Institut für Parallele und Verteilte Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Studienarbeit Nr. 2304

Optimierung der physischen Operatoren einer nativen RDF-Datenbank für moderne Prozessoren

Tim Waizenegger

Studiengang: Informatik

Prüfer: Prof. Dr. Berhard Mitschang

Betreuer: Dipl.-Inf. Andreas Brodt

begonnen am: 18. Oktober 2010

beendet am: 19. April 2011

CR-Klassifikation: H.2.4, H.3.4

Inhaltsverzeichnis

1. Einleitung **5**

2. Grundlagen **7**
2.1. Grenzen der Hardware 7
2.2. RDF 9
2.3. Datenbank-Architektur 12
2.4. RDF-3X 12

3. Verwandte Arbeiten **15**

4. Architektur **23**
4.1. Verbesserungsansatz 23
4.2. Systemarchitektur 23
4.3. Operatorbaum 24
4.4. Blöcke und Spalten 25

5. Sideways Information Passing **27**
5.1. SIP in RDF-3X 27
5.2. SIP in der blockweisen Verarbeitung 28

6. Implementierung **33**
6.1. Codegenerierung 33
6.2. Block-Register 33
6.3. Operatoren 35

7. Evaluation **41**
7.1. Testumgebung 41
7.2. Ergebnisse 43
7.3. Diskussion 45

8. Zusammenfassung **47**

A. Listings **49**

B. SPARQL- und XML-Anfragen **51**

Literaturverzeichnis **53**

Abbildungsverzeichnis

2.1. Zuordnung von Cache-Lines zu Speichersegmenten 8
2.2. RDF-Graph 11
2.3. Operator-Baum mit einfachen Registern 13

4.1. RDF-3X Architektur mit neuen Komponenten 24
4.2. Operator-Baum mit Block-Registern 25

5.1. Block-Operator-Baum mit SIP-Kanten und Partner-Listen 30
5.2. Detailansicht SIP Vergleich von Blöcken 31

6.1. Block-Spalten im Detail 34
6.2. Partitionierter Hash-Verbund 37

7.1. Messergebnisse – Bestimmung der Anzahl von Partitionen 44
7.2. Messergebnisse – Bestimmung der Anzahl von Blockzeilen 44

Tabellenverzeichnis

7.1. Messergebnisse – Vergleich block- und tupelweise Ausführung, hohe Selektivität – Datensatz A 44
7.2. Messergebnisse – Vergleich block- und tupelweise Ausführung, niedrige Selektivität – Datensatz B 44

1. Einleitung

Motivation

Die Leistungsfähigkeit handelsüblicher CPUs wächst seit langem schneller als die Geschwindigkeit des Speichers. Diese größer werdende Lücke macht die Optimierung von Speicherzugriffen zu einem immer wichtigeren Werkzeug bei der Optimierung von Datenbanksystemen.

In dieser Studienarbeit wird daher eine alternative Implementierung der *RDF-3X* Datenbank-Engine vorgestellt, mit dem Ziel die Zugriffsmuster auf den Prozessor-Cache, und den Hauptspeicher zu verbessern, und dadurch Effizienzgewinne zu erzielen.

Gliederung

Kapitel 2 – Grundlagen: Stellt die Grundlagen der verwendeten Technologien dar, und beleuchtet kurz die Prinzipien der Optimierung.

Kapitel 3 – Verwandte Arbeiten: Betrachtet andere Arbeiten die ähnliche Ansätze verfolgt haben, oder relevante Ergebnisse beitragen konnten.

Kapitel 4 – Architektur: Gibt einen Überblick über die Aspekte der Optimierung und der neuen Datenbankarchitektur.

Kapitel 5 – Sideways Information Passing: Erörtert Möglichkeiten wie SIP im Kontext von Datenbanksystemen eingesetzt werden kann.

Kapitel 6 – Implementierung: Beschreibt die durchgeführten Optimierungen an der Datenbank, und gibt konkrete Einblicke in die Implementierung.

Kapitel 6.3.3 – Probleme und Hürden: Stellt die größten Schwierigkeiten und Herausforderungen an dieser Arbeit dar.

Kapitel 7 – Evaluation: Um den Erfolg der Optimierung zu beziffern, wurden Leistungsmessungen an dem neuen System durchgeführt.

Kapitel 8 – Zusammenfassung: Bewertet die Ziele und Ergebnisse der Arbeit rückblickend und gibt Hinweise auf neue Lösungsansätze.

Kapitel A – Listings: Enthält Auszüge der Code-Basis, die zum Verständnis beitragen sollen.

Aufgabenstellung

Die Operatoren der *RDF-3X* Datenbank [NW10] sind im sogenannten *Volcano-Stil* nach [Gra94] implementiert. Jeder Operator hat daher drei Methoden (*open(), next() und close()*) die von seinem Elternoperator aufgerufen werden. Die Operatoren bilden einen Operatorbaum, dessen Blätter Index-Scans darstellen. Join- und andere Operatoren bilden die Knoten, sodass Tupel über die Stufen des Baumes propagiert werden bis an der Wurzel die Ergebnisse vorliegen. Die *open()* Methode, initialisiert den Operator, und bereitet die Produktion des ersten Tupels vor. Eine *next()* Methode wird wiederholt aufgerufen bis das Ergebnis vollständig ist, da in jedem Aufruf genau ein Tupel produziert wird. Schließlich beendet der Aufruf einer *close()* Methode die Produktion von Tupeln.

Die Operatoren arbeiten in einer Pipeline, die jedoch zu jedem Zeitpunkt nur ein Tupel enthält. Daher kann ein Operator wenn er aufgerufen wird, nur ein Tupel verarbeiten, bevor er die Ausführung an den Eltern-Operator abgibt. Dies führt zu einer schlechten Ausnutzung des Prozessorcaches.

Die Operatoren sollen optimiert werden, und die Ausführung der Pipeline angepasst, sodass ein Operator eine Menge von Tupeln produzieren kann und somit weniger Sprünge in der Ausführung entstehen.

2. Grundlagen

Dieses Kapitel behandelt kurz die Technologien die dieser Arbeit zu Grunde liegen, und soll einen Überblick der Rahmenbedingungen geben die in dem Verbesserungsansatz Beachtung finden.

2.1. Grenzen der Hardware

Die Entwicklung effizienter Software kann nur unter Berücksichtigung von Leistungsaspekten der verwendeten Hardware erfolgreich sein. Das Ziel dieser Arbeit soll daher erreicht werden, indem die Ausführung der Datenbank besser an die Rahmenbedingungen der Hardware angepasst wird. Dieses Kapitel soll die relevanten Aspekte moderner Rechnerarchitekturen kurz aufzeigen.

2.1.1. Speicherhierarchie und Cache-Lines

Um den großen Unterschied zwischen Speicher- und Rechengeschwindigkeit auszugleichen wird in modernen Rechnerarchitekturen der Speicher hierarchisch organisiert. So wird es möglich sowohl von schnellem, aber teurem Speicher, wie auch von langsamem Speicher mit großer Kapazität zu profitieren, indem direkte Speicherzugriffe nur auf der schnellsten Ebene erfolgen. Die Hierarchien der Speicherarchitektur sind daher ihrer Kapazität und Geschwindigkeit nach angeordnet. Die CPU greift dabei nur auf die unterste Ebene, den *Prozessor-Cache* direkt zu. Zugriffe auf höhere Ebenen erfolgen indirekt, indem der angeforderte Speicherbereich Ebene für Ebene kopiert wird, bis er schließlich in dem Prozessor-Cache angelangt ist [HR01].

Der Hauptspeicher aktueller Systeme umfasst üblicherweise mehrere GB, während der Prozessor-Cache nur wenige MB vorhalten kann. *Cache-Lines* bieten daher eine Möglichkeit den großen Adressraum des Hauptspeichers auf den Cache abzubilden. Eine Cache-Line repräsentiert einen zusammenhängenden Bereich des Hauptspeichers im Cache, und ist damit die kleinste Einheit von Daten, die der Prozessor aus dem Hauptspeicher lesen, oder schreiben kann. Da Cache-Lines eine konstante Größe haben, kann anhand der Kapazität des Caches die Anzahl der Cache-Lines einfach bestimmt werden. Eine heute übliche Cache-Line-Größe von *64 Byte* wie sie unter andrem in der *Intel Core2-Architektur* Einsatz findet, führt auf einer CPU mit *1 MB Cache* zu einer Anzahl von *16384 Cache-Lines*. Dies repräsentiert nun die Anzahl der verteilten Speicherbereiche des Hauptspeichers, auf welche die CPU gleichzeitig zugreifen kann.

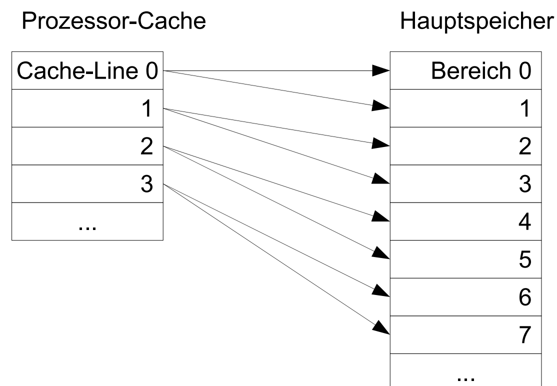

Abbildung 2.1.: Zuordnung von Cache-Lines zu Speichersegmenten

Um den gesamten Adressraum des Hauptspeichers abzudecken, sind jeder Cache-Line eine Vielzahl von zusammenhängenden Speicherbereichen zugeordnet wie Abbildung 2.1 verdeutlicht.

2.1.2. Translation Lookaside Buffer

Der *Translation Lookaside Buffer* (TLB) in modernen Prozessoren unterstützt das Betriebssystem bei der Verwaltung des virtuellen Speichers, indem er einen Teil der Pagingtabelle aus dem Hauptspeicher lokal vorhält. Findet ein Zugriff auf eine Speicheradresse statt, die dem TLB nicht bekannt ist, so muss zuerst der betreffende Teil der Pagingtabelle aus dem Hauptspeicher geladen werden, bevor der eigentliche Speicherzugriff erfolgen kann. Üblicherweise werden zwischen *64* und *128* Einträge im TLB vorgehalten. Dadurch sind bereits bei kleinen Datenstrukturen wie Hash-Tabellen, die eine ausreichende Menge an verteilten Speicheradressen referenzieren, zusätzliche Zugriffe zur Auflösung der virtuellen Adressen notwendig.

2.1.3. Prozessor-Pipeline

Die Ausführung einer Instruktion durch die CPU unterteilt sich in mehrere Phasen, welche sequenziell abzuarbeiten sind. Jede dieser Phasen wird von einer anderen Aufführungseinheit des Prozessors bearbeitet.

Um die hohe Taktgeschwindigkeit moderner Prozessoren zu ermöglichen, wird deshalb davon abgesehen eine Instruktion vollständig innerhalb eines Taktzyklus zu verarbeiten. Moderne CPUs führen Instruktionen daher in einer mehrstufigen Pipeline aus, um die unterschiedlichen Ausführungseinheiten des Prozessors zu jedem Zeitpunkt optimal auszulasten. Die Pipeline enthält stets eine Reihe von Instruktionen, die sich in unterschiedlichen Ausführungsphasen befinden. Das Ergebnis einer Instruktion ist erst verfügbar wenn sie die gesamte Pipeline durchlaufen hat. Bei optimaler Ausnutzung dieser Pipeline kann jedoch

in jedem Taktzyklus eine neue Instruktion die Pipeline betreten, während eine andere sie verlässt.

Diese Vorgehensweise erfordert, dass dem Prozessor eine lange Sequenz von Instruktionen vorliegt, welche er ununterbrochen ausführen kann. Befinden sich bedingte Sprünge in der Ausführung, so muss der Prozessor das Ergebnis der Bedingung vorhersagen um den nächsten Ausführungszweig in die Pipeline zu laden.

2.2. RDF

Das *Resource Description Framework* ist ein schemafreies Datenmodell zur Repräsentation strukturierter Informationen. RDF wurde im Kontext des *Semantic Web* entwickelt, und soll einen Standard bereitstellen, um semantische Informationen zu Web-Ressourcen zu beschreiben. Nachdem RDF 1999 vom *W3C* spezifiziert wurde, fand es kaum Beachtung bis es durch die wachsende Beliebtheit der No-SQL Datenbanken in den letzten Jahren wieder in den Fokus der Entwickler gerückt ist. Durch seine Schemafreiheit ist RDF gut geeignet für dynamisch wachsende Datenbestände wie Wissensdatenbanken, oder als offenes Austauschformat für soziale Netzwerke.

Hinter RDF steht die Idee, dass Gegenstände und Ressourcen Eigenschaften besitzen, und diese Eigenschaften mit einem Wert beschrieben werden können. Daher wird in RDF eine Ressource durch die Summe ihrer Eigenschaften beschrieben, welche als eine Menge von *Tripeln* dargestellt wird. Ein solches drei-stelliges Tupel setzt sich aus folgenden Bestandteilen zusammen:

Subjekt - Identifiziert die zu beschreibende Ressource,

Prädikat - Identifiziert eine bestimmte Eigenschaft dieser Ressource, und

Objekt - Beschreibt den Wert dieser Eigenschaft.

Wobei die ersten beiden Parameter Subjekt, und Prädikat immer eine *URI* darstellen (siehe unten). Das Objekt kann als *Literal* einen einfachen Wert darstellen, oder als URI andere Tripel referenzieren.

Folgende Aussage über einen Film: *Moonraker wurde 1979 gedreht* kann damit in RDF durch folgendes Tripel ausgedrückt werden:

```
<Moonraker> <releasedIn> 1979 .
```

RDF orientiert sich damit eng an der natürlichen Sprache und ihrer Grammatik, wodurch für Menschen das Lesen und Verstehen von RDF-Daten vereinfacht wird. Um jedoch den Austausch von Daten zwischen Computersystemen zu ermöglichen müssen weitere Anforderungen erfüllt sein. So ist es nötig, dass Subjekt, Prädikat und Objekt eindeutig identifizierbar sind, um die Verwechslung mit anderen Bezeichnern zu vermeiden die ähnlich, oder gleich geschrieben werden. Dieses Problem wird begünstigt durch die schemalose Natur von RDF, in der jeder Benutzer oder Entwickler seine eigenen Bezeichner vergeben kann. So könnten in verschiedenen Datenbanken mit dem Titel *Moonraker* unterschiedliche

Ressourcen identifiziert werden, oder das Erscheinungsjahr mit einem anderen Prädikat beschrieben werden.

Um diesen Anforderungen gerecht zu werden bedient sich RDF an dem weit verbreiteten Konzept des *URI* (Uniform Resource Identifier). Im Gegensatz zu einem *URL* (Uniform Resource Locator), der den Ort beschreibt an der eine bestimmte Ressource gefunden werden kann, identifiziert ein URI die Ressource lediglich. Da die Menge aller URLs eine Untermenge aller URIs darstellt ist es möglich, wenn auch nicht garantiert, dass ein in RDF vorkommender URI auf eine tatsächliche Ressource im Web verweist.

Der Menge aller URIs spannt einen hierarchischen Namensraum auf, an dem sich RDF bedient, um eindeutige Schlüssel zur Identifikation von Ressourcen zu erhalten.

Die Ressourcen werden daher um einen URI erweitert:

```
<http://example.org#Moonraker> <http://www.w3.org#releasedIn> 1979 .
```

Strukturen und Ressourcen werden in RDF durch eine Menge von Tripeln ausgedrückt, von denen jedes eine Eigenschaft abbildet.

```
<http://example.org#Moonraker> <http://www.w3.org#releasedIn> 1979 .
<http://example.org#Moonraker> <http://www.w3.org#runningTime> "126m" .
<http://example.org#Moonraker> <http://www.w3.org#writtenBy> "Ian Fleming" .
<http://example.org#Moonraker> <http://www.w3.org#hasCharacter>
        <http://example.org#James_Bond> .
<http://example.org#James_Bond> <http://www.w3.org#actedBy> "Roger Moore" .
```

Eine Menge solcher Tripel kann als Graph repräsentiert werden, indem zunächst für Ressourcen und Literale aus Subjekt und Objekt Knoten erzeugt werden. Die Kanten des Graphs bilden die Prädikate aus dem jeweiligen Tripel, wie in Abbildung 2.2 deutlich wird.

2.2.1. SPARQL

SPARQL (SPARQL Protocol and RDF Query Language) ist eine SQL-ähnliche Anfragesprache für RDF-Datenbanken, und wurde vom W3C mit speziellem Hinblick auf RDF entwickelt.

Um der Tripel-Repräsentation in RDF gerecht zu werden, bedient sich SPARQL einer musterbasierten Suche für Tripel. Eine Query setzt sich daher aus einer Menge von Tripel-Mustern zusammen, die über *Konstanten* bestimmte Komponenten einschränken, und über *Variablen* eine Verknüpfung von Tripeln erlauben.

Eine Anfrage die unser Beispieltripel aus dem vorherigen Kapitel benutzt, und das Erscheinungsjahr von Moonraker abfragt, könnte wie folgt aussehen:

```
select ?year
where { http://example.org#Moonraker http://www.w3.org#releasedIn ?year . }
```

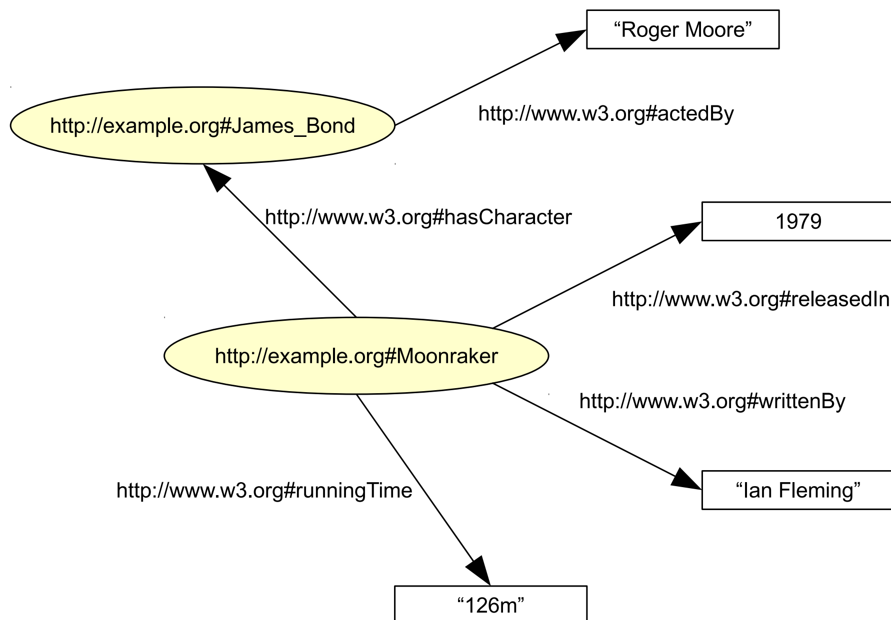

Abbildung 2.2.: RDF-Graph

Enthält die Datenbank noch weitere Informationen über Moonraker, so könnten alle entsprechenden (Prädikat, Objekt)-Paare mit folgender Anfrage produziert werden:

```
select ?predicate, ?object
where { http://example.org#Moonraker ?predicate ?object . }
```

Das Ergebnis dieser Anfrage wäre folgendes:

```
<http://www.w3.org#releasedIn> 1979 .
<http://www.w3.org#runningTime> "126m" .
<http://www.w3.org#writtenBy> "Ian Fleming" .
<http://www.w3.org#hasCharacter> <http://example.org#James_Bond> .
```

Eine Variable in einer SPARQL-Anfrage kann drei Zwecke erfüllen. Wie in den obigen Beispielen kann eine Komponente mit einer Variablen projiziert werden, indem sie mit `select` ausgewählt wird. Eine Variable kann auch als Platzhalter, oder *Wildcard* dienen, wenn sie nur einmal in der Anfrage vorkommt, und nicht zur Projektion ausgewählt wurde. Kommt eine Variable in mehreren Tripeln der Anfrage vor, so werden nur diejenigen Tripel als Ergebnis ausgewählt, in denen die entsprechende Komponente übereinstimmt. Folgende Anfrage verdeutlicht letzteren Fall, in dem die Namen aller Schauspieler ausgegeben werden, die in Moonraker aufgetreten sind.

```
select ?actor
where { ?character http://www.w3.org#actedBy ?actor .
        http://example.org#Moonraker http://www.w3.org#hasCharacter ?character . }
```

2.3. Datenbank-Architektur

Der für diese Arbeit relevante Teil moderner Datenbankarchitekturen, ist die Anfrageverarbeitung. Diese verwendet zur Produktion von Ergebnissen einen *Operatorgraph*, der in der verwendeten Datenbank RDF-3X die Form eines Baums annimmt. Die Produktion von Ergebnissen erfolgt in diesem Modell, indem eine Menge von Tupeln gelesen wird, um daraus anschließend das Ergebnis auszuwählen. Die Knoten dieses Graphen bilden Operatoren, die sich in drei Klassen einteilen lassen.

- Die Blätter bilden *Scanoperatoren* deren Aufgabe es ist, Tupel von Werten aus den physischen Indexen auf der Festplatte zu lesen.
- *Baumoperatoren* haben die Aufgabe, Tupel von einem oder mehreren Kindoperatoren zu verarbeiten, und dabei eine bestimmt Bedingung zu prüfen, welche die Teilnahme des Tupels an dem Ergebnis bestimmt.
- Der *Wurzeloperator* ist schließlich dafür zuständig, das nun vollständige Ergebnis welches er von seinem Kindoperator erhält, für den Benutzer auszugeben.

In dieser Arbeit werden zwei Typen von Baumoperatoren betrachtet, die beide *Verbundoperatoren* darstellen. Ein solcher Operator hat zwei Kindoperatoren, und vergleicht eine Komponente der Tupel beider Eingaberelationen. Er findet somit Partnertupel, in welchen die ausgewählte Komponente übereinstimmt, und produziert ein neues Verbundtupel für jedes Paar von Partnertupeln. Die zwei verwendeten Typen dieser Operatoren der Sortiereigenschaft der jeweiligen Eingaberelationen gerecht. Liegen beide dieser Relationen in sortierter Reihenfolge vor, so kann ein *Misch-Verbund* verwendet werden. Dieser Iteriert über seine Eingaberelationen, und produziert synchron dazu Ergebnisse. Die Sortiertheit der Relationen ermöglicht es dabei, Tupel ohne Partner schnell zu Identifizieren.

Liegen die Tupel der Eigaberelationen nicht in sortierter Reihenfolge vor, so kommt ein *Hash-Verbund* zum Einsatz. Dieser fügt sämtliche Tupel einer Relation in eine Hashtabelle ein, in welcher er anschließend für Tupel der anderen Relation Partner sucht.

2.4. RDF-3X

Dieser Arbeit liegt die Implementierung der nativen RDF-Datenbank *RDF-3X*[1] von Thomas Neumann zu Grunde. Die experimentelle Open-Source Datenbank *RDF-3X* (RDF Triple eXpress) stellt eine komplette Neuentwicklung dar, und wurde in Hinblick auf das schnelle Query-Processing auf großen Tripelmengen entworfen.

Dem zu Grunde liegt die physische Anordnung der Daten auf der Festplatte. Die Datenbank teilt sich dabei in zwei Bereiche. Ein *Dictionary*, und *Indexe*. Das Dictionary speichert die Werte der Literale und Ressourcen, und weist ihnen fortlaufende numerische Werte zu.

[1] http://www.mpi-inf.mpg.de/~neumann/rdf3x

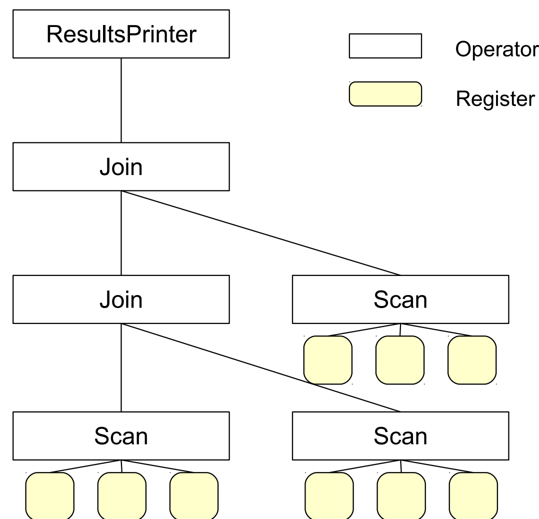

Abbildung 2.3.: Operator-Baum mit einfachen Registern

Diese Zuweisung erfolgt beim Erstellen einer Datenbank, sodass die Anfrageverarbeitung vollständig auf diesen *Dictionary-IDs* durchgeführt werden kann. Die *Indexe* stellen demnach sortierte Listen aus Dictionary-IDs dar. Um einen Index aufzubauen, werden sämtliche Tripel in Form ihrer numerischen IDs in eine Liste eingefügt, welche dann absteigend nach allen drei Spalten sortiert wird. In der Anfrageverarbeitung können Tripel anhand ihrer ID nun schnell in der sortierten Liste gefunden werden. Um auch Anfragen nach Prädikat oder Objekt schnell zu beantworten, existiert ein solcher Index für jede Permutation aus (S, P, O). Zusätzlich zu diesen vollständigen Indexen, werden aggregierte Indexe gebildet, welche nur noch zwei, respektive eine der Komponenten der Tripel enthalten. So existiert z.B. ein (S, P) Index mit dem sich Anfragen schnell beantworten lassen, die das Objekt ignorieren. Da die Indexe sortierte numerische Werte enthalten, werden sie zudem in komprimierter Form auf der Festplatte abgelegt. Zu diesem Zweck kommt eine *Delta-Kodierung* zum Einsatz, welche lediglich die Differenz von einem Wert zum nächsten Speichert. Dadurch müssen weniger Daten von der Festplatte gelesen werden, und die tatsächlichen IDs lassen sich leicht als Summe aus Anfangswert und Differenz bestimmen [NW09a].

Die Ausführung einer Anfrage teilt sich dann in zwei Phasen auf. Zunächst überführt der Parser die SPARQL-Anfrage in eine interne Repräsentation, den *Anfrage-Graph*. Anschließend wird mit Methoden der dynamischen Programmierung daraus eine Menge von möglichen Ausführungsplänen generiert, aus denen der optimale *Anfrage-Plan* ausgewählt wird.

Die Code-Generierung erzeugt anschließend daraus das Laufzeitsystem bestehend aus dem Operatorbaum und einer Laufzeitumgebung, dessen Ausführung die zweite Phase darstellt. Scans bilden die Blätter des Operator-Baums, und produzieren sortierte Listen von Tupeln, die sie aus den komprimierten Speicherseiten der Indexe lesen.

Operatoren bilden die Knoten, welche Verbundoperationen (*Join*) auf den Tupeln von zwei Kindknoten durchführen. In dieser Arbeit werden zwei Implementierungen von Join-

Operatoren verwendet, ein *Misch-Verbund* für Knoten an denen beide Eingabeströme in der gleichen, sortierten Reihenfolge ankommen, sowie einen *Hash-Verbund*, der keine bestimmte Reihenfolge der Eingabetripel voraussetzt.

Die Wurzel des Operator-Baums bildet schließlich ein Ausgabe-Operator, der die Ergebnisse zurück in Literale und Ressourcen auflöst.

Zur Zwischenspeicherung der Tupel dienen *Register*, welche genau einen einzelnen Wert aufnehmen. Ein Tupel wird daher durch eine Menge von Registern repräsentiert, von denen jedes eine Komponente des Tupels enthält.

Wie in Abbildung 2.3 zu erkennen ist, sind jedem Scan dabei drei solche Register zugewiesen in welche er die Komponenten seines zuletzt gelesenen Tripels schreibt. Die im Baum höher gelegenen Join-Operatoren greifen auf dieselben Register lesend zu. Sie vergleichen den Inhalt zweier Register, um festzustellen, ob die zugehörigen Tripel die Join-Bedingung erfüllen. Ist das der Fall, wird die Ausführung an den Elternknoten übergeben, welcher die nächste Join-Bedingung prüft. Wenn ein Tripel nicht zu einem Join-Ergebnis beiträgt, wird sein produzierender Operator aufgerufen um das nächste Tripel bereitzustellen. Dieser Aufruf pflanzt sich fort zu den Scan-Operatoren auf der untersten Ebene, sodass diese weitere Tupel aus dem Index lesen und damit die Ausführung der höher gelegenen Operatoren fortgesetzt werden kann.

3. Verwandte Arbeiten

Die Optimierung von Datenbanken ist eng an Rahmenbedingungen wie die Hardwarearchitektur, und Anwendungsfelder gebunden. Dies macht sie zu einem stets wachsenden Forschungsgebiet, aus dem im Folgenden relevante Erkenntnisse und Ansätze für diese Arbeit vorgestellt werden.

Die effiziente Ausführung der Operatoren sieht sich in zeilen-, wie in spaltenorientierten Datenbanken den selben Herausforderungen bezüglich Speicherlokalität und Prozessordurchsatz gegenübergestellt, weshalb Verbesserungsvorschläge aus beiden Welten betrachtet werden.

Der Ansatz der *blockweisen* Verarbeitung von Tupeln ist in zahlreichen Arbeiten und Implementierungen zu finden, und stellt einen wichtigen Grundbaustein der Datenbankoptimierung dar. Er kommt bei Datenbankarchitekturen nach dem *Volcano-Stil* [Gra94] zum Einsatz, und sieht einen Puffer zwischen produzierendem, und konsumierendem Operator vor.

In [SKN94] werden zwei allgemeine Ansätze zur Verbesserung der Cache-Lokalität von Daten in der Anfrageverarbeitung von Datenbanksystemen vorgestellt, die beide in dieser Arbeit Verwendung finden. Das Ziel beider Ansätze ist es, die Daten derart im Speicher anzulegen, dass ihre Anordnung den Zugriffsmustern der darauf zugreifenden Algorithmen entgegenkommt. Dazu wurde *Block-Bildung*, und *Partitionierung* vorgeschlagen. Partitionierung richtet sich an Algorithmen, die verteilte Zugriffsmuster auf große Datenstrukturen aufzeigen. Dies betrifft im vorliegenden Fall Suchoperationen wie sie in Hashtabellen vorkommen. Um verteilte Zugriffe zu vermeiden, werden die Daten in Partitionen aufgeteilt, deren Größe kleiner als der zur Verfügung stehende Prozessorcache sein soll. Der Algorithmus soll anschließend derart modifiziert werden, dass sich Speicherzugriffe zu jedem Zeitpunkt auf nur eine Partition beschränken. Auf diese Weise entfallen verteilte Zugriffe auf den Hauptspeicher, und die Ausführungszeit des Algorithmus verbessert sich. Die Block-Bildung sieht die Modifikation von Algorithmen vor, welchen einen Bereich des Speichers mehrfach lesen. Der Algorithmus wird so angepasst, dass sich wiederholte Zugriffe auf Speicherbereiche auf einen Block von Daten beschränken. Dadurch befinden sich diese Daten bereits im Prozessorcache, und müssen nicht erneut aus dem Hauptspeicher gelesen werden, was ebenfalls zu einer Steigerung der Ausführungsgeschwindigkeit führt.

In [BKM08] werden die leistungsbestimmenden Faktoren der spaltenorientierten Datenbank *MonetDB* vorgestellt, deren Architektur mit besonderem Hinblick auf die Speicherlokalität von Daten, sowie Instruktionen entwickelt wurde.

Im Gegensatz zu zeilenorientierten Datenbanken, verwendet MonetDB eine vertikale Speicherorganisation um relationale Tabellen zu repräsentieren. Dazu wird jede Spalte auf eine

zweispaltige Tabelle, die *binary association table (BAT)* abgebildet. Deren erste Spalte enthält einen Objekt-Bezeichner, welcher die ursprüngliche Zeile der relationalen Tabelle identifiziert, und somit die Verknüpfung mehrere BATs erlaubt. Die zweite Spalte enthält schließlich den zugehörigen physischen Wert als explizite Angabe, oder Referenz auf eine Zuordnungstabelle bei großen Werten.

Im Speicher werden die BATs jedoch in einer optimierten Form abgelegt. Diese macht sich die Anordung der Objekt-Bezeichner zu Nutze, welche stets ansteigend sortierte, numerische Werte sind. Aufgrund der relationalen Struktur der zugrunde liegenden Tabellen, ist die Reihenfolge dieser Bezeichner in allen BATs dieselbe. Daher kann die erste BAT-Spalte verdichtet werden, sodass sie eine ununterbrochene Sequenz bildet. Nun kann diese Bezeichner-Spalte vollständig verworfen, und als *virtuelle Spalte* betrachtet werden, deren Werte sich jederzeit aus der Zeilenposition der BAT rekonstruieren lassen.

Durch diese geschickte Spaltenorganisation lässt sich nicht nur der Speicherbedarf einer BAT reduzieren, sondern zugleich die Zugriffszeit auf korrespondierende Zeilen in anderen BATs verbessern. Letzteres geschieht indem die Funktionsweise der Speicherverwaltungseinheit der CPU ausgenutzt wird. Der Objekt-Bezeichner welcher aus der virtuellen ersten Spalte der BAT gewonnen wird, entspricht nun genau dem Versatz der Anfangsposition der BAT im Speicher zu der gewünschten Zeile.

In MonetDB findet sich zudem eine interessante Herangehensweise an den *Hash-Verbund Operator* die in dieser Arbeit verwendet wird (siehe Kapitel 6.3.1). Dabei sind zwei wesentliche Merkmale dieses Ansatzes festzustellen. Der Hash-Verbund verwendet *partitionierte Hash-Tabellen* [BMK99], wobei die Anzahl der Partitionen während der Aufbauphase *stufenweise* erhöht wird.

Der traditionelle Hash-Verbund fügt sämtliche Tupel einer Eingabe-Relation in eine Hash-Tabelle ein, um anschließend für jedes Tupel der andern Relation einen Partner in dieser zu finden. Der vorgestellte Ansatz des partitionierten Hash-Verbundes fügt die Tupel nicht in eine, sondern in mehrere Hash-Tabellen (Partitionen) ein, indem über eine weitere Hash-Funktion jedes Tupel einer Partition zugeordnet wird. Diese Partitionierung erfolgt auf beiden Eingabe-Relationen, wobei nur eine davon Hash-Tabellen als Partitionen verwendet.

Der Verbund kann nun erfolgen indem für jedes Tupel einer Partition ein Partner in der gegenüberliegenden Partition gesucht wird. Durch die Partitionierung nach Hash-Werten ist nun sichergestellt, dass sich ein potentielles Partner-Tupel stets in derselben Partition auf der gegenüberliegenden Seite befindet.

Die Größe der Partitionen wird dabei so gewählt, dass die Anzahl der Felder der Hashtabelle (Buckets) die Menge an Cache-Lines des Prozessors nicht übersteigt. Dadurch werden minimale Zugriffszeiten auf die Hashtabelle sichergestellt, da der zusätzliche Aufwand für verteilte Speicherzugriffe bei großen Hashtabellen entfällt.

Einen Nachteil dieses Verfahrens stellt jedoch die Aufbauphase der Partitionen dar. Da die maximale Größe der Hashtabellen beschränkt sein soll, steigt entsprechend die Anzahl der

Partitionen mit der Menge der einzufügenden Tupel. Übersteigt nun die Anzahl der Partitionen ihrerseits die Anzahl der Cache-Lines, entsteht an dieser Stelle zusätzlicher Aufwand für die verteilten Speicherzugriffe auf unterschiedliche Partitionen.

Der *Radix-Cluster* Algorithmus löst dieses Problem, da er die Anzahl der Partitionen stufenweise erhöht, indem die Eingabetupel zunächst in eine kleine Anzahl von Partitionen eingefügt werden. Anschließend wird der Algorithmus für jede dieser Partitionen erneut ausgeführt, wodurch sie in eine Anzahl kleinerer Partitionen aufgeteilt wird. Der Algorithmus kann auf den resultierenden Partitionen stets erneut ausgeführt werden, sodass die Anzahl mit jeder Stufe exponentiell ansteigt. So ist es möglich eine große Menge von ausreichend kleinen Partitionen zu erstellen, während der zusätzliche Aufwand für verteilte Speicherzugriffe vermieden wird.

Die blockweise Verarbeitung von Tupeln in Datenbanken hat stets zum Ziel, die zu Grunde liegende Rechnerarchitektur besser auszunutzen. Ein besonders hohes Potential verspricht diese Vorgehensweise daher auf Prozessoren deren Architektur auf die vektorbasierte Verarbeitung von Daten ausgelegt ist. In [HNZB07] wird deshalb eine Implementierung der MonetDB Anfrageverarbeitung auf dem *IBM Cell-Prozessor* vorgestellt. Dieser ist in der Lage einen Block von Tupeln vollständig parallel zu verarbeiten, weshalb die Aufteilung der Daten in Blöcke eine zwingende Voraussetzung auf derartigen Architekturen darstellt. Mit Hinblick auf künftige Prozessorgenerationen, wird dieses Vorgehen daher bei der Entwicklung von Datenbanksystemen eine bedeutende Rolle spielen.

Um die Ausführung von Programmen zu beschleunigen, muss dem Prozessor nicht nur ein schneller Zugriff auf Daten, sondern auch auf Instruktionen ermöglicht werden. Es ist daher ebenso wichtig verteilte Zugriffsmuster auf den Programmcode zu vermeiden, wie verteilte Zugriffe auf Daten zu vermeiden sind. In [ZR04] wurde eine Optimierung der Anfrageverarbeitung in *PostgreSQL* vorgestellt mit dem Ziel, durch das Puffern von Ergebnissen den zusätzlichen Aufwand durch Sprünge in der Ausführung, und damit verteilte Zugriffe auf Programmcode im Hauptspeicher, zu reduzieren.

Dazu wurde die Implementierung eines *Puffer-Operators* vorgeschlagen, der sich nahtlos in den bestehenden Operatorbaum integrieren lässt, da er sich transparent zwischen einen produzierenden, und einen konsumieren Operator einfügt. Die Realisierung des Puffers in Form einer Operators bietet zudem die Möglichkeit, diesen selektiv im Operatorbaum einzusetzen ohne die Verpflichtung einzugehen, nach jedem Operator einen Puffer zu verwenden.

Die Ausführung der Operatoren in PostgreSQL entspricht ebenfalls dem Volcano-Stil, sodass ein Operator bei jedem *next()*-Aufruf lediglich ein Tupel produziert, bevor er die Ausführung erneut an seinen aufrufenden Operator übergibt. Dadurch entstehen häufige Sprünge in der Ausführung, was zu verteilten Zugriffen auf Programminstruktionen im Hauptspeicher führt. Der Puffer-Operator reduziert die Anzahl dieser Sprünge, indem er bei seinem ersten next()-Aufruf eine große Anzahl an Tupeln von seinem Kind-Operator liest, und diese zwischenspeichert. Erhält er nun den nächsten next()-Aufruf von seinem Eltern-Operator, so genügt es den nächsten Wert aus dem Puffer zu lesen. Ein Aufruf des

Kind-Operators entfällt daher. Dieser minimal-invasive Ansatz kann die Anzahl der Ausführungssprünge deutlich reduzieren. Während ein konsumierender Operator nach wie vor für jedes Tupel sein Puffer-Operator aufrufen muss, kann dieser nun ohne weitere Aufrufe von Kind-Operatoren ein Tupel produzieren. Zudem stellt der Puffer einen leichtgewichtigen Operator dar, sodass durch dessen Aufruf nur eine geringe Anzahl von Instruktionen geladen werden muss, während ein komplexer Operator deutlich mehr Instruktionen enthält und damit auch diejenigen seines aufrufenden Operators aus dem Cache verdrängt.

Wie bereits in Kapitel 2.1.1 dargestellt wurde, finden Zugriffe auf den Hauptspeicher statt, indem die angeforderten Daten in den Prozessorcache geladen werden, wo sie schließlich der Verarbeitung zur Verfügung stehen. In [CMR09] wird daher die Rate von *Cache-Misses* (Zugriffe auf Hauptspeicherbereiche die sich noch nicht im Prozessorcache befinden) über die Laufzeit eines Operators untersucht. Dieses Cache-Miss-Verhalten stellt eine Funktion über die Laufzeit des Operators dar, die von dem spezifischen Zugriffsmuster abhängt. Operatoren welche Speicherbereiche mehrfach lesen, zeigen über ihre Laufzeit eine sinkende Cache-Miss-Rate, da Daten die bereits gelesen wurden sich nun im Cache befinden und keine weiteren Cache-Misses auslösen. Dieses Verhalten wurde im speziellen für den Hash-Verbund Operator untersucht, und konnte daher einen Beitrag zu dessen Implementierung in dieser Arbeit beitragen.

Als Ergebnis dieser Forschung kann festgehalten werden, dass die Dauer der ununterbrochenen Ausführung eines Operators eine große Auswirkung auf dessen Cache-Miss-Verhalten, und damit auf dessen Laufzeit hat. Um die Laufzeit der Operatoren in einer blockweisen Verarbeitung zu steuern, wird daher die geschickte Wahl der Blockgröße nahegelegt. Diese hat einen direkten Einfluss auf die Zeit die ein Operator ununterbrochen arbeiten kann, weshalb vorgeschlagen wird eine Blockgröße zu verwenden, die an den jeweils produzierenden, und konsumierenden Operator angepasst ist.

In [Aba08] wurde die Architektur der experimentellen, spaltenorientierten Datenbank *C-Store* vorgestellt. Konventionelle relationale Datenbanken verwenden eine zeilenweise Speicheranordnung, und Anfrageverarbeitung, welche auf die klassischen Anwendungsszenarien für Datenbanken ausgelegt ist, in welchen Anfragen das Finden und Vergleich von nur wenigen Zeilen umfassen. Für analytische Anwendungen sind hingegen Anfragen auf nur wenigen, aber dafür vollständigen Spalten notwendig. Vor diesem Hintergrund wurde C-Store für die schnelle Verarbeitung vollständiger Spalten entwickelt. Die physischen Anordnung der Daten auf der Festplatte ist dabei vergleichbar mit der von RDF-3X verwendeten Speicherstrategie (siehe Kapitel 2.4). Da es sich bei C-Store jedoch nicht um eine RDF- sondern eine relationale Datenbank handelt, ist die Anzahl der Spalten und möglichen Index-Permutationen wesentlich höher. Aus diesem Grund ist eine Beschränkung der Index-Permutationen nötig, welche aus dem Schema der relationalen Datenbank abgeleitet werden kann. Die blockweise Verarbeitung von Tupeln spielt eine untergeordnete Rolle in zeilenorientierten Datenbanken, da die Verarbeitungskosten einzelner Tupel aufgrund der nötigen Extraktionsschritte bereits sehr hoch sind. In spaltenorientierten Datenbanken hingegen entstehen nur geringe Verarbeitungskosten für einzelne Tupel, sodass die vektorbasierte Verarbeitung großer Tupelmengen in einem Operator eine wichtige Architekturkomponente darstellt.

Daher wird der next()-Aufruf in C-Store durch eine *nextBlock()*-Methode ersetzt. Anstatt jedoch einen tatsächlichen Block von Daten zwischen Operatoren auszutauschen, wird mit der nextBlock()-Methode lediglich ein Iterator zu einem Block übergeben. Der Grund dafür wird im Folgenden deutlich.

Die Ausführung der Operatoren in C-Store sieht sich eine besonderen Herausforderung gegenüber. Die zu Grunde liegenden Daten, und die Anfragestruktur entsprechen denen einer relationalen Datenbank, während die Architektur des System der einer spaltenorientierten Datenbank entspricht. Die Ausführung der Operatoren entspricht daher nicht dem klassischen Modell einer spaltenorientierten Datenbank, im speziellen ist dadurch die Baumstruktur des Operatorgraphen nicht mehr gewährleistet. Dies ist bedingt durch die Repräsentation der Daten mehrerer Spalten als virtuelle Zeile, aus der mehrere Attribute von unterschiedlichen Operatoren verarbeitet werden. Dies ermöglicht Ausführungspläne in denen ein Knoten nicht nur mehrere Kind-, sondern auch mehrere Elternoperatoren haben kann. Auf diesen Umstand musste bei der Implementierung der Operatorausführung besondere Rücksicht genommen werden, sodass die Produktion der Tupel im Graph stets synchron abläuft. Diese Bedingung muss erfüllt sein, da ein Operator mit mehreren Elternknoten seinen nextBlock()-Aufruf möglichst von allen synchron erhalten sollte, sodass er seine produzierten Tupel nicht zwischenspeichern muss für Elternoperatoren, die erst später als andere einen Block von Tupeln abfragen. Da zum Austausch der Daten zwischen Operatoren ein Iterator zum Einsatz kommt, ist es auch ausreichend für den Produzieren Operator eine einzelne physische Kopie der Tupel im Speicher zu halten, auf welche alle konsumierenden Operatoren zugreifen. Der produzierende Operator hat nun lediglich einen Block von Tupeln im Speicher, welchen er bei seinem nextBlock()-Aufruf verwirft, indem er die Werte mit neuen Tupeln überschreibt. Daher wird ein Tupel welches von den Blättern zur Wurzel produziert wird, In jedem Operator zwischengespeichert, sodass es eine Kaskade von Kopieroperationen durchläuft bevor es als Ergebnis an der Wurzel bereitsteht.

Die Ausführung von Hash-Verbund Operatoren auf modernen Rechnerarchitekturen wird in [GK07] untersucht. Ein besondere Schwerpunkt dieser Forschung war die Auswirkung der Materialisierungsstrategie auf die Cache-Lokalität, und damit das Laufzeitverhalten der Datenbank. Die Ergebnisse dieser Untersuchung konnten einen wichtigen Beitrag zu der Festlegung auf eine konkrete Implementierung der blockweisen Verarbeitung in dieser Arbeit liefern. Gegenstand der Forschung war eine Pipeline aus Hash-Verbund Operatoren, welche jeweils die Ergebnisse des vorgelagerten Operators konsumieren. In der dadurch entstehenden iterativen Ausführung der Operatoren bilden sich überlagernde Zugriffsmuster auf Speicherbereiche, da die produzierten Tupel eines Operators kurz darauf von einem weiteren konsumiert werden. Als Ergebnis kann festgehalten werden, dass voll-Materialisierte Tupel im Austausch zwischen unmittelbar aufeinanderfolgenden Operatoren, eine bessere Ausnutzung des Prozessorcaches erlauben als der Austausch von Zeigern auf Tupel. Dies liegt begründet in den Zugriffsmustern der Operatoren in einer Pipeline. Die produzierten Tupel eines Operators werden unmittelbar danach von dem konsumierenden Operator gelesen. Vollständig Materialisierte Tupel können daher direkt aus dem Cache geladen werden, während Zeigerlisten zusätzliche Speicherzugriffe erfordern.

Um die Ausführung von Datenbankoperationen zu beschleunigen, wurden in [ZCRS05] Möglichkeiten untersucht, die Ausführung der Operatoren in mehrere Threads aufzuteilen. In der Arbeit wurde die Optimierung von Daten-Cache-Lokalität angestrebt, und zudem die Möglichkeit untersucht, durch mehrere Threads auf einem Prozessor, die Auswirkungen von langsamen Speicherzugriffen zu minimieren. So wurde vorgeschlagen, die Zuordnung von Operatoren zu Threads so zu gestalten, dass während ein Thread auf Daten wartet, der andere Operationen ausführen kann deren Daten verfügbar sind. Dazu wird jeder Operator in zwei Threads aufgeteilt, von denen der erste (*helper Thread*) lediglich Daten in den Prozessorcache lädt, während der zweite (*computing Thread*) die tatsächliche Verarbeitung ausführt.

Der Computing-Thread erzeugt dafür eine Liste von Speicheradressen, auf welche er Zugriff benötigt, während der Helper-Thread diese Liste liest, und die Daten in den Prozessorcache lädt. Der Computing-Thread greift daher nur auf Daten zu, die er bereits von dem Helper-Thread angefordert hat. Um dieses Vorgehen zu realisieren, wird eine Kommunikations-Datenstruktur (*Work-ahead Set*) implementiert, auf die Helper- und Computing-Thread zugreifen. Diese Implementiert zwei Methoden, *post()* und *read()*. Der Computing-Thread schreibt mit post() die gewünschte Speicheradresse in das Work-ahead Set, während der Helper-Thread mit read() die Adressen liest, und anschließend in den Cache lädt. Das Work-ahead Set stellt ein zyklisches Array mit fester Größe dar, wodurch effiziente Zugriffe ermöglicht werden. Für beide,, post(), und read() Methoden, wird ein Iterator auf die zuletzt verarbeitete Postition in Work-ahead Set gehalten. Wenn die post()-Methode auf ein Feld stößt das bereits einen Wert enthält, bedeutet dies dass die betreffenden Daten vom Helper-Thread geladen wurden. Als Rückgabewert wird in diesem Fall die Speicheradresse der verfügbaren Daten übergeben. Der Computing-Thread kann diese Daten nun lesen, und verarbeiten. Die Größe des Work-ahead Sets muss dabei geschickt gewählt werden, denn eine zu kleine Liste lässt dem Helper-Thread nicht genug Zeit die Daten in den Cache zu laden bevor der Computing-Thread einen Zyklus vollzogen hat, und die Daten des korrespondierenden Feldes erwartet. Eine zu große Liste hingegen würde dazu führen dass die Menge an angeforderten Daten zu groß wird und andere Daten, oder die eigenen, aus dem Cache verdrängt.

In [bl001] wird die blockweise Verarbeitung von Daten in der relationalen Datenbank *DB2* untersucht. Der verwendete Ansatz soll dabei die Cache-Lokalität von Operatoren verbessern, indem die bestehende, tupelweise Verarbeitung durch die Verwendung von Blöcken ersetzt wird. In der vorgeschlagenen Implementierung werden die Operatoren derart modifiziert, dass sie einen Block von Tupeln, anstatt eines einzelnen produzieren, und konsumieren. Die DB2-Datenbank verfügt jedoch über zahlreiche leichtgewichtige Operatoren für mathematische, und andere schnell auszuführende Operationen. Diese können nur gering von Lokalität der Instruktionen im Cache profitieren, und würden den zusätzlichen Aufwand der Blockverwaltung, und des Kopierens von Tupeln nicht rechtfertigen. Um dieser Einschränkung gerecht zu werden, weist die Plangenerieung eine Gruppe solcher Operatoren, die im Plan hintereinander ausgeführt werden, einem Operatorblock zu. Der Operatorblock verhält sich dabei wie ein gewöhnlicher Operator, indem er einen Block von Werten am Stück konsumiert, und produziert. Innerhalb des Blocks arbeiten die Operatoren jedoch nach dem Tupel-orientierten Ansatz. Der erste Operator in einem solchen Block liest daher

ein Tupel nach dem anderen aus dem Eingabeblock, und gibt nach jedem Tupel die Ausführung an den nächsten Operator ab. Der Letze Operator schreibt seine Ergebnisse in den Ausgabeblock, sodass die Ausführung des Operatorblocks abgeschlossen wird, wenn den Ausgabeblock voll ist. Durch dieses Vorgehen wird ein geschickter Kompromiss gefunden zwischen der block- und tupelweisen Verarbeitung, welche je nach Typ des Operators einen Vorteil, oder Nachteil bietet.

4. Architektur

4.1. Verbesserungsansatz

Ziel dieser Arbeit ist die Verbesserung der Laufzeit des Operator-Baums durch die Ausnutzung von Cache-Lokalität und hohem Prozessor-Pipeline Durchsatz. Erreicht werden sollen diese Ziele mit der blockweisen Ausführung von Operatoren.

Damit ein Operator eine Menge von Tupeln am Stück produzieren kann, müssen seine Eingabetupel gepuffert sein. Dies soll mit *Block-Registern* realisiert werden. Ein solches Register kann eine definierte Menge von Tupeln aufnehmen, und dient als Puffer zwischen Ein- und Ausgabe zweier Operatoren.

Ein Operator kann nun so lange ununterbrochen Ergebnisse produzieren bis entweder sein Ausgabe-Block voll, oder einer seiner Eingabe-Blöcke leer ist.

4.2. Systemarchitektur

Um die geplanten Block-Register zu implementieren, waren Modifikationen an weiten Teilen der RDF-3X Anfrageverarbeitung erforderlich. Im Folgenden soll auf die wichtigsten Aspekte der neuen RDF-3X Architektur eingegangen werden, und ein Überblick des experimentellen Systems gegeben werden.

Abbildung 4.1 zeigt die neue parallele Architektur von RDF-3X, in der alternative Ausführungsmöglichkeiten über Aufrufparameter ausgewählt werden können. An zwei Stellen der Programmarchitektur wurden alternative Ausführungsmöglichkeiten hinzugefügt, die beliebig kombinierbar sind. Es entstehen daher vier mögliche Kombinationen. Zunächst kann das Format der Anfrage ausgewählt werden, deren jeweiliger Parser ein generisches Ausführungsplan-Objekt erzeugt.

Um die Implementierung der Operatoren zu unterstützen, und eine Möglichkeit zum selektiven Testen zu bieten, wird zusätzlich zu der SPARQL-Query Schnittstelle eine *XML-Ausführungsplan Schnittstelle* angeboten, auf die in Kapitel 7.1.1 genauer eingegangen wird.

Um sowohl die Ausführung mit block- als auch mit konventionellen Registern zu ermöglichen, soll eine zusätzliche Codegenerierung für blockbasierte Operatoren implementiert werden. Die Codegenerierung übersetzt den generischen Ausführungsplan in einen ausführbaren Operatorbaum, indem sie die nötigen Operator- und Register-Objekte anlegt und

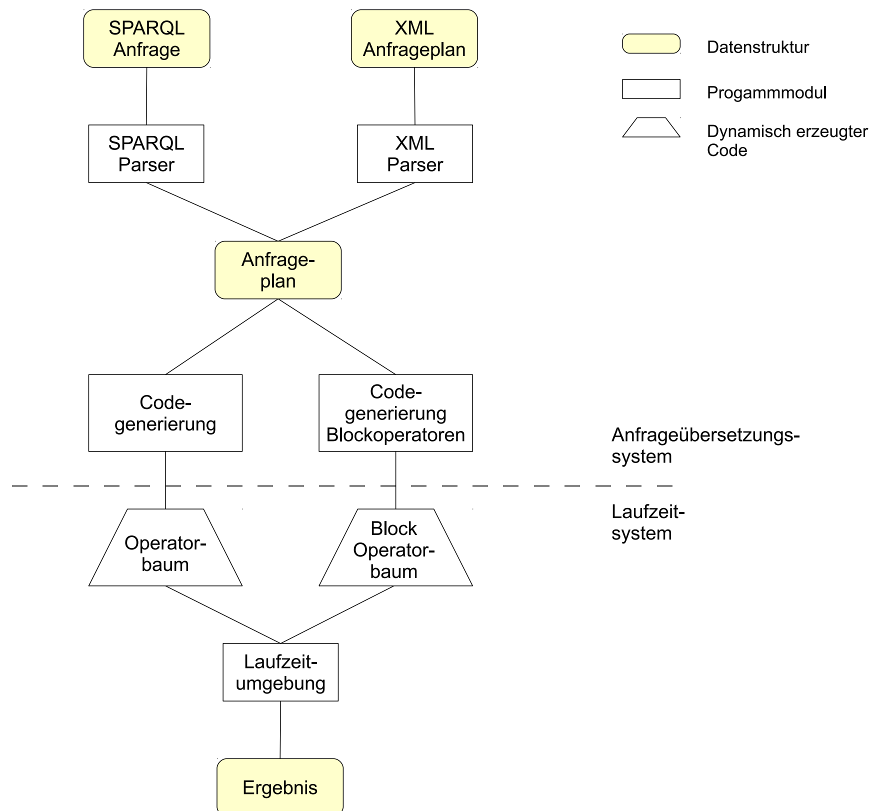

Abbildung 4.1.: RDF-3X Architektur mit neuen Komponenten

initialisiert. Der entstehende Block-Operatorbaum ist in der Aufrufsemantik kompatibel zu dem konventionellen Operatorbaum, sodass beide von einem generischen Scheduler ausgeführt werden können. In dieser Arbeit wird lediglich die Ausführung des Operatorbaums in einem Thread betrachtet.

4.3. Operatorbaum

In dem Block-Operatorbaum befindet sich ein Block-Register Knoten über jedem Operator. Dieses Register ersetzt die bisher direkte Verbindung der Operatoren, und soll einen effizienten Zugriff auf die gepufferten Werte bieten. Abbildung 4.2 zeigt den modifizierten Operator-Baum mit Block-Registern über jedem produzierenden Operator.

Das Zwischenspeichern der Ergebnisse jedes Operators bedeutet, dass auf jeder Ebene des Operator-Baums eine physische Kopie der Tupel angelegt wird. Dieser zusätzliche Speicheraufwand ist ein potentieller Nachteil des Lösungsansatzes, weshalb bei der Implementierung ein besonderer Schwerpunkt auf effiziente Lese- und Schreiboperationen in den Blöcken zu legen ist.

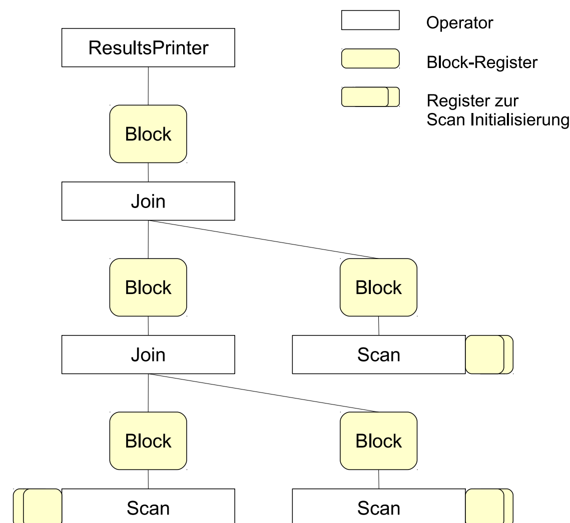

Abbildung 4.2.: Operator-Baum mit Block-Registern

Die Schnittstelle zwischen Operatoren und Blöcken soll transparent gestaltet werden, sodass sich bestehende RDF-3X Operatoren leicht in die neue Architektur integrieren lassen. Dem Open-Closed Prinzip entsprechend soll daher der Zugriff auf die Blöcke in den Registern implementiert sein. Um zudem spezifisches Wissen über die Zugriffsmuster und Arbeitsweisen einzelner Operatoren auszunutzen, sollen unterschiedliche Typen von Registern zum Einsatz kommen, die an den jeweils schreibenden Operator angepasst sind.

Operatoren finden nicht zu jedem Tupel einen Partner, und können nur im $N \times M$ *Join-Fall* (siehe Kapitel 6.3.2) tatsächlich mehr Ergebnisse produzieren als gelesen wurden. Daher wird angenommen, dass Leseoperationen auf den Registern in längeren, ununterbrochenen Sequenzen vorkommen als Schreiboperationen. Die Register sollen daher so angelegt werden, dass der unvermeidbare Aufwand der Blockverwaltung beim Schreiben, und nicht beim Lesen entsteht.

Leichtgewichtige Operatoren die zuvor eigenständig ausgeführt werden mussten, lassen sich nun mit der bestehenden Block-Logik vereinen. So können unnötige Ausführungssprünge vermieden werden und die Cache-Lokalität des Codes verbessert sich.

4.4. Blöcke und Spalten

Während in der konventionellen RDF-3X Architektur jedes der Register genau eine Komponente eines Tupels aufnimmt, bedingt der Einsatz von Blöcken als Puffer zwischen Operatoren die Zuordnung von Komponenten zu Spalten, welche zusammen einen Block bilden. Die Anzahl der Spalten eines Blocks wird also bestimmt von der Breite der zu speichernden Tupel. Die Blöcke der Scan-Operatoren können daher bis zu drei Spalten umfassen. Die Breite höher gelegener Blöcke kann jedoch beliebig groß sein, und ergibt sich aus der Anzahl der Spalten der darin vereinten Blöcke. Ein Block muss sämtliche Komponenten der

Ergebnistupels seines Operators aufnehmen, wobei nicht gegeben ist dass dies genau der Summe der Komponenten seiner Eingabetupel entspricht. Die genaue Blockbreite, und zu verwendende Spalten, werden in Kapitel 6.2 besprochen.

Da die Logik zur Verwaltung von Blöcken in den Registern enthalten sein soll, ist das Einfügen von Zeilen in den Block ebenfalls an dieser Stelle zu Implementieren. Ein Operator soll daher lediglich entscheiden welche Tupel akzeptiert werden, und anschließend seinem Ausgabe-Blockregister lediglich die Position dieser Tupel in seinen Eingabeblöcken mitteilen. Das Blockregister soll anschließend diese Tupel direkt aus den Quellblöcken lesen, und in seinen Block einfügen.

5. Sideways Information Passing

In einem Graph von kooperierenden Ressourcen, wie ihn der Operatorbaum darstellt, entsprechen die Kanten Kommunikationsverbindungen zwischen Ressourcen. In der Ausführung eines solchen Graphen können zwei Ressourcen daher nur direkt kommunizieren, wenn sie über eine Kante verbunden sind. Die Kanten des Operatorbaums sind zudem gerichtet, sodass Informationen nur an höher liegende Operatoren gesendet werden.

Der Sideways Information Passing Ansatz erweitert den Graph um zusätzliche SIP-Kanten, über welche eine direkte Kommunikation zwischen Operatoren möglich ist, die sonst nicht in direkter Verbindung stehen [IT08].

In der Ausführung des RDF-3X Operatorbaums wird SIP daher eingesetzt, um Informationen über verarbeitete Tupel zwischen Operatoren auszutauschen. So ist es möglich, Tupel die keinen Partner finden können, und daher nicht zum Ergebnis beitragen, frühzeitig zu verwerfen. Dies stellt einen besonders interessanten Ansatz dar, da SPARQL-Anfragen typischerweise viele selektive Verbundoperationen enthalten.

Im Folgenden sollen die SIP-Strategien der konventionellen RDF-3X Architektur vorgestellt, sowie neue Möglichkeiten für SIP in der Blockverarbeitung aufgezeigt werden.

5.1. SIP in RDF-3X

In RDF-3X kommen zwei unabhängige SIP-Strategien zum Einsatz, auf die im Folgenden kurz eingegangen werden soll.

Merge-Hints stellen eine leichtgewichtige Möglichkeit für Scan-Operatoren dar, Informationen über ihre aktuellen Tupel auszutauschen. Alle Register die sich in den Unterbäumen eines Merge-Verbunds befinden, und Werte für dieselbe Variable produzieren, werden einer *Äquivalenzklasse* zugeordnet. Zwei wichtige Eigenschaften gelten für die Register einer Äquivalenzklasse.

Die Werte dieser Register nehmen an einem gemeinsamen Merge-Verbund teil, weshalb jeder Wert einen äquivalenten Join-Partner in jedem anderen Register der Klasse benötigt. Des Weiteren ist durch den Merge-Verbund gegeben, dass die Tupel in sortierter Reihenfolge vorkommen.

Diese Eigenschaften können nun von den Scan-Operatoren ausgenutzt werden. Sobald ein Scan einen neuen Wert aus dem Index geladen hat, kan es diesen mit den aktuellen Werten der anderen Register aus seiner Äquivalenzklasse vergleichen. Hat ein anderer Scan bereits

einen größeren Wert produziert, kann der kleinere Wert verworfen werden da er keinen Join Partner mehr finden wird. Das Register in welchem der größere Wert steht, wird aufgrund der sortierten Indexe keine kleineren Werte mehr produzieren.

Um den Zusatzaufwand durch SIP gering zu halten, wird in RDF-3X ein etwas anderer Ansatz verfolgt. Anstatt jeden gelesenen Wert mit den anderen Registern zu vergleichen, findet der Vergleich nur beim Laden einer neuen Speicherseite statt [NW09b]. In der verwendeten Implementierung umfasst eine Speicherseite typischerweise etwa *5000* Einträge.

Die **Domänenbeschreibung** implementiert eine weitere SIP-Strategie, welche auf die Verarbeitung von Tupeln in Hash-Verbund Operatoren ausgelegt ist. Erneut werden Register in *Äquivalenzklassen* aufgeteilt, um den Registern einer Klasse eine gemeinsame Domänenbeschreibung zuzuweisen. Eine Äquivalenzklasse umfasst dabei sämtliche Register, welche derselben Variable zugeordnet sind. Im Gegensatz zu den Äquivalenzklassen der Merge-Hints, ist hier die Teilnahme an einem Merge-Verbund keine Bedingung. Es ist daher nicht gegeben, dass die Werte sortiert vorkommen, oder synchron gelesen werden. Ein einfacher Vergleich der aktuellen Werte wie ihn die Merge-Hints vornehmen, ist deshalb nicht ausreichend.

Die Domänenbeschreibung findet in Hash-Verbund Operatoren Verwendung, deren Verarbeitung in zwei Phasen abläuft. Zunächst wird in der Aufbauphase eine Hashtabelle aus einer der Eingaberelationen aufgebaut, um in der zweiten, Produktionsphase Partner für die Tupel der anderen Relationen in der Hashtabelle zu finden. Während der Aufbauphase erzeugt der Hash-Verbund Operator eine Liste aller Tupel die in der Eingaberelation vorgekommen. Da der Operator sämtliche Tupel der Relation in die Hashtabelle einträgt, ist diese Liste vollständig. Die Liste ist Teil der Domänenbeschreibung, und steht daher den anderen Operatoren in der Äquivalenzklasse zur Verfügung. Höher liegende Hash-Verbund Operatoren in anderen Unterbäumen, suchen nun jedes Tupel in dieser Liste, und können damit feststellen ob das Tupel einen Partner finden kann. Dasselbe gilt für Scan-Operatoren in anderen Unterbäumen, welche ebenfalls ihre gelesenen Werte mit der Liste vergleichen.

In der tatsächlichen Implementierung wird jedoch keine vollständige Liste von Tupeln erzeugt, da dies der Ausführung einen signifikanten Speicheraufwand hinzufügen würde. Stattdessen werden die gelesenen Werte in einen *Bloomfilter* eingetragen. Dieser kommt mit einer kleinen Datenstruktur aus, zu dem Preis einer gewissen Falschpositiv-Rate. Dadurch werden Tupel akzeptiert, die keinen Partner mehr finden können. Diese werden jedoch von der regulären Verbundoperation verworfen. Falschnegativ Ergebnisse sind mit einem Bloomfilter nicht möglich, weshalb sichergestellt ist, dass jedes Tupel das einen Partner finden kann, auch akzeptiert wird.

5.2. SIP in der blockweisen Verarbeitung

Die Block-Implementierung verändert die Semantik des Operatorbaums, und bedingt daher eine Neubewertung der SIP Strategie.

Einen wesentlichen Unterschied stellt die Pipeline-Architektur der Block-Implementierung dar. Anstatt eine Zeile von Registern auf der untersten Ebene zu benutzen, auf welche alle Operatoren abwechselnd zugreifen, werden Zwischenergebnisse nun auch in Block-Registern der höheren Ebenen gespeichert.

Verwerfen vollständiger Blöcke

Aufgrund der Puffer-Blöcke nach jedem Operator ist es nicht mehr möglich, ungültige Tupel durch einen Vergleich der zuletzt von Scans gelesenen Werte zu identifizieren. Dies wird bei Betrachtung der Pipeline deutlich. Ein Tupel aus einem Block kann seinen Join-Partner nicht nur in Blöcken derselben Ebene, sondern in Blöcken aus allen Ebenen finden. Der Ansatz eines rein horizontalen Vergleichs soll daher durch einen vertikal orientierten Vergleich ersetzt werden. Des Weiteren soll analog zu dem in [NW09b] vorgestellten Vorgehen, nicht für jedes Tripel ein Vergleich durchgeführt werden. Es ist daher naheliegend den Block als ausreichend große Einheit zum Gegenstand des Vergleichs zu machen. Die hier vorgeschlagene SIP-Strategie soll daher einen vollständigen Block als ungültig identifizieren können, wodurch mit geringem zusätzlichem Aufwand eine große Menge von Tupeln verworfen werden kann.

Der Vergleich soll durchgeführt werden zwischen einem *Quell-* und einem *Zielregister*. Er ist damit asymmetrisch, wobei nur das Quellregister seinen Block verwerfen kann nicht aber das Zielregister.

Ein solcher Vergleich kann angestellt werden zwischen Block-Registern, die folgenden Bedingungen erfüllen:

- Das Zielregister befindet sich in dem anderen Unterbaum eines gemeinsamen Elternknotens. (Für Knoten H in Abb. 5.1 kämen damit $\{B, D, E, F, I\}$ in Frage.)
- Das Zielregister ist das höchste in seinem Unterbaum. (Damit reduziert sich die Liste um $\{D, E\}$ auf $\{B, F, I\}$)

Die Bestimmung der SIP-Partner soll während der initialen Rekursion erfolgen, welche den Operatorbaum aufbaut. Ziel ist es, für jeden Operator eine Liste von SIP-Partnern zu finden. Daher wird folgender Algorithmus vorgeschlagen, der sich nahtlos in die bestehende Rekursion integrieren lässt, und mit geringem zusätzlichem Aufwand Partner-Listen für jeden Operator erzeugt.

- **Initialisierung:** Beginne mit der leeren Liste `{}` in der Wurzel `A`.
- **Rekursionsschritt:** Betrete die Rekursion für Kindknoten `B`, mit der um Kindknoten `C` erweiterten Liste. Betrete analog die Rekursion für Kindknoten `C`, mit der um Kindknoten `B` erweiterten Liste.
- **Abbruchbedingung:** Beende die Rekursion wenn keine weiteren Kindknoten mehr vorhanden sind. Die Semantik der Operatoren stellt dabei sicher, dass ein Elternknoten entweder zwei, oder keine Kindknoten hat.

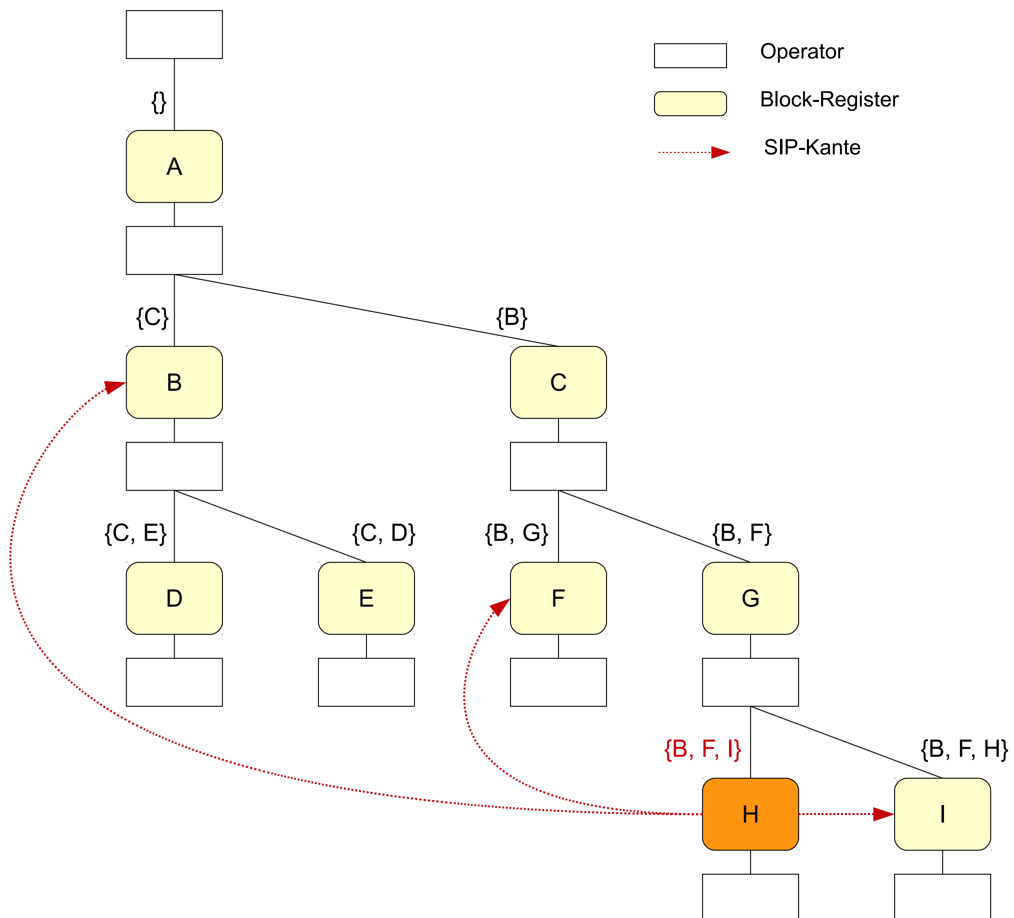

Abbildung 5.1.: Block-Operator-Baum mit SIP-Kanten und Partner-Listen

Algorithmus 5.1 Rekursives Bestimmen von SIP-Partnern

proc findPartners($node, partners$) $\equiv$
  $node.partners \leftarrow partners;$
  if $(node.left \wedge node.right)$
    then findPartners($node.left, (partners \cup node.right)$);
      findPartners($node.right, (partners \cup node.left)$);
  fi
.

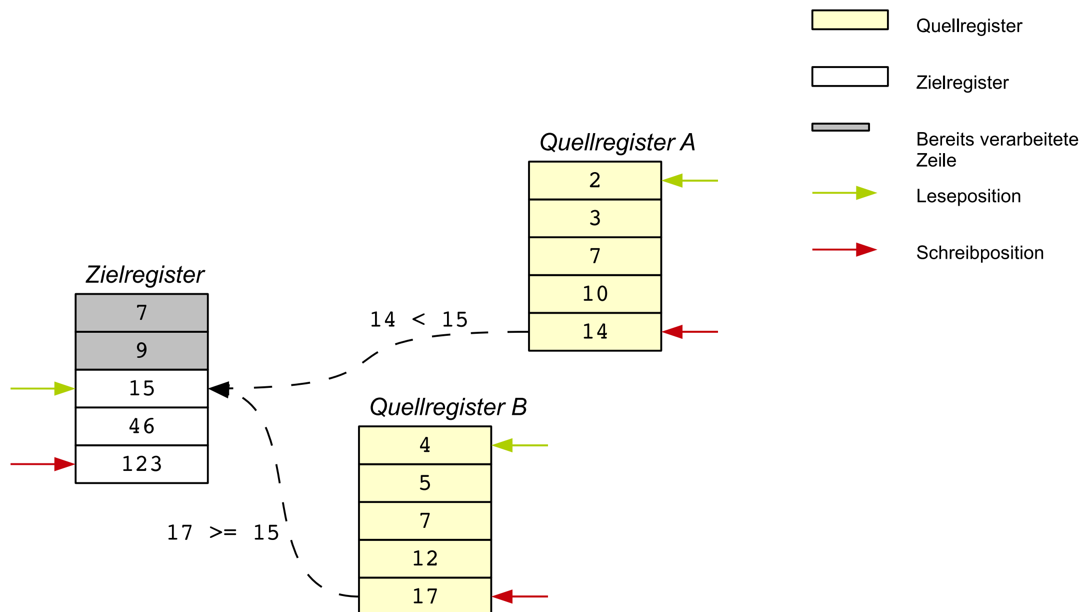

Abbildung 5.2.: Detailansicht SIP Vergleich von Blöcken

In Abbildung 5.1 ist ein beispielhafter Operatorbaum gegeben, dessen Register und Operatoren zur besseren Lesbarkeit vereint wurden. An Knoten H sind die SIP-Kanten erkennbar, sowie der Inhalt der Partner-Liste an jedem Knoten.

Ziel dieser SIP-Strategie ist es, einen ungültigen Block mit möglichst geringem zusätzlichem Aufwand zu verwerfen. Erreicht wird dies durch den Vergleich von nur einem Wert aus Quell- und Zielregister, wie in Abbildung 5.2 deutlich wird.

Die Register müssen in der Lage sein, ihren größten und kleinsten Wert schnell zu identifizieren, damit der Vergleich effizient abgewickelt werden kann. Da die hier vorgestellte SIP-Strategie nur auf die Unterbäume von Merge-Joins angewendet wird, ist die sortierte Reihenfolge der Werte gegeben. Die benötigten Werte befinden sich daher stets am Anfang und am Ende eines Blocks. Zeiger zu der aktuellen Schreib- und Leseposition hält jedes Block-Register ohnehin für die gewöhnlichen Block-Operationen. Diese Zeiger können nun zur Identifikation der Vergleichswerte dienen.

Partitionierter Hash Join

In der Block-Implementierung wurde der Hash-Verbund Algorithmus durch einen partitionierten Hash-Verbund (siehe Kapitel 6.3.1 und [BMK99]) ersetzt, der neue Möglichkeiten für SIP eröffnet.

Während der Aufbauphase der Hash-Partitionen für die rechte Relation, sind die Listen-Partitionen der linken Relation bereits vollständig verfügbar. Es lässt sich daher leicht feststellen, ob die Partner-Partition auf der linken Seite Einträge aufweist, oder nicht. In letzte-

rem Fall kann der Aufbau dieser Hash-Partition übersprungen werden, da kein potentieller Eintrag daraus Join-Partner in der leeren Listen-Partition finden kann.

Abgesehen von dieser neuen Möglichkeit, kann in der blockweisen Verarbeitung weiterhin die Domänenbeschreibung mit Bloomfilter aus 5.1 zum Einsatz kommen. Die Semantik des Hash-verbund Operators hat sich in dieser Hinsicht nicht verändert, da bereits in der tupelweisen Verarbeitung die Aufbauphase sämtliche Tupel gelesen hat, und daher die verwendete SIP-Strategie an dieser Stelle nicht von einer tupelweisen Verarbeitung abhängt.

6. Implementierung

6.1. Codegenerierung

Die Codegenerierung hat die Aufgabe das vollständige Laufzeitsystem bestehend aus Operatorbaum, und Blockregistern zu erzeugen. Diese Erzeugung geschieht in einer Rekursion über den Ausführungplan, dessen Knoten sämtliche Informationen zur Initialisierung der Operatoren enthalten. Die Rekursion entspricht dabei einer Tiefensuche, in welcher daher die Scan-Operatoren der jeweiligen Unterbäume als erstes aufgebaut werden. Dadurch wird es möglich, während dem Aufbau der Operatoren, Informationen über die Anzahl, und Position der Blockspalten zu sammeln, welche jeweils für die Erzeugung der Blöcke in höheren Ebenen benötigt werden. Während der Implementierung wurde Wert darauf gelegt, möglichst viele Informationen über das Laufzeitverhalten der Blockregister bereits während der Codegenerierung zu bestimmen. Die Blöcke können daher bereits mit den nötigen Informationen über Spaltenanzahl und -position Initialisiert werden, wodurch zur Laufzeit statischer Code ohne Sprünge ausgeführt werden kann. Um dies zu erreichen sind mehrere Implementierungen der Blockregister vorhanden, deren Verhalten über Template- und Konstruktorparameter angepasst wird. Die Codegenerierung bestimmt den Typ von Register anhand der Art von lesendem, und schreibendem Operator, sowie weiterer Parameter.

6.2. Block-Register

RDF verwendet zwar nur drei-wertige Tupel zur Repräsentation, das Ergebnis einer Query kann jedoch ein beliebig großes Tupel sein. Dies wird deutlich wenn man sich die Natur eine SPARQL-Query vor Augen hält. Es kann eine beliebige Anzahl von Variablen Projiziert werden. So ist es möglich nur ein- oder zweistellige Ergebnisse zu produzieren oder durch die Verknüpfung mehrere Tripel auch höherstellige Tupel zu bilden.

Diese Anforderung wurde von der konventionellen RDF-3X Implementierung implizit erfüllt, indem für jede Variable ein Register zu Verfügung steht. Um ein Ergebnis zu produzieren, müssen nur die Register der projizierten Variablen in der richtigen Reihenfolge ausgegeben werden.

Block-Register stehen ebenso für jede Variable zur Verfügung, haben jedoch eine andere Semantik als die konventionellen Register. Aufgrund der Puffer-Funktion der Block-Register, muss von den produzierten Ergebnissen jedes Operators eine physische Kopie in dessen Ausgabe-Block angelegt werden. Diese Kopie beinhaltet sämtliche Komponenten der Join-Tupel, wobei die Join-Komponente im Ergebnis nur einmal vorkommt. Es ist daher gegeben

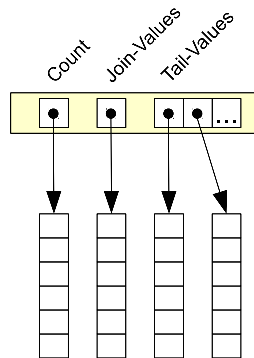

Abbildung 6.1.: Block-Spalten im Detail

dass der Ausgabeblock höchsten eine Spalte weniger als die Summe der Spalten seiner Eingabeblöcke enthält.

Ein Block enthält daher stets eine Spalte welche die Join-Komponente enthält (*Join-Value Spalte*), sowie eine beliebige Anzahl weiterer Spalten (*Tail-Value Spalten*). Zusätzlich enthält der Block noch eine *Count-Spalte*, deren numerischer Wert die Anzahl der Duplikate eines Tupels repräsentiert. Dieser wird benötigt um bei der Ausgabe Duplikate eines Tupels zu expandieren, worauf in dieser Arbeit jedoch nicht genauer eingegangen werden soll.

Jede dieser Spalten hat dieselbe Länge, weshalb in der Implementierung ein Speicherpool zum Einsatz kommt, welcher die Spalten in einem zusammenhängenden Stück des Hauptspeichers allokiert. Ein Block wird daher durch eine Datenstruktur repräsentiert (siehe Listings), welche Zeiger auf sämtliche Spalten dieses Blocks enthält, wie in Abbildung 6.1 deutlich wird (Code-Listing siehe Anhang).

Die Join-Value-Spalte enthält stets die Komponente des Tupels, auf welcher der lesende Operator den Vergleich durchführt. Dieses Vorgehen wurde gewählt, damit die verwendete Spalte in dem lesenden Operator fest zugewiesen werden kann, wodurch eine dynamische Bestimmung mit potentieller Auswirkung auf die Laufzeit vermieden wird. Diese Spalte muss aber nicht dieselbe sein, auf welcher der schreibende Operator den Join durchgeführt hat, weshalb das Blockregister beim Schreiben einer Zeile die Positionen der Spalten verändern kann. Ein solcher Fall wird bereits bei der Codegenerierung festgestellt, sodass ein Register mit entsprechendem Verhalten angelegt werden kann.

Um dieser, und den Anforderungen aus Kapitel 4.3 gerecht zu werden, sind drei grundlegende Typen von Registern Implementiert worden, deren spezielles Verhalten sich über *Mixins* erweitern lässt.

- Ein Register zur Initialisierung von Scan-Operatoren, welches nur einen einzelnen, konstanten Wert aufnimmt. Dieser Wert wird von der Codegenerierung aus dem Anfrageplan gelesen, und Form eines solchen Registers dem Scan-Operator übergeben. Diese Werte bestimmt dann die Konstanten des zu lesenden Tripel-Musters. Daher

wird jeder Scan mit keinem, einem, oder zwei solcher Register initialisiert. Die Different zu der gesamten Tripelbreite entspricht dann der Anzahl der Variablen, und damit der Spalten der Ausgaberegister.

- Für diese Scan-Ausgaberegister existiert ein weiterer Typ, welcher als schreibendes Registers für Scans, und lesendes Register für einen Verbund- oder Ausgabeoperator dient. Das Verhalten des Registers lässt sich mit drei Mixins um eine Filterfunktion für jede potentielle Variable erweitern. Diese akzeptieren eine Filterbedingung in Form von ein- oder auszuschließenden Dictionary-IDs, und wendet sie auf die entsprechende Komponente des Tupels an. Ist die Filterbedingung nicht erfüllt, so wird das gesamte Tupel verworfen.
- Als Ausgaberegister für Verbundoperatoren kommt ein weiterer Registertyp zum Einsatz, welches als schreibendes Register für einen Verbund-, und lesendes Register für einen Verbund- oder Ausgabeoperator dient.

  Sein Verhalten lässt sich über zwei Mixins erweitern, wovon das erste die bereits erwähnte Vertauschung der Spaltenpositionen ermöglicht. Das zweite wird verwendet um eine zusätzliche Verbund-Bedingung, oder Selektion auszuführen. Es kommt zum Einsatz wenn das Tripel-Muster der Anfrage mehr als eine identische Variable in mehreren Tripeln enthält. In diesem Fall wird von dem Mixin ein zusätzlicher Vergleich auf einem, oder mehreren Paaren von Tail-Values durchgeführt (Code-Listing siehe Anhang).

6.3. Operatoren

In der blockweisen Verarbeitung finden zwei der Operatoren von RDF-3X Verwendung in Form von eigenständigen Operatoren. Zwei weitere konnten, wie bereits erwähnt, direkt in die Blockregister integriert werden. Abgesehen von einem Sortieroperator unterstützt die vorgestellte Implementierung damit den vollständigen Funktionsumfang von RDF-3X, wodurch aussagekräftige Leistungsmessungen möglich sind.

Die Umstellung auf eine blockweise Verarbeitung von Tupeln könnte mit einer Schnittstelle zu Blockoperatoren realisiert werden, die den Puffer-Block vollständig verbirgt, und lediglich eine `nextTuple()`-Methode präsentiert. Operatoren könnten dadurch unverändert in die Implementierung mit Puffer-Blöcken übernommen werden.

Um jedoch die Vorteile der Blöcke vollständig auszunutzen, und unnötige Funktionsaufrufe für jedes Tupel zu vermeiden, wurde keine solche Schnittstelle verwendet. Stattdessen besteht die Schnittstelle zwischen Blöcken und lesendem Operator aus einer `nextBlock()`-Methode, die lediglich einen Zeiger auf den zu lesenden Block übergibt. Der Operator kann nun selbstständig über die Tupel des Blocks iterieren, und Werte direkt lesen.

Daher ist eine Modifikation der bestehenden Operatoren unumgänglich. Im Folgenden werden die Änderungen an den Operatoren, sowie deren grundlegendes Verhalten beschrieben.

6.3.1. Partitionierter Hash-Verbund

Der Hash-Verbund Operator wurde nicht in die neue Architektur übernommen, sondern auf Basis von [BMK99] vollständig neu Implementiert. Der Partitionierte Hash-Verbund verspricht eine deutlich beschleunigte Verbundphase, zum Preis einer aufwändigeren Aufbauphase.

Dies wird erreicht durch die Aufteilung der Hashtabelle in eine Vielzahl kleiner Partitionen. Jede dieser Partitionen enthält eine kleine Hashtabelle, die vollständig im Prozessorcache verarbeitet werden kann, wodurch teure Zugriffe auf den Hauptspeicher vermieden werden.

In der vorgestellten Implementierung wird zu jedem Tupel ein *32-Bit Hashwert* basierend auf dem jeweiligen Join-Wert berechnet. Je nach Konfiguration wird eine bestimmte Anzahl der niederwertigsten Bits dieses Hashwertes herangezogen, um die Partition zu bestimmen in welche der Wert einzufügen ist. Die verbleibenden Bits des Hashwertes werden zur Identifikation des *Buckets* in der Hashtabelle herangezogen. Diese Hashtabellen sind dabei mit dreifachem *Cuckoo-Hashing* [Ask09] implementiert, welches einen Worst-Case Suchaufwand von nur drei Zugriffen garantiert.

Die Partitionierung erfolgt auf beiden Eingaberelationen des Hash-Verbundes, wobei jedoch nur aus einer Relation tatsächliche Hashtabellen aufgebaut werden. Auf der anderen Relation kann das Einfügen in Hashtabellen entfallen, da diese Tupel lediglich iterativ gelesen, und in der entsprechenden Partner-Hashpartition gesucht werden. Abbildung 6.2 verdeutlicht diesen Sachverhalt. Auf der linken Seite sind die Hashtabellen zu erkennen, während die Partitionen der rechten Seite lediglich eine Liste aller Tupel enthalten. Die Partitionierung wurde auf zwei Bit durchgeführt, was eine Anzahl von vier Partitionen zur Folge hat.

Nachdem die Aufbauphase der Partitionen für beide Eingaberelationen abgeschlossen ist, kann die Produktion der Join-Ergebnisse beginnen. Dazu wird über die Listenpartitionen iteriert, und für jedes Tupel ein Join-Partner in der gegenüberliegenden Partition gesucht. Durch dieses Vorgehen wird der Zugriff zu jedem Zeitpunkt der Produktion auf lediglich zwei Partitionen beschränkt.

6.3.2. Misch-Verbund

Der Misch-Verbund Operator (*Merge Join*) wurde mit leichten Modifikationen aus der ursprünglichen Implementierung übernommen. Sein Algorithmus ist in Form einer Zustandsautomatmaschine Implementiert, um das Unterbrechen, und Fortsetzen der Ausführung nach jedem Tupel zu erlauben. Diese Anforderung besteht in abgeschwächter Form weiterhin in der blockweisen Verarbeitung, da die Ausführung nach der Produktion eines vollständigen Blocks ebenfalls unterbrochen werden muss.

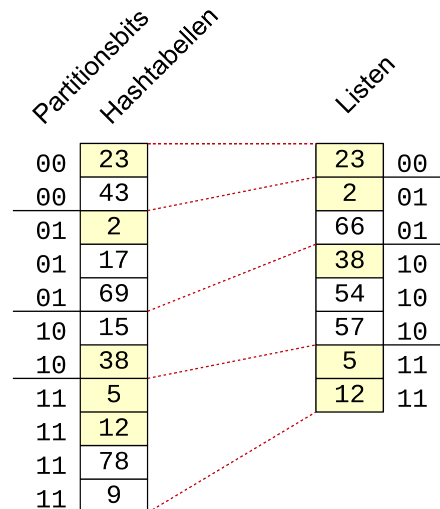

Abbildung 6.2.: Partitionierter Hash-Verbund

Um einen effizienten Zugriff auf die Puffer-Werte zu ermöglichen, wird nun der Zustandsautomat in einer Schleife ausgeführt, bis der Ausgabeblock keine weiteren Tupel mehr aufnehmen kann. Die benötigten Werte werden dabei über den oben erwähnten Blockiterator direkt gelesen.

Bei der Produktion von Ergebnissen im Merge Join sind drei wesentliche Fälle zu unterscheiden, die vom Hash-Verbund Algorithmus implizit abgedeckt, im Merge Join jedoch explizit zu modellieren sind. Diese Fälle ergeben sich aus den Tail-Values der Tupel. Jedes Tupel ist aufgrund der Semantik des Operatorbaums zwar eindeutig, kann jedoch mit identischem Join-Value mehrfach vorkommen, wenn sie die Tail-Values unterscheiden. Aufgrund der Sortiertheit der Tupel, ist jedoch gewährleistet dass eine solche Serie von Tupeln mit identischem Join-Value stets ununterbrochen aufeinanderfolgend in der Eingaberelation vorkommt.

- 1×1 Fall: Die gelesenen Tupel haben keine Duplikate mit unterschiedlichen Tail-Values, daher kann das Ergebnis unmittelbar produziert werden.
- $1 \times N$ Fall: in einer Relation kommt ein Tupel mit diesem Join-Value nur einmal vor, während in der anderen Relation N Tupel mit demselben Join-Value, aber unterschiedlichen Tail-Values stehen. Das erste Tupel wird daher festgehalten, und für jedes Partnertupel in der anderen Relation ein Ergebnis produziert, welches jeweils die Kombination aus beiden Tail-Value Zeilen enthält.
- $N \times M$ Fall: in beiden Relationen kommen mehrere Tupel mit übereinstimmendem Join-Value vor, es wird daher für jedes der M Tupel ein $1 \times N$ Join durchgeführt, um das Kreuzprodukt beider Tupel-Serien zu bilden.

Um die Abwicklung dieser Fälle zu verdeutlichen, und einen Einblick in die Arbeitsweise des Misch-Verbund Operators zu geben, werden im Folgenden kurz die wichtigsten Zustände des Zustandsautomaten aufgezeigt.

Da in der konventionellen, tupelweisen Verarbeitung mit jedem next() Aufruf des Kindoperators das zuvor gelesene Tupel aus den Registern verdrängt wird, ist es notwendig das aktuelle Tupel zu kopieren, sodass das nächste betrachtet werden kann um festzustellen um welchen der drei erwähnten Fälle es sich bei diesem Join handelt. Im $1 \times N$ und $N \times M$ Fall müssen sogar alle N Tupel kopiert werden bevor ein Ergebnis produziert werden kann.

Dieses Verhalten wurde in der Blockimplementierung beibehalten, da auch die Tupel in den Blöcken überschrieben werden sobald der nächste Block angefordert wird. Somit ist es nicht nötig die Blockposition zu berücksichtigen, und beim Übergang zu einem neuen Block Tupel erneut zu lesen, und kopieren.

Der Misch-Verbund hält daher zwei zusätzliche Datenstrukturen für Tupel um diese Fälle abzudecken. Ein *Schattenpuffer* der jeweils ein Tupel aus der rechten, und linken Relation aufnimmt, und dazu dient das aktuelle Tupel zu speichern während das nächste betrachtet wird. Und einen *Puffer* welcher eine beliebig große Menge an Tupeln aufnehmen kann, um den $1 \times N$ und $N \times M$ Fall abzudecken.

- **scanHasBoth:** Bildet den Anfangszustand indem die aktuellen Tupel beider Seiten in den Schattenpuffer kopiert werden, um die jeweils nächsten Tupel zu betrachten und den passenden Join-Fall zu bestimmen.
- **empty:** Eine der Eingaberelationen ist leer, daher kann die Produktion von Ergebnissen beendet werden, da keine weiteren Partner zur Verfügung stehen.
- **scanStepLeft:** Der Join-Value des linken Tupels ist kleiner als der des rechten, daher wird in der linken Relation das nächste, größere Tupel geladen.
- **scanStepRight:** Implementiert den analogen Fall für die rechte Relation.
- **scanStepBoth:** Wird betreten nachdem die Verarbeitung eines 1×1, $1 \times N$ oder $N \times M$ Falls abgeschlossen ist, um auf beiden Relationen das nächste Tupel zu erhalten.
- **1to1:** Der aktuelle Join-Value beider Seiten stimmt überein, der jeweils nächste Wert auf beiden Relationen unterscheidet sich aber von dem aktuellen. Es liegt also en 1×1 Fall vor und das Ergebnis kann unmittelbar produziert werden.
- **1toN:** Die zuletzt gelesenen Join-Values auf beiden Relationen stimmen überein, auf einer Relation ist jedoch auch der aktuelle Wert ein Join-Partner. Das eine Tupel wird daher im Schattenpuffer festgehalten, während auf der anderen Relation N Tupel gelesen werden.
- **NtoM:** Beide Relationen enthalten eine Serie von Tupeln mit übereinstimmendem Join-Value, daher werden N Tupel einer Relation in den Puffer geladen, um anschließend auf jedem davon einen $1 \times N$ Join auszuführen.

6.3.3. Probleme und Hürden

Eine besondere Herausforderung bei der Implementierung der blockweisen Verarbeitung, stellte die Programmarchitektur von RDF-3X dar. Sie ist bereits hoch optimiert, und stark an die tupelweise Verarbeitung zugeschnitten. Es waren daher weitreichende Modifikationen an der Codebasis nötig, die zu langen Entwicklungs- und Debuggingzyklen geführt haben.

Als Folge dieser Spezialisierung ergaben sich an vielen Stellen während der Implementierung Sonderfälle, die explizit zu lösen waren. Diese wurden in der ursprünglichen Implementierung implizit abgedeckt durch die geschickte Wahl der Programmarchitektur. Einer dieser Sonderfälle war das vertauschen der Join-Value Spalte, auf dessen Lösung in 6.2 eingegangen wird. In der tupelweisen Ausführung tritt dieses Problem nicht auf, da die Register eigenständige Objekte sind von denen jedes nur eine Komponente der Tupel enthält. Es besteht daher kein expliziter Zusammenhang zwischen diesen Registern, wie er bei der Blockimplemntierung eingeführt wurde.

Um diese Probleme zu umgehen, und eine potentiell bessere Blockimplementierung zu realisieren, muss über noch weitreichendere Änderungen an der RDF-3X Architektur nachgedacht werden.

7. Evaluation

7.1. Testumgebung

Die Messungen wurden durchgeführt auf einem *Dell Optiplex 755* mit einer *Intel Core2 Quad Q9300 CPU* bei *2,5GHz*. Das System verfügt über *4GB* Hauptspeicher, die Datenbanken sind auf einer lokalen Festplatte abgelegt. Als Betriebssystem kam ein *64 Bit Linux Kernel* der Version *2.6.31* zum Einsatz.

Die zu testende Anwendung wurde mit einem *GCC Version 4.5* kompiliert, wobei sämtliche Optimierungen der Stufe drei *(O3)*, sowie Optimierungen für die *Intel Core2-Architektur* aktiv waren.

7.1.1. XML-Query-Plan

Das Ziel hinter der Implementierung des XML-Anfrageplan ist, die Möglichkeit zu bieten direkten Einfluss auf den Anfrageplan, und damit auf die Struktur des Operatorbaums zu nehmen.

Der XML-Anfrageplan ist ein hilfreiches Werkzeug bei der Entwicklung und dem Debugging von Operatoren und Registern. Während der Evaluation bot er zudem die Möglichkeit, gezielt die Leistung von Operatoren und Registern in unterschiedlichen Situationen messen.

Der XML-Anfrageplan stellt eine unmittelbare Repräsentation des internen Query-Plans dar, und kann daher leicht durch rekursiven Aufbau in einen solchen überführt werden.

Ein beispielhafter Plan für einen Misch-Verbund könnte wie folgt aufgebaut sein.

```
<Parameters>
  <ProjectedVariable name="var" />
</Parameters>
<QueryPlan>
  <MergeJoin JoinOn="var">
    <IndexScan DataOrder="OPS" varSubject="var" constPredicate="hasCharacter" constObject="James Bond"/>
    <IndexScan DataOrder="OPS" varSubject="var" constPredicate="writtenBy" constObject="Ian Fleming"/>
  </MergeJoin>
</QueryPlan>
```

Die Struktur des Operatorbaums kann hier leicht in XML repräsentiert werden. Alle nötigen Parameter welche die Operatoren zur Initialisierung benötigen, werden an den entsprechenden XML-Objekten durch Attribute gesetzt. Ein Scan wird mit den drei Attributen eines Tripels initialisiert, die Konstanten oder Variablen repräsentieren. `DataOrder` bestimmt die Permutation des verwendeten Indexes. Verbund-Operatoren werden mit der Join-Variable initialisiert, und enthalten stets zwei weitere XML-Elemente welche die Unterbäume des Operators bilden. Schließlich können noch eine oder mehrere Variablen zur Projektion ausgewählt werden.

7.1.2. Testdaten

Einige Eigenschaften des experimentellen Systems können über Parameter festgelegt werden, die potentiellen Einfluss auf die Verarbeitungszeit haben. Dies umfasst die Anzahl der Partitionen des partitionierten Hash-Verbunds, sowie die Anzahl der Zeilen eines Block-Registers. Beide Parameter sollen anhand von Tests mit unterschiedlichen Werten bestimmt werden, indem derjenige Wert ausgewählt wird, welcher die niedrigste Antwortzeit erreicht.

Um die Anzahl der Partitionen zu bestimmen, sind Leistungsmessungen außschließlich am Hash-Verbund Operator erwünscht. Um andere Effekte auf die Laufzeit auszuschließen, wird daher mit Hilfe des XML-Anfrageplans ein Operatorbaum gewählt der lediglich einen Hash-Verbund, und einen Ausgabe-Operator umfasst. Da die Leistung des Operators unter realitätsnahen Bedingungen gemessen werden soll, ich es nötig dass er keine sortierten Werte als Eingaberelationen erhält. Dies ist jedoch mit dem vorgeschlagenen Operatorbaum nicht möglich, da Scan-Operatoren stets sortiere Relationen produzieren.

Daher wurde für diesen Test ein neuer Opertortyp, der *Random Generator* implementiert. Jede Instanz dieses Operators erzeugt einen unterschiedlichen, aber wiederholbaren Strom von pseudo-zufälligen Werten. Der Generator wurde für die Messungen so konfiguriert, dass er *eine Millionen* Werte aus dem Bereich *1 bis 500.000* produziert. Diese Kombination führ dazu, dass 1% der erzeugten Werte einen Join-Partner findet.

Für diese weiteren Tests wurden zwei Sätze von RDF-Tripeln nach folgendem Muster generiert.

```
class0 property0 "5" .
class0 property1 "8" .
class0 property2 "0" .
class0 property3 "2" .
class1 property0 "7" .
class1 property1 "8" .
class1 property2 "1" .
class1 property3 "8" .
```

Beide Datensätze enthalten *50 Millionen class*-Objekte, von denen jedes mit vier unterschiedlichen Prädikat/Objekt Kombinationen vorkommt.

Der erste Datensatz (A) enthält zu jedem Subjekt die Prädikate $property0$ bis $property3$, deren Objekte dabei von einem Zufallsgenerator aus den Zahlen 0 bis 9 ausgewählt wurden. Damit umfasst der gesamte Datensatz *200 Millionen* eindeutige Tripel.

Datensatz B enthält zu jedem Subjekt ebenfalls vier unterschiedliche Prädikate, die jedoch von einem Zufallsgenerator aus den Werten $property0$ bis $property39$ ausgewählt wurden. Im Durchschnitt hat damit jedes Subjekt ein bestimmtes Prädikat. Die Objekte wurden analog zu Datensatz A erzeugt, die Anzahl der gesamten Tripel ist ebenfalls dieselbe.

Durch die Auswahl dieser Datensätze ist es möglich, mit Anfragen auf A einen Operatorbaum zu erhalten in welchem ein Großteil der gelesenen Tupel ein Join-Partner findet, und damit eine große Ergebnismenge produziert (hohe Selektivität). Datensatz B hingegen enthält nur wenige Tripel mit übereinstimmenden Prädikat/Objekt-Kombinationen, sodass während der Verarbeitung ein Großteil der gelesenen Tupel verworfen wird (niedrige Selektivität).

7.2. Ergebnisse

Ein Messergebnis wurde bestimmt indem der zugehörige Testlauf zehn man wiederholt ausgeführt, um daraus das beste Ergebnis auszuwählen. Die Werte repräsentieren daher Messungen mit *warmem Cache*.

7.2.1. Bestimmung der Parameter

Abbildung 7.1 zeigt die Ergebnisse der Messungen des Hash-Verbund Operators mit zufällig generierten Werten. In der Abbildung nicht zu erkennen, betrug die Antwortzeit des nicht-partitionierten Hash-Verbunds unter identischen Bedingungen im besten Fall $0,59 Sekunden$. Die beste Laufzeit des partitionierten Hash-Verbunds liegt mit $0,51 Sekunden$ bei 128 Partitionen ($7 Bit$).

Abbildung 7.2 zeigt die Ergebnisse der Messungen mit unterschiedlichen Blockgrössen. Der verwendete XML-Anfrageplan findet sich im Anhang und SPARQL-Queries.

7.2.2. Leistungsmessung verschiedener Operatorbäume

Zur Beurteilung der Leistung des implementierten Ansatzes, wurden nun Vergleichsmessungen zwischen der tupelweisen, und blockweisen Ausführung durchgeführt.

Diese Tests umfassen fünf unterschiedliche SPARQL-Anfragen, um das Verhalten unterschiedlich hoher und breiter Operatorbäume zu untersuchen. Die Messungen wurden mit beiden Datensätzen A, und B für hohe, respektive niedrige Selektivität wiederholt. Die Ergebnisse finden sich in Tabelle 7.1 und Tabelle 7.2. Die verwendeten SPARQL-Anfragen sind im Anhang B aufgelistet.

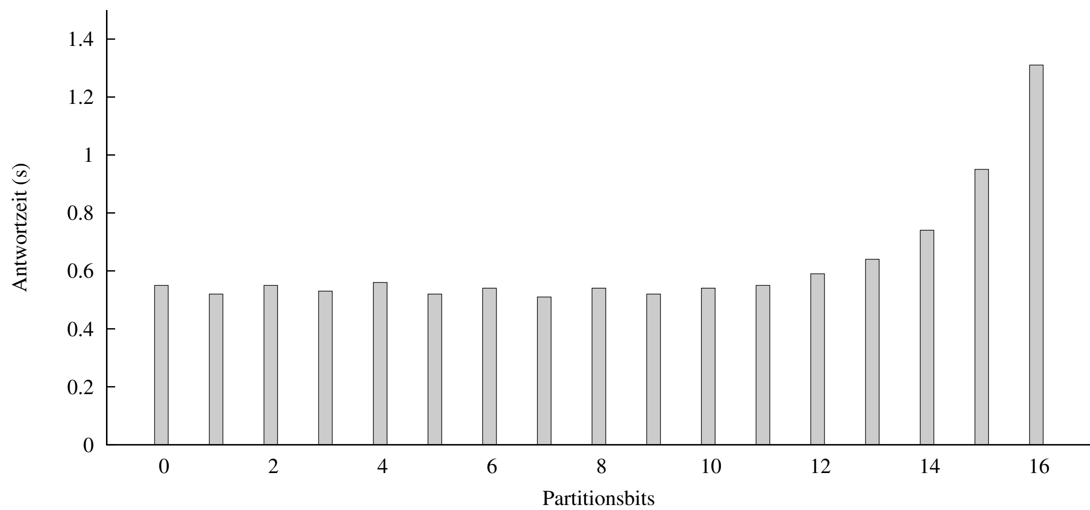

Abbildung 7.1.: Messergebnisse – Bestimmung der Anzahl von Partitionen

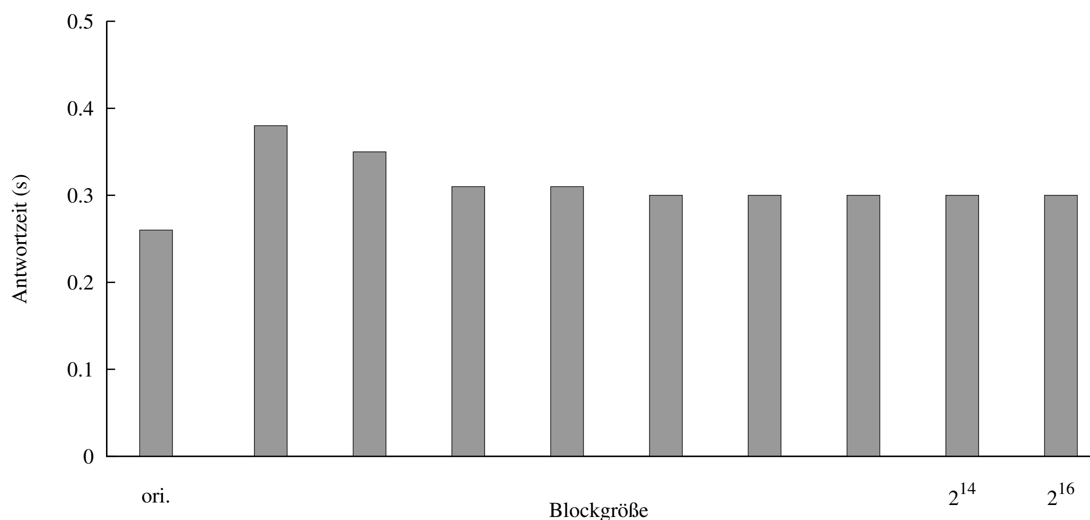

Abbildung 7.2.: Messergebnisse – Bestimmung der Anzahl von Blockzeilen

Ausführung	QA1	QA2	QA3	QB1	QB2
Blockweise	0,31	0,37	1,29	2,61	4,08
Tupelweise	0,27	0,30	1,05	2,06	3,50

Tabelle 7.1.: Messergebnisse – Vergleich block- und tupelweise Ausführung, hohe Selektivität – Datensatz A

Ausführung	QA1	QA2	QA3	QB1	QB2
Blockweise	0,00	0,03	0,04	0,00	0,13
Tupelweise	0,00	0,00	0,00	0,00	0,10

Tabelle 7.2.: Messergebnisse – Vergleich block- und tupelweise Ausführung, niedrige Selektivität – Datensatz B

7.3. Diskussion

Die Laufzeit des Hash-Verbund Operators hat wie erwartet eine Verschlechterung erfahren bei sehr hoher Anzahl von Partitionen. Es wäre zu erwarten gewesen, dass eine zu niedrige Anzahl an Partitionen ebenso einen Nachteil darstellt. Dieser zeigt sich jedoch nur in der Produktions-, nicht aber in der Aufbauphase. Letzere profitiert potentiell von wenigen Partitionen, da eine geringere Anzahl an verteilten Speicherzugriffen nötig ist. Daher lässt sich Schlussfolgern, was auch Erfahrungen aus der Implementierung nahelegten. Die Aufbauphase hat einen wesentlich höheren Anteil an der Gesamtlaufzeit des Algorithmus, als die Produktionsphase. Eine Beschleunigung der letzteren führt also lediglich zu geringen, oder nicht messbaren Verbesserungen der Laufzeit. Es kann jedoch eine leichte Verbesserung der Laufzeit verglichen mit der ursprünglichen Implementierung festgestellt werden, was nahelegt dass die Produktionsphase eine deutliche Beschleunigung erfährt, da die Aufbauphase des partitionierten Hash-Join deutlich aufwändiger ist als die er ursprünglichen Implementierung.

Die Bestimmung der Parameter hat bei der Messung mit unterschiedlichen Blockgrößen eine interessante Schlussfolgerung nahegelegt. Wie erwarten führen kleine Blockgrößen zu keiner Beschleunigung der Operatoren, da nach wie vor viele Sprünge in der Ausführung entstehen. Zusätzlich kommt jedoch der Aufwand der Blockverwaltung hinzu, was zu einem schlechten Laufzeitverhalten führt. Ab einer Blockgröße von 256 Zeilen, ist keine weitere Beschleunigung der Laufzeit durch seltenere Blockwechsel mehr zu beobachten. Es kann daher angenommen werden, dass Operatoren nicht, oder nur gering von der hier verwendeten blockweisen Verarbeitung profitieren, da sich sonst eine zunehmende Verbesserung der Laufzeit mit steigender Blockgröße einstellen sollte.

Die Messungen haben gezeigt dass die hier vorgestellte Blockimplementierung der Ausführung einen zusätzlichen Aufwand hinzufügt, welcher die potentielle Beschleunigung der Operatoren übersteigt.

8. Zusammenfassung

Mit der hier vorgestellten Implementierung einer blockweisen Verarbeitung konnte keine Verbesserung der Laufzeit erreicht werden, wie die Messungen aus dem vorherigen Kapitel gezeigt haben.

der Grund dafür liegt in der konkreten Umsetzung, welche das Kopieren von Tupeln nach jedem Operator erfordert. Dieser zusätzliche Speicheraufwand ist groß genug, um eine potentielle Beschleunigung durch die blockweise Verarbeitung zu überschatten.

Daher besteht noch Raum für Optimierungen an dem gewählten Ansatz, so sollte das Kopieren von Tupeln unterlassen werden, und stattdessen ein einmal geschriebenes Tupel möglichst lange wiederverwendet werden. Die Scan-Operatoren sollten zudem mit speziellem Hinblick auf die blockweise Verarbeitung weiterentwickelt werden, da sie bereits Blöcke von Daten lesen, und zwischenspeichern. Ein zusätzliches Kopieren in eine andere Block-Struktur sollte daher unterbleiben. Der Misch-Verbund Operator kann ebenfalls noch weiter von Puffer-Blöcken profitieren, indem die in Kapitel 6.3.2 angesprochenen Kopieroperationen zum Betrachten weiterer Tupel durch Leseoperationen auf dem bereits verfügbaren Block ersetzt werden.

Abschließend lässt sich sagen, dass der Ansatz der blockweisen Verarbeitung von Tupeln speziell in RDF-3X noch unausgeschöpftes Potential bietet, was auch ähnliche Arbeiten aus Kapitel 3 nahelegen.

A. Listings

```
struct Block {
  unsigned* counts;
  unsigned* values;
  unsigned* tails[];
};
```

Listing 1: Datenstruktur für Blockspalten

```
void writeNextJoinResult(const unsigned value, const unsigned* leftTail, const unsigned* rightTail) {

  // Apply additional Join-Conditions if necessary
  if (SelectionType::selection(leftTail, rightTail, selectionPairs)) {

    // Switch the Position of the Join-Value Column if necessary
    const unsigned newValue = SwapType::getJV(value, leftTail, rightTail, jvSwapPosition);
    block->values[writePosition] = newValue;

    // Copy only the Columns that are still needed
    unsigned tailPos = 0;
    for(VarList::const_iterator i = leftProj.begin(); i != leftProj.end(); i++, tailPos++){
      block->tails[tailPos][writePosition] = leftTail[*i];
    }

    for(VarList::const_iterator i = rightProj.begin(); i != rightProj.end(); i++, tailPos++){
      block->tails[tailPos][writePosition] = rightTail[*i];
    }
    writePosition++;
  }
}
```

Listing 2: Einfügen von Join-Ergebnissen in den Ausgabeblock

B. SPARQL- und XML-Anfragen

```
<QueryPlan>
  <HashJoin JoinOn="var">
    <RandomGenerator produce="var" />
    <RandomGenerator produce="var" />
  </HashJoin>
</QueryPlan>
```

Listing 3: XML-Anfrageplan zur Bestimmung der Anzahl von Partitionen

```
<QueryPlan>
  <MergeJoin JoinOn="var">
      <MergeJoin JoinOn="var">
        <IndexScan DataOrder="OPS" varSubject="var" constPredicate="property0" constObject="1"/>
        <IndexScan DataOrder="OPS" varSubject="var" constPredicate="property1" constObject="2"/>
      </MergeJoin>
      <IndexScan DataOrder="OPS" varSubject="var" constPredicate="property2" constObject="3"/>
    </MergeJoin>
</QueryPlan>
```

Listing 4: XML-Anfrageplan zur Bestimmung der Blockgröße

```
select ?n where {
?n property0 "1".
?n property1 "1".
}
```

Listing 5: SPARQL-Anfrage QA1

```
select ?n where {
?n property0 "1".
?n property1 "1".
?n property2 "1".
?n property3 "1".
}
```

Listing 6: SPARQL-Anfrage QA2

```
select ?n where {
?n property0 "1".
?n property1 "1".
?n property2 "1".
?n property3 "1".
?n ?a "1".
?n ?a "1".
}
```

Listing 7: SPARQL-Anfrage QA3

```
select ?n ?a where {
?n property0 "1".
?n property1 "1".
?n property2 ?a.
?n property3 ?a.
}
```

Listing 8: SPARQL-Anfrage QB1

```
select ?n where {
?n property0 "1".
?n property1 ?a.
?m property2 ?a.
?m property3 "1".
}
```

Listing 9: SPARQL-Anfrage QB2

Literaturverzeichnis

[Aba08] D. J. Abadi. Query Execution in Column-Oriented Database Systems. MIT PhD Dissertation, 2008. PhD Thesis. (Zitiert auf Seite 18)

[Ask09] N. Askitis. Fast and compact hash tables for integer keys. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, ACSC '09, pp. 113–122. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2009. URL http://portal.acm.org/citation.cfm?id=1862659.1862675. (Zitiert auf Seite 36)

[BKM08] P. A. Boncz, M. L. Kersten, S. Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, 2008. doi:http://doi.acm.org/10.1145/1409360.1409380. (Zitiert auf Seite 15)

[blo01] Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *ICDE '01: Proceedings of the 17th International Conference on Data Engineering*, p. 567. IEEE Computer Society, Washington, DC, USA, 2001. (Zitiert auf Seite 20)

[BMK99] P. A. Boncz, S. Manegold, M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pp. 54–65. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. URL http://portal.acm.org/citation.cfm?id=645925.671364. (Zitiert auf den Seiten 16, 31 und 36)

[CMR09] J. Cieslewicz, W. Mee, K. A. Ross. Cache-conscious buffering for database operators with state. In *DaMoN '09: Proceedings of the Fifth International Workshop on Data Management on New Hardware*, pp. 43–51. ACM, New York, NY, USA, 2009. doi:http://doi.acm.org/10.1145/1565694.1565704. (Zitiert auf Seite 18)

[GK07] P. Garcia, H. F. Korth. Pipelined hash-join on multithreaded architectures. In *DaMoN '07: Proceedings of the 3rd international workshop on Data management on new hardware*, pp. 1–8. ACM, New York, NY, USA, 2007. doi:http://doi.acm.org/10.1145/1363189.1363191. (Zitiert auf Seite 19)

[Gra94] G. Graefe. Volcano—An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.*, 6:120–135, 1994. doi:http://dx.doi.org/10.1109/69.273032. URL http://dx.doi.org/10.1109/69.273032. (Zitiert auf den Seiten 6 und 15)

[HNZB07] S. Héman, N. Nes, M. Zukowski, P. Boncz. Vectorized data processing on the cell broadband engine. In *Proceedings of the 3rd international workshop on Data management on new hardware*, DaMoN '07, pp. 4:1–4:6. ACM, New York, NY, USA, 2007. doi:http://doi.acm.org/10.1145/1363189.1363195. URL http://doi.acm.org/10.1145/1363189.1363195. (Zitiert auf Seite 17)

[HR01] T. Härder, E. Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung, 2. Auflage*. Springer, 2001. (Zitiert auf Seite 7)

[IT08] Z. G. Ives, N. E. Taylor. Sideways Information Passing for Push-Style Query Processing. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pp. 774–783. IEEE Computer Society, Washington, DC, USA, 2008. doi:http://dx.doi.org/10.1109/ICDE.2008.4497486. (Zitiert auf Seite 27)

[NW09a] T. Neumann, G. Weikum. The RDF-3X Engine for Scalable Management of RDF Data. Research Report MPI-I-2009-5-003, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2009. (Zitiert auf Seite 13)

[NW09b] T. Neumann, G. Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pp. 627–640. ACM, New York, NY, USA, 2009. doi:http://doi.acm.org/10.1145/1559845.1559911. (Zitiert auf den Seiten 28 und 29)

[NW10] T. Neumann, G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19:91–113, 2010. doi:http://dx.doi.org/10.1007/s00778-009-0165-y. URL http://dx.doi.org/10.1007/s00778-009-0165-y. (Zitiert auf Seite 6)

[SKN94] A. Shatdal, C. Kant, J. F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pp. 510–521. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994. URL http://portal.acm.org/citation.cfm?id=645920.758363. (Zitiert auf Seite 15)

[ZCRS05] J. Zhou, J. Cieslewicz, K. A. Ross, M. Shah. Improving database performance on simultaneous multithreading processors. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pp. 49–60. VLDB Endowment, 2005. URL http://portal.acm.org/citation.cfm?id=1083592.1083602. (Zitiert auf Seite 20)

[ZR04] J. Zhou, K. A. Ross. Buffering databse operations for enhanced instruction cache performance. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pp. 191–202. ACM, New York, NY, USA, 2004. doi:http://doi.acm.org/10.1145/1007568.1007592. URL http://doi.acm.org/10.1145/1007568.1007592. (Zitiert auf Seite 17)

Alle URLs wurden zuletzt am 10.04.2011 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Tim Waizenegger)