Philipps-Universität Marburg
Fachbereich Mathematik und Informatik

Anfrageoptimierung auf Datenströmen

Diplomarbeit

von

Tobias Schäfer

Betreuer: Prof. Dr. Bernhard Seeger

Marburg/Lahn, Oktober 2003

Zusammenfassung

Im Rahmen dieser Arbeit werden Techniken der Anfrageoptimierung auf ihre Anwendbarkeit im Kontext von Datenströmen hin untersucht. Zu diesem Zweck wird zunächst die herkömmliche Optimierung von Anfragen in Datenbanksystemen eingehend betrachtet und mit den Anforderungen, welche Datenströme an eine Anfrageoptimierung stellen, verglichen. Weiterhin werden mit der anfragenübergreifenden und der dynamischen Optimierung Erweiterungen der Anfrageoptimierung untersucht, welche sich die Lösung spezieller Probleme, die auch im Kontext von Datenströmen eine zentrale Bedeutung besitzen, zum Ziel gemacht haben. Aus diesen Untersuchungen geht eine Optimierungsstrategie hervor, deren Schwerpunkt in der Erkennung und Wiederverwendung gemeinsamer Teilanfragen liegt. Die Entscheidung zugunsten dieses Schwerpunkts liegt in den Charakteristika aktiver Datenquellen und den Eigenschaften von Anfragen an diese begründet. Da jede Teilanfrage einer solchen Anfrage wiederum eine Datenquelle definiert und weiterhin jede Datenquelle ihre Daten jedermann zugänglich macht, drängt sich diese Entscheidung geradezu auf.

Ein wichtiger Aspekt der Arbeit liegt außerdem in der Integration einer solchen Anfrageoptimierung in die Bibliothek XXL. Da die Bibliothek neben einer Algebra objekt-relationaler Operatoren für aktive Datenquellen bereits über eine flexible Anfrageoptimierung für Anfragen an relationale Datenbanken verfügt, ist die Frage nach einem Brückenschlag zwischen diesen beiden Modellen von besonderem Interesse. Als Antwort auf diese Frage entsteht eine Algebra logischer Operatoren, mit deren Hilfe sowohl Anfragen an passive als auch Anfragen an aktive Datenquellen formuliert werden können. Dies wird jedoch erst durch ein dynamisches Metadatenkonzept ermöglicht, mit dessen Hilfe zu Objekten beliebige Informationen zur Verfügung gestellt werden können. Mit Hilfe dieser Konzepte kann die bestehende Anfrageoptimierung derart erweitert werden, dass auch Anfragen auf Datenströmen mit ihrer Hilfe optimiert werden können. Dies bildete den Ausgangspunkt für die Definition von speziellen Transformationsregeln und Optimierungsverfahren für Datenströme, so dass letzten Endes mit der bestehenden Anfrageoptimierung, welche ursprünglich allein für die Optimierung von Anfragen an passive Datenquellen bestimmt war, eine vollwertige Anfrageoptimierung auf Datenströmen vorliegt.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die zunehmende Bedeutung von aktiven Datenquellen für unsere heutige Gesellschaft bildet die Grundlage für einen neuen Interessenschwerpunkt in der Informationsverarbeitung. Der Vorteil dieser Datenquellen gegenüber ihren passiven Geschwistern liegt in ihrer Fähigkeit begründet, Informationen selbständig an ihre Umwelt weiterleiten zu können. Diese Fähigkeit befreit sie von der Notwendigkeit, Informationen bis zu ihrer Abfrage lokal zwischenzuspeichern, und die von einer aktiven Datenquelle gelieferten Informationen erhalten einen starken zeitlichen Bezug. Dieser bleibt jedoch nur erhalten, wenn die Empfänger solcher Informationen ihrerseits ebenfalls umgehend auf sie reagieren (und sie ggf. weiterleiten).

Die folgenden Beispiele sollen verdeutlichen, dass aktive Datenquellen – und Anwendungen, die aktive Quellen nutzen – in allen Bereichen des täglichen Lebens gefunden werden können.

- In Wohn- und Arbeitsstätten werden Gebäudemanagementsysteme verwendet, um die Regelung von Zentralheizungen, Beleuchtungseinrichtungen, Jalousien etc. zentral zu verwalten und teilweise zu automatisieren. Die hierfür benötigten Informationen über Außenklima, Raumtemperatur, Lichtverhältnisse und Nutzung der Räumlichkeiten erhalten diese Systeme über eine Reihe von Sensoren, die als aktive Datenquellen angesehen werden können.

- Um dem täglichen Verkehrsaufkommen Herr zu werden und bestehende Streckenkapazitäten optimal auszunutzen, werden Verkehrsmanagementsysteme zur intelligenten Steuerung des Verkehrs eingesetzt. Neben den in Datenbanksystemen gespeicherten Informationen über Streckenkapazitäten, Ausweichrouten und Baustellen benötigen solche Systeme auch aktuelle Informationen über Verkehrsfluss und -dichte, um realistische Verkehrsprognosen stellen und Teile des Verkehrsaufkommens auf Ausweichrouten verlagern zu können. Induktionsschleifen und optische Sensoren, welche die benötigten Informationen liefern, stellen ebenfalls aktive Datenquellen dar.

- Mittels elektronischer Etiketten werden Unternehmen in die Lage versetzt, den Weg ihrer Ware zu verfolgen. Auf diese Art können automatisch Lagerbestände

verwaltet, Waren nachbestellt, Auslieferungen organisiert und Verkaufstransaktionen analysiert werden. Auch hier stellt das elektronische Etikett eine aktive Datenquelle dar, die bei einem Ortswechsel der Waren diese Information an ihre Umwelt weiterleitet.

- Letztendlich lässt sich jeder Informationsdienst, wie beispielsweise ein Newsletter-Dienst, der via Email oder SMS Nachrichten an registrierte Nutzer versendet, als aktive Datenquelle betrachten.

Alle diese Beispiele haben gemein, dass herkömmliche Techniken der Informationsverarbeitung, wie das Speichern in großen Datenbanksystemen zur anschließenden Analyse, für die Verarbeitung der von aktiven Datenquellen gelieferten Informationen eher ungeeignet sind.

Um diesen Nachteil auszugleichen, werden derzeit sogenannte Datenstrommanagementsysteme[1] entwickelt, welche die Verwaltung aktiver Datenquellen und die direkte Ausführung von Anfragen auf ihnen erlauben. Wie aus den Beispiel eines Verkehrsmanagementsystems ersichtlich wird, müssen diese Systeme nicht nur dem zeitlichen Bezug und den enormen Mengen der von den Datenquellen gelieferten Informationen gerecht werden, sondern auch in der Lage sein, große Zahlen von gleichzeitig aktiven Anfragen zu bewältigen. Hierfür ist eine effiziente Optimierung der an das System gestellten Anfragen unumgänglich, um einerseits Anfragen möglichst schnell beantworten zu können und andererseits den Speicherbedarf für die Beantwortung der Anfragen möglichst gering zu halten.

Diesem interessanten Themenbereich widmet sich die vorliegende Arbeit. Ihre Aufgabe besteht darin, existierende Konzepte der Anfrageoptimierung mit den Anforderungen von Anfragen an aktive Datenquellen in Einklang zu bringen und im Rahmen der Bibliothek XXL[2] verfügbar zu machen. Bei XXL handelt es sich um eine in *Java*[3] entwickelte Bibliothek, welche eine umfassende Infrastruktur von Algorithmen und Datenstrukturen für die Implementierung komplexer Datenbankfunktionalität bereitstellt. Sie bietet neben einer Fülle von Index- und Speicherstrukturen auch relationale Operatoren, mit denen Datenbankanfragen direkt formuliert werden können. Zusätzlich enthält die Bibliothek XXL eine regelbasierte Optimierung für Anfragen auf passiven Datenquellen und eine Algebra für die Arbeit mit aktiven Datenquellen. Nähere Informationen zu dieser Bibliothek finden sich in [BDS00] und [BBD+01], sowie im Internet unter [Phi03].

1.1 Gliederung der Arbeit

Dieser Abschnitt bietet einen Überblick über die Kapitel, in die sich die vorliegende Arbeit untergliedert. Eine allgemeine Einführung in die Optimierung von Anfragen in

[1] Der Begriff *data stream management system* (DSMS) wurde in [BBD+02] in Anlehnung an *database management system* (DBMS) geprägt.

[2] Abkürzung für *eXtensible and fleXible Library*

[3] Informationen zu der Programmiersprache *Java* finden sich in [Sun03].

Datenbanksystemen bietet Kapitel 2. Dabei werden die Abläufe der statischen und der dynamischen Optimierung näher erläutert und wichtige Verfahren, welche hierbei Verwendung finden, vorgestellt. Anschließend folgt ein Überblick über die im Rahmen der Bibliothek XXL implementierte Anfrageoptimierung für Anfragen an passive Datenquellen. Aktive Datenquellen und Datenströme werden im Kapitel 3 eingeführt, wobei ihre speziellen Eigenschaften verdeutlicht und die daraus resultierenden Anforderungen an Datenstrommanagementsysteme herausgestellt werden. Besonderes Augenmerk gilt hier dem Unterschied zwischen Datenbank- und Datenstrommanagementsystemen, um die veränderten Anforderungen an eine Anfrageoptimierung zu verdeutlichen. Den Abschluss dieses Kapitels bildet eine Vorstellung der in XXL integrierten Algebra objektrelationaler Operatoren für aktive Datenquellen. Das darauffolgende Kapitel 4 bietet einen Überblick über Erweiterungen der herkömmlichen Anfrageoptimierung, welche im Rahmen von Datenströmen von besonderem Interesse sind. Dabei wird sowohl die anfragenübergreifende als auch die dynamische Optimierung näher betrachtet. Kapitel 5 beschäftigt sich mit dem Entwurf einer Anfrageoptimierung im Kontext von Datenstrommanagementsystemen. Neben den für die Optimierung von Datenströmen notwendigen Voraussetzungen werden Datenstrukturen zur Darstellung von Anfrageplänen erläutert. Anschließend werden Verfahren für die statische und die dynamische Anfrageoptimierung auf Datenströmen vorgestellt und analysiert. Mit der Implementierung dieser Datenstrukturen und Algorithmen und ihrer Anwendung befasst sich schließlich Kapitel 6. Dort werden die Schnittstellen erläutert und exemplarisch auf die einzelnen Klassen eingegangen. Dabei wird besonderes Augenmerk auf den generischen Aufbau der Optimierung gelegt. Den Abschluss bildet eine Zusammenfassung und anschließende Diskussion der Ergebnisse dieser Arbeit in Kapitel 7, welches mit einem Ausblick auf neue und verbleibende Fragen der Forschung beendet wird.

Kapitel 2

Anfrageoptimierung

Das folgende Kapitel soll einen Überblick über die Möglichkeiten und den Ablauf der Anfrageoptimierung in herkömmlichen Datenbanksystemen bieten. Ein Datenbanksystem besteht aus einer Datenbank und einem Datenbankmanagementsystem. Während die Datenbank die vom Datenbanksystem zu verwaltenden Daten enthält, besteht das Datenbankmanagementsystem aus der zur Verwaltung der Daten benötigten Software. D. h. Anfragen an das Datenbanksystem werden korrekterweise an das Datenbankmanagementsystem gestellt, dort in eine ausführbare Form übersetzt und schließlich auf der eigentlichen Datenbank ausgeführt. Einen wichtigen Teil des Datenbankmanagementsystems stellt die Anfrageoptimierung dar. Ihre Aufgabe besteht darin, Anfragen in eine ausführbare Form zu übersetzen, die mit möglichst geringem Ressourcenbedarf die Ergebnisse der Anfrage ermittelt. Obwohl die Aufgabe der Anfrageoptimierung im Grunde nur darin besteht, eine Anfrage in eine günstigere Form zu übertragen, wird normalerweise auch die Übersetzung einer – in einer beliebigen Anfragesprache formulierten – Anfrage in eine logische Repräsentation und die Übersetzung der optimierten logischen Repräsentation in einer ausführbare Form zu ihren Aufgaben gezählt. Diese Ausweitung ihrer Aufgaben ermöglicht es, die Anfrageoptimierung außerhalb von Datenbankmanagementsystemen zu betrachten.

Grundsätzlich lässt sich die Anfrageoptimierung in zwei Klassen unterteilen. Die statische Optimierung umfasst die oben beschriebenen Schritte, die komplett vor der Ausführung der Anfrage durchgeführt werden. Dahingegen bietet eine dynamische Optimierung die Möglichkeit der Umstrukturierung von Anfragen zu ihrer Laufzeit, um auf veränderte Bedingungen reagieren zu können. Dieses Kapitel beschäftigt sich mit der statischen Optimierung einzelner Anfragen, welche gewöhnlich in herkömmlichen Datenbankmanagementsystemen zur Anfrageoptimierung eingesetzt wird. Die statische Optimierung von Anfragemengen und die dynamische Optimierung werden in Kapitel 4 näher erläutert. Auf diese eher theoretische Darstellung der statischen Anfrageoptimierung folgt die Vorstellung einer im Rahmen der Bibliothek XXL implementierten Anfrageoptimierung und ihrer Funktionalität.

2.1 Statische Anfrageoptimierung

Eine Anfrage an ein Datenbanksystem wird allein mit den Daten beantwortet, die zum Zeitpunkt der Anfrage in der Datenbank des Systems verwaltet werden. D. h. die Hauptaufgabe der Anfrageoptimierung besteht darin, den Zeitaufwand, der zur Bestimmung der Ergebnisse einer Anfrage notwendig ist, zu minimieren. Dadurch entstehen im Allgemeinen Anfragen, deren Lebensdauer im System recht kurz sind. Für solche Anfragen empfiehlt sich eine statische Optimierung, während dynamische Optimierungen eher unpassend sind, da die kurze Lebensdauer der Anfragen die mit dynamischer Optimierung verbundenen teuren Umstrukturierungen nicht rechtfertigt.

Die statische Optimierung lässt sich in mehrere logische Phasen unterteilen. Als Eingabe erhält sie eine in einer beliebigen Anfragesprache formulierte Anfrage. Hierfür stehen neben der wohl bekanntesten Anfragesprache SQL[1] noch eine ganze Reihe weiterer, wie beispielsweise QUEL[2] oder QBE[3], zur Verfügung. Nach der Eingabe der Anfrage muss diese nun zunächst auf ihre korrekte Form hin überprüft werden. Hierfür wird sie erst geparst und anschließend einer syntaktischen und semantischen Analyse unterzogen. Liegt die Anfrage in einer korrekten Form vor, so kann sie nun in eine interne logische Repräsentation übertragen werden. Für diese interne Repräsentation werden meist formalisierte mathematische Modelle wie ein relationaler Kalkül oder die (erweiterte) relationale Algebra[4] verwendet und mittels baumartiger Strukturen repräsentiert. Den Beispielen dieses Kapitels liegt der Einfachheit halber stets ein Operatorbaum der relationalen Algebra als interne logische Repräsentation der Anfrage zu Grunde. Die Blätter eines solchen Operatorbaumes werden durch die verwendeten Datenquellen gebildet, während die inneren Knoten Operationen der relationalen Algebra entsprechen. Nun folgt die eigentliche Optimierung der Anfrage. Zu diesem Zweck wird die Anfrage normalisiert und anschließend logisch und physisch optimiert. Das Ergebnis der physischen Optimierung ist ein Anfrageplan, der genau bestimmt, welche Algorithmen in welcher Reihenfolge auf die verwendeten Datenquellen angewandt werden, um das Ergebnis der Anfrage zu bestimmen. Während der logischen und der physischen Optimierung finden Kostenanalysen statt, welche die erzeugten Anfragen bzw. Anfragepläne hinsichtlich ihrer Qualität bewerten. Nach der eigentlichen Optimierung wird der optimale Anfrageplan in eine ausführbare Form übersetzt. Hierbei kann es sich um Maschinencode, generierte Funktionen oder einen physikalischen Operatorbaum handeln. Mit der Ausgabe der ausführbaren Repräsentation der eingegebenen Anfrage endet die Anfrageoptimierung schließlich.

Lässt man die notwendigen Übersetzungsphasen außer Betracht, so reduziert sich die eigentliche Anfrageoptimierung auf drei Phasen. Diese drei Phasen – Normalisierung, logische Optimierung und physische Optimierung – werden im folgenden betrachtet.

[1] Abkürzung für *Structured Query Language*

[2] Abkürzung für *QUEry Language*

[3] Abkürzung für *Query-By-Example*

[4] Einen guten Überblick über gängige relationale Anfragesprachen und mathematische Modelle bietet [KE99].

2.1.1 Normalisierung

Die Normalisierung verfolgt das Ziel, Anfragen für die folgenden Phasen der Optimierung in eine einheitliche Form zu bringen. Zu diesem Zweck wird die Anfrage zunächst standardisiert. Der erste Schritt der Standardisierung erfolgt durch die Ersetzung aller in der Anfrage auftretender Sichten durch ihre zugrundeliegenden Anfragen. Anschließend werden in der Anfrage enthaltene Unteranfragen – wenn möglich – aufgelöst. In der relationalen Algebra beispielsweise lassen sich Unteranfragen häufig mittels Verbund- oder Divisionsoperationen in die umgebende Anfrage integrieren. Nachdem die Anfrage selbst standardisiert wurde, müssen jedoch noch die darin verwendeten Selektionsbedingungen in eine solche Form gebracht werden. Zu diesem Zweck werden häufig die aus der mathematischen Aussagenlogik bekannten Normalformen – wie konjunktive und disjunktive Normalform – verwendet.

Nachdem die gesamte Anfrage einschließlich aller Selektionsbedingungen in standardisierter Form vorliegt, findet nun eine Vereinfachung der Anfrage statt. Den günstigsten Ansatzpunkt für diese Vereinfachung bilden dabei die Selektionsbedingungen. Hier lassen sich Konstanten propagieren und konstante Teilausdrücke vorberechnen. Auch die Behandlung und/oder Eliminierung gemeinsamer Teilausdrücke bietet sich zur Vereinfachung von Selektionsbedingungen an. Anschließend kann die Anfrage selbst eventuell noch vereinfacht werden. Selektionen mit nicht erfüllbaren Selektionsbedingungen lassen sich durch eine leere Datenquelle ersetzen, während solche mit immer erfüllten Selektionsbedingungen aus der Anfrage entfernt und durch ihre zugrundeliegende Datenquelle ersetzt werden können. Desweiteren lassen sich auch hier konstante Teilanfragen vorberechnen und gemeinsame Teilausdrücke behandeln.

Abschließend kann die vereinfachte Anfrage noch verbessert werden. Die Verbesserung stellt eine rudimentäre Optimierung dar, die unabhängig von der Anfrage durchgeführt wird. D. h. die während der Verbesserung durchgeführten Transformationen der Anfrage haben die Eigenschaft, dass sie in jeder beliebigen Anfrage zu einer Optimierung führen. Transformationen, die diesen Anforderungen genügen, sind beispielsweise in der relationalen Algebra die Verknüpfung von Selektions- und Projektionsoperationen in einer Operation oder das Ersetzen einer Selektionsoperation auf einem kartesischen Produkt durch eine Verbundoperation.

Mit der Verbesserung endet schließlich die Phase der Normalisierung, die in Datenbankmanagementsystemen häufig bereits mit der Übersetzung der Anfrage in die interne logische Repräsentation verknüpft wird. Dies hat den Vorteil, dass Standardisierung, Vereinfachung und Verbesserung der Anfrage schon während seines Aufbaus – also ohne wiederholtes Durchlaufen der Anfrage – durchgeführt werden kann. Unabhängig davon, ob die Normalisierung als eigene Phase oder innerhalb der Übersetzung der Anfrage durchgeführt wird, schließt sich an sie die logische Optimierung an.

2.1.2 Logische Optimierung

Während der logischen Optimierung wird die normalisierte Anfrage mit rein algebraischen Mitteln optimiert. D. h. aus der Anfrage werden durch Anwendung von Trans-

formationen äquivalente Anfragen generiert, die anschließend bewertet werden, um sie dann weiter zu betrachten oder zu verwerfen. Die hierfür verwendeten Transformationen stellen dabei einfache Äquivalenzumformungen der Anfrage dar und berücksichtigen ausschließlich deren algebraische Eigenschaften. Aus diesem Grund wird die logische Optimierung auch häufig als algebraische oder logisch-algebraische Optimierung bezeichnet. In der relationalen Algebra ist ein häufiges Ziel solcher Transformationen beispielsweise das *Herabdrücken* von Selektionsoperationen zu den Datenquellen, um die Menge der zu verarbeitenden Daten möglichst früh zu reduzieren. Eine Transformation, die diesen Zweck verfolgt, ist das Herabdrücken von Selektionsoperationen unter Verbundoperationen. An diesem Beispiel zeigt sich auch die Notwendigkeit der algebraischen Struktur für die Optimierung, da eine Selektionsoperation nur dann zu einer einzigen Datenquelle der Verbundoperation herabgedrückt werden kann, wenn sie allein mit den von der Datenquelle bereitgestellten Daten ihre Selektionsbedingung auswerten kann. Ist dies nicht der Fall, so muss die Selektionsbedingung aufgeteilt werden. Die Teile der Selektionsbedingung, welche sich mit Hilfe der Daten einer einzigen Datenquelle auswerten lassen, können durch eine neue Selektionsoperation direkt auf sie angewandt werden, während die Teile, welche die Daten mehrerer Datenquellen miteinander verknüpfen, in der ursprünglichen Selektionsoperation oberhalb der Verbundoperation verbleiben.

Die durch Transformation entstandenen äquivalenten Anfragen müssen nun bewertet werden, um zu bestimmen, welche dieser Anfragen eine Optimierung der ursprünglichen Anfrage darstellen. Hierzu wird meist ein einfaches Kostenmodell verwandt, welches die Anzahl der Zwischenergebnisse einer Operation unter Annahme von Gleichverteilung der Daten abschätzt. Ein solches Kostenmodell bevorzugt also Anfragen, die wenig Zwischenergebnisse produzieren und damit weniger Speicher verbrauchen als andere. Jedoch ist die Aufgabe, zu einer gegebenen Kostenfunktion diejenige Anfrage zu bestimmen, welche die Kostenfunktion minimiert, NP-hart, da zu deren Lösung alle äquivalenten Anfragen erzeugt und mittels der Kostenfunktion bewertet werden müssen. Aus diesem Grund lassen sich nur für relativ kleine Anfragen exakte Lösungen berechnen. Für die Betrachtung komplexerer Anfragen werden dahingegen effizientere Algorithmen benötigt, die den Suchraum, in dem die optimale Anfrage gesucht wird, genügend weit einschränken.

Heuristische Verfahren verfügen zu diesem Zweck über eine methodische Anleitung, wie der Suchraum einzuschränken ist. Solche Heuristiken können beispielsweise festlegen, dass nach einer festen Zahl von Transformationen nur die n günstigsten Anfragen oder die Anfragen, bei denen die Selektionen am weitesten herabgedrückt wurden, weiter betrachtet werden. Im Gegensatz zu heuristischen Verfahren, führt bei randomisierten Verfahren keine feste Anleitung sondern der Zufall zur Reduktion des Suchraumes. Dabei kann der Zufall auf verschiedenen Wegen an der Optimierung beteiligt sein. Einerseits können aus der ursprünglichen Anfrage zufällig n äquivalente – aber nicht notwendigerweise günstigere – Anfragen erzeugt und anschließend einzeln optimiert werden und die günstigste dieser n Anfragen bildet dann das Ergebnis der logischen Optimierung. Andererseits können während der Optimierung auch Transformationen, welche die Anfrage verschlechtern, mit einer gewissen Wahrscheinlichkeit erlaubt werden, um

so auf der Suche nach dem globalen Maximum lokale Maxima verlassen zu können. Während bei den zuvor genannten Verfahren die Transformationsregeln einen festen Bestandteil der Verfahren bilden, erlauben regelbasierte Optimierungsstrategien diese als Parameter an das Verfahren zu übergeben. Damit erhält man eine generische Optimierung, deren Transformationsregeln ohne Änderung der eigentlichen Optimierung variiert werden können. Um diese Freiheiten nutzen zu können, müssen die Transformationsregeln jedoch in einem speziellen – der Optimierung verständlichen – Format vorliegen. Hierfür bieten die meisten regelbasierten Optimierungen, wie beispielsweise das in [GD87] vorgestellte System EXODUS[5], eine eigene Regeldefinitionssprache an. Da eine regelbasierte Optimierung im Voraus keine Informationen über die Transformationsregeln besitzt, gestaltet sich die Bestimmung ihrer Ausführungsreihenfolge als besonders schwierig. Systeme, die solche Informationen zur Laufzeit bestimmen, sind meist sehr komplex und belasten den für die Optimierung benötigten Zeitaufwand noch zusätzlich. Daher überlassen viele Systeme diese Aufgabe dem Programmierer, der dann die Möglichkeit hat, Regeln bezüglich ihrer Aufgabe zu gruppieren und diesen Gruppen eine bestimmte Ausführungsreihenfolge zuzuteilen. Die oben genannten Verfahren bilden nur einen kleinen Überblick über mögliche Ansätze zur Begrenzung des Suchraumes und sind bei weitem nicht vollständig. Eine ausführlichere Darstellung der regelbasierten Optimierung bietet Abschnitt 2.3, welcher die in XXL verfügbare Anfrageoptimierung vorstellt.

Unabhängig von dem gewählten Verfahren, bestimmt die logische Optimierung eine Menge von günstigen Anfragen, die anschließend an die physische Optimierung weitergereicht werden.

2.1.3 Physische Optimierung

Die physische Optimierung bestimmt zu einer gegebenen Anfrage den optimalen physischen Anfrageplan. Zu diesem Zweck werden den logischen Operationen nun physische Algorithmen zugeordnet, welche die gewünschten Ergebnisse mit den vorliegenden Informationen möglichst effizient berechnen. In dieser Phase werden nun auch physische Eigenschaften wie beispielsweise eine Sortierung und das Vorhandensein von Indexen auf den Daten beachtet, aber auch statistische Informationen über die Daten selbst. Obwohl inhaltlich eigentlich mehr der logischen Optimierung zuzuordnen, ist eine weitere wichtige Aufgabe der physischen Optimierung die Bestimmung der optimalen Ausführungsreihenfolge von Verbundoperationen. Da für eine sinnvolle Erfüllung dieser Aufgabe physische Informationen von großer Wichtigkeit sind, werden in der Phase der logischen Optimierung zweistellige Verbundoperationen zu n-stelligen zusammengefasst, um dann in der Phase der physischen Optimierung in einen sogenannten *Joinplan* überführt zu werden. Der Joinplan besteht nun wieder aus zweistelligen Verbundoperationen und ordnet ihnen einen physischen Algorithmus zu. Wie der Anfrageplan auch, bestimmt der Joinplan die Beziehung zwischen den einzelnen (Verbund-)Operationen und legt damit die Reihenfolge ihrer Ausführung fest. Nach dem Aufbrechen der n-stelligen Verbundoperation in $(n-1)$ zweistellige, können nun aber wiederum Trans-

[5] Abkürzung für *EXtensible Object-oriented Database System*

formationen der logischen Optimierung anwendbar sein. Dies führt zu einer leichten Vermischung von logischer und physischer Optimierung. Am Beispiel des Verbunds lässt sich auch am einfachsten die Bedeutung der physischen Eigenschaften für die Wahl des Algorithmus verdeutlichen. So kann ein Verbund über zwei Datenquellen, die über eine gemeinsame Sortierung verfügen, sehr effizient mittels eines *Sort-Merge Join* berechnet werden, während ein bestehender Index auf einer der beiden Datenquellen durch einen *Indexed Nested-Loops Join* genutzt werden kann.

Auch die physische Optimierung benötigt ein Kostenmodell, um die Qualität der erzeugten Anfragepläne zu bewerten. In einem solchen Modell können viele Größen, wie beispielsweise I/O-Kosten, Haupt- und Externspeicherverbrauch und CPU-Kosten, berücksichtigt werden. Meist werden jedoch nur die relativ teuren Externspeicherzugriffe und die CPU-Kosten abgeschätzt, da diese einen Großteil der Gesamtkosten verursachen und so das Kostenmodell nicht unnötig komplex wird. Neben den verwendeten physischen Algorithmen tragen maßgeblich auch die Größe und die Verteilung der Daten der beteiligten Datenquelle zu den zu schätzenden Kosten bei. Damit diese Informationen auch bei der Kostenschätzung von Algorithmen, die Zwischenergebnisse konsumieren, zur Verfügung stehen, enthalten die meisten Kostenmodelle auch einen Schätzer für die Größe und die Verteilung der Daten von Zwischenergebnissen. Wie auch schon bei der physischen Optimierung, ist das Problem, zu einem gegebenem Kostenmodell den physischen Anfrageplan zu bestimmen, der die Kosten minimiert, NP-hart.[6] Aus diesem Grund sind auch hier wieder effiziente Verfahren zur Einschränkung des Suchraumes notwendig. Neben den bereits in Abschnitt 2.1.2 angesprochenen Verfahren, ist hier das in [SAC+79] vorgestellte dynamische Programmieren[7] besonders erwähnenswert. Dieses Verfahren eignet sich besonders zur effizienten Berechnung von Joinplänen. Zur Berechnung des optimalen Joinplans für eine n-stellige Verbundoperation werden zunächst alle möglichen Joinpläne über zwei Datenquellen erzeugt. Anschließend werden alle Joinpläne über drei Datenquellen erzeugt, indem der jeweils günstigste Joinplan über zwei Datenquellen mit der dritten Datenquelle kombiniert wird. Dieser Vorgang wird solange fortgeführt, bis Joinpläne über n Datenquellen erreicht werden. Schließlich wird der günstigste Joinplan zurückgegeben.

Mit der Rückgabe des optimalen physischen Anfrageplans endet die physische Optimierung und damit auch der Kern der Anfrageoptimierung. Nun folgt die im Vorfeld erwähnte Übersetzung in eine ausführbare Repräsentation, mit der die Anfrageoptimierung insgesamt beendet wird.

2.2 Metadaten

In den vorhergehenden Abschnitten wurde oftmals auf Informationen Bezug genommen, welche die algebraischen Eigenschaften von Aussagen und Operationen bzw. die

[6] Schon 1984 zeigten Toshihide Ibaraki und Tiko Kameda in [IK84], dass selbst das Teilproblem, zu einer n-stelligen Verbundoperation unter ausschließlicher Verwendung von *Nested-Loops Joins* den optimalen Joinplan zu berechnen, bereits NP-hart ist.

[7] Da dieses Verfahren 1979 in System R (siehe auch [ABC+76]) implementiert wurde, ist es auch unter dem Namen System R Algorithmus bekannt.

physischen Eigenschaften von Datenquellen und Algorithmen charakterisieren. Solche Informationen werden gemeinhin als Metadaten[8] bezeichnet.

> "Metadaten sind maschinenlesbare Informationen über elektronische Ressourcen oder andere Dinge."

Mit dieser Definition verdeutlicht Tim Berners-Lee, Direktor des *World Wide Web Consortium* (W3C), die Bedeutung von Metadaten in großen Informationssystemen, in denen sie zur Beschreibung von Informationsressourcen genutzt werden. Jedoch ist nicht nur die Ressourcenbeschreibung selbst von Interesse, sondern auch die Informationsverbreitung stellt einen wichtigen Aspekt dieses Interessengebietes dar. So genügt es nicht, Dokumente mittels Metadaten zu beschreiben, auch der effiziente und kostengünstige Einsatz dieser Metadaten in elektronischen Netzen ist von großer Wichtigkeit. Einen gewissen Standardisierungsgrad vorausgesetzt, ermöglichen diese Techniken neue und einfache Erschließungsmöglichkeiten für Informationssysteme.[9] Im Kontext von Datenbanksystemen beschreiben Metadaten dahingegen meist die Struktur der in der Datenbank gespeicherten Daten und Eigenschaften der zur Speicherung dieser Daten verwendeten Datenstrukturen. Am Beispiel einer relationalen Datenbank heißt dies konkret, dass die zugehörigen Metadaten die zugrundeliegenden Relationen beschreiben. Dabei werden die Domänen der Attribute meist über einen eindeutigen Namen und den in der Datenbank verwendeten Datentyp charakterisiert. Neben den Relationen selbst können Metadaten aber auch die konkrete Instanz einer Relation beschreiben. Diese Informationen können beispielsweise die Größe der Instanz, die Zahl der unterschiedlichen Werte pro Attribut und deren Verteilung oder Sortierungen auf einzelnen Attributen beinhalten.

Verallgemeinert verbergen sich hinter dem Begriff der Metadaten also strukturierte Daten, welche reale oder virtuelle Objekte und Ereignisse beschreiben. Da ihr Ursprung in der Informatik zu finden ist, wird meist zusätzlich gefordert, dass Metadaten in einer maschinenlesbaren Form vorliegen.

2.3 Anfrageoptimierung in XXL

Die Bibliothek XXL stellt mit dem Paket `xxl.relational` eine Algebra relationaler Operatoren bereit, mit deren Hilfe Anfragen an relationale Datenbanken formuliert werden können. Diese relationale Algebra stellt eine Spezialisierung der bereits in Kapitel 1 erwähnten *cursor*-Algebra dar. Dies bedeutet unter anderem, dass alle Operatoren der relationalen Algebra *cursor* sind und damit auch die in [Gra94] vorgestellte *open-next-close* Schnittstelle implementieren. Unter Verwendung dieser Algebra können Anfragen in Form von Operatorbäumen *bottom-up* aufgebaut und an relationale

[8] Obwohl $\mu\epsilon\tau\acute{\alpha}$ im Griechischen für räumlich und zeitlich *nach* steht, verwandten die Philosophen Alfred Tarski und Rudolf Carnap 1933–34 erstmals den Begriff Metasprache in Sinne einer *Sprache über eine Sprache*. Dieser Semantik folgend beschreibt der Begriff Metadaten *Daten über Daten*.

[9] Die automatisierte Erschließung von Informationen ist auch unter dem Begriff *information retrieval* bekannt.

Datenbanken gestellt werden; die Ergebnisse der Anfragen sind wiederum direkt über die *open-next-close* Schnittstelle der Wurzel des jeweiligen Operatorbaumes zugänglich.

Da die Implementierung der relationalen Algebra jedoch nur physische Operatoren zur Verfügung stellt, birgt diese Vorgehensweise den Nachteil, dass Anwender Anfragen nur in Form von physischen Anfrageplänen erstellen können. D. h. ein Anwender muss nicht nur die Anfrage selbst als Operatorbaum formulieren, sondern auch den Operatoren die anzuwendenden Algorithmen selbständig zuordnen. Die Entscheidung, ob eine Verbundoperation als *Nested-Loops Join*, als *Hash Join* oder als *Sort-Merge Join* ausgeführt werden soll, setzt jedoch ein hohes Maß an Kenntnis von der Struktur der zugrundeliegenden Datenbank und der Organisation ihrer Daten seitens der Anwender voraus. Aus diesem Grunde stellt die Bibliothek XXL mit dem Paket `xxl.relationalLog` eine Algebra logischer Operatoren mitsamt einer Anfrageoptimierung bereit.[10] Mit deren Hilfe können Anwender Anfragen als logische Operatorbäume formulieren und müssen damit ausschließlich die algebraischen Eigenschaften der Anfragen festlegen. Die Optimierung der Anfragen – einschließlich der Wahl der optimalen physischen Algorithmen – liegt dadurch in der Verantwortung der Anfrageoptimierung. Diese Anfrageoptimierung und die verfügbare logische Algebra werden in den folgenden Abschnitten näher erläutert.

2.3.1 Logische Operatoren

Im Gegensatz zu den physischen Operatoren, welche – einmal zu einer Anfrage zusammengestellt – nur noch sehr begrenzt in ihrer Ausführungsreihenfolge umgestellt oder gegen andere Operatoren ausgetauscht werden können, bieten logische Operatoren ein höheres Maß an Flexibilität. Logische Operatoren werden durch Knoten in einer baumartigen Struktur repräsentiert.[11] Dabei repräsentiert die Wurzel des Operatorbaumes die Anfrage selbst und die Kindern eines logischen Operators dessen Eingaben. Jeder Operator kann eine beliebige aber feste[12] Anzahl an Eingaben besitzen, wobei jede Eingabe durch eine Liste von äquivalenten, logischen Operatoren gebildet wird. Somit können logisch äquivalente Teilanfragen direkt im Operatorbaum repräsentiert werden. Gleichzeitig kann jeder logische Operator mehrere Eltern besitzen, wodurch gemeinsame Teilanfragen redundanzfrei gespeichert werden können.

Zur Zeit beschränkt sich die Implementierung der logischen Algebra auf SPJ-Anfragen[13] einschließlich dem kartesischen Produkt und der Umbenennung. Jeder dieser Operatoren wird durch eine eigene Klasse repräsentiert, welche jeweils den logischen Operator erweitert. Dabei sind die notwendigen Parameter der einzelnen Operatoren in

[10] Die Anfrageoptimierung und die zugrundeliegende logische Algebra wurden im Rahmen einer Diplomarbeit entwickelt und in [Ber02] vorgestellt.

[11] Korrekterweise handelt es sich bei einem Operatorbaum um einen gerichteten azyklischen Graphen mit einem ausgezeichneten Knoten, der sogenannten Wurzel, welche keine Eltern besitzt und von der aus jeder Knoten im Graphen über eine Pfad erreicht werden kann.

[12] Die Zahl der Eingaben eines logischen Operators wird bei seiner Konstruktion festgelegt und ist während seiner Lebensdauer konstant.

[13] SPJ-Anfragen sind Anfragen, welche ausschließlich aus Selektionen, Projektionen und Verbünden (*Joins*) bestehen.

den Klassen selbst gespeichert, während der logische Operator zusätzliche Metadaten speichert, welche die Struktur der Daten beschreiben und statistische Informationen über Größe und Verteilung der Daten und Eigenschaften der zur Verfügung stehenden Algorithmen wiedergeben. Die notwendigen Parameter der einzelnen logischen Operatoren ergeben sich aus der relationalen Algebra und sind für die Selektion die Selektionsbedingung, für die Projektion die zu projizierenden Attribute, für die Umbenennung die umzubenennenden Attribute und ihre neuen Bezeichnungen und für den Verbund die Verbundbedingung. Grundsätzlich betrachtet die logische Algebra nur allgemeine Verbünde (*Theta-Join*), bei denen nur solche Tupel ein Ergebnis bilden, welche eine explizit anzugebende Verbundbedingung erfüllen. Für den natürlichen Verbund (*Natural Join*)[14] und den *Equi-Join*[15] werden die Verbundbedingungen implizit aus den gleichbenannten bzw. angegebenen Attributen der Eingaberelationen erzeugt. Die Metadaten der logischen Operatoren werden im folgenden Abschnitt betrachtet.

2.3.1.1 Metadaten

Java's Bibliothek stellt mit dem Paket `java.sql` bereits Werkzeuge zur Verfügung, mit deren Hilfe SQL-Anfragen an eine relationale Datenbank gestellt werden können. Das Ergebnis einer solchen Anfrage wird dem Anwender als Instanz einer – durch die zugrundeliegenden Anfrage definierten – Relation zugänglich gemacht. Zur Beschreibung dieser Relation dient hierbei die Schnittstelle `ResultSetMetaData`, welche Informationen über die Typen und Eigenschaften der Attribute einer solchen Ausgaberelation bereitstellt. Im Paket `xxl.relationalLog` wird diese Schnittstelle erweitert, um die Metadaten logischer Operatoren zu repräsentieren. Dabei lassen sich logische Operatoren auf verschiedenen Abstraktionsebenen betrachten. Die Schnittstelle `OperatorOutputMetaData` beschreibt allein die Ausgaberelation eines logischen Operators und ergänzt diese Informationen durch einige Eigenschaften der konkreten Instanz. So sind über diese Schnittstelle neben der Größe der Instanz, der Zahl der unterschiedlichen Werte pro Attribut und Informationen über eine eventuell vorhandene Sortierung bezüglich einem oder mehreren Attributen auch die Kosten für die Berechnung der konkreten Instanz verfügbar. Verfügt der logische Operator zusätzlich über Eingaberelationen, so kann mit Hilfe der Schnittstelle `OperatorInputMetaData` direkt auf deren Metadaten zugegriffen werden.

Neben den Metadaten der logischen Operatoren, benötigt eine Anfrageoptimierung jedoch noch weitere Informationen. Um beispielsweise eine Aggregation oder eine Abbildung[16] optimieren zu können, werden Informationen darüber benötigt, welche Attribute

[14]Der natürliche Verbund liefert genau die Tupel als Ergebnis, welche in allen gleichbenannten Attributen übereinstimmen. Das Ergebnistupel besteht dabei aus allen Attributen der am Ergebnis beteiligten Tupel, wobei die gleichbenannten Attribute jedoch nur einmalig vorkommen.

[15]Beim *Equi-Join* bilden die Tupel ein Ergebnis, welche in bestimmten, explizit angegebenen Attributen übereinstimmen. Dabei werden, wie auch beim *Theta-Join*, alle Attribute der am Ergebnis beteiligten Tupel ins Ergebnistupel aufgenommen. Darum dürfen die am *Join* beteiligten Eingaberelationen keine gleichbenannten Attribute besitzen.

[16]Eine Abbildung oder *map* ist ein Operator, der mittels einer Abbildungsfunktion Tupel der Eingaberelation auf Tupel der Ausgaberelation abbildet. Die bekannten Operatoren Projektion und Umbenennung bilden damit Spezialfälle der Abbildung.

der Eingaberelation die zugehörige Aggregations- oder Abbildungsfunktion verwendet. Neben diesen Informationen, gibt die Klasse `FunctionMetaData` zusätzlich darüber Auskunft, ob die Ausführung der betrachteten Funktion rechenintensiv ist. Dies hat besonderen Einfluss auf die Anfrageoptimierung, da *teure* Operatoren möglichst weit oben im Operatorbaum platziert und damit möglichst selten ausgeführt werden sollen. Eine Erweiterung dieser Metadaten stellt die Klasse `PredicateMetaData` im Paket `xxl.relationalLog.predicates` dar. Neben den Metadaten von Funktionen, stellt sie zur Beschreibung von Selektionsbedingungen, Informationen über deren Selektivität bereit.

Schließlich bietet das Paket `xxl.relationalLog.physical` mit der Klasse `AlgorithmMetaData` noch eine Möglichkeit, mit deren Hilfe logischen Operatoren physische Algorithmen zugeordnet werden können. Neben der Identifizierung des zugrundeliegenden Algorithmus, bietet die Klasse Informationen darüber, ob der Algorithmus eine Sortierung auf einer Instanz der Eingaberelation erhält und über welche Sortierungen die Instanz der Ausgaberelation nach seiner Ausführung verfügt.

2.3.2 Normalisierung

Mit der im vorherigen Abschnitt vorgestellten Algebra logischer Operatoren steht die interne Repräsentation von Anfragen für die Optimierung fest. Nun folgen die einzelnen Phasen der Anfrageoptimierung. Dabei wird zunächst auf eine explizite Normalisierung der Anfrage verzichtet. Stattdessen beschränkt man die Menge der zulässigen Anfragen auf einfache SPJ-Anfragen ohne Unteranfragen und erlaubt nur Selektionsbedingungen in konjunktiver Normalform, welche ausschließlich die Attribute der Eingaberelationen miteinander oder mit Konstanten vergleichen. Eine in Rahmen der Normalisierung durchgeführte Vereinfachung oder Verbesserung der Anfrage findet ebenfalls nicht statt.

2.3.3 Logische Optimierung

Als nächste Schritt in der Optimierung der Anfrage folgt die Phase der logischen Optimierung. Um ein Höchstmaß an Flexibilität zu garantieren, steht für diese Aufgabe im Paket `xxl.relationalLog.ruleOptimizer` eine regelbasierte Anfrageoptimierung bereit. Mit deren Hilfe können die Transformationsregeln der logischen Optimierung frei gewählt und somit Optimierungsstrategien ausgetauscht oder neue logische Operatoren einfach integriert werden. Im Gegensatz zu anderen Implementierungen einer regelbasierten Anfrageoptimierung, benötigt die in XXL implementierte Variante keine zusätzliche Regeldefinitionssprache. Stattdessen werden Transformationsregeln direkt in Java definiert und sind sofort ausführbar. Desweiteren verzichtet die vorliegende Implementierung auf eine Bestimmung der optimalen Ausführungsreihenfolge für Transformationsregeln und überlässt diese Aufgabe dem Anwender. Dieser hat dabei die Möglichkeit, Regeln, welche einer gemeinsamen Zielsetzung folgen, zu Regelgruppen zusammenzufassen und sowohl die Ausführungsreihenfolge von Regeln in einer Regelgruppe als auch der einzelnen Regelgruppen eigenhändig zu bestimmen. Die Umsetzung von Regeln und Regelgruppen sowie die implementierte Ausführungsstrategie werden

in den folgenden Abschnitten erläutert.

2.3.3.1 Regeln

Eine regelbasierte Anfrageoptimierung benötigt als Grundlage für die Optimierung von Anfragen eine Menge von Transformationsregeln. Diese Regeln sind – im Gegensatz zu anderen Optimierungsverfahren – kein fester Bestandteil des Optimierungsalgorithmus und damit diesem auch nicht vorab bekannt. Aus diesem Grund müssen solche Regeln neben der eigentlichen Transformation zusätzliche Informationen darüber beinhalten, auf welche Anfragen sie anwendbar sind. Die Anwendbarkeit solcher Regeln hängt nicht nur allein vom Vorhandensein einer bestimmten Folge von Operatoren als Ausgangspunkt für die Transformation ab – oft wird sie noch durch zusätzliche Bedingungen eingeschränkt. So ist beispielsweise eine Transformation, welche eine Projektion von der Ausgabe einer Selektion in deren Eingabe verschiebt, nicht bereits beim Vorhandensein der beiden Operatoren in korrekter Reihenfolge anwendbar. Erst die zusätzliche Bedingung, dass die Projektion alle Attribute der Eingaberelation erhalten muss, welche von der Selektionsbedingung benötigt werden, vervollständigt die notwendige Bedingung für die Anwendbarkeit der Transformation. Mit der Klasse `Rule` stellt die Bibliothek XXL eine abstrakte Implementierung einer allgemeinen Regel zur Verfügung. Um nun konkrete Regeln zu erzeugen, genügt es diese abstrakte Klasse um eine Bedingung, welche prüft, ob die Regel auf eine bestimmte Folge von Operatoren anwendbar ist, und eine Transformationsvorschrift zu ergänzen. Ein Sammlung häufig verwendeter, vordefinierter Regeln findet sich in der Klasse `Rules`. Wie sich diese Regeln zu Regelgruppen zusammenfassen lassen, wird im folgenden Abschnitt erläutert.

2.3.3.2 Regelgruppen

Häufig ist mit der Bestimmung einer Ausführungsreihenfolge auf den Regeln einer regelbasierten Anfrageoptimierung ein nicht unerheblicher Aufwand verbunden. Die automatische Bestimmung der optimalen Ausführungsreihenfolge stellt selbst wieder ein Optimierungsproblem dar und belastet den für die Optimierung benötigten Zeitaufwand noch zusätzlich. Aus diesem Grund wird diese Aufgabe oft dem Anwender übertragen. Da jedoch die Menge der Regeln einer solchen Optimierung gewöhnlich recht umfangreich und eine optimale Ausführungsreihenfolge meist nicht offensichtlich ist, werden auch Anwender häufig durch diese Aufgabe überfordert. Um dies zu verhindern, gruppiert man Regeln mit einer gemeinsamen Zielsetzung zu einer gemeinsamen Regelgruppe.[17] So können beispielsweise alle Regeln, welche eine Selektion aus der Ausgabe eines beliebigen Operators in dessen Eingabe oder Eingaben verschiebt, zu einer gemeinsamen Regelgruppe *"Selektionen frühzeitig ausführen"* zusammengefasst werden. Die Regeln innerhalb einer solchen Gruppe benötigen keine spezielle Ausführungsreihenfolge, so dass eine solche nur noch für die einzelnen Regelgruppen zu spezifizieren ist.

[17] Bei der Gruppierung von Regel ist zu beachten, dass jede Regel einer Regelgruppe den logischen Operatorbaum in die gleiche Richtung durchläuft. Eine Regelgruppe, welche das Ziel verfolgt, einen bestimmten Operator frühzeitig auszuführen, durchläuft den logischen Operatorbaum *top-down*, während andere ihn *bottom-up* durchlaufen.

Für diese Aufgabe bietet die Klasse `RuleGroup` alle notwendigen Eigenschaften. Neben der Gruppierung von Regeln, lässt sich mit Hilfe dieser Klasse auch eine Ordnung auf den Regelgruppen definieren. Existieren regelübergreifende Bedingungen, welche stets erfüllt sein müssen, damit eine beliebige Regel der Regelgruppe ausgeführt werden kann, so lassen sind diese Bedingungen der gesamten Regelgruppe zugeordnen. Neben der eigentlichen Regelgruppe, bietet die Bibliothek XXL mit der Klasse `RuleGroups` eine Auswahl gebräuchlicher Regelgruppen an. Der nun folgende Abschnitt beschreibt schließlich, wie die regelbasierte Anfrageoptimierung die vom Anwender spezifizierte Ausführungsreihenfolge auf den Regelgruppen realisiert.

2.3.3.3 Regelbasierte Anfrageoptimierung

Mit den im vorhergehenden Abschnitt beschriebenen Regelgruppen bietet sich dem Anwender die Möglichkeit, eine Ausführungsreihenfolge auf ihnen festzulegen. Der Algorithmus 2.1 zeigt in leicht vereinfachter Weise[18], wie diese Ausführungsreihenfolge in die Optimierung von Anfragen integriert wird. Um einen logischen Operatorbaum zu optimieren, durchläuft der Algorithmus eine übergebene Folge von Regelgruppen. Erfordert eine Regelgruppe einen *bottom-up* Durchlauf durch den Operatorbaum, so müssen zunächst die Teilbäume der Wurzel des Operatorbaumes optimiert werden. Anschließend werden die Transformationsregeln der Regelgruppe so lange auf den Operatorbaum angewandt, bis keine dieser Regeln mehr auf die Struktur des Operatorbaumes passt. Bevor schließlich die nächste Regelgruppe verarbeitet wird, wird noch geprüft, ob die aktuelle Regelgruppe einen *top-down* Durchlauf des Operatorbaumes erfordert und eventuell die Teilbäume der Wurzel des Operatorbaumes optimiert. Nachdem alle Regelgruppen durchlaufen wurden, liegt der logische Operatorbaum in optimierter Form vor. Den vorgestellten Algorithmus implementiert die Klasse `AlgebraicRuleOptimizer`, welche zusätzlich eine regelbasierte Anfrageoptimierung mit Vorbelegung der Regelgruppen zur Verfügung stellt. Diese fertig konfigurierte logische Optimierung führt zunächst Selektionen spät aus, um anschließend natürliche Verbünde aufzulösen und schließlich Projektionen spät auszuführen. Danach werden erst Umbenennungen und dann Selektionen frühzeitig ausgeführt, um daraufhin Verbünde zusammenzufassen und abschließend Projektionen frühzeitig auszuführen. Mit der eigentlichen regelbasierten Anfrageoptimierung stehen nun alle Werkzeuge zur logischen Optimierung von Anfragen bereit.

2.3.4 Physische Optimierung

Den Abschluss der Anfrageoptimierung bildet die physische Optimierung, welche sich in dem Paket `xxl.relationalLog.physical` in Form der Klasse `PhysicalOptimizer` befindet. Ihre Aufgabe besteht darin, den logischen Operatoren unter Einbeziehung der physischen Eigenschaften ihrer Eingaberelationen den optimalen Algorithmus zu-

[18] Der komplette Algorithmus bietet zusätzlich die Möglichkeit, Teilbäume der Anfrage, welchen durch die Anwendung von Regeln neue Operatoren hinzugefügt wurden, nochmals mit bereits verarbeiteten Regelgruppen zu optimieren.

Algorithm 2.1 Regelbasiert Anfrageoptimierung

IN	: $(G_n)_{1 \leq n \leq N}$	Folge von N Regelgruppen
	Q	logischer Operatorbaum
OUT	: Q_{opt}	optimierter logischer Operatorbaum

```
1: for n ← 1 to N do
2:     if group G_n contains bottom-up transformations then
3:         for all subqueries Q_sub such that Q_sub is input of Q's root operator do
4:             optimize Q_sub
5:         end for
6:     end if
7:     changed ← false
8:     repeat
9:         for all rules R ∈ G_n do
10:            if R fits Q then
11:                execute R on Q
12:                changed ← true
13:            end if
14:        end for
15:    until ¬changed
16:    if group G_n performs top-down transformations then
17:        for all subqueries Q_sub such that Q_sub is input of Q's root operator do
18:            optimize Q_sub
19:        end for
20:    end if
21: end for
22: Q_opt ← Q
```

zuordnen. Um hierbei auch die Erweiterbarkeit der physischen Optimierung garantieren zu können, werden alle Informationen, welche mit der eigentlichen Optimierung eines logischen Operators in Verbindung stehen, aus dem Optimierungsalgorithmus herausgezogen und in separate Funktionen gekapselt. Diese Funktionen können einfach ausgetauscht und ergänzt werden, um die Optimierungsstrategie für einzelne Operatoren zu ändern oder neue Operatoren zu unterstützen. Der verbleibende Optimierungsalgorithmus hat nun nur noch die Aufgabe, den zu optimierenden Operatorbaum *bottom-up*[19] zu durchlaufen, für jeden Operator die entsprechende Optimierungsfunktion herauszusuchen und auf diesen anzuwenden. Aus den im Laufe dieser Optimierung entstandenen physischen Anfrageplänen wird abschließend derjenige ausgewählt und als Ergebnis der Anfrageoptimierung zurückgeliefert, welcher bezüglich eines gegebenen Kostenmodells die geringsten Kosten verursacht. In den nun folgenden Abschnitten werden Optimie-

[19]Der logische Operatorbaum wird in der physischen Optimierung *bottom-up* durchlaufen. Damit wird garantiert, dass wann immer ein logischer Operator physisch optimiert werden soll, seine Eingaberelationen bereits optimiert vorliegen.

rungsfunktionen und das der physischen Optimierung zugrundeliegende Kostenmodell näher beleuchtet.

2.3.4.1 Optimierungsfunktionen

Die Klasse `AlgorithmOptimizeFunctions` stellt eine Reihe Funktionen für die Optimierung gebräuchlicher logischer Operatoren bereit. Die Aufgabe einer solchen Funktion besteht darin, einem gegebenen Operator und dessen bereits physisch optimierten Eingaberelationen, alle möglichen physischen Implementierungen zuzuordnen. Dabei übernimmt die Klasse `Algorithms` die Funktion einer zentralen Instanz, bei der physische Implementierungen von logischen Operatoren registriert werden, um diese im Laufe der physischen Optimierung abfragen zu können. Da für einen logischen Operator meist mehrere verschiedene Implementierungen zur Verfügung stehen, werden die erzeugten Anfragepläne anschließend in der Klasse `QuerySortingGrouper` verwaltet. Die Aufgabe dieser Klasse besteht darin, alle physischen Anfragepläne zu sammeln und nach der Sortierung ihrer Ausgabe zu gruppieren. Damit können äquivalente Anfragepläne erkannt und aussortiert werden, so dass die physischen Eigenschaften der verschiedenen Implementierungen einer Eingaberelation anderen Optimierungsfunktionen zur Verfügung stehen. Besonders erwähnenswert ist hierbei die Optimierungsfunktion für Verbünde. Da die logische Optimierung alle Verbünde der Anfrage zu *Theta-Joins* transformiert und weitestgehend zusammenfasst, muss die physische Optimierung aus einem n-stelligen Verbund wieder einen Joinplan binärer Verbünde erzeugen. Dies geschieht durch dynamisches Programmieren, welches bereits in Anschnitt 2.1.3 vorgestellt wurde. Dabei wird auf eine zusätzliche logische Optimierung der neu erzeugten Operatoren verzichtet.

2.3.4.2 Kostenmodell

Die Grundlage der physischen Optimierung bildet ein Kostenmodell, mit dessen Hilfe die zu erwartenden Kosten eines Algorithmus anhand der physischen Eigenschaften der Eingaberelationen möglichst realistisch abgeschätzt werden können. Zu diesem Zweck stellt das Paket `xxl.relationalLog.optimizer` die Klasse `CostModel` bereit. Hier können für physische Algorithmen Kostenfunktionen zentral registriert und jederzeit wieder angefordert werden, um die Kosten eines Algorithmus zu bestimmen. Als Grundlage für die Kostenschätzung dienen dabei sowohl die Größen der Ein- und Ausgaberelationen als auch die Verteilung der Werte auf den einzelnen Attributen. Da diese Informationen nur für die in der Datenbank gespeicherten Relationen vorliegen, wird im gleichen Paket mit der Klasse `SDEstimator` ein Schätzer für diese Werte bereitgestellt. Auch diese Klasse stellt wiederum nur eine zentrale Instanz dar, bei der für die logischen Operatoren Funktionen, welche die benötigten Werte abschätzen, registriert werden können. Hierbei ist zu beachten, dass diese Funktionen logischen Operatoren und nicht Algorithmen zugeordnet werden, da die Größe der Ergebnismenge und die Verteilung der Werte auf den einzelnen Attributen unabhängig von den gewählten Algorithmen sind. Nachdem mit Hilfe des Kostenmodells die zu erwartenden Kosten aller Operatoren eines Anfrageplans bestimmt wurden, werden diese zu seinen Gesamtkos-

ten aufsummiert, um anhand dieser Wertes die verschiedenen Anfragepläne bewerten zu können.

2.4 Zusammenfassung

Im Zuge dieses Kapitels wurde gezeigt, dass die Anfrageoptimierung eine zentrale Position in der Verarbeitung von Anfragen innehat und welche Aufgaben mit ihr verbunden sind. Die Anfrageoptimierung selbst gliedert sich in die Normalisierung, die logische Optimierung und die physische Optimierung. Da die logische und physische Optimierung jeweils NP-harte Problemstellungen darstellen, versuchen viele Optimierungsansätze den Suchraum für eine mögliche Lösung einzuschränken. Obwohl als Folge aus der Verwendung eines solchen Ansatzes nicht mehr garantiert werden kann, dass dieser die optimale Lösung findet, rechtfertigt die Verringerung der Laufzeit eines solchen Ansatzes die geringfügige Verschlechterung seiner Ergebnisse in den meisten Fällen.

Das Paket `xxl.relationalLog.optimizer` stellt mit der Klasse `Planer` eine komplette Anfrageoptimierung bereit, welche mit Hilfe einer Algebra logischer Operatoren formulierte Anfragen optimiert. Die Anfrageoptimierung setzt sich aus einer logischen Optimierung in Form einer regelbasierten Optimierung und einer dynamischen Optimierung, welche unter anderem dynamisches Programmieren einsetzt, zusammen. Dabei bildet die Erweiterbarkeit der Anfrageoptimierung ein wichtiges Merkmal, da sich sowohl die unterstützten logischen Operatoren und Algorithmen als auch die Optimierungsstrategie ohne Änderung der zugrundeliegenden Schnittstellen und Klassen einfach austauschen oder erweitern lassen.

Kapitel 3

Aktive Datenquellen und Datenströme

Obwohl aktive Datenquellen schon seit längerem bekannt sind, werden erst seit kurzem adäquate Techniken zur Verarbeitung der gelieferten Daten untersucht. Stattdessen sehen gängige Verfahren die Speicherung dieser Daten in großen Datenbanksystemen vor, um sie mit Hilfe wohlbekannter und hochentwickelter Techniken aus dem Bereich der Datenbankforschung auszuwerten. Die stetig ansteigende Flut von Informationen, die täglich aufs Neue über unsere heutige Gesellschaft hereinbricht, führt diese Vorgehensweise jedoch trotz der steigenden Leistungsfähigkeit und Speicherkapazität von Datenbanksystemen[1] unweigerlich an ihre Grenzen, so dass nun der Ruf nach neuen Methoden zur Informationsverarbeitung laut wird. Aus diesem Grund stellt die Bibliothek XXL eine Algebra für aktive Datenquellen bereit, mit deren Hilfe Anfragen an aktive Datenquellen direkt formuliert und ausgeführt werden können.

Um einen Überblick über die Verarbeitung von Anfragen an aktive Datenquellen bieten zu können, stellt dieses Kapitel zunächst aktive Datenquellen vor und erläutert ihre Eigenschaften. Anschließend wendet es sich den Besonderheiten der von ihnen ausgehenden Datenströme zu und erläutert Unterschiede zu der herkömmlichen Verarbeitung von Anfragen. Im Anschluss an diese theoretischen Grundlagen wird die in XXL enthaltene Algebra für aktive Datenquellen näher betrachtet. Dabei gilt der einfachen Erweiterbarkeit und dem breiten Spektrum an Anwendungsmöglichkeiten ein besonderes Augenmerk.

3.1 Aktive Datenquellen

Datenquellen, welche ihre Daten selbständig versenden, werden als aktive Datenquellen bezeichnet. Im Gegensatz zu ihren passiven Geschwistern, wie beispielsweise die

[1] Gordon Moore erkannte 1965, dass sich die Transistorendichte integrierter Schaltkreis alle 18 Monate verdoppelt und schloss daraus, dass damit die Rechenleistung von Computern exponentiell wachsen muss. Diese Regelmäßigkeit ist seitdem als *Moore's Law* bekannt und trotz technischer Schwierigkeiten bis heute gültig.

Instanz einer Relation in einer Datenbank oder eine Datei in einem Dateisystem, werden ihre Daten nicht als Antwort auf eine explizite Anfrage hin bereitgestellt. Vielmehr versenden aktive Datenquelle ihre Daten, sobald diese vorliegen, an interessierte Empfänger. Solche Datenquellen können in vielerlei Gestalt auftreten – beispielsweise ein Sensor, welcher Ereignisse registriert oder in festen Zeitabständen Messwerte bestimmt und diese Daten an eine zentrale Messstation sendet, oder ein Software-Agent, welcher ein Netzwerk nach bestimmten Informationen durchsucht und Ergebnisse umgehend an seinen Auftraggeber weiterleitet. Allen diesen Datenquellen ist gemeinsam, dass ihre Daten nicht in persistenter Form vorliegen. D. h. nach dem erstmaligen Versand eines Datenelements wird dieses verworfen und steht für zukünftige Anfragen grundsätzlich nicht mehr zur Verfügung. Aus diesem Grund können mit Hilfe von aktiven Datenquellen ausschließlich kontinuierliche Anfragen[2] beantwortet werden. Diese Form der Anfrage stellt auch den Schlüssel zum aktiven Versand der Daten dar. Um zu verhindern, dass aktive Datenquellen ihre Daten in der Hoffnung, dass sie einen interessierten Empfänger erreichen, wahllos verbreiten, müssen solche Empfänger sich mittels kontinuierlicher Anfragen bei ihnen anmelden. Dadurch wird erzielt, dass die Datenquelle jederzeit weiß, an welche Empfänger sie ihre Daten versenden muss. Solche *publish/subscribe*-Mechanismen benötigen als Grundlage jedoch wiederum Metadaten[3], mit deren Hilfe Datenquellen Informationen über sich und ihre Daten öffentlich zugänglich machen können. Neben Informationen über die Struktur der gelieferten Daten, können diese Metadaten auch eine Beschreibung der aktiven Datenquelle, Zeitangaben für den Versand der Daten oder die mittlere Ausgaberate[4] der Datenquelle bereitstellen. Mit Hilfe solcher Metadaten können Netzwerke nach aktiven Datenquellen, welche bestimmte Eigenschaften innehaben, durchsucht werden, ohne dass diese im Voraus bekannt sind.

3.2 Datenströme

Aktive Datenquellen produzieren eine kontinuierliche Folge von Datenelementen, welche man auch Datenstrom nennt.[5] Meldet sich nun ein interessierter Empfänger bei einer solchen Datenquelle an, so hat dies zur Folge, das die Datenquelle fortan ihre Daten nicht nur an die bereits registrierten Empfänger, sondern zusätzlich auch an den neu hinzugekommenen versendet. Damit teilt sich der Datenstrom in den bereits bestehenden Teil und einen Nebenarm, welcher zu dem neuen Empfänger strömt. Aus diesem Grund wird ein solcher Empfänger auch Datensenke genannt. Da eine aktive

[2] In [BBD+02] werden historischen und kontinuierlichen Anfragen unterschieden, wobei historische Anfragen mit einem bereits vorliegenden Datenbestand und kontinuierliche Anfragen mit den Daten, welche ab einem bestimmten Zeitpunkt eintreffen, beantwortet werden. D. h. historische Anfragen beziehen sich auf Daten, welche in der Vergangenheit erzeugt wurden, während kontinuierliche Anfragen sich auf zukünftige Daten beziehen.

[3] Eine nähere Betrachtung von Metadaten und den Möglichkeiten ihrer Anwendung bietet Abschnitt 2.2.

[4] Unter der Ausgaberate einer Datenquelle versteht man die Anzahl der pro Zeiteinheit gesendeten Datenelemente.

[5] Einige Quellen verwenden den Begriff Datenfluss synonym zu dem hier verwendeten Datenstrom.

Datenquelle kontinuierlich Daten produziert und versendet und ihre Lebensdauer im Grunde einzig durch die Lebensdauer der zugrundeliegenden Hardware oder Software bestimmt wird, ist der von ihr ausgehende Datenstrom in seiner Größe potentiell unbeschränkt. Dies hat allerdings zur Folge, dass Datenströme zur Beantwortung von Anfragen nicht komplett konsumiert werden können und damit Anfragen, welche dies erfordern, nicht exakt beantwortet werden können. Beispielsweise muss ein Verbund zweier Datenströme beide Datenströme komplett konsumieren, um das exakte Ergebnis zu ermitteln. Da dies bei potentiell unbeschränkten Datenströmen schon allein aus Gründen des verfügbaren Speichers nicht möglich ist, kann das Ergebnis des Verbunds nur näherungsweise berechnet werden, indem man den Verbund auf Ausschnitte der Datenströme beschränkt. Ein weiteres Problem stellt die Ausgaberate einer aktiven Datenquelle dar, welche sehr starken Schwankungen unterworfen sein kann. Betrachtet man beispielsweise eine Induktionsschleife in einer Straße, so erkennt man schnell, dass diese nachts und in den frühen Morgenstunden nur wenige Daten aussenden wird, während zu den Hauptverkehrszeiten die Datenrate rapide ansteigt. Für aufwendig zu berechnende Anfragen bedeutet dies, dass bei hohen Datenraten die Zeit zwischen den Eintreffen zweier Datenelemente nicht mehr für die notwendigen Berechnungen ausreicht. Um eine solche Überlastung der Anfrage zu vermeiden, müssen Vorkehrungen wie beispielsweise ein Senken der Datenrate oder eine Erhöhung der Systemressourcen getroffen werden.

3.2.1 Datenstrommanagementsysteme

Datenstrommanagementsysteme stellen eine Analogie zu Datenbankmanagementsystemen für Anfragen an aktive Datenquellen dar. Mit ihrer Hilfe ist es möglich, kontinuierliche Anfragen an aktive Datenquellen zu stellen und diese dann vom System überwachen und verwalten zu lassen. Da kontinuierliche Anfragen gewöhnlich von langer Lebensdauer sind, ist die Verwaltung von parallelen Anfragen und die Verteilung der zur Verfügung stehenden Ressourcen auch die Hauptaufgabe solcher Systeme. Diese Aufgabe setzt sich aus vielen verschiedenen Aspekten zusammen, von denen einige hier näher erläutert werden sollen.

3.2.1.1 Ressourcenverwaltung

Eine anfragenübergreifende effiziente Verwaltung von Ressourcen zählt zu den wichtigsten Aufgaben solcher Systeme. Nicht nur das die vorhandenen Ressourcen auf eine hohe Zahl gleichzeitig aktiver Anfragen verteilt werden müssen – bei einem solchen System werden auch fortwährend Anfragen an- und abgemeldet, so dass eine kontinuierliche und dynamische Umverteilung von Ressourcen notwendig wird. Bei der Verteilung des Speichers treten dabei besondere Probleme auf, da bereits sehr einfache Anfragen, aufgrund der potentiell unbeschränkten Größe ihrer Eingabedatenströme, einen potentiell unbeschränkten Speicherbedarf haben. Da diese Forderung technisch jedoch nicht umgesetzt werden kann, werden für die Auswertung von Anfragen Verfahren benötigt, deren Speicherbedarf begrenzt ist. Die Verwendung solcher Verfahren hat

jedoch zur Folge, dass viele Anfragen nicht mehr exakt beantwortet werden können. Bereits die Auswertung eines kartesischen Produkts zweier Datenströme verursacht einen unbegrenzten Speicherbedarf. Wird der zur Speicherung der Datenströme verfügbare Hauptspeicher nun begrenzt, so lassen sich im Allgemeinen nur noch Teile der Datenströme speichern und zur Berechnung des kartesischen Produktes verwenden. Dadurch kann das Ergebnis der Anfrage nur noch näherungsweise bestimmt werden und jede weitere Reduktion des verfügbaren Hauptspeichers führt zu einer erneuten Verschlechterung der Approximation. Die Verwendung von Externspeicher, um einen Großteil der Datenströme zu speichern und damit die Approximationsgüte zu verbessern, verspricht in diesem Fall wenig Aussicht auf Erfolg. Hierbei steht zu befürchten, dass die Verarbeitungszeit größer als die Zeit zwischen dem Eintreffen zweier Datenelemente ist. Dies würde wiederum die Notwendigkeit einer Pufferung der Eingabedatenströme und damit zusätzlichen Speicheraufwand bedeuten. Letztendlich gilt es, die Qualität der Ergebnisse einer Anfrage gegen den damit verbundenen Speicheraufwand abzuwägen, um eine möglichst gute Qualität für alle Anfragen des Systems bieten zu können.

3.2.1.2 Ablaufsteuerung

Die Verwaltung von CPU-Zeit, welche auch häufig als Ablaufsteuerung (*scheduling*) bezeichnet wird, stellt nur eine weitere Form der Ressourcenverwaltung dar. Sie ist jedoch für die Verwaltung parallel ablaufender Anfragen von besonderem Interesse, da hier neben der großen Anzahl von gleichzeitig ablaufenden Anfragen auch Abhängigkeiten innerhalb der Anfragen und Qualitätsanforderungen an die Ausführung von Anfragen berücksichtigt werden müssen. So ist es beispielsweise unnötig, der Auswertung einer Teilanfrage, welche auf Eingaben wartet, CPU-Zeit zuzuteilen, bevor ihre Eingaben vorliegen. Auch sollten Anfragen, deren Ergebnisse besonders schnell vorliegen müssen, bei der Vergabe von CPU-Zeit bevorzugt behandelt werden, um so den Ansprüchen zeitkritischer Anwendungen zu genügen. Im Zusammenhang mit der Ablaufsteuerung von Anfragen auf aktiven Datenquellen verdeutlicht [CÇC+02] zwei interessante Eigenschaften. Einerseits führt eine Verarbeitung der Datenelemente, bei der die Elemente direkt bei Eintreffen verarbeitet und weitergeleitet werden, dazu, dass diese Elemente zwischen Operationen der Anfrage nicht zwischengespeichert und bei der weiteren Verarbeitung wieder geladen werden müssen. Andererseits können durch das Sammeln von Datenelementen, um diese anschließend gemeinsam durch eine Operation der Anfrage verarbeiten zu lassen, teure Umschaltvorgänge zwischen Operationen, wenn diese in unterschiedlichen Prozessen ablaufen, vermieden werden. Beide Ansätze zielen auf eine Verminderung der CPU-Kosten bei der Auswertung von Anfragen ab, wobei der zweite Ansatz davon ausgeht, dass viele Operationen in eigenen Prozessen ablaufen.

3.2.1.3 Approximation

Wie bereits erwähnt, hat in vielen Fällen auch die effizienteste Ressourcenverwaltung ihre Grenzen, so dass kein exaktes Ergebnis bestimmt werden kann, weil nicht genügend Speicher zur Verfügung steht oder die Datenraten so stark steigen, dass die Zeit

zwischen den Eintreffen zweier Datenelemente nicht mehr ausreicht, um notwendige Berechnungen auszuführen. In diesen Fällen ist es notwendig, das exakte Ergebnis durch eine Approximation anzunähern. Für diese Aufgabe stehen verschiedene Techniken zur Verfügung.

- Im Allgemeinen ist die Speicherung eines Datenstroms aufgrund seiner potentiellen Unbeschränktheit nicht möglich. Daher werden häufig sogenannte gleitende Fenster (*sliding windows*) eingesetzt, um Ausschnitte des Datenstroms zu speichern. Das gleitende Fenster wird dabei durch eine Bedingung definiert, welche die Größe des Fensters reguliert. Trifft ein neues Datenelement aus den Datenstrom ein, so wird es in das Fenster eingefügt und anschließend werden alle Elemente, welche die Bedingung nicht erfüllen, aus dem Fenster entfernt, um den Speicherbedarf des Fensters einzuschränken. Häufig werden solche Fenster als Zeitfenster realisiert, welche die Elemente einer gewissen Zeitspanne speichern. Dies erfordert jedoch, dass die vom Datenstrom gelieferten Datenelement über einen Zeitstempel verfügen. Dieser kann explizit Teil des Datenelements sein oder ihm implizit vom System zugewiesen werden. Jedoch besitzen auch Zeitfenster eine potentiell unbeschränkte Größe, so dass bei hohen Datenraten die zu speichernde Zeitspanne durch das Datenbankmanagementsystem angepasst werden muss.

- Die Stapelverarbeitung (*batch processing*) von Datenelementen eines Datenstroms stellt eine Möglichkeit dar, mit hohen Datenraten umzugehen. Zu diesem Zweck werden ankommende Datenelemente nicht sofort verarbeitet, sondern gepuffert, um diese anschließend gemeinsam zu verarbeiten. Mit diesem Mittel lassen sich Operationen entlasten und es besteht weiterhin die Möglichkeit günstigere Operationen zu verwenden, welche Mengen von Elementen verarbeiten. Jedoch ist mit diesem Verfahren eine deutliche Verzögerung zwischen dem Eintreffen und der Verarbeitung eines Datenelements verbunden, so dass es für zeitkritische Anwendungen ungeeignet scheint.

- Bei einem weiteren Anwachsen der Datenrate oder einer gleichbleibend hohen Datenraten, verspricht auch die Pufferung von Datenelementen keine Aussicht auf Erfolg. Hier bleibt meist nur die Möglichkeit, den Datenstrom selbst zu reduzieren. Dies kann einerseits durch willkürliches Löschen (*load shedding*) von Elementen erfolgen. Andererseits erlaubt das Ziehen einer zufälligen Stichprobe (*random sampling*) Aussagen über die statistische Qualität der Ergebnisse zu treffen.

- Sogenannte Synopsen dienen dem Zweck, Zusammenfassungen von Datenströmen bereit zu stellen. Mit ihrer Hilfe ist es möglich, sowohl kontinuierliche als auch historische Anfragen auf Datenströmen zu beantworten. Um die Gefahr eines unbeschränkten Anwachsens der Synopse zu verhindern, werden in diesem Bereich häufig verlustbehaftete Kompressionstechniken die einem konstanten Speicherplatz genügen eingesetzt. Eine zusätzliche Begrenzung von Synopsen auf Zeitfenster kann weiterhin dazu beitragen, den notwendigen Verwaltungsaufwand zu reduzieren.

3.2.1.4 Anfrageoptimierung

Als letzte, aber gewiss nicht geringste Aufgabe soll an dieser Stelle die Optimierung von Anfragen genannt werden. Da Datenströme meist nur wenige Informationen über ihre Datenelemente zur Verfügung stellen, welche für die Anfrageoptimierung verwendet werden können, sind die Möglichkeiten einer solchen Optimierung eher begrenzt. Informationen wie die Größe eines Datenstroms oder dessen Datenrate oder die Verteilung der Daten selbst können allenfalls im aktuellen Zeitpunkt bestimmt werden – ihre Vorhersage ist jedoch kaum möglich. Zusätzlich müssen einige Phasen der Anfrageoptimierung an die Bedürfnisse von Datenströmen angepasst werden. Die logische Optimierung bedarf einer Überarbeitung der zugrundeliegenden Transformationen zur Erzeugung äquivalenter Anfragen, da die Verwendung gleitender Fenster in algebraischen Operationen deren Semantik ändert.[6] Dahingegen erfordert die physische Optimierung neue Optimierungsstrategien, welche die Verwendung von Algorithmen, welche kontinuierlich Ergebnisse produzieren, berücksichtigen. Damit verbunden werden neue Kostenmodelle benötigt, da die Abschätzung der Größe von Zwischenergebnissen aufgrund deren potentieller Unbeschränktheit wenig sinnvoll erscheint. Geeigneter erscheinen dahingegen Kostenmodelle, welche die Datenraten der entstehenden Datenströme berücksichtigen. Neben der herkömmlichen Anfrageoptimierung sind im Kontext von Datenströmen zwei Zweige der Anfrageoptimierung von besonderem Interesse.

- Im Rahmen der statischen Optimierung erlangt die Optimierung von Anfragemengen besondere Bedeutung. Die in Abschnitt 4.1 näher erläuterte Variante der herkömmlichen Anfrageoptimierung verarbeitet nicht nur einzelne Anfragen, sondern ganze Anfragemengen. Hierbei stellt das Erkennen gemeinsamer Teilanfragen[7] innerhalb der Anfragemenge den Kernpunkt der Optimierung dar, da die Ergebnisse dieser Teilanfragen nach einmaliger Auswertung beliebig oft wiederverwendet werden können. Daher zielt die Optimierung der einzelnen Anfragen darauf ab, möglichst große gemeinsame Teilanfragen zu erzeugen.

- Veränderungen in den Datenraten der vom System verwalteten Datenströme können dazu führen, dass optimierte Anfragen plötzlich ein völlig anderes Verhalten an den Tag legen. Beispielsweise können einzelne Operatoren plötzlich durch ansteigende Datenraten überlastet werden oder eine Pufferung, welche einen Operator vor Überlastung schützen sollte, nun aufgrund anhaltend niedriger Datenraten zu unnötigen Verzögerungen der Datenelemente führen. Häufig schafft hierbei nur eine Reorganisation der vom System verwalteten Anfragen Abhilfe. Dies ist die Aufgabe der dynamischen Optimierung, welche in Abschnitt 4.2 vorgestellt wird. Für die Re-Optimierung der im Datenstrommanagementsystem befindlichen aktiven Anfragen stehen neben den statischen Informationen der Datenströme auch vom System aufgezeichnete Informationen über die zeitliche Entwicklung der Da-

[6] Die Änderung der Semantik algebraischer Operationen durch die Verwendung gleitender Fenster wird in Abschnitt 3.2.2 näher erläutert.

[7] Gemeinsame Teilanfragen, sind Teilausdrücke, welche in mindestens zwei Anfragen auftauchen. Im Falle von logischen Operatorbäumen entspricht eine gemeinsame Teilanfrage einem Teilbaum, welcher in mehreren Operatorbäumen enthalten ist.

tenraten zur Verfügung. Jedoch sind mit der dynamischen Optimierung auch gewisse Probleme verbunden, da hierbei aktive Anfragen zur Laufzeit umgestellt werden. Zu diesem Zweck müssen die zu optimierenden Anfragen vor ihrer Reorganisation in eine Art Ruhezustand überführt werden, damit während der Umstellung keine Datenelemente verloren gehen. Zusätzlich erfordern statusbehaftete Operatoren ein besonderes Augenmerk, da deren interner Status eventuell ebenfalls angepasst werden muss. Wird beispielsweise eine Selektion aus der Ausgabe eines Joins in seine Eingaben verschoben, so müssen auch die über diese Datenströme gleitenden Fenster angepasst werden, indem die Selektion auf sie angewandt wird. Dadurch werden alle Elemente, welche die Selektion aus den Datenströmen herausfiltern würde, aus den jeweiligen Fenstern entfernt.

3.2.2 Algebra für aktive Datenquellen

Die bereits in Abschnitt 2.1 erwähnte relationale Algebra stellt ein mengenbasiertes[8] mathematisches Modell dar. D. h. eine n-stellige Operation der relationalen Algebra, welche n Eingaberelationen auf eine Ausgaberelation abbildet, erwartet für jede der n Eingaberelationen eine Instanz – also eine Menge von Tupeln – und bildet diese auf eine Instanz der Ausgaberelation ab. Da hierbei grundsätzlich die Instanzen der Eingaberelationen komplett vorliegen müssen, ist diese Algebra nicht auf Datenströme anwendbar. Daher wird eine neue formale Sprache benötigt, mit deren Hilfe Anfragen an aktive Datenquellen formuliert werden können.

Eine formale Definition einer solchen Sprache stellt [Krä03] in Form einer Algebra auf Datenströmen bereit. Eine n-stellige Operation der Algebra ist hierbei eine Abbildung von n Eingabedatenströmen auf m Ausgabedatenströme, welche die über die Eingabedatenströme eintreffenden Datenelemente verarbeitet und ihre Ergebnisse über die Ausgabedatenströme wiederum versendet. Dabei kann man grundsätzlich zwischen zwischen zwei Kategorien von Operationen unterscheiden.

3.2.2.1 Exakte Operationen

Die erste Kategorie umfasst alle Operationen, welche die Elemente eines Eingabedatenstroms ohne Kenntnis der restlichen Eingabedatenströme verarbeiten. Operationen dieser Kategorie, wie beispielsweise Filteroperationen, Abbildungen der Datenelemente oder Vereinigungen von Datenströmen, können jedes Datenelement direkt verarbeiten und eventuelle Ergebnisse sofort versenden. Diese Ergebnisse stellen stellen eine exakte Antwort für jene Teile der Eingabedatenströme dar, welche durch die Operation verarbeitet wurden. Die Semantik solcher Operationen entspricht, bis auf den Umstand, dass sie ihre Ergebnisse kontinuierlich weitergeben, statt sie in Form einer Ergebnismenge bereitzustellen, der Semantik ihrer aus der relationalen Algebra bekannten Pendants.

[8] Für den Umgang mit Multi-Relationen, d. h. Relationen, welche Multimengen von Tupeln als Instanzen zulassen, existiert die sogenannte erweiterte relationale Algebra. Sie bietet die Möglichkeit, bereits innerhalb des mathematischen Modells Duplikate zu berücksichtigen und stellt damit die zur Verdichtung der Daten benötigte Funktionalität zur Verfügung.

3.2.2.2 Approximative Operationen

Die zweite Kategorie wird durch die Operationen, welche für die Verarbeitung der Elemente eines Eingabedatenstroms andere Eingabedatenströme berücksichtigen müssen, gebildet. Im Gegensatz zu Operationen der ersten Kategorie benötigen Operationen, wie beispielsweise Differenz- und Verbundoperationen oder Durchschnitte von Datenströmen, weitere Datenströme, um die Datenelemente eines Eingabedatenstroms verarbeiten zu können. Eine Verbundoperation beispielsweise muss beide Datenströme komplett vorhalten, um beim Eintreffen eines Datenelements über einen Datenstrom den anderen nach Verbundpartnern zu durchsuchen. Da dies zwar im mathematischen Modell kein Problem darstellt, sich aber in einer späteren Implementierung nicht umsetzen lässt, schränkt man die Operationen dahingegen ein, nur jeweils ein gleitendes Fenster der Datenströme vorzuhalten. D. h. beim Eintreffen eines Datenelements aus einem der beiden Eingabedatenströme stehen nun nicht mehr alle möglichen Verbundpartner des zweiten Eingabedatenstroms zur Verfügung, sondern nur noch die explizit im gleitenden Fenster gespeicherten. Durch diese Definition entsteht eine Verbundoperation, welche in der Lage ist mit beschränktem Speicheraufwand ein approximatives Ergebnis des Verbunds zweier potentiell unbeschränkter Datenströme zu berechnen. Operatoren dieser Kategorie sind daher nur in der Lage, approximative Ergebnisse zu liefern. Damit ändert sich auch die Semantik dieser Operationen gegenüber der Semantik ihrer Pendants der relationalen Algebra.

3.3 Datenströme in XXL

Parallel zu der im Paket `xxl.cursors` enthaltenen *cursor*-Algebra, welche bereits in Kapitel 1 erwähnt wurde, stellt die Bibliothek XXL im Paket `xxl.pipes` eine Algebra objekt-relationaler Operatoren für aktive Datenquellen bereit.[9] Mittels dieser physischen Operatoren wird eine Implementierung der in Abschnitt 3.2.2 vorgestellten algebraischen Operationen zur Verfügung gestellt. Die Operatoren lassen sich mit Hilfe von *publish/subscribe*-Mechanismen zu Anfragen zusammenstellen und ermöglichen zudem die Kontrolle der so entstehenden Datenströme. Weiterhin bietet die Bibliothek XXL Hilfsmittel zur Bewältigung der in Abschnitt 3.2.1 beschriebenen Aufgaben. Neben gleitenden Fenstern und speicheradaptiven Datenstrukturen umfasst dies auch Verfahren zur Ablaufsteuerung, mit deren Hilfe einzelne Operatoren, aber auch komplette Teilanfragen durch leichtgewichtige Prozesse gekapselt werden können. In den folgenden Abschnitten werden die oben genannten Punkte näher beleuchtet.

3.3.1 Schnittstellen

Ähnlich wie bei der *cursor*-Algebra werden auch bei der Datenstromalgebra Anfragen durch die Zusammenstellung algebraischer Operatoren erzeugt. Aufgrund der in Abschnitt 3.2.2 vorgestellten Definition n-stelliger Operationen, entstehen dabei jedoch

[9] Die Datenstromalgebra wurde in Rahmen einer Diplomarbeit entwickelt und in [Krä03] vorgestellt.

anstelle von Operatorbäumen sogenannte Operatorgraphen[10]. Dies ermöglicht eine einfache gemeinsame Nutzung häufig wiederkehrender Anfragen. Zusätzlich zu den Operatoren, welche Datenströme konsumieren und gleichzeitig ihre Ergebnisse über weitere Datenströme versenden, werden noch pure Datenquellen, von denen nur Datenströme ausgehen, und Datensenken, welche Datenströme konsumieren, aber keine Quelle weiterer Ergebnisse darstellen, benötigt, um sinnvoll Anfragen generieren zu können. Die folgenden Abschnitte erläutern die Schnittstellen, welchen Datenquellen, Datensenken und Operatoren zugrundeliegen, und stellen einige Implementierungen derselben vor.

3.3.1.1 Datenquellen

Beliebige Datenquellen, welche ihre Daten autonom an registrierte Datensenken versenden, werden mittels der Schnittstelle `Source` modelliert. Im Rahmen dieser Schnittstelle wird die Funktionalität, welche zur Nutzung einer Datenquelle notwendig ist, bereitgestellt. Datensenken sind über die Methode `subscribe` in der Lage, ein Abonnement auf der Datenquelle zu erwerben, und können dieses mittels der Methode `unsubscribe` wieder kündigen. Dabei zeigen die booleschen Rückgabewerte der Methoden jeweils an, ob die zugehörige Aktion erfolgreich ausgeführt wurde. Zusätzlich ermöglichen diese Methoden die Datenquelle mittels einer Identifikationsnummer zu kennzeichnen, wodurch Operatoren, welche mehrere Eingabedatenströme verarbeiten, in der Lage sind, die eintreffenden Datenelemente den jeweiligen Datenquellen zuzuordnen. Die Methoden `open` und `close` öffnen beziehungsweise schließen die Datenquelle. Das Öffnen einer Datenquelle hat zur Folge, dass diese beginnt über die Methode `transfer` Datenelemente an alle registrierten Datensenken zu versenden. Der Rückgabewert dieser Methode bezeichnet die Anzahl der Datensenken, an die das Datenelement versandt wurde. Sind Datensenken temporär nicht an den Datenelementen der Datenquelle interessiert, so sind diese über die Methoden `pause` und `resume` in der Lage, den zugehörigen Datenstrom zu unterbrechen und zu einem späteren Zeitpunkt wieder in Gang zu setzen. Sind alle Datenströme einer Datenquelle unterbrochen, so verfällt diese in einen inaktiven Zustand, aus dem sie erst wieder erwacht, wenn der erste Datenstrom wieder in Gang gesetzt wird. Mit dem Schließen gibt die Datenquelle schließlich alle belegten Ressourcen frei und teilt allen registrierten Datensenken mit, dass keine weiteren Datenelemente mehr versandt werden. Weiterhin stellen Datenquellen einen automatischen Schließmechanismus bereit, der es ermöglicht, Datenquellen nach der nächsten Kündigung eines Abonnements beziehungsweise der Kündigung aller Abonnements automatisch zu schließen.

Mit Hilfe der abstrakten Klasse `AbstractSource` wird eine Vorimplementierung der allgemeinen Funktionen einer Datenquelle bereitgestellt, welche einzig um die Erzeugung der zu sendenden Datenelemente erweitert werden muss. Sie bildet die Grundlage für alle weiterführenden Implementierungen von Datenquellen. Weiterhin erwähnenswert ist die Klasse `CursorSource`, welche die Daten eines *cursor*s aktiv versendet. Mit

[10] Operatorgraphen sind ein gerichtete Graphen, deren Knoten Datenquellen, Datensenken und Operatoren repräsentieren, welche über gerichtete Kanten in Beziehung gesetzt werden. Weist die Richtung dieser Kanten jeweils von der Datenquelle zur Datensenke, so entspricht ein Operatorgraph einem Datenflussgraph. Zyklen sind in Operatorgraphen ausdrücklich erlaubt – wenn auch selten erwünscht.

Hilfe dieses Adapters[11] lassen sich auch passive Datenquellen direkt mit den Operatoren der Datenstromalgebra verarbeiten.

3.3.1.2 Datensenken

Datensenken subskribieren sich auf einem oder mehreren Datenströmen und konsumieren die von ihnen gelieferten Datenelemente. Eine Datensenke wird mittels der Schnittstelle `Sink` repräsentiert, welche deren Funktionalität formalisiert und mittels Methoden zur Verfügung stellt. Parallel zu den Methoden `subscribe` und `unsubscribe` der Datenquelle, ermöglichen die Methoden `addSource` und `removeSource` eine Datenquelle, auf der die Datensenke ein Abonnement erworben hat, zu registrieren beziehungsweise nach der Kündigung des Abonnements wieder zu deregistrieren. Dadurch entsteht eine doppelte Verkettung zwischen Datenquellen und Datensenken, über die jede der beteiligten Parteien auf ihr jeweiliges Gegenüber zugreifen kann. Beide Methode teilen über einen booleschen Rückgabewert den Erfolg ihrer Aktivitäten mit und ermöglichen die Kennzeichnung des entstehenden Datenstroms mittels einer Identifikationsnummer. Eintreffende Datenelemente der Eingabedatenströme werden mittels der Methode `process` verarbeitet, welche gewöhnlich direkt durch die Methode `transfer` der dem Eingabedatenstrom zugrundeliegenden Datenquelle aufgerufen wird. Weiterhin ermöglicht diese Methode einer Datenquelle, der Datensenke die mit dem Datenstrom verbundene Identifikationsnummer mitzuteilen. Versiegt schließlich eine Datenquelle, so informiert diese alle Datensenken, welche auf ihr ein Abonnement besitzen mittels deren Methode `done` über diesen Umstand und teilt ihnen die jeweils zugehörigen Identifikationsnummern mit.

Wie auch im Fall der Datenquelle, bietet die abstrakte Klasse `AbstractSource` eine Vorimplementierung der grundlegenden Funktionalität einer allgemeinen Datensenke an. Da sie nur noch um die Verarbeitung der eintreffenden Datenelemente erweitert werden muss, bildet sie die Grundlage für die Implementierung spezieller Datensenken. Eine solche Datensenke stellt die Klasse `SinkCursor` dar, welche die Daten einer aktiven Datenquelle zwischenspeichert und in Form eines *cursor*s zur Verfügung stellt. Zusammen mit der Klasse `CursorSource` bildet dieser Adapter den Brückenschlag zwischen aktiver und passiver Datenverarbeitung.

3.3.1.3 Operatoren

Operatoren sind sowohl Datensenken als auch Datenquellen. Einerseits sind sie in der Lage, bei Datenquellen ein Abonnement zu erwerben und die daraufhin eintreffenden Datenelemente zu verarbeiten. Andererseits erzeugen sie aus diesen Datenelementen Ergebnisse und versenden diese an registrierte Datensenken. Die Schnittstelle `Pipe`, welche die Funktionalität eines solchen Operators bereitstellt, erweitert folglich auch die Schnittstellen `Sink` und `Source`. Durch die Verschmelzung von Datensenke und

[11] Bei einem Adapter handelt es sich um ein Entwurfsmuster, welches die Schnittstelle einer Klasse auf eine benötigte Schnittstelle abbildet. Dies ermöglicht die Zusammenarbeit verschiedener Klassen trotz inkompatibler Schnittstellen. Eine gute Darstellung des vorliegenden Entwurfsmusters, welches auch unter der Bezeichnung *wrapper* bekannt ist, bietet [GHJV95].

Datenquelle treten Wechselwirkungen zwischen beiden auf. So bewirkt beispielsweise das Öffnen eines Operators, dass dieser auch seine registrierten Datenquellen öffnet, während beim Schließen nicht nur die registrierten Datensenken benachrichtigt, sondern auch die registrierten Datenquellen abgemeldet und ebenfalls zu schließen versucht werden. Sind dahingegen alle registrierten Datenquellen eines Operators erschöpft, so ist dieser in der Lage, sich automatisch zu schließen.

Da die Programmiersprache Java keine Mehrfachvererbung erlaubt, implementiert die abstrakte Klasse `AbstractPipe` neben den neuen Wechselwirkungen zwischen Datensenke und Datenquelle auch deren Grundfunktionalität nochmals. Wie schon deren abstrakte Implementierungen, bildet diese Klasse eine Basis für die Implementierung spezieller Operatoren, welche die Verarbeitung der eintreffenden Daten bereitstellen muss. Ein Beispiel für eine solche Implementierung stellt die Klasse `Filter` dar. Diese Klasse implementiert allein durch geeignetes Überschreiben der Methode `process` eine Selektion, indem sie für jedes eintreffende Datenelement eine übergebende Selektionsbedingung überprüft und das Element dann gegebenenfalls versendet. Eine weitere grundlegende Funktionalität der Datenstromalgebra stellt die Abbildung dar, welche durch die Klasse `Mapper` implementiert wird. Auf eintreffende Datenelemente wendet diese Klasse eine übergebene Abbildungsfunktion an, deren Ergebnis sie anschließend versendet. Von besonderem Interesse ist hierbei der allgemeine Gruppierungsoperator, welcher durch die abstrakte Klasse `AbstractGrouper` implementiert wird. Anders als die aus der relationalen Algebra bekannte Gruppierung werden in der Datenstromalgebra die einzelnen Gruppen als Ausgabedatenströme dargestellt und die eintreffenden Datenelemente auf diese Gruppen aufgeteilt. Mit Hilfe der Klasse `Aggregator` können diese Gruppen nun mittels spezieller Funktionen aggregiert werden. Weiterhin von Interesse ist die Implementierung approximativer Operationen, welche jedoch erst im Abschnitt 3.3.4 näher dargestellt wird.

3.3.2 *publish/subscribe*-Mechanismen

Durch die Verwendung von *publish/subscribe*-Mechanismen werden kontinuierliche Anfragen in die Lage versetzt, sich auf Teilanfragen bereits aktiver Anfragen anzumelden, um deren Ergebnisse gemeinsam zu nutzen. Da Anfragen an aktive Datenquellen über Operatorgraphen repräsentiert werden, bietet jede Datenquelle und jeder Operator des Graphen einen Ansatzpunkt für eine gemeinsame Nutzung der Ergebnisse des entsprechenden Teilgraphen. Diese *publish/subscribe*-Mechanismen stellen eine Implementierung des in [GHJV95] vorgestellten Entwurfsmuster Beobachter dar. Dabei bieten Datenquellen den von ihnen ausgehenden Datenstrom zur Beobachtung an. Datensenken, welche an der Beobachtung des Datenstroms interessiert sind, melden sich über die Methode `subscribe` bei der Datenquelle an.[12] Sobald der Datenstrom sich ändert – und damit ein neues Datenelement vorliegt – benachrichtigt die Datenquelle alle regis-

[12] Die vollständig Erzeugung eines Datenstroms von einer Datenquelle zu einer Datensenke muss sich sowohl die Datensenke mittels der Methode `subscribe` bei der Datenquelle, als auch die Datenquelle mittels der Methode `addSource` bei der Datensenke anmelden. Zu diesem Zweck stellt die Klasse `Pipes` die Methoden `connect` und `disconnect` bereit, mit deren Hilfe Datenquellen und Datensenken direkt verbunden werden können.

trierten Beobachter über diese Änderung, indem es mittels der Methode `transfer` das neue Datenelement an sie versendet. Erlischt das Interesse einer Datensenke an einem Datenstrom, so ist diese über die Methode `unsubscribe` in der Lage, die Beobachtung des Datenstroms wieder einzustellen. Der Vorteil dieser Mechanismen besteht darin, dass Datensenken Abonnements einer Datenquelle zur Laufzeit erwerben und wieder kündigen können, gleichzeitig aber vollkommen unabhängig voneinander sind. Mit Hilfe dieser Mechanismen können Anfragen beziehungsweise die ihnen zugrundeliegenden Operatorgraphen *bottom-up* erzeugt werden. Zu diesem Zweck wird zunächst geprüft, ob zu der Wurzel des zu erzeugenden Operatorgraphen bereits eine Datenquelle im System existiert. Ist dies nicht der Fall, so werden zunächst die Datenquellen, welche von der Wurzel konsumiert werden, und ein Operator, welche die Wurzel repräsentiert, erzeugt. Anschließend wird der Operator bei den erzeugten Datenquellen angemeldet und als Datenquelle zurückgeliefert.

3.3.3 Daten- und Kontrollfluss

Wie bereits erwähnt, wird durch die Methoden `subscribe` der Datensenke und `addSource` der Datenquelle eine doppelte Verkettung zwischen ihnen erzeugt, so dass eine Kommunikation sowohl von der Datenquelle zur Datensenke als auch in umgekehrter Richtung ermöglicht wird. Wird nun ein Operator geöffnet, so hat dies zur Folge, dass der Operator auch alle abonnierten Quellen öffnet. Die selbe Vorgehensweise wird auch eingesetzt, wenn der betreffende Operator wieder geschlossen wird. Somit erzeugen die Methoden `open` und `close` einen Kontrollfluss, welche im Operatorgraphen *top-down* verläuft. Nachdem die Datenquellen geöffnet wurden, beginnen diese, Datenelemente an alle registrierten Datensenken zu versenden. Der betrachtete Operator, bei dem die Datenelemente eintreffen, verarbeitet diese zunächst, um dann wiederum die Ergebnisse dieser Verarbeitung zu versenden. Durch dieses Zusammenspiel der Methoden `transfer` und `process` wird im Operatorgraphen ein Datenfluss in *bottom-up* Richtung erzeugt. Wird der Datenstrom von dem betrachteten Operator zu einer seiner Datensenke kurzzeitig unterbrochen, so überprüft der Operator, ob alle von ihm ausgehenden Datenströme pausieren und damit keine Datensenke momentan an seinen Ergebnissen interessiert ist. Ist dies der Fall, so werden auch alle Datenströme von einer Datenquelle zum betreffenden Operator unterbrochen. Wird nun einer der von Operator ausgehenden Datenströme wieder reaktiviert, hat dies zur Folge, dass auch wieder alle in den Operator einmündenden Datenströme reaktiviert werden müssen. Damit erzeugen auch die Methoden `pause` und `resume` einen Kontrollfluss in *top-down* Richtung. Versiegt schließlich eine Datenquelle, auf welcher der betreffende Operator ein Abonnement besitzt, so teilt dieser, falls alle anderen abonnierten Datenquellen ebenfalls versiegt sind, allen registrierten Datenquellen mit, dass diese keine Datenelemente mehr von ihm zu erwarten haben und schließt sich anschließend selbst. Dies hat zur Folge, dass durch die Methode `done` im Operatorgraph ein Kontrollfluss in *bottom-up* Richtung verursacht wird.

Somit verläuft der Kontrollfluss im Operatorgraphen sowohl in *top-down* als auch in *bottom-up* Richtung, während der Datenfluss allein in *bottom-up* Richtung verläuft.

3.3.4 Approximation

Approximative Operatoren berechnen Näherungslösungen auf der Basis gespeicherter Daten eines Datenstroms. Da diese Datenströme nicht komplett gespeichert werden können, werden für diese Aufgabe spezielle Datenstrukturen benötigt, um die zu speichernde Datenmenge effizient zu verwalten. Eine solche Klasse von Datenstrukturen wird durch *sweep line* Statusstrukturen gebildet. Unter einer *sweep line* Statusstruktur versteht man eine statusbehaftete Datenstruktur, welche effiziente Methoden für das Einfügen und Suchen von Datenelementen bereitstellt und ihren Datenbestand kontinuierlich reorganisiert. Der Ursprung dieser Datenstrukturen ist in der algorithmischen Geometrie[13] zu finden. Der *plane sweep* Algorithmus nutzt beispielsweise eine *sweep line* Statusstruktur, um in einer Ebene verteilte geometrische Objekte zu verwalten. Hierbei wird die zu erfassende Ebene durch eine *sweep line* durchlaufen und, wann immer die *sweep line* ein Objekt erreicht, dieses in ihre Statusstruktur übernommen. Um ein unkontrolliertes Anwachsen der Statusstruktur zu verhindern, wird ihr Datenbestand kontinuierlich reorganisiert und nach bestimmten Kriterien Objekte aus ihr entfernt. Um mit Hilfe einer *sweep line* Statusstruktur einen Datenstrom zu speichern, wird der aktuelle Zeitpunkt als *sweep line* betrachtet, welche über eine die Zeit repräsentierende Gerade wandert. Der Zeitpunkt, zu dem ein Datenelement den zu speichernden Datenstrom passiert, bestimmt seine Position auf der Zeitgeraden. Auf diese Weise lassen sich mit Hilfe von *sweep line* Statusstrukturen gleitende Fenster realisieren. Passt man nun noch die *sweep line* Statusstruktur derart an, dass Datenelemente nach einer Zeitspanne Δt wieder auf ihr entfernt werden, so erhält man ein Zeitfenster der Größe Δt.

In dem Paket `xxl.collections` werden sowohl *sweep line* Statusstrukturen als auch gleitende Fenster zur Verfügung gestellt. Die Schnittstelle `SweepArea` modelliert eine *sweep line* Statusstruktur , welche Einfüge- und Suchoperationen zur Verfügung stellt und mittels der Methode `reorganize` eine benutzerdefinierte Reorganisation des Datenbestands vornimmt. Eine Suche auf dem Datenbestand einer *sweep line* Statusstruktur wird mittels einer zweistelligen Bedingung realisiert, welche zu einem übergebenen Datenelement entscheidet, ob dieses ein Ergebnis der Suche nach einem zweiten übergebenen Element darstellt. Die abstrakt Klasse `SlidingWindow` implementiert diese Schnittstelle und repräsentiert damit ein gleitendes Fenster. Um nun mit Hilfe gleitender Fenster beispielsweise einen approximativen Verbund zweier Datenströme zu realisieren, werden diese mittels gleitender Fenster gespeichert. Als Bedingung für die Suche auf den gleitenden Fenstern wird die Verbundbedingung genutzt. Trifft nun ein Datenelement eines Datenstroms ein, so wird es zunächst in das entsprechende gleitende Fenster eingefügt. Anschließend wird auf dem gleitenden Fenster des zweiten Datenstroms eine Suche nach dem betreffenden Datenelement durchgeführt und so werden alle Verbundpartner bestimmt. Aus ihnen und dem Datenelement werden nun die Ergebnisse des Verbunds erzeugt und über den Ausgabedatenstrom versendet. Zu guter letzt wird das gleitende Fenster des ersten Datenstroms noch bezüglich des betreffenden Elements reorganisiert.

[13]Das Betätigungsfeld der algorithmischen Geometrie liegt in dem Entwurf und der Analyse von Algorithmen zur Lösung von geometrischen Problemen für Objekte in mehrdimensionalen Räumen.

3.3.5 Adaptivität

Neben einer anpassungsfähigen Ablaufsteuerung, welche im nachfolgenden Abschnitt behandelt wird, bildet die adaptive Speicherverwaltung eine Grundvoraussetzung für die Verwaltung kontinuierlicher Anfragen. Ein Datenstrommanagementsystem muss in der Lage sein, Anfragen Speicher frei zuzuteilen und im Bedarfsfall wieder zu entziehen. Dies ermöglicht die Schnittstelle `Adaptive` im Paket `xxl.collections`. Sie stellt Methoden bereit, mit deren Hilfe einer speicheradaptiven Datenstruktur Speicher zugewiesen werden kann. Mittels des verfügbaren Speicherplatzes und des Speicherbedarfs der einzelnen Datenelemente wird die Anzahl der Elemente bestimmt, die in der Datenstruktur gespeichert werden können. Über die Methoden `detectOverflow` und `handleOverflow` kann überprüft werden, ob es zu einem Speicherüberlauf gekommen ist, und dieser gegebenenfalls behandelt werden. Die abstrakte Klasse `MemoryManager` implementiert die Schnittstelle `Adaptive` und kann dazu verwendet werden, Datenstrukturen ein speicheradaptives Verhalten zu ermöglichen. Zu diesem Zweck genügt es, den Speichermanager um eine Implementierung der abstrakten Methoden `size` und `handleOverflow`, welche die Anzahl der in der speicheradaptiven Datenstruktur gespeicherten Elemente bestimmen beziehungsweise einen Speicherüberlauf behandeln, zu ergänzen. Nachteile dieser Vorgehensweise ergeben sich daraus, dass ein Speichermanager nur dann die korrekte Anzahl der maximal zu speichernden Datenelemente bestimmt, wenn alle über denselben Speicherbedarf verfügen, und der zusätzliche Speicherbedarf der eigentlichen Datenstruktur vernachlässigt wird.

3.3.6 Ablaufsteuerung

Das Paket `xxl.pipes.processors` stellt eine Reihe leichtgewichtiger Prozesse[14] zur Verfügung, mit deren Hilfe die Aktivität einer Datenquelle gesteuert werden kann. Die abstrakte Klasse `Processor` implementiert die Grundfunktionalität eines solchen leichtgewichtigen Prozesses, welcher, sobald er gestartet wird, solange die Methode `delay`, welche die Ausführung des Prozesses kurzfristig verzögert, und die abstrakt Methode `process` im Wechsel ausführt, bis der Prozess beendet wird. Mit Hilfe der Methode `process` wird die eigentliche Aufgabe des Prozesses festgelegt, welche im Allgemeinen darin besteht, die Methode `transfer` einer Datenquelle aufzurufen, um das nächste Datenelement zu versenden. Mit der Klasse `SourceProcessor` steht ein solcher leichtgewichtiger Prozess zur Verfügung, welcher den Versand der Datenelemente einer übergebenen Datenquelle steuert und, über die durch den Aufruf der Methode `delay` verursachte Verzögerung zwischen dem Versand zweier Datenelemente, die Datenrate der Datenquelle festlegt. Unter Zuhilfenahme der Klassen `RandomPeriodSourceProcessor` und `PoissonSourceProcessor` können zusätzlich Schwankungen in der Datenrate einer Datenquelle simuliert werden. Dabei wird die Verzögerung zwischen dem

[14] Leichtgewichtige Prozesse, welche im Kontext der Programmiersprache Java häufig auch als *threads* bezeichnet werden, stellen eine Nachbildung der Prozesse eines Betriebssystems dar. Im Gegensatz zu diesen Prozessen, laufen leichtgewichtige Prozesse jedoch im selben Speicherbereich des Hauptspeichers ab, so dass der Wechsel zwischen zwei leichtgewichtigen Prozessen nur geringe Kosten verursacht.

	Datenbank-managementsystem (DBMS)	**Datenstrom-managementsystem (DSMS)**
Datenquellen	passiv	aktiv
Daten	persist	flüchtig
Zugriff	wahlfrei	sequentielle
Anfragen	historisch	kontinuierlich
Antworten	exakt	approximativ
Speicher	Externspeicher	Hauptspeicher
Optimierung	statisch	dynamisch

Tabelle 3.1: Vergleich der Anforderungen von Datenbankmanagementsystemen und Datenstrommanagementsystemen

Versand zweier Datenelemente durch die erste Klasse zufällig bestimmt, während die Verzögerungen der zweiten Klasse einer POISSON-Verteilung[15] genügen.

Sobald ein leichtgewichtiger Prozess den Versand eines Datenelements anstößt, werden durch das Zusammenspiel der Methoden `transfer` und `process` auch die registrierten Datensenken aktiv. Daher wird ein solches Datenelement von einer Reihe von Operatoren bis zu einer finalen Datensenke in einem Zug verarbeitet. Dies hat aber auch zur Folge, dass der leichtgewichtige Prozess die Aktivität aller Datensenken, welche direkt oder über weitere Operatoren Datenelemente der Datenquelle erhalten, steuert. Um diese enge Bindung aufzubrechen, wird die Klasse `Decoupler` verwendet. Innerhalb dieser Klasse werden alle eintreffenden Datenelemente zwischengespeichert und durch einen zweiten leichtgewichtigen Prozess an die registrierten Datensenken versendet. Auf diese Weise wird die Kontrolle der Datensenken, welche direkt oder über weitere Operatoren Datenelemente vom `Decoupler` erhalten, in die Verantwortung des zweiten leichtgewichtigen Prozesses übergeben, während der erste Prozess weiterhin die restlichen Datensenken steuert. Die Zuteilung der CPU-Zeit zu den einzelnen leichtgewichtigen Prozesse wird dabei jedoch der Laufzeitumgebung – in diesem Fall also der *Java virtual machine* – überlassen.

3.4 Zusammenfassung

Aktive Datenquellen versenden selbsttätig Datenelemente an registrierte Datensenken und erzeugen somit Datenströme von potentiell unbeschränkter Größe. Die Verwaltung solcher Datenströme erfordert aus diesem Grund eine effiziente Handhabung der verfügbaren Speicher- und CPU-Ressourcen. Dies umfasst sowohl eine flexible Zuteilung dieser Ressourcen als auch einen schonenden Umgang mit ihnen. Von besonderem In-

[15]Bei der POISSON-Verteilung handelt es sich um eine diskrete, asymmetrische Verteilung, welche einen Spezialfall der Binomialverteilung bildet. Durch ihre Verwendung wird erreicht, dass die Datenrate des betreffenden Datenstroms leicht um eine übergebende Datenrate λ schwankt, ihr jedoch im Mittel entspricht.

teresse sind hierbei die dynamische Optimierung und Verfahren, welche den Speicherbedarf oder die Datenraten der Eingabedatenströme auf Kosten der Exaktheit der Ergebnisse zu reduzieren vermögen. Die unterschiedlichen Anforderungen, welche an Datenstrommanagementsysteme und Datenbankmanagementsysteme gestellt werden, fasst Tabelle 3.1 nochmal kurz zusammen. Einen weiteren interessanten Punkt stellt die Definition einer formalen Sprache zur Formulierung kontinuierlicher Anfrage dar. Obwohl eine solche Sprache, welche [Krä03] in Form einer Datenstromalgebra definiert, grosse Ähnlichkeit zur erweiterten relationalen Algebra besitzt, haben die potentielle Unbeschränktheit von Datenströmen und die Art, wie diese ihre Datenelemente bereitstellen, einige grundlegende Unterschiede zur Folge, welche nähere Untersuchungen dringend erforderlich machen.

Im zweiten Teil des Kapitels wird die im Paket `xxl.pipes` bereitgestellte Implementierung der Datenstromalgebra vorgestellt. Neben den *publish/subscribe*-Mechanismen, welche die dynamische Erzeugung und Umgestaltung kontinuierlicher Anfragen ermöglicht, bieten vor allem die flexible Zuteilung von Operatoren zu leichtgewichtigen Prozessen, die Verwendung speicheradaptiver Datenstrukturen und die Verwendung approximativer Verfahren zur Berechnung von Anfrageergebnissen Datenstrommanagementsystemen viele Ansatzpunkte für effiziente Verfahren zur Bewältigung ihrer Verwaltungsaufgaben.

Kapitel 4

Erweiterte Anfrageoptimierung

Während Kapitel 2 einen Überblick über die statische Optimierung einzelner Anfragen bietet und die im Rahmen der Bibliothek XXL implementierten Verfahren erläutert, möchte dieses Kapitel zwei interessante Erweiterungen der Anfrageoptimierung vorstellen. Bei der ersten Erweiterung handelt es sich um eine Variante der statischen Anfrageoptimierung, welche anstelle einzelner Anfragen komplette Anfragemengen optimiert. Hierbei wird das Ziel verfolgt, Teilanfragen, welche in mehreren Anfragen in identischer Form vorliegen, nur einmalig auszuwerten, um diese Ergebnisse anschließend allen beteiligten Anfragen zur Verfügung zu stellen. Die zweite Erweiterung der Anfrageoptimierung geht einen völlig anderen Weg. Statt eine Anfrage zunächst komplett zu optimieren und sie anschließend auszuwerten, verlegt die dynamische Optimierung den Vorgang der Optimierung in die eigentliche Auswertung der Anfrage hinein. Dies hat den Vorteil, dass neben den normalen Metadaten auch solche, welche erst im Laufe der Auswertung einer Anfrage verfügbar werden, für die Optimierung genutzt werden können. Die nun folgenden Abschnitte sollen einen Überblick über diese Erweiterungen bieten und ihre Vor- und Nachteile aufzeigen.

4.1 Anfragenübergreifende Optimierung

Treffen Anfragen gleichzeitig oder in rascher Folge bei einem Datenbankmanagementsystem ein, so ist dieses gefordert, die Anfragen zügig zu optimieren und zur Ausführung zu bringen. Dieses zeitliche Zusammentreffen von Anfragen kann eine direkte Folge einer Stapelverarbeitung (*batch processing*) der Anfragen sein – aber auch in stark frequentierten Systemen tritt dieser Effekt häufig auf. Desweiteren kann die mehrfache Definition einer Regel in deduktiven Datenbanksystemen[1] dazu führen, dass eine

[1]Deduktive Datenbanksysteme stellen eine Verschmelzung aus Datenbanksystem und Logikprogrammierung dar. Statt – wie bei einem relationalen Datenbanksystem – alle Daten explizit zu speichern, besteht eine deduktive Datenbank aus einem extensionalen und einem intensionalen Teil. Der extensionale teil besteht aus einer Menge von Fakten, welche die Wissensbasis des Systems bildet. Dahingegen stellt der intensionale Teil eine Menge von Ableitungsregeln und Beschränkungen zur Verfügung, mit deren Hilfe nach dem Wenn-Dann-Prinzip aus den Fakten des extensionalen Teils neue Fakten gewonnen werden können.

einzelne an das Datenbanksystem gestellte Anfrage eine Reihe von Anfragen auf der unterliegenden Datenbank zur Folge hat. Werden solche Anfragen nun im Zuge einer statischen Anfrageoptimierung isoliert optimiert, so trägt dieses Vorgehen nicht dem Umstand Rechnung, dass die optimierten Anfragen nahezu gleichzeitig ausgewertet werden. Stattdessen legt der enge zeitliche Zusammenhang in der Ausführung der einzelnen Anfragen den Entschluss nahe, gemeinsame Teilanfragen einmalig auszuwerten und ihre Ergebnisse in Form von temporären Relationen den betreffenden Anfragen zur Verfügung zu stellen. Hierbei gilt es, bereits im Rahmen der Anfrageoptimierung, die notwendigen Vorbereitungen für dieses Vorgehen zu treffen. Statt zu den einzelnen Anfragen lokale Anfragepläne zu generieren und mittels eines Kostenmodells zu bewerten, erzeugt eine anfragenübergreifende Optimierung (*multiple-query optimization*) globale Anfragepläne, welche die Anfragepläne aller beteiligten Anfragen enthalten und die einmalige Ausführung gemeinsamer Teilanfragen berücksichtigen. Das verwendete Kostenmodell muss nun, neben den Kosten der im globalen Anfrageplan vorkommenden Operatoren, zusätzlich die Kosten, welche durch die Erzeugung einer temporären Relation und die Zwischenspeicherung der Ergebnisse einer gemeinsamen Teilanfrage entstehen, berücksichtigen. Damit ist es nun möglich, auch die gleichzeitige Ausführung mehrerer Anfragen zu optimieren. Jedoch ist auch die anfragenübergreifende Optimierung – wie so viele Optimierungsprobleme – eine NP-harte Aufgabe.[2] Aus diesem Grund werden auch für die anfragenübergreifende Optimierung Verfahren benötigt, welche den Suchraum für die Bestimmung eines globalen Anfrageplans geeignet einschränken.

4.1.1 Anfragenübergreifende Optimierung nach Sellis

Ausgehend von einer eingeschränkten relationalen Algebra[3], werden in [Sel88] zwei Verfahren zur anfragenübergreifenden Optimierung einer Menge von Anfragen vorgestellt. Als Referenzalgorithmus wird dabei ein Verfahren verwendet, welches eine Menge optimaler lokaler Anfragepläne als Eingabe erhält und diese in einer beliebigen Reihenfolge ausführt. Dieser Algorithmus entspricht dem herkömmlichen Verfahren, bei dem Anfragen einzeln optimiert und ausgeführt werden, und wird als Algorithmus AS (*Arbitrary Serial execution*) bezeichnet. Die Kosten des globalen Anfrageplans GP ergeben sich damit direkt aus den Kosten der lokal optimalen Anfragepläne und entsprechen

$$cost(GP) = \sum_{i=1}^{n} bestcost(Q_i) \tag{4.1}$$

wobei *bestcost* einer Anfrage Q_i, $1 \leq i \leq n$, die Kosten ihres lokal optimalen Anfrageplans zuordnet. Das erste Verfahren, welches Algorithmus IE (*Interleaved Execution*)

[2] Timos K. Sellis und Subrata Ghosh haben in [SG90] gezeigt, dass bereits die Aufgabe, zu n gegebenen Mengen von Anfrageplänen $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n$, wobei $\mathcal{P}_i = \{P_{i1}, P_{i2}, \ldots, P_{ik_i}\}$ eine Menge möglicher Anfragepläne zur Auswertung der Anfrage Q_i, $1 \leq i \leq n$, repräsentiert, einen globalen Anfrageplan durch die Auswahl eines Anfrageplans aus jeder Menge $\mathcal{P}_i$ zu finden, dessen Kosten minimal sind, NP-hart ist.

[3] In [Sel88] werden nur Selektionsbedingungen, welche ein Attribut der Eingaberelation mit einer Konstanten vergleichen, und Verbundbedingungen, welche ein Attribut der ersten Eingaberelation und ein Attribut der zweiten auf Gleichheit überprüfen, betrachtet. Desweiteren wird angenommen, dass Anfragen nur in Form einer Konjunktion von Selektions- und Verbundbedingungen auftreten.

genannt wird, erwartet als Eingabe ebenfalls eine Menge optimaler lokaler Anfragepläne, welche in kleinere Teilanfragen zerlegt werden. Diese Teilanfragen werden nun einzeln ausgeführt und ihre Ergebnisse über temporäre Relationen für die weitere Verarbeitung bereitgestellt. Mit Hilfe des Algorithmus IE können bestehende Datenbankmanagementsysteme einfach um eine anfragenübergreifende Optimierung erweitert werden, indem er zwischen die herkömmliche Anfrageoptimierung und die Übersetzung der optimierten Anfragepläne in eine ausführbare Form eingefügt wird. Bei dem zweiten Verfahren mit dem Namen Algorithmus HA (*Heuristic Algorithm*) handelt es sich um ein heuristisches Verfahren, welches aus einer Menge von lokalen Anfragepläne einen globalen Anfrageplan durch Wahl eines lokalen Anfrageplans pro Anfrage aufbaut. Da der Algorithmus im Allgemeinen für jede zu optimierende Anfrage mehrere lokale Anfragepläne erwartet, ersetzt er in einem Datenbankmanagementsystem einen Großteil der herkömmlichen Anfrageoptimierung. Lediglich die Erzeugung von geeigneten Anfragepläne bleibt dabei in ihrer Verantwortung. Eine exakte Beschreibung der beiden Algorithmen bieten die nun folgenden Abschnitte.

4.1.1.1 Algorithmus IE

Zu einer gegebenen Menge $\mathcal{P}^* = \{P_1^*, P_2^*, \ldots, P_n^*\}$ optimaler lokaler Anfragepläne P_i^*, $1 \leq i \leq n$, bestimmt der Algorithmus IE zunächst diejenigen Anfragepläne, welche möglicherweise gemeinsame Teilanfragen besitzen, indem er ihre Basisrelationen vergleicht. Alle Anfragepläne, welche mindestens eine gemeinsame Basisrelation mit einem anderen Anfrageplan besitzen, werden in den globalen Anfrageplan eingefügt. Die restlichen Anfragepläne werden auf normalem Wege zur Ausführung gebracht, da sie keine gemeinsamen Teilanfragen enthalten können. Auf den so erzeugten globalen Anfrageplan werden anschließend die folgenden Transformationen angewendet.

IE1. Sei s eine Selektion auf der Relation R, deren Selektionsbedingung auf der Attributmenge $\mathcal{A}_s \subseteq RS_R$ überprüft wird. Sei weiterhin $S(s) = \{s_1, s_2, \ldots, s_m\}$ die Menge der Selektionen s_i, $1 \leq i \leq m$, auf der selben Relation, deren Selektionsbedingung auf einer Attributmenge $\mathcal{A}_{s_i} \subseteq \mathcal{A}_s$ überprüft wird und welche für jede Instanz I der Relation R $s(I) \subseteq s_i(I)$ erfüllen. Bestimme die Selektion $s^* \in S(s)$, für die eine Zwischenspeicherung ihrer Ergebnisse in einer temporären Relation den geringsten Speicherbedarf verursacht, und ersetze die Eingaberelation R von s durch die von s^* erzeugte temporäre Relation.

IE2. Sei $O = \{o_1, o_2, \ldots, o_m\}$ eine Menge der Operatoren, welche die selben Eingaberelationen $R_1, R_2, \ldots, R_k$ verarbeiten und dabei identische temporäre Relationen erzeugen. Wähle den Operator $o_{j^*} \in O$, welcher in dem optimalen lokalen Anfrageplan P_j^* mit dem kleinsten Index j vorkommt, und ersetze die Ausgaberelationen der o_i, $1 \leq i \leq m$, $i \neq j^*$, durch die von o_{j^*} erzeugte temporäre Relation.

IE3. Um identische Operationen, welche durch die Schritte IE1 und IE2 erzeugt wurden, zu eliminieren, wird Schritt IE2 wiederholt, bis keine Reduktion des globalen Graphen mehr möglich ist.

Die Vollendung von Schritt IE3 erzeugt einen optimalen globalen Anfrageplan GP', der nun, ausgehend von den Basisrelationen, dem Datenfluss folgend ausgewertet werden kann.

Die Kosten des globalen Anfrageplans GP' ergeben sich aus den Kosten der lokal optimalen Anfragepläne und entsprechen

$$cost(GP') = \sum_{i=1}^{n} bestcost(Q_i) - \sum_{s \in CS} savings(s) \tag{4.2}$$

Hierbei bezeichnet CS die Menge aller gemeinsamen Teilanfragen, welche in den optimalen lokalen Anfragepläne P_i^*, $1 \leq i \leq n$, gefunden wurden, und $savings(s)$ benennt die Kosten, welche durch die Verwendung der temporären Relation einer gemeinsamen Teilanfrage s anstelle der Basisrelationen eingespart werden. Dabei ergeben sich die eingesparten Kosten bei Ausführung von Schritt IE1 aus den Kosten C_R für den Zugriff auf die Relation R abzüglich der Kosten C_{s^*} für den Zugriff auf die temporäre Relation von s^* und bei Ausführung von Schritt IE2 für jeden Operator o_i, $1 \leq i \leq m$, aus den Kosten C_{R_j} für den Zugriff auf die Relationen R_j, $1 \leq j \leq k$, zuzüglich der Kosten C_{o_i} für das Anlegen der temporären Relation von o_i.

Die Komplexität des Algorithmus IE liegt in der Größenordnung von $N \cdot \prod_{i=1}^{k} n_i$, wobei k die Anzahl der durch GP repräsentierten Anfragepläne, n_i die Anzahl der Operatoren im optimalen lokalen Anfrageplan P_i^* und N die Anzahl der Durchläufe von Schritt IE2, welche im schlimmsten Fall der Tiefe des längsten Anfrageplans entspricht, ist.

4.1.1.2 Algorithmus HA

Basierend auf einer gegebenen Menge $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_n\}$ von Anfragen, bestimmt der Algorithmus HA zu jeder Anfrage Q_i, $1 \leq i \leq n$, eine Menge möglicher Anfragepläne $\mathcal{P}_i = \{P_{i1}, P_{i2}, \ldots, P_{ik_i}\}$ für deren Auswertung. Mit Hilfe dieser Mengen von Anfrageplänen wird nun der Zustandsraum $\mathcal{S}$ des globalen Anfrageplans GP definiert.

Ein Zustand s wird durch ein n-Tupel $\langle P_{1j_1}, P_{2j_2}, \ldots, P_{nj_n} \rangle$ mit $P_{ij_i} \in \mathcal{P}_i \cup \{\text{NULL}\}$ repräsentiert, dessen i-te Stelle den zur Auswertung von Q_i gewählten Anfrageplan enthält. Der Wert NULL deutet dabei an, dass der betreffenden Anfrage noch kein Anfrageplan zugeordnet wurde. Eine Transition zwischen zwei Zuständen ersetzt den NULL-Eintrag mit dem niedrigsten Index i im ersten Zustand durch einen Anfrageplan $P_{ij} \in \mathcal{P}_i$ und weist damit der zugehörigen Anfrage Q_i einen Anfrageplan zu. Die Kosten einer solchen Transition werden durch die zusätzlichen Kosten, welche entstehen, wenn der Anfrageplan P_{ij} auf den bereits berechneten Ergebnissen der im ersten Zustand enthaltenen Anfragepläne ausgewertet wird, festgelegt.

Werden nun der Zustand $s_0 = \langle \text{NULL}, \text{NULL}, \ldots, \text{NULL} \rangle$ als Anfangszustand und die Zustände $s_F = \langle P_{1j_1}, P_{2j_2}, \ldots, P_{nj_n} \rangle$ mit $P_{ij_i} \neq \text{NULL}$, $1 \leq i \leq n$ als Endzustände im Zustandsraumes $\mathcal{S}$ markiert, so lässt sich das vorliegende anfragenübergreifende Optimierungsproblem auf eine Wegsuche im Zustandsraum $\mathcal{S}$ zurückführen. Diese Aufgabe kann beispielsweise mit dem A* Algorithmus[4] effizient gelöst werden. Das Ergebnis die-

[4]Bei dem A* Algorithmus handelt es sich um einen *branch and bound* Algorithmus, welcher in

ses Algorithmus ist ein Pfad vom Anfangszustand s_0 zu einem Endzustand s_F, dessen Kosten den Kosten des globalen Anfrageplans entsprechen. Die durch s_F bezeichneten Anfragepläne müssen letztendlich noch zu einem globalen Anfrageplan GP zusammengefügt werden. Dieser Vorgang entspricht im Wesentlichen den Schritten IE2 und IE3 des Algorithmus IE.

Die Kosten des globalen Anfrageplans GP ergeben sich aus den Kosten der gewählten Anfragepläne und entsprechen

$$cost(GP) = \sum_{P \in \mathcal{P}} cost(P) - \sum_{s \in CS} savings(s) \tag{4.3}$$

Hierbei ist $\mathcal{P}$ die Menge der in s_F bezeichneten Anfragepläne – die Menge CS und die Kostenfunktion $savings(s)$ sind analog zu Gleichung 4.2 definiert.

Die Komplexität des Algorithmus HA entspricht im Wesentlichen der Komplexität des A* Algorithmus, welche im schlimmsten Fall exponentiell in der Anzahl der zu optimierenden Anfragen ist. Im Mittel hängt die Komplexität des A* Algorithmus jedoch sehr stark von der Fähigkeit der Kostenfunktion $h(s)$, die tatsächlichen Kosten eines Pfades im Zustandsraum abzuschätzen, ab.

4.1.2 Anfragenübergreifende Optimierung nach Roy et al.

Auch in [RSSB00] (siehe auch [Roy98] und [Roy00]) werden Verfahren für die anfragenübergreifende Optimierung präsentiert, welche das in [Gra94] vorgestellte System Volcano um eine solchen Optimierung erweitert. Anders als bei den im vorherigen Kapitel vorgestellten Verfahren, werden in Volcano jedoch anstelle von Operatorgraphen UND/ODER-Graphen zur Repräsentation von Anfragepläne eingesetzt. Bei einem UND/ODER-Graph handelt es sich um einen gerichteten azyklischen Graph, dessen Knoten in UND-Knoten und ODER-Knoten unterteilt werden können. Dabei gilt jedoch die Einschränkung, dass die Kinder eines UND-Knoten ausschließlich ODER-Knoten und die Kinder eines ODER-Knotens ausschließlich UND-Knoten sein dürfen. Der Unterschied zwischen UND- und ODER-Knoten liegt in ihrem Verhältnis gegenüber ihren Kindern begründet. Während für die Verarbeitung eines UND-Knotens grundsätzlich alle Kinder des Knotens verarbeitet werden müssen, genügt es für die Verarbeitung eines ODER-Knotens, ein einziges Kind des Knotens zu verarbeiten.

Um nun mit Hilfe eines UND/ODER-Graphen einen Anfrageplan repräsentieren zu können, werden den beiden Knotentypen unterschiedliche Funktionen zugewiesen. Während die Operatoren eines Anfrageplans mittels UND-Knoten modelliert werden, dienen ODER-Knoten dem Zweck, eine Menge äquivalenter Teilausdrücke zu repräsentieren. Ausgehend von diesen Funktionen werden UND-Knoten auch als Operationskno-

einem gewichteten gerichteten Graphen einen Pfad von einem Anfangszustand s_0 zu einer Menge von Endzuständen s_F berechnet, der unter allen solchen Pfaden minimale Kosten verursacht. Dabei dient eine Funktion $h(s)$, welche eine unterer Grenze für die zusätzlichen Kosten eines Weges vom Zustand s zu einem Endzustand s_F berechnet, dem Zweck, den Suchraum für eine Lösung des Problems einzuschränken. Um die Konvergenz des A* Algorithmus zu beschleunigen, stellt [Sel88] ferner einen Algorithmus vor, der zu einer gegebenen Menge von Anfragen, ihren Anfragepläne und der Menge der darin enthaltenen identischen Operatoren eine möglichst gute Kostenfunktion bestimmt.

ten bezeichnet, während ODER-Knoten unter dem Namen Äquivalenzknoten bekannt sind. Der Vorteil der Verwendung von UND/ODER-Graphen zur Repräsentation von Anfrageplänen liegt in deren Fähigkeit begründet, äquivalente Teilanfragepläne mit Hilfe von Äquivalenzknoten direkt im ursprünglichen Anfrageplan verwalten zu können. Um nun einen Anfrageplan in Form eines Operatorgraphen in einen UND/ODER-Graphen zu transformieren, werden die Basisrelationen des Anfrageplans mittels Äquivalenzknoten dargestellt. Die Operatoren werden dahingegen jeweils durch einen Operationsknoten mit einem Äquivalenzknoten als Elternknoten repräsentiert. Damit erhält man einen UND-ODER-Graphen, dessen Wurzel ein Äquivalenzknoten ist und der keine äquivalenten Teilanfragepläne besitzt. Anschließend wird der UND/ODER-Graph expandiert, um neben dem ursprünglichen auch alle äquivalenten Anfragepläne zu repräsentieren. Zu diesem Zweck werden auf die Operationen des Anfrageplans alle möglichen Transformationen angewendet und die entstehenden äquivalenten Teilanfragepläne an den Äquivalenzknoten des ursprünglichen Teilanfrageplans angehängt.

Um eine Menge von Anfragepläne gemeinsam zu optimieren, werden diese durch einen einzigen UND/ODER-Graphen repräsentiert. Zu diesem Zweck wird ein neuer Operationsknoten erzeugt und die Wurzel-Äquivalenzknoten der zu optimierenden Anfragepläne als dessen Eingaben angehängt. Zusätzlich werden zwei Erweiterungen der anschließenden Expansion eingeführt. Die erste Erweiterung dient der Erkennung logisch äquivalenter Teilanfragepläne. Sobald im Zuge der Expansion zwei Äquivalenzknoten entdeckt werden, deren Teilanfragepläne logisch äquivalent sind, so werden diese zu einem einzigen Äquivalenzknoten zusammengefasst. Diese Erweiterung entspricht im Wesentlichen dem im vorherigen Abschnitt vorgestellten Schritt IE2 des Algorithmus IE. Die zweite Erweiterung beschäftigt sich mit der Identifikation von Implikationen zwischen Operatoren. Parallel zum Schritt IE1 im vorherigen Abschnitt, werden hierbei Operatoren auf gemeinsamen Eingaberelationen gesucht, so dass sich die Ergebnisse des ersten Operators unabhängig von den Instanzen der Eingaberelationen stets aus den Ergebnissen des zweiten Operators berechnen lassen. Im Detail bedeutet dies, dass für eine Reihe von Selektionen, welche auf einer gemeinsamen Relation ausgeführt werden, eine neue Selektion erzeugt wird, welche die Disjunktion all dieser Selektionen bildet und ihnen fortan als Eingabe dient. Aggregationen lassen sich auf diese Art ebenfalls behandeln, indem eine neue Aggregation eingeführt wird, welche über alle später benötigten Attribute der Eingaberelation gruppiert und alle benötigten Aggregate bereitstellt. Dabei besteht das Ziel dieser Erweiterung darin, gemeinsame Berechnungen aus Operatoren, welche nicht logisch äquivalent sind, herauszuziehen und nur einmalig auszuführen.

Der nun vorliegende UND/ODER-Graph enthält alle möglichen Anfragepläne der zu optimierenden Anfragen und repräsentiert gemeinsame Teilanfragen durch die mehrfache Verwendung eines Äquivalenzknotens. Die in den folgenden Abschnitten vorgestellten Verfahren dienen nun dem Zweck, eine Menge von Äquivalenzknoten zu bestimmen, deren Teilanfragen im Vorfeld ausgewertet und deren Ergebnisse über temporäre Relationen zur Wiederverwendung bereitgestellt werden. Die ersten zwei Verfahren, die Volcano-SH und Volcano-RU Algorithmen, stellen dabei Erweiterungen der normalen in Volcano integrierten Optimierungsalgorithmus dar. Dieser Algorithmus, welcher

in [GM93] näher erläutert wird, führt auf dem expandierten UND/ODER-Graphen eine Traversierung mittels Tiefensuche durch. Im Zuge dieser Traversierung berechnet der Algorithmus anhand eines Kostenmodell die Kosten jedes Knotens und wählt für jeden Äquivalenzknoten den Teilanfrageplan mit den geringsten Kosten aus. Dabei sind die Kosten eines Operationsknotens o durch die Formel

$$cost(o) = cost\_of\_execution(o) + \sum_{e_i \in children(o)} cost(e_i) \tag{4.4}$$

und die Kosten eines Äquivalenzknotens e durch die Formel

$$cost(e) = min\{cost(o_i) | o_i \in children(e)\} \tag{4.5}$$

bestimmt, wobei $cost\_of\_execution(o)$ die Kosten, welche durch die Auswertung der Operation o entstehen, bezeichnet. Bei dem dritten Verfahren handelt es sich um einen *Greedy*-Algorithmus, welcher zu einer Menge von Äquivalenzknoten, deren Ergebnisse materialisiert, d. h. in Form temporärer Relationen zur Wiederverwendung bereitgestellt, werden sollen, den optimalen Anfrageplan bestimmt.

4.1.2.1 Volcano-SH Algorithmus

Im Rahmen des Volcano-SH Algorithmus wird der expandierte UND/ODER-Graph zunächst mit Hilfe des in Volcano integrierten Optimierungsalgorithmus verarbeitet. Der dadurch erzeugte Anfrageplan enthält die Anfragepläne für alle zu optimierenden Anfragen und repräsentiert gemeinsame Teilanfragepläne durch mehrfach verwendete Äquivalenzknoten. Nun entscheidet der Volcano-SH Algorithmus mittels eines Vergleiches der Kosten darüber, welche gemeinsamen Teilanfragepläne materialisiert und welche mehrfach ausgewertet werden. Dabei wird ein Äquivalenzknoten e materialisiert, wenn die Bedingung

$$cost(e)+mat\_cost(e)+reuse\_cost(e)*(num\_uses(e)-1) < num\_uses(e)*cost(e) \tag{4.6}$$

erfüllt ist, wobei $mat\_cost(e)$ die Kosten für die Materialisierung der Ergebnisse von e, $reuse\_cost(e)$ die Kosten der Wiederverwendung dieser Ergebnisse und $num\_uses(e)$ die Häufigkeit dieser Wiederverwendung bezeichnen. Da die Kosten eines Äquivalenzknotens und die Häufigkeit der Wiederverwendung seiner materialisierten Ergebnisse davon abhängen, welche anderen Knoten materialisiert wurden, trifft der Algorithmus die Entscheidung, ob ein Knoten materialisiert wird oder nicht, in einem *bottom-up* Durchlauf durch den UND/ODER-Graphen.

4.1.2.2 Volcano-RU Algorithmus

Auch der Volcano-RU Algorithmus erhält einen expandierten UND/ODER-Graphen als Eingabe. Im Gegensatz zu Volcano-SH, optimiert Volcano-RU jedoch die enthalten Anfragepläne einzeln nacheinander. Nachdem ein Anfrageplan P_i mittels des normalen Optimierungsalgorithmus von Volcano optimiert wurde, werden die Äquivalenzknoten

Algorithm 4.1 *Greedy*-Algorithmus zur Bestimmung der materialisierten Knoten

IN	: DAG	expandierter UND/ODER-Graph
OUT	: X	Menge der materialisierten Knoten in DAG

1: $X \leftarrow \emptyset$
2: let Y be a set of equivalence nodes in DAG
3: **while** $Y \neq \emptyset$ **do**
4: pick $x \in Y$ which minimizes $bestcost(DAG, \{x\} \cup X)$
5: **if** $bestcost(DAG, \{x\} \cup X) < bestcost(DAG, X)$ **then**
6: $Y \leftarrow Y \setminus \{x\}$
7: $X \leftarrow X \cup \{x\}$
8: **else**
9: $Y \leftarrow \emptyset$
10: **end if**
11: **end while**
12: return X

des UND/ODER-Graphen, welche Teil des Anfrageplans P_i sind, auf eine mögliche Materialisierung hin untersucht. Falls Gleichung 4.6 für eine einzige Wiederverwendung der materialisierten Ergebnisse, d. h. $num\_uses(e) = 2$, erfüllt ist, so wird der Knoten e fortan als materialisiert betrachtet und steht fortan für eine Wiederverwendung zur Verfügung. Nachdem alle Anfragepläne des UND/ODER-Graphen auf diese Weise optimiert wurden, wird der Algorithmus Volcano-SH auf ihn angewendet, um letztendlich zu entscheiden, welche Äquivalenzknoten materialisiert werden.

4.1.2.3 *Greedy*-Algorithmus

Im Gegensatz zu den bisher vorgestellten Algorithmen, nutzt der *Greedy*-Algorithmus ein völlig andere Vorgehensweise für die anfragenübergreifende Optimierung. Aus dem expandierten UND/ODER-Graphen wählt der Algorithmus eine beliebige Menge von Äquivalenzknoten, welche fortan als materialisiert betrachtet werden, und bestimmt dazu den Anfrageplan, der die materialisierten Knoten optimal einsetzt. Dieser Vorgang wird für verschiedene Mengen von Knoten wiederholt, um eine möglichst gute Menge materialisierter Knoten zu erhalten. Dabei findet eine Kostenfunktion $bestcost(DAG, X)$ Verwendung, welche zu einem expandierten UND/ODER-Graph DAG und einer Menge materialisierter Knoten X die Kosten des optimalen Anfrageplans bestimmt. Algorithmus 4.1 skizziert das Vorgehen des *Greedy*-Algorithmus. Um eine vollständige Überprüfung aller möglichen Mengen von Äquivalenzknoten zu verhindern, erfordert der *Greedy*-Algorithmus eine Reihe von Optimierungen, mit dem Ziel den optimalen Anfrageplan eines UND/ODER-Graphen effizient bestimmen zu können.

- Zunächst wird die Menge der zu untersuchenden Knoten auf die im expandierten UND/ODER-Graph mehrfach verwendeten Äquivalenzknoten beschränkt, da nur

deren Materialisierung sinnvoll erscheint.

- Zusätzlich können die Kosten, welche durch den häufigen Aufruf der Kostenfunktion *bestcost* in Zeile 4 des Algorithmus verursacht werden, durch ein Verfahren zur inkrementellen Berechnung von *bestcost* reduziert werden.

- Durch die Verwendung einer monotonen Kostenfunktion $benefit$, welche die durch die Materialisierung eines Knotens eingesparten Kosten bestimmt, wird zusätzlich verhindert, dass die Funktion *bestcost* für jeden Knoten der zu überprüfenden Menge aufgerufen werden muss.

Die vorangegangene Aufzählung stellt nur einen kurzen Überblick über die in [RSSB00] vorgeschlagenen Möglichkeiten zur Optimierung des *Greedy*-Algorithmus dar. Für nähere Informationen wird an dieser Stelle auf die Quelle selbst verwiesen.

4.2 Dynamische Optimierung

Im Gegensatz zu den bisher angesprochenen Formen der statischen Optimierung, verfolgt die dynamische Optimierung einen anders gearteten Ansatz. Statt Anfragen mit Hilfe statischer Metadaten, welche bereits im Vorfeld der eigentlichen Ausführung bekannt sind, zu optimieren, führt die dynamische Optimierung den eigentlichen Vorgang der Optimierung erst zur Ausführungszeit der Anfrage durch. Diese Vorgehen wird beispielsweise notwendig, wenn wichtige Metadaten, welche für die Optimierung von Anfragen benötigt werden, im Vorfeld ihrer Ausführung noch nicht feststehen. Zusätzlich können starke Änderungen der Metadaten, auf deren Basis die Optimierung einer Anfrage durchgeführt wurde, zur Ausführungszeit der Anfrage dazu führen, dass die Entscheidungen, welche im Rahmen der Anfrageoptimierung getroffen wurden, nicht mehr mit den neuen Metadaten zu vereinbaren sind und damit der zugehörige Anfrageplan erhebliche Leistungseinbußen erleidet. Die folgenden Abschnitte sollen diese Anwendungen dynamischer Optimierung kurz erläutern.

4.2.1 Optimierung in verteilten Datenbanksystemen

Bei einem verteilten Datenbanksystem handelt es sich um ein Datenbanksystem, dessen gespeicherte Daten auf eine Menge von Datenbanken verteilt sind. Dabei residieren die einzelnen Datenbanken auf mehreren über ein Kommunikationsnetz miteinander verbundenen Rechnern, welche auch Knoten des Netzes genannt werden. Der Anwender eines solchen Datenbanksystems erscheinen die unterliegenden verteilten Datenbanken als logische Einheit und er benötigt keinerlei Wissen über die Verteilung der Daten innerhalb des Systems oder über die speziellen Datenbanken auf den einzelnen Rechnern, um Anfragen an das System zu formulieren. Eine wichtige Eigenschaft verteilter Datenbanksysteme ist hierbei die sogenannte Ortstransparenz des Systems. Darunter versteht man die Eigenschaft, dass ein verteiltes Datenbanksystem den physischen Aufenthaltsort der verteilten Daten verbirgt. Da zusätzlich die Abhängigkeit von zentralen

Systemfunktionen vermieden wird, um die Verfügbarkeit des verteilten Datenbanksystems zu erhöhen, kann der Fall auftreten, dass für die Optimierung einer Anfrage in einem Knoten des verteilten Systems nur die Metadaten der lokal im Knoten gespeicherten Daten vorliegen. Da diese Informationen häufig nicht zur Anfrageoptimierung ausreichen, wird in [ONK+96] ein Form der Anfrageoptimierung vorgestellt, welche die Anfrage an alle durch die Anfrage betroffenen Knoten versendet und die lokalen Teilanfragen vor Ort optimieren lässt. Der Optimierung der lokalen Teilanfragen steht damit die komplette Palette der Metadaten des Knotens zur Verfügung, wobei der anfragende Knoten nur noch die lokalen Anfragepläne zu einem globalen Anfrageplan zusammenfügen muss. Da ein Ziel einer solchen Optimierung in der Reduzierung der Kommunikation zwischen Knoten liegt, müssen Kostenmodelle, welche hier Anwendung finden, besonders die mit der begrenzten Netzwerkkapazität verbundenen I/O-Kosten berücksichtigen. Die Analyse der Eigenschaften und Anforderungen verteilter Anfragen und deren Integration in die Bibliothek XXL ist Thema der laufenden Diplomarbeit [Mic03].

4.2.2 Optimierung von langlebigen Anfragen

Eine weitere Anwendung der dynamischen Optimierung bietet sich mit der Optimierung beziehungsweise Re-Optimierung von langlebigen Anfragen. Diese Art von Anfragen ist nicht mit kontinuierlichen Anfragen zu verwechseln, da ihre lange Lebensdauer nicht auf der kontinuierlichen Verarbeitung von Daten beruht, sondern durch die großen, zu verarbeitenden Datenmengen oder besonders teuren Berechnungen auf den Daten zustande kommt. Daher treten langlebige Anfragen meist in sehr großen Datenbanksystemen, wie beispielsweise *Data Warehouses*[5] oder Multimedia-Datenbanksystemen[6], auf. Verbunden mit der langen Ausführungszeit solcher Anfragen, besteht hierbei die Gefahr, dass Metadaten, welche für die Anfrageoptimierung herangezogen wurden, sich während der Lebensdauer einer solchen Anfrage derartig verändern, dass der zugehörige Anfrageplan nur noch als bestenfalls suboptimal zu betrachten ist. Die Änderungen der Metadaten können sowohl zwischen der Optimierung der Anfrage und der Ausführung des zugehörigen Anfrageplans, so dass der Anfrageplan sich konstant schlecht verhält, als auch während der Ausführung des Anfrageplans, was üblicherweise zu einem Einbruch seiner Leistung führt, erfolgen. Eine Re-Organisation des Anfrageplans zur Laufzeit erweist sich dabei jedoch als schwierig. Zwar lassen sich zustandslose physische Operatoren noch recht leicht in einen Ruhezustand versetzen und im Anfrageplan verschieben, bei zustandsbehafteten physischen Operatoren muss auch der interne Zustand des Operators an seine neue Position im Anfrageplan angepasst werden.

In [CG94] wird dieses Problem mittels eines sogenannten *choose-plan* Operators behandelt. Im Rahmen einer statischen Optimierung werden dabei Anfragen unter

[5] Bei einem *Data Warehouse* handelt es sich im ein großes Datenbanksystem, in welchem die gemeinsamen Daten einer Organisation gesammelt und zu Analysezwecken in aufbereiteter Form zur Verfügung gestellt werden.

[6] Ein Multimedia-Datenbanksystem dient der Speicherung und Verwaltung digitaler Medien, wie beispielsweise Ton-, Text-, Bild- oder Filmdokumente. Von besonderem Interesse ist in diesem Kontext der hohe Speicherbedarf solcher Dokumente, die Definition von Relationen zwischen ihnen und ihre aufwändige Verarbeitung.

Zuhilfenahme der verfügbaren Metadaten wie gewohnt optimiert. Fehlen jedoch Metadaten, welche für die Optimierung einer Teilanfrage notwendig sind, so werden alle in Frage kommenden Teilanfragepläne erzeugt und durch einen *choose-plan* Operator gesammelt, der anstelle des entsprechenden Teilanfrageplans in den Anfrageplan eingefügt wird. Der Operator selbst hat dabei die Aufgabe, mit Beginn der Anfrageauswertung, wenn auch die erst zur Laufzeit verfügbaren Metadaten vorliegen, auf Basis einer Kostenschätzung einen der gespeicherten Anfragepläne auszuwählen. Damit können Anfragepläne generiert werden, welche einerseits dynamisch optimiert werden und andererseits einen Großteil des Optimierungsaufwands innerhalb der statischen Optimierung erledigen, wodurch eine zusätzliche Vergrößerung der zur Anfrageauswertung benötigten Zeitspanne weitestgehend vermieden werden kann.

4.3 Zusammenfassung

Die in diesem Kapitel vorgestellten erweiterten Verfahren der Anfrageoptimierung stellen einen Überblick über die Möglichkeiten jenseits der statischen Optimierung einzelner Anfragen dar. Die anfragenübergreifende Optimierung optimiert eine Menge von Anfragen gemeinsam und untersucht sie auf Teilanfragen, welche in mehreren Anfragen vertreten sind, um sie einmalig auszuwerten und ihre Ergebnisse den betreffenden Anfragen kostengünstig zur Verfügung zu stellen. Da auch die anfragenübergreifende Optimierung ein NP-hartes Problem darstellt, werden wiederum Verfahren benötigt, mit deren Hilfe der Suchraum, der nach einer Lösung des Problems durchsucht wird, begrenzt werden kann. Von besonderem Interesse ist hierbei die Verwendung von UND/ODER-Graphen zur Repräsentation von Anfragepläne, da diese die Möglichkeit bieten, äquivalente Teilanfragen innerhalb eines Graphen zu verwalten.

Die dynamische Anfrageoptimierung stellt dahingegen lediglich eine Verschiebung der Optimierung von einem Zeitpunkt vor Beginn der Anfrageauswertung in die eigentliche Anfrageauswertung hinein dar. Diese Maßnahme kann aufgrund fehlender oder stark schwankender Metadaten zum Zeitpunkt der statischen Anfrageoptimierung notwendig werden, um optimale Ergebnisse erzielen zu können. Da die Reorganisation von Anfragepläne zu einem Zeitpunkt, zu dem sie bereits ausgewertet werden, jedoch mit gewissen Schwierigkeiten verbunden ist und die zur Ausführung der Anfrage benötigte Zeit zusätzlich belastet, sind hier Verfahren von besonderem Interesse, welche die Hauptarbeit einer solchen dynamischen Optimierung innerhalb einer statischen Optimierung leisten, um zur Laufzeit der Anfrage nur die notwendigsten Entscheidungen treffen zu müssen.

Kapitel 5

Anfrageoptimierung auf Datenströmen

Die bisher betrachteten Formen der Anfrageoptimierung haben alle zur Aufgabe, Anfragen, welche an ein Datenbanksystem gerichtet werden, zu optimieren. Das Ziel einer solchen Optimierung liegt einerseits darin, die zur Ausführung einer Anfrage benötigte Zeit zu reduzieren, um deren Ergebnisse möglichst schnell an den Anwender weiterreichen zu können, und andererseits in dem umsichtigen Umgang mit den einem Datenbanksystem zur Auswertung von Anfragen zur Verfügung stehenden Ressourcen. Auch eine Anfrageoptimierung auf Datenströmen verfolgt dieses Ziel – jedoch stellt die Verarbeitung von Datenströmen andere Anforderungen an die Anfrageoptimierung. Zunächst einmal weisen einige Operationen der Datenstromalgebra eine gegenüber den entsprechenden Operationen der erweiterten relationalen Algebra geänderte Semantik auf, so dass deren mathematische Eigenschaften näher betrachtet werden müssen. Dies ist besonders im Hinblick auf die Transformationsregeln der herkömmlichen Anfrageoptimierung von besonderem Interesse, deren Korrektheit in Bezug auf die Datenstromalgebra noch nachzuweisen ist. Um mit Hilfe der Datenstromalgebra formulierte Anfragen einer weiteren Verarbeitung zugänglich machen zu können, wird weiterhin eine Algebra flexibler logischer Operatoren benötigt, mit deren Hilfe Operatorgraphen realisiert werden können. Die Anfrageoptimierung selbst entspricht aufgrund der Eigenschaften von Datenströmen einer anfragenübergreifenden Optimierung. Parallel dazu ist auch eine dynamische Optimierung, mit deren Hilfe Anfragen, welche bereits ausgeführt werden, reorganisiert werden können.

Die folgenden Abschnitte beleuchten die Anforderungen und Probleme einer Anfrageoptimierung auf Datenströmen näher und zeigen Lösungsansätze auf. Dabei wird der Lauf einer Anfrage von ihrer algebraischen Definition über ihre logische Repräsentation bis zur Optimierung ihres Anfrageplans verfolgt.

5.1 Äquivalenzen der Datenstromalgebra

Die in [Krä03] vorgestellte Datenstromalgebra stellt Operationen der erweiterten relationalen Algebra für die Verarbeitung aktiver Datenquellen bereit. Obwohl sich damit die Semantik dieser Operationen an der ihrer relationalen Vorbilder orientiert, bestehen teilweise gravierende Unterschiede. Hierfür gibt es verschiedene Gründe. Einerseits werden die relationalen Schemata, welche der erweiterten relationalen Algebra zugrundeliegen, verallgemeinert und durch die Verwendung beliebiger Datenobjekte ersetzt, um die Anwendbarkeit der Datenstromalgebra nicht unnötig einzuschränken. Andererseits fordert der Einsatz approximativer Techniken seinen Tribut, indem dieser die Semantik solcher Operationen entschieden beeinflusst. Im Rahmen dieses Abschnitts werden die wichtigsten algebraischen Regeln vorgestellt und kurz erläutert. Anstelle jede einzelne Äquivalenz explizit zu beweisen, wird im Anschluss an ihre Vorstellung ein beispielhafter Beweis durchgeführt, um den formalen Umgang mit approximativen Techniken und ihre Auswirkungen auf ein Operation zu verdeutlichen, ohne der Rahmen dieser Arbeit zu sprengen. Die folgenden Regeln stellen einen Überblick über die Äquivalenzen der Datenstromalgebra bereit. Dabei werden durch S beziehungsweise S_i Datenströme, durch p beziehungsweise p_i Filterprädikate, durch f beziehungsweise f_i Abbildungsfunktionen und durch p_w beziehungsweise p_{w_i} Fensterprädikate bezeichnet.

1. Vereinigung, Schnitt, kartesisches Produkt und Verbund sind weiterhin kommutativ, wenn sie (Schnitt, kartesisches Produkt und Verbund) ohne gleitende Fenster bestimmt werden:

$$S_1 \cup S_2 = S_2 \cup S_1 \tag{5.1}$$
$$S_1 \cap S_2 = S_2 \cap S_1 \tag{5.2}$$
$$S_1 \times S_2 = S_2 \times S_1 \tag{5.3}$$
$$S_1 \bowtie S_2 = S_2 \bowtie S_1 \tag{5.4}$$

   Die Eingabedatenströme von Schnitt, kartesischem Produkt und Verbund, welche mit Hilfe von gleitenden Fenstern berechnet werden, lassen sich dahingegen nur dann vertauschen, wenn auch die zugrundeliegenden Fensterprädikate vertauscht werden:

$$S_1 \cap_{p_{w_1},p_{w_2}} S_2 = S_2 \cap_{p_{w_2},p_{w_1}} S_1 \tag{5.5}$$
$$S_1 \times_{p_{w_1},p_{w_2}} S_2 = S_2 \times_{p_{w_2},p_{w_1}} S_1 \tag{5.6}$$
$$S_1 \bowtie_{p_{w_1},p_{w_2}} S_2 = S_2 \bowtie_{p_{w_2},p_{w_1}} S_1 \tag{5.7}$$

   Sind die verwendeten Fensterprädikate identisch, so verhalten sich auch Schnitt, kartesisches Produkt und Verbund auf Basis von gleitenden Fenstern wieder uneingeschränkt kommutativ.

2. Vereinigung, Schnitt, kartesisches Produkt und Verbund sind weiterhin assoziativ, wenn sie (Schnitt, kartesisches Produkt und Verbund) ohne gleitende Fenster

bestimmt werden:

$$S_1 \cup (S_2 \cup S_3) = (S_1 \cup S_2) \cup S_3 \quad (5.8)$$
$$S_1 \cap (S_2 \cap S_3) = (S_1 \cap S_2) \cap S_3 \quad (5.9)$$
$$S_1 \times (S_2 \times S_3) = (S_1 \times S_2) \times S_3 \quad (5.10)$$
$$S_1 \bowtie (S_2 \bowtie S_3) = (S_1 \bowtie S_2) \bowtie S_3 \quad (5.11)$$

Die Reihenfolge von Schnitt, kartesischem Produkt und Verbund, welche mit Hilfe von gleitenden Fenstern berechnet werden, lassen sich dahingegen nur dann ändern, wenn jedem Eingabedatenstrom sein zugehöriges Fensterprädikat zugeordnet bleibt und zusätzlich das gleitende Fenster über die zuerst berechnete Operation nur diejenigen Ergebnisse enthält, deren erzeugende Datenelemente auch in den gleitenden Fenstern dieser Operation enthalten sind. Seien p_{w_i} und p_{w_j} Fensterprädikate über die Eingabedatenströme S_i und S_j der zuerst berechneten Operation, so wird durch die Verwendung der Konjunktion $p_{w_{ij}} = p_{w_i} \wedge p_{w_j}$ dieser beiden Fensterprädikate erreicht, dass – sobald ein Datenelement aus einem der beiden zugehörigen Fenster W_i oder W_j entfernt wird – alle Ergebnisse, an deren Produktion das betreffende Datenelement beteiligt war, aus dem gleitenden Fenster W_{ij} der zuerst berechneten Operation entfernt werden. Damit enthält das Fenster W_{ij} genau diejenigen Elemente, welche durch Anwendung der Operation auf die unterliegenden Fenster W_i und W_j rekonstruiert werden können.

$$S_1 \cap_{p_{w_1}, p_{w_2} \wedge p_{w_3}} (S_2 \cap_{p_{w_2}, p_{w_3}} S_3) = (S_1 \cap_{p_{w_1}, p_{w_2}} S_2) \cap_{p_{w_1} \wedge p_{w_2}, p_{w_3}} S_3 \quad (5.12)$$
$$S_1 \times_{p_{w_1}, p_{w_2} \wedge p_{w_3}} (S_2 \times_{p_{w_2}, p_{w_3}} S_3) = (S_1 \times_{p_{w_1}, p_{w_2}} S_2) \times_{p_{w_1} \wedge p_{w_2}, p_{w_3}} S_3 \quad (5.13)$$
$$S_1 \bowtie_{p_{w_1}, p_{w_2} \wedge p_{w_3}} (S_2 \bowtie_{p_{w_2}, p_{w_3}} S_3) = (S_1 \bowtie_{p_{w_1}, p_{w_2}} S_2) \bowtie_{p_{w_1} \wedge p_{w_2}, p_{w_3}} S_3 \quad (5.14)$$

Sind die verwendeten Fensterprädikate identisch, so verhalten sich auch Schnitt, kartesisches Produkt und Verbund auf Basis von gleitenden Fenstern wieder uneingeschränkt assoziativ.

3. Filter sind vertauschbar und können durch Konjunktion der Filterprädikate zu einem einzigen Filter zusammengefasst werden:

$$\sigma_{p_1}(\sigma_{p_2}(S)) = \sigma_{p_2}(\sigma_{p_1}(S)) \quad (5.15)$$
$$\sigma_{p_1}(\sigma_{p_2}(\ldots(\sigma_{p_n}(S))\ldots)) = \sigma_{p_1 \wedge p_2 \wedge \ldots \wedge p_n}(S) \quad (5.16)$$

4. Abbildungen sind vertauschbar, falls die zur Abbildung der Datenelemente verwendeten Funktionen kommutieren, und lassen sich durch die Hintereinanderausführung dieser Funktionen zu einer einzigen Abbildung zusammenfassen:

$$\mu_{f_1}(\mu_{f_2}(S)) = \mu_{f_2}(\mu_{f_1}(S) \Leftrightarrow f_1 \circ f_2 \equiv f_2 \circ f_1 \quad (5.17)$$
$$\mu_{f_1}(\mu_{f_2}(\ldots(\mu_{f_n}(S))\ldots)) = \mu_f(S) \Leftrightarrow f \equiv f_1 \circ f_2 \circ \ldots \circ f_n \quad (5.18)$$

5. Filter und Abbildung sind vertauschbar, wenn das Filterprädikat Datenelemente und ihre Abbilder identisch bewertet:

$$\sigma_p(\mu_f(S)) = \mu_f(\sigma_p(S)) \Leftrightarrow p \equiv p \circ f \quad (5.19)$$

6. Filter können mit Vereinigungen, Schnitten und Differenzen vertauscht werden, wenn diese (Schnitt und Differenz) ohne gleitende Fenster berechnet werden:

$$\sigma_p(S_1 \cup S_2) = \sigma_p(S_1) \cup \sigma_p(S_2) \tag{5.20}$$
$$\sigma_p(S_1 \cap S_2) = \sigma_p(S_1) \cap \sigma_p(S_2) \tag{5.21}$$
$$\sigma_p(S_1 - S_2) = \sigma_p(S_1) - S_2 \tag{5.22}$$

Dahingegen sind Filter nur dann mit Schnitten und Differenzen, welche mit Hilfe von gleitenden Fenstern berechnet werden, vertauschbar, wenn die Fenster nach dem Einfügen eines Elements, welches durch das Filterprädikat herausgefiltert würde, keine Reorganisation vornehmen beziehungsweise im Zuge der Reorganisation kein Element aus dem Fenster entfernt werden würde.

$$\sigma_p(S_1 \cap_{p_{w_1} \vee \neg(p \circ \pi_2), p_{w_2} \vee \neg(p \circ \pi_2)} S_2) = \sigma_p(S_1) \cap_{p_{w_1}, p_{w_2}} \sigma_p(S_2) \tag{5.23}$$
$$\sigma_p(S_1 -_{p_w \vee \neg(p \circ \pi_2)} S_2) = \sigma_p(S_1) -_{p_w} S_2 \tag{5.24}$$

Dabei bezeichnet π_2 die Projektionsabbildung auf die zweite Stelle.

7. Filter können teilweise mit kartesischen Produkten und Verbünden vertauscht werden, wenn sich ihr Filterprädikat p in eine Konjunktion von Filterprädikaten p_1, p_2 und p_{12} umformulieren lässt, wobei die Prädikate p_1 und p_2 sich jeweils mit den Datenelementen der Eingabedatenströme S_1 und S_2 auswerten lassen und p_{12} ausschließlich auf einem Ergebnis der zugehörigen Operation ausgewertet werden kann:

$$\sigma_{p_1 \wedge p_2 \wedge p_{12}}(S_1 \times S_2) = \sigma_{p_{12}}(\sigma_{p_1}(S_1) \times \sigma_{p_2}(S_2)) \tag{5.25}$$
$$\sigma_{p_1 \wedge p_2 \wedge p_{12}}(S_1 \bowtie S_2) = \sigma_{p_{12}}(\sigma_{p_1}(S_1) \bowtie \sigma_{p_2}(S_2)) \tag{5.26}$$

Auch hier können Filter nur dann mit kartesischen Produkten und Verbünden, welche mit Hilfe gleitender Fenster berechnet werden, vertauscht werden, wenn die Reorganisation eines Fensters aufgrund eines Datenelements, welches durch das Filterprädikat herausgefiltert würde, keine Elemente aus dem Fenster verdrängt.

$$\sigma_{p_1 \wedge p_2 \wedge p_{12}}(S_1 \times_{p_{w_1} \vee \neg(p_1 \circ \pi_2), p_{w_2} \vee \neg(p_2 \circ \pi_2)} S_2) = \sigma_{p_{12}}(\sigma_{p_1}(S_1) \times_{p_{w_1}, p_{w_2}} \sigma_{p_2}(S_2)) \tag{5.27}$$
$$\sigma_{p_1 \wedge p_2 \wedge p_{12}}(S_1 \bowtie_{p_{w_1} \vee \neg(p_1 \circ \pi_2), p_{w_2} \vee \neg(p_2 \circ \pi_2)} S_2) = \sigma_{p_{12}}(\sigma_{p_1}(S_1) \bowtie_{p_{w_1}, p_{w_2}} \sigma_{p_2}(S_2)) \tag{5.28}$$

8. Abbildungen können mit Vereinigungen vertauscht werden:

$$\mu_f(S_1 \cup S_2) = \mu_f(S_1) \cup \mu_f(S_2) \tag{5.29}$$

9. Eine Besonderheit der Datenstromalgebra bildet die Gruppierung, welche im Gegensatz zur erweiterten relationalen Algebra kein blockierendes Verhalten an den

Tag legt, sondern anhand einer Funktion φ die eintreffenden Elemente Gruppen zuordnet und an die entsprechenden Ausgabedatenströme versendet. Daher lassen sich Filter und Gruppierungen vertauschen:

$$\forall i.\sigma_p(\pi_i(\gamma_\varphi(S))) = \pi_i(\gamma_\varphi(\sigma_p(S))) \tag{5.30}$$

Weiterhin lassen sich Abbildungen und Gruppierungen vertauschen, falls die Funktion φ Datenelemente und ihre Abbilder auf die selben Gruppen verteilt:

$$\forall i.\mu_f(\pi_i(\gamma_\varphi(S))) = \pi_i(\gamma_\varphi(\mu_f(S))) \quad \text{gdw.} \quad \varphi \equiv \varphi \circ f \tag{5.31}$$

Beweis 5.27: Seinen S_1, S_2 Datenströme über den jeweiligen Grundmengen A_1, A_2 mit $a_1 \in A_1$ und $a_2 \in A_2$. Seien p_1, p_2, p_{12} Filterprädikate, welche jeweils auf S_1, S_2 und $S_1 \times S_2$ ausgewertet werden, und p_{w_1}, p_{w_2} Fensterprädikate. Sei weiterhin $t \in T$ ein Zeitpunkt. Dann gilt:

$$\begin{aligned}
&(\sigma_{p_1 \wedge p_2 \wedge p_{12}}(S_1 \times_{p_{w_1} \vee \neg(p_1 \circ \pi_2), p_{w_2} \vee \neg(p_2 \circ \pi_2)} S_2))(t) = (a_1, a_2) \\
&\quad \text{gdw.} \quad (S_1 \times_{p_{w_1} \vee \neg(p_1 \circ \pi_2), p_{w_2} \vee \neg(p_2 \circ \pi_2)} S_2)(t) = (a_1, a_2) \wedge p_1(a_1) \wedge p_2(a_2) \wedge p_{12}((a_1, a_2)) \\
&\quad \text{gdw.} \quad (S_1(t) = a_1 \wedge \exists t_2 \in T.(t_2 \leq t \wedge S_2(t_2) = a_2 \\
&\qquad \wedge \forall t_2' \in T.(t_2 \leq t_2' \leq t \Rightarrow (p_{w_2} \vee \neg(p_2 \circ \pi_2))(a_2, S_2(t_2')))) \\
&\qquad \vee S_2(t) = a_2 \wedge \exists t_1 \in T.(t_1 \leq t \wedge S_1(t_1) = a_1 \\
&\qquad \wedge \forall t_1' \in T.(t_1 \leq t_1' \leq t \Rightarrow (p_{w_1} \vee \neg(p_1 \circ \pi_2))(a_1, S_1(t_1'))))) \\
&\qquad \wedge p_1(a_1) \wedge p_2(a_2) \wedge p_{12}((a_1, a_2)) \\
&\quad \text{gdw.} \quad (S_1(t) = a_1 \wedge \exists t_2 \in T.(t_2 \leq t \wedge S_2(t_2) = a_2 \\
&\qquad \wedge \forall t_2' \in T.(t_2 \leq t_2' \leq t \Rightarrow (p_{w_2}(a_2, S_2(t_2')) \vee \neg p_2(S_2(t_2'))))) \\
&\qquad \vee S_2(t) = a_2 \wedge \exists t_1 \in T.(t_1 \leq t \wedge S_1(t_1) = a_1 \\
&\qquad \wedge \forall t_1' \in T.(t_1 \leq t_1' \leq t \Rightarrow (p_{w_1}(a_1, S_1(t_1')) \vee \neg p_1(S_1(t_1')))))) \\
&\qquad \wedge p_1(a_1) \wedge p_2(a_2) \wedge p_{12}((a_1, a_2)) \\
&\quad \text{gdw.} \quad (S_1(t) = a_1 \wedge p_1(a_1) \wedge \exists t_2 \in T.(t_2 \leq t \wedge S_2(t_2) = a_2 \wedge p_2(a_2) \\
&\qquad \wedge \forall t_2' \in T.(t_2 \leq t_2' \leq t \Rightarrow (p_{w_2}(a_2, S_2(t_2')) \vee \neg p_2(S_2(t_2'))))) \\
&\qquad \vee S_2(t) = a_2 \wedge p_2(a_2) \wedge \exists t_1 \in T.(t_1 \leq t \wedge S_1(t_1) = a_1 \wedge p_1(a_1) \\
&\qquad \wedge \forall t_1' \in T.(t_1 \leq t_1' \leq t \Rightarrow (p_{w_1}(a_1, S_1(t_1')) \vee \neg p_1(S_1(t_1'))))) \\
&\qquad \wedge p_{12}((a_1, a_2)) \\
&\quad \text{gdw.} \quad ((\sigma_{p_1}(S_1))(t) = a_1 \wedge \exists t_2 \in T.(t_2 \leq t \wedge (\sigma_{p_2}(S_2))(t_2) = a_2 \\
&\qquad \wedge \forall t_2' \in T.(t_2 \leq t_2' \leq t \Rightarrow p_{w_2}(a_2, (\sigma_{p_2}(S_2))(t_2')))) \\
&\qquad \vee (\sigma_{p_2}(S_2))(t) = a_2 \wedge \exists t_1 \in T.(t_1 \leq t \wedge ((\sigma_{p_1}(S_1))(t_1) = a_1 \\
&\qquad \wedge \forall t_1' \in T.(t_1 \leq t_1' \leq t \Rightarrow p_{w_1}(a_1, (\sigma_{p_1}(S_1))(t_1'))))) \\
&\qquad \wedge p_{12}((a_1, a_2)) \\
&\quad \text{gdw.} \quad (\sigma_{p_1}(S_1) \times_{p_{w_1}, p_{w_2}} \sigma_{p_2}(S_2))(t) = (a_1, a_2) \wedge p_{12}((a_1, a_2)) \\
&\quad \text{gdw.} \quad (\sigma_{p_{12}}(\sigma_{p_1}(S_1) \times_{p_{w_1}, p_{w_2}} \sigma_{p_2}(S_2)))(t) = (a_1, a_2)
\end{aligned}$$

q.e.d.

5.2 Algebra logischer Operatoren

Ebenso wie die in der Bibliothek XXL integrierten *cursor*-Algebra stellt auch die Datenstromalgebra eine Reihe physischer Operatoren bereit. Der Einsatz dieser Algebren birgt damit das Problem, dass Anfragen durch physische Anfragepläne formuliert werden müssen. Die Nachteile dieser Vorgehensweise liegen auf der Hand – entweder muss der Anwender über profunde Kenntnisse der unterliegenden Datenbank beziehungsweise der verwendeten Datenströme verfügen, um effiziente Anfragepläne erzeugen und ihnen sinnvoll physische Algorithmen zuordnen zu können, oder der Anfrageplan muss nachträglich durch das Datenbank- beziehungsweise Datenstrommanagementsystem optimiert werden. Für die gebräuchlichere zweite Variante stellt jedoch die Formulierung physischer Anfragepläne keine günstige Ausgangslage dar, da diese im Zuge der Optimierung mit großer Wahrscheinlichkeit komplett umgestellt und den Operatoren neue Algorithmen zugewiesen werden. Außerdem müssen die physischen Operatoren im Zuge der logischen Optimierung von ihren Algorithmen abstrahiert werden, um rein algebraische Transformationen vornehmen zu können. Aus diesem Grund wird eine Algebra logischer Operatoren benötigt, mit deren Hilfe Anfragen durch logische Operatorbäume formuliert werden können.

Im Bereich relationaler Datenbanken wird dieser Bedarf durch die in Abschnitt 2.3.1 vorgestellte Algebra logischer Operatoren gedeckt. Jedoch weist diese logische Algebra einige gravierende Nachteile auf, welche sie für die Formulierung logischer Anfragen auf Datenströmen ungeeignet erscheinen lässt. Die Tatsache, dass die Anzahl der Eingaben eines logischen Operators stets fest gewählt werden muss und damit keine Möglichkeit besteht, eine beliebige Anzahl von Eingaben zuzulassen, um die effizienten *publish/subscribe*-Mechanismen von Datenströmen nachbilden zu können, stört nur am Rande. Schwerer wiegt dagegen schon der Umstand, dass die verfügbaren logischen Operatoren auf die Verarbeitung relationaler Daten festgelegt sind. Diese Festlegung erfolgt auf zweierlei Wegen. Zum einen können die logische Operatoren nur mit einer Art von Metadaten umgehen, welche eine Erweiterung der relationalen Metadaten darstellt, und zum anderen sind eine Reihe von Informationen, welche ausschließlich für die Optimierung relationaler Operatoren mit Hilfe der vorliegenden Anfrageoptimierung benötigt werden, fest in den Operator integriert. Damit erweisen sich die verfügbaren logischen Operatoren als zu starr für die Verwendung bei der Optimierung von Datenströmen.

Aus diesem Grund wird auch für die Anfrageoptimierung auf Datenströmen eine Algebra logischer Operatoren benötigt. Jedoch steht mit der vorliegenden Anfrageoptimierung ein äußerst flexibles Werkzeug zur Verfügung, welches durch eine Anpassung der Optimierungsstrategie ohne weiteres auch zur Optimierung von Anfragen der Datenstromalgebra eingesetzt werden kann. Daher macht es keinen Sinn, eine weitere logische Algebra zu entwickeln, welche für die Formulierung von Anfragen auf aktiven Datenquellen eingesetzt werden kann, da für deren Optimierung entweder die bestehende Anfrageoptimierung an die Verwendung der neuen logischen Operatoren angepasst oder eine komplett neue Anfrageoptimierung implementiert werden müsste. Die logische Folge beider Vorgehensweisen wären zwei relativ ähnliche Systeme zur Anfra-

geoptimierung, die jedoch untereinander vollkommen inkompatibel wären. Stattdessen zeigt eine Verallgemeinerung der logischen Algebra mehr Aussicht auf Erfolg, um das Potential der bestehenden Anfrageoptimierung ausschöpfen zu können. Die Integration einer solchen Algebra allgemeiner logischer Operatoren hat zwar immer noch weitgehende Änderungen der bestehenden Anfrageoptimierung zur Folge, jedoch entsteht so ein System, welches zur Optimierung beliebiger Anfragen eingesetzt werden kann.

Zu diesem Zweck wird grundsätzlich ein Rahmenwerk benötigt, mit dessen Hilfe Operatorgraphen formuliert werden können. Um die flexiblen *publish/subscribe*-Mechanismen der Datenstromalgebra unterstützen zu können, müssen die Knoten eines solchen Graphen in der Lage sein, eine beliebige Anzahl von Eltern- und Kindknoten zu verwalten. Alle weiteren Informationen, welche nichts mit der Verwaltung seiner Eltern- und Kindknoten zu tun haben, schränken die Wiederverwendbarkeit des Knotens zu stark ein. So hat zum Beispiel die feste Zuordnung eines Operatortyps zu einem Knoten des Rahmenwerks zur Folge, dass dieses nicht mehr für die Modellierung von Prädikaten verwendet werden kann. Diese Vorgehensweise erfordert jedoch das Vorhandensein einer flexiblen Möglichkeit, mit deren Hilfe einem Knoten die Eigenschaften des Objekts, welches er repräsentieren soll, zugewiesen werden können. Die Bereitstellung eines universellen Metadatenkonzeptes bietet eine Möglichkeit, diesen Anforderungen genüge zu tun. Dabei werden alle Informationen, welche das Objekt, das durch einen Knoten repräsentiert wird, beschreiben, als Metadaten betrachtet und in Form eines Metadatenobjekts im Knoten abgelegt. Damit wird der Knoten selbst unabhängig von dem zugehörigen repräsentierten Objekt und kann durch Anpassung der Metadaten für die Repräsentation beliebiger Objekte wiederverwendet werden.

5.2.1 Dynamische Metadaten

Für die im vorherigen Abschnitt beschriebene Verwendung der Metadaten wird ein Konzept benötigt, dass es erlaubt, Metadaten für jeden möglichen Anwendungsfall zu generieren und im Laufe der Optimierung zu variieren. Zusätzlich wäre es wünschenswert, Metadaten zur Laufzeit dynamisch erweitern zu können. Mit Hilfe dieser Eigenschaft könnten so im Laufe der Anfrageoptimierung neu gewonnene Informationen den Metadaten hinzugefügt werden, ohne dass die zur Speicherung dieser Informationen benötigten Strukturen bereits zu Beginn der Optimierung bekannt und innerhalb der Metadaten verfügbar sein müssen.

Durch die Bereitstellung eines Basis-Metadatenobjekts und dessen Erweiterung, wie im Rahmen der bestehenden logischen Algebra geschehen, lassen sich diese Anforderungen nur äußerst unzureichend umsetzen. Bereits das Hinzufügen einer einzelnen Information zu den Metadaten erfordert die Erweiterung zumindest eines der Metadatenobjekte. Damit stehen die Strukturen zur Speicherung dieser Informationen bereits bei der Erzeugung der Metadaten zur Verfügung und können potentiell zugegriffen werden, auch wenn die zugehörigen Informationen erst in einer späteren Phase der Anfrageoptimierung erzeugt und gespeichert werden. Dies erfordert zusätzlich explizite Sicherheitsvorkehrungen, um den Zugriff auf noch nicht verfügbare Informationen zu unterbinden. Zusätzlich müssen alle in einer Anfrage verwendeten Datenquellen

dahingehend geändert werden, dass sie fortan statt ihrer bisherigen Metadaten die erweiterte Version zur Verfügung stellen. Damit stellt die Erweiterung von Metadaten, selbst wenn eine dynamische Erweiterbarkeit der Metadaten unberücksichtigt bleibt, keine zufrieden stellende Lösung des Problems dar.

Aus diesem Grund wird ein neues Metadatenkonzept benötigt, welches die beschriebenen Anforderungen besser erfüllt. Dabei scheint eine Aggregation[1] der einzelnen Informationen besser geeignet zu sein, um diese gemeinsam zur Verfügung zu stellen, als eine starre Erweiterung der Metadaten um diese Informationen. Zu diesem Zweck wird das eigentliche Metadatenobjekt durch eine Art Container ersetzt, welcher Metadatenfragmente, d. h. kleine Einheiten zusammengehöriger Informationen, sammelt und gemeinsam zur Verfügung stellt. Mit Hilfe eines solchen Containers lassen sich somit einzelne Metadatenfragmente zu einem gemeinsamen Metadatenobjekt zusammensetzen. Durch die Verwendung einer dynamischen Datenstruktur zur Speicherung der aggregierten Metadatenfragmente wird weiterhin erzielt, dass neue Metadatenfragmente zur Laufzeit dem Metadatenobjekt angefügt oder Metadatenfragmente, welche nicht mehr benötigt werden, wieder von ihm abgetrennt werden können. Um einen logischen Operator im oben beschriebenen Rahmenwerk zu beschreiben, kann somit ein Metadatenfragment, welches den Typ des Operators enthält, eines, welches die Parameter des Operators beschreibt, und eines, welches die relationalen Metadaten des Ergebnisses des Operators enthält, zu einem gemeinsamen Metadatenobjekt zusammengefügt und vom Knoten bereitgestellt werden. Wird dem logischen Operator im Rahmen der physischen Optimierung schließlich ein Algorithmus zugewiesen, so kann diese Information wiederum in einem Metadatenfragment gespeichert und dem Metadatenobjekt angefügt werden.

Gerade in Verbindung mit der Anfrageoptimierung ist hierbei die automatische Erzeugung von Metadaten von besonderem Interesse. Während bei der Erweiterung eines Metadatenobjekts stets bekannt ist, welche Informationen ein Metadatenobjekt enthält, und sich damit Verfahren angeben lassen, wie diese Informationen gewonnen werden können, erweist sich diese Aufgabe bei dynamischen Metadaten als schwieriger. Zunächst benötigt jedes einzelne Metadatenfragment ein solches Verfahren, welches die Informationen des Metadatenfragments bestimmt. Um die dynamischen Eigenschaften des Metadatenobjekts nicht einzuschränken, müssen diese Verfahren ebenfalls dynamisch verwaltet werden. Zusätzlich muss die Möglichkeit bestehen, Abhängigkeiten in der Erzeugung der Metadatenfragmente zu definieren, um sicherzustellen, dass Informationen von Metadatenfragmenten, welche für die Erzeugung weiterer Metadatenfragmente benötigt werden, auch vor diesen bestimmt werden. So muss zum Beispiel sichergestellt werden, dass bei der Erzeugung der Metadaten einer Projektion stets das Metadatenfragment, welches deren projizierte Attribute enthält, vor dem Metadatenfragment, welches die relationalen Metadaten der Projektion beschreibt, erzeugt wird, da für die Erzeugung des zweiten Metadatenfragments die Informationen des ersten benötigt werden. Dabei gilt es natürlich zu beachten, dass keine Zyklen in dem durch

[1] Bei der Aggregation handelt es sich um eine Beziehung zwischen Objekten, wobei Objekte, die sogenannten Komponenten, als Teil eines weiteren Objekts, des sogenannten Aggregats, aufgefasst werden.

die gegenseitigen Abhängigkeiten definierten Graphen entstehen.

5.3 Statische Optimierung

Nun, da die algebraischen Eigenschaften der Operationen der Datenstromalgebra feststehen und eine Algebra allgemeiner logischer Operatoren für die Formulierung von Anfragen zur Verfügung steht, folgt die Betrachtung der eigentlichen Optimierung. Dabei liegt das Augenmerk der Anfrageoptimierung auf Datenströmen auf der anfragenübergreifenden Optimierung. Der Grund hierfür liegt in dem Umstand, dass einer Anfrage, welche an das Datenbanksystem gerichtet wird, mit den im System aktiven Anfragen ein Vielzahl Datenquellen zur Verfügung stehen, welche optimal genutzt werden wollen. Daher ist das Ziel einer Anfrageoptimierung auf Datenströmen hauptsächlich darin zu sehen, neue Anfragen möglichst effizient in den Anfragegraphen der aktiven Anfragen einzufügen.

Bei der Optimierung von Datenströmen beschränkt man sich auf jene Datenströme, welche relationale Daten liefern, und ausschließlich Zeitfenster zum Zweck der Approximation einsetzen, obwohl die zur Verfügung stehende Datenstromalgebra diesen Einschränkungen nicht unterworfen ist. Desweiteren beschränkt man die zulässigen Anfragen an die aktiven Datenquellen auf SPJ-Anfragen. Die erste Einschränkung hat zur Folge, dass Datenströme mittels relationaler Metadaten Informationen über die Struktur der von ihnen gelieferten Daten bereitstellen können. Dies dient dem Zweck, auch Transformationsregeln, welche gewissen Eigenschaften von des Eingaben des zu transformierenden Operators verlangen, bei der Anfrageoptimierung einsetzen zu können. Dahingegen bewirkt die zweite Einschränkung, dass die Menge der Fensterprädikate, welche innerhalb einer Anfrage auftreten können, auf einen einzigen Typ beschränkt wird. Dadurch hängt die lokale Speicherung von Datenelementen allein von deren zeitlichem Eintreffen ab und Transformationen zustandsbehafteter Operatoren vereinfachen sich stark. Da bei der Verarbeitung von Datenelementen aus mehreren aktiven Datenquellen meist ein zeitlicher Zusammenhang der Daten gefordert wird, stellt diese Einschränkung keine große Beeinträchtigung der möglichen Anfragen dar. Die letzte Einschränkung ist in der Datenbankforschung weit verbreitet und dient dem Zweck, zunächst die Komplexität des Problems zu begrenzen.

Zur Repräsentation von Anfragen während der Optimierung werden die in Abschnitt 4.1.2 vorgestellten expandierten UND/ODER-Graphen verwendet, welche auch als Grundlage für die in [GD87] erläuterte Anfrageoptimierung in Volcano dienen. Hierbei erweist sich die Verwendung der allgemeinen logischen Algebra als besonderer Vorteil, da sich einerseits die vorhandenen logischen Operatoren direkt als Operationsknoten verwenden lassen und mit Hilfe eines kleinen Metadatenfragments aus den Knoten des allgemeinen Rahmenwerks Äquivalenzknoten erzeugt werden können. Die Erzeugung eines UND/ODER-Graphen aus einem übergebenen Operatorgraphen kann als Teil der Normalisierung angesehen werden und stellt eine triviale Aufgabe dar. Da kontinuierliche Anfragen unter Zuhilfenahme der effizienten *publish/subscribe*-Mechanismen von Datenströmen eine problemlose gemeinsame Nutzung einmal be-

rechneter Ergebnisse ermöglichen, bietet es sich an, die Optimierung um Techniken der anfragenübergreifenden Optimierung zu erweitern. Hierfür eignen sich die im gleichen Abschnitt vorgestellten Erweiterungen der Expansion von UND/ODER-Graphen, welche in [RSSB00] veröffentlicht wurden, besonders. In den folgenden Abschnitten werden die einzelnen Phasen der Anfrageoptimierung auf Datenströmen näher erläutert.

5.3.1 Logische Optimierung

Die logische Optimierung erhält als Eingabe einen Anfrageplan in Form eines UND/ODER-Graphen und unterteilt sich in zwei Phasen. Dabei besteht die Aufgabe der ersten Phase in der Expansion des zu optimierenden Anfrageplans. Zu diesem Zweck werden mit Hilfe der bestehenden regelbasierten Anfrageoptimierung alle möglichen Transformationen auf die Operatoren des Anfrageplans angewendet und die entstehenden äquivalenten Teilanfragen an den Äquivalenzknoten des ursprünglichen Teilanfrageplans mit angehängt. Hierbei bedarf es einer effizienten Verwaltung der auf einen Operator angewendeten Transformationsregeln, um zu verhindern, dass zueinander inverse Transformationen mehrfach in Folge auf einen Operator angewendet werden. Zusätzlich zur herkömmlichen Expansion wird die in [RSSB00] vorgestellte Erweiterung zur Identifikation und Nutzung von Implikationen eingesetzt. Im Rahmen dieser Erweiterung wird, sobald ein Äquivalenzknoten e entdeckt wird, der mehreren Selektionen σ_{p_i} als Eingabe dient, eine neue Selektion $\sigma_{p'}$ erzeugt, deren Selektionsbedingung p' durch die Disjunktion der Selektionsbedingungen p_i der Selektionen σ_{p_i} gebildet wird. Diese neue Selektion $\sigma_{p'}$ erhält nun e als Eingabe und dient wiederum den Selektionen σ_{p_i} als Eingabe. Die zweite vorgestellt Erweiterung, mit deren Hilfe Äquivalenzknoten, deren Anfragepläne zwar syntaktisch verschieden aber semantisch äquivalent sind, erkannt und zu einem zusammengeführt werden, kann ebenfalls im Rahmen der Expansion durchgeführt werden. Sie kann jedoch auch ignoriert werden, da durch das Einfügen des Anfrageplans in den Anfragegraphen des Datenstrommanagementsystem implizit semantisch äquivalente Äquivalenzknoten erkannt und entfernt werden.

Die Transformationsregeln, welche im Laufe der Expansion auf den Anfrageplan angewendet werden, entsprechen im Grunde den in [Ber02] vorgestellten Transformationsregeln. Jedoch ist bei der Transformation von Verbünden aufgrund der Verwendung von Zeitfenstern besondere Vorsicht geboten. Im Folgenden werden die Transformationsregeln, welche der Expansion zur Verfügung stehen, namentlich erwähnt. Regeln, welche nicht in [Ber02] erwähnt werden oder deren Definition sich von der dortigen unterscheidet, werden neu definiert.

1. Umbenennung
   (a) Umbenennung löschen
   (b) Umbenennungen vereinigen
   (c) Umbenennung – Projektion tauschen (1)
   (d) Umbenennung – Projektion tauschen (2)
   (e) Umbenennung – Selektion tauschen

(f) Umbenennung – Verbund tauschen

Input:	$\rho_{B_1 \leftarrow A_1,\ldots,B_n \leftarrow A_n}(S_1 \bowtie_{p_w,p_w} S_2)$
Bedingung:	$true$
Output:	$\rho_{B_{i_1} \leftarrow A_{i_1},\ldots,B_{i_{n_i}} \leftarrow A_{i_{n_i}}}(S_1) \quad \bowtie_{p_w,p_w} \quad \rho_{B_{j_1} \leftarrow A_{j_1},\ldots,B_{j_{n_j}} \leftarrow A_{j_{n_j}}}(S_2)$, wobei die A_{i_k}, $1 \leq i_k \leq i_{n_i} \leq n$, und A_{j_k}, $1 \leq j_k \leq i_{n_j} \leq n$, jeweils die Attribute bezeichnen, welche in $RS(S_1)$ beziehungsweise $RS(S_2)$ vorkommen

Dabei sind die Umbenennungen $\rho_{B_{i_1} \leftarrow A_{i_1},\ldots,B_{i_{n_i}} \leftarrow A_{i_{n_i}}}$ und $\rho_{B_{j_1} \leftarrow A_{j_1},\ldots,B_{j_{n_j}} \leftarrow A_{j_{n_j}}}$ jeweils auch auf die Daten der durch p_w definierten Zeitfenster anzuwenden, um den Zustand des Verbunds konsistent zu halten.

2. Projektion

(a) Projektion löschen

(b) Projektionen vereinigen

(c) Projektion – Umbenennung tauschen

(d) Projektion – Selektion tauschen (1)

(e) Projektion – Selektion tauschen (2)

(f) Projektion – Verbund tauschen

Input:	$\pi_{\mathcal{A}}(S_1 \bowtie_{p_w,p_w} S_2)$
Bedingung:	$true$
Output:	$\pi_{\mathcal{A}}(\pi_{(\mathcal{A} \cup RS(S_2)) \cap RS(S_1)}(S_1) \bowtie_{p_w,p_w} \pi_{(\mathcal{A} \cup RS(S_1)) \cap RS(S_2)}(S_2))$

Dabei sind die Projektionen $\pi_{(\mathcal{A} \cup RS(S_2)) \cap RS(S_1)}$ und $\pi_{(\mathcal{A} \cup RS(S_1)) \cap RS(S_2)}$ jeweils auch auf die Daten der durch p_w definierten Zeitfenster anzuwenden, um den Zustand des Verbunds konsistent zu halten.

3. Selektion

(a) Selektion löschen

(b) Selektionen vereinigen

(c) Selektion – Umbenennung tauschen

(d) Selektion – Projektion tauschen

(e) Selektion – Verbund tauschen

Input:	$\sigma_{p_1 \wedge p_2 \wedge p_{12}}(S_1 \bowtie_{p_w,p_w} S_2)$
Bedingung:	$true$
Output:	$\sigma_{p_{12}}(\sigma_{p_1}(S_1) \bowtie_{p_w,p_w} \sigma_{p_2}(S_2))$, wobei p_1, p_2 und p_{12} jeweils jeweils Prädikate bezeichnen, welche auf $RS(S_1)$, $RS(S_2)$ beziehungsweise $RS(S_1) \cup RS(S_2)$ ausgewertet werden können

Dabei sind die Selektionen σ_{p_1} und σ_{p_2} jeweils auch auf die Daten der durch p_w definierten Zeitfenster anzuwenden, um den Zustand des Verbunds konsistent zu halten.

4. Verbünde

   (a) Verbund – Selektion tauschen

Input:	$\sigma_{p_1}(S_1) \bowtie_{p_w,p_w} \sigma_{p_2}(S_2)$
Bedingung:	$true$
Output:	$\sigma_{p_1 \wedge p_2}(S_1 \bowtie_{p_w,p_w} S_2)$

   (b) Verbund – Verbund tauschen

Input:	$S_1 \bowtie_{p_w,p_w} (S_2 \bowtie_{p_w,p_w} S_3)$
Bedingung:	$true$
Output:	$(S_1 \bowtie_{p_w,p_w} S_2) \bowtie_{p_w,p_w} S_3$

   Dabei sind die durch p_w definierten Zeitfenster W_i mit ihren zugehörigen Eingaben S_i zu verschieben, damit deren Zustand erhalten bleibt. Weiterhin ist das Zeitfenster W_{12} über die Ergebnisse des zuerst ausgeführten Verbunds $S_1 \bowtie_{p_w,p_w} S_2$ durch einen Verbund über die Daten der Zeitfenster W_1 und W_2 zu erzeugen.

Im Rahmen der zweiten Phase wird der expandierte Anfrageplan in den Anfragegraphen der im Datenstrommanagementsystem aktiv ablaufenden Anfragen eingefügt. Hierfür sind lediglich zwei einfache Algorithmen notwendig. Algorithmus 5.1 bestimmt die gemeinsamen Teilanfragepläne eines Anfrageplans und eines Anfragegraphen. Mit Hilfe des rekursive Aufrufs in Zeile 8, wird der Anfrageplan zunächst *top-down* durchlaufen, um dann *bottom-up* die gemeinsamen Teilanfragepläne aufzubauen. Trifft der Algorithmus im Anfrageplan auf eine Datenquelle S, so bricht die Rekursion ab und es wird überprüft, ob diese Datenquelle bereits im Anfragegraphen vorhanden ist. Ist dies der Fall, so wird die Information, dass die Datenquelle S bereits im Anfragegraphen vorhanden ist, den Metadaten von S hinzugefügt (Zeile 3). Sind alle Eingaben eines Operationsknotens O bereits im Anfragegraphen vorhanden, so wird nach einem gemeinsamen Elternknoten der Eingaben im Anfragegraphen gesucht, welcher O entspricht. Auch hier werden die Metadaten genutzt, um O einen entsprechenden Knoten im Anfragegraphen zuzuordnen (Zeile 12). Bei einem Äquivalenzknoten E wird dahingegen nur eine einzige Eingabe benötigt, welche bereits im Anfragegraphen vorhanden ist. Ist dies der Fall, so wird der Elternknoten der Eingabe im Anfragegraphen über die Metadaten von E diesem zugeordnet (Zeile 17).[2] Wurden auf diese Weise die gemeinsamen Teilanfragepläne des einzufügenden Anfrageplans und des Anfragegraphen im Datenbankmanagementsystem bestimmt, so kann mit Hilfe des Algorithmus 5.2 der Anfrageplan, welcher die geringsten Kosten verursacht, gewählt werden und in den

[2]Da Operationsknoten nur jeweils einen Äquivalenzknoten als Elternknoten besitzen können, entfällt hier die Entscheidung, welcher Elternknoten dem Knoten im Anfrageplan entspricht.

Algorithm 5.1 Bestimmung der gemeinsamen Teilanfragepläne eines Anfrageplans und eines Anfragegraphen

IN : E Äquivalenzknoten des einzufügenden Anfrageplans
$\mathcal{S}$ Menge der Datenquellen des Anfragegraphen

1: **if** $children(E) = \emptyset$ **then**
2: **if** $\exists \tilde{E} \in \mathcal{S}$ such that $\tilde{E} \equiv E$ **then**
3: add $\tilde{E}$ as `common node` to E's meta data
4: **end if**
5: **else**
6: **for all** operation nodes $O' \in children(E)$ **do**
7: **for all** equivalence nodes $E' \in children(O')$ **do**
8: get common query of E' by using $\mathcal{S}$
9: **end for**
10: **if** $\forall E' \in children(O')$ such that E''s meta data contains `common node` **then**
11: **if** $\exists \tilde{O'} \in \cap_{E' \in children(O')} parents($`common node` of $E')$ such that $\tilde{O'} \equiv O'$ **then**
12: add $\tilde{O'}$ as `common node` to O''s meta data
13: **end if**
14: **end if**
15: **end for**
16: **if** $\exists O' \in children(E)$ such that O's meta data contains `common node` **then**
17: add $\tilde{E} \in parents($`common node` of $O')$ as `common node` to E's meta data
18: **end if**
19: **end if**

Anfragegraphen eingefügt werden. Auch hier durchläuft der Algorithmus mittels des rekursiven Aufrufs in Zeile 6 den Anfrageplan *top-down*. Sobald er dabei auf einen Äquivalenzknoten E trifft, der bereits im Anfragegraphen vorhanden ist, wird der Knoten E bei seinen Elternknoten durch den entsprechenden Knoten des Anfragegraphen ersetzt (Zeile 2). Ansonsten wird der Kindknoten O von E gewählt, der die geringsten Kosten verursacht (Zeile 4) und mit dessen Eingaben der Algorithmus rekursiv aufgerufen.

Mittels des Algorithmus 5.3 lassen sich Anfragepläne wieder korrekt aus dem Anfragegraphen des Datenbankmanagementsystems ausschneiden, ohne gemeinsam verwendete Teilanfragen zu beschädigen. Dabei wird der Anfrageplan wiederum *top-down* durchlaufen und jeweils die Verbindung zwischen einem Äquivalenzknoten und seinem Elternknoten durchtrennt (Zeile 12). Trifft der Algorithmus dabei auf einen Äquivalenzknoten, der mehrfach verwendet wird, so endet die Rekursion an dieser Stelle.

5.3.2 Physische Optimierung

Nachdem die logische Struktur des Anfrageplans und dessen gemeinsame Teilanfragepläne feststehen, kommt nun der Zeitpunkt, den nicht wiederverwendeten Operatoren geeignete Algorithmen zuzuweisen. Auch in diesem Fall kann die bestehende Anfra-

Algorithm 5.2 Einfügen eines übergebenen Anfrageplans in einen Anfragegraphen

IN	: E	Äquivalenzknoten des einzufügenden Anfrageplans
	: O	Operationsknoten, in den E eingefügt werden soll

```
1: if E's meta data contains common node then
2:     replace O's children E by common node of E
3: else
4:     pick operation node O' ∈ children(E) which minimizes cost(O')
5:     for all equivalence nodes E' ∈ children(O') do
6:         paste query E' in O'
7:     end for
8: end if
```

Algorithm 5.3 Ausschneiden eines Anfrageplans aus einem Anfragegraphen

IN	: E	Äquivalenzknoten des auszuschneidenden Anfrageplans
	O	Operationsknoten, aus dem E ausgeschnitten werden soll
	$\mathcal{S}$	Menge der Datenquellen des Anfragegraphen

```
1: if parents(E) = {O} then
2:     if children(E) = ∅ then
3:         R ← S \ {E}
4:     else
5:         for all operation nodes O' ∈ children(E) do
6:             for all equivalence nodes E' ∈ children(O') do
7:                 cut query E' from O' by using S
8:             end for
9:         end for
10:    end if
11: end if
12: remove E from O
```

geoptimierung genutzt werden. Zu diesem Zweck werden, wie in Abschnitt 2.3.4 beschrieben, die mit einem bestimmten Operatortyp verbundenen Verfahren zur logischen Optimierung desselben mittels Funktionen gekapselt und der vorhandenen physischen Optimierung zur Verfügung gestellt.

Die Definition eines geeigneten Kostenmodells stellt dabei ein zentrales Problem der Anfrageoptimierung auf Datenströmen dar. Herkömmliche Kostenmodelle aus der Datenbankforschung schätzen meist die Größe der Eingaberelationen und die Verteilung ihrer Werte ab; da solche Informationen über Datenströme nur in den seltensten Fällen existieren, erscheint dieses Vorgehen als wenig sinnvoll. Die Anzahl der nicht wiederverwendeten Operatoren lässt sich dahingegen minimieren, wenn die Kosten eines einmalig verwendeten Operators mit 1 und eines wiederverwendeten Operators mit 0 bewertet werden. Vielversprechender scheinen jedoch Verfahren, welche die Kosten eines Operators mit Hilfe dessen Datenraten definieren, um so einen optimalen Da-

tenfluss im Anfragegraphen zu gewährleisten. Ansätze solcher Kostenmodelle werden in [VN02] und [KNV02] vorgestellt.

Im speziellen Fall der vorhandenen Datenstromalgebra stellt sich die logische Optimierung als relativ einfach heraus. Einerseits werden Verbünde im Rahmen der Expansion nicht zusammengefasst, so dass damit auch keine Joinpläne erzeugt werden müssen, und andererseits besteht für die logischen Operatoren keine Auswahl an physischen Algorithmen, wodurch die Wahl desselben besonders leicht fällt.

5.4 Dynamische Optimierung

Im Rahmen der dynamischen Optimierung gilt es nun, auf veränderte Bedingungen zu reagieren, um eine gleichbleibend gute Qualität der Anfragen zu garantieren. Ihre Aufgabe besteht dabei darin, den Anfragegraphen der im System aktiven Anfragen zu optimieren. Mit Hilfe der in den vorherigen Abschnitten vorgestellten Techniken lassen sich große Teile dieser Aufgabe auf Verfahren der statischen Optimierung zurückführen. Dabei wird zunächst der Anfragegraph des Systems wie im Rahmen der logischen Optimierung expandiert, so dass der Anfragegraph anschließend alle äquivalenten Anfragen der im System aktiven Anfragen repräsentiert. Dabei muss eine genaue Protokollierung der Transformationen, welche auf zustandsbehaftete Operatoren angewendet werden, durchgeführt werden, um diese Transformationen später auch auf die physischen Operatoren anwenden zu können. Durch die Erkennung semantisch äquivalenter Äquivalenzknoten während der Expansion kann auf einen Einsatz des Algorithmus 5.1 komplett verzichtet werden. Stattdessen werden die Kosten der einzelnen Operatoren in einem *bottom-up* Durchlauf durch den Anfragegraphen und die optimalen Anfragepläne in einem anschließenden *top-down* Durchlauf bestimmt.

Die Reorganisation aktiver physischer Anfragen – speziell die Umordnung zustandsbehafteter Operatoren – ist jedoch mit einigen Schwierigkeiten verbunden. Die einfachste Lösung besteht darin, die Daten der Datenquellen des Anfragegraphen temporär zu ignorieren und abzuwarten, bis alle Zeitfenster im Anfragegraphen leer gelaufen sind. Zu diesem Zeitpunkt hat der Anfragegraph einen Ruhezustand erreicht und kann beliebig umstrukturiert werden. Diese Vorgehensweise ist jedoch nicht praktikabel, da sie einen langfristigen Stillstand aller im System aktiven Anfragen verursacht. Dahingegen lassen sich die in Abschnitt 5.3.1 vorgestellten Transformationsregeln auf aktive zustandsbehaftete Operatoren anwenden, deren Zeitfenster durch identische Fensterprädikate p_w definiert werden. Zu diesem Zweck müssen jedoch zunächst die Zeitfenster aneinander angepasst werden. Diese Anpassung kann einerseits durch ein Vergrößerung des kleineren oder eine Verkleinerung des größeren Zeitfensters erfolgen. Dabei hat die erste Variante den Nachteil, dass das zu vergrößernde Zeitfenster erst langsam gefüllt werden muss, bis die Transformationen durchgeführt werden können. Wird also ein Zeitfenster, welches die Daten der letzten Stunde speichert, auf zwei Stunden vergrößert, so benötigt dieser Vorgang eine Stunde, in der das Zeitfenster gefüllt wird. Dahingegen kann im Zuge der zweiten Variante das Zeitfenster einfach auf die gewünschte Größe reduziert werden. Sobald das nächste Element in das Zeitfenster eingefügt wird, werden alle äl-

teren Datenelemente daraus verdrängt. Dies hat jedoch eine sprunghafte Verringerung der Qualität des Operators zur Folge, da zur Berechnung von Ergebnissen schlagartig weniger Datenelemente zur Verfügung stehen.

5.5 Zusammenfassung

Im Rahmen dieses Kapitels wird ein Überblick über die Anfrageoptimierung auf Datenströmen vermittelt. Die Optimierung verfolgt dabei das Ziel, neue Anfragen in den durch die aktiven Anfragen des Datenbankmanagementsystems definierten Anfragegraphen derart einzufügen, dass gemeinsame Teilanfragen nur ein einziges Mal aufgewertet werden. Hierfür wird die Anfrage mittels eines UND/ODER-Graphen dargestellt und anschließend expandiert, um alle äquivalenten Anfragepläne der Anfrage verfügbar zu halten. Anschließend werden die gemeinsamen Teilanfragepläne *bottom-up* bestimmt und der Anfrageplan in den Anfragegraphen des Systems eingefügt. Die dynamische Optimierung nutzt die selben Techniken, um die Anfragegraphen der im System aktiven Anfragen zu re-optimieren.

Das Ziel, die im Rahmen der Bibliothek XXL angebotene Anfrageoptimierung relationaler Anfragen an passive Datenquellen weitgehend wiederzuverwenden, wird durch eine Algebra allgemeiner, logischer Operatoren und ein dynamisches Metadatenkonzept zur Spezialisierung dieser Operatoren verfolgt. Mit Hilfe dieser Werkzeuge kann die bestehende Anfrageoptimierung zu einer allgemeinen Anfrageoptimierung mit regelbasierter logischer Optimierung und flexibler physischer Optimierung erweitert werden.

Kapitel 6

Implementierung und Integration in XXL

Nachdem der Ablauf der Anfrageoptimierung auf Datenströmen feststeht und die für diese Aufgabe erforderlichen Verfahren und Techniken im vorherigen Kapitel erläutert wurden, wendet sich dieses Kapitel nun der konkreten Umsetzung dieser Verfahren und deren Integration in die Bibliothek XXL zu. Der Schwerpunkt liegt dabei nicht in der expliziten Betrachtung aller implementierten Klassen. Stattdessen sollen die grundlegenden Klassen der Implementierung vorgestellt und ihre Nutzung verdeutlicht werden.

Im Anschluss an diese Darstellung folgt ein tabellarischer Überblick über alle im Rahmen dieser Diplomarbeit implementierten Klassen, in dem diese nochmals im Einzelnen kurz erläutert werden.

6.1 Logische Anfragegraphen

Aufgrund der uneingeschränkten *publish/subscribe*-Mechanismen, welche auf Datenströmen zur Verfügung stehen, können Operatoren der Datenstromalgebra von einer beliebigen Anzahl von Datenquellen Daten beziehen und diese wiederum an eine beliebige Anzahl von Datensenken versenden. Daher werden für die Darstellung von Anfragen der Datenstromalgebra Anfragegraphen benötigt.

Zu diesem Zweck stellt das Paket `xxl.logical` mit der Klasse `Node` die Vorimplementierung eines Knotens in einem gerichteten Graphen zur Verfügung. Der Knoten selbst beschränkt sich dabei allein auf die Verwaltung der Eltern- und Kindknoten. Um sowohl Operatoren der Datenstromalgebra mit ihren flexiblen *publish/subscribe*-Mechanismen als auch Operatoren mit einer festen Anzahl von Eingaben unterstützen zu können, wird die Anzahl der Kindknoten bei der Konstruktion eines Knotens entweder festgelegt oder mittels der Konstanten `VARIABLE` variabel gehalten. Anschließend lassen sich die Kindknoten über die Methoden `addChild`, `getChild` und `removeChild` verwalten. Dabei wird jede dieser Methoden in zwei Varianten bereitgestellt. Bei der Verwaltung einer festen Anzahl von Kindknoten wird auf diese über einen Index zuge-

griffen, während bei einer variablen Anzahl der Kindknoten selbst benötigt wird, um ihn beispielsweise zu löschen. Das Einfügen eines Kindknoten erzeugt dabei eine doppelte Verkettung zwischen den Knoten, so dass mittels der Methoden `getChildren` und `getParents` über die Kind- beziehungsweise Elternknoten iteriert werden kann. Näher bestimmen lässt sich ein solcher Knoten ausschließlich über seine Metadaten, auf welche über die Methode `getMetaData` zugegriffen werden kann. Zuvor muss der Knoten jedoch noch mittels der Methode `open` geöffnet werden. Damit wird sichergestellt, dass die Metadaten eines Knotens verfügbar sind. Stehen dabei in einem Knoten noch keine Metadaten bereit, so werden diese über die abstrakte Methode `createMetaData` erzeugt.

6.1.1 Metadaten

Die Metadaten, welche für diese Vorgehensweise von zentraler Bedeutung sind, werden mittels der Klasse `CompositeMetaData` im Paket `xxl.util` bereitgestellt. Diese Klasse dient dem Zweck, Metadatenfragmente, welche in Form beliebiger Objekte vorliegen können, zu verwalten und zu einen gemeinsamen Metadatenobjekt zusammenzufügen. Mittels der Methode `add` lässt sich ein neues Metadatenfragment an das Metadatenobjekt anfügen. Dabei muss ein innerhalb des Metadatenobjekts eindeutiger Schlüssel spezifiziert werden, über den später wieder auf das Metadatenobjekt zugegriffen werden kann. Zu diesem Zweck verfügen die vorimplementierten Metadatenfragmente über eine Konstante `META_DATA_TYPE`, welche einen solchen Schlüssel zur Verfügung stellt. Die Methoden `replace`, `get` und `remove` benötigen wiederum diesen Schlüssel, um ihrer jeweiligen offensichtlichen Aufgabe an dem zugehörigen Metadatenfragment nachkommen zu können. Jedoch wird die Freiheit, Metadatenfragmente beliebigen Typs zu einem Metadatenobjekt zusammenfügen zu können, durch die Notwendigkeit erkauft, dass Anwender dieser Metadaten der Typ des Objekts, welches sich hinter einem Schlüssel verbirgt, bekannt sein muss.

Mit Hilfe dieser Metadaten können nun die Knoten des vorher erwähnten Graphen näher bestimmt werden. So stehen zum Beispiel mit den vordefinierten Metadatenfragmenten des Pakets `xxl.logical.operators.metaData` alle notwendigen Komponenten zur Verfügung, um einen Operator zu beschreiben. Zu diesem Zweck werden mittels eines Metadatenfragments vom Typ `OperatorMetaData` der Typ des Operators und die Anzahl seiner Eingaben festgelegt. Anschließend wird ein weiteres Metadatenfragment, welches den Operator näher beschreibt, erzeugt. So enthält ein Metadatenfragment vom Typ `ProjectionMetaData` beispielsweise die projizierten Spalten eines Projektionsoperators. Diese beiden Metadatenfragmente können nun zu den Metadaten eines Projektionsoperators zusammengefügt werden. Handelt es sich weiterhin um einen relationalen Operator, so können diese jederzeit als Metadatenfragment angefügt werden.

Von besonderem Interesse ist hierbei die automatische Erzeugung von Metadaten. So ist es beispielsweise wünschenswert, dass ein relationaler Operator seine relationalen Metadaten selbständig aus den Metadaten seiner Eingaben erzeugt. Um dies zu ermöglichen, werden die Verfahren, mit deren Hilfe die Informationen eines bestimmten Metadatenfragments gewonnen werden, in Funktionen gekapselt. Anschließend werden

diese Funktionen unter einem eindeutigen Schlüssel bei einem Objekt von Typ `Translator`, welches durch das Paket `xxl.logical.translation` bereitgestellt wird, registriert. Dabei besteht die Aufgabe dieser Klasse darin, bei einem Aufruf der abstrakten Methode `translate` die registrierten Funktionen auf ein übergebenes Objekt anzuwenden und deren Ergebnisse zurückzuliefern. Um diesen Mechanismus zu unterstützen, bieten die implementierten Metadatenfragmente der Operatoren bereits vorimplementierten Funktionen zur Erzeugung relationaler Metadaten an. Weiterhin stellt die Klasse `Operators` im Paket `xxl.logical.operators` eine Vorimplementierung eines solchen Übersetzers bereit, der die registrieren Funktionen auf die Metadaten eines übergebenen Operators anwendet, um diese zu initialisieren.

Die vorliegende Implementierung stellt alle notwendigen Hilfsmittel bereit, um Operatorgraphen und baumartige Darstellungen von Prädikaten zu erzeugen. Um diese Aufgabe noch zu vereinfachen, stellen die Klassen `Operators` in `xxl.logical.operators` und `Predicates` in `xxl.logical.predicates` statische Methoden zur Verfügung, mit deren Hilfe direkt Operatoren im Operatorgraphen beziehungsweise Prädikate in einer baumartigen Darstellung erzeugt werden können.

6.2 Komponenten der Anfrageoptimierung

Im Paket `xxl.system` wird mit der Klasse `QueryCoordinator` die zentrale Instanz der Anfrageverarbeitung bereitgestellt. Ihre Aufgabe besteht darin, mittels der Methode `addQuery` Anfragen entgegenzunehmen und in Form einer physischen Datenquelle, welche die optimierte Anfrage repräsentiert, zurückzuliefern. Dabei wird die eigentliche Anfrageoptimierung in der abstrakten Klasse `QueryModule` gekapselt, mit deren Hilfe Optimierungsstrategien einfach ausgetauscht werden können. Ebenso wie beim Anfragekoordinator werden dem Anfragemodul Anfragen mittels der Methode `addQuery` übergeben und in Form einer physischen Datenquelle zurückgeliefert. Mit den Klassen `DAGQueryModule` und `QueryShareableDAGQueryModule` stehen zwei Implementierungen des Anfragemoduls bereit. Dabei verwendet die erste Variante einen UND/ODER-Graphen, welcher keine gemeinsamen Teilanfragen ermittelt und wiederverwendet, um die aktiven Anfragen des Moduls zu verwalten. Die zweite Variante verwendet ebenfalls einen UND/ODER-Graphen zur Verwaltung der Anfragen, jedoch sind hier zusätzlich die Algorithmen zur Erkennung gemeinsamer Teilanfragen implementiert und werden genutzt, um solche Anfragen zu erkennen und wiederzuverwenden.

Die anschließende physische Optimierung wird in Form einer einfachen Übersetzung mittels der Klasse `QueryTranslator` durchgeführt, welche die bereits vorgestellte Klasse `Translator` erweitert. Zu diesem Zweck werden in der Klasse `QueryTranslator` Optimierungsfunktionen für die einzelnen Operatoren definiert und in einem *bottom-up* Durchlauf durch den Anfragegraphen auf dessen Operatoren angewendet. Da für die Operatoren der logischen Algebra jeweils nur eine einzige Implementierung existiert, besteht bisher keine Notwendigkeit einer Kostenanalyse. Diese lässt sich jedoch durch Verwendung der bestehenden physischen Optimierung im Paket `xxl.relationalLog.physical` problemlos erweitern.

6.3 Übersicht der implementierten Klassen

Schließlich soll dieser Abschnitt noch einen Überblick über die im Rahmen dieser Diplomarbeit implementierten Klassen und Schnittstellen vermitteln. Dabei wird die tabellarische Aufzählung der Klassen und Schnittstellen jeweils um eine kurze Beschreibung derselben ergänzt.

6.3.1 Paket `xxl.logical`

In dem Paket `xxl.logical` werden die Klassen, welche für die Erstellung gerichteter Graphen benötigt werden, bereitgestellt.

Klassen- bzw. Schnittstellenbezeichnung	Beschreibung
`Node`	Abstrakte Implementierung eines nicht näher bestimmten Knotens in einem gerichteten Graphen; nähere Bestimmung des Knotens nur über dessen Metadaten möglich
`NodeMetaData`	Metadatenfragment, welches Informationen über einen Knoten bereitstellt

Tabelle 6.1: Kurzbeschreibung der Klassen und Schnittstellen des Pakets `xxl.logical`

6.3.2 Paket `xxl.logical.operators`

Das Paket `xxl.logical.operators` enthält eine Erweiterung der abstrakten Klasse `Node` für logische Operatoren und bietet statische Methoden zu deren Erzeugung.

Klassen- bzw. Schnittstellenbezeichnung	Beschreibung
`Operator`	Implementierung eines logischen Operators auf Basis der abstrakten Klasse `Node`
`Operators`	Sammlung statischer Methoden für die Erzeugung spezieller logischer Operatoren und den Zugriff auf deren Metadaten

Tabelle 6.2: Kurzbeschreibung der Klassen und Schnittstellen des Pakets `xxl.logical.operators`

6.3.3 Paket `xxl.logical.operators.metaData`

Das Paket `xxl.logical.operators.metaData` stellt eine Reihe von Metadatenfragmenten für die Beschreibung von Operatoren bereit. Dabei bestimmt die Klasse `OperatorMetaData` den Typ eines Operators. Im Metadatenobjekt wird dieser Typ zusätzlich als Schlüssel für die näheren Eigenschaften des Operators genutzt, welche mit Hilfe der restlichen Klassen des Pakets beschrieben werden können.

Klassen- bzw. Schnittstellenbezeichnung	**Beschreibung**
`FileOperatorMetaData`	Metadatenfragment, welches Informationen über einen relationalen Operator bereitstellt, der Daten aus einer Datei liest und aktiv versendet
`NaturalJoinMetaData`	Metadatenfragment, welches Informationen über einen natürlichen Verbund bereitstellt
`OperatorMetaData`	Metadatenfragment, welches Informationen über einen beliebigen Operator bereitstellt
`PrintOperatorMetaData`	Metadatenfragment, welches Informationen über einen relationalen Operator bereitstellt, der Daten einer Datenquelle konsumiert und auf dem Bildschirm ausgibt
`ProjectionMetaData`	Metadatenfragment, welches Informationen über eine Projektion bereitstellt
`RenamingMetaData`	Metadatenfragment, welches Informationen über eine Umbenennung bereitstellt
`SelectionMetaData`	Metadatenfragment, welches Informationen über eine Selektion bereitstellt

Tabelle 6.3: Kurzbeschreibung der Klassen und Schnittstellen des Pakets `xxl.logical.operators.metaData`

6.3.4 Paket `xxl.logical.predicates`

Das Paket `xxl.logical.predicates` enthält eine Erweiterung der abstrakten Klasse `Node` für logische Prädikate und bietet statische Methoden zu deren Erzeugung.

Klassen- bzw. Schnittstellenbezeichnung	Beschreibung
`Predicate`	Implementierung eines logischen Prädikats auf Basis der abstrakten Klasse `Node`
`Predicates`	Sammlung statischer Methoden für die Erzeugung spezieller Prädikate und den Zugriff auf deren Metadaten

Tabelle 6.4: Kurzbeschreibung der Klassen und Schnittstellen des Pakets `xxl.logical.predicates`

6.3.5 Paket `xxl.logical.predicates.metaData`

Das Paket `xxl.logical.predicates.metaData` stellt eine Reihe von Metadatenfragmenten für die Beschreibung von Prädikaten bereit. Dabei bestimmt die Klasse `PredicateMetaData` den Typ eines Prädikats. Im Metadatenobjekt wird dieser Typ zusätzlich als Schlüssel für die näheren Eigenschaften des Prädikats genutzt, welche mit Hilfe der restlichen Klassen des Pakets beschrieben werden können.

Klassen- bzw. Schnittstellenbezeichnung	Beschreibung
`ColumnComparisonPredicateMetaData`	Metadatenfragment, welches Informationen über ein Prädikat bereitstellt, das die Werte zweier Spalten eines Tupels miteinander vergleicht
`ComparisonPredicateMetaData`	Abstrakte Implementierung eines Metadatenfragments, welches Informationen über ein Prädikat bereitstellt, das einen Vergleich auf einer Spalte eines Tupels durchführt
`ConstantComparisonPredicateMetaData`	Metadatenfragment, welches Informationen über ein Prädikat bereitstellt, das den Wert einer Spalte eines Tupels mit einer Konstanten vergleicht
`MultiPredicateMetaData`	Metadatenfragment, welches Informationen über ein Prädikat bereitstellt, das mehrere Prädikate miteinander verknüpft
`NullComparisonPredicateMetaData`	Metadatenfragment, welches Informationen über ein Prädikat bereitstellt, das den Wert einer Spalte eines Tupels auf den Wert `null` hin untersucht

`PredicateMetaData`	Metadatenfragment, welches Informationen über ein beliebiges Prädikat bereitstellt

Tabelle 6.5: Kurzbeschreibung der Klassen und Schnittstellen des Pakets `xxl.logical.predicates.metaData`

6.3.6 Paket `xxl.logical.translation`

Das Paket `xxl.logical.translation` enthält Klassen, welche für die automatische Übersetzung von Objekten benötigt werden.

Klassen- bzw. Schnittstellenbezeichnung	**Beschreibung**
`Translator`	Abstrakte Implementierung einer Klasse, welche ein mit Hilfe von Metadaten beschriebenes Objekt übersetzt

Tabelle 6.6: Kurzbeschreibung der Klassen und Schnittstellen des Pakets `xxl.logical.translation`

6.3.7 Paket `xxl.pipes`

Das bestehende Paket `xxl.pipes` wird um Adapter für Datenquellen, Operatoren und Datensenken erweitert.

Klassen- bzw. Schnittstellenbezeichnung	**Beschreibung**
`DecoratorPipe`	Umsetzung des Entwurfsmusters Adapter für einen physischen Operator der Datenstromalgebra
`DecoratorSink`	Umsetzung des Entwurfsmusters Adapter für eine physische Datensenke der Datenstromalgebra
`DecoratorSource`	Umsetzung des Entwurfsmusters Adapter für eine physische Datenquelle der Datenstromalgebra

Tabelle 6.7: Kurzbeschreibung der neuen Klassen und Schnittstellen des Pakets `xxl.pipes`

Da aufgrund der wechselseitigen Verknüpfung zwischen den Operatoren und deren bidirektionaler Kommunikation nicht sichergestellt werden kann, dass Zugriffe stets über den Adapter erfolgen, sind diese Adapter nur für die Erweiterung der Schnittstellen von Datenquelle, Operator und Datensenke anwendbar.

6.3.8 Paket `xxl.pipes.relational`

In dem Paket `xxl.pipes.relational` werden spezielle Operatoren der Datenstromalgebra, welche auf die Verarbeitung von relationalen Daten spezialisiert sind, bereitgestellt.

Klassen- bzw. Schnittstellenbezeichnung	**Beschreibung**
`ResultSetPrinter`	Implementierung einer physischen Datensenke der Datenstromalgebra, welche relationale Daten konsumiert und über den Bildschirm ausgibt

Tabelle 6.8: Kurzbeschreibung der Klassen und Schnittstellen des Pakets `xxl.pipes.relational`

6.3.9 Paket `xxl.relational`

Das bestehende Paket `xxl.relational` wird um Repräsentationen relationaler Metadaten erweitert.

Klassen- bzw. Schnittstellenbezeichnung	**Beschreibung**
`StoredResultSetMetaData`	Implementierung der Schnittstelle `ResultSetMetaData` aus dem Paket `java.sql`, welche ihre Informationen explizit speichert
`WrappedResultSetMetaData`	Umsetzung des Entwurfsmusters Adapter für Objekte vom Typ `ResultSetMetaData` aus dem Paket `java.sql` mit einigen Hilfsfunktionen auf Basis der Klasse `ObjectMetaData` aus dem Paket `xxl.util`

Tabelle 6.9: Kurzbeschreibung der neuen Klassen und Schnittstellen des Pakets `xxl.relational`

6.3.10 Paket `xxl.relationalLog`

Das bestehende Paket `xxl.relationalLog` wird um Repräsentationen statistischer Metadaten erweitert.

Klassen- bzw. Schnittstellenbezeichnung	Beschreibung
`SizeDistributionableMetaData`	Umsetzung des Entwurfsmusters Adapter für Objekte vom Typ `SizeDistributionable` aus dem Paket `xxl.relationalLog` auf Basis der Klasse `ObjectMetaData` aus dem Paket `xxl.util`
`StoredSizeDistributionableMetaData`	Implementierung der Schnittstelle `SizeDistributionable` aus dem Paket `xxl.relationalLog`, welche ihre Informationen explizit speichert

Tabelle 6.10: Kurzbeschreibung der neuen Klassen und Schnittstellen des Pakets `xxl.relationalLog`

6.3.11 Paket `xxl.system`

In dem Paket `xxl.system` werden die Komponenten der Anfrageoptimierung auf Datenströmen bereitgestellt

Klassen- bzw. Schnittstellenbezeichnung	Beschreibung
`DAGQueryModule`	Implementierung der abstrakten Klasse `QueryModule`, welche aktive Anfragen in einem Anfragegraphen ohne Wiederverwendung gemeinsamer Teilanfragen verwaltet
`EquivalenceNode`	Implementierung eines Äquivalenzknotens auf Basis der abstrakten Klasse `xxl.logical.Node`
`EquivalenceNodeMetaData`	Metadatenfragment, welches Informationen über einen Äquivalenzknoten bereitstellt und ihm eine physische Anfrage zuordnen kann
`LogicalQueryMetaData`	Metadatenfragment, welches Informationen über den logischen Anfragegraphen einer aktiven physischen Anfrage bereitstellt

`Nodes`	Sammlung statischer Methoden für den Zugriff auf die Metadaten eines Knotens
`QueryCoordinator`	Anfragekoordinatoren dienen dem Zweck, die Ausführung von Anfragen zu koordinieren; die Anfragen selbst werden dabei in einem Objekt der Klasse `QueryModule` verwaltet
`QueryModule`	Abstrakte Implementierung eines austauschbaren Moduls zur Verwaltung von aktiven Anfragen
`QueryShareableDAGQueryModule`	Implementierung der abstrakten Klasse `QueryModule`, welche aktive Anfragen in einem Anfragegraphen unter Wiederverwendung gemeinsamer Teilanfragen verwaltet
`QueryTranslator`	Implementierung der abstrakten Klasse `Translator` aus dem Paket `xxl.logical.translation`, mit deren Hilfe ein logischer Anfragegraph in eine physische Anfrage übersetzt werden kann; stellt die Optimierungsfunktionen für die physische Optimierung bereit
`SharingMetaData`	Metadatenfragment, welches Informationen über die Wiederverwendung gemeinsamer Teilanfragen bereitstellt

Tabelle 6.11: Kurzbeschreibung der Klassen und Schnittstellen des Pakets `xxl.system`

6.3.12 Paket `xxl.util`

Das Paket `xxl.util` stellt die Komponenten des dynamischen Metadatenkonzeptes bereit.

Klassen- bzw. Schnittstellenbezeichnung	Beschreibung
`CompositeMetaData`	Container, mit dessen Hilfe Metadatenfragmente zu einem gemeinsamen Metadatenobjekt zusammengefügt werden können; Metadatenfragmenten muss ein im Metadatenobjekt eindeutiger Schlüssel zugeordnet werden; Zugriff auf die Metadatenfragmente erfolgt über diesen Schlüssel
`FunctionStore`	Container, mit dessen Hilfe Funktionen verwaltet und direkt ausgewertet werden können
`MetaDataException`	Ausnahme, welche auf einen Fehler in der Verarbeitung von Metadaten hinweist
`MetaDataProviders`	Sammlung statischer Methoden für den Zugriff auf die Metadaten eines Objekts vom Typ `MetaDataProvider` aus dem Paket `xxl.util`
`ObjectMetaData`	Klasse, welche ein beliebiges (Metadaten-) Objekt adaptiert und den Zugriff darauf regelt; dient als Basis für alle Adapter für Metadaten
`OrderPreservingFunctionStore`	Container, mit dessen Hilfe Funktionen verwaltet und direkt ausgewertet werden können; zusätzlich wird für Iterationen über alle verwalteten Funktionen die Reihenfolge, in der die Funktionen in den Container eingefügt wurden, erhalten

Tabelle 6.12: Kurzbeschreibung der neuen Klassen und Schnittstellen des Pakets `xxl.util`

Kapitel 7

Diskussion und Ausblick

Im Rahmen dieser Arbeit wurden Techniken der Anfrageoptimierung auf ihre Anwendbarkeit im Kontext von Datenströmen hin untersucht. Zu diesem Zweck wurde zunächst die herkömmliche Optimierung von Anfragen in Datenbanksystemen eingehend betrachtet und mit den Anforderungen, welche Datenströme an eine Anfrageoptimierung stellen, verglichen. Weiterhin wurden mit der anfragenübergreifenden und der dynamischen Optimierung Erweiterungen der Anfrageoptimierung untersucht, welche sich die Lösung spezieller Probleme, die auch im Kontext von Datenströmen eine zentrale Bedeutung besitzen, zum Ziel gemacht haben. Aus diesen Untersuchungen ist eine Optimierungsstrategie hervorgegangen, deren Schwerpunkt in der Erkennung und Wiederverwendung gemeinsamer Teilanfragen liegt. Die Entscheidung zugunsten dieses Schwerpunkts liegt in den Charakteristika aktiver Datenquellen und den Eigenschaften von Anfragen an diese begründet. Da jede Teilanfrage einer solchen Anfrage wiederum eine Datenquelle definiert und weiterhin jede Datenquelle ihre Daten jedermann zugänglich macht, drängt sich diese Entscheidung geradezu auf.

Ein wichtiger Aspekt der Arbeit lag außerdem in der Integration einer solchen Anfrageoptimierung in die Bibliothek XXL. Da die Bibliothek neben einer Algebra objekt-relationaler Operatoren für aktive Datenquellen bereits über eine flexible Anfrageoptimierung für Anfragen an relationale Datenbanken verfügt, war die Frage nach einem Brückenschlag zwischen diesen beiden Modellen von besonderem Interesse. Als Antwort auf diese Frage entstand eine Algebra logischer Operatoren, mit deren Hilfe sowohl Anfragen an passive als auch Anfragen an aktive Datenquellen formuliert werden können. Dies wurde jedoch erst durch ein dynamisches Metadatenkonzept ermöglicht, mit dessen Hilfe zu Objekten beliebige Informationen zur Verfügung gestellt werden können. Dieses Metadatenkonzept konnte mit seiner dauerhaften Integration in der Datenstromalgebra bereits erste Erfolge für sich verbuchen. Mit Hilfe dieser Konzepte kann nun die bestehende Anfrageoptimierung derart erweitert werden, dass auch Anfragen auf Datenströmen mit ihrer Hilfe optimiert werden können. Dies bildete den Ausgangspunkt für die Definition von speziellen Transformationsregeln und Optimierungsverfahren für Datenströme, so dass letzten Endes mit der bestehenden Anfrageoptimierung, welche ursprünglich allein für die Optimierung von Anfragen an passive Datenquellen bestimmt war, eine vollwertige Anfrageoptimierung auf Daten-

strömen vorliegt.

Im Anschluss an diese Arbeit verbleiben noch eine Reihe weiterer Aufgaben, die einer näheren Betrachtung würdig erscheinen. So muss zunächst die allgemeine logische Algebra in die bestehende Anfrageoptimierung integriert werden, so dass diese als Grundlage für die Optimierung von Datenströmen verwendet werden kann. Ausgehend von dieser zugegebenermaßen arbeitsintensiven aber prinzipiell unproblematischen Aufgabe eröffnet sich ein weites Feld an interessanten Themen, die es noch zu betrachten gilt.

Da mit der Durchführung einer dynamischen Optimierung ein gewaltiger Aufwand verbunden ist, erscheinen nähere Betrachtungen in dieser Gebiet durchaus lohnenswert. Hierbei wird ein großer Teil dieses Aufwands allein durch die Forderung nach Zeitfenstern identischer Größe verursacht. Um nach der Optimierung des logischen Anfragegraphen die notwendigen Transformationen auf den zugehörigen physischen Anfragen nachvollziehen zu können, müssen zunächst alle beteiligten Zeitfenster auf eine identische Größe gebracht werden. Dieser Vorgang ist entweder mit einem großen Zeitaufwand oder einem Verlust von potentiellen Ergebnissen der Operation verbunden. Um diesen Aufwand abzuwenden, ist die Untersuchung von Transformationsregeln, welche nicht auf Operatoren mit Zeitfenstern identischer Größe beschränkt sind, von großem Interesse.

Weiterhin besteht ein begründetes Interesse an der Betrachtung alternativer Kostenmodelle. Dabei scheinen prinzipiell Modelle auf Basis einer Abschätzung der Datenraten von Operatoren für die Verwendung im Kontext von Datenströmen besser geeignet zu sein als herkömmliche Modelle auf Basis einer Abschätzung der Zwischenergebnisse eines Operators. Dabei sind vor allem die Auswirkungen der Verwendung solcher Kostenmodelle auf den Grad der Wiederverwendung gemeinsamer Teilanfragen ein zentraler Interessenschwerpunkt.

Aufgrund des hohen Aufwands, der mit der Durchführung einer dynamischen Optimierung der aktiven Anfragen eines Systems verbunden ist, erscheint auch die Definition geeigneter Kriterien für diese Aufgabe durchaus als sinnvoll. Hierbei sind insbesondere die engen, wechselseitigen Beziehungen zwischen der Ressourcenverwaltung, der Ablaufsteuerung und der Anfrageoptimierung zu berücksichtigen. Weiterhin gilt es zu überprüfen, in wie weit eine dynamische Optimierungen mittels einer sorgfältigen Neuaufteilung des verfügbaren Speichers und einer entsprechenden Bevorzugung der betreffenden Anfragen in der Ablaufsteuerung aufgefangen werden kann.

Auch verspricht die Erweiterung der Anfrageoptimierung um weiterführende Konzepte wie beispielsweise die in [BKK+01] beziehungsweise [Bra02] erläuterten *quality-of-service* Aspekte von Anfragen die Erschließung weiterer interessante Themengebiete. Für die Anfrageoptimierung stellen solche Qualitätsanforderungen, wie beispielsweise die Vorgabe einer Zeitspanne, in der die Ergebnisse einer Anfrage vorliegen müssen, eine zusätzliche Herausforderung dar, bei der Bevorzugung einzelner Anfragen zur Erfüllung deren Qualitätsanforderungen und der daraus resultierenden Benachteiligung der restlichen im System aktiven Anfragen abgewogen werden muss.

Zusammenfassend bleibt somit zu erwähnen, dass die vorliegende Arbeit sich mit einem sehr interessanten und hoch aktuellen Forschungsgebiet auseinandergesetzt hat.

Dabei sind aus ihr sind einige sehr interessante Erkenntnisse in Bezug auf die Optimierung von Datenströmen hervorgegangen. Jedoch hat sie auch mindestens ebenso viele neue und spannende Fragen aufgeworfen, welche ein lohnendes Ziel weiterer Forschung darstellen könnten.

List of Algorithms

Tabellenverzeichnis

Literaturverzeichnis

[ABC+76] Astrahan, Morton M., Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade und Vera Watson: *System R: Relational Approach to Database Management.* ACM Transactions on Database Systems, 1(2):97–137, 1976.

[BBD+01] Bercken, Jochen van den, Björn Blohsfeld, Jens-Peter Dittrich, Jürgen Krämer, Tobias Schäfer, Martin Schneider und Bernhard Seeger: *XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries.* In: *Proceedings of the International Conference on Very Large Data Bases*, Seiten 39–48. Morgan Kaufmann, 2001.

[BBD+02] Babcock, Brian, Shivnath Babu, Mayur Datar, Rajeev Motwani und Jennifer Widom: *Models and Issues in Data Stream Systems.* In: *Proceedings of the Symposium on Principles of Database Systems*, Seiten 1–16. ACM Press, 2002.

[BDS00] Bercken, Jochen van den, Jens-Peter Dittrich und Bernhard Seeger: *javax.XXL: A Prototype for a Library of Query Processing Algorithms.* In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Band 29, Seite 588. ACM Press, 2000.

[Ber02] Beringer, Jürgen: *Anfrageoptimierung in XXL.* Diplomarbeit, Philipps-Universität Marburg, September 2002.

[BKK+01] Braumandl, Reinhard, Markus Keidl, Alfons Kemper, Donald Kossmann, Alexander Kreutz, Stefan Seltzsam und Konrad Stocker: *ObjectGlobe: Ubiquitous query processing on the Internet.* The International Journal on Very Large Data Bases, 10(1):48–71, 2001.

[Bra02] Braunmandl, Reinhard: *Quality of Service and Optimization in Data Integration Systems.* Doktorarbeit, Universität Passau, Februar 2002.

[CÇC+02] Carney, Don, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker,

Nesime Tatbul und Stan Zdonik: *Monitoring Streams – A New Class of Data Management Applications.* In: *Proceedings of the International Conference on Very Large Data Bases*, Seiten 215–226. Morgan Kaufmann, 2002.

[CG94] Cole, Richard L. und Goetz Graefe: *Optimization of Dynamic Query Evaluation Plans.* In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Band 23, Seiten 150–160. ACM Press, 1994.

[GD87] Graefe, Goetz und David J. DeWitt: *The EXODUS Optimizer Generator.* In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Band 16, Seiten 160–172. ACM Press, 1987.

[GHJV95] Gamma, Erich, Richard Helm, Ralph Johnson und John M. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Professional, 1995.

[GM93] Graefe, Goetz und William J. McKenna: *The Volcano Optimizer Generator: Extensibility and Efficient Search.* In: *Proceedings of the International Conference on Data Engineering*, Seiten 209–218. IEEE Computer Society, 1993.

[Gra94] Graefe, Goetz: *Volcano - An Extensible and Parallel Query Evaluation System.* IEEE Transactions on Knowledge and Data Engineering, 6(1):120–135, 1994.

[IK84] Ibaraki, Toshihide und Tiko Kameda: *On the Optimal Nesting Order for Computing N-Relational Joins.* ACM Transactions on Database Systems, 9(3):482–502, 1984.

[KE99] Kemper, Alfons und André Eickler: *Datenbanksysteme : eine Einführung.* R. Oldenbourg Verlag, 3. Auflage, 1999.

[KNV02] Kang, Jaewoo, Jeffrey F. Naughton und Stratis D. Viglas: *Evaluating Window Joins over Unbounded Streams.* University of Wisconsin-Madison, 2002.

[Krä03] Krämer, Jürgen: *Entwurf und Implementierung objekt-relationaler Operatoren für aktive Datenquellen.* Diplomarbeit, Philipps-Universität Marburg, April 2003.

[Mic03] Michel, Sebastian: *Verteilte Anfrageverarbeitung mit Web Services.* Diplomarbeit, Philipps-Universität Marburg, Dezember 2003. Noch nicht erschienen.

[ONK+96] Ozcan, Fatma, Sena Nural, Pinar Koksal, Cem Evrendilek und Asuman Dogac: *Dynamic Query Optimization on a Distributed Object Management Platform.* In: *Proceedings of the International Conference*

on Information and Knowledge Management, Seiten 117–124. ACM Press, 1996.

[Phi03] Philipps-Universität Marburg: Arbeitsgruppe Datenbanksysteme: *XXL – the eXtensible and fleXible Library*. Online im Internet, URL: http://www.mathematik.uni-marburg.de/DBS/xxl/, 2003.

[Roy98] Roy, Prasan: *Optimization of DAG-Structured Query Evaluation Plans*. Diplomarbeit, Indian Institute of Technology Bombay, Januar 1998.

[Roy00] Roy, Prasan: *Multi-Query Optimization and Applications*. Doktorarbeit, Indian Institute of Technology Bombay, 2000.

[RSSB00] Roy, Prasan, S. Seshadri, S. Sudarshan und Siddhesh Bhobe: *Efficient and Extensible Algorithms for Multi Query Optimization*. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Band 29, Seiten 249–260. ACM Press, 2000.

[SAC+79] Selinger, Patricia G., Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie und Thomas G. Price: *Access Path Selection in a Relational Database Management System*. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Seiten 23–34. ACM Press, 1979.

[Sel88] Sellis, Timos K.: *Multiple-Query Optimization*. ACM Transactions on Database Systems, 13(1):23–52, 1988.

[SG90] Sellis, Timos K. und Subrata Ghosh: *On the Multiple-Query Optimization Problem*. IEEE Transactions on Knowledge and Data Engineering, 2(2):262–266, 1990.

[Sun03] Sun Microsystems, Inc.: *Java^{TM}2 Platform, Standard Edition, Version 1.4.2 API Specification*. Online im Internet, URL: http://java.sun.com/j2se/1.4.2/docs/api/, 2003.

[VN02] Viglas, Stratis D. und Jeffrey F. Naughton: *Rate-based query optimization for streaming information sources*. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Seiten 37–48. ACM Press, 2002.

Danksagung

Abschließend möchte ich meine Diplomarbeit dazu nutzen, allen Personen zu danken, welche direkt oder indirekt an ihrer Entstehung beteiligt waren. Hierbei geht mein Dank in erster Linie an meine Familie, welche mir während meines gesamten Studiums zur Seite stand und mich stets unterstützt hat. Ausdrücklich bedanken möchte ich mich auch bei meiner Lebensgefährtin Monique Riemenschneider, die mir ein Quell der Ruhe, der Inspiration und der Kreativität zugleich war und mir stets aufmunternde Worte entgegenbrachte, wenn ich ihrer bedurfte.

Selbstverständlich danke ich auch meinem Betreuer Prof. Dr. Bernhard Seeger, welcher jederzeit Fragen offen gegenüber stand und auch in kritischen Zeiten meine Schritte sicher in Richtung einer erfolgreichen Beendigung dieser Diplomarbeit lenkte. Weiterhin gebührt Jürgen Beringer, Michael Cammert, Christoph Heinz und Sebastian Michel, die kurzfristig Zeit aufbrachten, um gemeinsam diese Diplomarbeit Korrektur zu lesen, besonderer Dank. Schließlich möchte ich all den ungenannten Mitgliedern der Arbeitsgruppe Datenbanksysteme für ihre Unterstützung, welche sie mir jederzeit entgegenbrachten, danken.

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Marburg, im Oktober 2003

(*Tobias Schäfer*)