributes in the input histogram relation which are not included in the predicate or which are part of a complex predicate get adapted to the cardinality of the output histogram relation keeping the relative data distribution of the histograms in the input histogram relation. This means, the number of buckets does not change as well as the bucket bounds do not change, only the cardinality of the buckets is being modified (and the number of distinct values if necessary). First, we estimate the selectivity of the predicate. Then, we use the selectivity to compute the cardinality of the output histogram relation. Finally, we adapt the cardinalities (and the number of distinct values) in the buckets of the histograms accordingly. Some of the complex predicates can be transformed into simple predicates as follows. A predicate that consists of several simple predicates which are all combined by AND can be treated as a sequence of selections with a separate selection operation for each of those simple predicates.

Similar to arithmetic operations in projections, we presume the *Attribute Value Independence Assumption* and enumerate all bucket combinations that can be built up of the histograms corresponding to the operands of the comparison operation. When an operand is not just an attribute but an arithmetic expression, we first have to compute the histogram of this arithmetic expression according to the computation of result histograms for arithmetic expressions in projections (see Section 5.2.2). Then, we apply the comparison operation to each bucket combination and modify the buckets accordingly. For those operands that are just attributes, we can use the histogram modified by the comparison operation as histogram for this attribute in the output histogram relation of the selection operation. In the following, we describe for some comparison operations how they modify the buckets in the histograms of their operands.

Equal

The comparison operation *equal* can be used for predicates of the form $E_1 = E_2$ and E_1 LIKE C where E_1 and E_2 are arithmetic expressions and C is a search pattern without wildcards.

Figure 5.8 shows two partially overlapping buckets, B_1 and B_2, where B_1 belongs to the input histogram of the first operand and B_2 belongs to the input histogram of the second

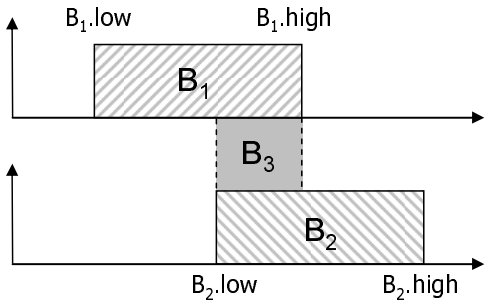


Figure 5.8: Two buckets B_1 and B_2 and their overlapping area B_3.

operand. The bucket denoted as B_3 covers the range in which the two buckets overlap. It specifies the part of the two original buckets which satisfies the predicate. The properties of bucket B_3 can be calculated as follows (presuming that the two buckets overlap):

$$
\begin{aligned}
B_3.low &= max(B_1.low, B_2.low) \\
B_3.high &= min(B_1.high, B_2.high) \\
B_3.dv &= min\left(\frac{length(B_3)}{length(B_1)} \cdot B_1.dv, \frac{length(B_3)}{length(B_2)} \cdot B_2.dv\right) \\
B_3.card &= \left(\frac{B_1.card}{HR.card} \cdot \frac{1}{B_1.dv}\right) \cdot \left(\frac{B_2.card}{HR.card} \cdot \frac{1}{B_2.dv}\right) \cdot HR.card \cdot B_3.dv
\end{aligned}
$$

$B_3.low$ and $B_3.high$ are the bounds of the interval, where buckets B_1 and B_2 overlap, assuming that the bounds appear in the respective value sets of both buckets. The fraction of distinct values for bucket B_1 in the overlapping interval and the fraction of distinct values for bucket B_2 in the overlapping interval are being calculated under the *Continuous Value Assumption*. We choose the minimum of these two values as $B_3.dv$ relying on the *Inclusion Assumption*. In the formula for $B_3.card$, the first expression in brackets computes the relative frequency of a distinct value in bucket B_1, the second expression in brackets does the same for bucket B_2. The product of these two expressions and the cardinality $HR.card$ of the input histogram relation is the frequency of a distinct value in bucket B_3, i.e., the frequency of a combination of distinct values in bucket B_1 and B_2 that satisfies the predicate. This value multiplied by $B_3.dv$ results in the total frequency of all distinct values in bucket B_3, which is the cardinality of this bucket.

The algorithm for the comparison operation *equal* is very similar to the join algorithm for histograms introduced in [BC02]. There, the buckets of both histograms are split in a way that both histograms have buckets with the same lower and upper bounds, i.e., there is no partial overlapping of any buckets. Afterwards, the pairs of buckets with same bounds are compared producing a bucket with the same bounds as well. In our approach, we do it the other way round. This is, we first compare each bucket of one histogram with each bucket of the other histogram and, when they overlap, we split the buckets by determining the overlapping interval.

As great benefit of supporting comparison operations where both operands are attributes, the same implementation of the comparison operation *equal* can be used for

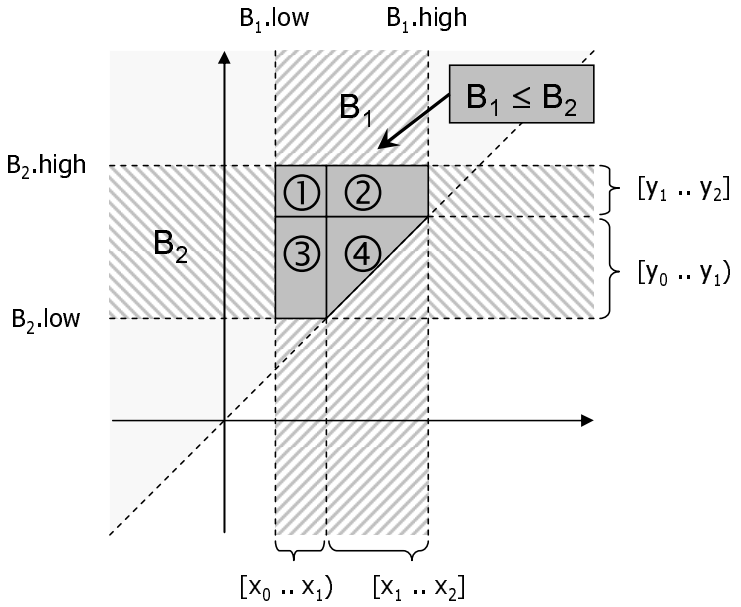


Figure 5.9: Application of the comparison operation *less equal* to two buckets B_1 and B_2.

different purposes:

- $\sigma_{A_i=A_j}(R)$: A predicate comparing two attributes.
- $\sigma_{A_i=C}(R)$: A predicate comparing an attribute and a constant value, because constant values are represented by histograms, too.
- $R_1 \bowtie_{A_i=A_j} R_2 \Rightarrow \sigma_{A_i=A_j}(R_1 \times R_2)$: A join condition. As we can transform a join operation into a Cartesian product followed by a selection with the join condition as predicate, we need no separate implementation of a join operation.
- $\sigma_{A_i \ IN \ (C_1,\ldots,C_n)}(R) \Rightarrow R \bowtie_{A_i=A} R_C \Rightarrow \sigma_{A_i=A}(R \times R_C)$: An IN predicate. IN predicates comparing an attribute with a set of values can be realized by a join (Cartesian product followed by a selection) with an additional table R_C that contains the values as rows. The additional table is represented by a histogram relation with a single attribute A and a corresponding histogram that contains a single bucket for each value in the value set.

This uniform treatment similarly applies to the other comparison operations, *less equal* and *less*, too.

Less Equal

The comparison operation *less equal* can be used for predicates of the form $E_1 \leq E_2$ where E_1 and E_2 are arithmetic expressions. Moreover, the same operation can be used for predicates of the form $E_1 \geq E_2$, because we can transform these predicates by interchanging the two arithmetic expressions into predicates of the form $E_2 \leq E_1$.

As the comparison operation *less equal* is not a symmetric comparison operation like the comparison operation *equal*, the two input buckets are adapted differently and therefore result in two different output buckets, one for each input bucket (see Figure 5.10). Hence, the computations are also somehow more complex. Figure 5.9 shows the application of the comparison operation *less equal* to the two partially overlapping buckets B_1 and B_2. Bucket B_1 belongs to the histogram that corresponds to the first operand of the comparison operation, whereas Bucket B_2 belongs to the histogram that corresponds to the second operand of the comparison operation. The Figure shows a 2-dimensional diagram with the domains of the two operands as axes. The two hatched bars represent the intervals of the two buckets. The gray area above the bisecting line, where both bars overlap, marks the area that contains the value combinations that satisfy the predicate. When both bars overlap completely below the bisecting line, none of the value combinations satisfies the predicate, because each value that falls into the interval of bucket B_2 is smaller than all values that fall into the interval of bucket B_1. The other extreme is when they overlap completely above the bisecting line. In this case, all value combinations satisfy the predicate, because each value that falls into the interval of bucket B_1 is smaller than all values that fall into the interval of bucket B_2. If the bisecting line intersects the area of overlapping, only part of the value combinations satisfy the predicate and we can distinguish up to four subareas. These subareas are marked with numbers 1 to 4 in Figure 5.9. The intervals that determine the size of these subareas are specified at the right side and at the bottom side of the Figure. The bounds of these intervals are defined as follows:

$$\begin{aligned} x_0 &= B_1.low \\ x_1 = y_0 &= max(B_1.low, B_2.low) \\ x_2 = y_1 &= min(B_1.high, B_2.high) \\ y_2 &= B_2.high \end{aligned}$$

Based on this information, we can calculate the number of valid combinations of distinct values for both buckets under the *Attribute Value Independence Assumption*. We do this separately for each subarea, i.e., for both buckets, we compute the fractions of distinct values that fall into the intervals of the different subareas. Afterwards, for each subarea, we multiply the corresponding fraction of bucket B_1 with the corresponding fraction of bucket B_2. The result is the number of possible combinations of distinct values for that subarea. For the subarea denoted as subarea 4, we multiply the resulting value with 0.5 due to the triangular form of that subarea. So, for integer histograms we compute the number of combinations, dv_1, dv_2, dv_3, and dv_4, as follows (for decimal histograms, replace $x_1 - 1$ by x_1 and $y_1 - 1$ by y_1):

$$\begin{aligned} lx &= length(B_1) \\ ly &= length(B_2) \\ lx_{01} &= length(x_0, x_1 - 1) \\ lx_{12} &= length(x_1, x_2) \\ ly_{01} &= length(y_0, y_1 - 1) \\ ly_{12} &= length(y_1, y_2) \end{aligned}$$

$$
\begin{aligned}
dv_1 &= \frac{lx_{01}}{lx} \cdot B_1.dv \cdot \frac{ly_{12}}{ly} \cdot B_2.dv \\
dv_2 &= \frac{lx_{12}}{lx} \cdot B_1.dv \cdot \frac{ly_{12}}{ly} \cdot B_2.dv \\
dv_3 &= \frac{lx_{01}}{lx} \cdot B_1.dv \cdot \frac{ly_{01}}{ly} \cdot B_2.dv \\
dv_4 &= \frac{lx_{12}}{lx} \cdot B_1.dv \cdot \frac{ly_{01}}{ly} \cdot B_2.dv \cdot \frac{1}{2}
\end{aligned}
$$

For each bucket combination where the buckets in the 2-dimensional diagram overlap above the bisecting line, we take over a single modified bucket to each of the modified histograms that belong to the operands of the comparison operation. We compute the cardinality of these buckets using the previously computed number of combinations of distinct values for each subarea. The modified bucket B'_1 that corresponds to bucket B_1 has the following properties:

$$
\begin{aligned}
B'_1.low &= x_0 \\
B'_1.high &= x_2 \\
B'_1.card &= \left(\frac{B_1.card}{HR.card} \cdot \frac{1}{B_1.dv}\right) \cdot \left(\frac{B_2.card}{HR.card} \cdot \frac{1}{B_2.dv}\right) \\
&\quad \cdot HR.card \cdot (dv_1 + dv_2 + dv_3 + dv_4) \\
B'_1.dv &= \frac{lx_{02}}{lx} \cdot B_1.dv
\end{aligned}
$$

And the modified bucket B'_2 that corresponds to bucket B_2 has the following properties:

$$
\begin{aligned}
B'_2.low &= y_0 \\
B'_2.high &= y_2 \\
B'_2.card &= \left(\frac{B_1.card}{HR.card} \cdot \frac{1}{B_1.dv}\right) \cdot \left(\frac{B_2.card}{HR.card} \cdot \frac{1}{B_2.dv}\right) \\
&\quad \cdot HR.card \cdot (dv_1 + dv_2 + dv_3 + dv_4) \\
B'_2.dv &= \frac{ly_{02}}{ly} \cdot B_2.dv
\end{aligned}
$$

The cardinalities of both buckets have to be equivalent as well as the total cardinality of the modified histograms has to be equivalent, because the modified histograms belong to attributes in the same histogram relation or expressions defined on the attributes of the same histogram relation. As mentioned before, the two input buckets of the comparison operation are adapted differently and therefore result in two different output buckets, one for each input bucket. Figure 5.10 shows the different cases that can appear when comparing two buckets B_1 and B_2. There may be no overlap between the buckets or the buckets may partially overlap or one bucket may completely cover the other bucket. Furthermore, its important which of the two buckets contains the lower values and which of the two buckets contains the higher values. Figure 5.10 also shows how the comparison operation *less equal* modifies the boundaries of the two input buckets resulting in the modified buckets B'_1 and B'_2.

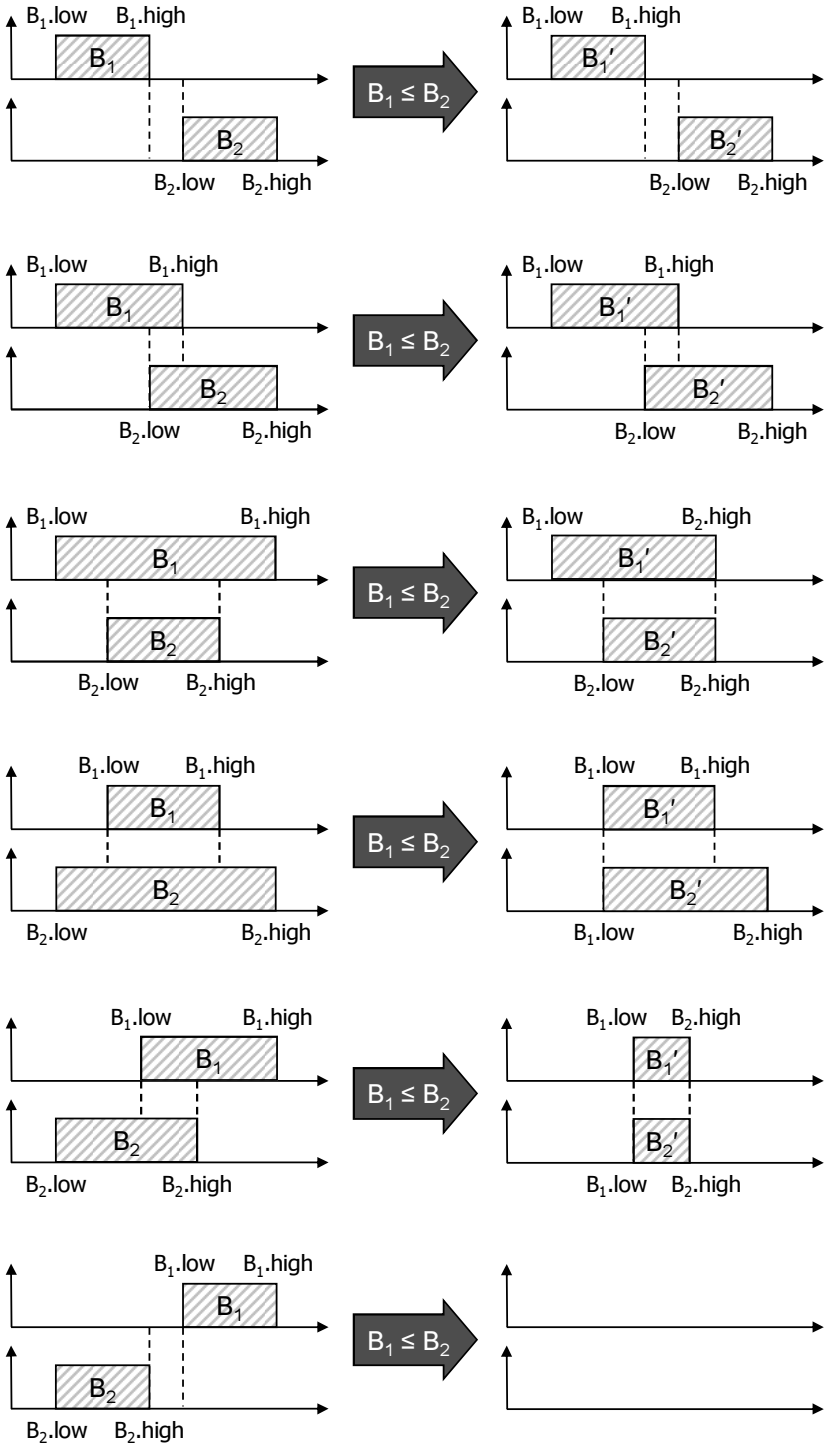


Figure 5.10: Different cases that can appear when applying the comparison operation *less equal* to two buckets B_1 and B_2.

Less

The only difference between the comparison operation *less equal* and the comparison operation *less* is that for the comparison operation *less equal* the area of value combinations that satisfy the predicate does not include the bisecting line. This has to be considered when computing subarea 4 (see Figure 5.9).

For integer histograms, we even can exploit the comparison operation *less equal* to realize the comparison operation *less*. This is due to the fact that $E_1 < E_2$ equals $E_1 + 1 \leq E_2$. Therefore, when we first add a constant with value 1 to the histogram of the left expression, afterwards apply the comparison operation *less equal* instead of the comparison operation *less*, and finally subtract a constant with value 1 from the modified histogram corresponding to the left expression, we get the expected result.

5.2.4 Grouping

Grouping is a unary operation that has one input histogram relation and one output histogram relation. Additionally, it has a list of grouping expressions, a list of aggregate expressions, and a corresponding list of attributes as parameters. When we concatenate the list of grouping expressions and the list of aggregate expressions, we get a new list where each element in this new list builds a pair with the attribute at the same position in the list of attributes. Furthermore, each attribute in the list of attributes represents an attribute in the output histogram relation.

In the relational algebra of multi-relations, the grouping operation is used to summarize all tuples that are equal in the values of the grouping expressions to a single group tuple. Simultaneously, the aggregates specified by the list of aggregate expressions are calculated over the tuples of a group. Aggregate expressions are aggregate functions applied to arithmetic expressions. We adapt this to histogram relations, i.e., the histograms in the output histogram relation of the grouping operation represent the content of the grouping expressions and aggregates after grouping.

Due to the *Attribute Value Independence Assumption*, each possible combination of histogram buckets with respect to the grouping attributes has to be considered when computing the groups and their aggregates. For each such bucket combination, we first compute the number of groups that can be built from this bucket combination and the average size of these groups. Afterwards, we compute the aggregates for each bucket combination, i.e., we compute the aggregates for the groups that can be built from each bucket combination.

The calculation of the amount of groups for a bucket combination and the corresponding group size is based on the *Uniform Distribution Assumption* and on the *Attribute Value Independence Assumption*. Thus, the amount of groups *groupcount* is the product of the number of distinct values of all the buckets in a bucket combination $\langle B_1, ..., B_n \rangle$ where n is the number of grouping expressions (B_1 is a bucket in the histogram of the first expression in the list of grouping expressions, B_2 is a bucket in the histogram of the second expression in the list of grouping expressions, and so on):

$$groupcount = \prod_{B_i \in \{B_1,\ldots,B_n\}} B_i.dv$$

The average size of these groups *groupsize* is calculated similarly to the calculation of the cardinality of a bucket combination in the selection and projection operation:

$$groupsize = HR.card \cdot \prod_{B_i \in \{B_1,\ldots,B_n\}} \frac{B_i.card}{HR.card} \cdot \frac{1}{B_i.dv} = \frac{HR.card \cdot \prod_{B_i \in \{B_1,\ldots,B_n\}} \frac{B_i.card}{HR.card}}{groupcount}$$

When the number of distinct values in the grouping expressions is high, but the cardinality of the input histogram relation is low, it is possible that the maximum number of groups for a bucket combination is greater than the cardinality of this bucket combination in the output histogram-relation. In this case, the *groupsize* is less than 1 and this indicates that not all possible value combinations / groups really appear in the input relation, because the size of a group which is the number of tuples that build this group can practically not be less than 1. Therefore, we provide some correction step for this problem. This means, we reduce the number of groups such that we get a *groupsize* of 1 ($groupcount'$ and $groupsize'$ represent the values of *groupcount* and *groupsize* after the correction step):

$$\begin{aligned} groupcount' &= groupcount \cdot groupsize \\ groupsize' &= 1 \end{aligned}$$

The calculation of the aggregate values is not trivial, it's a sophisticated combinatorial problem. For example, given a histogram for an attribute A, an aggregate expression $SUM(A)$, and an average group size of 10, the data distribution of the aggregation result has to reflect all the combinations to take 10 values out of the value set of attribute A which is represented by the given histogram. In our case, this becomes even more difficult, because we store bucket cardinalities as decimal numbers. Moreover, real-world data always contains noise and a deviation from the *Uniform Distribution Assumption* within a bucket. Hence, there is also some deviation in the group sizes for a single bucket combination. To overcome this problem, we developed some heuristics to calculate the histograms resulting from the application of the aggregate functions $COUNT$, SUM, and AVG as long as the attribute or arithmetic expression to be aggregated contains none or only negligibly few *null* values. In the following, we highlight the heuristics used for the different aggregate functions. The aggregate functions get a histogram as input as well as a list that contains the values *groupcount* and *groupsize* for each bucket combination of the grouping expressions. The histogram represents the data distribution of the argument of the aggregate function. When this argument is not just an attribute but an arithmetic expression, we first have to compute the histogram of this arithmetic expression according to the computation of result histograms for arithmetic expressions in projections (see Section 5.2.2).

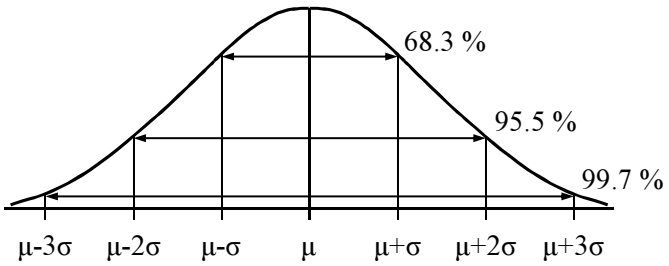


Figure 5.11: Normal distribution / Gaussian distribution.

COUNT and COUNT(*)

A trivial solution is to take the average group size for each bucket combination as value for *COUNT(*)*. This means, we would add a bucket for each bucket combination where the lower bound and the upper bound equals *groupsize* and the cardinality equals *groupcount*. However, real world data often presents some deviation in group sizes. Since many measurements ranging from psychological to physical phenomena can be approximated to varying degrees by the normal distribution [Wik], we also presume that the distribution of the group sizes follows the normal distribution with the average group size *groupsize* as mean μ. Figure 5.11 shows how the normal distribution which is also called Gaussian distribution looks like. The figure also depicts that about 68.3% of the values lie within one standard deviation of the mean, about 95.5% of the values are within two standard deviations and about 99.7% lie within three standard deviations. We approximate this distribution by five buckets with the following ranges, where σ denotes the standard deviation: $[\mu - 3\sigma \; .. \; \mu - 2\sigma)$, $[\mu - 2\sigma \; .. \; \mu - \sigma)$, $[\mu - \sigma \; .. \; \mu + \sigma]$, $(\mu + \sigma \; .. \; \mu + 2\sigma]$, and $(\mu + 2\sigma \; .. \; \mu + 3\sigma]$. However, we have to make use of some heuristics to estimate a value for the standard deviation, because the histogram data does not capture the standard deviation within buckets. We approximate the standard deviation by the square root of the average group size *groupsize*.

The aggregate function *COUNT* is very similar to the aggregate function *COUNT(*)*. The only difference is that *COUNT(*)* counts all tuples of a group, whereas *COUNT* only counts those tuples of a group where the argument of the aggregate function is not equal to *null*. We consider this by reducing *groupcount* proportionally by the number of *null* values in the argument of the aggregate function, before we determine the buckets that represent the distribution of the aggregate values. The computation of the buckets equals the computation of the buckets for the aggregate function *COUNT(*)*.

SUM

Again, we take deviations in the group sizes into account. Thus, we determine the minimum group size as $groupsize - 3\sigma$ and the maximum group size as $groupsize + 3\sigma$. Afterwards, we sum up the buckets with highest values until we reach the maximum group size starting with the bucket which has the highest bounds. Accordingly, we sum up the buckets with lowest values until we reach the minimum group size starting with the

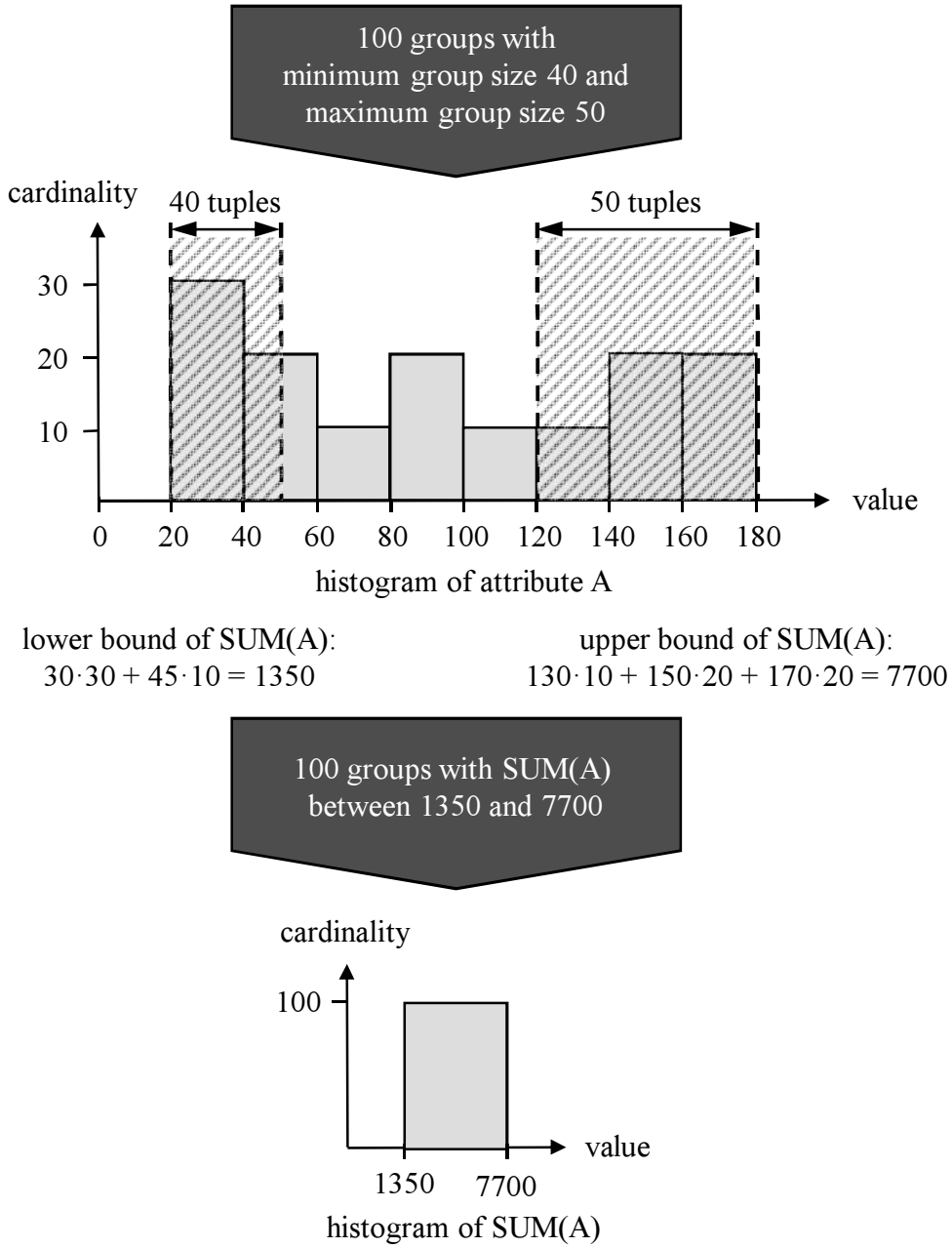


Figure 5.12: Sample of applying aggregate function $SUM(A)$ to a bucket combination $\{B_1, ..., B_n\}$ with $groupcount = 100$, $groupsize = 45$ and $3\sigma = 5$.

null bucket or, if a *null* bucket does not exist, with the bucket having the lowest bounds. In this case, we treat the *null* value as 0. Figure 5.12 shows some sample application, where we presume a bucket combination with $groupcount = 100$, a minimal group size of 40, and a maximal group size of 50. The computation of the SUM aggregate over a single bucket follows the *Continuous Value Assumption*. Thus, the sum of a complete bucket can be computed by multiplying the mean between the lower and upper bound of the bucket by the cardinality of the bucket (see lowest bucket in Figure 5.12, for example). For computing the SUM aggregate over a bucket fraction, we have to take the mean of the range of the bucket fraction instead of the mean of the range of the whole bucket (see second lowest bucket in Figure 5.12, for example).

AVG

First, we compute the minimum SUM aggregate and the maximum SUM aggregate for each bucket combination as described before. Afterwards, we divide the minimum SUM aggregate by the maximum group size and the maximum SUM aggregate by the minimum group size to get the minimum and maximum AVG aggregate values.

5.3 Database Interface

Query sequences can be executed by every relational DBMS that supports the SQL subset specified in Appendix A. Additionally, the cost-based approach requires that the query optimizer of the DBMS is a cost-based optimizer that uses histograms for query optimization. Furthermore, the DBMS has to support the access to meta data and statistics like histograms as well as it has to support the retrieval of cost estimates for a given SQL statement. However, there exists no standardized interface for this purpose. Moreover, each DBMS only supports a special type of histogram and stores this histogram in its own format. Therefore, we define a common interface called *Statistics API* that offers uniform DBMS-independent access to statistics, meta data, and optimizer estimates. It abstracts from the DBMS-specific APIs and data structures. Hence, there has to exist an implementation of Statistics API for each DBMS that should be supported (see Figure 5.13). Such an implementation behaves like a wrapper that maps between the DBMS-independent methods and data structures of Statistics API and the DBMS-specific API and data structures of the target database system.

Statistics API is a JAVA interface. Figure 5.16 shows a UML class diagram of this interface including the method signatures. It is a 'low-level' interface that offers a set of basic methods that can be used to compose more powerful methods consisting of multiple basic method calls. Therefore, Statistics API has an explicit connection management that allows to run multiple methods of Statistics API using the same connection and therefore the same transaction, i.e., the application has to open a connection to the target database, before it can use the access methods, and, afterwards, it has to close the connection again. For this reason, Statistics API includes a *connect* and a *disconnect* method. This may be helpful, e.g., for an application that has to retrieve the names and the statistics for all columns of a given table. This application can open a connection,

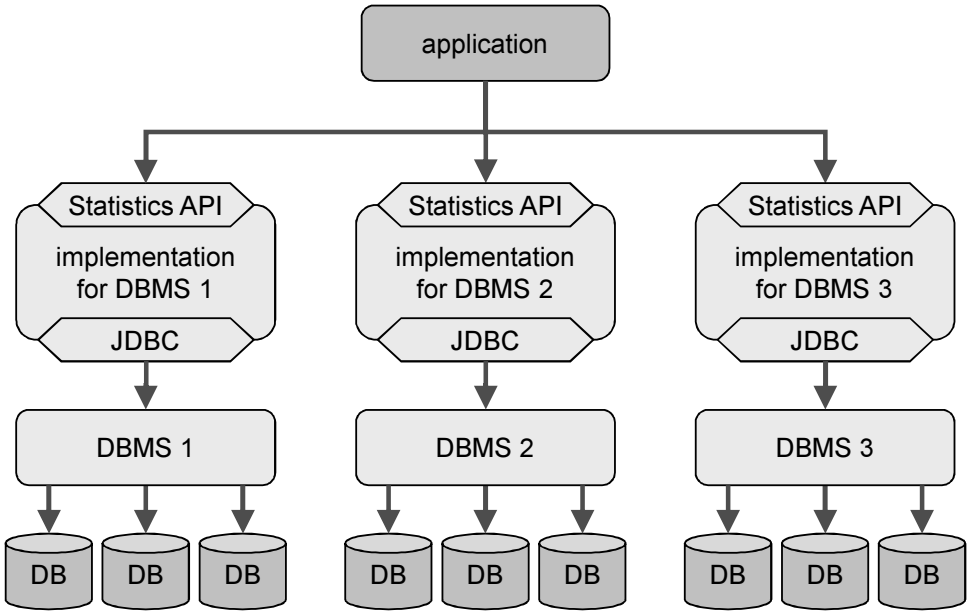


Figure 5.13: Statistics API in an application scenario.

retrieve the column names of the table stored as meta data, retrieve the statistics for each column, and close the connection again. The connections used in Statistics API are ordinary JDBC connections. Therefore, these connections can also be reused by the application for regular data management purposes, when necessary.

In the following, we first explain which statistics, meta data, and optimizer estimates are supported by Statistics API. Furthermore, we introduce the JAVA classes whose instances are used as arguments or return values by the methods of Statistics API. These classes define the data structures to store and exchange statistics, meta data, and optimizer estimates. The description of the classes also contains a description of the methods these classes provide. Afterwards, we focus on the API itself, i.e., we introduce the methods that Statistics API provides to access statistics, meta data, and optimizer estimates in a target database system.

5.3.1 Data Structures

We have analyzed the three commercial DBMSs IBM DB2, Oracle, and Microsoft SQL Server to select the statistics and meta data that should be supported by Statistics API. Each of these three DBMSs provides a huge set of statistics and meta data concerning the different database objects and data structures. Due to the properties of the query sequences that we cover in this work, we focus on meta data and statistics about typical database objects in a relational DBMS such as tables and indexes. Unlike statistics, meta data regarding tables and indexes are very similar in all considered DBMSs. Regarding statistics, we have selected those elements that are supported by at least two of the three analyzed DBMSs. There is a great overlap in logical table statistics and logical

column statistics, i.e., statistics that describe the data distribution within a table and therefore directly influence selectivity and cardinality estimation. However, there are great differences in index statistics, statistics regarding character string data (prefix and postfix patterns) and physical statistics. For example, DB2 and Oracle both have statistics that reflect the degree of index clustering, but their definitions and dimension units are totally different. This is due to the fact that various DBMSs support different index types and that each index type or index implementation has specific statistics.

We grouped the statistics, meta data, and optimizer estimates that should be supported by Statistics API into seven groups according to the type of information they provide and according to the type and granularity of the database object to which they refer: *table statistics*, *column statistics*, *index statistics*, *table meta data*, *column meta data*, *index meta data*, and *estimates*. Figure 5.14 lists the name, the associated JAVA data type, and a short description for each element of these seven groups.

Statistics API provides a JAVA class for each group with the associated elements as attributes. We store elements of primitive data type (like long, int and boolean) in attributes of the associated wrapper classes. This allows to distinguish whether a value has been assigned to this element or not, i.e., if no value has been assigned to it, the associated attribute is set to the JAVA *null* value. Therefore, the classes provide a *get* method and an *isAvailable* method for each of their attributes. The *get* method (e.g. *getCardinality*) returns the attribute value. The *isAvailable* method (e.g. *isCardinalityAvailable*) returns whether a value for the attribute is available or not, i.e., whether the associated attribute is set to the JAVA *null* value or contains a JAVA object. In addition, attributes that represent statistics have a *clear* and a *set* method assigned. The *clear* method (e.g. *clearCardinality*) resets the attribute to the JAVA *null* value. The *set* method (e.g. *setCardinality*) sets the attribute to the new value passed on as argument. *clear* and *set* methods are not available for meta data and optimizer estimates, because the classes associated with meta data and optimizer estimates are only used as return values in the methods of Statistics API but not as arguments (see Section 5.3.2). So, the values stored in these classes needn't be set or changed by the application.

For histogram data, Statistics API provides a data structure that conforms to the histogram definition in Section 3.3. Figure 5.15 shows a UML class diagram comprising all classes related to histograms. Due to readability, we omitted to list the methods of the classes in this figure. Irrespective of their data type, all histograms are represented by instances of the *Histogram* class. The *Histogram* class contains a vector with instances of the *Bucket* class as elements. Instances of the *Bucket* class represent buckets which are defined by a lower bound *low*, an upper bound *high*, a cardinality *card*, and a number of distinct values *dv*. The objects stored in *low* and *high* are instances of subclasses of the abstract superclass *Value*. For each data type that is supported by Statistics API, an appropriate class exists (e.g. *CharValue*, *DateValue*, *DecimalValue*, *IntegerValue*). These classes cover the specific properties of the corresponding data type. The methods of the *Histogram* class and the methods of the *Bucket* class ensure that all bucket bounds within a histogram have the same data type. The *null* value that may appear instead of a regular value in histograms of any data type is represented by a bucket where the lower and upper bound is set to the JAVA *null* value.

TableStatistics		
cardinality	long	number of rows in the table
pagesAllocated	long	number of pages allocated for the table (used + unused)
pagesUsed	long	number of pages containing rows of the table

ColumnStatistics		
avgLength	long	average length of the column (in bytes)
histogram	Histogram	a histogram that approximates the data distribution of the column

IndexStatistics		
distinctKeys	long	number of distinct values in the index key columns
distinctKeysFirstNColumns	long[]	array containing the number of distinct values for several prefixes of the index key columns*
leafPages	long	number of index leaf pages
levels	long	number of index levels

TableMetaData		
columns	String[]	array containing the column names of the table
indexes	String[][]	array containing a pair for each index on the table which consists of the name and the schema of the index
pageSize	long	size of a page in the associated tablespace (in bytes)

ColumnMetaData		
dataTypeName	String	name of the data type of the column (DBMS-dependent)
sqlDataTypeNumber	int	SQL data type of the column from *java.sql.Types*
histogramDataTypeNumber	int	data type of the histogram (indicates the *Value* class)
size	int	maximum number of characters for columns which store strings or dates, precision for columns which store numbers
digits	int	number of fractional digits
nullable	boolean	*true* if the column allows *null* values, *false* otherwise

IndexMetaData		
indexColumns	String[]	array containing the column names of the index key
indexColumnsAscending	boolean[]	array containing the order of the values in the columns of the index key; *true* for ascending order, *false* for descending order
includedColumns	String[]	array containing the names of the columns that are included in the index, but that are not part of the index key
unique	boolean	*true* if the index key is unique, *false* otherwise
clustered	boolean	*true* if the index is a clustered index, *false* otherwise
pageSize	long	size of a page in the associated tablespace (in bytes)

Estimates		
cardinality	double	cardinality estimate for the result of the given statement
cost	double	cost estimate for the given statement (DBMS-dependent)

Figure 5.14: Statistics, meta data, and optimizer estimates supported by Statistics API.

* The i-th entry of the array stores the number of distinct values for the subset consisting of the first i columns of the index key. For example, for an index key $C_1C_2C_3C_4$ the array *distinctKeysFirstNColumns* contains the number of distinct values for the prefixes C_1, C_1C_2, $C_1C_2C_3$ and $C_1C_2C_3C_4$. Some DBMSs (like DB2) store the number of distinct values only for a limited number of prefixes. In that case, the array contains only the values of the available prefixes.

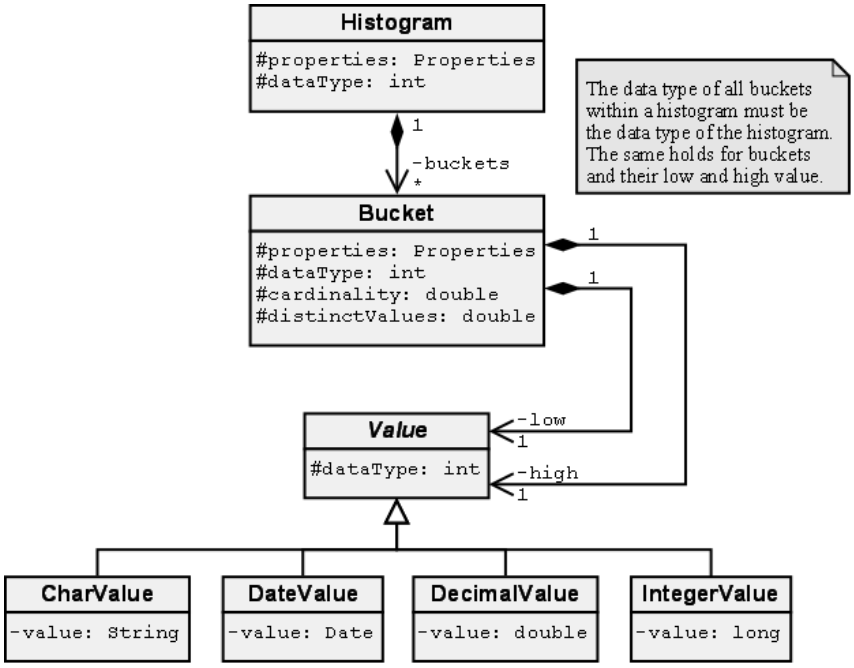


Figure 5.15: Class diagram of the classes related to histograms (methods are omitted).

Besides the methods for adding buckets, the *Histogram* class contains methods to transform the stored histogram into a serial histogram, an equi-width histogram, or an equi-height histogram, and methods to reduce the number of buckets by merging them. The corresponding algorithms rely on the *Uniform Distribution Assumption* and on the *Continuous Value Assumption*. These transformation methods are necessary because some DBMSs can only store histograms of a certain type or size.

5.3.2 API Methods

Figure 5.16 shows a UML class diagram of the JAVA interface that contains the access methods of Statistics API. For each statistics class, there exists a *delete* method, a *get* method, and a *set* method in Statistics API. The *delete* method (e.g. *deleteTableStatistics*) deletes existing statistics data in the target database. The *get* method (e.g. *getTableStatistics*) retrieves statistics from the target database and returns an appropriate statistics object. Some of the statistic values in the returned statistics object may be unavailable, because not every DBMS supports all statistics covered by the statistics object or, because some statistics may not have been gathered yet. In this case, the associated attribute is set to the JAVA *null* value (see Section 5.3.1). The *set* method (e.g. *setTableStatistics*) replaces the statistics values in the database with the new values in the appropriate statistics object which we pass as argument to the method. For all statistics where the value of the associated attribute in the statistics object equals the JAVA *null* value, the value of the associated statistic will not be deleted in the database, but the old value will be kept. For meta data, only *get* methods are available, because

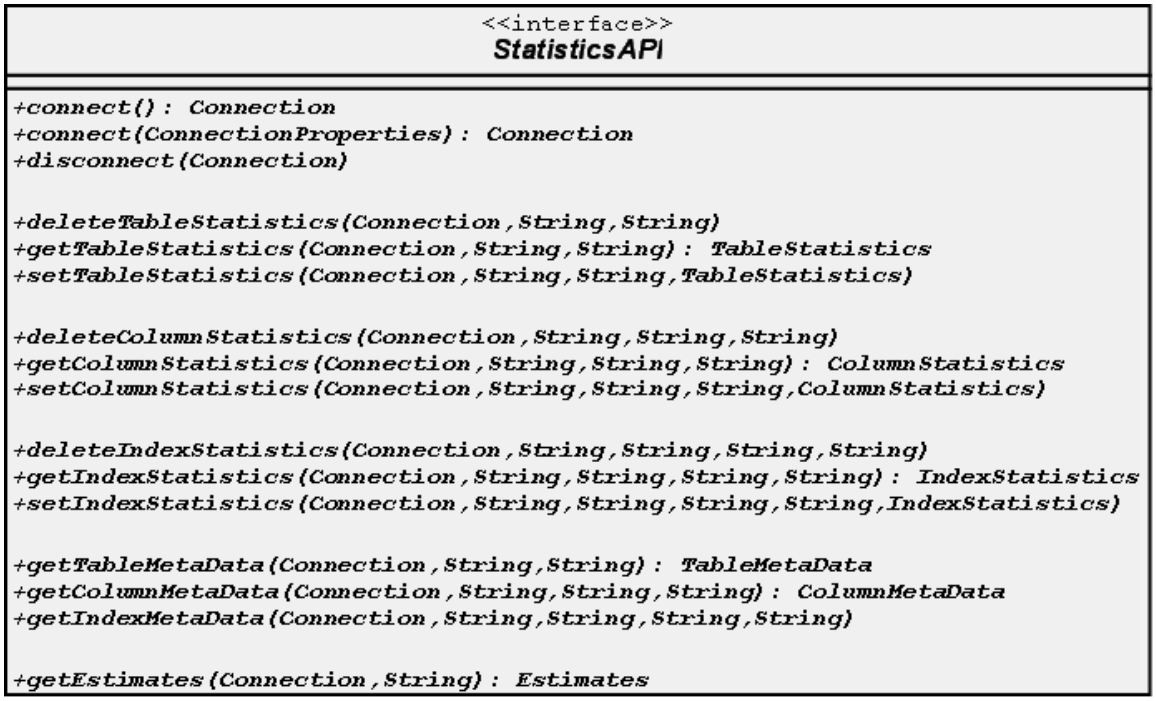


Figure 5.16: Methods of the Statistics API interface.

these data contains information about the database structure and physical layout that cannot be changed directly from outside. The same holds for optimizer estimates.

The methods of Statistics API are not case-sensitive regarding identifiers of database objects such as table names or schema names, i.e., identifiers of database objects can be passed on in lower-case letters, upper-case letters, or a mixture of both. Since indexes and statements which create, alter, or drop indexes are still not part of the SQL standards, different alternatives to identify an index exist. In some DBMSs, an index is a separate database object that can be stored in another schema than the associated table. In this case, an index is identified by its name and schema. In other DBMSs, an index belongs to the associated table and therefore is identified by its name and the name and schema of the associated table. To support both alternatives of index identification, the index-related methods of Statistics API require the name and schema of the index as well as the name and schema of the associated table as input. Even when an index can be identified by its name and schema in the target DBMS, an interface implementation should check whether the given name and schema of the associated table is correct. When the index is identified by its name and the associated table, an implementation should ignore the value passed on as index schema.

Statistics API provides its own exception classes *StatisticsAPIException*, *NoSuchObjectException* and *UnsupportedMethodException*. The two latter ones are specializations of the first one. A *NoSuchObjectException* is being thrown when the object addressed by the input parameters of a method does not exist. An *UnsupportedMethodException* is being thrown when a method of Statistics API is being called that is not supported by

the current implementation. Additionally, the histogram-related classes provide their own exception class *HistogramException*. When such an exception occurs during the execution of an API method, it is being encapsulated into a *StatisticsAPIException* object by this method.

5.4 Control Strategy

```
Input:   A query sequence S.
Output:  Sequence with lowest costs.

1    SearchSpace ← ∅;
2    NextSequences ← {S};

3    while NextSequences ≠ ∅ do begin
4        SearchSpace ← SearchSpace ∪ NextSequences;
5        PreviousSequences ← NextSequences;
6        NextSequences ← ∅.
7        foreach sequence S_i in PreviousSequences do begin
8            foreach rule R in CGO rule set do begin
9                NewSequences ← result of applying R to S_i;
10               NextSequences ← NextSequences ∪ NewSequences;
11           end;
12       end;
13       NextSequences ← NextSequences − SearchSpace;
14   end;

15   OptimalSequence ← sequence in SearchSpace with lowest costs;

16   return OptimalSequence;
```

Figure 5.17: Exhaustive control strategy.

The objective of a cost-based control strategy is to find the query sequence with lowest costs. A straightforward solution enumerates the complete search space with respect to the CGO rewrite rules, retrieves costs for each query sequence in the search space, and selects that query sequence from the search space which has lowest costs. When multiple query sequences exist with lowest costs, the control strategy has to choose one of them randomly or with respect to any other characteristics of the query sequences. Figure 5.17 shows the corresponding algorithm. In each iteration step, it tries to apply each rule once to each query / query combination within the query sequences resulting from the previous iteration step. Since a single rule possibly can be applied to multiple

queries / query combinations within a query sequence, the application of a rule to a query sequence results not just in a single new sequence but in a set of new sequences (line 9). The query sequences which are the result of the previous iteration step build the set *PreviousSequences* (line 5) and the results of the current iteration step build the set *NextSequences* (line 10). Before the algorithm can proceed with the next iteration step, it has to remove those sequences from *NextSequences* which are already part of the current search space (line 13). Otherwise, depending on the concrete rule set, the algorithm could end up in an infinite loop due to cyclic rule applications among the sequences. When *NextSequences* is still not empty after removing the sequences contained in the current search space (line 3), the algorithm proceedes with the next iteration step and adds the sequences within *NextSequences* to the search space (line 4). Now, the content of *NextSequences* becomes the content of *PreviousSequences* (line 5) and *NextSequences* becomes the empty set in order to store the new sequences of this new iteration step (line 6).

```
Input:   A query sequence S.
Output:  Sequence with lowest costs.

1    OptimalSequence ← S;

2    do
3        PreviousSequence ← OptimalSequence;
4        NextSequences ← ∅;
5        foreach rule R in CGO rule set do begin
6            NewSequences ← result of applying R to PreviousSequence;
7            NextSequences ← NextSequences ∪ NewSequences;
8        end;
9        if NextSequences ≠ ∅ then begin
10           S_min ← sequence in NextSequences with lowest costs;
11           if costs of S_min ≤ costs of OptimalSequence
12           then OptimalSequence ← S_min;
13       end;
14   while OptimalSequence ≠ PreviousSequence;

15   return OptimalSequence;
```

Figure 5.18: Greedy control strategy.

Since an exhaustive algorithm is not practical for large search spaces, we also introduce a greedy algorithm that significantly reduces the search space. The greedy algorithm presumes that we get the query sequence with lowest costs, when we always apply that rule of the rule set to the current sequence which reduces the costs the most. Figure 5.18

shows the corresponding algorithm. In a loop, the control strategy once applies each rule whose condition is satisfied to the current query sequence (line 5 to 8) and selects that sequence from the resulting query sequences as the sequence for the next iteration step which has lowest costs (line 9 to 13). When multiple query sequences exist with lowest costs, the control strategy has to choose one of them randomly or with respect to any other characteristics of the query sequences. The control strategy exits the loop when the sequence of the next iteration step is the same as the sequence of the current iteration step, i.e., when none of the rules could be applied to the current sequence or when none of the sequences resulting from rewriting has less or equal costs than the original sequence (line 14). However, since the greedy algorithm only considers a subset of the complete search space, it may find a local optimum in the search space instead of the global optimum.

Besides these two control strategies, there are some other control strategies that prune the search space such as simulated annealing. More details about these strategies can be found in literature on search space problems and in literature on cost-based query optimization (e.g., see [Cha98] for some references). In combination with the control strategy, it is also possible to optimize the cost estimation process. This means that it is advisable to cache the costs retrieved for the statements of a sequence, since the retrieval of the costs from the underlying database system is time-consuming. So, in the best case, costs must only be recomputed for those parts of a query sequence that are modified by a rewrite rule, but costs for those parts that stay unchanged can be taken from the cache.

5.5 Prototype

This section addresses the prototypic implementation of the cost estimation component for query sequences. Besides the cost estimation component, our prototype called *CEOPS* [Kra07] includes an SQL parser, Statistics API implementations, and a GUI. The SQL parser translates an incoming query sequence into a query dependency graph. Then, the cost estimation component uses the parsed representation of the INSERT statements in the query dependency graph as basis to transform them into a relational algebra tree for histogram propagation. The prototype provides implementations of Statistics API for three commercial database systems. The GUI explains the cost estimation process and in particular the histogram propagation process. Therefore, it has access to the subcomponents of the cost estimation component. See Figure 5.19 for an architectural overview of the CEOPS system.

In the following, we concentrate on the GUI, the mapping from SQL to relational algebra as it is being used within our prototype, and some implementation issues concerning the three Statistics API implementations.

5.5.1 GUI

The GUI has been developed for demonstration purposes. It explains the cost estimation process and in particular the histogram propagation process. This means, it displays the tree of algebraic operations that we obtain by translating the queries of the INSERT statements of a given sequence into our algebra. Each algebraic operation provides a set

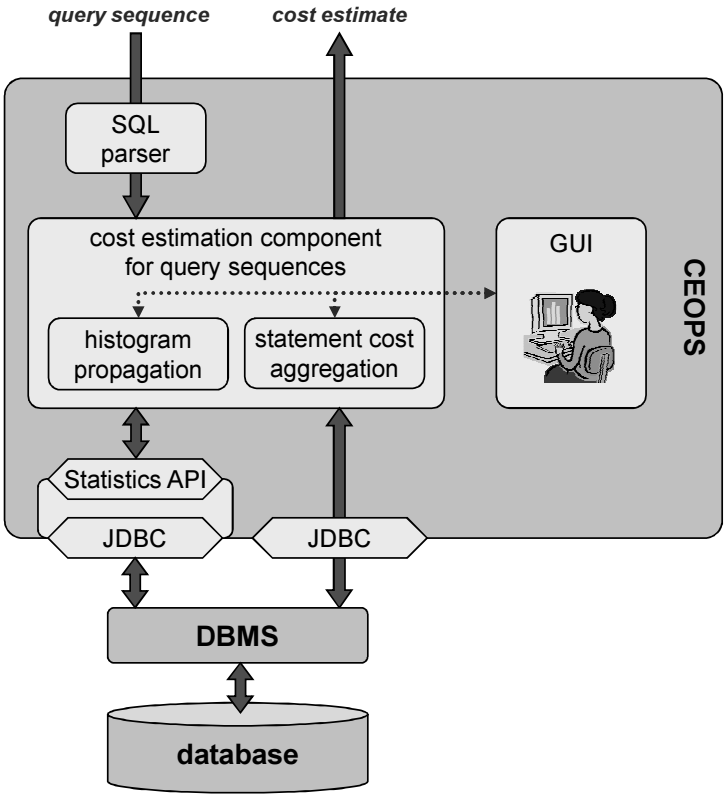


Figure 5.19: Architecture of the CEOPS system.

of attributes and a corresponding histogram for each attribute in its output histogram relation. The GUI is able to display these histograms. Moreover, the user can choose between different representations, i.e., for readability, the histogram resulting from propagation may be transformed into an equi-width or an equi-depth histogram. Besides, the GUI presents operation-specific information. Furthermore, for each INSERT statement in the sequence, the GUI reports the cost estimate and cardinality estimate retrieved from the optimizer of the underlying database system as well as the cardinality estimate obtained by histogram propagation. The GUI also allows to modify configuration parameters concerning the algorithms used for histogram propagation. For example, you can turn on and off the normalization step and you can vary the maximum number of buckets concerning normalization.

5.5.2 Mapping SQL to Relational Algebra

The mapping implemented in the prototype is similar to the mapping described in Section 3.2.3. Additionally, we take into account that the same arithmetic expression may appear multiple times in different clauses of the same SQL statement, i.e., we consider common subexpressions when mapping SQL to relational algebra. The goal is to improve the quality of the histograms resulting from histogram propagation.

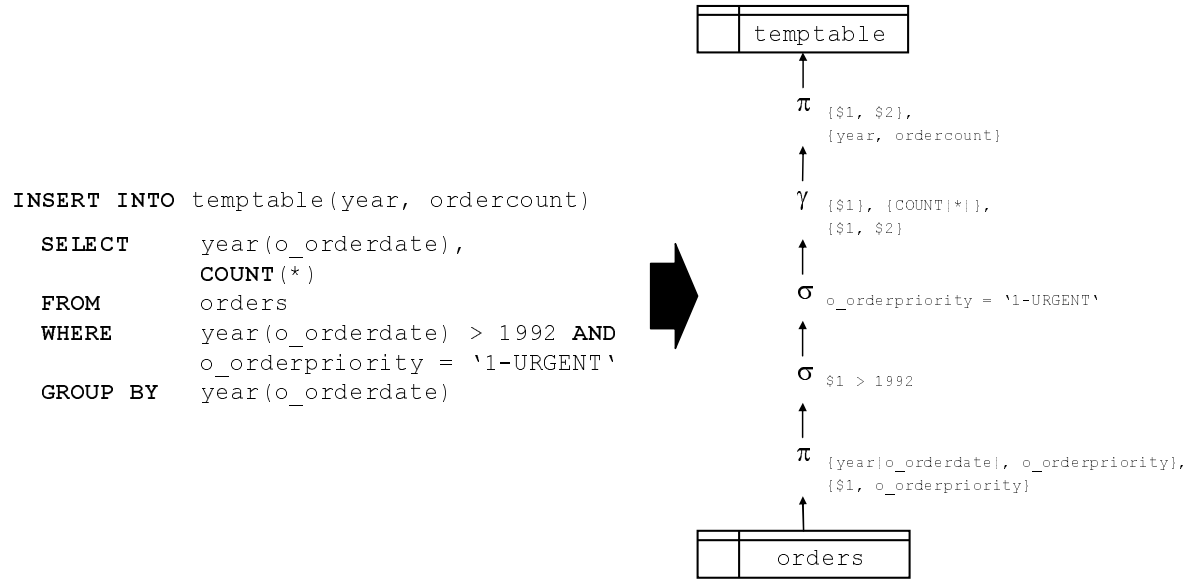


Figure 5.20: Mapping SQL to relational algebra with respect to common subexpressions.

Figure 5.20 shows an INSERT statement where arithmetic expression *year*(*o_orderdate*) occurs three times in different clauses of the query which is the body of the INSERT statement. When we would not take this into account, we would have to compute the result of this arithmetic expression each time it appears and these computations would always be based on the same relative data distribution. This relative data distribution is the relative data distribution of the histogram that belongs to attribute *o_orderdate* in table *orders*. This means, the histogram which represents the result of the arithmetic expression *year*(*o_orderdate*) always contains not only buckets, where the values of the boundaries are greater than 1992, but buckets with the boundaries of all years that appear in the histogram of attribute *o_orderdate*.

In the prototypic implementation, we account for such common subexpressions. For this reason, we add appropriate projections in between the operations where necessary and replace common subexpressions by internal attributes as early as possible within the tree. Thus, a histogram can be assigned to each of these expressions and this histogram can be reused each time the corresponding expression occurs in the original tree of operations and it can be adapted by all subsequent operations. In Figure 5.20, we add an additional projection to the original tree of operations below the first selection. This projection replaces the arithmetic expression *year*(*o_orderdate*) by the internal attribute $1. Thus, we also replace all subsequent occurrences of this arithmetic expression by the internal attribute $1. So, the inserted projection computes a histogram for the arithmetic expression *year*(*o_orderdate*) based on the histogram of attribute *o_orderdate* and this histogram is being reused and adapted by all subsequent operations. This means that the histogram of *year*(*o_orderdate*), which is part of the output histogram-relation of the projection that represents the SELECT clause, contains only buckets where the values of the boundaries are greater than 1992.

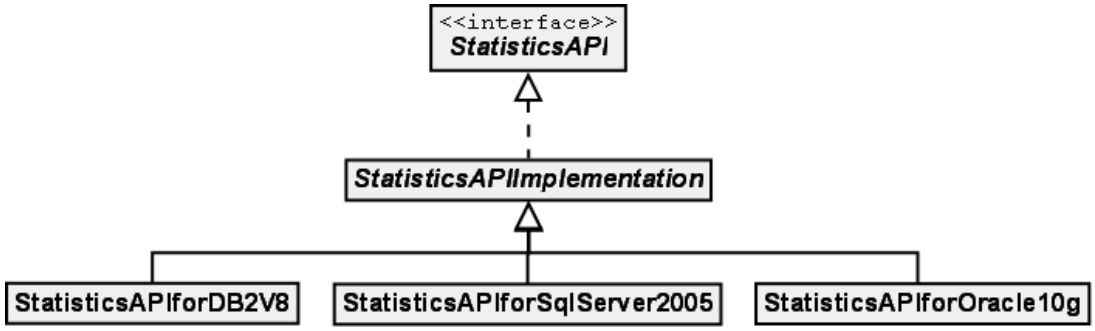


Figure 5.21: Class diagram showing the Statistics API interface and the inheritance between the implementing classes (attributes and methods are omitted).

5.5.3 Database Interface

The prototype comprises interface implementations for the three commercial database systems IBM DB2, Oracle, and Microsoft SQL Server. As the UML class diagram in Figure 5.21 shows, the implementation classes do not directly implement the *StatisticsAPI* interface but inherit from the abstract superclass *StatisticsAPIImplementation*, which implements the *StatisticsAPI* interface. *StatisticsAPIImplementation* implements some common behavior such as creating and closing a JDBC connection and some helper methods that are presumably useful for all implementations of Statistics API. The three interface implementations in the prototype solely use JDBC to communicate with the target database but no further tools or libraries. They behave like wrappers that map the DBMS-independent methods of Statistics API to one or more DBMS-specific SQL statements. These SQL statements directly access catalog tables or call stored procedures in the target database that represent proprietary APIs provided by the DBMS. To perform the SQL statements, the account used for the connection to the target database system must own the necessary authorizations. Which authorities are necessary, depends on the underlying DBMS and whether you want to use the full functionality of Statistics API or only a non-critical part of it. There is only one exception where the implementations do not directly access the catalog tables or call stored procedures. When retrieving column meta-data, the implementations make use of the JDBC DatabaseMetaData interface due to exploit the functionality of mapping DBMS-specific data types to the data types of *java.sql.Types*. However, in all other methods of Statistics API, the implementations do not make use of the JDBC DatabaseMetaData interface.

In the following sections, we discuss the capabilities of the three analyzed DBMSs to access statistics, meta data, and optimizer estimates, and we show how to use these capabilities in Statistics API implementations. For these purpose, we studied the manuals and optimization-related literature offered by the DBMS vendors. However, the management of statistics is often sparsely documented and examples that show how to use these features are hard to find. A short description of the catalog tables of IBM DB2 can be found in the IBM DB2 Universal Database SQL Reference [IBM04b]. Further information, about the usage of statistics in the optimizer and how to update the statistics tables,

can be found in the IBM DB2 Universal Database Administration Guide: Performance [IBM04a]. Similarly, information about the usage of statistics in Oracle and their storage in the catalog tables can be found in the Oracle Database Reference [Ora03b] and in the Oracle Database Performance Tuning Guide [Ora03a]. The package DBMS STAT which enables to modify and delete statistics is documented in the PL/SQL Packages and Types Reference [Ora03c]. Additional information can be found in Oracle's web forum AskTom [Ora]. For Microsoft SQL Server, we made use of the Transact-SQL Reference [Mic06], which is available online at the Microsoft Developer Network (MSDN).

5.5.3.1 IBM DB2 V8.2 Implementation

DB2 offers no special API to access statistics data and meta data, but allows for direct access to the appropriate catalog views. Thus, retrieving this data can simply be realized by querying these catalog views. As the columns regarding statistics are updatable in the catalog views, the deletion or modification of statistics can be realized by updating these views. However, DB2 does some consistency checks when updating these views to avoid serious inconsistencies within the database catalog. Thus, the order in which the tuples are updated is crucial. The tuples that contain the statistics as attribute values already exist in the catalog views and do not have to be created. As long as no statistics have been gathered, the corresponding attributes contain a default value that marks them as not available. However, this is different for histograms. DB2 stores histograms in the form of quantiles. Each quantile stores a cumulative frequency for the upper bound of a bucket and this cumulative frequency is the sum of frequencies of the actual bucket and all previous buckets. The tuples that contain the quantile data do not exist initially in the corresponding catalog view *syscat.coldist*. They are created by the DBMS as soon as statistics are being gathered and statistics are being gathered when the RUNSTATS command is being called. As direct insertions into *syscat.coldist* are not possible, the RUNSTATS command has to be called to create the tuples and, afterwards, the attribute values of these tuples have to be modified to store a given histogram. The DB2 implementation of Statistics API converts between the bucket-based format used by Statistics API and the quantile-based format used by DB2. When a histogram should be stored in DB2 that contains more buckets as there are tuples available for quantile data, the implementation has to compact the histogram by merging adjacent buckets which is a functionality provided by the *Histogram* class. Since DB2 doesn't store the lowest and highest, but the second-lowest and second-highest value of a column in its catalog views, we treat the second-lowest as the lowest and the second-highest as the highest value in the prototypic implementation.

We use the following views to retrieve and modify statistics data: *syscat.tables*, *syscat.columns*, *syscat.coldist*, and *syscat.indexes*. And these are the views that contain the relevant meta data: *syscat.tables*, *syscat.columns*, *syscat.indexes*, *syscat.indexcoluse* and *syscat.tablespaces*. The last one is used to retrieve the page size of tables and indexes. The prototypic implementation just accesses the quantile data in the catalog view *syscat.coldist*, but it does not consider the list of most frequent values, which is stored in the same catalog view, too. Finally, we use the EXPLAIN tool to get cost and cardinality

estimates. We call EXPLAIN with a given SQL statement and query the cost and cardinality information by joining the EXPLAIN tables *explain_statement*, *explain_operator*, and *explain_stream*.

The identifiers of database objects such as schema names, table names, column names, and index names are stored in upper-case letters in the catalog tables of DB2. Therefore, the DB2 implementation of Statistics API converts the identifiers passed on as arguments to upper case, before they are used for comparison within a query. In DB2, indexes can be identified by their name and schema. Anyhow, the implementation also asks for the correct name and schema of the associated table.

The API implementation for IBM DB2 V8.2 also works with IBM DB2 V9. However, the methods *deleteTableStatistics* and *deleteIndexStatistics* do not reset the values of the statistics columns that have been added in IBM DB9 V9.

5.5.3.2 Oracle 10g Implementation

In an Oracle database system, statistics can reside in two different locations: in the database catalog tables or in tables created in the user's schema for this purpose. Due to the fact that only statistics stored in the database catalog have an impact on the cost-based optimizer, the Oracle implementation of Statistics API only operates on the database catalog. Oracle provides a special package, called DBMS_STATS, to gather and manage statistics data. Furthermore, statistics data and meta data can be retrieved by directly querying the appropriate catalog views. However, these views are not updatable. Except for the retrieval of histogram data, the implementation uses the procedures of the DBMS_STATS package to retrieve, delete, and modify statistics data in the database. We do not use DBMS_STATS for histogram retrieval, because the associated procedure returns the histogram data in VARRAYs which can only be retrieved when using Oracle's extensions to JDBC. Instead, we decided to query the appropriate statistics table *all_tab_histograms* instead of using Oracle's extended JDBC classes. As Oracle only supports equi-width histograms, the Oracle implementation of Statistics API converts a histogram, passed on as parameter to the *setColumnStatistics* method, into an equi-width histogram, before it calls the corresponding procedure in the DBMS_STATS package. For the retrieval of meta data, we make use of the appropriate catalog views which are: *all_tables*, *all_part_tables*, *all_tab_columns*, *all_indexes*, *all_part_indexes*, *all_ind_columns*, *all_ind_expressions*, and *user_tablespaces*. Finally, we use the EXPLAIN PLAN command to retrieve the cost and cardinality estimates for a given SQL statement.

As Oracle supports partitioning of tables and indexes, statistics can be collected separately for each partition as well as for the whole table or index. However, the prototypic implementation of Statistics API for Oracle has no special support for partitioning, i.e., it reads and manipulates the statistics of the whole table but not the statistics of the partitions. The only methods that also affect the partition statistics are the *delete* methods, because they cascade to each partition. Since the index statistic *distinctKeysFirstNColumns* is not available in Oracle, it is also not available in *IndexStatistics* objects returned by *getIndexStatistics* and it is not considered when setting index statistics. Furthermore, the prototypic implementation does not support object tables and nested tables.

However, unlike JDBC DatabaseMetaData, the implementation considers cluster indexes. Therefore, a *TableMetaData* object returned by the *getTableMetaData* method contains a cluster index when the target table is part of a cluster. Moreover, *getTableMetaData* retrieves the page size for 'normal' tables as well as for index-organized tables (IOT) and for partitioned tables as well as for non-partitioned tables. This is not trivial because, depending on the kind of table and depending on partitioning, the information of the page size is stored in different views. Retrieving the columns of an index meets another problem. When columns in an index are denoted to be sorted in descending order, Oracle internally adds a hidden column to the associated table. This hidden column is defined by an expression which equals the original column, but which signals Oracle that the values in this column are sorted in descending order. Therefore, in this case, the hidden column is part of the index key instead of the original column. We consider this and return the column name in the expression that defines the hidden column by removing the surrounding quotation marks. For indexes defined on expressions, we also return the expression instead of the column name in the *IndexMetaData* object.

The identifiers of database objects such as schema names, table names, column names, and index names, are stored in upper-case letters in the catalog tables of Oracle. The procedures of the DBMS_STATS package also ask for identifiers in upper-case letters. Therefore, the Oracle implementation of Statistics API converts the identifiers passed on as arguments to upper case, before they are used for comparison within a query or as arguments in a procedure call. In an Oracle database, indexes can be identified by their name and schema. Anyhow, the implementation also asks for the correct name and schema of the associated table.

To get access to the full functionality of Statistics API, the account used by the application to connect to the database must own the necessary rights to access catalog views and to execute the procedures in the DBMS_STATS package.

5.5.3.3 Microsoft SQL Server 2005 Implementation

In SQL Server, the access to catalog views is limited to retrieval. However, the manipulation of statistics data is possible but not documented anywhere. Hence, the implementation of Statistics API for Microsoft SQL Server 2005 is restricted to the *get* methods. When a *delete* or *set* method is being called, the implementation throws an *UnsupportedMethodException*.

Many of the system tables from earlier versions of SQL Server are now implemented as a set of compatibility views, but they are intended for backward compatibility. Therefore, the implementation makes use of the catalog views introduced in SQL Server 2005. To retrieve statistics data, we access the following catalog views: *sys.dm_db_partition_stats*, *sys.stats*, *sys.stats_columns*, *sys.columns*, *sys.indexes*, and *sys.index_columns*. Due to the fact that most of the column and index statistics are not available in these catalog views, we have to call the DBCC command SHOW_STATISTICS (DBCC stands for Database Console Commands). The DBCC command SHOW_STATISTICS returns current statistics stored in the database for a given index or statistics group. A statistics group defines a group of columns for which the database stores statistics, which can be used by the

query optimizer. Similar to indexes, a column can be part of multiple statistics groups and a statistics group can contain multiple columns. For both, indexes and statistics groups, SQL Server stores the density (reciprocal of the number of distinct rows) and the average row length for the prefixes of the index key and for the prefixes of the group of columns, respectively. Additionally, SQL Server stores a histogram for the first column of the index key and for the first key of the group of columns, respectively. So, to retrieve the histogram for a given column, we first have to query the catalog views for the name of an index or a statistics group where the given column is in the first place. When this query returns multiple occurrences, we choose the latest one, i.e., the one where the statistics have been updated last. Then we call SHOW_STATISTICS with the name of this index or statistics group and get the histogram that we are searching for.

For the retrieval of meta data, we access the following views: *sys.columns*, *sys.indexes*, *sys.index_columns*, and *sys.columns*. To avoid the execution of a query but to get the associated cost and cardinality estimate, we must set SHOWPLAN_ALL ON. Afterwards, when we send an SQL statement to the database, it returns the execution plan in a tabular format as result set including some additional information like cardinality estimates and cost estimates. We read the cardinality estimate and the cost estimate in the first row of the result set and set SHOWPLAN_ALL back to OFF.

Indexes can be identified by their name and the name and schema of the associated table. This is due to the fact that SQL Server tightly couples an index with the associated table. Hence, the same name can be used for multiple indexes as long as they are not associated with the same table.

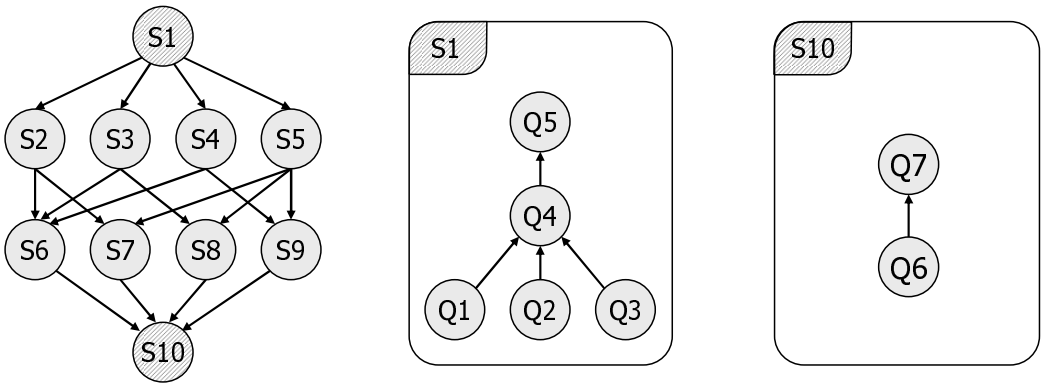
To get access to the full functionality of Statistics API, the account used by the application to connect to the database must own the server role *sysadmin* or the database role *db_owner*. This is due to the fact that some catalog views are only accessible to owners of one of those roles.

5.6 Experiments

In this section, we present the results of some experiments conducted with our prototypic implementation of a cost estimation component for query sequences. The objective of these experiments is to show that histogram propagation can successfully be exploited to provide useful cost estimates for query sequences.

All experiments were performed using the same hardware and software configuration as the experiments with the heuristic optimizer (see Section 4.4), i.e., the database server is a Windows XP machine with two 1.53GHz AMD Athlon 1800+ processors and 1GB main memory. However, for the experiments in this section, we only made use of the DB2 database system. The database is still a TPC-H database which contains TPC-H data created with a scaling factor of 10. Indexes exist on all foreign keys and statistics including histograms have been gathered on all columns of all tables that belong to the TPC-H schema.

Our prototype of a cost estimation component for query sequences did the histogram propagation and provided the cost estimates for the sequences that we examined in our



(a) CGO search space (b) Original sequence S1 (c) Rewritten sequence S10

Figure 5.22: Sequences used in the experiments.

experiments. So, Statistics API was used to retrieve and modify the statistics stored in the underlying database system and to retrieve cost estimates for the statements of a sequence. Remember that DB2 stores histograms as quantiles, i.e., each bucket stores a cumulative frequency including the frequencies of all previous buckets. When retrieving histograms from DB2, Statistics API transforms these quantile data into our own histogram representation. To store the histograms resulting from histogram propagation, Statistics API has to compact histograms with more than 20 buckets and it has to transform the buckets into quantiles. The number of buckets is reduced by merging adjacent buckets with respect to keeping cardinality errors in the data distribution minimal. Moreover, Statistics API also updates the corresponding logical table statistics and logical column statistics in the database catalog.

The left side of Figure 5.22 shows a complete CGO search space containing 10 syntactically different but semantically equivalent query sequences (*S1* to *S10*). Starting point is query sequence *S1*, which has been created by the MicroStrategy DSS tools. The rest of the search space has been built by applying the CGO rewrite rules as implemented in the heuristic optimizer prototype. As the figure shows, sometimes a certain query sequence can be reached by different sequences of rule applications. Query sequence *S10* is the sequence with the most rules applied, i.e., subsequently, we applied three rules to rewrite query sequence *S1* into query sequence *S10*. Figure 5.22 also shows the query dependency graph of the original sequence *S1* and the query dependency graph of sequence *S10*. Query sequence *S1* consists of a final-result query (*Q5*) and four intermediate-result queries (*Q1*, *Q2*, *Q3*, and *Q4*), whereas query sequence *S10* consists of a final-result query (*Q7*) and a single intermediate-result query (*Q6*). Query *Q1* calculates the turnover $(extendedprice \cdot (1 - discount) \cdot (1 + tax))$ for each line item ordered in the year 1992 and sums it up for each customer. Query *Q2* and query *Q3* provide the same for the years 1993 and 1994. Query *Q4* joins these intermediate results and selects those customers that have a turnover that is greater than a given constant parameter value in each of the years. Query *Q5* joins the target table of query *Q4* with the customer table to look up the customer names. Query *Q6* of the rewritten sequence *S10* calculates the turnover

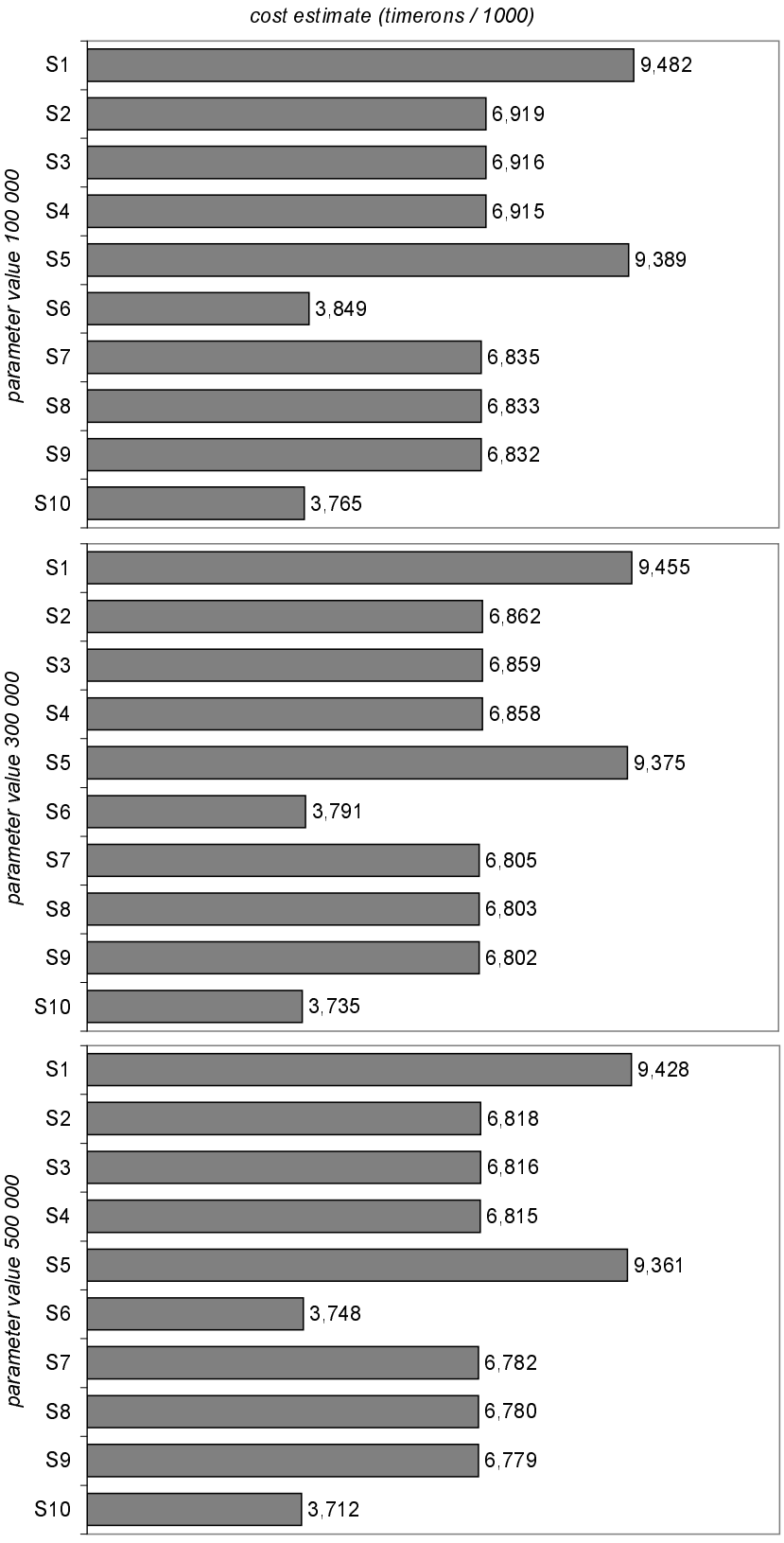


Figure 5.23: Cost estimates of all sequences in the search space for three different parameter values.

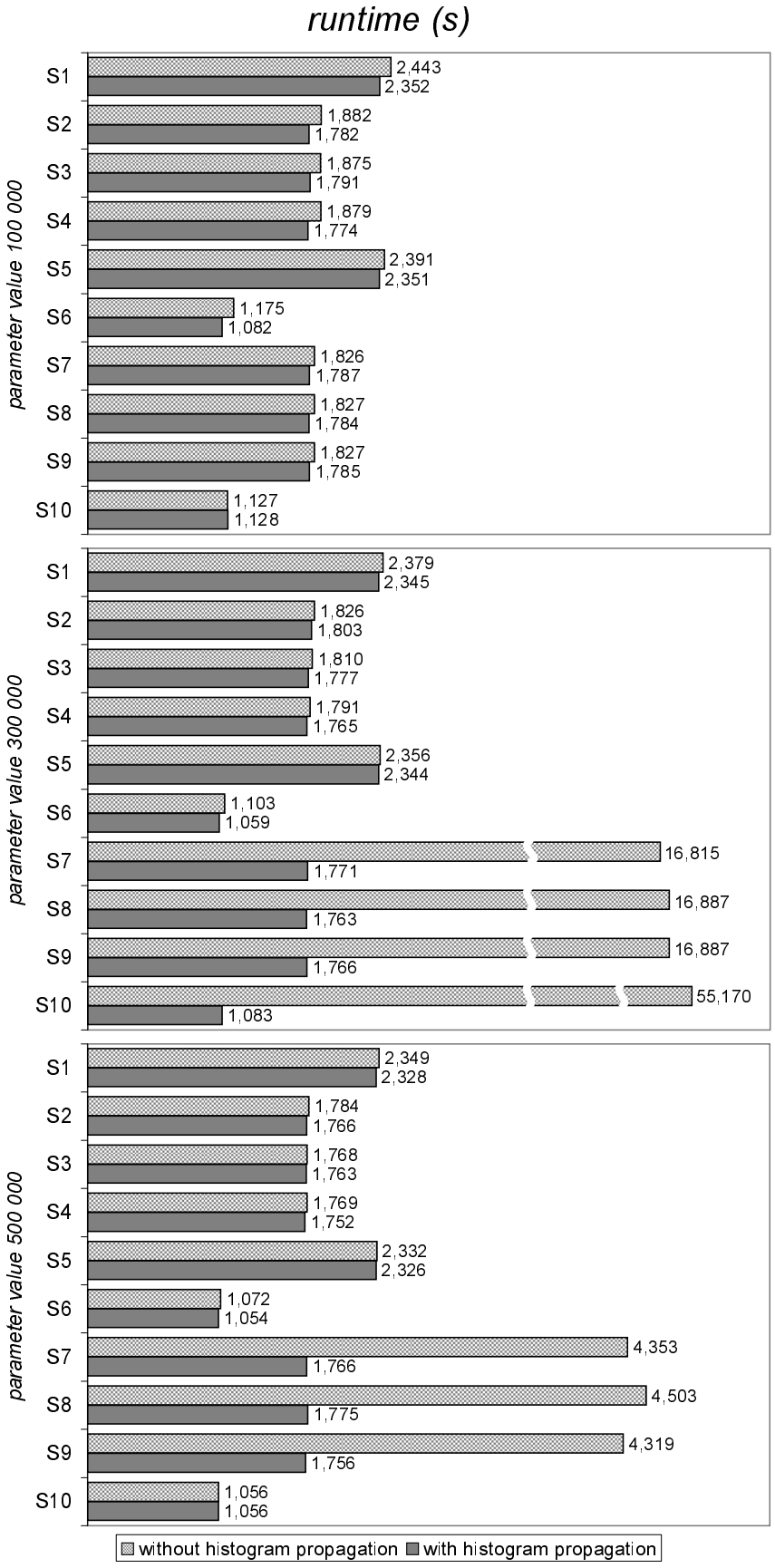


Figure 5.24: Runtimes of all sequences in the search space for three different parameter values.

for each customer and each year between 1992 and 1994 and selects those tuples where the turnover is greater than a given constant parameter value. Hence, the target table of query *Q6* contains the union of query *Q1*, query *Q2*, and query *Q3*, with the filter predicates of query *Q4* applied. Query *Q7* accesses the target table of query *Q6* three times selecting each of the included years once and joins this data with the customer table. So, query *Q7* is not just a merge of query *Q4* and query *Q5*.

We performed the experiments with three different parameter values for the filter: 100 000, 300 000, and 500 000. A higher parameter value denotes a smaller selectivity. The different sequences have been executed in isolation with empty buffer pool and empty statement cache. The runtimes in Figure 5.24, 5.25, and 5.26 are average runtimes of 3 subsequent runs. We distinguish between executions where the propagated histograms for the intermediate-result tables have been stored in the catalog tables of the underlying database and executions without statistics for the intermediate-result tables. In the latter case, we execute the sequence as is, whereas, in the first case, we make use of the histograms that we actually computed to support the cost estimation process. The figures show pure runtimes, i.e., overhead for histogram propagation is not included. However, in our experiments, this overhead is less than 1% for a single sequence. Moreover, the overhead for propagating histograms and estimating costs for all ten sequences adds up to less than 5% of the runtime of the original sequence *S1*. This overhead is acceptable as we could identify *S10* as the most efficient sequence resulting in a performance gain of more than 50% in comparison to the original sequence *S1*. Moreover, the overhead for histogram propagation does not depend on the runtime of a sequence but on the complexity of the corresponding operation trees, the type of the operations used in these operation trees, and the number of buckets in the histograms of the base tables accessed by the sequence. Thus, this approach is especially profitable for long running sequences or sequences with a high optimization potential. Additionally, as stated in Section 5.2, the complexity of the calculations during histogram propagation can be reduced by adding a normalization step prior to or after an operation, if necessary.

Figure 5.23 shows the cost estimates for all sequences of the search space using the three different parameter values for filtering. Figure 5.24 shows the corresponding runtimes. These two figures depict the retrieved cost estimates as a good indicator for the corresponding runtimes. When we use the histograms derived from histogram propagation, the cost estimates exactly reflect the runtimes. When we execute the query sequences as is, i.e., without using the propagated histograms, there are some extreme outliers in the runtimes (see query sequences *S7* to *S10* for parameter value 300 000 and query sequences *S7* to *S9* for parameter value 500 000). The extremely long runtimes of the affected sequences are a result of missing statistics on the target tables of the intermediate-result queries. Due to missing statistics, the query optimizer of the underlying database system makes wrong assumptions regarding table cardinality and predicate selectivity. Therefore, it chooses a bad execution plan that does not fit to the real table cardinalities and predicate selectivities. So, these figures also show how the optimizer of the underlying database system benefits from the statistics available through histogram propagation and that bad execution plans could be avoided when using these statistics. Moreover, they also show the quality of these derived statistics.

Figure 5.25 and Figure 5.26 highlight the behavior of a single query sequence under various parameter values. The first figure refers to the original query sequence *S1*, the second figure refers to the fully optimized query sequence *S10*. Both figures show that the cost estimates also reflect the changes in the runtimes when various parameter values are used in the filter condition. Without using histograms for execution, the underlying database system would provide the same cost and cardinality estimate for query *Q4* independent of the parameter value that is actually used. This is due to the fact that the comparison with the parameter value is applied to an aggregate and, in this case, DB2 uses a default selectivity. However, since we support aggregate functions in our propagation approach, we get different histograms for the attribute that stores the aggregate and different cardinality estimates for the target table of query *Q4* when using different parameter values in the filter predicate.

We summarize that the cost estimates which our cost estimation approach provides are a good indicator for the corresponding runtimes. The experiments have also shown that histogram propagation is necessary to retrieve appropriate cost estimates for the INSERT statements that read from intermediate-result tables. In this context, we could also show that the extensions which we added to related approaches on histogram propagation, like support for aggregate functions, are also very important for query sequences. More details on the experimental results, i.e., a table containing the measured values of the three runs for each query sequence, can be found in Appendix D.2.

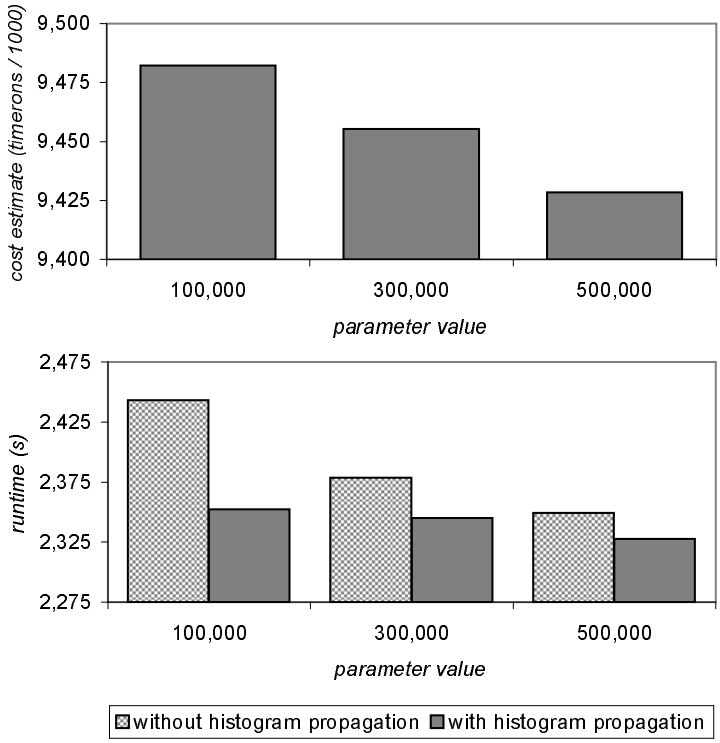


Figure 5.25: Cost estimates and runtimes of sequence *S1* for three different parameter values.

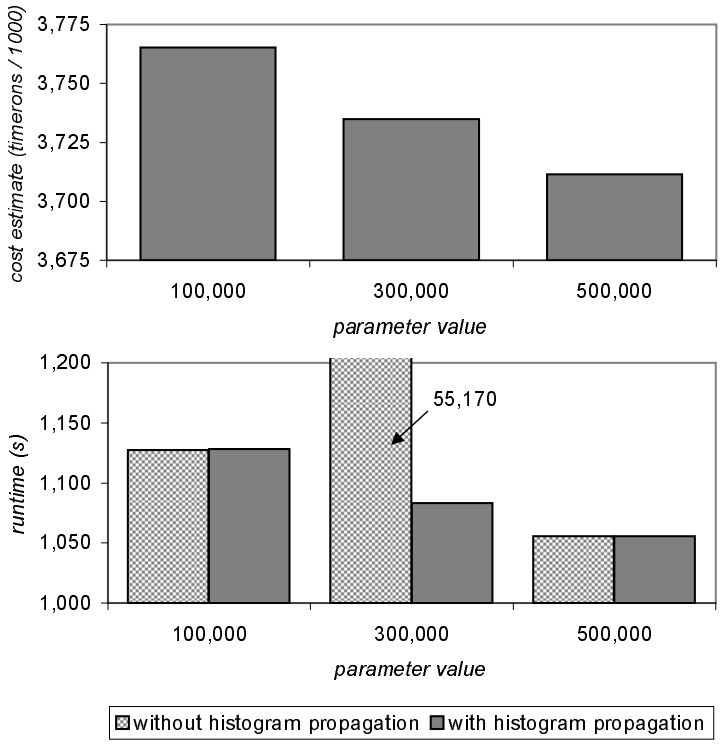


Figure 5.26: Cost estimates and runtimes of sequence *S10* for three different parameter values.

6 Conclusion

In the following, we summarize the results of this thesis and suggest some topics that could be interesting to investigate in further research.

6.1 Summary

In this thesis, we highlighted the optimization of statement sequences. Therefore, we first specified how the statement sequences look like which are the focus of our work and denoted this subset of statement sequences as query sequences. Besides defining query sequences, we also specified an internal representation for query sequences called query dependency graph. Its a graph-based representation that groups all statements with the same target table and that explicitly models consumer-producer dependencies between the INSERT statements of a query sequence.

We classified our optimization problem with respect to the related work on query optimization in database systems. This revealed that most of database research covers SQO problems or MQO problems. Moreover, only few optimization approaches address sequences of statements where the statements are not independent of each other. However, none of the approaches that we looked at considers the specific characteristics and optimization potential of query sequences.

We started with a heuristic optimization approach. This comprises a set of rewrite rules and a priority-based heuristic control strategy. We showed the correctness of our rules by describing their effects on the queries in the relational algebra. Furthermore, we proved that the presented rule set guarantees that the optimization process terminates after a finite number of rule applications. We implemented a prototype of the heuristic optimizer in JAVA. Finally, we performed some experiments with this prototype. This means, we applied the prototype to some query sequences generated by a commercial OLAP tool. The results of the experiments showed that rewriting query sequences can significantly

improve performance. So, they confirm the effectiveness and the optimization potential of our approach.

Since some rule applications caused a deterioration of performance, we extended our optimizer by a cost estimation component in order to build a cost-based optimizer for query sequences. The decision whether a rule should be applied as well as the decision which rule should be applied is now based on cost estimates. As our approach is placed on top of the DBMS which executes the query sequences, building our own cost model means to rebuild the cost model implemented by the query optimizer of the underlying DBMS. This would have been no feasible solution, especially, when we consider that our approach should be independent of a certain DBMS. Thus, we decided to employ the optimizer of the underlying DBMS and make use of its cost estimation capabilities. Since missing statistics for the intermediate-result tables created by the statements of the sequence may lead to suboptimal execution plans and useless cost estimates, we decided to make use of histogram propagation to increase the quality of the cost estimates. Thus, we retrieve the base table statistics from the underlying database, propagate them through the INSERT statements of the query sequence and store the propagation results as statistics for the intermediate-result tables in the underlying database. For histogram propagation, we adapted and extended techniques known from approximate query answering to our needs. We added support for arithmetic expressions and for grouping and aggregation. Furthermore, we provide comparison operators that allow both operands to be attributes. Hence, we do not need a separate implementation of the join operation, but we can represent it by a Cartesian product followed by a selection. Because a common interface is missing to access the statistics, meta data, and optimizer estimates required by the cost estimation component, we proposed a DBMS-independent JAVA interface called Statistics API to access these data. Besides this, Statistics API also provides its own data structures for the data passed on as arguments to the methods of the interface or returned by the methods of the interface. We implemented Statistics API for the three commercial DBMSs IBM DB2, Oracle, and Microsoft SQL Server. Moreover, we implemented a prototype of the cost estimation component to perform some experiments. In our experiments, we retrieved cost estimates for a set of syntactically different but semantically equivalent query sequences and compared these cost estimates to the corresponding execution times. The set of query sequences represents the search space for a generated query sequence with respect to the rule set introduced in this document. The results showed that the cost estimates are a good indicator for the execution times of the query sequences and that our approach is a feasible solution to compare the alternative query sequences within a search space. Moreover, the experiments showed that the propagated statistics support the query optimizer in the underlying DBMS and that these statistics should also be used when executing a query sequence and not only when estimating the costs of a query sequence.

6.2 Future Work

Most potential for future research lies in extending the rules or the rule set and in extending the definition of query sequences. First, existing rules could be extended so that they consider more special cases. For example, the *ConcatQueries* rule as described in this document only fires, if the target query is referenced by not more than a single query. However, we could change the *ConcatQueries* rule in a way that, if the target query is referenced by more than one query, we merge it with each of those referencing queries. Alternatively, we could also merge it just with one of those queries, when we do not remove the target query afterwards. Secondly, we could add new rules to the rule set. For example, this could be inverse rules of the *Merge* rules, i.e., rules that split a single query into several queries instead of merging several queries into a single one. New rules also arise when we extend the subset of SQL that can be used in query sequences. For example, we could add support for nested queries or set operations which offers further potential for optimization. However, when we apply these extensions to the rules or to the rule set, we also have to consider that this adds to the complexity of the search space and that this bears the risk of oscillating rule applications. Hence, with each change applied to a rule or to the rule set, we have to check whether a termination of the optimization process is guaranteed with respect to the heuristic approach. Furthermore, with respect to the cost-based approach, we have to consider pruning techniques to reduce the complexity of the search space when enumerating the alternative query sequences.

Histogram propagation was in the focus of our work, when we tried to support the query optimizer of the underlying DBMS with statistics for the intermediate-result tables. However, it could be interesting to investigate some other techniques of approximate query answering to provide statistics for intermediate-result tables. Since sampling is supported by more and more DBMSs, this could be a good alternative. This means, we execute the INSERT statements on samples of the base tables instead on the full content of the base tables and derive statistics for the intermediate-result tables from their content after this execution. Alternatively, we could even use the runtimes of the query sequences executed on samples of the base tables to estimate costs for the query sequences.

Another interesting question is: What happens when we combine our rewriting approach with other approaches suggested by related work? Maybe, the different approaches could benefit from each other. For example, combining our rewrite approach with indexes and statistics on the intermediate-result tables could be interesting. However, for this purpose, the definition of query sequences would have to be extended so that the statements that create the indexes and that gather the statistics are supported, too.

We also see directions of future research in the optimization of programs that make use of embedded SQL or workflows that contain SQL activities. We mentioned a first approach regarding this topic in the related work but there is still potential for further investigations. However, this adds much to the complexity of the search space and it is hard to argue for correctness in this context.

A

SQL Syntax Diagrams

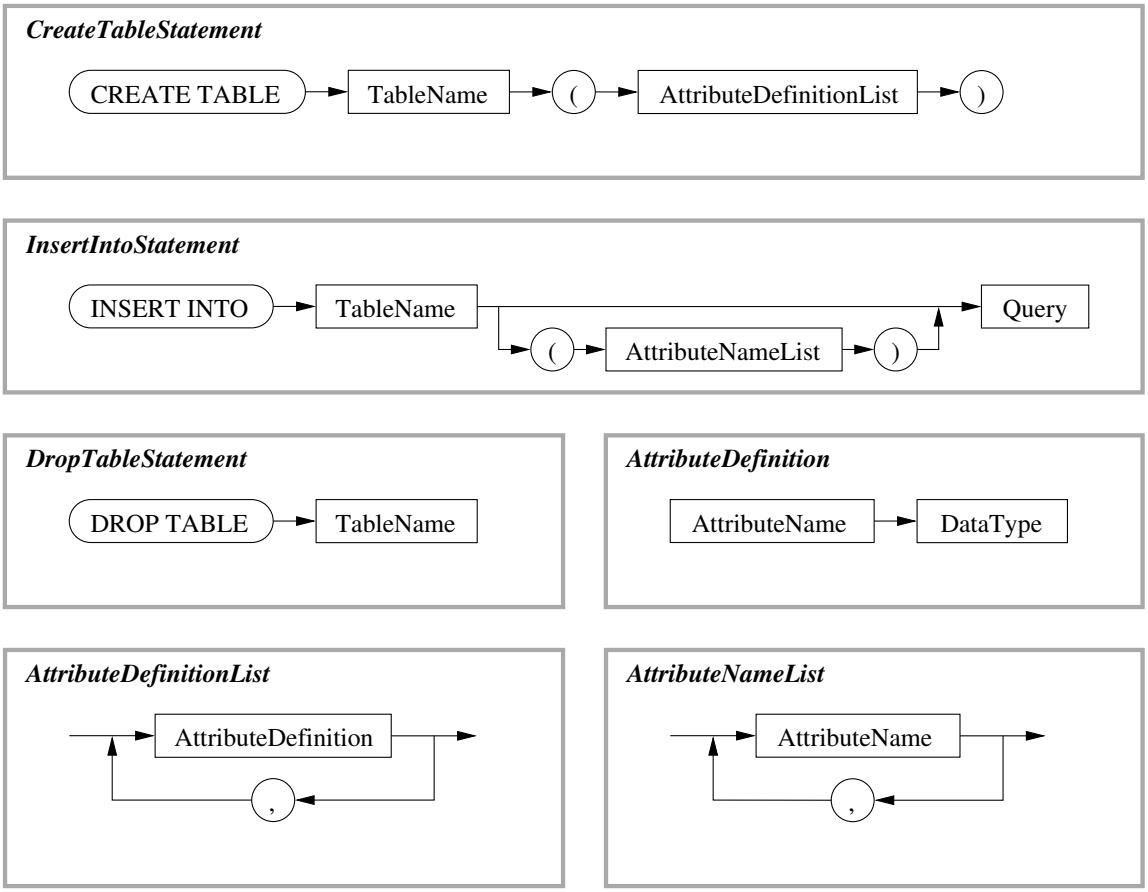


Figure A.1: SQL syntax for query sequences – part 1.

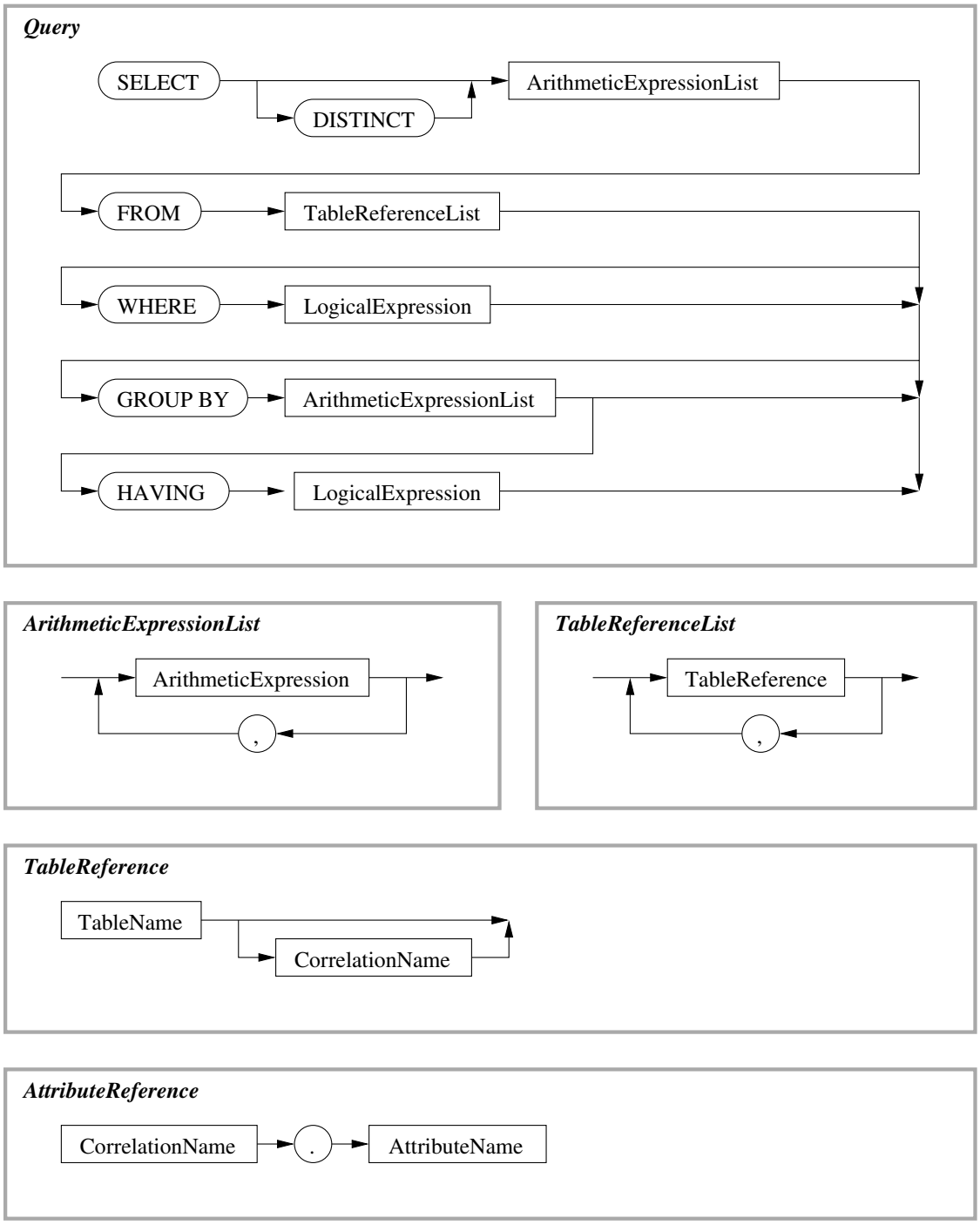


Figure A.2: SQL syntax for query sequences – part 2.

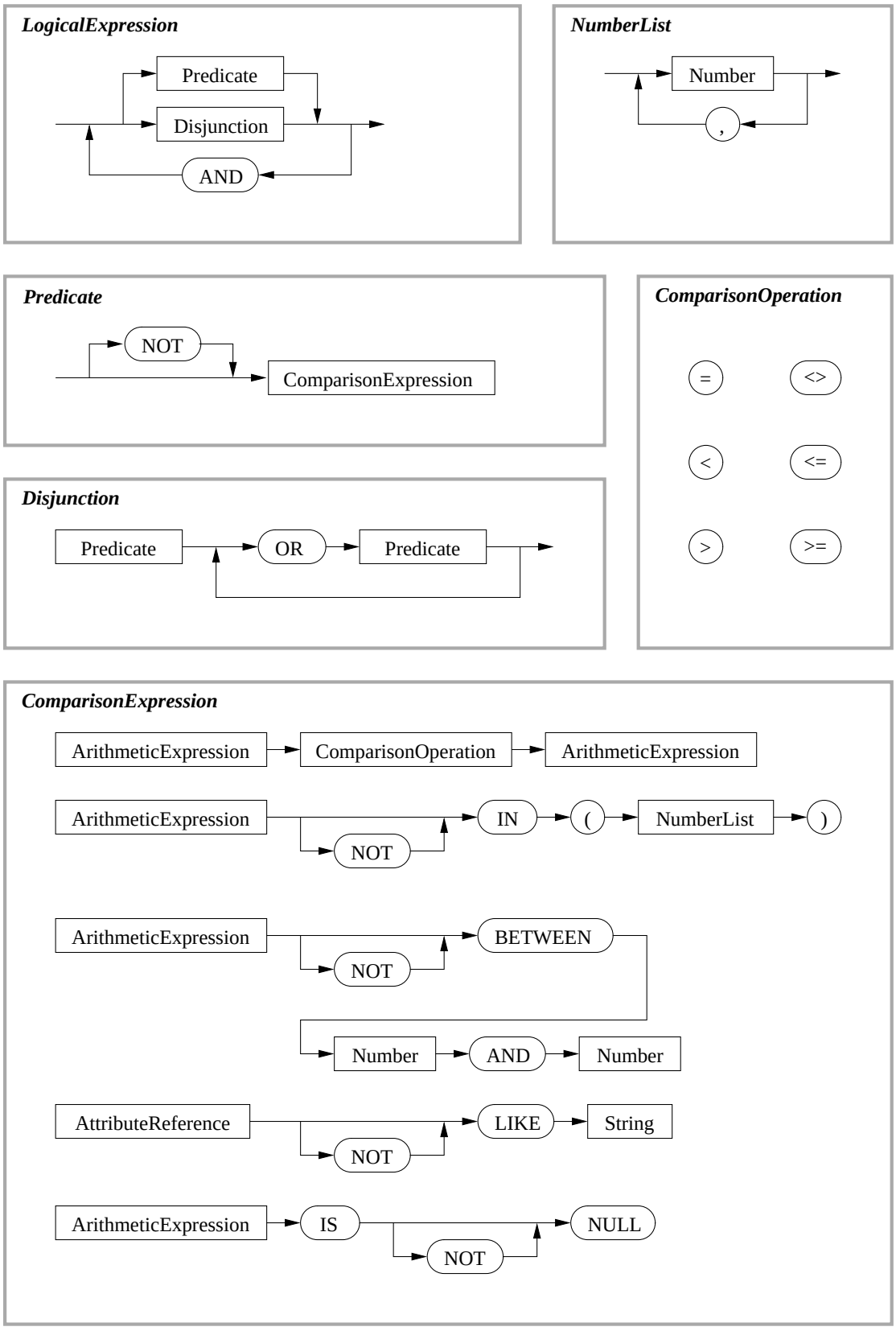


Figure A.3: SQL syntax for query sequences – part 3.

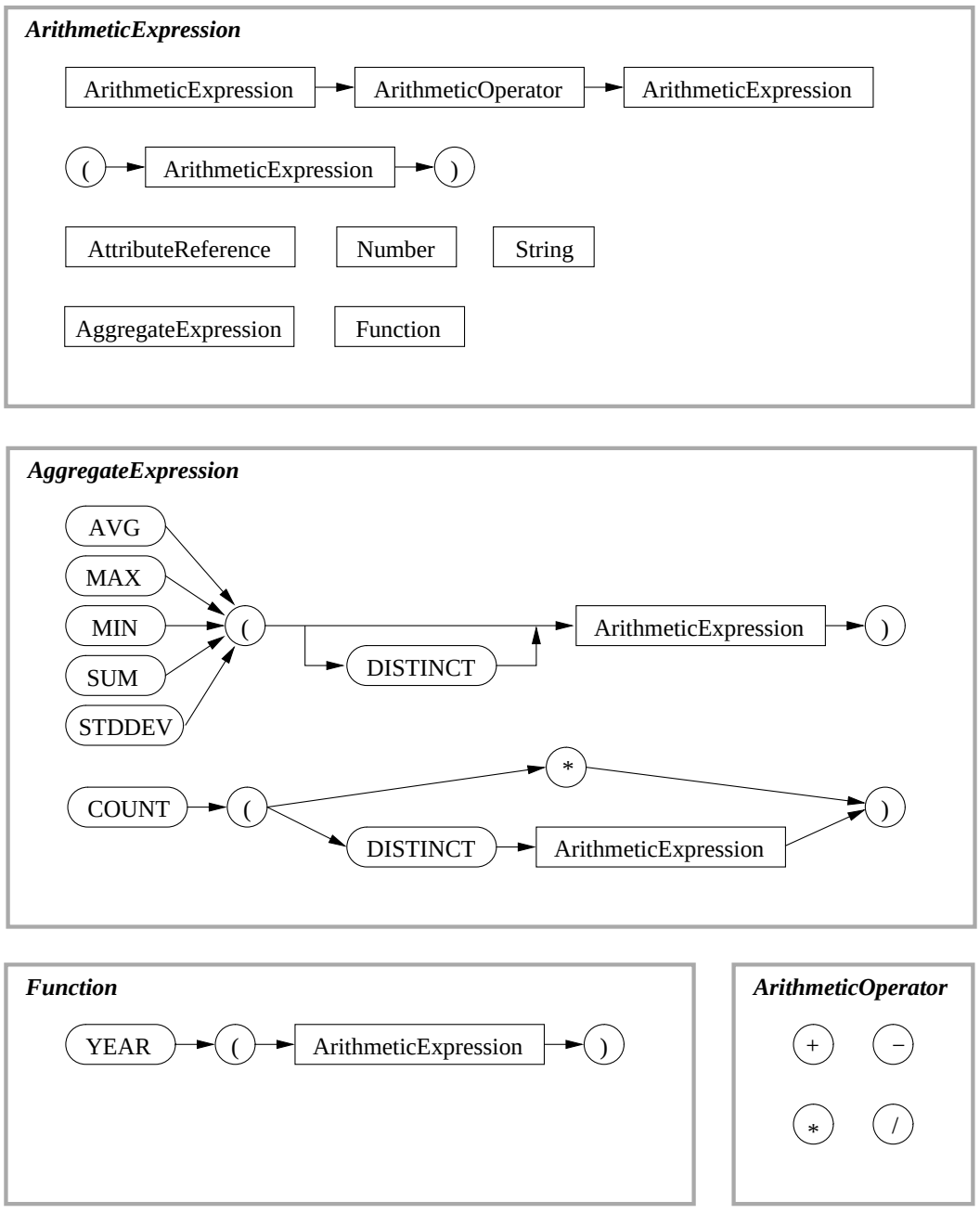


Figure A.4: SQL syntax for query sequences – part 4.

B
Internal Representation

B.1 DTD of CGO-XML

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!ELEMENT   sequence          (query)*>
<!ELEMENT   query             (referenced-by, select-clause,
                               from-clause, where-clause?,
                               (group-by-clause, having-clause?)?)>
<!ATTLIST   query             name      ID              #REQUIRED
                              distinct  (no | yes)      #REQUIRED
                              result    (no | yes)      #REQUIRED>

<!ENTITY    % aggr            "(avg | max | min | sum | count | stddev)">
<!ENTITY    % arithm          "(plus | minus | mul | div)">
<!ENTITY    % comp            "(equal | not-equal | less | greater |
                               less-equal | greater-equal |
                               between | not-between |
                               in | not-in | like | not-like |
                               is-null | is-not-null)">
<!ENTITY    % function        "(year)">
<!ENTITY    % expr            "(%aggr; | %arithm; | %function;
                               constant | string | attribute)">

<!ELEMENT   referenced-by     (referencing-query)*>
<!ELEMENT   select-clause     (attribute-definition)+>
<!ELEMENT   from-clause       (source)+>
<!ELEMENT   where-clause      (%comp; | disjunction)+>
```

```
<!ELEMENT   group−by−clause      (%expr;)+>
<!ELEMENT   having−clause        (%comp; | disjunction)+>

<!ELEMENT   referencing−query    EMPTY>
<!ATTLIST   referencing−query    name  IDREF #REQUIRED>
<!ELEMENT   attribute−definition (%expr;)>
<!ATTLIST   attribute−definition name  CDATA #REQUIRED
                                 type  CDATA #REQUIRED>
<!ELEMENT   disjunction          (%comp;, (%comp;)+)>
<!ELEMENT   source               EMPTY>
<!ATTLIST   source               name  CDATA #REQUIRED
                                 alias CDATA #REQUIRED>

<!ELEMENT   constant             EMPTY>
<!ATTLIST   constant             value CDATA #REQUIRED>
<!ELEMENT   string               EMPTY>
<!ATTLIST   string               value CDATA #REQUIRED>
<!ELEMENT   attribute            EMPTY>
<!ATTLIST   attribute            name  CDATA #REQUIRED
                                 source CDATA #REQUIRED>

<!ELEMENT   equal                (%expr;, %expr;)>
<!ELEMENT   not−equal            (%expr;, %expr;)>
<!ELEMENT   less−equal           (%expr;, %expr;)>
<!ELEMENT   greater−equal        (%expr;, %expr;)>
<!ELEMENT   less                 (%expr;, %expr;)>
<!ELEMENT   greater              (%expr;, %expr;)>
<!ELEMENT   in                   (%expr;, constant+)>
<!ELEMENT   not−in               (%expr;, constant+)>
<!ELEMENT   between              (%expr;, constant, constant)>
<!ELEMENT   not−between          (%expr;, constant, constant)>
<!ELEMENT   like                 (attribute, string)>
<!ELEMENT   not−like             (attribute, string)>
<!ELEMENT   is−null              (%expr)>
<!ELEMENT   is−not−null          (%expr)>

<!ELEMENT   plus                 (%expr;, %expr;)>
<!ELEMENT   minus                (%expr;, %expr;)>
<!ELEMENT   mul                  (%expr;, %expr;)>
<!ELEMENT   div                  (%expr;, %expr;)>
<!ELEMENT   avg                  (%expr;)>
<!ATTLIST   avg                  distinct (no | yes) #REQUIRED>
<!ELEMENT   max                  (%expr;)>
<!ATTLIST   max                  distinct (no | yes) #REQUIRED>
```

```
<!ELEMENT   min      (%expr;)>
<!ATTLIST   min      distinct (no | yes) #REQUIRED>
<!ELEMENT   sum      (%expr;)>
<!ATTLIST   sum      distinct (no | yes) #REQUIRED>
<!ELEMENT   stddev   (%expr;)>
<!ATTLIST   stddev   distinct (no | yes) #REQUIRED>
<!ELEMENT   count    (%expr;? | EMPTY)>
<!ATTLIST   count    distinct (no | yes) #REQUIRED>

<!ELEMENT   year     (%expr;)>
<!ATTLIST   year     EMPTY>
```

B.2 Sample

```
<query name="t1" distinct="no" result="no">
  <referenced-by>
    <referencing-query name="t1"/>
  </referenced-by>
  <select-clause>
    <attribute-definition name="custkey" type="INTEGER">
      <attribute name="o_custkey" source="o"/>
    </attribute-definition>
    <attribute-definition name="year" type="INTEGER">
      <year>
        <attribute name="o_orderdate" source="o"/>
      </year>
    </attribute-definition>
    <attribute-definition name="turnover" type="FLOAT">
      <sum distinct="no">
        <attribute name="o_totalprice" source="o"/>
      </sum>
    </attribute-definition>
  </select-clause>
  <from-clause>
    <source name="orders" alias="o"/>
  </from-clause>
  <where-clause>
    <in>
      <year>
        <attribute name="o_orderdate" source="o"/>
      </year>
      <constant value="1990"/>
      <constant value="1991"/>
    </in>
  </where-clause>
  <group-by-clause>
    <attribute name="o_custkey" source="o"/>
    <year>
      <attribute name="o_orderdate" source="o"/>
    </year>
  </group-by-clause>
</query>
```

Figure B.1: Internal representation of the intermediate-result query in the lower query sequence of Figure 3.1.

```
<query name="t3" distinct="no" result="yes">
  <referenced-by/>
  <select-clause>
    <attribute-definition name="custkey" type="INTEGER">
      <attribute name="c_custkey" source="c"/>
    </attribute-definition>
    <attribute-definition name="name" type="VARCHAR(25)">
      <attribute name="c_name" source="c"/>
    </attribute-definition>
  </select-clause>
  <from-clause>
    <source name="t1" alias="t1"/>
    <source name="t1" alias="t2"/>
    <source name="customer" alias="c"/>
  </from-clause>
  <where-clause>
    <equal>
      <attribute name="custkey" source="t1"/>
      <attribute name="c_custkey" source="c"/>
    </equal>
    <equal>
      <attribute name="custkey" source="t1"/>
      <attribute name="custkey" source="t2"/>
    </equal>
    <greater>
      <attribute name="turnover1991" source="t2"/>
      <attribute name="turnover1990" source="t1"/>
    </greater>
    <equal>
      <attribute name="year" source="t1"/>
      <constant value="1990"/>
    </equal>
    <equal>
      <attribute name="year" source="t2"/>
      <constant value="1991"/>
    </equal>
  </where-clause>
</query>
```

Figure B.2: Internal representation of the final-result query in the lower query sequence of Figure 3.1.

C Pseudo Code

In the following, we present the rewrite rules in a pseudo code notation. Therefore, we first introduce the pseudo code notation which we developed to specify the rewrite rules in a more formal way than in natural language. Afterwards, we provide pseudo code for the condition part as well as for the action part of each rewrite rule.

C.1 Notation

Most of the elements in our pseudo code notation are self-explanatory since this language derives much from imperative programming languages as well as from Boolean algebra and set algebra. A program specified in this pseudo code notation is just a sequence of statements where statements are separated by semicolons. A statement can be an assignment statement, a control flow statement, a return statement, or a procedure call. For simplicity, variables and their data type do not have to be declared beforehand. Therefore, we provide no declare statement.

C.1.1 Variables and Expressions

A variable is implicitly declared the first time it is used. Variable names consists of a character string which may be followed by a subscript and / or a superscript. The data type of a variable results from the data type of the result of the expression that is assigned to it. The pseudo code notation supports primitive data types like integers as well as structured data types like lists. For understanding the code fragments, it's mostly not necessary to know how the data structures returned by the functions look like in detail. Thus, we omit to discuss these structures in more detail.

Expressions are used in assignment statements, in return statements, or as arguments in function calls and procedure calls. They consist of variables, constant values, operations, and function calls.

Besides the values of the various data types, our pseudo code notation supports two reserved values, $\emptyset$ and ε. $\emptyset$ represents the empty list whereas ε indicates a non-existing value. Both are mostly used as return values by functions, e.g., the functions that retrieve a certain clause in an INSERT statement return $\emptyset$ when the requested clause does not exist in the given INSERT statement. (In programming languages, such values are often named *null* or *nil*.)

A comma-separated sequence of variables and / or constant values which is surrounded by angle brackets represents a list. The values in the sequence represent the list items. Unlike sets, lists are ordered collections of items and a certain value can appear more than once within the same list. For example, $\langle\ 1, 2, 3, 1\ \rangle$ represents a list that contains the four values 1, 2, 3, and 1 in that order. For lists, the pseudo code notation provides the following operations known from set algebra: union ($\cup$), intersection ($\cap$), and difference ($-$). These operations treat the lists as sets and remove duplicate rows from the final result set. Additionally, the pseudo code notation supports a concatenation operation ($\uplus$) that joins two lists end to end without removing duplicates. The length of a list can be retrieved by surrounding the variable that stores the list by vertical bars, e.g., when variable a currently stores the list $\langle\ 1, 2, 3, 1\ \rangle$, the expression $|a|$ evaluates to 4.

A right-pointing arrow ($\rightarrow$) with a list on both sides of the arrow creates a mapping between the items of the two lists. So, a mapping is a list of pairs where each element from the list on the left side of the arrow builds a pair with the element at the same position in the list on the right side of the arrow. Therefore, the two lists must have same length.

Expressions used as conditions within control flow statements are logical expressions. Logical expressions evaluate to one of the two Boolean values, $true$ and $false$. A logical expression consists of predicates, Boolean values, and logical operations like NOT ($\neg$), AND ($\wedge$), and OR ($\vee$). In the simplest case, a predicate is a function call that returns a Boolean value. Otherwise, a predicate is either an equivalence test that checks two operands for equivalence ($=$) or non-equivalence ($\neq$) or an inclusion test that checks whether a list includes a certain value ($\in$) or not ($\notin$). Two lists are regarded equivalent when they contain the same values without considering how often these values appear and without considering the order of the values within the lists. For example, $\langle\ 1, 1, 2, 3\ \rangle$ equals $\langle\ 3, 2, 2, 1\ \rangle$ since both lists consist of the values 1, 2, and 3.

C.1.2 Assignment Statement

An assignment statement is indicated by a left-pointing arrow ($\leftarrow$) where the result of the expression on the right side of the arrow is assigned to the variable on the left side of the arrow.

Later on, we use lists on the left side of the arrow when the right side of the arrow is also a list and we want to assign each value of this list to a separate variable. For example, the following statement assigns value 1 to variable a, value 2 to variable b, and value 3 to variable c: $\langle\ a, b, c\ \rangle\ \leftarrow\ \langle\ 1, 2, 3\ \rangle$. Hence, the length of the list on the right side has to be known in advance and the list on the left side of the arrow must be of same length as the list on the right side of the arrow.

C.1.3 Control Flow Statements

The pseudo code notation provides control flow statements to realize conditionals and loops. The *if-then-else* statement allows for conditional processing. The syntax is as follows:

if *condition*
then $statement_1$;
else $statement_2$;

When the condition after keyword *if* evaluates to *true*, the statement following keyword *then* is executed. Otherwise, the execution continues with the statement following keyword *else*. After executing the statement following keyword *then* or following keyword *else*, execution continues with the statement following the *if-then-else* statement. The *else* part of the conditional statement is optional. When the *else* part is missing and the condition evaluates to $false$, execution directly continues with the statement that follows the statement in the *then* branch.

The pseudo code notation supports two types of loops, count-controlled loops and collection-controlled loops. The *for* statement realizes a count-controlled loop and has the following syntax:

for *counter* **from** *start* **to** *end* **do**
    statement;

The *for* statement enumerates each of the values within a numeric integer range and in each enumeration step, it executes the statement in its body. The range is specified by a start value which follows keyword *from* and an end value which follows keyword *to*. In each iteration step, the current value of the counter is bound to the variable which follows keyword *for* and the statement following keyword *do* is being executed under the current binding.

The *foreach* statement realizes a collection-controlled loop, i.e., a loop that iterates over the items in a collection such as a list and that executes the statement in its body in each iteration step. The syntax is as follows:

foreach *item* **in** *collection* **do**
    statement;

The collection is given by the expression that follows keyword *in*. In each iteration step, an item of the collection is bound to the variable which follows keyword *foreach* and the statement following keyword *do* is being executed under the current binding.

When more than just a single statement should be executed within a branch of the *if-then-else* statement or within a loop, we have to define a block. A block is a sequence of statements surrounded by the keywords *begin* and *end*. For example, the syntax of an *if-then-else* statement using blocks in the *then* and *else* branch is as follows:

if *condition* **then begin**
    $statement_1$;
    ...
    $statement_n$;
end else begin
    $statement_{n+1}$;
    ...
    $statement_{n+m}$;
end;

When execution enters the *then* branch, the statements in the sequence starting with $statement_1$ and ending with $statement_n$ will be subsequently executed. Accordingly, when execution enters the *else* branch, the statements in the sequence starting with $statement_{n+1}$ and ending with $statement_{n+m}$ will be subsequently executed.

C.1.4 Return Statement

The return statement completes the execution of a sequence and returns the value of the expression which follows the keyword *return*. When not just a single value but multiple values should be returned, a list has to be used.

C.1.5 Functions and Procedures

The names of the functions and procedures are mostly self-explanatory. When a function name starts with prefix *get*, it retrieves the value of something whereas when a procedure starts with prefix *set*, it sets the value of something. A procedure starting with prefix *replace* replaces some elements within a target structure according to a mapping which is passed on as argument. A procedure that starts with prefix *add* adds something to a target structure whereas a procedure that starts with prefix *remove* removes something from a target structure. A function starting with prefix *is* tests whether the argument of this function possesses certain characteristics. Functions starting with *contains* perform an inclusion test.

Functions and Procedures Regarding Queries

- getCreateTableStatement(q)
  This function returns the CREATE TABLE statement of query q.
- getInsertStatement(q)
  This function returns the INSERT statement of query q.
- setInsertStatement(q, i)
  This procedure replaces the INSERT statement of query q by the new INSERT statement i.

- getTargetTable(q)

  This function returns the name of the target table of query q.

- removeQuery(qdg, q)

  This procedure removes query q from query dependency graph qdg.

- isIntermediateResultQuery(q)

  This function returns $true$ when query q is an intermediate-result query and $false$ when query q is a final-result query.

- containsImplicitTypeCastDuringInsert(qdg, q)

  This function returns $true$ when the data type of at least a single element in the SELECT clause in the INSERT statement of query q in query dependency graph qgd does not match with the data type of the corresponding attribute definition in the CREATE TABLE statement of query q in query dependency graph qdg, $false$ otherwise.

Functions and Procedures Regarding INSERT Statements

- isDistinct(i)

  This function returns $true$ when the body of INSERT statement i explicitly eliminates duplicates, i.e., when the body starts with SELECT DISTINCT, $false$ otherwise.

- getClauseList(i)

  This function returns a list that contains all (empty as well as non-empty) clauses in INSERT statement i.

- setClauseList($i, \langle\ s, f, w, g, h\ \rangle$)

  This procedure replaces all clauses in INSERT statement i by the corresponding clauses in the list $\langle\ s, f, w, g, h\ \rangle$.

- getSelectClause(i)

  This function returns the SELECT clause in INSERT statement i as a list of arithmetic expressions.

- setSelectClause(i, s)

  This procedure replaces the SELECT clause in INSERT statement i by the list of arithmetic expressions s.

- getFromClause(i)

  This function returns the FROM clause in INSERT statement i as a list of table references consisting of a table name and a correlation name.

- getWhereClause(i)

  This function returns the WHERE clause in INSERT statement i as a list of dis-

junctions and / or predicates. This list represents a conjunction since the WHERE clause is in conjunctive normal form.

- setWhereClause(i, w)

  This procedure replaces the WHERE clause in INSERT statement i by the list of disjunctions and / or predicates w. This list represents a conjunction since the WHERE clause is in conjunctive normal form.

- getHavingClause(i)

  This function returns the HAVING clause in INSERT statement i as a list of disjunctions and / or predicates. This list represents a conjunction since the HAVING clause is in conjunctive normal form.

- setHavingClause(i, h)

  This procedure replaces the HAVING clause in INSERT statement i by the list of disjunctions and / or predicates w. This list represents a conjunction since the HAVING clause is in conjunctive normal form.

- getGroupByClause(i)

  This function returns the GROUP BY clause in INSERT statement i as a list of arithmetic expressions.

- setGroupByClause(i, g)

  This procedure replaces the GROUP BY clause in INSERT statement i by the list of arithmetic expressions g.

Functions and Procedures Regarding Mapping

Some of the mappings passed on as arguments in the following functions and procedures are not complete but partial. When there is no entry for some source element, it is being kept as it is, i.e., the mapping is only applied to those elements that appear as source element in the mapping. For example, a mapping of attribute names may not have an entry for each attribute name that appears in the structure to which we apply the mapping. Thus, when there exists no mapping for an attribute name that appears in the structure, we keep the old attribute name.

All of the replacement functions just return the result of the replacement but do not modify the data structure which has been passed on to the function as an argument.

- getCorrelationNameMappings(i_1, i_2)

  This function returns all valid mappings from the correlation names of the FROM clause elements in INSERT statement i_2 onto the correlation names of the FROM clause elements in INSERT statement i_1. In this case, valid means that the mapping is bijective and two correlation names which are mapped onto each other refer to the same table. The mapping is represented by a list of pairs of correlation names.

- replaceCorrelationNames(i, m)

  This function replaces the correlation names in INSERT statement i according to mapping m.

- replaceAttributeNames(adl, m)

  This function replaces the attribute names in the attribute definitions in the list of attribute definitions adl according to mapping m.

- replaceAttributeNames(i, t, m)

  In INSERT statement i, this function replaces the attribute names in the attribute references which refer to attributes in table t according to mapping m.

- replaceTable(i, t_1, t_2)

  This function replaces table name t_1 in the table references of the FROM clause in INSERT statement i by table name t_2.

- replaceExpressions(x, m)

  This function replaces expressions in x according to mapping m. Since an attribute is the simplest form of an expression, this function can also be used to replace attributes. x can be a clause, a list of predicates, or a predicate.

Functions and Procedures Regarding the SELECT Clause

- containsAggregateExpression(s)

  This function returns $true$ when at least a single arithmetic expression in SELECT clause s contains an aggregate-function call, $false$ otherwise.

- containsAttributeInDenominator(s)

  This function returns $true$ when at least a single arithmetic expression in SELECT clause s contains a fraction and the denominator of this fraction contains an attribute, $false$ otherwise.

Functions and Procedures Regarding the WHERE Clause

- combineViaOR(w_1, w_2)

  This function combines the WHERE clauses, w_1 and w_2, which are represented by two lists of disjunctions and / or predicates via a logical OR. It returns the result in conjunctive normal form, too, i.e., it returns the result as a list of disjunctions and / or predicates.

- addToWhereClause(i, t, w)

  For each appearance of table name t in the table references of INSERT statement i, this procedure adds the elements of WHERE clause w to the WHERE clause of INSERT statement i. Before adding the elements, the procedure replaces the correlation names of the attributes in WHERE clause w according to the current

table reference. The procedure only adds an element of WHERE clause w to the WHERE clause of INSERT statement i when neither the same nor a stronger element already exists in the WHERE clause of INSERT statement i.

- removeFromWhereClause(i, t, pl)

  For each appearance of table name t in the table references of INSERT statement i, this procedure removes all predicates contained in predicate list pl from the WHERE clause of INSERT statement i. Before removing a predicate listed in pl, the procedure has to replace the correlation names of the attributes in this prediacte according to the current table reference.

Functions and Procedures Regarding Query Dependencies

- getDependentQueries(qdg, q)

  This function returns a list of queries which contains all queries that directly depend on query q, i.e., all queries that appear as destination in a direct query dependency which has query q as source.

- getDependenciesBySource(qdg, q)

  This function returns all direct query dependencies which have query q as source.

- getDependenciesByDestination(qdg, q)

  This function returns all direct query dependencies which have query q as destination.

- removeDependency(qdg, d)

  This procedure removes direct query dependency d from query dependency graph qdg.

- setSource(d, q)

  This procedure replaces the source of direct query dependency d by query q.

- setDestination(d, q)

  This procedure replaces the destination of direct query dependency d by query q.

Functions and Procedures Regarding Predicates

- isEquivalenceTestWithConstant(p)

  This function returns $true$ when predicate p is an equivalence test where one operand is an arithmetic expression that contains an attribute and the other operand is a constant value, $false$ otherwise.

- isEquivalenceTestWithConstantList(p)

  This function returns $true$ when predicate p is an equivalence test where one operand is an arithmetic expression that contains an attribute and the other operand is a constant value or a list of constant values, $false$ otherwise.

- getArithmeticExpression(p)

  Assuming that predicate p is an equivalence test where one operand is an arithmetic expression that contains an attribute and the other operand is a constant value or a list of constant values, this function returns the operand that is an arithmetic expression.

- mergePredicates(p_1, p_2)

  This function merges the two characteristic predicates p_1 and p_2 into a single predicate according to the *WhereToGroup* rule (see Section 4.1.1.4). This means, p_1 and p_2 perform an equivalence test on an arithmetic expression and a constant value or they check whether the result of an arithmetic expression is in a list of constant values. Since the arithmetic expression is the same in both predicates, these predicates can be merged into a single one which checks whether the result of the arithmetic expression is included in a list that is the union of the lists of constant values of both predicates.

- getPushdownPredicateListByTableReference(w, cn)

  This function returns a list of predicates which contains those predicates in WHERE clause w that only include attribute references where the correlation name equals cn. So these predicates are possible candidates for a predicate pushdown according to the *PredicatePushdown* rule (see Section 4.1.2.2).

Functions and Procedures Regarding Attribute Definitions

- createAttributeDefinitionList(anl, dtl)

  This function creates a list of attribute definitions based on the list of attribute names anl and the list of data types dtl, i.e., anl and dtl must have same size and each data type in dtl corresponds to the attribute name at the same position in anl.

- getAttributeDefinitionList(c)

  This function returns a list that contains all attribute definitions of CREATE TABLE statement c.

- setAttributeDefinitionList(c, adl)

  This procedure replaces the attribute definitions in CREATE TABLE statement c by the attribute definitions in the list adl.

- getAttributeNameList(c)

  This function returns a list that contains the attribute names used in the attribute definitions of CREATE TABLE statement c.

- getAttributeName(ad)

  This function returns the attribute name of the attribute defined by attribute definition ad.

- getDataType(ad)

  This function returns the data type of the attribute defined by attribute definition ad.

Functions and Procedures Regarding Attributes and Expressions

- createAttributeList(anl, t)

  This procedure creates a list of attributes based on the list of attribute names anl and the table name t, i.e., table t is being assigned to each attribute name as the source table.

- getAttributeList(c)

  This function returns a duplicate-free list of all attributes that appear in clause c.

- getAggregateExpressionList(c)

  This function returns a duplicate-free list of all aggregate expressions that appear in clause c. (Aggregate expressions are aggregate-function calls.)

- createUniqueAttributeNameList(l)

  This function creates a list of length l that consists of unique attribute names, i.e., attribute names that have not been used yet.

- computeDataTypeList(el, qdg, i)

  This function returns a list which contains the data type for each of the expressions in the expression list el where i is the INSERT statement in query dependency graph qdg from whom the expressions have been extracted. So, the tables to which the correlation names in the attributes of the expressions refer can be found in the FROM clause of INSERT statement i.

- getAttributeNameListByTable(i, t)

  This function returns a duplicate-free list of the names of those attributes of table t which are accessed at least once within the body of INSERT statement i.

Functions and Procedures Regarding Table References

- isReferencedOnce(f, t)

  This function returns $true$ when just a single table reference in FROM clause f has t as table name, i.e., when table t is referenced just once in FROM clause f, $false$ otherwise.

- getTableReferenceListByTableName(f, t)

  This function returns a list that contains those table references in FROM clause f that have t as table name, i.e., it returns a list of all table references that refer to table t.

- createUniqueCorrelationNameList(l)

  This function creates a list of length l that consists of unique correlation names, i.e., correlation names that have not been used yet.

- getCorrelationNameList(f)

  This function returns a list that contains the correlation names of the table references in FROM clause f.

- getCorrelationName(tr)

  This function returns the correlation name of table reference tr.

Functions and Procedures Regarding Lists

- getFirstElement(l)

  This function retrieves the first element in list l.

- getElement(l, i)

  This function retrieves the i-th element in list l where the first element of the list is at position $i = 1$.

Functions and Procedures Regarding Equivalence Classes

- computeEquivalenceClasses(i)

  This function returns a list of equivalence classes regarding the arithmetic expressions within the body of INSERT statement i. When two arithmetic expressions are tested for equivalence in the predicates of the WHERE or HAVING clause, they fall into the same equivalence class.

- elementOfSameEquivalenceClass(ec, ae_1, ae_2)

  Given a list of equivalence classes ec, this function returns $true$ when the arithmetic expressions ae_1 and ae_2 fall into the same equivalence classes, $false$ otherwise. So, this function returns $true$ when the two arithmetic expressions are equivalent or when the two arithmetic expressions are tested for equivalence in the predicates of the WHERE or HAVING clause.

C.2 Rule Actions and Rule Conditions

C.2.1 MergeSelect Rule

Rule Condition

Input: A query dependency graph qdg and two queries q_1 and q_2.
Output: The result of the condition test and an appropriate correlation-name mapping.

```
1   if isIntermediateResultQuery(q1) ∧ isIntermediateResultQuery(q2) then begin
2     i1 ← getInsertStatement(q1);
3     i2 ← getInsertStatement(q2);
4     if ¬ isDistinct(i1) ∧ ¬ isDistinct(i2) then begin
5       foreach m in getCorrelationNameMappings(i1, i2) do begin
6         i2' ← replaceCorrelationNames(i2, m);
7         ⟨ s1, f1, w1, g1, h1 ⟩ ← getClauseList(i1);
8         ⟨ s2, f2, w2, g2, h2 ⟩ ← getClauseList(i2');
9         if (
10              ( g1 ≠ ∅ ∧ g2 ≠ ∅ ∧ g1 = g2 ) ∨
11              ( g1 = ∅ ∧ g2 = ∅ ∧
12                containsAggregateExpression(s1) ∧
13                containsAggregateExpression(s2) ) ∨
14              ( g1 = ∅ ∧ g2 = ∅ ∧
15                ¬ containsAggregateExpression(s1) ∧
16                ¬ containsAggregateExpression(s2) )
17            ) ∧
18            w1 = w2 ∧
19            h1 = h2
20          then return ⟨ true, m ⟩;
21        end;
22        return ⟨ false, ∅ ⟩;
23    end else return ⟨ false, ∅ ⟩;
24  end else return ⟨ false, ∅ ⟩;
```

Rule Action

Input: A query dependency graph qdg, two queries q_1 and q_2 and a correlation-name mapping m.
Output: None.

```
1   c1 ← getCreateTableStatement(q1);
2   c2 ← getCreateTableStatement(q2);
3   i1 ← getInsertStatement(q1);
4   i2 ← getInsertStatement(q2);
5   i2' ← replaceCorrelationNames(i2, m);
6   t1 ← getTargetTable(q1);
7   t2 ← getTargetTable(q2);

8   s1 ← getSelectClause(i1);
9   s2 ← getSelectClause(i2');
10  s1' ← s1 ⊎ s2;
11  setSelectClause(i1, s1');

12  adl1 ← getAttributeDefinitionList(c1);
13  adl2 ← getAttributeDefinitionList(c2);
14  anl2 ← getAttributeNameList(c2);
15  anl2' ← createUniqueAttributeNameList(|s2|);
16  adl2' ← replaceAttributeNames(adl2, anl2 → anl2');
17  adl1' ← adl1 ⊎ adl2';
18  setAttributeDefinitionList(c1, adl1');

19  foreach q in getDependentQueries(qdg, q2) do begin
20      i ← getInsertStatement(q);
21      i' ← replaceAttributeNames(i, t2, anl2 → anl2');
22      i'' ← replaceTable(i', t2, t1);
23      setInsertStatement(q, i'');
24  end;

25  foreach d in getDependenciesBySource(qdg, q2) do
26      setSource(d, q1);
27  foreach d in getDependenciesByDestination(qdg, q2) do
28      removeDependency(qdg, d);

29  removeQuery(qdg, q2);
```

C.2.2 MergeWhere Rule

Rule Condition

Input: A query dependency graph qdg and two queries q_1 and q_2.
Output: The result of the condition test and an appropriate correlation-name mapping.

```
1   if isIntermediateResultQuery(q1) ∧ isIntermediateResultQuery(q2) then begin
2     i1 ← getInsertStatement(q1);
3     i2 ← getInsertStatement(q2);
4     if ¬ isDistinct(i1) ∧ ¬ isDistinct(i2) then begin
5       foreach m in getCorrelationNameMappings(i1, i2) do begin
6         i2' ← replaceCorrelationNames(i2, m);
7         ⟨ s1, f1, w1, g1, h1 ⟩ ← getClauseList(i1);
8         ⟨ s2, f2, w2, g2, h2 ⟩ ← getClauseList(i2');
9         if ¬ containsAggregateExpression(s1) ∧
10           ¬ containsAggregateExpression(s2) ∧
11           ¬ containsAttributeInDenominator(s1) ∧
12           ¬ containsAttributeInDenominator(s2) ∧
13           w1 ≠ w2 ∧
14           g1 = ∅ ∧ g2 = ∅ ∧
15           h1 = ∅ ∧ h2 = ∅
16         then return ⟨ true, m ⟩;
17       end;
18       return ⟨ false, ∅ ⟩;
19     end else return ⟨ false, ∅ ⟩;
20  end else return ⟨ false, ∅ ⟩;
```

Rule Action

Input: A query dependency graph qdg, two queries q_1 and q_2 and a correlation-name mapping m.
Output: None.

1 $c_1 \leftarrow$ getCreateTableStatement(q_1);
2 $c_2 \leftarrow$ getCreateTableStatement(q_2);
3 $i_1 \leftarrow$ getInsertStatement(q_1);
4 $i_2 \leftarrow$ getInsertStatement(q_2);
5 $i_2' \leftarrow$ replaceCorrelationNames(i_2, m);
6 $t_1 \leftarrow$ getTargetTable(q_1);
7 $t_2 \leftarrow$ getTargetTable(q_2);

8 $w_1 \leftarrow$ getWhereClause(i_1);
9 $w_2 \leftarrow$ getWhereClause(i_2');
10 **if** $w_1 \neq \emptyset \wedge w_2 \neq \emptyset$
11 **then** $w_1' \leftarrow$ combineViaOR(w_1, w_2);
12 **else** $w_1' \leftarrow \emptyset$;
13 setWhereClause(i_1, w_1');

14 $s_1 \leftarrow$ getSelectClause(i_1);
15 $s_2 \leftarrow$ getSelectClause(i_2');
16 $al_3 \leftarrow$ getAttributeList(w_1);
17 $al_4 \leftarrow$ getAttributeList(w_2);
18 $s_1' \leftarrow s_1 \uplus s_2 \uplus al_3 \uplus al_4$;
19 setSelectClause(i_1, s_1');

20 $adl_1 \leftarrow$ getAttributeDefinitionList(c_1);
21 $adl_2 \leftarrow$ getAttributeDefinitionList(c_2);
22 $anl_2 \leftarrow$ getAttributeNameList(c_2);
23 $anl_2' \leftarrow$ createUniqueAttributeNameList($|s_2|$);
24 $adl_2' \leftarrow$ replaceAttributeNames($adl_2, anl_2 \rightarrow anl_2'$);
25 $anl_3 \leftarrow$ createUniqueAttributeNameList($|al_3|$);
26 $dtl_3 \leftarrow$ computeDataTypeList(al_3, qdg, i_1);
27 $adl_3 \leftarrow$ createAttributeDefinitionList(anl_3, dtl_3);
28 $anl_4 \leftarrow$ createUniqueAttributeNameList($|al_4|$);
29 $dtl_4 \leftarrow$ computeDataTypeList(al_4, qdg, i_2');
30 $adl_4 \leftarrow$ createAttributeDefinitionList(anl_4, dtl_4);
31 $adl_1' \leftarrow adl_1 \uplus adl_2' \uplus adl_3 \uplus adl_4$;
32 setAttributeDefinitionList(c_1, adl_1');

33 $al_3' \leftarrow$ createAttributeList(anl_3, ε);
34 $w_3 \leftarrow$ replaceExpressions($w_1, al_3 \rightarrow al_3'$);

```
35  al'_4 ← createAttributeList(anl_4, ε);
36  w_4 ← replaceExpressions(w_2, al_4 → al'_4);

37  if w_3 ≠ ∅ then
38      foreach q in getDependentQueries(qdg, q_1) do begin
39          i ← getInsertStatement(q);
40          addToWhereClause(i, t_1, w_3);
41      end;
42  if w_4 ≠ ∅ then
43      foreach q in getDependentQueries(qdg, q_2) do begin
44          i ← getInsertStatement(q);
45          addToWhereClause(i, t_2, w_4);
46          i' ← replaceAttributeNames(i, t_2, anl_2 → anl'_2);
47          i'' ← replaceTable(i', t_2, t_1);
48          setInsertStatement(q, i'');
49      end;

50  foreach d in getDependenciesBySource(qdg, q_2) do
51      setSource(d, q_1);
52  foreach d in getDependenciesByDestination(qdg, q_2) do
53      removeDependency(qdg, d);

54  removeQuery(qdg, q_2);
```

C.2.3 MergeHaving Rule

Rule Condition

Input: A query dependency graph qdg and two queries q_1 and q_2.
Output: The result of the condition test and an appropriate correlation-name mapping.

```
1   if isIntermediateResultQuery(q1) ∧ isIntermediateResultQuery(q2) then begin
2     i1 ← getInsertStatement(q1);
3     i2 ← getInsertStatement(q2);
4     if ¬ isDistinct(i1) ∧ ¬ isDistinct(i2) then begin
5       foreach m in getCorrelationNameMappings(i1, i2) do begin
6         i2' ← replaceCorrelationNames(i2, m);
7         ⟨ s1, f1, w1, g1, h1 ⟩ ← getClauseList(i1);
8         ⟨ s2, f2, w2, g2, h2 ⟩ ← getClauseList(i2');
9         if ¬ containsAttributeInDenominator(s1) ∧
10          ¬ containsAttributeInDenominator(s2) ∧
11          w1 = w2 ∧
12          g1 ≠ ∅ ∧ g2 ≠ ∅ ∧ g1 = g2 ∧
13          h1 ≠ h2
14        then return ⟨ true, m⟩;
15      end;
16      return ⟨ false, ∅ ⟩;
17    end else return ⟨ false, ∅ ⟩;
18  end else return ⟨ false, ∅ ⟩;
```

Rule Action

Input: A query dependency graph qdg, two queries q_1 and q_2 and a correlation-name mapping m.
Output: None.

```
1   c_1 ← getCreateTableStatement(q_1);
2   c_2 ← getCreateTableStatement(q_2);
3   i_1 ← getInsertStatement(q_1);
4   i_2 ← getInsertStatement(q_2);
5   i'_2 ← replaceCorrelationNames(i_2, m);
6   t_1 ← getTargetTable(q_1);
7   t_2 ← getTargetTable(q_2);

8   h_1 ← getHavingClause(i_1);
9   h_2 ← getHavingClause(i'_2);
10  if h_1 ≠ ∅ ∧ h_2 ≠ ∅
11  then h'_1 ← combineViaOR(h_1, h_2);
12  else h'_1 ← ∅;
13  setHavingClause(i_1, h'_1);

14  s_1 ← getSelectClause(i_1);
15  s_2 ← getSelectClause(i'_2);
16  ael_3 ← getAggregateExpressionList(h_1);
17  ael_4 ← getAggregateExpressionList(h_2);
18  s'_1 ← s_1 ⊎ s_2 ⊎ ael_3 ⊎ ael_4;
19  setSelectClause(i_1, s'_1);

20  adl_1 ← getAttributeDefinitionList(c_1);
21  adl_2 ← getAttributeDefinitionList(c_2);
22  anl_2 ← getAttributeNameList(c_2);
23  anl'_2 ← createUniqueAttributeNameList(|s_2|);
24  adl'_2 ← replaceAttributeNames(adl_2, anl_2 → anl'_2);
25  anl_3 ← createUniqueAttributeNameList(|ael_3|);
26  dtl_3 ← computeDataTypeList(ael_3, qdg, i_1);
27  adl_3 ← createAttributeDefinitionList(anl_3, dtl_3);
28  anl_4 ← createUniqueAttributeNameList(|ael_4|);
29  dtl_4 ← computeDataTypeList(ael_4, qdg, i'_2);
30  adl_4 ← createAttributeDefinitionList(anl_4, dtl_4);
31  adl'_1 ← adl_1 ⊎ adl'_2 ⊎ adl_3 ⊎ adl_4;
32  setAttributeDefinitionList(c_1, adl'_1);

33  al_3 ← createAttributeList(anl_3, ε);
34  w_3 ← replaceExpressions(h_1, ael_3 → al_3);
```

```
35  al_4 ← createAttributeList(anl_4, ε);
36  w_4 ← replaceExpressions(h_2, ael_4 → al_4);

37  if w_3 ≠ ∅ then
38      foreach q in getDependentQueries(qdg, q_1) do begin
39          i ← getInsertStatement(q);
40          addToWhereClause(i, t_1, w_3);
41      end;
42  if w_4 ≠ ∅ then
43      foreach q in getDependentQueries(qdg, q_2) do begin
44          i ← getInsertStatement(q);
45          addToWhereClause(i, t_2, w_4);
46          i' ← replaceAttributeNames(i, t_2, anl_2 → anl'_2);
47          i'' ← replaceTable(i', t_2, t_1);
48          setInsertStatement(q, i'');
49      end;

50  foreach d in getDependenciesBySource(qdg, q_2) do
51      setSource(d, q_1);
52  foreach d in getDependenciesByDestination(qdg, q_2) do
53      removeDependency(qdg, d);

54  removeQuery(qdg, q_2);
```

C.2.4 WhereToGroup Rule

Rule Condition

Input: A query dependency graph qdg and two queries q_1 and q_2.
Output: The result of the condition test, an appropriate correlation-name mapping and the two characteristic predicates.

```
1   if isIntermediateResultQuery(q1) ∧ isIntermediateResultQuery(q2) then begin
2     i1 ← getInsertStatement(q1);
3     i2 ← getInsertStatement(q2);
4     if ¬ isDistinct(i1) ∧ ¬ isDistinct(i2) then begin
5       foreach m in getCorrelationNameMappings(i1, i2) do begin
6         i2' ← replaceCorrelationNames(i2, m);
7         ⟨ s1, f1, w1, g1, h1 ⟩ ← getClauseList(i1);
8         ⟨ s2, f2, w2, g2, h2 ⟩ ← getClauseList(i2');
9         if ¬ containsAttributeInDenominator(s1) ∧
10          ¬ containsAttributeInDenominator(s2) ∧
11          ( g1 ≠ ∅ ∨ containsAggregateExpression(s1) ) ∧
12          ( g2 ≠ ∅ ∨ containsAggregateExpression(s2) ) ∧
13          h1 = h2
14        then begin
15          pl1 ← w1 − ( w1 ∩ w2 );
16          pl2 ← w2 − ( w1 ∩ w2 );
17          if |pl1| = 1 ∧ |pl2| = 1 then begin
18            p1 ← getFirstElement(pl1);
19            p2 ← getFirstElement(pl2);
20            ae1 ← getArithmeticExpression(p1);
21            ae2 ← getArithmeticExpression(p2);
22            ael1 ← g1 − (g1 ∩ g2);
23            ael2 ← g2 − (g1 ∩ g2);

24            if |ael1| = 0 ∧ |ael2| = 0 ∧
25              isEquivalenceTestWithConstantValue(p1) ∧
26              isEquivalenceTestWithConstantValue(p2) ∧
27              ae1 = ae2 ∧
28              ae1 ∉ g1 ∧
29              ae2 ∉ g2
30            then return ⟨ true, m, p1, p2 ⟩;

31            else if |ael1| = 1 ∧ |ael2| = 0 ∧
32              isEquivalenceTestWithConstantValueList(p1) ∧
33              isEquivalenceTestWithConstantValue(p2) ∧
34              ae1 = ae2 ∧
35              ae1 ∈ g1 ∧
```

```
36                          ae_2 ∉ g_2
37                      then return ⟨ true, m, p_1, p_2 ⟩;

38                      else if |ael_1| = 0 ∧ |ael_2| = 1 ∧
39                          isEquivalenceTestWithConstantValue(p_1) ∧
40                          isEquivalenceTestWithConstantValueList(p_2) ∧
41                          ae_1 = ae_2 ∧
42                          ae_1 ∉ g_1 ∧
43                          ae_2 ∈ g_2
44                      then return ⟨ true, m, p_1, p_2 ⟩;

45                      else if |ael_1| = 0 ∧ |ael_2| = 0 ∧
46                          isEquivalenceTestWithConstantValueList(p_1) ∧
47                          isEquivalenceTestWithConstantValueList(p_2) ∧
48                          ae_1 = ae_2 ∧
49                          ae_1 ∈ g_1 ∧
50                          ae_2 ∈ g_2
51                      then return ⟨ true, m, p_1, p_2 ⟩;

52                  end;
53              end;
54          end;
55          return ⟨ false, ∅, ε, ε ⟩;
56      end else return ⟨ false, ∅, ε, ε ⟩;
57  end else return ⟨ false, ∅, ε, ε ⟩;
```

Rule Action

Input: A query dependency graph qdg, two queries q_1 and q_2
a correlation-name mapping m and
the two characteristic predicates p_1 and p_2.
Output: None.

1 $c_1 \leftarrow$ getCreateTableStatement(q_1);
2 $c_2 \leftarrow$ getCreateTableStatement(q_2);
3 $i_1 \leftarrow$ getInsertStatement(q_1);
4 $i_2 \leftarrow$ getInsertStatement(q_2);
5 $i_2' \leftarrow$ replaceCorrelationNames(i_2, m);
6 $t_1 \leftarrow$ getTargetTable(q_1);
7 $t_2 \leftarrow$ getTargetTable(q_2);

8 $ae_1 \leftarrow$ getArithmeticExpression(p_1);
9 $ael_3 \leftarrow \langle\ ae_1\ \rangle$;

10 $s_1 \leftarrow$ getSelectClause(i_1);
11 $s_2 \leftarrow$ getSelectClause(i_2');
12 $s_1' \leftarrow s_1 \uplus s_2 \uplus ael_3$;
13 setSelectClause(i_1, s_1');

14 $adl_1 \leftarrow$ getAttributeDefinitionList(c_1);
15 $adl_2 \leftarrow$ getAttributeDefinitionList(c_2);
16 $anl_2 \leftarrow$ getAttributeNameList(c_2);
17 $anl_2' \leftarrow$ createUniqueAttributeNameList($|s_2|$);
18 $adl_2' \leftarrow$ replaceAttributeNames($adl_2, anl_2 \rightarrow anl_2'$);
19 $anl_3 \leftarrow$ createUniqueAttributeNameList($|ael_3|$);
20 $dtl_3 \leftarrow$ computeDataTypeList(ael_3, qdg, i_1);
21 $adl_3 \leftarrow$ createAttributeDefinitionList(anl_3, dtl_3);
22 $adl_1' \leftarrow adl_1 \uplus adl_2' \uplus adl_3$;
23 setAttributeDefinitionList(c_1, adl_1');

24 $g_1 \leftarrow$ getGroupByClause(i_1);
25 $g_1' \leftarrow g_1 \cup ael_3$;
26 setGroupByClause(i_1, g_1');

27 $p_1' \leftarrow$ mergePredicates(p_1, p_2);
28 $w_1 \leftarrow$ getWhereClause(i_1);
29 $w_1' \leftarrow (\ w_1 - \langle\ p_1\ \rangle\ ) \uplus \langle\ p_1'\ \rangle$;
30 setWhereClause(i_1, w_1');

31 $al_3 \leftarrow$ createAttributeList(anl_3, ε);

32 $p_3 \leftarrow$ replaceExpressions$(p_1, ael_3 \rightarrow al_3)$;
33 $p_4 \leftarrow$ replaceExpressions$(p_2, ael_3 \rightarrow al_3)$;

34 **foreach** q **in** getDependentQueries(qdg, q_1) **do begin**
35 $i \leftarrow$ getInsertStatement(q);
36 addToWhereClause(i, t_1, p_3);
37 **end**;
38 **foreach** q **in** getDependentQueries(qdg, q_2) **do begin**
39 $i \leftarrow$ getInsertStatement(q);
40 addToWhereClause(i, t_2, p_4);
41 $i' \leftarrow$ replaceAttributeNames$(i, t_2, anl_2 \rightarrow anl_2')$;
42 $i'' \leftarrow$ replaceTable(i', t_2, t_1);
43 setInsertStatement(q, i'');
44 **end**;

45 **foreach** d **in** getDependenciesBySource(qdg, q_2) **do**
46 setSource(d, q_1);
47 **foreach** d **in** getDependenciesByDestination(qdg, q_2) **do**
48 removeDependency(qdg, d);

49 removeQuery(qdg, q_2);

C.2.5 ConcatQueries Rule

Rule Condition

Input: A query dependency graph qdg and a query q_1.
Output: The result of the condition test and the query that depends on q_1.

```
1   if intermediateResultQuery(q1) then begin
2     ql1 ← getDependentQueries(qdg, q1);
3     if |ql1| = 1 then begin
4       q2 ← getFirstElement(ql1);
5       c1 ← getCreateTableStatement(q1);
6       i1 ← getInsertStatement(q1);
7       i2 ← getInsertStatement(q2);
8       t1 ← getTargetTable(q1);
9       ⟨ s1, f1, w1, g1, h1 ⟩ ← getClauseList(i1);
10      ⟨ s2, f2, w2, g2, h2 ⟩ ← getClauseList(i2);
11      if isReferencedOnce(f2, t1) ∧
12        ¬ containsImplicitTypeCastDuringInsert(qdg, q1) then
13        if ¬ isDistinct(i1) ∧
14          ¬ containsAggregateExpression(s1) ∧
15          g1 = ∅ ∧ h1 = ∅
16        then return ⟨ true, q2 ⟩;
17        else if isDistinct(i1) ∧ isDistinct(i2) ∧
18          ¬ containsAggregateExpression(s1) ∧
19          ¬ containsAggregateExpression(s2) ∧
20          g1 = ∅ ∧ h1 = ∅ ∧
21          g2 = ∅ ∧ h2 = ∅
22        then return ⟨ true, q2 ⟩;
23        else if ¬ containsAggregateExpression(s2) ∧
24          g1 ≠ ∅ ∧
25          g2 = ∅ ∧ h2 = ∅ ∧
26          |f2| = 1
27        then return ⟨ true, q2 ⟩;
28        else return ⟨ false, ε ⟩;
29      else return ⟨ false, ε ⟩;
30    end else return ⟨ false, ε ⟩;
31  end else return ⟨ false, ε ⟩;
```

Rule Action

Input: A query dependency graph qdg, a query q_1 and a query q_2 that depends on q_1.
Output: None.

```
1   c1 ← getCreateTableStatement(q1);
2   i1 ← getInsertStatement(q1);
3   i2 ← getInsertStatement(q2);
4   t1 ← getTargetTable(q1);
5   ⟨ s1, f1, w1, g1, h1 ⟩ ← getClauseList(q1);
6   ⟨ s2, f2, w2, g2, h2 ⟩ ← getClauseList(q2);

7   cnl1 ← getCorrelationNameList(f1);
8   cnl1' ← createUniqueCorrelationNameList(|f1|);
9   ⟨ s1', f1', w1', g1', h1' ⟩ ← replaceCorrelationNames(⟨ s1, f1, w1, g1, h1 ⟩, cnl1 → cnl1');

10  tr2 ← getFirstElement(getTableReferenceListByTableName(f2, t1));
11  cn2 ← getCorrelationName(tr2);
12  anl1 ← getAttributeNameList(c1);
13  al1 ← createAttributeList(anl1, cn2);
14  s2' ← replaceExpressions(s2, al1 → s1');
15  f2' ← ( f2 − ⟨ tr2 ⟩ ) ⊎ f1';
16  w2' ← replaceExpressions(w2, al1 → s1');
17  g2' ← replaceExpressions(g2, al1 → s1');
18  h2' ← replaceExpressions(h2, al1 → s1');

19  if g1' = ∅ then begin
20      w2'' ← w2' ⊎ w1';
21      setClauseList(i2, ⟨ s2', f2', w2'', g2', h2' ⟩);
22  end else begin
23      g2'' ← g1';
24      h2'' ← w2' ⊎ h1';
25      w2'' ← w1';
26      setClauseList(i2, ⟨ s2', f2', w2'', g2'', h2'' ⟩);
27  end;

28  foreach d in getDependenciesBySource(qdg, q1) do
29      removeDependency(qdg, d);
30  foreach d in getDependenciesByDestination(qdg, q1) do
31      setDestination(d, q2);

32  removeQuery(qdg, q1);
```

C.2.6 PredicatePushdown Rule

Rule Condition

Input: A query dependency graph qdg and a query q_1.
Output: The result of the condition test and a list of predicates that can be pushed down.

```
1   if isIntermediateResultQuery(q1) then begin

2       c1 ← getCreateTableStatement(q1);
3       t1 ← getTargetTable(q1);

4       pl1 ← ∅;

5       foreach q in getDependendentQueries(qdg, q1) do begin
6           i ← getInsertStatement(q);
7           f ← getFromClause(i);
8           w ← getWhereClause(i);
9           trl ← getTableReferenceListByTableName(f, t1);
10          foreach tr in trl do begin
11              cn ← getCorrelationName(tr);
12              pl ← getPushdownPredicateListByTableReference(w, cn);
13              pl' ← replaceCorrelationNames(pl, ⟨cn⟩ → ⟨ε⟩);
14              pl1 ← pl1 ∩ pl';
15          end;
16      end;

17      if pl1 ≠ ∅
18      then return ⟨ true, pl1 ⟩;
19      else return ⟨ false, ∅ ⟩;

20  end else return ⟨ false, ∅ ⟩;
```

Rule Action

Input: A query dependency graph qdg, a query q_1 and a list of predicates that can be pushed down pl_1.
Output: None.

```
1   c_1 ← getCreateTableStatement(q_1);
2   i_1 ← getInsertStatement(q_1);
3   t_1 ← getTargetTable(q_1);

4   s_1 ← getSelectClause(i_1);
5   anl_1 ← getAttributeNameList(c_1);
6   al_1 ← createAttributeList(anl_1, ε);
7   pl'_1 ← replaceExpressions(pl_1, al_1 → s_1);

8   g_1 ← getGroupByClause(i_1);
9   if g_1 ≠ ∅ ∨ containsAggregateExpression(s_1) then begin
10      h_1 ← getHavingClause(i_1);
11      h'_1 ← h_1 ⊎ pl'_1;
12      setHavingClause(i_1, h'_1);
13  end else begin
14      w_1 ← getWhereClause(i_1);
15      w'_1 ← w_1 ⊎ pl'_1;
16      setWhereClause(i_1, w'_1);
17  end;

18  foreach q in getDependentQueries(qdg, q_1) do begin
19      i ← getInsertStatement(q);
20      removeFromWhereClause(i, t_1, pl_1);
21  end;
```

C.2.7 EliminateUnusedAttributes Rule

Rule Condition

Input: A query dependency graph qdg and a query q_1.
Output: The result of the condition test and
a list of names belonging to attributes
that are not used by queries that depend on q_1.

```
1   if intermediateResultQuery(q1) then begin
2     i1 ← getInsertStatement(q1);
3     if ¬ isDistinct(i1) then begin

4       t1 ← getTargetTable(q1);
5       anl ← ∅;
6       foreach q in getDependentQueries(qdg, q1) do begin
7         i ← getInsertStatement(q);
8         anl ← anl ∪ getAttributeNameListByTable(i, t1);
9       end;

10      c1 ← getCreateTableStatement(q1);
11      anl1 ← getAttributeNameList(c1);
12      anld ← anl1 − anl;

13      if anld ≠ ∅
14      then return ⟨ true, anld ⟩;
15      else return ⟨ false, ∅ ⟩;

16    end else return ⟨ false, ∅ ⟩;
17  end else return ⟨ false, ∅ ⟩;
```

Rule Action

Input: A query dependency graph qdg, a query q_1 and a list of names anl belonging to attributes that are not used by queries that depend on q_1.
Output: None.

```
1   c1 ← getCreateTableStatement(q1);
2   i1 ← getInsertStatement(q1);

3   adl1 ← getAttributeDefinitionList(c1);
4   s1 ← getSelectClause(i1);

5   adl'1 ← ∅;
6   s'1 ← ∅;
7   for k from 1 to |s1| do begin
8       adk ← getElement(adl1, k);
9       ank ← getAttributeName(adk);
10      aek ← getElement(s1, k);
11      if ank ∉ anl then begin
12          adl'1 ← adl'1 ⊎ ⟨ adk ⟩;
13          s'1 ← s'1 ⊎ ⟨ aek ⟩;
14      end;
15  end;

16  setAttributeDefinitionList(c1, adl'1);
17  setSelectClause(i1, s'1);
```

C.2.8 EliminateRedundantAttributes Rule

Rule Condition

Input: A query dependency graph qdg and a query q_1.
Output: The result of the condition test.

```
1    if intermediateResultQuery(q1) then begin
2        c1 ← getCreateTableStatement(q1);
3        i1 ← getInsertStatement(q1);

4        adl1 ← getAttributeDefinitionList(c1);
5        s1 ← getSelectClause(i1);

6        ec ← computeEquivalenceClasses(i1);

7        for k from 1 to |s1| − 1 do begin
8            adk ← getElement(adl1, k);
9            dtk ← getDataType(adk);
10           aek ← getElement(s1, k);
11           for l from k + 1 to |s1| do begin
12               adl ← getElement(adl1, l);
13               dtl ← getDataType(adl);
14               ael ← getElement(s1, l);
15               if elementOfSameEquivalenceClass(ec, aek, ael) ∧
16                   dtk = dtl
17               then return true;
18           end;
19       end;
20       return false;

21   end else return false;
```

Rule Action

Input: A query dependency graph qdg and a query q_1.
Output: None.

```
1   c_1 ← getCreateTableStatement(q_1);
2   i_1 ← getInsertStatement(q_1);
3   t_1 ← getTargetTable(q_1);

4   adl_1 ← getAttributeDefinitionList(c_1);
5   s_1 ← getSelectClause(i_1);

6   ec ← computeEquivalenceClasses(i_1);

7   adl'_1 ← ∅;
8   s'_1 ← ∅;
9   x ← ∅;
10  anl_old ← ∅;
11  anl_new ← ∅;

12  for k from 1 to |s_1| − 1 do
13      if k ∉ x then begin
14          ad_k ← getElement(adl_1, k);
15          dt_k ← getDataType(ad_k);
16          ae_k ← getElement(s_1, k);
17          adl'_1 ← adl'_1 ⊎ ⟨ ad_k ⟩;
18          s'_1 ← s'_1 ⊎ ⟨ ae_k ⟩;
19          for l from k + 1 to |s_1| do
20              if l ∉ x then begin
21                  ad_l ← getElement(adl_1, l);
22                  dt_l ← getDataType(ad_l);
23                  ae_l ← getElement(s_1, l);
24                  if elementOfSameEquivalenceClass(ec, ae_k, ae_l) ∧
25                      dt_k = dt_l
26                  then begin
27                      x ← x ∪ ⟨ l ⟩;
28                      an_k ← getAttributeName(ad_k);
29                      an_l ← getAttributeName(ad_l);
30                      anl_old ← anl_old ⊎ ⟨ an_l ⟩;
31                      anl_new ← anl_new ⊎ ⟨ an_k ⟩;
32                  end;
33              end;
34      end;
```

```
35  setAttributeDefinitionList(c_1, adl'_1);
36  setSelectClause(i_1, s'_1);

37  foreach q in getDependentQueries(qdg, q_1) do begin
38      i ← getInsertStatement(q);
39      i' ← replaceAttributeNames(i, t_1, anl_old → anl_new);
40      setInsertStatement(q, i');
41  end;
```

D
Experimental Results

D.1 Heuristic Approach

The following tables summarize the experimental results for the five original sequences, for the sequences after each rewritte step, and for the alternative single-query representations. The five original sequences are named *S1* to *S5*. The abbreviations after the name of an original sequence identify the rules applied to this original sequence. The abbreviation *wtg* stands for the WhereToGroup rule, *ms* stands for the MergeSelect rule, and *cq* stands for the ConcatQueries rule. They are listed in the order of application, e.g., *S1_wtg_wtg_cq* identifies sequence *S1* after applying the WhereToGroup rule twice and subsequently applying the ConcatQueries rule once. The abbreviation *sq* denotes the deep-nested single-query representation whereas *with* denotes the single-query representation that makes use of the WITH clause. For each sequence which we considered in our experiments, the following tables list the sequence identifier, the runtime of three runs, the average runtime of these three runs, and the runtime in percent relative to the runtime of the corresponding original sequence.

DB2 Runtimes

sequence	1st run	2nd run	3rd run	average	relative
S1	15,535	15,541	15,534	15,537	100%
S1_wtg	10,476	10,498	10,470	10,481	67%
S1_wtg_wtg	9,976	9,575	9,594	9,715	63%
S1_wtg_wtg_cq	2,989	3,005	3,004	2,999	19%
S1_wtg_wtg_cq_cq	3,110	3,006	3,016	3,044	20%
S1_sq	14,225	14,371	14,343	14,313	92%
S1_with	10,575	10,784	10,865	10,741	69%
S2	14,903	14,941	14,921	14,922	100%
S2_wtg	2,853	2,814	2,839	2,835	19%
S2_wtg_ms	2,156	2,133	2,125	2,138	14%
S2_wtg_ms_cq	35,118	34,119	34,644	34,627	232%
S2_wtg_ms_cq_cq	33,928	33,870	33,886	33,895	227%
S2_sq	7,671	7,653	7,684	7,669	51%
S2_with	2,722	2,746	2,708	2,725	18%
S3	23,417	23,312	23,526	23,418	100%
S3_wtg	18,457	18,349	18,411	18,406	79%
S3_wtg_wtg	17,530	17,451	17,601	17,527	75%
S3_wtg_wtg_ms	9,642	9,590	9,600	9,611	41%
S3_wtg_wtg_ms_cq	2,999	3,011	3,002	3,004	13%
S3_wtg_wtg_ms_cq_cq	3,017	3,014	3,015	3,015	13%
S3_sq	25,371	25,249	25,314	25,311	108%
S3_with	18,934	18,891	18,893	18,906	81%
S4	14,331	14,224	14,495	14,350	100%
S4_wtg	11,996	11,969	12,096	12,020	84%
S4_wtg_wtg	11,121	11,081	11,164	11,122	78%
S4_wtg_wtg_wtg	10,347	10,276	10,286	10,303	72%
S4_wtg_wtg_wtg_ms	6,386	6,123	6,169	6,226	43%
S4_wtg_wtg_wtg_ms_cq	3,523	3,544	3,507	3,525	25%
S4_wtg_wtg_wtg_ms_cq_cq	3,516	3,505	3,628	3,550	25%
S4_sq	20,939	20,781	20,851	20,857	145%
S4_with	12,195	12,198	12,245	12,213	85%
S5	18,504	18,464	18,642	18,537	100%
S5_wtg	16,200	16,157	16,261	16,206	87%
S5_wtg_wtg	15,385	15,503	15,249	15,379	83%
S5_wtg_wtg_wtg	14,461	14,578	14,491	14,510	78%
S5_wtg_wtg_wtg_ms	10,410	10,281	10,301	10,331	56%
S5_wtg_wtg_wtg_ms_ms	6,278	6,150	6,138	6,189	33%
S5_wtg_wtg_wtg_ms_ms_cq	3,529	3,520	3,520	3,523	19%
S5_wtg_wtg_wtg_ms_ms_cq_cq	3,541	3,506	3,528	3,525	19%
S5_sq	29,357	29,576	29,378	29,437	159%
S5_with	16,689	16,648	16,657	16,665	90%

Oracle Runtimes

sequence	1st run	2nd run	3rd run	average	relative
S1	5,129	4,865	5,349	5,114	100%
S1_wtg	4,576	4,555	4,597	4,576	89%
S1_wtg_wtg	4,043	4,023	4,129	4,065	79%
S1_wtg_wtg_cq	2,372	2,376	2,461	2,403	47%
S1_wtg_wtg_cq_cq	2,438	2,385	2,427	2,417	47%
S1_sq	7,290	7,640	7,424	7,451	146%
S1_with	18,665	18,020	17,732	18,139	355%
S2	8,046	7,627	7,665	7,779	100%
S2_wtg	2,827	2,833	2,772	2,811	36%
S2_wtg_ms	4,363	4,293	4,360	4,339	56%
S2_wtg_ms_cq	2,060	2,057	2,045	2,054	26%
S2_wtg_ms_cq_cq	2,038	2,053	2,052	2,048	26%
S2_sq	10,983	10,934	10,934	10,950	141%
S2_with	3,208	3,049	3,204	3,154	41%
S3	7,424	7,502	7,510	7,479	100%
S3_wtg	7,095	7,042	7,135	7,091	95%
S3_wtg_wtg	6,600	6,543	6,595	6,579	88%
S3_wtg_wtg_ms	4,064	4,042	4,073	4,060	54%
S3_wtg_wtg_ms_cq	2,414	2,388	2,417	2,406	32%
S3_wtg_wtg_ms_cq_cq	2,425	2,420	2,458	2,434	33%
S3_sq	12,248	12,235	12,172	12,218	163%
S3_with	4,109	4,136	4,155	4,133	55%
S4	7,121	7,151	7,102	7,125	100%
S4_wtg	6,776	6,772	6,802	6,783	95%
S4_wtg_wtg	6,220	6,262	6,324	6,269	88%
S4_wtg_wtg_wtg	5,834	5,824	5,842	5,833	82%
S4_wtg_wtg_wtg_ms	3,810	3,908	3,877	3,865	54%
S4_wtg_wtg_wtg_ms_cq	2,896	2,885	2,916	2,899	41%
S4_wtg_wtg_wtg_ms_cq_cq	2,887	2,849	2,888	2,875	40%
S4_sq	13,732	13,594	14,079	13,802	194%
S4_with	5,013	4,985	5,049	5,016	70%
S5	9,981	9,087	9,220	9,429	100%
S5_wtg	8,841	8,788	8,690	8,773	93%
S5_wtg_wtg	8,236	8,284	8,257	8,259	88%
S5_wtg_wtg_wtg	7,736	7,786	7,800	7,774	82%
S5_wtg_wtg_wtg_ms	5,777	5,805	5,834	5,805	62%
S5_wtg_wtg_wtg_ms_ms	3,823	3,882	3,876	3,860	41%
S5_wtg_wtg_wtg_ms_ms_cq	2,904	2,849	2,872	2,875	30%
S5_wtg_wtg_wtg_ms_ms_cq_cq	2,886	2,837	2,872	2,865	30%
S5_sq	18,867	18,637	18,678	18,727	199%
S5_with	6,013	5,961	6,026	6,000	64%

D.2 Cost-Based Approach

The following tables summarize the experimental results for the 10 query sequences, *S1* to *S10*, that span the search space for optimization of original sequence *S1*. We performed the experiments for each sequence with three different values for the constant used in the filter predicates within the query sequences. For each combination of sequence and constant value, runtimes have been measured with and without using the propagated statistics and cost estimates have been retrieved. The tables regarding runtimes show the runtime of three runs and the average runtime of these three runs. All values are in milliseconds. The table regarding cost estimates lists the summarized cost estimates. Since we retrieve these cost estimates from DB2, we use the DB2-specific cost unit which is called *timeron*.

DB2 Runtimes Not Using Propagated Statistics

sequence	constant value	1st run	2nd run	3rd run	average
S1	100,000	2,458,437	2,431,906	2,439,235	2,443,193
S2	100,000	1,871,812	1,874,890	1,900,501	1,882,401
S3	100,000	1,883,219	1,873,875	1,868,126	1,875,073
S4	100,000	1,888,875	1,875,281	1,873,032	1,879,063
S5	100,000	2,396,282	2,390,937	2,387,235	2,391,485
S6	100,000	1,170,000	1,173,391	1,182,781	1,175,391
S7	100,000	1,832,390	1,818,610	1,827,125	1,826,042
S8	100,000	1,827,985	1,824,891	1,827,470	1,826,782
S9	100,000	1,816,197	1,814,500	1,850,280	1,826,992
S10	100,000	1,122,485	1,131,484	1,127,640	1,127,203
S1	300,000	2,376,265	2,375,516	2,384,578	2,378,786
S2	300,000	1,863,454	1,805,828	1,808,892	1,826,058
S3	300,000	1,790,141	1,808,374	1,832,109	1,810,208
S4	300,000	1,779,421	1,794,125	1,798,202	1,790,583
S5	300,000	2,361,609	2,351,515	2,355,390	2,356,171
S6	300,000	1,126,609	1,090,469	1,091,734	1,102,937
S7	300,000	16,814,281	16,826,265	16,805,452	16,815,333
S8	300,000	16,900,297	16,872,078	16,889,001	16,887,125
S9	300,000	16,870,751	16,914,984	16,876,047	16,887,261
S10	300,000	55,205,094	55,155,547	55,149,459	55,170,033
S1	500,000	2,348,374	2,350,607	2,348,408	2,349,130
S2	500,000	1,798,125	1,781,063	1,772,204	1,783,797
S3	500,000	1,770,406	1,769,515	1,763,642	1,767,854
S4	500,000	1,790,280	1,755,766	1,760,531	1,768,859
S5	500,000	2,341,001	2,317,453	2,337,469	2,331,974
S6	500,000	1,071,688	1,065,219	1,077,672	1,071,526
S7	500,000	4,364,016	4,352,672	4,341,843	4,352,844
S8	500,000	4,384,968	4,364,500	4,758,797	4,502,755
S9	500,000	4,327,891	4,312,422	4,317,657	4,319,323
S10	500,000	1,053,437	1,057,203	1,056,656	1,055,765

DB2 Runtimes Using Propagated Statistics

sequence	constant value	1st run	2nd run	3rd run	average
S1	100,000	2,346,640	2,359,596	2,350,797	2,352,344
S2	100,000	1,780,843	1,788,984	1,777,219	1,782,349
S3	100,000	1,781,812	1,799,890	1,792,361	1,791,354
S4	100,000	1,769,172	1,785,905	1,766,843	1,773,973
S5	100,000	2,341,375	2,358,609	2,351,593	2,350,526
S6	100,000	1,074,766	1,087,297	1,084,611	1,082,225
S7	100,000	1,776,485	1,793,171	1,790,080	1,786,579
S8	100,000	1,785,609	1,770,375	1,797,345	1,784,443
S9	100,000	1,765,890	1,814,125	1,776,391	1,785,469
S10	100,000	1,138,032	1,132,047	1,114,657	1,128,245
S1	300,000	2,333,671	2,330,655	2,371,047	2,345,124
S2	300,000	1,803,921	1,782,297	1,823,515	1,803,244
S3	300,000	1,760,047	1,784,673	1,786,531	1,777,084
S4	300,000	1,766,923	1,770,295	1,758,360	1,765,193
S5	300,000	2,331,016	2,366,219	2,335,251	2,344,162
S6	300,000	1,051,454	1,068,156	1,056,454	1,058,688
S7	300,000	1,772,297	1,781,937	1,759,485	1,771,240
S8	300,000	1,769,188	1,764,610	1,754,470	1,762,756
S9	300,000	1,769,313	1,763,673	1,764,908	1,765,965
S10	300,000	1,086,517	1,079,187	1,083,890	1,083,198
S1	500,000	2,319,016	2,334,017	2,330,048	2,327,694
S2	500,000	1,772,594	1,761,219	1,763,094	1,765,636
S3	500,000	1,770,484	1,760,125	1,757,000	1,762,536
S4	500,000	1,745,546	1,750,250	1,759,985	1,751,927
S5	500,000	2,318,798	2,326,921	2,331,549	2,325,756
S6	500,000	1,054,125	1,049,515	1,058,765	1,054,135
S7	500,000	1,773,188	1,758,766	1,766,172	1,766,042
S8	500,000	1,762,577	1,760,782	1,802,484	1,775,281
S9	500,000	1,751,672	1,762,656	1,754,735	1,756,354
S10	500,000	1,060,281	1,059,172	1,047,766	1,055,740

DB2 Cost Estimates

sequence	constant value	cost estimate
S1	100,000	9,482,170
S2	100,000	6,918,625
S3	100,000	6,916,166
S4	100,000	6,915,253
S5	100,000	9,389,211
S6	100,000	3,848,696
S7	100,000	6,835,205
S8	100,000	6,832,746
S9	100,000	6,831,831
S10	100,000	3,765,246
S1	300,000	9,455,241
S2	300,000	6,861,590
S3	300,000	6,859,131
S4	300,000	6,858,195
S5	300,000	9,374,996
S6	300,000	3,791,365
S7	300,000	6,805,123
S8	300,000	6,802,663
S9	300,000	6,801,744
S10	300,000	3,734,821
S1	500,000	9,428,311
S2	500,000	6,818,254
S3	500,000	6,815,795
S4	500,000	6,814,817
S5	500,000	9,360,771
S6	500,000	3,747,687
S7	500,000	6,782,203
S8	500,000	6,779,744
S9	500,000	6,778,813
S10	500,000	3,711,545

Bibliography

[AAA+] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guízar, N. Kartha, C. K. Liu, R. Khalaf, D. König, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web Services Business Process Execution Language Version 2.0 - oasis standard. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf. Last visited on August 15, 2008.

[ACN00] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated Selection of Materialized Views and Indexes for SQL Databases. In *Proceedings of International Conference on Very Large Data Bases (VLDB), Cairo, Egypt*, September 2000.

[ACN06] S. Agrawal, E. Chu, and V. R. Narasayya. Automatic Physical Design Tuning: Workload as a Sequence. In *Proceedings of ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA*, June 2006.

[BC02] N. Bruno and S. Chaudhuri. Exploiting Statistics on Query Expressions for Optimization. In *Proceedings of ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA*, June 2002.

[BCG01] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: A Multidimensional Workload-Aware Histogram. *SIGMOD Record*, 30(2), 2001.

[Bel57] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[BW01] S. Babu and J. Widom. Continuous Queries over Data Streams. *SIGMOD Record*, 30(3), 2001.

[CDTW00] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data, Dallas, Texas, USA*, May 2000.

[CG94] R. L. Cole and G. Graefe. Optimization of Dynamic Query Evaluation Plans. In *Proceedings of ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA*, May 1994.

[Cha98] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proceedings of ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), Seattle, Washington, USA*, June 1998.

[Che06] K. K. Chen. Influence query optimization with optimization profiles and statistical views in DB2 9. http://www.ibm.com/developerworks/db2/library/techarticle/dm-0612chen/index.html, December 2006.

[Chr84] S. Christodoulakis. Implications of Certain Assumptions in Database Performance Evaluation. *ACM Transactions on Database Systems (TODS)*, 9(2), 1984.

[CKPS95] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing Queries with Materialized Views. In *Proceedings of International Conference on Data Engineering (ICDE), Taipei, Taiwan*, March 1995.

[CN07] S. Chaudhuri and V. R. Narasayya. Self-Tuning Database Systems: A Decade of Progress. In *Proceedings of International Conference on Very Large Data Bases (VLDB), Vienna, Austria*, September 2007.

[Cod69] E. F. Codd. Derivability, Redundancy and Consistency of Relations Stored in Large Data Banks. *IBM Research Report*, RJ599, 1969.

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6), 1970.

[DSRS01] N. N. Dalvi, S. K. Sanghai, P. Roy, and S. Sudarshan. Pipelining in Multi-Query Optimization. In *Proceedings of ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), Santa Barbara, California, USA*, May 2001.

[Fin82] S. Finkelstein. Common Subexpression Analysis in Database Applications. In *Proceedings of ACM SIGMOD International Conference on Management of Data, Orlando, Florida, USA*, June 1982.

[GG01] M. Garofalakis and P. Gibbons. Approximate Query Processing: Taming the TeraBytes. In *Proceedings of International Conference on Very Large Data Bases (VLDB), Roma, Italy*, September 2001.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-Query Processing in Data Warehousing Environments. In *Proceedings of International Conference on Very Large Data Bases (VLDB), Zürich, Switzerland*, September 1995.

[GL01] J. Goldstein and P.-A. Larson. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. In *Proceedings of ACM SIGMOD International Conference on Management of Data, Santa Barbara, California, USA*, May 2001.

[GMP02] P. Gibbons, Y. Matias, and V. Poosala. Fast Incremental Maintenance of Approximate Histograms. *ACM Transactions on Database Systems*, 27(3), 2002.

[GMUW01] H. Garcia-Molina, J. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, 2001.

[Gra95] G. Graefe. The Cascades Framework for Query Optimization. *IEEE Data Engineering Bulletin*, 18(3), 1995.

[Gra96] G. Graefe. The Microsoft Relational Engine. In *Proceedings of International Conference on Data Engineering (ICDE), New Orleans, Louisiana, USA*, February/March 1996.

[GW89] G. Graefe and K. Ward. Dynamic Query Evaluation Plans. In *Proceedings of ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA*, May/June 1989.

[HFLP89] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible Query Processing in Starburst. In *Proceedings of ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA*, May/June 1989.

[HK93] E. Hanson and L. Kollar. Statistics Used by the Query Optimizer in Microsoft SQL Server 2005. *Microsoft SQL Server TechCenter*, 1993.

[IBM04a] IBM Corp. *IBM DB2 Universal Database, Administration Guide: Performance, Version 8.2*, 2004.

[IBM04b] IBM Corp. *IBM DB2 Universal Database, SQL Reference Volume 1, Version 8.2*, 2004.

[IC91] Y. E. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *Proceedings of ACM SIGMOD International Conference on Management of Data, Denver, Colorado, USA*, May 1991.

[IC93] Y. E. Ioannidis and S. Christodoulakis. Optimal Histograms for Limiting Worst-Case Error Propagation in the Size of Join Results. *ACM Transactions on Database Systems*, 18(4), 1993.

[IMH+04] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies. In *Proceedings of ACM SIGMOD International Conference on Management of Data,*, June 2004.

[Ioa96] Y. E. Ioannidis. Query Optimization. *ACM Computing Surveys (CSUR)*, 28(1), 1996.

[Ioa03] Y. E. Ioannidis. The History of Histograms (abridged). In *Proceedings of International Conference on Very Large Data Bases (VLDB), Berlin, Germany*, September 2003.

[IP95a] Y. E. Ioannidis and V. Poosala. Balancing Histogram Optimality and Practicality for Query Result Size Eestimation. In *Proceedings of ACM SIGMOD International Conference on Management of Data, San Jose, California, USA*, May 1995.

[IP95b] Y. E. Ioannidis and V. Poosala. Histogram-Based Solutions to Diverse Database Estimation Problems. *Data Engineering Bulletin*, 18(3), 1995.

[IP99] Y. E. Ioannidis and V. Poosala. Histogram-Based Approximation of Set-Valued Query-Answers. In *Proceedings of International Conference on Very Large Data Bases (VLDB), Edinburgh, Scotland*, September 1999.

[KD98] N. Kabra and D. J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *Proceedings of ACM SIGMOD International Conference on Management of Data, Seattle, Washington, USA*, June 1998.

[KM07] T. Kraft and B. Mitschang. Statistics API: DBMS-independent Access and Management of DBMS Statistics in Heterogeneous Environments. In *Proceedings of International Conference on Enterprise Information Systems (ICEIS), Funchal, Madeira, Portugal*, June 2007.

[Kra07] T. Kraft. A Cost-Estimation Component for Statement Sequences. In *Proceedings of International Conference on Very Large Data Bases (VLDB), Vienna, Austria*, September 2007.

[KS04] T. Kraft and H. Schwarz. CHICAGO: A Test and Evaluation Environment for Coarse-Grained Optimization. In *Proceedings of International Conference on Very Large Data Bases (VLDB), Toronto, Canada*, August / September 2004.

[KSM07] T. Kraft, H. Schwarz, and B. Mitschang. A Statistics Propagation Approach to Enable Cost-Based Optimization of Statement Sequences. In *Proceedings of East-European Conference on Advances in Databases and Information Systems (ADBIS), Varna, Bulgaria*, September / October 2007.

[KSRM03] T. Kraft, H. Schwarz, R. Rantzau, and B. Mitschang. Coarse-Grained Optimization: Techniques for Rewriting SQL Statement Sequences. In *Proceedings of International Conference on Very Large Data Bases (VLDB), Berlin, Germany*, September 2003.

[LLS+02] S. Lightstone, G. M. Lohman, B. Smith, R. Horman, and J. Teng. A SMARTer DB2. *DB2 magazine*, 4, 2002.

[LMS95] A. Y. Levy, A. O. Mendelzon, and Y. Sagiv. Answering Queries Using Views. In *Proceedings of ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), San Jose, California, USA*, May 1995.

[LY85] P.-A. Larson and H. Z. Yang. Computing Queries from Derived Relations. In *Proceedings of International Conference on Very Large Data Bases (VLDB), Stockholm, Sweden*, August 1985.

[Mic] Microsoft Research. AutoAdmin: Self-Tuning and Self-Administering Databases. http://research.microsoft.com/dmx/AutoAdmin/. Last visited on August 15, 2008.

[Mic06] Microsoft Corp. SQL Server 2005 Books Online - Transact-SQL Reference. http://msdn2.microsoft.com/ en-us/library/ms189826.aspx, 2006.

[Mit95] B. Mitschang. *Anfrageverarbeitung in Datenbanksystemen - Entwurfs- und Implementierungskonzepte*. Vieweg, 1995.

[MRS+04] V. Markl, V. Raman, D. Simmen, G. M. Lohman, H. Pirahesh, and Miso Cilimdzic. Robust Query Processing through Progressive Optimization. In *Proceedings of ACM SIGMOD International Conference on Management of Data, Paris, France*, June 2004.

[MRSR01] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized View Selection and Maintenance Using Multi-Query Optimization. In *Proceedings of ACM SIGMOD International Conference on Management of Data, Santa Barbara, California, USA*, May 2001.

[Ora] Oracle Corp. Ask Tom. http://asktom.oracle.com/.

[Ora02] Oracle Corp. *Oracle9i Database Performance Tuning Guide and Reference, Release 2 (9.2)*, 2002.

[Ora03a] Oracle Corp. *Oracle Database Performance Tuning Guide, 10g Release 1 (10.1)*, 2003.

[Ora03b] Oracle Corp. *Oracle Database Reference, 10g Release 1 (10.1)*, 2003.

[Ora03c] Oracle Corp. *PL/SQL Packages and Types Reference, 10g Release 1 (10.1)*, 2003.

[PGI99] V. Poosala, V. Ganti, and Y. E. Ioannidis. Approximate Query Answering using Histograms. *IEEE Data Engineering Bulletin*, 22(4), 1999.

[PHIS96] V. Poosala, P. Haas, Y. E. Ioannidis, and E. Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. In *Proceedings of ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada*, June 1996.

[PI97] V. Poosala and Y. E. Ioannidis. Selectivity Estimation Without the Attribute Value Independence Assumption. In *Proceedings of International Conference on Very Large Data Bases (VLDB), Athens, Greece*, August 1997.

[PLH97] H. Pirahesh, T. Y. C. Leung, and W. Hasan. A Rule Engine for Query Transformation in Starburst and IBM DB2 C/S DBMS. In *Proceedings of International Conference on Data Engineering (ICDE), Birmingham, UK*, April 1997.

[PP98] M. Petkovic and L. Petkovic. *Complex Interval Arithmetic and Its Applications*. Wiley-VCH, 1998.

[PS88] J. Park and A. Segev. Using Common Subexpressions to Optimize Multiple Queries. In *Proceedings of International Conference on Data Engineering (ICDE), Los Angeles, California, USA*, February 1988.

[RC88] A. Rosenthal and U. Chakravarthy. Anatomy of a Modular Multiple Query Optimizer. In *Proceedings of International Conference on Very Large Data Bases (VLDB), Los Angeles, California, USA*, August/September 1988.

[RSSB00] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In *Proceedings of ACM SIGMOD International Conference on Management of Data, Dallas, Texas, USA*, May 2000.

[SAC+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA*, May 1979.

[Sel86] T. K. Sellis. Global Query Optimization. In *Proceedings of ACM SIGMOD International Conference on Management of Data, Washington, D.C., USA*, May 1986.

[Sel88] T. K. Sellis. Multiple-Query Optimization. *ACM Transactions on Database Systems (TODS)*, 13(1), 1988.

[SLMK01] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *Proceedings of International Conference on Very Large Data Bases (VLDB), Roma, Italy*, September 2001.

[SS91] T. K. Sellis and L. D. Shapiro. Query Optimization for Nontraditional Database Applications. *IEEE Transactions on Software Engineering*, 17(1), 1991.

[SS07] T. K. Sellis and A. Simitsis. ETL Workflows: From Formal Specification to Optimization. In *Proceedings of East-European Conference on Advances in Databases and Information Systems (ADBIS), Varna, Bulgaria*, September/October 2007.

[SVS05a] A. Simitsis, P. Vassiliadis, and T. K. Sellis. Optimizing ETL Processes in Data Warehouses. In *Proceedings of International Conference on Data Engineering (ICDE), Tokyo, Japan*, April 2005.

[SVS05b] A. Simitsis, P. Vassiliadis, and T. K. Sellis. State-Space Optimization of ETL Workflows. *IEEE Transactions on Knowledge Data Engineering.*, 17(10), 2005.

[Tra] Transaction Processing Performance Council (TPC). *TPC BENCHMARK^{TM}H (Decision Support), Standard Specification, Revision 2.7.0.*

[VSRM08] M. Vrhovnik, H. Schwarz, S. Radeschuetz, and B. Mitschang. An Overview of SQL Support in Workflow Products. In *Proceedings of International Conference on Data Engineering (ICDE), Cancún, México*, April 2008.

[VSS+07] M. Vrhovnik, H. Schwarz, O. Suhre, B. Mitschang, V. Markl, A. Maier, and T. Kraft. An Approach to Optimize Data Processing in Business Processes. In *Proceedings of International Conference on Very Large Data Bases (VLDB), Vienna, Austria*, September 2007.

[Wik] Wikipedia, the free encyclopedia. Normal Distribution. http://en.wikipedia.org/wiki/Normal_distribution. Last visited on August 15, 2008.

[WMHZ02] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback. Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. In *Proceedings of International Conference on Very Large Data Bases (VLDB), Hong Kong, China*, August 2002.

[ZCL+00] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering Complex SQL Queries Using Automatic Summary Tables. In *Proceedings of ACM SIGMOD International Conference on Management of Data, Dallas, Texas, USA*, May 2000.

[ZRL+04] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. J. Storm, C. Garcia-Arellano, and S. Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *Proceedings of International Conference on Very Large Data Bases (VLDB), Toronto, Canada*, August / September 2004.

[ZZL+04] D. C. Zilio, C. Zuzarte, S. Lightstone, W. Ma, G. M. Lohman, R. Cochrane, H. Pirahesh, L. S. Colby, J. Gryz, E. Alton, D. Liang, and G. Valentin. Recommending Materialized Views and Indexes with IBM DB2 Design Advisor. In *Proceedings of International Conference on Autonomic Computing (ICAC), New York, New York, USA*, May 2004.

Curriculum Vitae

Tobias Kraft

Date and place of birth:	February 4^{th}, 1976; Stuttgart, Germany
Nationality:	German

8/1982 – 7/1986	Primary School at the Osterholzschule in Ludwigsburg, Germany
8/1986 – 6/1995	Secondary School at the Otto-Hahn-Gymnasium in Ludwigsburg, Germany Degree: Abitur
9/1995 – 9/1996	Civilian service at a retirement home in Stuttgart, Germany
10/1996 – 4/2002	Studies in Computer Science at the Universität Stuttgart, Germany Degree: Diplom-Informatiker (Dipl. inf.)
6/2002 – 2/2009	Research staff member in the Department of Applications of Parallel and Distributed Systems at the Institute of Parallel and Distributed Systems, Universität Stuttgart, Germany

---