Generische Joinverarbeitung am Beispiel des Similarity Join

Dissertation zur Erlangung des
Doktorgrades der Naturwissenschaften
(Dr. rer. nat.)

dem Fachbereich Mathematik und Informatik
der Philipps-Universität Marburg vorgelegt von

Jens-Peter Dittrich
aus Alfeld

Marburg/Lahn
Mai 2002

Vom Fachbereich Mathematik und Informatik
der Philipps-Universität Marburg als Dissertation am 24. Juni 2002
angenommen.

Erstgutachter:	Prof. Dr. Bernhard Seeger
Zweitgutachter:	Prof. Dr. Peter Widmayer (ETH Zürich)

Tag der mündlichen Prüfung am 2. Juli 2002.

Zusammenfassung

Der Ähnlichkeitsverbund (engl.: Similarity Join) ist eine bedeutende Grundoperation für zahlreiche Data Mining Anwendungen wie Clustering und Outlier Detection. Ein Similarity Join zweier Mengen hochdimensionaler Punkte R und S berechnet alle Tupel (r, s), die für eine gegebene Distanzfunktion einen Abstand ε nicht überschreiten.

Alle bekannten effizienten Similarity Join Verfahren müssen die Eingaberelationen vor der eigentlichen Joinberechnung zunächst sortieren. Während dieser Sortieroperation produzieren diese Joinverfahren keine Ergebnistupel und *blockieren* den Datenfluß innerhalb des Anfragebaums. Diese Verfahren sind somit ungeeignet für wichtige Anwendungen wie z.B. online Data Mining.

Die vorliegende Arbeit behandelt Algorithmen für Similarity Joins, die den Datenfluß nicht blockieren.

Ausgehend von den Verbundoperationen, die von Orenstein und Koudas & Sevcik vorgeschlagen wurden, werden zunächst die Auswirkungen der kontrollierten Datenreplikation untersucht. Wir präsentieren ein theoretisches Modell für Replikation in beliebig dimensionalen Räumen und untersuchen die Auswirkung auf die Anzahl der Distanzberechnungen. Wir schlagen einen neuen generischen Algorithmus namens *Generic External Space Sweep (GESS)* vor, der die Verfahren ORE, MSJ und viele andere als Spezialfälle einschließt. GESS ersetzt das I/O-Modell von MSJ und erweitert das Verfahren um einen generischen Replikations- und Partitionierungsalgorithmus. Wir nutzen kontrollierte Replikation, um die Anzahl der teuren Distanzberechnungen zu reduzieren. Duplikate in der Antwortmenge werden durch eine effiziente online Duplikateliminierung erkannt.

Unser analytisches Modell sagt eine hohe Reduktion (mehrere Größenordnungen) der Distanzberechnungen im Vergleich zu MSJ voraus. Ferner zeigen unsere Ergebnisse, daß der Hauptspeicherbedarf von GESS im Vergleich zu MSJ um mehrere Größenordnungen geringer ist. Dies mag zunächst überraschen, da dies durch die Replikation der Daten erzielt wird. Die I/O-Kosten unseres Algorithmus sind um Faktor 2 und mehr besser (gegenüber den klassischen Verfahren). Unsere analytischen Ergebnisse werden bestätigt durch eine Serie von Simulationen und Experimenten mit synthetischen und realen hochdimensionalen Datensätzen.

Ein weiteres fundamentales Problem bisheriger Verfahren ist das Blockieren des Datenflusses durch die Joinberechnung. Ausgehend vom Similarity Join präsentieren wir zunächst

ein generisches Framework für alle uns bekannten sortierbasierten Joinalgorithmen. Wir präsentieren *Progressive Merge Join (PMJ)*, das nach unserem Wissen erste nichtblockierende Similarity Join Verfahren. Auch wenn PMJ im Kontext des Similarity Joins vorgestellt wird, ist der Algorithmus nicht hierauf beschränkt und kann für alle sortierbasierten Joinverfahren genutzt werden, so z.B. zur Berechnung des Equi Joins, des raumbezogenen Verbunds (Spatial Join) und des zeitbezogenen Verbunds (Temporal Joins).

Die wesentliche Idee von PMJ ist es, beide Datenmengen — beginnend mit der Run-Erzeugung — gleichzeitig zu sortieren und fortlaufend Ergebnistupel des Joins von den im Hauptspeicher vorhandenen Teilmengen zu berechnen. Ergebnistupel werden direkt weitergereicht und können für nachfolgende Berechnungen bzw. Visualisierungen genutzt werden. Die Kosten von PMJ entsprechen asymptotisch denen der blockierenden Verfahren. In einer Reihe von Experimenten für relationale und räumliche Verbundoperationen sowie Similarity Joins zeigen wir, daß eine beträchtliche Anzahl von Ergebnistupeln produziert wird, noch bevor die Sortierphase der konventionellen Techniken beendet ist. Unsere Ergebnisse zeigen nur einen geringen Anstieg der absoluten Laufzeit der Algorithmen im Vergleich zu den korrespondierenden blockierenden Techniken.

Die in dieser Arbeit vorgestellten Algorithmen sind als Teil der objektrelationalen Cursoralgebra *eXtensible and fleXible Library (XXL)* implementiert. XXL ist eine plattformunabhängige Javabibliothek von Frameworks zur Anfrageverarbeitung. Wir diskutieren die wichtigsten Designprinzipien, Konzepte und Schnittstellen und beschreiben die Integration der in dieser Arbeit vorgestellten Algorithmen.

Danksagungen

Mein herzlicher Dank gilt Herrn Prof. Dr. Bernhard Seeger für die hervorragende fachliche und menschliche Betreuung dieser Arbeit. Seine fachlichen Anregungen und die zahlreichen gemeinsamen Diskussionen haben viel zum Gelingen der vorliegenden Arbeit beigetragen.

Desweiteren möchte ich mich bei Herrn Prof. Dr. Peter Widmayer (ETH Zürich) für die Zweitbegutachtung und die hilfreichen Anmerkungen zu dieser Arbeit bedanken.

Prof. Dr. Peter Widmayer und Dr. David Scot Taylor (ETH Zürich, UC L.A.) danke ich ferner für die zahlreichen spannenden Diskussionen. Insbesondere die interessante gemeinsame theoretische Arbeit über nichtblockierende Joinalgorithmen war sehr inspirierend.

Prof. Dr. Thomas Schwentick (Uni Marburg) und Prof. Dr. Volkmar Welker (Uni Marburg) danke ich für interessante Diskussionen zum Thema Komplexität von Joinalgorithmen.

Dr. Bernhard Braunmüller (LMU München), Prof. Dr. Christian Böhm (LMU München, UMI Tirol) sowie Prof. Dr. Scott T. Leutenegger (University of Denver) danke ich für das Bereitstellen von Datensätzen.

Evelyn Wander und MA Martin Esters danke ich für das Korrekturlesen der vorliegenden Arbeit.

Alexander Markowetz danke ich für die gemeinsame Zeit mit viel Spaß und anregenden fachlichen Diskussionen und gemeinsamen Projekten.

Bei Evelyn Wander, Wido Wirsam, Brigitte Klein, Michael Baldeweg und Heike Luu sowie meinen Eltern möchte ich mich ganz besonders für die persönliche Unterstützung in den vergangenen drei Jahren bedanken.

Meinem Sohn Laurin danke ich dafür, daß er es immer wieder erfolgreich geschafft hat, seinen Papa gedanklich von der Informatik abzulenken.

Marburg, im September 2002

Inhaltsverzeichnis

Zusammenfassung i
Danksagungen iii

1 Einleitung **1**

1.1 Motivation 1
1.1.1 Beispiel: Clustering 2
1.1.2 Beispiel: Outlier Detection 3
1.2 I/O-Modell und Notation 3
1.3 Namensgebung und Begriffe 4
1.4 Grundlegende Definitionen 4
1.4.1 Verbund 4
1.4.2 Ähnlichkeitsverbund 5
1.4.3 Mehrschrittverarbeitung 5
1.4.4 Die Auswirkung der Dimension 6
1.5 Problemstellung und Voraussetzungen 7
1.5.1 Datenreplikation in hochdimensionalen Datenräumen 7
1.5.2 Joinverfahren blockieren den Datenfluß 7
1.5.3 Zusätzliche Voraussetzungen 8
1.6 Gliederung dieser Arbeit 8

2 Joinverarbeitung in Datenbanken **10**

2.1 Prinzipielle Vorgehensweisen 10
2.1.1 Nested-Loops Join 10
2.1.2 Indexed Nested-Loops Join 11
2.1.3 Hashjoin 11

2.1.4 Sort-Merge Join 12
2.1.5 Schachteln von Algorithmen 13
2.2 Literatur 14
2.2.1 Spatial Join 14
2.2.2 Similarity Join 16
2.2.3 Nichtblockierende Joinalgorithmen 17

3 GESS **20**

3.1 Generischer Rahmen 21
3.1.1 Orensteins Algorithmus 22
3.1.2 Der MSJ-Algorithmus 23
3.2 Generic External Space Sweep 24
3.2.1 Der Replikationsalgorithmus 25
3.2.2 Warum wir keine Ebenendateien brauchen 27
3.2.3 RPM: Die Referenzpunktmethode 28
3.3 CPU-Kosten Analyse 29
3.3.1 Analytisches Modell von MSJ 30
3.3.2 Analytisches Modell von GESS 31
3.4 I/O-Kosten Analyse 33
3.4.1 I/O-Kosten von MSJ 33
3.4.2 I/O-Kosten von GESS 34
3.5 Implementierungsaspekte 37
3.6 Experimente 39
3.6.1 Simulationen 39
3.6.2 Implementierungen 43
3.6.3 Die Auswirkung der Aufteilungsstrategie 44
3.6.4 Replikationsraten 53
3.6.5 Overhead der Referenzpunktmethode 55
3.7 Experimente mit der Fraktalen Dimension 57
3.7.1 Die fraktale Dimension 57
3.7.2 Erweiterung des Kostenmodells 59
3.7.3 Diskussion 63

3.8 Zusammenfassung 64

4 PMJ **66**

4.1 Progressive Merge Join 67

4.1.1 Join während der Run-Erzeugung 68

4.1.2 Join während des Mischens 72

4.1.3 Erweiterungsmöglichkeiten 74

4.2 Vollständigkeit 76

4.3 Eindeutigkeit 79

4.4 Realisierung der Eindeutigkeit 80

4.4.1 Duplikatvermeidung 80

4.4.2 Duplikateliminierung 80

4.5 Laufzeit 82

4.6 Experimente 84

4.6.1 Vergleich von Joinalgorithmen 84

4.6.2 Die Auswirkung der Hauptspeichergröße 91

4.6.3 Selektivitätsschätzung 94

4.7 Zusammenfassung 96

5 XXL **98**

5.1 Designprinzipien 98

5.2 Aufbau der Bibliothek 100

5.2.1 Iteratoren und Cursor 101

5.2.2 Funktionen und Prädikate 102

5.2.3 Queues 103

5.3 Integration der Joinalgorithmen 105

5.3.1 xxl.spatial 105

5.3.2 Sort-Merge Join 106

5.3.3 Orenstein 108

5.3.4 GESS 112

5.3.5 PMJ 118

5.4 Zusammenfassung 119

6 Zusammenfassung und Ausblick **120**

6.1 Beiträge dieser Arbeit 120

6.1.1 GESS 120

6.1.2 PMJ 121

6.1.3 XXL 122

6.2 Ausblick 122

Literaturverzeichnis **125**

Abbildungsverzeichnis **132**

Tabellenverzeichnis **136**

Abkürzungsverzeichnis **137**

Index **140**

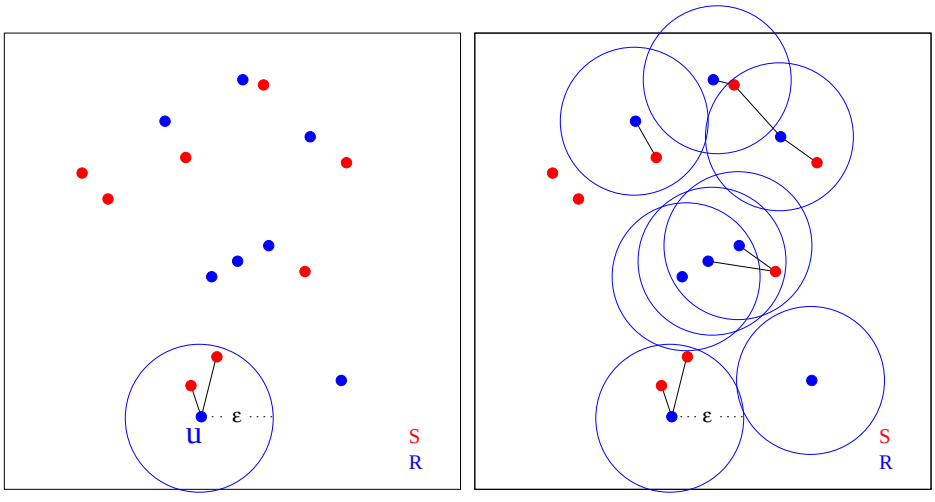
Lebenslauf **143**

Kapitel 1

Einleitung

1.1 Motivation

Ein *Ähnlichkeitsverbund (engl.: Similarity Join)* zweier Mengen hochdimensionaler Punkte R und S berechnet alle Tupel (u, v) $(u \in R,\ v \in S)$, die für eine gegebene Distanzfunktion einen Abstand ε nicht überschreiten. Der Similarity Join ist damit eine Verallgemeinerung der *Kugelanfrage*, bei welcher die Anfragemenge R aus nur einem Element besteht. Abbildung 1.1a zeigt ein Beispiel für eine Kugelanfrage eines Punktes $u \in R$ an die Menge S.



(a) Kugelanfrage
$K_\varepsilon(u, S) = \{v \mid \mathrm{d}(u, v) \leq \varepsilon, v \in S\}$

(b) Similarity Join
$R \bowtie_\varepsilon S = \{(u, \mathrm{K}_\varepsilon(u, S)) \,|u \in R\}$

Abbildung 1.1: Der Similarity Join als Massenanfrage

Abbildung 1.1b zeigt den Similarity Join für dasselbe Beispiel. Die mit einer durchgezogenen Line verbundenen Punkte markieren die Ergebnistupel.

Ein Similarity Join kann somit im Prinzip durch eine Menge einzelner Kugelanfragen berechnet werden. Für große Anfragemengen ist dieses Verfahren allerdings nicht effizient und es wird beim Similarity Join versucht, die einzelnen Anfragen intern geschickt zu bündeln, um — z.B. unter Ausnutzung von Lokalitätsprinzipien — den Gesamtaufwand der Berechnung zu reduzieren.

Wir geben im folgenden zunächst einige wichtige Beispiele für die Anwendung von Similarity Joins, im Anschluß wird der Similarity Join formal definiert.

1.1.1 Beispiel: Clustering

Ein Cluster kennzeichnet eine Teilmenge ähnlicher Datenpunkte, die sich von den Punkten anderer Cluster unterscheiden. Die Berechnung von *Clustern (Clustering)* stellt eine der wichtigsten Data Mining Operationen dar.

Beispielsweise kann ein Mobilfunkbetreiber durch Clustering Kundenprofile ermitteln.

Die Gruppierung der Daten erfolgt z.B. beim dichtebasierten Clustering [EKSX96] bezüglich eines gegebenen Abstandsmaßes. Ein Punkt r heißt zugehörig zu einem Cluster, wenn mindestens j Punkte in einem Abstand kleiner ε existieren. In Abhängigkeit von ε entstehen Gruppen (Cluster) ‚ähnlicher Punkte'. $||\mathrm{K}_\varepsilon(x, A)||$ bezeichne die Anzahl der Datenpunkte aus A, die in der Kugel mit Mittelpunkt x und Radius ε liegen.

Cluster-Algorithmen selektieren aus einer gegebenen Menge von Punkten R alle diejenigen Punkte, für die die Menge R in einem Abstand von höchstens ε mindestens j weitere Datenpunkte enthält. Die so selektierten Punkte werden zu Clustern A_i gruppiert. Um welche bzw. wieviele Cluster es sich hierbei handelt, steht vor der Ausführung des Algorithmus nicht notwendigerweise fest:

$$\mathrm{Cluster}(R) = \Big\{ A_i \mid A_i \subseteq R,\ \forall x \in A_i : ||\mathrm{K}_\varepsilon(x, A_i)|| \geq j \Big\}.$$

Viele bekannte Cluster-Algorithmen wie z.B. DBSCAN [EKSX96] und OPTICS [ABKS99] müssen intern für jeden Punkt der Eingabemenge R eine Kugelanfrage berechnen. Diese Menge von Anfragen läßt sich durch einen Ähnlichkeitsselbstverbund ersetzen. In [BBBK00] wurde gezeigt, daß durch Verwendung eines Similarity Joins als internen Algorithmus für Cluster-Algorithmen große Leistungsgewinne erzielt werden können.

1.1.2 Beispiel: Outlier Detection

Outlier („Ausreißer“) kennzeichnen generell ‚auffällige‘ Datenpunkte:

> “An outlier is an observation that deviates so much from other observations as to arouse suspicions that it was generated by a different mechanism.” [Haw80]

Beispielsweise kennzeichnen Outlier in Data Mining Anwendungen Betrugsfälle bzw. Datenfehler.

Outlier können mit Hilfe des Differenzoperators ebenfalls durch einen Similarity Join berechnet werden[1]:

$$\text{Outlier}(R) = R \setminus \bigcup_i A_i.$$

Die Menge Outlier(R) enthält folglich alle diejenigen Punkte aus R, die nicht Teil eines Clusters sind. Outlier und Cluster können somit algebraisch mit Hilfe eines Ähnlichkeitsverbunds als Grundlage berechnet werden.

1.2 I/O-Modell und Notation

Mit R und S seien die Eingabemengen einer Joinoperation bezeichnet. Z bezeichne die Anzahl der Ergebnistupel (‚Antworten‘). In dieser Arbeit gehen wir i.A. davon aus, daß die Problemgröße der Verfahren den verfügbaren Hauptspeicher übersteigt. Grundlage unserer Analysen ist ein zweistufiges Speichermodell bestehend aus *Hauptspeicher* und *Externspeicher* (z.B. Festplatte). Wir nehmen an, daß die Daten zu Speicherseiten einer festen Größe zusammengefaßt sind. Daten werden grundsätzlich nur in Speicherseiten gelesen und geschrieben. Eine Schreib- bzw. Leseoperation wird in den I/O-Modellen als eine I/O-Operation gezählt.

Eine Speicherseite faßt B Elemente. Der verfügbare Hauptspeicher in Seiten wird mit m bezeichnet. $|R|$ bzw. $|S|$ bezeichne die Anzahl der Speicherseiten von R bzw. S. $||R||$ bzw. $||S||$ bezeichne die Anzahl der in R bzw. S enthaltenen Elemente. Für Similarity Joins werden hochdimensionale Punkte der Dimension d betrachtet. Diese Punkte sind üblicherweise *Merkmalsvektoren (Feature Vectors)* und werden mit x, die Komponente von x in der i-ten Dimension mit x_i bezeichnet.

[1] Diese Darstellung kann auch auf andere Outlier-Definitionen wie z.B. *lokale Outlier (LOFs)* [BKNS00] erweitert werden.

Die übrige Notation wird in den entsprechenden Kapiteln eingeführt. In Anhang 6.2 ist eine tabellarische Auflistung der von uns verwendeten Notation zu finden.

1.3 Namensgebung und Begriffe

Nach Möglichkeit werden deutsche Bezeichnungen in dieser Arbeit verwendet. Für einige Fachtermini weichen wir aus Gründen der besseren Lesbarkeit davon ab. Insbesondere haben wir auch bei Methodennamen die englischen Bezeichnungen beibehalten, da diese in der deutschen Informatikliteratur erstens sehr geläufig sind und zweitens die deutschen Übersetzungen sehr sperrig klingen würden (Beispiel: `insert(...)`, `query(..)`).

Bei der Diskussion von Pseudocode verwenden wir in Anlehnung an die objektorientierte Programmierung den Begriff *Methode* statt *Funktion*.

1.4 Grundlegende Definitionen

Im folgenden wird aus Gründen der Übersicht angenommen, daß die Merkmalsvektoren im d-dimensionalen Einheitswürfel $U = [0,1)^d$ enthalten sind und daß das Distanzmaß die L_p-Metrik ist. Zunächst benötigen wir einige grundlegende Definitionen:

Definition 1 (n-äres Prädikat). *Sei Y eine Menge. Unter einem n-ären Prädikat P verstehen wir eine Abbildung*

$$P : \underbrace{Y \times \ldots \times Y}_{n} \rightarrow \{false, true\}.$$

1.4.1 Verbund

Definition 2 (Verbund (engl.: Join)). *Sei P ein binäres Prädikat. Der Verbund zweier Datenmengen R und S ist gegeben durch*

$$R \bowtie S \;=\; \{(x,y) \mid P(x,y),\; x \in R,\; y \in S\}\,.$$

Definition 3 (Selbstverbund (engl.: Self Join)). *Sei P ein binäres Prädikat. Der Selbstverbund einer Datenmenge R ist gegeben durch*

$$R \bowtie R \;=\; \{(x,y) \mid P(x,y),\; x,y \in R,\; x \neq y\}\,.$$

1.4.2 Ähnlichkeitsverbund

Definition 4 (Distanzprädikat). *Seien $x = (x_1, \ldots, x_d) \in R$ und $y = (y_1, \ldots, y_d) \in S$ zwei d-dimensionale Punkte eines metrischen Raumes. Dann ist das Distanzprädikat unter der L_p-Metrik für $x \in R$ und $y \in S$ wie folgt definiert:*

$$P_\varepsilon(x,y) = \left(\sum_{i=1}^{d} |x_i - y_i|^p \right)^{1/p} \leq \varepsilon.$$

Definition 5 (Ähnlichkeitsverbund (engl.: Similarity Join)). *Seien R und S zwei Mengen d-dimensionaler Punkte eines metrischen Raumes. Dann ist der Ähnlichkeitsverbund für zwei Datenmengen wie folgt definiert:*

$$R \bowtie_\varepsilon S = \{ (x,y) \mid P_\varepsilon(x,y),\ x \in R,\ y \in S \}.$$

Für den wichtigen Spezialfall des Ähnlichkeitsselbstverbund ($R = S$) gilt die entsprechende Definition des allgemeinen Selbstverbunds:

$$R \bowtie_\varepsilon R = \{ (x,y) \mid P_\varepsilon(x,y),\ x,y \in R,\ x \neq y \}.$$

Für das Auswerten des Distanzprädikates sind d Potenzoperationen sowie eine Wurzeloperation notwendig. Die Wurzeloperation ist streng monoton auf $[0;\infty)$, besitzt folglich eine Umkehrfunktion und kann eingespart werden, d.h. das Distanzprädikat wird modifiziert zu

$$P_\varepsilon^p(x,y) = \sum_{i=1}^{d} |x_i - y_i|^p \leq \varepsilon^p.$$

Für die d Potenzoperationen des Prädikates ist dies leider nicht möglich.

1.4.3 Mehrschrittverarbeitung

Die Auswertung der Distanzfunktion kann — wie im letzten Abschnitt gezeigt — sehr teuer sein. Deshalb verwenden wir eine Mehrschrittverarbeitung [Ore86, BKSS94]. Hierfür berechnen wir zunächst mit Hilfe eines Filters eine Obermenge des Joins und selektieren hieraus dann in einem zweiten Schritt die tatsächlich gesuchten Ergebnistupel:

1. **Filterschritt:** Wir berechnen zunächst den Join, der die L_∞-Metrik als Prädikat verwendet. Der Similarity Join unter der L_∞-Metrik ist äquivalent zu einer räumlichen Verbundoperation zweier Mengen von Hypercubes $H(x) := H_{\varepsilon,d}(x)$. Hierbei

entspricht ε der Seitenlänge des Hypercubes[2] in Dimension i, $1 \leq i \leq d$, $x \in [0;1)^d$ ist ein Merkmalsvektor. Die Berechnung dieses Prädikates verursacht maximal d Intervallschnittoperationen. In der Regel sind aber weniger als d Operationen notwendig, da die Auswertung des Prädikates sofort abgebrochen werden kann, wenn sich die Hypercubes bereits in einer Dimension nicht überlappen.

2. **Verfeinerungsschritt:** Wir selektieren aus den berechneten ‚Kandidaten' alle diejenigen Tupel, die die eigentliche L_p-Metrik erfüllen. Man beachte, daß diese Überprüfung des Verfeinerungsschrittes direkt ausgeführt werden kann. Dies bedeutet insbesondere, daß die Relation der Kandidaten nicht zwischengespeichert werden muß. Wir werden in dieser Arbeit — wie in der Literatur zum Similarity Join üblich — annehmen, daß $p = 2$ gilt, d.h. die genutzte Metrik ist die Euklidische. Unsere Verfahren sind aber nicht hierauf beschränkt.

Dieses zweistufige Vorgehen ist korrekt, da $P_\varepsilon^p(x,y) \Rightarrow P_\varepsilon^\infty(x,y)$ für $p \in \mathbb{N}$ gilt. Die Leistung des Similarity Join wird stark beeinflußt von der Performanz des Filterschrittes. Es sollten nicht zuviele 'false drops' erzeugt werden, d.h. Ergebnistupel, die vom Filterschritt als Kandidat erzeugt, dann aber vom Verfeinerungsschritt nicht als Ergebnistupel akzeptiert werden.

1.4.4 Die Auswirkung der Dimension

Ein wichtiges Problem bei der Verarbeitung hochdimensionaler Daten ist der sogenannte *curse of dimensionality*, ein Problem, das in der Statistik seit langem bekannt ist [Sco92]. Unter der Annahme, daß hochdimensionale Daten gleichverteilt und die Dimensionen unabhängig sind, scheint es schwierig zu sein, einen effizienten Joinalgorithmus zu entwickeln. Um das Problem zu illustrieren, haben wir in Abbildung 1.2 die Seitenlänge ε eines Hypercubes in Abhängigkeit der Anzahl der Dimensionen d abgetragen (für verschiedene feste Volumina V). Diese Ergebnisse gelten, wenn die Annahme gemacht wird, daß die Daten gleichförmig und unabhängig verteilt sind. Für fünfzig Dimensionen ist ε fast so groß wie der Datenraum. Glücklicherweise trifft obige Annahme für Daten der realen Welt nicht zu, in denen eine starke Korrelation zwischen den einzelnen Dimensionen beobachtet werden kann. Die Ergebnisse der Experimente in [FSTT00] legen nahe, daß die sogenannte *fraktale (oder Hausdorff) Dimension* ein exzellentes Maß für die **wahre** Dimension eines Datensatzes ist. Die Ergebnisse unter der Annahme der Gleichverteilung sind dann anwendbar,

[2] Dies entspricht einer Anfragedistanz von $\varepsilon/2$ unter der L_p-Metrik. Da beide Punktmengen R und S transformiert werden, werden im Filterschritt alle Tupel mit Abstand ε gefunden.

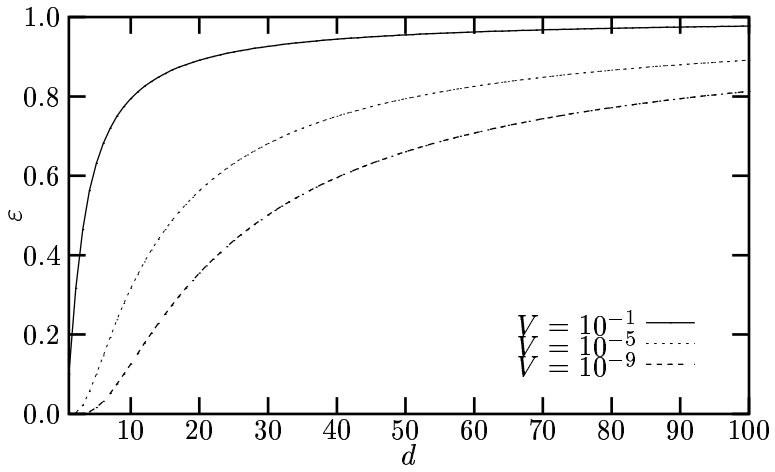


Abbildung 1.2: Länge ε eines Hypercube in einer Dimension in Abhängigkeit von d (Gesamtzahl der Dimensionen)

wenn d durch die fraktale Dimension ersetzt wird [BF98]. Wir werden weiter unten noch detailliert auf dieses Phänomen eingehen.

1.5 Problemstellung und Voraussetzungen

1.5.1 Datenreplikation in hochdimensionalen Datenräumen

In [Dit99, DS00] wurde festgestellt, daß kontrollierte Datenreplikation für den räumlichen Verbund einen großen Leistungsgewinn zur Folge hat. Aber lohnt sich Datenreplikation auch in hochdimensionalen Datenräumen? Wie sind die Auswirkungen für den *Ähnlichkeitsverbund*?

1.5.2 Joinverfahren blockieren den Datenfluß

Zunächst benötigen wir eine weitere Definition:

Definition 6 (Blockieren). *Ein Algorithmus heißt blockierend, wenn gilt, daß der Algorithmus erst alle Eingabedaten einlesen muß, bevor er das erste Ergebnis der Ausgabe berechnen kann.*

Diese Eigenschaft trifft insbesondere auf Sortieralgorithmen zu. Damit sind alle Joinverfahren, die die Daten zunächst sortieren müssen, notwendigerweise blockierend. Dies ist illustriert in Abbildung 1.3: auf der x-Achse ist die Laufzeit, auf der y-Achse die Anzahl

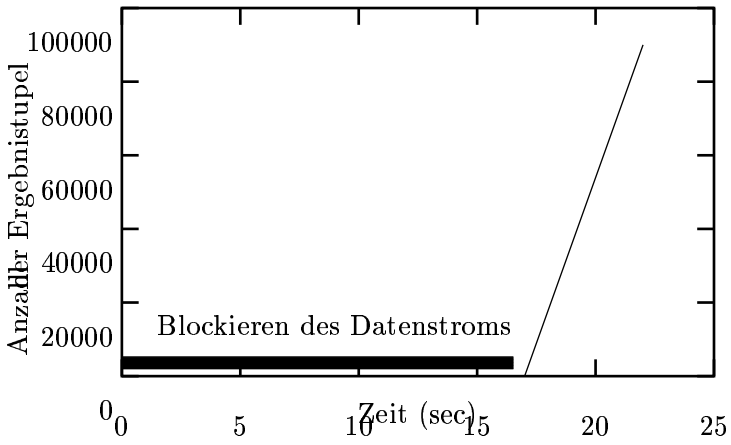


Abbildung 1.3: Das Blockierverhalten sortierbasierter Verbundoperationen

der gefundenen Ergebnistupel abgetragen. Die Abbildung zeigt, daß in der ersten Phase des Algorithmus (0–17 Sekunden) kein einziges Ergebnistupel berechnet wird. Danach wird eine große Zahl von Tupeln in relativ kurzer Zeit produziert. D.h. in den ersten 17 Sekunden der Laufzeit wird der Datenstrom *blockiert*: nachfolgende Operatoren müssen warten, da sie keine Ergebnistupel erhalten. Diese blockierende Verhalten ist typisch für viele Joinalgorithmen, u.a. auch für Similarity Joins und ein ernstes Problem in interaktiven Data Mining Anwendungen wie z.B. Clusteranalysen und Visualisierungen.

Gibt es ein Verfahren für den Ähnlichkeitsverbund, das Daten sortiert, aber den Datenfluß trotzdem nicht blockiert?

1.5.3 Zusätzliche Voraussetzungen

Wir gehen davon aus, daß die in dieser Arbeit behandelten Verfahren Teil eines komplexen Operatorbaums sind und deshalb kein Index auf den Eingabedaten existiert. Ferner nehmen wir an, daß die Problemgröße die Größe des verfügbaren Hauptspeichers übersteigt (Externspeicherverfahren).

1.6 Gliederung dieser Arbeit

- In Kapitel 2 (**Joinverarbeitung**) werden zunächst prinzipielle Vorgehensweisen bei der Verarbeitung von Joinoperationen vorgestellt. Es werden die wichtigsten algorithmischen Paradigmen und ausgehend hiervon die für die vorliegende Arbeit relevante Literatur diskutiert.

- Kapitel 3 (**GESS**) untersucht ausgehend vom *Multidimensional Spatial Join (MSJ)* die Auswirkungen von Replikation bei der Verarbeitung von Similarity Joins. Es wird ein generisches Verfahren namens *Generic External Space Sweep (GESS)* präsentiert. GESS umschließt zahlreiche andere Joinverfahren als Spezialfälle. Es wird ein Kostenmodell für Replikation und die Anzahl der Distanzberechnungen für MSJ und GESS vorgestellt. Das I/O-Modell von MSJ wird ersetzt und eine online Methode zur Duplikateliminierung vorgestellt. Das Kostenmodell wird mit Ergebnissen von Simulationen validiert. In einer Reihe von Experimenten wird das I/O-Verhalten, die Anzahl der Distanzberechnungen, der Hauptspeicherbedarf, die Replikation und die Laufzeit von GESS gegenüber MSJ untersucht. Ferner werden die fraktale Dimension der verwendeten Datensätzen sowie deren Auswirkung auf das Kostenmodell untersucht.

- Kapitel 4 (**PMJ**) präsentiert einen nichtblockierenden, sortierbasierten, generischen Joinalgorithmus namens *Progressive Merge Join (PMJ)*. Die wesentlich Idee von PMJ ist es, beide Sortieroperatoren mit dem eigentlichen Joinoperator zu einem Operator zu verschmelzen. Hierdurch können bereits während der Run-Erzeugung Ergebnistupel produziert werden. Folglich kann PMJ extrem früh Ergebnistupel an den nächsten Operator weiterreichen.

  Diese Kapitel stellt den Grundalgorithmus von PMJ sowie verschiedene Anwendungsfälle mit verschiedenen Jointypen vor. Insbesondere für den Similarity Join wird das erste nichtblockierende Verfahren durch Anwendung von PMJ auf GESS präsentiert.

  In einer Reihe von Experimenten wird gezeigt, daß eine beträchtliche Anzahl von Ergebnistupeln produziert wird, noch bevor die Sortierphase der konventionellen Techniken beendet ist. Die Ergebnisse zeigen nur einen geringen Anstieg der Laufzeit der Algorithmen im Vergleich zu den korrespondierenden blockierenden Techniken.

- Kapitel 5 (**XXL**) gibt eine kurze Übersicht über die *eXtensible and fleXible Library (XXL)*. XXL ist eine plattformunabhängige Javabibliothek von Frameworks und Algorithmen zur Anfrageverarbeitung. XXL wird in der AG Datenbanksysteme der Universität Marburg unter Leitung von Prof. Dr. Bernhard Seeger entwickelt. Es werden die wichtigsten Konzepte der Bibliothek diskutiert und die Integration der in dieser Arbeit vorgestellten Algorithmen beschrieben.

- Kapitel 6 (**Zusammenfassung und Ausblick**) faßt die Beiträge dieser Arbeit zusammen und beschließt die Arbeit.

Kapitel 2

Joinverarbeitung in Datenbanken

Zunächst diskutieren wir prinzipielle Vorgehensweisen für die Verarbeitung von Joinoperationen (Abschnitt 2.1). Im Anschluß präsentieren wir die für Similarity Joins und insbesondere nichtblockierende Joinoperationen relevante Literatur (Abschnitt 2.2).

2.1 Prinzipielle Vorgehensweisen

In diesem Abschnitt diskutieren wir zunächst prinzipielle Vorgehensweisen bei der Verarbeitung von Joinoperationen. Für Übersichtsartikel zu Techniken der Joinverarbeitung siehe auch [ME92, Gra93].

2.1.1 Nested-Loops Join

Die einfachste Art, den Join von zwei Datenmengen R und S zu berechnen, besteht darin, das Kreuzprodukt von R und S zu bilden und diejenigen Tupel zu selektieren, für die das Joinprädikat erfüllt ist. Dies entspricht obiger mathematischer Definition des Joins (Def. 2). Der Aufwand dieses Verfahrens ist jedoch $O(||R|| \cdot ||S||)$ und damit für große Datenmengen zu teuer. Implementiert wird dieses Verfahren durch zwei geschachtelte Schleifen (*"Nested-Loops"*), die iterativ das Joinprädikat P für jedes mögliche Tupel (r, s), $r \in R$, $s \in S$ auswerten. Abbildung 2.1 zeigt den Algorithmus in Pseudocode.

Liegt mindestens eine der Eingabemengen auf dem Externspeicher, werden aus Effizienzgründen die Daten zu Blöcken fester Größe (z.B.: Speicherseiten) gruppiert. Der Algorithmus verarbeitet dann diese Gruppen und nicht mehr einzelne Elemente. Der resultierende Algorithmus wird *Seitenweises Nested-Loops* genannt.

```
1 Sei RES = {};
2 Für alle r aus R:
3       Für alle s aus S:
4             Falls P(r,s) :
5                   RES = RES ∪ {r,s};
```

Abbildung 2.1: Nested-Loops Join

```
1 Sei RES = {};
2 Sei S_ind ein Index auf S;
3 Für alle r aus R:
4       RES = RES ∪ {(r,s) | s ∈ S_ind.query(r)};
```

Abbildung 2.2: Indexed Nested-Loops Join

2.1.2 Indexed Nested-Loops Join

Für den Fall, daß auf einer der Datenmengen ein Index existiert, können die Elemente der einen Datenmenge als Anfragen an diesen Index genutzt werden. Angenommen für S existiert ein Index. Dann wird für jedes Element aus $r \in R$ eine Anfrage an den Index von S gestellt. Dieser Algorithmus heißt *Indexed Nested-Loops Join*. Die innere Schleife fällt hierbei weg und wird durch eine Anfrage an den Index von S ersetzt (siehe Abbildung 2.2). Dies illustriert die Idee, den Join als eine *Massenanfrage* zu betrachten, wobei R die Anfragemenge und S die Datenmenge bezeichnet.

2.1.3 Hashjoin

Beim *Hashjoin* werden beide Eingaben mit Hilfe einer *Hashfunktion* partitioniert. Die kleinere der beiden Eingabemengen ist der so genannte *Build Input* und wird rekursiv solange partitioniert, bis alle Partitionen in den Hauptspeicher passen. Die größere Eingabemenge, der *Probe Input*, wird mit derselben Hashfunktion partitioniert. Dann werden die Partitionen des Build Input nacheinander in den Hauptspeicher geladen und mit den korrespondierenden Partitionen des Probe Inputs verknüpft. Letztere müssen nicht komplett in den Hauptspeicher passen. Abbildung 2.3 zeigt den Algorithmus in Pseudocode.

```
1 Sei RES = {};
2 Sei R der Build Input;
3 Partitioniere R rekursiv solange
      bis alle Partitionen in den HS passen;
4 Partitioniere S mit der gleichen Hashfunktion;
5 Für jedes Partitionspaar:
6         Joine Partitionspaar und füge
              Ergebnistupel zu RES hinzu;
```

Abbildung 2.3: Hashjoin

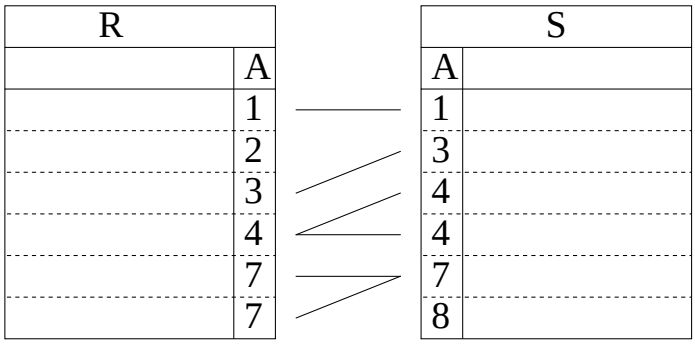


Abbildung 2.4: Beispiel für Equi Join

2.1.4 Sort-Merge Join

Das Prinzip des Sort-Merge Join ist von fundamentaler Bedeutung für die vorliegende Arbeit. Wir skizzieren an dieser Stelle zunächst nur die Grundidee und kommen später hierauf zurück.

Beim Sort-Merge Join werden die Eingabemengen R und S zunächst in Folgen R' und S' sortiert. Für beide Mengen wird ein ‚Datenfenster' (Sweep Line, Sweep Area) im Hauptspeicher gehalten. Die Sortierung der Daten wird so ausgenutzt, daß in den Datenfenstern bei der Verarbeitung des Joins nur Teilmengen der Eingabemengen gehalten werden müssen.

In den Datenfenstern werden nur die für die Joinverarbeitung relevanten Datenobjekte gespeichert. Von den sortierten Folgen R' und S' wird jeweils das kleinste Element bezüglich der genutzten Ordnung gewählt und verarbeitet. Zunächst wird für dieses Element überprüft, ob Elemente aus den Datenfenstern gelöscht werden können. Danach wird dieses Element an das Datenfenster der anderen Datenmenge angefragt.

Abbildung 2.4 veranschaulicht dieses Prinzip für den relationalen Equi Join. In dem Beispiel

werden die Relationen R und S mittels des Attributs A verknüpft. Beide Relationen sind in einem ersten Schritt zunächst nach dem Attribut A sortiert worden (*Sort*). Im nachfolgenden *Vermischen (Merge)* werden die Relationen dann synchron durchlaufen. Hierfür wird für jede Relation ein ‚Zeiger' auf das aktuelle Element gehalten. Zeigen die Zeiger beider Relationen auf einen gleichen Wert, ist ein Ergebnistupel gefunden worden. Danach wird jeweils der Zeiger mit dem kleinsten Nachfolgeelement erhöht und erneut verglichen. Dieser Vorgang wird solange wiederholt, bis beide Zeiger das Ende ihrer Relationen erreicht haben.

Bezeichne N die Anzahl der Elemente in R bzw. S. Wir nehmen der Einfachheit an, daß $N = ||R|| + ||S||$. Die I/O-Komplexität des Joins wird durch die Kosten für das externe Sortieren bestimmt. Die Komplexität für den eigentlichen Join (ohne Sortieren) hingegen hängt von der verwendeten Hauptspeicherstruktur ab. Für den Equi Join ist diese $O(N + Z)$, wobei Z die Anzahl der Ergebnistupel des Joins bezeichne. Für den allgemeinen Fall bestimmt sich die Komplexität durch die Anfrage- und Löschkomplexität der verwendeten Hauptspeicherdatenstruktur und führt zu einem Aufwand von $O(N \cdot Y(N) + Z)$, wobei $Y(N)$ den Aufwand für das Anfragen und Löschen in der Hauptspeicherdatenstruktur bezeichne. Für einen Spatial Join mit einem balancierten Intervallbaum als Datenstruktur [APR+98] ergibt sich z.B. ohne Sortieren ein Hauptspeicheraufwand von $O(N \log N + Z)$.

In der Literatur werden die zahlreichen Joinalgorithmen, die auf dem Sort-Merge-Paradigma beruhen, als unterschiedliche Verfahren dargestellt. So wird z.B. unterschieden zwischen dem *Sort-Merge Join* [BE77] für relationale Daten (Equi Join bzw. Theta Join), *Plane Sweep* [PS85, APR+98] für räumliche Daten und *Temporal Joins* [SSJ94] bzw. *Similarity Joins* [Ore91] und *Band Joins* [DNS91].

In Kapitel 4 wird ein generisches Konzept vorgestellt, mit dem alle sortierbasierten Joinalgorithmen als Spezialfälle unseres Algorithmus formuliert werden können. In Kapitel 3.1 wird insbesondere gezeigt, daß die unterschiedlichen Partitionierungsstrategien von Spatial- und Similarity Joins Spezialfälle unseres Algorithmus sind.

2.1.5 Schachteln von Algorithmen

Die verschiedenen Joinprinzipien können beliebig gruppiert/geschachtelt werden. So kann z.B. in einem ersten Schritt eine Aufteilung der Daten mittels eines Hashjoins erfolgen. In einem zweiten Schritt können die entstandenen Partitionspaare dann mit einem sortierbasierten Verfahren weiterverarbeitet werden. Ein Beispiel hierfür ist der Algorithmus für den räumlichen Verbund *PBSM* [PD96] bzw. [DS00]. Dieses Verfahren teilt die Daten zunächst mit Hilfe eines räumlichen Gitters auf und berechnet Partitionspaare. Die Partitionspaare

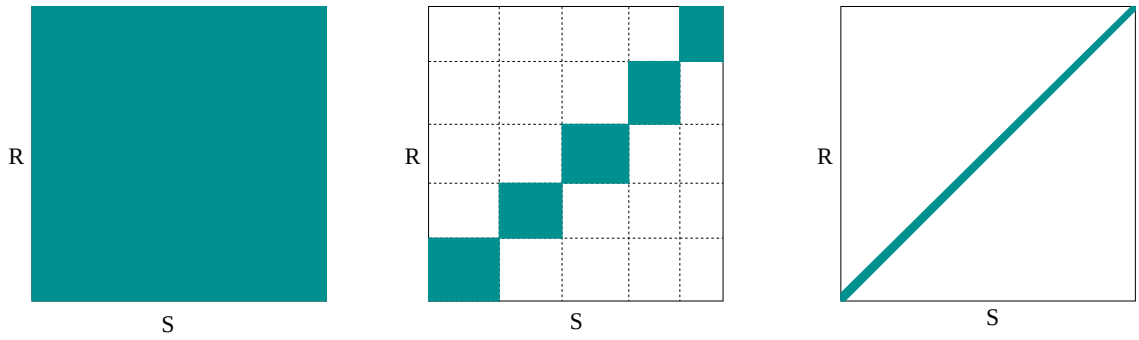


Abbildung 2.5: Auswirkung der Partitionierung bei verschiedenen Joinalgorithmen (Nested-Loops Join, Hashjoin, Sort-Merge Join, vgl. Mishra und Eich [ME92])

werden dann mittels eines Plane Sweep-Algorithmus [BKS93] verknüpft. Ein Vorteil von geschachtelten Joins ist u.a., daß Joinverfahren hiermit zur Laufzeit einen geeigneten (inneren) Joinalgorithmus auswählen können, welcher für das konkrete Problem effizienter sein kann.

2.2 Literatur

In der folgenden Literaturübersicht wird für jedes Verfahren jeweils das verwendete Joinprinzip angegeben. Zunächst wird ein Überblick über die Literatur zum Spatial Join gegeben (Abschnitt 2.2.1). Danach wird die Literatur zum Similarity Join vorgestellt (Abschnitt 2.2.2). Anschließend diskutieren wir die Literatur zu nichtblockierenden Joinalgorithmen (Abschnitt 2.2.3).

Da die Annahme gemacht wurde, daß kein Index auf den Eingabemengen existiert, wird an dieser Stelle nicht näher auf indexbasierte Joins eingegangen.

2.2.1 Spatial Join

Da einige Hashverfahren Indexstrukturen als Partitionierungsfunktion nutzen, seien kurz die wichtigsten Vertreter erwähnt.

Die wichtigsten Indexstrukturen für Geodaten sind der R-Baum [Gut84], R^+-Baum [SRF87] und R*-Baum [BKSS90], sowie Quadtrees [Sam90b, Sam90a]. Für den Fall, daß beide Datenmengen indiziert sind, kann für R*-Bäume der von Brinkhoff, Kriegel und Seeger [BKS93]

vorgeschlagene Traversierungsalgorithmus genutzt werden. Für Quadtrees wurde von Hoel und Samet in [HS95] ein Verfahren präsentiert.

2.2.1.1 Hashbasierte Verfahren

SHJ [LR96] partitioniert zunächst die innere Datenmenge in überlappende Regionen. Als Partitionierungsfunktion wird hierbei ein R-Baum mit nur einer Baumebene genutzt. Dieser wird mit Hilfe der *bootstrap seeding* Technik aus [LR95] erzeugt. Danach wird die äußere Datenmenge mit derselben Partitionierungsfunktion aufgeteilt. Hierbei kann es passieren, daß Elemente der äußeren Datenmenge mehreren Partitionen zugeordnet werden müssen (Replikation).

In der Joinphase werden die in der Partitionierungsphase gewonnenen Paare von überlappenden Partitionen verknüpft. Hierfür wird der Indexed Nested-Loops Join genutzt, d.h. für die innere Partition wird jeweils ein R-Baum konstruiert. Die Elemente der äußeren Partition dienen als Anfragen an diesen R-Baum. Diese Vorgehensweise entspricht dem oben erwähnten Schachteln von Algorithmen: zunächst wird ein Hashverfahren genutzt, um die Daten in Partitionen aufzuteilen, danach werden die Partitionspaare mit einem anderen Verfahren (in diesem Fall Indexed Nested-Loops) verknüpft. Es entstehen keine Duplikate in der Antwortmenge.

Ein ähnliches Verfahren wurde von Patel und DeWitt auf der selben Konferenz wie SHJ präsentiert. *PBSM* [PD96] teilt die Daten mit einem festen Gitter in sogenannte *Kacheln* auf. Mehrere Kacheln werden dann zu einer Partition zusammengefaßt. Korrespondierende Partitionspaare werden mit einem Plane Sweep-Verfahren [BKS93] verknüpft. Da durch Replikation Duplikate in der Antwortmenge entstehen können, müssen die Ergebnistupel noch sortiert und Duplikate eliminiert werden. In [Dit99, DS00] wird eine verbesserte Variante von PBSM vorgeschlagen, die ohne diese Nachbehandlung der Ergebnistupel auskommt.

Als Partitionierungsfunktion kann für den Hashjoin im Prinzip ein beliebiger Index genutzt werden. So nutzen Lo und Ravishankar einen R-Baum für den ersten Partitionierungsschritt [LR94, LR98]. Van den Bercken, Schneider und Seeger verallgemeinern diese Idee und stellen ein generisches Verfahren namens Plug&Join [BSS00] vor, das mit beliebigen Grow&Post-Bäumen [Lom91] funktioniert und eine rekursive Partitionierung unterstützt.

2.2.1.2 Sortierbasierte Verfahren

Der erste Vorschlag für sortierbasierte Spatial Joins stammt von Orenstein [Ore86]. Dieser Algorithmus nutzt raumfüllende Kurven, um den Datenraum in gleichförmige Partitionen aufzuteilen. Das Verfahren von Orenstein ist ein Ausgangspunkt dieser Arbeit. In Kapitel 3.1 wird detailliert hierauf eingegangen.

Eine ähnliche Idee verfolgen Koudas und Sevcik mit S^3J [KS97] bzw. *MSJ* [KS98, KS00]. Im Gegensatz zu Orenstein partitionieren sie die Daten in sogenannte *Ebenendateien (Level-Files)*. Auch auf diesen Algorithmus wird detailliert in Kapitel 3.1 eingegangen.

Von Arge et.al. stammt der *SSSJ*-Algorithmus [APR+98]. Die Experimente in [APR+98] bestätigen die sogenannte *Wurzelregel* aus [GS87]. Diese besagt, daß die Sweepline eines Plane Sweep Algorithmus ca. $\sqrt{N}$ Elemente überdeckt — nur diese Elemente müssen für den eigentlichen Mergejoin im Hauptspeicher gehalten werden. SSSJ versucht während des Sortierens die Größe der Sweep Line zu schätzen und partitioniert die Daten nur dann, wenn diese voraussichtlich die Größe des Hauptspeichers übersteigt.

Trotz der Tatsache, daß das strikte Sort-Merge Join in den Experimenten in [APR+98] genutzt wurde, war dieses Verfahren in [APR+98] bereits effizienter als PBSM [PD96]. Wir werden in Kapitel 4 hierauf zurückkommen.

2.2.2 Similarity Join

2.2.2.1 Hashbasierte Verfahren

Prinzipiell fallen in diese Klasse Hashverfahren für den Spatial Join wie [PD96, LR96, BSS00]. Auf Grund der hohen Replikationsraten, die diese Verfahren für hochdimensionale Daten zur Folge haben, sind diese jedoch nicht praktikabel.

2.2.2.2 Sortierbasierte Verfahren

Das Verfahren von Orenstein [Ore91] ist der erste Vorschlag für einen hochdimensionalen Join, der die Daten mit Hilfe einer raumfüllenden Kurve sortiert. Ein ähnlicher Ansatz wurde in [KS00] verfolgt. Letzterer Algorithmus partitioniert die Daten in sogenannte *Ebenendateien (Level-Files)*. Diese beiden Verfahren bilden den Ausgangspunkt des Algorithmus *GESS* in Kapitel 3. Wir werden dort detailliert auf dieses Verfahren eingehen.

Der *ε-KDB-tree* [SSA97] sortiert die Daten bezüglich einer Dimension und hält für jede

Datenmenge einen Streifen der Breite 2ε im Hauptspeicher. Überlappende Streifen werden verknüpft. Diese Vorgehensweise ist nur dann erfolgreich, wenn zwei benachbarte Streifen gleichzeitig im Hauptspeicher gehalten werden können. Für den Fall, daß der Hauptspeicher überläuft, schlagen die Autoren in [SSA97] eine Page-Scheduling-Strategie vor, die Seiten in Nested-Loops Manier verarbeitet.

In [BBKK01] wird *Epsilon Grid Order* vorgestellt. Epsilon Grid Order partitioniert den Datenraum in Würfel der Kantenlänge ε. Die Würfel werden sortiert und lexikographisch verknüpft. Hierbei entsteht das Problem, daß während der Joinphase durch mehrfaches Nachladen von Partitionen *buffer thrashing* auftreten kann. Konsequenterweise entwickeln die Autoren verschiedene Strategien, um letzteren Effekt zu mildern. Keine dieser Strategien kann jedoch verhindern, daß Seiten in der Joinphase u.U. mehrfach gelesen werden müssen.

2.2.3 Nichtblockierende Joinalgorithmen

In diesem Abschnitt geben wir eine Übersicht über die Literatur zu nichtblockierenden Joinverfahren.

Für den konventionellen Equi Join wurde der Hashjoin als die effizienteste Methode identifiziert [Gra94a]. Für Nicht Equi Joins basieren die führenden Joinverfahren jedoch auf dem Sort-Merge Paradigma.

2.2.3.1 Hashbasierte Verfahren

In [WA91] präsentieren Wilschut und Apers den *Symmetric Hashjoin (SHJ)* für die Verarbeitung von Equi Joins. Eine ähnliche Idee wurde von Raschid und Su in [RS86] vorgestellt. SHJ hält für jede Eingabemenge eine Hashtabelle im Hauptspeicher. Jedes zu behandelnde Tupel wird zunächst in die zugehörige Hashtabelle eingefügt und danach gegen die gegenüberliegende Hashtabelle angefragt. Es werden keine Annahmen über die Menge der eintreffenden Tupel gemacht. Für den Fall, daß eine Eingabe zeitweise keine Tupel liefert, können mit Hilfe der anderen Eingabe weiterhin Tupel produziert werden.

Eine ernste Beschränkung von SHJ ist die Anforderung, daß beide Hashtabellen im Hauptspeicher gehalten werden müssen. Dies ist für die Verarbeitung großer Datenmengen oft nicht möglich. In [UF00] schlagen Urhan und Franklin deshalb den *XJoin* vor, eine multi threaded Erweiterung von SHJ, welche Teile der Hashtabellen auf dem Sekundärspeicher hält. Der XJoin partitioniert die Daten in Behälter. Ist kein Hauptspeicher mehr allokierbar, wird der größte Hashbehälter auf Festplatte geschrieben und für eine spätere Bearbei-

tung durch einen parallelen Thread markiert. Da hierbei Duplikate in der Antwortmenge entstehen können, stellt XJoin ferner eine effiziente Methode zur Duplikateliminierung — basierend auf Zeitmarken — zur Verfügung. Ein ähnlicher Ansatz wird in [IFF+99] vorgestellt, wo dieser Algorithmus zur Zusammenführung von Daten aus verschiedenen aktiven Quellen genutzt wird.

Vor kurzem wurde von Luo, Naughton und Ellmann [LNE02] ein nichtblockierender Spatial Join Algorithmus vorgeschlagen. Dies ist nach unserem Wissen die einzige Arbeit, die frühe Ergebnistupelproduktion für Nicht-Equi Joins behandelt. Der Ansatz basiert auf einer Kombination von SHJ und PBSM [PD96]. Ferner wird eine modifizierte Version der in [DS00] vorgeschlagenen Referenzpunktmethode zur Duplikateliminierung verwendet.

Im Kontext von Algorithmen für online Aggregation haben Haas und Hellerstein [HH99] die sogenannten *Ripple Joins* präsentiert. Die Ergebnisse des Ripple Joins werden kontinuierlich an ein online Aggregationssystem weitergereicht. Die spezielle Aggregationsoperation steuert die Verarbeitungsstrategie des Ripple Joins zur Laufzeit.

2.2.3.2 Sortierbasierte Verfahren

Bis jetzt wurden sortierbasierte Joinverfahren immer als eine blockierende Operation betrachtet: erste Ergebnistupel werden erst produziert, nachdem ein Großteil der gesamten Laufzeit des Algorithmus verstrichen ist und insbesondere alle Daten der Eingaberelationen zur Verfügung stehen. Dies gilt insbesondere für den ursprünglichen Sort-Merge Join für relationale Daten [BE77], welcher beide Eingabemengen zunächst vollständig sortiert und dann verknüpft. In [NP85] präsentieren Negri und Pelagatti eine Verbesserung dieses Verfahrens, welches das Blockierverhalten des Sort-Merge Join verbessert. Die Autoren schlagen vor, die letzte Mischoperation der beiden Sortieroperatoren online durchzuführen. D.h. anstatt die sortierten Datensequenzen vollständig auf Festplatte zu schreiben, werden die sortierten Teilsequenzen beider Datenmengen gemischt bis nur noch $2 < F \leq m$ Teilsequenzen auf Festplatte liegen. Für jede Teilsequenz wird dann eine Speicherseite im Hauptspeicher reserviert. Mit Hilfe eines Heaps wird ermittelt, welche Seite das nächste zu behandelnde Element enthält (bezogen auf die genutzte Ordnung). Durch diesen Ansatz wird ein komplettes Schreiben und Lesen der Daten eingespart. Diese Technik wird heutzutage in zahlreichen Systemen genutzt. Das Blockierverhalten des Sort-Merge Join ist damit allerdings nicht verschwunden.

Trotz der Tatsache, daß sortierbasierte Joins blockierend sind, nutzen viele moderne Joinalgorithmen diese Technik. Die in [Ore86, KS97, APR+98, DS00] vorgestellten Spatial Join

Algorithmen basieren auf diesem Prinzip. Das gleiche gilt für Similarity Joins, hier basieren alle bekannten Ansätze auf dem Sort-Merge Paradigma ([SSA97, Ore91, KS00, BBKK01]). Der im nächsten Kapitel vorgestellte GESS-Algorithmus basiert ebenfalls auf diesem Paradigma.

Kapitel 3

GESS

Eines der effizientesten Verfahren zur Berechnung des Similarity Join ist der Multidimensional Spatial Join (MSJ) von Koudas und Sevcik. In einer früheren Arbeit [DS00] — durchgeführt für 2-dimensionale Daten — haben wir jedoch festgestellt, daß MSJ bezüglich CPU- und I/O-Kosten sowie Hauptspeicherbedarf Leistungseinbrüche zeigt.

Das folgende Kapitel stellt einen neuen Algorithmus namens *Generic External Space Sweep (GESS)* vor. GESS nutzt kontrollierte Replikation, um die Anzahl der teuren Distanzberechnungen zu reduzieren. Es werden ein Kostenmodell für Replikation, ein I/O-Modell und eine effiziente Methode zur online Duplikateliminierung präsentiert. Eine wesentliche Komponente des Verfahrens ist ein flexibler Algorithmus zur Steuerung der Replikation.

Unser analytisches Modell sagt eine hohe Reduktion (mehrere Größenordnungen) der Distanzberechnungen im Vergleich zu MSJ voraus. Ferner zeigen unsere Ergebnisse, daß der Hauptspeicherbedarf von GESS um mehrere Größenordnungen geringer ist im Vergleich zu MSJ. Die I/O-Kosten unseres Algorithmus sind um den Faktor 2 und mehr besser (unabhängig davon ob Replikation genutzt wird oder nicht). Unsere analytischen Ergebnisse werden bestätigt durch eine Serie von Simulationen und Experimenten mit synthetischen und realen hochdimensionalen Datensätzen.

Dieses Kapitel gliedert sich wie folgt: der nachfolgende Abschnitt 3.1 stellt einen generischen Rahmen für die Joinalgorithmen aus [Ore91, KS00] vor. Danach wird gezeigt, daß letztere Verfahren Spezialfälle dieses Rahmens sind. Kapitel 3.2 stellt GESS vor, ein neues, generisches Verfahren für d-dimensionale Verbundoperationen. Es wird ein Kostenmodell für Replikation und die Anzahl der Distanzberechnungen in Abschnitt 3.3 präsentiert. Abschnitt 3.4 analysiert die I/O-Kosten von MSJ und GESS. Abschnitt 3.5 diskutiert Implementierungsaspekte des Verfahrens. Abschnitt 3.6 untersucht in einer Reihe von Experi-

menten die Laufzeit, die Anzahl der Distanzberechnungen und den Hauptspeicherbedarf. Hierfür wird sowohl eine Simulation als auch eine Implementierung genutzt. In Abschnitt 3.7 wird die fraktale Dimension der genutzten Datensätze bestimmt und mit Voraussagen des Kostenmodells verglichen.

3.1 Generischer Rahmen

In diesem Abschnitt wird zunächst ein generischer Rahmen für die Joinverfahren aus [Ore91, KS00] präsentiert. Danach wird gezeigt, daß letztere Algorithmen Spezialfälle dieses Rahmens sind.

Definition 7 (Rekursive Partitionierung). *Seien n, l, mL positive ganze Zahlen mit $l < mL$. Sei D^l eine Menge. Sei $D_0^{l+1}, ..., D_{n-1}^{l+1}$ eine Folge von Teilmengen von D^l, die die folgende Bedingung erfüllen:*

- $D^l = \bigcup_{0 \le i < n} D_i^{l+1}$
- $D_i^{l+1} \cap D_j^{l+1} = \varnothing$ *für* $i \neq j$

Dann ist

$$Part(D^l) := \begin{cases} \{D^l\} \cup \bigcup_{0 \le i < n} Part(D_i^{l+1}) & , falls\ l < mL \\ \{D^l\} & , sonst\ (l = mL) \end{cases} \tag{3.1}$$

eine n-äre rekursive Partitionierung von D^l. $D_0^{l+1}, ..., D_{n-1}^{l+1}$ werden die Direkten Teilräume von D^l genannt. Im folgenden gelte $D := D^0$.

Man beachte, daß Baumstrukturen wie kd-tries [Ore91] und Quadtrees [Sam90b] eine rekursive Partitionierung für $n = 2$ bzw. $n = 2^d$ induzieren, denn jeder Knoten repräsentiert einen Teilraum des zugrundeliegenden Datenraums. Das nächste Lemma faßt einige wichtige Eigenschaften n-ärer rekursiver Partitionierungen zusammen.

Lemma 1. *Sei D^l eine Menge und $Part(D^l)$ eine n-äre rekursive Partitionierung von D^l.*

1. *Dann gilt die folgende Bedingung für $A, B \in Part(D^l)$:*
   $A \cap B = \varnothing \quad \vee \quad A \supset B \quad \vee \quad A \subseteq B$
2. *Seien $A \in Part(D^l)$ und $D_0^{l+1}, ..., D_{n-1}^{l+1}$ die Direkten Teilräume von D^l. Falls $A \neq D^l$, dann existiert genau ein i, so daß*
   $A \subseteq D_i^{l+1} \quad \wedge \quad A \cap D_j^{l+1} = \varnothing$ *für* $j \neq i$.

Beweis Der Beweis ergibt sich direkt aus Definition 7. □

Definition 8. *Sei D^l eine Menge und Part(D^l) eine n-äre rekursive Partitionierung. Seien $D_0^{l+1}, ..., D_{n-1}^{l+1}$ die Direkten Teilräume von D^l. Sei $A \in Part(D^l)$. Dann gilt*

$$Code_{D^l}(A) := \begin{cases} < i \oplus Code_{D_i^{l+1}}(A) > & , falls\ A \subseteq D_i^{l+1} \\ <> & , sonst\ (A = D^l) \end{cases} \tag{3.2}$$

Die Länge von $Code_{D^l}(A)$ wird die Ebene von A genannt.

Die oben definierte Funktion *Code* gibt einen String mit Ziffern aus $\{0, ..., n-1\}$ zurück. Um Datenobjekte in einer rekursiven Partitionierung zu sortieren, nutzen wir die *lexikographische Ordnung* auf den Strings. Offensichtlich gilt: falls String s_1 Präfix von $s_2 \Rightarrow s_1 < s_2$.

Eine rekursive Partitionierung Part(A) stellt ein Skelett zur Verfügung, wobei ein Hypercube $H(x)$ des Merkmalsvektors x einem oder mehreren direkten Teilräumen von Part(A) zugewiesen wird. Der Code des Hypercubes entspricht dem des zugeordneten Teilraums. Die nachfolgenden Joinverfahren unterscheiden sich im wesentlichen durch ihre verschiedenen Zuordnungsstrategien.

3.1.1 Orensteins Algorithmus

In diesem Abschnitt behandeln wir den Joinalgorithmus von Orenstein [Ore86, Ore91] (im folgenden *ORE* bezeichnet). Dieses Verfahren basiert auf einer binären rekursiven Partitionierung ($n = 2$), siehe Definition 7, wobei der binäre Code die sogenannte *Z-Ordnung* repräsentiert.

ORE ordnet jedem Hypercube der Eingaberelationen eine Menge disjunkter Teilräume der rekursiven Partitionierung zu, wobei die Vereinigung dieser Teilräume den Hypercube vollständig überdeckt. ORE sortiert die beiden so erzeugten Relationen dann bezüglich der binären Codes ihrer Elemente (lexikographische Ordnung). Danach werden die beiden Relationen mit Hilfe von 2 Stacks *Stack_R* und *Stack_S* verbunden (siehe Abbildung 3.1). Bei der Verarbeitung wird sichergestellt, daß für zwei benachbarte Hypercubes im Stack die Präfixeigenschaft erfüllt ist für ihre zugehörigen Codes. Diese Präfixeigenschaft wird in der dritten Zeile des Algorithmus sichergestellt. Aus diesem Grund brauchen nur diejenigen Hypercubes verknüpft werden (Zeile 5 und 9), deren Codes dasselbe Präfix haben.

Ein Nachteil des Verfahrens von Orenstein ist, daß die in [Ore91] untersuchten verschiedenen Zuordnungsstrategien sehr hohe Replikationsraten zur Folge haben. D.h. die Anzahl der zu

```
1  Solange (beide sortierten Sequenzen nicht leer) {
2        Wähle Hypercube H(x) mit kleinstem Code
           von den Eingaben;
3        Stelle Präfixeigenschaft für Stack_R und Stack_S sicher;
4        Falls (H(x) ist Hypercube aus R) {
5              Stack_S.query(H(x));
6              Stack_R.insert(H(x));
7        }
8        Sonst { //H(x) ist Hypercube aus S
9              Stack_R.query(H(x));
10             Stack_S.insert(H(x));
11       }
12 }
```

Abbildung 3.1: Orensteins Merge-Algorithmus

```
Für beide Datenmengen:
    Für jeden Hypercube:
        Berechne Code und Ebene l des Hypercube;
        Schreibe Hypercube in Ebenendatei l;
    Für jede Ebenendatei:
        Sortiere Ebenendatei bezüglich der lexikographischen Ordnung der Z-Werte;
    Führe ein synchronisierten Einlesen der Ebenendateien durch und
      rufe Orensteins Merge-Algorithmus;
```

Abbildung 3.2: Der MSJ-Algorithmus

verarbeitenden Elemente steigt stark an und folglich steigen die Kosten für das externe Sortieren der Daten. Ein weiterer Nachteil von ORE ist, daß keine Methode zum effizienten Löschen von Duplikaten der Ergebnismenge bereitgestellt wurde.

3.1.2 Der MSJ-Algorithmus

MSJ geht ähnlich vor wie ORE, mit zwei wesentlichen Unterschieden: zum ersten ist Replikation von Daten nicht erlaubt, zum zweiten wird eine I/O-Strategie basierend auf sogenannten *Ebenendateien (Level-Files)* eingesetzt. Außerdem wird eine n-äre rekursive Partitionierung genutzt wobei $n = 2^d$ (Quadtree-Partitionierung).

Zunächst wollen wir für MSJ betrachten, wie ein Hypercube einem Teilraum der rekursiven

Partitionierung zugeordnet wird. Unter den Teilräumen, die den Hypercube umschließen, wählen wir den kleinsten (längster möglicher Code).

Der Algorithmus partitioniert die Hypercubes zunächst in Ebenendateien (siehe Abbildung 3.2). Folglich wird ein Hypercube der Ebene l in der l-ten Ebenendatei gespeichert. Danach werden die Ebenendateien bezüglich der lexikographischen Ordnung der Codes sortiert. Die Ebenendateien werden dann synchron gelesen und für die Elemente wird der Merge-Algorithmus von Orenstein gerufen.

Ein wesentlicher Nachteil dieses Verfahrens für hochdimensionale Daten ist, daß ein großer Teil der Eingabedaten der obersten Ebene 0 zugeordnet wird [BK01]. Die Hypercubes aus Ebene 0 müssen aber gegen die komplette Eingaberelation der anderen Datenmenge getestet werden — was hohe CPU-Kosten verursacht.

In einer früheren Arbeit [DS00] haben wir für zwei Dimensionen bereits experimentell gezeigt, daß kontrollierte Replikation die Ausführungszeit von MSJ erheblich verringert.

3.2 Generic External Space Sweep

In diesem Abschnitt stellen wir unseren Similarity Join Algorithmus *Generic External Space Sweep (GESS)* vor. GESS basiert auf einem generischen Replikationsalgorithmus, der dazu dient, die Anzahl der teuren Distanzberechnungen zwischen Merkmalsvektoren zu reduzieren. Mögliche Duplikate in der Antwortmenge werden durch eine effiziente online Technik entdeckt und eliminiert. Unser Algorithmus geht in drei Phasen vor, die wir im folgenden zunächst kurz erklären und dann jeweils in einem eigenen Unterabschnitt erläutern.

Seiten R und S zwei Mengen von Merkmalsvektoren. In der ersten Phase des Verfahrens wird jeder Vektor aus R (bzw. S) in einen Hypercube transformiert und an den *Replikationsalgorithmus* weitergereicht (siehe Abschnitt 3.2.1).

Der Replikationsalgorithmus erzeugt Codes, die die Teilräume des Hypercube repräsentieren. Die Hypercubes werden dann direkt an den Sortieroperator weitergeleitet, der die Elemente bezüglich der lexikographischen Ordnung ihrer Codes sortiert. Wir partitionieren die Hypercubes dabei nicht physikalisch in verschiedene Ebenendateien (Level-Files) sondern überlassen die I/O-Strategie dem zugrundeliegenden Sortieroperator (siehe Abschnitt 3.2.2). Danach verknüpfen wir die sortierten Daten mit Hilfe von Orensteins Merge-Algorithmus (siehe Abbildung 3.1). Der Algorithmus gibt die Ergebnistupel des Verfahrens direkt an die *Referenzpunktmethode* weiter (siehe Abschnitt 3.2.3), die Duplikate sofort erkennt und

```
Für beide Datenmengen:
    Für jeden Hypercube:
        Rufe Replikationsalgorithmus und
            berechne Codes;
    Gebe Codes direkt an den Sortieroperator weiter;
    Rufe Orensteins Merge-Algorithmus;
    Für jedes Ergebnistupel:
        Rufe Referenzpunktmethode;
```

Abbildung 3.3: Der GESS-Algorithmus

eliminiert. Unser Algorithmus ist zusammengefaßt in Abbildung 3.3.

3.2.1 Der Replikationsalgorithmus

Aus Gründen der Übersicht betrachten wir zunächst eine binäre rekursive Partitionierung ($n = 2$). Später gehen wir dann zu einer Quadtree-Partitionierung über, wie wir sie für die Simulation nutzen. Abbildung 3.4 zeigt den Algorithmus in Pseudocode. Der Replikationsalgorithmus erwartet einen Hypercube $H(x)$ und einen Teilraum D als Eingabe. Beim ersten Aufruf entspricht D dem gesamten Datenraum.

Der Algorithmus überprüft ob $H(x) \subseteq D_0 \vee H(x) \subseteq D_1$ gilt. Falls das der Fall ist, wird der Replikationsalgorithmus rekursiv mit dem umschließenden Teilraum gerufen. Anderenfalls (d.h. wenn $H(x) \not\subseteq D_0 \wedge H(x) \not\subseteq D_1$) wird das benutzerdefinierte Prädikat *split_erlaubt* ausgewertet. Falls das Prädikat *true* zurückliefert, wird $H(x)$ in zwei Replikate $H(x) \cap D_0$ und $H(x) \cap D_1$ aufgeteilt. Der Algorithmus wird dann rekursiv für jedes Replikat gerufen. Gibt das Prädikat *false* zurück, stoppt der Partitionierungsprozeß für den entsprechenden Teilast und der Code, der den aktuellen Teilraum repräsentiert, wird zurückgegeben.

Unser Replikationsalgorithmus basiert auf einem benutzerdefinierten Prädikat *split_erlaubt*, welches die Aufteilungsstrategie bestimmt. Diese Strategie ist nicht festgeschrieben und kann — sogar zur Laufzeit — geändert werden.

Für die folgende Analyse nehmen wir eine Quadtree-Partitionierung an ($\mathrm{Part}(D^l) = \{D^l\} \cup \bigcup_{0 \le i < 2^d} \mathrm{Part}(D_i^{l+1})$). Wir konzentrieren uns dabei auf folgende wichtige Strategien:

Strategie 1 (Maximale Anzahl von Aufteilungslinien). *Aufteilung eines Hypercube $H(x)$ ist dann erlaubt, wenn nicht mehr als k Hyperebenen von $H(x)$ der aktuellen Quadtree-Ebene geschnitten werden.*

```
1  Prädikat split_erlaubt; //aktuelle Aufteilungsstrategie

2  Methode repliziere(Hypercube H(x), Teilraum D){
3        Part(D) := {D} ∪ D0 ∪ D1;
4        Falls ( H(x) ⊆ D0) {
5              repliziere( H(x), D0 );
6        }
7        Sonst Falls ( H(x) ⊆ D1) {
8              repliziere( H(x), D1 );
9        }
10       Sonst Falls ( split_erlaubt( H(x), Part(D) ) ) {
               //Teile H(x) in zwei Replikate auf:
11             repliziere( H(x) ∩ D0, D0 );
12             repliziere( H(x) ∩ D1, D1 );
13       }
14       Sonst {
15             return (H(x), Code_D(D));
16       }
17 }
```

Abbildung 3.4: Der Replikationsalgorithmus von GESS

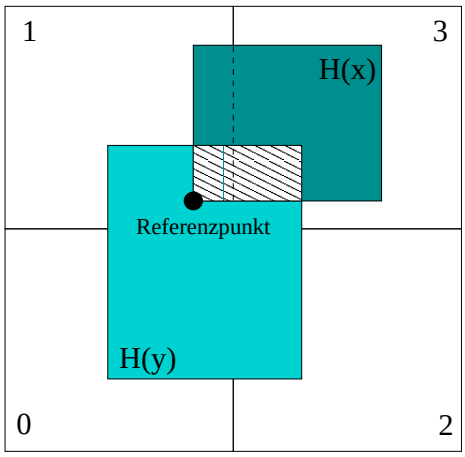


Abbildung 3.5: Die Referenzpunktmethode (RPM)

Strategie 2 (Maximale Aufteilungsebene). *Aufteilung eines Hypercube $H(x)$ ist erlaubt, falls die aktuelle Ebene l kleiner als eine gegebene Schranke ist, d.h. $l \leq msl$, wobei $msl \in [0, \ldots, mL]$ die maximale Aufteilungsebene ist. mL bezeichne den Index der tiefsten Partitionierungsebene.*

Strategie 3 (Kompositionsstrategie). *Aufteilung eines Hypercube $H(x)$ ist erlaubt, falls die Bedingungen aus Strategie 1 und Strategie 2 gelten.*

3.2.2 Warum wir keine Ebenendateien brauchen

Wichtig für den Merge-Algorithmus von Orenstein (siehe Abbildung 3.1) ist die lexikographische Ordnung der Z-Werte. Eine physikalische Partitionierung in Ebenendateien (Level-Files) — so wie sie von MSJ vorgenommen wird — ist nicht notwendig. Um Orensteins Merge-Algorithmus für große Datenmengen nutzbar zu machen, nutzen wir einen vorhandenen Sortieroperator. Unser Operator implementiert externes Merge-Sort basierend auf Replacement Selection [Knu73] für das Erzeugen der Runs. Das Sortierkriterium ist modifiziert, um die Daten bezüglich der lexikographischen Ordnung der Merkmalsvektoren zu sortieren. Die letzte Mischoperation wird online durchgeführt [NP85], d.h. die vollständig sortierten Daten werden nicht auf den Externspeicher geschrieben, sondern direkt im Hauptspeicher zusammengeführt und an den Joinalgorithmus weitergereicht.

3.2.3 RPM: Die Referenzpunktmethode

Da GESS Replikation von Hypercubes zuläßt, müssen wir betrachten, wie mögliche Duplikate in der Antwortmenge entdeckt und gelöscht werden können.

Betrachten wir zunächst die Situation in Abbildung 3.5: der Hypercube $H(x)$ wird in die Partitionen 1 und 3 der Ebene 1 repliziert, $H(y)$ befindet sich in Ebene 0. Beide Replikate von $H(x)$ müssen in der Joinphase mit $H(y)$ verglichen werden. Dies bedeutet, daß das Tupel (x, y) zweimal als Ergebnistupel des Joins berichtet werden würde.

Es gibt zwei prinzipielle Vorgehensweisen, um Duplikate zu eliminieren. Die erste ist, eine Hashtabelle zu nutzen, die alle Ergebnistupel des Joins speichert. Der Speicherbedarf dieses Verfahrens ist $O(Z)$, wobei Z die Anzahl der Ergebnistupel bezeichne. Die zweite Möglichkeit ist, die Ergebnistupel des Joins zu sortieren [PD96]. Dies hat zum einen zusätzliche I/O-Kosten zur Folge, zum anderen kann eine solche Sortieroperation erst dann Ergebnistupel produzieren, wenn alle Ergebnistupel des Merge-Algorithmus vorliegen.

Wir schlagen deshalb eine effiziente online Methode namens *Referenzpunktmethode (RPM)* vor. Diese Methode benötigt weder zusätzlichen Speicher, noch verursacht sie zusätzliche I/O-Operationen. Seien $x \in R$, $y \in S$ zwei Merkmalsvektoren, wobei gilt $H(x) \cap H(y) \neq \emptyset$. Die Grundidee von RPM ist es, einen Referenzpunkt rp zu definieren, mit

$$rp := f_{rp}\big(H(x), H(y)\big) \in H(x) \cap H(y),$$

D.h., der Referenzpunkt ist im Schnitt von $H(x)$ und $H(y)$ enthalten. Die Referenzpunktmethode geht dann folgendermaßen vor:

Definition 9 (RPM). *Sei H ein Hypercube mit der tiefsten Ebene (längster Code) von $H(x)$ und $H(y)$. Ein Ergebnistupel (x, y) wird berichtet von RPM, wenn*

$$\Longleftrightarrow \mathit{Code}_D(H) \textit{ ist Präfix von } \mathit{Code}_D\Big(f_{rp}\big(H(x), H(y)\big)\Big).$$

Betrachten wir noch einmal die Situation in Abbildung 3.5. Es gilt $H = H(x)$, da $H(x)$ auf einer tieferen Partitionierungsebene liegt. Der Code des Referenzpunktes rp ist

$$\mathrm{Code}_D\Big(f_{rp}\big(H(x), H(y)\big)\Big) =< 1.. > .$$

$H(x)$ wurde repliziert und wird deswegen von zwei Codes, $< 1 >$ und $< 3 >$, repräsentiert. Da $< 1 >$ Präfix von $< 1.. >$ ist, wird das Ergebnistupel für dieses Replikat berichtet. Code $< 3 >$ hingegen ist kein Präfix von $< 1.. >$. Folglich berichtet RPM dieses Ergebnistupel nicht. D.h. insgesamt wird das Ergebnistupel genau einmal berichtet.

Theorem 1 (Korrektheit von RPM). *Die Referenzpunktmethode berichtet jedes Ergebnistupel des Joins genau einmal.*

Beweis Sei Part(D) eine n-äre rekursive Partitionierung von D. Sei (x, y) ein Ergebnistupel des Joins, wobei $H(x)$ und $H(y)$ die zugehörigen Hypercubes sind. Der Replikationsalgorithmus von GESS berechnet eine Menge disjunkter Teilräume, $Rep(x) \subseteq \text{Part}(D)$ und $Rep(y) \subseteq \text{Part}(D)$ für jeden Hypercube $H(x)$ und $H(y)$. Die Elemente von $Rep(x)$ und $Rep(y)$ sind konservative Approximationen von $H(x)$ und $H(y)$.

Da $\forall\, A, B \in Rep(x), A \neq B, A \cap B = \emptyset$ und $\forall\, A, B \in Rep(y), A \neq B, A \cap B = \emptyset$, folgt, daß der Referenzpunkt nur in einem der Teilräume von $Rep(x)$ und $Rep(y)$ liegen kann. Hieraus folgt, daß nur ein Paar existieren kann $(A, B), A \in Rep(x), B \in Rep(y)$, für welches gilt

$$A \cap B \neq \emptyset \wedge rp \in A \cap B.$$

Da $A, B \in \text{Part}(D)$, folgt, daß $A \subseteq B$ oder $B \subseteq A$ gelten muß. Sei o.B.d.A. $A \subseteq B$. Dann folgt

$$\begin{aligned}
& rp \in H(x) \cap H(y) \\
\Longleftrightarrow\ & rp \in A \cap B \\
\Longleftrightarrow\ & rp \in A \subseteq B \\
\Longleftrightarrow\ & \begin{cases} & \text{Code}_D(B) \text{ ist Präfix von Code}_D(A) \\ \wedge\ & \text{Code}_D(A) \text{ ist Präfix von Code}_D(rp) \end{cases}
\end{aligned}$$

□

3.3 CPU-Kosten Analyse

Im folgenden betrachten wir die CPU-Kosten des Intersection Joins für MSJ und GESS.

Beide Verfahren partitionieren die Eingaberelationen R und S mit Hilfe einer rekursiven Partitionierung ($n = 2^d$, Quadtree-Partitionierung). Mit R_i (bzw. S_i) bezeichnen wir die Teilmenge von Daten aus R (bzw. S), die in die Ebene i partitioniert wird. Da Hypercubes repliziert werden können, enthält die Summe $\sum_{i=0}^{mL} R_i$ (S_i) für gewöhnlich mehr Elemente als R (bzw. S) für GESS. Die CPU-Kosten der Verfahren werden dominiert durch die Anzahl der für die Joinphase benötigten Distanzberechnungen.

Lemma 2. *Die durchschnittliche Anzahl der Distanzberechnungen DC für einen Intersection Join von R und S ist gegeben durch*

$$DC = \sum_{i=0}^{mL} \left(||R_i|| \cdot \left[\sum_{l=0}^{i} \frac{||S_l||}{n^l} + \frac{1}{n^i} \sum_{l=i+1}^{mL} ||S_l|| \right] \right). \tag{3.3}$$

Beweis R_i (S_i) bezeichne alle Elemente aus R (S) die in die Ebene i partitioniert werden ($0 \leq i \leq mL$). Sei x ein beliebiges Element aus R_i. x muß mit allen Elementen aus S verknüpft werden, die dasselbe Präfix haben. Es entstehen 2 Fälle:

1. **Fall:** Ebene $0 \leq l \leq i$: pro Ebene entstehen mit dem Element $||S_l||/n^l$ Vergleiche.
2. **Fall:** Ebene $i < l \leq mL$: pro Ebene entstehen mit dem Element $||S_l||/n^i$ Vergleiche.

Durch Aufsummieren ergibt sich obige Gleichung. □

Um die Anzahl der Distanzberechnungen DC bestimmen zu können, müssen wir zunächst ermitteln, wieviele Hypercubes auf den verschiedenen Ebenen liegen. Wir nehmen in unserem Modell an, daß die Hypercubes vollständig im Einheitskubus $[0,1)^d$ enthalten sind. Diese Eigenschaft ist für das Similarity Join Problem momentan noch nicht erfüllt. Durch eine einfache lineare Transformation der Merkmalsvektoren können wir jedoch das gegebene Joinproblem in ein äquivalentes Joinproblem, das die Bedingung erfüllt, umwandeln. Wir transformieren jede Komponente x_i eines Merkmalsvektors x wie folgt:

$$x_i' = \frac{\varepsilon}{2} + x_i \cdot (1 - \varepsilon), (1 \leq i \leq d).$$

Desweiteren modifizieren wir das Joinprädikat des Intersection Joins:

$$|x_i - y_i| \leq (1 - \varepsilon) \cdot \varepsilon, (1 \leq i \leq d).$$

Aufgrund dieser Transformation des Similarity Join in einen Intersection Join können wir uns im folgenden auf die Diskussion von Intersection Joins beschränken. Randeffekte treten nicht auf [BBKK97].

Wir machen für die nachfolgende Analyse die Annahme, daß die Hypercubes iid-verteilt sind in $[0;1)^d$.

Da die Größe eines Hypercubes beschränkt ist durch ε in jeder Dimension, ist die maximale Partitionierungsebene beschränkt durch $mL = \lceil -\log \varepsilon \rceil - 1 \geq l \geq 0$.

3.3.1 Analytisches Modell von MSJ

Die Wahrscheinlichkeit $P_{NC}(,,\text{trifft } l\text{“})$, daß ein Hypercube eine Hyperebene der Ebene l trifft, ist gegeben durch

$$P_{NC}(,,\text{trifft } l\text{“}) = 1 - \left(\frac{1 - 2^{l+1}\varepsilon}{1 - 2^l \varepsilon}\right)^d. \tag{3.4}$$

Für die Ebene $l \geq 1$ brauchen wir nur diejenigen Hypercubes zu betrachten, die die darüberliegenden Ebenen $0, \ldots, l-1$ passiert haben. Die *bedingte* Wahrscheinlichkeit $p_l := P_C(,,\text{trifft } l\text{“})$, daß ein Hypercube eine Hyperebene der Ebene l trifft, ist dann gegeben durch

$$p_l = \begin{cases} P_{NC}(,,\text{trifft } l\text{“}) \cdot \left(1 - \sum_{i=0}^{l-1} P_{NC}(,,\text{trifft } i\text{“})\right) & , l < mL \\ 1 - \sum_{i=0}^{l-1} P_{NC}(,,\text{trifft } i\text{“}) & , l = mL \end{cases} \tag{3.5}$$

3.3.2 Analytisches Modell von GESS

In diesem Abschnitt untersuchen wir die Effekte von Replikation in GESS. Wir verwenden hierfür die Replikationsstrategie 1 für ein gegebenes k, $1 \leq k \leq d$. D.h., wann immer ein Hypercube q Hyperebenen ($q \leq k$) in Ebene l schneidet, wird der Hypercube aufgeteilt in 2^q Teile, so daß jeder Teil vollständig in einem Teilraum enthalten ist. Diese Teile ‚fallen' weiter zur Ebene $l+1$, wo sie erneut betrachtet werden. Wichtig für unsere Analyse ist, daß jeder Hypercube pro Dimension nie mehr als eine Hyperebene schneiden kann.

Sei l die aktuelle Ebene. Wir suchen die Zufallsvariable X_l, die die Anzahl der geschnittenen Hyperebenen in Ebene l für einen gegebenen Hypercube zurückgibt. Die nichtbedingte Wahrscheinlichkeit p, eine Hyperebene einer Dimension der Ebene l zu treffen, ist gegeben durch

$$p = 1 - \frac{1 - 2^{l+1}\varepsilon}{1 - 2^l \varepsilon}.$$

Die nichtbedingte Wahrscheinlichkeit, genau q Hyperebenen in Ebene l zu treffen, ist binomial verteilt:

$$P_{NC}(X_l = q) = b_{d,p}(q) = \binom{d}{q} \cdot (1-p)^q \cdot p^{d-q}.$$

Die Wahrscheinlichkeit, mehr als k Hyperebenen zu treffen, ist gegeben durch

$$P_{NC}(X_l > k) = \sum_{i=k+1}^{d} P_{NC}(X_l = i).$$

Für $d \to \infty$ und $d \cdot p \to \lambda$, kann $b_{d,p}(k)$ durch eine Poisson-Verteilung mit Mittelwert λ approximiert werden. Wir erhalten dann folgende Approximation:

$$P_{NC}(X_l > k) \approx \frac{(p \cdot d)^{k+2}}{(k+2)!}. \tag{3.6}$$

Dies bedeutet, die Wahrscheinlichkeit, mehr als k Hyperebenen zu treffen, fällt exponentiell mit steigendem k. Man beachte, daß $p \cdot d < 1$ gilt, da wir annehmen, daß pro Dimension ein Hypercube höchstens eine Hyperebene schneiden kann.

Für die Berechnung der Anzahl der Hypercubes in Ebene l dürfen wir nur diejenigen Hypercubes berücksichtigen, die die darüberliegenden Ebenen $0, \ldots, l-1$ passiert haben, d.h. wir müssen übergehen zu bedingten Wahrscheinlichkeiten.

Aus Gründen der Übersichtlichkeit nehmen wir zunächst an, daß Replikation die Anzahl der Elemente in den tieferen Ebenen nicht erhöht. Die bedingte Wahrscheinlichkeit, mehr als k Hyperebenen in Ebene l zu treffen, ist gegeben durch

$$P(X_l > k) = P_{NC}(X_l > k | X_{l-1} \leq k | \ldots | X_0 \leq k). \tag{3.7}$$

Da die Ereignisse auf den verschiedenen Ebenen unabhängig sind, kann die bedingte Wahrscheinlichkeit auf der rechten Seite von Gleichung 3.7 durch ein Produkt dargestellt werden. Insgesamt bekommen wir die folgende Gleichung:

$$P(X_l > k) = \begin{cases} P_{NC}(X_l > k) \left(1 - \sum_{i=0}^{l-1} P(X_i > k)\right) & , l < mL \\ 1 - \sum_{i=0}^{l-1} P(X_i > k) & , l = mL \end{cases} \tag{3.8}$$

Die erwartete Anzahl von Hypercubes in Ebene l ist dann gegeben durch

$$||R|| \cdot P(X_l > k). \tag{3.9}$$

Als nächstes erweitern wir unser Modell und berücksichtigen zusätzlich, daß Replikation die Anzahl der Elemente in tieferen Ebenen erhöht. Im allgemeinen ist es schwierig, eine exakte Formel für die Verteilung unter Replikation zu bekommen, da die Verteilungen auf den verschiedenen Ebenen nicht mehr unabhängig sind. Deswegen sind wir an einer einfachen aber akkuraten Schätzung interessiert und machen hierfür folgende Annahmen: Erstens nehmen wir an, daß die replizierten Hypercubes die gleiche Verteilung haben wie die nicht replizierten Hypercubes. Zweitens nehmen wir an, daß die Replikate die gleiche Größe haben wie die Ausgangshypercubes. Letztere Annahmen vereinfachen die mathematische Handhabung unseres Modells, liefern aber trotzdem sehr genaue Voraussagen[1]. Man beachte, daß die letzten beiden Annahmen in Bezug auf die zu berechnende Anzahl der Distanzberechnungen DC (Formel 3.3) pessimistisch sind. Unser Modell berechnet somit eine obere Schranke für die Anzahl der Distanzberechnungen von GESS.

Sei $\hat{r}_l$ unsere Schätzung für die Anzahl der Hypercubes der Ebene l. Desweiteren sei $\hat{r}_{>l}$ die geschätzte Anzahl von Hypercubes, die die Ebene l passiert haben. Wir setzen $\hat{r}_{-1} = ||R||$. Dann sind unsere Schätzer $\hat{r}_l$ und $\hat{r}_{>l}$ rekursiv definiert durch die folgenden Formeln:

$$\begin{aligned} \hat{r}_l &= \hat{r}_{>l-1} \cdot P_{NC}(X_l > k) && , l < mL \\ \hat{r}_{>l} &= \hat{r}_{>l-1} \cdot \sum_{i=0}^{k} 2^i P_{NC}(X_l = i) && , l < mL \end{aligned} \tag{3.10}$$

[1] siehe Abschnitt 3.6.1

Außerdem setzen wir $\hat{r}_{>l}$ und $\hat{r}_l = \hat{r}_{>l-1}$ für $l = mL$. In Abschnitt 3.6.1 werden wir Ergebnisse von Simulationen mit gleichverteilten Daten diskutieren, die bestätigen, daß unsere Schätzung sehr genau ist.

3.4 I/O-Kosten Analyse

In diesem Abschnitt betrachten wir die I/O-Kosten von MSJ und GESS. R und S seien die Eingaberelationen. Die I/O-Kosten ergeben sich aus der Anzahl der Lese- und Schreiboperationen zwischen Haupt- und Externspeicher.

Wir nehmen an, daß höchstens m Seiten im Speicher gehalten werden können. Eine Lese- bzw. Schreiboperation entspricht einer I/O-Operation. Eine wesentliche Idee von MSJ ist, die Daten aus R und S in Ebenendateien $R_0, R_1, \ldots$ bzw. $S_0, S_1, \ldots$ aufzuteilen. Für eine Partition A einer Relation bezeichne $|A|$ die Anzahl der Seiten in A. Insbesondere bezeichnen $|R|$ und $|S|$ die Anzahl der Seiten in den Eingaberelationen R bzw. S.

Zunächst führen wir die Analyse ohne Annahmen über die Verteilung der Hypercubes durch. Später erweitern wir die Analyse und nehmen an, daß die Hypercubes iid-verteilt sind im Einheitswürfel. D.h. die ‚linken unteren' Kanten der Hypercubes sind gleichmäßig verteilt in $[0, 1-\varepsilon)^d$, und die verschiedenen Dimensionen sind unabhängig verteilt.

3.4.1 I/O-Kosten von MSJ

Wir nehmen in unserem Modell an, daß zwei Operatoren die Eingabedaten liefern [Gra94b]. Aus diesem Grund zählen wir die Kosten für das erste Einlesen der Daten nicht mit.

MSJ berechnet zunächst den Hilbertwert [Hil91] für jeden Merkmalsvektor. Die entstehenden Tupel bestehend aus dem Merkmalsvektor und dem Hilbertwert werden dann in die entsprechende Ebenendatei geschrieben. Dieser erste Schritt benötigt $\sum_{i=0}^{mL}(|R_i| + |S_i|) \approx |R| + |S|$ I/O-Operationen wobei mL den Index der tiefsten Partitionierungsebene bezeichnet. Im folgenden Sortierschritt wird jede Ebenendatei vom Externspeicher gelesen und in sortierte Runs aufgeteilt. Dies erfordert

$$\sum_{i=0}^{mL} 2(|R_i| + |S_i|) \approx 2(|R| + |S|)$$

I/O-Operationen. Die Anzahl der Mischoperationen für Ebenendatei R_i (bzw. S_i) ergibt

sich durch

$$f_{\text{mo}}(R_i) = \begin{cases} \lceil \log_{m-1}(|R_i|/m) \rceil & , |R_i|/m > 1 \\ 0 & , |R_i|/m \leq 1 \end{cases}$$

Jede Mischoperation kostet eine I/O-Operation für das Lesen sowie eine I/O-Operation für das Schreiben je Seite des Eingabe-Runs. Die Gesamtkosten für die Ebenendatei R_i sind dann näherungsweise $2|R_i|$. Insgesamt erfordert das externe Sortieren der Ebenendateien

$$2 \cdot \sum_{i=0}^{mL} \Big((1 + f_{\text{mo}}(R_i)) \cdot |R_i| + (1 + f_{\text{mo}}(S_i)) \cdot |S_i| \Big)$$

$$= 2(|R| + |S|) + 2 \sum_{i=0}^{mL} \Big(f_{\text{mo}}(R_i) \cdot |R_i| + f_{\text{mo}}(S_i) \cdot |S_i| \Big)$$

I/O-Operationen. In der Joinphase des Algorithmus müssen die verschiedenen Ebenendateien gelesen und gemischt werden. Damit ergeben sich die Gesamtkosten von MSJ durch

$$MSJ_{\text{I/O-Kosten}} = 4(|R| + |S|) + \underbrace{2 \sum_{i=0}^{mL} f_{\text{mo}}(R_i)|R_i| + f_{\text{mo}}(S_i)|S_i|}_{=:\Delta_{\text{MSJ}}} . \tag{3.11}$$

Dies bedeutet, daß MSJ mindestens 4 I/O-Operationen für jede Seite der Eingaberelationen benötigt sowie zusätzliche I/O-Operationen für Mischoperationen des Externen Sortierens.

Um die Kosten für Mischoperationen von Runs abschätzen zu können — siehe Term Δ_{MSJ} in Gleichung 3.11 — benötigen wir Informationen über die Verteilung der Hypercubes in den verschiedenen Ebenen. Mittels Gleichung 3.5 können wir die erwarteten I/O-Kosten von MSJ schätzen. Abbildung 3.6 zeigt die zusätzlichen I/O-Kosten, die für MSJ durch Mischoperationen des externen Sortierens entstehen. Auf der x-Achse ist die Puffergröße abgetragen. Die y-Achse zeigt die zusätzlichen I/O-Operationen normiert auf die Eingabegröße. D.h. für einen y-Wert von 1 entstehen insgesamt genau $|R| + |S|$ zusätzliche I/O-Operationen.

In unseren Experimenten haben wir festgestellt, daß der Term Δ_{MSJ} größer 0 war für Datensätze, die ca. 10-mal größer als der verfügbare Hauptspeicher sind.

3.4.2 I/O-Kosten von GESS

In diesem Abschnitt betrachten wir die I/O-Kosten von GESS. Zunächst ermitteln wir die I/O-Kosten ohne Berücksichtigung von Replikation. Später erweitern wir dann unser Modell.

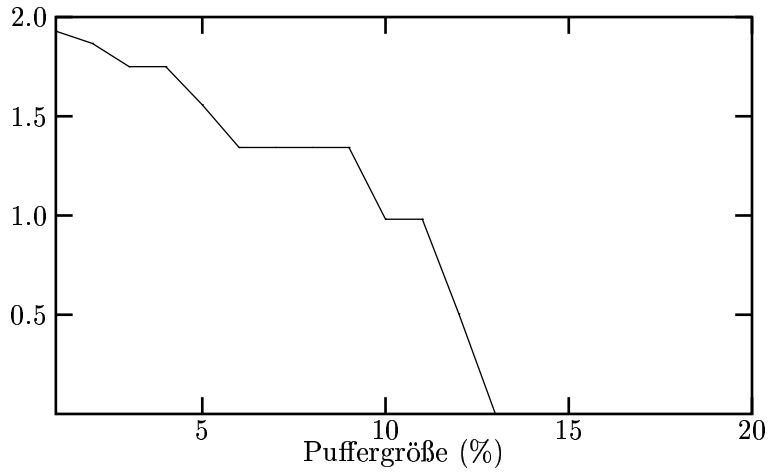


Abbildung 3.6: Zusätzliche I/O-Kosten von MSJ für Mischoperationen (untere Schranke)

Jede der beiden Eingaberelationen wird von dem Replikationsalgorithmus an einen Sortieroperator weitergereicht. Diese Operatoren erzeugen zunächst sortierte Runs. Dieses erfordert $|R| + |S|$ I/O-Operationen. Die Gesamtzahl der Mischoperationen ist wieder gegeben durch

$$f_{\text{mo}}(R) = \begin{cases} \lceil \log_{m-1}(|R|/m) \rceil & , |R|/m > 1 \\ 0 & , |R|/m \leq 1. \end{cases}$$

Man beachte, daß die vollständigen sortierten Runs der letzten Mischoperationen nicht auf den Externspeicher geschrieben, sondern direkt an den nächsten Operator weitergereicht werden [NP85]. Desweiteren verursachen Mischoperationen vor dem letzten online Mischen I/O-Kosten. Die Anzahl dieser Mischoperationen ist

$$\bar{f}_{\text{mo}}(R) = \max(f_{\text{mo}}(R) - 1, 0).$$

Die gesamten I/O-Kosten von GESS sind dann

$$GESS_{\text{I/O-Kosten}} = 2\,(|R| + |S|) + \underbrace{2\left(\bar{f}_{\text{mo}}(R) \cdot |R| + \bar{f}_{\text{mo}}(S) \cdot |S|\right)}_{=:\Delta_{\text{GESS}}}. \tag{3.12}$$

Dies bedeutet, daß GESS mindestens zwei I/O-Operationen für jede Eingabeseite der Eingaberelation benötigt. Mögliche zusätzliche Mischoperationen (siehe Term Δ_{GESS} in Gleichung 3.12) verursachen weitere Kosten.

Im folgenden zeigen wir, daß der Term Δ_{GESS} mit hoher Wahrscheinlichkeit Null ist für die meisten realistischen Situationen. GESS benötigt Hauptspeicher für die beiden Sortieroperatoren und für die eigentliche Joinoperation. Man beachte, daß während der Run-Erzeugung

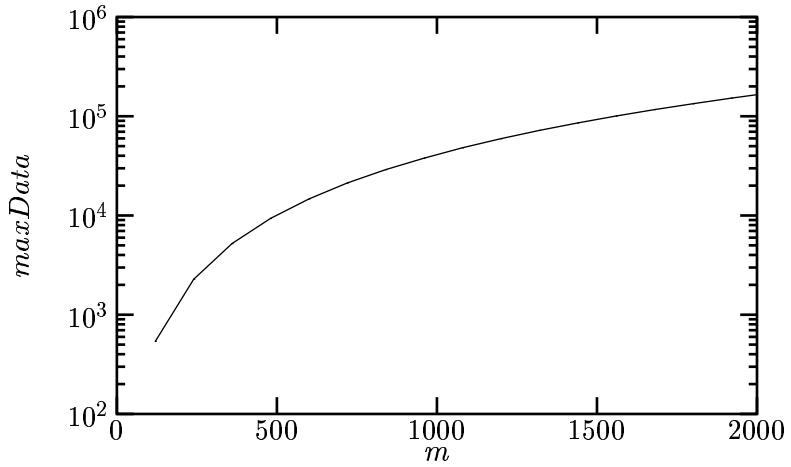


Abbildung 3.7: Maximale Größe der Eingaberelationen ohne Nutzen von Mischoperationen

und den Mischoperationen der open-Phase ein einzelner Sortieroperator zunächst den kompletten Hauptspeicher nutzen kann. Unsere Sortieroperatoren basieren auf Replacement Selection [Knu73], welches sortierte Runs der durchschnittlichen Länge $2m$ erzeugt. Während der letzten online Mischoperation müssen die Sortieroperatoren den Hauptspeicher teilen. Sei $m_{\text{Sorter R}}$ der verfügbare Hauptspeicher (in Seiten) für den Sortieroperator für Eingaberelation R während der next-Phase [Gra94b] des Joins (wobei $m_{\text{Sorter R}} + m_{\text{Sorter S}} < m$). Der maximale Fanin der letzten online Mischoperation ist $m_{\text{Sorter R}} - 1$. Wenn wir sortierte Eingabe-Runs durchschnittlicher Größe $2m$ annehmen, läßt sich schließen, daß Eingaberelationen der Größe

$$max_{data} = (m_{\text{Sorter R}} - 1)\, 2m$$

sortiert werden können, ohne zusätzliche Mischoperationen in der open-Phase zu benötigen. Abbildung 3.7 zeigt max_{data} als eine Funktion von m ($m_{\text{Sorter R}} = m/3$). Der Graph zeigt, daß sogar für kleine Werte von m sehr große Datenmengen sortiert werden können ohne daß zusätzliche Mischoperationen erforderlich sind. Z.B. können mit nur 2 MB Hauptspeicher bis zu 690 MB große Relationen sortiert werden.

Wir schließen hieraus, daß für viele praktische Situationen GESS keine zusätzliche Mischoperationen benötigt. Für diese Fälle sind die I/O-Kosten von GESS

$$GESS_{\text{I/O-Kosten}} \approx 2(|R| + |S|) \tag{3.13}$$

ohne Berücksichtigung von Replikation. Abbildung 3.8 zeigt den Verbesserungsfaktor

$$IF_{\text{I/O-Kosten}} = \frac{MSJ_{\text{I/O-Kosten}}}{GESS_{\text{I/O-Kosten}}}$$

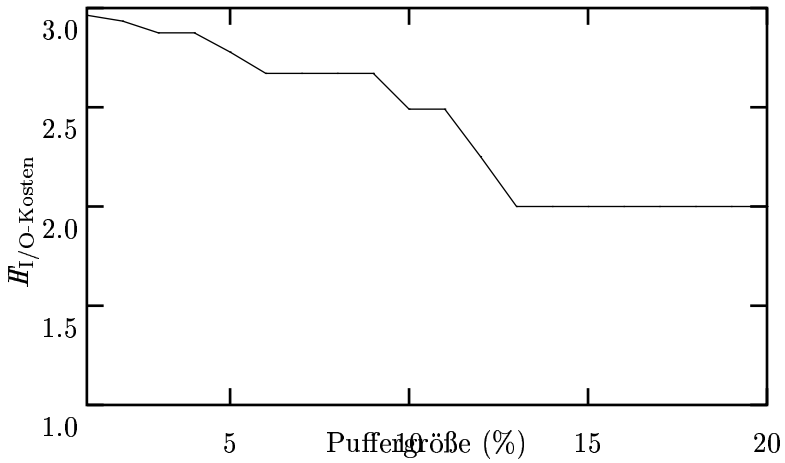


Abbildung 3.8: Verbesserung von GESS gegenüber MSJ bezüglich I/O-Kosten

von GESS gegenüber MSJ. GESS benötigt höchstens die Hälfte der I/O-Operationen für alle Puffergrößen. Falls der Puffer kleiner ist als 10% der Datenmenge, ist GESS sogar 2.5 mal besser. Für sinkende Puffergrößen steigt die Verbesserung auf 2.7 und mehr.

Im folgenden berücksichtigen wir die durch Replikation entstehenden zusätzlichen I/O-Kosten. Sei $||R||$ die Anzahl der Elemente der Eingabemenge R. Durch Replikation werden Elemente aus R mehrfach abgespeichert. D.h. aus R entsteht die Multimenge $\tilde{R}$ mit $||\tilde{R}|| \geq ||R||$. Wir definieren den Replikationsfaktor w durch

$$w = \frac{||\tilde{R}||}{||R||}.$$

D.h. w beschreibt das Verhältnis der Anzahl der Elemente der Eingabemenge im Vergleich zur Anzahl der Elemente der resultierenden Multimenge des Replikationsprozesses. Die I/O-Kosten von GESS ergeben sich damit aus Gleichung 3.13 zu

$$GESS_{\text{I/O-Kosten}} \approx w \cdot 2(|R| + |S|). \tag{3.14}$$

D.h., daß selbst für einen Replikationsfaktor von $w = 2$ (durchschnittlich wird jedes Eingabeelement einmal aufgeteilt) die I/O-Kosten von GESS für alle Puffergrößen unter denen von MSJ liegen.

3.5 Implementierungsaspekte

In diesem Abschnitt diskutieren wir einige interessante Implementierungsaspekte unseres Algorithmus. Auf die softwaretechnische Realisation von GESS im Rahmen von XXL wird in Abschnitt 5.3 eingegangen.

- **Integration in XXL-Bibliothek:** GESS ist vollständig integriert in die XXL-Bibliothek. Insbesondere wurde hierbei auf **Generizität** geachtet. Wir werden hierauf in einem separatem Kapitel (Kapitel 5: XXL) eingehen.

- **Datentypunabhängig:** GESS kann beliebige Eingabedaten verarbeiten. Für den in dieser Arbeit behandelten Similarity Join wird zunächst für jeden Eingabepunkt ein Hypercube mit Seitenlänge ε berechnet. Für jeden Eingabepunkt wird ein Tupel (Punkt,Z-Wert) erzeugt. Eine Erweiterung auf k-Nearest-Neighbor Joins ist hiermit leicht möglich z.B. durch individuell berechnete ε für jeden Punkt (bzw. Teilräume).

- **Festkommaarithmetik:** Intern wird von der Fließkommaarithmetik der Datentypen *double* bzw. *float* auf eine Festkommaarithmetik übergegangen. Hierfür werden mittels Bitoperationen die Mantissen der skalaren Werte extrahiert und intern durch *long*-Typen dargestellt. Auf der so gewonnen Festkommarepräsentation lassen sich sehr effizient Z-Wert-Berechnungen bzw. Replikatberechnungen von Hypercubes durchführen. Es sind keine Exponentenabgleiche mehr notwendig. Die Genauigkeit dieser Arithmetik beträgt 63 Bit für den gesamten Datenraum wohingegen für die Fließkommadarstellung diese Genauigkeit nur auf dem Intervall $[0; 2^{-10})$ erreicht wird.

- **Prädikatgesteuerte Replikation:** Der generische Replikationsalgorithmus von GESS entscheidet mit Hilfe benutzerdefinierter Prädikate welche Replikations-Strategie angewandt wird. Zahlreiche vordefinierte Strategien werden in unserer Implementierung bereits zur Verfügung gestellt. Erweiterungen hiervon sind durch Implementierung eigener Prädikate leicht möglich. Ein Modifizieren des eigentlichen Algorithmus ist hierbei nicht notwendig. Intern verwaltet der Replikationsalgorithmus die Elemente in einem Heap. Dieser kann zusätzlich für prioritätsgesteuerte Replikation genutzt werden.

- **Referenzpunktmethode:** Die Referenzpunktmethode ist als binäres Prädikat P_{RPM} implementiert. D.h. wir ersetzen das eigentliche Joinprädikat $P_{join} = P_{overlap}$ durch

$$P_{join} = P_{overlap} \wedge P_{RPM}.$$

  Wir gehen hierbei davon aus, daß der Ausdruck von links nach rechts ausgewertet wird. D.h. nur für diejenigen Elemente, für die $P_{overlap}$ gilt, wird P_{RPM} ausgewertet. Ferner nutzen wir hier noch die Optimierung, daß die Referenzpunktmethode nur dann angewandt werden braucht, wenn mindestens eines der Elemente ein Replikat ist. Für jedes Tupel (Punkt,Z-Wert) merken wir uns, ob das Tupel ein Replikat ist, d.h. wir betrachten Tupel (Punkt,Z-Wert, bool). In der Joinphase überprüfen wir dann

für jedes Kandidatenelement, ob mindestens eines der Tupel ein Replikat ist: nur für diese Fälle kann die Referenzpunktmethode `false` zurückgeben. Sei $P_{ist_Replikat}(u)$ ein einstelliges Prädikat, daß genau dann gilt, wenn u Replikat ist. Dann erhalten wir insgesamt folgendes Joinprädikat:

$$P_{join}(u,v) \;=\; P_{overlap}(u,v) \wedge \Big(\big(\neg P_{ist_Replikat}(u) \wedge \neg P_{ist_Replikat}(v) \big) \vee P_{RPM}(u,v) \Big).$$

3.6 Experimente

3.6.1 Simulationen

Um unser analytisches Modell zu bestätigen, haben wir einen Simulator basierend auf dem Zufallsgenerator der *COLT-Bibliothek* [COL] implementiert. Unser Simulator erzeugt gleichverteilte Hypercubes in $[0;1)^d$ und berechnet für diese den Code sowie die zugehörige Ebene.

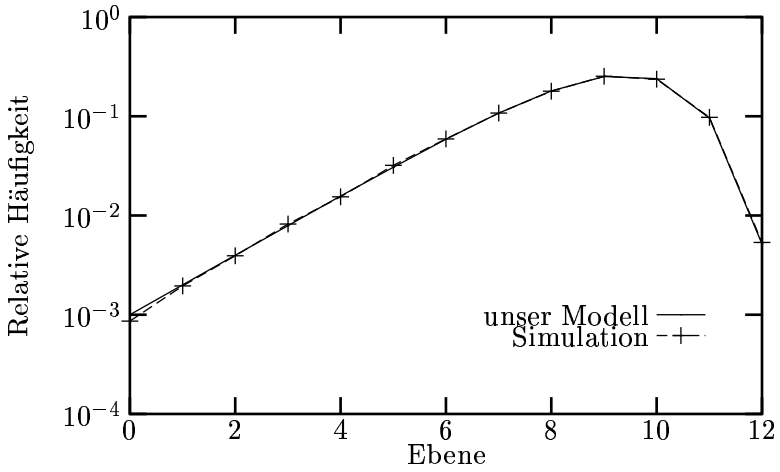
Die Ergebnisse in Abbildung 3.9a und 3.9b zeigen eine sehr gute Übereinstimmung der Simulation mit den Voraussagen des Modells ($d = 10$, $\varepsilon = 10^{-4}$). Die Kurven zeigen die Häufigkeit der Hypercubes für die verschiedenen Ebenen.

Die Simulationen hatten drei wesentliche Ergebnisse:

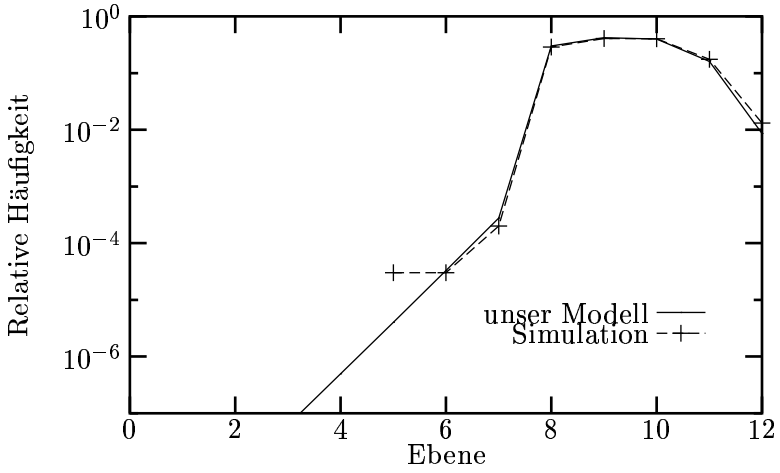
1. Das analytisches Modell sagt die in der Simulation gemessenen relativen Häufigkeiten sehr genau voraus. Abbildung 3.9a zeigt die erwarteten und die gemessenen Häufigkeiten für gleichverteilte Daten. Wie man sieht, sind die Kurven nahezu identisch.

2. Das analytische Modell ist leicht pessimistisch wenn Replikation betrachtet wird. In der Simulation fällt eine geringe Zahl von Hypercubes tiefer als durch das Modell vorhergesagt. Dies liegt daran, daß wir die pessimistische Annahme gemacht haben, daß die Replikate auf tieferen Ebenen gleichverteilt sind. In der Simulation treten Replikate aber gehäuft um die Hyperebenen auf. Dies hat den gewünschten Effekt, daß die von unserem Modell geschätzte Anzahl der Distanzberechnungen DC größer ist als in der Simulation gemessen.

3. Für die Simulation ist zu beachten, daß die gemessenen Häufigkeiten durch eine diskrete Anzahl von Ereignissen entstehen. Dies bedeutet z.B., falls $r_l \cdot ||R|| < 1$ in Ebene l gilt, wird ‚weniger als ein Hypercube' erwartet. Dieser Effekt ist in Abbildung 3.9b zu sehen: die Kurve zeigt die erwartete und die beobachtete Verteilung für $\varepsilon = 10^{-4}$ und $d = 10$ für unseren Algorithmus GESS. 100.000 Hypercubes wurden für die Simulation genutzt. Die erwartete Anzahl von Hypercubes für Ebene 2 ist



(a) Ohne Replikation



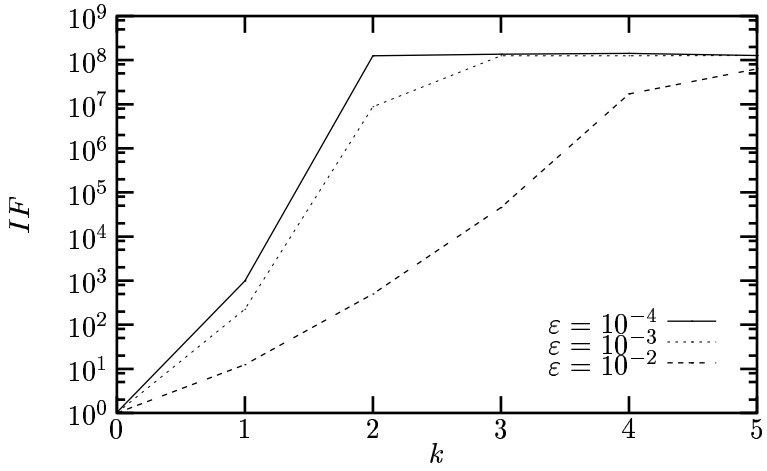
(b) Mit Replikation

Abbildung 3.9: Vergleich des analytischen Modells mit Ergebnissen der Simulation

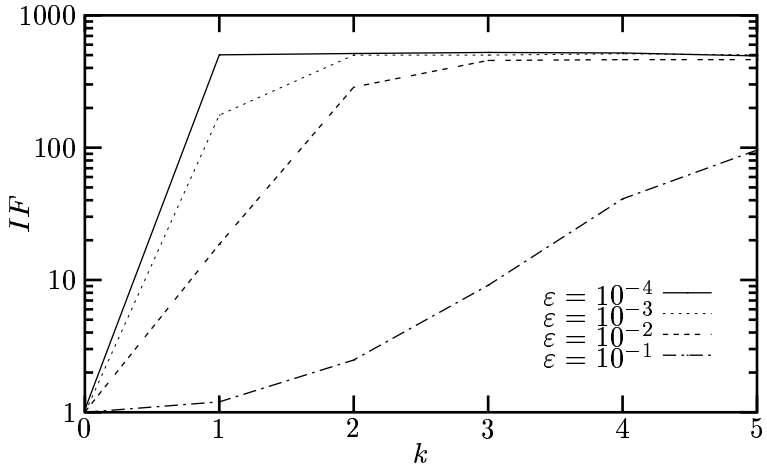
$r_2 \cdot ||R|| = 7.2 \cdot 10^{-6} \cdot ||R|| = 0.72 < 1$. Für das konkrete Experiment bedeute dies, daß Ebene 2 einen Hypercube (in 72% der Simulationen), Ebene 0 und 1 keine Hypercubes enthalten.

3.6.1.1 Distanzberechnungen

Sei $IF = DC_{\text{MSJ}}/DC_{\text{GESS}}$ der Verbesserungsfaktor von GESS gegenüber MSJ bezüglich der Distanzberechnungen. Abbildung 3.10 zeigt IF für einen Selbstverbund gleichverteilter Daten. Abbildung 3.10a zeigt IF für $d = 10$, $msl = 2$ in Abhängigkeit von k (logarithmische Skala, $k = 0$ entspricht der Partitionierungsstrategie von MSJ). Für $\varepsilon = 10^{-4}$ ($k = 1$) steigt



(a) Replikation erlaubt in den Ebenen $l \leq 2$ ($msl = 2$)



(b) Replikation nur erlaubt in Ebene $l = 0$ ($msl = 0$)

Abbildung 3.10: Distanzberechnungen: Verbesserungsfaktor IF für GESS gegenüber MSJ

IF auf den Wert 10^3 und $1.2 \cdot 10^8$ für $k = 2$. IF erreicht mit steigendem k Werte im Bereich von 8 Größenordnungen. Für große Werte von ε sinkt IF.

In unseren Simulationen haben wir nur geringe Replikationsraten gemessen für $\varepsilon = 10^{-3}$ und $\varepsilon = 10^{-4}$ (weniger als 1%). Für $\varepsilon = 10^{-2}$ war die Replikationsrate unter Faktor 2. Da die I/O-Verbesserung von GESS gegenüber MSJ größer ist als Faktor 2 (vgl. Abschnitt 3.4), bedeutet dies, daß GESS trotzdem für dieses Experiment insgesamt weniger I/O-Operationen benötigt als MSJ. Für größere Werte von ε haben wir höhere Replikationsraten festgestellt.

Um die Replikationsraten gering zu halten, haben wir eine andere Aufteilungsstrategie in Abbildung 3.10b verwendet. Hier ist $msl = 0$, d.h. das Aufteilen von Hypercubes ist nur in

Ebene 0 erlaubt. Für $k \geq 3$ zeigt das Diagramm einen IF von ca. 500 für $\varepsilon = 10^{-4}, 10^{-3}$ und 10^{-2} welcher konstant bleibt für höhere Werte von k. Dies erklärt sich durch die Tatsache, daß für $k = 1$ und $\varepsilon = 10^{-4}, 10^{-3}$ die Ebene 0 fast keine Hypercubes mehr enthält — nur wenige Hypercubes treffen mehr als eine Hyperebene. Für größere Werte von ε hingegen steigt die Wahrscheinlichkeit, mehrere Hyperebenen zu treffen. Für diese Fälle kann k — z.B. in Abhängigkeit von ε — erhöht werden, um die Anzahl der Elemente in Ebene 0 gering zu halten. In unseren Experimenten waren die Replikations-Raten kleiner als 11% für alle gezeigten Parameter. Nur für $\varepsilon = 10^{-1}$ stieg die Replikation auf 2.85. Die Verbesserung IF hingegen war trotz dieser Tatsache 95.

3.6.1.2 Hauptspeicherbedarf

In diesem Abschnitt untersuchen wir den Hauptspeicherbedarf von GESS und MSJ. Die relative Größe des benötigten Hauptspeichers für GESS und MSJ kann geschätzt werden durch

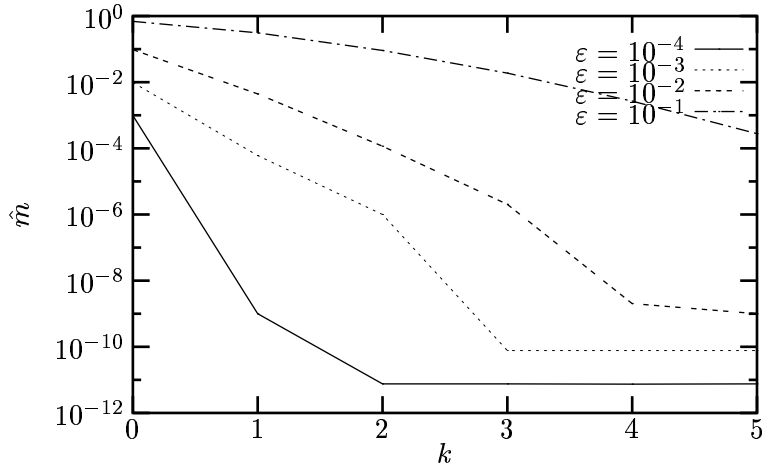
$$\hat{m} = \frac{1}{||R|| + ||S||} \sum_{i=0}^{mL} \frac{||R_i|| + ||S_i||}{2^{d \cdot i}}$$

wobei für einen Selbstverbund gilt

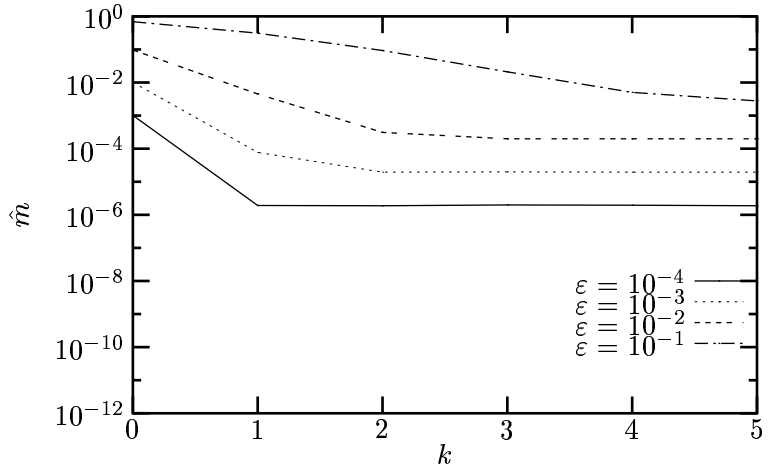
$$\hat{m} = \frac{1}{||R||} \sum_{i=0}^{mL} \frac{||R_i||}{2^{d \cdot i}}.$$

Abbildung 3.11 zeigt den Hauptspeicherbedarf $\hat{m}$ von GESS für die Experimente des letzten Abschnitts[2]. Abbildung 3.11a zeigt, daß mit steigendem k der Hauptspeicherbedarf von GESS exponentiell abnimmt. Der Hauptspeicherbedarf in der Joinphase ist um einen Faktor 10^{-4} geringer im Vergleich zu MSJ für $k = 1$ und $\varepsilon = 10^{-3}$. Für kleinere Werte von ε steigt dieser Faktor sogar noch. Für große Werte von ε ist der Faktor geringer aber ist immer noch exponentiell abhängig von k.

Abbildung 3.11b zeigt die Ergebnisse eines Experimentes für $msl = 0$. Der Hauptspeicherbedarf in dem Experiment sinkt zunächst exponentiell und bleibt dann auf einem Niveau. Letzterer Effekt erklärt sich durch unsere Beobachtung, daß für hohe Werte von k die Ebene 0 keine Hypercubes mehr enthält.



(a) Replikation erlaubt in den Ebenen ≤ 2 ($msl = 2$)



(b) Replikation nur erlaubt in Ebene 0 ($msl = 0$)

Abbildung 3.11: Hauptspeicherbedarf von GESS in Abhängigkeit von k ($k = 0$: MSJ)

3.6.2 Implementierungen

In diesem Abschnitt präsentieren wir Ergebnisse von Experimenten mit realen Datensätzen. Die Experimente wurden durchgeführt auf einem AthlonTB 700 mit 256 MB Hauptspeicher und 30 GB Festplatte. Alle Algorithmen sind in Java 1.3 basierend auf XXL [BDS00, BBD+01] implementiert[3].

Um die Gesamtlaufzeit der Algorithmen zu ermitteln, wurde der Puffer des Betriebssystems ausgeschaltet durch das Nutzen von raw-Devices. Die Runs, die während des Sortierens

[2] $\hat{m} = 1$ heißt somit, daß die gesamte Eingaberelation im Hauptspeicher gehalten werden muß.
[3] siehe Kapitel 5

Daten-satz	Beschreibung	# Punkte	d	Größe
CAD	Merkmalsvektoren von CAD-Bauteilen	1.312.173	16	80,1 MB
CH	Farbhistogramme von Fernsehbildern	112.361	64	27,4 MB
CM	Corel-Moments	68.040	9	2,4 MB
CT	Corel-Texture	68.040	16	4,2 MB

Tabelle 3.1: Beschreibung der in den Experimenten genutzten Datensätze

erzeugt werden, werden in große, kontinuierliche Partitionen auf das raw-Device geschrieben.

Der verfügbare Hauptspeicher war 10% der Größe der Eingabedaten. Für die Experimente haben wir verschiedene hochdimensionale Datensätze genutzt, die bereits in anderen Experimenten Verwendung gefunden haben [BK01, LLL01]. Tabelle 3.1 zeigt die genutzten Datensätze. Die Dimensionalität der Daten liegt im Bereich von 9–64. Der größte verfügbare Datensatz (CAD) enthält 1,3 Millionen Punkte und ist 80 MB groß.

3.6.3 Die Auswirkung der Aufteilungsstrategie

Um die hohen Auswirkungen kontrollierter Replikation zu zeigen, variieren wir in den folgenden Unterabschnitten die Replikationsstrategie unseres Algorithmus GESS. Die in Abschnitt 3.2 eingeführte Kompositionsstrategie 3 besagt, daß ein Hypercube nur dann aufgeteilt werden darf, wenn die aktuelle Ebene kleiner als eine gegebene Schranke msl ist und wenn höchstens k Hyperebenen in der aktuellen Ebene getroffen werden.

Wir variieren im folgenden den Parameter msl im Bereich von 0 bis 5 und den Parameter k im Bereich von 0 bis 5. Der Wert für $k = 0$ entspricht in den Graphiken jeweils der Partitionierungsstrategie von MSJ.

Wir berichten die Laufzeit, die Anzahl der Distanzberechnungen, die Replikationsraten sowie den Hauptspeicherbedarf in Abhängigkeit der Aufteilungsstrategie.

3.6.3.1 Laufzeit

Die Abbildung 3.12 zeigt das Ergebnis von Ähnlichkeitsselbstverbünden des Datensatzes CAD. Die x-Achse zeigt jeweils den Parameter k, der die Aufteilungsstrategie bezeichnet. Die maximale Aufteilungsebene msl war auf den Wert 0 gesetzt in dem Experiment. Auf

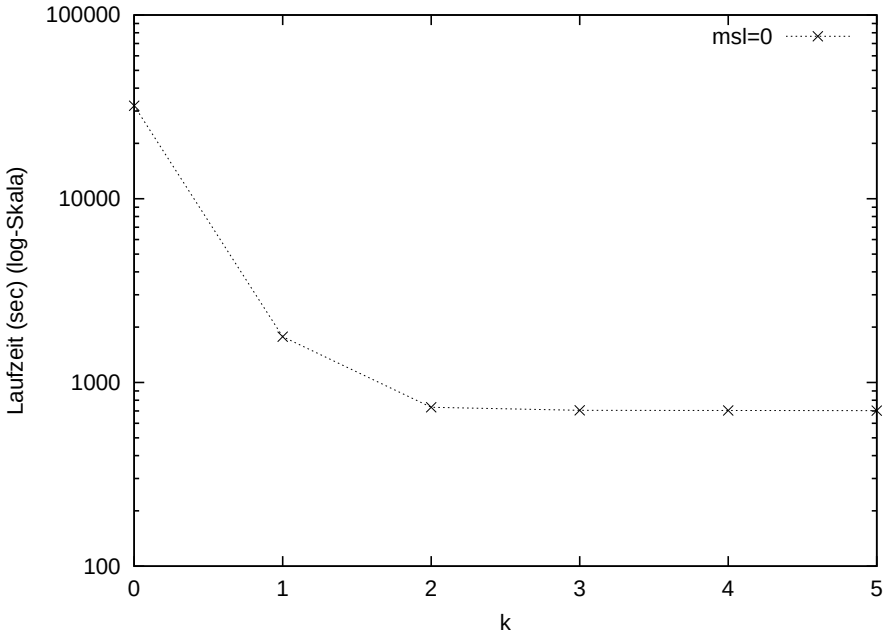


Abbildung 3.12: Laufzeit für Ähnlichkeitsselbstverbund von CAD für verschiedene k ($k = 0$: MSJ), $\varepsilon = 10^{-3}$ (log-Skala auf y-Achse)

der y-Achse ist die Gesamtlaufzeit in Sekunden abgetragen (logarithmische Skala). Der Datenpunkt bei $k = 0$ entspricht dem Ergebnis ohne Replikation. Dies entspricht der Vorgehensweise von MSJ. Hierbei ist zu beachten, daß für MSJ noch zusätzliche Kosten durch das ungünstigere I/O-Verhalten hinzukommen. Der Parameter ε betrug 10^{-3}. Für dieses ε wurde bereits eine beträchtliche Anzahl von Ergebnistupeln produziert. Wir werden auf die Anzahl der produzierten Ergebnistupel in Abschnitt 3.7 noch genauer eingehen.

Die Ergebnisse zeigen eine Laufzeit von über 31610 Sekunden für $k = 0$. Für $k = 1$ sinkt die Laufzeit bereits um Faktor 15 auf ca. 1700 Sekunden. Für $k = 2$ benötigt das Verfahren nur 720 Sekunden. für $k \geq 3$ beträgt die Laufzeit 695 Sekunden und ändert sich nicht mehr für größere k. Dies liegt darin begründet, daß für $k = 3$ die Ebene 0 der Partitionierungshierarchie keine Datenelemente mehr enthält. Somit hat das Zulassen von mehr Replikation keine Auswirkung mehr. Insgesamt ist GESS für dieses Experiment für $k \geq 3$ um Faktor 45 schneller als das ursprüngliche Verfahren. Die nächste Abbildung (3.13) zeigt dasselbe Experiment mit erweiterterten Aufteilungsstrategien ($msl > 0$). Für $msl = 1$ und $msl = 2$ zeigen die Kurven eine weitere Leistungssteigerung des Verfahrens (660 bzw. 635 sec). Für $msl \geq 3$ steigt die Gesamtlaufzeit allerdings wieder an. Die beste Aufteilungsstrategie ($msl = 2,\ k = 5$) erreicht gegenüber MSJ einen Laufzeitgewinn von Faktor 50.

Die Abbildungen 3.14, 3.15 und 3.16 zeigen das gleiche Experiment für die Datensätze CT,

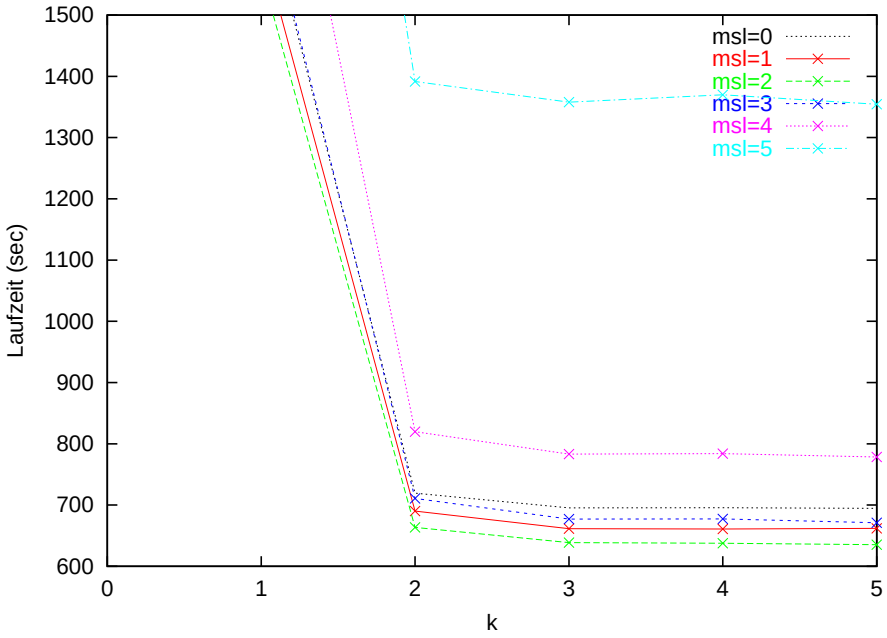


Abbildung 3.13: Laufzeit für Ähnlichkeitsselbstverbund von CAD für verschiedene k und msl, $\varepsilon = 10^{-3}$

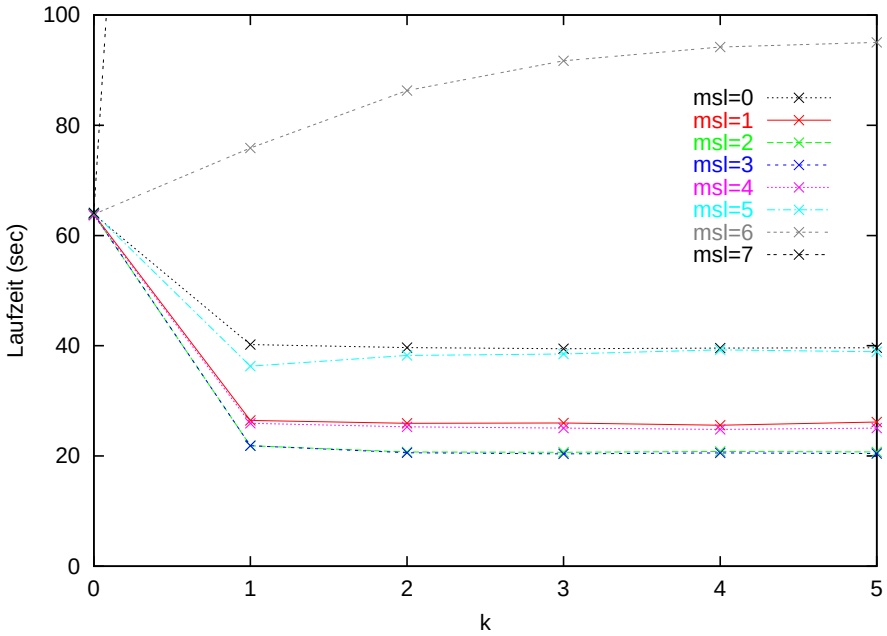


Abbildung 3.14: Laufzeit für Ähnlichkeitsselbstverbund von CT für verschiedene k und msl, $\varepsilon = 10^{-3}$

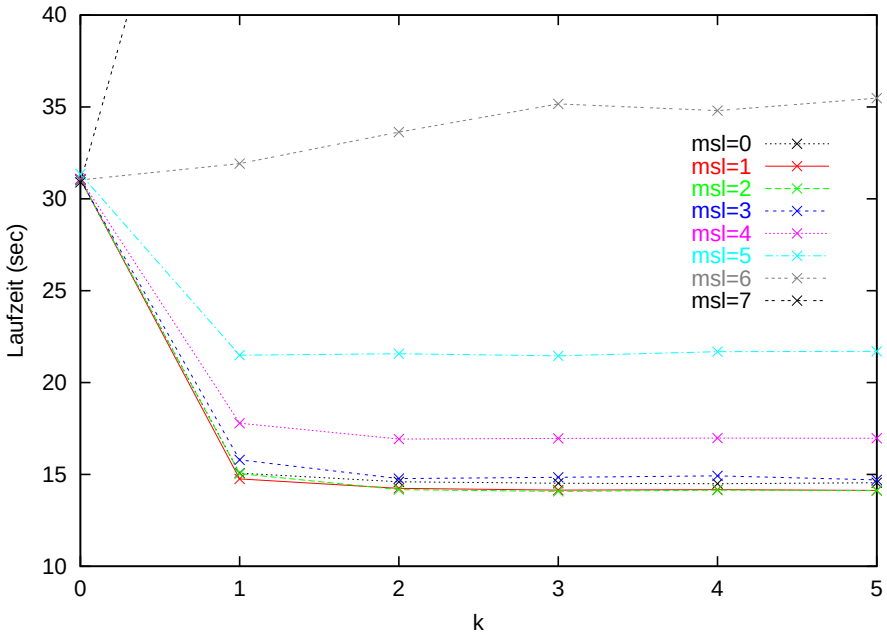


Abbildung 3.15: Laufzeit für Ähnlichkeitsselbstverbund von CM für verschiedene k und msl, $\varepsilon = 10^{-3}$

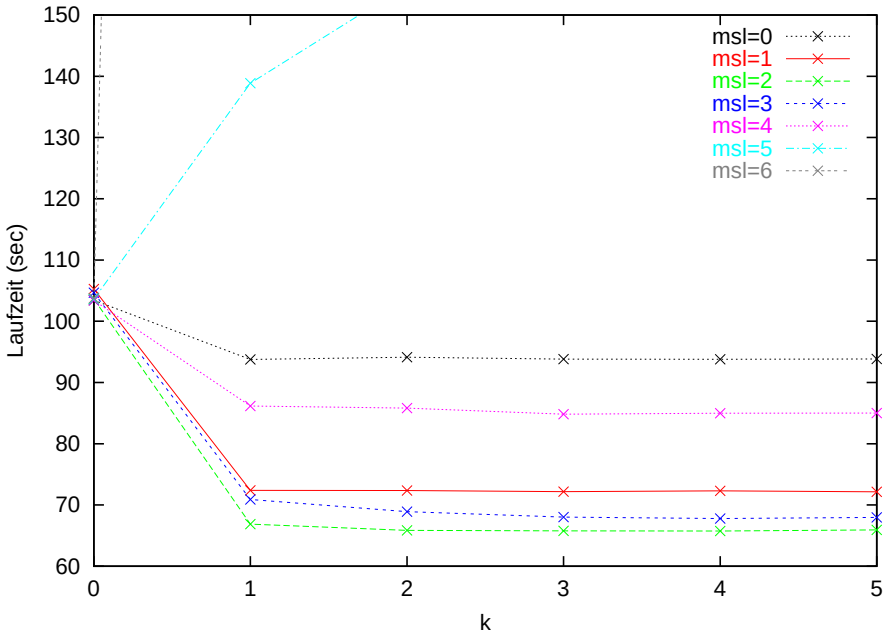


Abbildung 3.16: Laufzeit für Ähnlichkeitsselbstverbund von CH für verschiedene k und msl, $\varepsilon = 10^{-3}$

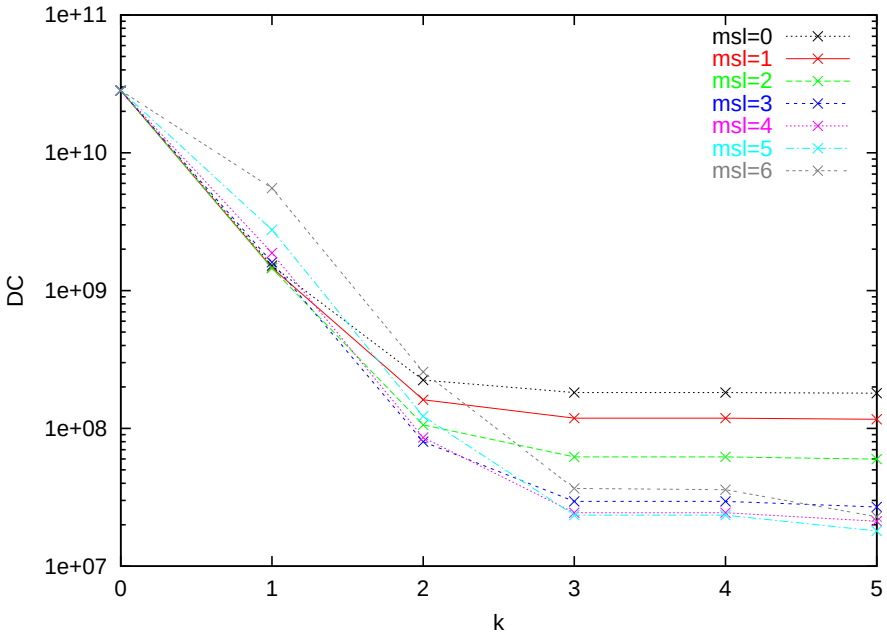


Abbildung 3.17: Distanzberechnungen DC für Ähnlichkeitsselbstverbund für CAD für verschiedene k ($k = 0$: MSJ), $\varepsilon = 10^{-3}$ (log-Skala auf y-Achse)

CM und CH. Für die Aufteilungsstrategie mit $msl = 0$ sind die Leistungsgewinne nicht sehr hoch im Vergleich zu den CAD-Daten. Die Kurven zeigen aber ein ähnliches Verhalten: für $k = 0$ ist die Laufzeit am größten, für $k = 1$ sinkt die Laufzeit um 30–40%. Für $k \geq 2$ bleibt die Laufzeit auf einem konstanten niedrigen Niveau.

Läßt man allerdings mehr Replikation zu ($msl > 0$), sinkt die Gesamtlaufzeit in allen Experimenten um den Faktor 2 bis 3. Insbesondere für $msl = 2$ bzw. $msl = 3$ scheint die Laufzeit am geringsten zu sein. Für höhere Werte von msl steigt die Laufzeit wieder an. Für Werte von $msl > 5$ erhöht sich die Laufzeit um mehrere Größenordnungen.

In den folgenden Abschnitten versuchen wir diese Ergebnisse näher zu erklären und betrachten hierzu die Auswirkung der Aufteilungsstrategie auf die Anzahl der Distanzberechnungen bzw. auf die Replikationsraten.

3.6.3.2 Distanzberechnungen

Abbildung 3.17 zeigt die Anzahl der Distanzberechnungen DC in Abhängigkeit der Aufteilungs-Strategie (k und msl) für Ähnlichkeitsselbstverbünde des CAD-Datensatzes. Für $k = 0$ werden $2,8 \cdot 10^{10}$ Distanzberechnungen durchgeführt. Für $k = 1$ sinkt dieser Wert um eine Größenordnung auf $1,5 \cdot 10^{9}$. Bei $k = 2$ beträgt DC $2,2 \cdot 10^{8}$, für $k > 2$ $1,8 \cdot 10^{8}$. D.h.

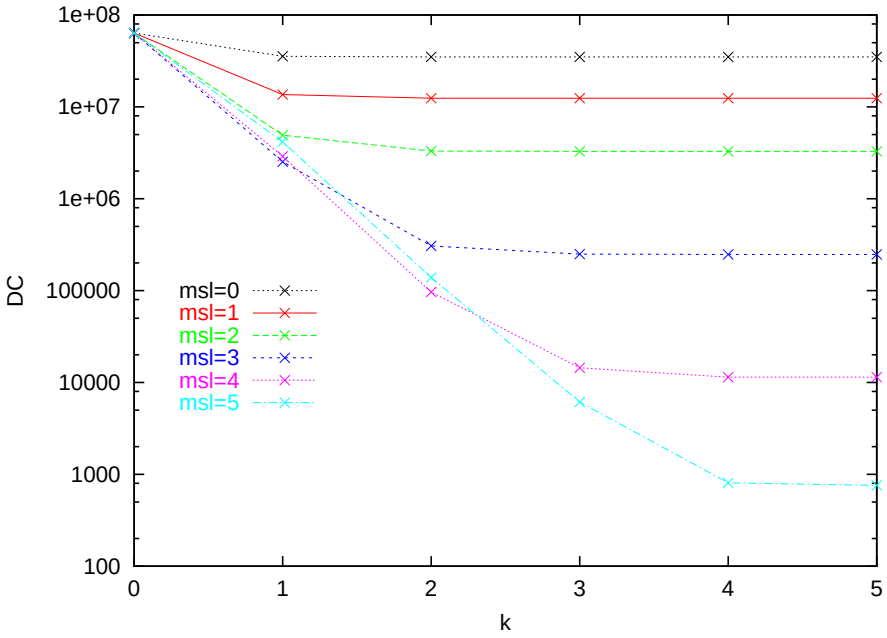


Abbildung 3.18: Distanzberechnungen DC für Ähnlichkeitsselbstverbund von CT für verschiedene k und msl ($k = 0$: MSJ), $\varepsilon = 10^{-3}$ (log-Skala auf y-Achse)

insgesamt sinkt die Anzahl der Distanzberechnungen um mehr als 2 Größenordnungen für dieses Experiment.

Für die Aufteilungsstrategien mit $msl > 0$ sinkt die Anzahl der Distanzberechnungen weiter und erreicht bei $k = 5$, $msl = 5$ ein Minimum von $1,8 \cdot 10^7$. Eine hohes Maß an Replikation ist somit bezüglich der Anzahl der Distanzberechnungen in jedem Fall vorteilhaft.

Die Abbildungen 3.18, 3.19 und 3.20 zeigen das gleiche Experiment für die Datensätze CT, CM und CH. Die Einsparung in der Anzahl der Distanzberechnungen betragen hier jeweils 44%, 97% und 18% für die einfache Aufteilungsstrategie mit $msl = 0$. Auch hier läßt sich der Effekt beobachten, daß für $k \geq 2$ die Kurve konstant bleibt.

Bei Zulassen von mehr Replikation ($msl > 0$) sinkt die Anzahl der Distanzberechnungen für alle 3 Datensätze allerdings ebenfalls exponentiell. Ein Erhöhen von msl führt jeweils zu einem Absinken des Wertes DC. Ein interessanter Effekt ist in Abbildung 3.19 für die Kurven $msl = 4$ sowie $msl = 5$ zu beobachten: für $msl = 5$ steigt die Anzahl der Distanzberechnungen teilweise gegenüber $msl = 4$ leicht an. Dies könnte an den für diese Aufteilungsstrategien hohen Replikationsraten liegen. Dies wollen wir im folgenden Abschnitt näher untersuchen.

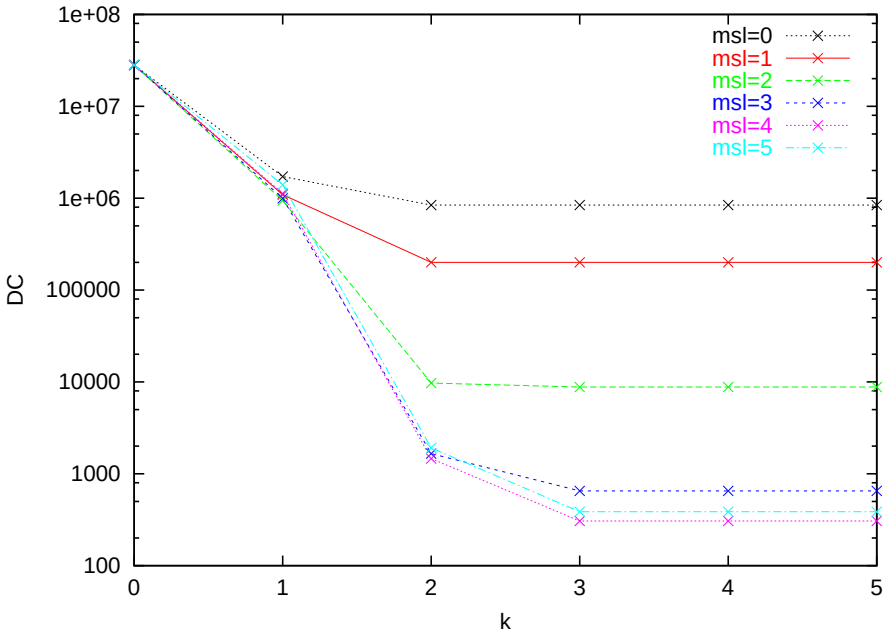


Abbildung 3.19: Distanzberechnungen DC für Ähnlichkeitsselbstverbund von CM für verschiedene k und msl ($k = 0$: MSJ), $\varepsilon = 10^{-3}$ (log-Skala auf y-Achse)

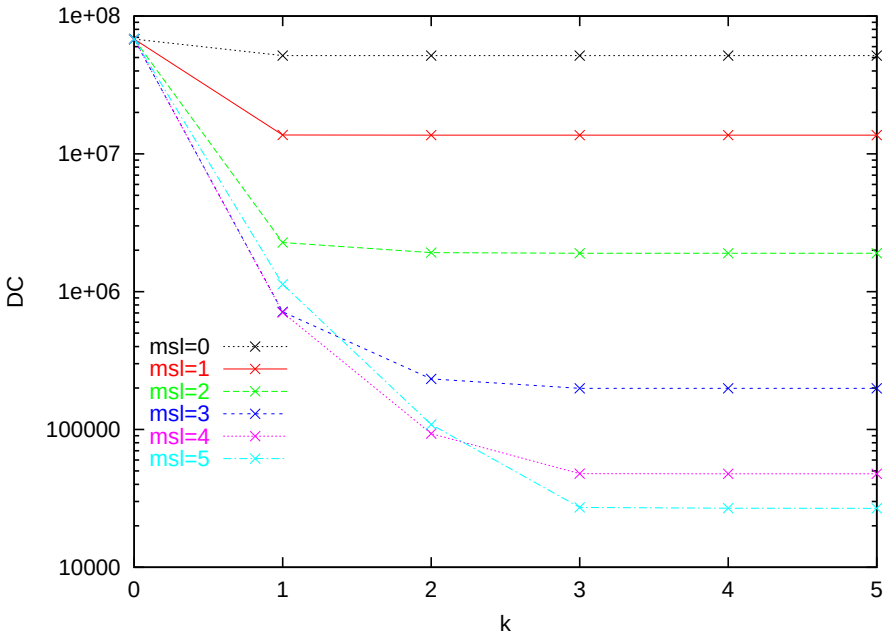


Abbildung 3.20: Distanzberechnungen DC für Ähnlichkeitsselbstverbund von CH für verschiedene k und msl ($k = 0$: MSJ), $\varepsilon = 10^{-3}$ (log-Skala auf y-Achse)

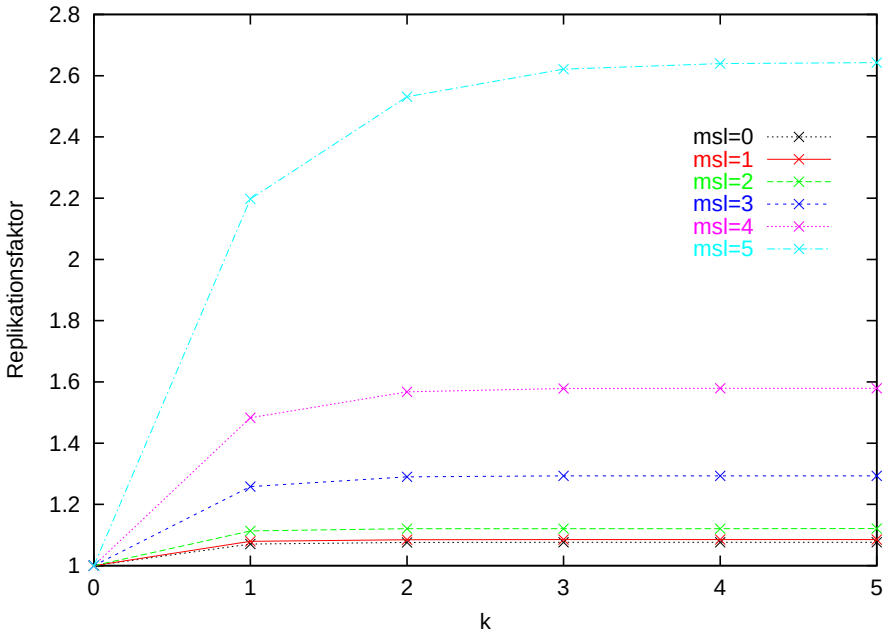


Abbildung 3.21: Replikationsraten für Ähnlichkeitsselbstverbund für CAD für verschiedene k ($k = 0$: MSJ), $\varepsilon = 10^{-3}$

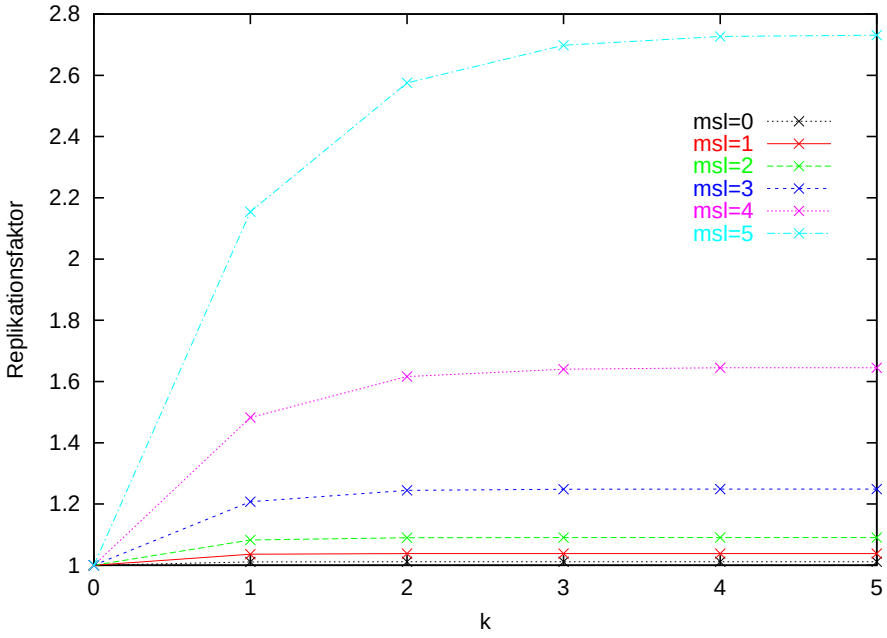


Abbildung 3.22: Replikationsraten für Ähnlichkeitsselbstverbund von CT für verschiedene k und msl ($k = 0$: MSJ), $\varepsilon = 10^{-3}$

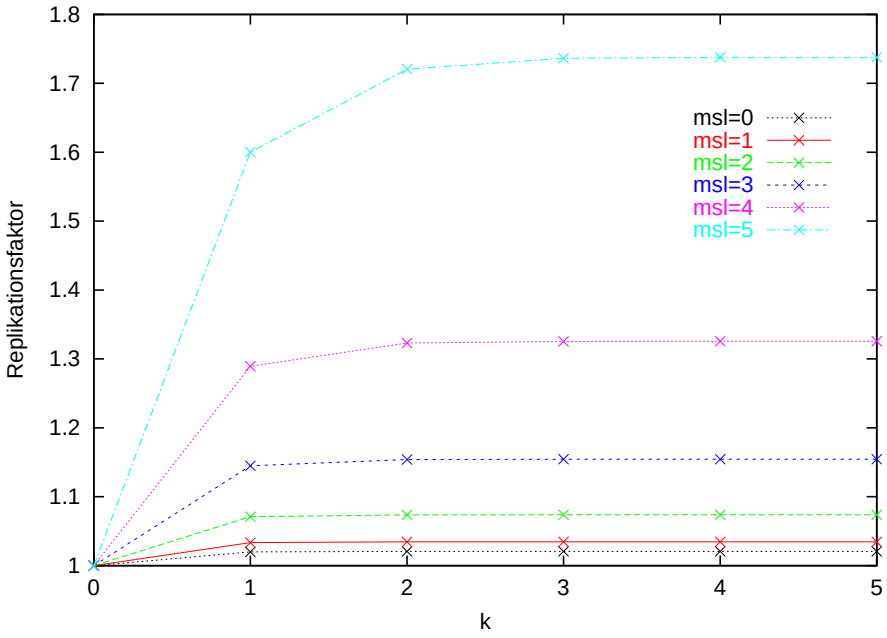


Abbildung 3.23: Replikationsraten für Ähnlichkeitsselbstverbund von CM für verschiedene k und msl ($k = 0$: MSJ), $\varepsilon = 10^{-3}$

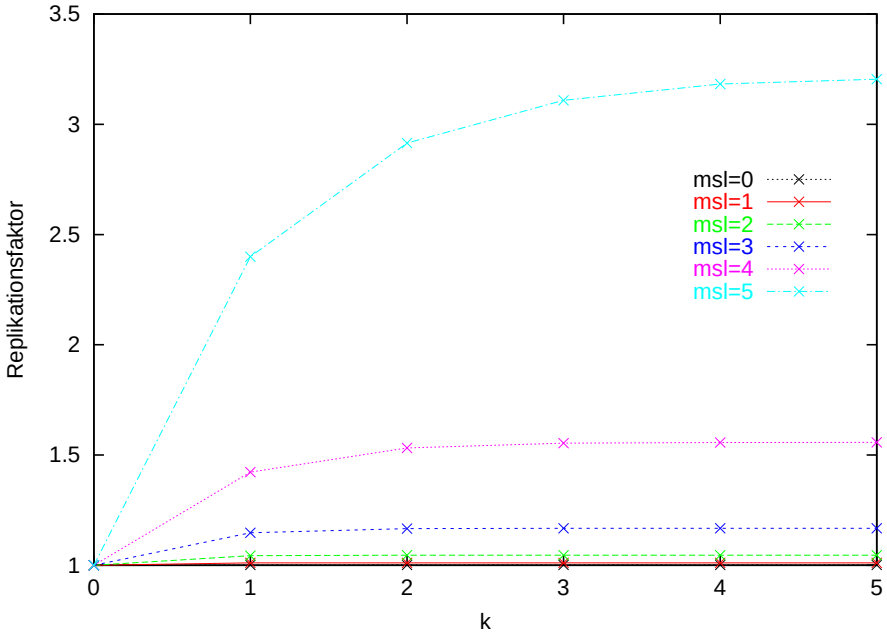


Abbildung 3.24: Replikationsraten für Ähnlichkeitsselbstverbund von CH für verschiedene k und msl ($k = 0$: MSJ), $\varepsilon = 10^{-3}$

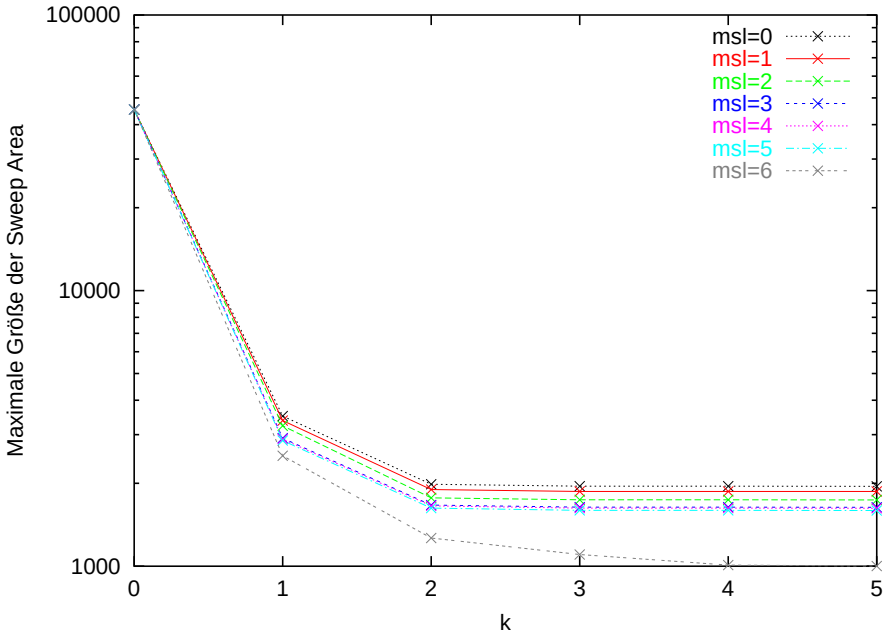


Abbildung 3.25: Maximale Größe der Sweep Area für Ähnlichkeitsselbstverbund von CAD für verschiedene k und msl ($k = 0$: MSJ), $\varepsilon = 10^{-3}$ (log-Skala auf y-Achse)

3.6.4 Replikationsraten

Die Abbildungen 3.21 bis 3.24 zeigen die beobachteten Replikationsraten für obige Experimente. Für $msl \leq 2$ liegt die beobachtete Replikation für alle 4 Datensätze unter 10%, für $msl \leq 3$ unter 25%. Für $msl = 4$ steigt die Replikation auf bis zu 60%, für $msl = 5$ auf Werte von bis über 300%. Für solch hohen Replikationsraten steigen die Kosten für das externe Sortieren der Daten stark an. Dies erklärt vermutlich, daß trotz einer sehr geringen Anzahl von Distanzberechnungen für Werte von $msl > 3$ die Laufzeit des Verfahrens trotzdem schlechter wird.

3.6.4.1 Größe der Sweep Area und Hauptspeicherbedarf

Der Hauptspeicherbedarf von MSJ und GESS wird domiert von der Anzahl der Elemente, die maximal in den Sweep Areas gehalten werden müssen. Die Abbildungen 3.25 bis 3.28 zeigen die maximale Anzahl der Elemente, die bei obigen Experimenten in der Sweep Area gespeichert werden müssen.

Für das Experiment mit den CAD-Daten (Abbildung 3.25) muß ohne Replikation Hauptspeicher für 45356 Elemente zur Verfügung stehen, um ein Auslagern der Sweep Area auf

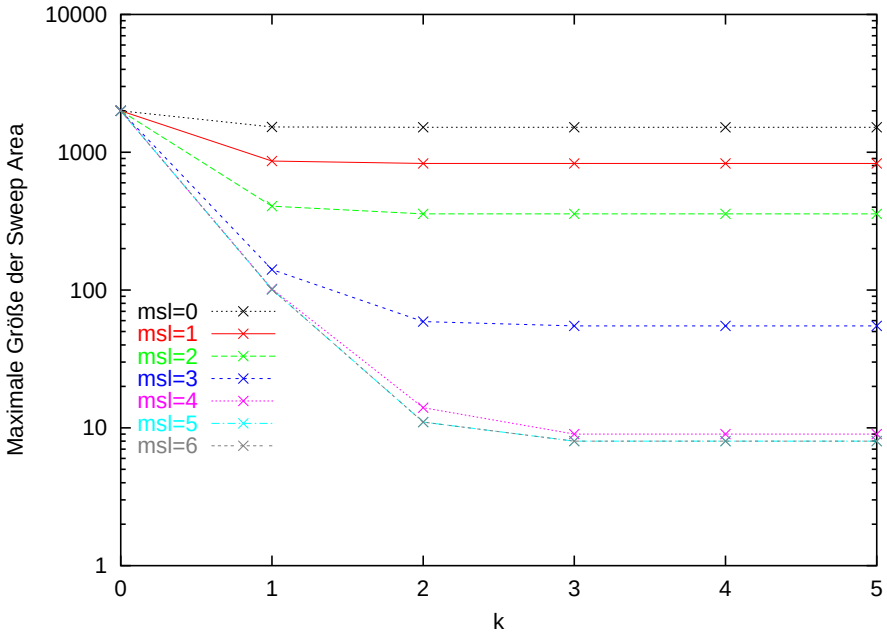


Abbildung 3.26: Maximale Größe der Sweep Area für Ähnlichkeitsselbstverbund von CT für verschiedene k und msl ($k = 0$: MSJ), $\varepsilon = 10^{-3}$ (log-Skala auf y-Achse)

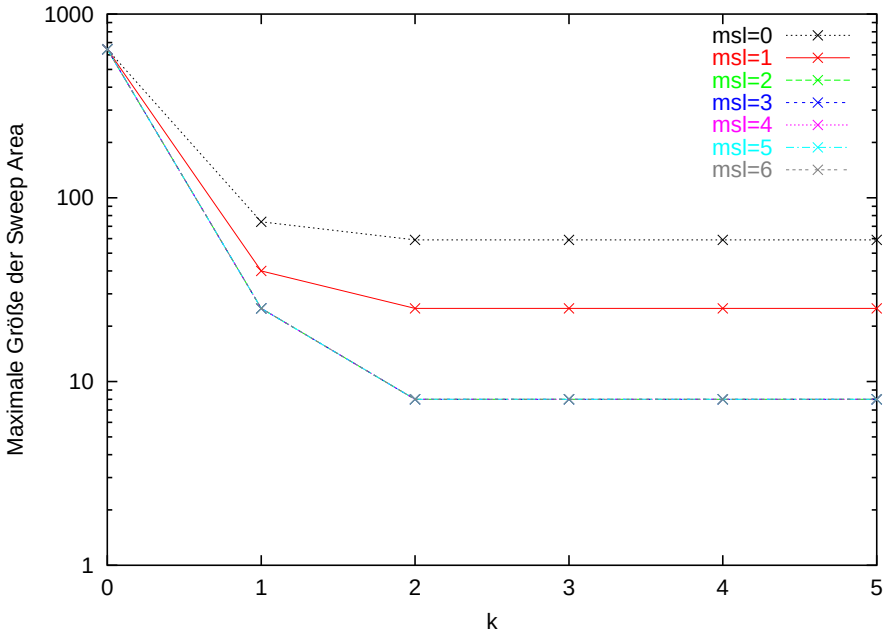


Abbildung 3.27: Maximale Größe der Sweep Area für Ähnlichkeitsselbstverbund von CM für verschiedene k und msl ($k = 0$: MSJ), $\varepsilon = 10^{-3}$ (log-Skala auf y-Achse)

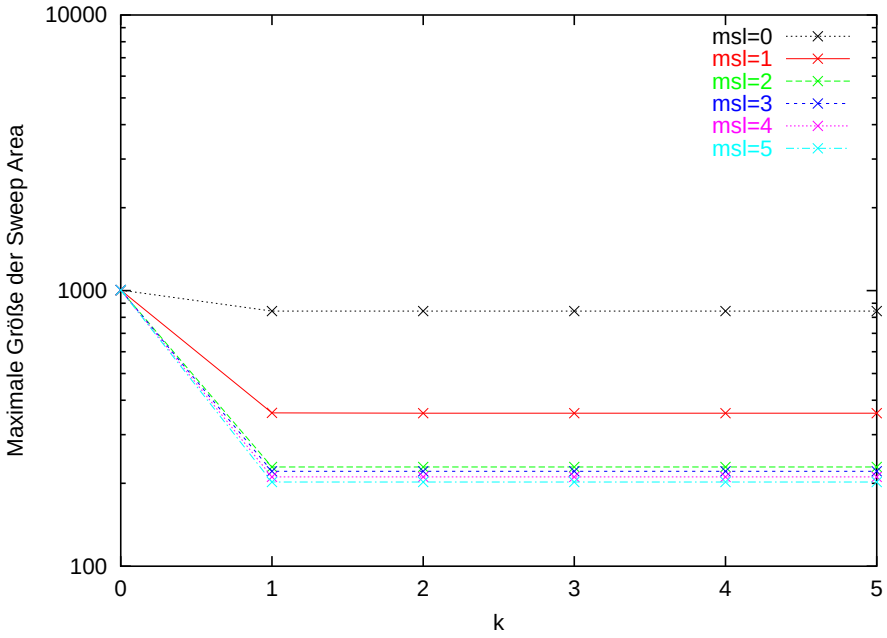


Abbildung 3.28: Maximale Größe der Sweep Area für Ähnlichkeitsselbstverbund von CH für verschiedene k und msl ($k = 0$: MSJ), $\varepsilon = 10^{-3}$ (log-Skala auf y-Achse)

den Externspeicher zu vermeiden. Für $k = 1$ sinkt dieser Wert bereits auf 3503, für $k = 2$ auf 1984 Elemente. D.h. GESS braucht nur 1/22 des Hauptspeichers für die Sweep Area im Vergleich zu MSJ. Abbildungen 3.26, 3.27 und 3.28 zeigen das gleiche Experiment für die Datensätze CT, CM und CH. Die maximale Anzahl der Elemente sinkt hier jeweils um 24%, 91% und 3% für die einfache Strategie mit $msl = 0$.

Für Aufteilungsstratgien mit $msl > 0$ ist ein stärkeres Abfallen des Hauptspeicherbedarfs zu beobachten. Der Hauptspeicherbedarf sinkt in diesen Fällen um den Faktor 45, 250, 80 und 5 für die Datensätze CAD, CT, CM bzw. CH.

3.6.5 Overhead der Referenzpunktmethode

Abbildung 3.29 zeigt den zusätzlichen Aufwand der Referenzpunktmethode für einen Selbstverbund auf den CAD-Daten. Auf der x-Achse ist der Parameter ε abgetragen. Ein höherer Wert für ε bedeutet eine höhere Anzahl von Ergebnistupeln. Die y-Achse zeigt die Gesamtlaufzeit des Verfahrens für GESS mit bzw. ohne Ausführung der Referenzpunktmethode RPM. Die Ergebnismengen für GESS ohne RPM beinhalten zum Teil erhebliche Mengen von Duplikaten. Für $\varepsilon = 0.0001$ ist der zusätzliche Aufwand der Referenzpunktmethode vernachlässigbar. Die Anzahl der Ergebnistupel beträgt hier $2.4 \cdot 10^5$. Die Anzahl der Duplikate

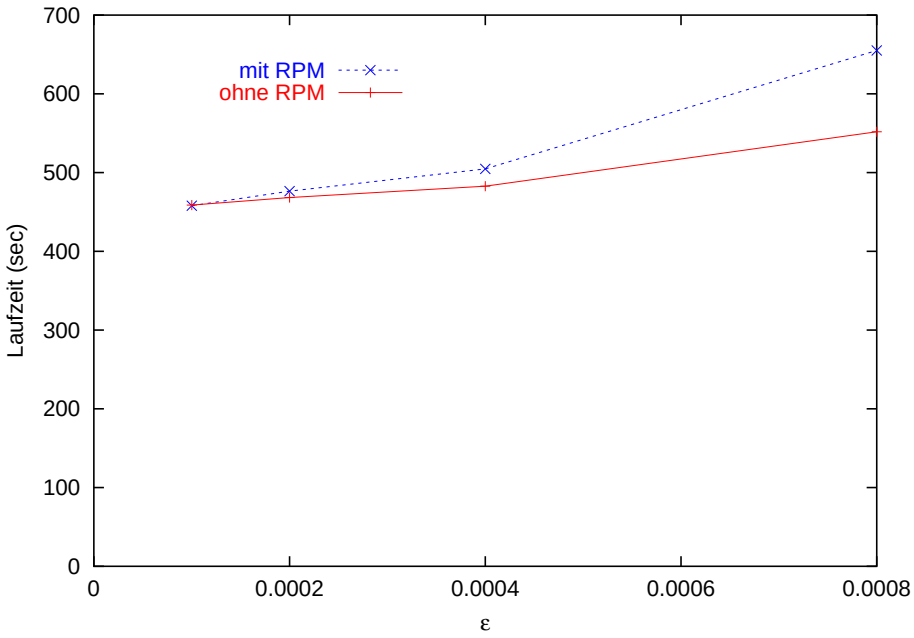


Abbildung 3.29: Ausführungszeit von GESS mit und ohne RPM für Ähnlichkeitsselbstverbund von CAD in Abhängigkeit von ε, $k = 3$

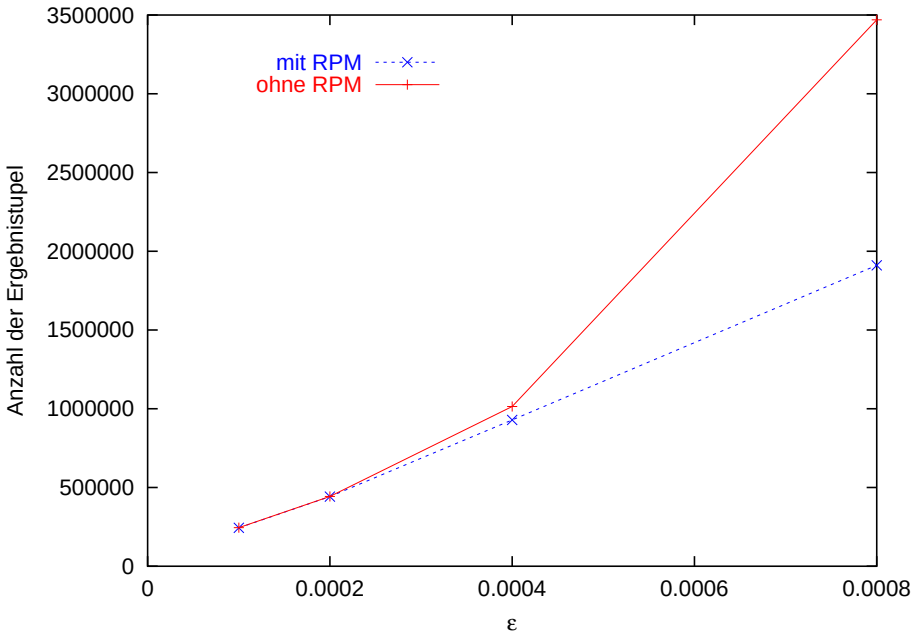


Abbildung 3.30: Anzahl der Ergebnistupel von GESS mit und ohne RPM für Ähnlichkeitsselbstverbund von CAD in Abhängigkeit von ε, $k = 3$

ist nur geringfügig höher. Mit steigendem ε steigt die Anzahl der Duplikate und damit auch die Anzahl der Aufrufe der Referenzpunktmethode. Dies ist dargestellt in Abbildung 3.30. Hier sind auf der y-Achse die Anzahl der Ergebnistupel mit und ohne Duplikateliminierung durch RPM aufgetragen. Für $\varepsilon = 0.0008$ beträgt die Anzahl der zu eliminierenden Duplikate bereits $1,5 \cdot 10^6$. Dies macht sich in Abbildung 3.29 durch die ansteigende zusätzliche Laufzeit bemerkbar.

Die Experimente zeigen, daß für eine hohe Anzahl von Duplikaten die Kosten für RPM ansteigen. Dies könnte zum einen durch eine „low-level"-Implementierung der Referenzpunktmethode verbessert werden, die in XXL sehr generisch implementiert ist. Hierbei stellt sich allerdings die Frage, ob es für realistische Applikationen noch Sinn macht, größere Ergebnismengen zu betrachten. Macht man die Annahme, daß jedes Ergebnistupel je zwei Referenzen à 8 Bytes auf die beteiligten Punkte hält, bekommt man für $\varepsilon = 0.0008$ bereits eine Ergebnisgröße von 29 MB nur für die Tupel (!) — ohne die dahinterliegenden Punkte mitzuzählen. Der Optimierer einer Datenbank- bzw. eines Data Mining-Systems würde zunächst versuchen, ein solch großes Zwischenergebnis zu vermeiden, um in nachfolgenden Operatoren hohe Kosten durch eine hohe Anzahl von Ergebnistupeln zu vermeiden.

Es stellt sich somit generell die Frage, wie der Parameter ε sinnvoll gewählt werden kann. Diese Problem betrachten wir im folgenden Abschnitt.

3.7 Experimente mit der Fraktalen Dimension

Unter der Annahme der *Gleichverteilung* kann mit Hilfe analytischer Modelle zu einer gesuchten Anzahl von Ergebnistupeln der Wert für ε geschätzt werden. Nutzt man diesen Wert ε für die Berechnung eines Ähnlichkeitsverbunds eines realen Datensatzes, zeigt sich, daß der vom Modell vorausgesagte Wert ε um mehrere Größenordnungen zu groß ist. Die Ursache hierfür liegt darin begründet, daß die Merkmalsvektoren realer Datensätze nicht unabhängig verteilt sind bezüglich der einzelnen Dimensionen.

3.7.1 Die fraktale Dimension

In [BF95, FSTT00] wurde festgestellt, daß die Anzahl der Ergebnistupel in Abhängigkeit von ε einem sogenannten *Power Law* genügt, d.h. es gilt

$$||R \bowtie S|| = c \cdot \varepsilon^{\mathcal{P}} \tag{3.15}$$

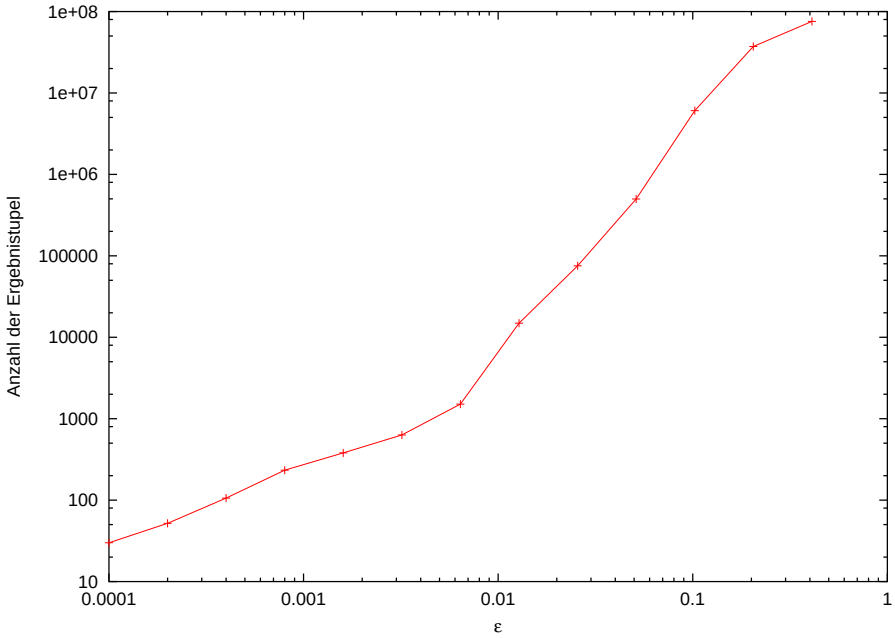


Abbildung 3.31: Anzahl der Ergebnistupel für Ähnlichkeitsselbstverbund von CAD (Stichprobe 1%) in Abhängigkeit von ε (log-log-Skala)

wobei c eine Konstante und $\mathcal{P}$ der sogenannte *Ergebnistupelexponent (pair-count exponent)* ist. Für einen Selbstverbund ($R = S$) entspricht $\mathcal{P}$ der *Fraktalen Dimension* (oder: *Hausdorff Dimension*) des Datensatzes und ist ein Maß für die **wahre** Dimensionalität der Daten [BF95]. Die Fraktale Dimension ist immer kleiner gleich der ‚einbettenden' Dimension d. Der Exponent $\mathcal{P}$ ist invariant bezüglich der verwendeten L_p-Metrik [FSTT00]. Es gilt sogar, daß $\mathcal{P}$ invariant ist bezüglich Sampling [FSTT00]. Hiermit kann in der Praxis ein effizientes Schätzverfahren für den Wert ε gewonnen werden — sofern ein solches Sample zur Verfügung steht.

Abbildung 3.31 zeigt die Anzahl der Ergebnistupel für einen Ähnlichkeitsselbstverbund einer Stichprobe von CAD (1%). Auf der x-Achse ist der Paramter ε, auf der y-Achse die Anzahl der Ergebnistupel abgetragen. Beide Achsen sind in logarithmischer Skala abgetragen (*log-log Skala*). Abbildung 3.32 zeigt dasselbe Experiment für die Datensätze CT, CM und CH. Für alle vier Datensätze gilt, daß die Anzahl der Ergebnistupel für ‚kleine' ε nur gering ist und sehr langsam ansteigt. Ab einer bestimmten Schranke nimmt die Anzahl der Ergebnistupel dann stark zu und gehorcht obigem Power Law. Für die Bestimmung der fraktalen Dimension $\mathcal{P}$ wird die Anfangsphase mit wenigen Ergebnistupeln nicht berücksichtigt und nur der ‚interessanten' Teil [FSTT00] genutzt. Dieser Teil der Kurven läßt sich — wie man in den Graphiken gut erkennen kann — in der log-log-Skala mit einer Gera-

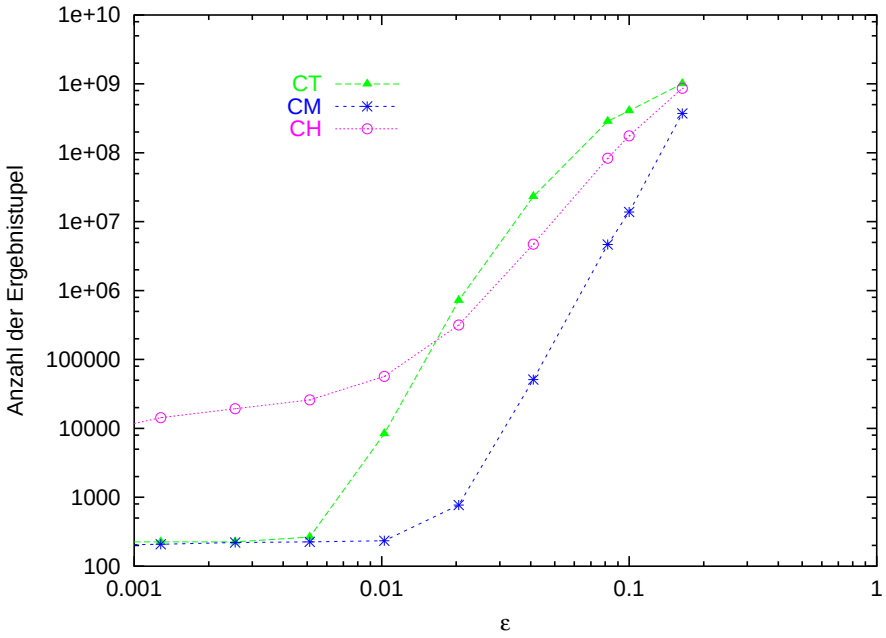


Abbildung 3.32: Anzahl der Ergebnistupel für Ähnlichkeitsselbstverbund von CT, CM und CH in Abhängigkeit von ε (log-log-Skala)

den approximieren. Die Steigung dieser Geraden entspricht der fraktalen Dimension des jeweiligen Datensatzes [BF95].

Hiermit ergeben sich folgende fraktale Dimensionen:

Datensatz	fraktale Dimension ($\mathcal{P}$)	einbettende Dimension (d)
CAD	2,9	16
CH	3,8	64
CM	6,4	9
CT	5,7	16

3.7.2 Erweiterung des Kostenmodells

Analytische Modelle, die auf der Gleichverteilungsannahme beruhen, lassen sich laut [BF98] auch für reale Datensätze verwenden. Hierzu wird die Dimension d im analytischen Modell durch die jeweilige fraktale Dimension des Datensatzes ersetzt. Mit den im letzten Abschnitt berechneten fraktalen Dimensionen können wir diese Aussage für das Kostenmodell für die Anzahl der Distanzberechnungen aus Abschnitt 3.3.2 überprüfen. Die Abbildungen 3.33

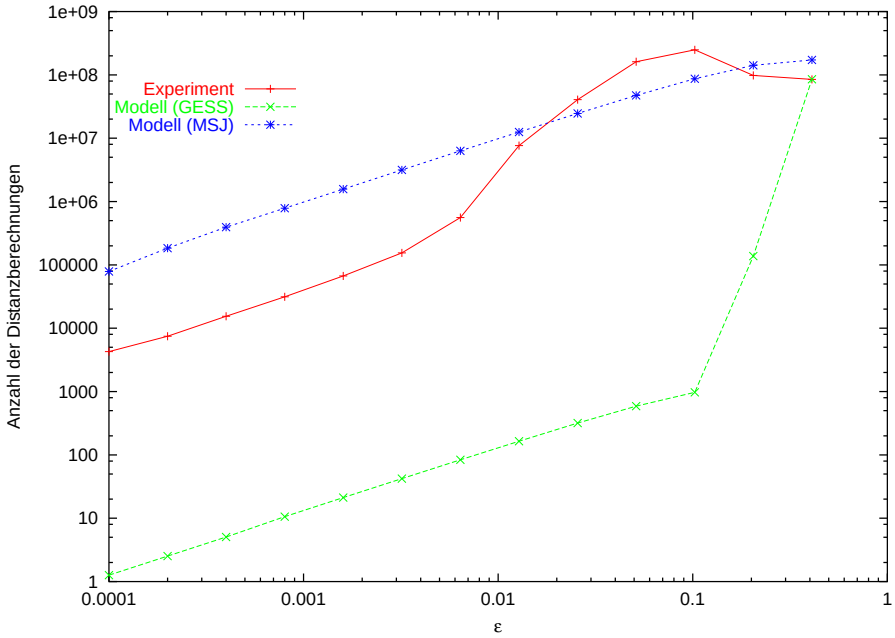


Abbildung 3.33: Vergleich von Kostenmodell und Experiment für Ähnlichkeitsselbstverbund von CAD (Stichprobe 1%, log-log-Skala)

bis 3.36 zeigen die Anzahl der Distanzberechnungen in Abhängigkeit von ε für Stichproben der Datensätze CAD, CT, CM und CH. Beide Achsen sind wieder logarithmisch abgetragen. Die rote Kurve zeigt jeweils die im Experiment gemessene Anzahl von Distanzberechnungen. Die grüne Kurve markiert die Voraussage des Kostenmodells von GESS, die blaue Kurve zeigt die Voraussage des Kostenmodells von MSJ. Für die Kostenmodelle wurde die jeweilige fraktale Dimension der Datensätze verwendet. Die Aufteilungsstrategie in den Experimenten war $msl = 0$ und $k = 3$.

Wir betrachten zunächst Abbildung 3.33. Die gemessene Anzahl von Distanzberechnungen liegt für dieses Experiment für $\varepsilon < 0,01$ im Bereich von 4000 bis über 500000. Ab $\varepsilon > 0,01$ steigt dieser Wert in stärkerem Maße an (auf über 10^7), was sich vermutlich durch hohe Replikationsraten erklärt.

Das Kostenmodell von GESS sagt für dieses Experiment Werte voraus, die für nahezu alle ε ca. 3 Größenordnungen unter dem experimentell gemessenen Wert liegen. Das Kostenmodell von MSJ liegt maximal eine Größenordnung neben dem tatsächlichen Wert.

Die Abbildung für den Datensatz CT (Abbildung 3.34) zeigt ein ähnliches Ergebnis. In diesem Experiment liegt die Kurve des Kostenmodells von MSJ sehr nahe am gemessenen Wert. Die Kurve für das Kostenmodell von GESS hingegen liegt für fast alle ε um mehr als 4 Größenordnungen unterhalb des gemessenen Wertes. Die Kurve des Kostenmodells

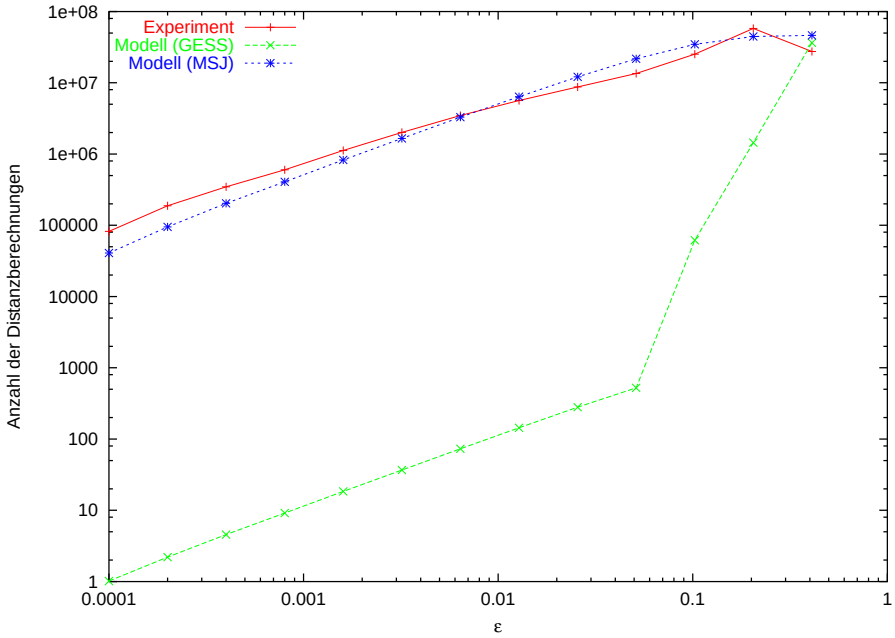


Abbildung 3.34: Vergleich von Kostenmodell und Experiment für Ähnlichkeitsselbstverbund von CT (Stichprobe 10%, log-log-Skala)

von liegt MSJ somit näher an der tatsächlichen Anzahl der Distanzberechnungen, obwohl in diesem Experiment Replikation erlaubt war(!). Für das Experiment mit dem Datensatz CM (Abbildung 3.35) liegt für $\varepsilon < 0.02$ die Kurve des Kostenmodells von GESS näher am gemessenen Wert. Für größere Werte von ε ist die Voraussage des Kostenmodells von MSJ besser. Die Abbildung erweckt den Eindruck, als würde die experimentell gemessene Anzahl der Distanzberechnungen in einem ‚Zwischenraum' liegen, der durch die beiden Kurven der Kostenmodelle aufgespannt wird.

Für den Datensatz CH (Abbildung 3.36) scheint das Kostenmodell von MSJ die im Experiment gemessene Anzahl von Distanzberechnungen exakt vorauszusagen. Allerdings ist hierbei zu beachten, daß in Abbildung 3.36 auf der y-Achse 25 Größenordnungen abgetragen sind. Abbildung 3.37 zeigt dasselbe Experiment deshalb für das y-Intervall 10^4 bis $2 \cdot 10^8$. Zusätzlich sind in dieser Abbildung weitere Kurven (gestrichelte Linien) eingezeichnet, die die Voraussagen des Kostenmodells von MSJ markieren, wenn in dieses beliebige Werte für die Dimension d eingesetzt werden ($d \in \{1, 2, 3, 6, 64\}$). Die Voraussagen des Kostenmodells für $d = 1$ und $d = 64$ spannen in der Abbildung einen Bereich von ca. 2 Größenordnungen auf. Die im Experiment gemessene Anzahl von Distanzberechnungen liegt für nahezu alle ε unter der Kurve für $d = 6$. Insgesamt läßt sich feststellen, daß das Kostenmodell von MSJ unter Einsetzen der fraktalen Dimension des Datensatzes ($d = 3, 8$) für dieses Experiment

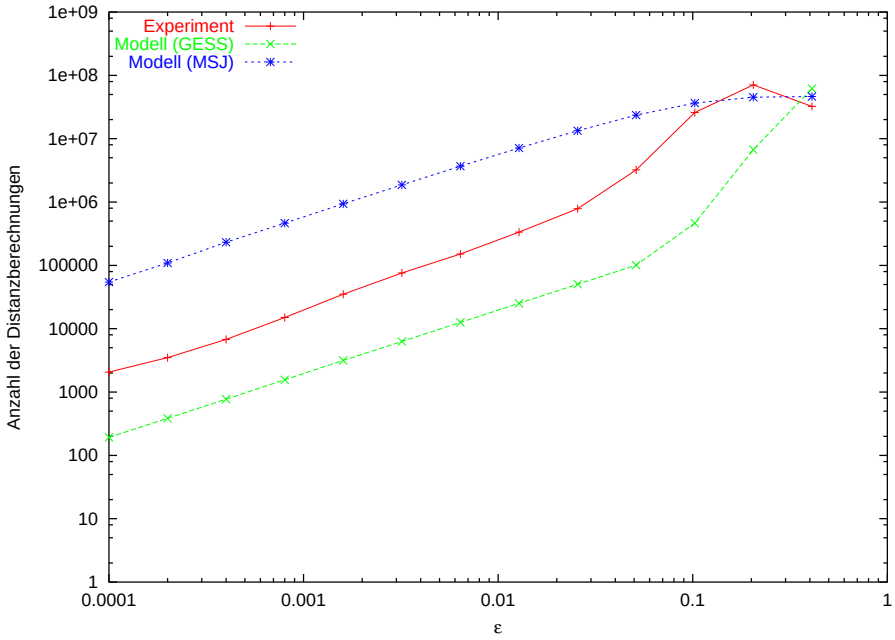


Abbildung 3.35: Vergleich von Kostenmodell und Experiment für Ähnlichkeitsselbstverbund von CM (Stichprobe 10%, log-log-Skala)

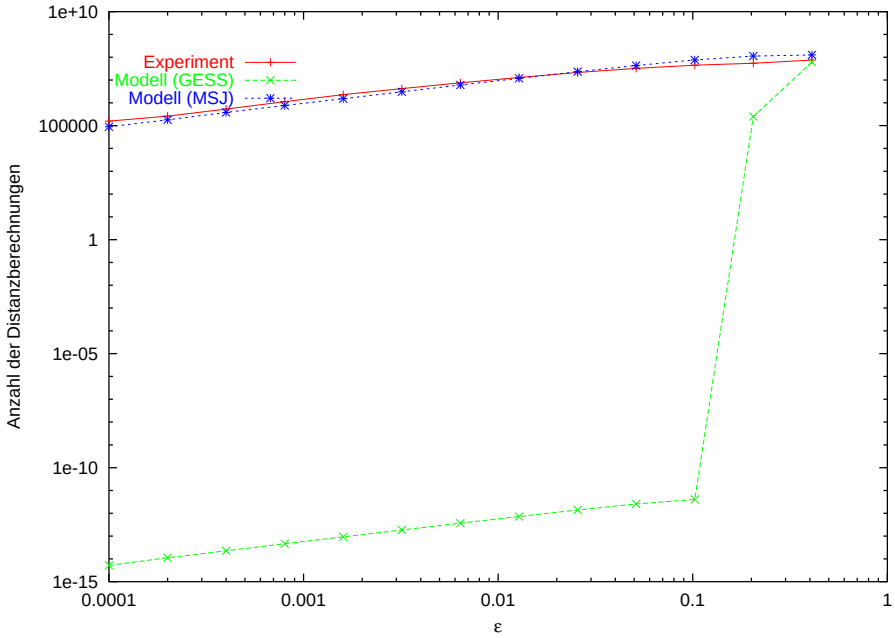


Abbildung 3.36: Vergleich von Kostenmodell und Experiment für Ähnlichkeitsselbstverbund von CH (Stichprobe 10%, log-log-Skala)

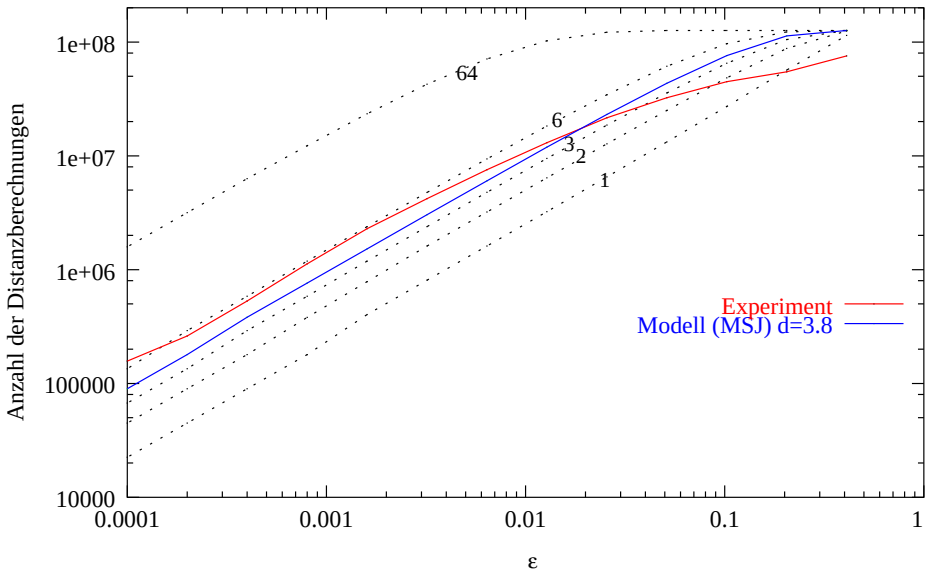


Abbildung 3.37: Vergleich von Kostenmodell und Experiment für Ähnlichkeitsselbstverbund von CH unter Variation der Dimension von 1–64 (Stichprobe 10%, log-log-Skala)

die tatsächlich anfallenden Distanzberechnungen recht gut voraussagt.

3.7.3 Diskussion

Insgesamt wirft der Vergleich von Experiment und Kostenmodell unter Einsetzen der fraktalen Dimension mehr Fragen auf als er beantwortet. Die Aussage aus [BF98] scheint durch die Experimente nicht bestätigt. Das Kostenmodell von GESS gibt die Situation für reale Datensätze nur bedingt wieder. Das Kostenmodell von MSJ liegt in einigen Experimenten (Datensätze CT und CH) erstaunlich nahe an der gemessenen Anzahl von Distanzberechnungen, obwohl in den Experimenten Replikation erlaubt war. In den Experimenten mit den Datensätzen CAD und CM scheint keines der beiden Kostenmodelle die reale Situation zufriedenstellend modellieren zu können.

Woran dieses liegt, kann verschiedenen Gründe haben: Zum einen reagiert das Kostenmodell von GESS sehr sensibel auf nicht-gleichverteilte Daten. Dies kann man sich an einem Beispiel klarmachen: für gleichverteilte Daten und der gewählten Aufteilungsstrategie ($msl = 0,\ k = 3$) ist der Erwartungswert für einen Hypercube mit $\varepsilon < 0,1$ mehr als 3 Aufteilungslinien in Ebene 0 zu schneiden nahe 0 (siehe Formel 3.8). Gibt es in einem echten Datensatz hingegen nur einen Hypercube, der in Ebene 0 mehr als 3 Aufteilungsebenenen schneidet, so verursacht dieser Hypercube alleine $||R||$ zusätzliche Distanzberechnungen. We-

nige Hypercubes können somit einen großen Einfluß auf die Voraussage des Modells haben.

Zum anderen war der Leistungsgewinn für die Datensätze CT und CH in den Experimenten bei der gewählten Aufteilungsstrategie nicht besonders groß, so daß gefolgert werden kann, daß Replikation bei dieser Strategie kaum Wirkung zeigt. Dies mag die Nähe zum Kostenmodell von MSJ erklären.

Es stellt sich aber auch die prinzipielle Frage, wie es möglich sein soll, daß ein einziger Wert, die fraktale Dimension eines Datensatzes, die ‚inhärente Eigenschaft' eines Datensatzes so weitgehend beschreibt, daß ein analytisches Modell unter Einsetzen dieses Wertes exakte Voraussagen liefert. Für einfache geometrische Objekte wie Linien, Flächen und fraktale Kurven wie die Sierpinski Kurve liefern die Experimente in [BF98] starke Belege für das Konzept der fraktalen Dimension. Es wäre aber denkbar, daß ein echter Datensatz nicht durch einen einzigen Wert zu beschreiben ist sondern vielmehr in Regionen unterschiedlicher fraktaler Dimensionen zerfällt. Die ‚fraktale Dimension eines Datensatzes' wäre in einem solchen Modell nur eine Aggregation der fraktalen Dimensionen der einzelnen Regionen.

3.8 Zusammenfassung

In diesem Abschnitt haben wir einen neuen generischen Similarity Join Algorithmus namens GESS vorgestellt. GESS nutzt kontrollierte Replikation, um die Gesamtlaufzeit und den Hauptspeicherbedarf zu reduzieren. Wir haben GESS analytisch und experimentell mit MSJ verglichen. Sowohl in unserem analytischen Modell als auch in unseren Experimenten haben wir festgestellt, daß GESS bezüglich Gesamtlaufzeit, I/O-Kosten und Hauptspeicherbedarf MSJ klar überlegen ist.

Unsere Experimente zeigen das hohe Potential unseres Ansatzes. Wie wir gesehen haben, spielen sowohl die Aufteilungsstrategie als auch der Parameter ε eine große Rolle, um eine gute Balance zwischen eingesparten Distanzberechnungen, zusätzlicher I/O und einer nicht zu großen Ergebnismenge zu erreichen.

Die Definition des ε-Similarity Join setzt voraus, daß à priori ein sinnvoller Wert für ε bestimmt werden kann. Mit Hilfe analytischer Modelle kann ein Schätzwert für ε zwar bestimmt werden, diese Modelle gehen jedoch meist von Gleichverteilung der Daten aus und schätzen ε um mehrere Größenordnungen zu groß.

Es ist zwar möglich, die fraktale Dimension [BF98] der genutzten Datensätze zu bestimmen und damit ein besseres ε zu schätzen [FSTT00]. Gerade diese Bestimmung der fraktalen

Dimension aber erfordert das mehrfache Ausführen eines Similarity Joins für verschiedene ε. D.h. es werden bereits im umfangreichem Maße Vorberechnungen durchgeführt, die weit über den Aufwand des eigentlichen Similarity Join hinausgehen. Ferner scheidet diese Vorberechnung aus, wenn der Eingabedatensatz dynamisch von einem unterliegenden Operator erzeugt wird. In diesem Fall ist es notwendig, zur Laufzeit das ε zu bestimmen.

Im nächsten Kapitel werden wir ein nichtblockierendes generisches Joinverfahren vorstellen. Dieses Verfahren liefert nicht nur nach sehr kurzer Zeit erste Ergebnistupel, sondern hat auch den großen Vorteil, daß das Joinprädikat – entweder durch Benutzerinteraktion oder aufgrund statistischer Informationen — zur Laufzeit verändert werden kann.

Kapitel 4

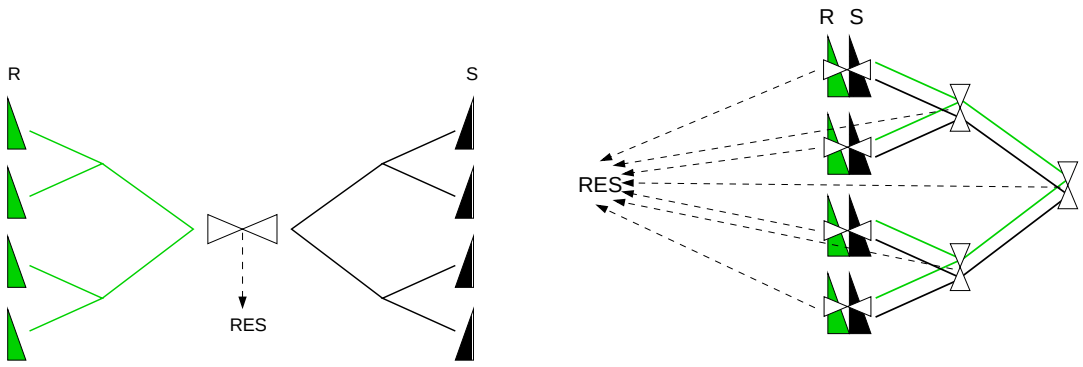
PMJ

Viele Joinalgorithmen erfordern es, daß die Eingabedaten vor der eigentlichen Joinverarbeitung komplett sortiert vorliegen. Dies bedeutet, daß diese Techniken erste Ergebnistupel erst dann liefern können, nachdem eine beträchtliche Zeitspanne für das Sortieren der Eingabedaten vergangen ist. Dieses *Blockieren* des Datenstroms ist ein ernstes Problem, wenn nachfolgende Operatoren auf die Ergebnistupel des Joins warten müssen. Ferner ist dieses Verhalten nicht akzeptabel, wenn das Ergebnis visualisiert werden soll und/oder Benutzerinteraktion erfordert. Letztere sind typische Anforderungen von Data Mining-Applikationen. Die *Auszeit* derzeit existierender Techniken steigt zudem noch mit der Problemgröße.

In diesem Kapitel schlagen wir deshalb einen generischen, sortierbasierten Algorithmus namens *Progressive Merge Join (PMJ)* vor, der den Datenstrom nicht blockiert. Die wesentliche Idee von PMJ ist es, beide Sortieroperatoren mit dem eigentlichen Joinoperator zu einem Operator zu verschmelzen. Frühe Ergebnistupel werden dann dadurch produziert, daß beide Datensätze vom Beginn der Run-Erzeugung an gleichzeitig sortiert und Teilfolgen sehr früh verknüpft werden. Hierdurch können Ergebnistupel fortlaufend produziert und extrem früh an den nächsten Operator weitergereicht werden.

PMJ basiert auf einer sogenannten generischen *Sweep Area*, die auf alle sortierbasierten Joinverfahren wie Band Joins [DNS91], Temporal Joins [SSJ94], Spatial Joins [APR+98] und Similarity Joins [Ore91, DS01] angepaßt werden kann. D.h. sehr viele existierende Algorithmen können von dem hier vorgestellten Verfahren profitieren.

Diese Kapitel stellt den grundlegenden Algorithmus von PMJ sowie Anwendungsfälle mit verschiedenen Jointypen vor. Insbesondere für den Similarity Join präsentieren wir das erste nichtblockierende Verfahren durch Anwendung von PMJ auf GESS.



(a) Sort-Merge Join

(b) Progressive Merge Join

Abbildung 4.1: Sortierbasierter Join zweier Datensätze R und S

In einer Reihe von Experimenten zeigen wir, daß eine beträchtliche Anzahl von Ergebnistupeln produziert wird, noch bevor die Sortierphase der konventionellen Techniken beendet ist. Unsere Ergebnisse zeigen nur einen geringen Anstieg der Laufzeit der Algorithmen im Vergleich zu den entsprechenden blockierenden Techniken.

Im folgenden werden zunächst die wichtigsten Algorithmen von PMJ präsentiert (Abschnitt 4.1). Abschnitt 4.2 und Abschnitt 4.3 beweisen die Vollständigkeit und Eindeutigkeit von PMJ. Abschnitt 4.4 beschreibt die verschiedenen Möglichkeiten zur Realisierung der Eindeutigkeit von PMJ. Abschnitt 4.5 diskutiert die Komplexität unseres Verfahrens. Abschnitt 4.6 präsentiert Resultate von Experimenten mit PMJ.

4.1 Progressive Merge Join

In Analogie zum externen Sortieren [Knu73] besteht PMJ aus 2 Phasen. Im folgenden beschreiben wir kurz beide Phasen und betrachten danach beide Phasen ausführlich in separaten Unterabschnitten.

Seien R und S zwei Eingabemengen. In der ersten Phase beginnt PMJ zunächst damit, Teilmengen von R und S in den verfügbaren Hauptspeicher zu lesen. Beide Teilmengen werden dann im Hauptspeicher mit einem Verfahren wie z.B. Quicksort sortiert. Die sortierten Folgen werden anschließend verknüpft. Ergebnistupel des Joins werden direkt an den nächsten Operator weitergeliefert [Gra93].

PMJ lädt Teilmengen in den Hauptspeicher, sortiert und verknüpft diese, bis alle Elemente der Eingabemengen behandelt wurden. Anstatt Quicksort zu nutzen, kann für PMJ auch

Replacement Selection [Knu73] genutzt werden, welches den Vorteil hat, daß die erzeugten Ausgabefolgen im Schnitt doppelt so lang sind wie bei Quicksort. Dies führt außerdem zu einer höheren Zahl von Ergebnistupeln in der ersten Phase. Replacement Selection wird auch in unserer Implementierung genutzt. Um die Algorithmen übersichtlich zu halten, haben wir dieses Detail in den folgenden Beschreibungen ausgelassen.

In der zweiten Phase mischt PMJ die in der ersten Phase erzeugten Folgen zu längeren Folgen zusammen. Wiederum werden hierbei bereits Ergebnistupel durch eine Kombination aus Mischen und Verknüpfen produziert. Die erzeugten Folgen werden temporär auf den Externspeicher geschrieben (mit Ausnahme der letzten Mischoperation).

Betrachten wir zunächst den Algorithmus in Abbildung 4.2.

Der Algorithmus erwartet 4 Eingabeparameter: die Eingabemengen R und S, die Anzahl der Seiten, die im Hauptspeicher gehalten werden können (m), und den maximalen Fanin eines Merges (F). $|R|$ bzw. $|S|$ bezeichne die Anzahl der Seiten in R bzw. S. $||Q||$ bezeichne die Anzahl der Elemente einer Menge Q.

In der ersten Phase beginnt PMJ zunächst damit, sortierte Teilfolgen ('Runs') zu erzeugen. Dies entspricht der Run-Erzeugung des Externen Sortierens, aber im Gegensatz hierzu verarbeitet PMJ zwei Eingabemengen gleichzeitig (siehe Zeilen 2 und 3 in Abbildung 4.2), wobei $|\hat{R}| + |\hat{S}| \leq m$ gilt.

Der Einfachheit halber nehmen wir an, daß $|\hat{R}| \simeq |\hat{S}| \simeq m/2$. Dies ist nicht vorgeschrieben und kann für Optimierungen verändert werden. Beide Teilmengen werden mittels eines Verfahrens im Hauptspeicher (z.B. mit Quicksort) in sortierte Teilsequenzen $\hat{R}'$ und $\hat{S}'$ sortiert (Zeile 5).

Die sortierten Teilfolgen werden dann im Hauptspeicher verknüpft (Zeile 6). Das Paar ($\hat{R}'$, $\hat{S}'$) wird in der Menge Q zur weiteren Bearbeitung gespeichert (Zeile 8). Die sortierten Teilfolgen werden temporär auf den Externspeicher geschrieben (Zeile 7).

PMJ erzeugt solange Paare von sortierten Teilfolgen, bis alle Eingabeelemente verarbeitet sind (Zeile 1).

4.1.1 Join während der Run-Erzeugung

Der interne Joinalgorithmus der ersten Phase wird gerufen mittels der Methode `earlyJoinInitialRuns`, siehe Abbildung 4.4. Diese Methode basiert auf dem *Plane Sweep Paradigma* [PS85], auf welches wir separat eingehen werden. Entscheidend für dieses Ver-

Gegeben:

-Eingabemengen R und S

-Anzahl der Seiten m, die in den Hauptspeicher passen

-Maximaler Fanin eines Merges F

Sei Q eine leere Menge;
Sei RES die leere Ergebnismenge;

Phase 1 („Join während der Run-Erzeugung"):

1 Solange $(R \neq \{\} \ \vee \ S \neq \{\})$ {
2 Sei $\hat{R}$ eine Teilmenge von R und $\hat{S}$ eine Teilmenge von S
3 wobei $|\hat{R}| + |\hat{S}| \leq m$
4 $R = R \backslash \hat{R}; \ S = S \backslash \hat{S};$
5 Sortiere $\hat{R}$ in Folge $\hat{R}'$, sortiere $\hat{S}$ in Folge $\hat{S}'$;
6 $RES = RES \cup$ **earlyJoinInitialRuns**$(\hat{R}', \hat{S}')$;
7 Schreibe $\hat{R}'$ und $\hat{S}'$ auf den Externspeicher;
8 $Q = Q \ \cup \ \{(\hat{R}', \hat{S}')\}$;
9 }

Phase 2 („Join während des Mischens"):

10 Solange$(||Q|| > 1)$ {
11 Sei $\hat{Q}$ eine Teilmenge von Q, $||\hat{Q}|| \leq F/2$;
12 Sei $(\hat{R}', \hat{S}')$ ein Tupel bestehend aus zwei leeren Folgen;
13 $RES = RES \cup$ **earlyJoinMergedRuns**$(\hat{Q}, \ (\hat{R}', \hat{S}'))$;
14 Schreibe $\hat{R}'$ und $\hat{S}'$ auf den Externspeicher;
15 $Q = \{Q \backslash \hat{Q}\} \ \cup \ \{(\hat{R}', \hat{S}')\}$;
16 }

Abbildung 4.2: Progressive Merge Join

```
1  Klasse SweepArea {
2        Prädikat P_join;
3        Prädikat P_rm;
4        Sei X = {};
5        Methode insert(Element v) {
6              X = X ∪ {v};
7        }
8        Methode query(Element v) {
9              X = X \ {u | P_rm(u,v), u ∈ X};
10             return { (u,v) | P_join(u,v), u ∈ X};
11       }
12 }
```

Abbildung 4.3: Der Datentyp `SweepArea`

fahren ist eine Hauptspeicherstruktur namens *Sweep Area*, siehe Abbildung 4.3, welche eine Datenmenge mit Bezug auf ein bestimmtes Löschprädikat speichert.

Die sortierten Folgen werden elementweise verarbeitet. Zunächst wird ermittelt, ob eine der beiden Eingabefolgen R' und S' leer ist (siehe Zeile 5 und 6, Abbildung 4.4). Nur für die nichtleeren Eingabefolgen wird das erste Element der Folge betrachtet und an r bzw. s zugewiesen (siehe Zeilen 7–12). Es wird jeweils das kleinste Element bezüglich der verwendeten Ordnung gewählt bzw. falls eine Eingabefolge leer ist, wird die nichtleere Folge zur weiteren Bearbeitung ausgewählt (siehe Zeile 13). Das gewählte Element wird dann in die zugehörige Sweep Area eingefügt (siehe Zeilen 14 bzw. 19). Dann wird dieses Element genutzt, um Anfragen an die Sweep Area der anderen Datenmenge zu stellen (Zeilen 15 bzw. 20). Die Ergebnistupel dieser Anfragen sind Ergebnistupel des Joins.

Anfragen an die Sweep Areas lösen zudem Löschoperationen von Elementen in der Sweep Area aus: vor jeder Anfrage werden alle diejenigen Elemente gelöscht, die nicht mehr als Ergebnistupel einer Anfrage auftauchen können (siehe Zeile 9 von Abbildung 4.3). Hierdurch verbessert sich die Effizienz dieser Anfragen erheblich.

Wir möchten an dieser Stelle auch nochmals darauf hinweisen, daß das eigentliche Joinprädikat nur in der Sweep Area auftaucht. Dies ist der eigentliche Grund, warum unser Joinverfahren für eine große Klasse von Joins anwendbar ist.

Für den Equi Join wählen wir die Prädikate wie folgt: das Joinprädikat $P_{join}(u, v)$ testet,

```
1  Methode earlyJoinInitialRuns (Folgen R', S') {
2        Sei RES = {};
3        SweepArea R_M = {}, S_M = {};
4        Solange ( R' ≠ {} ∨ S' ≠ {} ) {
5              bool R_empty = (R' == {});
6              bool S_empty = (S' == {});
7              Falls (¬R_empty ) {
8                    r = First(R');
9              }
10             Falls (¬S_empty ) {
11                   s = First(S');
12             }
13             Falls (¬R_empty ∧ (S_empty ∨ r ≤ s)) {
14                   R_M.insert(r);
15                   RES = RES ∪ S_M.query(r);
16                   R' = R'\{r};
17             }
18             Sonst {
19                   S_M.insert(s);
20                   RES = RES ∪ R_M.query(s);
21                   S' = S'\{s};
22             }
23       }
24       return RES;
25 }
```

Abbildung 4.4: Die Methode `earlyJoinInitialRuns`

ob u und v gleich sind. Das Löschprädikat $P_{rm}(u, v)$ testet, ob u kleiner als v ist[1].

4.1.2 Join während des Mischens

In der zweiten Phase mischt PMJ die sortierten Teilfolgen ('Runs') zu größeren zusammen. Dies entspricht der Mischphase des externen Sortierens, aber im Gegensatz hierzu mischt PMJ gleichzeitig Teilfolgen beider Datenmengen. Der Parameter F bezeichnet den maximalen Fanin, d.h. die maximale Anzahl von Folgen, die von einer Operation gemischt werden können. Q enthält alle Tupel von Folgen, die noch bearbeitet werden müssen. PMJ wählt eine Teilmenge von Tupeln $\hat{Q} \subseteq Q$, wobei gilt $||\hat{Q}|| \leq F/2$ (Zeile 11 in Abbildung 4.2). Hiernach wird die Methode `earlyJoinMergedRuns` gerufen mit der Teilmenge $\hat{Q}$ und dem Tupel $(\hat{R}', \hat{S}')$, welche zwei leere Folgen enthält, die als Ausgabefolgen genutzt werden (Zeilen 12/13 in Abbildung 4.2). Die Methode `earlyJoinMergedRuns` (siehe Abbildung 4.5) mischt die Eingabefolgen zu zwei größeren Folgen $\hat{R}', \hat{S}'$ zusammen, welche dann direkt verknüpft und auf den Externspeicher geschrieben werden (Zeile 14, Abbildung 4.2). Man beachte, daß das Schreiben und Verknüpfen der Folgen nicht nacheinander erfolgt — wie der Leser von der Beschreibung vermuten mag — sondern gleichzeitig, um das temporäre Speichern von Ausgabefolgen zu vermeiden. Schließlich wird das Paar sortierter Folgen $(\hat{R}', \hat{S}')$ zu Q hinzugefügt (Zeile 15, Abbildung 4.2). Das Mischen und Joinen der Folgen wird solange wiederholt (Zeile 10, Abbildung 4.2) bis Q nur noch ein einziges Tupel (R', S') enthält, d.h. zwei sortierte Folgen der Eingabemengen R und S.

Die Methode `earlyJoinMergedRuns` (siehe Abbildung 4.5) ist der wichtigste Teil des Algorithmus. Diese Methode erzeugt gleichzeitig sowohl zwei gemischte Folgen der Eingabemengen als auch deren Ergebnistupel.

Zunächst wird überprüft, ob für beide Datensätze R und S noch Elemente in den zugehörigen Folgen R'_i und S'_i vorhanden sind (siehe Zeilen 5 und 6). Aus den Folgen R_i und S_i wird danach das kleinste Element bezüglich der verwendeten Ordnung gewählt und an r bzw. s zugewiesen (siehe Zeilen 8 bzw. 11). Anschließend wird das kleinste Element von r und s bezüglich der verwendeten Ordnung gewählt bzw. falls eine Eingabe leer ist, wird die nichtleere Folge zur weiteren Bearbeitung ausgewählt (Zeile 13). Dieses Element wird als nächstes verarbeitet und zunächst in die zugehörige Sweep Area eingefügt (Zeilen 14 bzw. 21). Dann wird eine Anfrage an die andere Sweep Area ausgeführt (Zeilen 15 bzw.

[1]Nachdem das Löschprädikat bearbeitet wurde, enthält die Sweep Area für den Equi Join nur noch Ergebnistupel, so daß das eigentliche Joinprädikat für diesen Anwendungsfall nicht mehr ausgeführt werden muß.

```
1  Methode earlyJoinMergedRuns( { (R'_0, S'_0), ..., (R'_n, S'_n) }, (R̂', Ŝ') ) {
2      Sei RES = {};
3      SweepArea R_M = {}, S_M = {};
4      Solange (∃m : R_m ≠ {} ∨ ∃S_m ≠ {}) {
5          bool R_empty = (∀_i : R'_i == {});
6          bool S_empty = (∀_i : S'_i == {});
7          Falls (¬R_empty ) {
8              r = min_{i∈{ l | R'_l≠{}}} First(R'_i);
9          }
10         Falls (¬S_empty ) {
11             s = min_{j∈{ l | S'_l≠{}}} First(S'_j);
12         }
13         Falls (¬R_empty ∧ (S_empty ∨ r ≤ s)) {
14             R_M.insert(r);
15             RES = RES ∪ S_M.query(r)
16                                \ {(r, x)| x ∈ S_i};
17             R'_i = R'_i \ {r};
18             R̂' = R̂' ∪ {r};
19         }
20         Sonst {
21             S_M.insert(s);
22             RES = RES ∪ R_M.query(s)
23                                \ {(x, s)| x ∈ R_j};
24             S'_j = S'_j \ {s};
25             Ŝ' = Ŝ' ∪ {s};
26         }
27     }
28     return RES;
29 }
```

Abbildung 4.5: Die Methode `earlyJoinMergedRuns`

22). Um Duplikate zu eliminieren, müssen alle Ergebnistupel verworfen werden, die bereits vorher berichtet wurden (Zeilen 16 bzw. 23). Nach dieser Duplikateliminierung entfernt der Algorithmus das Element aus der zugehörigen Eingabefolge (Zeilen 17 bzw. 24) und hängt es an die Ausgabefolge an (Zeilen 18 bzw. 25).

Dieser Prozeß des Mischens und Verbindens von Daten wird solange fortgeführt, bis alle Eingabefolgen vollständig konsumiert wurden (Zeile 4).

Im nachfolgenden Abschnitt werden wichtige Erweiterungsmöglichkeiten von PMJ beschrieben. Die nachfolgenden Abschnitte zeigen die Vollständigkeit (4.2) sowie Eindeutigkeit (4.3) von PMJ. Abschnitt 4.4 beschreibt verschiedene Möglichkeiten der Realisierung der Eindeutigkeit. Anschließend wird die Effizienz von PMJ betrachtet (Abschnitt 4.5). Abschnitt 4.6 präsentiert Resultate von Experimenten mit PMJ.

4.1.3 Erweiterungsmöglichkeiten

In diesem Abschnitt möchten wir auf einige mögliche Erweiterungen unseres Algorithmus hinweisen.

- **Semijoin:** Zunächst ist PMJ nicht darauf beschränkt den *Inner Join* zweier Datenmengen zu berechnen, sondern kann auch für die Berechnung des *Outer Join* bzw. des *Semijoin* verwendet werden. Für einen Semijoin der Relationen R und S erreicht man einen Leistungsgewinn gegenüber dem konventionellen Ansatz folgendermaßen: wann immer PMJ ein Ergebnistupel für $r \in R$ generiert, braucht r nicht mehr weiter betrachtet werden, d.h. r wird nicht mehr in die Ausgabefolge geschrieben (siehe Zeile 18 in Abbildung 4.5). Dies reduziert die I/O-Kosten des Verfahrens.

  Für den Semijoin ergeben sich weitere Vorteile durch die online Fähigkeit von PMJ: der Semijoin wird z.B. typischerweise genutzt, um einen Netzwerkflaschenhals zu umgehen. Dieses Verfahren besteht aus zwei aufeinanderfolgenden, blockierenden Joinoperationen.

  1. Zunächst wird mittels eines Selbstverbunds auf Rechner $R1$ ermittelt, welche Elemente an einem Join teilhaben.
  2. Danach werden die vollständigen Elemente — und zwar nur die für den Join benötigten — von Rechner $R1$ an Rechner $R2$ verschickt und der Join auf Rechner $R2$ durchgeführt.

  Man beachte, daß sich dieses Verfahren nur für eine gewisse Selektivität rentiert, d.h.

wenn $||\delta_R(R \bowtie S)|| << ||R||$ gilt.

Beide Joinoperationen für dieses Szenario sind blockierend, d.h. Rechner $R1$ kann das erste vollständige Tupel erst dann verschicken, wenn der Semijoin das erste Ergebnistupel produziert. Der nachfolgende Join auf Rechner $R2$ kann erst dann ein Endergebnis produzieren, wenn alle Ergebnistupel des Semijoins eingetroffen sind.

Wird für den Semijoin und den nachfolgenden Join PMJ eingesetzt, können sofort Ergebnistupel von Rechner $R1$ nach Rechner $R2$ verschickt werden. Rechner $R2$ kann sofort mit der Joinberechnung beginnen und braucht nicht zu warten.

- **Speicherverteilung:** Die Speicheraufteilung von PMJ muß nicht vor Ausführung des Joins festgelegt werden, sondern kann zur Laufzeit dynamisch den Gegebenheiten angepaßt werden. Dies ist von großem Vorteil, wenn der verfügbare Hauptspeicher nicht konstant ist. Die vom speicheradaptiven Sortieren bekannten Techniken [ZL97] können ebenfalls für PMJ verwendet werden.

- **Erster Merge:** Der Parameter M ist derzeit festgelegt durch den verfügbaren Hauptspeicher. Um sehr früh Ergebnistupel weiterreichen zu können, kann es Sinn machen, einen kleineren Wert für M zu wählen als möglich. Zwar verringert sich hierdurch die Gesamtzahl der in der ersten Mischoperation gefundenen Ergebnistupel, allerdings sinkt der Aufwand für das Aufbauen der beiden Heaps für Replacement Selection.

  Sei Z die Anzahl der Ergebnistupel des Joins $R \bowtie S$, d.h. $Z = ||R \bowtie S||$. Es gelte o.B.d.A. $N = ||R|| = ||S||$. Wir nehmen an, daß genau $Z' < Z$ Ergebnistupel gesucht sind. Bei Nutzung von Replacement Selection und einem verfügbaren Hauptspeicher für M Elemente erzeugt PMJ für das erste Paar von Eingabefolgen eine erwartete Anzahl von $(M/N)^2 \cdot Z$ Elementen. Die I/O-Kosten hierfür sind genau $2m$ ($m = \lfloor M/B \rfloor$)[2]. D.h. bezüglich I/O-Kosten sollte m maximal gewählt werden. Wir nehmen an, daß $Z' < (M/N)^2 \cdot Z$. Dann könnten wir M verringern auf

$$M_{neu} = N \cdot \sqrt{\frac{Z'}{Z}},$$

  um die geschätzte Anzahl $\hat{Z}'$ von Ergebnistupeln bei minimaler I/O zu bekommen. Dieses Vorgehen entspricht dem *Budgeted Sort-Merge Join*, den wir in [DSTW02a] beschrieben haben. Hierbei ist noch nicht die CPU-Zeit für die Heaps von Replacement Selection berücksichtigt. Diese benötigen am Anfang zunächst $O(M/2)$ Tauschoperationen, um die Heap-Eigenschaft herzustellen.

[2] 2 Runs der Länge m werden geschrieben

Im Rahmen dieser Arbeit kommen wir im experimentellen Teil dieses Abschnitts auf den diskutierten Effekt zurück.

- **Selbstverbund:** Der Selbstverbund hat insbesondere für Data Mining Applikationen wie Clustering [BBBK00] und Outlier-Detection [BKNS00] eine große Bedeutung. Die Sweep Area für solche Joinalgorithmen ist i.d.R. relativ klein (vgl. Abschnitt 3.6.1.2). Vernachlässigt man den Hauptspeicherbedarf der Sweep Areas für die Berechnung der I/O-Kosten für einen Selbstverbund von PMJ, dann entsprechen diese exakt denen des semistrikten Sort-Merge Joins (nicht nur asymptotisch). Dies macht PMJ für Data Mining Applikationen sehr interessant.

- **Unendliche Datenströme:** Seit kurzem sind Algorithmen für unendliche Datenströme (*continuous queries*) ins Blickfeld der Datenbankforschung gerückt [GKS01, MF02]. Ein wichtiges Joinverfahren für solche Situationen ist der *XJoin*[UF00]. Der XJoin besteht aus drei separaten Threads. Liefern die Quellen temporär keine Daten mehr, kann XJoin dies erkennen und auf einen anderen Thread umschalten, der dann die in externen Hashbehältern zwischengespeicherten Daten weiterbearbeitet.

  In Analogie hierzu könnte eine parallele Implementierung von PMJ bei Blockieren einer Eingabe dazu übergehen interne Mischoperationen auszuführen, um die Wartezeit auf die Eingaben zu nutzen.

- **Generelle Erweiterungsmöglichkeiten:** Grundsätzlich kann PMJ durch die Techniken verbessert werden, die auch für externes Sortieren anwendbar sind.

- In einer separaten Arbeit [DSTW02a] haben wir Erweiterungsmöglichkeiten von PMJ betrachtet, die die **Traversierungsstrategie des Mischbaums** betreffen. Neben einem detaillierten theoretischen Modell, das die Anzahl der Ergebnistupel bezüglich eines vorgegebenen Kostenmaßes voraussagt, untersuchen wir die Komplexität verschiedener Traversierungsstrategien des Mischbaums, und modellieren ihre Auswirkung auf die Erzeugung früher Ergebnistupel.

4.2 Vollständigkeit

In diesem Abschnitt zeigen wir die Korrektheit von PMJ. Hierfür müssen wir zunächst zeigen, daß PMJ alle Tupel (u, v) des Ergebnisses findet (Vollständigkeit). Zunächst betrachten wir zwei Regeln, die für den generischen Algorithmus erfüllt sein müssen.

Die wesentliche Idee des *Plane Sweep Paradigmas* ist es, für jede Eingabemenge ein kleines

Datenfenster (unter Berücksichtigung der verwendeten Ordnung) zu definieren. Man beachte, daß unser Verfahren Ähnlichkeiten zu SHJ [WA91] zeigt, wenn wir keine Elemente aus der Sweep Area entfernen. Wäre die Sweep Area als Liste organisiert, würde das Verfahren einem symmetrischen Nested-Loops entsprechen. Es ist eine Frage der Effizienz, daß das Löschprädikat P_{rm} vor der Anfrage überprüft wird. Dies hält die Sweep Area klein und reduziert die Anzahl der Objekte, die in der nachfolgenden Anfrage berücksichtigt werden müssen. Damit das Verfahren alle Ergebnistupel des Joins findet, müssen das Löschprädikat P_{rm} und das Joinprädiakt P_{join} folgende Bedingungen $\forall u$ und $\forall v$ erfüllen:

$$\forall w > v : P_{rm}(u, v) \Longrightarrow \neg P_{join}(u, w) \tag{4.1}$$

$$\forall w \leq v : \neg P_{rm}(u, v) \Longrightarrow \neg P_{rm}(u, w) \tag{4.2}$$

Die erste Bedingung stellt folgendes sicher: wenn v das Element w aus der Sweep Area löscht, können alle Elemente größer als v kein Joinpartner von u sein. Die zweite Bedingung garantiert, daß, falls v das Element u nicht aus der Sweep Area löscht, alle Elemente $w \leq v$ das Element u nicht aus der Sweep Area entfernen. Beide Bedingungen sind notwendig für die Korrektheit des Joins. Insbesondere gilt das folgende Lemma.

Lemma 3. *Sei P_{rm} ein Löschprädikat, das die Bedingungen 4.1 und 4.2 erfüllt. Seien R' und S' zwei sortierte Folgen. Dann gibt die Methode* `earlyJoinInitialRuns` *(siehe Abbildung 4.4) $R' \bowtie S'$ zurück.*

Als eine direkte Konsequenz aus Lemma 3 ergibt sich, daß der konventionelle Sort-Merge Join von R und S korrekt ist, wenn er in unserem algorithmischen Rahmen ausgeführt wird. Dies liegt daran, daß für diesen Fall die Eingabemengen zunächst vollständig sortiert werden und dann die Methode `earlyJoinInitialRuns` gerufen wird mit (R', S') als Eingabeparameter.

Theorem 2 (Vollständigkeit von PMJ). *Gegeben zwei Eingabemengen R, S und ein binäres Joinprädikat P_{join}. Dann gilt: PMJ berichtet alle Ergebnistupel von $R \bowtie S$.*

Beweis [Vollständigkeit von PMJ]

Wir zeigen, daß jedes Ergebnistupel des konventionellen Sort-Merge Joins auch von PMJ berichtet wird. Sei (u, v) ein Ergebnistupel, das vom konventionellen Sort-Merge Join berichtet wird und nehmen wir o.B.d.A. an, daß $u \leq v$ gilt und u vor v in die Sweep Area eingefügt wurde. Wir folgern, daß $P_{join}(u, v)$ gelten muß. Mit Bedingung 4.1 folgt $\neg P_{rm}(u, v)$.

Betrachten wir nun die Situation für PMJ. PMJ mischt die Daten in einem *Mischbaum* für die Datenmengen R und S. Wir stellen sicher, daß beide Mischbäume identisch sind, indem wir leere Knoten hinzufügen (welche leere Mengen repräsentieren), falls notwendig.

Join-algorithmus	Datentyp	Ordnung	$P_{join}(u,v)$	$P_{rm}(u,v)$
Equi Join	Attribut A	A	$u.A = v.A$	$u.A < v.A$
Band Join	Attribut A	A	$\|u.A - v.A\| \leq \varepsilon$	$u.A < v.A - \varepsilon$
Temporal Join	Intervall $I = [min, max]$	I.min	$u.I \cap v.I \neq \emptyset$	$u.I.max < v.I.min$
Spatial Join	Rechteck $R = [I_x, I_y]$	R[0].min	$u.R \cap v.R \neq \emptyset$	$u.R[0].max <$ $v.R[0].min$
Similarity Join	Punkt P	ZCode(P)	Euclid$(u.P, v.P) \leq \varepsilon$	ZCode($u.P$) ist nicht Präfix von ZCode($v.P$)

Tabelle 4.1: Die Join- und Löschprädikate sowie Sortierungen verschiedener Joinalgorithmen

Wir unterscheiden zwei Fälle:

1. **Fall:** u und v gehören zur selben Folge (Blatt des Mischbaums). Für eine solche Folge wird die Methode `earlyJoinInitialRuns` gerufen. Wegen Lemma 3 gilt dann: (u, v) ist in der Antwortmenge von PMJ enthalten.

2. **Fall:** u und v gehören nicht zur selben Folge. Dann gibt es einen Knoten im Mischbaum, wo sich die Elemente u und v zum ersten Mal treffen, d.h. das Ergebnistupel (u, v) konnte nicht vorher erzeugt werden. Die Methode `earlyJoinMergedRuns` (siehe Abbildung 4.3) berechnet die Mischoperation. Da $u \leq v$, wird u in die Sweep Area eingefügt (bevor v eingefügt wird). Daraus folgt: die nachfolgende Anfrage an die Sweep Area kann das gesuchte Ergebnistupel (u, v) nicht produzieren.

Als nächstes betrachten wir die Situation, wenn v in die Sweep Area eingefügt wird. Wir zeigen, daß u zu diesem Zeitpunkt noch in der Sweep Area enthalten sein muß und deshalb PMJ das Joinergebnis (u, v) berechnet. Da (u, v) ein Ergebnistupel des Joins ist, gilt $P_{join}(u, v)$ und aus Bedingung 4.1 folgt $\neg P_{rm}(u, v)$. Mit Hilfe von Bedingung 4.2 erhalten wir $\forall w \in S,\ u \leq w \leq v :\ \neg P_{rm}(u, w)$. Da nur letztere Elemente von S verarbeitet werden während des Einfügens von u und v, folgt, daß u in der Sweep Area enthalten sein muß, wenn v verarbeitet wird. □

Im folgenden zeigen wir, daß verschiedene Arten von Joins in unserem algorithmischen Framework unterstützt werden. Tabelle 4.1 zeigt für verschiedene Jointypen ihre entsprechenden Prädikate und Ordnungen. Zusätzlich ist der zugrundeliegende Datentyp der Joins aufgeführt. Z.B. ist der Spatial Join (Filterschritt) für Rechtecke definiert, wobei wir unter einem Rechteck eine feste Zahl von d Intervallen verstehen. Der Similarity Join ist definiert für Punkte, wobei die Joinbedingung auf der euklidischen Distanz zweier Punkte (oder auf einem anderen Distanzmaß) basiert. Sowohl die Sortierung als auch das Löschprädikat des

Similarity Join basieren auf den Z-Werten der hochdimensionalen Punkte [Ore86].

Lemma 4. *Das Löschprädikat der verschiedenen Joins erfüllt die Bedingungen 4.1 und 4.2.*

Beweis [Spatial Join]

Wir beweisen obiges Lemma beispielhaft für den Spatial Join. Seien u und v zwei Tupel. Wir zeigen, daß das Löschprädikat die beiden Bedingungen erfüllt:

1. Zunächst nehmen wir an, daß $P_{rm}(u, v)$ gilt. D.h. u löscht v aus der Sweep Area. Daraus folgt $u.R[0].max < v.R[0].min$. Sei w ein Tupel und gelte $w > v$. Es folgt, daß $w.R[0].min > v.R[0].min$ $(> u.R[0].max)$. Folglich gilt $P_{rm}(u, w)$.

2. Nun nehmen wir an, daß $P_{rm}(u, v)$ nicht gilt. Sei w ein Tupel mit $w \leq v$. Dann gilt $w.R[0].min \leq v.R[0].min \leq u.R[0].max$. Folglich gilt nicht $P_{rm}(u, w)$.

□

4.3 Eindeutigkeit

In diesem Abschnitt zeigen wir, daß jedes Ergebnistupel des Joins genau einmal berichtet wird. Die Methode `earlyJoinMergedRuns` von PMJ berichtet nicht nur die gefundenen Ergebnistupel sondern entfernt auch mögliche Duplikate (siehe Zeilen 16 und 23 in Abbildung 4.5).

Theorem 3 (Eindeutigkeit von PMJ). *PMJ berichtet jedes Ergebnistupel des Joins genau einmal.*

Beweis [Eindeutigkeit von PMJ] Der Beweis wird mittels Induktion über die Höhe des Mischbaums geführt.

$\mathbf{n = 0}$: Für einen Mischbaum der Höhe 0 ergibt sich trivialerweise, daß keine Duplikate produziert werden, da dies einem Join zweier sortierter Folgen entspricht.

$\mathbf{n \rightarrow n + 1}$: Sei n eine ganze Zahl und nehmen wir an, daß die Mischbäume der Höhe j, $j \leq n$ keine Duplikate produzieren. Wir nehmen einen Mischbaum der Höhe $n + 1$ an. Die letzte Mischoperation wird durch einen Aufruf der Methode `earlyJoinMergedRuns` berechnet, wobei die Parameter der Methode die Folgen $(R_0, \ldots, R_m, S_0, \ldots S_m)$ sind, welche vom

Mischbaum der Höhe n erzeugt wurden. Die Menge der bereits berechneten Ergebnistupel ist gegeben durch

$$\bigcup_{i=1}^{m} R_i \bowtie S_i.$$

Genau diese Ergebnistupel werden nicht produziert, da sie in den Zeilen 16 und 23 aus der Antwortmenge entfernt werden. □

Im folgenden Abschnitt diskutieren wir die verschiedenen Möglichkeiten, die Eindeutigkeit von PMJ zu realisieren.

4.4 Realisierung der Eindeutigkeit

4.4.1 Duplikatvermeidung

Unsere derzeitige Implementierung von PMJ realisiert die Eindeutigkeit von PMJ (Abschnitt 4.3 bzw. Zeilen 16 und 23 in Abbildung 4.5) wie folgt: anstatt eine Sweep Area für jede Datenmenge zu nutzen, halten wir eine Sweep Area für jeden Eingabefolge im Hauptspeicher. D.h. es existieren bei einem Fanin der Mischoperation von F genau $2F$ Sweep Areas. Jedesmal wenn ein Element r der i-ten Eingabefolge zur Verarbeitung gewählt wird (Zeilen 5–13, Abbildung 4.5), werden Joinpartner nur von denjenigen Sweep Areas angefragt, die nicht aus der i-ten Eingabefolge stammen. Dieses Vorgehen erzeugt nur Ergebnistupel und keine Duplikate (die dann später wieder eliminiert werden müßten). Dieses Vorgehen hat auch den Vorteil, daß die Implementierung existierender Sweep Areas für die Nutzung von PMJ nicht modifiziert werden muß.

4.4.2 Duplikateliminierung

Eine andere Möglichkeit, die Eindeutigkeit von PMJ zu realisieren, besteht darin, wie beim ursprünglichen Sort-Merge Join nur eine Sweep Area pro Datenmenge zu verwenden. Die bei diesem Vorgehen erzeugten Duplikate müssen dann nachträglich erkannt und eliminiert werden. Hierfür ist es notwendig, für jedes Element, das in den Sweep Areas gespeichert wird, die ID der zugehörigen Eingabefolge zu speichern.

Sei o.B.d.A. $r \in R'_i$ das nächste zu verarbeitende Element. Durch die Anfrage von r an S_M (Zeile 15, Abbildung 4.5) ergibt sich eine Antwortmenge der Form $A = \{(r, u) \mid u \in S_M\}$. Wir benötigen eine Abbildung der Form

$$f_{ID}(u) = \text{„ID der Eingabefolge von } u \in S_M\text{“},$$

so daß mit Hilfe eines Prädikates

$$P(r,u) = (f_{ID}(u) \neq i)$$

alle Duplikate aus A gelöscht werden können. D.h. zur Antwortmenge RES werden nur die Elemente $\{(r,u) \mid P(r,u)\}$ hinzugefügt.

Die Funktion f_{ID} kann auf verschiedene Arten realisiert werden:

- Zum einen ist das **‚Anhängen' der jeweiligen ID** an die einzelnen Datenelemente denkbar. Allerdings muß hierfür u.U. der Datentyp von u erweitert werden bzw. Anpassungen der Implementierung der Sweep Area[3] vorgenommen werden.

- Eine andere Möglichkeit besteht darin, sich in einer **separaten Datenstruktur** (z.B. einer Hashmap) die IDs für jedes Element zu merken. Allerdings besteht hier das Problem, daß bei dieser Lösung ungeklärt ist, wie Elemente aus dieser Datenstruktur wieder gelöscht werden. Elemente müssen gelöscht werden, da sonst die Datenstruktur die Größe der Eingabeströme erreicht, was nicht akzeptabel ist.

  Selbstverständlich ist ein Löschen aus der Datenstruktur durch ein allgemeines Löschprädikat P_{rm} denkbar. Dieses erfordert aber Kosten in Höhe $O(n)$ für jeden Aufruf, gleiches gilt für das Joinprädikat P_{join}. Aus Effizienzgründen wird die Löschstrategie daher implizit in der `reorganize`-Methode der entsprechenden Sweep Area festgelegt und steht nicht explizit als Prädikat zur Verfügung. Diese Methode hat Zugriff auf interne Attribute der Sweep Area, was es ihr ermöglicht, das Löschprädikat effizient zu unterstützen. Der Rückgabetyp ist `void`. Dieses Vorgehen ist nur möglich, wenn die Implementierung jeder einzelnen Sweep Area geändert wird.

- **Lösung:** Für unseren Algorithmus PMJ wäre es ideal, wenn die Methode `reorganize` des allgemeinen `SweepArea`-Interfaces die Menge der gelöschten Elemente zurückgeben würde[4]. Diese Information steht innerhalb der Sweep Areas zur Verfügung und könnte nach außen gegeben werden[5]. In diesem Fall könnten nach jedem Aufruf von `reorganize` alle zurückgegebenen Elemente aus der Datenstruktur für die IDs gelöscht werden (im Falle einer Hashmap mit Aufwand $O(1)$ pro Element. D.h. der Gesamtaufwand bleibt asymptotisch gleich.).

[3] wir gehen in Kapitel 5.3 näher auf die Modellierung des Sort-Merge Join in XXL ein
[4] als `Cursor`
[5] Hierfür müßte in XXL das Interface `SortMergeJoin.SweepArea` entsprechend angepaßt werden.

4.5 Laufzeit

Im folgenden untersuchen wir die I/O-Kosten von PMJ. Die Kosten für das Lesen der Eingabedaten und das Schreiben der Ergebnistupel werden nicht berücksichtigt [Gra94b]. Wir nehmen an, daß die Daten in Seiten organisiert sind und sind interessiert an der Anzahl der Seitenzugriffe. Seien B und M die Anzahl der Elemente, die in eine Speicherseite bzw. in den Hauptspeicher passen. N bezeichne die Anzahl der Elemente in der Eingabemenge. Wir nehmen der Einfachheit halber an, daß M und N Vielfache von B sind. Um mindestens zwei Eingabeströme und einen Ausgabestrom zulassen zu können, nehmen wir an, daß $M \geq 6B$ gilt.

Wir nehmen zunächst an, daß die Größe der Sweep Area (M_{Sweep}) kleiner als $M - 6B$ ist. Dann gilt: $M - M_{\text{Sweep}}$ Elemente können für die Erzeugung der Runs und die Mischphasen des Externen Sortierens genutzt werden. Die I/O-Kosten von PMJ ergeben sich durch die Kosten der Mischphasen. Jede Eingabemenge kann hierbei den halben Hauptspeicher nutzen. Dann ergeben sich die I/O-Kosten von PMJ durch die folgende Formel:

$$\Theta(N/B \; log_{(M-M_{\text{Sweep}})/B} N/B).$$

Bleibt zu klären, was in dem Fall passiert, wenn die Sweep Area nicht in den Hauptspeicher paßt. Dieser tritt auf, wenn die Selektivität des Joins sehr hoch ist. Z.B. kann das für den Equi Join passieren, wenn ein Joinattribut fast M mal vorkommt — ein Problem, daß in unserem Algorithmus mindestens genausogut behandelt werden kann wie im konventionellen Ansatz.

Wir diskutieren diesen Fall zunächst für den Equi Join, Band Join und Temporal Join. Wenn wir PMJ derart modifizieren, daß ein kleiner aber konstanter Teil des Hauptspeichers größer B für die Sweep Area reserviert ist, ergeben sich die gesamten I/O-Kosten von PMJ wie folgt:

$$\Theta(N/B \; log_{M/B} N/B + Z/B),$$

wobei Z die Anzahl der Ergebnistupel des Joins bezeichnet. Für den Spatial Join erhalten wir dieselbe Formel, wenn wir Techniken wie *Distribution Sweeping* [APR+98] anwenden. Jedoch bemerken die Autoren von [APR+98], daß für realistische Datensätze die Sweep Area nie größer als $O(\sqrt{N})$ war. Hieraus schließen sie, daß die Sweep Area in fast allen Fällen in den Hauptspeicher einer modernen Workstation paßt. Eine ähnliche Beobachtung wurde für den Similarity Join in [DS01] gemacht, wo die Sweep Area in den Experimenten nur einen marginalen Anteil am Hauptspeicher einnimmt.

Die hier durchgeführte Analyse beschreibt die Gesamtlaufzeit der Algorithmen. Für online

Algorithmen ist eine Analyse der Anzahl der Ergebnistupel in Abhängigkeit der Laufzeit interessant. Dieses fundamentale Problem ist nicht Teil dieser Arbeit. Wir haben es in einer separaten Arbeit untersucht [DSTW02a]. Ferner wird in [DSTW02b] ein Schätzer für die Anzahl der berechneten Ergebnistupel vorgeschlagen.

4.6 Experimente

In diesem Abschnitt präsentieren wir Ergebnisse von Experimenten mit drei Typen von Joins (Equi Joins, Spatial Joins und Similarity Joins) jeweils bezogen auf drei verschiedene sortierbasierte Algorithmen.

Für jeden Jointyp nutzen wir *PMJ*, den strikten Sort-Merge Join (*striktes SMJ*) [BE77] und den semistrikten Sort-Merge Join (*semistriktes SMJ*) [NP85].

Alle Algorithmen wurden in Java 1.3 basierend auf XXL [BDS00, BBD+01] implementiert. Die Implementierungen sind generisch in dem Sinne, daß die verschiedenen Jointypen von einer einzigen Implementierung unterstützt werden. Dies wird durch ein generisches `SweepArea`-Interface erreicht (siehe Kapitel 5). Alle Algorithmen basieren ferner auf dem Iterator-Interface. Das nächste Element wird vom Nutzer mittels der Methode `next` angefordert. Thread-Wechsel sind hierfür nicht notwendig.

In unseren Experimenten messen wir sowohl die Gesamtzahl der Externspeicherzugriffe als auch die Gesamtlaufzeit. Wir zählen einen I/O für jedes Schreiben oder Lesen einer Speicherseite. Die Seitengröße ist 4 KB. Wir nehmen an, daß die Algorithmen Teil eines Operatorbaums sind. Deswegen werden die Kosten für das erste Lesen der Daten als auch für Weiterreichen der Ergebnistupel an den nächsten Operator — welche gleich sind für alle Algorithmen – nicht mitgezählt.

Um die Gesamtlaufzeit der Algorithmen zu ermitteln, wurde der Puffer des Betriebssystems ausgeschaltet durch das Nutzen von raw-Devices. Die Runs, die während des Sortierens erzeugt werden, werden in große, kontinuierliche Partitionen auf das raw-Device geschrieben. Der verfügbare Hauptspeicher war 10% der Größe der Eingabedaten.

In unseren Experimenten haben wir sowohl echte als auch generierte Datensätze genutzt. Tabelle 4.2 zeigt, bezogen auf den jeweiligen Join, die verwendeten Datensätze. Für den Equi Join haben wir gleichverteilte (iid) Daten verwendet wohingegen wir für die anderen Joins die Datensätze (ST, RR, STPT, RRPT) (TIGER-Daten [Bur96]) bzw. hochdimensionale Daten einer CAD-Applikation (CAD1, CAD2, CAD1_Shuf, CAD2_Shuf) genutzt haben. Diese Daten wurden bereits in zahlreichen anderen Experimenten verwendet.

4.6.1 Vergleich von Joinalgorithmen

In diesem Abschnitt präsentieren wir Ergebnisse eines experimentellen Vergleichs für PMJ, striktes SMJ und semistriktes SMJ. Wir berichten jeweils die Anzahl der gefundenen Er-

Jointyp	Datensatz	Beschreibung	Datentyp	#Punkte	dim	Größe [MB]
Equi Join	UNI1	gleichverteilt	Integer	2,000,000	1	8.0
	UNI2	gleichverteilt	Integer	2,000,000	1	8.0
Spatial Join	ST	Straßensegmente von L.A.	Rechtecke	131,461	2	4.5
	RR	Bahn, Hydro & Verwaltungsgrenzen von L.A.	Rechtecke	128,971	2	4.4
Similarity Join	STPT	linke untere Ecke von ST	Punkte	131,461	2	4.5
	RRPT	linke untere Ecke von RR	Punkte	128,971	2	4.4
	CAD1	Merkmalsvektoren von CAD-Teilen	Punkte	657,048	16	40.1
	CAD2	Merkmalsvektoren von CAD-Teilen	Punkte	655,125	16	40.0
	CAD1_Shuf	CAD1 gemischt	Punkte	657,048	16	40.1
	CAD2_Shuf	CAD2 gemischt	Punkte	655,125	16	40.0

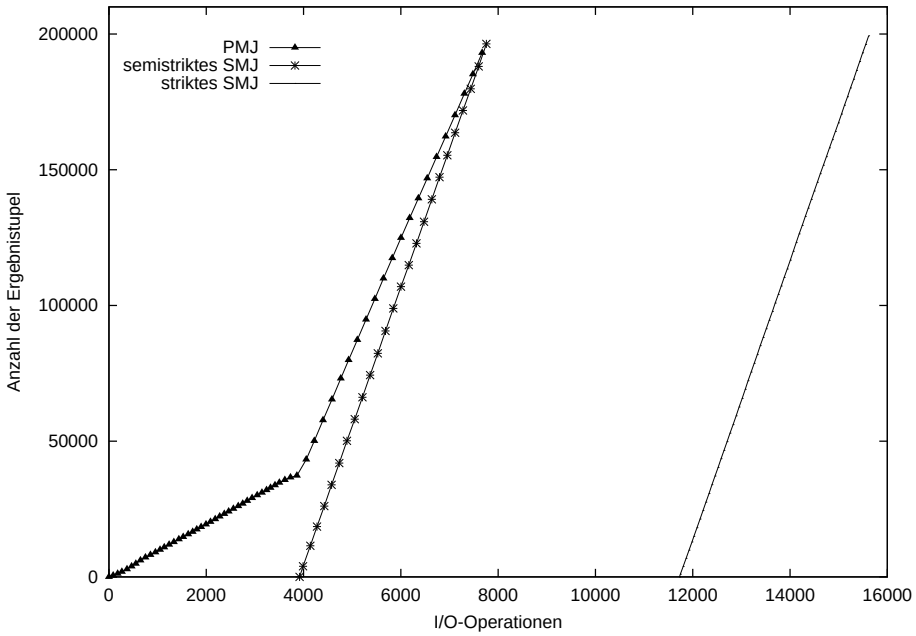
Tabelle 4.2: Die in den Experimenten verwendeten Datensätze

gebnistupel als eine Funktion der Anzahl der I/O-Operationen bzw. der Laufzeit.

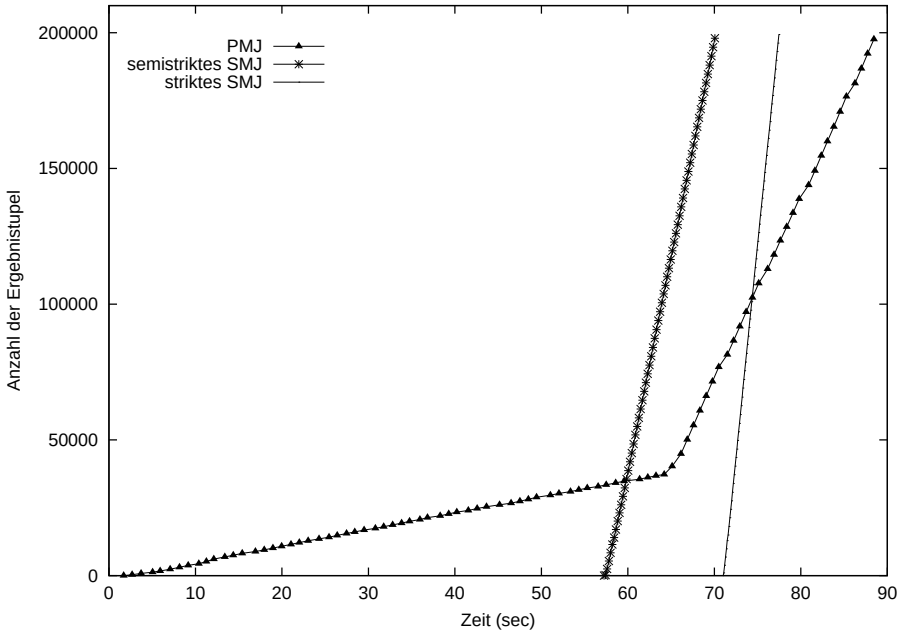
4.6.1.1 Equi Join

In unserem ersten Experiment berechnen wir den Equi Join von UNI1 und UNI2 (Tabelle 4.2). Jede der Datenmengen besteht aus 2,000,000 gleichverteilten Integer. Die Ergebnisse sind dargestellt in Abbildung 4.6. Die obere Abbildung zeigt die Anzahl der Ergebnistupel als eine Funktion der Anzahl der I/Os, die untere Abbildung als eine Funktion der Zeit. Striktes SMJ und semistriktes SMJ produzieren erste Ergebnistupel erst nach 11500 bzw. 4000 I/O-Operationen, wohingegen PMJ von Anfang an Ergebnistupel produziert. Es ist interessant, daß die Kurve von PMJ eine stückweise lineare Funktion ist. Das erste lineare Teilstück entspricht der Run-Erzeugungsphase wohingegen das zweite Teilstück der Merge-Phase entspricht.

Die Ergebnisse bezogen auf die Gesamtlaufzeit (siehe Abbildung 4.6b) sind ähnlich für striktes SMJ und semistriktes SMJ. Beide Algorithmen produzieren Ergebnistupel erst nach 58 bzw. 72 Sekunden. Danach produzieren beide Algorithmen eine große Zahl von Ergebnistupeln in sehr kurzer Zeit. Im Gegensatz hierzu liefert PMJ die ersten Ergebnistupel nach weniger als einer Sekunde. PMJ berechnet kontinuierlich Ergebnistupel bis zum Ende der Run-Erzeugungsphase. Die Anzahl der berechneten Ergebnistupel steigt dabei linear in Abhängigkeit der Laufzeit. In der Mischphase ist die Kurve wiederum linear, aber steiler



(a) I/O-Kosten



(b) Laufzeit (sec)

Abbildung 4.6: Equi Join UNI1 $\bowtie$ UNI2: Anzahl der Ergebnistupel als eine Funktion von I/O und Zeit

als die der Run-Erzeugung. Dies zeigt, daß pro Zeiteinheit mehr Ergebnistupel produziert werden.

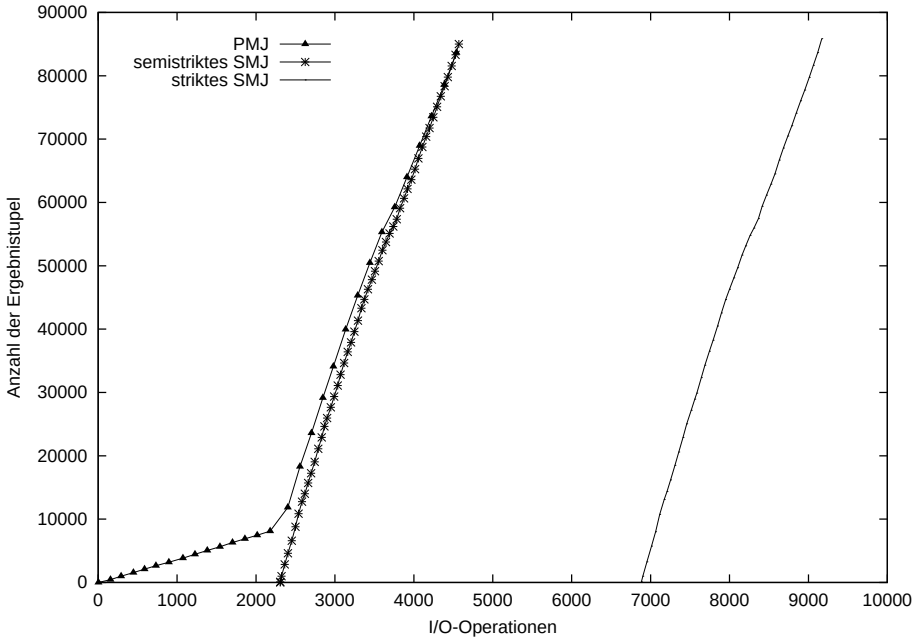
4.6.1.2 Spatial Join

Im folgenden diskutieren wir Ergebnisse des Spatial Joins der Datensätze ST und RR. Die Sweep Area ist hashbasiert wie in [APR+98] beschrieben. Trotz der Tatsache, daß das strikte SMJ in den Experimenten in [APR+98] genutzt wurde, war dieses Verfahren in [APR+98] bereits effizienter als PBSM [PD96].

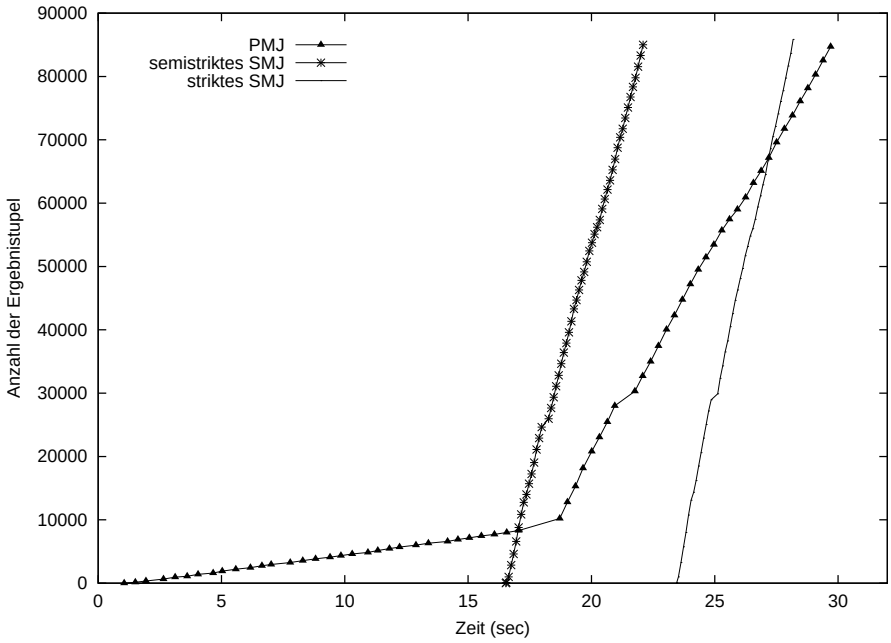
Die Kurven in Abbildung 4.7 zeigen, daß PMJ von Beginn an eine beträchtliche Anzahl von Ergebnistupeln produziert. Bevor das semistrikte SMJ das erste Tupel findet, hat PMJ bereits 8000 Tupel gefunden. Die Gesamtzahl der I/O-Operationen ist fast identisch für PMJ und das semistrikte SMJ, wohingegen das strikte SMJ doppelt soviele I/O-Operationen benötigt. Abbildung 4.7b zeigt die Anzahl der Ergebnistupel als eine Funktion der Laufzeit. PMJ berichtet wieder von Beginn an Ergebnistupel wohingegen die Konkurrenzverfahren erst nach 16 bzw. 23 Sekunden das erste Tupel finden. Die Anzahl der pro Zeiteinheit produzierten Tupel ist allerdings höher für letztere Verfahren. Dies liegt vermutlich an dem zusätzlichen Aufwand für die Duplikatvermeidung bei PMJ, welche pro Element mehrere Anfragen auf Sweep Areas erfordert. Insgesamt ist die Gesamtlaufzeit nur um ca. 1/3 höher als die des semistrikten SMJ.

4.6.1.3 Similarity Join

Eine der wichtigsten Anwendungen für nichtblockierende Joinalgorithmen liegt im Bereich des Data Mining, wo Nutzerinteraktion unverzichtbar ist. Dies kann nur dann zufriedenstellend unterstützt werden, wenn Similarity Joins und andere Grundoperationen das frühe Produzieren von Ergebnistupeln unterstützen. Im folgenden präsentieren wir Ergebnisse für den in Kapitel 3 vorgestellten Similarity Join GESS. Abbildung 4.8 zeigt das Ergebnis für den Similarity Join der Datensätze STPT und RRPT. Die Kurven sind nicht linear wie im letzten Experiment aber vom Trend her ähnlich. Bezüglich I/O-Kosten haben PMJ und semistriktes SMJ die gleichen Kosten. Im Gegensatz zu beiden Versionen von SMJ berichtet PMJ von Beginn an wiederum zahlreiche Ergebnistupel (einige Tausend). Die Gesamtlaufzeit von PMJ ist nur geringfügig höher als die des semistrikten SMJ und geringer als die des strikten SMJ. Zu dem Zeitpunkt, an dem das semistrikte SMJ das erste Ergebnistupel berichtet, hat PMJ bereits 10% der Ergebnismenge berechnet.

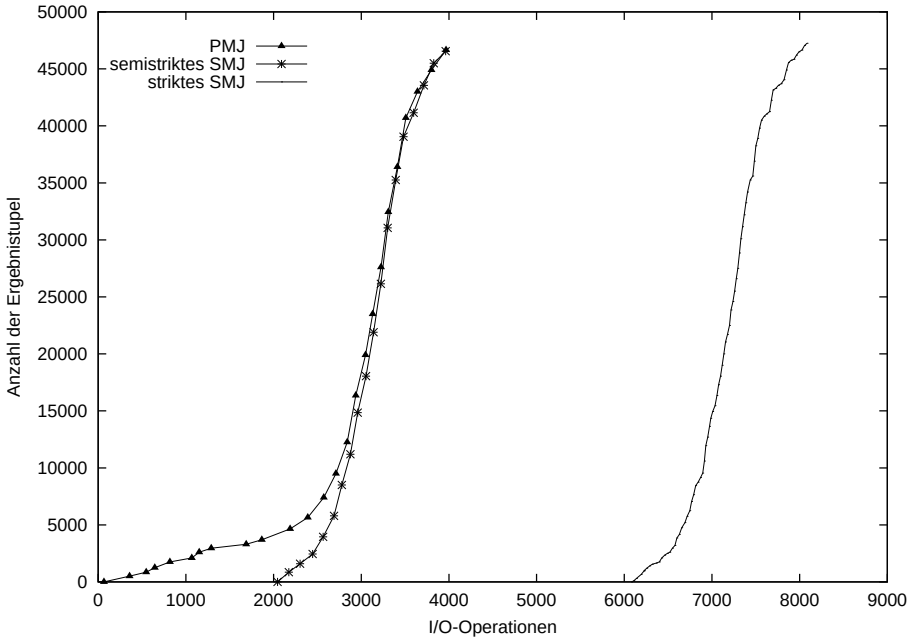


(a) I/O-Kosten

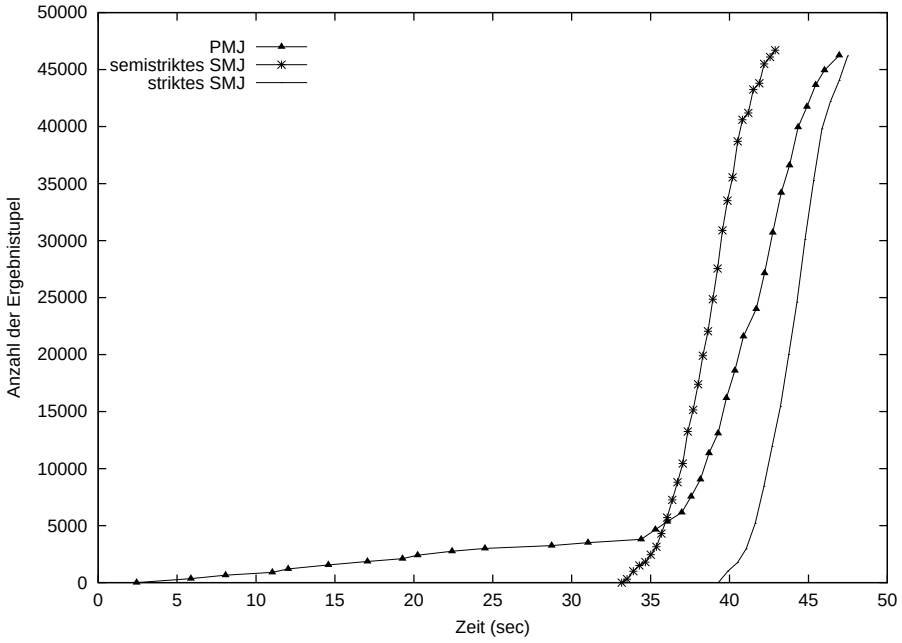


(b) Laufzeit (sec)

Abbildung 4.7: Spatial Join ST $\bowtie$ RR: Anzahl der Ergebnistupel als eine Funktion von I/O und Zeit

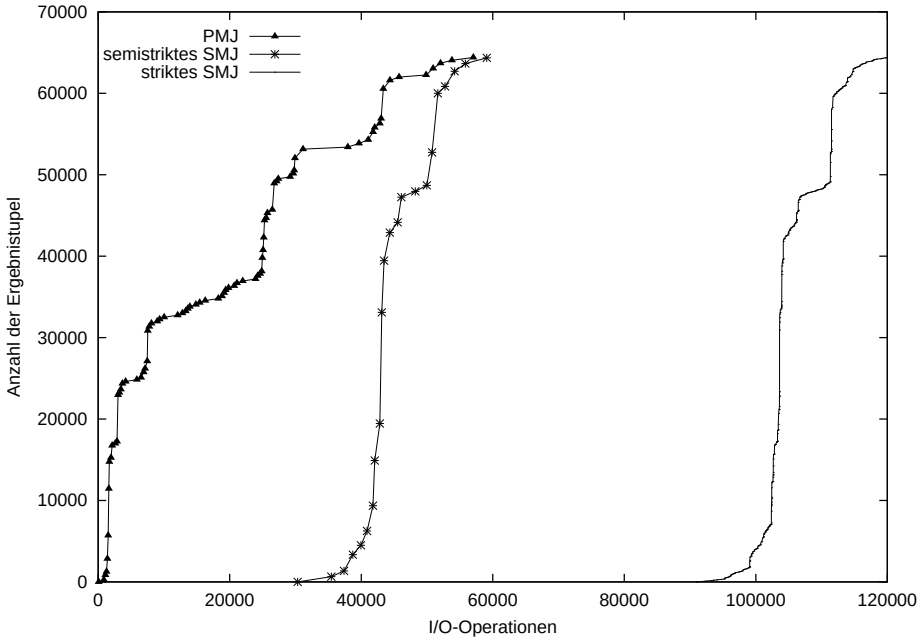


(a) I/O-Kosten

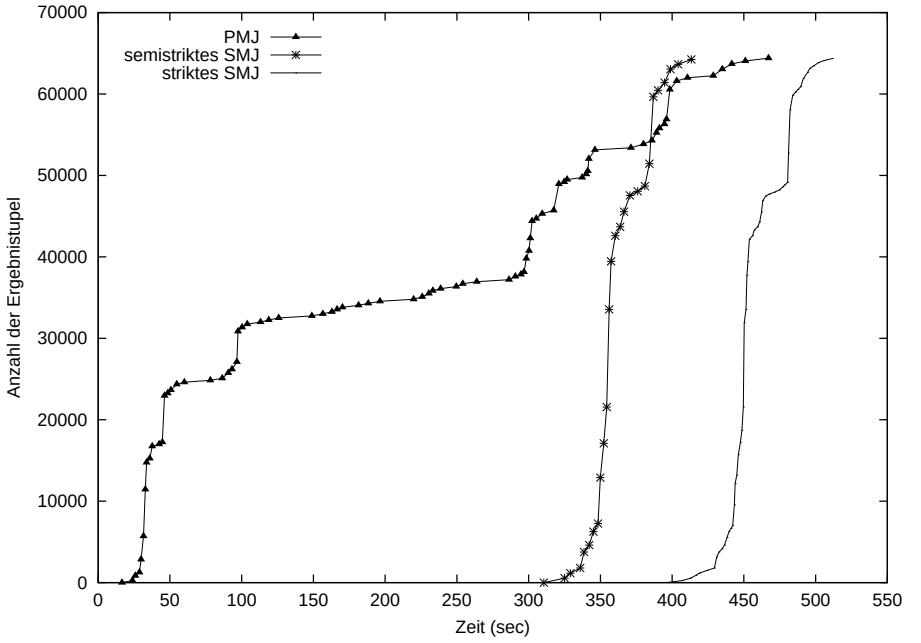


(b) Laufzeit (sec)

Abbildung 4.8: Similarity Join STPT $\bowtie_\varepsilon$ RRPT: Anzahl der Ergebnistupel als eine Funktion von I/O und Zeit



(a) I/O-Kosten



(b) Laufzeit (sec)

Abbildung 4.9: Similarity Join CAD1 $\bowtie_\varepsilon$ CAD2: Anzahl der Ergebnistupel als eine Funktion von I/O und Zeit

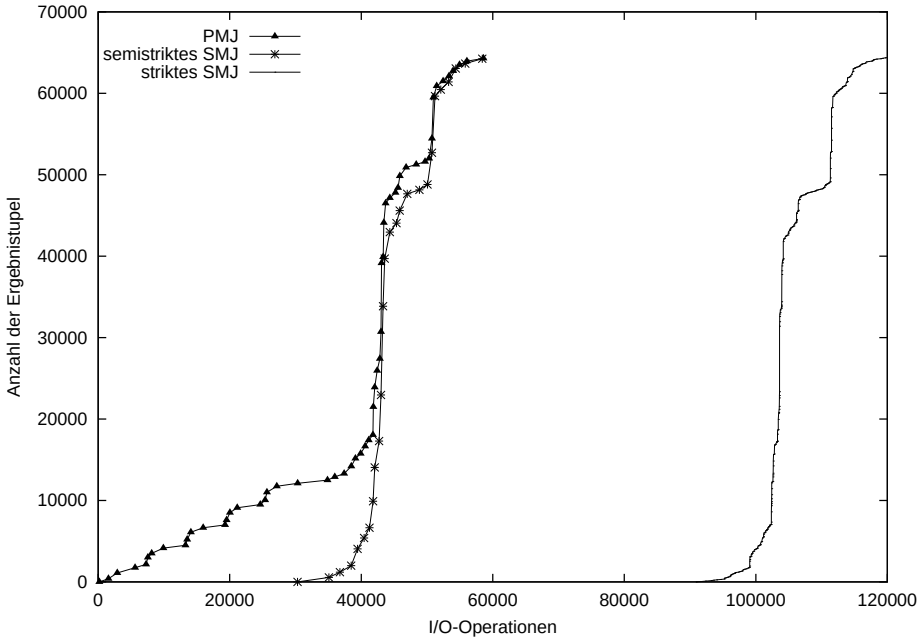
Die nächsten Kurven in Abbildung 4.9 zeigen Ergebnisse eines Similarity Joins der 16-dimensionalen Datensätze CAD1 und CAD2. Diese Datensätze wurden durch Aufteilung eines Datensatzes erzeugt. Dies erfolgte so, daß ein Tupel mit einer Wahrscheinlichkeit von 0.5 zu Datensatz CAD1 bzw. CAD2 zugeordnet wurde. Aufgrund von Cluster-Effekten der ursprünglichen Datensätze sind die Datensätze CAD1 und CAD2 stark korreliert. Dies ist der Grund dafür, daß die Gesamtlaufzeit von PMJ nur wenig höher ist als die von SMJ und außergewöhnlich viele Ergebnistupel während der Run-Erzeugung produziert werden (Abbildung 4.9). Nach 120 Sekunden sind bereits mehr als 50% der Ergebnismenge berichtet, wohingegen das semistrikte SMJ das erste Tupel erst nach 310 Sekunden findet. PMJ benötigt keinen zusätzlichen I/O-Aufwand für dieses Experiment.

In einem zweiten Experiment haben wir den Similarity Join für die gemischten Datensätze CAD1_Shuf und CAD2_Shuf berechnet. Die Ergebnisse sind in Abbildung 4.10 zu sehen. Die Ergebnisse für das strikte und semistrikte SMJ gleichen denen des letzten Experiments. Für PMJ hingegen unterscheiden sich die Kurven. Für die durchmischten Daten wird ein geringerer Anteil an Ergebnistupeln während der Run-Erzeugung produziert. Im Vergleich zu den Experimenten mit Equi und Spatial Joins ist die Anzahl der in der Mischphase produzierten Ergebnistupel allerdings deutlich höher. Die Gesamtlaufzeit von PMJ entspricht der des letzten Experiments.

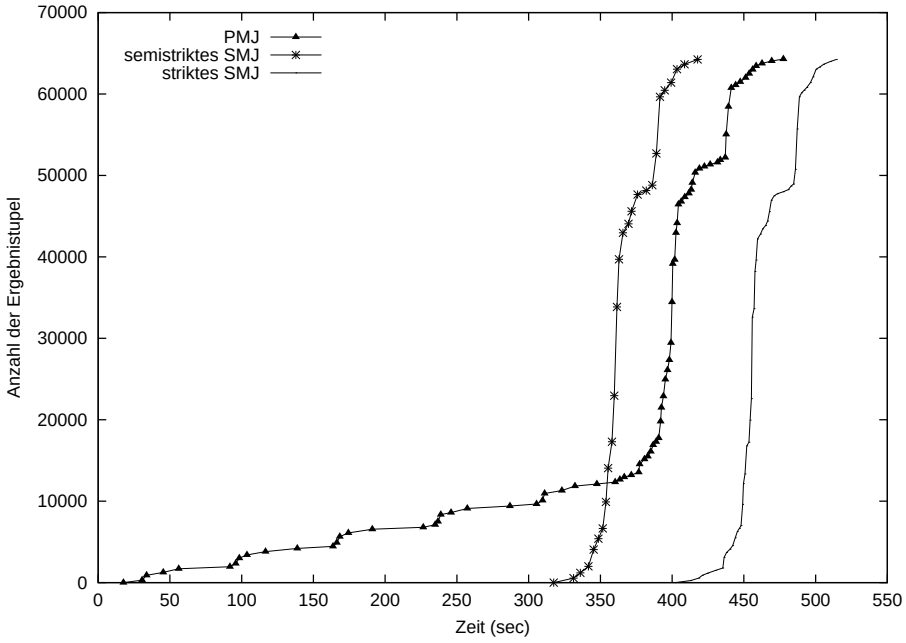
4.6.2 Die Auswirkung der Hauptspeichergröße

In diesem Experiment haben wir den verfügbaren Hauptspeicher variiert. Abbildung 4.11 zeigt die Zeit die benötigt wird, die ersten 100 Ergebnistupel zu berichten für den Spatial Join von RR und ST. Die Größe des Hauptspeichers ist normiert bezüglich der Größe der Eingabemengen. Für alle Hauptspeichergrößen findet berichtet PMJ die ersten 100 Ergebnistupel mindestens eine Größenordnung früher als das semistrikte SMJ. Für kleinere Hauptspeichergrößen ($< 5\%$) erhöht sich dieser Vorteil auf zwei Größenordnungen und mehr.

Tabelle 4.3 zeigt die erforderlich Zeit bzw. Anzahl der I/O-Operationen, um das 1., 100. und 1000. Ergebnistupel eines Spatial Joins von RR und ST zu berichten. Tabelle 4.3 zeigt die I/O-Operationen bzw. die Laufzeit, die benötigt wird für Hauptspeichergrößen von 5%, 10% und 20%. Für eine Hauptspeichergröße von 10% benötigt das strikte SMJ 30.3 bzw. 30.4 Sekunden, um das erste bzw 1000. Tupel zu berichten. Im Gegensatz hierzu benötigt PMJ nur 0.7 bzw. 5.1 Sekunden für das erste bzw. 1000. Tupel. Für größere Hauptspeichergrößen nimmt die Laufzeit leicht zu aufgrund der Tatsache, daß der Heap für Replacement Selection

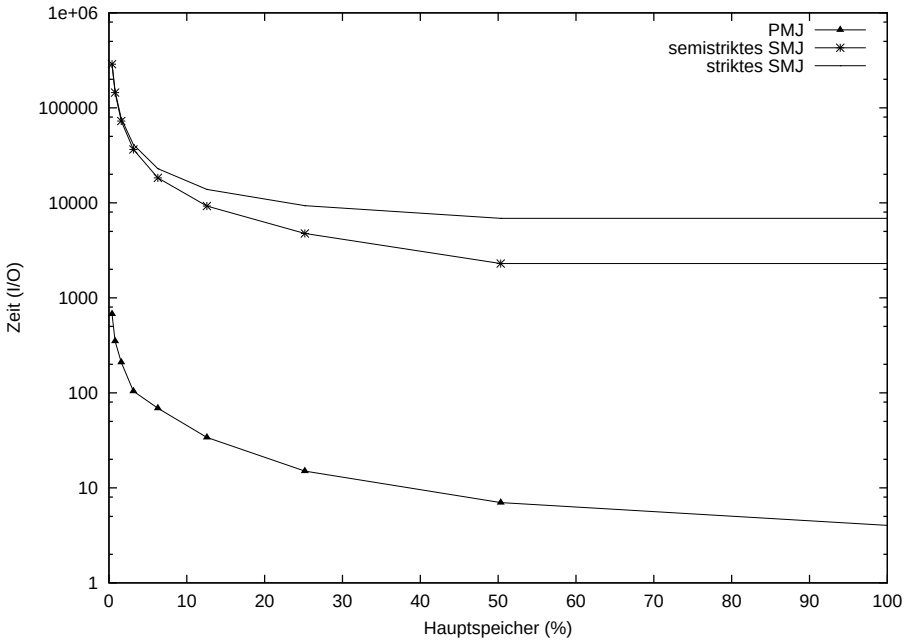


(a) I/O-Kosten

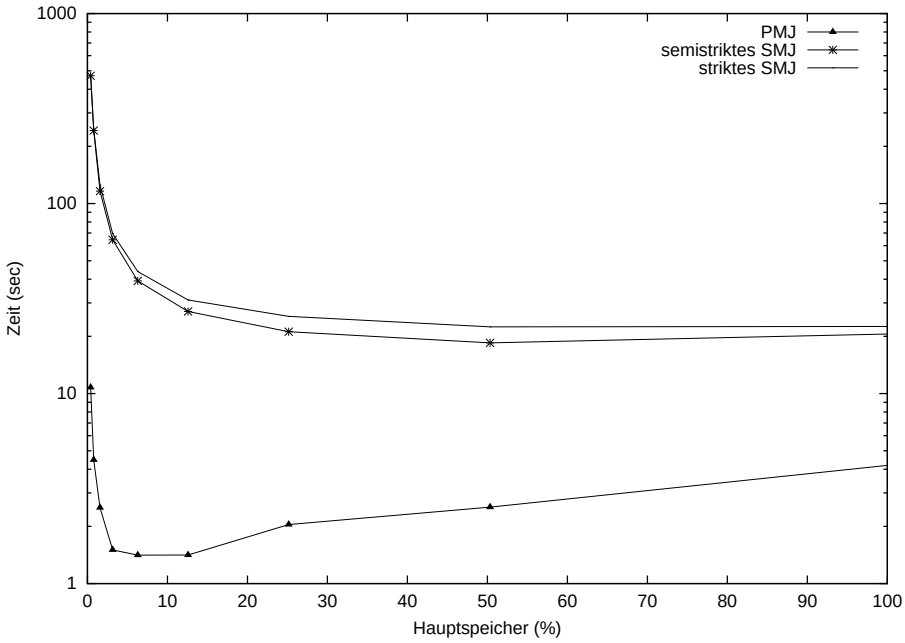


(b) Laufzeit (sec)

Abbildung 4.10: Similarity Join CAD1_Shuf $\bowtie_\varepsilon$ CAD2_Shuf: Anzahl der Ergebnistupel als eine Funktion von I/O und Zeit



(a) I/O-Kosten



(b) Laufzeit (sec)

Abbildung 4.11: Zeit, die benötigt wird, um die ersten 100 Ergebnistupel zu berechnen, Plane Sweep ST $\bowtie$ RR

Algorithmus	verfügbarer HS	1. Tupel		100. Tupel		1000. Tupel	
		I/O	Zeit (sec)	I/O	Zeit (sec)	I/O	Zeit (sec)
striktes SMJ	5%	10878	30.3	10880	30.3	10898	30.4
	10%	6878	22.0	6880	22.0	6898	22.1
	20%	6876	22.3	6878	22.3	6896	22.4
semistriktes SMJ	5%	6302	22.2	6303	22.2	6319	22.3
	10%	2305	16.0	2305	16.0	2321	16.1
	20%	2300	17.2	2301	17.2	2318	17.3
PMJ	5%	1	0.7	80	1.4	568	5.1
	10%	3	0.9	42	1.4	311	3.3
	20%	0	1.3	21	1.6	185	3.0

Tabelle 4.3: Benötigte Laufzeit bzw. I/O-Operationen für einen Spatial Join von RR und ST (Hauptspeichergröße: 5%, 10%, 20%)

größer ist und deshalb am Anfang höhere Kosten verursacht.

4.6.3 Selektivitätsschätzung

In diesem Abschnitt präsentieren wir Ergebnisse von Experimenten mit dem in [DSTW02b] vorgeschlagenen Selektivitätsschätzer.

Abbildung 4.12 zeigt die Anzahl der berechneten Ergebnistupel in Abhängigkeit der Zeit für einen Similarity Join der künstlich gemischten Datensätze CAD1_Shuf und CAD2_Shuf. Um eine hohe Zahl von Schätzungen zu bekommen, ist der verfügbare Hauptspeicher in dem Experiment auf 1% gesetzt. Man beachte, daß das der Grund für die höhere Laufzeit im Vergleich zum Experiment aus Abbildung 4.10 verursacht, wo der verfügbare Hauptspeicher um Faktor 10 größer war. Abbildung 4.12 zeigt die erwartete Anzahl der Ergebnistupel sowie das zugehörige Konfidenzintervall (95% Wahrscheinlichkeit) in Abhängigkeit der Laufzeit. Die grüne Kurve markiert die von PMJ berechneten Ergebnisse in Abhängigkeit der Laufzeit. Die blaue Linie markiert die tatsächliche Anzahl von Ergebnistupeln.

Bereits nach wenigen Sekunden ist der geschätzte Wert nahe an der Gesamtzahl der Ergebnistupel. Die Güte des Schätzers wird vom Konfidenzintervall bestätigt.

Abbildung 4.13 zeigt dasselbe Experiment für die nicht künstlich vermischten Datensätze CAD1 und CAD2. Der Schätzwert liegt in diesem Experiment um 1.5 bis 2 Größenordnungen über der tatsächlichen Anzahl der Ergebnistupel. Entsprechendes gilt für das zugehörige

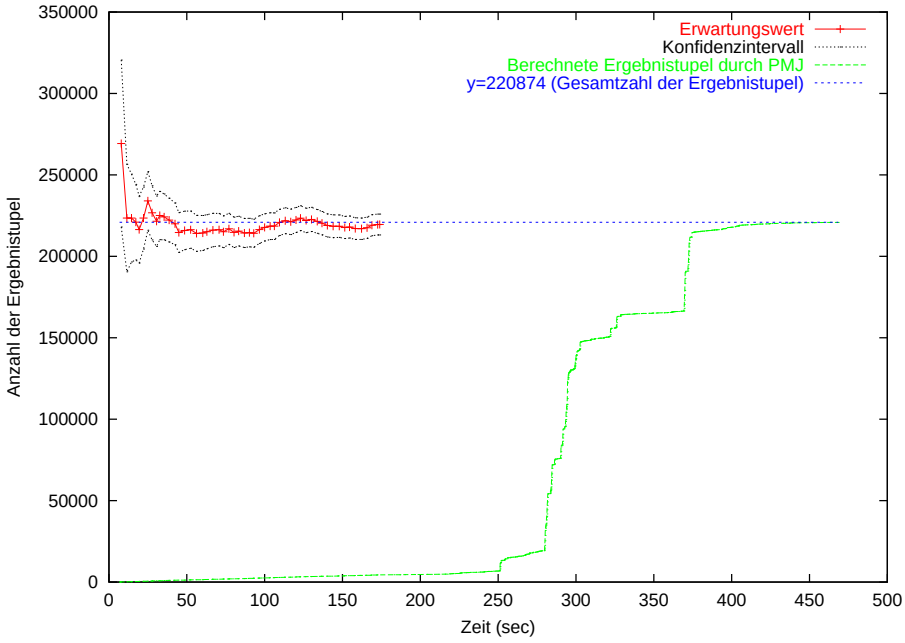


Abbildung 4.12: Selektivitätsschätzung für den Similarity Join CAD1_Shuf $\bowtie_\varepsilon$ CAD2_Shuf (Hauptspeichergröße 1%)

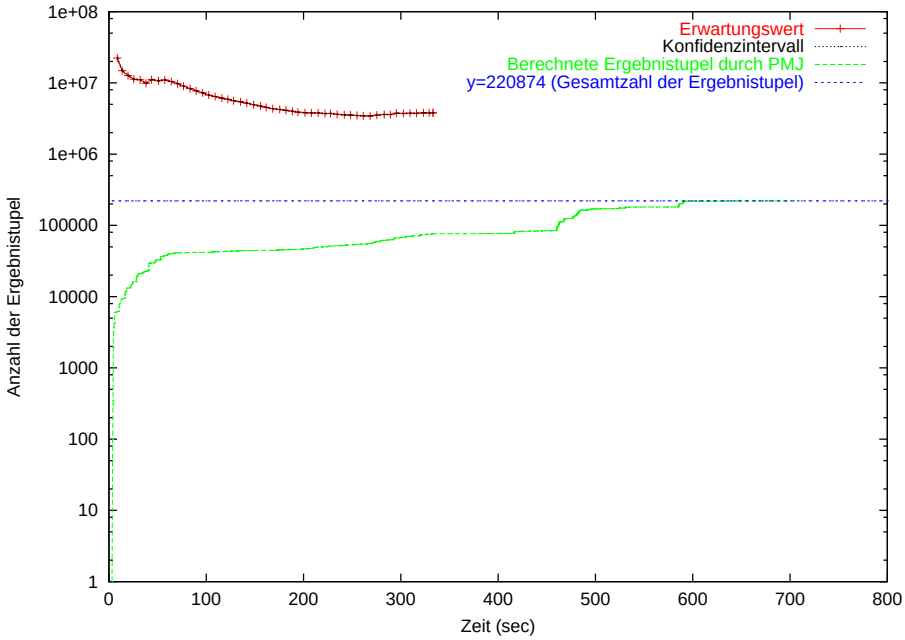


Abbildung 4.13: Selektivitätsschätzung für den Similarity Join CAD1 $\bowtie_\varepsilon$ CAD2 (Hauptspeichergröße 1%, log-Skala auf y-Achse)

Konfidenzintervall.

Dieses Ergebnis läßt sich dadurch erklären, daß der in [DSTW02b] vorgeschlagene Schätzer die Annahme macht, daß die bereits verarbeiteten Daten eine Sichprobe der gesamten Datenmenge sind. D.h., es wird postuliert, daß die Verteilung der Stichprobe, die Verteilung der gesamten Datenmenge repräsentiert. Diese Annahme ist in dem Experiment für die Datensätze CAD1 und CAD2 — wie wir auch bereits in den Abbildungen 4.9&4.10 gesehen haben — offensichtlich verletzt.

Die Ergebnisse des Schätzers für reale und künstlich vermischte Datensätze differieren in dem Experimenten in hohem Maße.

In Abschnitt 3.7 haben wir bereits gesehen, daß die Gleichverteilungsannahme bezüglich der Unabhängigkeit der einzelnen Dimensionen für reale Datensätze nicht zutrifft. Das vorliegende Experiment zeigt eine zusätzliche Abhängigkeit von der Ankunftsreihenfolge der Daten. Diese Ankunftsreihenfolge kann in einer realen Data Mining Applikationen nicht garantiert werden kann. Dies ist ein prinzipielles Problem von online Schätzverfahren. Es läßt sich schließen, daß entsprechende Schätzverfahren im Kontext von Data Mining Applikationen mit einer gewissen Vorsicht zu betrachten sind.

4.7 Zusammenfassung

In diesem Kapitel haben wir uns mit dem Problem beschäftigt, effiziente, nichtblockierende Joinalgorithmen zu entwickeln, die früh und fortlaufend Ergebnistupel produzieren. Unser primäres Ziel hierbei war es, solche Methoden für fortgeschrittene Jointechniken wie den Spatial- und Similarity Join zu präsentieren. Da viele moderne Joinverfahren auf dem Sort-Merge Paradigma basieren, haben wir uns auf sortierbasierte Verfahren konzentriert. Inbesondere haben wir eine neue, nichtblockierende Jointechnik namens *Progressive Merge Join (PMJ)* vorgeschlagen. PMJ läßt sich immer dann anwenden, wenn ein korrespondierendes blockierendes Verfahren existiert.

Nach unsere Einschätzung ist PMJ das erste nichtblockierende Verfahren für den Similarity Join, ein Jointyp von großer Bedeutung für Data Mining-Anwendungen. PMJ ist insbesondere sehr nützlich für das interaktive Erkunden großer Datenmengen in Situationen, wo der Nutzer nicht bereit ist, minutenlang zu warten, bis das erste Ergebnistupel produziert wurde.

In diesem Kapitel haben wir den generischen PMJ-Algorithmus vorgestellt. Die wesentliche

Idee von PMJ ist es, erste Ergebnisse des Joins bereits während des Sortierens zu produzieren. Die Sortierphase wird hierbei für beide Datenmengen gleichzeitig ausgeführt. Wir haben formal bewiesen, daß der generische PMJ-Algorithmus korrekt ist. Ferner haben wir gezeigt, wie verschiedene Jointypen durch spezialisierte Versionen von PMJ unterstützt werden. Hierzu gehören Equi Joins, Temporal Joins, Band Joins, Spatial Joins und Similarity Joins. Alle diese Varianten sind nichtblockierend und produzieren früh Ergebnistupel.

Desweiteren haben wir die Gesamtlaufzeit von PMJ untersucht und gezeigt, daß diese asymptotisch derjenigen der korrespondierenden blockierenden Algorithmen entspricht.

Schließlich habe wir verschiedene Joinvarianten in einer Serie von Experimenten mit verschiedenen Datensätzen gegenübergestellt. Wir haben gezeigt, daß PMJ bei geringfügig höherer Laufzeit Ergebnistupel wesentlich früher produziert als die korrespondierenden blockierenden Algorithmen.

In einer separaten Studie [DSTW02a] haben wir ein theoretisches Modell entwickelt, daß die Anzahl der Ergebnistupel in Abhängigkeit der Laufzeit und der verwendeten Traversierungsstrategie vorhersagt.

Grundsätzlich kann PMJ durch alle Techniken, die auch für das Externe Sortieren durch Mischen verwendet werden können, optimiert werden.

Kapitel 5

XXL

In diesem Kapitel geben wir eine kurze Übersicht über XXL (eXtensible and fleXible Library) und beschreiben die Integration der in dieser Arbeit vorgestellten Algorithmen. XXL ist eine plattformunabhängige Javabibliothek von Frameworks und Algorithmen zur Anfrageverarbeitung. XXL wird in der AG Datenbanksysteme der Universität Marburg unter Leitung von Prof. Dr. Bernhard Seeger entwickelt.

Alle in dieser Arbeit vorgestellten Verfahren wurden basierend auf XXL entwickelt und sind in die Bibliothek integriert. Alle Algorithmen sind im aktuellen Release[1] von XXL enthalten und stehen zum Download auf der XXL-Homepage [XXL] bereit[2]. XXL wird unter der *Lesser Gnu Public License* [GNU] vertrieben.

Wir diskutieren im folgenden zunächst die Designprinzipien, die bei der Erstellung von XXL ausschlaggebend waren. Danach erläutern wir die wichtigsten Schnittstellen und Konzepte von XXL. Im Anschluß beschreiben wir die Integration der Joinalgorithmen in XXL.

Für eine detaillierte Beschreibung von XXL siehe auch [BDS00, BBD+01].

5.1 Designprinzipien

Beim Design von XXL wurde auf folgende Aspekte besonders Wert gelegt:

1. **Physische Datenunabhängigkeit:** Alle Algorithmen arbeiten auf Schnittstellen, hierdurch wird ein beliebiger Austausch von z.B. Hauptspeicher-Queues mit externen Queues möglich. Beispielsweise arbeitet das ‚Externe Sortieren durch Mischen' [Knu73]

[1] Zum Zeitpunkt der Erstellung dieser Arbeit: 0.99b Build 2002-02-13
[2] PMJ erscheint im nächsten offiziellen Release

(`MergeSorter`) auf einem `Queue`-Interface. Der Nutzer muß dem Algorithmus mitteilen, ob das Sortierverfahren extern oder im Hauptspeicher ablaufen soll. Dies erfolgt durch Übergabe einer *Factory-Method* [GHJV94].

2. **Plattformunabhängigkeit:** Das Projekt wurde vollständig in Java realisiert (SDK 1.3). Java-Interpreter (inklusive Just-in-Time bzw. Hot-Spot-Compilern) sind heute auf fast allen Architekturen vorhanden. Insbesondere die Browser-Fähigkeit von Java macht XXL für entsprechende Anwendungen interessant. Ein mögliches Anwendungsgebiet liegt auch in der Nutzung von XXL für Java Server Pages (JSP) bzw. Enterprise Java Beans. Ferner bieten viele Datenbanksysteme eine exzellente Java-Anbindung. So wurde in [BBD+01] gezeigt, wie im Rahmen der vorliegenden Arbeit entwickelten Joinalgorithmen für den räumlichen Verbund aus XXL in einem kommerziellen Datenbanksystem (Cloudscape) genutzt werden können.

3. **Demand-Driven:** Alle Operatoren sollen das *open-next-close (ONC)* [Gra94b] Interface erfüllen. (siehe unten).

4. **Sauberes Design:** Es wurde großen Wert gelegt auf die ‚korrekte' Modellierung der Algorithmen. Insbesondere wurde intensiv Gebrauch gemacht von Design Patterns [GHJV94], um doppelten Code zu vermeiden.

   Im allgemeinen ist es schwierig, ein generisches Verfahren direkt so zu entwerfen, so daß es für alle Anwendungsfälle sofort geeignet ist. Die Erstellung generischer Frameworks scheint mit den traditionellen Ansätzen des Software Engineerings (wie z.B. dem Wasserfall- [Roy70] oder Spiralmodell [Boe88]) nicht möglich. Generische Verfahren kristallisieren sich in einem hochevolutionären Prozeß heraus. Meist ergeben sich erst aus der Spezialisierung eines Algorithmus neue Anregungen und Ideen für den allgemeinen Ansatz. Die generische Funktionalität wird nach und nach aus den Spezialfällen ‚herausgezogen'. Hierdurch hat sich im Laufe der Zeit u.a. die sehr mächtige Cursoralgebra von XXL entwickelt.

5. **Unterstützung für funktionale Programmierung in Java:**

   Java bietet zwar die Möglichkeit, die Ordnung auf einem Datentyp mittels `java.util.Comparator` zu definieren, ein allgemeiner Funktionstyp existiert in Java aber leider nicht. Aus diesem Grund wurden in XXL die abstrakten Klassen `Function` und `Predicate` eingeführt. Die packages `xxl.functions` und `xxl.predicates` enthalten zudem zahlreiche wichtige Spezialisierungen wie z.B. `And`, `Or` und `Not`.

6. **Generizität:** Alle Algorithmen werden nach Möglichkeit so entworfen, daß sie nicht an einen bestimmten Typ gebunden sind. Die Typisierung erfolgt erst in Parameter-

funktionen und Prädikaten sowie den Comparatoren. D.h. die eigentliche Funktionalität des Algorithmus wird vom konkreten Datentyp getrennt.

7. **Wiederverwendung von Code:**

   Die Erstellung spezialisierter Verfahren erfolgt i.d.R. durch Erweiterung der allgemeinen Algorithmen `xxl.cursors`. Dies trifft z.B. auf das package `xxl.relationalNEU`[3] zu. Alle relationalen Operatoren, wie z.B. `xxl.relationalNeu.Selection` oder `xxl.relationalNeu.SortMergeJoin` erben von den entsprechenden generischen Verfahren `xxl.cursors.Filter` bzw. `xxl.cursors.SortMergeJoin`. Selbiges trifft auf die in dieser Arbeit vorgestellten Similarity Joins zu. Wir kommen weiter unten hierauf zurück.

8. **Dokumentation:** Die Klassen werden mit `javadoc` dokumentiert. Die aktuelle API-Dokumentation findet sich auf der XXL-Homepage [XXL].

5.2 Aufbau der Bibliothek

In diesem Abschnitt erläutern wir den prinzipiellen Aufbau von XXL und beschreiben die wichtigsten Schnittstellen.

Die Bibliothek ist in sogenannte *packages* (Pakete) gegliedert. Die wichtigsten Pakete sind `xxl.functions`, `xxl.predicates`, `xxl.cursors` sowie `xxl.collections` und `xxl.io`. In diesen Paketen wird eine Vielzahl von Schnittstellen und abstrakten Vorimplementierungen sowie Toolboxes bereitgestellt.

Desweiteren stehen spezialisierte Pakete zur Verfügung. Das Paket `xxl.indexStructures` enthält wichtige Indexstrukturen wie B-, R*-, M- und X-Baum. `xxl.binarySearchTrees` stellt balancierte Binärbäume bereit. `xxl.spatial` enthält Bausteine und Algorithmen sowohl für Spatial- als auch Similarity Joins. `xxl.statistics` beinhaltet statistische Methoden für Datenbanksysteme. In `xxl.relationalNEU` werden Operatoren der relationalen Algebra bereitgestellt. Schließlich enthält `xxl.util` verschiedene Hilfsklassen, die keiner anderen Kategorie zufallen.

[3] Das package `xxl.relationalNEU` soll das alte Paket `xxl.relational` in einer zukünftigen Version von XXL ersetzen.

5.2.1 Iteratoren und Cursor

Von fundamentaler Bedeutung für das Verständnis von XXL ist das *Iterator-Design Pattern*. In der Datenbankliteratur wird auch von *ONC*-Schnittstelle[4] gesprochen [Gra94b]. Ein Iterator liefert eine Folge von Objekten. Die zugrundeliegende Datenquelle wird hierbei nicht preisgegeben. Das Iterator-Interface von Java besteht aus 3 Methoden:

```
interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

Die Methode `hasNext()` ermittelt, ob der Iterator noch weitere Datenelemente liefern kann. `next()` liefert das nächste Datenelement des Iterators zurück.

Die `remove()`-Methode ist eine sogenannte *optionale Methode*, die nicht implementiert werden muß. Ist sie implementiert, löscht sie dasjenige Element, das als letztes vom Iterator geliefert wurde.

In XXL wurde das Iterator-Interface in zwei Schritten um 5 Methoden erweitert:

```
interface PeekIterator extends Iterator {
    Object peek();
    boolean supportsPeek();
}


interface Cursor extends PeekIterator {
    void update(Object o);
    void reset();
    void close();
}
```

Die `peek()`-Methode zeigt das nächste Element, ohne es dem Iterator zu entnehmen. `supportsPeek()` ermittelt, ob eine konkrete Implementierung des `PeekIterator` letztere Methode implementiert.

`update()` ersetzt das aktuell von `peek()` oder `next()` gelieferte Objekt mit dem übergebenen Objekt. Die `reset()`-Methode setzt den Cursor auf den Anfangszustand zurück.

[4] ONC steht für die drei Methodennamen `open()`, `next()` und `close()`.

Wichtig für den Cursor ist die Methode `close()`, welche es dem Cursor ermöglicht, allokierte Ressourcen (z.B. temporäre Dateien) freizugeben.

Man beachte, daß es keine `open()`-Methode gibt. Die Open-Phase [Gra94b] wird implizit im Konstruktor des Cursors ausgeführt.

Das Paket `xxl.cursors` enthält eine vollständige Cursoralgebra mit einer Vielzahl wichtiger Grundoperationen wie z.B. *nest-* und *unnest-* sowie Sortieroperationen. Ein Cursor hat grundsätzlich 0, 1 oder 2 Eingabe-Cursor, verarbeitet diese Eingaben und gibt die Ausgaben über die `next()`-Methode weiter. Somit können die verschiedenen Cursor aus XXL leicht zu komplexeren Anfragebäumen (=Operatorbäumen) zusammengesetzt werden. Hierbei ist zu betonen, daß die Cursor in `xxl.cursors` nicht auf einen bestimmten Datentyp festgelegt sind. Die Anpassung an konkrete Daten erfolgt erst durch die Übergabe spezialisierter Funktionsargumente (vom Typ `Function`, `Predicate` oder `Comparator`). Ein ähnlicher Ansatz wurde in [Gra94b] genutzt (*support functions*). In der funktionalen Programmierung [Hug89] kann eine komplexe Anfrage grundsätzlich als eine Funktion höherer Ordnung betrachtet werden, die dynamisch zur Laufzeit erzeugt wird.

5.2.2 Funktionen und Prädikate

Funktionsparameter ergänzen Java sowohl um funktionale Typen als auch um Funktionen höherer Ordnung. In XXL enthält die abstrakte Klasse `Function` die folgenden `invoke()`-Methoden:

```
abstract class Function {
    Object invoke();
    Object invoke(Object[] args);
    Object invoke(Object argument);
    Object invoke(Object argument1, Object argument2);
}
```

D.h. es werden `invoke()`-Methoden für 0-, 1-, 2- und n-stellige Funktionen bereitgestellt. Ferner stehen `compose()`-Methoden zur Erzeugung von Funktionen höherer Ordnung zur Verfügung. Die verschiedenen `invoke()`-Methoden rufen sich in der abstrakten Klasse `Function` rekursiv gegenseitig auf. Um die Klasse sinnvoll nutzen zu können, muß der Nutzer diesen Rekursionsmechanismus durchbrechen und mindestens eine der `invoke()`-Methoden überschreiben. Dies erfolgt meist durch Spezifikation einer *anonymen Klasse*.

Im Gegensatz zu allgemeinen Funktionen ist ein *Prädikat* eine spezielle, beliebigstellige Funktion, die auf *bool* abbildet. Um Prädikate semantisch von Funktionen zu unterscheiden,

existiert in XXL eine eigene abstrakte Klasse `Predicate`:

```
abstract class Predicate {
    boolean invoke();
    boolean invoke(Object[] args);
    boolean invoke(Object argument);
    boolean invoke(Object argument1, Object argument2);
}
```

`Predicate` basiert auf demselben Rekursionsmechanismus wie `Function`. Prädikate spielen z.B. eine wichtige Rolle für den Partitionierungsprozeß von GESS: die Aufteilungsstrategie von Hypercubes wird dem Replikationsalgorithmus durch Aufteilungsprädikate mitgeteilt (siehe `xxl.spatial.Replicator.Split`). Die logischen Prädikate `And` und `Or` ermöglichen ein einfaches Verknüpfen verschiedener Aufteilungsstrategien.

Eine zukünftige Erweiterung des Funktionskonzeptes in XXL könnte in der Trennung von `Function` in Funktionen, die einen Wert zurückgeben, und Prozeduren bestehen. Letztere Klasse ist in `xxl.util.concurrency` bereits vorhanden. Das Konzept eines Prozedurparameters ist auch eng verwandt mit dem *Strategy Pattern* [GHJV94]. Desweiteren ist eine Aufspaltung der abstrakten Klassen `Function` und `Predicate` in separate Schnittstellen für verschiedenstellige `invoke()`-Methoden bereits in Planung.

5.2.3 Queues

Zur Speicherung von Datenelementen stehen in XXL die wichtigen Schnittstellen `Queue` und `Container` bereit.

Container spielen insbesondere für Indexstrukturen eine große Rolle, haben für die vorliegende Arbeit aber keine Bedeutung. Deswegen wollen wir hier nicht weiter darauf eingehen und verweisen den Leser auf [BBD+01].

Von wesentlicher Bedeutung für Joinverfahren ist das Queue-Interface.

Betrachten wir zunächst noch einmal das *Iterator-Pattern*: für einen Iterator gilt, daß der Nutzer Datenelemente iterativ aus dem Iterator ‚zieht' durch Aufruf der Methode `next()` („PULL"-Technologie). D.h., der Nutzer fordert ein neues Element vom Iterator an und muß solange warten, bis der Iterator dieses Element produziert und zurückgegeben hat. Der eigentliche Iterator ist *passiv* und arbeitet nur, um das nächste angeforderte Element zu berechnen. Der Datenfluß erfolgt durch *Zug/Pull* vom Nutzer.

Im Gegensatz dazu kann der Nutzer einer `Queue` ein Datenelement *aktiv* übergeben („PUSH"-

Technologie) . Dies geschieht durch Aufruf der Methode `insert(Object object)`. In diesem Fall muß der Nutzer nicht warten, bis die `insert`-Operation beendet ist, sondern kann sofort im Programmablauf fortfahren[5]. Der Datenfluß erfolgt durch *Druck/Push* der Datenquelle.

In einer *Queue* fließen beide Konzepte (Push und Pull) zusammen.

Würde man in Analogie zum `[Pull]Iterator`-Interface ein `PushIterator`-Interface definieren, so könnte dieses folgendermaßen aussehen:

```
interface PushIterator {
    boolean takesNext();
    void next(Object object);  //entspricht insert-Operation
}
```

`takesNext()` gibt `true` zurück wenn der PushIterator noch weitere Elemente entgegennimmt. Mit `next(Object)` wird das nächste Element an den PushIterator übergeben.

Damit ergäbe sich ein Queue-Interface wie folgt:

```
interface Queue extends [Pull]Iterator, PushIterator {
    int size();
    void clear();
 }
```

Derzeit ist diese Trennung in separate Interfaces noch nicht in XXL realisiert.

Die Einführung von *PushIteratoren* hat weitreichende Folgen insbesondere für *Continuous Queries* [BW01, MF02, NACP01]. Mit Hilfe von *Push-Iteratoren* läßt sich letztendlich jedes Programm als Operatorbaum bestehend aus Push- und Pull-Iteratoren formulieren. Desweiteren ist in Analogie zu `xxl.cursors` eine Push-basierte Algebra denkbar. Hierfür wären auch Operatoren erforderlich, die Push- und Pull verbinden. Es gibt auch enge Beziehungen zu den Design Pattern *Multicast* sowie *Observer*. Letztere lassen sich vollständig durch atomare Push-Iteratoren formulieren.

Desweiteren haben hashbasierte Joinverfahren und auch Indexstrukturen enge Zusammenhänge mit diesem Konzept, da sie z.B. bei einer Einfügeoperation ein Datenobjekt in einen Behälter bzw. Teilast *pushen* müssen. Ein solches Einfügen ist immer ein Hinweis auf eine mögliche Parallelisierung eines Algorithmus.

Im Rahmen dieser Arbeit können wir aus Zeitgründen nicht weiter auf das *Push-Iterator Design Pattern* eingehen und kehren zum Queue-Interface zurück.

[5]Somit verweist die `insert`-Operation immer auf eine mögliche Parallelisierung von Programmcode.

Das Queue-Interface im aktuellen Release von XXL sieht — etwas vereinfacht — wie folgt aus:

```
interface Queue extends Iterator {
    void insert(Object object);
    int size();
    void clear();
 }
```

`insert()` fügt das übergebene Argument in die Queue ein. `size()` gibt die aktuelle Anzahl der in der Queue enthaltenen Elemente zurück. `clear()` entfernt alle Elemente aus der Queue.

In XXL stehen verschiedene Implementierungen des Queue-Interfaces bereit, u.a. gibt es eine `ListQueue` und `ArrayQueue` (Hauptspeicher) sowie eine `RandomAccessFileQueue` (Externspeicher). Für die Experimente in der vorliegenden Arbeit wurde eine modifizierte `RandomAccessFileQueue` genutzt, die auf raw-Devices arbeitet.

5.3 Integration der Joinalgorithmen

Im diesem Abschnitt beschreiben wir die Integration der sortierbasierten Algorithmen für den Ähnlichkeitsverbund `Orenstein` und `GESS` sowie des `Replicator` mit einem Fokus auf Klassenmodellierung und Datenfluß. Für die algorithmische Darstellung des generischen Sort-Merge Join siehe Kapitel 4.

5.3.1 xxl.spatial

Die in dieser Arbeit besprochenen Algorithmen werden im XXL-Paket `xxl.spatial` zur Verfügung gestellt. `xxl.spatial` enthält eine Vielzahl von Operatoren und Toolboxes zur Verarbeitung von ‚nichtstandard' Joinoperationen. Neben dem Joinverfahren für den räumlichen Verbund von Arge et.al. `PlaneSweep` [APR+98] werden die Similarity Joinverfahren `Orenstein` [Ore86, Ore91], `MSJ` [KS97, KS98, KS00] und `GESS` (Kapitel 3) bereitgestellt. Der Replikationsmechanismus von GESS steht in einer eigenen Klasse `Replicator` zur Verfügung, da er unabhängig von GESS verwendet werden kann. Er ist so generisch implementiert, daß er in einer zukünftigen Version von XXL z.B. für k-nearest-neighbor Joins und auch als Partitionierungsfunktion für Hashverfahren genutzt werden kann[6].

[6]Die `main()`-Methoden obiger Klassen enthalten vollständig implementierte Anwendungsfälle für den Spatial- bzw. Similarity Join.

Betrachten wir nun, wie obige Joinalgorithmen in XXL integriert sind.

5.3.2 Sort-Merge Join

Abbildung 5.3.2 zeigt das vereinfachte UML-Diagramm[7] für die behandelten Verfahren. An der Spitze des Diagramms ist zunächst das Interface `Iterator` aus `java.util` zu sehen. Dieses wird in XXL in zwei Schritten erweitert. Zunächst wird in `PeekIterator` die Peek-Funktionalität hinzugefügt. Das Interface `Cursor` schließlich erweitert `PeekIterator` um die Methoden `reset()`, `close()` und `update()`.

Das sogenannte *Marker-Interface* `Join` erweitert `Cursor`. Dieses Interface hat keine zusätzlichen Methoden. Es dient lediglich dazu, semantisch zu kennzeichnen, daß alle implementierenden Cursor Joinverfahren sind[8]. D.h. das `Join`-Interface entspricht dem *logischen* Joinoperator.

Der eigentliche generische `SortMergeJoin` implementiert das `Join`-Interface. Der Name der Klasse signalisiert, daß es sich hierbei um einen *physikalischen* Joinoperator — eine physikalische Ausprägung des logischen Operators — handelt, der auf dem Sort-Merge-Paradigma beruht. Die eigentliche Joinfunktionalität wird nur einmal — im `SortMergeJoin` — zur Verfügung gestellt.

Der `SortMergeJoin` stellt ein inneres Interface `SweepArea` zur Verfügung. Dieses muß von allen erbenden Joins implementiert werden: es legt für spezielle Joins die zu verwendende Hauptspeicherstruktur fest. Ferner werden hier die Löschstrategie spezifiziert und das Joinprädikat ausgewertet (vgl. Kapitel 4). Die Löschstrategie wird durch Implementierung der Methode `reorganize(Object currentStatus)` spezifiziert. Diese Methode löscht alle diejenigen Elemente aus der Sweep Area, die bezüglich des aktuell behandelten Elementes `reorganize` eine bestimmte Bedingung erfüllen[9]. D.h. diese Methode implementiert das in Kapitel 4 erwähnte Löschprädikat P_{rm}.

Die Methode `query` legt fest, wie Anfragen bezüglich des Joinprädikates beantwortet werden[10]. Aus Effizienzgründen hat diese Methode Zugriff auf interne Attribute der Sweep Area. Zurückgegeben wird ein Cursor mit den Elementen, die sich bezüglich des Prädikates qualifizieren.

[7] Für jede Klasse/Interface werden aus Platzgründen jeweils nur die wichtigsten Attribute und Methoden gezeigt. Jede Klasse/Interface zeigt nur diejenigen Methoden, die neu hinzukommen.

[8] Dies entspricht der Vorgehensweise des SDK (vgl. z.B. `java.io.Serializable`).

[9] Bei einem Equi Join z.B. würden alle diejenigen Elemente gelöscht werden, die kleiner als `currentStatus` sind.

[10] Bei einem Equi Join alle diejenigen Elemente, die ‚gleich' dem `currentStatus` sind.

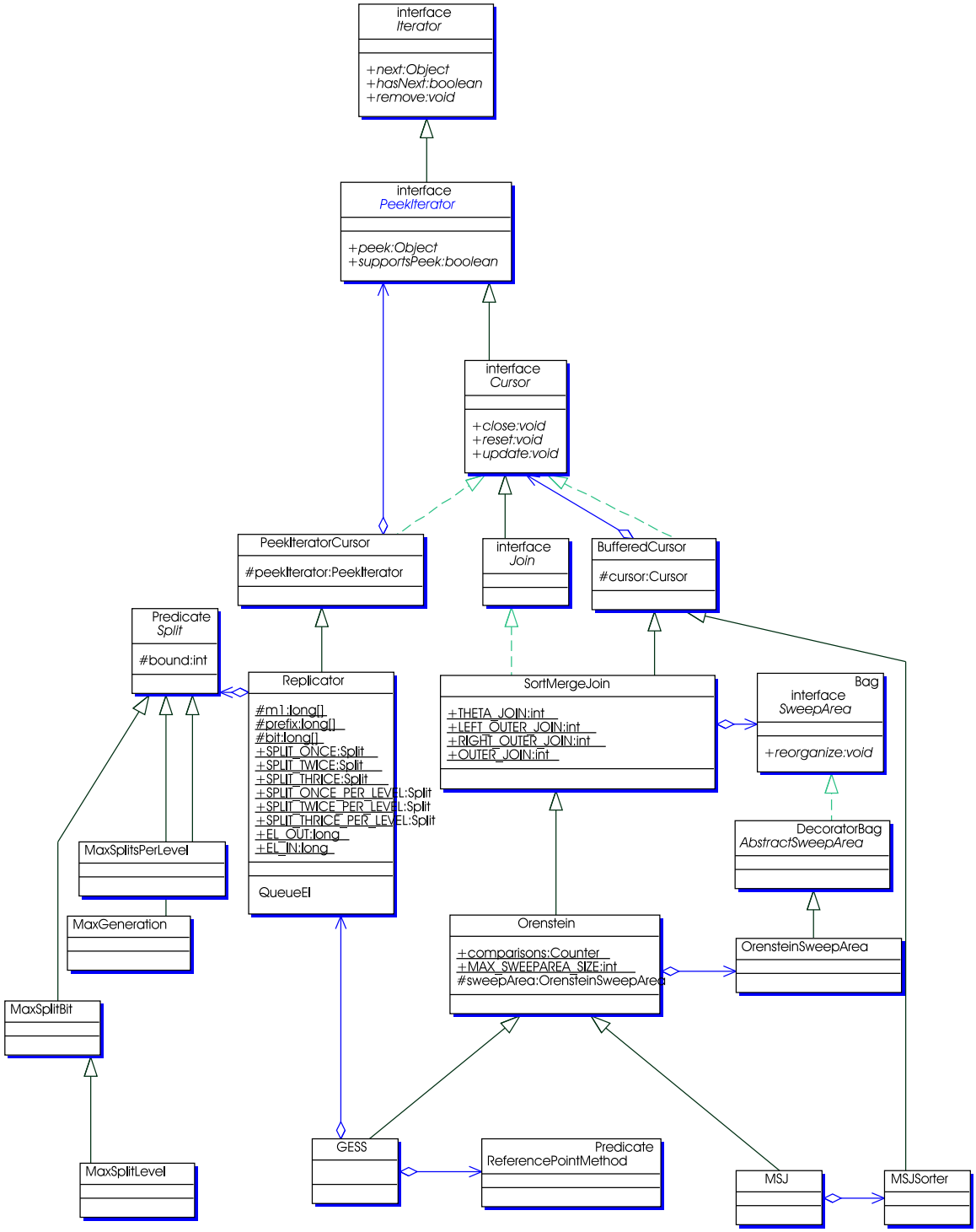


Abbildung 5.1: Integration der sortierbasierten Similarity Joins in XXL

Alle Joinalgorithmen, die auf dem Sort-Merge-Paradigma basieren, brauchen nur das Interface `SweepArea` zu implementieren und eventuell zusätzlich die Eingabedaten geeignet mappen. Eine Duplikation von Funktionalität findet nicht statt.

5.3.3 Orenstein

```
1  public class Orenstein extends SortMergeJoin {
2      public static class OrensteinSweepArea extends
                                          SortMergeJoin.AbstractSweepArea {
3          protected Cursor cursor;
4          public OrensteinSweepArea (LIFOBag lifoBag){
5              super(lifoBag);
6              cursor = lifoBag.lifoCursor();
7          }
8          public OrensteinSweepArea (int initialCapacity){
9              this(new ArrayBag(initialCapacity));
10         }
11         public void reorganize (final Object currentStatus) {
12             for(cursor.reset(); cursor.hasNext();){
13                 BitSet top = ((KPEzCode)cursor.next()).getzCode();
14                 BitSet query = ((KPEzCode)currentStatus).getzCode();
15                 if ((query.precision() < top.precision()) ||
                                          (query.compare( top ) != 0) )
16                     cursor.remove();
17                 else
18                     break;
19             }
20         }
21     }
22     //Konstruktoren:
23     public Orenstein(...){
24     }
25 }
```

Abbildung 5.2: Generische Implementierung des Algorithmus von Orenstein in XXL

Das Similarity Join Verfahren `xxl.spatial.Orenstein` erweitert den generischen `xxl.cursors.SortMergeJoin`. Zusätzlich stellt diese Klasse eine Implementierung der

Sweep Area (`OrensteinSweepArea`) bereit. Diese Sweep Area organisiert die Daten in einer speziellen `LIFOBag`[11]. Hierdurch können die Datenelemente im Hauptspeicher in Präfixordnung gehalten werden. Beim Löschen von Daten aus der Sweep Area wird diese LIFO-Strategie ausgenutzt: Elemente, die die Präfixeigenschaft verletzen, können nur an der Spitze der `LIFOBag` liegen[12].

Die Klasse `xxl.spatial.Orenstein` enthält in ihrer `main`-Methode einen vollständig implementierten Anwendungsfall für die Berechnung des Similarity-Join. Der Aufruf des Orenstein-Verfahrens erfolgt wie folgt:

```
Iterator input = new FloatPointInputIterator( // Erzeuge Iterator
                    fileName,                 // Eingabedatei
                    1024*1024,                // Groesse des Lesepuffers
                    dim                       // Dimensionalitaet der Daten
);

input = new FloatPointKPEzCodeMapper(         // Erzeuge Z-Wertberechnung
                    input,                    // Eingabeiterator
                    epsilon,                  // Anfragedistanz
                    maxLevel                  // groesste Partitionierungsebene
);

Orenstein orenstein = new Orenstein(          // Erzeuge Joinoperator
                    input,                    // Eingabeiterator
                    joinPredicate,            // Joinpraedikat
                    newSorter,                // Factory fuer Sortierroutine
                    Tuplify.DEFAULT_INSTANCE, // Factory fuer Ergebnistupel
                    capacity                  // Groesse der Sweep Area
);

while(orenstein.hasNext()){                   // Solange es Ergebnistupel gibt
    orenstein.next();                         // produziere das Ergebnistupel
}
```

Das Beispiel geht davon aus, daß die Daten — in diesem Fall ein Selbstverbund — in einer Datei vorliegen. Der `FloatPointInputIterator` erzeugt für die Datei einen Iterator von Punkten, d.h. die Objekte des Iterators sind vom Typ `FloatPoint`. Für jeden

[11] Im Prinzip ist dies ein Stack. Im Unterschied zum Stack erlaubt die `LIFOBag` allerdings Zugriff auf alle Datenelemente und nicht nur auf das top-Element. Diese Funktionalität wird für Anfragen benötigt.

[12] siehe Kapitel 3.1

der Eingabepunkte muß der Z-Wert berechnet werden. Dies geschieht mit Hilfe der Klasse `FloatPointKPEzCodeMapper`. Diese erzeugt für jeden Eingabepunkt und eine gegebene Anfragedistanz `epsilon` ein Tupel[13] bestehend aus Eingabepunkt und dem zugehörigen Z-Wert.

Die entstehenden Tupel werden an den Joinalgorithmus von Orenstein weitergereicht. Dieser bekommt zusätzlich ein Joinprädikat übergeben, in unserem Fall die im Filterschritt benötigte L_∞-Metrik vom Typ `DistanceWithinMaximum`. Der `Orenstein`-Operator bekommt ferner eine Factory-Methode für einen Sortieroperator übergeben: dieser muß die Eingabedaten bezüglich der Z-Ordnung aufsteigend sortieren. Die Funktion `Tuplify.DEFAULT_INSTANCE` legt fest, wie Ergebnistupel des Joins erzeugt werden sollen. Der Parameter `capacity` legt die Kapazität des verwendeten Stacks fest.

Abbildung 5.3 illustriert den Datenfluß und die Parametrisierung des Orenstein-Algorithmus gegenüber dem generischen Sort-Merge Join. In der Mitte der Graphik ist der generische `SortMergeJoin`-Operator zu sehen. Der linke Teil der Graphik zeigt die für den Orenstein-Algorithmus notwendigen Parametrisierungen. Der Datenfluß erfolgt aus Gründen der besseren Lesbarkeit von oben nach unten. Ein Datenelement der Eingabe wird zunächst auf ein spezielles Tupelformat (`KPEzCode`) gemappt[14]. Dieses Tupel enthält sowohl das Eingabeelement als auch den zugehörigen Z-Wert. Danach wird das so gewonnene Tupel an den generischen `SortMergeJoin` weitergereicht. Dieser sortiert alle Eingaben bezüglich ihrer Z-Werte. Die sortierten Datenströme werden dann an den eigentlichen Merge Join weitergeleitet. Dieser berechnet mit Hilfe der speziellen `OrensteinSweepArea` die Ergebnistupel.

Um den generischen `SortMergeJoin` nutzen zu können, muß dieser geeignete Parameter übergeben bekommen. Dies erfolgt mittels verschiedener Funktionsparameter. Es müssen übergeben werden: die Sortierordnung (mittels `java.util.Comparator`), eine Factory-Methode für Queues (`xxl.functions.Function`), das zu verwendende Joinprädikat (`xxl.predicates.Predicate`) und die zu verwendende Hauptspeicherstruktur (`xxl.spatial.Orenstein.OrensteinSweepArea`).

Abbildung 5.2 zeigt den vollständigen Code der entscheidenden `OrensteinSweepArea`. Diese enthält bereits einige Optimierungen: so wird beim Aufruf der Methode `reorganize()` nicht jedesmal ein neuer Cursor erzeugt[15] sondern intern eine Referenz auf einen Cursor gehalten (Zeile 3), welcher dann jeweils zurückgesetzt wird (Zeile 12). Ließe man diese Optimierung

[13] vom Typ `KPEzCode`. Die Namensgebung ist historisch begründet. Der Begriff *‚kpe‘* steht für *key-pointer-element* und wurde von Koudas und Sevcik eingeführt [KS97]

[14] z.B. für Punkte als Eingabe mit Hilfe des `FloatPointKPEzCodeMapper`

[15] Ein Aufruf von `new` ist in Java extrem teuer. Vgl. [Eck98].

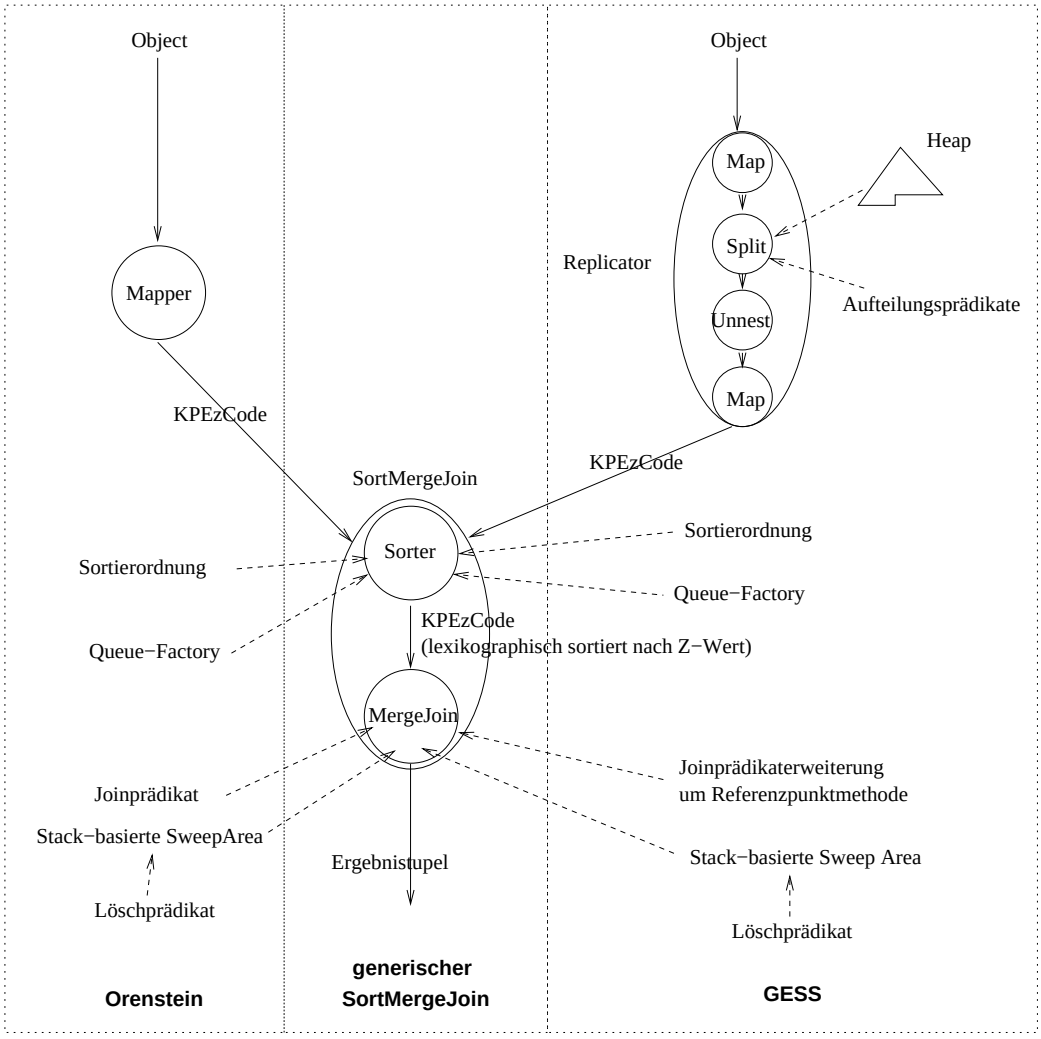


Abbildung 5.3: Datenfluß und Parameter für Orenstein und GESS in XXL

weg, würde sich der gesamte Code für die `OrensteinSweepArea` in XXL auf ca. 7 Zeilen reduzieren.

5.3.4 GESS

Das Similarity Join Verfahren `xxl.spatial.GESS` erweitert das Verfahren von Orenstein (`xxl.spatial.Orenstein`). Auch für diese Verfahren ist ein vollständig implementierter Anwendungsfall in der `main`-Methode von `xxl.spatial.GESS` zu finden. Wir zeigen zunächst wieder den Beispielaufruf in Java:

```
Iterator input = new FloatPointInputIterator( // Erzeuge Iterator
                    fileName,                 // Eingabedatei
                    1024*1024,                // Groesse des Lesepuffers
                    dim                       // Dimensionalitaet der Daten
);

GESS gess = new GESS(                         // Erzeuge Joinoperator
                    input,                    // Eingabeiterator
                    inputMapping,             // Mapping fuer Eingabedaten
                    joinPredicate,            // Joinpraedikat
                    splitAllowed,             // Aufteilungsstrategie
                    minBitIndex,              // maximale Partitionierungsebene
                    newSorter,                // Factory fuer Sortierroutine
                    Tuplify.DEFAULT_INSTANCE, // Factory fuer Ergebnistupel
                    dim,                      // Dimensionalitaet der Daten
                    initialCapacity           // Groesse der Sweep Area
);

while(gess.hasNext()){                        // Solange es Ergebnistupel gibt
    gess.next();                              // produziere das Ergebnistupel
}
```

Der Aufruf von `GESS` erfolgt ähnlich wie der von `Orenstein`. Es kommen allerdings einige wichtige Aufrufparameter hinzu: insbesondere muß der Parameter `inputMapping` definiert werden. Dies ist eine Abbildung, die ein beliebiges Objekt als Eingabe erwartet und ein Rechteck in Festkommazahldarstellung[16] zurückgibt. Mit Hilfe dieser Abbildung ist es möglich, GESS für beliebige Ausgangsdaten zu verwenden[17]

[16] siehe unten

[17] Für Similarity-Joins steht hierfür in `xxl.spatial.PointFixedPointRectangleNEWMapper` die innere

Ein zweiter wichtiger Parameter ist das Prädikat `splitAllowed`. Dieses legt die Aufteilungsstrategie von GESS fest. Für die im GESS-Kapitel untersuchten Strategien lautet der Aufruf wie folgt:

```
Predicate splitAllowed = new And(
                    new Replicator.MaxSplitsPerLevel( k ), //Strategie 1
                    new Replicator.MaxSplitLevel( msl )    //Strategie 2
);
```

Ferner ist es notwendig, das Joinprädikat zu modifizieren, um die Referenzpunktmethode zu nutzen. Bei Orenstein sieht die Initialisierung des Joinprädikates folgendermaßen aus:

```
Predicate joinPredicate =
                    new DistanceWithinMaximum( epsilon );  // Maximumsmetrik
```

Für GESS wird das Joinprädikat mittels einer Und-Operation erweitert:

```
Predicate joinPredicate =
           new And(
                    new DistanceWithinMaximum( epsilon ),  // Maximumsmetrik
                    new GESS.ReferencePointMethod( epsilonDiv2 ),
           );
```

D.h. die Duplikateliminierung wird nicht fest in GESS integriert und dann durch , Schalter' von außen gesteuert, sondern erfolgt auch in der Implementierung durch die im GESS-Kapitel beschriebene Prädikaterweiterung.

Kehren wir noch einmal zur graphischen Darstellung des Datenflusses in Abbildung 5.3 zurück. Im rechten Teil der Abbildung ist der Datenfluß und die Parametrisierung des GESS-Algorithmus gegenüber dem generischen Sort-Merge Join und dem Verfahren von Orenstein dargestellt.

Wir präsentieren zunächst den prinzipiellen Datenfluß für GESS. Danach gehen wir etwas näher auf die Festkommadarstellung und den `Replicator` ein.

Ein Datenelement der Eingabe wird zunächst im `Replicator` verarbeitet. Dieser bildet jedes Eingabeobjekt mit Hilfe eines vom Benutzer definierten Mappings zunächst auf die Festkommadarstellung ab und berechnet dann für die Eingabedaten die Replikate.

Hierbei entstehen für GESS ebenfalls Tupel vom Typ `KPEzCode`. Dieses Tupel enthält sowohl das Eingabeelement als auch den zugehörigen Z-Wert. Zusätzlich vermerken diese

Klasse `PointEpsilonFixedPointRectangleNEWMapping` zur Verfügung. Vgl. auch den Anwendungsfall in der `main`-Methode von `xxl.spatial.GESS`.

Tupel in einem Flag, ob es sich bei dem jeweiligen Tupel um ein Replikat handelt. Die so gewonnenen Tupel werden dann an den generischen `SortMergeJoin` weitergereicht. Dieser sortiert alle Eingabe bezüglich ihrer Z-Werte. Die sortierten Datenströme werden dann an den eigentlichen Merge Join weitergeleitet. Dieser berechnet mit Hilfe der speziellen `OrensteinSweepArea` die Ergebnistupel. Da aufgrund der Replikation Duplikate in der Antwortmenge entstehen können, wird das eigentliche Joinprädikat erweitert. Ist ein Ergebnistupel des zu berechnenden Joins gefunden worden, wird sofort überprüft, ob eines der beiden Objekte des Tupels ein Replikat ist. Ist dies der Fall, wird für das Tupel die Referenzpunktmethode gerufen (`GESS.ReferencePointMethode`). Da diese als Prädikat implementiert ist, kann das eigentliche Joinprädikat leicht mittels eines logischen `AND` um die Referenzpunktmethode erweitert werden.

5.3.4.1 Festkommadarstellung

GESS verwendet für die Berechnung von Replikaten aus Effizienzgründen eine *Festkommazahldarstellung.*

Die Java-Typen `double` und `float` nutzten intern eine *Fließkommazahldarstellung.* Nach IEEE-Standard 754 wird eine 64-Bit Fließkommazahl unterteilt in 3 Teile:

- **Bit 63:** Vorzeichen $+/-$
- **Bits 62–52:** Exponent $EXP = e_{65}, \ldots, e_{52}$
- **Bits 51–0:** Mantisse $m_{51}, \ldots, m_0$

Eine reellwertige Zahl wird dann angenähert durch eine Potenzsumme. Die Zahl wird hierbei normiert zur Basis zwei, d.h. der Exponent wird so gewählt, daß die Zahl in der Form

$$+/- 1.m_{51}, \ldots, m_0 \cdot 2^{EXP}$$

dargestellt werden kann. Der Exponent EXP wird hierbei als vorzeichenlose 11-Bit Zahl gespeichert. Dies wird erreicht indem zum eigentlichen Wert von EXP ein sogenannter *Bias* von $2^{10} - 1$ hinzugezählt wird.

Die Fließkommazahldarstellung ist für die Berechnung von Replikaten und Z-Werten sehr ungünstig, da bei jeder Operation der Exponent mitberücksichtigt werden muß. Ferner ist zu berücksichtigen, daß hochdimensionale Daten meist normiert auf den Einheitswürfel $[0;1)^d$ vorliegen oder leicht entsprechend normiert werden können.

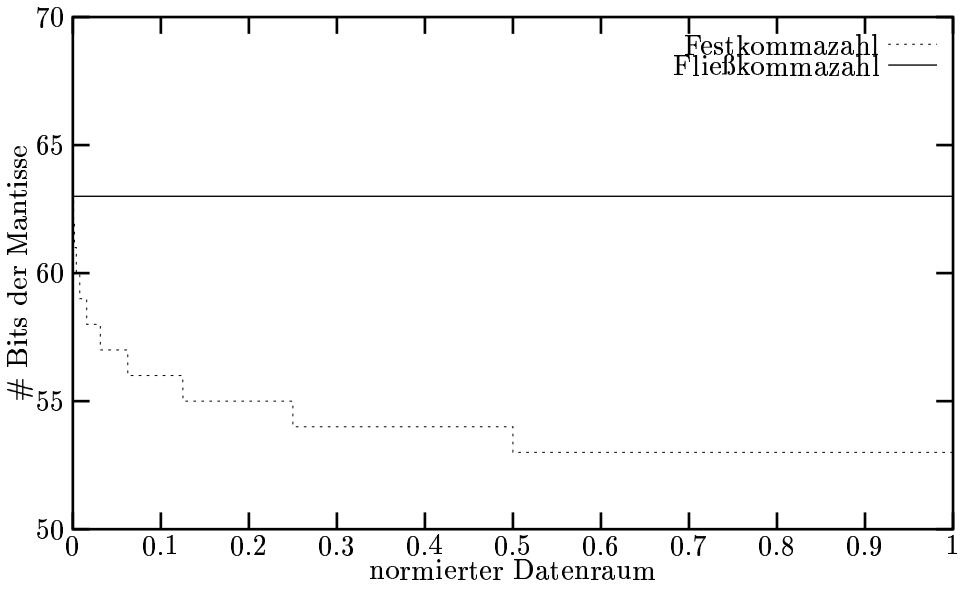


Abbildung 5.4: Präzision von gebrochenen Zahlen im normierten Raum $[0;1)$

Aus diesem Grund gehen wir zu einer Festkommazahldarstellung über, d.h. wir nutzen *alle* Bits (bis auf das Vorzeichenbit) als Mantisse. Unsere Festkommazahl sieht dann folgendermaßen aus:

$$+ < 0_{63}, m_{62}, \ldots, m_0 > .$$

Ein Exponent braucht nicht mitgespeichert zu werden. Die Zahl wird interpretiert als die Summe von

$$m_{62} \cdot 2^{-1} + m_{61} \cdot 2^{-2} + \ldots + m_0 \cdot 2^{-63}$$

D.h. dieser Zahlentyp hat eine *Genauigkeit* von 2^{-63} (63 Mantissenbits) über den gesamten Bereich des Intervalls $[0;1)$. Eine Fließkommazahl hingegen hat im allgemeinen eine Genauigkeit von 2^{-52}. Für Werte nahe 0 erhöht sich für Fließkommazahlen die Genauigkeit durch die Normierung, d.h. die Zahlen im Intervall $[2^{-i}; 2^{-i+1})$ $(i \geq 1)$ haben eine Genauigkeit von 2^{-i-52}. Hieraus folgt, daß eine Fließkommazahl nur im Intervall $[0; 2^{-10}) = [0; 0,0009766)$ (=1/1024 des Datenraums) eine höhere Genauigkeit als die Festkommadarstellung besitzt.

In der Festkommazahldarstellung besteht kein Unterschied mehr zwischen den gesetzten Bits der Mantisse der gebrochenen Zahl und den Partitionierungslinien (Aufteilungslinien) der betrachteten Dimension. D.h. die Bits der Mantisse entsprechen direkt den Aufteilungslinien eines Quadtrees bzw. kd-Tries (vergleiche auch [Dit99]).

Für die Berechnung, ob ein d-dimensionaler Hypercube eine Aufteilungslinie einer gegebenen Dimension i $(0 \leq i \leq d)$ der Partitionierungsebenen $l > 0$ schneidet, brauchen lediglich der min- und max-Wert des Hypercube der Dimension i des Bit $62-l$ betrachtet werden. Sind die

Bits unterschiedlich, schneidet der betreffende Hypercube die Aufteilungslinie. Anderenfalls liegt kein Schnitt mit der betreffenden Aufteilungslinie vor.

Beliebigdimensionale Z-Werte lassen sich in der Festkommadarstellung ebenso elegant berechnen: für einen d-dimensionalen Hypercube brauchen lediglich beginnend bei Bit 62 die einzelnen Bits ‚eingesammelt' und zu einem Z-Wert zusammengeführt werden. Dies kann vollständig ohne Multiplikationsoperationen und Exponentenanpassungen geschehen.

5.3.4.2 Replicator

Der `Replicator` erzeugt aus dem Eingabeobjekt mit Hilfe des vorgegebenen ε einen Hypercube[18]. Die interne Darstellung der Hypercubes basiert auf der oben beschriebenen *Festkommadarstellung.*

Sei $I = [a;b) \subset [0;1)$ ein eindimensionales Intervall eines Hypercubes der Dimension i $(0 \leq i \leq d)$. Sei t $(62 \geq t \geq 0)$ der Index des aktuell zu betrachtenden Bits. Es gelte $a.m_i = b.m_i \; \forall j > t$ sowie $a.m_t \neq b.m_t$. Da $b \geq a$, folgt $a.m_t = 0 \wedge b.m_t = 1$. Gesucht sind die disjunkten Replikatintervalle $I1$ und $I2$ von I mit

$$I1 \cup I2 = I, \; I1 \cap I2 = \emptyset.$$

Dann berechnen sich $I1$ und $I2$ zu

$$I1 = [\; a \; ; \; < a.m_{62} \ldots a.m_{t+1} \; 0_t \; 1_{t-1} \ldots 1_0 > \;]$$

$$I2 = [\; < a.m_{62} \ldots a.m_{t+1} \; 1_t \; 0_{t-1} \ldots 0_0 > \; ; \; b \;]$$

D.h. die Berechnung von Replikatintervallen kann vollständig durch Bitoperationen erfolgen.

Im `Replicator` werden die d-dimensionalen Hypercubes rekursiv betrachtet und in Abhängigkeit des *Aufteilungsprädikates*[19] repliziert. Die Rekursion wird mit Hilfe einer Queue aufgebrochen. Berechnete Replikate werden sofort an den nächsten Operator weitergereicht (und nicht zwischengespeichert).

5.3.4.3 Local Split Decision

Wir betrachten zunächst den ‚Standardfall' der *lokalen Split-Entscheidung* (vgl. auch [HSW89]). Hierbei wird *lokal* für einen Hypercube und eine Aufteilungslinie entschieden, ob eine Aufteilung erfolgen darf.

[18] Das ε kann für jedes Objekt verschieden gewählt werden.

[19] Zahlreiche wichtige Aufteilungssprädikate stehen als innere Klassen von `Replicator` bereit.

Der Replikationsprozeß für einen Hypercube H startet mit Bit 62 und Dimension 0. Es wird überprüft, ob eine Aufteilungslinie in dieser Dimension getroffen wird. Ist dies der Fall, wird das *Aufteilungsprädikat* gerufen. Falls letzteres für das momentane Paar (Bit,Dimension) Replikation erlaubt, wird der Hypercube H bezüglich der aktuellen Dimension in zwei Replikate gespalten. Beide Teile werden dann in die Queue eingefügt und im nächsten Schritt weiterbehandelt. Der Replikationsprozeß wird für beide Teile beim selben Bit der nächsten Dimension fortgeführt. Wurde die letzte Dimension betrachtet, wird das zu betrachtende Bit um eines heruntergezählt und wieder bei Dimension 0 angefangen. Dieser Vorgang wird rekursiv solange fortgesetzt, bis alle Replikate nach außen gegeben wurden. Für alle Replikate, die nach außen gegeben wurden, gilt, daß sie entweder die tiefste Partitionierungsebene erreicht haben bzw. das Aufteilungsprädikat `false` zurückgegeben hat.

5.3.4.4 Global Split Decision

Um eine *globale Aufteilungsentscheidung* handelt es sich immer dann, wenn nicht für eine einzelne Aufteilungslinie entschieden werden kann, ob der aktuelle Hypercube H aufgeteilt werden darf oder nicht. Ein Beispiel hierfür ist die in Kapitel 3.1 erwähnte Quadtree-Partitionierung. Das Prädikat

„Replikation ist dann erlaubt, wenn in dieser Partitionierungsebene nicht mehr als k Aufteilungslinien getroffen werden.“

kann nur berechnet werden, indem das aktuelle Bit *in allen Dimensionen* gleichzeitig berücksichtigt wird. Dies bedeutet, die Entscheidung kann nicht ‚lokal‘ an der Aufteilungslinie getroffen werden — andere ‚globale‘ Informationen sind notwendig. Auch dieses Vorgehen wird vom `Replicator` unterstützt. Hierfür existiert ein besonderes Prädikat `Replicator.MaxSplitsPerLevel`, welches nicht nur eine einzelne Aufteilungslinie sondern die kompletten Partitionierungsebene berücksichtigt. Es berechnet für die aktuelle Ebene einmalig die Anzahl der notwendigen Aufteilungen (=Anzahl der ungleichen Bits) und trifft dann die Entscheidung.

5.3.5 PMJ

PMJ ist ein `Cursor` der das Interface `xxl.cursors.Join` aus XXL implementiert. Das Verfahren basiert auf der generischen Sweep Area aus `xxl.cursors.SortMergeJoin`. D.h. PMJ wurde als generischer Joinoperator entwickelt: es werden keine Annahmen über die zu verwendenden Datentypen, Sweep Areas oder Sortierordnungen gemacht.

Der Aufruf für die Nutzung von PMJ mit GESS sieht folgendermaßen aus:

```
Function newSweepArea = new Function(){       // Factory fuer Sweep Area
            public Object invoke(){
                 return new Orenstein.OrensteinSweepArea(initialCapacity);
            }
};

Iterator input0 = new FloatPointInputIterator( // Erzeuge Iterator
                    fileName0,                 // Eingabedatei fuer R
                    dim                        // Dimensionalitaet der Daten
);

Iterator input1 = new FloatPointInputIterator( // Erzeuge Iterator
                    fileName1,                 // Eingabedatei fuer S
                    dim                        // Dimensionalitaet der Daten
);

input0  = new Replicator( inputMapping, input0, dim, splitAllowed, minBitIndex );
input1  = new Replicator( inputMapping, input1, dim, splitAllowed, minBitIndex );

Join join = new PMJ(                           // Erzeuge Joinoperator
                    input0,                    // Eingabeiterator fuer R
                    input1,                    // Eingabeiterator fuer S
                    comparator,                // Sortierkriterium
                    newQueue,                  // Factory fuer Queue
                    newSweepArea,              // Factory fuer Sweep Area
                    M,                         // Verfuegbarer Hauptspeicher
                    F                          // Maximaler Fanin
);

while(join.hasNext()){                         // Solange es Ergebnistupel gibt
    join.next();                               // produziere das Ergebnistupel
}
```

Wir gehen in dem Beispiel davon aus, daß das oben definierte Joinprädikat `splitAllowed` sowie die Erweiterung um die Referenzpunktmethode zur Verfügung stehen. Ferner nutzen wir das oben beschriebene `inputMapping`.

Zunächst wird in dem Beispiel eine *Factory-Methode* für die zu verwendende Sweep Area definiert. Anschließend werden die Eingabeiteratoren erzeugt. Danach wird für beide Eingaben eine Instanz des Replikators erzeugt. Schließlich wird PMJ instanziiert. Hierfür werden die beiden Eingabeiteratoren (`input0` und `input1`), die Sortierordnung (`comparator`), eine Factory für Queues (`newQueue`), die Factory für die Sweep Area (`newSweepArea`) sowie die Parameter für Hauptspeicher und Fanin M und F übergeben.

Für jeden konventionellen `SortMergeJoin` in XXL muß das innere Interface `SweepArea` implementiert sein. Letztere Implementierungen stehen in XXL zur Verfügung und können ohne Änderung für PMJ genutzt werden. Dies bedeutet insbesondere, daß der Nutzer von PMJ keine Funktionalität des eigentlichen Joinverfahrens nachimplementieren muß. Jeder konventionelle sortierbasierte Joinalgorithmus aus XXL ‚induziert' die jeweilige PMJ-Variante. Das Verwenden von PMJ ist somit sehr einfach möglich.

5.4 Zusammenfassung

XXL ist eine plattformunabhängige Javabibliothek von Frameworks und Algorithmen zur Anfrageverarbeitung. XXL wird in der AG Datenbanksysteme der Universität Marburg unter Leitung von Prof. Dr. Bernhard Seeger entwickelt.

Wir haben in diesem Kapitel die wichtigsten Designprinzipien und Konzepte der Bibliothek diskutiert. Insbesondere sind wir hierbei auf das Iterator Design Pattern eingegangen. Ferner haben wir die Integration der in dieser Arbeit vorgestellten Algorithmen beschrieben und haben über die algorithmischen Fragestellungen hinausgehende softwaretechnische Aspekte behandelt. Wir haben die Integration des Verfahrens für den Ähnlichkeitsverbund GESS und dessen Festkommazahlrepräsentation und Replikationsalgorithmus vorgestellt. Ferner sind wir auf das Zusammenspiel von PMJ mit GESS eingegangen.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Beiträge dieser Arbeit

6.1.1 GESS

Ausgehend vom *Multidimensional Spatial Join (MSJ)* wurden die Auswirkungen von Replikation auf die Laufzeit, die Anzahl der Distanzberechnungen, Hauptspeicherbedarf und Replikationsraten untersucht. Das I/O-Modell von MSJ wurde ersetzt und ein generischer Algorithmus namens *GESS (Generic External Space Sweep)* vorgeschlagen, der MSJ und andere Verfahren als Spezialfälle enthält.

Die Beiträge dieses Kapitels waren wie folgt:

1. Es wurde ein Kostenmodell für Replikation und die Anzahl der Distanzberechnungen für MSJ und GESS präsentiert. Das Kostenmodell wurde validiert mit Ergebnissen von Simulationen, die zeigen, daß GESS MSJ klar überlegen ist. Es wurde gezeigt, daß eine geringe Rate von kontrollierter Replikation (weniger als 15%) ausreicht, um die Anzahl der Distanzberechnungen um mehrere Größenordnungen zu reduzieren. Mögliche Duplikate in der Antwortmenge werden durch eine effiziente online Methode (*Referenzpunktmethode*) eliminiert.

2. Ferner wurde gezeigt, daß Replikation den Hauptspeicherbedarf in hohem Maße reduziert. Der benötigte Hauptspeicher für GESS ist mehrere Größenordnungen unter dem für MSJ.

3. GESS basiert auf einer gegenüber MSJ effizienteren I/O-Strategie. Es wurde eine I/O-Kosten-Analyse von GESS präsentiert und gezeigt, daß letzteres Verfahren mit der

Hälfte oder weniger der I/O-Operationen von MSJ auskommt. Diese Verbesserung ist unabhängig davon, ob Replikation genutzt wird.

4. Es wurden Ergebnisse von Experimenten mit echten hochdimensionalen Datensätzen präsentiert. Die Experimente bestätigen die analytischen Ergebnisse und zeigen die klare Überlegenheit von GESS gegenüber MSJ.

5. Schließlich wurde das Kostenmodell für die Anzahl der Distanzberechnungen um das Konzept der fraktalen Dimension erweitert. Die Voraussagen des Kostenmodells wurden mit Ergebnissen von Experimenten mit realen Datensätzen verglichen.

6.1.2 PMJ

Viele nicht-standard Joinalgorithmen — so auch der Similarity Join — erfordern das Sortieren der Eingabedaten vor der Berechnung des eigentlichen Joins. Während des Sortierens wird der Datenstrom blockiert, so daß nachfolgende Operatoren warten müssen.

Die Beiträge dieses Kapitels waren wie folgt:

1. Es wurde ein nichtblockierender generischer Joinalgorithmus namens *Progressive Merge Join (PMJ)* präsentiert. PMJ ist immer anwendbar, wenn ein blockierender konventioneller Joinalgorithmus existiert. Insbesondere für den Similarity Join ist das Verfahren der — nach unserem Wissen — erste nichtblockierende Joinalgorithmus. Der Algorithmus ist leicht in kommerzielle Datenbanksysteme zu integrieren.

2. Es wurde die Korrektheit des Verfahrens bewiesen und die Laufzeit untersucht. Die Laufzeit von PMJ entspricht asymptotisch der des konventionellen Sort-Merge Join.

3. In einer Reihe von Experimenten wurde die Anzahl der produzierten Ergebnistupel bezüglich der Laufzeit (Anzahl der I/O-Operationen bzw. Ausführungszeit) des Verfahrens untersucht. Es wurden für den generischen Progressive Merge Join drei Anwendungsfälle untersucht: 1.) der relationale Verbund (Equi Join), 2.) der räumliche Verbund (Plane Sweep) sowie 3.) der Ähnlichkeitsverbund (GESS). Die Ergebnisse haben gezeigt, daß PMJ wesentlich früher Ergebnistupel produziert als die konventionellen Ansätze. Ferner ist die Gesamtlaufzeit von PMJ nur wenig höher. Ferner wurde die Erweiterung von PMJ um ein einfaches Schätzverfahren betrachtet.

6.1.3 XXL

Heutige Datenbanksysteme sind zu inflexibel, um neue Techniken der Anfragebearbeitung zu integrieren. Aus diesem Grund wird in der AG Datenbanksysteme unter Leitung von Prof. Seeger die Software-Bibliothek *XXL (eXtensible and fleXible Library)* entwickelt. XXL ist eine plattformunabhängige Javabibliothek von Frameworks und Algorithmen zur Anfrageverarbeitung. Anfrageoperatoren aus XXL können leicht in kommerzielle Datenbanksystemen integriert werden. So wurde in [BBD+01] gezeigt, wie Spatial Join Operatoren aus XXL in ein kommerzielles Datenbanksystem (Cloudscape [CLO]) integriert und effizient genutzt werden können.

Die Beiträge dieses Kapitels waren wie folgt:

1. Es wurden die Designprinzipien von XXL vorgestellt sowie die wichtigsten Schnittstellen und Konzepte präsentiert. Insbesondere wurde das Iterator Design Pattern diskutiert.

2. Es wurden die Realisation und Integration der in dieser Arbeit behandelten Joinalgorithmen beschrieben. Hierbei wurden insbesondere über die eigentlichen algorithmischen Fragestellungen hinausgehende softwaretechnische Aspekte behandelt.

3. Ferner wurde die Festkommazahldarstellung und der Replikationsalgorithmus von GESS dargestellt.

4. Schließlich wurde die softwaretechnische Realisation von PMJ sowie die Integration von GESS in PMJ beschrieben.

6.2 Ausblick

In dieser Arbeit wurden nichtblockierende Algorithmen für den Ähnlichkeitsverbund untersucht. Der Fokus lag hierbei auf Ansätzen und Lösungen, die zum einen für eine Vielzahl von Problemstellungen genutzt und zum anderen leicht an neue Problemstellungen angepaßt werden können. Algorithmen, die diese Anforderungen erfüllen, heißen *generische Algorithmen.*

Der Vorteil einer solchen Herangehensweise entspricht zum einen den Forderungen der Datenbankhersteller, die nicht an einer Vielzahl spezialisierter Lösungen sondern an einer Handvoll flexibel und vielseitig einsetzbarer Algorithmen interessiert sind. Don Haderle,

Chefarchitekt des Datenbanksystems DB2 bei IBM, formulierte dies auf der ACM SIGMOD 2000 folgendermaßen:

> "Please don't give me another join algorithm, unless it is *the* join algorithm!"

Zum anderen schaffen generische Algorithmen vom wissenschaftlichen Standpunkt betrachtet eine bessere Übersicht über bestehende Ansätze. Dies geschieht immer dann, wenn es gelingt, ‚unterschiedliche Algorithmen' in einer gemeinsamen Kernidee — dem generischen Algorithmus — zusammenzuführen.

Die in dieser Arbeit entwickelten Algorithmen GESS und PMJ, sowie die Software-Bibliothek XXL haben wesentliche Schritte in diese Richtung gemacht.

Die Algorithmen wurden zwar im Kontext des Similarity Join präsentiert, sind aber nicht hierauf beschränkt und können für zahlreiche andere Jointypen genutzt werden. So kann GESS auch für Spatial oder Temporal Joins, PMJ für beliebige Joins genutzt werden.

Bei der konkreten Untersuchung des Similarity Joins hat sich insbesondere gezeigt, daß die Gleichverteilungsannahme für Datensätze der realen Welt nicht zutrifft. Konsequenterweise schätzen analytische Modelle den Wert für ε um mehrere Größenordnungen zu hoch.

Unsere Experimente stützen das Konzept der *fraktalen Dimension* und legen nahe, daß dieses ein besseres Maß für die wahre Dimensionalität eines Datensatzes ist. Hierbei stellt sich aber für Datensätze der realen Welt die Frage, ob ein einziger Wert — die fraktale Dimension — wirklich ausreicht, die ‚inhärente Eigenschaft' eines Datensatzes ausreichend zu beschreiben. Wir schlagen deshalb vor, das Konzept der fraktalen Dimension zu erweitern um **lokale fraktale Dimensionen**. Diese beschreiben disjunkte Teilmengen eines Datensatzes unterschiedlicher fraktaler Dimensionen. Die ‚fraktale Dimension eines Datensatzes' wäre in einem solchen Konzept nur eine Aggregation der fraktalen Dimensionen der einzelnen Teilmengen.

Für ein solches Konzept wäre auch die Verwendung einer Indexstruktur als Aggregationshierarchie denkbar. Desweitern könnte eine solche Indexstruktur genutzt werden, um die Joinstrategie festzulegen bzw. zur Laufzeit auszutauschen. Ferner wäre mit der lokalen fraktalen Dimensionen eine Erweiterung von Similarity Joins auf k-Nächste-Nachbar-Joins möglich, da hiermit das ε für jeden Datenpunkt anhand der entsprechenden lokalen fraktalen Dimension festgelegt werden könnte.

Eine zukünftige Variante von PMJ könnte diese Techniken integrieren und damit die Wahl des Joinprädikates bzw. der Joinstrategie automatisieren — in Analogie zu den Ripple Joins.

Ein weiterer interessanter Aspekt wäre die Evaluierung und Erweiterung von PMJ für unendliche, aktive Datenströme, sowie die Untersuchung von Varianten für Multiprozessor bzw. Multidisksysteme.

Literaturverzeichnis

[ABKS99] Ankerst, Mihael, Markus M. Breunig, Hans-Peter Kriegel und Jörg Sander: *OPTICS: Ordering Points To Identify the Clustering Structure.* ACM SIGMOD, Seiten 49–60, 1999.

[APR+98] Arge, Lars, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel und Jeffrey Scott Vitter: *Scalable Sweeping-Based Spatial Join.* VLDB, Seiten 570–581, 1998.

[BBBK00] Böhm, Christian, Bernhard Braunmüller, Markus M. Breunig und Hans-Peter Kriegel: *High Performance Clustering Based on the Similarity Join.* ACM CIKM, Seiten 298–313, 2000.

[BBD+01] Bercken, Jochen Van den, Björn Blohsfeld, Jens-Peter Dittrich, Jürgen Krämer, Tobias Schäfer, Martin Schneider und Bernhard Seeger: *XXL — A Library Approach to Supporting Efficient Implementations of Advanced Database Queries.* VLDB, Seiten 39–48, 2001.

[BBKK97] Berchtold, Stefan, Christian Böhm, Daniel A. Keim und Hans-Peter Kriegel: *A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space.* ACM PODS, Seiten 78–86, 1997.

[BBKK01] Böhm, Christian, Bernhard Braunmüller, Florian Krebs und Hans-Peter Kriegel: *Epsilon Grid Order: An Algorithm for the Similarity Join on Massive High-Dimensional Data.* ACM SIGMOD, Seiten 379–388, 2001.

[BDS00] Bercken, Jochen Van den, Jens-Peter Dittrich und Bernhard Seeger: *javax.XXL: a Prototype for a Library of Query Processing Algorithms.* ACM SIGMOD, Seite 588, 2000.

[BE77] Blasgen, Mike W. und Kapali P. Eswaran: *Storage and Access in Relational Data Bases.* IBM Systems Journal, 16(4):362–377, 1977.

[BF95] Belussi, Alberto und Christos Faloutsos: *Estimating the Selectivity of Spatial Queries Using the 'Correlation' Fractal Dimension.* VLDB, Seiten 299–310, 1995.

[BF98] Belussi, Alberto und Christos Faloutsos: *Self-Spatial Join Selectivity Estimation Using Fractal Concepts.* ACM TOIS, 16:161–201, 1998.

[BK01] Böhm, Christian und Hans-Peter Kriegel: *A Cost Model and Index Architecture for the Similarity Join.* IEEE ICDE, Seiten 411–420, 2001.

[BKNS00] Breunig, Markus M., Hans-Peter Kriegel, Raymond T. Ng und Jörg Sander: *LOF: Identifying Density-Based Local Outliers.* ACM SIGMOD, Seiten 93–104, 2000.

[BKS93] Brinkhoff, Thomas, Hans-Peter Kriegel und Bernhard Seeger: *Efficient processing of spatial joins using R-trees.* ACM SIGMOD, Seiten 237–246, 1993.

[BKSS90] Beckmann, Norbert, Hans-Peter Kriegel, Ralf Schneider und Bernhard Seeger: *The R*-tree: An Efficient and Robust Access Method for Points and Rectangles.* ACM SIGMOD, Seiten 322–331, 1990.

[BKSS94] Brinkhoff, Thomas, Hans-Peter Kriegel, Ralf Schneider und Bernhard Seeger: *Multi-Step Processing of Spatial Joins.* ACM SIGMOD, Seiten 197–208, 1994.

[Boe88] Boehm, Barry: *A spiral model of software development and enhancement.* IEEE Computer, 25:61–71, May 1988.

[BSS00] Bercken, Jochen Van den, Martin Schneider und Bernhard Seeger: *Plug&Join: An easy-to-use Generic Algorithm for Efficiently Processing Equi and Non-Equi Joins.* EDBT, Seiten 495–509, 2000.

[Bur96] Bureau of Census: *TIGER/Line precensus files (1996 Technical Documentation)*, 1996.

[BW01] Babu, Shivnath und Jennifer Widom: *Continuous Queries over Data Streams.* ACM SIGMOD, Seiten 109–120, 2001.

[CLO] *http://www.cloudscape.com.* [Stand 2002-4-15].

[COL] *http://tilde-hoschek.home.cern.ch/~hoschek/colt/.* [Stand 2002-4-15].

[Dit99] DITTRICH, JENS-PETER: *Implementierung und Vergleich von Verfahren zur Berechnung des räumlichen Verbunds*, 1999. Diplomarbeit, Fachbereich Mathematik und Informatik, Universität Marburg.

[DNS91] DEWITT, DAVID J., JEFFREY F. NAUGHTON und DONOVAN A. SCHNEIDER: *An Evaluation of Non-Equijoin Algorithms.* VLDB, Seiten 443–452, 1991.

[DS00] DITTRICH, JENS-PETER und BERNHARD SEEGER: *Data Redundancy and Duplicate Detection in Spatial Join Processing.* IEEE ICDE, Seiten 535–546, 2000.

[DS01] DITTRICH, JENS-PETER und BERNHARD SEEGER: *GESS: a Scalable Similarity-Join Algorithm for Mining Large Data Sets in High Dimensional Spaces.* ACM SIGKDD, Seiten 47–56, 2001.

[DSTW02a] DITTRICH, JENS-PETER, BERNHARD SEEGER, DAVID SCOT TAYLOR und PETER WIDMAYER: *On the Efficiency of computing Join Results early.* 2002. In Vorbereitung.

[DSTW02b] DITTRICH, JENS-PETER, BERNHARD SEEGER, DAVID SCOT TAYLOR und PETER WIDMAYER: *Progressive Merge Join: A Generic and Non-Blocking Sort-Based Join Algorithm.* VLDB, 2002. Im Erscheinen.

[Eck98] ECKEL, BRUCE: *Thinking in Java.* Prentice Hall PTR, 1998.

[EKSX96] ESTER, MARTIN, HANS-PETER KRIEGEL, JÖRG SANDER und XIAOWEI XU: *A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise.* Seiten 226–231, 1996.

[FSTT00] FALOUTSOS, CHRISTOS, BERNHARD SEEGER, AGMA J. M. TRAINA und CAETANO TRAINA JR.: *Spatial Join Selectivity Using Power Laws.* ACM SIGMOD, Seiten 177–188, 2000.

[GHJV94] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON und JOHN VLISSIDES: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Massachusetts, 1994.

[GKS01] GEHRKE, JOHANNES, FLIP KORN und DIVESH SRIVASTAVA: *On Computing Correlated Aggregates Over Continual Data Streams.* ACM SIGMOD, Seiten 13–24, 2001.

[GNU] *http://www.gnu.org*. [Stand 2002-4-15].

[Gra93] GRAEFE, GOETZ: *Query Evaluation Techniques for Large Databases.* ACM Computing Surveys, 25(2):73–170, 1993.

[Gra94a] GRAEFE, GOETZ: *Sort-Merge-Join: An Idea Whose Time Has(h) Passed?* IEEE ICDE, Seiten 406–417, 1994.

[Gra94b] GRAEFE, GOETZ: *Volcano - An Extensible and Parallel Query Evaluation System.* IEEE TKDE, 6(1):120–135, 1994.

[GS87] GÜTING, RALF H. und W. SCHILLING: *A Practical Divide-and-Conquer Algorithm for the Rectangle Intersection Problem.* Information Sciences, 42:95–112, 1987.

[Gut84] GUTTMAN, ANTONIN: *R-trees: A Dynamic Index Structure for Spatial Searching.* ACM SIGMOD, Seiten 47–57, 1984.

[Haw80] HAWKINS, DOUGLAS M.: *Identification of outliers.* Chapman and Hall, 1980.

[HH99] HAAS, PETER J. und JOSEPH M. HELLERSTEIN: *Ripple Joins for Online Aggregation.* ACM SIGMOD, Seiten 287–298, 1999.

[Hil91] HILBERT, DAVID: *Über die stetige Abbildung einer Linie auf ein Flächenstück.* Mathematische Annalen, 38:459–460, 1891.

[HS95] HOEL, ERIK G. und HANAN SAMET: *Benchmarking Spatial Join Operations with Spatial Output.* VLDB, Seiten 606–618, 1995.

[HSW89] HENRICH, ANDREAS, HANS-WERNER SIX und PETER WIDMAYER: *The LSD tree: Spatial Access to Multidimensional Point and Nonpoint Objects.* VLDB, Seiten 45–53, 1989.

[Hug89] HUGHES, JOHN: *Why Functional Programming Matters.* Computer Journal, 32(2):98–107, 1989.

[IFF+99] IVES, ZACHARY G., DANIELA FLORESCU, MARC FRIEDMAN, ALON Y. LEVY und DANIEL S. WELD: *An Adaptive Query Execution System for Data Integration.* ACM SIGMOD, Seiten 299–310, 1999.

[Knu73] KNUTH, DONALD E.: *The Art of Computing, vol. 3 – Sorting and Searching.* Addison-Wesley, 1973.

[KS97] KOUDAS, NICK und KENNETH C. SEVCIK: *Size Separation Spatial Join.* ACM SIGMOD, Seiten 324–335, 1997.

[KS98] Koudas, Nick und Kenneth C. Sevcik: *High Dimensional Similarity Joins: Algorithms and Performance Evaluation.* IEEE ICDE, Seiten 466–475, 1998.

[KS00] Koudas, Nick und Kenneth C. Sevcik: *High Dimensional Similarity Joins: Algorithms and Performance Evaluation.* IEEE TKDE, 12:3–18, 2000.

[LLL01] Liao, S., Mario A. Lopez und Scott T. Leutenegger: *High Dimensional Similarity Search With Space Filling Curves.* IEEE ICDE, Seiten 615–622, 2001.

[LNE02] Luo, Gang, Jeffrey F. Naughton und Curt Ellmann: *A Non-blocking Parallel Spatial Join Algorithm.* IEEE ICDE, 2002. Im Erscheinen.

[Lom91] Lomet, David B.: *Grow and Post Index Trees: Roles, Techniques and Future Potential.* SSD, Seiten 183–206, 1991.

[LR94] Lo, Ming-Ling und Chinya V. Ravishankar: *Spatial Joins Using Seeded Trees.* ACM SIGMOD, Seiten 209–220, 1994.

[LR95] Lo, Ming-Ling und Chinya V. Ravishankar: *Generating Seeded Trees from Data Sets.* SSD, Seiten 328–347, 1995.

[LR96] Lo, Ming-Ling und Chinya V. Ravishankar: *Spatial Hash-Joins.* ACM SIGMOD, Seiten 247–258, 1996.

[LR98] Lo, Ming-Ling und Chinya V. Ravishankar: *The Design and Implementation of Seeded Trees: An Efficient Method for Spatial Joins.* IEEE TKDE, 10(1):136–152, 1998.

[ME92] Mishra, Priti und Margaret H. Eich: *Join Processing in Relational Systems.* ACM Computing Surveys, 24(1):63–113, 1992.

[MF02] Madden, Sam und Michael J. Franklin: *Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data.* IEEE ICDE, 2002.

[NACP01] Nguyen, Benjamin, Serge Abiteboul, Gregory Cobena und Mihai Preda: *Monitoring XML Data on the Web.* ACM SIGMOD, Seiten 437–448, 2001.

[NP85] Negri, Mauro und Giuseppe Pelagatti: *Join During Merge: An Improved Sort Based Algorithm.* IPL, 21(1):11–16, 1985.

[Ore86] Orenstein, Jack A.: *Spatial Query Processing in an Object-Oriented Database System.* ACM SIGMOD, Seiten 326–336, 1986.

[Ore91] Orenstein, Jack A.: *An Algorithm for Computing the Overlay of k–Dimensional Spaces.* SSD, Seiten 381–400, 1991.

[PD96] Patel, Jignesh M. und David J. DeWitt: *Partition Based Spatial-Merge Join.* ACM SIGMOD, Seiten 259–270, 1996.

[PS85] Preparata, Franco P. und Michael I. Shamos: *Computational Geometry.* Springer-Verlag, New York, 1985, 1985.

[Roy70] Royce, Walker W.: *Managing the development of large software systems: concepts and techniques.* Proc. IEEE WESTCON, Los Angeles, Seiten 1–9, 1970.

[RS86] Raschid, Louiqa und Stanley Y. W. Su: *A Parallel Processing Strategy for Evaluating Recursive Queries.* VLDB, Seiten 412–419, 1986.

[Sam90a] Samet, Hanan: *Applications of Spatial Data Structures.* Addison-Wesley, 1990.

[Sam90b] Samet, Hanan: *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.

[Sco92] Scott, David W.: *Multivariate Density Estimation.* Wiley-Interscience, 1992.

[SRF87] Sellis, Timos K., Nick Roussopoulos und Christos Faloutsos: *The R+-Tree: A Dynamic Index for Multi-Dimensional Objects.* VLDB, Seiten 507–518, 1987.

[SSA97] Shim, Kyuseok, Ramakrishnan Srikant und Rakesh Agrawal: *High-Dimensional Similarity Joins.* IEEE ICDE, Seiten 301–313, 1997.

[SSJ94] Soo, Michael D., Richard T. Snodgrass und Christian S. Jensen: *Efficient Evaluation of the Valid-Time Natural Join.* IEEE ICDE, Seiten 282–292, 1994.

[UF00] Urhan, Tolga und Michael J. Franklin: *XJoin: A Reactively-Scheduled Pipelined Join Operator.* IEEE Data Engineering Bulletin, 23(2):27–33, 2000.

[WA91] Wilschut, Annita N. und Peter M.G. Apers: *Pipelining in Query Execution.* Conference on Databases, Parallel Architectures and their Applications, Miami, USA, Seiten 68–77, 1991.

[XXL] *http://www.mathematik.uni-marburg.de/DBS/xxl*. [Stand 2002-4-15].

[ZL97] Zhang, Weiye und Per-Åke Larson: *Dynamic Memory Adjustment for External Mergesort*. VLDB, Seiten 376–385, 1997.

Abbildungsverzeichnis

1.1 Der Similarity Join als Massenanfrage 1

1.2 Länge ε eines Hypercube in einer Dimension in Abhängigkeit von d (Gesamtzahl der Dimensionen) 7

1.3 Das Blockierverhalten sortierbasierter Verbundoperationen 8

2.1 Nested-Loops Join 11

2.2 Indexed Nested-Loops Join 11

2.3 Hashjoin 12

2.4 Beispiel für Equi Join 12

2.5 Auswirkung der Partitionierung bei verschiedenen Joinalgorithmen (Nested-Loops Join, Hashjoin, Sort-Merge Join, vgl. Mishra und Eich [ME92]) . . . 14

3.1 Orensteins Merge-Algorithmus 23

3.2 Der MSJ-Algorithmus 23

3.3 Der GESS-Algorithmus 25

3.4 Der Replikationsalgorithmus von GESS 26

3.5 Die Referenzpunktmethode (RPM) 27

3.6 Zusätzliche I/O-Kosten von MSJ für Mischoperationen (untere Schranke) . 35

3.7 Maximale Größe der Eingaberelationen ohne Nutzen von Mischoperationen 36

3.8 Verbesserung von GESS gegenüber MSJ bezüglich I/O-Kosten 37

3.9 Vergleich des analytischen Modells mit Ergebnissen der Simulation 40

3.10 Distanzberechnungen: Verbesserungsfaktor IF für GESS gegenüber MSJ . . 41

3.11 Hauptspeicherbedarf von GESS in Abhängigkeit von k ($k = 0$: MSJ) 43

3.12 Laufzeit für Ähnlichkeitsselbstverbund von CAD für verschiedene k ($k = 0$: MSJ), $\varepsilon = 10^{-3}$ (log-Skala auf y-Achse) 45

3.13 Laufzeit für Ähnlichkeitsselbstverbund von CAD für verschiedene k und msl, $\varepsilon = 10^{-3}$ 46

3.14 Laufzeit für Ähnlichkeitsselbstverbund von CT für verschiedene k und msl, $\varepsilon = 10^{-3}$ 46

3.15 Laufzeit für Ähnlichkeitsselbstverbund von CM für verschiedene k und msl, $\varepsilon = 10^{-3}$ 47

3.16 Laufzeit für Ähnlichkeitsselbstverbund von CH für verschiedene k und msl, $\varepsilon = 10^{-3}$ 47

3.17 Distanzberechnungen DC für Ähnlichkeitsselbstverbund für CAD für verschiedene k ($k = 0$: MSJ), $\varepsilon = 10^{-3}$ (log-Skala auf y-Achse) 48

3.18 Distanzberechnungen DC für Ähnlichkeitsselbstverbund von CT für verschiedene k und msl ($k = 0$: MSJ), $\varepsilon = 10^{-3}$ (log-Skala auf y-Achse) 49

3.19 Distanzberechnungen DC für Ähnlichkeitsselbstverbund von CM für verschiedene k und msl ($k = 0$: MSJ), $\varepsilon = 10^{-3}$ (log-Skala auf y-Achse) 50

3.20 Distanzberechnungen DC für Ähnlichkeitsselbstverbund von CH für verschiedene k und msl ($k = 0$: MSJ), $\varepsilon = 10^{-3}$ (log-Skala auf y-Achse) 50

3.21 Replikationsraten für Ähnlichkeitsselbstverbund für CAD für verschiedene k ($k = 0$: MSJ), $\varepsilon = 10^{-3}$ 51

3.22 Replikationsraten für Ähnlichkeitsselbstverbund von CT für verschiedene k und msl ($k = 0$: MSJ), $\varepsilon = 10^{-3}$ 51

3.23 Replikationsraten für Ähnlichkeitsselbstverbund von CM für verschiedene k und msl ($k = 0$: MSJ), $\varepsilon = 10^{-3}$ 52

3.24 Replikationsraten für Ähnlichkeitsselbstverbund von CH für verschiedene k und msl ($k = 0$: MSJ), $\varepsilon = 10^{-3}$ 52

3.25 Maximale Größe der Sweep Area für Ähnlichkeitsselbstverbund von CAD für verschiedene k und msl ($k = 0$: MSJ), $\varepsilon = 10^{-3}$ (log-Skala auf y-Achse) . . 53

3.26 Maximale Größe der Sweep Area für Ähnlichkeitsselbstverbund von CT für verschiedene k und msl ($k = 0$: MSJ), $\varepsilon = 10^{-3}$ (log-Skala auf y-Achse) . . 54

3.27 Maximale Größe der Sweep Area für Ähnlichkeitsselbstverbund von CM für verschiedene k und msl ($k = 0$: MSJ), $\varepsilon = 10^{-3}$ (log-Skala auf y-Achse) . . 54

3.28 Maximale Größe der Sweep Area für Ähnlichkeitsselbstverbund von CH für verschiedene k und msl ($k = 0$: MSJ), $\varepsilon = 10^{-3}$ (log-Skala auf y-Achse) . . 55

3.29 Ausführungszeit von GESS mit und ohne RPM für Ähnlichkeitsselbstverbund von CAD in Abhängigkeit von ε, $k = 3$ 56

3.30 Anzahl der Ergebnistupel von GESS mit und ohne RPM für Ähnlichkeitsselbstverbund von CAD in Abhängigkeit von ε, $k = 3$ 56

3.31 Anzahl der Ergebnistupel für Ähnlichkeitsselbstverbund von CAD (Stichprobe 1%) in Abhängigkeit von ε (log-log-Skala) 58

3.32 Anzahl der Ergebnistupel für Ähnlichkeitsselbstverbund von CT, CM und CH in Abhängigkeit von ε (log-log-Skala) 59

3.33 Vergleich von Kostenmodell und Experiment für Ähnlichkeitsselbstverbund von CAD (Stichprobe 1%, log-log-Skala) 60

3.34 Vergleich von Kostenmodell und Experiment für Ähnlichkeitsselbstverbund von CT (Stichprobe 10%, log-log-Skala) 61

3.35 Vergleich von Kostenmodell und Experiment für Ähnlichkeitsselbstverbund von CM (Stichprobe 10%, log-log-Skala) 62

3.36 Vergleich von Kostenmodell und Experiment für Ähnlichkeitsselbstverbund von CH (Stichprobe 10%, log-log-Skala) 62

3.37 Vergleich von Kostenmodell und Experiment für Ähnlichkeitsselbstverbund von CH unter Variation der Dimension von 1–64 (Stichprobe 10%, log-log-Skala) 63

4.1 Sortierbasierter Join zweier Datensätze R und S 67

4.2 Progressive Merge Join 69

4.3 Der Datentyp `SweepArea` 70

4.4 Die Methode `earlyJoinInitialRuns` 71

4.5 Die Methode `earlyJoinMergedRuns` 73

4.6 Equi Join UNI1 $\bowtie$ UNI2: Anzahl der Ergebnistupel als eine Funktion von I/O und Zeit 86

4.7 Spatial Join ST $\bowtie$ RR: Anzahl der Ergebnistupel als eine Funktion von I/O und Zeit 88

4.8 Similarity Join STPT $\bowtie_\varepsilon$ RRPT: Anzahl der Ergebnistupel als eine Funktion von I/O und Zeit 89

4.9 Similarity Join CAD1 $\bowtie_\varepsilon$ CAD2: Anzahl der Ergebnistupel als eine Funktion von I/O und Zeit 90

4.10 Similarity Join CAD1_Shuf $\bowtie_\varepsilon$ CAD2_Shuf: Anzahl der Ergebnistupel als eine Funktion von I/O und Zeit 92

4.11 Zeit, die benötigt wird, um die ersten 100 Ergebnistupel zu berechnen, Plane Sweep ST $\bowtie$ RR 93

4.12 Selektivitätsschätzung für den Similarity Join CAD1_Shuf $\bowtie_\varepsilon$ CAD2_Shuf (Hauptspeichergröße 1%) 95

4.13 Selektivitätsschätzung für den Similarity Join CAD1 $\bowtie_\varepsilon$ CAD2 (Hauptspeichergröße 1%, log-Skala auf y-Achse) 95

5.1 Integration der sortierbasierten Similarity Joins in XXL 107

5.2 Generische Implementierung des Algorithmus von Orenstein in XXL 108

5.3 Datenfluß und Parameter für Orenstein und GESS in XXL 111

5.4 Präzision von gebrochenen Zahlen im normierten Raum $[0; 1)$ 115

Tabellenverzeichnis

3.1 Beschreibung der in den Experimenten genutzten Datensätze 44

4.1 Die Join- und Löschprädikate sowie Sortierungen verschiedener Joinalgorithmen 78

4.2 Die in den Experimenten verwendeten Datensätze 85

4.3 Benötigte Laufzeit bzw. I/O-Operationen für einen Spatial Join von RR und ST (Hauptspeichergröße: 5%, 10%, 20%) 94

Abkürzungsverzeichnis

Symbol	Definition
B	Anzahl der Elemente pro Speicherseite
c	Konstante für Power Law
$\text{Code}_D(A)$	Code des Teilraums $A \in \text{Part}(D)$
D	$:= D^0$ beliebige Ausgangsmenge für rekursive Partitionierung
D_i^l	i-te Teilmenge von D^{l-1} $(l > 0)$ des l-ten rekursiven Partitionierungsschrittes
d	Dimension
DC	Anzahl der Distanzberechnungen für einen Similarity Join
δ_R	Projektion auf Attribute von R
ε	Anfragedistanz in jeder Dimension für Merkmalsvektor x
F	maximaler Fanin einer Mergeoperation
$f_{\text{mo}}(A)$	Anzahl der Mischoperationen für Menge A
$\bar{f}_{\text{mo}}(A)$	Anzahl der ‚mittleren' Mischoperationen für Menge A $(=f_{\text{mo}}(A)$ ohne letzten Merge)
f_{rp}	allgemeine Referenzpunktfunktion
$H(x)$	$= H_{\varepsilon,d}(x)$
$H_{\varepsilon,d}(x)$	d-dimensionaler Hypercube der Seitenlänge ε
IF	Verbesserungsfaktor von GESS gegenüber MSJ bezüglich Distanzberechnungen (DC)
$IF_{\text{I/O-Kosten}}$	Verbesserungsfaktor von GESS gegenüber MSJ bezüglich I/O-Kosten
k	Anzahl der Hyperebenen, die ein Hypercube maximal schneiden darf
$K_\varepsilon(u, S)$	Teilmenge aller Punkte aus S, die einen Abstand nicht größer als ε vom Punkt u haben (Kugelanfrage)
l	Rekursionsstufe der rekursiven Partitionierung
M	Verfügbarer Hauptspeicher in Anzahl der Elemente
m	$= \lfloor M/B \rfloor$, Verfügbarer Hauptspeicher in Seiten

weiter auf der nächsten Seite

Fortsetzung

Symbol	Definition
M_{Sweep}	Anzahl der Elemente in der Sweep Area
mL	Maximale Rekursionsstufe der rekursiven Partitionierung
msl	Maximale Aufteilungsebene
N	Anzahl der Elemente einer Eingabemenge
n	$= \lceil N/B \rceil$, Anzahl der Speicherseiten von N
P	Prädikat
$\mathcal{P}$	Ergebnistupelexponent
p_l	bedingte Wahrscheinlichkeit, daß ein Hypercube H eine Hyperebene der Ebene l trifft
$P(X_l > k)$	bedingte Wahrscheinlichkeit mehr als k Hyperebenen in Ebene l zu treffen
$P_{NC}(X_l = q)$	nichtbedingte Wahrscheinlichkeit genau q Hyperebenen in Ebene l zu treffen
$P_{NC}(X_l > k)$	nichtbedingte Wahrscheinlichkeit mehr als k Hyperebenen in Ebene l zu treffen
$P_{join}(u, v)$	binäres Joinprädikat
$P_{rm}(u, v)$	binäres Löschprädikat
$P_{\varepsilon}(u, v)$	binäres Distanzprädikat (spezielles Joinprädikat)
Part(D)	Rekursive Partitionierung von D
R, S	Name der Eingabemengen
R', S'	Sortierte Folge von R bzw. S
$\|R\|, \|S\|$	Anzahl der Seiten in R (bzw. S)
$\|\|R\|\|, \|\|S\|\|$	Anzahl der Elemente in R (bzw. S)
R_i, S_i	i-te Teilmenge von R (bzw. S)
$\hat{R}, \hat{S}$	Teilmengen von R bzw. S
$\hat{R}', \hat{S}'$	Sortierte Folge von $\hat{R}$ bzw. $\hat{S}$
$\hat{r}_l$	Schätzer für die Anzahl der Hypercubes in Ebene l
$\hat{r}_{>l}$	Schätzer für die Anzahl der Hypercubes, die die Ebene l passieren
rp	Referenzpunkt
U	Einheitskubus: $U = [0, 1)^d$
V	Volumen eines Hypercube

weiter auf der nächsten Seite

Fortsetzung	
Symbol	Definition
w	Replikationsfaktor
x	Merkmalsvektor
x_i	Komponente von x in Dimension i
Z	Anzahl der Ergebnistupel

Index

B, 3
D, 21
L_p-Metrik, 4
R, 3
S, 3
Z, 3
$\text{Code}_{D^l}(A)$, 22
$\text{Part}(D^l)$, 21
k, 25
m, 3
msl, 27
rp, 28
Ähnlichkeitsselbstverbund, 5
Ähnlichkeitsverbund, 1, 5

Aufteilungsstrategie, 25

Blockieren, 7
blockierend, *siehe* Blockieren
Build Input, 11

Cloudscape, 99
Clustering, 2
`Comparator`, 99
continuous queries, 76, 104
curse of dimensionality, 6
`Cursor`, 101

Datenfenster, 12, 76
Design Pattern, 99
Dimension, 3
Direkter Teilraum, 21
Distanzmaß, 4
Distanzprädikat, 5
Duplikatvermeidung, 80

`earlyJoinInitialRuns`, 71
`earlyJoinMergedRuns`, 73
Ebene, 22
Ebenendatei, 16, 23, 24, 27
Einheitswürfel, 4
Equi Join
    Beispiel, 12
Ergebnistupel, 3
Ergebnistupelexponent, 58
Externspeicher, 3
Externspeicherverfahren, 8

Factory-Method, 99
Feature Vector, *siehe* Merkmalsvektor
Filter, 5
Filterschritt, 5
Fraktale Dimension, 6, 58
`Function`, 99, 102
Funktionale Programmierung, 102

GESS, 24

Hashfunktion, 11
Hashjoin, 11
Hauptspeicher, 3
Hausdorff Dimension, *siehe* Fraktale Dimension

I/O-Modell, 3

I/O-Operation, 3
Index, 8
Inner Join, 74
`Iterator`, 101

Java, 99
Join, *siehe* Verbund
Joinprädikat, 70, 77

Kugelanfrage, 1

Löschprädikat, 70, 72, 77
Level-File, *siehe* Ebenendatei
Lexikographische Ordnung, 22

Massenanfrage, 11
Mehrschrittverarbeitung, 5
Merge, *siehe* Vermischen
Merkmalsvektor, 3
    Komponente von, 3
Methode, 4
MSJ, 23

Nested-Loops Join, 10
    Indexed, 11
    seitenweises, 10
Notation, 3

Obermenge, 5
ONC-Paradigma, 99
Operatorbaum, 8
ORE, 22
Orenstein, 22
Outer Join, 74
Outlier, 3

package, *siehe* Paket
pair-count exponent, *siehe* Ergebnistupel-exponent
Paket, 100
Partitionierung
    n-äre, 21
    Quadtree, 23
    rekursive, 21
    Zuordnungsstrategien, 22
`PeekIterator`, 101
Power Law, 57
Prädikat
    n-äres, 4
    `split_erlaubt`, 25
Präfix, 22
Präfixeigenschaft, 22
`Predicate`, 99, 103
Probe Input, 11
`PushIterator`, 104

Quadtree, 14
`Queue`, 99, 104, 105
Quicksort, 67

R-Baum, 14
Referenzpunkt, 28
Referenzpunktmethode, 24, 28
Replacement Selection, 67
Replikat, 25
Replikation, 22
Replikationsalgorithmus, 24, 25
Replikationsfaktor, 37
RPM, *siehe* Referenzpunktmethode

Selbstverbund, 4, 76
Self Join, *siehe* Selbstverbund
Semijoin, 74
Similarity Join, *siehe* Ähnlichkeitsverbund
Sort-Merge Join
    Grundidee, 12
    Ordnung, 12

Speichermodell, 3
Speicherseite, 3
Stack, 22
Strategy Pattern, 103
Sweep Area, 12
    Klassendefinition, 70
Sweep Line, 12

unendliche Datenströme, *siehe* continuous
        queries

Verbund, 4
Verfeinerungsschritt, 6
Vermischen, 13

Wurzelregel, 16

Z-Ordnung, 22

Lebenslauf

Jens-Peter Dittrich wurde am 20. Mai 1972 in Gronau (Leine) geboren. Nach der Schulzeit in Alfeld (Leine) und Abitur (1991), leistete er Zivildienst beim DRK Alfeld. 1992 folgte ein 9 monatiger Aufenthalt in Portugal (Sprachkurse, Sprachstudium in Coimbra). Im Sommersemester 1993 begann er sein Studium an der Universität Göttingen (Englisch/Französisch/Portugiesisch).

Zum Wintersemester 1993 wechselte er an die *Philipps-Universität Marburg* und begann das Studium der Informatik (Diplom) mit Nebenfach Medienwissenschaften. Im Frühjahr 1996 legte er das Vordiplom, im März/April 1998 die Diplomprüfungen ab. Studienschwerpunkte waren Software Engineering, Graphik-Programmierung, Datenbanksysteme und Java. Seine Diplomarbeit behandelte ein Thema aus dem Bereich Geodatenbanksysteme (*,,Implementierung und Vergleich von Verfahren zur Berechnung des räumlichen Verbunds“*) und wurde von Prof. Dr. Bernhard Seeger, AG Datenbanksysteme, Universität Marburg, betreut. Im Februar 1999 schloß er sein Studium mit dem Titel *Diplom-Informatiker* ab.

Im März 1999 begann Jens-Peter Dittrich seine Arbeit in der Arbeitsgruppe von Prof. Dr. Bernhard Seeger im Rahmen des von der Deutschen Forschungsgemeinschaft geförderten Vorhabens ,,Verarbeitung von Massenanfragen in Geo- und Zeitdatenbanken“. Seine Forschungsschwerpunkte liegen im Bereich generische Joinalgorithmen, flexible Software-Komponenten und generische Bibliotheken, Data Mining sowie Design Patterns.