TECHNISCHE UNIVERSITÄT ILMENAU

Fakultät für Informatik und Automatisierung
Institut für Praktische Informatik und Medieninformatik
Fachgebiet Datenbanken und Informationssysteme

Dynamische Anfrageoperatoren für CAN-basierte P2P-Systeme

Diplomarbeit zur Erlangung des akademischen Grades Diplominformatiker vorgelegt der Fakultät für Informatik und Automatisierung der Technischen Universität Ilmenau von Christian von der Weth

vorgelegt von: *Christian von der Weth*
Hertzstr. 15
76187 Karlsruhe
Betreuer: *Prof. Dr.-Ing. habil. Kai-Uwe Sattler*
Tag/Ort der Einreichung: *12.11.2004, Ilmenau*
Inventarisierungsnummer: *2004-12-01/126/IN99/2254*

Abstract

One of the main tasks of database or information systems is to provide an effective access to the stored data. Responsible for this task is the query proccessing. In large federated and shared-nothing environments, like Peer-to-Peer, the query processing has to deal with widely fluctuating characteristics of the resources. A basic approach is the development of methods for an adaptive query processing. In contrast to static query optimization and execution techniques, adaptive query processing tries to adapt optimization and execution to the characteristics of the system throughout the duration of a query.

The existing implementations of an adaptive query processing are not developed specifically and therefore optimized for Peer-to-Peer systems. In this work an implementation of an adaptive query processing method is introduced, which is designed for CAN-based Peer-to-Peer systems. A CAN is a special overlay-network for the distribution and indexing of the data in the network. In particular the implementation copes with the most important advantages of Peer-to-Peer systems: scalability and robustness. The adaptive query processing mechanism of this paper is called P2P-Eddy and enhances the original eddy mechanisms for Peer-to-Peer environments.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

1.1 Zielstellung

Datenbankmanagementsysteme erheben den Anspruch, große Datenbestände effizient und effektiv zu erzeugen, zu manipulieren und zu verwalten. Aus Sicht eines Nutzers oder einer Anwendung steht in erster Linie ein schneller Zugriff auf die Daten im Vordergrund [Vos00]. Zuständig hierfür ist die sogenannte Anfrageverarbeitung. Neben anderen Teilaufgaben umfasst sie auch die Optimierung und die eigentliche Ausführung einer Anfrage. Die Umsetzung und Performanz der Anfrageverarbeitung hängt stark von der gesamten Systemumgebung ab. Nur wenn sie optimal an das System und die Umgebung angepasst ist, kann eine Anfrageverarbeitung optimale Ergebnisse liefern.

Ein wichtiger Parameter dafür ist die Art und Weise, wie die Informationen der Datenbank gespeichert werden. Man unterscheidet im Wesentlichen zwischen verteilter und nichtverteilter Datenhaltung. Unterteilt man die Daten in die eigentlichen Nutzdaten und die Metadaten, kann die Datenhaltung in drei große Klassen eingeteilt werden. In diesem Kontext sollen unter Metadaten vor allem die Indexstrukturen verstanden werden.

1. *Nichtverteilte Nutzdaten. Nichtverteilte Metadaten.*
   Sämtliche Daten der Datenbank werden von einem System verwaltet. Quasi alle Informationen, die eine Datenbank betreffen stehen hier jederzeit zur Verfügung.

2. *Verteilte Nutzdaten. Nichtverteilte Metadaten.*
   Die eigentlichen Nutzdaten der Datenbank liegen verteilt auf verschiedenen Rechnern. Alle Zugriffe auf die Datenbank werden von einer zentralen Stelle aus organisiert, überwacht und ausgeführt. Es existiert also eine Instanz, die eine globale Sicht auf die Datenbank besitzt.

3. *Verteilte Nutzdaten. Verteilte Metadaten.*
   Sowohl Nutz- als auch Metadaten liegen auf verschiedenen vernetzten Rechnern. Keine Instanz im Netz besitzt ein globales Wissen über die Datenbank. Dadurch stehen in der Regel auch deutlich weniger Metadaten zur Verfügung.

Gerade für die Anfrageverarbeitung spielen die Metadaten ein wichtige Rolle, da die Optimierung einer Anfrage zum großen Teil auf Kenngrößen der Datenbank (Größe und

Anzahl der Relationen, Schemainformationen, ...) und andere verwaltete Statistiken (Verteilungen der Attributwerte, Selektivität von Operatoren, ...) zurückgreift. Eine Anfrageverarbeitung, die für ein System aus einer der oben vorgestellten Klasse optimiert wurde, eignet sich also nicht automatisch für Systeme der anderen Klassen. Prinzipiell ist ein Einsatz zwar möglich, doch werden in der Regel damit nur suboptimale Ergebnisse erzielt.

Besonders Datenbanksysteme der ersten, aber auch der zweiten Klasse, zeichnen sich dadurch aus, dass eine Vielzahl von Metadaten global erfasst werden können. Zudem unterliegen die Metadaten in diesen Systemen typischerweise keinen starken Schwankungen über der Zeit. Deshalb hat sich hier eine strikte Trennung zwischen der Optimierung und Ausführung eine Anfrage durchgesetzt. Man spricht auch von einer statischen Anfrageverarbeitung. Eine solche Trennung ist zwar weniger flexibel, bedeutet aber einen wesentlich geringeren Aufwand. Und da sich die Parameter für die Optimierung nur wenig ändern, ist dieses Vorgehen für solche Systeme durchaus praktikabel.

In komplett verteilten Umgebungen, wie z.B. Peer-to-Peer Netzen, gelten ganz andere Voraussetzungen. Hier stehen typischerweise deutlich weniger Metadaten zur Verfügung, auf die nicht global zugegriffen werden kann. Besonderes Augenmerk muss dabei auf die Indexierung der Daten gerichtet werden. Ein geeigneter Mechanismus sind Overlay-Netze wie das Content Adressable Network. Diese logische Struktur wird über die physikalische P2P-Umgebung gesetzt und übernimmt die Verwaltung bzw. Indexierung der Daten.

Umgebungen wie Peer-to-Peer Netze besitzen typischerweise eine hohe Dynamik, d.h. dass sich die Parameter für die Anfrageverarbeitung bzw. -optimierung über die Zeit schnell stark ändern können. Je weitvereilter das Netz dabei ist, desto länger ist auch die Dauer für die Ausführung einer Anfrage. Bei einer Trennung zwischen Optimierung und Ausführung, kann das Resultat der Optimierung noch zur Ausführungszeit hinfällig werden und eventuell nur noch suboptimale Ergebnisse liefern.

Ziel der Arbeit ist die Entwicklung einer Anfrageverarbeitung für eine CAN-basierte Peer-to-Peer Umgebung, welche auf eine Trennung zwischen Optimierung und Ausführung einer Anfrage verzichtet. Zur Ausführungszeit soll also entschieden werden, welche Schritte als nächstes abgearbeitet werden. Als Basis für diese Entscheidungsfindung werden dabei Kenngrößen dienen, die zur Laufzeit ermittelt bzw. erlernt werden, um den jeweils aktuellen Zustand der Systemumgebung einfließen zu lassen.

Ein derart flexibler und dynamischer Mechanismus führt zwangsläufig zu einem deutlichen Mehraufwand. In Abhängigkeit der Systemumgebung kann der zusätzliche Overhead der dynamischen Anfragverarbeitung, die Effizienz negativ beeinflussen. Dennoch kann eine Trennung von Optimierung und Ausführung im Mittel bessere Ergebnisse erzielen.

1.2 Überblick

Kapitel 2 gibt zunächst einen kurzen Überblick über die Anfrageverarbeitung, wie sie in den meisten kommerziellen, nichtverteilten Datanbanksystemen zum Einsatz kommt. Die Phase der Optimierung wird dabei besonders hervorgehoben. Über die Eigenschaften die-

ser Anfrageverarbeitung lassen sich dann auch deren Nachteile für verteilte Umgebungen ableiten.

Im Anschluss daran werden die grundlegenden Ansätze und Konzepte adaptiver Verfahren für die Anfrageverarbeitung vorgestellt. Es werden die wichtigsten Charakteristiken genannt, mit derer sich die verschiedenen adaptiven Verfahren klassifizieren lassen. Zur Veranschaulichung soll auf verschiedene konkrete Umsetzungen näher eingegangen werden.

Kapitel 3 befasst sich mit der zugrundeliegenden Systemumgebung. Dazu zählen vor allem die allgemeinen Eigenschaften und Besonderheiten des Peer-to-Peer Netzmodells, sowie dessen Vor- und Nachteile.

Danach wird das Konzept des *Content-Adressable Networks* (CAN) vorgestellt. Diese logische Netzstruktur eignet sich Dank seiner Vorteile gerade für den Einsatz auf einem Peer-to-Peer System.

Da auch in dieser Arbeit mit relationalen Daten in Form von Tupeln gearbeitet wird, soll im letzten Unterpunkt noch die Organisation dieser Daten innerhalb eines CANs demonstriert werden. Diese Ergebnisse haben einen entscheidenden Einfluss auf die spätere Implementierung.

Der Entwurf für die in dieser Arbeit vorgestellten adaptiven Anfrageverarbeitung, im Folgenden P2P-Eddy genannt, ist Thema von Kapitel 4. Es werden prinzipielle Überlegungen angestellt, wie die gegebenen Anforderungen, für die anschließende Implementierung, erfüllt werden können. Dieses Kapitel ist in zwei Hauptabschnitte unterteilt.

Zunächst wird das Konzept soweit ausgearbeitet, dass eine adaptive Anfrageverarbeitung möglichst flexibel und dynamisch umgesetzt werden kann. Es wird sozusagen die technische Voraussetzung entwickelt.

Um die Dynamik und Flexibiltät effizient zu nutzen, werden Srategien benötigt, die anhand unterschiedlicher Kriterien, einen Einfluss auf die Anfrageverarbeitung besitzen. Im zweiten Abschnitt werden dazu verschiedene Verfahren und Parameter vorgestellt.

In den Kapiteln 5 und 6 wird die Implementierung des P2P-Eddies behandelt. Dabei wird in erster Linie auf die Besonderheiten eingegangen, die aus der dynamischen Anfrageverarbeitung heraus entstehen. Daneben werden einige Kernalgorithmen vorgestellt.

Schwerpunkt von Kapitel 5 ist die Implementierung der dynamischen Anfrageoperatoren. Diese bilden die Grundlage für eine variable Operatorreihenfolge für verschiedene Tupel und damit für eine dynamischen Anfrageverarbeitung.

Strategien für die effiziente Ausnutzung dynamischer Anfrageoperatoren sind Thema von Kapitel 6. Es werden vor allem zwei Ziele verfolgt. Zum einen sollen möglichst kostengünstige Operatorreihenfolgen für die Tupel gefunden werden. Zum anderen soll die erzeugte Last einer Anfrage möglichst fair im Netz verteilt werden.

Ein wichtiger Punkt bei der Entwicklung neuer Verfahren ist die Evaluierung, denn nicht immer stimmen die praktischen Ergebnisse mit den theoretischen Annahmen überein. Auch für diese adaptive Anfrageverarbeitung wurden einige Test durchgeführt,

um zu überprüfen, ob die Umsetzung den Erwartungen entspricht.

Die Evaluierung ist Thema von Kapitel 7. Anhand ausgewählter Tests soll die Einsatztauglichkeit der neuen Anfrageverarbeitung demonstriert werden. Überprüft wird das Verhalten in verschiedenen Netzgößen sowie das Verhalten bei der Manipulation einiger wichtiger Kenngrößen, die Einfluss auf die Arbeitsweise der Anfrageverarbeitung nehmen.

Kapitel 2

Grundlagen der Anfrageverarbeitung

2.1 Traditionelle Anfrageverarbeitung und -optimierung

2.1.1 Anfrageverarbeitung

Die Anfrageverarbeitung ist Kernstück eines jeden Datenbankmanagementsystems, da sie die eigentliche Zugriffsmöglichkeit auf die Daten der Datenbank darstellt. Sie ist somit als Schnittstelle zwischen der Anfragesprache und dem Dateisystem anzusehen [Vos00]. Aus diesem Grund kommt der Anfrageverarbeitung eine große Bedeutung zu.

Der gesamte Vorgang der Verarbeitung einer Anfrage lässt sich grob in folgende vier Phasen unterteilen [Vos00]:

1. Vorverarbeitung
2. Anfrageoptimierung
3. Code-Erzeugung
4. Ausführung

Aufgabe der Vorverarbeitung ist die Umwandlung einer Anfrage, von der Syntax einer Anfragesprache in eine interne Darstellung für die Weiterverarbeitung. Dazu gehört zunächst das Scannen des Anfrage-Strings. Dabei werden z.B. die Schlüsselworte, Attribut- und Relationennamen identifiziert. Der Parser prüft daraufhin, ob es sich gemäß den Grammatikregeln der Anfragesprache um eine korrekte Anfrage handelt. Letzter Teilschritt ist die Validierung. Die Validierung ist dann erfolgreich, wenn alle Attribute und Relationennamen positiv auf ihre Gültigkeit geprüft wurden.

Die zweite Phase ist die Anfrageoptimierung. Was eine Optimierung überhaupt erst notwendig macht, ist die Deskriptivität als wichtiges Kriterium für Anfragesprachen [HS00]. Deskriptive Sprachen sind zwar schwieriger zu implementieren, garantieren aber die gewünschte Unabhängigkeit vom zugrundeliegenden Datenmodell. Daneben werden solche Sprachen von den Anwendern bevorzugt [Ull88]. Mit deskriptiven Anfragesprachen wird quasi nur das Ergebnis der Anfrage formuliert, aber nicht auf welchem Wege dies geschehen soll. Da für die Ausführung letztlich eine prozedurale Darstellung (ausführbarer Code) der Anfrage benötigt wird, muss eine Umsetzung aus der deskriptiven

Formulierung erfolgen. In der Regel gibt es für eine Anfrage viele verschiedene Möglichkeiten für eine prozedurale Darstellung. Genau an dieser Stelle greift die Anfrageoptimierung ein. Sie sorgt dafür, dass eine möglichst effiziente Ausführungsstrategie gefunden wird.

Aufgabe der dritten Phase ist die Erzeugung von ausführbarem Code für den gewählten Anfrageplan. Der Code selbst kann entweder direkt ausgeführt (interpretierter Modus) oder gespeichert und erst bei Bedarf ausgeführt werden (kompilierter Modus) [KE99].

Die eigentliche Ausführung übernimmt in der letzten Phase der Laufzeitdatebankprozessor, im kompilierten oder interpretierten Modus. Eine Technik, die bei der Ausführung einer Anfrage eingesetzt wird, ist das so genannte Pipelining, auch als strombasierte Verarbeitung bezeichnet [EN02]. Ohne Pipelining werden die einzelnen Operatoren einer Anfrage getrennt hintereinander ausgeführt. Dies führt dazu, dass die Zwischenergebnisse jedes Operators temporär gespeichert werden müssen. Ein meist unnötiger Aufwand wenn man bedenkt, dass die Ausgaben eines Operators in der Regel die Eingaben für den folgenden Operator sind. Das Pipelining versucht diesem Nachteil zu umgehen, indem aktuell erzeugte Zwischenergebnisse eines Operators sofort zum nächsten weitergereicht werden. Diese Art der Parallelisierung bedeutet besonders auch in verteilten Datenbanksystemen einen deutlichen Performanzgewinn.

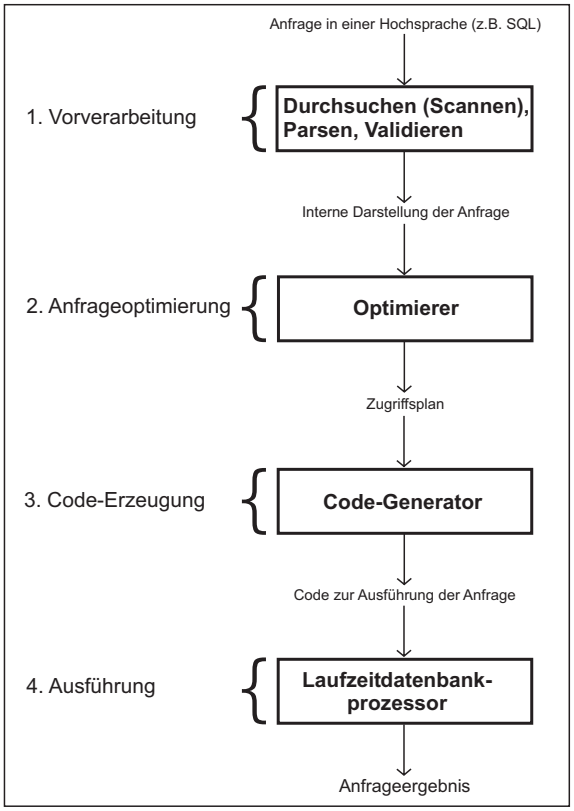

Abbildung 2.1: Typische Schritte der traditionellen Anfrageverarbeitung

2.1.2 Anfrageoptimierung

Wie bereits erwähnt, begründet sich die Notwendigkeit einer Anfrageoptimierung aus dem Ziel, eine deskriptiv formulierte Anfrage in einen effizienten Ausführungsplan umzuwandeln. Ein solcher Plan wird dann als effizient betrachtet, wenn er bei seiner Ausführung möglichst wenig geringe verursacht [Vos00]. Die Gesamtkosten setzen sich wie folgt zusammen [EN02]:

- *Zugriffskosten auf Sekundärpeicher*
  Kosten für das Durchsuchen, Lesen und Schreiben von Datenblöcken, die auf dem Sekundärspeicher gespeichert sind.
- *Speicherkosten*
  Kosten für die Speicherung von temporären Dateien.
- *Abarbeitungskosten*
  Kosten für die Durchführung von Speicheroperationen auf die Datenpuffer während der Anfragenausführung
- *Hauptspeicherkosten*
  Anzahl der während der Anfragenausführung benötigten Puffer im Hauptspeicher
- *Kommunikationskosten*
  Kosten der Übertragung der Anfrage und ihrer Resultate vom Datenbankrechner an den Anfragensteller. In verteilten Datenbanken kommt die Anzahl der benötigten verschickten Nachrichten dazu.

Wie bei den meisten Formen der Optimierung, handelt es sich auch bei der Anfrageoptimierung um ein kombinatorisches Problem mit hoher Komplexität. Aus diesem Grund lässt sich die beste Strategie nicht in vertretbarem Zeitaufwand finden. In der Praxis wird deshalb weniger versucht die beste, sondern eine möglichst optimale Ausführungsstrategie zu ermitteln bzw. schlechte Strategien zu vermeiden.

Als wichtiger Bestandteil der Anfrageverarbeitung, besteht die Optimierung ihrerseits typischerweise aus drei Phasen. Dazu zählen zum einen die beiden prinzipiellen Techniken der logischen und physischen Optimierung [KE99] und zum anderen die kostenbasierte Auswahl des letztlich auszuführenden Anfrageplans. Die Aufgabe der Optimierungstechniken ist die Erzeugung äquivalenter Alternativpläne. In der Phase der kostenbasierten Auswahl wird aus allen Anfrageplänen derjenige ausgewählt, der auf Basis von Kostenabschätzungen am besten ist.

Die logische Optimierung, oder auch High-Level-Optimierung, befindet sich auf der Ebene der logischen Algebra, also der Relationenalgebra im relationalen Datenmodell. Da die Umformung auf syntaktischer Ebene geschieht, wird diese Technik auch als Rewriting bezeichnet. Dieses Niveau erlaubt eine Optimierung unabhängig von der eigentlichen Implementierung des Datenbanksystems.

Für das Erzeugen der Alternativanfragepläne werden im Wesentlichen heuristische Regelen angewendet, welche die Reihenfolge der Operatoren im Anfrageplan festlegen. Eine Heuristik ist eine Regel, die aus Erfahrung in den meisten Fällen gute Ergebnisse liefert, aber nicht garantieren kann [EN02]. Die Heuristiken bei der logischen Optimierung

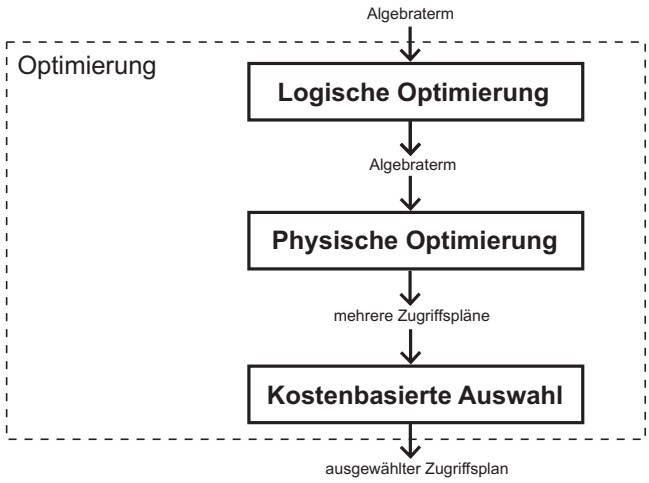

Abbildung 2.2: Phasen der Optimierung

zielen darauf ab, möglichst kleine Zwischenergebnisse zu erzeugen. Mit diesem Ansatz lassen sich die meisten vorgestellten Kosten (Zugriffskosten, Abarbeitungskosten,...) minimieren. Im Detail fallen folgende Heuristiken darunter [KE99]:

- Aufbrechen von Selektionen
- Verschieben der Selektionen soweit wie möglich nach unten im Operatorbaum
- Zusammenfassen von Selektionen und Kreuzprodukten zu Joins
- Bestimmung der Reihenfolge der Joins in der Form, dass möglichst kleine Zwischenergebnisse entstehen
- unter Umständen Einfügen von Projektionen
- Verschieben der Projektionen soweit wie möglich nach unten im Operatorbaum

Um die Äquivalenz der Anfragepläne durch das Vertauschen der Operatorreihenfolge zu gewährleisten, müssen die Umformungsregeln der Relationenalgebra eingehalten werden. Eine vollständige Auflistung dieser Umformungsregeln findet sich z.B. in [EN02].

Bei der physischen Optimierung, oder auch Low-Level-Optimierung, werden für die Optimierung die Interna des Datenbanksystems herangezogen. Diese Technik arbeitet auf der sogenannten physischen Algebra, deren Operatoren die implementierten Gegenstücke der abstrakten Operatoren der logischen Algebra darstellen. So gibt es in der Regel mehrere Möglichkeiten, einen logischen Operator physisch zu implementieren. Vor allem für komplexe Operatoren wie z.B. dem Join existieren unterschiedliche Implementierungen (Nested-Loop-Join, Merge-Join, Hash-Join,...).

Die physische Optimierung erzeugt für einen Anfrageplan, als Ergebnis der logischen Optimierung, mehrere Zugriffspläne, indem für die logischen Operatoren des Anfrageplans, die verschiedenen Implementierungen der Operatoren eingesetzt werden. Logischerweise können nur diejenigen Implementierungen genutzt werden, die auch tatsächlich im verwendeten Datenbankmanagementsystem umgesetzt sind.

Wird bei der Anfrageverarbeitung das vorgestellte Pipelining eingesetzt, entstehen noch weitere Möglichkeiten für eine physische Optimierung. Für einige Implementierungen binärer Operatoren (in erster Linie Joins) ist der Aufwand für die Abarbeitung abhängig von der Reihenfolge der Eingaberelationen (z.B. Nested-Loop-Join). Implementierungen, die im Gegensatz dazu ihre Eingaberelationen gleichberechtigt behandeln, werden auch als symmetrisch bezeichnet (z.B. Merge-Join) [AH00]. Mit dieser Technik kann sich die Abarbeitungsdauer mit der der Reihenfolge der Eingaberelationen ändern, wenn sich die Zeit zwischen den Eintreffen der Tupel beider Relationen unterscheidet. In diesem Fall sollte bei der Optimierung abgeschätzt werden, welche Reihenfolge der Eingaberelationen am sinnvollsten ist.

Da Heuristiken gute Ergebnisse nicht garantieren können, verlässt man sich nicht allein auf diese Techniken. Um die Güte von Anfrageplänen quantifizierbar und damit vergleichbar zu machen, wird ein Kostenmodell benötigt. Ein Kostenmodell stellt Funktionen zur Verfügung, die den Aufwand bzw. die Laufzeit ermitteln. Doch auch diese Kostenfunktionen sind lediglich nur Schätzungen, so dass die gewählte Ausführungsstrategie nicht zwangsweise optimal ist.

Kenngrößen für das Kostenmodell sind vor allem [KE99]:

- Indexinformationen
- Clustering-Informationen
- Kardinalitäten der Datenbank
- Attributverteilungen, u.a.

Diese Informationen werden im Datenbankkatalog gespeichert und gepflegt.

2.1.3 Nachteile der traditionellen Anfrageverarbeitung

Die vorgestellten Konzepte der Anfrageverarbeitung und -optimierung liefern in einem Ein-Prozessor-Datenbanksystem meist nahezu optimale Ausführungsstrategien. In solchen Systemen steht der kostenbasierten Auswahl eine Vielzahl von Parametern zur Verfügung, welche sich dazu in kurzen Zeiträumen kaum ändern. Mit dieser Voraussetzung lassen sich so relativ aussagekräftige Kostenabschätzungen für die verschiedenen Ausführungsstrategien treffen.

In verteilten Datenbanksystemen müssen dagegen andere Annahmen gemacht werden. Diese Systeme verfügen typischerweise über eine weitaus höhere Komplexität als nichtverteilte Systeme. Der Anstieg der Komplexität lässt sich wie folgt einteilen [AH00]:

- *Komplexität bezüglich der Hardware und Auslastung*
  In weitverteilten Umgebungen - vor allem in heterogenen, aber auch in homogenen - ist die Performanz und Auslastung der gesamten Hardware kaum vorhersagbar. Antowrtzeiten können nicht garantiert werden, was Verzögerungen zur Folge haben kann.

- *Komplexität bezüglich der Daten*
  Die Schätzungen für die Selektivität der Operatoren sind in weitverteilten Datenbanksystemen oft unzureichend, da die dazu benötigten statistischen Informationen über die Datenverteilung in der Regel nicht verfügbar sind.

- *Komplexität bezüglich der Nutzerschnittstelle*
  In weitverteilten Umgebungen dauert der Großteil der Anfragen deutlich länger als in Ein-Prozessor-Systemen. Deshalb soll der Nutzer Einfluss auf die Ausführung einer Anfrage besitzen.

Man muss also davon ausgehen, dass sich die Parameter für eine Anfrageoptimierung laufend ändern. Die drei wesentlichen Kenngrößen sind dabei [AH00]:

- Kosten der Operatoren
- Selektivität der Operatoren
- Ankunftsrate der Tupel

Das eigentliche Problem liegt nun in der strikten Trennung von Optimierung und Ausführung einer Anfrage innerhalb der herkömmlichen Anfrageverarbeitung. Bei diesem statischen Verfahren, kann auf Veränderungen zur Laufzeit der Anfrage nicht eingegangen werden. Sinnvoll wäre eine kontinuierliche Optimierung zur Ausführungszeit.

2.2 Formen adaptiver Anfrageverarbeitung

2.2.1 Grundlegende Eigenschaften

Da der Begriff „adaptives System“ nicht immer gleichbedeutend verwendet wird, soll zunächst definiert werden, was in diesem Kontext ein adaptives System ausmacht. Dafür müssen drei Eigenschaften erfüllt sein [HFC+00]:

1. Das System erhält Informationen aus seiner Umgebung.
2. Diese Informationen haben Einfluss auf das Verhalten des Systems.
3. Der gesamte Prozess ist iterativ. Es entsteht eine Rückkopplung zwischen dem Zustand der Umgebung und dem Systemverhalten.

Man beachte, dass auch die statische Anfrageverarbeitung bereits die ersten beiden Punkte erfüllt. Der wesentliche Unterschied ist der Verzicht auf eine Schleife über dem Erfassen des Umgebungszustands und der Anpassung des Systemverhaltens. Doch genau hier steckt das Potential für die Entwicklung effizienter, adaptiver Verfahren für die Anfrageverarbeitung. Durch obige Definition können drei Haupteigenschaften adaptiver Systeme extrahiert werden [HFC+00]:

1. *Häufigkeit der Adaption*
   Gibt an, wie oft das System die Parameter der Umgebung erfasst und wie oft es sich danach anpasst.

2. *Auswirkung der Adaption*
   Spezifiziert, welche Systemeigenschaften bei einer Anpassung geändert werden können.

3. *Dauer der Adaption*
   Damit wird die Länge des Zeitraumes beschrieben, in dem die Rückkopplung zwischen Umgebung und System aufrecht gehalten wird.

2.2.2 Möglichkeiten für eine adaptive Anfrageverarbeitung

Eine adaptive Anfrageverarbeitung unterscheidet sich von einer statischen nur dadurch, dass die Anpassung an die Systemumgebung ein iterativer Prozess während der Anfrage ist. Deshalb können die Möglichkeiten für eine adaptive Anfrageverarbeitung aus der statischen Anfrageverarbeitung abgeleitet werden. Die Entwicklung adaptiver Verfahren für die Anfrageverarbeitung basiert auf zwei Prinzipien. Zum einen wird auf die Veränderungen der Systemumgebung eingegangen, indem der Ausführungsplan einer Anfrage zur Laufzeit modifiziert wird. Dies entspricht den Rewriting-Techniken der statischen Anfrageverarbeitung. Zum anderen werden spezielle physische Operatoren entwickelt, die ihr Verhalten an unvorhersagbaren Bedingungen bzw. Veränderungen der Umgebung anpassen. Das Gegenstück innerhalb der statischen Anfrageverarbeitung ist die physische Optimierung. Auf beide Punkte soll im Folgenden näher eingegangen werden [GPFS02].

Eine Modifikation des Anfrageplans kann unabhängig auf zwei Ebenen erfolgen: der logischen und der physischen Ebene. Auf der logischen Ebene unterscheidet man zwei Varianten:

- *Erzeugung eines Alternativplans*
  Für den restlichen Plan einer Anfrage wird ein komplett neuer Alternativplan erstellt. Dabei können neue Operatoren hinzugefügt, Operatoren geändert und die Form der Baumstrukter des Planes geändert werden.

- *Neuordnung des Anfrageplans*
  Hier darf lediglich die Reihenfolge der Operatoren des restlichen Anfrageplans verändert werden. Operatoren können weder hinzugefügt noch verändert werden.

Die Methoden sind offensichtlich nicht disjunkt, da die Erzeugung eines Alternativplans die Neuordnung der Operatoren mit einschließt. In beiden Fällen muss gewährleistet sein, dass durch eine Modifikation nur äquivalente Anfragpläne entstehen. In relationalen Datenbanksystemen müssen die Modifikationen deswegen gemäß den Regeln der Relationenalgebra erfolgen.

Adaptive Algorithmen für Operatoren können ihr Verhalten zur Laufzeit ändern, in Abhängigkeit veränderter Bedingungen und den zur Verfügung stehenden Informationen über die Systemumgebung. Dazu werden die entsprechenden Parameter kontinuierlich erfasst und ausgewertet. Die Anpassung der Algorithmen geschieht vollkommen autonom und somit unabhängig vom Datenbankmanagementsystem.

Auch wenn die Erfassung und Auswertung der Systemparameter kontinuierlich erfolgt, kann die Anpassung des Algorithmus nicht zu beliebigen Zeitpunkten durchgeführt

werden. Zur Laufzeit entstehen in der Regel immer Zustände, in denen keine Veränderungen vorgenommen werden dürfen, ohne dass das Ergebnis verfälscht wird. Um den Grad der Adaptivität von Algorithmen zu quantifizieren, bedient man sich zweier Begriffe aus der Parallelprogrammierung, mit denen sich die Zustände der Algorithmen beschreiben lassen [AH00]:

- *Synchronisationsschranke*
  Zeitpunkt, an dem ein Algorithmus in einen Zustand übergeht, in dem keine Veränderungen am Verhalten vorgenommen werden dürfen.

- *Symmetriemoment*
  Zeitpunkt, an dem ein Algorithmus angepasst werden kann, ohne dass sein Zustand verändert und dadurch das Ergebnis verfälscht wird.

Je weniger Synchronisationsschranken und je mehr Symmetriemomente ein Algorithmus besitzt, desto höher ist dessen Adaptivität. Bei deren Entwicklung darf allerdings die Komplexität und damit die absoluten Kosten nicht unbeachtet bleiben.

2.2.3 Merkmale für eine Klassifikation adaptiver Verfahren

Eine konkrete Umsetzung einer adaptiven Anfrageverarbeitung kann anhand einiger charakteristischer Merkmale beschrieben werden. Damit ist es möglich, verschiedene Verfahren zu klassifizieren bzw. zu vergleichen.

Einflussgrößen für die Anpassung. Abhängig vom verwendeten Verfahren, werden nur bestimmte Kenngrößen der Umgebung durch die Adaption beeinflusst. Die wichtigsten davon sind:

- *Speicherschwankungen*
  Das System versucht, sich auf die Verfügbarkeit des Hauptspeichers und Speicherknappheiten anzupassen.

- *Präferenzen des Anwenders*
  Der Anwender kann einen indirekten Einfluss auf die Anfrageverarbeitung haben. Dazu gehört z.B., dass der Anwender möglichst schnell Teilergebnisse der Anfrage erwartet. Der Anwender kann auch die Anfrageergebnisse unterschiedlich wichten. Höher gewichtete Daten werden dann vom System schneller verarbeitet.

- *Ankunftsrate der Daten*
  In parallelen und verteilten Systemen wird typischerweise versucht, die Ankunftsrate der Daten anzupassen.

- *aktuelle Statistiken*
  Viele statistische Größen stehen zum Beginn einer Anfrage nicht zur Verfügung oder unterliegen während der Laufzeit starken Schwankungen. Diese Größen müssen während der Ausführung der Anfrage bestimmt bzw. aktualisiert werden.

- *Performanzschwankungen*
  Vor allem in parallelen Systemen kommt es oft zu unvorhersagbaren Leistungseinbrüchen, auf die intelligent reagiert werden muss.

- *Kombination oben genannter Parameter*
  Viele adaptive Verfahren haben Einfluss auf mehrere Parameter der Umgebung.

Ziel der Anpassung. Obwohl als Hauptziel immer eine Steigerung der Effizienz und Effektivität der Anfrageverarbeitung angestrebt wird, können dennoch drei wesentliche Teilziele unterschieden werden.

- *Minimierung der gesamten Antwortzeit*
  Die Zeit vom Absetzen der Anfrage bis zu deren vollständigen Verarbeitung soll möglichst kurz sein.

- *Minimierung der initialen Antwortzeit*
  Die Zeit vom Absetzen der Anfrage bis zum Eintreffen der ersten Teilergebnisse soll möglichst kurz sein.

- *Maximierung des Durchsatzes*
  Das System soll pro Zeiteinheit möglichst viele Daten einer oder mehrerer Anfragen verarbeiten.

Parameter für die Rückkopplung. Für die Anpassung können verschiedene Parameter der Systemumgebung erfasst und ausgewertet werden.

- *Verfügbarkeit des Speichers*
  Überwacht wird vor allem die Auslastung des Hauptspeichers bzw. des Puffers.

- *Nutzereingaben*
  Umfasst die nutzerspezifischen Prioritäten für Teile einer Anfrage und das Update von Teilergebnissen.

- *Verfügbarkeit des Inputs für Operatoren*
  Die Eingabedaten mancher Operatoren können blockiert sein. Im diesem Fall muss laufend geprüft werden, wann weiterer Input zur Verfügung steht.

- *Auslastung*
  Darunter versteht man in erster Linie die Auslastung der Operatoren. Diese spiegelt sich beispielsweise in der Länge der Eingangswarteschlangen wider.

- *Datenrate*
  Rate, mit der neue Tupel erzeugt werden.

- *Statistiken*
  Dazu gehören Kenngrößen wie die Größe der Relationen, Anzahl verschiedener Werte für ein Attribute, die Verfügbarkeit von Indexen und andere. Welche Größen zur Verfügung stehen und ob es sich nur um Schätzungen handelt, hängt von der Systemumgebung ab.

Häufigkeit der Rückkopplung. Beschreibt, wann und wie oft eine Anpassung auf die veränderte Systemumgebung erfolgt.

- *inter-operator*
  Die Anpassung an die Systemumgebung erfolgt jedes Mal zwischen zwei physischen Operatoren.

- *intra-operator*
  Die Anpassung an die Systemumgebung erfolgt zur Laufzeit der physischen Operatoren.

Zielumgebung. Adaptive Verfahren sind in der Regel für eine bestimmte Systemumgebung optimiert, da in verschiedenen Umgebungen nicht die gleichen Voraussetzungen gelten bzw. die gleichen Annahmen gemacht werden können. Zwar ist ein konkretes Verfahren nicht zwangsläufig an seine Zielumgebung gebunden, liefert aber dort die besten Ergebnisse.

- *Ein-Prozessor-System*
  Sämtliche Operationen werden durch einen Prozessor verarbeitet.

- *paralleles System*
  In diesem Kontext wird unter einem parallelen System ein eng verbundenes Mehrprozesser-System verstanden (geringe räumliche Verteilung).

- *verteiltes System*
  Loser Verbund unabhängiger Rechner, die über ein Netzwerk miteinander in Verbindung stehen (große räumliche Verteilung).

Verantwortliche Komponenten für die Anpassung.

- *physische Operatoren*
  Es werden lediglich physische Operatoren eingesetzt, die ihr Verhalten zur Laufzeit anpassen können, unabhängig vom restlichen Datenbankmanagementsystem.

- *lokale Entscheidungsfindung*
  Trifft zu, wenn der Anfrageoptimierer oder eine andere Komponente des Datenbankmanagementsystems, den aktuellen Anfrageplan zur Laufzeit auswertet.

- *globale Entscheidungsfindung*
  In parallelen und verteilten Systemen wird hierfür eine globale Sicht mehrerer beteiligter Knoten benötigt.

Art der Umsetzung. Die Strategien für die Implementierung einer adaptiven Anfrageverarbeitung können in folgende drei Kategorien eingeteilt werden:

- *physische Operatoren*
  Die sämtliche Adaptivität wird durch physische Operatoren realisiert. Weitere Eingriffe in des Datenbankmanagementsystem müssen nicht vorgenommen werden.

- *konkreter Algorithmus*
  Die Adaptivität wird durch Erweiterungen und gezielte Veränderungen der Operatoren des Anfrageplanes erreicht.

- *System*
  Die Adaptivität wird durch mehrere unabhängige Techniken realisiert, die zu einer Einheit zusammengefasst werden.

2.2.4 Bestehende adaptive Anfrageverarbeitungen

Die Fortschritte in der Entwicklung adaptiver Anfrageverarbeitungen lässt sich gut an der Häufigkeit der Adaption zeigen [HFC+00]. In diesem Abschnitt sollen einige Konzepte inkl. konkreter Umsetzungen vorgestellt werden.

Batch-Optimierung und *Late Binding Schemes*
Obwohl beide Verhahren nicht direkt der Definition für adaptive Systeme (siehe Abschnitt 2.2) entsprechen, sollen sie aus Gründen der Vollständigkeit dennoch kurz genannt werden.

Der Anfrageoptimierer des *System R* [SAC+79], dessen Grundideen Bestandteil der meisten relationalen Datanbankmanagementsysteme ist, verwaltet in einem Katalog Statistiken (Kardinalitäten der Tabellen, Verteilung der Attributwerte,...) für die kostenbasierte Auswahl der verschiedenen Anfragepläne. Die Anpassung an das System besteht nun darin, diesen Katalog periodisch vom System aktualisieren zu lassen. Die Aktualisierung wird dabei manuell gestartet und ist unabhängig von der eigentlichen Anfrageverarbeitung. Die Häufigkeit der Adaption ist vergleichsweise selten, die Aktualisierung des Katalogs erfolgt typischerweise ein Mal pro Tag oder Woche.

Late Binding Schemes sind eine spezielle Erweiterung des Anfrageoptimierers des *System R*. Ziel hierbei ist es, in Laufe der Abarbeitung von Anfragen, sich häufig wiederholende Teilanfragen zu erkennen. Diese Teilanfragen werden dann als vollständig kompilierter Maschinencode im Datenbanksystem hinterlegt. Muss eine solche Teilanfrage dann erneut ausgeführt werden, wird sofort auf den kompilierten Code zurückgegriffen.

***Per-Query* Adaptivität**
Die Anpassung an das System erfolgt hier zwischen der Verarbeitung von Anfragen bzw. nach der Verarbeitung von Anfragen. Eine Umsetzung dieser Idee ist die *Adaptive Selectivity Estimation* [CR94], welche wiederum ein Erweiterung des *System R* Optimierers darstellt. Hier werden die Größen aller Teilergebnisse als Metainformation innerhalb der Anfrage gespeichert. Nach jeder Verarbeitung einer Anfrage wird der Datenbankkatalog gemäß den gesammelten Metadaten aktualisiert, welcher somit für die weitere Optimierung verwendet wird.

Competition* und *Sampling
Beim *Competition*-Verfahren [AZ97] wird unter den verschiedenen Zugriffsmöglichkeiten auf eine Tabelle die geeignetste ausgewählt. Es starten zunächst alle Möglichkeiten. Nach kurzr Zeit kann anhand der ersten Ergebnisse die vielversprechendste weiter ausgeführt werden. Alle anderen werden abgebrochen. Die Häufigkeit der Adaption ist bereits intra-operator, auch wenn die Optimierung relativ beschränkt ist. Innerhalb einer Anfrage wird lediglich eine Entscheidung pro Tabelle getroffen.

Ganz ähnlich ist die Arbeitsweise des sogenannten *Sampling* [BDF+97]. Beim *Sampling* werden Teilanfragen stichprobenartig durchgeführt, um den Aufwand für

die Verarbeitung der gesamten Anfrage abzuschätzen. Bezüglich der Häufigkeit der Adaptivität bewegt sich dieses Verfahren in der Gegend der *Per-Query* Adaptivität.

Inter-Operator Optimierung und *Query Scrambling*
Nach der *Per-Query* Adaptivität ist die Inter-Operator Optimierung der nächste logische Schritt. In einem ersten Ansatz für verteilte Systeme werden Teilanfragen an verschiedene Knoten geschickt und die zurückgegebenen Ergebnisse zur Entscheidungsfindung für das weitere Vorgehen verwendet [ONP+96].

Durch das *Query Sampling* werden Anfragepläne zu bestimmten Zeitpunkten innerhalb der Anfrageverarbeitung modifiziert [AFTU96]. Solche Zeitpunkte können z.B. nach der Ausführung blockierender Operatoren (Sortierung,...) sein oder wenn signifikante Performanzeinbrüche auftreten.

Intra-Operator Optimierung (Adaptive Anfrageoperatoren)
Bei der Sortierung und dem Hashing handelt es sich in beiden Fällen um einen Operator, dessen Kosten abhängig vom zur Verfügung stehenden Hauptspeicher sind. Um auf Schwankungen bei der Vergabe von Hauptspeicherressourcen besser reagieren zu können, kommen für das Sortieren und das Hashing spezielle Algoritmen zum Einsatz, die ihr Verhalten den Schwankungen anpassen. Die Adaption erfolgt sowohl beim Verlust von Hauptspeicher als auch bei der Allokation neuer Bereiche [PCL93].

Ein weiterer Operator der immer wieder gesondert betrachtet wird, ist der Join. Als binärer Operator verbindet er Tupel unterschiedlicher Relationen miteinander. Vor allem in verteilten Datenbanksystemen ist die Ausführungszeit des Joins deshalb abhängig von der Ankunftsrate der Tupel aus beiden Relationen. Spezielle Join-Algorithmen wie die *Ripple Join* Familie passen ihr Verhalten automatisch an die Ankunftsraten der Tupel an [HH99].

Adaptive Partitionierung von Anfragen
In verteilten Datenbanksystemen kann eine Intra-Operator Optimierung erreicht werden, indem die Daten aufgeteilt und an verschiedene Knoten im Netz verteilt werden. In traditionellen Systemen wird die Partitionierung statisch durch *Round-Robin* oder Hash-Verfahren verteilt. Bei der adaptiven Partitionierung ist die Aufteilung abhängig vom aktuellen Zustand der Systemumgebung. Eine konkrete Umsetzung ist *River* [ADAT+99].

Eddies: Kontinuierliche Adaption
Eddies erreichen eine kontinuierliche Adaption durch die Verschmelzung von Intra- und Inter-Operator Optimierung. Der Eddy ist somit einer der „aggressivsten“ Umsetzungen einer adaptiven Anfrageverarbeitung [AH00]. Ein Eddy ermöglicht es, die Abarbeitungsreihenfolge der Operatoren einer Anfrage für die einzelnen Tupel kontinuierlich neu zu ordnen. Da seine grundlegenden Konzepte und Ideen die Basis für den in dieser Arbeit vorgestellten Mechanismus sind, soll auf den Eddy-Mechanismus an dieser Stelle näher eingegangen werden.

Zunächst wurde der Eddy als zentrale Komponente implementiert. Gemäß den vorgestellten Charakteristiken für eine adaptive Anfrageverarbeitung, lässt sich der zentralisierte Eddy wie folgt einordnen [GPFS02]:

- Neuordnung der Operatorreihenfolge für die Modifikation des Anfrageplans
- kein Einfluss auf den physischen Zugriffsplan
- keine Neupartitionierung der Anfrage
- mehrere Zielgößen für die Anpassung (Speicherschwankungen, Tupelankunftsraten, ...)
- Hauptziel: Minimierung der Antwortzeit
- Feedback durch statistische Parameter
- Intra-Operator Feedback
- Zielumgebung: Ein-Prozessor-Systeme
- lokale Entscheidungsfindung für die Optimierung
- Umsetzung als konkreter Algorithmus

Abbildung 2.3 zeigt schematisch die Arbeitsweise eine zentralisierten Eddies.

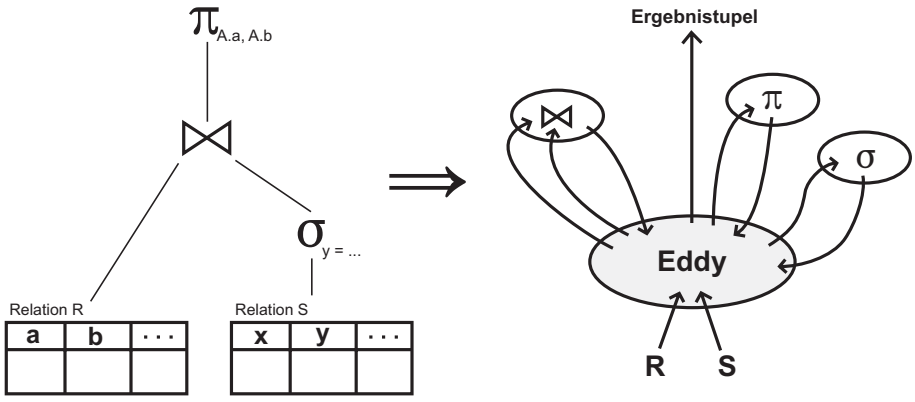

Abbildung 2.3: Zentralisierter Eddy

Der Eddy ist eine Art Verteiler oder Pipeline, der die Tupel der Eingangsrelationen zu den verschiedenen Operatoren der Anfrage leitet und die Ergebnisse derer auch wieder erhält. Damit wird quasi die dynamische, logische Optimierung realisiert. Die Entscheidung, in welcher Reihenfolge die Operatoren abgearbeitet werden, treffen unterschiedliche Routing-Strategien. Der Eddy-Mechanismus als solches kann auch die Konzepte der dynamischen, physischen Optimierung ausnutzen, falls die Operatoren dementsprechend implementiert sind.

Innerhalb des Routing muss sichergestellt werden, dass für alle Tupel nur legale Wege erzeugt werden. Hilfsmittel hierfür sind Statusbits. Um eine doppelte Ausführung einer

Operation zu vermeiden, erhält jedes Tupel pro Operator ein sogenanntes Done-Bit. Bei der Initialisierung sind alle Done-Bits ungesetzt. Passiert ein Tupel einen Operator erfolgreich, wird das entsprechende Done-Bit gesetzt. Ein unkontrolliertes Routing birgt darüber hinaus die Gefahr, dass Operatorreihenfolgen entstehen können, die nicht mehr äquivalent sind, da sie die Umformungsregeln der Relationenalgebra verletzen. Als Gegenmaßnahme wird jedes Tupel um ein Ready-Bit pro Operation erweitert. Das Ready-Bit zeigt an, wann ein Operator ausgeführt werden darf. Diese Ready-Bits müssen nach jeder Ausführung eines Operators anhand der Umformungsregeln aktualisiert werden. Ein Tupel darf also immer nur zu einem Operator mit ungesetzten Done-Bit und gesetzten Ready-Bit geschickt werden.

Für das Routing der Tupel sind verschiedene Strategien denkbar. So kann z.B. die Länge der Eingangswarteschlangen der Operatoren als Maß für deren Kosten angesehen werden. Ein Tupel wird dann zu dem Operator mit der kürzesten Warteschlange geleitet. Eine weitere Strategie nutzt einen Ticket-Mechanismus als Indikator für die Selektivität. Für jeden Operator besitzt der Eddy einen Zähler, der inkrementiert wird, wenn ein Tupel zu dem dazugehörigen Operator geschickt wird. Kommt danach ein Ergebnistupel zurück, wird der Zähler wieder dekrementiert. Operatoren mit einer hohen Selektivität erzeugen also einen hohen Zählerstand. Bei dieser Routing-Strategie werden genau diese Operatoren bevorzugt angelaufen. Eine genaue Umsetzung der kurz vorgestellten Strategien sowie noch weiterer, findet sich in [AH00].

Vorteil des zentralisierten Eddies ist die relativ einfache Implementierung, bei einer dennoch höchst dynamischen Ausführung. Als zentrale Instanz hat der Eddy aber auch folgende Nachteile:

- alle Original-, Zwischenergebnis- und Ergebnistupel muss der Eddy verarbeiten
- der Eddy muss sämtliche Routing-Entscheidungen für jedes Tupel treffen
- hohe Netzlast durch das ständige Hin- und Herschicken der Tupel

Der Eddy kann dadurch selbst schnell zum Flaschenhals werden und somit die Ausführung bremsen [TD03]. Vor allem in einer P2P-Umgebung ist der Einsatz eines zentralisierten Eddies nicht praktikabel. Er widerspricht der Idee von gleichberechtigten Netzknoten und hebt somit die genannten Vorteile von P2P-Netzen auf.

Die nächste logische Weiterentwicklung sind die verteilten Eddies. Bezüglich der Charakteristiken für adaptive Anfrageverarbeitungen, ist die Zielumgebung ein verteiltes Datenbanksystem. Ansonsten unterscheiden sich beide Eddy-Varianten nur wenig. Kernstück ist auch hier die dynamische Auswahl des nächsten Operators, aber ohne Hilfe einer zentralen Komponente. Nachdem ein Operator ein Ergebnistupel erzeugt hat, wird dieses direkt an den nächsten Operator geschickt (siehe Abbildung 2.4). Die Routing-Entscheidungen treffen hier also die Operatoren.

In Abbildung 2.4 stellen die gestrichelten Linien alle möglichen Wege für die Ausführung dar. Die dicke Linie zeigt beispielhaft die Operatorfolge für ein beliebiges Tupel. Auch bei verteilten Eddies werden Ready- und Done-Bits benötigt, um eine doppelte Ausführung von Operatoren und Verletzungen der Umformungsregeln der Relationenalgebra

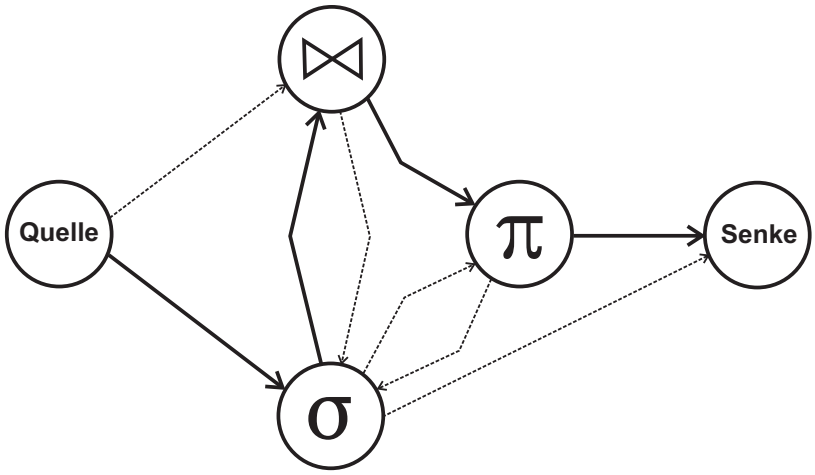

Abbildung 2.4: Verteilter Eddy

zu vermeiden. Im Graph fehlen bereits alle Pfade, die aufgrund der Ready-Bits nicht möglich sind.

Praktikable Routing-Strategien basieren auf ähnlichen Parametern wie schon beim zentralisierten Eddy. Mögliche Parameter sind beispielsweise [TD03]:

- Länge der Eingangswarteschlangen
- erlernte Selektivität durch einen Ticket-Mechanismus
- berechnete Selektivität durch einen Monitor für jeden Operator
- durchschnittliche Verweilzeit eines Tupels in einem Operator als Maß für dessen Kosten

Dazu kommen noch Strategien, die auf den Kombinationen der verschiedenen Parametern basieren. In [TD03] werden konkrete Routing-Strategien sowie deren Vor- und Nachteile im Detail vorgestellt.

Im Normalfall befindet sich ein Operator immer auf dem gleichen Peer. Um Flaschenhalseffekte aufgrund überlasteter Peers zu vermeiden, werden Techniken für eine Lastverteilung benötigt. Beherbergt ein Peer mehr als einen Operator, kann der Peer bei zu hoher Last einen oder mehrere Operatoren an Nachbar-Peers abgeben (*Box-Sliding*). Ist ein Peer mit nur einem Operator überlastet, kann dieser Operator auf zwei Peers verteilt werden (*Box-Splitting*). In beiden Fällen muss das Routing auf die veränderte Situation angepasst werden. Genaueres zu diesem Thema auch in [TD03].

Der zentralisierte, aber vor allem der verteilte Eddy-Mechanismus, setzen bereits einige interessante Konzepte für eine adaptive Anfrageverarbeitung um. Die Eingriffe sowohl in die logische als auch in die physische Optimierung, erlauben dem Eddy ein höchst dynamische Ausführung einer Anfrage.

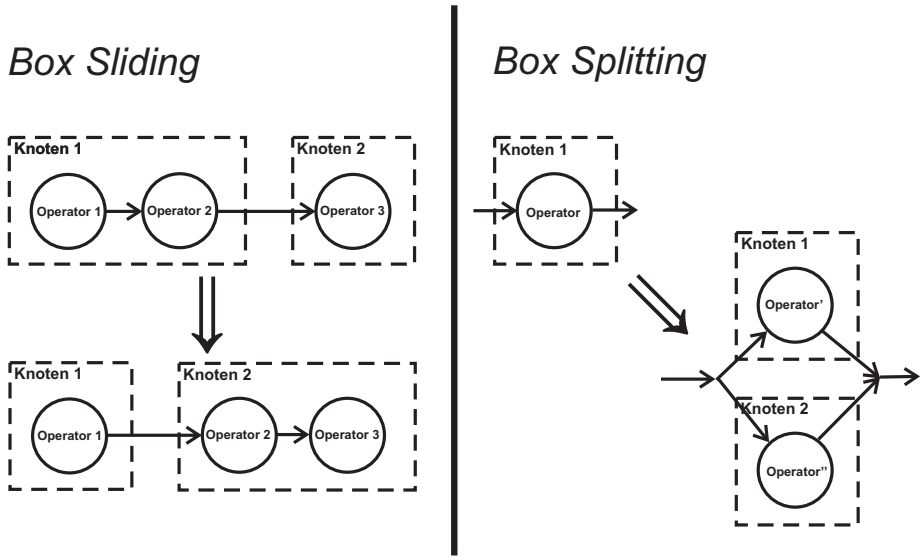

Abbildung 2.5: Box Sliding und Box Splitting

Kapitel 3

CAN-basierte P2P-Netze

3.1 Das Peer-to-Peer Netzmodell

Das Peer-to-Peer Modell (kurz: P2P) ist einer der beiden grundlegenden Ansätze für die Vernetzung von Rechnern. Der zweite Ansatz ist die Client/Server-Architektur, siehe Abbildung 3.1. Im Gegensatz zu dieser, besteht ein P2P-Netz aus gleichberechtigten Knoten, den sogenannten Peers [Ber98]. Innerhalb solcher Umgebungen existieren keine dedizierten Rechner für die Bereitstellung von bestimmten Server-Diensten. Jeder Peer kann gleichzeitig Client und Server sein. Alle Netzknoten arbeiten im Wesentlichen autonom (Abbildung 3.2). Für die Ressourcenverwaltung, Optimierung und andere Aufgaben, ist jeder Peer selbst verantwortlich. Auch die Kommunikation findet direkt zwischen den Peers statt, ohne den Umweg über eine zentrale Instanz.

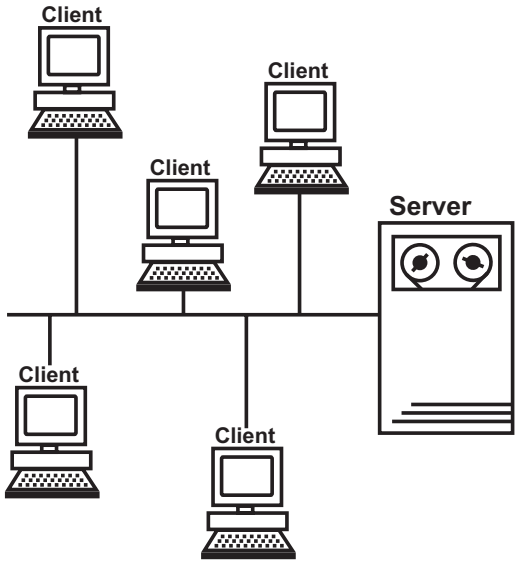

Abbildung 3.1: Client/Server-Architektur

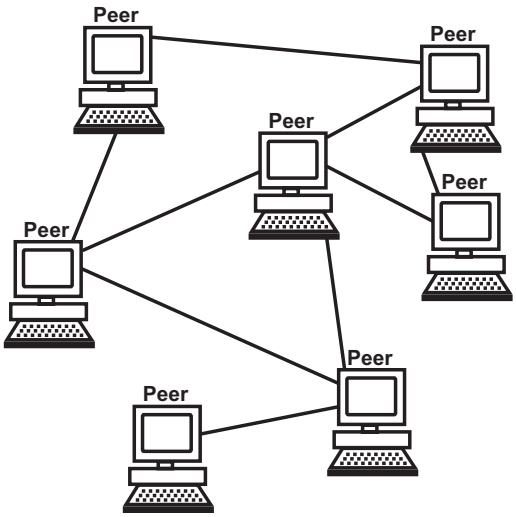

Abbildung 3.2: Peer-to-Peer Netzwerk

Da es in reinen P2P-Umgebungen keine zentrale Koordination und keine zentrale Datenbasis gibt, besitzt kein Peer eine globale Sicht auf das gesamte Netz. Ein Peer kennt

in den meisten Fällen nur seine direkten Nachbarn, seine Sicht ist also immer nur lokal. Das globale Verhalten ergibt sich erst als Summe aller lokalen Interaktionen der Peers.

Diese Art der Vernetzung bietet einige klare Vorteile. Da alle Knoten gleichwertig sind und weitestgehend autonom arbeiten, wird auch Last im Netz auf die Knoten verteilt. Damit ist die Größe von P2P-Netzen quasi unbeschränkt.

Durch das Fehlen von dedizierten Rechnern oder sonstigen Netzhierarchien, sind Flaschenhalseffekte (Single Point of Failure) durch Überlastung einzelner Rechner nicht möglich. Diese Eigenschaft macht solche Systeme auch äußerst tolerant gegenüber Ausfällen und Angriffen. Fällt ein Peer aufgrund eines Fehlers oder Angriffes aus, ist die Funktionstüchtigkeit des restlichen Netzes in der Regel nicht davon betroffen. Es stehen hauptsächlich die Ressourcen des fehlenden Peers nicht mehr zur Verfügung.

P2P-Netze sind typischerweise besser skalierbar als Client/Server-Umgebungen, da das Hinzufügen bzw. Entfernen (gewollt oder ungewollt) von Knoten nur Aktionen innerhalb der direkten Nachbarschaft des Knotens zur Folge hat. Aus diesem Grund eignen sich P2P-Netze besonders für sehr dynamische Umgebungen, in denen es im Betrieb häufig zum Einbinden oder Entfernen von Teilnehmern kommt.

Mit dem P2P-Modell können also schnell große Ressourcen (Rechenleistung, File-Sharing, ...) bereitgestellt werden, ohne dass eine besondere Netzplanung oder hohe Kosten für eine leistungsstarke Hardware (Server, Netzkomponenten, ...) notwendig sind [RFH+01].

P2P-Netze haben natürlich auch Nachteile. Aufgrund ihrer großen Dynamik gibt es keine Garantien für die Existenz von Peers und Verbindungen zwischen Peers. Ohne zusätzliche Mechanismen sind die Ressourcen fehlender oder nicht mehr erreichbarer Peers nicht zugänglich. Je nach Einsatzgebiet kann dieser Umstand toleriert oder auf geeignete Weise möglichst umgangen werden.

Doch vor allem das Finden von Daten, ohne die Verwendung einer zentralen Koordination bzw. Datenhaltung, ist schwierig. Um nicht jedes Mal das gesamte Netz mit Suchanfragen zu fluten, werden skalierbare, dezentrale Indexierungsmechanismen benötigt. Diese erlauben einen effizienten Zugriff auf die Daten, in einem vertretbaren Aufwand.

Viele existierende Implementierungen weichen das P2P-Konzept auf, indem sie zentrale Instanzen für die Indexierung der Daten einsetzen. Der eigentliche Datenaustausch findet weiterhin direkt zwischen den Peers statt (z.B. Napster). Diese Systeme sind allerdings dann deutlich schlechter skalierbar und anfälliger gegenüber Überlastung, Ausfall oder Angriff der zentralen Instanzen.

3.2 Das Content-Adressable Network

Ein *Content-Addressable Network* (kurz: CAN) [RFH+01] ist eine logische Struktur, die sich gerade für den Einsatz als Overlay-Netzwerk auf einer P2P-Umgebung eignet. Als Basis für das CAN dient ein *d*-dimensionaler, kartesischer Koordinatenraum auf einem

d-Torus. Dadurch besitzt der Koordinatenraum keine Ränder, so dass von einem Punkt aus in alle beliebigen Richtungen gegangen werden kann, ohne den Raum zu verlassen. Abbildung 3.3 zeigt einen Torus für einen 2-dimensionalen Koordinatenraum.

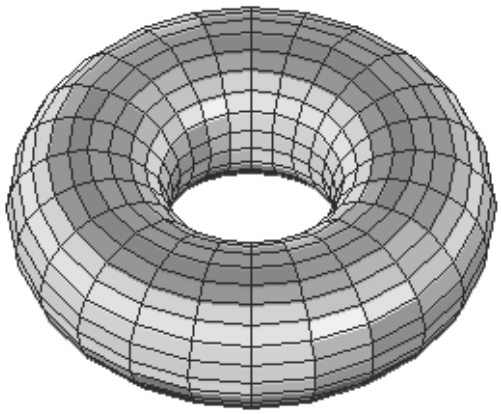

Abbildung 3.3: Torus

Dieser Koordinatenraum wird nun in disjunkte Zonen unterteilt, die durchaus unterschiedlich groß sein dürfen. Jede Zone wird von einem Knoten des unterliegenden Netzwerkes verwaltet. Es muss dabei die Bedingung erfüllt sein, dass zu jedem Zeitpunkt der gesamte Raum abgedeckt wird. In einem CAN-basierten Netz kennen alle Rechnerknoten ihre eigene Zone, sowie die Zonen ihrer direkten Nachbarn. In einem *d*-dimensionalen Raum sind zwei Zonen genau dann benachbart, wenn sie entlang einer Dimension eine gemeinsame Begrenzung haben und sie sich dort entlang der restlichen *(d-1)* Dimensionen berühren.

Abbildung 3.4 zeigt ein solche disjunkte Zerlegung des 2-dimensionalen Koordinatenraums. Die Ausdehnung des Raumes ist dabei in beiden Dimensionen auf „1“ normiert.

Die eigentliche Datenorganisation geschieht mit Hilfe einer verteilten Hash-Tabelle (DHT - *Distributed Hash Table*). Ganz allgemein versteht man unter Hashing die Abbildung von Schlüsseln auf Werte mit Hilfe der sogenannten Hash-Funktion *h(x)*. Im Falle eines CAN werden Schlüssel-Werte-Paare der Form *(key, value)* gespeichert, indem der Schlüssel *key* durch die Hash-Funktion auf einen Punkt *P* im Koordinatenraum abgebildet wird (*h(key)=P*). Auf dem Peer, der die Zone verwaltet in dem *P* liegt, wird das Datum *(key, value)* abgelegt. Das Vorgehen für das Auslesen der Daten gestaltet sich analog.

Für die gesamte Organisation der Daten werden im Kern folgende zwei Basisoperationen benötigt:

1. *put(key, value)*
   Lokales Speichern des Paares *(key, value)* auf einem Peer.

2. *get(key)->V*
   Ermitteln des Wertes *value* zum Schlüssel *key* auf einem Peer.

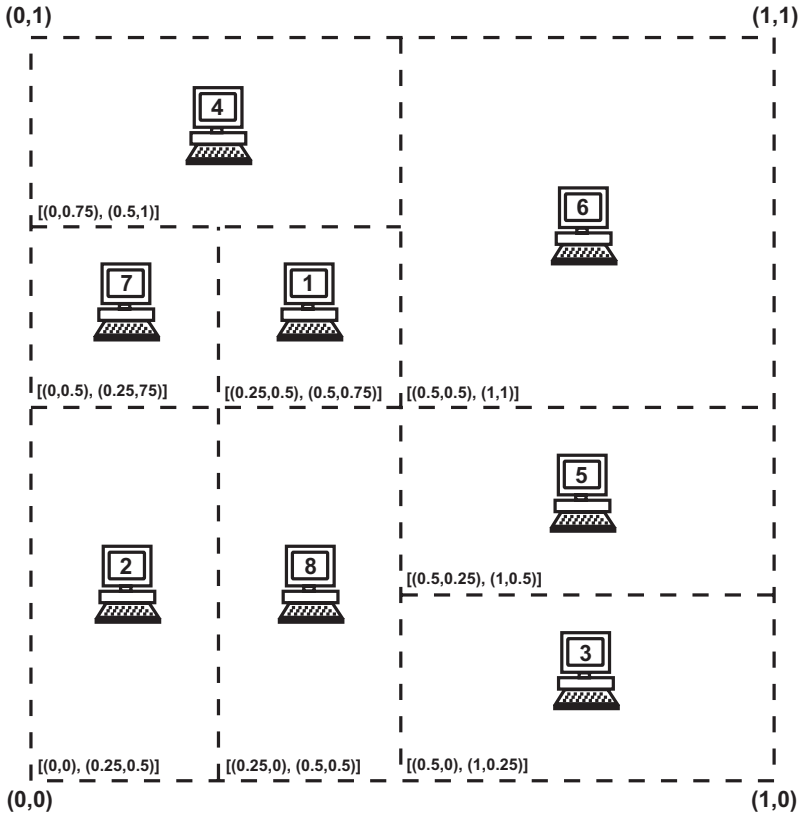

Abbildung 3.4: Beispiel CAN

Sowohl `put` als auch `get` sind lokale Operationen. Befindet sich der berechnete Punkt nicht auf dem angesprochenen Peer, muss die Anfrage zunächst an den richtigen Peer weitergeleitet werden. Diese Aufgabe übernimmt die methode `lookup(key)`, die beide Basisoperationen bei Bedarf aufrufen können. Das *lookup* ist von besonderem Interesse, da für ein schnelles Auffinden des richtigen Peers ein entsprechend effizienter Routing-Algorithmus benötigt wird.

Ein intuitiver Ansatz für einen Routing-Algorithmus ist die Weiterleitung der Anfrage entlang der Verbindungslinie zwischen dem aktuellen Peer und dem berechneten Punkt *P* im kartesischen Koordinatenraum. Da jeder Peer die Zonen seiner direkten Nachbarn kennt, kann er berechnen, wessen Zone den Punkt *P* enthält bzw. am nächsten an *P* liegt. Zu diesem Nachbarn wird nun die *lookup*-Anfrage geschickt. Abbildung 3.5 zeigt die Wegwahl für ein *lookup*, für den Fall dass Peer 1 angefragt wird, das Datum aber auf Peer 3 gespeichert ist.

Für einen *d*-dimensionalen Datenraum der in *n* gleich große Zonen unterteilt ist, lassen sich einige Kenngrößen abschätzen:

- durchschnittliche Pfadlänge einer *lookup*-Operation (Anzahl der Hops): $(d/4)n^{1/d}$
- minimale Anzahl von Nachbarn: $2d$
- Anstieg der Pfadlänge bei wachsender Anzahl von Peers: $O(n^{1/d})$

Der Exponent $(1/d)$ macht deutlich, dass die durchschnittliche Pfadlänge bzw. deren

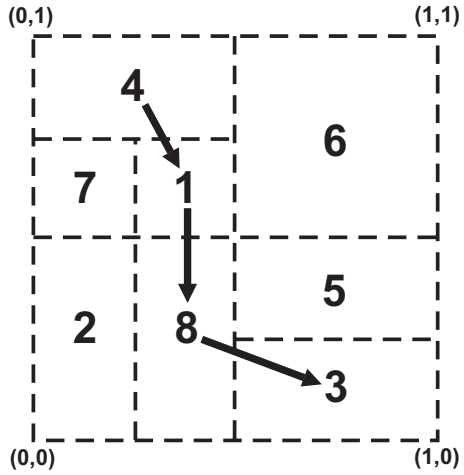

Abbildung 3.5: Beispiel für ein *lookup*

Wachstum bei einer Skalierung des Netzes mit dem Einsatz weiterer Dimensionen geringer wird.

Für das Routing existieren immer mehrere Wege zwischen zwei Punkten im Raum, so dass bei Ausfällen von Peers Alternativrouten gewählt werden können. Die Anzahl der möglichen Wege steigt mit der Dimension des Koordinatenraums.

Durch die Voraussetzung, dass der gesamte Koordinatenraum jederzeit vollständig abgedeckt sein muss, werden geeignete Algorithmen für den Beitritt und das Verlassen von Netzknoten benötigt.

Soll ein neuer Knoten in das CAN eingegliedert werden, müssen folgende drei Schritte durchgeführt werden:

1. *Finden einer belegten Zone im CAN*
   Der neue Knoten wählt einen zufälligen Punkt *P* im Koordinatenraum und schickt an den Peer, der die Zone verwaltet die *P* enthält, eine JOIN-Nachricht.

2. *Split der Zone*
   Die betreffende Zone wird in zwei (typischerweise gleich große) Teilzonen unterteilt und die Hälfte die *P* enthält an den neuen Peer übergeben. Die Wertpaare der ursprünglichen Zone werden gemäß der Aufteilung auf die beiden Peers verteilt.

3. *Aktualisierung der Nachbarschaftsbeziehungen*
   Alle betroffenen Knoten (die beiden Peers der geteilten Zone sowie alle direkten Nachbarn) müssen ihre Informationen über die Zonen ihrer Nachbarn auf die veränderte Aufteilung des Koordinatenraums anpassen.

Wenn *d* die Anzahl der Dimensionen des CANs beschreibt, dann ist die maximale Anzahl der betroffenen Knoten in der Größenordnung von *O(d)*. Der Aufwand steigt also linear mit der Anzahl der Dimensionen an.

Für das Verlassen von Peers gibt es zwei prinzipielle Möglichkeiten. Zum einen kann der Peer explizit bekannt geben, dass er das Netz verlassen will und zum anderen, wenn der Peer durch einen Ausfall, Überlastung oder Angriff nicht mehr erreichbar ist.

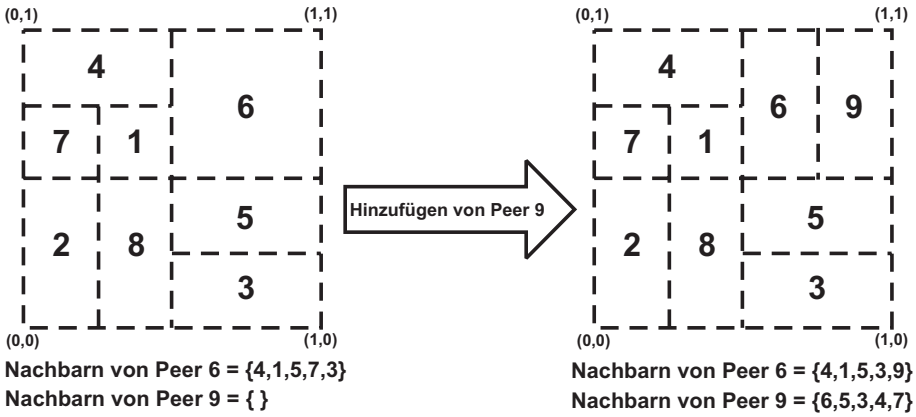

Abbildung 3.6: Einfügen eines neuen Peer in ein CAN

Gibt ein Knoten sein Verlassen explizit bekannt, genügen folgende Schritte, um eine vollständige Überdeckung des Koordinatenraums wiederherzustellen:

1. *Abgabe der verwalteten Zone*
   Der verlassende Peer übergibt seine Zone in der Regel an den Nachbarn mit der kleinsten Zone ab. Dabei müssen auch die Daten an diesen Peer übergeben werden.

2. *Benachrichtigung aller Nachbarn*
   Sowohl der Peer mit der vereinigten Zone als auch alle seine direkten Nachbarn müssen ihr Nachbarschaftsinformationen aktualisieren.

Fällt ein Knoten aus, oder ist aus anderen Gründen nicht mehr erreichbar, müssen zusätzliche Maßnahmen getroffen werden. Um überhaupt festzustellen, dass ein Nachbar nicht mehr zu erreichen ist, werden periodisch Update-Nachrichten zwischen den Peers ausgetauscht. Beim Wegfall eines Peers bleibt dessen Update-Nachricht aus. Ist dies der Fall, werden folgende Schritte durchgeführt:

1. *Aushandeln, welcher Nachbar die Zone übernimmt*
   Nach dem Ausbleiben der Update-Nachricht eines Peers, tauschen dessen Nachbarn sogenannte TAKEOVER-Nachrichten aus. Diese beinhalten im Wesentlichen die Größe der vom Peer verwalteten Zone, der die Nachricht abgeschickt hat. Dadurch kann festgestellt werden, welcher Nachbar die kleinste Zone verwaltet.

2. *Übernahme der Zone*
   Das restliche Vorgehen gestaltet sich analog zum expliziten Verlassen eine Peers (Verschmelzung der Zone, Benachrichtigung aller direkten Nachbarn). Allerdings gehen hier die Daten des ausgefallenen Peers verloren.

Um den Verlust von Daten durch einen Ausfälle zu umgehen, bedarf es zusätzlicher Sicherungsmechanismen. So können z.B. die Daten periodisch neu eingefügt werden oder eine replizierte Datenhaltung zum Einsatz kommen.

Die vorgestellten Eigenschaften und Verfahren gelten für die ursprüngliche Version eines Content-Adressable Networks. Doch es gibt noch verschiedene Erweiterungen, mit denen ein CAN verbessert werden kann. Verbesserungen können bezüglich des Routings,

der Skalierbarkeit, der Ausfallsicherheit und anderer Punkte gemacht werden. Mögliche Erweiterungen sind z.B.:

- *Erhöhung der Dimensionalität*
  Wegen $(d/4)n^{1/d}$ verringert sich die durchschnittliche Pfadlänge beim Routing, bei einem vergleichsweise geringen Mehraufwand für das Verwalten der zusätzlichen Routing-Einträge auf jedem Peer. Mit steigender Dimension wächst auch die Anzahl der Alternativrouten zwischen zwei Peers und damit die Ausfallsicherheit.

- *Verwaltung mehrer unabhängiger Koordinatenräume*
  Es kommen für die Datenorganisation mehrere Hash-Funktionen zum Einsatz. Ist *n* die Anzahl der Hash-Funktionen, so wird ein Datum auf *n* verschiedene Stellen gespeichert. Während des Routings kann dann zwischen den verschiedenen Hash-Funktionen gewechselt werden. Dieses Verfahren erhöht zum einem die Ausfallsicherheit (Daten werden mehrfach gespeichert) und zum anderen die Effizienz des Routings (es wird die Hash-Funktion mit dem kürzesten Abstand zum gesuchten Punkt gewählt).

- *Zusätzliche Routing-Metriken*
  Die vorgestellte Strategie, den schnellste Weg für das Routing auf Basis der geringsten Distanz im Koordinatenraum zu finden, beschränkt sich allein auf die Ebene des CAN. Peers, die im Koordinatenraum direkte Nachbarn sind, können auf physischer Ebene sehr weit auseinander liegen (hohe Kommunikationskosten). Es ist daher sinnvoll für das Routing auch Parameter des unterliegenden physischen Netzes heranzuziehen.

- *Überladen von Zonen*
  Anstatt jede Zone von genau einem Peer verwalten zu lassen, sind hier mehrere Peers für eine Zone zuständig. Dabei muss der Peer nicht nur seine Nachbarn, sondern auch die Peers, mit denen er sich seine Zone teilt, kennen. Die Daten einer Zone können entweder fragmentiert oder repliziert auf die Peers verteilt werden, je nachdem ob die Ausfallsicherheit oder die Performanz verbessert werden soll.

Alles in allem ist das Content-Adressable Network, aufgrund seine völlig dezentralen Datenorganisation und der sehr guten Skalierbarkeit, bestens geeignet zur Indexierung von Daten in einer Peer-to-Peer Umgebung.

3.3 Organisation relationaler Daten

Mit Hilfe der verteilten Hash-Tabelle können einzelne Schlüssel in einem CAN gut verwaltet werden. In den meisten Anwendungen werden allerdings strukturierte Daten verarbeitet. In der Datenbankwelt ist vor allem das relationale Datenmodell zu finden. Auch in dieser Arbeit kommen relationale Daten in Form von Tupeln zum Einsatz. Im Folgenden soll kurz eine Möglichkeit vorgestellt werden, wie sich Relationen in eine verteilte Hash-Tabelle speichern lassen. Eine genaure Beschreibung des Verfahrens findet

sich in [RvdWSB04].

Das Einfügen von Tupeln geschieht mit Hilfe der *put*-Operation. Gegebenenfalls wird zuvor ein *lookup* benötigt um den entsprechenden Peer zu finden, der dann das lokale *put* durchführt.

Beide Operationen benötigen als Eingabeparameter eine *key*-Wert. Für relationale Daten eignet sich das Wertepaar aus dem Relationennamen und dem Attributwert des Primärschlüssels, da diese beiden Größen ein Tupel innerhalb einer Datenbank eindeutig identifizieren. In welcher Form der Relationenname und der Attributwert dargestellt werden, ob als String, als numerischer oder anderweitig abgeleiteter Wert, spielt dabei keine Rolle. Der *value*-Wert für das *put(key, value)* ist in der Regel das gesamte Tupel.

Die besonderen Eigenschaften des CAN zeigen sich gerade bei der Suche nach Tupeln. Bestimmt wird das Vorgehen durch das Konzept der verteilten Hash-Tabelle. Wie beim Hashing üblich, kann auf die Daten nur über die Schlüsselwerte zugegriffen werden, nach denen die Daten auch verteilt wurden. Aus diesem Grund können Anfrage in drei Klassen eingeteilt werden. Für nachfolgende Beispielanfragen soll die Relation `student(MatNr, Matrikel, Studiengang)` dienen.

Punktanfragen auf Werten des Primärschlüsselattributes. Dazu zählt z.B. eine Anfrage der Form:

- `SELECT * FROM student WHERE MatNr=29093`

Diese Art Anfrage stellen den Idealfall dar, alle Größen die für das *lookup* und *get* benötigt werden, stehen direkt zur Verfügung. Der *key*-Wert kann direkt aus dem Relationennamen und dem gesuchten Wert für das Primärschlüsselattribut gebildet und in die Basisoperationen eingesetzt werden. Für obige Anfrage würde sich also ergeben: *key=(student, 29093)*.

Bei solchen Anfragen zeigen sich die Vorteile des Hash-Verfahrens. Auf die Tupel kann quasi direkt zugegriffen werden, da mit den gleichen Schlüsseln gesucht wird, mit denen die Tupel eingefügt worden sind.

Bereichsanfragen auf Werten des Primärschlüsselattributes. Bespiele für diese Art Anfragen sind:

- `SELECT * FROM student` *oder*
- `SELECT * FROM student WHERE 20000<MatNr<30000`

Da hier der *key*-Wert nicht eindeutig bestimmt werden kann, ist auch eine gezielte Anfrage an den entsprechenden Peer nicht möglich. Um das Problem zu lösen, stehen zwei Ansätze zur Verfügung.

1. *Broadcast/Multicast der Anfrage.*
   Die Anfrage wird an alle Peers im Netz verteilt. Jeder Peer prüft, ob er Tupel der entsprechenden Relation besitzt (hier: `student`). Werden Tupel gefunden, wird lokal die Selektionsbedingung überprüft und bei Erfolg das Ergebnistupel zum anfragenden Peer zurückgeschickt.

Ohne eindeutige Schlüsselwerte können bei diesem Verfahren nicht die Basisoperationen eingesetzt werden. Somit führt hier das Hashing eher zu Problemen.
Falls es das CAN unterstützt, werden nur die Teilbereiche des Netze angesprochen, die tatsächlich Tupel für die Basisrelationen besitzten. Aus dem „teuren“ Broadcast wird so im Mittel mehrere „günstigere“ Multicast. Diese Fähigkeit ist allerdings von einigen Bedingungen abhängig (z.B. der Datenverteilung) und soll an dieser Stelle nicht weiter betrachtet werden.

2. *Mehrere Punktanfragen auf die Elemente des Wertebereiches des Primärschlüsselattributes.*
   Diese Option steht immer dann zur Verfügung, wenn der Wertebereich des Primärschlüsselattributes durch die Anfrage beidseitig begrenzt ist (hier: `20000<MatNr<30000`). In solchen Fällen kann für alle Zwischenwerte Punktanfragen gestellt werden. Obige Beispielanfrage müsste also in 9999 einzelne Punktanfragen unterteilt werden.

Der Broadcasting-Ansatz ist immer durchführbar, bedeutet aber in der Regel immer eine hohen Aufwand, besonders in großen und weitverteilten Netzen. Ein Broadcast und damit der Verzicht auf den Einsatz der Basisfunktionen für das Hashing, mach dieses Verfahren auch praktikabel für Anfragen an Nichtschlüsselattribute.

Ob eine Aufteilung in Punktanfragen möglich ist, hängt von der beidseitigen Beschränktheit des Wertebereich des Primärschlüsselattributes ab. Je größer der Wertebereich, desto größer ist auch die Anzahl der resultierenden Punktanfragen. Es sollte also abgeschätzt werden, welches Verfahren sich für eine aktuelle Anfrage eignet.

Anfragenverarbeitung mit Hilfe einer Neuverteilung der Tupel benötigen. Die beiden bisherigen Klassen von Anfragen sind in erster Linie für Anfrage bzw. Datenbankoperationen praktikabel, die immer nur einzelne Tupel verarbeiten. Solche Operationen sind beispielsweise die Selektion und Projektion. Doch bei einigen Datenbankoperationen werden zwei oder mehr Tupel im Verbund verarbeitet, es existieren quasi Abhängigkeiten zwischen den Eingangstupeln. Zu solchen Operatoren zählen die binären Operatoren (Joins) und Operatoren wie die Gruppierung und Aggregation. Ein mögliche Anfrage auf die Beispielrelation `student` ist:

- `SELECT MAX(MatNr) FROM student`

Bei dieser Anfrage müssen alle Tupel als Ganzes bearbeitet werden, um das korrekte Ergebnis zu erhalten. Das Problem liegt nun darin, dass die Tupel ein Relation typischerweise auf mehr als einem Peer gespeichert sind, aber jeder Operator letztlich lokal auf einem Peer ausgeführt werden muss.

Ein intelligentes Verfahren, um dieses Problem zu beheben, ist eine Neuverteilung der betreffenden Tupel, das so genannte Re-Hashing. Die Tupel, zwischen denen für eine Operation eine Abhängigkeit besteht, werden dabei so neu verteilt, dass sich zum Schluss alle Tupel auf einem Peer befinden. Grundlage dafür ist das Finden eines geeigneten *key*-Wertes für die Neuverteilung. Verteilt werden natürlich nur Kopien der Tupel, da die Originaltupel der Relation weiterhin dort zu finden sein müssen, wo sie durch das initiale Einfügen gespeichert wurden. Die Reihenfolge der Abarbeitung einer Datenbankoperation unter zu Hilfenahme einer Neuverteilung besteht aus drei Schritten:

1. *Finden aller betreffenden Tupel.*
   Dies geschieht entweder durch einen Broadcast an alle Peers oder, falls es die Anfrage erlaubt, durch die Aufteilung in Punktanfragen (vgl. *Bereichsanfragen auf Werte des Primärschlüsselattributes*)

2. *Re-Hashing der Tupel.*
   Kopien benötigten Tupel werde mit Hilfe eines eindeutigen *key*-Wertes neu verteilt. Dieser Wert muss so gewählt oder berechnet werden, dass die Tupel garantiert auf demselben Peer landen.

3. *Ausführen der lokalen Datenbankoperation.*
   Da nun alle benötigten Tupel lokal zur Verfügung stehen, kann die Operation ausgeführt werden.

3.4 Fazit

Durch die Verwendung einer verteilten Hash-Tabelle eignet sich ein Contant Adressable Network zur Verteilung bzw. Indexierung auch strukturierter Daten in weitverteilten Umgebungen. Der Verzicht auf eine zentrale Koordination und die gute Skalierbarkeit machen eine Kombination aus CAN und P2P-Netze besonders interessant.

Diese Eigenschaften sind es auch, die für den Einsatz einer dynamischen Anfrageverarbeitung sprechen. Die Gründe hierfür wurden in Abschnitt 2.1.3 näher erläutert.

Alle bisherigen Umsetzungen adaptiver Anfrageverarbeitungen sind nicht gezielt für P2P-Netze entwickelt worden und liefern dadurch keine zufriedenstellenden Ergebnisse. Selbst verteilte Eddies stellen nicht die optimale Lösung dar, da sie dem Konzept gleichberechtigter Knoten in einem P2P-Netz widersprechen. Doch gerade in solchen Systemen kann die Verteilung der Operatoren sinnvoll sein, um nicht auf dedizierte Knoten angewiesen zu sein.

Benötigt wird also nicht nur eine Anfrageverarbeitung, die sich dynamisch an veränderte Systemumgebungen anpassen kann, sondern auch auf die Ziele von P2P-Umgebungen (Skalierbarkeit, Dynamik, Robustheit,...) eingeht. Genau diese Lücke soll der P2P-Eddy schließen, indem er in weiten Bereichen alle Peers gleichberechtigt in die Verarbeitung einer Anfrage miteinbezieht und so auch eine Verteilung von Operatoren ermöglicht.

Kapitel 4

Entwurfskonzept

4.1 Ausgangssituation

Nicht alle Phasen der Anfrageverarbeitung sollen in dieser Arbeit betrachtet werden. Im Kern werden dies die Optimierung und die Ausführung einer Anfrage sein.

Vorausgesetzt wird zum einen eine vollständige Vorverarbeitung. Dies beinhaltet im Wesentlichen die Umformung einer deskriptiv formulierten Anfrage in eine interne Darstellung des Datenbanksystems (Anfragebaum). Wie in Punkt 2.1.1 beschrieben, umfasst dieser Schritt auch den Syntaxcheck der Anfrage sowie deren Validierung.

Weiterhin wird eine Voroptimierung vorgenommen. Dabei wird versucht, die interne Baumstruktur der Anfrage zu minimieren. Eine Minimierung ist immer dann möglich, wenn der Baum redundante Teilbäume oder auch unnötige Operationen enthält. Solche Fehler können durch „ungeschickt“ gestellte Anfragen der Nutzer oder automatisierte Anfrageerzeugung von Anwendungsprogrammen auftreten. Auch die Auflösung von Sichten ist ein häufiger Grund für Redundanzen im Baum.

Um den Aufwand für die Implementierung des P2P-Eddies in Grenzen zu halten, wird dessen Leistungsumfang bewusst eingeschränkt. Zum einen werden nur die wichtigsten Operationen umgesetzt (Selektion, Projektion, Join) und zum anderen werden auch bestimmte Voraussetzungen für die Operatoren selbst getroffen. So wird z.B. ein Join immer nur über einem Attribut ausgeführt. Dazu mehr bei der konkreten Implementierung der Operatoren.

Ausgangsbasis für das weitere Vorgehen ist also eine Anfrage mit folgenden Eigenschaften:

- Darstellung als Baum
- auf Gültigkeit geprüft
- voroptimiert
- eingeschränkter Leistungsumfang

Zusätzlich befindet sich die Anfrage in einer Form, die direkt weiterverarbeitet werden kann.

4.2 Grundidee P2P-Eddies

Wie schon bei den beiden vorgestellten Eddy-Mechanismen, soll die Dynamik der Verarbeitung einer Anfrage durch eine Modifikation des Anfrageplans erreicht werden. Die Modifikation umfasst dabei lediglich eine Neuordnung der logischen Operatoren. Es werden keine Alternativpläne, durch Hinzufügen, Entfernen oder Veränderung der Operatoren, erzeugt.

Um den Unterschied zwischen P2P-Eddy und den urprünglichen Eddy-Varianten zu verdeutlichen, ist eine kurze Vorüberlegung hilfreich. Um eine Anfrage zu bearbeiten, sind grob betrachtet drei Dinge notwendig:

1. die Nutzdaten (Tupel)
2. die Vorschrift, wie die Daten verarbeitet werden sollen (z.B. Anfrageplan)
3. ein Rechner (CPU), der die Verarbeitung ausführt

Nutzdaten und Verarbeitungsvorschrift müssen sich also zur gleichen Zeit auf dem gleichen Peer befinden.

Der zentralisierte Eddy und die verteilten Eddies gehen dabei so vor, dass sie die Verarbeitungsvorschrift fest an einen Rechner binden, sei es in einer zentralen Instanz oder dedizierten Peers (Abbilding 4.1). Für die konkrete Anfrageverarbeitung werden nun die Tupel zur zentralen Komponente bzw. den entsprechenden Knoten gebracht.

Um die Gleichberechtigung aller Rechner in einem P2P-System zu erhalten, geht der P2P-Eddy einen anderen Weg. Hier wird die Verarbeitungsvorschrift direkt an die Tupel gebunden (Abbildung 4.2). Für die Anfrageverarbeitung wird nur noch ein beliebiger Rechnerknoten benötigt.

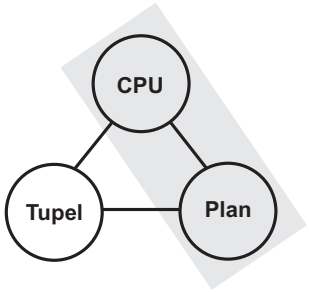

Abbildung 4.1: Verarbeitungsprinzip in herkömmlichen Anfrageverarbeitungen

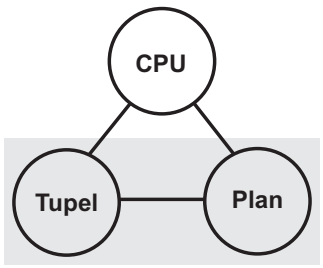

Abbildung 4.2: Verarbeitungsprinzip des P2P-Eddies

Die Darstellung einer Anfrage als Baum gibt durch ihre Struktur implizit eine Verarbeitungsreihenfolge für die Tupel vor. Die Abarbeitung erfolgt dabei von den Blättern in Richtung der Wurzel. Im ersten Schritt wird also eine Darstellungsform benötigt, die auf eine festgelegte Reihenfolge ihrer Elemente verzichtet. Eine solche Struktur ist z.B. eine einfache Aufzählungsliste. Die Elemente dieser Liste sind die logischen Operatoren aus

dem Anfragebaum, die auf den Tupeln ausgeführt werden. Im Folgenden wird diese Liste deshalb auch als Todo-Liste bezeichnet. Ziel ist es nun, jedes Tupel mit einer solchen Liste von Operatoren zu versehen. Prinzipiell können auch mehrere gleichartige Tupel (Tupel mit gleichem Schema) zusammengefasst werden. Für die Vorstellung des gesamten Verfahrens wird aus Gründen der Übersichtlichkeit, jedes Tupel mit einer Todo-Liste versehen.

Beim Routing durch das Netz kann jetzt jeder beliebige Peer die Tupel verarbeiten, da jedes Tupel die Informationen mitbringt, welche Operatoren noch ausgeführt werden müssen. Dieses Verfahren verzichtet auf die Auswahl bestimmter Knoten im Netz und ist somit für P2P-Umgebungen bestens geeignet.

Die Todo-Liste darf nur die Operatoren enthalten, die auch tatsächlich auf die zugehörigen Tupel angewendet werden dürfen. Die Information, welche Operatoren für welche Tupel gelten, steckt vollständig in der ursprünglichen Baumstruktur. Die Blätter des Anfrageplans sind immer die benötigten Relationen für die Anfrage.

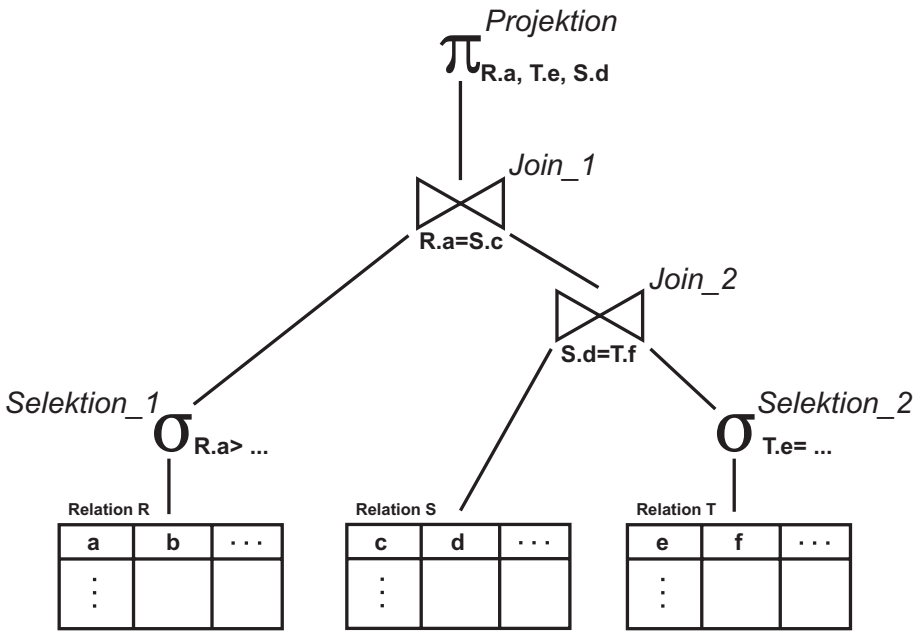

Abbildung 4.3: Beispiel-Anfragebaum

Um die richtigen Operatoren für die Tupel einer Ursprungsrelation zu finden, muss der Weg von der Relation bis zur Wurzel verfolgt werden. Pro Relation gibt es demnach immer eine Todo-Liste. Enthält der Plan auch binäre Operatoren, müssen nicht zwangsläufig alle Operatoren auf dem Pfad für die Tupel einer Ursprungsrelation gelten. Durch die Eigenschaften der Operatoren, ist die eindeutige Zuordnung zu den Tupeln möglich. Für die Beispielanfrage (Abbbildung 4.3) ergeben sich somit diese drei Todo-Listen (Abbildung 4.4).

Bevor mit der eigentlichen Abarbeitung der einzelnen Todo-Listen begonnen werden kann, müssen diese zunächst an alle Peers verteilt werden. Jeder Peer prüft darauf hin, ob er für die aktuelle Todo-Liste zugehörige Tupel besitzt. Ist dies der Fall, werden die Tupel eindeutig ihrer Todo-Liste zugeordnet. Danach sind die Tupel bereit für die Verarbeitung der Anfrage.

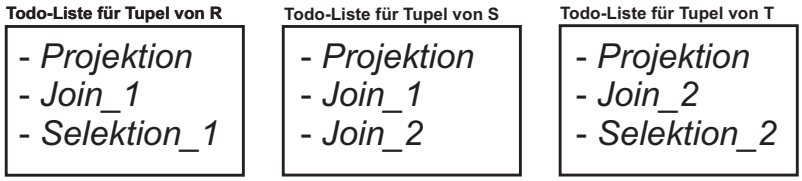

Abbildung 4.4: Todo-Listen für den Beispiel-Anfragebaum

Obwohl die Todo-Liste selbst keine Reihenfolge für die Abarbeitung der Operatoren vorgibt, müssen dennoch Verletzungen von Umformungsregeln der Relationenalgebra abgefangen werden. Dafür wird jedes Element der Todo-Liste mit einem Ready-Bit versehen. Nur bei gesetzten Ready-Bit darf der Operator ausgeführt werden. Nach jeder Abarbeitung eines Operators, müssen die Ready-Bits der restlichen Elemente der Todo-Liste aktualisiert werden. Auf ein Done-Bit, wie es bei den ursprünglichen Eddy-Mechanismen zum Einsatz kommt, kann hier verzichtet werden. Ein Operator wird nach seiner Ausführung einfach aus der Todo-Liste entfernt.

Bei unären Operatoren wie der Selektion oder Projektion, ist die Abarbeitung relativ problemlos. Erhält ein Peer ein Paket aus Tupel und Todo-Liste, wählt er einen Operator aus und arbeitet diesen auf dem zugehörigen Tupel ab. Im Anschluss wird dieser Operator aus der Liste gelöscht und die Ready-Bits der restlichen Operatoren gemäß der veränderten Situation aktualisiert. Hat der Operator ein Ergebnistupel hervorgebraucht, wird dies der aktualisierten Todo-Liste zugeordnet und weitergeleitet.

Etwas komplexer stellt sich de Sachverhalt bei binären Operatoren dar. Da hier Tupel unterschiedlicher Herkunft miteinander verknüpft werden, müssen auch die zugehörigen Todo-Listen entsprechend verarbeitet werden. Die Todo-Liste für mögliche Ergebnistupel muss dabei alle Elemente der beiden ursprünglichen Todo-Listen enthalten. Zunächst wird, wie gehabt, der aktuell abgearbeitete, binäre Operator aus den beiden Listen entfernt. Im zweiten Schritt werden beide Listen verschmolzen. Dieser Vorgang entspricht einer Vereinigung von Mengen. Ist ein Operator in beiden Eingangs-Todo-Listen enthalten, wird er nur einmal in die Ausgangs-Todo-Liste übernommen. Die restlichen Schritte werden analog zu den unären Operatoren durchgeführt. Es werden die Ready-Bits aktualisiert und die Todo-Liste mit den Ergebnistupeln verknüpft.

Abbildung 4.5 zeigt schematisch das Verschmelzen zweier Todo-Listen nach Ausführung des Joins *Join_1* für zwei Tupel aus den Relationen R und S der Beispielanfrage (Abbildung 4.3).

Für die Ausführung der logischen Operatoren werden auch weiterhin Implementierungen der verschiedenen Datenbankoperationen benötigt. Alle anderen Aufgaben werden durch eine Art Superoperator erledigt, im Folgenden als Eddy-Operator oder einfach als (P2P-)Eddy bezeichnet. Zu dessen Hauptaufgaben zählen:

- Erzeugung der Todo-Listen
- Broadcast der Todo-Listen
- Zuordnung zwischen Tupeln und Todo-Listen herstellen
- Verteilung der Pakete

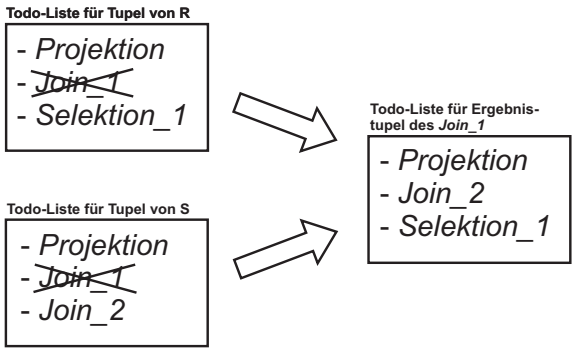

Abbildung 4.5: Verschmelzung zweier Todo-Listen nach einem Join

- Auswahl der Operatoren aus den Todo-Listen
- Aktualisierung der Todo-Listen

Ein vollständiges, im Netz verschicktes Paket besteht also den zwei Komponenten:

1. Tupel (oder Menge gleichartiger Tupel)
2. Todo-Liste

Dieser zusammengehörige Verbund soll im Weiteren als Nachrichtenpaket bezeichnet werden. Aus welchen Teilen sich die Komponenten im Einzelnen zusammensetzen, wird sich im Rahmen der Implementierung zeigen.

Abgesehen von den initialen Operationen des Eddies, wie dem Erzeugen und Verteilen der Todo-Listen, läuft die Abarbeitung der Todo-Listen nach folgendem Schema ab:

1. *Ankunft eines Nachrichtenpaketes bei einem Peer.*
   Dies kann jeder beliebige Peer im Netz sein. Der Peer hat ab diesem Zeitpunkt Zugriff auf das Tupel, den Ausführungsplan in Form der Todo-Liste.

2. *Auswahl des nächsten Operators für die Ausführung.*
   Der Eddy-Operator wählt auf Basis unterschiedlicher Strategien einen Operator aus der Todo-Liste aus. Dieser Operator wird dabei gleich aus der Liste entfernt.

3. *Ausführung des Operators.*
   Es wird eine geeignete Implementierung für den gewählten logischen Operator auf dem Tupel ausgeführt. Liefert die Operation kein Ergebnistupel, kann abgebrochen werden. Sonst weiter mit 4.

4. *Aktualisierung der Todo-Liste.*
   Umfasst das Setzen von Ready-Bits und das Verschmelzen von Todo-Listen nach binären Operatoren. Ist die Todo-Liste leer, ist die Bearbeitung beendet und das Ergebnistupel kann ausgegeben werden. Sonst weiter mit 5.

5. *Erzeugung des neuen Nachrichtenpaketes.*
   Das Ergebnistupel, der Eddy-Operator und die aktualisierte Todo-Liste werden zu einem neuen Nachrichtenpaket verschnürt und weitergeleitet.
   Weiter mit 1.

Erzeugt ein Operator kein Ergebnistupel, ist eine weitere Verarbeitung nicht möglich und das gesamte Nachrichtenpaket kann verworfen werden.

Abbildung 4.6 veranschaulicht noch einmal graphisch das Prinzip des P2P-Eddies. Auf jedem Peer läuft lokal eine Instanz des P2P-Eddies. Der Eddy nimmt Nachrichtenpa-

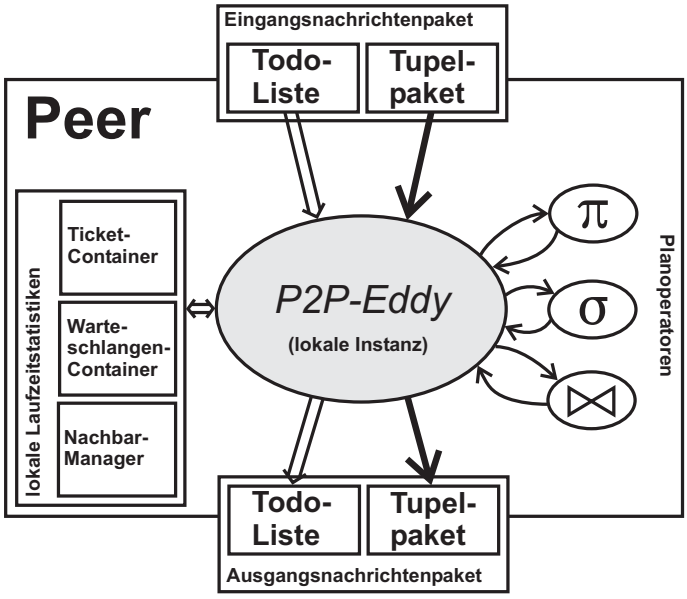

Abbildung 4.6: Prinzip des P2P-Eddies

kete (Tupel + Todo-Liste) entgegen und verarbeitet diese. Dazu gehören das Bestimmen des nächsten Operators aus der Todo-Liste, die Ausführung des Operators mit Hilfe der Planoperatoren und ggf. das Weiterschicken möglicher Ergebnistupel.

Für die Auswahl eines Operators oder eines Ziel-Peers dienen verschiedene Routing-Strategien. Diese Strategien basieren teilweise auf Laufzeitstatistiken, wie der Warteschlangenlänge, Selektivität und Informationen über die direkten Nachbarn eines Peers.

4.3 Routing-Strategien

4.3.1 Allgemeines

Bisher wurde das Konzept für die dynamische Anfrageverarbeitung nur soweit vorgestellt, dass eine variable Operatorreihenfolge für die Tupel auf beliebigen Peers überhaupt möglich ist. Jetzt fehlen noch die Mechanismen für die Entscheidungsfindungen, mit deren Hilfe die Anfrageverarbeitung auch effizient wird. Es müssen zwei grundsätzliche Entscheidungen getroffen werden:

1. *Operator-Routing*
   „Welcher Operator soll als nächstes ausgeführt werden?“

2. *Peer-Routing*
   „Zu welchem Peer soll ein Nachrichtenpaket als nächstes geschickt werden?“

Für beide Varianten werden Strategien benötigt, die obige Fragestellung so beantworten können, dass die Anfrageverarbeitung möglichst effizient geschieht.

Daneben sollte darauf geachtet werden, dass sich der Aufwand für die Routing-Strategien in Grenzen hält. Der Mehraufwand für die Entscheidungsfindungen setzt sich dabei wie folgt zusammen:

- *Verschicken zusätzlicher Nachrichten*
  Der Einsatz von Nachrichten allein für die Verteilung von Routing-Informationen sollte vermieden werden. Gerade in weitverteilten Systemen tragen die Kommunikationskosten entscheidend zum Gesamtaufwand bei. Es wird daher versucht, die Informationen ausschließlich in den vorhandenen Nachrichtenpaketen unterzubringen.

- *Umfang von Zusatzinformationen*
  Die Routing-Strategien basieren in der Regel auf verschiedenen Parametern. Diese Parameter müssen an geeigneter Stelle hinterlegt und auch dort gepflegt werden. Vor allem ein Erweiterung der Nachrichtenpakete kann sich auf die Performanz negativ auswirken, da mit wachsender Paketgröße auch die Netzlast steigt. Zusatzinformationen auf den Peers spielen eine untergeordnete Rolle.

- *zeitlicher Overhead*
  Wie beschrieben, werden sämtliche Routing-Entscheidungen durch den Eddy-Operator getroffen. Dafür werden zusätzliche Methoden eingesetzt, deren Ausführung logischerweise Zeit in Anspruch nimmt. Neben der eigentlichen Entscheidung, ist der Eddy auch für die Aktualisierung der benötigten Parameter zuständig. Dies bedeutet wiederum mehr Methoden und damit mehr Overhead.

Die vorgestellten Strategien unterscheiden sich teilweise sehr stark hinsichtlich ihres Aufwandes. Ob sich der Mehraufwand für komplexe Routing-Strategien auch lohnt, wird sich später im realen Einsatz zeigen.

4.3.2 Operator-Routing

Für das Operator-Routing werden vier verschiedene Strategien umgesetzt. Aus der Todo-Liste darf nur ein Operator gewählt werden, dessen Ready-Bit gesetzt ist. Andernfalls werden die Regeln der Relationenalgebra verletzt und es kommt zu falschen Ergebnissen oder gar Fehlern.

Zufällige Auswahl. Der nächste Operator für die Ausführung wird einfach per Zufall aus der Todo-Liste entnommen. Der Aufwand ist minimal. Zum einen sind keine weiteren Parameter nötig und zum anderen kann auch der zeitliche Overhead vernachlässigt werden. Allerdings können besonders ungünstige Operatorreihenfolgen nicht vermieden werden.

Auswahl nach Priorität. Für jeden Operatortyp (Selektion, Projektion, Join) werden Prioritäten vergeben. Diese können ein einfacher Zahlenwert sein. Operatoren mit einer erfahrungsgemäß kleineren Selektivität erhalten eine höhere Priorität als andere.

Damit können z.B. Heuristiken wie „Selektion vor Join“ leicht umgesetzt werden. Im Mittel werden dadurch besonders schlechte Entscheidungen vermieden.

Je mehr Operatoren eines Typs vorhanden sind, desto weniger können gute Operatorreihenfolgen garantiert werden. Besitzen mehrere Operatoren die gleiche höchste Priorität, muss quasi wieder zufällig ein Operator aus dieser Teilmenge bestimmt werden. Vorteil ist aber auch wieder der verhältnismäßig geringe Aufwand. Neben dem zusätzlichen Parameter, beträgt die Suche nach dem Operator mit der höchsten Priorität maximal einen vollständigen Durchlauf durch die Todo-Liste.

Auswahl nach Länge der Eingangswarteschlange. Wie schon bei den bekannten Eddy-Mechanismen, wird auch hier die Länge der Eingangswarteschlange eines Operators als Maß für dessen Kosten angesehen werden. Schnelle Operatoren können ihre Tupel zügig verarbeiten und die Warteschlange umso schneller leeren. Je länger die Eingangswarteschlange eines Operators also ist, desto höher sind typischerweise auch dessen Kosten.

Im Gegensatz zu den verteilten Eddies, kann hier jeder Operator quasi auf jedem beliebigen Peer ausgeführt werden. Durch diese Verteilung ist aber auch die Warteschlange des Operators verteilt. Dies bringt einige Nachteile mit sich:

- *eingeschränkte Aussagekraft*
  Die Entscheidung für das Routing wird lokal auf den Peers getroffen. In Abhängigkeit von deren Vorgeschichte, können von Peer zu Peer unterschiedliche Entscheidungen für den gleichen Operator getroffen werden.

- *eindeutige Entscheidungen nicht immer möglich*
  Enthält die Todo-Liste Operatoren, die einem Peer noch unbekannt sind (sind also auf diesem Peer noch nicht ausgeführt worden), können keine Aussagen über dessen Kosten über die Warteschlangenlänge gemacht werden. In diesem Fall wird aus der Vereinigung der unbekannten Operatoren und dem Operator mit der kürzesten Warteschlange der Operator mit der höchsten Priorität gewählt. Gibt es mehrere Operatoren mit einer kürzesten Warteschlange, ist das Verhalten analog.

- *Degenerierung des Verfahrens zu „Auswahl nach Prioritäten“ im Worst-Case*
  Erzeugt eine Anfrage nur wenige Nachrichtenpakete, die zusätzlich weit über das Netz verteilt sind bzw. werden, erhöht sich die Wahrscheinlichkeit, dass die Pakete bei ihrer Abarbeitung auf Peers landen, die für diese Anfrage noch nicht verwendet wurden. Somit sind dem Peer sämtliche Operatoren unbekannt (oder zumindest der größte Teil) und die Auswahl des Operators wird aufgrund der Prioritäten vorgenommen. Diese Wahrscheinlichkeit steigt weiterhin mit der Größe des Netzes und der Verteilung der Daten.

Damit ein Peer einen Operator immer eindeutig zuordnen kann, wird für jeden Operator eine ID benötigt. Alle anderen Erweiterungen müssen nur auf dem Peer gemacht werden. Dadurch ist der Mehraufwand für dieses Verfahrens eher gering.

Auswahl nach erlernter Selektivität. Die Verwendung der Selektivität von Operatoren als Auswahlkriterium für eine Operatorreihenfolge findet man nahezu in allen

Datenbanksystemen. Bei der traditionellen Anfrageverarbeitung werden die Selektivitäten anhand von Statistiken berechnet bzw. abgeschätzt. In verteilten Umgebungen stehen diese Informationen in den meisten Fällen nicht zu Verfügung. Beim zentralisierten und verteilten Eddy-Mechanismus [AH00, TD03] werden die Selektivitäten zu Ausführungszeit, mit Hilfe eines Ticket-Mechanismus, erlernt.

Ein ähnliches Ziel soll auch bei diesem Verfahren verfolgt werden. Durch die verteilten Operatoren wird allerdings eine andere Art der Umsetzung benötigt. Ein Ticket besteht im Wesentlichen aus zwei Zählern für die Eingangs- und die Ergebnistupel eines Operators. Jedem Peer wird für jeden Operator einer Anfrage ein solches Ticket zugewiesen. Kommt ein Operator auf einem Peer zur Ausführung, wird zunächst der Zähler der Eingangstupel gemäß der Tupelanzahl des Nachrichtenpaketes erhöht. Verlassen Tupel erfolgreich den Operator, wird deren Anzahl auf den Ergebnistupelzähler des Tickets aufaddiert. Ein Ticket ist demnach umso aussagekräftiger, je öfter ein Operator auf dem gleichen Peer ausgeführt wurde.

Die Selektivität eines Operators berechnet sich anhand seines Tickets nach folgender Formel:

$$Selektivität = \frac{Anzahl\ der\ Ergebnisstupel}{Anzahl\ der\ Eingangstupel}$$

Für die Abarbeitung wird also immer nach dem Operator mit der kleinsten Selektivität gesucht. Bei diesem ist die Wahrscheinlichkeit, dass keine bzw. wenig Ergebnistupel entstehen am größten. Auch hier kann es vorkommen, dass ein Peer keine eindeutige Entscheidung treffen kann, falls noch kein Ticket für einen Operator auf dem Peer vorliegt oder es mehrere Operatoren mit minimaler Selektivität gibt. Auf dieser Teilmenge wird dann das „Auswahl nach Priorität“-Verfahren angewandt.

Die Nachteile des Ticket-Mechanismus können 1:1 von der Warteschlangen-Methode übernommen werden, da diese alle aufgrund der verteilten Operatoren entstehen. Der Nachteil der lokalen Entscheidungsfindung muss aber relativiert werden. Man kann problemlos Fälle konstruieren, bei denen unterschiedliche Selektivitäten für einen Operator durchaus sinnvoll sind. So können z.B. Tupel, die eine Selektionsbedingung erfüllen, räumlich eng im Netz verteilt sind. In diesem Bereich werden die Peers eine hohe Selektivität für diese Selektion erlernen, was diesen Operator dort unattraktiv macht.

Für die unären Planoperatoren Selektion und Projektion kann die Selektivität durch den Ticket-Mechanismus direkt berechnet werden, da sowohl die Anzahl der Eingangs- und Ausgangstupel für ein Nachrichtenpaket bekannt sind. Problematisch wird die Bestimmung der Selektivität für Join-Operatoren, da sich die Anzahl der Eingangstupel aus dem Produkt der Kardinalitäten beider Partnerrelationen ergibt. Durch die Verteilung der Operatoren sind diese Kardinaltiäten aber nicht bekannt. Für Join-Operatoren kann die Selektivität bestenfalls geschätzt werden.

Suche nach dem „nächsten“ Join. Wie bereits in Punkt 3.3 kurz erwähnt, setzt die Abarbeitung des Join-Operators eine Neuverteilung der Tupel voraus. Stehen mehrere Join-Operatoren zur Auswahl, werden sich die Abstände zwischen dem aktuellen Peer und dem Ziel-Peer für die Neuverteilung der verschiedenen Joins in der Regel unterscheiden.

Die Idee ist also, den Join-Operator auszuwählen, bei dem das Tupel bei der Neuverteilung den kürzesten Weg gehen muss. Dadurch kann die Anzahl der benötigten Nach-

richten minimiert werden. Auch der Sonderfall, dass der aktuelle Peer und der Ziel-Peer identisch sind wird dadurch ausgenutzt. Dazu müssen im Vorfeld die Abstände vom aktuellen Peer zu den resultierenden Zielpunkten aller Joins berechnet und miteinander verglichen werden. Enthält ein Nachrichtenpaket mehrere Tupel, wird der mittlere Abstand als Vergleichswert herangezogen. Zusätzliche Informationen werden nicht benötigt, der Mehraufwand ergibt sich allein aus den Abstandsberechnungen.

Offen bleibt die Frage, wann nach dem nächsten Join gesucht werden soll. Intuitiv bietet sich diese Berechnung an, wenn innerhalb der Todo-Liste nur Join-Operatoren ein gesetztes Ready-Bit besitzen. Es handelt sich also genau genommen nicht um eine eigene Routing-Strategie, sondern um eine Hilfsfunktion für andere Strategien.

4.3.3 Peer-Routing

Das Peer-Routing ist kein Bestandteil der ursprünglichen Eddy-Varianten. Selbst bei den verteilten Eddies reicht das Operator-Routing aus, da jeder Operator quasi fest mit einem Peer verknüpft ist. Erst durch die nahezu beliebige Verteilung der Operatoren stellt sich die Frage nach einer sinnvollen Auswahl der Peers.

Auch beim P2P-Eddy ist das Peer-Routing nicht völlig unabhängig von der Wahl des nächsten Operators. Durch den als *Symmetric Hash Join* implementierten Join-Operator, der auf einer Neuverteilung der Tupel basiert, wird der Ziel-Peer direkt vom Join-Operator vorgegeben.

Die Auswahl des nächsten Peers ist nur bei den unären Operatoren Selektion und Projektion losgelöst vom Operator-Routing. Damit eignen sich diese Operatoren besonders gut für eine gezielte Lastverteilung. Das Delegieren von Last auf andere Peers ist besonders auch bei aufwendigen Operatoren wie Sortieren und Aggregation sinnvoll.

Im ersten Schritt muss beim Peer-Routing entschieden werden, ob ein Nachrichtenpaket überhaupt weitergeleitet werden soll. Um die Entscheidung dynamisch treffen zu können, wird ein Parameter benötigt, anhand dessen die Aussage getroffen werden kann, ob eine Weiterleitung sinnvoll ist oder nicht. Hierfür bietet sich die aktuelle Auslastung des Peers an. Überschreitet diese einen gewissen Wert, wird das Nachrichtenpaket weitergeleitet.

Wie die Auslastung bestimmt oder abgeschätzt wird, ist eine andere Frage. In der bisherigen Umsetzung wird die Auslastung eines Peers durch zwei verschiedene Parameter abgeschätzt. Zum einen ist dies die Summe aller aktuellen Warteschlangenlängen auf dem Peer und zum anderen die Anzahl aller eingegangen Nachrichten innerhalb eines festgelegten Zeitintervalls. Eine andere Möglichkeit ist das Auslesen der konkrete Systeminformationen (Prozessorlast, Speicherbelegung, ...).

Wird die Entscheidung gefällt, ein Nachrichtenpaket auf demselben Peer weiterzuverarbeiten, ist das Peer-Routing abgeschlossen. Im anderen Fall muss nun festgelegt werden, wohin das Packet tatsächlich geschickt werden soll. Da jeder Peer nur seine direkten Nachbarn und deren Bereiche des CAN-Koordinatenraums kennt, ist das Weiterleiten zu einen direkten Nachbarn am sinnvollsten. Im letzten Schritt muss nur noch bestimmt werden, welcher Nachbar als Routing-Ziel ausgewählt wird.

Gleicher Peer. Die Tupel werden nur noch bei einer Neuverteilung aufgrund von Joins verschickt. Selektion und Projektion werden immer auf dem aktuellen Peer ausgeführt. Dadurch erhöht sich zwar das Risiko einer Überlastung des Peers, reduziert aber im Mittel den benötigten Netzwerk-Traffic.

Zufällige Auswahl. Wie schon beim Operator-Routing kann die Entscheidung durch Zufall getroffen werden. Dabei kann es allerdings passieren, dass ein Nachbar bevorzugt das Ziel einer Weiterleitung und dadurch möglicherweise überlastet wird.

Zyklische Auswahl. Alle Nachbarn werden nacheinander als Ziel-Peer ausgewählt. Dies garantiert zumindest eine faire Verteilung der Nachrichtenpakete und damit eine ausgewogene Lastverteilung.

Suche nach der schnellsten Verbindung. Beim Nachrichtenaustausch zwischen den Peers wird die Zeit berechnet, wie lange eine Nachricht von einem Peer zu dessen Nachbarn gebraucht hat. Jeder Peer weiß somit, wie schnell er seine Nachbarn erreichen kann. Ein Nachrichtenpaket wird immer zu dem Nachbarn geschickt, bei dem die Übertragungsdauer am geringsten ist.

Gerade in weitverteilten Netzen (Internet) können sich die Anbindungen eines Peers zu seinen Nachbarn deutlich unterscheiden. Diese Strategie ist natürlich nur dann wirklich aussagekräftig, wenn im gesamten Netz ein globale Zeit garantiert werden kann. Anderfalls ist die Berechnung der Übertragungsdauer einer Nachricht nicht korrekt.

Auswahl nach Auslastung der Nachbarn. Ziel soll es sein, die Auslastung der Nachbarn abzuschätzen. Das Verfahren arbeitet ähnlich wie das für die Bestimmung der Auslastung eines Peers. Jeder Peer merkt sich für jeden seiner Nachbarn, wie viele direkte Nachrichten er in einem bestimmten Zeitintervall von diesem bekommen hat.

Die Annahme ist also, dass ein Nachbar immer dann höher ausgelastet ist als ein anderer, wenn er im gleichen Zeitraum mehr Nachrichten geschickt hat. Diese Annahme berücksichtigt natürlich nur die Auslastung eines Peers bzgl. der Anfrageverarbeitung. Natürlich kann ein Peer auch aus anderen Gründen unter hoher Last laufen. Solche Fälle können durch dieses Verfahren nicht abgefangen werden.

Richtig gute Aussagen über die Auslastung der Nachbarn sind nur dann möglich, wenn diese Information direkt periodisch ausgetauscht werden würde. Dazu wären allerdings zusätzliche Nachrichen notwendig, was als Vorbedingung aber vermieden werden sollte. Darüberhinaus müssten der Informationsaustausch in kurzen Zeitintervallen erfolgen, da die Auslastung typischerweise eine stark schwankende Größe ist.

Durch den Test auf Auslastung und der freien Wahl der Strategie für das Peer-Routing, kann es unter Umständen zu widersprüchlichen Entscheidungen kommen. Es entsteht beispielsweise ein Konflikt, wenn die Auslastung eines Peers den Grenzwert übersteigt, aber die Strategie „Gleicher Peer“ angewendet werden soll. Da der Test auf Auslastung abschaltbar ist und alle Strategien explizit getestet werden können, bildet der Entscheidungsbaum aus 4.7 die Basis für das Peer Routing.

Die Abarbeitung des Entscheidungsbaums und damit das Peer-Routing wird an zwei

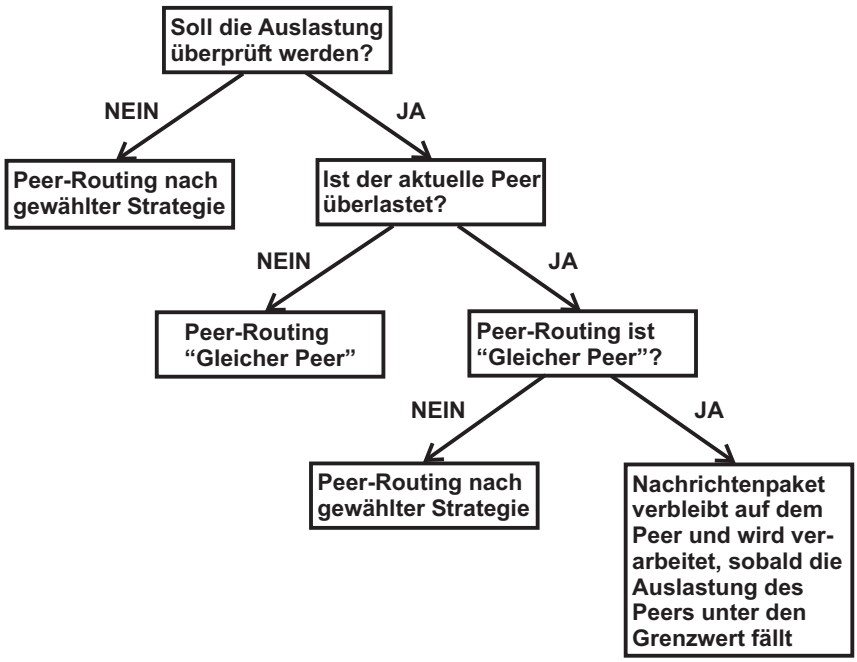

Abbildung 4.7: Entscheidungsbaum für das Peer-Routing

Zweitpunkten ausgeführt. So wird nach der Verarbeitung eines Nachrichtenpaketes entschieden, wohin des Ergebnis geschickt werden soll. Aber auch bereits beim Eintreffen von Nachrichtenpaketen wird entschieden, ob ein Paket auf dem Peer ausgeführt oder weitergeschickt wird. Um zu vermeiden, dass ein Paket zu lange unverarbeitet durch das Netz geleitet wird, sind geeignete Gegenmaßnahmen notwendig.

4.3.4 Anmerkungen

Die vorgestellten Routing-Strategien müssen nicht zwangsläufig exklusiv eingesetzt werden, sondern können auch für jede Anfrage parallel laufen. Als Voraussetzung muss dann gelten, dass alle benötigten Parameter aktualisiert werden. Wird dieser Aufwand in Kauf genommen, kann zur Ausführungszeit einer Anfrage zwischen den Strategien gewechselt werden.

Diese zusätzliche Flexibilität führt automatisch zu der Frage, welche Routing-Strategie tatsächlich zum Einsatz kommt, falls dem Eddy-Operator mehrere zur Auswahl stehen. Die Auswahl könnte beispielsweise in Abhängigkeit der aktuell zur Verfügung stehenden Parameter des Peers geschehen. Sind für eine Strategie zu wenige Daten vorhanden, um ein aussagekräftiges Ergebnis zu erhalten, kann die Routing-Entscheidung durch ein anderes Verfahren abgenommen werden.

Im Normalfall bleibt das gesamte Routing dem Anwender gegenüber verborgen. Allerdings kann es sinnvoll sein, direkten oder indirekten Einfluss auf das Routing zu haben.

Durch eine direkte Vorgabe einer Strategie für das Operator- und Peer-Routing, kann z.B. deren Ergebnisse für die gleiche Anrage miteinander verglichen werden. Damit können Aussagen gemacht werden, welche Strategie in welchen Fällen am günstigsten ist.

Wird für beiden Teilbereiche des Routings jeweils nur eine Strategie verwendet, kann darüber hinaus der Aufwand minimal gehalten werden. Lediglich die Parameter für die jeweilige Routing-Strategie werden benötigt.

Als indirekten Einfluss auf die Auswahl der Routing-Strategie kann das Einbringen von Vorwissen des Anwenders bezeichnet werden. Dazu gehören z.B. globale Informationen über die Systemumgebung. Besteht das P2P-Netz aus leistungsschwachen Peers, die über schnelle Verbindungen kommunizieren, entscheidet sich der Eddy-Operator in der Regel für möglichst weite Verteilung der Anfrage im Netz. Sind stattdessen leistungsstarke Peers über ein langsames Netz verbunden, sollten möglichst viele Operatoren auf demselben Peer ausgeführt werden, um unnötigen Netzverkehr zu vermeiden.

Sind dem Anwender solche oder ähnliche Informationen bekannt, kann er dadurch das Routing in bestimmte Richtungen lenken.

Sowohl die Verarbeitung als auch das Routing der Nachrichtenpakete wurde so umgesetzt, dass sich prinzipiell beliebig viele Tupel zusammenfassen lassen. Damit lässt sich der benötigte Kommunikationsaufwand für die Verarbeitung einer Anfrage im Mittel erheblich mindern. Es muss lediglich die Bedingung eingehalten werden, dass alle Tupel eines Paketes gleichartig sind. Was unter gleichartigen Tupeln verstanden werden soll, wird in Abschnitt 5.3.1 erläutert. Die Anzahl der Tupel ist so mit abhängig von folgenden Punkten:

- *Verteilung der Basisrelationen*
  Die initialen Nachrichtenpakete können maximal so viele Tupel enthalten, wie es Tupel der zugehörigen Basisrelationen auf einem Peer gibt. Je verteilter eine Tabelle also ist, umso kleiner wird die maximale Größe der Pakete.

- *Ergebnisse der Planoperatoren*
  Alle Nachrichtenpakete werden getrennt voneinander verarbeitet. Für jedes Eingangsnachrichtenpaket wird in Falle von Ergebnistupeln genau ein Ergebnistupelpaket erzeugt. Ergebnisse zweier Nachrichtenpakete können nicht verschmolzen werden.

- *festgelegte Maximalgröße*
  Durch den Nutzer oder die Anwendung kann die maximale Größe der Nachrichtenpakete vorgegeben werden. Dadurch kann bei der Evaluierung der Einfluss der Paketgröße auf die Verarbeitung einer Anfrage untersucht werden.

Im Normalfall wird sich die Zahl der Tupel eines Nachrichtenpaketes im Laufe der Verarbeitung immer verringern. Lediglich durch Join-Operatoren können mehr Ergebnistupel als Eingangstupel erzeugt werden. Im Allgemeinen legt also die Verteilung die maximale verfügbare Größe der Pakete fest.

Kapitel 5

Implementierung der dynamischen Operatoren

5.1 Überblick

Bevor die einzelnen Klassen des P2P-Eddies genauer vorgestellt werden, sollen zunächst die wichtigsten Klassen in einer Art Übersicht präsentiert werden, um deren Zusammenspiel zu veranschaulichen. Ausgangspunkt ist die Abbildung 4.6 aus Abschnitt 4.2. Diese Abbildung zeigt schematisch alle Komponenten des P2P-Eddies.

In Abbildung 5.1 wurden die Komponenten um die Klassen erweitert, welche die Kompenenten umsetzen. Dadurch zeigt sich automatisch, welche Beziehungen zwischen den Klassen bestehen. Weggelassen wurden Klassen, die zwar notwendig sind, aber nicht direkt die Grundkonzepte des P2P-Eddies widerspiegeln.

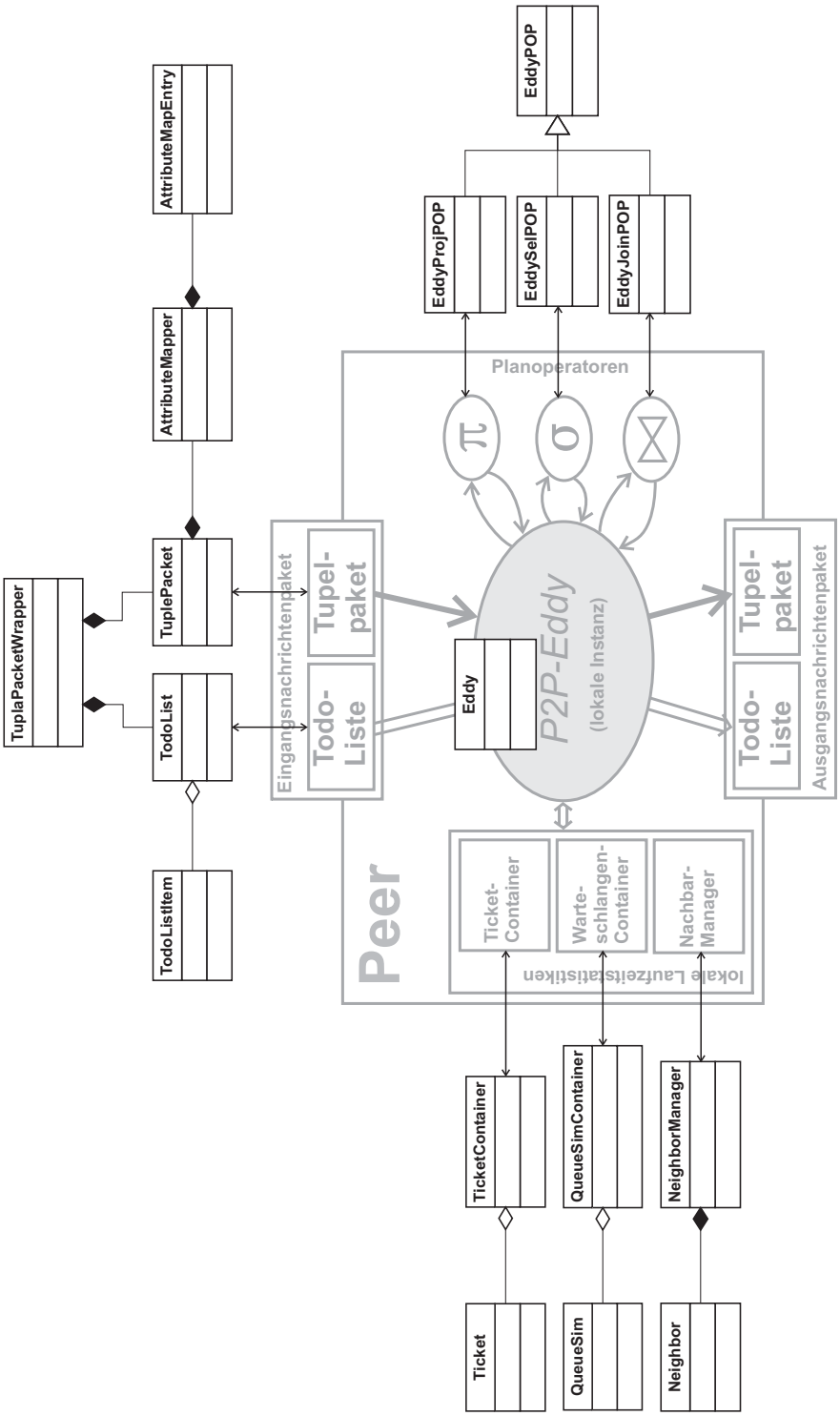

Abbildung 5.1: Übersicht über die wichtigsten Klassen des P2P-Eddies

5.2 Umsetzung der Todo-Listen Idee

5.2.1 Die Klasse `TodoListID`

Ein eindeutiger Identifier für jede Todo-Liste wird für deren Verteilung im Netz benötigt. Enthält die Todo-Liste keine Selektion, die einer Exact-Match-Anfrage auf das Primärschlüsselattribut entspricht, muss ein Broadcast der Listen durchgeführt werden. Jeder Peer muss jede Todo-Liste genau einmal verarbeiten und sollte sie genau einmal an seine Nachbarn verschicken. Dafür merkt sich jeder Peer in einem Vektor die TodoListIDs aller bereits verarbeiteten Todo-Listen.

Innerhalb einer Anfrage wird eine Todo-Liste eindeutig durch den Relationennamen bestimmt, für deren Tupel die Liste bestimmt ist. Um die parallele Verarbeitung mehrerer Anfrage zu gewährleisten, muss jede Anfrage mit Hilfe einer `QueryID` näher spezifiziert werden. Diese beiden Größen sind somit auch die Attribute der Klasse `TodoListID`.

Neben den Methoden zum Auslesen der Atribute, dient `equals` dazu, zwei Instanzen dieser Klasse zu vergleichen. Mit dieser Methode kann ein Peer überprüfen, ob eine Todo-Liste bereits für eine Verteilung verarbeitet wurde. Ist die List noch nicht bekannt, wird deren TodoListID im entsprechenden Vektor des Peers aufgenommen.

5.2.2 Die Klasse `TodoList`

Die Klasse `TodoList` stellt die eigentliche Umsetzung der Todo-Liste dar. Sie kapselt im Wesentlichen eine Instanz der Klasse `java.util.Vector` (`listItems`). Die Elemente des Vektors sind aber nicht direkt die Operatoren, sondern Objekte vom Typ `TodoListItem` (Siehe 5.2.3).

Die verschiedenen Methoden für das Einfügen, Entfernen und Auslesen von Listenelementen, das Auslesen der Anzahl aller Elemente und andere Methode, können direkt auf die Methoden der Klasse `Vector` abgebildet werden.

Wichtig für binäre Operatoren ist die Methode `joinTodoLists`, welche das Verschmelzen zweier Todo-Listen realisiert. Da die Reihenfolge innerhalb einer Liste egal ist, kann der Algorithmus wie in Abbildung 5.2 implementiert werden

joinTodoLists(*TodoList*)
1 for each *item* from *listItems*
2 if *item* is also element of *TodoList*
4 *TodoList.removeListItem(item)*
5 end if
6 end for
7 if *TodoList.getSize() > 0*
8 for each *item* from *TodoList*
9 *resultTodoList.insertListItem(item)*
10 end for
11 end if
12 return *resultTodoList*

Abbildung 5.2: Algorithmus für das Verschmelzen zweier Todo-Listen

Der binäre Operator, der die Verschmelzung der Todo-Listen hervorgerufen hat, wurde schon bei seiner Auswahl aus beiden Listen entfernt.

5.2.3 Die Klasse `TodoListItem`

Als Elemente der Klasse `TodoList`, enthält eine Instanz der Klasse `TodoListItem` den eigentlichen Planoperator. Diese zusätzliche Kapselung der Operatoren ist sinnvoll, da die Klasse `TodoListItem` noch weitere Attribute enthält, die für Operatoren nur im Umfeld der Todo-Liste gelten.

So wird hier z.B. das vorgstellte Ready-Bit umgesetzt, welches anzeigt, ob ein Operator ausgeführt werden darf oder nicht. Durch das Löschen abgearbeiteter Operatoren aus der Liste, kann auf ein Done-Bit, wie es bei den ursprünglichen Eddy-Varianten zum Einsatz kommt, verzichtet werden.

Ein weiteres Attribut ist die Operatorpriorität, die für die Strategie „Auswahl nach Priorität“ beim Operator-Routing benötigt wird. Dabei handelt es sich um einen einfachen Zahlenwert, der für die Operatortypen Selektion, Projektion und Join entsprechend gesetzt wird.

Für weitere Routing-Stratgien kann die Klasse `TodoListItem` einfach erweitert werden. Denkbar wäre z.B. eine zusätzliche Priorität, so dass auch Operatoren eines Typs unterschiedlich behandelt werden. Möglich ist auch eine vorher berechnete Selektivität, die sich während der Abarbeitung der Todo-Liste nicht ändert.

Die Methoden der Klasse beschränken sich auf das Auslesen und ggf. das Setzen ihrer Attribute.

5.2.4 Die Klasse `TodoListContainer`

Diese Klasse dient lediglich dazu, ein Objekt vom Typ `java.util.Vector` zu kapseln dessen Elemente vom Typ `TodoList` sind. Die Methoden zum Hinzufügen und Entfernen von den Listen können direkt auf die Methode der Vektor-Klasse abgebildet werden.

Verwendung findet der Container beim Broadcast der Todo-Listen. Anstatt jede Todo-Liste getrennt im gesamten Netz zu verteilen, werden sie gemeinsam von Peer zu Peer geschickt. Unabhängig von den benötigten Ursprungsrelationen, reduziert sich der Aufwand auf höchstens einen Broadcast pro Anfrage.

5.3 Tupelverwaltung

5.3.1 Die Klasse `TuplePacket`

Diese Klasse ermöglicht es, mehrere Tupel gleichzeitig zu verarbeiten. Hauptkomponente ist eine Instanz der Klasse `java.util.Vector`, welche Elemente vom Typ `Tuple` aufnehmen soll. Die verschiedenen Operationen zum Einfügen und Entfernen von Tupeln können direkt auf die entsprechenden Methoden der Vektorklasse abgebildet werden.

In einem Tupelpaket können aber keine beliebigen Tupel gemeinsam gesammelt werden. Da alle Tupel eines Paketes auf die gleiche Art und Weise verarbeitet werden sollen, müssen die Tupel folgenden Bedingungen genügen:

- bei der initialen Erzeugung des Paketes müssen die Tupel aus der gleichen Basisrelation stammen
- alle Tupel besitzen die gleiche Vergangenheit, d.h. sie haben im Laufe der Verarbeitung die gleichen Operatoren in gleicher Reihenfolge angelaufen
- würden alle Tupel eine eigene Todo-Liste besitzen, wären diese identisch

Diese Bedingungen garantieren, dass alle Tupel das gleiche Schema besitzen. Aus diesem Grund kann jedem Tupelpaket der notwendige Attribute-Mapper (siehe Abschnitt 5.4.3) zugeordnet werden.

Der letzte Bestandteil der Klasse `TuplePacket` ist ein Zähler (`hopCounter`). Dieser hält fest, wie oft in Folge ein Tupelpaket von Peer zu Peer geschickt wurde, ohne verarbeitet worden zu sein. Ziel dabei ist eine bessere Verteilung der Auslastung über das Netz bzw. die Vermeidung einer Überlastung von Peers. Damit ein Paket nicht zu lange unverarbeitet im Netz verbringt, *muss* es von einem Peer verarbeitet werden, sobald der Hop-Zähler einen vorher festgesetzten Maximalwert erreicht.

5.3.2 Die Klasse `TuplePacketWrapper`

Diese Hilfsklasse dient dazu ein Problem zu lösen, dass bei der Neuverteilung der Tupel entsteht. Da eine Neuverteilung einem temporären Einfügen entspricht, kann nur für `Tuple`-Objekte das Re-Hashing ausgeführt werden. Ohne geeignete Maßnahmen würde dabei die nötige Verbindung der Tupel mit ihrer Todo-Liste verloren gehen. Weiterhin wäre es so nicht möglich, mehrere Tupel gleichzeitig neu zu verteilen.

Aus diesem Grund kapselt die Klasse `TuplePacketWrapper` ein Objekt vom Typ `TuplePacket` und vom Typ `TodoList`. Diese Kapselung ist natürlich nur ein Zwischenschritt, da Instanzen dieser Klasse nicht neuverteilt werden können. Um das Re-Hashing zu ermöglichen, wird der Wrapper in den Datenvektor eines Hilfstupel eingefügt.

Mit diesem Verfahren können nun mehrere Tupel auf einmal neuverteilt werden, inkl. der zugehörigen Todo-Liste.

5.3.3 Die Klasse `TuplePacketContainer`

Mit Hilfe der Klasse `TuplePacketContainer` können mehrere Tupelpakete oder Tupelpaket-Wrapper in einem Vektor gesammelt und so gemeinsam behandelt werden. Eine Vermischung der beiden Objektypen ist allerdings nicht zulässig.

Mehr Methoden als die üblichen für das Einfügen und Entfernen oder die Bestimmung der Anzahl der Elemente werden nicht benötigt.

5.4 Weitere Hilfsklassen

5.4.1 Die Klasse `QueryID`

Um mehrere Anfragen parallel verarbeiten zu können, müssen sie eindeutig unterscheidbar gemacht werden. Durch zwei Komponenten wird die Eindeutigkeit erreicht:

1. *Zeitstempel*
   Zeitpunkt, an dem eine Anfrage initiiert wurde.

2. *Peer-Identifier*
   Peer, an den die Anfrage gestellt wurde; notwendig, da in einer verteilten Umgebung eine globale Zeit nicht immer zur Verfügung steht; ohne globale Zeit kann die Eindeutigkeit des Zeitstempels nicht garantiert werden.

Verwendung findet der Anfrage-Identifier vor allem als Komponente der Klasse `TodoListID`, welche für den korrekten Broadcast aller Todo-Listen zuständig ist (siehe Abschnitt 5.2.1).

5.4.2 Die Klasse `AID`

`AID` steht für Attribute-Identifier und dient dazu, Tupelattribute anzusprechen. Benötigt wird dieser für die Planoperatoren, um Selektionsattribut, die beiden Attribute eines Joins und die zu projezierenden Attribute einer Projektion festzulegen.

Innerhalb einer Datenbank wird ein Attribut eindeutig bestimmt durch den Identifier der Basisrelation und der Position des Attributes im Schema der Relation. Beide Größen sind damit auch die beiden Komponenten der Klasse `AID`.

5.4.3 Die Klassen `AttributeMapper` und `AttributeMapEntry`

Wie eben beschrieben, werden Attribute unter anderem über die Position im Datenvektor angesprochen. Durch Planoperatoren wie Projektion oder Joins kann sich das Schema der Tupel verändern und somit auch die Position der Attribute. Bei einer statischen Anfrageverarbeitung, bei dem die Operatorreihenfolge für alle Tupel gleich ist, kann die Attributposition bereits im Vorfeld korrekt gesetzt werden.

Ist die Operatorreihenfolge nicht bekannt, wie beim P2P-Eddy, müssen die Attributpositionen im Laufe der Verarbeitung immer wieder aktualisiert werden. Da nahezu jeder Planoperator in den Todo-Listen auf eine oder mehrere Instanzen der Klasse `AID` angewiesen ist, empfiehlt sich die Aktualisierung in einer gemeinsam genutzten Komponente. Andernfalls müssten alle restlichen Operatoren der Todo-Liste aktualisiert werden, sobald ein schemaverändernder Operator ausgeführt wurde. Diese Komponente ist der Attribute-Mapper, über den die Planoperatoren auf die Attribute im Datenvektor der Tupel zugreifen.

Die Idee des Attribute-Mappers ist es, die Position der Attribute im Datenvektor der Tupel einer Basisrelation auf die Position im aktuellen Datenvektor abzubilden. Die Attribute-Identifier der Planoperatoren können deswegen einfach auf die Position der Attribute innerhalb der Basistupel gesetzt werden. Muss ein Operator auf ein Attribut zugreifen, ruft er zunächst den Attribute-Mapper des Tupelpaketes auf und erhält von diesem die aktuelle Position des Attributes im Datenvektor.

Die eigentliche Abbildung übernehmen Instanzen der Klasse `AttributeMapEntry`. Neben dem Relationennamen gehört zu dieser Klasse ein Array, dessen Größe der Anzahl der Attribute in der Basisrelation entspricht. Die Position im Array entspricht dabei

der Position eines Attributes in der Basisrelation. Der jeweilige Inhalt steht für dessen Position im aktuellen Datenvektor.

Werden Tupel aufgrund binärer Operatoren miteinander verknüpft, vergrößert sich dadurch der Attribute-Mapper für die Ergebnistupel. Die Anzahl der Mapper-Einträge entspricht somit immer der Anzahl der enthaltenen Basisrelationen im Datenvektor der Tupel.

Abbildung 5.3 zeigt einen initialisierten Attribute-Mapper für die Basisrelation R mit vier Attributen. Zu diesem Zeitpunkt sind die Attributpositionen in den Basistupeln mit

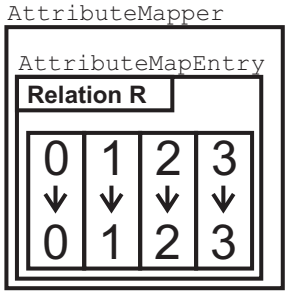

Abbildung 5.3: Beispiel für eine neu erzeugten Attribute-Mapper

denen im aktuelle Datenvektor noch identisch, so dass die Positionen auf sich selbst abgebildet werden.

Jeder schemaverändernde Planoperator muss nun gemäß seiner Eigenschaften den Attribute-Mapper so verändern, dass nachfolgende Operatoren auf die richtigen Attribute zugreifen können. Beispiele dafür finden sich bei den entsprechenden Operatoren.

5.5 Planoperatoren

5.5.1 Die Klasse `EddyPOP`

Die Planoperatoren (kurz: POPs) stellen die Implementierung der logischen Operatoren dar. Die abstrakte Klasse `EddyPOP` umfasst alle Attribute und Methoden, die allen Operatoren eigen ist. Von dieser Klasse können demnach keine Instanz erzeugt werden. Instantiierbar sind die von `EddyPOP` abgeleiteten Klassen, die umgesetzten Operationen Projektion, Selektion und Join (siehe Abbildung 5.4)

Um die Operatoren innerhalb einer Anfrage unterscheidbar zu machen, wird ein eindeutiger Identifier (`operatorID`) benötigt. Zum Auslesen dieses Attributes dient die Methode `getOperatorID`.

Gemäß den Voraussetzungen aus Abschnitt 4.1 muss die Anfrage als Baum übergeben werden. Um die Struktur aus den Operatoren zu konstruieren, wird jeder Operator um Attribute erweitert, welche die Kind-Objekte des Operators im Baum repräsentieren. Da der Anfragebaum maximal der Ordnung 2 ist, sind zwei solche Attribute nötig (`leftChild`, `rightChild`). Für unäre Operatoren wird eines der Attribute auf `null` gesetzt.

Die Verarbeitung der Tupel durch einen Operator soll über zwei verschieden Wege realisiert werden. Im ersten Fall werden alle Eingangstupel am Stück abgearbeitet. Dazu wird lediglich eine Methode (`processEddyPOP`) benötigt, welche die übergebenen Tupel verarbeitet und mögliche Ergebnistupel wieder zurückgibt.

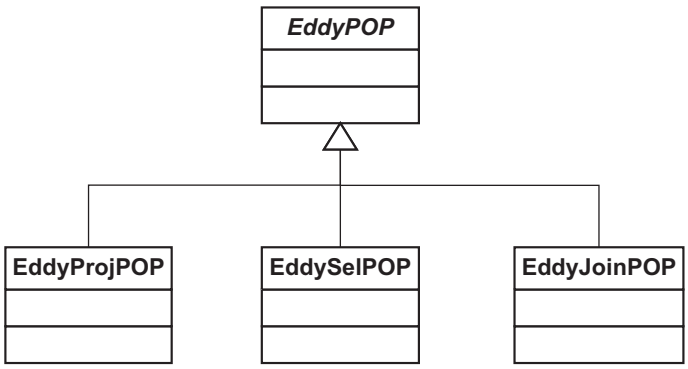

Abbildung 5.4: Vererbungshierarchie der Klassen der Planoperatoren

Bei der zweiten Variante werden mögliche Ergebnistupel schrittweise angefordert. Dadurch können bereits die Zwischenergebnisse eines Operators weiterverabeitet oder auch weitergeschickt werden (vgl. Pipelining). Hierfür wird die Klasse um ein Attribut `tuplePacket` erweitert, welches zunächst alle Eingangstupel aufnimmt (über die Methode `setTuplePacket`). Die eigentliche Abarbeitung der Tupel übernimmt die Methode `getNextOutputTuple`. Diese arbeitet die Eingangstupel nur soweit ab, bis ein erstes Ergebnistupel entsteht. Für die komplette Verarbeitung muss `getNextOutputTuple` solange aufgerufen werden, bis alle Eingangstupel abgearbeitet wurden.

Die beiden Methoden für die Abarbeitung der Tupel müssen als „abstrakt“ definiert werden, da deren Implementierung abhängig vom konkreten Operator ist.

5.5.2 Die Klasse `EddyProjPOP`

Der Planoperator der Projektion wird durch die Klasse `EddyProjPOP` implementiert. Die beispielhafte Darstellung einer Projektion im Anfragbaum (Abbildung 5.5) liefert alle spezifischen Informationen.

Abbildung 5.5: Darstellung einer Projektion im Anfragebaum

- Liste alle projezierten Attribute inkl. der zugehörigen Relationen (Vektor mit Elementen vom Typ `AID`)

Die Projektion gehört zur der Sorte Operatoren, die in der Regel das Schema ihrer Eingangstupel verändern. Deshalb wird eine Methode benötigt die den entsprechenden Attribute-Mapper der Tupel modifiziert. Die verbliebenen Attribute müssen dabei auf die Position abgebildet werden, an derer sich die zugehörigen AIDs im Vektor der Projektion befinden. Alle herausprojezierten Attribute werden auf eine Konstante abgebildet, die kenntlich macht, dass diese Tupel nicht mehr gültig sind.

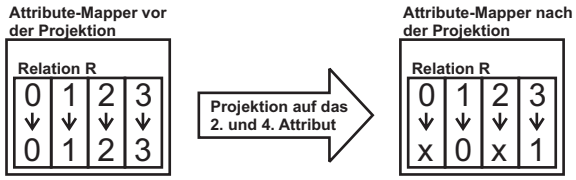

Abbildung 5.6: Aktualisierung eines Attribute-Mappers nach einer Projektion

5.5.3 Die Klasse `EddySelPOP`

Der zweite unäre Planoperator ist die Selektion. Im Vergleich zur Projektion, umfasst die Selektion wesentlich mehr beschreibende Parameter.

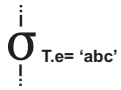

Abbildung 5.7: Darstellung einer Selektion im Anfragebaum

- Selektionsattribut inkl. zugehöriger Relation (repräsentiert durch ein Attribut vom Typ `AID`)
- Vergleichsoperation
- Vergleichswert
- Typ des Vergleichswertes (`String`, `Double`, `Integer`)

Natürlich sind auch Selektionen möglich, bei denen der Wertebereich des Selektionsattributes beidseitig begrenzt ist (z.B. `20000<MatNr< 30000`). Solche Selektionen müssen in Teilselektionen umgewandelt werden, wobei jede für eine Teilbedingung verantwortlich ist.

Da eine Selektion die Schemas ihrer Ergebnistupel nicht verändert, bedarf es keiner Anpassung des Attribute-Mappers.

5.5.4 Die Klasse `EddyJoinPOP`

Als binärer Operator muss der Join, im Gegensatz zur Projektion oder Selektion, mehrere Tupel miteinander in Beziehung bringen. Durch diese notwendige Koordination ist der Join zum einen weniger flexibel und zum anderen deutlich komplexer als die unären Operatoren.

Implementiert wurde der Join-Operator als SHJ (*Symmetric Hash Join*). Die Idee des SHJ ist es, potentielle Join-Partner auf den selben Peer zu bringen und dort einen lokalen Join durchzuführen. Damit ist der Join ein zweistufiger Prozess bestehend aus der Neuverteilung der Tupel und dem lokalen Join. Die charakteristischen Kenngrößen des Join-Operators sind:

- die beiden Join-Attribute inkl. der zugehörigen Relationen (repräsentiert durch jeweils ein AID: `leftAID`, `rightAID`)

Abbildung 5.8: Darstellung einer Projektion im Anfragebaum

- eindeutiger `namespace` (notwendig für die Neuverteilung)
- alle Eingangstupel des Join-Operators müssen aus ihrer Todo-Liste entnehmen können, aus welchem der beiden Teilbäume sie zum Join kommen;
  diese Information wird im Attribut `joinPartnerSide` hinterlegt und beim Erzeugen der Todo-Listen entsprechend gesetzt

Obwohl für die Join-Bedingung verschiedene Vergleichsoperationen möglich sind, wird hier nur auf Gleichheit geprüft (Equi-Join). Diese Einschränkung wird durch die Neuverteilung der Tupel bedingt.

Die Neuverteilung oder auch das Re-Hashing von Tupeln ist ein temporäres Einfügen von Kopien der Originaltupel. Da es sich bei den Tupeln der Nachrichtenpakete bereits um Kopien handelt, können diese direkt verwendet werden. Obwohl das Re-Hashing durch den Eddy-Operator durchgeführt wird (nur dieser kann Nachrichten verschicken), soll der prinzipielle Ablauf dennoch hier vorgestellt werden.

Um sicherzustellen, dass potentielle Join-Partner nach der Neuverteilung auf einem Peer landen, muss für alle Tupel der *key*-Wert für die Basisoperationen des CANs geeignet gewählt werden. Um dies zu erreichen, setzt sich hier der *key* aus dem festgelegten *namespace* des Join-Operators und dem Wert des Join-Attributes eines jeden Tupel zusammen. Die Neuverteilung anhand des Attributwertes hat den Nachteil, dass lediglich die Gleichheit als Join-Bedingung genutzt werden kann.

Obwohl sich in einem `TuplePacket` immer nur gleichartige Tupel befinden dürfen, können die Tupel dennoch unterschiedliche Werte für das Join-Attribut besitzen. Enthält ein Paket also mehr als ein Tupel, werden diese in der Regel auf unterschiedliche Peers verteilt.

Das Re-Hashing der Tupel wurde in zwei Varianten implementiert. Die trivialste Lösung ist die Neuverteilung jedes einzelnen Tupels aus dem Paket. Dadurch werden aber immer genau so viele Nachrichten erzeugt, wie es Tupel gibt. In Bezug auf die Kommunikationskosten ist diese Art der Neuverteilung somit ungeeignet.

Bei der zweiten Variante wird vor der eigentlichen Neuverteilung ein Sortieralgorithmus auf das `TuplePacket` angewendet. Dieser Algorithmus (Abbildung 5.9) packt alle Tupel mit gleichem Wert für das Join-Attribut in neue Pakete ein. Für jedes Paket kann nun garantiert werden, dass dessen Tupel den gleichen Ziel-Peer für das Re-Hashing besitzen.

Abbildung 5.10 veranschaulicht die Arbeitsweise des Sortieralgorithmus. In diesem Beispiel kann die Zahl der benötigten Re-Hash-Nachrichten von sieben auf drei reduziert werden.

Im Mittel kann so die Anzahl der benötigten Nachrichten minimiert werden. Im Worst-Case enthält das Originalpaket nur Tupel mit unterschiedlichen Attributwerten, falls z.B.

Die Position des Join-Attributes im Datenvektor wird ausserhalb der Methode bestimmt und als Parameter `position` übergeben. Der Test der Join-Bedingung in Zeile 10 garantiert, dass beide Attributwerte identisch sind. Für diesen Test ist der Parameter `EddyJoinPOP` notwendig, da diese Klasse die benötigte Methode enthält.

buildReHashTuplePackets(*EddyJoinPOP, TuplePacket, position*)

```
1  originalAttirbuteMapper = TuplePacket.getAttributeMapper
2  for each tuple from TuplePacket
4      currentValue = tuple.getData(position)
5      inserted = false
6      for each entry from resultContainer
7          firstTuple = entry.getFirstTuple()
8          firstTupleValue = firstTuple.getData(position)
9          resultTodoList.insertListItem(item)
10         if both values fulfil the join condition
11             inserted = true
12             entry.insertTuple(tuple)
13         end if
14     end for
15     if inserted == false
16         TuplePacket newTuplePacket = new TuplePacket()
17         newTuplePacket.setAttributeMapper(originalAttributeMapper)
18         newTuplePacket.insertTuple(tuple)
19         resultContainer.insertObject(newTuplePacket)
20     end if
21 end for
22 return resultContainer
```

Abbildung 5.9: Algorithmus für die Vorsortierung von Tupel vor einem Re-Hashing

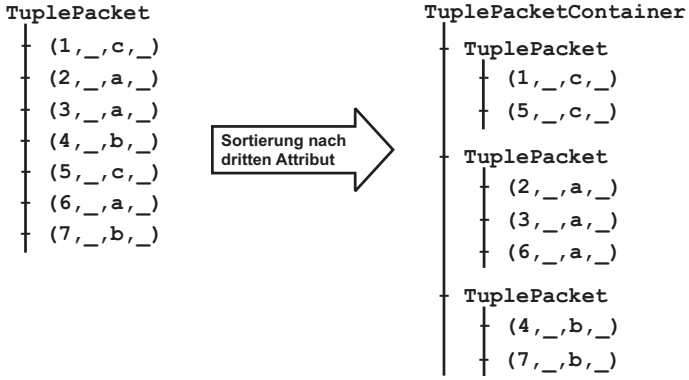

Abbildung 5.10: Beispiel für die Vorsortierung der Tupel für eine Neuverteilung

das Join-Attribut der Primärschlüssel ist. In solchen Fällen degeneriert das Verfahren zur Neuverteilung jedes einzelnen Tupels.

Um noch mehr Nachrichten einzusparen, wird vor jeder Neuverteilung überprüft, ob Ziel-Peer und aktueller Peer identisch sind. Ist dies der Fall, kann die weitere Abarbeitung sofort lokal fortgesetzt werden.

Systembedingt können im CAN nur Tupel eingefügt und somit auch nur Tupel neu verteilt werden. Um Tupelpakete inkl. Attribute-Mapper und Todo-Liste zu verteilen, müssen sämtliche zusammengehörigen Komponenten in eine Instanz der Klasse `TuplePacketWrapper` gesteckt werden. Diese wird Element des Datenvektors eines Hilfstupels, welches nun neu verteilt werden kann. Der vollständige Algorithmus für das Re-Hashing wird in Abbildung 5.11 gezeigt. Erreicht ein solches Hilsftupel seinen

reHashTuplePackets(*EddyJoinPOP, EddyProcessTodoListMessage*)
1 *todoList = EddyProcessTodoListMessage.getTodoList()*
2 *position* <- use AttributeMapper to get the position of the join attribute
3 *container = buildReHashTuplePackets(EddyJoinPOP, TuplePacket, position*
4 for each *tuplePacket* from *container*
5 *firstTuple = tupelPacket.getFirstTuple()*
6 *value* <- firstTuple.getData(position)
7 *namespace = EddyJoinPOP.getNamespace()*
8 *currentWrapper = new TuplePacketWrapper(tuplePacket, todoList)*
9 *reHashTuple = new Tuple(namespace,* `DONT_CARE_PRIMKEY`*, currentWrapper)*
10 *reHashTuple.setLifeTime(*`LIFETIME`*)*
11 *targetPoint* <- *lookup((namespace, value))*
12 send *reHashTuple* to peer which administrates the *targetPoint*
13 end for

Abbildung 5.11: Algorithmus für das Re-Hashing von Tupeln

Ziel-Peer, wird der lokale Join ausgeführt.

Der lokale Join wird mittels einer Nested-Loop-Implementierung durchgeführt. Durch die Dynamik der Anfrageverarbeitung, können sich die Schemas der möglichen Partnertupel, in Abhängigkeit ihrer Vorgeschichte unterscheiden. Daraus folgt, dass auch die Ergebnistupel nicht die gleichen Schemas besitzen müssen. Da sich aber in einem Tupelpaket nur gleichartige Tupel befinden dürfen, muss eine Vermischung ungleichartiger Tupel abgefangen werden.

Ähnlich wie beim Re-Hashen, können auch hier alle Ergebnistupel gesondert betrachtet werden. Jedes Ergebnistupel wird dabei in ein eigenes Tupelpaket gesteckt und mit dem entsprechenden Attribute-Mapper und entsprechender Todo-Liste versehen. Dies bedeutet also, dass sich nach einem Join nur noch ein Tupel in jedem Tupelpaket befindet, was wiederum den Netzverkehr unnötig ansteigen lassen würde.

Um dies zu umgehen, werden gleichartige Ergebnistupel in einem Tupelpaket gesammelt. Ob ein Tupel in ein bereits existierendes Paket gepackt werden darf, entscheiden folgende drei Bedingungen, die alle erfüllt sein müssen:

1. Die Attribute-Mapper des Ergebnistupels und des Tupelpaketes müssen identisch sein
2. Die Todo-Listen des Ergebnistupels und des Tupelpaketes müssen identisch sein
3. Die Anzahl der Attribute im Datenvektor des Ergebnistupels muss gleich der Anzahl der Attribute im Datenvektor der Tupel im Tupelpaket sein

Sobald eine der drei Bedingungen nicht erfüllt ist, wird ein neues Tupelpaket angelegt. Abbildung 5.12 zeigt den Algorithmus für das Erzeugen des Rückgabewertes eines lokalen Joins. Ein Beispiel für die Vorteile des gezielten Einfügens der Ergebnistupel ist in Abbildung 5.13 zu finden. In diesem Beipiel sind alle drei Ergebnistupel gleichartig, für die nachfolgenden Verarbeitungsschritte wird also nur ein Nachrichtenpaket benötigt anstatt drei.

Durch den binären Charakter des Join-Operators müssen sowohl die Todo-Listen

```
insertResultTupleIntoContainer(TuplePacketContainer, TuplePacketWrapper)
1  inputTuplePacket.TuplePacketWrapper.getTuplePacket()
2  inputTuple = inputTuplePacket.getFirstTuple()
4  inputTupleData = inputTuple.getAllData()
5  inputTodoList = TuplePacketWrapper.getTodoList()
6  inputAtributeMapper = inputTuplePacket.getAttributeMapper()
7  found = false
8  for each wrapper from TuplePacketContainer
9      partnerTuplePacket = wrapper.getTuplePacket()
10     partnerTuple = partnerTuplePacket.getFirstTuple()
11     partnerTupleData = partnerTuple.getAllData()
12     partnerTodoList = wrapper.getTodoList()
13     partnerAttributeMapper = partnerTuplePacket.getAttributeMapper()
14     if (inputAttributeMapper == partnerAtributeMapper AND
15         inputTodoList == partnerTodoList AND
16         inputTuple.getSize() == partnerTuple.getSize())
17             found = true
18             partnerTuplePacket.insertTuple(inputTuple)
19     end if
20 end for
21 if found == false
22     TuplePacketContainer.insertObject(TuplePacketWrapper)
23 end if
```

Abbildung 5.12: Algorithmus für das Einfügen eines Ergebnistupels eines Joins in den Rückgabe-Container

als auch die Attribute-Mapper der ursprünglichen Teile eines Ergbnistupels verschmolzen werden. Diese Vorgänge waren bereits Schwerpunkte in den Abschnitten 5.2.2 bzw 5.4.3.

Neben der Verschmelzung der Attribute-Mapper, muss der neue Attribute-Mapper noch aktualisiert werden. Da die Konvention eingehalten wird, dass die Attribute des Tupels aus dem rechten Teilbaum des Join-Operators an die Attribute des Tupels aus dem

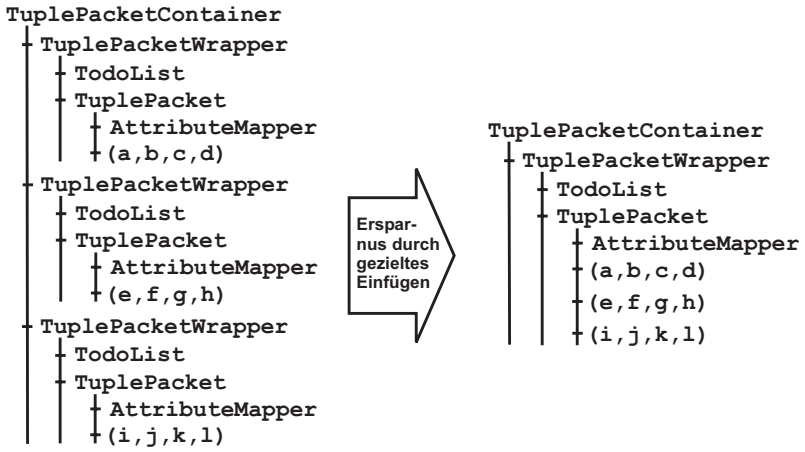

Abbildung 5.13: Vergleich der beiden Varianten für die Ergebnisse eines lokalen Joins

linken Teilbaum gehängt werden, müssen auch die beiden Attribute-Mapper in dieser Reihenfolge miteinander verbunden werden.

In diesem Fall können die Einträge des „linken“ Attribute-Mappers unangetastet bleiben. Auf alle Einträge des „rechten“ Attribute-Mappers muss die Anzahl der Attribute des „linken“ Tupels addiert werden, da sich die Position der Attribute des „rechten“ Tupels genau um diese Anzahl verschoben hat. Abbildung 5.14 zeigt die Verschmelzung und Aktualisierung zweier initialer Attribute-Mapper durch einen Join.

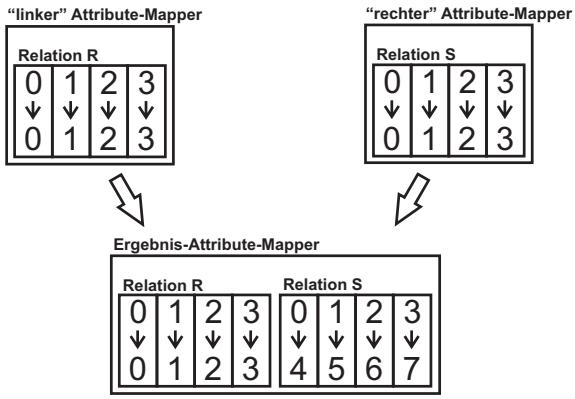

Abbildung 5.14: Erzeugung eines neuen Attribute-Mappers für die Ergebnistupel eines Joins

5.6 Eddy-Operator

Die Klasse `Eddy` ist das Herzstück der P2P-Eddy-Implementierung. Die Aufgaben und Arbeitsweise des Eddy-Operators lassen sich als Graph, wie in Abbildung 5.15 gezeigt, darstellen.

Erzeugung der Todo-Listen. Diese Teilaufgabe umfasst die Umsetzung der Anfrage in Baumdarstellung in die Todo-Listen für die Tupel der Basisrelationen. Die Abarbeitung findet vollständig auf dem Peer statt, an den die Anfrage gestellt wurde.

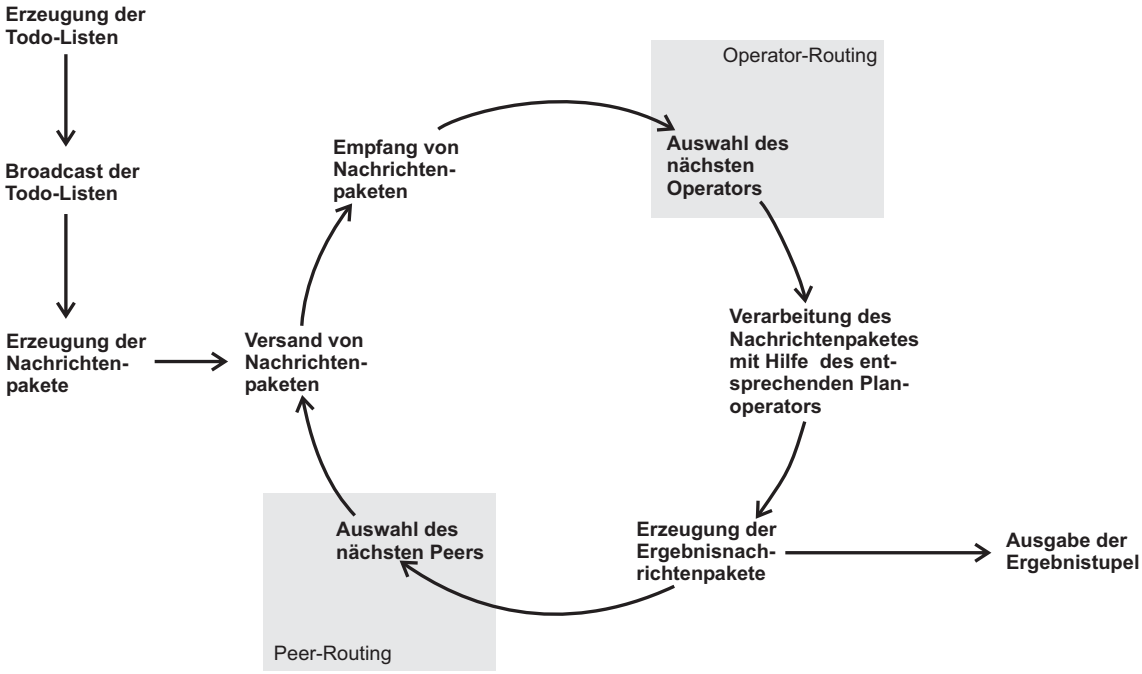

Abbildung 5.15: Überblick über die Aufgaben des Eddy-Operators

Alle Operatoren des Anfragebaumes, die auf die Tupel einer Basisrelation angewendet werden müssen, liegen auf dem Pfad zwischen Relation und Wurzeloperator. Durch binäre Operatoren (Joins) kann es allerdings vorkommen, dass auf diesem Pfad auch Operatoren liegen, die nicht für die Tupel gelten. Diese Operatoren greifen nicht auf Attribute der aktuellen Basisrelation zu und dürfen somit nicht in der entsprechenden Todo-Liste eingetragen werden.

Algorithmisch lässt sich diese Aufgabe durch eine Tiefensuche durch den Anfragebaum, beginnend beim Wurzeloperator, lösen. Auf den Weg zu den Blättern (Basisrelationen), werden alle Operatoren in Richtung Wurzel in einer vorläufigen Todo-Liste gesammelt. Wird eine Basisrelation erreicht, müssen aus der Liste zunächst alle Operatoren herausgefiltert werden, die nicht auf die Tupel der Relation anzuwenden sind. Im letzten Schritt werden alle fertigen Todo-Listen in einem Objekt vom Typ `TodoListContainer` gesammelt.

Abbildung 5.16 zeigt die notwendigen Algorithmen in Pseudocode-Notation. Endergebnis ist ein Container, der sämtliche Todo-Listen für die Basisrelationen enthält. Die Anzahl der Todo-Liste entspricht logischerweise der Anzahl der Basisrelationen, die für die Anfrage benötigt werden.

Die Methode `filterTodoList` erfüllt noch eine weitere Aufgabe, die in Abbildung 5.16 fehlt. Beim Durchlauf durch alle Operatoren wird nach einer Selektion gesucht, die einer Punktanfrage auf das Primärschlüsselattribut entspricht. Dazu muss die Selektion zwei Bedingungen genügen:

1. Die Vergleichsoperation ist „=“
2. Das Vergleichsattribut ist das Primärschlüsselattribut

Im Falle eine solchen Selektion, wird dieses Wissen in der Todo-Liste hinterlegt

Der Algorithmus `buildTodoList` wird zu Beginn mit dem Wurzelelement des Operatorbaumes und einer leeren Todo-Liste und einem leeren Container aufgerufen. Die Zeilen 7-11 sind nur für binäre Operatoren relevant. Für unäre Operatoren wird der rechte Sohnknoten auf `null` gesetzt.

buildTodoList(*Operator, TodoList, TodoListContainer*)
1 *TodoList.insert(Operator)*
2 if *Operator.lefChild* is an operator
3 *buildTodoList(Operator.leftChild, TodoList, TodoListContainer)*
4 else if *Operator.leftChild* is an relation with ID *relID*
5 *TodoListContainer.insert(filterTodoList(TodoList, relID))*
6 end if
7 if *Operator.rightChild* is an operator
8 *buildTodoList(Operator.rightChild, TodoList, TodoListContainer)*
9 else if *Operator.rightChild* is an relation with ID *relID*
10 *TodoListContainer.insert(filterTodoList(TodoList, relID))*
11 end if

filterTodoList(*TodoList, relID*)
1 *resultTodoList = new TodoList()*
2 for each *op* from *TodoList*
3 if *op* uses relation with ID *relID*
4 *resultTodoList.insert(op)*
5 end if
6 end for
7 return *resultTodoList*

Abbildung 5.16: Algorithmen für die Erzeugung der Todo-Listen

(`exactMatchPossible=true`).

Verteilung der Todo-Listen. Bevor der Container mit den Todo-Listen an alle Peers verteilt wird, wird nach Todo-Listen gesucht, deren Attribut `exactMatchPossible` auf `true` gesetzt ist. Für diese Listen wird kein Broadcast benötigt, da sie gezielt zu den Peer geschickt werden können, auf dem sich das mögliche Ergebnistupel befinden muss (falls es tatsächlich existiert). Das gezielte Senden ist möglich, da durch die Punktanfrage auf das Primärschlüsselattribut der *key*-Wert für eine *lookup*-Operation direkt gebildet werden kann. Der *key* besteht in diesem Fall aus:

- dem Identifier der Basisrelation und
- dem Wert für das Primärschlüsselattribut (= Vergleichswert des entsprechenden Selektion-Planoperators)

Durch die Hash-Funktion wird der Peer berechnet, zu dem die Todo-Liste geschickt werden muss. Alle Todo-Liste für die das möglich ist, werden aus dem Container entfernt.

Für alle anderen Todo-Listen mus ein Broadcast des Containers durchgeführt werden. Dieser Vorgang zerfällt ein zwei Teile. Im ersten Schritt sendet ein Peer den Container an seine direkten Nachbarn. Um die Zahl der Nachrichten etwas zu minimieren, wird sich in einem Vektor gemerkt, welche Peers den Container bereits verarbeitet haben.

Schritt Nummer 2, der auch für gezielt verteilte Todo-Listen ausgeführt werden muss, ist die Suche nach Tupeln der zugehörigen Basisrelation. Verwaltet ein Peer Tupel für eine Todo-Liste, kann mit der nächsten Teilaufgabe im Ausführungsgraph (Abbildung 5.15) begonnen werden.

Um das korrekte Ergebnis zu erhalten, muss jeder Peer den Container genau einmal verarbeiten. Dafür ist aber nicht der Eddy-Operator verantwortlich, sondern die Kommunikationskomponente des Peers. Dazu wird die `TodoListID` der ersten Todo-Liste des Containers in einem Vektor auf dem Peer hinterlegt. Jedesmal wenn der Container einen Peer erreicht, wird in diesem Vektor nach der ID gesucht. Nur wenn diese nicht enthalten ist, wird der Container verarbeitet.

Eine Alternative zum Broadcast des Containers ist ein Multicast jeder Todo-Liste. Dazu müssen die Daten so im CAN verteilt worden sein, dass die Teilbereiche des Netzes bestimmt werden können, in denen sich die Tupel einer Relation befinden. Grundvoraussetzung für den Multicast ist die Eineindeutigkeit der Hash-Funktion, die im Allgemeinen nicht gegeben ist. Auf diese Möglichkeit soll nicht weiter eingegangen werden, da dies Aufgabe der CAN-Umgebung und nicht der Anfrageverarbeitung ist. Wird im Folgenden von der Verteilung und dem Broadcast der Todo-Liste gesprochen, schließt das die Ausnutzung von Multicasts mit ein.

Erzeugung der Nachrichtenpakete Wie in Kapitel 4 bereits beschrieben, setzt sich ein Nachrichtenpaket aus einem Tupel-Paket und zugehöriger Todo-Liste zusammen.

Bei der ersten Erzeugung der Nachrichtenpakete werden Kopien aller gefundenen Tupel einer Basisrelation in einem Objekt vom Typ `TuplePacket` verpackt. Dabei wird gleichzeitig der Attribute-Mapper des Tupel-Paketes initialisiert (siehe Abschnitt 5.4.3).

Zusammen mit der Todo-Liste kann nun mit der eigentlichen Verarbeitung der Anfrage begonnen werden.

Versand der Nachrichtenpakete. Nach der Bestimmung eines Ziel-Peers, werden die Tupel samt Todo-Liste weitergeleitet. Sind Ziel-Peer und aktueller Peer identisch, können gleich die Methoden für eine weitere Verarbeitung aufgerufen werden. Die Altenativmöglichkeit wäre, dass sich ein Peer in solchen Fällen selbst eine Nachricht schickt.

Da mit der Abarbeitung der initialen Nachrichtenpakete auf den Peers begonnen wird, auf denen sie erzeugt wurden, ist zu Beginn kein Versenden der Nachrichtenpakete erforderlich.

Auswahl des nächsten Operators. Die Vorstellung des Operator-Routings ist Schwerpunkt von Kapitel 6 und soll an dieser Stelle nicht weiter behandelt werden.

Sobald aber ein Operator ausgewählt wurde, wird dieser sofort aus der Todo-Liste entfernt.

Verarbeitung der Nachrichtenpakete mit Hilfe des entsprechenden Planoperators. Diese Aufgabe wird nahezu vollständig an den ausgewählten Planoperator delegiert. Der Eddy-Operator entscheidet lediglich, ob die Tupel des Paketes am Stück oder Schritt für Schritt abgearbeitet werden sollen (vergleiche Abschnitt 5.5.1).

Im Falle einer Projektion oder Selektion sind die Rückgabewerte wieder vom Typ `TuplePacket`. Diese unären Operatoren sind damit vollständig durchlaufen.

Joins müssen die Schleife aus Abbildung 5.15 quasi zweimal durchlaufen. Im ersten Durchlauf besteht die Verarbeitung des Nachrichtenpaketes aus dem Re-Hashing der Tupel. Dies legt automatisch den Ziel-Peer für das Peer-Routing fest. Erreichen die neuverteilten Tupel ihr Ziel beginnt der zweite Durchlauf. Dabei muss das Operator-Routing übersprungen werden. Die Verarbeitung entspricht dann der Ausführung des lokalen Joins. Wie in Abschnitt 5.5.4 gezeigt, ist der Rückgabewert des Join-Planoperators ein Container der Elemente vom Typ `TuplePacketWrapper` enthält. Jeder Wrapper steht dabei wieder für ein komplettes Nachrichtenpaket.

Erzeugung der Ergebnisnachrichtenpakete. Bevor die Ergebnistupel weiter verarbeitet werden können, müssen sowohl Tupelpaket als auch Todo-Liste aktualisiert werden.

Für Tupelpakete bedeutet dies die Anpassung des zugehörigen Attribute-Mappers. Dazu ruft der Eddy-Operator die Konvertierungsmethoden des eben ausgeführten Planoperators auf.

Die Aktualisierung der Todo-Liste bezieht sich auf das korrekte Setzen der Ready-Bits der restlichen Operatoren, um Verletzungen der Umformungsregeln der Relationenalgebra zu vermeiden. Die Verschmelzung von Todo-Listen aufgund von Joins wird bereits durch den Join-Planoperator ausgeführt. Da Selektionen und Joins keine Attribute aus den Tupeln entfernen, können diese zu jeder Zeit ausgeführt werden. Somit kann das Ready-Bit für diese Operatoren von Anfang an gesetzt sein. Durch eine Projektion kann es allerdings passieren, dass Selektions- oder Join-Attribute herausprojeziert werden. Dies muss durch die Ready-Bits verhindert werden. Daneben würde die Ausführung einer Projektion vor einem Join ggf. eine Nachprojektion mitsichbringen, um das gleiche Ergebnis zu erzielen, wie es in umgekehrter Reihenfolge der Ausführung entstehen würde. Da Pro-

jektionen ohnehin eine der letzten Operationen einer Anfrage darstellen, müssen folgende Bedingungen erfüllt sein, damit das Ready-Bit einer Projektion gesetzt werden darf:

1. Die Todo-Liste darf keine weiteren Joins enthalten.
2. Die Projektion darf Selektionsattribute der restlichen Selektionen nicht herausprojezieren (der `AID`-Vektor der Projektion muss die AIDs sämtlicher Selektionen der Todo-Liste enthalten)

Abbildung 5.17 zeigt den Pseudocode für das Setzen der Ready-Bits für die Projektionen.

Auswahl des nachsten Peers. Wie schon das Operator-Routing, ist auch das Peer-

```
updateTodoList(TodoList)
1   joinCounter = 0
2   collectedAIDs = new Vector()
3   projections = new Vector()
4   for each item from TodoList
5       currentOperator = item.getOperator()
6       if currentOperator is an EddySelPOP
7           collectedAIDs = currentOperator.getAIDs()
8       if currentOperator is an EddyProjPOP
9           projections.add(currentOperator)
10      if currentOperator is an EddyJoinPOP
11          joinCounter++
12      end if
13  end for
14  if joinCounter == 0
15      hits = 0
16      for each proj from projections
17          projAIDs = proj.getAIDs()
18          for each aid from projAIDs
19              if aid is element of collectedAIDs
20                  hits++
21              end if
22          end for
23          if hits == collectedAIDs.getSize
24              proj.setReadyBite(true)
25          end if
26      end for
27  end if
```

Abbildung 5.17: Algorithmus für das Setzen der Ready-Bits

Routing Thema von Kapitel 6.

Ausgabe der Ergebnistupel. Die Tupel eines Paketes sind dann Ergebnistupel,

wenn die zugehörige Todo-Liste keine Operatoren mehr enthält. In diesem Fall wird das Tupelpaket an den Peer geschickt, an den die Anfrage ursprünglich gestellt wurde. Diese Information steckt als Peer-Identifier in der `QueryID`, die bei jedem Nachrichtenaustausch mitgeschickt wird.

Im Rahmen seiner Aufgaben stellt der Eddy-Operator die Schnittstelle zwischen P2P-Eddy und dem CAN dar. So hat hat der Eddy-Operator als einziger direkten Zugriff auf folgende Bestandteile der Peers:

- *Kommunikationskomponente*
  Als einziger Teil des P2P-Eddies ist der Eddy-Operator in der Lage Nachrichen zu verschicken.

- *Datenbestand*
  Die Planoperatoren arbeiten alle auf Kopien der Originaltupel der Basisrelationen. Der Zugriff auf die Basisrelationen wird allein bei der Erzeugung der initialen Tupelpakete benötigt.

- *lokale Laufzeitstatistiken*
  Der Zugriff auf die Warteschlangenlängen und die erlernten Selektivitäten der Operatoren sowie die zusätzlichen Informationen über die direkten Nachbarn des Peers, sind Grundlage einiger Routing-Strategien.

Eine dynamische Anfrageverarbeitung wie der P2P-Eddy, lässt einen großen Spielraum für die Ausführung. Gerade der Eddy-Operator verfügt über eine Vielzahl von Parametern, deren Werte dessen Arbeitsweise beeinflussen.

- `operatorRoutingMethod`
  legt die Strategie fest, nach welcher der nächste Operator für die Abarbeitung ausgewählt wird

- `peerRoutingMethod`
  legt die Strategie fest, nach welcher der nächste Ziel-Peer für ein Nachrichtenpaket ausgewählt wird (falls diese Auswahl unabhängig vom Operator-Routing ist)

- `checkWorkload`
  Ja/Nein-Entscheidung, ob die Auslastung eines Peers Einfluss auf das Peer-Routing hat

- `processingMethod`
  legt fest, ob die Tupel eines Nachrichtenpaketes am Stück oder in Form von Teilpaketen verarbeitet werden sollen

- `usingSharedData`
  Ja/Nein-Entscheidung, ob die Operator-Routing-Strategien „Warteschlangenlänge“ und „Selektivität“ auf ein globales Wissen zurückgreifen können

- `findNearestJoin`
  Ja/Nein-Entscheidung, ob die Zusatz-Routing-Strategie „Suche nach den nächsten Join“ verwendet werden soll

- `maxPacketSize`
  skalarer Zahlenwert, der die maximale Anzahl von Tupeln in einem Tupelpaket vorgibt

- `maxHops`
  skalarer Zahlenwert, der die Anzahl festlegt, wie oft in Folge ein Nachrichtenpaket von Peer zu Peer geschickt werden kann, ohne verarbeitet werden zu müssen

- `maxWorkloadClock`
  skalarer Zahlenwert, der den Grenzwert festlegt, ab wann ein Peer als ausgelastet angesehen wird; ein Peer ist dann ausgelastet, sobald die Anzahl aller eingehenden Nachrichten in einem festen Zeitintervall diesen Grenzwert überschreitet

- `maxWorkloadQueue`
  skalarer Zahlenwert, der den Grenzwert festlegt, ab wann ein Peer als ausgelastet angesehen wird; ein Peer ist dann ausgelastet, sobald die Summe aller Warteschlangen auf dem Peer diesen Grenzwert überschreitet

- `packTuples`
  Ja/Nein-Entscheidung, ob ein Re-Hashing für alle Tupel eines Tupelpaketes einzeln durchgeführt werden soll; die Alternative ist eine Vorsortierung der Tupel (vorgestellt im Abschnitt 5.5.4)

Die vielen möglichen Parameterkombinationen machen den Eddy-Operator äußerst flexibel. Obwohl im Normalfall die Arbeitsweise dem Anwender oder der Applikation verborgen bleiben soll, können sämtliche Parameter von Außen über den Konstruktor gesetzt werden. Dadurch ist es möglich, sinnvolle Kombinationen der Parameter gezielt miteinander zu vergleichen.

5.7 Nachrichtenklassen

5.7.1 Grundprinzip der Kommunikation

Innerhalb der Systemumgebung erfolgt die Kommunikation zwischen Peers über spezielle Nachrichtenklassen. So existiert für jede Aufgabe eine eigene Klasse, mit den jeweils benötigten Informationen.

Für den P2P-Eddy wurden fünf neue Typen von Nachrichtenklassen implementiert. Sie werden alle von der Oberklasse `DirectedMessage` abgeleitet (Abbildung 5.18). Diese Klasse besteht aus folgenden Attributen:

- `sender`
  Peer, von dem die Nachricht aus geschickt wurde

- `messageID`
  eindeutiger Identifier der Nachricht

- `messageTarget`
  Zielpunkt im Koordinatenraums des CAN

In den weiteren Abschnitten sollen die neuen Nachrichtenklassen und ihre zusätzlichen Attribute vorgestellt werden.

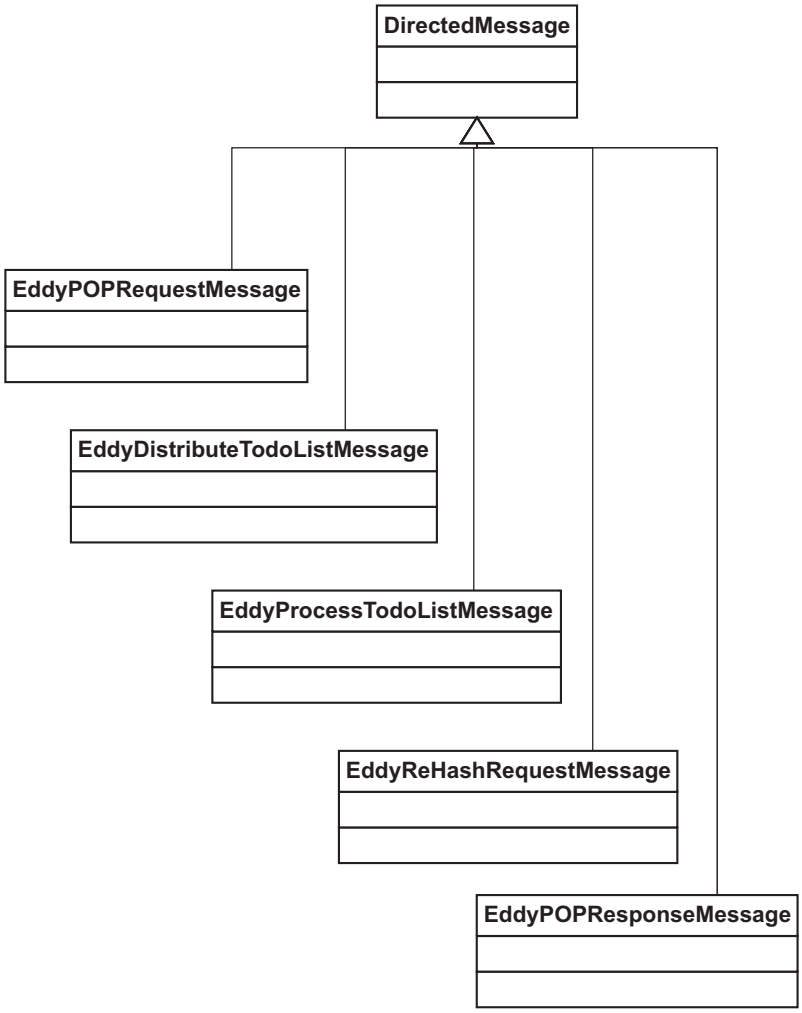

Abbildung 5.18: Vererbungshierarchie der neuen Nachrichtenklassen für den P2P-Eddy

5.7.2 Die Klasse `EddyPOPRequestMessage`

Mit dieser Nachricht wird eine Anfrage für den P2P-Eddy initiiert. Sie wird an den Peer geschickt, von dem aus die Anfrage gestartet werden soll. Die Attribute der Klasse `EddyPOPRequestMessage` sind:

- `EddyPOP`
  Anfrage in Baumstruktur (übergebener Operator ist Wurzelelement)

- `Eddy`
  Eddy-Operator mit den gewählten Parametern (vergleiche Abschnitt 5.6)

- `QueryID`
  eindeutiger Identifier der Anfrage
- `PeerDescriptor`
  Peer an dem Anfrage gestartet werden soll

5.7.3 Die Klasse `EddyDistributeTodoListMessage`

Diese Nachricht ist für den Broadcast des Containers mit den Todo-Listen zuständig. Für die korrekte Durchführung sind folgende Attribute notwendig:

- `TodoListContainer`
  Container mit den Todo-Listen
- `Eddy`
  Eddy-Operator mit den gewählten Parametern (vergleiche Abschnitt 5.6)
- `visitedPeers` Vektor mit alle Peers, auf dem der Container bereits verarbeitet wurde
- `PeerDescriptor`
  Peer, von dem aus die Anfrage gestartet wurde
- `QueryID`
  eindeutiger Identifier der Anfrage
- `startTime`
  Sendezeitpunkt der Anfrage (dient zur Bestimmung der Übertragungsdauer zwischen zwei Peers)

5.7.4 Die Klasse `EddyProcessTodoListMessage`

Innerhalb der Klasse `EddyProcessTodoListMessage` werden die Tupelpakete inkl. ihrer Todo-Listen von Peer zu Peer geschickt. Neben der Klasse `EddyReHashRequestMessage` gehört sie zu den beiden Nachrichtentypen, die für die eigentliche Verarbeitung der Tupel benötigt wird. Die Attribute der Klasse sind:

- `Eddy`
  Eddy-Operator mit den gewählten Parametern (vergleiche Abschnitt 5.6)
- `TuplePacket`
  Tupelpaket mit gleichartigen Tupeln
- `TodoList`
  zugehörige Todo-Liste für das Tupelpaket
- `TodoListItem`
  Element aus der Todo-Liste, falls der nächste Operator bereits vom Sender-Peer festgelegt wurde

- `PeerDescriptor`
  Peer, von dem aus die Anfrage gestartet wurde
- `QueryID`
  eindeutiger Identifier der Anfrage
- `startTime`
  Sendezeitpunkt der Anfrage (dient zur Bestimmung der Übertragungsdauer zwischen zwei Peers)

5.7.5 Die Klasse `EddyReHashRequestMessage`

Wie der Name bereits andeutet, werden mit dieser Nachricht Tupel neu verteilt. Sie wird zu dem Peer geschickt auf dem dann die Tupel temporär eingefügt werden. Die Klasse enthält folgende Attribute:

- `EddyPOP`
  Join-Planoperator der für den lokalen Join auf dem Ziel-Peer benötigt wird
- `Eddy`
  Eddy-Operator mit den gewählten Parametern (vergleiche Abschnitt 5.6)
- `Tuple` Hilfstupel, welches neu verteilt werden soll (dieses Tupel enthält eine Objekt vom Typ `TuplePacketWrapper`, welcher wiederum das Tupelpaket mit den eingentlichen Datentupeln und die Todo-Liste enthält)
- `TodoList`
  zugehörige Todo-Liste für das Tupelpaket
- `PeerDescriptor`
  Peer, von dem aus die Anfrage gestartet wurde
- `QueryID`
  eindeutiger Identifier der Anfrage

5.7.6 Die Klasse `EddyPOPResponseMessage`

Mittels dieser Nachricht werden letztlich die Ergebnistupel einer Anfrage zu dem Peer geschickt, an den die Anfrage gestartet wurde. Die Attribute der Klasse sind:

- `Eddy`
  Eddy-Operator mit den gewählten Parametern (vergleiche Abschnitt 5.6)
- `TuplePacket`
  Tupelpaket mit den Ergebnistupeln
- `PeerDescriptor`
  Peer, von dem aus die Anfrage gestartet wurde
- `QueryID`
  eindeutiger Identifier der Anfrage

Kapitel 6

Routing-Strategien

6.1 Allgemeines

Sowohl Operator-Routing als auch Peer-Routing sind Aufgaben des Eddy-Operators. Für beide Varianten wird die Klasse `Eddy` um jeweils eine Methode erweitert. Die konkrete Routing-Strategie wird den Methoden als Parameter übergeben.

Für das Operator-Routing ist die Methode `chooseNextTodoListItem` zuständig. Die übergebenen Parameter sind die Todo-Liste des aktuellen Nachrichtenpaketes und die Strategie, nach welcher der nächste Operator ausgewählt werden soll.

Abbildung 6.1 zeigt die Struktur der Methode `chooseNextTodoListItem`. Die Konstanten für die `switch`-Anweisung repräsentieren die im Abschnitt 4.3.2 vorgestellten Strategien für das Operator-Routing.

Die Hilfsstrategie zum Finden des „nächsten“ Joins kann nicht für sich allein verwendet werden. Sie kann nur an gegebener Stelle zusätzlich aufgerufen werden. Mehr dazu bei der Implementierung der Operator-Routing-Strategien.

Die Methode `chooseNextPeer` wählt den nächsten Peer für ein Nachrichtenpaket aus, falls das Ziel nicht durch das Operator-Routing implizit vorgegeben ist. Der Übergabeparameter ist die verwendete Strategie. Realisiert wurde der Entscheidungsbaum aus Abbildung 4.7. Dazu muss neben den bereits vorgestellten Strategien für das Peer-Routing auch der Test auf Auslastung integriert werden. Die Grobstruktur der Methode `chooseNextPeer` zeigt Abbildung 6.2.

Vom Eddy-Operator wird die Methode immer mit der Konstante `NEXT_PEER_WORKLOAD` aufgerufen. Im entsprechenden Teilzweig wird zunächst überprüft, ob die Auslastung eines Peers herangezogen werden soll oder nicht. In Abhängigkeit davon und der gewählten Routing-Strategie, wird die Methode rekursiv mit der entsprechenden Konstante aufgerufen.

In einem Zwischenschritt (Zeilen 2-7) werden alle Elemente der übergebenen Todo-Liste haerausgefiltert, deren Ready-Bit gesetzt ist. Nur aus dieser Teilmenge darf der nächste Operator ausgewählt werden.

chooseNextTodoListItem(*TodoList, operatorRoutingMethod*)
```
1  trueList = new TodoList()
2  for each item from TodoList
3      if item.getReadyStatus() == true
4          trueList.add(item)
5      end if
6  end for
7  resultItem = new TodoListItem()
8  switch (operatorRoutingMethod)
9      case NEXT_OPERATOR_RANDOM:
10         ...
11     case NEXT_OPERATOR_HIGHEST_PRIORITY:
12         ...
13     case NEXT_OPERATOR_MIN_QUEUE_LENGTH:
14         ...
15     case NEXT_OPERATOR_TICKET:
16         ...
17 end switch
18 return resultItem
```

Abbildung 6.1: Struktur der Methode `chooseNextTodoListItem`

chooseNextPeer(*method*)
```
1  resultPeer = new PeerDescritpor()
2  switch (method)
3      case NEXT_PEER_WORKLOAD:
4          ...
5      case NEXT_PEER_SAME_PEER:
6          ...
7      case NEXT_PEER_RANDOM_NEIGHBOR:
8          ...
9      case NEXT_PEER_CYCLIC_NEIGHBOR:
10         ...
11     case NEXT_PEER_PING_TIME:
12         ...
13     case NEXT_PEER_SENT_MESSAGES:
14         ...
15 end switch
16 return resultPeer
```

Abbildung 6.2: Struktur der Methode `chooseNextPeer`

6.2 Operator-Routing

6.2.1 Zufällige Auswahl

Diese einfachste aller Strategien für das Operator-Routing kommt ohne zusätzliche Informationen aus. Das ausgewählte Element der Todo-Liste wird über die Position im Vektor der Todo-Liste angesprochen. Die Position berechnet sich dabei als zufälliger Wert aus dem Intervall [0,...,(*n*-1)], wobei *n* die Anzahl der Elemente der Todo-Liste ist. Abbildung 6.3 zeigt den betreffenden Ausschnitt der Methode `chooseNextTodoListItem`.

chooseNextTodoListItem(*TodoList, operatorRoutingMethod*)

```
1  ...
2  case NEXT_OPERATOR_RANDOM:
4      random = Math.random //random value between „0“ and „1“
5      position = Math.round(radom * (trueList.getSize() - 1))
6      resultItem = trueList.getListItemByPosition(position)
7  end case
8  ...
```

Abbildung 6.3: Algorithmus für die Operator-Routing-Strategie „Zufällige Auswahl“

Dieses Verfahren ermöglicht, mit Ausnahme der Einschränkungen durch die Ready-Bits, eine beliebige Operatorreihenfolge. Ungünstige Reihenfolgen können dadurch nicht ausgeschlossen werden. Damit ist die „Zufällige Auswahl“ keine echte Routing-Strategie, da hier die Effizienz der Anfrageverarbeitung nicht gezielt verbessert wird. Mit ihr lassen sich allerdings gut die Vorteile „richtiger“ Routing-Strategien demonstrieren.

6.2.2 Auswahl nach Priorität

Bei dieser Strategie wird die Erweiterung der Listenelemente um eine Operatorpriorität benötigt. Operatoren mit einer hohen Priorität werden dabei bevorzugt ausgewählt. Besitzen mehrere Operatoren die gleiche höchste Priorität, wird aus dieser Teilmenge der Operator zurückgegeben, der am weitesten oben in der Todo-Liste steht. Die Prioritäten für die drei implementierten Planoperatoren besitzen folgende Rangfolge:

$$prio(EddySelPOP) > prio(EddyProjPOP) > prio(EddyJoinPOP)$$

Abbildung 6.4 zeigt den Algorithmus für die Auswahl nach Priorität. Dieser besteht aus genau einem Durchlauf der gesamten Todo-Liste. Eine Optimierung wäre, wenn der Durchlauf abgebrochen werden würde, sobald die erste Selektion gefunden wurde. Da der Aufwand für lokale Operationen im Vergleich zum Kommunikationsaufwand aber vernachlässigt werden kann, wurde darauf verzichtet.

Enthält die Todo-Liste sowohl Selektionen als auch Joins, werden somit immer erst alle Selektionen ausgeführt. Besonders ungünstige Operatorreihenfolgen werden dadurch vermieden, die unnötig große Zwischenergebnisse erzeugen würde. Je kleiner die Zwischenergebnisse, umso geringer ist vor allem auch die Auslastung des Netzes.

chooseNextTodoListItem(*TodoList, operatorRoutingMethod*)

```
1  ...
2  case NEXT_OPERATOR_HIGHEST_PRIORITY:
3      oldHighestPriority = 0
4      for each item from TodoList
5          currentPriority = item.getPriority()
6          if currentPriority > oldHighestPriority
7              oldHighestPriority = currentPriority
8              resultItem = item
9          end if
10     end for
11 end case
12 ...
```

Abbildung 6.4: Algorithmus für die Operator-Routing-Strategie „Auswahl nach Priorität“

6.2.3 Auswahl nach Warteschlangenlänge

Da die Tupel nicht wirklich in Warteschlangen eingefügt werden und somit keine Warteschlangenlänge ausgelesen werden kann, muss dies geeignet simuliert werden. Erreicht wird dies durch einen Zähler für jeden Operator auf jedem Peer. Wird ein Operator ausgeführt, muss der Zähler um die Anzahl der ankommenden Tupel erhöht, nach der Abarbeitung um die gleiche Anzahl wieder erniedrigt werden.

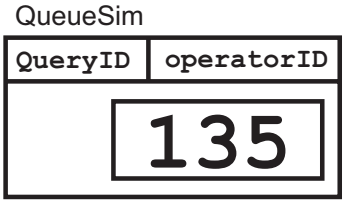

Abbildung 6.5: Beispiel für eine Instanz der Klasse `QueueSim`

Der Zähler muss seinem Operator eindeutig zugeordnet werden können, auch im Falle mehrerer parallel ausgeführter Anfragen. Aus diesem Grund muss der Zähler um die `QueryID` der Anfrage und der ID des Operators erweitert werden. Implementiert wird das Ganze durch die Klasse `QueueSim`. Um alle QueueSims auf einem Peer gemeinsam verwalten zu können, werden sie in einem Objekt vom Typ `QueueSimContainer` gepackt. Jeder Peer wird um einen solchen Container eweitert.

Ein `QueueSim` steht auf einem Peer allerdings erst dann zur Verfügung, wenn der zugehörige Operator mindestens einmal auf diesem Peer ausgeführt wurde. Sobald also die Todo-Liste Operatoren enthält, für die noch kein `QueueSim` existiert, kann für die Auswahl nach der Warteschlangenlänge keine eindeutige Entscheidung getroffen werden. Gleiches gilt, wenn mehrere Operatoren die gleiche minimale Warteschlangenlänge besitzen. Auf dieser Teilmenge aus unbekannte Operatoren und Operatoren mit gleicher minimaler Warteschlangenlänge wird die Strategie „Auswahl nach Priorität“ ausgewählt.

Den Code-Ausschnitt der Methode `chooseNextTodoListItem` für diese Stratgie zeigt Abbildung 6.6.

In Zeile 19 wird geprüft, ob eine eindeutige Entscheidung auf Basis der Warteschlangenlänge getroffen werden kann. Ist dies nicht der Fall, wird die Methode rekursiv mit neuem Parameter aufgerufen.

chosseOperator(*TodoList, operatorRoutingMethod*)

```
1  switch (operatorRoutingMethod)
2  ...
3  case NEXT_OPERATOR_MIN_QUEUE_LENGTH:
4      unknownOperators = new TodoList()
5      minimumOperators = new TodoList()
6      for each op in TodoList
7          if for op a queue does not exist on the current peer
8              unknownOperators.insert(op)
9          else
10             currentQueueLength = queue.getQueueLength()
11             if currentQueueLength == minimumQueueLength
12                 minimumOperators.insert(op)
13             else if currentQueueLength < minimumQueueLength
14                 minimumOperators.clearList()
15                 minimumOperators.insert(op)
16             end if
17         end if
18     end for
19     if minimumOperators.size() == 1 and unknownOperators.size() == 0
20         return minimumOperators.getElement()
21     else
22         combinedList = minimumOperators + unknownOperators
23         return chooseOperator(combinedList, NEXT_OPERATOR_HIGHEST_PRIORITY)
24     end if
25 end case
26 ...
```

Abbildung 6.6: Algorithmus für die Operator-Routing-Strategie „Auswahl nach Warteschlangenlänge“

6.2.4 Auswahl nach erlernter Selektivität

Obwohl mit diesem Verfahren ein ganz anderes Ziel verfolgt wird als mit der „Auswahl nach Warteschlangenlänge“, ist die Umsetzung durchaus vergleichbar. Sowhl die notwendigen Erweiterungen als auch der Algorithmus für die `chooseNextTodoListItem`-Methode sind sich äußerts ähnlich.

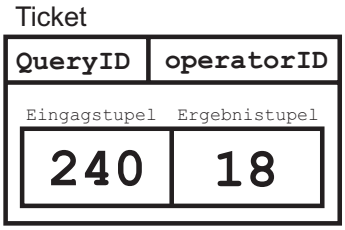

Abbildung 6.7: Beispiel für ein Ticket

Das Gegenstück zum `QueueSim` ist hier die Klasse `Ticket`. Der Unterschied liegt nur darin, dass ein Ticket zwei Zähler benötigt. Der eine Zähler hält fest, wie viele Tupel vom zugehörigen Operator auf einem Peer abgearbeitet wurden und der andere, wie viele Tupel den Operator erfolgreich passiert haben. Aus dem Verhältnis der beiden Zähler lässt sich die Selektivität des Operators abschätzen. Mit der Klasse `TicketContainer` lassen sich mehrere Tickets gemeinsam von einem Peer verwalten.

Auch bei diesem Ticket-Mechanismus kann es vorkommen, dass nicht alle Operatoren aus der Todo-Liste dem Peer bekannt sind. Gelöst wird dieses Problem wie bei der Strategie „Auswahl nach Warteschlangenlänge“ durch den rekursiven Aufruf der Methode `chooseNextTodoListItem`. Parameter sind wieder die Teilmenge aus unbekannten Operatoren und Operatoren mit gleicher minimaler Selektivität sowie die Konstante für die Strategie „Auswahl nach Priorität“.

Der Algorithmus des Verfahrens ist mit dem aus Abbildung 6.6 nahezu identisch. Lediglich die Berechnung der Selektivität kommt hier noch dazu (siehe Abbildung 6.8)

6.2.5 Hilfsmethode `findNearestJoin`

Abbildung 6.9 zeigt den Algorithmus für das Finden des „nächsten“ Joins. Für jeden Join-Operator der Todo-Liste werden die Abstände vom aktuellen Peer zum Ziel-Peer der Neuverteilung aller Tupel berechnet und gemittelt. Der Join mit dem kleinsten durchschnittlichen Abstand wird dann zurückgegeben.

Wie bereits erwähnt, ist diese Methode keine eigene Routing-Strategie, da sie operatorspezifisch ist. Bleibt also die Frage, wann im Laufe des Peer-Routings die Methode verwendet werden soll. Als geeignete Stelle hat sich der Aufruf innerhalb der Routing-Strategie „Auswahl nach Priorität“ erwiesen, wenn die Liste mit den Elementen, deren Ready-Bit gesetzt ist, nur Joins enthält. Damit kann sie auch ausgeführt werden, wenn für die Laufzeitstatistiken Warteschlangenlänge und Selektivität keine eindeutige Entscheidung getroffen werden kann. Der überarbeitete Algorithmus für die Strategie „Auswahl nach Priorität“ findet sich in Abbildung 6.10.

In Zeile 20 wird geprüft, ob eine eindeutige Entscheidung auf Basis der Tickets getroffen werden kann. Ist dies nicht der Fall, wird die Methode rekursiv mit neuem Parameter aufgerufen.

chosseOperator(*TodoList, operatorRoutingMethod*)

```
1  switch (operatorRoutingMethod)
2  ...
3  case NEXT_OPERATOR_TICKET:
4      unknownOperators = new TodoList()
5      minimumOperators = new TodoList()
6      oldMinimumSelectivity = 1.0
7      for each op in TodoList
8          if for op a ticket does not exist on the current peer
9              unknownOperators.insert(op)
10         else
11             currentSelectivity = ticket.computeSelectivity()
12             if currentSelectivity == oldMinimumSelectivity
13                 minimumOperatorurrentQueueLengths.insert(op)
14             else if currentSelectivity < oldMinimumSelectivity
15                 minimumOperators.clearList()
16                 minimumOperators.insert(op)
17             end if
18         end if
19     end for
20     if minimumOperators.size() == 1 and unknownOperators.size() == 0
21         return minimumOperators.getElement()
22     else
23         combinedList = minimumOperators + unknownOperators
24         return chooseOperator(combinedList, NEXT_OPERATOR_HIGHEST_PRIORITY)
25     end if
26 end case
27 ...
```

Abbildung 6.8: Algorithmus für die Operator-Routing-Strategie „Auswahl nach erlernter Selektivität“

findNearestJoin(*TodoList*)

```
1  minDistance = MAX_VALUE
2  for each item from TodoList
3      if item is an EddyJoinPOP
4          currentAccumulatedDistance = 0
5          currentAverageDistance = 0
6          currentJoin = item.getOperator()
7          currentNamespace = currentJoin. getNamespace()
8          for each tuple from msg.getTuplePacket
9              currentAttribute <- find join attribute
10             currentTargetPoint <- lookup((namespace, currentAttribute))
11             currentDistance <- calculate distance between currentTargetPoint and peer
12             currentAccumulatedDistance = currentAccumulatedDistance + currentDistance
13         end for
14         currentAverageDistance = currentAccumulatedDistance / (count of tuples)
15         if currentAverageDistance < minDistance
16             minDistance = currentAverageDistance
17             nearestJoin = item
18         end if
19     end if
20 end for
21 return nearestJoin
```

Abbildung 6.9: Algorithmus der Methode `findNearestJoin`

Durch die zusätzliche Bedingung in der if-Klausel kann die Verwendung der Methode `findNearestJoin` auch völlig ausgeschaltet werden.

chooseNextTodoListItem(*TodoList, operatorRoutingMethod*)

```
1  ...
2  case NEXT_OPERATOR_HIGHEST_PRIORITY:
3      oldHighestPriority = 0
4      if trueList.getSize() == (count of joins) and findNearesJoin == YES
5          resultItem = findNearesJoin(trueList)
6      else
7          for each item from TodoList
8              currentPriority = item.getPriority()
9              if currentPriority > oldHighestPriority
10                 oldHighestPriority = currentPriority
11                 resultItem = item
12             end if
13         end for
14     end if
15 end case
16 ...
```

Abbildung 6.10: Algorithmus für die Operator-Routing-Strategie „Auswahl nach Priorität“ inkl. dem Aufruf für die Methode `findNearestJoin`

6.3 Peer-Routing

6.3.1 Allgemeines

Für das Peer-Routing wurden zwei neue Klassen implementiert. Die Klasse `Neighbor` sammelt verschiedene Informationen über einen Peer in seiner Rolle als direkter Nachbar. Derzeit sind das folgende Größen:

- `pingTime`
  Zeitdauer zwischen Senden und Empfang der letzten Nachricht (Sender ist der zugehörige Nachbar; Empfänger der Peer, der die entsprechende Instanz der `Neighbor`-Klassen enthält)

- `sentMessages`
  Anzahl der gesendeten Nachrichten eines Nachbarn

Jeder Peer wird um eine Instanz der Klasse `NeighborManager` erweitert. Diese Klasse verwaltet in erster Linie einen Array (`neighbors`, dessen Elemente vom Typ Neighbor sind und alle direkten Nachbarn des Peers repräsentieren. Die Größe des Arrays entspricht somit die Anzahl der Nachbarn.

Die meisten Algortihmen für die Strategien des Peer-Routings befinden sich in der Klasse `NeighborManager` und werden lediglich von der Methode `chooseNextPeer` aufgerufen.

Die Auswahl des nächsten Peers soll auf Basis des Entscheidungsbaumes aus Abbildung 4.7. Dies bedeutet, dass das Peer-Routing auch den Test auf Auslastung eines Peers umsetzen muss. Die Methode `chooseNextPeer` wird dazu im ersten Schritt immer mit der Konstanten NEXT_PEER_WORKLOAD aufgerufen, unabhängig von der eigentlichen Peer-Routing-Strategie. Der zugehöre Code-Abschnitt realisiert genau den oben genannten Entscheidungsbaum.

Das Attribut `peerRoutingMethod` steht für die gewählte Routing-Strategeie. Durch die if-Klausel in Zeile 3 kann der Test auf Auslastung auch deaktiviert werden.

chooseNextPeer(*method*)

```
1  ...
2  case NEXT_PEER_WORKLOAD:
3      if checkWorkload == YES
4          workload = peerQueueSimContainer.getWorkload()
5          if workload < maxWorkload
6              resultPeer = chooseNextPeer(SAME_PEER)
7          else
8              resultPeer = chooseNextPeer(peerRoutingMethod)
9          end if
10     else
11         resultPeer = chooseNextPeer(peerRoutingMethod)
12     end if
13 end case
14 ...
```

Abbildung 6.11: Algorithmus für den Test auf Auslastung eines Peers

Zur Bestimmung der Auslastung eines Peers anhand der Anzahl der eingegangen Nachrichten in einem Zeitintervall, dient die Klasse `WorkloadManager` als Erweiterung für jeden Peer. Die Klasse sammelt in einem Vektor (`messages`) die Ankunftszeiten aller eingehenden Nachrichtenpakete. Bei jedem Einfügen einer neuen Ankunftszeit werden veraltete Zeiten aus dem Vektor entfernt. Eine Ankunftszeit ist dann veraltet, wenn sie vor dem Zeitintervall liegt, der sich durch die neuste Ankunftszeit und einem festgelegten Zeitabschnitt ergibt (siehe Algorithmus 6.12).

Je kürzer der Abstand zwischen den Ankünften eingehender Nachrichtenpakete ist, umso mehr Elemente befinden sich somit im Vektor `messages`. Die Größe des Vektors repräsentiert die Auslastung des Peers. Beim Test auf Überlastung wird die Vektorgröße mit einem festgelegten Grenzwert (`maxWorkloadClock`) verglichen.

Wird die Summe aller Eingangswarteschlangenlängen als Maß für die Auslastung eines Peers herangezogen, kann auf neue Klassen verzichtet werden. Dieser Parameter kann bereits durch die Klassen `QueueSim` und `QueueSimContainer` bereitgestellt werden. Lediglich ein Grenzwert muss festgelegt werden (`maxWokloadQueue`).

```
insertMessage(messageTime)
1  minimumTime = messageTime - INTERVAL
2  if minimumTime < 0
3      minimumTime = 0
4  end if
5  for each time from messages
6      if time < minimumTime
7          messages.removeMessageTime(time)
8      end if
9  end for
10 messages.addMessageTime(messageTime)
```

Abbildung 6.12: Algorithmus für die Aktulasierung des Vektor der Klasse `WorkloadManager`

6.3.2 Routing-Strategien

6.3.3 Gleicher Peer

Beim Aufruf der Methode `chooseNextPeer` mit der Konstanten `NEXT_PEER_SAME_PEER`, wird einfach der aktuelle Peer zurückgegeben (siehe Abildung 6.13)

```
chooseNextPeer(method)
1  ...
2  case NEXT_PEER_SAME_PEER:
3      resultPeer <- current peer
4  end case
5  ...
```

Abbildung 6.13: Algorithmus für die Peer-Routing-Strategie „Gleicher Peer“

6.3.4 Zufälliger Nachbar

Diese Routing-Strategie wählt aus allen direkten Nachbarn einen zufälligen aus (Abbildung 6.14). Eine faire Verteilung der Nachrichtenpakete im Netz kann dadurch nicht erreicht werden.

6.3.5 Zyklische Auswahl

Hier werden alle Nachbarn zyklisch hintereinander mit Nachrichtenpaketen bedient. Im Neighbor-Manager des Peers wird in einem Attribut (`lastRecepient`) der Nachbar gemerkt, der als letzter ein Paket erhalten hat.

```
chooseNextPeer(method)
1  ...
2  case NEXT_PEER_SAME_PEER:
3      resultPeer = peerNeighborManager.getRecepientByRandom()
4  end case
5  ...

getRecepientByRandom()
1  random = Math.random() //random value between „0“ and „1“
2  number = Math.round(random * (neighbors.length - 1))
3  return neighbors[number]
```

Abbildung 6.14: Algorithmen für die Peer-Routing-Strategie „Zufällige Auswahl“

```
chooseNextPeer(method)
1  ...
2  case NEXT_PEER_CYCLIC_NEIGHBOR:
3      resultPeer = peerNeighborManager.getRecepientByCycle()
4  end case
5  ...

getRecepientByCycle()
1  lastRecepient++
2  if lastRecepient > neighbors.length
3      lastRecepient = 0
4  end if
5  return neighbors[lastRecepient]
```

Abbildung 6.15: Algorithmen für die Peer-Routing-Strategie „Zyklische Auswahl“

6.3.6 Suche nach der schnellsten Verbindung

In einer Schleife über alle Nachbarn wird nach dem kleinsten Wert für das Attribut `pingTime` gesucht. Dieser Wert steht aber nur dann zur Verfügung, wenn mindestens eine Nachricht von diesem Peer empfangen wurde. Andernfalls ist die Güte für die Verbindung nicht bekannt und es kann keine eindeutige Entscheidung getroffen werden. In solchen Fällen wird auf die Strategie „Zyklische Auswahl“ zurückgegriffen.

Abbildung zeigt die beiden Algorithmen für die Strategie.

chooseNextPeer(*method*)

```
1  ...
2  case NEXT_PEER_PING_TIME:
3      peer = peerNeighborManager.getRecepientByPingTime()
4      if peer is not null
5          resultPeer = peer
6      else
7          resultPeer = chooseNextPeer(NEXT_PEER_CYCLIC_NEIGHBOR)
8      end if
9  end case
10 ...
```

getRecepientByPingTime()

```
1  currentMinPingTime = MAX_VALUE
2  for each neighbor from neighbors
3      if neighbor.getPingTime() is unknown
4          return null
5      else
6          if neighbor.getPingTime() < currentMinPingTime
7              currentReturnNeighbor = neighbor
8              currentMinPingTime = neighbor.getPingTime()
9          end if
10     end if
11 end for
12 return currentReturnNeighbor
```

Abbildung 6.16: Algorithmen für die Peer-Routing-Strategie „Suche nach der schnellsten Verbindung“

6.3.7 Auswahl nach Auslastung der Nachbarn

Gesucht wird innerhalb aller Nachbarn nach dem kleinsten Nachrichtenvektor `sentMessages` (siehe Abbildung 6.17). Die Aktualisierung des Vektors übernimmt ebenfalls der Neighbor-Manager, allerdings nicht in der Methode `getRecepientByLoad`. Nach veralteten Nachrichten wird immer dann gesucht, wenn eine neue Nachricht in den Vektor eingefügt wird.

```
chooseNextPeer(method)
1  ...
2  case NEXT_PEER_PING_TIME:
3      resultPeer = peerNeighborManager.getRecepientByLoad()
9  end case
10 ...

getRecepientByLoad()
1  currentMinLoad = MAX_VALUE
2  for each neighbor from neighbors
6      if neighbor.getLoad() < currentMinLoad
7          currentReturnNeighbor = neighbor
8          currentMinLoad = neighbor.getLoad()
9      end if
11 end for
12 return currentReturnNeighbor
```

Abbildung 6.17: Algorithmen für die Peer-Routing-Strategie „Auswahl nach Auslastung der Nachbarn“

Kapitel 7

Evaluierung

7.1 Testbedingungen

Testumgebung
Basis für die Experimente war ein in Java implementierter CAN-Prototyp, der sowohl als Simulator als auch als (verteiltes) CAN-Sytem eingesetzt werden kann [BB04]. Der Prototyp ermöglicht die Simulierung von Netzen aus mehreren tausend Peers, bei Bedarf auch auf einem Ein-Prozessor-Rechner. Dies ist auch notwendig, wenn aussagekräftige Ergebnisse erzielt werden sollen, da reale CAN-basierte Netze in solchen Dimensionen nicht existieren.

Testdaten
Verwendet wurden die TPC-H-Daten[1] für 1MB. I/O-Kosten für Externspeicherzugriffe konnten ignoriert werden, da die Daten für die einzelnen Peers in einfachen Hauptspeicherstrukturen gehalten werden konnten. Diese Einschränkung war akzeptabel, da als wesentlicher Faktor für den Gesamtaufwand die Kommunikation betrachtet wurde. Besonders in weitverteilten Netzen (Internet) ist diese Annahme realistisch.

Testanfragen
Der verwendete Anfragemix bestand im Kern aus drei Klassen mit je drei Anfragen. Innerhalb einer Klasse unterschieden sich die Anfragen hinsichtlich der Anzahl der Joins. Der Unterschied zwischen den Klassen bestand darin, wie stark die Basisrelationen selektiert wurden. Zu diesen neun Anfragen gehörten somit auch solche, die für ein fragmentierte Speicherung besonders ungünstig sind. Dadurch konnten auch Aussagen über das Worst-Case Verhalten der Implementierung gemacht werden.

Untersuchte Kenngrößen
Um quantitative Aussagen und damit Vergleiche zwischen den Testergebnissen machen zu können, wurden folgende Parameter bertrachtet:

- *Anzahl der Hops*
  Ein Hop steht für das Senden einer Nachricht über eine direkte Verbindung zwi-

[1] siehe auch: www.tpc.org

schen zwei Peers. Dieser Parameter ist ein guter Indikator für den Gesamtaufwand einer Anfrage.

- *Anzahl der Nachrichten*
  Anzahl aller erzeugten Nachrichten innerhalb der Verarbeitung einer Anfrag. Abhängig ist diese Größe vor allem von der Verteilung der Daten und der Größe bzw. Anzahl der Zwischenergebnisse.

- *Zeit*
  Durch den CAN-Prototyp kann keine „echte“ Parallelität ermöglicht werden. Aus diesem Grund kann die Systemzeit nicht verwendet werden. Für brauchbare Ergebnisse muss also auch die parallel Zeit simuliert werden.
  Die Zeit wurde als Funktion über die Anzahl der Hops und der Auslastung der Peers realisiert, wobei die Hops stärker gewichtet wurden. Um die Parallelität zu simulieren, wurden nur die Hops und Auslastungen verwendet, die sich durch eine „echte“ Parallelität ergeben würde.

Alle drei Größen wurden nach der vollständigen Verarbeitung einer Anfrage ermittelt, auch wenn in weitverteilten Netzen alle Ergebnistupel bereits vor diesem Zeitpunkt zurückgegeben worden sein können. Der Erhalt aller Ergebnistupel (die in der Praxis üblicherweise nicht vorher bekannt sind) markiert also nicht das Ende einer Anfrageverarbeitung.

7.2 Ausgewählte Tests

7.2.1 Skalierbarkeit

Für Verfahren oder Meachanismen, die in weitverteilten Umgebungen zum Einsatz kommen, gehört die Skalierbarkeit mit zu den wichtigsten Charakteristiken. Die Skalierbarkeit beschreibt, wie sich der Aufwand eines Algorithmus mit der Größe des Netzes ändert. Gerade in CAN-basierten P2P-Netzen, deren große Stärke ein sehr gute Skalierbarkeit ist, werden hohe Ansprüche an die Algortihmen gesetzt.

Abbildung 7.1 zeigt das Verhalten des P2P-Eddies mit den Standardeinstellungen in den drei Netzgrößen von 1.000, 5.000 und 10.000 Peers. Gemessen wurde alle drei Kenngrößen Zeit, Anzahl der Hops und Anzahl der Nachrichten.

Anzahl der Nachrichten. Bei allen drei Netzgrößen ist die Anzahl der erzeugten Nachrichten nahezu identisch. Abhängig ist die Anzahl dabei vor allem von der Verteilung der Daten. Wie man Tabelle 7.1 entnehmen kann, sind die Relationen in allen drei Netzen ähnlich stark verteilt. Damit unterscheidet sich die Anzahl der initialen Nachrichtenpakete und die Verarbeitung von diesen nur gering. Aus diesem Grund werden in etwa die gleiche Anzahl von Nachrichten benötigt.

Anzahl der Hops. Die Summe aller Nachrichtenübertragungen muss in größeren Netzen zwangsläufig ansteigen. Besonders nachteilig wirkt sich der Broadcast des Containers mit den Todo-Listen aus. Aber auch die längeren Wege für das Erreichen von Ziel-Peers einer Neuverteilung besitzen einen großen Einfluss. Je mehr Ergebniszupel eine Anfrage erzeugt, umso stärker bestimmt auch das Zurückschicken der Ergebnisse die Anzahl der Hops.

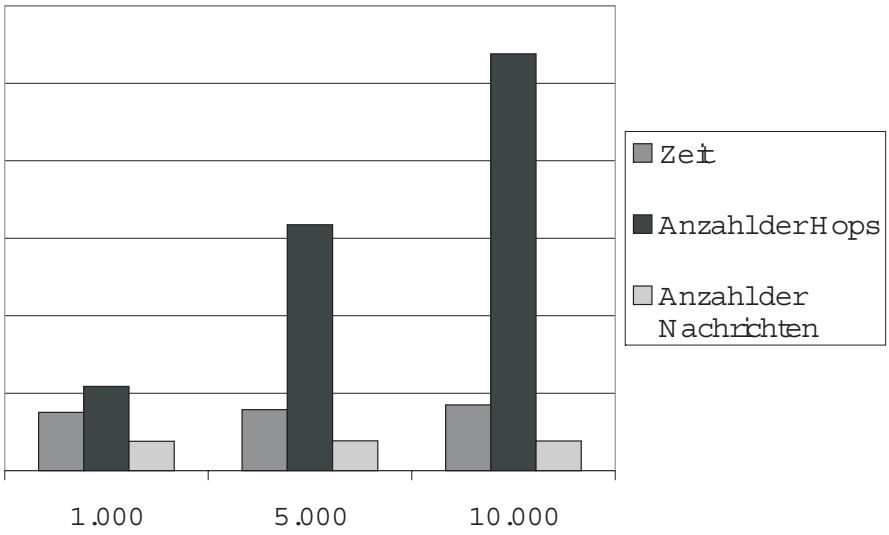

Abbildung 7.1: Skalierbarket des P2P-Eddies

Da die Anzahl der Hops ein gutes Maß für die Auslastung des Netzes darstellt, steigt die absolute Auslastung bei erster Betrachtung deutlich an. Das Verhältnis von Anzahl der Hops zur Anzahl der Peers sinkt aber mit der Größe des Netzes (Abbildung 7.2). Die Auslastung wächst somit nicht linear.

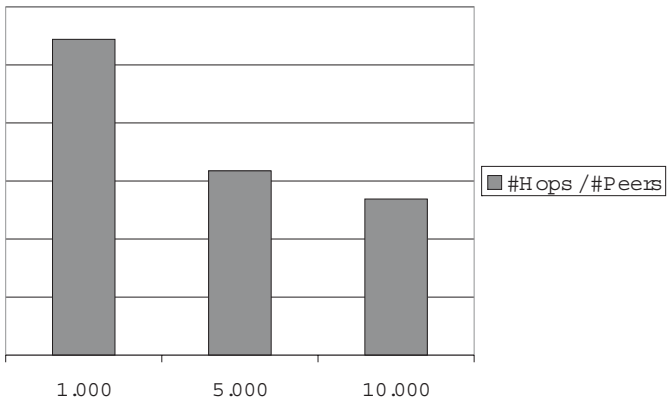

Abbildung 7.2: Entwicklung der Auslastun im Vergleich zur Netzgröße

Zeit. Aus Sicht eines Nutzers oder einer Anwendung ändert sich die Zeit für die Verarbeitung einer Anfrage mit steigender Anzahl der Peers nur gering. Grund hierfür ist zweifelsohne die starke Verteilung der Daten und die hohe Parallelität der Verarbeitung. Der Anstieg der Zeigt liegt in erster Linie an der Zunahme der benötigten Hops, begründet durch die längeren Wege durch das Netz.

7.2.2 Vergleich der Strategien für das Operator-Routing

Die Auswahl der Operatorreihenfolge für Tupel ist *der* Ansatz die für Optimierung in sämtlichen Eddy-Varianten. Der große Unterschied zwischen dem P2P-Eddy und den ur-

sprünglichen Umsetzungen, liegt in der Möglichkeit verteilter Operatoren. Diese Verteilung ist es auch, die die Ergebnisse für die Untersuchung der verschiedenen Strategien für das Operator-Routing prägt.

Abbildung 7.3 zeigt die Ergebnisse für die vier umgesetzten Strategien. Bis auf die Routing-Stratgie behielten die Parameter des Eddy-Operators ihre Standardwerte. Durchgeführt wurden die Tests in einem Netz mit 1000 Peers.

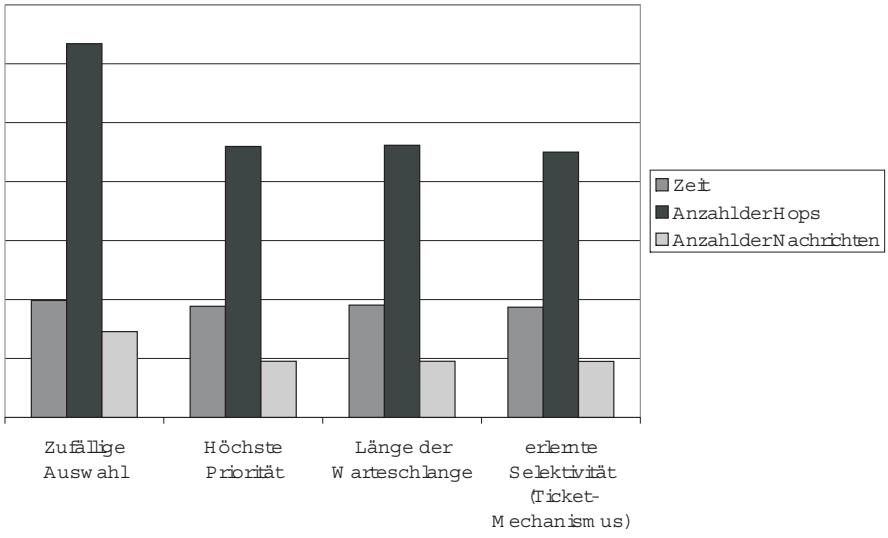

Abbildung 7.3: Vergleich der Strategien für das Operator-Routing

Es fällt auf, dass die Strategien, mit Ausnahme der zufälligen Auswahl, quasi identische Werte für alle drei untersuchten Kenngrößen besitzen. Dies ist auch nicht weiter überraschend, wenn man daran denkt, dass die Operatorpriorität immer dann als Kriterium herangezogen wird, wenn für die Laufzeitstatistiken Warteschlangenlänge und erlernte Selektivität (Ticket-Mechanismus) keine eindeutige Entscheidung getroffen werden kann. Und durch die starke Verteilung der Daten, waren selten alle Operatoren einer Todo-Liste dem jeweiligen bekannt. Es wurde also äußerst oft auf die „Auswahl nach Priorität“ zurückgegriffen.

Wie zu erwarten, fällt die zufällig Auswahl des nächsten Operators ungünstig aus, da besonders ungünstige Operatorreihenfolgen nicht vermieden werden können. Die Nachteile für die starke Verteilung und hohe Parallelität für die Laufzeitstatistiken ist hier allerdings von Vorteil. Wie man dem Diagramm entnehmen kann, ist der Anstieg der Zeit im Vergleich zur Anzahl der Hops und Nachrichten eher gering.

7.2.3 Vergleich der Strategien für das Peer-Routing

Ziel des Peer-Routings ist vor allem eine möglichst gute Verteilung der Last. Wirklich gute Aussagen wären also nur möglich, wenn der CAN-Prototyp auch Last von Peers und somit des gesamten Netzes simulieren könnte. Im aktuellen Stand war dies nicht der Fall. Wie bereits erwähnt, ist das Peer-Routing nicht unabhängig von der Auswahl des nächsten Operators. Richtige Unterschiede sind dadurch mit Anfragen, die vor allem Join-Operatoren enthalten, nicht zu erwarten. Für eine ersten Vergleich der Strategien

wurde also nur auf die Anfrage mit den 10 Selektionen zurückgegriffen. Um untypische Schwankungen etwas abzufangen, wurde für jede Strategie die Anfrage dreimal ausgeführt. Um die Routing-Strategien für eine echte Weiterleitung auch wirklich zum Einsatz kommen zu lassen, wurde der Schwellwert für die maximale Auslastung eines Peers stark verringert. Abbildung 7.4 zeigt das Ergebnis des Tests.

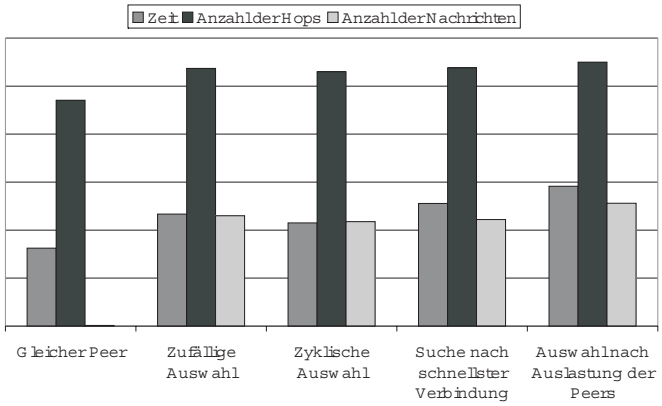

Abbildung 7.4: Vergleich der Strategien für das Peer-Routing

Wie oben angedeutet, bewirkt das Fehlen einer Lastsimulation eine eingeschränkte Aussagekraft des Tests. Die fünf implementierten Strategien liefern recht ähnliche Werte. Alleine die Auswahl „Gleicher Peer“ und „Auswahl nach Auslastung der Nachbarn“ zeigen im gewissen Umfang Abweichungen. Das gute Abschneiden für die Zeit von „Gleicher Peer“ liegt an der höheren Wichtung der Hops gegenüber der Auslastung der Peers. Da sämtliche Nachrichtenpakete auf dem gleichen Peer ausgeführt werden konnten, wurden auch nur drei Nachrichten benötigt, die Broadcast-Nachricht und zwei für das Zurückschicken des Ergebnisses. Die dennoch recht hohe Anhzahl von Hops macht klar, wie aufwendig allein die Verteilung des Todo-Listen-Containers ist.

Die eher schlechten Werte für die Strategie „Auswahl nach Auslastung der Nachbarn“ resultiert daher, dass die eigentliche Verarbeitung der Nachrichtenpakete einen geringen Kommunikationsaufwand besitzt. Es stehen also kaum Informationen über die Nachbarn zur Verfügung. Da der zugehörige Algorithmus so arbeitet, dass er den ersten Nachbarn mit minimaler (abgeschätzter) Auslastung liefert, wird die Last nicht fair verteilt. Da für die wenigsten Nachbarn von Peers Aussagen über deren Auslastung getroffen werden kann, wird sehr oft der gleiche Nachbar zum Ziel-Peer beim Peer-Routing. Die möglichst faire Verteilung der Last und damit auch die Parallelität der Verarbeitung einer Anfrage wird damit untergraben.

Relativ gut konnte die Güte der Verbindungen und damit die Vorzüge der Strategie „Suche nach der schnellsten Verbindung“ getestet werden. Die nötigen Erweiterungen konnten unabhängig von der eigentlichen CAN-Umgebung vorgenommen werden. Wie schon beschrieben, wird die Verbindungsgeschwindigkeit zwischen zwei Peers anhand der Übertragungsdauer ermittelt. Wird eine Nachrichtenpaket nun an einen Nachbarn geschickt, wird die aktuelle Übertragungsdauer mit der durchschnittlichen Dauer verglichen. Gemäß dem Verhältnis wird die Kommunikation zusätzlich gewichtet. Für klare Ergebnisse wurde der Test auf Auslastung der Peers abgeschaltet und die zusätzliche Wichtung besonders hochgesetzt. Das Resultat zeigt Abbildung 7.5.

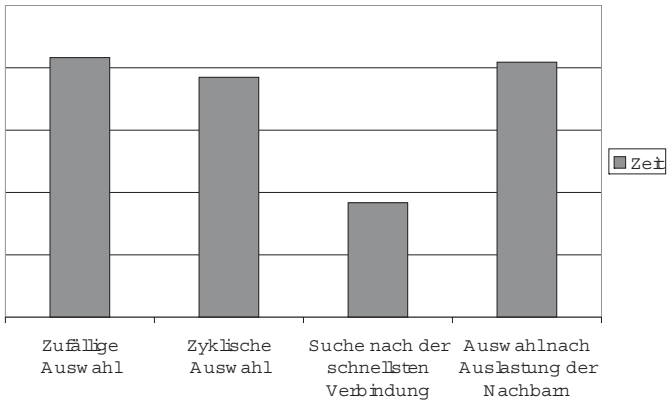

Abbildung 7.5: Test auf „Suche nach der schnellsten Verbinding“

Wie man sehr gut erkennen kann, macht es durchaus Sinn, auf die Parameter des physikalsichen P2P-Netzes einzugehen. Je weitverteilter und ausgelasteter das Netz, umso stärker können die Parameter schwanken und umso brauchbarer können solche Strategien sein.

7.2.4 Suche nach dem „nächsten“ Join

Als Hilfsstrategie für das Peer-Routing soll der Nutzen der Methode `findNearestJoin` gesondert untersucht werden. Ziel dieser Methode war die Minimierung der Hops durch die gezielte Auswahl des Joins, bei dem die durchschnittliche Entfernung zu den Ziel-Peers für die Neuverteilung am geringsten ist.

Bis auf die Verwendung der Methode `findNearestJoin` wurden die Standardwerte für den Eddy-Operator verwendet. Abbildung 7.6 zeigt das Ergbnis.

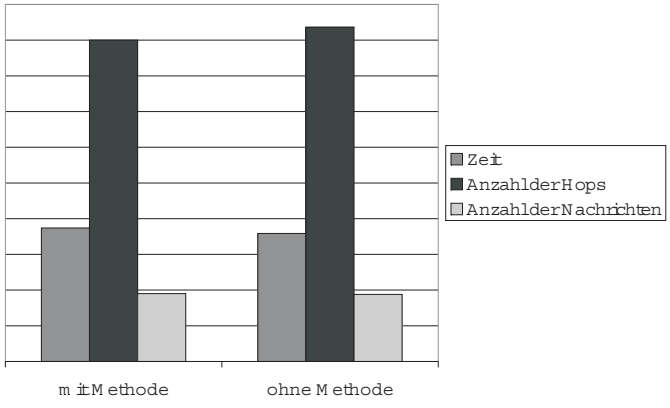

Abbildung 7.6: Test auf Sinn von `findNearestJoin`

Das Diagramm zeigt klar, dass sich die Ergebnisse nur geringfügig unterschieden. Die Anzahl der Hops ist sogar leicht angestiegen. Erklärt kann dieses Verhalten erneut durch die starke Verteilung der Daten. Je verteilter die Daten, umso weniger unterscheiden sich im Mittel die Entfernungen zu den Ziel-Peers.

7.2.5 Zusatzalgorithmen für die Ausführung von Join-Operatoren

Die Algorithmen sind die Vorsortierung der Tupel für das Re-Hashing und das Zusammenfassen gleichartiger Ergbnistupel eines Joins. Hauptziel waren dabei die Verringerung der Nachrichten- und damit auch der Hop-Anzahl. Die Auswirkungen sind dabei umso größer, je mehr Tupel sich in den Eingangsnachrichtenpaketen befinden. Falls im ungünstigen Fall die Pakete nur ein Tupel enthalten, sind die Vorteile der Algorithmen hinfällig.

Abbildung 7.8 vergleicht die Verarbeitung des Anfragemixes mit und ohne dem Packen von Tupeln. Alle anderen Parameter des Eddy-Operators behalten die Standardwerte. Die zwei „teuersten“ Anfragen mussten allerdings herausgenommen werden. Die Anzahl der Nachrichtenpakte war bei diesen so groß, dass der Simulator für die Verarbeitung zu viele Threads benötigt hat.

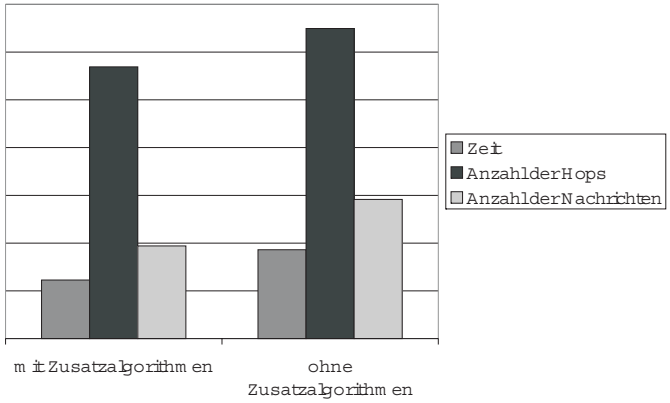

Abbildung 7.7: Test auf Sinn der Zusatzalgorithmen für den Join-Operator

Obwohl die Daten weit im Netz verteilt liegen und die initialen Nachrichtenpakete im Mittel nur wenige Tupel enthalten, ist der Vorteil, der sich durch die Zusatzalgorithmen ergibt, offensichtlich. Alle drei untersuchten Kenngrößen steigen beim Verzicht auf die Algorithmen merklich an.

7.2.6 Einfluss von globalen Wissen

Simuliert wurde das globale Wissen durch eine Klasse mit statischen Attributen und Methoden. Somit konnte jeder Peer auf diese Daten zugreifen. Damit sollte untersucht werden, welche Auswirkungen die verteilten Laufzeitstatistiken haben.

In beiden Testläufen arbeitete der Eddy-Operator mit den Standardwerten für seine Parameter. Lediglich im zweiten Lauf wurde das globale Wissen hinzugeschaltet. Graphisch präsentiert wird das Ergebnis in Abbildung 7.8.

Wie man sieht, ist das Ergebnis bei der Verwendung von globalem Wissen merklich schlechter, besonders bei Anzahl der Hops und Anzahl der Nachrichten. Die Zeit bleibt durch die hohe Parallelität weitgehend unberührt. Der Grund für dieses Resultat liegt im Ticket-Mechanismus begründet. Durch das globale Wissen sind kurz nach dem Start der Anfrage alle Operatoren bekannt, so dass ab diesem Zeitpunkt immer eine eindeutige Entscheidung aufgrund der Tickets getroffen werden kann. Dabei kann es durchaus passieren,

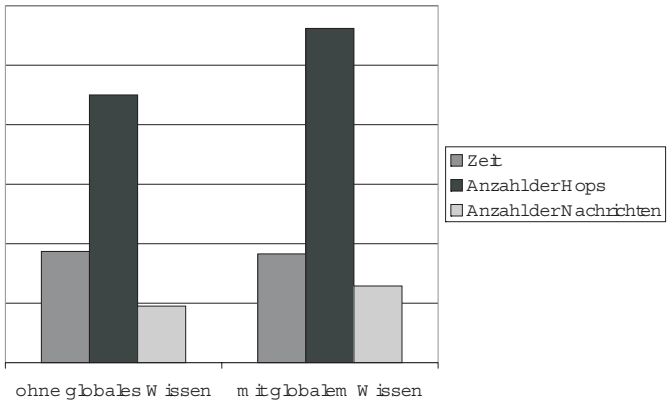

Abbildung 7.8: Test auf Einfluss von globalen Wissen

dass ein Join-Operator eine geringere Selektivität besitzt als eine Selektion. Dadurch muss für mehr Tupel der Join durchgeführt werden, was einen Anstieg der Hop- und Nachrichtenanzahl begünstigt. Offensichtlich ist der Ticket-Mechanismus nicht völlig ausgereift. So kann z.B. für Joins die Selektivität nur abgeschätzt werden, was an der *Symmetric Hash Join*-Implementierung liegt. Desweiteren sollten die erlernten Selektivitäten der Operatoren unterschiedlich gewichtet werden, um die Komplexität der Algorithmen für die Planoperatoren miteinzubeziehen.

7.3 Test auf Auslastung

Bei diesem Experiment wurde untersucht, ob und wie sinnvoll es ist, ein Nachrichtenpaket unverarbeitet weiterzuschicken, falls ein Peer überlastet ist. Ein Anstieg bei der Anzahl von Hops ergibt sich selbstverständlich automatisch. Da für die Berechnung der Zeit auch die Last der Peers miteinfließt, sollte sich die Vermeidung stark ausgelasteter Peers positiv auf die Zeit auswirken.

Natürlich ist auch der Aufwand (lokaler Aufwand; nicht die Übertragungskosten) für das Weiterleiten von Paketen nicht unabhängig von der Auslastung eines Peers. Allerdings ist die Verarbeitung von Paketen im Mittel deutlich teurer als die pure Weiterleitung. Innerhalb diesen Tests wurde ein Verhältnis von 20:1 angenommen.

Die Auslastung der Peers wurde durch zufällige Werte aus einem Intervall $I = [0..n]$ simuliert. Dieses Vorgehen ist für diesen Test völlig ausreichend und spiegelt zusätzlich die nicht vorhersagbare Dynamik in solchen Systemen wider. Eine wird als ausgelastet angesehen, wenn für seine aktuelle Last $x \in I$ gilt: $x > 0.9 * n$. Sobald ein Peer ein Nachrichtenpaket verarbeitet, wurde der Zeitzähler des Paketes um x erhöht, bei einer direkten Weiterleitung um $0.05 * n$. Der Hop-Zähhler im Nachrichtenpaket sorgt dafür, dass ein Paket bei hoher Auslastung vieler Peers zu lange unverarbeitet verschickt wird (siehe Abschnitt 5.3.1).

Als Vergleichskonfiguration mit den Standardwerten für den Eddy-Operator, wurde die Strategie „Gleicher Peer “ für das Peer-Routing verwendet. Siehe dazu auch den Entscheidungsbaum aus Abbildung 4.7. Um aussagekräftige Ergebnisse zu erhalten, wurden Anfragen ohne Joins und mit vielen Selektionen pro Basisrelation verwendet. Diese An-

fragen erlauben eine häufige freie Wahl von Ziel-Peers für das Peer-Routing. Das Ergebnis des Tests zeigt Abbilung 7.9.

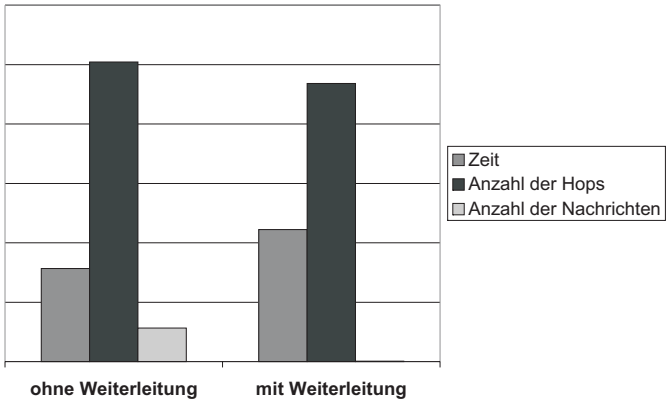

Abbildung 7.9: Test auf Auslastung

Die Anzahl der Hops geht ohne ein zusätzliches Verschicken von Tupeln erwartungsgemäß nach unten. Genaugenommen erzeugt dieses Vorgehen das Minimum an benötigten Hops für die Verarbeitung einer Anfrage.

Da alle Selektionen direkt auf den Peers ausgeführt werden, auf denen Tupel der entsprechenden Basisrelationen liegen, beschränken sich die benötigten Nachrichten auf die Broadcast-Nachricht und die Nachrichten für die Ergebnistupel. Erst durch die Weiterleitung von Nachrichten aufgrund überlasteter Peers steigt die Nachrichtenanzahl an.

Vorteilhaft wird der Test auf Auslastung beim Parameter Zeit. Durch die Vermeidung von einer Verarbeitung von Nachrichtenpaketen auf stark ausgelasteten Peers, wird der Zeitzähler der Pakete nie um maximale Werte erhöht. Für einen Nutzer oder eine Anwendung ist die Weiterleitung von Nachrichtenpaketen weg von überlasteten Peers somit vorteilhaft.

Der Unterschied zwischen beiden Vorgehen ist in erster Linie abhängig von der Anfrage, da Operator- und Peer-Routing nicht orthogonal zueinander sind. Enthält eine Anfrage beispielsweise nur Joins, ist eine freie Wahl der Ziel-Peers und damit eine zusätzliche Verteilung der Last nicht möglich.

Ob ein Test auf Auslastung sinnvoll ist, wird auch durch das Verhältnis zwischen dem Aufwand für das Verschicken und der Verarbeitung von Nachrichtenpaketen bestimmt. Sind Hops sehr „teuer“, kann die Zeitersparnis aus der geringer Auslastung der Peers durchaus von den Kommunikationskosten verdrängt werden. Andererseits kann durch den Parameter `maxHops` des Eddy-Operators (siehe Abschnitt 5.6) die maximale Anzahl zusätzlicher Übertragungen nach oben begrenzt werden. Außerdem werden Pakete bei freier Peer-Wahl immer nur zu direkten Nachbarn geschickt, was somit immer genau einem Hop entspricht. Darum relativiert sich vor allem in großen Netzen der extra Kommunikationsaufwand, erst recht, wenn die Anfrage auch Joins enthält.

7.4 Vergleich mit einem zentralisierten Verfahren

Obwohl mit dem P2P-Eddy gerade die Gefahr von Flaschenhalseffekten aufgrund dedizierter Peers, wurde der P2P-Eddy nachträglich um einen Modus erweitert, der in etwa einen zentralisierten Eddy simuliert. Dazu wurden folgende Erweiterungen umgesetzt:

- alle initial erzeugten Nachrichtenpakete werden zum Peer geschickt, auf dem die Anfrage gestartet wurde
- alle Operatoren werden auf diesem Peer ausgeführt; somit ist dieser Peer auch das Ziel sämtlicher Neuverteilungen
- das globale Wissen wird verwendet

Bei dieser einfachen Umsetzung werden also alle Tupel auf den Initiator-Peer geholt und von diesem verarbeitet. Verglichen wurde die zentralisierte Variante mit den Standardwerten für den Eddy-Operator des P2P-Eddies. Abbildung 7.10 zeigt das Ergebnis.

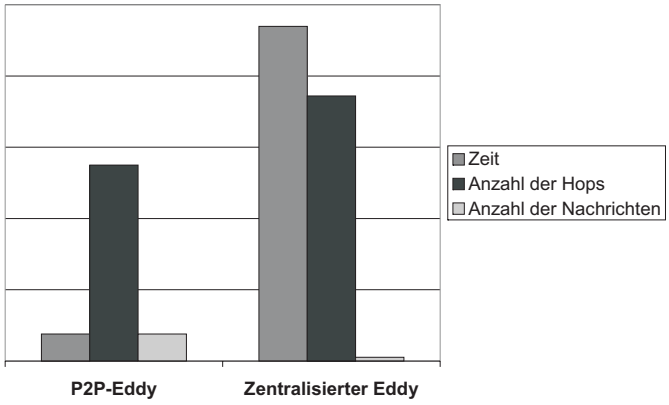

Abbildung 7.10: Vergleich des P2P-Eddies mit einer zentralisierten Verarbeitung

Auffallend ist der deutliche Anstieg der Zeit. Bedingt wird dies durch den Verlust der Parallelität bei der Verarbeitung der Tupel.

Auch die Anzahl der Hops steigt bei der zentralisierten Variante im Mittel an, da zunächst immer alle Tupel der betroffenen Basisrelationen zum Initiator-Peer werden. Selektionen auf den Tupeln können vorher nicht ausgenutzt werden. Erst wenn, bedingt durch die Anfrage, vollständige Relationen verbunden werden müssen, sinkt die Anzahl der benötigten Hops im Vergleich zum P2P-Eddy. Sobald sich alle Tupel einmal auf dem Start-Peer befinden, bedarf es keiner weiteren Kommunikation.

Die realisierte Umsetzung eines zentralisierten Eddies ist allerdings nicht optimal. Sinnvoller wäre es z.B. die Abarbeitung der Planoperatoren auf andere Knoten zu delegieren und lediglich die Aufgabe des Eddy-Operators vom Initiator-Peer durchführen zu lassen. Damit sinkt zwar die Auslastung dieses Peers aber gleichzeitig steigt wieder der Kommunikationsaufwand. Ausserdem müssten die Fragen beantwortet werden, welcher Peer welchen Planoperator ausführt.

Kapitel 8

Zusammenfassung und Ausblick

Mit dem P2P-Eddy wurden die Ideen der urprünglichen Eddy-Umsetzungen auf die Charakteristika von P2P-Systeme angepasst. Kern ist auch hier die dynamsiche Auswahl der Operatorreihenfolge für die Tupel. Die grundlegende Neuerung ist die Unterstützung verteilter Operatoren, erreicht durch die Verbindung der Tupel mit der zugehörigen Verarbeitungsvorschrift. Dieses Konzept ist natürlich nicht auf den Einsatz in P2P-Netzen beschränkt, sondern kann ohne großen Aufwand auch für andere verteilte Umgebungen umgesetzt werden.

Die dadurch gewonnene Flexibilität für die Auswahl von Rechnerknoten bringt dabei einige Vorteile mit sich. So existieren z.B. keine dedizierten Rechnerknoten, deren Ausfall (durch Überlastung, Angriff,...) automatisch einen vollständigen Abbruch der Anfrage zur Folge hätte. Weiterhin kann eine wesentlich bessere Lastverteilung erreicht werden. Vor allem wird aber so ein Maximum an Parallelität ermöglicht, was sowohl der Skalierbarkeit als auch der Robustheit des P2P-Eddies zu Gute kommt, den Hauptkriterien für den Einsatz in P2P-Netzen. Die teilweise freie Auswahl von Peers, erlaubt dem P2P-Eddy eine noch „agressivere“ Adaption an die Systemumgebung als die ersten Eddy-Varianten.

Im Mittelpunkt der Adaptivität stehen die Strategien, welche die Operatorreihenfolge und die Peer-Auswahl festlegen. Für beiden Fragestellungen wurden verschiedene Strategien umgesetzt, die mit der Steigerung der Effizienz ein gemeinsames Ziel verfolgen, aber verschiedene Wege gehen. Sie unterscheiden sich dabei in Verwendung der rückgekoppelten Systemparameter und ggf. der Bestimmung bzw. Pflege zugehöriger Statistiken.

Die Verteilung von Operatoren und der Verzicht auf eine zentrale Koordination haben auch ihre Nachteile. Ganz allgemein wird die hohe Flexibilität und Dynamik mit einem nicht unerheblichen Mehraufwand erkauft. Dieser setzt sich in erster Linie aus Aktualisierung benötigter Parameter und aus den Kosten für die Entscheidungsfindungen zusammen.

Mit der Verteilung von Operatoren werden auch die zugehörigen Laufzeitstatistiken verteilt. Je verteilter ein Operator, umso verteilter sind auch dessen Statistiken und umso geringer ist somit die Aussagekraft der Kenngrößen für die lokalen Routing-Entscheidungen. Dies betrifft vor allem das Operator-Routing. Dadurch, dass auch Routing-Strategien, die auch ohne die lokalen Lauftzeitstatistiken gute Ergebnisse erzielen, existieren, relativiert sich dieses Problem.

Die Möglichkeiten des P2P-Eddies wurden noch lange nicht ausgeschöpft. So feh-

len für einen vollständigen Anfrageprozessor noch die nötigen Planoperatoren. Dazu zählen Operatoren wie Aggregation, Gruppierung oder Sortierung. Auch die ausschließliche Verwendung von Equi-Joins schränkt den Leistungsumfang der Anfrageverarbeitung ein. Hinsichtlich der Flexibilität und Effizienz sind auch verschiedene Implementierungen vor allem für komplexe oder blockierende Operatoren denkbar (Stichwort: adaptive dynamische Operatoren).

Das größte Potential für eine weitere Optimierung steckt zweifellos in den Routing-Strategien. Durch die verteilten Operatoren sind neue Strategien für das Operator-Routing interessant, die auf verteilte Laufzeitstatistiken verzichten. Denkbar wäre beispielsweise eine feste Priorität oder Selektiviät für jeden einzelnen Operator im Anfragebaum. Welche zusätzlichen Erweiterung (Stichwort: verteilter Datenbankkatalog) dafür nötig wären, steht auf einem anderen Blatt. Wie die Evaluierung gezeigt hat, sind aber auch die bestehenden Routing-Strategien noch nicht optimal.

Unzureichend evaluiert wurde das Peer-Routing mit dem Ziel einer möglichst effizienten Lastverteilung. Für reprodizierbare und aussagekräftige Ergebnisse müsste der CAN-Prototyp sinnvollerweise erweitert werden, um auch Last im Netz bzw. auf die Peers zu simulieren, vorzugsweise unabhängig von der Anfrageverarbeitung.

Alles in allem wurde mit dem P2P-Eddy ein wichtiger Schritt in Richtung Anfrageverarbeitung in massiv verteilten Umgebungen gemacht. Obwohl die maximale Effizienz sicher noch nicht erreicht wurde, zeigen die hohe Parallelität, Dynamik und Flexibilität, dass der P2P-Eddy durchaus auf dem richtigen Weg ist. Bezüglich der Aggressivität der Adaptivität, hat der P2P-Eddy die Meßlatte für adaptive Anfrageverarbeitungen noch einmal höher gelegt. Und das Potential ist noch lange nicht ausgeschöpft.

Eidestattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben. Die wörtlich oder inhaltlich entnommenen Stellen wurden als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Ort, Datum **Christian von der Weth**

Literaturverzeichnis

[ADAT+99] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, Joseph M. Hellerstein David E. Culler, David Patterson, and Katherine Yelick. Cluster I/O with River: Making the Fast Case Common. In *Sixth Workshop on I/O in Parallel and Distributed Systems (IOPADS '99), Atlanta, USA*, pages 10–22, may 1999.

[AFTU96] Laurent Amsaleg, Michael J. Franklin, Anthony Tomasic, and Tolga Urhan. Scrambling Query Plans to Cope With Unexpected Delays. In *4th International Conference on Parallel and Distributed Information Systems (PDIS), Miami Beach, USA*, december 1996.

[AH00] Ron Avnur and Joseph M. Hellerstein. Tuple Routing Strategies for Distributed Eddies. In *Proceedings of the ACM SIGMOD, Dallas, Texas, USA*, may 2000.

[AZ97] Gennady Antoshenkov and Mohammed Ziauddin. Query Processing and Optimization in Oracle Rdb. In *VLDB Journal*, pages 5(4):229–237, 1997.

[BB04] Erik Buchmann and Klemens Boehm. How to Run Experiments with Large Peer-to-Peer Data Structures. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium, Santa Fe, USA*, 2004.

[BDF+97] Daniel Barbara, William DuMouchel, Christos Faloustsos, Peter J. Haas, Joseph M. Hellerstein, Yannis E. Ioannidis, H. V. Jagadish, Theodore Johnson, Raymond T. Ng, Viswanath Poosala, Kenneth A. Ross, and Kenneth C. Sevcik. The New Jersey Data Reduction Report. In *IEEE Data Engineering Bulletin*, 1997.

[Ber98] Glenn Berg. *MCSE Network Essentials*. New Riders Publishing, Indianapolis, second edition, 1998. ISBN: 1-56205-919-X.

[CR94] Chung-Min Chen and Nick Roussopoulos. Adaptive selectivity estimation using query feedback. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 24–27, 1994.

[EN02] Ramez Elmasri and Shamkant B. Navathe. *Grundlagen von Datenbanksystemen*. Pearson Education Deutschland GmbH, München, third edition, 2002. ISBN: 3-8273-7021-3.

[GPFS02] Anastasios Gounaris, Norman W. Paton, Alvar A.A. Fernandes, and Rizos Sakkellariou. Adaptive Query Processing: A Survey. In *19th British National Conference on Databases, BNCOD 19, Sheffield, UK*, july 2002.

[HFC+00] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekeran, Amol Deshpande, Kris Hildrum, Sam Madden, Vijayshankar Raman, and Mehul A. Shah. Adaptive Query Processing: Technology in Evolution. In *IEEE Data Engineering Bulletin*, june 2000.

[HH99] Peter J. Haas and Joseph M. Hellerstein. Ripple Joins for Online Aggregation. In *Proceedings ACM SIGMOD International Conference on Management of Data, Tucson*, 1999.

[HS00] Andreas Heuer and Gunter Saake. *Datenbanken: Konzepte und Sprachen*. mitp-Verlag Bonn, second edition, 2000. ISBN: 3-8266-0619-1.

[KE99] Alfons Kemper and Andre Eickler. *Datenbanksysteme*. Oldenbourg Verlag München Wien, third edition, 1999. ISBN: 3-486-25053-1.

[ONP+96] F. Ozcan, S. Nural, P.Koksal, C. Evrendilek, and A. Dogac. Dyamic query optimization on a distributed object management platform. In *Conference on Information and Knowledge Management, Baltimore, Maryland, USA*, 1996.

[PCL93] HweeHwa Pang, Michael J. Carey, and Miron Livney. Memory-adaptive external sorting. In *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, pages 618–629, 1993.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richsrd Karp, and Scott Shenker. A Scaleable Content-Adressable Network. In *Proceedings of the SIGCOMM, San Diego, California, USA*, aug 2001.

[RvdWSB04] Philipp Rösch, Christian von der Weth, Kai Uwe Sattler, and Erik Buchmann. Verteilte Anfrageverarbeitung in DHT-basierten P2P-Systemen. In *To appear.*, october 2004.

[SAC+79] Patricia G. Selinger, Morton M. Astrahen, Donald D. Chamberlain, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1*, pages 23–34, 1979.

[Spo98] Mark Sportack. *Network Essentials*. Sams Publishing, Indianapolis, first edition, 1998. ISBN: 0-672-31210-7.

[TD03] Feng Tian and David J. DeWitt. Tuple Routing Strategies for Distributed Eddies. In *Proceedings of the 29th VLDB Conference, Berlin, Germany*, 2003.

[Ull88] Jeffrey D. Ullman. *Principles of databse and Knowledge-Base Systems, Volume 1: Classical Database Systems*, volume One. Computer Science Press Rockville, first edition, 1988. ISBN: 0-7167-8158-1.

[Vos00] Gottfried Vossen. *Datenmodelle, Datenbanksprachen und Datenbankmanagementsysteme*. Oldenbourg Verlag München Wien, forth edition, 2000. ISBN: 3-486-25339-5.