UNIVERSITÄT PASSAU

Diplomarbeit

Methoden zur Berechnung von besten Zuordnungen von Objekten zweier Datenströme

Richard Kuntschke
Neuburgerstr. 19a
94032 Passau

Erstgutachter:	Prof. Alfons Kemper, Ph. D.
Zweitgutachter:	Prof. Dr. Burkhard Freitag
Betreuer:	Dipl.-Inf. Bernhard Stegmaier

Lehrstuhl für Dialogorientierte Systeme
Fakultät für Mathematik und Informatik
Universität Passau

Zusammenfassung

Diese Arbeit beschäftigt sich mit Verfahren zur Bestimmung von besten Zuordnungen von Objekten bezüglich eines gegebenen Vergleichsmaßes. Dazu werden Paare von Objekten zweier Datenquellen gebildet und anhand des Vergleichsmaßes miteinander verglichen. Am Ende sollen schließlich genau die Paare von Objekten erzeugt werden, die bezogen auf die verwendete Vergleichsordnung von keinem anderen Objektpaar dominiert werden. Erschwerend kommt hinzu, dass die zu betrachtenden Objekte Datenströmen entstammen, die mitunter auch unendlich sein können. Dies müssen die zur Berechnung eingesetzten Algorithmen adäquat berücksichtigen. Für die Ermittlung bester Zuordnungen von Elementen zweier Datenströme wird die neue Klasse der Bestmatch-Join Operatoren eingeführt. Der Bestmatch-Join stellt im Rahmen dieser Arbeit das zentrale Verfahren zur Berechnung von besten Zuordnungen dar.

Grundsätzlich sind unter den gegebenen Bedingungen nur approximative Verfahren anwendbar, sofern keine zusätzlichen Voraussetzungen gefordert und eingehalten werden. Um korrekte Ergebnisse erzeugen zu können, müssen die von den Datenströmen gelieferten Eingabedaten bestimmte Eigenschaften besitzen, die von den Berechnungsverfahren ausgenutzt werden können. Dabei handelt es sich etwa um die Einhaltung einer vorgegebenen Reihenfolge, z. B. durch die Sortierung der Datenelemente nach den Werten eines ausgewählten Attributs. Außerdem sind Restriktionen zu formulieren, welche die Menge der für das Ergebnis interessanten Kombinationen von Elementen der beiden Eingaben einschränken. Dies bewirkt, dass nicht alle möglichen Paare von Elementen gebildet und betrachtet werden müssen.

Zur Berechnung korrekter Ergebnisse auf der Basis der oben genannten Voraussetzungen wird der Bestmatch-Join mit Einschränkungen vorgestellt und eingesetzt. Weiterhin werden Methoden zur Bestimmung des Ergebnisses dieses Operators behandelt, die auch auf unendlichen Datenströmen stets korrekte Resultate erzeugen.

Zur Bewertung der Leistungsfähigkeit der neuen Operatoren und der ihnen zugrunde liegenden Algorithmen wurde ein Prototyp implementiert und unter Leistungsgesichtspunkten analysiert. Der Prototyp ermöglicht dabei den Vergleich einer zahlreiche Nachteile aufweisenden naiven Implementierung mit einem deutlich bessere Eigenschaften besitzenden fensterbasierten Algorithmus. Dieser fensterbasierte Ansatz lässt neben der Berechnung korrekter Ergebnisse auf mitunter unendlichen Datenströmen auch die Erzeugung eines kontinuierlichen Stroms von Ausgabedaten aus zwei kontinuierlichen Eingabedatenströmen zu, d. h. er besitzt die wichtige Eigenschaft der Pipeline-Fähigkeit.

Danksagung

Vorab ein Dankeschön an alle, die mit ihrer Unterstützung dazu beigetragen haben, dass diese Diplomarbeit entstehen konnte. Insbesondere gilt mein Dank Herrn Prof. Alfons Kemper, Ph. D., der durch zahlreiche Anregungen die Entstehung dieser Arbeit maßgeblich mitbeeinflusst hat, sowie meinem Betreuer Herrn Dipl.-Inf. Bernhard Stegmaier, der sich durch meine wiederholten Besuche in seinem Büro und die dabei vorgetragenen Fragen und entstandenen Diskussionen nie aus der Ruhe bringen ließ und mir immer mit Rat und Tat zur Seite stand. Ebenfalls Dank an Herrn Prof. Dr. Burkhard Freitag, der sich bereit erklärt hat, die Rolle des Zweitgutachters zu übernehmen.

Besondere Erwähnung verdienen auch Herr Thomas Zimmermann, der in seiner unnachahmlichen Art auch diesmal wieder in technischen Fragen, insbesondere zu LaTeX und Gnuplot, sehr hilfsbereit und hilfreich war, und Herr Paul Holleis, der es unter anderem geschafft hat, dass das Mausrad im XEmacs funktioniert.

Schließlich nicht unerwähnt bleiben sollen meine Eltern, die es mit ihrer uneingeschränkten Unterstützung möglich gemacht haben, dass ich mich jederzeit mit voller Aufmerksamkeit meinem Studium widmen konnte.

Diese Diplomarbeit wurde mit dem Textsatzsystem LaTeX 2_{ε} unter Einbeziehung verschiedener Zusatzpakete erstellt. Die verwendeten Diagramme wurden mit Gnuplot erzeugt. Die Bilder entstanden mit dem Zeichenprogramm Xfig, die Entwurfsdiagramme mit Together 6.0 von TogetherSoft.

Die Implementierung des in der Arbeit vorgestellten Prototyps erfolgte in der Programmiersprache Java unter Verwendung des JDK 1.4.0 von Sun Microsystems, Inc. Als Editor kam der XEmacs zum Einsatz.

Inhaltsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Mit dem Beginn des Internet-Zeitalters in den neunziger Jahren des vergangenen Jahrhunderts erlangten Unternehmen wie Privatanwender gleichermaßen innerhalb relativ kurzer Zeit Zugriff auf ein weltumspannendes Datennetz, das bis dahin dem Militär und den Universitäten vorbehalten war. Damit begann eine Entwicklung, die weg von autarken Einzelplatzrechnern und kleinen lokalen Netzwerken hin zu einem globalen Verbund von Rechnern führte, der Informationsaustausch über Ländergrenzen und Kontinente hinweg ermöglicht. Die voranschreitende Globalisierung brachte multinationale Konzerne hervor, die Niederlassungen in den verschiedensten Ländern der ganzen Welt unterhalten. Diese Veränderungen hatten natürlich auch maßgeblichen Einfluss auf die Informatik und den Datenbanksektor. Verteilte Datenbanken haben durch das Bedürfnis der Unternehmen, lokale Datenbestände der einzelnen Niederlassungen dem Gesamtkonzern verfügbar zu machen und dabei stets hochaktuell zu sein, zunehmend an Bedeutung gewonnen.

1.1 Motivation

The network is the computer. Diese Aussage bringt wie wohl keine andere zum Ausdruck, welche Bedeutung Netze in der heutigen Welt der Informatik haben. Netze bedeuten immer auch Datenströme, die durch diese Netze fließen. Einige Zukunftsvisionen beinhalten gar die Vorstellung, Daten würden in Zukunft nicht mehr auf lokalen Datenträgern wie Festplatten gespeichert, sondern strömten kontinuierlich durch ein Datennetz. Wer bestimmte Daten benötigt, kann sich in diesem Szenario einfach in das Netz einloggen und die gewünschten Informationen aus dem kontinuierlichen Strom herausfiltern. Vor diesem Hintergrund sind Operatoren gefragt, welche die Daten aus den Datenströmen nach bestimmten Kriterien filtern und verarbeiten.

Von besonderem Interesse sind dabei Operatoren, die nicht nur auf einem Datenstrom arbeiten, sondern die Daten zweier oder mehrerer Datenströme miteinander verbinden. Dies ist vergleichbar mit einem Join, der die Inhalte zweier Datenbankrelationen unter Berücksichtigung gegebener Prädikate verbindet. Oftmals will oder kann man dabei aber keine exakten Angaben über die Bedingungen machen, die von den Joinpartnern erfüllt werden müssen, damit sie sich für das Joinergebnis qualifizieren. Vielmehr hätte man gern Paare von solchen Objekten im Ergebnis, die bezüglich eines gegebenen Vergleichsmaßes bestmöglich zueinander passen, also von keiner anderen möglichen Kombination von Objekten dominiert werden. Die Berechnung solcher bester Zuordnungen von Objekten zweier Datenströme stellt den Kern dieser Arbeit dar.

Ein praktisches Beispiel für ein Anwendungsszenario liefert der Arbeitsmarkt. Hier kommt es darauf an, eine möglichst gute Zuordnung von Arbeitssuchenden zu freien Arbeitsstellen zu finden. Eine Stelle wird mit dem Arbeiter besetzt, dessen Profil am besten von allen auf das Profil der jeweiligen Stelle passt. Das Profil kann dabei mehrere Aspekte umfassen. Bei einem Arbeitnehmer sind beispielsweise Qualifikation, Alter und Gehaltsvorstellung denkbare Attribute. Auf Seiten der Arbeitsstelle kann man entsprechend Qualifikationsanforderung, Anforderung an die Erfahrung oder die Belastbarkeit eines Bewerbers und Gehaltsvorgabe des Arbeitgebers nennen. Da es in der Regel keinen Arbeiter gibt, der alle Anforderungen der zu besetzenden Stelle exakt erfüllt, sind hier Verfahren, die auf exakte Übereinstimmung prüfen, im Allgemeinen nicht anwendbar. Stattdessen wählt man denjenigen Arbeitnehmer aus, der in Bezug auf alle Aspekte die beste Übereinstimmung mit dem Arbeitsplatz hat. Allerdings ist eine eindeutige Entscheidung hier unter Umständen nicht möglich. Dies ist dann der Fall, wenn es einen Arbeiter gibt, der bezüglich eines Attributs am besten zu der betreffenden Arbeitsstelle passt, und einen anderen, der in einem anderen Attribut die beste Übereinstimmung aufweist. Die beiden Arbeiter sind dann unvergleichbar und müssen beide als Ergebnis der Berechnung ausgegeben werden. Hier muss schließlich von Hand eine Entscheidung für einen Kandidaten gefällt werden. Solche Fälle lassen sich allerdings durch geeignete Einschränkungen auf den Attributen reduzieren. Obiges Beispiel lässt sich analog auch anwenden auf das Problem der innerbetrieblichen Arbeitsteilung, also der Verteilung der Mitarbeiter eines Betriebes auf die Aufgaben im Betrieb, für die sie am besten geeignet sind.

Datenströme kommen nun ins Spiel, wenn die Daten der Arbeitssuchenden und Arbeitsstellen nicht zentral gespeichert, sondern über mehrere Datenbanken verteilt sind, beispielsweise in regional verteilten Datenbanken des Arbeitsamtes und der Betriebe. Dann müssen die Informationen über ein Netz zu einem Ort transportiert werden, an dem die Berechnung der besten Zuordnungen von Arbeitnehmern zu Arbeitsplätzen erfolgt.

Abbildung 1.1 visualisiert eine einfache Ausprägung des obigen Anwendungsszenarios. Der Bewerber John ist für beide dargestellten Firmen von Interesse, während die Bewerberin Susan nur für die Firma Enrun, Inc. in Frage kommt. Das liegt daran, dass Susan in Bezug auf ComWorld, Inc. von John dominiert wird, da dieser die Qualifikationsanforderung exakt und die Alters- und Gehaltsvorgaben besser erfüllt als Susan. In Bezug auf Enrun, Inc. hingegen sind Susan und John unvergleichbar, da Susan zwar im Gegensatz zu John die Qualifikationsanforderung genau erfüllt, dieser dafür aber den Altersvorstellungen des Unternehmens eher und dessen Gehaltsvorstellungen sogar exakt entspricht. Der Bewerber Bill ist hingegen für keinen der beiden Arbeitgeber interessant, da er im Vergleich zu John und Susan sowohl überqualifiziert als auch zu alt ist und zudem zu hohe Gehaltsvorstellungen hat.

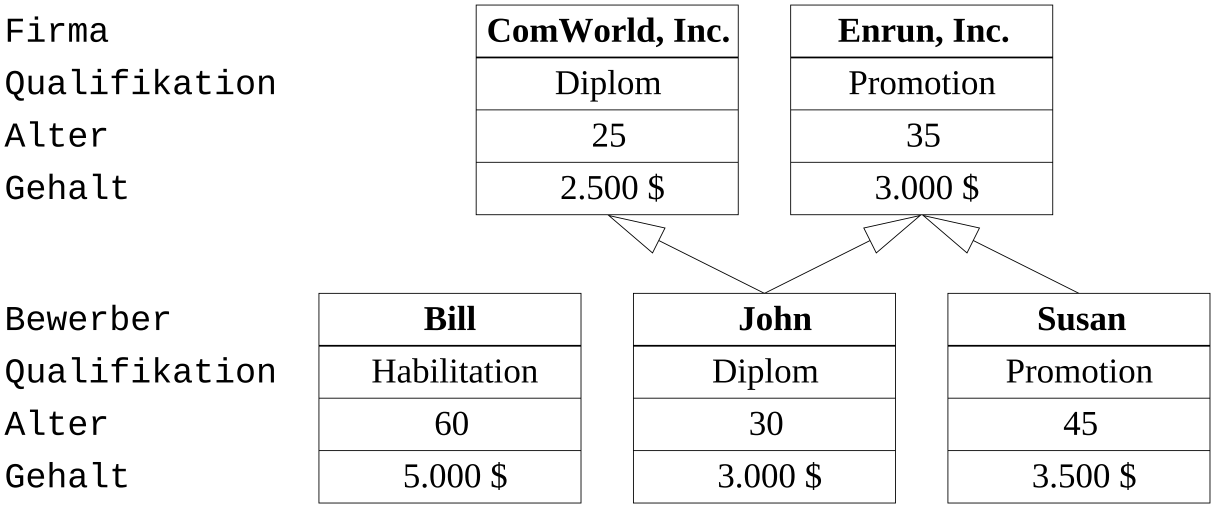


Abb. 1.1: Beispiel für ein mögliches Anwendungsszenario

1.2 Zielsetzung

Das Ziel dieser Arbeit besteht darin, eine Methode zur Berechnung von besten Zuordnungen von Objekten zweier Datenströme vorzustellen, die zum einen möglichst effizient ist und zum anderen korrekte Ergebnisse auf Datenströmen erzeugt, ohne zu blockieren. Es soll also möglich sein, aus zwei kontinuierlichen Eingabedatenströmen einen kontinuierlichen Strom von Ausgabedaten zu generieren. Da Datenströme unendlich sein können, sind hierfür im Allgemeinen nur approximative Verfahren einsetzbar. Die Berechnung korrekter Ergebnisse würde einen blockierenden Ansatz erfordern, der beide oder zumindest eine der beiden Eingaben bereits vor Beginn der Ergebnisberechnung vollständig einliest. Dieser komplette Überblick über die zu verarbeitenden Daten ist nötig, um alle relevanten Kombinationen von Datenelementen bilden und vergleichen zu können. Bei unendlichen Datenströmen ist dies aber nicht möglich. Hier muss man sich stattdessen mit approximativen Resultaten, die auf der Basis des bis zum Zeitpunkt ihrer Generierung gesammelten Wissens über die Eingabedaten erzeugt werden, begnügen. Da dieses Wissen aufgrund nicht komplett eingelesener Eingaben und der fehlenden Möglichkeit zur Speicherung unendlich großer Zustände in der Regel nicht vollständig ist, können in den approximativen Ergebnissen sowohl Ergebnisdaten fehlen als auch Datenelemente, die nicht zum korrekten Endergebnis gehören, enthalten sein. Zur Bestimmung korrekter Ergebnisse ist es daher erforderlich, dass die Eingabe- und die Ergebnisdaten bestimmten Anforderungen und Einschränkungen genügen. Worum es sich dabei handelt, wird in dieser Arbeit geklärt. Eine weitere Zielsetzung ist die Untersuchung der Leistungsfähigkeit der vorgestellten Verfahren anhand einer Prototyp-Implementierung.

Grundsätzlich können die hier behandelten Methoden auf unterschiedlichsten, z. B. relational, objektorientiert, objektrelational oder nach einem XML-Schema organisierten Eingabedaten arbeiten. Aus Gründen der einfacheren Darstellung wird im Kontext der Arbeit aber davon ausgegangen, dass die von den zu verarbeitenden Datenströmen gelieferten Daten aus einer Folge von Tupeln bestehen, die einem relationalen Schema genügen.

1.3 Überblick

Die Diplomarbeit beinhaltet insgesamt sieben Kapitel und einen Anhang bestehend aus zwei Kapiteln.

Im Anschluss an diese Einleitung folgt in **Kapitel 2: Verwandte Arbeiten** eine kurze Übersicht über mit dem Thema dieser Diplomarbeit verwandte Problemstellungen.

Kapitel 3: Bestmatch-Join auf Datenströmen beschreibt die Theorie des für die Lösung der Aufgabenstellung eingesetzten Bestmatch-Joins.

In **Kapitel 4: Algorithmen zur Berechnung des ϵ-LOBMJ** werden Algorithmen vorgestellt, mit deren Hilfe sich beste Zuordnungen berechnen lassen. Dabei liegt das Hauptaugenmerk auf einem effizienten Algorithmus zur Berechnung des Left-Outer-Bestmatch-Joins mit Einschränkungen, kurz ϵ-LOBMJ genannt, und auf einer naiven Variante, die als Vergleichsreferenz dienen soll.

Das folgende **Kapitel 5: Entwurf und Implementierung eines Prototyps** beschreibt den objektorientierten Entwurf und die Realisierung einer Prototyp-Implementierung der in Kapitel 4 vorgestellten Algorithmen.

Kapitel 6: Leistungsmessung geht auf die bei den Messungen verwendete Umgebung und die Messparameter ein und legt die Ergebnisse der Untersuchungen mit der Prototyp-Implementierung der Algorithmen dar.

Das abschließende **Kapitel 7: Zusammenfassung und Ausblick** bietet eine Zusammenstellung der gewonnenen Erkenntnisse und einen kurzen Ausblick auf weitere interessante Forschungsthemen in dem behandelten Gebiet.

Anhang A: Bedienung des Prototyps beinhaltet einen Überblick über die Bedienung des implementierten Prototyps inklusive Datengenerator, I/O-Testprogramm und Konfigurationsdateien, sowie über das Format der Eingabedateien.

Schließlich enthält **Anhang B: Entwurfsdiagramme** die Entwurfsdiagramme des objektorientierten Entwurfs aller fünf Pakete des Prototyps.

Kapitel 2

Verwandte Arbeiten

Dieses Kapitel stellt in einer kurzen Übersicht Arbeiten vor, die mit dem in dieser Diplomarbeit behandelten Thema verwandt sind. Dazu gehören klassische graphentheoretische Probleme ebenso wie neuere Entwicklungen auf dem Gebiet der Datenbanken.

2.1 Skyline

Der Skyline-Operator, vorgestellt in [BKS01], ist ein Operator zur Bestimmung derjenigen Tupel einer Menge von Tupeln, die bestimmte Vorgaben bestmöglich erfüllen. Dabei wird jede für die Berechnung des Ergebnisses interessante Dimension der Tupel separat betrachtet. Ein Tupel ist genau dann im Ergebnis enthalten, wenn es von keinem anderen Tupel der betrachteten Menge dominiert wird. Das ist der Fall, wenn es kein anderes Tupel in dieser Menge gibt, das in jeder relevanten Dimension mindestens genauso gut und in mindestens einer Dimension besser ist als das aktuell betrachtete Tupel. Nach welchem Kriterium entschieden wird, ob ein Tupel in einer bestimmten Dimension besser, schlechter oder genauso gut ist wie ein anderes Tupel, muss vorher festgelegt werden. Natürlich kann es hier auch unvergleichbare Tupel geben. Zwei Tupel sind genau dann unvergleichbar, wenn sie in allen relevanten Dimensionen gleich gut sind oder wenn ein Tupel in mindestens einer relevanten Dimension besser und in mindestens einer anderen schlechter ist als das andere Tupel. Unvergleichbare Tupel dominieren sich selbstverständlich nicht gegenseitig. Ein Beispiel für eine Anwendung dieses Operators ist die Suche nach denjenigen Hotels in einer Hotel-Datenbank, die möglichst billig sind und möglichst nah am Strand liegen. In diesem zweidimensionalen Fall gelangen alle Tupel in das Ergebnis, die nicht in beiden Dimensionen von einem anderen Tupel dominiert werden. Anders ausgedrückt ist ein Hotel genau dann im Ergebnis enthalten, wenn es kein anderes Hotel in der betrachteten Menge von Hotels gibt, das sowohl billiger ist als auch näher am Strand liegt. Abbildung 2.1 auf der nächsten Seite zeigt ein einfaches Beispiel einer Skyline für eine Menge zweidimensionaler Punkte. Ein Punkt ist hier in einer Dimension umso besser, je näher er bezüglich dieser Dimension am Ursprung des Koordinatensystems liegt. Die grau unterlegten Punkte sind untereinander unvergleichbar und bilden die Skyline, da sie verglichen mit jedem anderen Punkt der Menge in jeweils mindestens einer Dimension

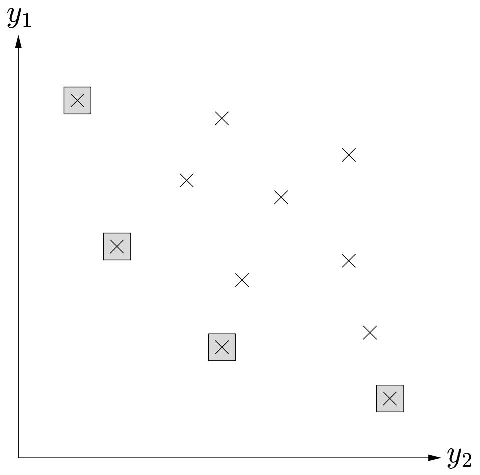


Abb. 2.1: Skyline einer Menge zweidimensionaler Punkte

besser sind. Alle anderen Punkte werden von mindestens einem Punkt der Menge dominiert, sind also in beiden Dimensionen schlechter als ein anderer Punkt und deshalb nicht im Ergebnis der Skyline-Berechnung enthalten.

Für die Implementierung des Skyline-Operators werden in [BKS01] mehrere Algorithmen vorgestellt und hinsichtlich ihrer Leistungsfähigkeit untersucht. Die beiden wesentlichen Gruppen von Algorithmen sind dabei Block-Nested-Loops Algorithmen und Divide & Conquer Algorithmen. Während der naive Ansatz zur Berechnung der Skyline eine Komplexität von $\mathcal{O}(n^2)$ besitzt, wobei n die Anzahl der betrachteten Tupel bezeichnet, erreichen die Block-Nested-Loops Algorithmen im besten Fall $\mathcal{O}(n)$, falls die Skyline in den Hauptspeicher passt. Im schlimmsten Fall liegen sie allerdings ebenfalls bei $\mathcal{O}(n^2)$. Die Divide & Conquer Algorithmen besitzen eine Komplexität von $\mathcal{O}(n \cdot (\log n)^{d-2}) + \mathcal{O}(n \cdot \log n)$. Dabei steht d für die Anzahl der Dimensionen der Skyline. Die Wahl eines besten Algorithmus ist vom jeweiligen Anwendungsszenario abhängig. In [BKS01] werden daher auch Empfehlungen gegeben, unter welchen Bedingungen welche Klasse und welche Ausprägung der vorgestellten Algorithmen zu bevorzugen ist. Eine grundlegende Arbeit, auf der die Skyline-Arbeiten aufbauen und die den dort benutzten Divide & Conquer Algorithmus eingeführt hat, ist [KLP75]. Weiterhin beschreibt [KRR02] Verfahren zur Online-Berechnung der Skyline.

Die größte Gemeinsamkeit zwischen der Berechnung der Skyline und den in dieser Arbeit beschriebenen Verfahren besteht in der Art und Weise, wie die Tupel bezüglich der Dominanz verglichen werden. Die separate Betrachtung aller relevanten Dimensionen und die sich daraus ergebende Möglichkeit des Auftretens unvergleichbarer Tupel sind hier die wichtigsten Übereinstimmungen. Die Standard-Algorithmen zur Berechnung der Skyline sind allerdings blockierend und deshalb für die Bearbeitung von Datenströmen nicht geeignet. Außerdem muss für die Anwendung der Skyline-Algorithmen zur Bestimmung der besten Zuordnungen von Objekten zweier Eingaben zunächst das Kreuzprodukt der Eingabedaten berechnet und dann darauf die Skyline ermittelt werden. In der Praxis ist das aber im Allgemeinen zu aufwendig. Deshalb wird in dieser Arbeit ein anderer Ansatz verfolgt.

2.2 Matching

Matching ist ein klassisches graphentheoretisches Problem. Ein Matching eines Graphen ist definiert als eine Teilmenge der Kantenmenge des Graphen mit der Eigenschaft, dass kein Knoten des Graphen mit mehr als einer Kante des Matchings inzident ist. Meistens sucht man ein *maximales* Matching, also ein Matching mit maximaler Anzahl an Kanten. Ein Matching heißt *perfekt*, wenn jeder Knoten des Graphen mit genau einer Kante des Matchings inzident ist. Es gibt im Wesentlichen vier Varianten von Matchings[1]. Diese beruhen auf der Art des Graphen – bipartit oder allgemein – und darauf, ob die Kanten des Graphen gewichtet oder ungewichtet sind. Bipartite Graphen zeichnen sich dadurch aus, dass sich die Menge ihrer Knoten in zwei disjunkte Knotenmengen aufteilen lässt, so dass die Kanten des Graphen nur Knoten aus der einen Knotenmenge mit Knoten aus der jeweils anderen Knotenmenge verbinden. Allgemeine Graphen unterliegen dieser Einschränkung hingegen nicht. Beim ungewichteten Matching in bipartiten Graphen wird wie bereits erwähnt meist eine maximale Teilmenge der Kanten des Graphen gesucht, so dass keine zwei Kanten des Matchings mit demselben Knoten inzident sind. Die Variante des bipartiten gewichteten Matchings wählt die Kanten des Matchings dabei so aus, dass die Summe der Kantengewichte der zum Matching gehörenden Kanten maximal oder minimal wird, oft unter der Nebenbedingung, dass das Matching perfekt sein soll. Eine bekannte Anwendung dieses Verfahrens ist das sogenannte Heiratsproblem, bei dem heiratswillige Männer und Frauen die beiden Knotenmengen bilden und anhand ihrer Präferenzen einander bestmöglich zugeordnet werden. Die erzeugte Zuordnung heißt *instabil*, wenn es mindestens zwei Personen gibt, die sich gegenseitig als Partner bevorzugen würden, im aktuellen Matching aber anderen Partnern zugeordnet sind. Anderenfalls heißt die Zuordnung *stabil*. Praktische Anwendung finden Methoden zur Berechnung des Matchings auf bipartiten gewichteten Graphen tatsächlich bei der zentralen Verteilung von Absolventen des Studiums der Medizin auf die Krankenhäuser in den USA. Auch hier können Absolventen und Krankenhäuser zunächst Präferenzen für zukünftige Arbeitsstellen bzw. Ärzte angeben, bevor die bestmögliche Zuordnung auf der Grundlage dieser Wünsche erfolgt.[2] Zur Berechnung des Matchings auf bipartiten Graphen existieren sowohl für den ungewichteten als auch für den gewichteten Fall relativ einfache Algorithmen, die polynomiale Laufzeiten besitzen. Sie sind z. B. in [Jun99] und [OW02] beschrieben. Abbildung 2.2 zeigt ein perfektes maximales Matching eines bipartiten ungewichteten Graphen. Die breiteren Kanten gehören zum Matching.

Das Matching-Problem lässt sich allerdings auch auf allgemeine Graphen erweitern. Die Problemlösung wird in diesem Fall etwas komplizierter, aber auch hierfür existieren effiziente Lösungsverfahren. Für den ungewichteten Fall gibt es einen polynomialen Algorithmus, der eine Erweiterung und Verallgemeinerung des Standard-Algorithmus für bipartites ungewichtetes Matching darstellt und somit dessen Lösung quasi als Sonderfall enthält. Dieser Algorithmus aus dem Jahr 1965 ist bekannt unter dem Namen Blütenalgorithmus von Edmonds und wird neben [Jun99] unter anderem auch in [AMO93] und

[1]Matchings werden in der deutschsprachigen Literatur oft auch als Zuordnungen bezeichnet, Matching-Probleme entsprechend auch als Zuordnungsprobleme.

[2]Siehe hierzu auch [Sed88].

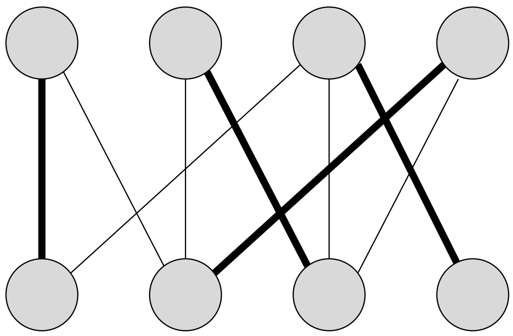


Abb. 2.2: Maximales Matching auf einem bipartiten ungewichteten Graphen

[Edm65] vorgestellt. Schwieriger ist das Finden eines allgemeinen gewichteten Matchings. Doch auch für dieses Problem gibt es effiziente Verfahren, wie z. B. in [OW02] erklärt wird. Bemerkenswert ist dabei, dass sowohl im bipartiten als auch im allgemeinen Fall die Lösung des gewichteten Matching-Problems zwar konzeptionell schwieriger ist als die des ungewichteten Matching-Problems, dass sie aber dennoch keine höhere algorithmische Komplexität besitzt. Alle vier Matching-Varianten sind mit effizienten polynomialen Algorithmen lösbar, die höchstens in $\mathcal{O}(n^3)$ liegen, wobei n die Anzahl der Knoten des zugrunde liegenden Graphen bezeichnet.

Doch sind auch die Matching-Algorithmen für Berechnungen auf Datenströmen nicht einsetzbar. Außerdem werden sie der Anforderung, im Falle mehrdimensionaler Präferenzen jede Präferenz separat zu betrachten, nicht gerecht. Stattdessen funktionieren die Matching-Verfahren nur auf eindimensionalen Präferenzen. Bei mehreren Dimensionen müssen sie die besten Zuordnungen zweier Objekte in der Regel dadurch bestimmen, dass sie basierend auf den vorliegenden Präferenzen anhand einer Bewertungsfunktion Gewichtungen errechnen und mittels dieser Gewichtungswerte, welche den Kantengewichten des Graphen entsprechen, die Zuordnung vornehmen.

2.3 Nearest-Neighbor und Closest-Point Probleme

Aus dem Bereich der algorithmischen Geometrie stammen die Nearest-Neighbor und Closest-Point Probleme. Dabei geht es im Wesentlichen um die Bestimmung von Punkten, die geometrisch bezüglich einer bestimmten Metrik, wie beispielsweise der euklidischen oder der Manhattan-Metrik, den geringsten Abstand voneinander haben. Es gibt sehr viele Varianten von Problemen dieser Art. So kann es z. B. interessant sein, das Paar derjenigen beiden Punkte einer Menge zu bestimmen, die am nächsten von allen zusammenliegen. Diese Ausprägung des Nearest-Neighbor Problems wird auch als Closest-Pair Problem bezeichnet. Darüber hinaus ist es in bestimmten Anwendungsfällen eventuell auch nötig, zu einem gegebenen Punkt denjenigen Punkt einer Menge zu finden, der dem gegebenen Punkt am nächsten liegt, oder zu jedem Punkt der Menge den nächstgelegenen Punkt aus dieser oder einer anderen Menge zu bestimmen. Es existieren vielfältige praktische Anwendungen für Algorithmen zur Lösung von Nearest-Neighbor Problemen, etwa bei Routenplanern und Navigationssystemen oder auch in Grafikprogrammen.

Nearest-Neighbor und Closest-Point Probleme sind in der algorithmischen Geometrie schon lange bekannt und relativ gut erforscht. Entsprechend effiziente Algorithmen gibt es zur Lösung dieser Probleme. Während naive Verfahren für das Closest-Pair Problem quadratische Komplexität in der Zahl der zu betrachtenden Punkte aufweisen, gibt es höher entwickelte Methoden, die eine Komplexität von $\mathcal{O}(n \cdot \log n)$ besitzen, wobei n die Zahl der Punkte bezeichnet, oder gar in linearer Zeit ablaufen. Näheres hierzu findet sich in [SU00]. Weiterhin beschreibt [HAK00] die Bestimmung von nächsten Nachbarn in höherdimensionalen Räumen. Hierbei werden im Gegensatz zu anderen Arbeiten nicht alle Dimensionen gleich behandelt, sondern anhand eines Qualitätskriteriums die für den jeweiligen Anwendungsfall relevanten Dimensionen ausgewählt. In [Epp98] werden Datenstrukturen und Implementierungsvarianten vorgestellt, die eine effiziente Bestimmung der Lösung des Closest-Pair Problems erlauben. Schließlich behandelt [CP00] die Auswertung approximativer Nearest-Neighbor Anfragen in mehrdimensionalen Räumen.

Im Vergleich zu den in dieser Arbeit behandelten Verfahren besitzen die Methoden zur Lösung des Nearest-Neighbor Problems ähnliche Unterschiede, wie sie bereits beim Matching dargelegt wurden. Zum einen sind die gängigen Algorithmen zur Berechnung der nächsten Nachbarn in der Regel nicht auf Datenströmen einsetzbar und zum anderen besteht das Ziel der im Folgenden vorgestellten Methoden darin, Datenpunkte anhand jeder einzelnen relevanten Dimension miteinander zu vergleichen, anstatt wie die Nearest-Neighbor Verfahren einen einzigen, mittels einer Metrik berechneten Abstandswert als Entscheidungskriterium zu verwenden. Deshalb wird eine Suche nach nächsten Nachbarn im Allgemeinen andere Ergebnisse erzeugen als beispielsweise die Berechnung der Skyline, die dieser Arbeit in dieser Beziehung näher steht.

2.4 Top N Anfragen

Ähnlich verhält es sich mit Top N Anfragen. Dabei handelt es sich um Datenbankanfragen, die nicht alle Tupel, die die angegebenen Prädikate der Anfrage erfüllen, ausgeben, sondern nur die N besten. Ein Beispiel dafür wäre eine Anfrage eines Managers, der die zehn Produkte aus seiner Abteilung sucht, die im letzten Quartal den größten Umsatz erzielt haben. Wenn es in der Abteilung insgesamt 100 Produkte gibt, dann ist ein relativ großer Teil der Produkte, genauer gesagt 90 Prozent, für diese Anfrage eigentlich uninteressant. Die Herausforderung besteht nun darin, das Wissen darüber, dass nur ein Teil des Ergebnisses gebraucht wird, bereits bei der Anfragebearbeitung gewinnbringend zur Effizienzsteigerung einzusetzen. Das bedeutet, dass man die Verschwendung von Ressourcen zur Berechnung nicht benötigter Ergebnistupel, im Beispiel etwa des elft- und zwölftbesten Produktes, vermeiden will.

In [CK97a] ist dargestellt, wie dies geschehen kann. Dazu wird eine SQL Erweiterung, der `STOP AFTER` Ausdruck, eingeführt. Weiter beschreibt die oben genannte Arbeit Implementierungsmöglichkeiten dieser Erweiterung durch die Einführung von Stop-Operatoren sowie eine konservative und eine aggressive Strategie zur Generierung von Auswertungsplänen. Dieselben Autoren greifen in [CK97b] ihre Vorschläge wieder auf und gehen auf Nachteile aktueller Datenbanksysteme bei der Auswertung von Top N Anfragen sowie auf

Verbesserungsmöglichkeiten ein. In [CK98] schließlich zeigen sie Möglichkeiten auf, wie die Effizienz der Berechnung solcher Anfragen weiter gesteigert werden kann.

Da Top N Anfragen ein anderes Problem als die Berechnung bester Zuordnungen darstellen, werden sie im Normalfall auch andere Ergebnisse erzeugen. Insbesondere hängt das Ergebnis einer Top N Anfrage natürlich stark von dem Parameter N ab. Einen vergleichbaren Einflussfaktor gibt es bei den Verfahren zur Bestimmung von besten Zuordnungen, auf die sich diese Arbeit konzentriert, nicht. Deshalb sind hier andere Wege zu verfolgen.

2.5 Similarity-Join

Similarity-Joins werden in Datenbanken in verschiedenen Bereichen eingesetzt, vor allem beim Data Mining oder bei der Ähnlichkeitssuche in Bilddatenbanken. Sie arbeiten auf zwei Mengen von Punkten eines multidimensionalen Vektorraums, die beispielsweise bei der Extraktion von sogenannten Feature-Vektoren entstehen. Jeder Punkt entspricht dann einem solchen Feature-Vektor, der eine skalare Repräsentation der Merkmale der ursprünglichen Daten, aus denen der Vektor extrahiert wurde, darstellt. Im Falle von Bilddatenbanken könnten das z. B. Farb- oder Helligkeitseigenschaften der einzelnen Bildpunkte sein. Der Similarity-Join kombiniert die Punkte der beiden Mengen derart, dass am Ende diejenigen Paare von Punkten ausgegeben werden, deren Abstand kleiner als eine vorgegebene Schranke ϵ ist.

Der Similarity-Join war in jüngerer Vergangenheit Gegenstand intensiver Forschungsarbeit. In [BK01] wird er eingehend unter Leistungsgesichtspunkten untersucht. Außerdem werden ein Kostenmodell und eine neue Index-Architektur vorgeschlagen und es wird versucht, den auftretenden Konflikt zwischen CPU- und I/O-Optimierung zu entschärfen. Weiterhin stellt [FTTF01] das sogenannte OMNI-Konzept zur Beschleunigung von Ähnlichkeitssuchen in höherdimensionalen Räumen vor und [LLL01] präsentiert ein Verfahren zur approximativen Berechnung von nächsten Nachbarn in Mengen mehrdimensionaler Punkte. Böhm, Braunmüller, Krebs und Kriegel schlagen in [BBKK01] einen neuen Algorithmus zur effizienteren Berechnung des Similarity-Joins auf großen Datenmengen vor. Ihr Ansatz basiert auf einer speziellen Sortierordnung der Datenpunkte, der Epsilon-Grid-Ordnung, die in einer leicht abgewandelten Version auch in dieser Arbeit eine Rolle spielt.

Außer dieser Ordnung, auf deren Variante in Abschnitt 4.3.1 eingegangen wird, hat das Gebiet der Similarity-Joins noch eine weitere Gemeinsamkeit mit dem zentralen Verfahren, das in den folgenden Kapiteln behandelt wird. Es handelt sich dabei um die Einschränkung auf eine vorgegebene ϵ-Umgebung. Diese wird in Abschnitt 3.2.2 über den Bestmatch-Join mit Einschränkungen, auch kurz ϵ-BMJ genannt, wieder aufgegriffen.

Kapitel 3

Bestmatch-Joins auf Datenströmen

Dieses Kapitel gibt eine Einführung in den Bestmatch-Join Operator und seine Verwendung auf Datenströmen. Der Bestmatch-Join ist die zentrale Methode zur Berechnung von besten Zuordnungen von Objekten zweier Datenströme, die in den restlichen Kapiteln der Arbeit im Wesentlichen behandelt wird. Zunächst soll dargestellt werden, was unter einem Datenstrom zu verstehen ist und welche Vergleichsordnungen für den Join eingesetzt werden können. Danach folgt die Definition des Bestmatch-Joins und seiner Varianten nach [KS02a] sowie eine Beschreibung seiner wichtigsten Eigenschaften. Schließlich werden zwei unterschiedliche Ansätze zur Berechnung des Bestmatch-Joins auf Datenströmen vorgestellt.

3.1 Einführung

Um Kombinationen von Datensätzen aus zwei verschiedenen Eingaben unter Berücksichtigung bestimmter Prädikate zu berechnen, bietet es sich an, einen Join-Operator zu verwenden. Die Charakteristik eines Joins ist es ja gerade, die Tupel zweier Eingaben, meistens Datenbankrelationen, nach vorgegebenen Prädikaten miteinander zu vergleichen und zu verbinden. Deshalb liegt es nahe, einen solchen Ansatz auch für die hier behandelte Problemstellung zu verwenden. Im Arbeitsmarkt-Beispiel aus der Einleitung bedeutet das einen Join der Daten der Arbeitssuchenden mit denen der Stellenangebote mit dem Prädikat, dass die gefundenen Zuordnungen von Arbeitnehmern zu Arbeitsstellen in den relevanten Attributen wie Qualifikation, Alter und Gehalt bestmöglich zueinander passen. Aufgrund der separaten Betrachtung der einzelnen Attribute und der daraus resultierenden Möglichkeit der Unvergleichbarkeit zweier Kombinationen ist es dabei durchaus möglich, dass einer arbeitssuchenden Person mehrere interessante Stellenangebote zugeordnet und dass umgekehrt zu einem Stellenangebot mehrere interessante Bewerber gefunden werden. Die Schwierigkeit ist nun, einen Join-Operator zu entwickeln, der auf möglicherweise unendlichen Datenströmen statt auf zur Berechnungszeit vollständig und unveränderlich vorliegenden Relationen arbeitet, dabei noch effizient ist und korrekte Ergebnisse erzeugt. Wie das in der Theorie aussehen kann ist Gegenstand dieses und des nächsten Kapitels. Während in diesem Kapitel der Operator und seine Varianten definiert werden, beschreibt das folgende Kapitel 4 Algorithmen zur Implementierung einer Variante des Operators. Dazu vorab ein Wort zur Notation. Eine Bezeichnung ähnlich der Form $1, \ldots, n$ oder $\{1, \ldots, n\}$ bedeutet in den gesamten folgenden Ausführungen immer alle Elemente i bzw. die Menge aller i mit den Eigenschaften $i \in \mathbb{N}$, $i \geq 1$ und $i \leq n$.

3.1.1 Datenströme

Bevor man sich mit Operatoren beschäftigen kann, die auf Datenströmen arbeiten, muss man sich zunächst einmal darüber im Klaren sein, was unter einem Datenstrom zu verstehen ist.

Definition 3.1 (Datenstrom) *Ein Datenstrom R ist eine geordnete Folge von Datenobjekten* $< r_1, r_2, \ldots, r_n >$ *mit* $n \in \mathbb{N}$. *Aus dem Datenstrom kann immer nur das jeweils nächste Datenobjekt in der Reihenfolge der Anlieferung durch den Strom gelesen werden. Nachdem ein Datenobjekt* r_i *gelesen wurde ist auf dem Datenstrom kein Zugriff mehr auf ein Datenobjekt* r_j *mit* $j \in \mathbb{N}$ *und* $j \leq i$ *möglich.*

Das bedeutet, dass nach dem Lesen eines Datenobjektes aus dem Datenstrom dieses und jedes zuvor gelesene Objekt des Stroms dem Strom nicht noch einmal entnommen werden kann. Die Datenobjekte müssen also entweder sofort verarbeitet oder zur späteren Verarbeitung gepuffert werden. Für n kann insbesondere auch $n = \infty$ gelten, was einen unendlichen Datenstrom bedeutet.

Die Reihenfolge der Datenobjekte eines Datenstroms hängt von der Sortierung des Stroms ab. Ein Datenstrom kann seine Objekte sowohl unsortiert als auch nach einer bestimmten Ordnung sortiert liefern.

Definition 3.2 (Sortierung) *Ein Datenstrom heißt sortiert nach einer Ordnung* $\trianglelefteq$, *falls gilt:*

$$\forall i, j \in \mathbb{N} : i < j \Leftrightarrow r_i \trianglelefteq r_j$$

Die Sortierung wird in Abschnitt 4.2 noch eine wichtige Rolle spielen.

Hellersteins Forderung nach einer Online-Verarbeitung von Datenströmen in [Hel97], welche mit den hier vorgestellten Verfahren schließlich erreicht werden soll, bedingt eine Reihe von Forderungen, die an die verwendeten Operatoren zu stellen sind. Die herausragendste dieser Forderungen ist, dass die Operatoren alle Ein- und Ausgaben als Datenströme verarbeiten bzw. liefern müssen. Unter der Annahme, dass die verwendeten Eingabedaten einer Datenquelle entstammen, die ihre Daten über das Internet oder ein anderes Netz anliefert, ist es ferner nicht möglich, für die Anfragebearbeitung geeignete Statistiken vorzuberechnen und zu speichern. Darüber hinaus ist es wünschenswert, den Benutzer entweder ständig oder zumindest in periodischen Abständen mit Updates, also mit aktuellen Ergebnissen seiner Anfrage, zu versorgen. Um diese Anforderung zu erfüllen, müssen die Operatoren während der Bearbeitung der Anfrage kontinuierlich Zwischenergebnisse erzeugen. Wie diese Zwischenergebnisse bezüglich ihrer Qualität zu beurteilen sind, hängt allerdings vom verwendeten Berechnungsalgorithmus ab. Während es sich bei den zwischenzeitlich generierten Resultaten auch um approximative Ergebnisse handeln kann, stellen sie bei den zentralen Algorithmen dieser Arbeit Teile des korrekten Endergebnisses dar. Sie bilden den kontinuierlichen Strom von Ausgabedaten und ergeben in ihrer Gesamtheit das vollständige Join-Ergebnis.

Grundsätzlich muss ein Operator, der die obigen Anforderungen erfüllt, uneingeschränkt pipelinefähig sein, d. h. er muss Teile des Ergebnisses bereits weiterreichen können, *bevor* er die gesamte Eingabe gelesen hat. Solche Operatoren nennt man *nicht-blockierend*. Hingegen heißen Operatoren, die zunächst die komplette Eingabe einlesen müssen, bevor sie auch nur einen Teil des Ergebnisses berechnen und ausgeben können, *blockierend* oder auch *Pipeline-Breaker*. Letztere Bezeichnung verweist darauf, dass derartige Operatoren an ihrer Verwendungsstelle die Vorteile der Pipeline-Berechnung zunächst einmal zunichte machen, wenn sie in einen Berechnungsplan eingeführt werden, der zuvor nur aus nicht-blockierenden Operatoren bestand. Das liegt daran, dass die komplette Eingabe vor einem solchen Operator aufgestaut werden muss, bevor er sein Ergebnis berechnen kann. Beispiele für nicht-blockierende Operatoren sind die Selektion und die Projektion. Diese können ihre Aufgaben sogar tupelweise erfüllen, so dass sie jedes Eingabetupel für sich bearbeiten und sofort weiterreichen können. Blockierende Operatoren sind z. B. die Sortierung und die Berechnung von Maxima oder Minima. Vor dem Hintergrund, dass Datenströme unendlich sein können, sind blockierende Operatoren für Berechnungen auf Datenströmen nicht anwendbar, da es aufgrund von Zeit- und Speicherplatzbeschränkungen unmöglich ist, einen unendlichen Datenstrom zur Verarbeitung komplett zwischenzuspeichern. Für diese Anwendungen sind daher nicht-blockierende Verfahren unumgänglich. Außerdem sind blockierende Operatoren nicht in der Lage, verlässliche Zwischenergebnisse während des Berechnungsvorgangs zu liefern. Solche Zwischenergebnisse, die generiert werden, bevor der Operator die gesamten Eingaben verarbeitet hat, sind immer mit Unsicherheiten behaftet. Dies beruht auf den folgenden beiden möglichen Situationen:

- Es kann sein, dass einzelne Ergebnisobjekte des korrekten Ergebnisses in einem Zwischenergebnis nicht vorhanden sind, da die zur Berechnung dieser Ergebnisobjekte benötigten Eingaben zum Zeitpunkt der Generierung des Zwischenergebnisses noch nicht verarbeitet worden waren. Dieser Punkt relativiert sich allerdings bei unendlichen Datenströmen, da hier zu keinem Zeitpunkt ein vollständiges Endergebnis existiert.

- Andererseits ist nicht auszuschließen, dass ein Zwischenergebnis Objekte enthält, die nicht Teil des korrekten Endergebnisses sind. Dies ist dadurch zu erklären, dass diese Datenobjekte zum Zeitpunkt der Generierung des Zwischenergebnisses auf der Grundlage der bis dahin verarbeiteten Eingabedaten zwar in das vorläufige Ergebnis aufgenommen wurden, bei der weiteren Verarbeitung der Eingaben aber durch bessere Ergebnisobjekte invalidiert werden.

Insgesamt können in von blockierenden Operatoren erzeugten Zwischenergebnissen also sowohl korrekte Ergebnistupel fehlen, als auch inkorrekte Tupel enthalten sein. Deshalb ist hier zwischen einer blockierenden Ausführung der Operatoren, die in jedem Fall korrekte Ergebnisse erzeugt aber nicht immer anwendbar ist, und einer nicht-blockierenden

Ausführung, die approximative Ergebnisse von mehr oder weniger guter Qualität generiert und immer funktioniert, zu unterscheiden.

Um die in der Realität bestehende Beschränkung des zur Verfügung stehenden Hauptspeichers adäquat berücksichtigen zu können, arbeiten alle hier betrachteten Join-Operatoren auf Datenfenstern. Sowohl die aktuellen Eingaben der Operatoren als auch deren gegenwärtige Zwischenergebnisse[1] werden in solchen Fenstern gespeichert.

Definition 3.3 (Datenfenster) *Es gibt größenbeschränkte und nicht größenbeschränkte Datenfenster. Das größenbeschränkte Datenfenster W_R^t einer Eingabe R zum Zeitpunkt t ist definiert durch $W_R^t \subseteq \{r_1, \ldots, r_i\}$, wobei r_i das letzte Datenobjekt von R bezeichnet, das bis zum Zeitpunkt t in W_R^t eingelesen wurde. Die Größe des Datenfensters ist durch $size_{W_R}^t$ beschränkt, es gilt also $|W_R^t| \leq size_{W_R}^t$. Diese maximale Anzahl an Datenelementen, die das Datenfenster aufnehmen kann, ist von der Zeit t abhängig, um z. B. bei mehreren Datenfenstern den gesamten zur Verfügung stehenden Hauptspeicher variabel auf die Fenster aufteilen zu können. Falls die Bedingung $i \leq size_{W_R}^t$ erfüllt ist, so tritt für obige Teilmengenbeziehung der Gleichheitsfall ein, es gilt also $W_R^t = \{r_1, \ldots, r_i\}$. Weiterhin gelten folgende Definitionen:*

$$min_{y_i}(W_R^t) := min\{r.y_i \mid r \in W_R^t\}$$

$$max_{y_i}(W_R^t) := max\{r.y_i \mid r \in W_R^t\}$$

Das nicht größenbeschränkte Datenfenster W_R einer Eingabe R ist analog zu dem größenbeschränkten Fenster W_R^t definiert, mit dem Unterschied, dass die Beschränkung der Anzahl der zu einem Zeitpunkt enthaltenen Datenelemente auf $size_{W_R}^t$ entfällt. Ein solches Datenfenster kann also theoretisch beliebig viele Datenobjekte enthalten. Es kann aber auch bestimmten andersartigen Beschränkungen unterliegen. Diese bedingen, dass nur solche Tupel aus $\{r_1, \ldots, r_i\}$ in dem Fenster W_R enthalten sein dürfen, welche die geltenden Einschränkungen nicht verletzen. Ein Beispiel für eine derartige Beschränkung ist die Limitierung eines Datenfensters auf Tupel, deren Werte für ein oder mehrere Attribute innerhalb vorgegebener Intervalle liegen.

Das größenbeschränkte Datenfenster W_R^t findet später bei der Berechnung des approximativen Bestmatch-Joins in Abschnitt 3.2.1 Verwendung. Das nicht größenbeschränkte Fenster W_R wird in Kapitel 4 im fensterbasierten Algorithmus zur Berechnung des eingeschränkten Left-Outer-Bestmatch-Joins eingesetzt.

3.1.2 Vergleichsordnungen

Bevor der Bestmatch-Join formal definiert wird, stellt dieser Abschnitt mögliche Vergleichsordnungen vor, mit denen der Operator arbeiten kann. Der Bestmatch-Join verwendet diese Ordnungen, um zwei Paare von Tupeln miteinander zu vergleichen. Im Folgenden wird davon ausgegangen, dass die Eingaben des Joins aus den beiden Relationen

[1] Die aktuellen Zwischenergebnisse eines Operators zu einem bestimmten Zeitpunkt entsprechen dem Zustand dieses Operators zu dem betreffenden Zeitpunkt.

R und S bestehen, die wie folgt aufgebaut sind:

$$R : \{[x_1 : type(x_1), \ldots, x_n : type(x_n), y_1 : type(y_1), \ldots, y_d : type(y_d)]\}$$

$$S : \{[y_1 : type(y_1), \ldots, y_d : type(y_d), z_1 : type(z_1), \ldots, z_m : type(z_m)]\}$$

Weiterhin gilt $d, m, n \in \mathbb{N}$. Ferner bezeichnen r und r' Tupel aus R, sowie s und s' Tupel aus S. Der Bestmatch-Join generiert aus Tupeln $r \in R$ und $s \in S$ Paare von Tupeln $(r \times s)$. Das Ergebnis der Berechnung des Bestmatch-Joins der beiden Relationen R und S enthält alle solchen Paare von Tupeln, die bezüglich der verwendeten Vergleichsordnung bestmöglich zueinander passen. Dabei sind für den Vergleich nur explizit als solche gekennzeichnete *Join-Attribute* oder *Join-Dimensionen* relevant. Die restlichen Attribute der Eingaberelationen enthalten Nutzdaten, die für die Join-Berechnung nicht von Interesse sind. In der hier verwendeten Notation sind die Attribute $y_1, \ldots, y_d$ die Join-Attribute der beiden Relationen. Entsprechend bezeichnet d die Anzahl der Join-Dimensionen. Selbstverständlich müssen korrespondierende Join-Attribute aus den beiden Relationen in Bezug auf den Join miteinander kompatibel sein, d. h. je zwei sich entsprechende Join-Attribute aus R und S müssen in der Regel den gleichen Typ haben. In obiger Definition der Schemata von R und S ist dies sichergestellt. Die übrigen Attribute $x_1, \ldots, x_n$ und $z_1, \ldots, z_m$ haben dagegen keinen Einfluss auf die Berechnung des Ergebnisses des Bestmatch-Joins. Sie enthalten Nutzdaten und unterliegen im Rahmen der Join-Berechnung keinerlei besonderen Bedingungen.

Nun folgt die Betrachtung der Vergleichsordnungen, die der Bestmatch-Join zur Bestimmung bestmöglicher Zuordnungen von Tupeln der beiden Eingaberelationen verwenden kann. Dabei ist der Join-Operator nicht auf eine bestimmte Ordnung festgelegt. Vielmehr kann ihm die zu verwendende Ordnung als benutzerdefinierte Funktion übergeben werden, deren innere Arbeitsweise dem Join nicht bekannt ist, die aber die vom Join-Operator vorgegebene Signatur einhält. Der Operator bestimmt dann durch Anwendung der übergebenen Funktion auf je zwei zu vergleichende Paare von Tupeln, ob ein Paar das andere dominiert, oder ob die beiden Paare unvergleichbar sind. Da im Allgemeinen jede Join-Dimension separat betrachtet wird, müssen die Vergleichsordnungen nicht notwendigerweise total sein. Stattdessen sind auch partielle Ordnungen erlaubt und sogar der Regelfall. Zwei Paare sind genau dann unvergleichbar, wenn sie bezüglich einer partiellen Vergleichsordnung unvergleichbar sind. Das ist dann der Fall, wenn jedes der beiden Paare in mindestens einer Join-Dimension besser ist als das jeweils andere Paar. Im Ergebnis des Bestmatch-Joins können somit auch mehrere unvergleichbare Tupelpaare enthalten sein, sofern sie nicht von einem besseren Paar dominiert werden. Darüber hinaus kann es auch Paare geben, die bezüglich der verwendeten Vergleichsordnung gleich, also in allen Join-Dimensionen gleich gut sind. Auch solche gleichen Paare dominieren sich nicht gegenseitig und werden deshalb, sofern sie nicht wiederum von anderen Paaren dominiert werden, alle im Ergebnis enthalten sein. Hingegen dominiert ein Paar ein anderes, wenn es in jeder Join-Dimension mindestens genauso gut und in mindestens einer besser ist, als das andere Paar.

Als Nächstes folgen einige Beispiele für partielle Ordnungen, anhand derer der Bestmatch-Join zwei Paare von Tupeln miteinander vergleichen kann. Dazu muss zunächst für jedes

Join-Attribut y_i mit $i \in \{1, \ldots, d\}$ eine partielle oder totale Ordnung $\leq_{y_i}$ definiert werden, die einen Vergleich in dieser einen Dimension ermöglicht. Bezogen auf zwei Paare von Tupeln $(r \times s)$ und $(r' \times s')$ bedeutet $(r \times s) \leq_{y_i} (r' \times s')$ dann, dass das erste Paar in der betrachteten Dimension y_i mindestens genauso gut ist wie das zweite Paar. Diese Ordnungen entsprechen den vom Benutzer gewünschten Kriterien, nach denen die einzelnen Dimensionen zu vergleichen sind. Durch die Zusammenfassung der Ordnungen aller Join-Dimensionen in einem Vergleichsoperator erhält man die vom Bestmatch-Join zu verwendende Vergleichsordnung für die Tupelpaare. Die Ordnungen können frei definiert werden und bestimmen letztendlich, nach welcher Semantik der Bestmatch-Join entscheidet, welche Paare von Tupeln im Ergebnis enthalten sind und welche nicht.

Beispiel 3.1
Bei numerischen Join-Attributen y_i lässt sich eine Vergleichsordnung definieren, die auf dem Abstand der beiden in einem Paar zusammengefassten Tupel bezüglich der betrachteten Dimensionen beruht:

$$(r \times s) \leq_{y_i} (r' \times s') \quad \Leftrightarrow \quad |r.y_i - s.y_i| \leq |r'.y_i - s'.y_i|$$

Entsprechend gilt auch:

$$(r \times s) <_{y_i} (r' \times s') \quad \Leftrightarrow \quad |r.y_i - s.y_i| < |r'.y_i - s'.y_i|$$

Wenn alle Join-Dimensionen der betrachteten Relationen einen numerischen Typ besitzen, so kann also für jede Dimension dieselbe – entsprechend obigem Beispiel arbeitende – Vergleichsordnung benutzt werden. Ein Paar von Tupeln ist bezüglich dieser Ordnung in einer Dimension genau dann besser als ein Vergleichspaar, wenn der Betrag der Differenz der Attributwerte der beiden Tupel in dieser Dimension kleiner ist als bei dem Vergleichspaar. Fasst man nun die einzelnen Ordnungen auf jeder Dimension zu einer Vergleichsordnung $\prec_{y_1,\ldots,y_d}$ für den Bestmatch-Join zusammen, so kann diese wie folgt definiert werden:

$$\begin{aligned}(r \times s) \prec_{y_1,\ldots,y_d} (r' \times s') \quad \Leftrightarrow \quad & (\forall i \in \{1, \ldots, d\} : (r \times s) \leq_{y_i} (r' \times s')) \ \wedge \\ & (\exists i \in \{1, \ldots, d\} : (r \times s) <_{y_i} (r' \times s'))\end{aligned}$$

Nach dieser Definition dominiert das Tupelpaar $(r \times s)$ das Paar $(r' \times s')$ genau dann, wenn $(r \times s)$ bezüglich der Ordnung $\prec_{y_1,\ldots,y_d}$ in allen Join-Dimensionen $y_1, \ldots, y_d$ mindestens genauso gut ist wie $(r' \times s')$ und wenn es mindestens eine Dimension $i \in \{1, \ldots, d\}$ gibt, in der $(r \times s)$ besser ist. Hingegen dominieren sich Tupel, die bezüglich der Ordnung gleich oder unvergleichbar sind, nicht gegenseitig. Zu beachten ist hierbei, dass die obige Vergleichsordnung $\prec_{y_1,\ldots,y_d}$ des Bestmatch-Joins partiell ist, da jede Dimension separat betrachtet wird. Die weiter oben definierte Ordnung $\leq_{y_i}$ zum Vergleich der einzelnen Dimensionen ist hingegen total. □

Es gibt noch viele weitere Möglichkeiten, totale oder partielle Ordnungen $\leq_{y_i}$ auf den Join-Attributen zu definieren. Die wichtigsten sind nachfolgend aufgeführt. Sie lassen sich in zwei unterschiedliche Typen klassifizieren, nämlich Ordnungen, die auf der natürlichen Ordnung der reellen Zahlen basieren, und allgemeine partielle Ordnungen.

Ordnungen auf der Basis der natürlichen Ordnung der reellen Zahlen

Die Ordnungen dieses Typs sind allgemein wie folgt definierbar:

$$(r \times s) \leq_{y_i} (r' \times s') \quad \Leftrightarrow \quad f(r.y_i, s.y_i) \leq f(r'.y_i, s'.y_i)$$

oder alternativ

$$(r \times s) \leq_{y_i} (r' \times s') \quad \Leftrightarrow \quad f(r.y_i, s.y_i) \geq f(r'.y_i, s'.y_i)$$

Dabei ist y_i das betrachtete Join-Attribut. Die Funktion f besitzt folgende Signatur:

$$f : type(y_i) \times type(y_i) \rightarrow \mathbb{R}$$

Die Ordnung auf dem Join-Attribut y_i wird hier also definiert, indem sie mittels der Funktion f auf die natürliche Ordnung der reellen Zahlen zurückgeführt wird, und ist deshalb ebenso wie diese total.

Zwei Beispiele für totale Ordnungen dieses Typs, die beim Bestmatch-Join zum Einsatz kommen können, sind *minAttrDist* und *coverage*.

Bei *minAttrDist* handelt es sich um die in Beispiel 3.1 eingeführte Ordnung. Die Funktion f ist hier definiert als der Abstand[1] der Attributwerte des betreffenden Attributs der beiden Tupel eines Paares:

$$f_{dist}(a, b) := |a - b|$$

Die Ordnung ist entsprechend so definiert:

$$(r \times s) \leq_{y_i}^{minAttrDist} (r' \times s') \quad \Leftrightarrow \quad |r.y_i - s.y_i| \leq |r'.y_i - s'.y_i|$$

Im Folgenden wird, soweit nicht explizit etwas anderes angegeben ist, immer von numerischen Join-Attributen und dieser Ordnung ausgegangen.

Im Gegensatz zu *minAttrDist* ist *coverage* nicht auf numerischen, sondern auf mengenwertigen Attributen definiert. Ein Paar von Tupeln ist hier besser als ein anderes, wenn die Schnittmenge der Attributmengen der beiden Tupel mehr Elemente enthält als die Schnittmenge des anderen Paares. Das bedeutet, dass dasjenige Paar als besser eingestuft wird, dessen Tupel auf dem mengenwertigen Join-Attribut mehr Übereinstimmungen besitzen. Entsprechend ist die Funktion f als die Mächtigkeit der Schnittmenge der beiden Attributmengen definiert:

$$f_{intersect}(a, b) := |a \cap b|$$

Die Ordnung wird dann wie folgt definiert:

$$(r \times s) \leq_{y_i}^{coverage} (r' \times s') \quad \Leftrightarrow \quad |r.y_i \cap s.y_i| \geq |r'.y_i \cap s'.y_i|$$

[1] Betrag der Differenz

Allgemeine partielle Ordnungen

Im Gegensatz zum vorherigen Typ von totalen Ordnungen sind die Ordnungen dieses Typs, wie der Name schon sagt, partiell. Deshalb können Elemente bezüglich dieser Ordnungen nicht nur besser, gleich gut oder schlechter sein als ein Vergleichselement, sondern es gibt auch den Fall, dass zwei Elemente unvergleichbar sind. Mit Blick auf den Bestmatch-Join ist die interessanteste Frage aber, ob ein Datenelement im Vergleich zu einem anderen besser oder gleich gut ist.

Auch von diesem Typ folgen nun zwei Beispiele. Sie werden mit *subset* und *commonness* bezeichnet und sind auf mengenwertige Attribute bezogen.

Die partielle Ordnung *subset* betrachtet bezüglich des Join-Attributs y_i das Tupelpaar zweier zu vergleichender Paare als das bessere, bei dem die Schnittmenge der Attributwerte der beiden Tupel des Paares eine Obermenge der Schnittmenge der Attributwerte der Tupel des anderen Paares ist. Diese Ordnung ist z. B. sinnvoll anwendbar für das Arbeitsmarkt-Beispiel aus der Einleitung. Dort passt die Kombination eines Bewerbers und einer Arbeitsstelle in der Dimension `Qualifikation` besser zusammen als eine andere, wenn die Menge der Übereinstimmungen der Qualifikationsmerkmale des Bewerbers mit den Qualifikationsanforderungen der Stelle eine Obermenge der Übereinstimmungen der anderen Kombination darstellt. Formal kann die Ordnung so definiert werden:

$$(r \times s) \leq_{y_i}^{subset} (r' \times s') \quad \Leftrightarrow \quad (r'.y_i \cap s'.y_i) \subseteq (r.y_i \cap s.y_i)$$

Diese Ordnung ist offensichtlich partiell, denn zwei Paare von Tupeln sind genau dann unvergleichbar, wenn die Teilmengenbeziehung auf der rechten Seite der obigen Definition weder in der einen noch in der anderen Richtung erfüllt ist.

Wenn es von Bedeutung ist, dass ein mengenwertiges Attribut keine anderen Elemente enthält als solche, die in einer Referenzmenge enthalten sind, so kann man auf die Ordnung *commonness* zurückgreifen. Definiert man die Relation R als diejenige, deren Attribute die Referenzmengen und die Relation S als diejenige, deren Attribute die zu vergleichenden Ist-Mengen enthalten, so ist die Ordnung wiederum bezogen auf das Join-Attribut y_i folgendermaßen definiert:

$$(r \times s) <_{y_i}^{commonness} (r' \times s') \quad \Leftrightarrow \quad (s.y_i \subseteq r.y_i) \wedge (s'.y_i \nsubseteq r'.y_i)$$

Bei dieser partiellen Ordnung ergibt sich die Unvergleichbarkeit zweier Tupelpaare genau dann, wenn auf der rechten Seite obiger Definition entweder beide Teilmengenbeziehungen zugleich erfüllt sind oder wenn beide zugleich nicht erfüllt sind. Gleichheit ergibt sich für den Fall, dass statt beider Teilmengenbeziehungen ebenfalls Gleichheit gilt.

3.1.3 Bestmatch-Joins

Nachdem die Grundlagen von Datenströmen und Ordnungen dargelegt wurden, führt dieser Abschnitt nun die Klasse der Bestmatch-Join Operatoren ein. Diese Operatoren werden in der Folge zur Berechnung von besten Zuordnungen von Objekten zweier Datenströme eingesetzt. Zunächst wird der Bestmatch-Join definiert. Anschließend folgt eine Diskussion seiner Eigenschaften und eine Einführung in die Varianten des Joins.

Definition 3.4 (Bestmatch-Join) *Der Bestmatch-Join $R \bowtie_{y_1,\ldots,y_d} S$ zweier Relationen R und S auf den Join-Attributen $y_1, \ldots, y_d$ ist wie folgt definiert:*

$$R \bowtie_{y_1,\ldots,y_d} S := \left\{ (r \times s) \in (R \times S) \;\middle|\; \neg\exists (r' \times s') \in (R \times S) : (r' \times s') \prec_{y_1,\ldots,y_d} (r \times s) \right\}$$

Das Ergebnis des Bestmatch-Joins enthält also alle Paare von Tupeln aus den beiden Eingaberelationen, die nicht von einem anderen Paar dominiert werden. Das sind die Paare, die gemäß der gegebenen Vergleichsordnung $\prec_{y_1,\ldots,y_d}$ am besten von allen zueinander passen. Es ist also die Aufgabe des Bestmatch-Joins, alle Minima von $R \times S$ bezüglich der partiellen Ordnung $\prec_{y_1,\ldots,y_d}$ zu finden. Um die verwendete Ordnung zu kennzeichnen, kann der Bestmatch-Join von R und S auf den Join-Attributen $y_1, \ldots, y_d$ unter Verwendung der Ordnung $\prec_{y_1,\ldots,y_d}$ alternativ auch mit $R \bowtie_{\prec_{y_1,\ldots,y_d}} S$ bezeichnet werden. Da im Rahmen dieser Arbeit, sofern nicht anders angegeben, immer die Ordnung *minAttrDist* aus dem vorhergehenden Abschnitt benutzt wird, erhält hier die einfachere Schreibweise $R \bowtie_{y_1,\ldots,y_d} S$ den Vorzug. Weiterhin findet für den Bestmatch-Join mitunter auch die abkürzende Bezeichnung BMJ Verwendung.

Abbildung 3.1 veranschaulicht an einem einfachen Beispiel das Ergebnis des Bestmatch-Joins auf zwei Eingaben zweidimensionaler Punkte. Die Punkte der linken Eingabe sind durch Kreuze, die der rechten Eingabe durch Kreise symbolisiert. Die Join-Attribute heißen y_1 und y_2, als Ordnung wird *minAttrDist* verwendet. Es sind somit genau die Paare von Kreuzen und Kreisen im Ergebnis enthalten, die zumindest bezüglich einer Dimension näher aneinander liegen als alle anderen. Die Ergebnispaare sind in der Abbildung grau unterlegt. Es sind die beiden Paare (A,a) und (A,c). Diese sind unvergleichbar und werden von keinem anderen Paar dominiert, da die Tupel des Paares (A,a) in der Dimension y_2 und die Tupel des Paares (A,c) in der Dimension y_1 jeweils den geringsten Abstand aller möglichen Paare von Tupeln haben. Alle anderen Paare werden von (A,a) oder (A,c) dominiert. So ist z. B. das Paar (B,d) mit dem Paar (A,c) unvergleichbar, da die beiden Tupel des Paares bezüglich der Dimension y_1 zwar einen größeren, bezüglich der Dimension y_2 aber einen kleineren Abstand haben als c von A. Das Paar (A,a) hingegen dominiert (B,d), da die Abstände zwischen a und A in beiden Dimensionen kleiner sind als die Abstände zwischen d und B. Damit ist (B,d) nicht im Ergebnis enthalten. Ähnliches gilt für die restlichen dominierten Paare.

Es sollen nun kurz die für den praktischen Einsatz in Datenbanksystemen wichtigen Eigenschaften des Bestmatch-Joins in Bezug auf Kommutativität und Assoziativität dargestellt werden. Die Kommutativität des Joins ist abhängig von der zugrunde liegenden Vergleichsordnung $\prec_{y_1,\ldots,y_d}$. Wie aus der Definition des Bestmatch-Joins unschwer ersichtlich ist, ist der Join genau dann kommutativ, wenn für die eingesetzte Ordnung gilt:

$$(r \times s) \prec_{y_1,\ldots,y_d} (r' \times s') \quad \Leftrightarrow \quad (s \times r) \prec_{y_1,\ldots,y_d} (s' \times r')$$

Die Ordnungen *minAttrDist*, *coverage* und *subset* besitzen offensichtlich die obige Eigenschaft, die Ordnung *commonness* hingegen nicht.

Unabhängig von der Ordnung ist der Bestmatch-Join hingegen im Allgemeinen nicht assoziativ. Dies sei an einem Beispiel demonstriert.

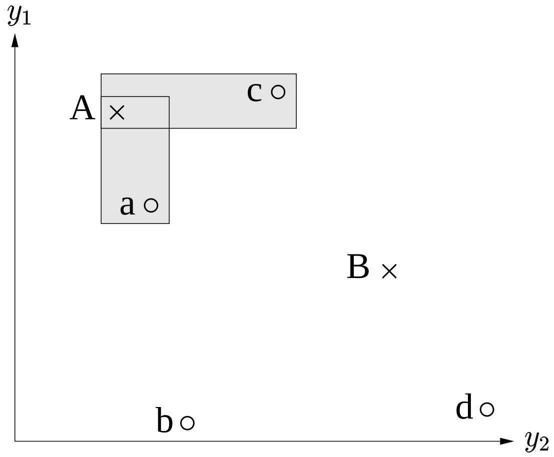


Abb. 3.1: Bestmatch-Join zweier Mengen zweidimensionaler Punkte

Beispiel 3.2
Man betrachte die drei Relationen R, S und T, deren Ausprägungen zweidimensionale Punkte als Tupel enthalten und wie folgt aussehen: $R : \{A = (2,0), B = (2,5)\}$, $S : \{a = (3,2), b = (3,4)\}$ und $T : \{\alpha = (0,2)\}$. Wiederum stellen die beiden Dimensionen die Join-Attribute dar. Als Ordnung werde *minAttrDist* verwendet. Entsprechend seien die partiellen Ordnungen $\prec_{RS}$ auf $R \times S$ und $\prec_{ST}$ auf $S \times T$ definiert. Damit erhält man für $(R \bowtie_{\prec_{RS}} S) \bowtie_{\prec_{ST}} T$ als Ergebnis $\{B \times b \times \alpha\}$, für $R \bowtie_{\prec_{RS}} (S \bowtie_{\prec_{ST}} T)$ hingegen $\{A \times a \times \alpha\}$. □

Der vorgestellte Bestmatch-Join besitzt einige praktisch relevante Varianten, die nun vorgestellt und definiert werden. Zunächst ist der Left-Outer-Bestmatch-Join zu nennen. Er berechnet im Gegensatz zum normalen Bestmatch-Join nicht nur die insgesamt besten Zuordnungen bezüglich der verwendeten partiellen Ordnung $\prec_{y_1,\ldots,y_d}$, sondern die besten Zuordnungen für jedes einzelne Tupel der linken Eingabe. Sein Ergebnis entspricht somit der Vereinigung der Ergebnisse mehrerer Bestmatch-Joins, die jeweils den Join der gesamten rechten Eingabe mit jedem einzelnen Tupel der linken Eingabe berechnen.

Definition 3.5 (Left-Outer-Bestmatch-Join) *Der Left-Outer-Bestmatch-Join zweier Relationen R und S auf den Join-Attributen $y_1, \ldots, y_d$ wird mit $R \ltimes_{y_1,\ldots,y_d} S$ bezeichnet und ist wie folgt definiert:*

$$R \ltimes_{y_1,\ldots,y_d} S := \left\{ (r \times s) \in (R \times S) \;\middle|\; \neg\exists (r \times s') \in (R \times S) : (r \times s') \prec_{y_1,\ldots,y_d} (r \times s) \right\}$$

Der Left-Outer-Bestmatch-Join wird im Rahmen dieser Arbeit auch kurz mit LOBMJ bezeichnet. Abbildung 3.2 zeigt das Ergebnis des Left-Outer-Bestmatch-Joins für das Beispiel aus Abbildung 3.1. Im Vergleich zum normalen Bestmatch-Join sind zu den beiden Ergebnispaaren (A, a) und (A, c) nun auch zwei Paare mit dem Punkt B im Ergebnis enthalten. Es wird davon ausgegangen, dass die Kreuze die linke Eingabe darstellen. Damit werden die Punkte A und B jeweils separat betrachtet, was im Falle von A das gleiche Ergebnis wie in Abbildung 3.1 ergibt. Für B kommen nun noch die Paare (B, a)

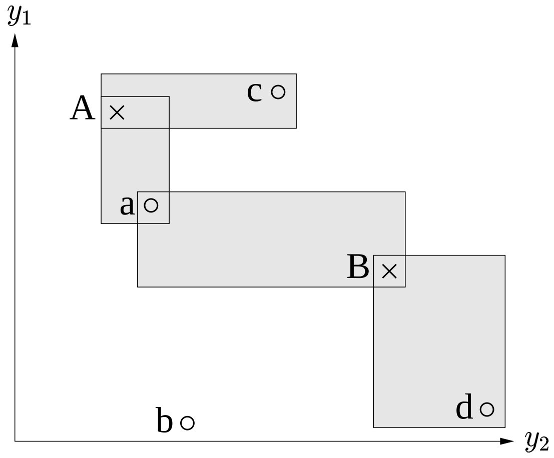


Abb. 3.2: Left-Outer-Bestmatch-Join zweier Mengen zweidimensionaler Punkte

und (B, d) im Ergebnis hinzu, denn bezüglich B sind diese beiden Paare unvergleichbar. Während (B, a) in der Dimension y_1 den geringeren Abstand aufweist, ist das Paar (B, d) in der Dimension y_2 besser. Die Paare (B, b) und (B, c) werden hingegen beide von (B, d) dominiert, da sowohl b als auch c in beiden Dimensionen weiter von B entfernt ist als d. Sie sind damit nicht im Ergebnis enthalten. Insgesamt ergibt sich somit $R ⟕_{y_1,y_2} S = \{(A, a), (A, c), (B, a), (B, d)\}$.

Der Left-Outer-Bestmatch-Join besitzt von allen Varianten des Bestmatch-Joins wohl die größte praktische Relevanz. Schließlich ist der häufigste Anwendungsfall eines derartigen Joins der Versuch, jedem Element aus einer Menge diejenigen Elemente aus einer anderen Menge zuzuordnen, die gemäß einer vorgegebenen Metrik am besten zu dem jeweiligen Element passen. Auf das Arbeitsmarkt-Beispiel aus der Einleitung angewandt bedeutet dies, dass nicht nur die Paare von Bewerbern und Stellen gebildet werden, die insgesamt am besten zusammenpassen, wodurch einzelne Bewerber ohne Arbeitsplatzangebot und einzelne Stellen unbesetzt bleiben können,[1] sondern dass entweder jedem Bewerber die am besten zu ihm passenden Arbeitsstellen oder jedem Stellenangebot die am besten zu ihm passenden Bewerber zugeordnet werden, je nachdem, ob Bewerber oder Stellen die linke Eingabe darstellen.

Analog zum LOBMJ lässt sich der Right-Outer-Bestmatch-Join, kurz ROBMJ genannt, definieren. Er berechnet entsprechend für jedes Tupel der rechten Eingabe die besten Zuordnungen aus allen Tupeln der linken Eingabe.

Definition 3.6 (Right-Outer-Bestmatch-Join) *Der Right-Outer-Bestmatch-Join zweier Relationen R und S auf den Join-Attributen $y_1, \ldots, y_d$ wird mit $R ⟖_{y_1,\ldots,y_d} S$ bezeichnet und ist wie folgt definiert:*

$$R ⟖_{y_1,\ldots,y_d} S := \left\{ (r \times s) \in (R \times S) \;\middle|\; \neg\exists (r' \times s) \in (R \times S) : (r' \times s) \prec_{y_1,\ldots,y_d} (r \times s) \right\}$$

[1] Vergleiche Punkt B in Abbildung 3.1 zum Bestmatch-Join.

Die letzte Variante ist der Full-Outer-Bestmatch-Join, der auch kurz FOBMJ genannt wird. Sein Ergebnis ist die Vereinigung der Ergebnisse des Left-Outer-Bestmatch-Joins und des Right-Outer-Bestmatch-Joins. Damit berechnet er also die besten Zuordnungen für jedes Tupel der linken und der rechten Eingabe.

Definition 3.7 (Full-Outer-Bestmatch-Join) *Der Full-Outer-Bestmatch-Join zweier Relationen R und S auf den Join-Attributen $y_1, \ldots, y_d$ wird mit $R \Join_{y_1,\ldots,y_d} S$ bezeichnet und ist wie folgt definiert:*

$$R \Join_{y_1,\ldots,y_d} S := \Big\{ (r \times s) \in (R \times S) \;\Big|\; (\neg\exists (r \times s') \in (R \times S) : (r \times s') \prec_{y_1,\ldots,y_d} (r \times s)) \vee (\neg\exists (r' \times s) \in (R \times S) : (r' \times s) \prec_{y_1,\ldots,y_d} (r \times s)) \Big\}$$

3.2 Berechnungsansätze

Nachdem im vorangegangenen Abschnitt der Bestmatch-Join und seine Varianten vorgestellt wurden, behandelt diese Sektion nun die Theorie zweier unterschiedlicher Ansätze zur Berechnung des Bestmatch-Joins auf Datenströmen. Da der Bestmatch-Join ein blockierender Operator ist, kann er nicht ohne weiteres auf Datenströmen arbeiten. Um ihn pipelinefähig zu machen und damit auch die Möglichkeit zur Verarbeitung unendlicher Datenströme zu erhalten, sind zusätzliche Überlegungen nötig. Dabei ist zu unterscheiden zwischen einer approximativen Berechnung des Join-Ergebnisses auf der einen und einer Einschränkung des Suchraums, verbunden mit zusätzlichen Anforderungen an die Art und Weise, wie der Datenstrom die Daten liefert, auf der anderen Seite. Diese beiden Alternativen sind Gegenstand der nun folgenden Betrachtungen.

3.2.1 Approximativer Bestmatch-Join

Dieser Ansatz aus [KS02b] ist dadurch gekennzeichnet, dass er in Kauf nimmt, mitunter inkorrekte Zwischenergebnisse wie in Abschnitt 3.1.1 beschrieben auszugeben, um die blockierende Eigenschaft des Bestmatch-Joins zu umgehen. Dazu verwendet er die in demselben Abschnitt definierten Datenfenster. Die grundlegende Architektur des Operators ist in Abbildung 3.3 dargestellt. Die beiden Eingaben R und S werden kontinuierlich in die zugehörigen Datenfenster W_R^t und W_S^t eingelesen und von dort dem Bestmatch-Join Operator als Eingabe zugeführt. Die Zwischenergebnisse werden in dem Datenfenster O gehalten, das vom Join-Operator während der Berechnung immer wieder auf den neuesten Stand gebracht wird. Der Inhalt von O wird auch als Zustand des Operators bezeichnet. Die Summe der Größen der drei Datenfenster $size_{W_R}^t$, $size_{W_S}^t$ und $size_O^t$ ist konstant. Sie bestimmt den maximalen Speicherbedarf des Operators. Ist eines der Datenfenster voll, so wird ein Teil seines Inhalts verworfen und mit neuen Daten gefüllt. Die verworfenen Daten sind an allen folgenden Berechnungen nicht mehr beteiligt. Der Vorteil dieser Methode gegenüber einem herkömmlichen Berechnungsverfahren liegt darin, dass nach dem Start einer Anfrage kontinuierlich Zwischenergebnisse generiert und an die folgenden Operatoren oder an den Benutzer weitergegeben werden. Diese Zwischenergebnisse sind allerdings

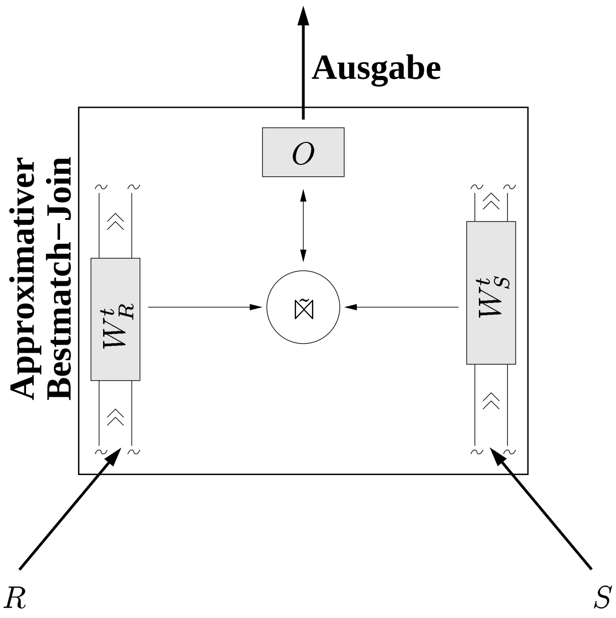


Abb. 3.3: Architektur des approximativen Bestmatch-Joins auf Datenströmen

approximativ, denn wie bereits in Abschnitt 3.1.1 dargestellt, fehlen in der Regel korrekte Ergebnistupel, da die zu ihrer Generierung benötigten Eingaben entweder noch nicht verarbeitet oder zu früh wieder aus den Datenfenstern verdrängt worden sind. Letzteres liegt daran, dass die für ein korrektes Ergebnistupel zu kombinierenden Eingabetupel der linken und der rechten Eingabe zu sehr unterschiedlichen Zeiten angeliefert werden können, so dass das eine Tupel aufgrund von Platzmangel in dem zugehörigen Datenfenster bereits wieder verworfen wird, bevor das andere angekommen ist. Das entsprechende Ergebnistupel wird dann nie generiert und ist somit auch nicht im Ergebnis enthalten. Andererseits können in den Zwischenergebnissen auch inkorrekte Tupel enthalten sein, die wegen der fehlenden korrekten Tupel nicht invalidiert worden sind. Im Falle endlicher Datenströme gelten die gleichen Überlegungen ferner auch für das endgültige Endergebnis der Join-Berechnung.

Für die Implementierung des approximativen Bestmatch-Joins sind unterschiedliche Strategien zur Verteilung des verfügbaren Speichers auf die Datenfenster und zur Bestimmung der zu verdrängenden Datenobjekte bei vollem Datenfenster denkbar. So kann man z. B. den zur Verfügung stehenden Speicher in drei feste Teile aufteilen, die den drei Datenfenstern zugewiesen werden. Bei dieser als *FIXED* bezeichneten Strategie ist der Speicher eines jeden Datenfensters also statisch und in seiner Größe während der Berechnung nicht variabel. Im Gegensatz dazu lässt die sogenannte *DYNAMIC*-Strategie eine dynamische Veränderung der Speichergröße der einzelnen Datenfenster während der Berechnung zu. Hier wird jedem Fenster initial ein bestimmter Teil des Speichers zugewiesen, der so lange vergrößert werden darf, bis der verfügbare Speicher aufgebraucht ist. Sollte danach die Vergrößerung eines Fensters nötig werden, so muss dafür ein anderes Fenster entsprechend verkleinert werden. Mögliche Verfahren zur Verdrängung von Datenobjekten bei vollem Datenfenster sind z. B. die *FIFO*-Strategie, bei der die ältesten Datenobjekte zuerst verworfen werden, und die *LFU*-Strategie, bei der diejenigen Datenobjekte als erstes aus dem Datenfenster entfernt werden, die bisher am wenigsten an der Generierung von Ergebnisobjekten beteiligt waren.

Im Folgenden werden die Algorithmen zur Berechnung des approximativen Bestmatch-Joins kurz vorgestellt. Aufgrund der engen Verwandtschaft des Bestmatch-Joins mit dem Skyline-Operator aus Abschnitt 2.1 – der Bestmatch-Join berechnet im Prinzip die Skyline des Kreuzprodukts der beiden Eingaberelationen – liegt es nahe, Abwandlungen der Skyline-Algorithmen für den Bestmatch-Join zu verwenden. Alle in dieser Arbeit vorgestellten Algorithmen für den Bestmatch-Join basieren auf Nested-Loops Algorithmen. Da es sich beim Bestmatch-Join ebenso wie beim Skyline-Operator im Grunde um einen blockierenden Operator handelt, ist hier einiges an Anpassungsarbeit zu leisten.

Algorithmus 3.1 (*UpdateApprox*) zeigt zunächst die Vorgehensweise bei der Aktualisierung des gegenwärtigen Ergebnisses O des approximativen Bestmatch-Joins bei Hinzukommen eines neuen Datenobjekts x unter Verwendung der partiellen Ordnung $\prec_{y_1,\ldots,y_d}$. Das neue Objekt wird dabei mit allen im Datenfenster von O bereits enthaltenen Objekten verglichen und verdrängt alle Elemente aus O, die von x bezüglich der partiellen Ordnung dominiert werden. Falls mindestens eine solche Verdrängung stattgefunden hat, wird x außerdem in O aufgenommen. Die Ausgabe dieses Algorithmus besteht aus einer Menge von Änderungsoperationen, welche die vorgenommenen Änderungen am Zustand O enthält. Diese Änderungen, die mit $[insert(x)]$ für das Einfügen eines neuen Datenelements in das aktuelle Ergebnis bzw. mit $[delete(o)]$ für das Löschen eines alten Datenelements aus dem aktuellen Ergebnis bezeichnet werden, propagiert der Algorithmus am Ende an den nächsten Operator. Wenn das Datenfenster des Zustands O voll ist, so werden gemäß der verwendeten Verdrängungsstrategie Datenelemente des Fensters verworfen. Diese sind dann auf jeden Fall im endgültigen Ergebnis des Joins enthalten, weil sie nach ih-

Algorithmus 3.1 Aktualisieren des approximativen Ergebnisses (*UpdateApprox*)

Eingabe: neues Datenobjekt x, Zustand O, partielle Ordnung $\prec_{y_1,\ldots,y_d}$
Ausgabe: Menge *result* von Update-Operationen

$result \leftarrow \emptyset$
$insert \leftarrow false$
for all $o \in O$ **do**
  if $x \prec_{y_1,\ldots,y_d} o$ **then**
    $insert \leftarrow true$
    $result \leftarrow result \cup \left\{[delete(o)]\right\}$
    $O \leftarrow O \setminus \{o\}$
  end if
end for
if $insert$ **then**
  $result \leftarrow result \cup \left\{[insert(x)]\right\}$
  $O \leftarrow O \cup \{x\}$
end if
return $result$

rer Entfernung aus O nicht mehr durch nachrückende Datenobjekte invalidiert werden können.

Unter Verwendung dieses Update-Algorithmus kann nun in Algorithmus 3.2 das Verfahren zur Berechnung des approximativen Bestmatch-Joins auf Datenströmen angegeben werden. Für jedes neue Datenelement r aus dem Datenstrom R werden alle Paare mit den aktuell im Datenfenster W_S^t enthaltenen Datenelementen des Datenstroms S gebildet. Mittels des Update-Algorithmus wird für jedes dieser Paare überprüft, ob es in das Ergebnis aufzunehmen ist. Die während der Aktualisierung mit einem neuen Tupelpaar erzeugte Menge von Änderungsoperationen wird mit der Menge Q^t der akkumulierten Änderungsoperationen für das neue Datenobjekt vereinigt. Am Ende der Bearbeitung eines neuen Datenelements speichert der Algorithmus das Element mittels der Methode *store* in dem zugehörigen Datenfenster und leitet die akkumulierte Menge von Änderungsoperationen an den nächsten Operator weiter. Jedes neue Datenobjekt s aus dem Datenstrom S wird analog behandelt. Der Algorithmus terminiert, wenn keiner der beiden Eingabedatenströme mehr neue Datenobjekte liefert.

Zu dieser Variante des Bestmatch-Joins auf Datenströmen ist abschließend zu sagen, dass die Qualität der approximativen Ergebnisse abhängig ist von der Größe der verwendeten Datenfenster, der Reihenfolge, in der die Eingabedaten geliefert werden, und der Verdrängungsstrategie für die Entfernung von Datenelementen bei vollem Fenster. Der hier beschriebene Ansatz zur approximativen Berechnung des Bestmatch-Joins auf Datenströmen wird in dieser Arbeit nicht weiter verfolgt. Nähere Ausführungen zu diesem Thema, insbesondere auch zu Optimierungsansätzen zur Reduzierung der Laufzeit und zur Steigerung der Ergebnisqualität sowie Ergebnisse von Laufzeit- und Qualitätsuntersuchungen an einer Test-Implementierung, finden sich in [KS02b].

Algorithmus 3.2 Approximativer Bestmatch-Join

Eingabe: Datenströme R und S, partielle Ordnung $\prec_{y_1,\ldots,y_d}$
Ausgabe: $Q^t(R \bowtie S)$

$O \leftarrow \emptyset$
while new objects available **do**
  $Q^t \leftarrow \emptyset$
  if new object $r \in R$ available **then**
    for all $s \in W_S^t$ **do**
      $Q^t \leftarrow Q^t \cup \mathit{UpdateApprox}((r \times s), O, \prec_{y_1,\ldots,y_d})$
    end for
    $W_R^t.\mathit{store}(r)$
  else if new object $s \in S$ available **then**
    for all $r \in W_R^t$ **do**
      $Q^t \leftarrow Q^t \cup \mathit{UpdateApprox}((r \times s), O, \prec_{y_1,\ldots,y_d})$
    end for
    $W_S^t.\mathit{store}(s)$
  end if
  output Q^t
end while

3.2.2 Bestmatch-Join mit Einschränkungen

Nachdem im vorangegangenen Abschnitt ein Verfahren zur näherungsweisen Berechnung des Bestmatch-Joins auf Datenströmen vorgestellt wurde, gibt dieser Teil eine Einführung in einen Ansatz, der unter bestimmten Bedingungen die Ermittlung des vollständig korrekten Ergebnisses erlaubt. Der Bestmatch-Join mit Einschränkungen, auch eingeschränkter Bestmatch-Join oder kurz ϵ-BMJ genannt, ist eine Variante des in Kapitel 3.1.3 definierten Bestmatch-Joins. Der Unterschied besteht dabei in einer Einschränkung des Suchraums auf eine ϵ-Umgebung. Die Ausführungen zum approximativen Bestmatch-Join haben gezeigt, dass bei begrenzter Größe der Datenfenster und ohne zusätzliche Annahmen über die Reihenfolge der angelieferten Eingabedaten die Bestimmung des korrekten Endergebnisses im Allgemeinen nicht möglich ist. Um dies dennoch zu ermöglichen, müssen sowohl eine Vorsortierung der Eingabedaten als auch die oben genannte Einschränkung der relevanten Attributwerte vorausgesetzt bzw. vorgenommen werden. Die Vorsortierung wird in Abschnitt 4.2.1 genauer behandelt. Die Einschränkung auf eine ϵ-Umgebung ist Teil der folgenden Definitionen und Ausführungen.

Der Bestmatch-Join mit Einschränkungen entspricht im Wesentlichen dem in Kapitel 3.1.3 vorgestellten Bestmatch-Join. Der einzige Unterschied besteht darin, dass für das Ergebnis des Joins nur solche Paare von Tupeln in Betracht kommen, bei denen die Abstände der beiden Tupel für alle Join-Attribute y_i mit $i \in \{1, \ldots, d\}$ jeweils vorgegebene Höchstabstände ϵ_i einhalten. Auch bei den weiteren Betrachtungen werden numerische Join-Attribute und die in Sektion 3.1.2 vorgestellte Vergleichsordnung *minAttrDist* vorausgesetzt. Der Abstand zweier Tupel in einer Dimension entspricht damit also wieder dem Betrag der Differenz der beiden Attributwerte. Die vorgenommene Einschränkung erweist sich in der Praxis als durchaus sinnvoll. Sie eliminiert sogenannte Ausreißer aus dem Join-Ergebnis. Dabei handelt es sich um Tupelpaare, die zwar in einer oder mehreren Join-Dimensionen sehr gut zusammenpassen, so dass sie in diesen Dimensionen von keiner anderen Paarung dominiert werden, die aber in anderen relevanten Dimensionen sehr weit auseinander liegen. Meistens interessieren aber nur Kombinationen von solchen Tupeln, die in allen Join-Attributen eine gewisse Übereinstimmung besitzen, also einen bestimmten Maximalabstand ϵ_i in der Dimension y_i nicht überschreiten. Bezogen auf das Beispiel aus der Einleitung könnte das so aussehen, dass etwa eine Luftfahrtgesellschaft einen Piloten sucht, der gewisse Altersvorgaben nicht über- oder unterschreiten darf. Die Gesellschaft sucht dann als idealen Kompromiss zwischen Erfahrung und Alter einen Piloten, der optimalerweise 35 Jahre alt ist und maximal fünf Jahre jünger oder älter sein darf. Dann ist $\epsilon_{Alter} = 5\ Jahre$ und die im Join-Ergebnis enthaltenen Piloten sind alle zwischen 30 und 40 Jahre alt. Ein jüngerer oder älterer Pilot ist selbst dann nicht im Ergebnis enthalten, wenn er alle anderen relevanten Bedingungen, wie Qualifikation und Gehaltsvorstellung, exakt erfüllt.

Um den Bestmatch-Join mit Einschränkungen definieren zu können, werden nun zunächst einige Bezeichnungen eingeführt. Der Abstand der an einem Paar beteiligten Tupel $r \in R$ und $s \in S$ bezüglich eines Join-Attributs y_i mit $i \in \{1, \ldots, d\}$ wird mit $a_i(r, s)$ bezeichnet und ist bei Verwendung der Ordnung *minAttrDist* wie folgt definiert:

$$a_i(r, s) := |r.y_i - s.y_i|$$

Mit dieser Festlegung lässt sich die Definition der Ordnung *minAttrDist* aus Kapitel 3.1.2 folgendermaßen formulieren:

$$(r \times s) \leq_{y_i}^{minAttrDist} (r' \times s') \quad \Leftrightarrow \quad a_i(r,s) \leq a_i(r',s')$$

Der vorgegebene Maximalabstand der Tupel eines Paares in der Dimension i werde mit ϵ_i bezeichnet. Dann sind die Vektoren $\vec{a}(r,s)$ und $\vec{\epsilon}$ so definiert:

$$\vec{a}(r,s) := \begin{pmatrix} a_1(r,s) \\ \vdots \\ a_d(r,s) \end{pmatrix}, \qquad \vec{\epsilon} := \begin{pmatrix} \epsilon_1 \\ \vdots \\ \epsilon_d \end{pmatrix}$$

Damit gilt:

$$\vec{a}(r,s) \leq \vec{\epsilon} \quad \Leftrightarrow \quad \forall i \in \{1,\ldots,d\} : a_i(r,s) \leq \epsilon_i$$

Die Bedingung $\vec{a}(r,s) \leq \vec{\epsilon}$ ist also genau dann erfüllt, wenn die $\leq$-Bedingung für jede Komponente der beiden Vektoren $\vec{a}(r,s)$ und $\vec{\epsilon}$ erfüllt ist. Im Übrigen seien die Attributwerte y_i und damit auch die Abstände $a_i(r,s)$ für jedes Paar von Tupeln $r \in R$ und $s \in S$ auf den Wertebereich $[0;1]$ und die Maximalabstände ϵ_i auf den Bereich $]0;1]$ normiert:

$$\forall r \in R,\ \forall s \in S,\ \forall i \in \{1,\ldots,d\} : y_i \in [0;1] \ \wedge\ a_i(r,s) \in [0;1] \ \wedge\ \epsilon_i \in]0;1]$$

Für den Fall $\epsilon_i = 0$ entartet der Bestmatch-Join in der Dimension i zu einem Equi-Join. Dieser Fall wird daher nicht näher betrachtet.

Mit diesen Bezeichnungen kann nun der Begriff der ϵ-Umgebung eingeführt werden.

Definition 3.8 (ϵ-Umgebung) *Die ϵ-Umgebung $U_S^{\epsilon_1,\ldots,\epsilon_d}$ eines Tupels $r \in R$ ist definiert als die Menge der Tupel $s \in S$, die in allen Join-Dimensionen $i \in \{1,\ldots,d\}$ von r jeweils höchstens den Abstand ϵ_i haben:*

$$U_S^{\epsilon_1,\ldots,\epsilon_d}(r) := \Big\{ s \in S \mid \vec{a}(r,s) \leq \vec{\epsilon} \Big\}$$

Analog ist die ϵ-Umgebung $U_R^{\epsilon_1,\ldots,\epsilon_d}$ eines Tupels $s \in S$ definiert als die Menge der Tupel $r \in R$, die in allen Join-Dimensionen $i \in \{1,\ldots,d\}$ von s jeweils höchstens den Abstand ϵ_i haben:

$$U_R^{\epsilon_1,\ldots,\epsilon_d}(s) := \Big\{ r \in R \mid \vec{a}(r,s) \leq \vec{\epsilon} \Big\}$$

Mit Hilfe der obigen Festlegungen ist es nun möglich, den Bestmatch-Join mit Einschränkungen zu definieren.

Definition 3.9 (Bestmatch-Join mit Einschränkungen) *Der Bestmatch-Join mit Einschränkungen zweier Relationen R und S auf den Join-Attributen $y_1,\ldots,y_d$ mit den Maximalabständen $\epsilon_1,\ldots,\epsilon_d$ wird mit $R \bowtie_{y_1,\ldots,y_d}^{\epsilon_1,\ldots,\epsilon_d} S$ bezeichnet und ist wie folgt definiert:*

$$R \bowtie_{y_1,\ldots,y_d}^{\epsilon_1,\ldots,\epsilon_d} S := \Big\{ (r \times s) \in (R \times S) \ \Big|\ \vec{a}(r,s) \leq \vec{\epsilon} \ \wedge\ \neg\exists (r' \times s') \in (R \times S) : \\ (\vec{a}(r',s') \leq \vec{\epsilon} \ \wedge\ (r' \times s') \prec_{y_1,\ldots,y_d} (r \times s)) \Big\}$$

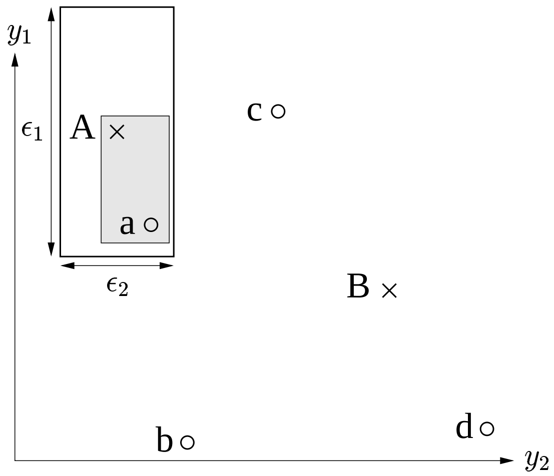


Abb. 3.4: Eingeschränkter Bestmatch-Join zweier Mengen zweidimensionaler Punkte

Alternativ kann der Bestmatch-Join mit Einschränkungen auch über die ϵ-Umgebungen definiert werden:

$$R \bowtie^{\epsilon_1,\dots,\epsilon_d}_{y_1,\dots,y_d} S := \Big\{ (r \times s) \in (R \times S) \;\Big|\; r \in U_R^{\epsilon_1,\dots,\epsilon_d}(s) \;\wedge\; s \in U_S^{\epsilon_1,\dots,\epsilon_d}(r) \;\wedge\; \neg\exists (r' \times s') \in (R \times S) : (r' \in U_R^{\epsilon_1,\dots,\epsilon_d}(s') \;\wedge\; s' \in U_S^{\epsilon_1,\dots,\epsilon_d}(r') \;\wedge\; (r' \times s') \prec_{y_1,\dots,y_d} (r \times s)) \Big\}$$

Das Ergebnis des Bestmatch-Joins mit Einschränkungen enthält also genau die Paare des entsprechenden Bestmatch-Joins ohne Einschränkungen, die aus solchen Tupeln bestehen, deren Abstände in allen relevanten Join-Dimensionen den für die jeweilige Dimension zulässigen Höchstabstand nicht überschreiten. Abbildung 3.4 zeigt hierfür ein Beispiel basierend auf Abbildung 3.1 zum nicht eingeschränkten Bestmatch-Join. In dem Bild ist die ϵ-Umgebung des Punktes A eingezeichnet. Wie man sieht, besteht das Ergebnis des ϵ-BMJ für dieses Beispiel nur noch aus dem Tupelpaar (A, a), während das Ergebnis des nicht eingeschränkten Bestmatch-Joins zusätzlich noch das Paar (A, c) enthält. Dieses Paar fehlt in der Variante mit Einschränkungen, da der Punkt c offensichtlich außerhalb der ϵ-Umgebung von A liegt und umgekehrt. Hingegen liegt a innerhalb der ϵ-Umgebung von A und damit auch A innerhalb der ϵ-Umgebung von a, da die Umgebung für alle Punkte in jeder Join-Dimension jeweils die gleiche Ausdehnung besitzt. Somit ist das Paar (A, a) im Ergebnis enthalten. Ergebnispaare, die den Punkt B enthalten, gibt es hingegen mit der gleichen Begründung wie beim uneingeschränkten Bestmatch-Join nicht.

Der Bestmatch-Join mit Einschränkungen besitzt die gleichen Varianten wie der in Kapitel 3.1.3 definierte Bestmatch-Join ohne Einschränkungen. Diese Varianten werden nun kurz definiert und erläutert. Den Anfang macht der Left-Outer-Bestmatch-Join mit Einschränkungen, der auch als eingeschränkter Left-Outer-Bestmatch-Join oder kurz als ϵ-LOBMJ bezeichnet wird.

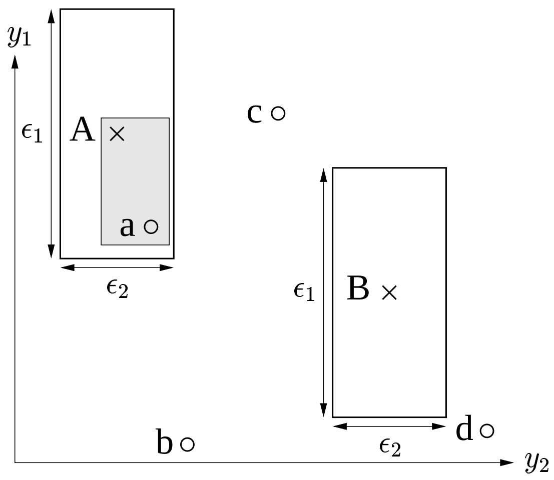


Abb. 3.5: Eingeschränkter Left-Outer-Bestmatch-Join zweier Mengen zweidimensionaler Punkte

Definition 3.10 (Left-Outer-Bestmatch-Join mit Einschränkungen) *Der Left-Outer-Bestmatch-Join mit Einschränkungen* $R ⟕^{\epsilon_1,\ldots,\epsilon_d}_{y_1,\ldots,y_d} S$ *zweier Relationen* R *und* S *auf den Join-Attributen* $y_1, \ldots, y_d$ *mit den Maximalabständen* $\epsilon_1, \ldots, \epsilon_d$ *ist wie folgt definiert:*

$$R ⟕^{\epsilon_1,\ldots,\epsilon_d}_{y_1,\ldots,y_d} S := \Big\{ (r \times s) \in (R \times S) \;\Big|\; \vec{a}(r,s) \leq \vec{\epsilon} \,\wedge\, \neg\exists (r \times s') \in (R \times S) : \\ (\vec{a}(r,s') \leq \vec{\epsilon} \,\wedge\, (r \times s') \prec_{y_1,\ldots,y_d} (r \times s)) \Big\}$$

Die alternative Definition mittels der ϵ-Umgebungen sieht wie folgt aus:

$$R ⟕^{\epsilon_1,\ldots,\epsilon_d}_{y_1,\ldots,y_d} S := \Big\{ (r \times s) \in (R \times S) \;\Big|\; r \in U_R^{\epsilon_1,\ldots,\epsilon_d}(s) \,\wedge\, s \in U_S^{\epsilon_1,\ldots,\epsilon_d}(r) \,\wedge \\ \neg\exists (r \times s') \in (R \times S) : \\ (r \in U_R^{\epsilon_1,\ldots,\epsilon_d}(s') \,\wedge\, s' \in U_S^{\epsilon_1,\ldots,\epsilon_d}(r) \,\wedge \\ (r \times s') \prec_{y_1,\ldots,y_d} (r \times s)) \Big\}$$

Entsprechend dem Bestmatch-Join mit Einschränkungen enthält auch der Left-Outer-Bestmatch-Join mit Einschränkungen diejenigen Tupel der Variante ohne Einschränkungen, die die zusätzliche Bedingung des Maximalabstands auf den einzelnen Join-Attributen erfüllen. Dies ist in Abbildung 3.5 dargestellt. Sie basiert ebenfalls auf der entsprechenden Abbildung 3.2 der uneingeschränkten Variante. In dem Bild sind die ϵ-Umgebungen der beiden Punkte A und B, die wieder als die Punkte der linken Eingabe betrachtet werden, eingezeichnet. Für A gilt das bereits zu Abbildung 3.4 Erläuterte. Hingegen erzeugt B nun im Gegensatz zum Join ohne Einschränkungen überhaupt kein Ergebnispaar mehr, da die Partnertupel a und d der beiden Ergebnispaare von B aus Bild 3.2 nun offensichtlich beide außerhalb der ϵ-Umgebung von B liegen. Insgesamt fallen also im Vergleich zum nicht eingeschränkten Join die Paare (A, c), (B, a) und (B, d) aus dem Ergebnis heraus, das nun nur noch aus dem Paar (A, a) besteht.

Wiederum ist die Variante des Left-Outer-Bestmatch-Joins mit Einschränkungen die in der Praxis relevanteste. Deshalb steht sie auch im Zentrum der folgenden Betrachtungen in Kapitel 4 und wurde in dem in Kapitel 5 beschriebenen und in Kapitel 6 untersuchten Prototyp implementiert.

Analog zum ϵ-LOBMJ kann der Right-Outer-Bestmatch-Join mit Einschränkungen, auch eingeschränkter Right-Outer-Bestmatch-Join oder kurz ϵ-ROBMJ genannt, definiert werden.

Definition 3.11 (Right-Outer-Bestmatch-Join mit Einschränkungen) *Der Right-Outer-Bestmatch-Join mit Einschränkungen* $R \text{⟖}^{\epsilon_1,\dots,\epsilon_d}_{y_1,\dots,y_d} S$ *zweier Relationen* R *und* S *auf den Join-Attributen* $y_1,\dots,y_d$ *mit den Maximalabständen* $\epsilon_1,\dots,\epsilon_d$ *ist wie folgt definiert:*

$$
\begin{aligned}
R \text{⟖}^{\epsilon_1,\dots,\epsilon_d}_{y_1,\dots,y_d} S := \Big\{(r \times s) \in (R \times S) \;\Big|\; & \vec{a}(r,s) \le \vec{\epsilon} \;\wedge\; \neg\exists (r' \times s) \in (R \times S) : \\
& (\vec{a}(r',s) \le \vec{\epsilon} \;\wedge\; (r' \times s) \prec_{y_1,\dots,y_d} (r \times s))\Big\}
\end{aligned}
$$

Alternativ dazu ist diese Definition möglich:

$$
\begin{aligned}
R \text{⟖}^{\epsilon_1,\dots,\epsilon_d}_{y_1,\dots,y_d} S := \Big\{(r \times s) \in (R \times S) \;\Big|\; & r \in U_R^{\epsilon_1,\dots,\epsilon_d}(s) \;\wedge\; s \in U_S^{\epsilon_1,\dots,\epsilon_d}(r) \;\wedge \\
& \neg\exists (r' \times s) \in (R \times S) : \\
& (r' \in U_R^{\epsilon_1,\dots,\epsilon_d}(s) \;\wedge\; s \in U_S^{\epsilon_1,\dots,\epsilon_d}(r') \;\wedge \\
& (r' \times s) \prec_{y_1,\dots,y_d} (r \times s))\Big\}
\end{aligned}
$$

Die letzte Variante ist der Full-Outer-Bestmatch-Join mit Einschränkungen, für den auch die Bezeichnung eingeschränkter Full-Outer-Bestmatch-Join und die Kurzbezeichnung ϵ-FOBMJ existieren.

Definition 3.12 (Full-Outer-Bestmatch-Join mit Einschränkungen) *Der Full-Outer-Bestmatch-Join mit Einschränkungen zweier Relationen* R *und* S *auf den Join-Attributen* $y_1,\dots,y_d$ *mit den Maximalabständen* $\epsilon_1,\dots,\epsilon_d$ *wird mit* $R \text{⟗}^{\epsilon_1,\dots,\epsilon_d}_{y_1,\dots,y_d} S$ *bezeichnet und ist wie folgt definiert:*

$$
\begin{aligned}
R \text{⟗}^{\epsilon_1,\dots,\epsilon_d}_{y_1,\dots,y_d} S := \Big\{(r \times s) \in (R \times S) \;\Big|\; & \vec{a}(r,s) \le \vec{\epsilon} \;\wedge \\
& ((\neg\exists (r \times s') \in (R \times S) : \\
& (\vec{a}(r,s') \le \vec{\epsilon} \;\wedge\; (r \times s') \prec_{y_1,\dots,y_d} (r \times s))) \;\vee \\
& (\neg\exists (r' \times s) \in (R \times S) : \\
& (\vec{a}(r',s) \le \vec{\epsilon} \;\wedge\; (r' \times s) \prec_{y_1,\dots,y_d} (r \times s))))\Big\}
\end{aligned}
$$

Alternativ kann die Definition unter Verwendung der ϵ-Umgebungen so erfolgen:

$$
\begin{aligned}
R \bowtie^{\epsilon_1,\ldots,\epsilon_d}_{y_1,\ldots,y_d} S := \Big\{(r \times s) \in (R \times S) \;\Big|\; & r \in U_R^{\epsilon_1,\ldots,\epsilon_d}(s) \;\wedge\; s \in U_S^{\epsilon_1,\ldots,\epsilon_d}(r) \;\wedge \\
& ((\neg\exists(r \times s') \in (R \times S) : \\
& (r \in U_R^{\epsilon_1,\ldots,\epsilon_d}(s') \;\wedge\; s' \in U_S^{\epsilon_1,\ldots,\epsilon_d}(r) \;\wedge \\
& (r \times s') \prec_{y_1,\ldots,y_d} (r \times s))) \;\vee \\
& (\neg\exists(r' \times s) \in (R \times S) : \\
& (r' \in U_R^{\epsilon_1,\ldots,\epsilon_d}(s) \;\wedge\; s \in U_S^{\epsilon_1,\ldots,\epsilon_d}(r') \;\wedge \\
& (r' \times s) \prec_{y_1,\ldots,y_d} (r \times s))))\Big\}
\end{aligned}
$$

Es wird darauf hingewiesen, dass eine Bedingung der Form $r \in U_R^{\epsilon_1,\ldots,\epsilon_d}(s)$ die jeweilige komplementäre Bedingung $s \in U_S^{\epsilon_1,\ldots,\epsilon_d}(r)$ impliziert. Dies rührt daher, dass davon ausgegangen wird, dass die Maximalabstände ϵ_i für die einzelnen Join-Dimension $i \in \{1,\ldots,d\}$ jeweils für alle Tupel $r \in R$ und $s \in S$ gleich sind. Somit besitzt jedes Tupel $r \in R$ bzw. $s \in S$ die gleiche Ausdehnung seiner ϵ-Umgebung, was z. B. in Abbildung 3.5 durch die beiden deckungsgleichen Rahmen für die ϵ-Umgebungen der Punkte A und B zum Ausdruck kommt. Damit können die vorgenannten alternativen Definitionen der Varianten des Bestmatch-Joins mit Einschränkungen, die auf den ϵ-Umgebungen basieren, dadurch vereinfacht werden, dass von je zwei derartigen Bedingungen eine weggelassen wird. Aus Gründen der Vollständigkeit wurden sie hier jedoch belassen.

Kapitel 4

Algorithmen zur Berechnung des ϵ-LOBMJ

Nachdem das vorangegangene Kapitel den Bestmatch-Join auf Datenströmen eingeführt hat, behandelt dieses Kapitel nun Algorithmen zur Berechnung des Left-Outer-Bestmatch-Joins mit Einschränkungen, kurz ϵ-LOBMJ genannt. Wie bereits angedeutet, handelt es sich bei dieser Variante des Bestmatch-Joins um diejenige mit der größten praktischen Relevanz. Außerdem kann der ϵ-LOBMJ auf Datenströmen unter bestimmten Bedingungen korrekt berechnet werden, was diesen Ansatz von der approximativen Berechnung aus Abschnitt 3.2.1 abhebt. Zunächst wird ein naives Verfahren vorgestellt, dass zwar den ϵ-LOBMJ korrekt berechnet, aber nicht sehr effizient und außerdem auf Datenströmen nicht praktikabel ist. Es dient in der Folge als Vergleichsinstanz, insbesondere für die Leistungsmessungen in Kapitel 6. Danach wird ein fensterbasiertes Verfahren beschrieben, das auf der Basis der Einschränkung des Suchraums und unter der Voraussetzung einer Vorsortierung der Eingabedaten arbeitet und deutlich bessere Eigenschaften besitzt. Schließlich gehen weitere Ausführungen dieses Kapitels auf Erweiterungen und Abwandlungen der fensterbasierten Methode ein, die diese noch leistungsfähiger und universeller einsetzbar machen. So werden die Sortierung nach Epsilon-Grid-Ordnung und der zugehörige Scheduling-Algorithmus zur Reduzierung der Anzahl an Seitenzugriffen auf den Hintergrundspeicher sowie eine Abwandlung der fensterbasierten Methode für die Berechnung des eingeschränkten Left-Outer-Bestmatch-Joins auf punktierten Datenströmen behandelt. Zu einem Teil der aufgeführten Algorithmen finden sich auch Erläuterungen in [KS02a].

4.1 Naiver Algorithmus

Das naive Verfahren zur Berechnung des Bestmatch-Joins besteht aus einem simplen Nested-Loops Algorithmus. Die von ihm erzielte Leistung dient als Vergleichsreferenz zur Bewertung der durch bessere Methoden erreichbaren Leistungssteigerung. Außerdem bildet der Nested-Loops Ansatz den Kern aller Algorithmen zur Berechnung des Bestmatch-Joins auf Datenströmen in dieser Arbeit.

4.1.1 Algorithmus

Zunächst wird eine Methode zur Berechnung des normalen Bestmatch-Joins angegeben. Diese ist in Algorithmus 4.1 in Pseudocode dargestellt. Der Algorithmus erhält als Eingaben die beiden zu kombinierenden Relationen R und S, hier in der Regel Datenströme, und die Vergleichsordnung $\prec_{y_1,\dots,y_d}$. Daraus berechnet er den Bestmatch-Join $R \bowtie_{y_1,\dots,y_d} S$. Mittels vier verschachtelter Schleifen vergleicht er jede Kombination von Tupeln $r \in R$ und $s \in S$ mit jeder anderen. Wird das Tupelpaar $(r \times s)$ von keinem anderen Paar $(r' \times s')$ dominiert, so wird es als Teil des Ergebnisses ausgegeben. Der Algorithmus terminiert, wenn alle möglichen Paare von Tupeln mit jeweils allen anderen Paaren verglichen worden sind. Die Ausgabe entspricht dann dem Ergebnis des Bestmatch-Joins von R und S.

Geht man davon aus, dass beide Eingaberelationen R und S jeweils gleich viele Datenobjekte enthalten und n die Anzahl der Datenobjekte je Relation bezeichnet, so ergibt sich für diesen Algorithmus offensichtlich eine Komplexität von $\mathcal{O}(n^4)$. Diese resultiert aus den vier verschachtelten Schleifen über die beiden Eingaben. Dieses Ergebnis deckt sich mit der Erkenntnis, dass der Bestmatch-Join die Skyline auf dem Kreuzprodukt der beiden Eingaberelationen berechnet. Die Skyline lässt sich mittels eines Nested-Loops Algorithmus in $\mathcal{O}(k^2)$ berechnen, wobei k die Anzahl der Punkte in der Eingabe des Skyline-Operators bezeichnet. Setzt man für k nun n^2 ein, was der Anzahl an Punkten im Kreuzprodukt von R und S entspricht, so erhält man obige Komplexität für den naiven Nested-Loops Algoritmus zur Berechnung des Bestmatch-Joins.

Um den eigentlich interessierenden Left-Outer-Bestmatch-Join berechnen zu können, muss Algorithmus 4.1 geringfügig abgeändert werden. Da nun für jedes einzelne Tupel der linken Eingabe die am besten dazu passenden Tupel der rechten Eingabe gesucht werden, muss die zweite Schleife über die Relation R in der vierten Zeile des Algorithmus entfallen.

Algorithmus 4.1 Naiver Algorithmus zur Berechnung des Bestmatch-Joins

Eingabe: Eingaberelationen R und S, partielle Ordnung $\prec_{y_1,\dots,y_d}$
Ausgabe: $R \bowtie_{y_1,\dots,y_d} S$
  for all $r \in R$ **do**
2:    **for all** $s \in S$ **do**
      $dominated \leftarrow false$
4:      **for all** $r' \in R$ **do**
        for all $s' \in S$ **do**
6:          **if** $(r' \times s') \prec_{y_1,\dots,y_d} (r \times s)$ **then**
            $dominated \leftarrow true$
8:          **end if**
        end for
10:      **end for**
      if $\neg dominated$ **then**
12:        output $(r \times s)$
      end if
14:    **end for**
  end for

Damit vergleicht das Verfahren jede mögliche Kombination aus dem aktuellen Tupel $r \in R$ und einem Tupel $s \in S$ mit jeder anderen solchen Kombination. Dies erfolgt wiederum für jedes $r \in R$. Ein Tupelpaar wird genau dann als Teil des Ergebnisses ausgegeben, wenn es von keinem Vergleichspaar dominiert wurde. Auch diese Methode terminiert, wenn alle Kombinationen für jedes Datenelement der linken Eingabe abgearbeitet worden sind. Das Verfahren ist in Algorithmus 4.2 dargestellt.

Unter den gleichen Annahmen wie für Algorithmus 4.1 besitzt die Methode für den Left-Outer-Bestmatch-Join eine Komplexität von $\mathcal{O}(n^3)$. Dies ergibt sich direkt aus der Reduzierung der Anzahl der verschachtelten Schleifen von vier auf drei.

Die beiden in diesem Abschnitt eingeführten Algorithmen lassen sich ein wenig optimieren. Da ein Tupelpaar $(r \times s)$ nicht mehr im Ergebnis enthalten sein kann, sobald es von mindestens einem anderen Paar dominiert wurde, können alle verbleibenden Vergleiche des Paares $(r \times s)$ mit anderen Paaren unterbleiben, sobald das erste dominierende Vergleichspaar gefunden wurde. In den Algorithmen kann dies dadurch berücksichtigt werden, dass als letzter Befehl in dem if-Block, der die beiden Tupelpaare vergleicht, ein break-Befehl eingeführt wird, der die innerste Schleife sofort beendet. In Algorithmus 4.1 ist dieser Befehl unmittelbar nach Zeile 7 einzufügen, in Algorithmus 4.2 unmittelbar nach Zeile 6, jeweils direkt nach dem Setzen des *dominated*-Flags. In Algorithmus 4.1 kann außerdem auch die zweitinnerste Schleife sofort beendet werden.

Die beiden Algorithmen dieses Kapitels berechnen in der dargestellten Form den Bestmatch-Join bzw. Left-Outer-Bestmatch-Join ohne Einschränkungen. Will man das Ergebnis des jeweiligen Joins mit Einschränkungen berechnen, so dürfen nur solche Paare von Tupeln in das Ergebnis aufgenommen werden, die die Einschränkungen erfüllen. Dies kann dadurch erreicht werden, dass in allen Schleifen über die rechte Eingabe S nur solche Tupel $s \in S$ in Betracht gezogen werden, die innerhalb der ϵ-Umgebung des zugehörigen Tupels der linken Eingabe liegen. In Algorithmus 4.1 ist dieses zugehörige Tupel der linken

Algorithmus 4.2 Naiver Algorithmus zur Berechnung des Left-Outer-Bestmatch-Joins

Eingabe: Eingaberelationen R und S, partielle Ordnung $\prec_{y_1,\ldots,y_d}$
Ausgabe: $R ⟕_{y_1,\ldots,y_d} S$

    for all $r \in R$ **do**
2:    **for all** $s \in S$ **do**
        dominated $\leftarrow$ *false*
4:      **for all** $s' \in S$ **do**
          if $(r \times s') \prec_{y_1,\ldots,y_d} (r \times s)$ **then**
6:          *dominated* $\leftarrow$ *true*
          end if
8:      **end for**
        if $\neg$*dominated* **then**
10:       output $(r \times s)$
        end if
12:   **end for**
    end for

Eingabe r für s und r' für s'. In Algorithmus 4.2 ist es r sowohl für s als auch für s'. Die Schleife muss also solche Tupel aus S überspringen, die außerhalb der ϵ-Umgebung des entsprechenden Tupels aus R liegen.

4.1.2 Diskussion

Der große Vorteil des naiven Nested-Loops Algorithmus liegt in seiner Einfachheit. Er ist sowohl leicht zu verstehen als auch leicht zu implementieren. Da er keinerlei komplizierte algorithmische Bestandteile besitzt, kommt er mit sehr wenig Zusatzaufwand aus und arbeitet deshalb auf kleinen Eingaben durchaus schnell. Zudem ist er auch als Bestandteil von speziellen Algorithmen zur Berechnung von besten Zuordnungen auf Datenströmen einsetzbar.

Allerdings ist er ohne besondere Erweiterungen nicht für Berechnungen auf Datenströmen geeignet, da es sich beim Nested-Loops Algorithmus offensichtlich um ein blockierendes Verfahren handelt. Für den Left-Outer-Bestmatch-Join muss zumindest die komplette rechte Eingabe im Hauptspeicher oder materialisiert auf dem Hintergrundspeicher vorliegen, bevor das erste Ergebnistupel erzeugt werden kann. Im Falle des normalen Bestmatch-Joins müssen sogar beide Eingaben vorab vollständig vorhanden sein. Damit ist dieser Algorithmus bei unendlichen Datenströmen überhaupt nicht benutzbar. Darüber hinaus wird er bei größeren Eingaben sehr langsam, da relativ viele verschachtelte Schleifen über die kompletten Eingaberelationen laufen. Für das Ziel der Berechnung von besten Zuordnungen von Objekten zweier Datenströme ist der Algorithmus deshalb in seiner reinen Form eher weniger geeignet.

4.2 Fensterbasierter Algorithmus

Um die Nachteile des naiven Nested-Loops Algorithmus auszuschalten und ein pipelinefähiges Verfahren zu erhalten, das auch auf möglicherweise unendlichen Datenströmen beste Zuordnungen berechnen kann und dabei sowohl einen kontinuierlichen Datenstrom korrekter Ergebnisse erzeugt als auch eine akzeptable Geschwindigkeit an den Tag legt, wurde der fensterbasierte Algorithmus entwickelt. In den folgenden Abschnitten wird zunächst ein Überblick über die Voraussetzungen für die Anwendbarkeit dieses Verfahrens gegeben. Einige Beispiele sollen die praktische Einsetzbarkeit veranschaulichen. Anschließend wird der Algorithmus vorgestellt und beschrieben. Schließlich fasst eine Diskussion die Vor- und Nachteile des fensterbasierten Ansatzes zusammen.

4.2.1 Voraussetzungen und Beispiele

Damit die Anwendbarkeit des fensterbasierten Algorithmus gewährleistet ist, müssen vorab die folgenden beiden Voraussetzungen erfüllt sein. Nur wenn beide Voraussetzungen gleichzeitig erfüllt sind, erzeugt das Verfahren mit Sicherheit korrekte Ergebnisse.

- Zunächst muss der Suchraum des Joins eingeschränkt werden. Die Tupel der im Endergebnis enthaltenen Tupelpaare müssen also innerhalb der ϵ-Umgebung des jeweiligen Partnertupels liegen, wie in Kapitel 3.2.2 beschrieben. Das ist nötig, um im Algorithmus später die Datenfenster entsprechend organisieren und aktualisieren zu können. Ohne Einschränkungen müssten die Datenfenster den gesamten Datenbestand auf einmal aufnehmen, was sich negativ auf die Leistung des Algorithmus auswirken und außerdem seine Anwendbarkeit auf unendlichen Datenströmen verhindern würde.

- Weiterhin müssen die Datenströme die Eingabedaten bereits vorsortiert liefern. Die Vorsortierung verlangt die aufsteigende[1] Sortierung der gelieferten Tupel nach den Werten eines gemäß der vorhergehenden Voraussetzung auf eine ϵ-Umgebung eingeschränkten Attributs. Dadurch wird sichergestellt, dass alle nachfolgenden Tupel für dieses Attribut keinen kleineren Wert besitzen als das letzte bis dahin aus dem Datenstrom gelesene Tupel. Anhand der sortierten Dimension verschiebt der Algorithmus dann die Datenfenster. Durch die Sortierung wird garantiert, dass die Fenster nur in einer Richtung, nämlich in der Richtung aufsteigender Attributwerte des sortierten Attributs, verschoben werden. Dadurch kann das Verfahren keine relevanten Tupelkombinationen übersehen, weil für ein einmal aus den Datenfenstern entferntes Tupel später kein Join-Partner mehr auftauchen kann, der in der sortierten Dimension innerhalb der ϵ-Umgebung des entfernten Tupels liegt.

Die angesprochene Vorsortierung genügt Definition 3.2 der Sortierung, wobei die Relation $\trianglelefteq$ durch die Relation $\leq$ auf den reellen Zahlen zu ersetzen ist. Im Folgenden sei o. B. d. A. die Dimension $i = 1$ die vorsortierte Dimension und damit y_1 das Attribut, nach dem die Tupel aus den Eingabedatenströmen aufsteigend sortiert sind. Mit den Bezeichnungen aus Abschnitt 3.1.1 und $r_i \trianglelefteq r_j \Leftrightarrow r_i.y_1 \leq r_j.y_1$ ist dann die Vorsortierung wie folgt definiert:

$$\forall i, j \in \mathbb{N} : i < j \Leftrightarrow r_i.y_1 \leq r_j.y_1$$

Betrachtet man das Arbeitsmarkt-Beispiel aus der Einleitung unter der Voraussetzung der vorsortierten Eingabedatenströme, so kann man sich ein Szenario vorstellen, in dem mehrere, auf unterschiedliche Regionen verteilte Betriebe ihre Stellenangebote und regionale Arbeitsämter die Profile von Bewerbern z. B. über das Internet verfügbar machen. Dann können die Datenströme der verschiedenen Betriebe und Arbeitsämter an einem Zielknoten, etwa einem Terminal eines Mitarbeiters in einem der Arbeitsämter, mittels eines Bestmatch-Joins ausgewertet werden. Die Vorsortierung der Datenströme könnte dabei u. a. nach Alter oder einer abstrakten numerischen Einstufung der vorhandenen Qualifikation erfolgen. Sucht man nun zu jedem Bewerber die am besten passenden Stellen oder umgekehrt, so liegt ein Left-Outer-Bestmatch-Join vor.[2]

[1] Grundsätzlich sind die nachfolgend vorgestellten Verfahren auch auf entsprechend absteigend sortierten Datenströmen anwendbar. Sie müssen lediglich an manchen Stellen geringfügig an die veränderte Sortierreihenfolge angepasst werden. Wichtig ist allerdings, dass beide Eingabedatenströme eines Joins jeweils gleichartig sortiert sind. Da die nötigen Anpassungen offensichtlich sind, gehen die folgenden Ausführungen ausschließlich von aufsteigenden Sortierungen aus.

[2] Falls ein Right-Outer-Bestmatch-Join vorliegt, so kann dieser natürlich einfach durch Vertauschen der Eingaberelationen in einen Left-Outer-Bestmatch-Join überführt werden. Diese beiden Join-Varianten sind insofern äquivalent. In dieser Arbeit wird per Konvention immer auf den Left-Outer-Bestmatch-Join Bezug genommen.

Es folgen nun weitere Beispiele für Anwendungsszenarien, welche die Voraussetzungen des fensterbasierten Algorithmus erfüllen.

Beispiel 4.1
Man betrachte einen Wetterdienst, der zum Zwecke der Vorhersage der Niederschlagswahrscheinlichkeit Sensoren zur Messung von Luftdruck und Luftfeuchtigkeit über das zu beobachtende Gebiet verteilt aufgestellt hat. Dabei ist nicht davon auszugehen, dass je ein Sensor für Luftdruck und Luftfeuchtigkeit am selben Platz installiert wurde. Vielmehr können die Sensoren für die beiden Messgrößen auch unabhängig voneinander platziert werden. Außerdem seien die Messzeitpunkte eines jeden Sensors nicht vorab festgelegt. So soll sich das Zeitintervall zwischen zwei aufeinanderfolgenden Messungen eines Sensors verkürzen, wenn der Messwert der letzten Messung relativ stark von dem der vorletzten Messung abweicht. Dadurch kann die Veränderung des Messwertes besser erfasst und verfolgt werden. Entsprechend verlängert sich das Messintervall, wenn aufeinanderfolgende Messwerte nur geringe Abweichungen voneinander aufweisen. Durch diese sich dynamisch an die jeweilige Situation anpassenden Messzeitpunkte ist eine statische Zuordnung von Luftdruck- und Luftfeuchtigkeitssensoren nach Messzeitpunkt und Standort nicht möglich. Jedes Messgerät schickt bei jeder Messung ein Datenpaket bestehend aus dem Zeitpunkt der Messung, dem Messwert und dem in der Form zweidimensionaler Koordinaten dargestellten Standort der Messanlage auf ein Netzwerk. Die Daten für Luftdruck und Luftfeuchtigkeit werden dabei in geeigneter Weise gebündelt, so dass letztendlich für jede der beiden Messgrößen ein eigener unabhängiger Datenstrom von Messwerten vorliegt. Eine denkbare Lösung hierfür ist die Funkübertragung der Daten von jedem Sensor zu einem die Informationen verarbeitenden Zentralknoten, unter der Voraussetzung, dass zuerst abgeschickte Datenpakete unabhängig von der räumlichen Entfernung zwischen Sender und Empfänger auch zuerst am Zentralknoten ankommen. Die aufsteigende Sortierung der Datenströme nach einem Attribut ergibt sich dann in natürlicher Weise durch die Zeit, zu der die Messdaten erzeugt wurden. Ziel des Wetterdienstes ist es nun, für jeden Luftdruckmesswert diejenigen Luftfeuchtigkeitsmesswerte zu finden, die einen möglichst geringen zeitlichen und räumlichen Abstand von dem jeweiligen Luftdruckwert haben, um auf dieser Basis gute Vorhersagen treffen zu können. Es wird also der Left-Outer-Bestmatch-Join der beiden Datenströme auf den Join-Attributen Zeit und Standort gebildet. Folgende Einschränkungen sind für aussagekräftige Ergebnisse auf diesen Join-Dimensionen sinnvoll:

- Der zeitliche Abstand der Messzeitpunkte für die Messwerte von Luftdruck und Luftfeuchtigkeit in einem Ergebnispaar darf nicht größer als zehn Minuten sein.
- Der räumliche Abstand der Sensoren darf nicht größer als 1000 Meter sein.

Ohne diese Einschränkungen könnten im Ergebnis Paare enthalten sein, deren Messwerte zwar von zwei eng benachbarten Messstationen stammen, die aber zu stark unterschiedlichen Zeiten gemessen wurden, z. B. mit einem zeitlichen Abstand von mehreren Stunden. Damit wäre ein solches Paar von Messwerten natürlich nicht aussagekräftig, da sich etwa der Luftdruck in der langen Zwischenzeit bis zur Messung der Luftfeuchtigkeit schon wieder grundlegend geändert haben könnte. Dasselbe gilt auch für Paare, deren Messwerte zwar zu ähnlichen Zeiten generiert wurden, deren zugehörigen Sensoren aber einen sehr großen räumlichen Abstand voneinander haben, beispielsweise mehrere Kilometer. In diesem Fall gehören die beiden Messwerte zu zwei verschiedenen Regionen mit eventuell völlig unterschiedlichen Wetterverhältnissen zum selben Zeitpunkt. Damit macht auch die Kombination zweier solcher Messwerte keinen Sinn.

Seien t, x und y die Attribute für die Zeit und die beiden Koordinaten des Sensorstandortes und $r, r' \in R$ bzw. $s, s' \in S$ Datenpakete der Luftdruck- bzw. Luftfeuchtigkeitssensoren. Dann kann man die Vergleichsordnung für den Left-Outer-Bestmatch-Join formal so definieren:

$$\begin{aligned}(r \times s) \prec_{t,x,y} (r' \times s') \quad \Leftrightarrow \quad & (\forall i \in \{t, x, y\} : |r.i - s.i| \leq |r'.i - s'.i|) \ \wedge \\ & (\exists i \in \{t, x, y\} : |r.i - s.i| < |r'.i - s'.i|)\end{aligned}$$

Damit wird jede der beiden Dimensionen des Standorts separat betrachtet. Die Berechnung des Abstands zweier Messpunkte erfolgt also individuell für jede Dimension. Stattdessen kann man den Standort auch als eine einzige Dimension auffassen und den Abstand mit der euklidischen Metrik berechnen:

$$\begin{aligned}(r \times s) \prec_{t,x,y} (r' \times s') \quad \Leftrightarrow \quad & (|r.t - s.t| \leq |r'.t - s'.t|) \ \wedge \\ & \Big(\sqrt{(r.x - s.x)^2 + (r.y - s.y)^2} \ \leq \\ & \sqrt{(r'.x - s'.x)^2 + (r'.y - s'.y)^2}\ \Big) \ \wedge \\ & \Big((|r.t - s.t| < |r'.t - s'.t|) \ \vee \\ & \Big(\sqrt{(r.x - s.x)^2 + (r.y - s.y)^2} \ < \\ & \sqrt{(r'.x - s'.x)^2 + (r'.y - s'.y)^2}\ \Big)\Big)\end{aligned}$$

Abbildung 4.1 auf der nächsten Seite zeigt eine abstrakte Visualisierung dieses Beispiels. In dem Gebiet G sind die drei Luftdrucksensoren p_1, p_2 und p_3 sowie die drei Luftfeuchtigkeitssensoren h_1, h_2 und h_3 aufgestellt. Sie senden ihre Daten gebündelt in den Datenströmen R bzw. S an einen Zentralknoten, der aus den beiden Eingaben den Left-Outer-Bestmatch-Join berechnet und das Ergebnis an einen Folgeoperator oder an den Benutzer weitergibt. □

Beispiel 4.2
Ein anderes Beispiel für die Einsetzbarkeit des eingeschränkten Left-Outer-Bestmatch-Joins auf vorsortierten Eingabedatenströmen ist die Materialprüfung in einem Produktionsbetrieb. Der Betrieb produziere zwei verschiedene Bauteile, die im weiteren Produktionsprozess zusammengefügt werden sollen. Die einzelnen Bauteile laufen dabei von einem Fließband. Weiter sei die Genauigkeit, mit der die Teile produziert werden können, stark eingeschränkt. Dies kann z. B. darin gründen, dass der Fertigungsprozess mindestens einen Fertigungsschritt enthält, der nicht für jedes zu bearbeitende Teil genau dasselbe Ergebnis erzeugt. Eine mögliche Erklärung dafür ist, dass der betreffende Fertigungsschritt von

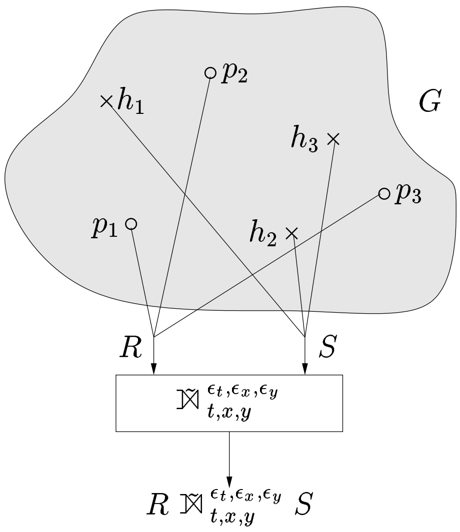


Abb. 4.1: Wetterdienst-Beispiel für den ϵ-LOBMJ

verschiedenen Arbeitern in Handarbeit ausgeführt wird. Aufgrund der abweichenden Maße der Bauteile passen nicht alle Paare von Teilen gleich gut zusammen. Deshalb wird vor dem Zusammensetzen der Einzelteile für jedes Bauteil der einen Baureihe das am besten dazu passende Teil der anderen Baureihe gesucht. Zu diesem Zweck werden die von den beiden Fließbändern laufenden Teile elektronisch vermessen. Die Messwerte der beiden Bänder bilden zwei Datenströme, deren Datenelemente alle gemessenen Daten sowie den Zeitpunkt der Vermessung des zugehörigen Teils beinhalten. Anhand der Messdaten dieser beiden Datenströme werden die Bauteile einander zugeordnet. Dazu wird ein Left-Outer-Bestmatch-Join auf den Daten der beiden Ströme berechnet. Wenn $y_1, \ldots, y_d$ die einzelnen Messwerte eines Bauteils und R bzw. S wieder die Datenströme bezeichnen, handelt es sich dabei also um den Join $R ⟕^{\epsilon_1,\ldots,\epsilon_d}_{y_1,\ldots,y_d} S$. Die Maximalabstände $\epsilon_1, \ldots, \epsilon_d$ geben an, wie groß die Abweichungen der jeweiligen Messwerte zweier zu kombinierender Teile höchstens sein dürfen, damit die Teile überhaupt noch zusammenpassen. Die Sortierung der Datenströme ergibt sich implizit aus den Messzeitpunkten, zu denen die Messdaten der einzelnen Bauteile erzeugt wurden. Aus diesem Beispiel geht auch hervor, dass es sich bei dem Attribut, nach dem die Eingabedatenströme aufsteigend sortiert sind, nicht notwendigerweise um ein Join-Attribut des Bestmatch-Joins handeln muss. Allerdings ist es erforderlich, dass auch auf dem sortierten Attribut ein einschränkender Maximalabstand ϵ existiert, da dieser für die Organisation der Datenfenster des fensterbasierten Algorithmus benötigt wird. Die Festlegung dieses ϵ-Wertes hat Einfluss auf das Join-Ergebnis. □

Beispiel 4.3

Auch im Bereich der Finanzwirtschaft finden sich Anwendungen für den fensterbasierten Algorithmus zur Berechnung des eingeschränkten Left-Outer-Bestmatch-Joins. Hier fällt etwa die Verarbeitung von Börsenkursen ins Auge. An der Börse liefern Datenströme ständig die zum aktuellen Zeitpunkt gültigen Aktienkurse. Die Daten aus den Strömen bestehen aus einem Zeitstempel, dem Namen der betreffenden Gesellschaft und deren zum angegebenen Zeitpunkt gültigen Aktienkurs. Der Zeitstempel für einen bestimmten Kurswert liefert wiederum implizit die aufsteigende Sortierung der Datenströme nach dem

Zeit-Attribut. Für einen Unternehmer oder Aktionär kann es zum Herausfinden ähnlich starker Wettbewerber interessant sein, diejenigen Aktiengesellschaften zu finden, deren Aktien in etwa den gleichen Kurs haben wie seine eigenen. Zu diesem Zweck lässt sich der Left-Outer-Bestmatch-Join mit den Aktienkursen der Wertpapiere des Unternehmers bzw. Aktionärs als linker Eingabe und den Kursen der interessanten Konkurrenzunternehmen als rechter Eingabe berechnen. Join-Attribute sind die Zeit und der Wertpapierkurs. Selbstverständlich sind aber noch weitere Join-Attribute denkbar, sofern deren Daten von den Datenströmen mitgeliefert werden, wie z. B. die Anzahl der Mitarbeiter eines Unternehmens oder der jährliche Umsatz und der Gewinn vor und nach Steuern, usw. Einschränkungen auf dem Zeit- und dem Kurs-Attribut sorgen dafür, dass nur Wettbewerber im Ergebnis enthalten sind, die zu ähnlichen Zeiten ähnliche Aktienkurse aufweisen. □

In obigen Anwendungsfällen sind herkömmliche, z. B. mit einer Bewertungsfunktion arbeitende Verfahren nicht sinnvoll anwendbar, da sie in der Regel keine zufriedenstellenden Ergebnisse erzeugen. Das liegt daran, dass es im Allgemeinen kein einzelnes, optimales Paar von Tupeln gibt. Stattdessen sind mehrere unvergleichbare Tupelpaare von Interesse, von denen jedes in mindestens einer Join-Dimension das Beste darstellt. Es existiert aber meist kein Paar, das in allen Dimensionen dominant ist. Bezogen auf das Beispiel 4.1 des Wetterdienstes kann etwa eine Kombination von Messwerten bezüglich der Zeit-Dimension optimal sein und dennoch von einem anderen Paar, dessen Messwerte zwar zeitlich weiter auseinanderliegen, dafür aber räumlich enger benachbarten Sensoren entstammen, in der Standort-Dimension dominiert werden. Für die vorgestellten Anwendungsszenarien ist deshalb der jede Join-Dimension individuell betrachtende Bestmatch-Join die Methode der Wahl.

4.2.2 Algorithmus

Dieser Abschnitt behandelt den fensterbasierten Algorithmus zur Berechnung des Left-Outer-Bestmatch-Joins mit Einschränkungen, der auch in dem in Kapitel 5 beschriebenen Prototyp implementiert wurde. Dabei sind zunächst zwei verschiedene Fensterstrategien zu unterscheiden.

Ein erster Ansatz für die Berechnung des ϵ-LOBMJ der beiden Datenströme R und S sieht für jede Eingabe ein Datenfenster W_R bzw. W_S vor. Dabei hält das Datenfenster W_R immer nur das gerade aktuelle Tupel $r \in R$, es gilt also $W_R = \{r\}$. Das Fenster W_S enthält alle Tupel $s \in S$, die bezüglich der sortierten Dimension y_1 innerhalb der ϵ-Umgebung von r liegen, also $W_S = \{s \in S \mid r.y_1 - \epsilon_1 \leq s.y_1 \leq r.y_1 + \epsilon_1\}$. Auf diesen Fenstern wird der ϵ-LOBMJ mittels des Nested-Loops Algorithmus berechnet, was offensichtlich das korrekte Ergebnis für das aktuelle Tupel $r \in R$ der linken Eingabe erzeugt. Danach wird r aus W_R entfernt, durch das nächste Tupel $r' \in R$ ersetzt und W_S entsprechend aktualisiert. Dazu entfernt das Verfahren alle $s' \in W_S$ aus dem Datenfenster der rechten Eingabe, für die $s' < r'.y_1 - \epsilon_1$ gilt und liest alle $s'' \in S$ neu aus dem Datenstrom S in W_S ein, welche die Bedingung $s'' \leq r'.y_1 + \epsilon_1$ erfüllen. Da die Datenströme nach den Werten des Attributs y_1 aufsteigend sortiert sind, ist dies durch sequenzielles Lesen und Einfügen aufeinanderfolgender Tupel aus S bis zum ersten Auftreten eines Tupels mit

$s'' > r'.y_1 + \epsilon_1$ möglich. Dieses zuletzt gelesene Tupel wird selbstverständlich vorerst nicht in das Datenfenster W_S eingefügt. Außerdem garantiert die Vorsortierung, dass immer alle für das aktuelle $r \in W_R$ relevanten Tupel aus S in dem Fenster W_S enthalten sind. Aus obiger Erklärung ist ferner ersichtlich, dass im Fall $r' = r$ keine Änderung an W_S erforderlich ist. Nach der Aktualisierung der Datenfenster kann wiederum der Join berechnet werden. Der Vorgang wiederholt sich so lange, bis alle $r \in R$ abgearbeitet sind. Ein Problem dieses Ansatzes ist, dass die Datenfenster in der Regel zu groß sind, um vollständig in den Hauptspeicher zu passen. Dies lässt sich durch das Materialisieren der Fenster auf dem Hintergrundspeicher lösen. Da aber viele Tupel in W_S im Allgemeinen öfter gebraucht werden, kann es zu zahlreichen ineffizienten Ladevorgängen kommen.

Um diesem Problem zu begegnen, ist eine Abwandlung des ersten Ansatzes möglich. Hierbei erhält auch der Datenstrom R ein größeres Datenfenster, das zu einem Zeitpunkt mehr als ein Tupel aus R halten kann. Dieses Verfahren speichert den Inhalt der beiden Datenfenster ebenfalls auf dem Hintergrundspeicher. Der Algorithmus lädt dann Teile der Fenster in geeigneter Weise in den Hauptspeicher, um die Berechnung des Join-Ergebnisses durchzuführen. Um dies effizient und mit einer möglichst geringen Anzahl an Ladevorgängen realisieren zu können, greift die Methode dabei auf eine leicht abgewandelte Version der in [BBKK01] vorgestellten Epsilon-Grid-Ordnung, kurz EGO genannt, zurück.[1] Auf die Ordnung geht Abschnitt 4.3.1 genauer ein. In dieser Arbeit findet die hier skizzierte zweite Fensterstrategie Verwendung. Sie wird nachfolgend genauer beschrieben.

Die grundlegende Architektur des Verfahrens ist in Abbildung 4.2 dargestellt. Wie auch beim approximativen Bestmatch-Join arbeitet der Join-Operator auf zwei Datenfenstern W_R und W_S, die ihren Inhalt aus den beiden Eingabedatenströmen R und S beziehen. Zu beachten ist hierbei, dass beide Datenfenster synchron organisiert sind, also gemeinsame Unter- und Obergrenzen $minRS$ und $maxRS$ für den Wert des sortierten Attributs y_1 besitzen. Diese Grenzen geben an, welchen Teil der Eingaben die Datenfenster enthalten. Alle Tupel aus R, deren y_1-Wert zwischen $minRS$ und $maxRS$ liegt, sind in W_R enthalten. Analoges gilt für Tupel aus S und das Datenfenster W_S. Tupel mit einem kleineren Wert für y_1 als der aktuelle Wert von $minRS$ oder einem größeren als der aktuelle Wert von $maxRS$ können und dürfen nicht im zugehörigen Datenfenster enthalten sein. Auch dies gilt für beide Datenfenster und beide Datenquellen gleichermaßen. Der Inhalt der Datenfenster ist somit formal folgendermaßen definiert:

$$W_R = \{r \in R \mid minRS \leq r.y_1 \leq maxRS\}$$

$$W_S = \{s \in S \mid minRS \leq s.y_1 \leq maxRS\}$$

Die Fenster werden mit fortschreitender Berechnung entlang der sortierten Dimension verschoben, wobei sich die neuen Unter- und Obergrenzen jeweils aus den y_1-Werten der Tupel im linken Datenfenster ergeben und auf beide Fenster angewandt werden. Aufgrund der aufsteigend vorsortierten Attributwerte erfolgt das Verschieben nur in Richtung aufsteigender Werte für $minRS$ und $maxRS$. Die Join-Berechnung greift außer auf die Tupel

[1] Die Bezeichnungen Epsilon-Grid-Ordnung und EGO beziehen sich im Rahmen dieser Arbeit, sofern nicht explizit anders angegeben, immer auf die in Kapitel 4.3.1 definierte modifizierte Version der Ordnung.

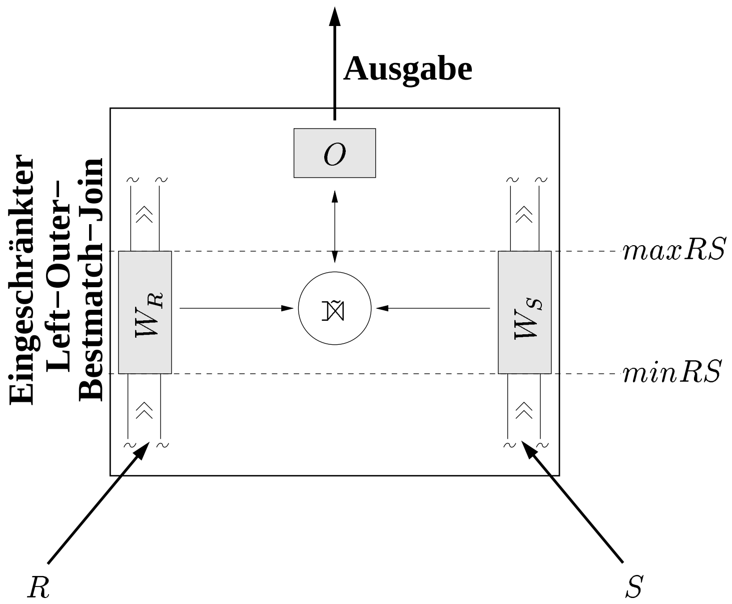


Abb. 4.2: Architektur des fensterbasierten Algorithmus zur Berechnung des ϵ-LOBMJ auf Datenströmen

aus den beiden Datenfenstern auch auf die im Zwischenergebnis O enthaltenen vorläufigen Ergebnispaare zurück. In jedem Berechnungsschritt bildet der Algorithmus Paare von Tupeln aus den aktuellen Datenfenstern und vergleicht diese mit bereits generierten Tupelpaaren aus O. Entsprechend der sich dabei ergebenden Verhältnisse werden dominierte Paare aus O entfernt und neue dominierende oder unvergleichbare Paare hinzugefügt. Danach zieht das Verfahren die beiden Datenfenster in y_1-Richtung nach, verwirft dabei Tupel, die aus den Fenstern herausfallen, und liest aus den Datenströmen neue ein. Auch der Puffer O wird entsprechend nachgezogen. Die dabei aus seinen Grenzen herausfallenden Paare werden in das Endergebnis übernommen.

Die Abbildungen 4.3 bis 4.6 zeigen die Vorgehensweise beim Einlesen neuer Datenelemente, der Join-Berechnung und dem Nachziehen der Fenstergrenzen für ein Beispiel mit zweidimensionalen Punkten. In Bild 4.3 ist das erstmalige Einlesen der Datenfenster dargestellt. Die Höhe der beiden Fenster W_R und W_S beträgt stets genau ϵ_1. Alle Tupel $r \in R$ mit $minRS \leq r.y_1 \leq maxRS$ werden in das Datenfenster W_R aufgenommen. Analoges gilt für Tupel $s \in S$ und W_S. Dabei liest das Verfahren immer das nächste Tupel aus dem jeweiligen Datenstrom und überprüft, ob es noch in das aktuelle Fenster passt. Falls ja, wird es in das Fenster geschrieben und das nächste Tupel aus dem Strom gelesen. Anderenfalls ist der Einlesevorgang beendet und das zuletzt gelesene Tupel, das nicht mehr in das Fenster passt, wird bis zur nächsten Aktualisierung der Datenfenster gepuffert. Die aufsteigende Sortierung der Tupel nach y_1-Werten stellt sicher, dass das Verfahren kein in die Fenster passendes Datenelement übersieht. Wegen des begrenzt zur Verfügung stehenden Hauptspeichers werden die Tupel bereits während des Einlesevorgangs auf dem Hintergrundspeicher materialisiert. Hierfür sind verschiedene Materialisierungsvarianten einsetzbar, die später noch ausführlich beschrieben werden. Sind die Fenster mit allen relevanten Tupeln gefüllt, erfolgt die Join-Berechnung wie in Abbildung 4.4 gezeigt.

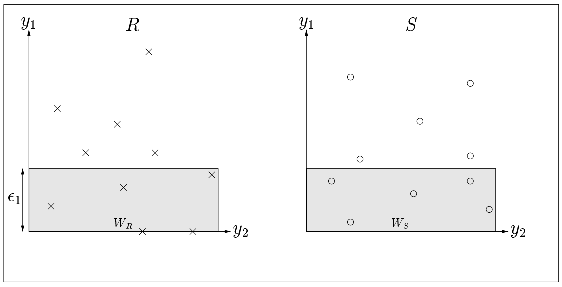


Abb. 4.3: Erstmaliges Einlesen der Datenfenster

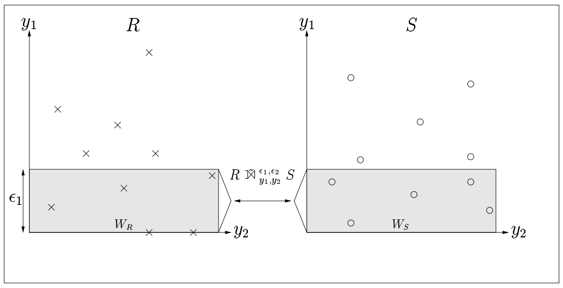


Abb. 4.4: Berechnung des ϵ-LOBMJ auf den Ausgangsfenstern

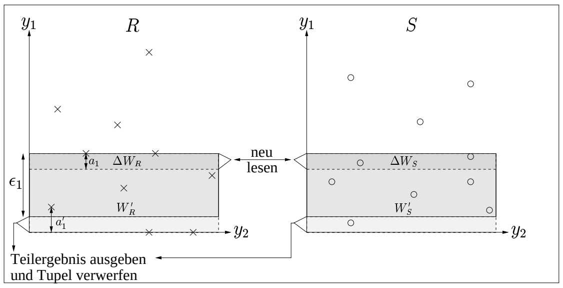


Abb. 4.5: Nachziehen der Fenster und Ausgeben des Teilergebnisses

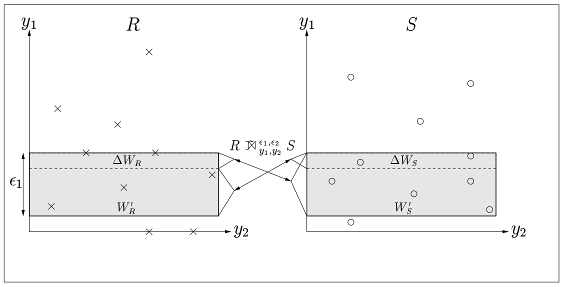


Abb. 4.6: Berechnung des ϵ-LOBMJ auf den aktualisierten Fenstern

Zu diesem Zweck werden die in Seiten auf dem Hintergrundspeicher abgelegten Tupel z. B. mittels des in Kapitel 4.3.2 eingeführten Scheduling-Algorithmus seitenweise geladen und von einer Variante des Nested-Loops Algorithmus verarbeitet. Nach Abschluss der Berechnung aktualisiert das Verfahren die gemeinsamen Fenstergrenzen $minRS$ und $maxRS$ der beiden Datenfenster W_R und W_S sowie deren Inhalt derart, dass die Fenster danach wieder genau die Tupel enthalten, deren y_1-Werte innerhalb der neuen Grenzen liegen. Dazu entfernt es zunächst alle Tupel r aus W_R, die auf der alten Untergrenze des Fensters liegen, für die also $r.y_1 = minRS$ gilt. In Bild 4.5 sind das die beiden untersten Tupel. Danach werden sowohl $minRS$ als auch $maxRS$ um das Minimum von a_1 und a'_1 auf $minRS'$ bzw. $maxRS'$ erhöht, es gilt also $minRS' = minRS + min(a_1, a'_1)$ und $maxRS' = maxRS + min(a_1, a'_1)$. Dabei bezeichnet a_1 den Abstand des gepufferten Tupels $r \in R$, das nicht mehr in das alte Fenster passte, von der alten Fensterobergrenze und a'_1 den Abstand des bezüglich y_1 kleinsten Tupels im Fenster von der alten Untergrenze. Es gilt also $a_1 = r.y_1 - maxRS$ und $a'_1 = min_{y_1}(W_R) - minRS$ mit der aus Definition 3.3 bekannten Festlegung $min_{y_1}(W_R) := min\{r.y_1 \mid r \in W_R\}$. Auch dies ist in Abbildung 4.5 dargestellt, wobei hier a_1 die Distanz der Verschiebung bestimmt. Anschließend sind aus W_S alle Tupel s zu entfernen, die aus den neuen Grenzen für W_R und W_S herausfallen, für die also die Bedingung $s.y_1 < minRS'$ erfüllt ist. Der Teil zwischen der neuen Untergrenze $minRS'$ und der alten Obergrenze $maxRS$, der nach dem Entfernen nicht mehr benötigter Tupel von den beiden Datenfenstern W_R und W_S noch übrig bleibt, wird mit W'_R bzw. W'_S bezeichnet. Nun sind die beiden Datenfenster noch mit neuen Tupeln zu füllen. Dazu werden wie schon beim erstmaligen Einlesen alle Tupel s bzw. r aus den Datenströmen R und S in die Fenster W_R und W_S gelesen, für die $r.y_1 \leq maxRS'$ bzw.

$s.y_1 \leq maxRS'$ gilt. Begonnen wird dabei mit dem gepufferten Tupel, das bereits beim vorangegangenen Einlesevorgang gelesen wurde, aber nicht mehr in das alte Fenster passte. Für alle weiteren Tupel liest das Verfahren die Eingaben wieder tupelweise aus den beiden Datenströmen und nimmt sie in das jeweilige Datenfenster auf, bis es das erste Tupel antrifft, dessen y_1-Wert größer ist als $maxRS'$. Dieses Tupel wird wieder bis zum nächsten Aktualisierungsschritt zwischengespeichert. Die Teile der Datenfenster zwischen der alten Obergrenze $maxRS$ und der neuen Obergrenze $maxRS'$ heißen ΔW_R und ΔW_S. Auch der Zwischenspeicher O für potenzielle Ergebnispaare unterliegt der Aktualisierung. Alle Tupelpaare $(r \times s)$, für deren linkes Tupel r die Bedingung $r.y_1 < minRS'$ gilt, können aus dem Puffer entfernt und als Teil des Endergebnisses ausgegeben werden. Aufgrund der Einschränkung des Suchraums auf das Intervall $[r.y_1 - \epsilon_1; r.y_1 + \epsilon_1]$ in der Dimension y_1 für jedes $r \in R$ und der aufsteigenden Sortierung der y_1-Werte der Tupel in den Datenströmen können diese Paare im weiteren Verlauf der Berechnung nicht mehr dominiert werden. Im nächsten Schritt muss der Algorithmus nun den Join der in den aktualisierten Datenfenstern enthaltenen Tupel berechnen. Dabei ist zu beachten, dass nicht einfach der vollständige Inhalt beider Fenster miteinander kombiniert werden darf, da sonst Duplikate erzeugt werden können. Die Situation ist in Abbildung 4.6 gezeigt. Die Pfeile deuten an, welche Teile der beiden Datenfenster zu kombinieren sind. Die bei der letzten Aktualisierung der Fenster neu in W_R hinzugekommenen Tupel müssen mit dem gesamten Inhalt, der Rest nur mit dem neuen Teil von W_S verbunden werden. Dies vermeidet, dass die bereits vor der vorhergehenden Verschiebung der Fenstergrenzen in den Datenfenstern vorhandenen und durch das Verschieben nicht verworfenen Tupel noch einmal miteinander kombiniert werden. Das Verfahren berechnet also zwei Joins auf jeweils unterschiedlichen Teilmengen der Fensterinhalte.

Die nachfolgenden Ausführungen erläutern die einzelnen Vorgänge anhand der Algorithmen 4.3 und 4.4 im Detail. Algorithmus 4.3 stellt den fensterbasierten Algorithmus zur Berechnung des ϵ-LOBMJ dar. Er erhält als Eingabe die beiden Datenströme R und S, die ϵ-Umgebung und die partielle Ordnung $\prec_{y_1,\dots,y_d}$. Als Ausgabe liefert das Verfahren einen Datenstrom mit den Ergebnistupeln des Joins $R \bowtie^{\epsilon_1,\dots,\epsilon_d}_{y_1,\dots,y_d} S$. In den ersten Zeilen erfolgen zunächst einige Initialisierungen. So werden der Zustand O und die Datenfenster W_R und W_S eingeführt und in der vierten Zeile wird aus beiden Datenströmen mittels der Funktion *next* das jeweils erste Tupel in eine Puffervariable eingelesen. Außerdem wird überprüft, ob eine der beiden Eingaben leer ist. Dies ist dann der Fall, wenn eine der beiden Puffervariablen nach dem Einlesen des ersten Tupels bereits den Wert *null* besitzt, der das Ende des entsprechenden Datenstroms markiert. Unter diesen Umständen terminiert der Algorithmus, da keine Ergebnistupel erzeugt werden können. In der achten Zeile erfolgt das initiale Setzen der Fenstergrenzen. Nur wenn das erste Tupel aus R einen y_1-Wert kleiner als die Höhe ϵ_1 der Datenfenster besitzt, haben die Grenzen initial die Werte $minRS = 0$ und $maxRS = \epsilon_1$. Anderenfalls muss die Obergrenze auf $r.y_1$ und die Untergrenze entsprechend auf $r.y_1 - \epsilon_1$ gesetzt werden. Damit liegt das erste Tupel der linken Eingabe in diesem Fall immer genau auf der Obergrenze des Fensters, so dass das Fenster für die rechte Eingabe nach dem erstmaligen Einlesen alle Tupel $s \in S$ enthält, für die $r.y_1 - \epsilon_1 \leq s.y_1 \leq r.y_1$ gilt, die also für das erste Tupel aus R relevante Join-Partner darstellen. Die übrigen relevanten Tupel $s \in S$ mit $r.y_1 < s.y_1 \leq r.y_1 + \epsilon_1$ folgen dann in späteren Berechnungsschritten. Weiterhin wird $oldMaxRS$ auf 0 gesetzt. Diese

Algorithmus 4.3 Fensterbasierter Algorithmus

Eingabe: Datenströme R und S, ϵ-Umgebung $\epsilon_1, \ldots, \epsilon_d$, partielle Ordnung $\prec_{y_1,\ldots,y_d}$

Ausgabe: Datenstrom von $R \bowtie^{\epsilon_1,\ldots,\epsilon_d}_{y_1,\ldots,y_d} S$

/* Initialisierung */
$O \leftarrow \emptyset$
$W_R \leftarrow \emptyset;\ W_S \leftarrow \emptyset$
$r \leftarrow R.next();\ s \leftarrow S.next()$
if $(r = null) \vee (s = null)$ **then**
  return
end if
$minRS \leftarrow max(0, r.y_1 - \epsilon_1);\ maxRS \leftarrow minRS + \epsilon_1;\ oldMaxRS \leftarrow 0$
/* Join-Berechnung */
while $(r \neq null) \vee (W_R \neq \emptyset)$ **do**
  $UpdateWindows$
  if $oldMaxRS = 0$ **then** /* komplett neuer Fensterinhalt */
    $Schedule(O, W_R, W_S, \prec_{y_1,\ldots,y_d})$
  else /* teilweise neuer Fensterinhalt */
    $Schedule(O, W'_R, \Delta W_S, \prec_{y_1,\ldots,y_d})$
    $Schedule(O, \Delta W_R, W'_S \cup \Delta W_S, \prec_{y_1,\ldots,y_d})$
  end if
  /* Aktualisierung von Fenstergrenzen und -inhalt */
  $oldMaxRS \leftarrow maxRS$
  $W_R.delete(\{r' \in W_R \mid r'.y_1 = minRS\})$
  if $(r \neq null) \vee (W_R \neq \emptyset)$ **then**
    if $(r \neq null) \wedge (W_R \neq \emptyset)$ **then**
      $minRS \leftarrow min(min_{y_1}(W_R), r.y_1 - \epsilon_1)$
    else if $(r = null) \wedge (W_R \neq \emptyset)$ **then**
      $minRS \leftarrow min_{y_1}(W_R)$
    else /* $(r \neq null) \wedge (W_R = \emptyset)$ */
      $minRS \leftarrow r.y_1 - \epsilon_1;\ oldMaxRS \leftarrow 0$
    end if
    $maxRS \leftarrow minRS + \epsilon_1$
    $W_S.delete(\{s' \in W_S \mid s'.y_1 < minRS\})$
    output and delete $\{(r' \times s') \in O \mid r'.y_1 < minRS\}$
  end if
end while
output $\{(r' \times s') \in O\}$

Variable hält während der Ausführung des Algorithmus die alte Obergrenze der Datenfenster, die unmittelbar vor dem bis zum gegenwärtigen Zeitpunkt letzten Nachziehen der Fenster aktuell war. Sie stellt damit die Trennlinie zwischen dem alten Datenbestand der Datenfenster und den beim letzten Einlesevorgang neu hinzugekommenen Tupeln dar. In den Abbildungen 4.5 und 4.6 ist das die gestrichelte Linie zwischen W'_R und ΔW_R bzw. zwischen W'_S und ΔW_S.

Algorithmus 4.4 Aktualisieren des Inhalts der Datenfenster ($UpdateWindows$)

```
while ((r ≠ null) ∧ (r.y1 ≤ maxRS)) ∨ ((s ≠ null) ∧ (s.y1 ≤ maxRS)) do
  if (r ≠ null) ∧ (r.y1 ≤ maxRS) then /* W_R füllen */
    if |M_R| = m_R − 1 then
      materialize(M_R); M_R ← ∅
    end if
    M_R.add(r); r ← R.next()
  end if
  if (s ≠ null) ∧ (s.y1 ≤ maxRS) then /* W_S füllen */
    if s.y1 ≥ minRS then
      if |M_S| = m_S − 1 then
        materialize(M_S); M_S ← ∅
      end if
      M_S.add(s)
    end if
    s ← S.next()
  end if
end while
materialize(M_R); materialize(M_S)
```

Die folgende while-Schleife beinhaltet das Einlesen und Verschieben der Datenfenster sowie das Anstoßen der Join-Berechnungen. Sie wiederholt diese Schritte so lange, bis sowohl der Datenstrom R der linken Eingabe als auch das aktuelle linke Datenfenster W_R keine Tupel mehr enthalten. Dann sind alle Tupel $r \in R$ der linken Eingabe abgearbeitet und der eingeschränkte Left-Outer-Bestmatch-Join ist vollständig berechnet.

Zunächst werden innerhalb der Schleife die Fenster mit Tupeln gefüllt. Dieser Vorgang ist in Algorithmus 4.4 dargestellt. Solange mindestens einer der beiden Datenströme R und S noch Daten liefert und das aktuell gelesene Tupel $r \in R$ bzw. $s \in S$ noch innerhalb der aktuellen Grenzen der Datenfenster W_R und W_S liegt, wird das Tupel in das entsprechende Fenster aufgenommen und das nächste Datum aus dem Datenstrom gelesen. Die Tupel werden auf dem Hintergrundspeicher in Seiten gleicher Kapazität gespeichert, wie es der Realität im Datenbankbereich entspricht. Mit M_R bzw. M_S werde der Puffer für Seiten des linken bzw. rechten Datenfensters im Hauptspeicher bezeichnet. Weiter seien m_R und m_S die maximale Anzahl an Seiten, die gleichzeitig in M_R bzw. M_S Platz finden. Dann können neue Tupel eines Datenstroms so lange mittels der Funktion *add* in den Hauptspeicher geschrieben werden, bis der Hauptspeicherpuffer des zugehörigen Datenfensters bis auf eine Seite voll ist, bis also $|M_R| = m_R - 1$ oder $|M_S| = m_S - 1$ gilt. Bei Verwendung der Epsilon-Grid-Ordnung und des auf dieser Ordnung aufbauenden, noch einzuführenden Scheduling-Algorithmus muss der Hauptspeicherinhalt durch die *materialize* Funktion so sortiert und materialisiert werden, dass er zusammen mit bereits früher materialisierten Bestandteilen des Fensters gemäß Epsilon-Grid-Ordnung sortiert auf dem Hintergrundspeicher liegt. Zum Zwecke des Mischens der zu materialisierenden Tupel mit bereits auf den Hintergrundspeicher ausgelagerten Datenobjekten muss dazu mindestens für eine Seite Platz im Hauptspeicher freigehalten werden. Entsprechende Materialisierungsvarianten folgen in Abschnitt 4.3.3. Der fensterbasierte Algorithmus kann aber auch ohne diese Erweiterung verwendet werden. Dann ist die Sortierung der Datenelemente nach Epsilon-Grid-Ordnung im Rahmen des Materialisierungsvorgangs nicht erforderlich. Nachdem die Materialisierung des Hauptspeicherpuffers abgeschlossen ist, wird dieser freigegeben und kann wieder mit neuen Tupeln gefüllt werden. Beim Einlesen von Datenelementen aus S ist außerdem darauf zu achten, dass die Elemente nur dann in W_S aufgenommen werden dürfen, wenn sie auch oberhalb der Fensteruntergrenze liegen. Es muss also $s.y_1 \geq minRS$ gelten. Dies ist dann relevant, wenn aufeinander folgende Tupel aus R bezüglich ihrer y_1-Werte weiter als $2\epsilon_1$ auseinander liegen. Dann nämlich werden die Fenstergrenzen, die sich ja allein an den y_1-Werten der Datenobjekte aus R orientieren, in einem Schritt um mehr als die komplette Fensterhöhe ϵ_1 verschoben. Das bedeutet, dass die neue Fensteruntergrenze nach dem Verschieben größer ist, als die alte Fensterobergrenze vor dem Verschieben war. Damit werden also komplett neue Fenster gebildet, die ausschließlich aus neu eingelesenen Tupeln bestehen und keine alten Datenelemente mehr beinhalten, die bereits in früheren Schritten eingelesen wurden. Dies wird im Algorithmus durch das Setzen von $oldMaxRS = 0$ markiert. Tupel aus S, deren y_1-Wert größer als die alte Fensterobergrenze und gleichzeitig kleiner als die neue Fensteruntergrenze ist, können dann aufgrund der Einschränkung auf ϵ_1 keinen Join-Partner in R finden und dürfen nie in W_S aufgenommen werden. Genau das verhindert die if-Abfrage in Zeile 9 von Algorithmus 4.4. Abbildung 4.7 zeigt dies an einem Beispiel. Im Datenstrom R liegen der dritte und vierte Datenpunkt in der Dimension y_1 um mehr als $2\epsilon_1$ auseinander. Das linke Bild zeigt den Zustand unmittelbar vor der nächsten Verschiebung der Datenfenster. Im rechten Bild ist die Situation nach dem Verschieben der beiden Fenster gezeigt. Ihre vorhergehenden Positionen sind gestrichelt eingezeichnet. Das fünfte Tupel des Datenstroms S ist weder in dem alten gestrichelten, noch in dem neuen Fenster enthalten. Es findet keinen Join-Partner in R, dessen y_1-Wert um höchstens ϵ_1 von seinem eigenen abweicht. Daher ist dieses Datenelement aus S für die Berechnung des Join-Ergebnisses irrelevant und taucht in keinem Datenfenster auf. Nachdem der Algorithmus beide Fenster mit allen relevanten Tupeln für die aktuellen Fenstergrenzen aufgefüllt hat, sind zum Schluss in Zeile 18 noch die in den Hauptspeicherpuffern verbliebenen Tupel zu materialisieren, damit der gesamte Datenbestand der Fenster, gegebenenfalls korrekt nach Epsilon-Grid-Ordnung sortiert, auf dem Hintergrundspeicher vorliegt.

In Algorithmus 4.3 geht es danach mit der Berechnung des Joins auf den aktuellen Datenfenstern weiter. Falls $oldMaxRS = 0$ gilt, liegen vollständig neu eingelesene Fenster vor, deren Inhalt wie in Abbildung 4.4 komplett miteinander kombiniert werden kann. Ansonsten ergibt sich die Situation aus Bild 4.6. Die Join-Berechnung lässt sich durch Anwendung des Nested-Loops Algorithmus auf den Fenstern erledigen, wobei die benötigten Seiten mit den Datenelementen der Fenster einfach bei Bedarf vom Hintergrundspeicher geladen werden, erfolgt aber idealerweise durch einen Aufruf des in Kapitel 4.3 beschriebenen Scheduling-Algorithmus, der die Sortierung der materialisierten Datenelemente nach Epsilon-Grid-Ordnung voraussetzt. Er erhält neben dem Zustand O und der partiellen Ordnung $\prec_{y_1,\ldots,y_d}$ auch die jeweils relevanten Teile der Datenfenster W_R und W_S übergeben. Da im Falle vollständig neu eingelesener Fenster jeweils der komplette Fensterinhalt von W_R mit dem von W_S verbunden werden muss, sind hier die gesamten Fenster W_R

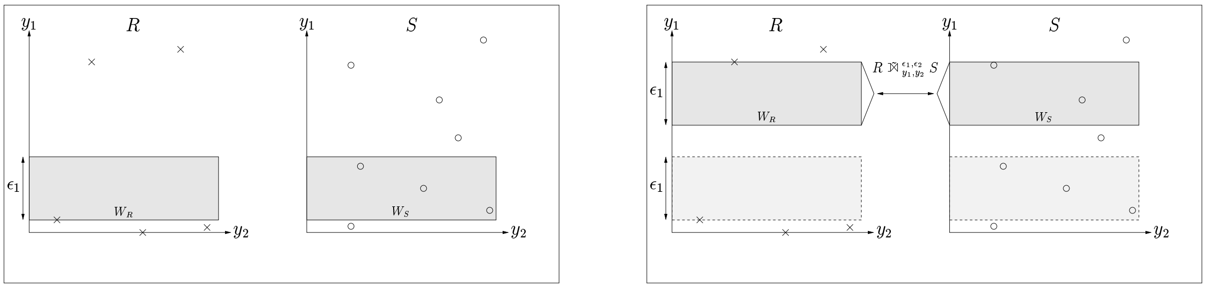


Abb. 4.7: Springen der Datenfenster bei einer Verschiebung um mehr als ϵ_1

und W_S angegeben. Dieser Fall tritt außer im ersten Schritt nach der Initialisierung auch jedesmal nach einer Verschiebung der Fenster um mehr als die Fensterhöhe ϵ_1, wie oben beschrieben, ein. Ansonsten sind entsprechend Abbildung 4.6 bestimmte Teile der Fenster zu kombinieren. Dafür sind jeweils zwei Aufrufe der Funktion *Schedule* notwendig, wobei der erste den alten Teil des linken Fensters mit dem neuen des rechten und der zweite den neuen Teil des linken Fensters mit dem gesamten rechten Fenster verbindet. Zu beachten ist hierbei, dass Tupel, die auf der Trennlinie $oldMaxRS$ zwischen altem und neuem Teil eines Datenfensters liegen, noch zum alten Teil gehören. Schließlich liegen sie auf der alten Obergrenze und wurden somit nicht beim bis dahin letzten Einlesevorgang neu eingelesen, sondern waren zu diesem Zeitpunkt bereits im Fenster vorhanden. Sie sind somit Elemente der Menge W'_R bzw. W'_S und nicht in ΔW_R oder ΔW_S enthalten.

Die Zeilen 18 bis 32 von Algorithmus 4.3 regeln das Nachziehen der Datenfenster. Zunächst wird die aktuelle Fensterobergrenze $maxRS$ in $oldMaxRS$ gesichert. Anschließend entfernt das Verfahren alle Tupel r' aus W_R, die auf der aktuellen Untergrenze von W_R liegen, für die also $r'.y_1 = minRS$ gilt. Die folgenden if-Abfragen dienen zur Unterscheidung von vier Fällen, die beim Berechnen und Setzen der neuen Fenstergrenzen auftreten können:

- Falls sowohl der Datenstrom R als auch das zugehörige Datenfenster W_R keine Tupel mehr enthalten, sind alle Datenelemente $r \in R$ bereits abgearbeitet und der ϵ-LOBMJ ist unter den gegebenen Bedingungen vollständig und korrekt berechnet worden. Damit ist die if-Bedingung aus Zeile 21 des Algorithmus nicht mehr erfüllt und die anderen drei Fälle werden übersprungen. Die while-Schleife terminiert anschließend und das Verfahren gibt zuletzt in Zeile 34 die in O verbliebenen Zwischenergebnisse zur Vervollständigung des Endergebnisses aus.

- Der gegenteilige Fall tritt ein, wenn sowohl in R als auch in W_R noch Tupel enthalten sind. Dies ist der in den Abbildungen 4.3 bis 4.6 dargestellte Fall. Hier wird $minRS$ auf das Minimum aus $min_{y_1}(W_R)$ und $r.y_1 - \epsilon_1$ gesetzt. Damit ist sichergestellt, dass jedes Tupel $r \in R$ mit jedem für r relevanten Tupel $s \in S$ mit der Eigenschaft $r.y_1 - \epsilon_1 \leq s.y_1 \leq r.y_1 + \epsilon_1$ zu irgendeinem Zeitpunkt genau einmal in miteinander kombinierten Teilen der beiden Datenfenster W_R und W_S enthalten ist. Somit werden weder Duplikate generiert, noch relevante Paarungen von Tupeln $r \in R$ und $s \in S$ übersehen.

- Sollte der Datenstrom R keine Tupel mehr enthalten, das Datenfenster W_R hingegen schon, so wird $minRS$ auf $min_{y_1}(W_R)$ gesetzt. Diese Situation tritt gegen Ende der

Berechnung auf, wenn der Datenstrom erschöpft ist, die noch im Fenster befindlichen Datenobjekte aber noch fertig bearbeitet werden müssen.

- Der vierte und letzte Fall tritt ein, wenn W_R leer ist und R noch neue Tupel liefert. Dies entspricht der oben beschriebenen und in Abbildung 4.7 dargestellten Situation, dass die Datenfenster um mehr als die Fensterhöhe ϵ_1 verschoben wurden. Hier setzt der Algorithmus $minRS$ auf $r.y_1 - \epsilon_1$ und $oldMaxRS$ auf 0, um den obigen Ausführungen entsprechend zu markieren, dass die aktuellen Fenster ausschließlich neu eingelesene Datenelemente enthalten.

Weiterhin wird in allen vier Fällen $maxRS$ nach der Aktualisierung von $minRS$ auf $minRS + \epsilon_1$ gesetzt, so dass anschließend Unter- und Obergrenzen aktuell sind und die Fensterhöhe wieder ϵ_1 beträgt. Außerdem muss in allen Fällen außer dem ersten nun noch der Inhalt von W_S und O auf den neuesten Stand gebracht werden. Dazu entfernt der Algorithmus in Zeile 30 alle Tupel aus W_S, die durch die Neuberechnung der Fenstergrenzen jetzt aus dem Datenfenster herausfallen. Das sind alle $s' \in W_S$, für die $s'.y_1 < minRS$ gilt. In ähnlicher Weise ist der Puffer O für die Zwischenergebnisse zu behandeln. Alle Paare $(r' \times s')$, für deren linkes Tupel r' die Bedingung $r'.y_1 < minRS$ erfüllt ist, können aus dem Puffer entfernt und als Teil des endgültigen Ergebnisses ausgegeben werden. Wegen der Einschränkung auf ϵ_1 und der aufsteigenden Sortierung der Tupel aus den Datenströmen in der Dimension y_1 kann kein Tupel $s'' \in S$ mehr auftauchen, so dass $(r' \times s'') \prec_{y_1,\ldots,y_d} (r' \times s')$ gilt.

Aus obigen Erläuterungen geht hervor, dass mit Ausnahme der Tupel, die zu Beginn des Algorithmus beim erstmaligen Einlesen der Datenfenster in die Fenster aufgenommen werden, jedes Tupel $r \in R$ zu irgendeinem Zeitpunkt genau auf der Obergrenze des aktuellen Datenfensters W_R liegt, nämlich in dem Schritt, in dem es in das Fenster eingelesen wurde. Deshalb können abgesehen vom ersten Einlesen bei einem Einlesevorgang nur solche Tupel $r \in R$ in W_R eingefügt werden, die alle den gleichen y_1-Wert besitzen. Damit ist der neu eingelesene Teil von W_R bis auf die Tupel, die auf der Obergrenze liegen, immer leer. Für W_S muss das natürlich nicht gelten. Dies garantiert, dass unmittelbar nach dem Einfügen eines Tupels $r \in R$ in das Fenster W_R alle Tupel $s \in S$ in W_S enthalten sind, für die $r.y_1 - \epsilon_1 \leq s.y_1 \leq r.y_1$ gilt. Entsprechend liegt jedes Tupel $r \in R$ auch zu irgendeinem Zeitpunkt genau auf der Untergrenze des aktuellen Datenfensters W_R, nämlich im letzten Berechnungsschritt, bevor es aus dem Fenster entfernt wird. In diesem Moment enthält das Datenfenster W_S alle Tupel $s \in S$ mit der Eigenschaft $r.y_1 \leq s.y_1 \leq r.y_1 + \epsilon_1$. Insgesamt wird also jedes Tupel $r \in R$ irgendwann während des Berechnungsvorgangs mit jedem für dieses r relevanten Tupel $s \in S$ kombiniert.

In diesem Abschnitt wurde der fensterbasierte Algorithmus als Verfahren zur Berechnung des ϵ-LOBMJ vorgestellt. Es bleibt hinzuzufügen, dass der fensterbasierte Ansatz auch zur Ermittlung des Ergebnisses des ϵ-BMJ verwendet werden kann. Dazu muss lediglich die Join-Berechnung auf den jeweils miteinander zu verbindenden Teilen der Datenfenster vom ϵ-LOBMJ auf den ϵ-BMJ umgestellt werden. Dann besteht der kontinuierliche Strom von Ausgabedaten des Verfahrens allerdings nur aus Bestandteilen eines approximativen Ergebnisses, da bei keinem Ergebnispaar ausgeschlossen werden kann, dass später noch ein besseres Tupelpaar generiert wird, das ein oder mehrere ältere Paare invalidiert. Bei unendlichen Datenströmen stellt diese approximative Lösung das bestmöglich Erreichbare dar. Bei endlichen Datenströmen ergibt sich am Ende der Berechnung schließlich das korrekte Endergebnis des Joins.

4.2.3 Diskussion

Abschließend lässt sich sagen, dass der fensterbasierte Algoritmus gegenüber der naiven Methode eine ganze Reihe von Vorteilen besitzt. So ist er, wie die Leistungsmessungen in Kapitel 6 noch zeigen werden, nicht nur wesentlich schneller und effizienter bei der Berechnung, sondern bietet darüber hinaus die Möglichkeit, aus kontinuierlichen Eingabedatenströmen einen kontinuierlichen Ausgabedatenstrom zu erzeugen. Dies scheitert beim naiven Algorithmus daran, dass er zunächst die komplette rechte Eingabe vollständig materialisieren muss, bevor er mit der Berechnung des Left-Outer-Bestmatch-Joins überhaupt beginnen kann. Den Vorteilen stehen nachteilig die Voraussetzungen, bestehend aus der Einschränkung auf eine ϵ-Umgebung und vorsortierten Eingabedatenströmen, und der deutlich höhere Implementierungsaufwand gegenüber. Dies wiegt allerdings nicht besonders schwer, da die Einschränkung des Suchraums in der Praxis oftmals Sinn macht, z. B. um Ausreißer in einzelnen Dimensionen zu eliminieren, moderne Datenquellen in der Regel in der Lage sind, ihre Daten auf Anforderung gemäß einer bestimmten Ordnung sortiert zu liefern und der Aufwand angesichts der erweiterten Möglichkeiten und gesteigerten Effizienz im Vergleich zum naiven Algorithmus absolut gerechtfertigt ist. Außerdem lässt sich beispielsweise die Voraussetzung vorsortierter Datenströme durch die Anwendung spezieller Techniken lockern, wie spätere Ausführungen noch verdeutlichen werden.

4.3 Scheduling-Algorithmus

Nachdem der vorhergehende Abschnitt den fensterbasierten Algorithmus im Detail vorgestellt hat, soll nun auf Möglichkeiten zur Effizienzsteigerung des Verfahrens eingegangen werden. Dabei ist insbesondere die Materialisierung der Datenfenster auf dem Hintergrundspeicher und das Laden der Speicherseiten vom Hintergrundspeicher in den Hauptspeicher zum Zwecke der weiteren Verarbeitung zu betrachten, verursachen doch diese Schritte den Hauptteil der Gesamtlaufzeit einer Implementierung des Algorithmus. Ein möglicher Ansatz ist, die Tupel eines Datenfensters nach einer abgewandelten Version der in [BBKK01] definierten Epsilon-Grid-Ordnung sortiert zu materialisieren. Dann können die Seiten auf dem Hintergrundspeicher, welche die gespeicherten Tupel enthalten, bei Bedarf mittels einer angepassten Version des ebenfalls in [BBKK01] vorgestellten Scheduling-Algorithmus geladen werden. Das Ziel dieser Strategie ist, die Anzahl an Lesezugriffen auf Seiten des Hintergrundspeichers möglichst gering zu halten und damit das gesamte Berechnungsverfahren zu beschleunigen. Nachfolgend wird die veränderte Version der Epsilon-Grid-Ordnung, nach der die Tupel im Zuge der Materialisierung sortiert werden, definiert. Anschließend folgen der Scheduling-Algorithmus und zwei mögliche Materialisierungsvarianten, die ihre ganz spezifischen Vor- und Nachteile besitzen. Zum Schluss werden die vorgestellten Verfahren noch einmal kritisch betrachtet.

4.3.1 Epsilon-Grid-Ordnung

Die ursprüngliche Epsilon-Grid-Ordnung war für die effiziente Berechnung von Similarity-Self-Joins gedacht. Dabei handelt es sich um Joins, die aus einer mehrdimensionalen Menge von Punkten diejenigen Paare herausfinden sollen, deren Abstand bezüglich der euklidischen Metrik kleiner als ein vorgegebener Maximalabstand ϵ ist. Diese Zielsetzung unterscheidet sich in zwei wesentlichen Punkten von der in dieser Arbeit verfolgten. Zum einen arbeitet ein Self-Join, wie der Name schon sagt, auf nur einer Menge von Eingabedaten, während die Verfahren zur Berechnung von besten Zuordnungen von Objekten zweier Datenströme naturgemäß auf zwei in der Regel unterschiedlichen Eingaben operieren. Zum anderen entspricht die Verwendung der euklidischen Metrik als Abstandsbegriff dem Einsatz einer Bewertungsfunktion, die aus allen Dimensionen einen einzelnen Abstandswert berechnet. Im Gegensatz dazu betrachtet der Bestmatch-Join im Stil der Skyline-Berechnung jede Join-Dimension individuell. Daher genügt es nun nicht mehr, nur einen Maximalabstand ϵ vorzugeben. Vielmehr muss eine ϵ-Umgebung existieren, die für jede Dimension $y_i \in \{y_1, \ldots, y_d\}$ einen eigenen Maximalabstand $\epsilon_i \in \{\epsilon_1, \ldots, \epsilon_d\}$ besitzt. Diese beiden Modifikationen werden an der Definition der Epsilon-Grid-Ordnung und dem Scheduling-Algorithmus aus [BBKK01] in diesem und dem nächsten Abschnitt vorgenommen.

Für die Definition der Epsilon-Grid-Ordnung betrachte man die beiden d-dimensionalen Vektoren $\vec{p}$ und $\vec{q}$. Sie enthalten die durch die ϵ-Umgebung eingeschränkten Attribute der zugehörigen Tupel $r \in R$ bzw. $s \in S$. Die i-te Dimension von $\vec{p}$ bzw. $\vec{q}$ werde mit p_i bzw. q_i bezeichnet. Ferner seien $\vec{\epsilon}$ und ϵ_i der Vektor bzw. die i-te Komponente des Vektors der ϵ-Umgebung.

Definition 4.1 (Epsilon-Grid-Ordnung $<_\epsilon$) *Die Epsilon-Grid-Ordnung zum Vergleich zweier d-dimensionaler Vektoren $\vec{p}$ und $\vec{q}$ wird mit $<_\epsilon$ bezeichnet und ist wie folgt definiert:*

$$\begin{aligned}\vec{p} <_\epsilon \vec{q} \;\Leftrightarrow\; & \left(\exists i \in \{1, \ldots, d\} : \left\lfloor \frac{p_i}{\epsilon_i} \right\rfloor < \left\lfloor \frac{q_i}{\epsilon_i} \right\rfloor\right) \wedge \\ & \left(\forall j \in \{1, \ldots, d\} \text{ mit } j < i : \left\lfloor \frac{p_j}{\epsilon_j} \right\rfloor = \left\lfloor \frac{q_j}{\epsilon_j} \right\rfloor\right)\end{aligned}$$

Es handelt sich hierbei also um eine lexikographische Ordnung. Der Vektor $\vec{p}$ ist nach EGO genau dann kleiner als der Vektor $\vec{q}$, wenn es eine Dimension i gibt, in der die auf die nächste natürliche Zahl abgerundete Division der Komponente p_i durch ϵ_i kleiner ist als der Wert der entsprechenden Berechnung für die Komponente q_i, und wenn dieselbe Berechnung auf den Komponenten aller vorhergehenden Dimensionen $j < i$ in jeder Dimension für p_j und q_j jeweils das gleiche Ergebnis erbracht hat. Wird die Epsilon-Grid-Ordnung im Folgenden auf zwei Tupel $r \in R$ und $s \in S$ angewandt, so ist damit immer der Vergleich der die eingeschränkten Attribute der Tupel enthaltenden Vektoren $\vec{p}$ und $\vec{q}$ gemeint, es gilt also:

$$r <_\epsilon s \;\Leftrightarrow\; \vec{p} <_\epsilon \vec{q}$$

In [BBKK01] wird bewiesen, dass die dort definierte Epsilon-Grid-Ordnung eine strikte Ordnung ist. Dazu ist zu zeigen, dass sie irreflexiv, asymmetrisch und transitiv ist. Dieser Beweis ist relativ einfach und lässt sich analog für die hier eingeführte modifizierte Ordnung durchführen.

Lemma 4.1 *Die Epsilon-Grid-Ordnung ist eine strikte Ordnung.*

Beweis:
Seien $\vec{p}$ und $\vec{q}$ zwei d-dimensionale Vektoren.

Irreflexivität ($\neg\vec{p} <_\epsilon \vec{p}$):
Es kann nie $\vec{p} <_\epsilon \vec{p}$ gelten, da es keine Dimension mit Index i gibt, für die $\lfloor p_i/\epsilon_i \rfloor < \lfloor p_i/\epsilon_i \rfloor$ gilt.

Asymmetrie ($\vec{p} <_\epsilon \vec{q} \;\Rightarrow\; \neg\vec{q} <_\epsilon \vec{p}$):
Wegen $\vec{p} <_\epsilon \vec{q}$ gibt es eine Dimension mit Index i, für die $\lfloor p_i/\epsilon_i \rfloor < \lfloor q_i/\epsilon_i \rfloor$ und $\lfloor p_j/\epsilon_j \rfloor = \lfloor q_j/\epsilon_j \rfloor$ für alle $j < i$ gilt. Somit gilt auch $\lfloor q_j/\epsilon_j \rfloor = \lfloor p_j/\epsilon_j \rfloor$. Es kann aber weder $\lfloor q_i/\epsilon_i \rfloor < \lfloor p_i/\epsilon_i \rfloor$ noch $\lfloor q_i/\epsilon_i \rfloor = \lfloor p_i/\epsilon_i \rfloor$ erfüllt sein. Damit ist auch $\vec{q} <_\epsilon \vec{p}$ nicht erfüllt.

Transitivität ($\vec{p} <_\epsilon \vec{q} \wedge \vec{q} <_\epsilon \vec{n} \;\Rightarrow\; \vec{p} <_\epsilon \vec{n}$):
Wegen $\vec{p} <_\epsilon \vec{q}$ gibt es wiederum eine Dimension mit Index i, für die $\lfloor p_i/\epsilon_i \rfloor < \lfloor q_i/\epsilon_i \rfloor$ und $\lfloor p_j/\epsilon_j \rfloor = \lfloor q_j/\epsilon_j \rfloor$ für alle $j < i$ gilt. Wegen $\vec{q} <_\epsilon \vec{n}$ gibt es entsprechend eine Dimension mit Index i', so dass $\lfloor q_{i'}/\epsilon_{i'} \rfloor < \lfloor n_{i'}/\epsilon_{i'} \rfloor$ und $\lfloor q_j/\epsilon_j \rfloor = \lfloor n_j/\epsilon_j \rfloor$ für alle $j < i'$ gilt. O. B. d. A. sei $i < i'$ (für die anderen Fälle verläuft der Beweis analog). Damit gilt $\lfloor p_j/\epsilon_j \rfloor = \lfloor q_j/\epsilon_j \rfloor = \lfloor n_j/\epsilon_j \rfloor$ für alle $j < i$ und weiter $\lfloor p_i/\epsilon_i \rfloor < \lfloor q_i/\epsilon_i \rfloor = \lfloor n_i/\epsilon_i \rfloor$. Insgesamt folgt somit aus der Definition der Epsilon-Grid-Ordnung $\vec{p} <_\epsilon \vec{n}$. □

Eine weitere wichtige Eigenschaft der Epsilon-Grid-Ordnung ist, dass kein Tupel $s \in S$ ein relevanter Join-Partner eines Tupels $r \in R$ sein kann, wenn der Vektor $\vec{q}$ der Join-Attribute von s außerhalb der ϵ-Umgebung des Vektors $\vec{p}$ der Join-Attribute von r liegt. Damit müssen z. B. beim Left-Outer-Bestmatch-Join mit Einschränkungen für jedes $r \in R$ nur die $s \in S$ als mögliche Join-Partner in Betracht gezogen werden, die sich innerhalb der ϵ-Umgebung um das jeweils betrachtete r befinden. Die folgenden beiden Lemmata nach [BBKK01] drücken dies formal aus.

Lemma 4.2 *Seien $\vec{p}$, $\vec{p'}$ und $\vec{q}$ jeweils d-dimensionale Vektoren. Wenn $\vec{q} <_\epsilon \vec{p} - (\epsilon_1, \ldots, \epsilon_d)$ gilt, dann ist $\vec{q}$ kein relevanter Join-Partner für $\vec{p}$, d. h. es gilt:*

$$\vec{q} \;\notin\; [p_1 - \epsilon_1] \times \cdots \times [p_d - \epsilon_d]$$

Weiter ist $\vec{q}$ dann auch kein relevanter Join-Partner für alle $\vec{p'}$, für die nicht $\vec{p'} <_\epsilon \vec{p}$ gilt, d. h. es gilt auch:

$$\vec{q} \;\notin\; [p'_1 - \epsilon_1] \times \cdots \times [p'_d - \epsilon_d]$$

Beweis:
Aus $\vec{q} <_\epsilon \vec{p} - (\epsilon_1, \ldots, \epsilon_d)$ folgt mit Definition 4.1 sofort:

$$\exists i \in \{1, \ldots, d\} : \left\lfloor \frac{q_i}{\epsilon_i} \right\rfloor < \left\lfloor \frac{p_i - \epsilon_i}{\epsilon_i} \right\rfloor$$

Aufgrund der Monotonie der Rundungsfunktion[1] gilt damit auch $q_i < p_i - \epsilon_i$. Es existiert also eine Dimension mit Index i, so dass $p_i - q_i > \epsilon_i$ gilt. Damit sind alle Vektoren $\vec{q}$ mit $\vec{q} <_\epsilon \vec{p} - (\epsilon_1, \ldots, \epsilon_d)$ nicht in dem Bereich $[p_1 - \epsilon_1] \times \cdots \times [p_d - \epsilon_d]$ enthalten.

Wegen der Transitivität von $<_\epsilon$ existiert auch eine Dimension mit Index i', so dass $q_{i'} < p'_{i'} - \epsilon_{i'}$ gilt. Analog zu oben folgt dann, dass $\vec{q}$ nicht in $[p'_1 - \epsilon_1] \times \cdots \times [p'_d - \epsilon_d]$ enthalten ist. □

Lemma 4.3 *Seien $\vec{p}$, $\vec{p'}$ und $\vec{q}$ jeweils d-dimensionale Vektoren. Wenn $\vec{p} + (\epsilon_1, \ldots, \epsilon_d) <_\epsilon \vec{q}$ gilt, dann ist $\vec{q}$ kein relevanter Join-Partner für $\vec{p}$, d. h. es gilt:*

$$\vec{q} \notin [p_1 + \epsilon_1] \times \cdots \times [p_d + \epsilon_d]$$

Weiter ist $\vec{q}$ dann auch kein relevanter Join-Partner für alle $\vec{p'}$, für die nicht $\vec{p} <_\epsilon \vec{p'}$ gilt, d. h. es gilt auch:

$$\vec{q} \notin [p'_1 + \epsilon_1] \times \cdots \times [p'_d + \epsilon_d]$$

Beweis:
Der Beweis verläuft analog zum Beweis von Lemma 4.2. □

Damit lässt sich die Epsilon-Grid-Ordnung zur effizienten Berechnung des Left-Outer-Bestmatch-Joins mit Einschränkungen $R \mathbin{⟕}^{\epsilon_1,\ldots,\epsilon_d}_{y_1,\ldots,y_d} S$ einsetzen. Dazu werden die Datenobjekte der Datenströme gemäß EGO sortiert und auf dem Hintergrundspeicher materialisiert. Für ein Tupel $r \in R$ gelte in Bezug auf die Join-Attribute $y_1, \ldots, y_d$:

$$r - \vec{\epsilon} := (r.y_1 - \epsilon_1, \ldots, r.y_d - \epsilon_d)$$

$$r + \vec{\epsilon} := (r.y_1 + \epsilon_1, \ldots, r.y_d + \epsilon_d)$$

Analoges gilt für Tupel $s \in S$. Wegen der oben gezeigten Eigenschaft liegen nur solche Tupel $s \in S$ innerhalb der interessierenden ϵ-Umgebung von r, für die gilt:

$$(s \not<_\epsilon (r - \vec{\epsilon})) \wedge ((r + \vec{\epsilon}) \not<_\epsilon s)$$

Für die Berechnung des Left-Outer-Bestmatch-Joins müssen also nur die Tupel s aus dem Datenstrom S betrachtet werden, die bezüglich der Epsilon-Grid-Ordnung im Bereich zwischen $r - \vec{\epsilon}$ und $r + \vec{\epsilon}$ liegen.

4.3.2 Algorithmus

Die Epsilon-Grid-Ordnung und ihre im vorhergehenden Abschnitt gezeigten Eigenschaften lassen sich ausnutzen, um die Anzahl an Seitenzugriffen des fensterbasierten Algorithmus auf materialisierte Daten auf dem Hintergrundspeicher zu verringern und damit die Ausführung des Algorithmus effizienter zu gestalten. Dazu speichern die *materialize* Funktionen in Algorithmus 4.4 den Inhalt der Datenfenster nach Epsilon-Grid-Ordnung

[1] $\lfloor a \rfloor < \lfloor b \rfloor \Rightarrow a < b$

sortiert in Seiten gleicher Kapazität auf dem Hintergrundspeicher. Die aktuell dort befindliche i-te Seite mit Tupeln des Datenstroms R werde mit P_R^i bezeichnet, die j-te Seite des Datenstroms S entsprechend mit P_S^j. Innerhalb einer Seite sind alle Tupel nach EGO aufsteigend sortiert, d. h. steht ein Tupel r_i in einer Seite vor einem Tupel r_j, dann kann nicht $r_j <_\epsilon r_i$ gelten. Die Sortierung erstreckt sich dabei seitenübergreifend auf den gesamten materialisierten Datenbestand. So kann für das erste Tupel $P_R^{i+1}.first$ der Seite P_R^{i+1} und das letzte Tupel $P_R^i.last$ der Seite P_R^i nicht gelten $P_R^{i+1}.first <_\epsilon P_R^i.last$. Entsprechendes gilt damit aufgrund der Transitivität von $<_\epsilon$ auch für alle anderen Datenelemente der beiden Seiten. Selbstverständlich ist dieses Beispiel auch auf Seiten, die nicht direkt aufeinanderfolgen, und entsprechend auf Seiten P_S^j mit Datenobjekten von S übertragbar.

Abbildung 4.8 auf Seite 57 zeigt an einem einfachen Beispiel, wie sich die in den einzelnen Dimensionen lexikographisch gemäß Epsilon-Grid-Ordnung sortierten Daten auf verschiedene Seiten auf dem Hintergrundspeicher verteilen können. Dargestellt ist die Unterteilung des Datenraums in das von der Ordnung erzeugte Gitter sowie die Zugehörigkeit einzelner Gitterteile zu Hintergrundspeicherseiten. Die einzelnen Gitterzellen orientieren sich an den Join-Dimensionen y_i und haben in jeder Dimension jeweils die Kantenlänge ϵ_i. Alle in derselben Zelle enthaltenen Datenelemente sind dann bezüglich der Epsilon-Grid-Ordnung gleich. Die zu einer Seite gehörenden Teile des Gitters sind in der Abbildung von durchgezogenen Linien eingegrenzt. Die Eingrenzungen der Bereiche der einzelnen Seiten deuten an, dass der Inhalt einer Seite nicht scharf mit der Grenze einer Gitterzelle zusammenfallen muss. So enthalten im Beispiel sowohl die erste als auch die zweite Seite Tupel aus dem Gitterbereich zwischen 0 und ϵ_2 sowie zwischen $3\epsilon_3$ und $4\epsilon_3$. Ferner muss auch die mitten durch eine Gitterzelle gezogene Grenze zwischen den Bereichen zweier Seiten in der Realität nicht so scharf sein, wie in der Abbildung dargestellt. Das liegt daran, dass nicht garantiert ist, dass die Datenelemente des Datenraums bezüglich irgendeiner Dimension, z. B. y_3, aufsteigend sortiert in den Datenfenstern und damit in den Seiten liegen. Dies wird vielmehr im Allgemeinen nicht der Fall sein. Selbst die ursprünglich vorhandene Sortierung der Eingabedaten nach aufsteigenden Werten in der Dimension y_1 wird bei der Materialisierung durch die Sortierung der Tupel nach Epsilon-Grid-Ordnung in der Regel zerstört. Dies sei an einem einfachen Beispiel demonstriert.

Beispiel 4.4
Man betrachte die beiden aus demselben Datenstrom stammenden und numerische Attributwerte besitzenden zweidimensionalen Tupel A und B, wobei $A = (0.1, 0.3)$ und $B = (0.11, 0.2)$ gelte. Sei die erste Dimension diejenige, nach der die Tupel des Datenstroms aufsteigend sortiert sind. Dann liefert der Datenstrom offensichtlich A vor B, da $0.1 < 0.11$ gilt. Sei weiter $\epsilon_1 = \epsilon_2 = 0.1$. Dann gilt $\lfloor 0.1/0.1 \rfloor = 1 = \lfloor 0.11/0.1 \rfloor$ und $\lfloor 0.3/0.1 \rfloor = 3 > 2 = \lfloor 0.2/0.1 \rfloor$. Also ist B gemäß Epsilon-Grid-Ordnung kleiner als A und es gilt $B <_\epsilon A$. Sofern diese beiden Tupel gleichzeitig in einem zu materialisierenden Datenfenster enthalten sind, kommt B nach der Sortierung der Tupel des Fensters nach Epsilon-Grid-Ordnung im Zuge der Materialisierung plötzlich vor A. Entsprechend steht B dann natürlich auch im materialisierten Datenbestand des Datenfensters vor A und kann deshalb gegebenenfalls sogar in einer kleineren Seite stehen, z. B. dann, wenn B das letzte Tupel ist, das noch in die aktuelle Seite passt, während A bereits in eine neue Seite geschrieben werden muss. □

Da die Vorsortierung aber nur für die Aktualisierung der Ober- und Untergrenzen der Datenfenster benötigt wird und diese jeweils bereits vor dem erneuten Füllen und Materialisieren der Fenster erfolgt, spielt die Zerstörung der Vorsortierung durch die Epsilon-Grid-Ordnung im Zuge der Materialisierung keine Rolle. In Abbildung 4.8 wurden im Übrigen die Dimensionen y_2 und y_3 verwendet und auf die Dimension y_1 verzichtet, da y_1 die Dimension ist, nach der die Fenstergrenzen aktualisiert werden. Da die Datenfenster aber nur die Höhe ϵ_1 haben, hätte sich in der Abbildung der Bereich einer Seite in y_1-Richtung maximal über zwei Gitterblöcke erstrecken können. Um mehr Möglichkeiten bei der Gestaltung des Beispiels zu haben und Seiten auch über mehr als zwei Gitterblöcke ausdehnen zu können, wie z. B. mit der zweiten Seite über drei Blöcke in y_2-Richtung und vier Blöcke in y_3-Richtung, wurde y_1 in dem Bild weggelassen und stattdessen auf y_2 und y_3 zurückgegriffen. Diese beiden Dimensionen stellen auch die beiden einzigen Join-Dimensionen in dem Beispiel dar.

Zur Berechnung des Left-Outer-Bestmatch-Joins mit Einschränkungen müssen nun die Gitterblöcke entsprechend der Epsilon-Grid-Ordnung abgearbeitet werden. Im Beispiel muss also erst die Dimension y_3 von links nach rechts vollständig traversiert werden, bevor in der Dimension y_2 der Übergang auf den nächsthöheren Gitterblock erfolgen kann. Dieser wird dann wiederum in y_3-Richtung komplett durchlaufen, usw. Dies muss natürlich für die sortierten und materialisierten Datenräume beider Datenfenster W_R und W_S geschehen. Um für die Berechnung des Join-Ergebnisses relevante Kombinationen von Tupeln zu erhalten, müssen jeweils solche Seiten P_R^i von W_R und P_S^j von W_S gemeinsam in den Hauptspeicher geladen und verarbeitet werden, deren Inhalte in jeder Dimension des Datenraums höchstens um den jeweiligen Maximalabstand ϵ_i der betreffenden Dimension auseinander liegen. In Abbildung 4.8 sind das alle Seiten, deren Grenzen sich irgendwo berühren. Abbildung 4.9 zeigt, welche Kombinationen von Seiten geladen und verarbeitet werden müssen, wenn die materialisierten Datenräume beider Datenfenster W_R und W_S die in Bild 4.8 dargestellte Gestalt haben. Diese Annahme dient nur der Übersichtlichkeit des Beispiels. In der Realität werden die Datenräume in aller Regel unterschiedlich gestaltet sein. Schließlich hängt die Aufteilung der Gitterblöcke auf die Seiten von der Verteilung der Attributwerte der Tupel ab. In Bild 4.9 ist dargestellt, in welcher Reihenfolge die einzelnen Seiten der beiden Datenfenster geladen werden, damit alle relevanten Kombinationen gebildet werden können. Die grauen Felder stellen dabei irrelevante Kombinationen von Seiten dar, da die Attributwerte der in den jeweiligen Seiten enthaltenen Tupel den vorgegebenen Höchstabstand überschreiten und diese Tupel daher nicht zur Erzeugung von Ergebnispaaren bei der Join-Berechnung beitragen können. Man betrachte beispielsweise die beiden Seiten P_R^1 und P_S^3. Aus Abbildung 4.8 geht hervor, dass alle Datenelemente der dritten Seite in der Dimension y_2 um mehr als ϵ_2 von den Datenelementen der ersten Seite entfernt sind. Deshalb ist die Kombination dieser Seiten für die Bestimmung des Join-Ergebnisses nicht relevant. In Bild 4.9 symbolisieren die schwarzen Ovale das Laden einer Seite P_R^i von W_R, die weißen das Laden einer Seite P_S^j von W_S vom Hintergrundspeicher in den Hauptspeicher. Die Pfeile deuten die Reihenfolge an, in der die Seiten geladen werden. Entsprechend der Aufteilung des Datenraums auf die einzelnen Seiten sind diese dann zu verarbeiten. Zunächst erfolgt das Laden der beiden ersten Seiten beider Eingaben. Daraus ergibt sich die Kombination (P_R^1, P_S^1). Aus Abbildung 4.8

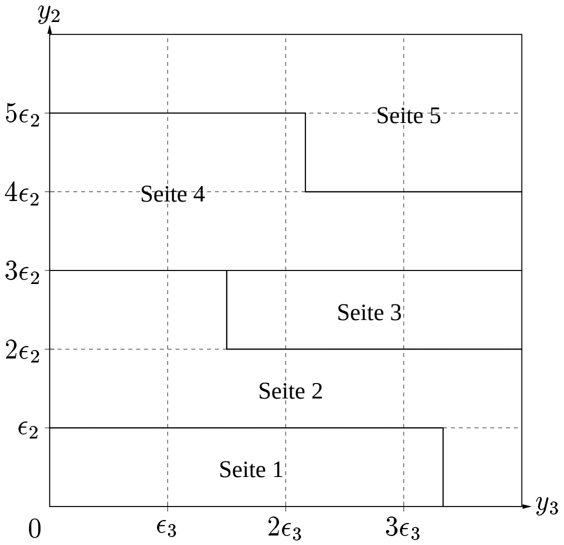


Abb. 4.8: Einteilung des Datenraums

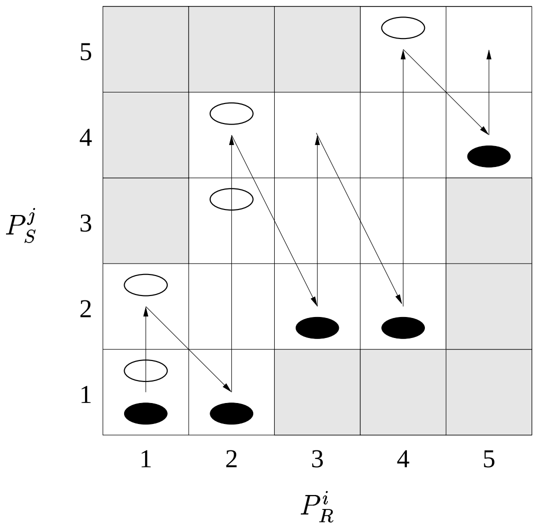


Abb. 4.9: Kombinieren der Seiten

ist ersichtlich, dass die erste Seite des Datenfensters W_R außer mit der ersten auch noch mit der zweiten Seite von W_S zu verbinden ist. Daher wird als nächstes die Seite P_S^2 geladen und die Kombination (P_R^1, P_S^2) gebildet. Danach sind alle Seiten von W_S, die für die Seite P_R^1 von W_R von Bedeutung sind, betrachtet. Deshalb kann die nächste Seite des Datenfensters W_R geladen werden. Dies ist die Seite P_R^2. Relevante Kombinationen für diese Seite sind nach Bild 4.8 (P_R^2, P_S^1), (P_R^2, P_S^2), (P_R^2, P_S^3) und (P_R^2, P_S^4). Entsprechend geht es weiter, bis alle zu betrachtenden Paare von Seiten abgearbeitet wurden.

Die Aufgabe des Scheduling-Algorithmus ist es nun, die benötigten Seiten in einer derartigen Reihenfolge vom Hintergrundspeicher in den Hauptspeicher zu laden, dass die Anzahl der Ladevorgänge und damit auch die dafür benötigte Zeit während der Join-Berechnung möglichst gering ausfällt. Dabei sei die Größe des zur Verfügung stehenden Hauptspeichers auf insgesamt $m = m_R + m_S$ Seiten begrenzt. Falls der Hauptspeicher bereits voll ist, wenn eine neue Seite geladen werden muss, wird nach der LRU-Strategie diejenige Seite aus dem Speicher verdrängt, die am längsten nicht mehr gebraucht wurde. Abbildung 4.10 veranschaulicht die nötigen Ladevorgänge des naiven Ansatzes, der nach [BBKK01] auch Gallop-Modus genannt wird, anhand eines weiteren Beispiels. Wiederum bedeuten schwarze Ovale das Laden einer Seite P_R^i und weiße Ovale das Laden einer Seite P_S^j. Grau unterlegte Felder kennzeichnen Kombinationen von Seiten, die aufgrund der Einschränkung auf eine ϵ-Umgebung nicht relevant sind, und die Pfeile zeigen die Reihenfolge der Ladevorgänge an. Der Gallop-Modus arbeitet auf dem neuen Beispiel so, wie es bereits für Abbildung 4.9 erklärt wurde, d. h. für jede Seite von W_R lädt das Verfahren alle Seiten von W_S in den Speicher, die mit der aktuell geladenen Seite von W_R zu kombinieren sind. Diese Vorgehensweise ist so lange optimal, wie keine später noch benötigten Seiten aus dem Hauptspeicher verdrängt werden müssen. Sobald dies aufgrund der Speicherplatzbeschränkung geschieht, kann die Methode aber sehr ineffizient werden. Sei im Folgenden $m = 4$, d. h. der Hauptspeicher hat Platz für vier Seiten. Bezogen auf

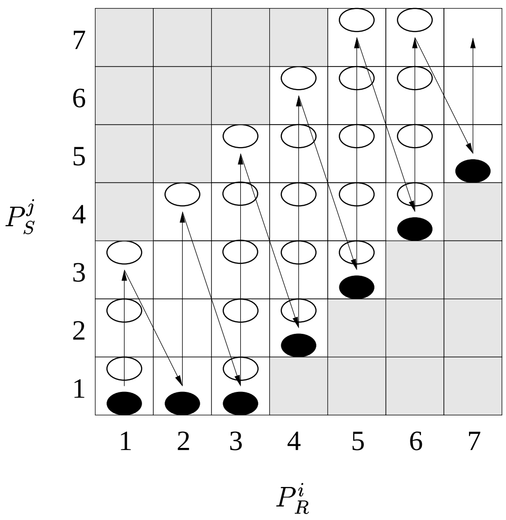


Abb. 4.10: Gallop-Modus

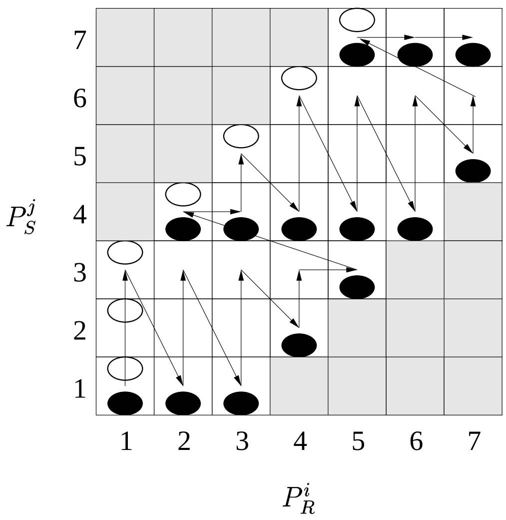


Abb. 4.11: Crabstep-Modus

Abbildung 4.9 ist der naive Ansatz dann optimal, denn hier werden stets nur solche Seiten verdrängt, die das Verfahren im weiteren Verlauf der Berechnung nicht mehr benötigt. Jede neu geladene Seite von W_R ersetzt die vorherige, so dass immer nur eine Seite des linken Datenfensters im Speicher ist. Wird die vierte Seite von W_S geladen, muss die erste Seite weichen. Diese ist jedoch für die verbleibenden Seiten aus W_R nicht relevant und muss deshalb im restlichen Verlauf der Berechnung nicht noch einmal geladen werden. Anders stellt sich die Situation aber in Abbildung 4.10 dar. Wiederum sei $m = 4$. Dann verdrängt bei der Bearbeitung von P_R^2 die vierte Seite P_S^4 von S die erste Seite P_S^1 aus dem Speicher. Diese ist aber für die Bearbeitung der nächsten Seite P_R^3 von W_R wieder nötig. Deshalb muss sie dort erneut geladen werden. Dabei verdrängt sie aber die bis dahin am längsten nicht mehr benötigte Seite. Dies ist die Seite P_S^2. Diese wird aber wiederum unmittelbar nach der Verarbeitung der eben geladenen Seite P_S^1 wieder gebraucht. Also muss auch sie erneut geladen werden. Dabei verdrängt sie ihrerseits aber die Seite P_S^3, die ebenfalls im nächsten Schritt wieder benötigt wird und deshalb geladen werden muss. Dieser Vorgang setzt sich für die weiteren Seiten von W_S und W_R fort und führt zu sehr vielen Ladevorgängen, wodurch das Verfahren relativ ineffizient wird. Man bezeichnet diese Erscheinung als Seitenflattern.

Um diesem Problem zu begegnen, wurde der Crabstep-Modus entwickelt. Er ist für die oben beschriebene Situation in Abbildung 4.11 dargestellt. Die Grundidee des Verfahrens ist relativ einfach. Falls durch das Laden einer neuen Seite des Datenfensters W_S eine bereits im Speicher befindliche Seite dieses Fensters verdrängt würde, wird diese neue Seite zunächst nicht geladen. Stattdessen setzt die Methode alle gerade im Speicher befindlichen Seiten P_S^j fest, lädt nacheinander alle für eine Kombination mit diesen Seiten relevanten Seiten des linken Datenfensters W_R und berechnet jeweils auf diesen Daten den Join. Danach werden alle Seiten des rechten Datenfensters W_S aus dem Hauptspeicher entfernt und die Berechnung fährt entsprechend mit der nächsten Seite von W_S, in Bild 4.11 sind das die Seiten P_S^4 bzw. P_S^7, fort. Während beim Gallop-Modus also jede Seite P_R^i höchstens einmal und die Seiten P_S^j mitunter mehrfach geladen werden, ist dies beim Crabstep-

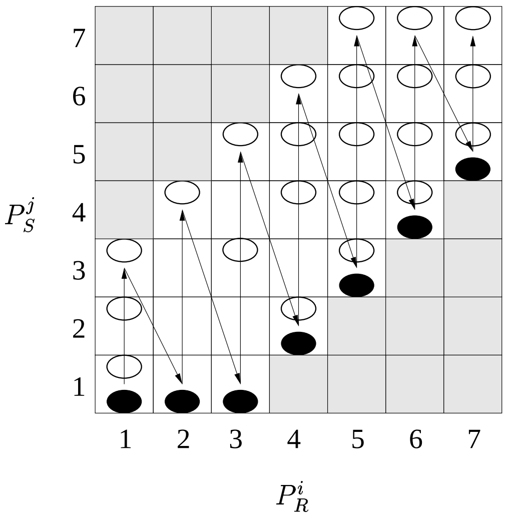


Abb. 4.12: Gallop-Modus mit MRU-Strategie

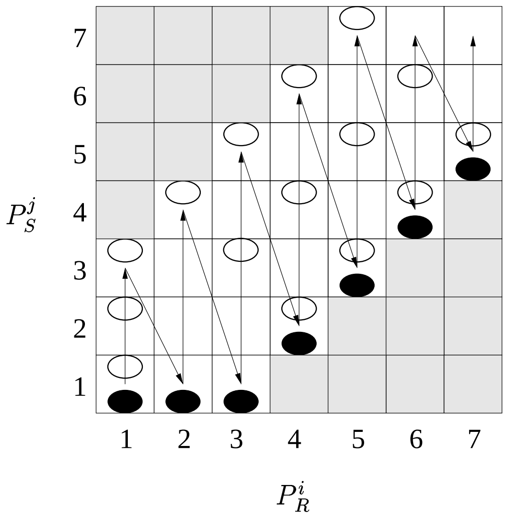


Abb. 4.13: Gallop-Modus mit modifizierter MRU-Strategie

Modus genau umgekehrt. Hier müssen zwar jetzt einige Seiten von W_R mehrmals geladen werden, dafür aber jede Seite von W_S höchstens einmal. Insgesamt verringert sich dadurch die Anzahl an Seitenzugriffen deutlich, im Beispiel von 30 auf 21. Dies entspricht einer Reduzierung um 30 Prozent.

Statt der LRU-Strategie zur Verdrängung von Seiten bei vollem Hauptspeicher kann im Gallop-Modus auch die MRU-Strategie eingesetzt werden. Bei dieser Strategie wird nicht die Seite verdrängt, auf die am längsten nicht mehr zugegriffen wurde, sondern die Seite, auf die der bis dahin letzte Zugriff stattgefunden hat. Abbildung 4.12 zeigt die bei dieser Strategie nötigen Ladevorgänge für das bekannte Beispiel. Hier kommt der Gallop-Modus auf 29 Seitenzugriffe und damit auf nur einen Seitenzugriff weniger als bei der LRU-Strategie. Er ist damit immer noch deutlich schlechter als der Crabstep-Modus. Im Allgemeinen dürfte die MRU-Variante sogar oft schlechter sein als LRU. So muss dieses Verfahren im Beispiel ab der Seite P_R^5 der linken Eingabe jede benötigte Seite der rechten Eingabe S bei jedem Zugriff erneut laden. Die Ursache hierfür liegt darin, dass sich nach der Verarbeitung der Seite P_R^4 die beiden Seiten P_S^1 und P_S^2 aufgrund der MRU-Verdrängungsstrategie noch im Hauptspeicher befinden und auch im weiteren Verlauf der Berechnung nicht mehr verdrängt werden. Diese beiden Seiten von S blockieren also, obwohl sie für die weitere Berechnung nicht mehr benötigt werden, zwei der insgesamt drei für Seiten von S im Hauptspeicher zur Verfügung stehenden Plätze. Damit fallen zwei Drittel des gesamten Hauptspeicherpuffers der rechten Eingabe für den Rest des Berechnungsvorgangs weg. Somit kann in der Folge nur noch jeweils eine neue Seite von S im Hauptspeicher gepuffert werden, weshalb jede einzelne Seite der rechten Eingabe bei Bedarf erneut geladen werden muss.

Eine Verbesserung des Verhaltens der MRU-Strategie ist durch folgende Modifikation möglich. Bei jedem Laden einer neuen Seite von R in den Hauptspeicher wird überprüft, ob die aktuell geladenen Seiten von S für die Join-Berechnung mit der neuen Seite von R noch relevant sind. Gibt es Seiten von S im Speicher, deren Tupel allesamt außerhalb der

ϵ-Umgebungen der Tupel der neuen Seite von R liegen, so werden diese Seiten aus dem Hauptspeicher entfernt. Man beachte, dass diese Überprüfung beim LRU-Ansatz unnötig ist, da hier ohnehin diejenigen Seiten aus dem Hauptspeicher verdrängt werden, auf die am längsten nicht mehr zugegriffen wurde. Mit der beschriebenen Modifikation der MRU-Strategie wird jede Seite, die nicht mehr gebraucht wird, auf jeden Fall entfernt, sobald der von ihr belegte Speicherplatz für eine neue Seite benötigt wird. In Abbildung 4.13 sind die für das oben eingeführte Beispiel nötigen Ladevorgänge des Gallop-Modus mit modifizierter MRU-Strategie dargestellt. Dieser Ansatz erfordert bei dem verwendeten Beispiel 22 Seitenzugriffe. Damit liegt er immer noch, wenn auch nur geringfügig, über dem nach wie vor besseren Crabstep-Modus, der sich mit 21 Seitenzugriffen begnügt. Insgesamt ist der Crabstep-Modus damit die beste Wahl. Dieser Eindruck verstärkt sich noch, wenn man sich überlegt, wieviele Seitenzugriffe auf den Hintergrundspeicher die einzelnen Verfahren benötigen würden, wenn im Beispiel jede mögliche Kombination von Seiten der beiden Eingaben R und S relevant wäre. Dann gäbe es in den Abbildungen 4.10 bis 4.13 keine grauen Felder. In diesem Fall käme der Crabstep-Modus offensichtlich auf 28 Seitenzugriffe, der Gallop-Modus mit LRU-Strategie auf 56 und mit MRU-Strategie auf 35. Die oben beschriebene Modifikation des MRU-Ansatzes hat hier keinen Effekt, da es keine Seiten von S gibt, die für irgendeine Seite von R irrelevant wären.

Algorithmus 4.5 zeigt den Scheduling-Algorithmus. Er erhält als Eingabe den aktuellen Zustand O der Join-Berechnung, die relevanten Teile der Datenfenster W_R und W_S, die zu verarbeiten sind, und damit auch die Referenzen auf die Seiten auf dem Hintergrundspeicher, die deren materialisierte Datenelemente enthalten, sowie die partielle Ordnung $\prec_{y_1,\ldots,y_d}$. Die Seiten, auf die sich der gesamte Inhalt der Datenfenster verteilt, werden mit $P_R^1, \ldots, P_R^{p_R}$ und $P_S^1, \ldots, P_S^{p_S}$ bezeichnet. Mit f_R bzw. f_S werde der Index der ersten in die Berechnung einzubeziehenden Hintergrundspeicherseite von W_R bzw. W_S benannt. Analog bezeichnen l_R und l_S den Index der letzten einzubeziehenden Seite des zugehörigen Datenfensters. Dies ist nötig, da bei der Join-Berechnung, wie in Abschnitt 4.2.2 dargestellt, meist nur Teile der Datenfenster kombiniert werden. Weiter stehen M_R und M_S wieder für den Hauptspeicherpuffer für Seiten von W_R und W_S und $M_S.lastPage$ für die letzte Seite in M_S. Schließlich beziehen sich h_R und h_S auf eine einzelne Seite von W_R bzw. W_S im Hauptspeicher, sowie $P.first$ und $P.last$ für eine beliebige Seite P auf das erste bzw. letzte Datenelement in der betreffenden Seite. Der Algorithmus wurde im Vergleich zu der Version in [BBKK01], wo er für einen Self-Join verwendet wird, für die Anwendung auf zwei verschiedenen Eingaben erweitert.

Zunächst werden die Indizes i und j initialisiert und in der zweiten Zeile lädt das Verfahren mittels der Funktion *Load* die erste relevante Seite von W_R in den Hauptspeicher. Die äußere while-Schleife läuft dann so lange, bis alle relevanten Seiten beider Eingaben verarbeitet worden sind. Dabei beginnt der Algorithmus im Gallop-Modus und verbleibt in diesem Modus, solange im Hauptspeicher neben den bereits geladenen Seiten von W_S und der einen geladenen Seite von W_R noch Platz ist, solange also $|M_S| + 1 < m$ gilt.

Der Gallop-Modus beginnt in Zeile 5 von Algorithmus 4.5 und endet in Zeile 19. Er lädt die nächste relevante Seite von W_S, sofern vorhanden, in M_S und ersetzt falls nötig die aktuell geladene Seite von W_R durch die nächste. Dies muss immer dann geschehen, wenn alle Tupel in der zuletzt in M_S hinzugefügten Seite bereits außerhalb des interessanten Bereichs

Algorithmus 4.5 Scheduling-Algorithmus ($Schedule$)

Eingabe: Zustand O, relevante Teile von W_R und W_S, partielle Ordnung $\prec_{y_1,\ldots,y_d}$

$j \leftarrow f_S - 1$
2: $i \leftarrow f_R;\ h_R \leftarrow Load(P_R^i)$
while $(i \leq l_R) \vee (j \leq l_S)$ **do**
4: **if** $(|M_S| + 1) < m$ **then** /* Gallop-Modus */
$notloaded \leftarrow true$
6: **if** $j < l_S$ **then**
$j \leftarrow j + 1;\ M_S \leftarrow M_S \cup Load(P_S^j);\ notloaded \leftarrow false$
8: **end if**
if $((h_R.last + \vec{\epsilon}) <_\epsilon M_S.lastPage.first) \vee notloaded$ **then**
10: $i \leftarrow i + 1;\ h_R \leftarrow Load(P_R^i)$
for all $h_s \in M_S$ **do** /* nicht mehr benötigte Seiten von S entfernen */
12: **if** $h_s.last <_\epsilon (h_R.first - \vec{\epsilon})$ **then**
$M_S \leftarrow M_S \setminus \{h_s\}$
14: **end if**
end for
16: $Update(O, \{h_R\}, M_S, \prec_{y_1,\ldots,y_d})$
else
18: $Update(O, \{h_R\}, \{M_S.lastPage\}, \prec_{y_1,\ldots,y_d})$
end if
20: **else** /* Crabstep-Modus */
$nextpage \leftarrow i;\ i \leftarrow i + 1;\ h_R \leftarrow Load(P_R^i)$
22: **while** $M_S.lastPage.last \not<_\epsilon (h_R.first - \vec{\epsilon})$ **do**
for all $h_S \in \{h'_S \in M_S \mid h'_S.last \not<_\epsilon (h_R.first - \vec{\epsilon})\}$ **do**
24: $Update(O, \{h_R\}, \{h_S\}, \prec_{y_1,\ldots,y_d})$
end for
26: $i \leftarrow i + 1;\ h_R \leftarrow Load(P_R^i)$
end while
28: $i \leftarrow nextpage;\ h_R \leftarrow Load(P_R^i);\ M_S \leftarrow \emptyset$
end if
30: **end while**

um die Datenelemente der aktuellen Seite h_R liegen oder wenn bereits alle relevanten Seiten von W_S geladen wurden und deshalb im gegenwärtigen Berechnungsschritt keine weitere mehr geladen werden konnte. Diese Bedingungen sind in Zeile 9 des Algorithmus formal ausgedrückt. Sollte die aktuell geladene Seite h_R von W_R durch die nächste Seite ersetzt worden sein, so entfernt das Verfahren noch alle Seiten h_S aus M_S, die für die neue Seite h_R nicht mehr relevant sind, bevor in Zeile 16 das Zwischenergebnis O der Join-Berechnung durch den Aufruf der Funktion $Update$ aktualisiert wird. Der entsprechende Aktualisierungsalgorithmus ist in Algorithmus 4.6 dargestellt. Falls h_R in der zehnten Zeile tatsächlich mit einer neuen Seite geladen wurde, dann muss nun der Join der Tupel in h_R mit allen Tupeln in M_S berechnet werden. Anderenfalls ist die bereits in früheren Schritten in h_R befindliche Seite dort verblieben und muss nur mit der in der siebten Zeile neu in M_S

hinzugekommenen Seite kombiniert werden. Die Aktualisierung des Zwischenergebnisses mit h_R und den anderen Seiten in M_S hat dann schon in früheren Schritten stattgefunden. Dieses Vorgehen entspricht der in Abbildung 4.10 gezeigten Situation.

Sollte dem Verfahren in der vierten Zeile der Platz für neue Seiten P_S^j im Hauptspeicher ausgehen, so schaltet es in den Crabstep-Modus um. Dieser ist in den Zeilen 21 bis 28 von Algorithmus 4.5 dargestellt. Zunächst sichert dieser Modus den Index i der aktuell geladenen Seite von W_R in der Variable *nextpage*, um später an diese Stelle zurückkehren zu können. Dann lädt er die nächste Seite von W_R in h_R. In der folgenden while-Schleife werden so lange aufeinanderfolgende Seiten von W_R geladen und in der eingeschachtelten for-Schleife mit allen für die jeweilige Seite h_R relevanten Seiten in M_S kombiniert, bis eine Seite h_R geladen wurde, für die es in M_S keine relevante Seite mehr gibt. Danach lädt das Verfahren in Zeile 28 die Seite von W_R mit dem gemerkten Index *nextpage* in h_R und leert M_S, da alle bis dahin geladenen Seiten von W_S nun vollständig bearbeitet worden sind. Im nächsten Durchlauf der äußeren while-Schleife fährt das Verfahren dann mit der aktuell in h_R geladenen Seite im Gallop-Modus fort. Dies entspricht der in Abbildung 4.11 dargestellten Vorgehensweise. Das Laden von $P_R^{nextpage}$ in Zeile 28 und das Fortfahren im Gallop-Modus findet dabei im Bild seine Entsprechung in den Sprüngen von der Zelle mit den Koordinaten $(5,3)$ nach $(2,4)$ bzw. von $(7,6)$ nach $(5,7)$. Tatsächlich funktioniert dies im Algorithmus ein wenig komplizierter. Bei der Bearbeitung des in der Abbildung dargestellten Beispiels mit dem angegebenen Algorithmus wird vor dem ersten Umschalten in den Crabstep-Modus der Index 1 in der Variable *nextpage* abgelegt, vor dem zweiten Umschalten der Index 4. Daher schaltet der Algorithmus am Ende des Crabstep-Modus zunächst nicht wie in dem Bild dargestellt auf das Rechteck $(2,4)$ bzw. $(5,7)$ um, sondern auf $(1,4)$ bzw. $(4,7)$. Da im Beispiel die Kombination der Seite P_R^1 mit der Seite P_S^4 und die Kombination der Seite P_R^4 mit der Seite P_S^7 für die Join-Berechnung wegen der Einschränkung auf die ϵ-Umgebung nicht relevant ist, müssen diese Kombinationen nicht betrachtet werden. Der Algorithmus bemerkt dies in Zeile 9 im Gallop-Modus und lädt danach in Zeile 10 sofort die nächste Seite von W_R in h_R. Damit befindet er sich an der in der Abbildung dargestellten Stelle $(2,4)$ bzw. $(5,7)$. Der Zwischenschritt ist in dem Bild aus Gründen der Übersichtlichkeit nicht eingezeichnet. In der Praxis kann es natürlich durchaus auch vorkommen, dass das ursprüngliche Sprungziel des Algorithmus, hier also das Rechteck mit den Koordinaten $(1,4)$ oder das mit den Koordinaten $(4,7)$, für die Berechnung relevant ist. Auch das erkennt das Verfahren, denn dann ist die if-Abfrage in Zeile 9 nicht erfüllt und die entsprechenden Kombinationen von Seiten werden korrekt in die Berechnung miteinbezogen.

In Algorithmus 4.6 ist die Arbeitsweise der Funktion *Update* gezeigt. Es handelt sich hierbei um eine modifizierte Variante des Nested-Loops Algorithmus 4.2 von Seite 35 zur Berechnung des Left-Outer-Bestmatch-Joins. Die Variante unterscheidet sich von der ursprünglichen Version im Wesentlichen dadurch, dass die innere Schleife nicht über die rechte Eingaberelation, sondern über den Inhalt des Puffers O für das aktuelle Zwischenergebnis läuft. Der Update-Algorithmus erhält beim Aufruf vom Scheduling-Algorithmus neben dem Zustand O und der partiellen Ordnung $\prec_{y_1,\ldots,y_d}$ auch zwei Mengen von Hauptspeicherseiten H_R und H_S übergeben. Diese enthalten die vom Scheduling-Algorithmus in den Hauptspeicher geladenen und nun zu betrachtenden Seiten $h_R \in H_R$ und $h_S \in H_S$.

Algorithmus 4.6 Update-Algorithmus (*Update*)

Eingabe: Zustand O, Hauptspeicherseiten H_R und H_S, partielle Ordnung $\prec_{y_1,\ldots,y_d}$

for all relevant $r \in H_R$ **do**
2: **for all** relevant $s \in (H_S \cap U_S^{\epsilon_1,\ldots,\epsilon_d}(r))$ **do**
$dominated \leftarrow false$
4: **for all** $o \in O$ **do**
if $o.r' = r$ **then**
6: **if** $o \prec_{y_1,\ldots,y_d} (r \times s)$ **then**
$dominated \leftarrow true$
8: **break**
else if $(r \times s) \prec_{y_1,\ldots,y_d} o$ **then**
10: $O \leftarrow O \setminus \{o\}$
end if
12: **end if**
end for
14: **if** $\neg dominated$ **then**
$O \leftarrow O \cup (r \times s)$
16: **end if**
end for
18: **end for**

Die beiden äußeren Schleifen des Verfahrens laufen über alle relevanten Tupel $r \in H_R$ und $s \in H_S$. Das sind alle Tupel in allen Seiten von H_R und H_S, die für die Join-Berechnung interessant sind, die also innerhalb der zu bearbeitenden Teile der Datenfenster liegen und die gegebenen Einschränkungen erfüllen. Da in den Seiten alle Datenelemente der zugehörigen Datenfenster enthalten sind, meist aber nur Teile der Fenster miteinander kombiniert werden[1] und die Grenzen dieser Teile aufgrund der Sortierung nach Epsilon-Grid-Ordnung nicht scharf zwischen bzw. in den Seiten verlaufen,[2] muss der Update-Algorithmus die relevanten Datenelemente aus den übergebenen Hauptspeicherseiten herausfiltern. Bei den Datenobjekten $s \in H_S$ ist bei dieser Filterung ferner zu berücksichtigen, dass die interessanten Tupel innerhalb der ϵ-Umgebung $U_S^{\epsilon_1,\ldots,\epsilon_d}(r)$ des jeweils aktuellen Tupels $r \in H_R$ der linken Eingabe liegen.

In der vierten Zeile des Algorithmus beginnt die über die Tupelpaare $o \in O$ laufende innere Schleife. Da das Zwischenergebnis O potenzielle Ergebnispaare von Tupeln $r \in R$ und $s \in S$ der beiden Eingabedatenströme enthält, dürfen für die Berechnung des Left-Outer-Bestmatch-Joins nur solche Paare $o \in O$ herangezogen werden, deren linkes Tupel $o.r'$ dem aktuellen Tupel $r \in H_R$ der äußeren Schleife entspricht. Es muss also $o.r' = r$ gelten, damit ein Paar $(r \times s)$ mit einem Paar o vergleichbar ist. Falls nun für ein solches o die Bedingung $o \prec_{y_1,\ldots,y_d} (r \times s)$ erfüllt ist, o also das Paar $(r \times s)$ dominiert, setzt der Algorithmus den Flag *dominated* auf *true* und die innere Schleife kann mittels eines break-Befehls sofort beendet werden. Das liegt daran, dass ein einmal dominiertes Tupelpaar auf keinen Fall mehr in das Zwischenergebnis aufgenommen werden kann. Dominiert stattdessen $(r \times s)$

[1] Vergleiche hierzu Algorithmus 4.3 auf Seite 46.
[2] Vergleiche hierzu Beispiel 4.4 auf Seite 55.

das Paar o, so wird o aus O entfernt. Das potenzielle Ergebnispaar o wurde also durch das Paar $(r \times s)$ invalidiert und kann deshalb nicht Teil des korrekten Endergebnisses sein. In diesem Fall muss die innere Schleife bis zum Ende weiterlaufen, da das Paar $(r \times s)$ noch weitere Paare $o \in O$ invalidieren könnte. Letztendlich wird $(r \times s)$ dann in Zeile 15 als neues potenzielles Ergebnispaar in O aufgenommen.

Man beachte, dass der Abbruch der inneren Schleife in Zeile 8 keinen Einfluss auf die Korrektheit des Verfahrens hat. Ein Tupelpaar $(r \times s)$, das von irgendeinem potenziellen Ergebnispaar $o \in O$ dominiert wird, kann seinerseits niemals irgendein anderes Paar $o' \in O$ dominieren. Da die verwendete Vergleichsordnung $\prec_{y_1,\ldots,y_d}$ wie jede Ordnung transitiv ist, würde o' in einem solchen Fall auch von o dominiert. Damit ist es aber unmöglich, dass sich o' zusammen mit o im aktuellen Zwischenergebnis O befindet, denn entweder wäre o' von o invalidiert worden oder o' hätte gar nicht erst in O eingefügt werden können, da es von dem bereits in O enthaltenen o dominiert worden wäre. Aus demselben Grund kann es umgekehrt auch nicht passieren, dass ein Paar $(r \times s)$ ein Paar $o' \in O$ invalidiert und dann seinerseits von einem anderen Paar $o \in O$ dominiert wird. Insgesamt kann festgehalten werden, dass sich für ein Tupelpaar $(r \times s) \in (R \times S)$ bei einem kompletten Durchlauf durch die innere Schleife für alle $o \in O$ die if-Bedingungen in der sechsten und neunten Zeile gegenseitig ausschließen, d. h. es kann für ein festes $(r \times s)$ nur entweder die eine oder die andere Bedingung für ein oder mehrere $o \in O$ erfüllt sein. Dies setzt natürlich voraus, dass der Inhalt von O konsistent ist, also keine sich gegenseitig dominierenden Tupelpaare enthält. Da aber nur Algorithmus 4.6 Paare zu O hinzufügt und dabei die Konsistenzbedingung offensichtlich nicht verletzt, ist diese Voraussetzung immer erfüllt.

Der Ansatz, die innere Schleife des Nested-Loops Algorithmus über den Puffer O mit den Zwischenergebnissen laufen zu lassen und die Invalidierung bereits darin enthaltener Tupelpaare zuzulassen, ist notwendig, da aufgrund der fensterbasierten Berechnung bei einem Lauf des Update-Algorithmus mitunter noch nicht alle für die Bestimmung des korrekten Ergebnisses benötigten Datenelemente vorliegen. Diese werden dann in späteren Läufen verarbeitet, so dass hier später noch einmal bessere Kombinationen gefunden werden können als bei früheren Berechnungsschritten. Dann muss es natürlich auch möglich sein, das Zwischenergebnis entsprechend zu korrigieren, also bereits enthaltene Tupelpaare zu entfernen und durch bessere zu ersetzen. Außerdem kann durch dieses Verfahren die Effizienz der Berechnung gesteigert werden, wenn z. B. der Algorithmus in einem frühen Stadium der Auswertung sehr gute Paare in O aufgenommen hat, die in der Folge viele weniger gute Paare von vornherein dominieren und dadurch eliminieren. Dann kann das Verfahren die innere Schleife oft schon sehr früh mit dem break-Befehl in der achten Zeile des Algorithmus abbrechen und dadurch viel Rechenzeit einsparen.

Wie bereits am Ende von Abschnitt 4.2.2 erläutert wurde, kann die fensterbasierte Methode auch zur Berechnung des ϵ-BMJ eingesetzt werden. Dazu muss die eigentliche Join-Berechnung vom ϵ-LOBMJ auf den ϵ-BMJ umgestellt werden. In Algorithmus 4.6 kann dies sehr einfach durch Weglassen der if-Abfrage in Zeile 5 erfolgen.

4.3.3 Materialisierungsvarianten

Die vorangegangenen Abschnitte haben gezeigt, wie die Datenfenster verschoben und ihre Inhalte eingelesen, aktualisiert und geladen werden können. Die folgenden Ausführungen stellen nun zwei mögliche Varianten vor, mit denen sich die Tupel in den Fenstern durch den Aufruf der Funktion *materialize* in Algorithmus 4.4 nach Epsilon-Grid-Ordnung sortiert auf dem Hintergrundspeicher materialisieren lassen.

Standard-Materialisierung

Die erste Variante wird als Standard-Materialisierung bezeichnet und lehnt sich in der Vorgehensweise an das Verfahren des externen Sortierens an. Dabei wird der Hauptspeicherpuffer des betreffenden Datenfensters, wie in Algorithmus 4.4 dargestellt, zunächst gefüllt, bis er nur noch für eine Seite Platz bietet. Danach werden alle Tupel im Hauptspeicher gemäß Epsilon-Grid-Ordnung sortiert. Dazu lässt sich ein normaler Hauptspeicher-Sortieralgorithmus verwenden, z. B. Quicksort. Anschließend müssen die im Hauptspeicher befindlichen Tupel gemäß der Verfahrensweise bei externem Sortieren mit den Tupeln des in früheren Schritten bereits materialisierten Teils des Datenfensters gemischt und in einem neuen Lauf wieder auf den Hintergrundspeicher geschrieben werden. Dazu lädt das Verfahren zunächst die erste Seite des materialisierten Teils vom Hintergrundspeicher in die freigebliebene Seite des Hauptspeicherpuffers und schreibt dann abwechselnd jeweils entweder das nächste Tupel dieser Seite oder das nächste Tupel des zuvor sortierten Hauptspeicherbereichs in den neuen Lauf auf dem Hintergrundspeicher, je nachdem, welches dieser Tupel gemäß Epsilon-Grid-Ordnung das kleinere ist. Ist die aktuell geladene Seite des bereits zuvor materialisierten Teils des Datenfensters vollständig abgearbeitet, wird sie aus dem Hauptspeicher entfernt und stattdessen die nächste Seite des bereits materialisierten Datenbestands geladen. Das Verfahren fährt fort, bis alle neuen Datenelemente aus dem Hauptspeicher mit den schon früher materialisierten Tupeln gemischt und der neue Gesamtdatenbestand korrekt nach Epsilon-Grid-Ordnung sortiert auf den Hintergrundspeicher geschrieben worden ist. Nach der Materialisierung kann der Hauptspeicher geleert werden und anschließend neue Datenobjekte aufnehmen.

Eine mögliche Modifikation der Standard-Materialisierung ist, den Hauptspeicherinhalt nach der Sortierung gemäß Epsilon-Grid-Ordnung nicht sofort mit dem bereits materialisierten Teil des Datenfensters zu mischen, sondern zunächst in einem eigenen Lauf auf den Hintergrundspeicher zu schreiben. Dies wird während der Aktualisierung des Inhalts eines Datenfensters nach der Berechnung neuer Fenstergrenzen so oft wiederholt, bis alle neu hinzukommenden Tupel in solchen Läufen gespeichert sind. Der letzte Lauf muss dabei natürlich auch geschrieben werden, wenn der Hauptspeicher nicht ganz voll geworden ist. Anschließend können alle Läufe und der bereits zuvor materialisierte Teil des Datenfensters miteinander gemischt und in einem Lauf gespeichert werden. Dazu muss das Verfahren die jeweils erste Seite eines jeden Laufs und des schon materialisierten Fensterteils in den Hauptspeicher laden und dann wieder wie oben gemäß der Vorgehensweise bei externem Sortieren zurück auf den Hintergrundspeicher schreiben. Der Vorteil dieses Ansatzes ist, dass einmal materialisierte Teile nicht so oft geladen und neu geschrieben werden müssen, wenn bei einem Aktualisierungsschritt eines Datenfensters so viele Tupel neu hinzukommen, dass sie während der Aktualisierung den Hauptspeicher mehrmals füllen. Reichen die neu hinzukommenden Datenelemente hingegen höchstens aus, um den Hauptspeicher einmal zu füllen, dann gibt es auch nur einen Lauf und die hier beschriebene Modifikation hat keinen Vorteil mehr gegenüber dem oben beschriebenen nicht modifizierten Verfahren. Genau das ist aber in der Praxis häufig der Fall. In das linke Datenfenster werden z. B. durch den fensterbasierten Algorithmus immer nur die Tupel neu eingelesen, die genau auf der neuen Obergrenze des Fensters liegen. Das ist, wie in Abschnitt 4.2.2 dargestellt, für die Korrektheit der Methode nötig. Daher werden in einem Aktualisierungsschritt wahrscheinlich meist eher wenige neue Tupel in das Fenster aufgenommen, die alle in den zur Verfügung stehenden Hauptspeicherpuffer passen. Außerdem ist es für die Modifikation erforderlich, dass die Anzahl der Läufe, die am Ende mit dem bereits materialisierten Teil zu mischen sind, nicht größer ist als $m - 1$, wobei m wiederum die Anzahl an gleichzeitig in den Hauptspeicher passenden Seiten sei. Schließlich muss für das Mischen jeweils eine Seite aus jedem Lauf und zusätzlich eine Seite aus dem schon materialisierten alten Datenbestand gleichzeitig im Speicher sein. Das lässt sich aber nicht garantieren, da im Voraus nicht bekannt ist, wieviele Läufe bei einer Aktualisierung eines Datenfensters entstehen können. Insgesamt ist diese Modifikation deshalb eher ungeeignet und wird im Weiteren nicht benutzt.

Grid-Materialisierung

Die zweite Materialisierungsvariante basiert auf der Erkenntnis, dass die Epsilon-Grid-Ordnung den Datenraum bezüglich der Join-Attribute y_i, wie in Abbildung 4.8 exemplarisch dargestellt, in Zellen mit den Kantenlängen ϵ_i einteilt. Dies lässt sich bei der Materialisierung wie folgt ausnutzen. Als Datenstruktur zur Speicherung der Datenelemente eines zu materialisierenden Datenfensters wird im Hauptspeicher eine d-dimensionale Matrix angelegt. Deren Einträge repräsentieren die entsprechenden Zellen der Epsilon-Grid-Ordnung und enthalten jeweils eine Liste derjenigen Seiten auf dem Hintergrundspeicher, welche die in der zugehörigen Zelle befindlichen Datenelemente enthalten. Aus einem gegebenen Tupel lässt sich dann einfach mittels der Definition der Epsilon-Grid-Ordnung der Index der Zelle in der Matrix bestimmen, in die das betreffende Tupel gehört. Mit Hilfe dieses Index ist es wiederum möglich, diese Zelle und damit auch die Seite, in die das Tupel geschrieben werden muss, schnell und effizient zu finden. Da es aber bei einer beliebigen Folge von Tupeln wegen der vielen Schreib- und Lesezugriffe relativ ineffizient wäre, für jedes Tupel den Index zu berechnen, die entsprechende Seite in den Hauptspeicher zu laden, das Tupel einzufügen und dann die Seite wieder auf den Hintergrundspeicher zurückzuschreiben, geht das Verfahren etwas anders vor. Zunächst wird der Hauptspeicher wieder mit Datenobjekten gefüllt, bis nur noch für eine Seite Platz ist. Analog zur Standard-Materialisierung sortiert das Verfahren den Hauptspeicherinhalt dann gemäß Epsilon-Grid-Ordnung. Nun kann die Seite für das erste zu speichernde Tupel aus dem Hauptspeicher wie oben beschrieben geladen und das Tupel eingefügt werden. Aufgrund der Sortierung der Datenelemente können auch alle anderen Tupel, die in dieselbe Gitterzelle gehören, sofort sequenziell in diese Seite geschrieben werden. Ist die Seite voll, so schreibt das Verfahren sie auf den Hintergrundspeicher zurück, legt eine neue Seite an, fügt sie am Ende der Liste von Seiten der aktuellen Gitterzelle ein und schreibt die weiteren Tupel in die neue Seite. Dies setzt sich fort, bis das erste Datenobjekt im Hauptspeicher angetroffen wird, das in eine andere Zelle gehört. Die Sortierung nach EGO garantiert, dass alle vorhandenen Tupel für die aktuelle Zelle dann bereits geschrieben worden sind. Die aktuell geladene Seite kann nun auf den Hintergrundspeicher zurückgeschrieben und im Hauptspeicher durch die letzte Seite aus der Liste der Gitterzelle, in die das nächste zu materialisierende Tupel gehört, ersetzt werden. Dies setzt sich fort, bis der gesamte Hauptspeicherpuffer materialisiert worden ist. Aufgrund des über den Datenraum gelegten Gitters in Form der d-dimensionalen Matrix wird diese Materialisierungsvariante als Grid-Materialisierung bezeichnet.

Vor- und Nachteile der beiden Varianten

Der große Vorteil der Standard-Materialisierung liegt darin, dass sie relativ einfach und ressourcenschonend implementiert werden kann. Außerdem wird durch das sequenzielle Schreiben der Datenelemente in die Seiten wenig Speicherplatz verschwendet, da die Seiten immer voll geschrieben werden, bevor das Verfahren eine neue anlegt. Somit kann es nur bei der zuletzt angelegten Seite und durch das Entfernen von Datenelementen zu nicht ganz vollen Seiten kommen. Der Nachteil der Standard-Variante ergibt sich aus dem Ansatz des externen Sortierens. Hierfür müssen beim Mischen der neuen Datenobjekte mit dem bereits materialisierten Datenbestand diese alten Daten erneut geladen, mit den neuen Daten gemischt und wieder geschrieben werden. Dies bedeutet zusätzlichen Lese- und Schreibaufwand.

Genau dieser Nachteil wird bei der Grid-Materialisierung eliminiert. Hier muss der alte Datenbestand bei der Materialisierung neuer Datenelemente mit Ausnahme der jeweils zum Schreiben benötigten Seite nicht eingelesen und erneut geschrieben werden. Das liegt daran, dass die Tupel während des Schreibvorgangs durch die Verteilung auf die zugehörigen Gitterzellen implizit sortiert und mit dem bereits materialisierten Datenbestand gemischt werden. Dies spart also I/O-Kosten. Allerdings ist dafür vor allem bei dünn besetzten Gitterzellen die Speicherverschwendung wesentlich höher, da jede Gitterzelle, die mindestens ein Tupel enthält, auch eine komplette Seite bekommt. Daher kann es viele Seiten auf dem Hintergrundspeicher geben, die nur sehr wenige Datenobjekte enthalten. Weiter gibt es dann im Vergleich zur Standard-Materialisierung natürlich auch mehr Seiten, um dieselbe Menge an Tupeln zu speichern. Dies führt zu einer größeren Zahl an Seitenladevorgängen bei der Join-Berechnung. Ein weiteres Problem der Grid-Variante ist die d-dimensionale Matrix. Sie enthält umso mehr Zellen, je mehr Dimensionen es gibt und je kleiner die Kantenlängen ϵ_i der einzelnen Zellen sind. Daher wird die Anzahl an Zellen für kleiner werdende ϵ_i und vor allem für eine steigende Anzahl d an Dimensionen schnell sehr groß. Dies führt letztendlich dazu, dass die Matrix irgendwann nicht mehr in den Hauptspeicher passt. Daher ist die Grid-Materialisierung nur bei größeren ϵ_i und wenigen Join-Dimensionen geeignet.

Insgesamt hängt es von der Anwendungssituation ab, welche der beiden Varianten die besseren Ergebnisse erzielt. Deshalb findet in Kapitel 6 unter anderem auch ein Vergleich dieser beiden Verfahren unter verschiedenen Bedingungen statt.

4.3.4 Diskussion

Offenbar erlauben die Sortierung nach Epsilon-Grid-Ordnung und der Einsatz des diese Sortierung ausnutzenden Scheduling-Algorithmus eine weitere Verbesserung der Effizienz der fensterbasierten Methode. Zwar führt dieser Ansatz zu einer zusätzlichen Verkomplizierung des Berechnungsverfahrens und zu einer beträchtlichen Erhöhung des Implementierungsaufwands, gleichzeitig ermöglicht er aber eine deutliche Verringerung der Laufzeiten. Dies ist auf die starke Reduzierung der Anzahl an Lesezugriffen auf Seiten des Hintergrundspeichers zurückzuführen. Gerade diese Zugriffe haben aufgrund der Zugriffslücke zwischen Haupt- und Hintergrundspeicher einen großen Anteil an den vom Verfahren zur Generierung von Ergebnistupeln benötigten Ausführungszeiten. Natürlich sollen diese möglichst klein gehalten werden, nicht zuletzt, um einen kontinuierlichen Strom von Ausgabetupeln zu erhalten. Insgesamt verspricht der Mehraufwand also eine lohnende Investition zu sein. Dies wird in Kapitel 6 noch konkret untersucht.

4.4 Ausnutzung von Punktierungen

Um die relativ strenge Voraussetzung vorsortierter Eingabedatenströme für die Anwendbarkeit des fensterbasierten Algorithmus abzuschwächen und das Verfahren universeller einsetzbar zu machen, sind verschiedene Ansätze denkbar. Einer der naheliegendsten und praktikabelsten ist die Ausnutzung von punktierten Datenströmen. Eine darauf aufbauende Modifikation des fensterbasierten Algorithmus soll exemplarisch demonstrieren, wie die Methode bei Bedarf an andere als die in Abschnitt 4.2.1 vorausgesetzten Bedingungen angepasst und damit die Voraussetzung der aufsteigenden Vorsortierung der Eingabedaten nach den Werten eines bestimmten Attributs gelockert werden kann. Sie ist jedoch nicht in der in den Kapiteln 5 und 6 vorgestellten und untersuchten Prototyp-Implementierung enthalten und wird deshalb über diese kurze Einführung hinaus im Rahmen dieser Arbeit nicht weiter betrachtet. Näheres zur Erweiterung des fensterbasierten Verfahrens für die Anwendbarkeit unter abgeschwächten Vorbedingungen findet sich in [KS02a]. Die folgenden Abschnitte geben eine kurze Einführung in punktierte Datenströme und beschreiben die nötigen Anpassungen, um den fensterbasierten Algorithmus auf solchen Strömen einsetzbar zu machen. Eine abschließende Diskussion verdeutlicht die Konsequenzen der Modifikation.

4.4.1 Punktierte Datenströme

Das Konzept der Punktierung von Datenströmen, wie es in [TM02] vorgestellt wird, sieht vor, dass Datenströme neben den normalen Datenelementen mit Nutzdaten auch Metadaten enthalten können, die eine Aussage über den Zustand des Stroms treffen. Diese Metadaten, bei denen es sich in der Regel um ein oder mehrere Prädikate handelt, nennt man Punktierungen. Sie werden im Verlauf des Datenstroms zwischen den Datenobjekten des Stroms übermittelt und bringen zum Ausdruck, dass nach ihrem Auftreten im Strom kein Datenelement mehr folgen wird, welches das Prädikat der Punktierung erfüllt. Ein

Beispiel für ein solches Prädikat wäre etwa, dass der Wert eines bestimmten Attributs eine festgelegte Grenze nicht überschreitet. Nach dem Eintreffen einer solchen Punktierung folgen dann nur noch Tupel aus dem Datenstrom, deren entsprechende Attributwerte größer sind als die in der Punktierung genannte Grenze. Sei beispielsweise a ein Attribut numerischen Typs. Dann könnte eine mögliche Punktierung aus dem Prädikat $a \leq 10$ bestehen. In diesem Fall ist sichergestellt, dass nach dieser Punktierung nur noch Tupel aus dem Datenstrom gelesen werden, für die $a > 10$ gilt. Bezogen auf das Beispiel 4.1 eines Wetterdienstes muss das Netzwerk nun nicht mehr alle Daten nach dem Wert des Zeitstempels streng aufsteigend sortiert liefern. Stattdessen können aufgrund unterschiedlicher Übertragungszeiten zwischen den einzelnen Sensoren und dem Zentralknoten auch einmal jüngere Messdaten vor älteren am Ziel ankommen. Durch den Einsatz von Punktierungen, die in bestimmten Abständen gesendet werden und garantieren, dass alle Messdaten, die bis zu einem bestimmten Zeitpunkt generiert wurden, nun bereits eingegangen sind, lässt sich auch in diesem Szenario der Left-Outer-Bestmatch-Join mit Einschränkungen mittels eines leicht modifizierten fensterbasierten Algorithmus berechnen.

4.4.2 Anpassung des fensterbasierten Algorithmus

Für den fensterbasierten Algorithmus bedeutet die Anpassung an das Konzept der Punktierungen eine deutliche Erweiterung der praktischen Einsatzmöglichkeiten. Es ist dadurch nämlich möglich, die strenge Sortierung der einzelnen Datenelemente der Eingabedatenströme nach aufsteigenden Attributwerten eines ausgewählten Attributs aufzugeben. Stattdessen verwendet man punktierte Datenströme mit Prädikaten ähnlich dem im vorhergehenden Abschnitt genannten. Dadurch wird eine Art unscharfe Sortierung der Eingaben erreicht. Zwischen zwei Punktierungen müssen die aus einem Datenstrom gelesenen Tupel zwar keiner Sortierung genügen, es ist aber garantiert, dass bestimmte Attributwerte innerhalb bestimmter Grenzen liegen. Es wird dabei davon ausgegangen, dass sich die Punktierungen auf ein einziges Attribut beziehen, das dann auch zur Aktualisierung der Grenzen der Datenfenster des fensterbasierten Algorithmus verwendet wird. Dieses Attribut entspricht somit in seiner Bedeutung dem Attribut, nach dessen Werten die Tupel der Datenströme in der nicht modifizierten Version des Algorithmus aus Kapitel 4.2 aufsteigend sortiert sind.

Die notwendigen Modifikationen an den Algorithmen 4.3 und 4.4 sind geringfügig und einfach. Sie betreffen das Einlesen von Datenelementen in die Datenfenster und die Aktualisierung der Fenstergrenzen. Bei jedem Einlesen von Tupeln in die Fenster kann nun nicht mehr beim Antreffen des ersten Tupels, das nicht mehr in das Fenster passt, aufgehört werden. Schließlich könnten aufgrund der unscharfen Sortierung später noch Tupel folgen, die innerhalb der aktuellen Fenstergrenzen liegen. Deshalb muss jetzt bis zur ersten Punktierung gelesen werden, die garantiert, dass alle Tupel, die in das aktuelle Fenster passen, schon geliefert worden sind. Wenn die Fensterobergrenze z. B. bei $y_1 = 0.1$ liegt, die erste Punktierung aus dem Prädikat $y_1 \leq 0.07$ und die zweite aus dem Prädikat $y_1 \leq 0.15$ besteht, dann müssen in jedem Fall alle Tupel bis zur zweiten Punktierung gelesen und in das zugehörige Datenfenster eingefügt werden. Selbstverständlich können dabei auch Datenelemente in das Fenster gelangen, die eigentlich außerhalb der Fenstergrenzen liegen, etwa ein Tupel mit dem Wert 0.12 für das Attribut y_1. Diese Datenelemente müssen in den folgenden Berechnungsschritten zunächst ignoriert werden, indem aus den Datenfenstern zur Berechnung nur die Tupel herausgefiltert werden, deren Wert für y_1 innerhalb der Grenzen des Fensters liegt. Bei der Aktualisierung der Fenstergrenzen ist außerdem darauf zu achten, dass nun für die entsprechenden Berechnungen in Algorithmus 4.3 nicht mehr der Wert $r.y_1$ des nächsten in das Fenster einzufügenden Tupels r benutzt werden darf. Vielmehr muss hier der kleinste y_1-Wert aller gemäß oben beschriebener Vorgehensweise bis zur aktuellen Punktierung gelesenen und in das Fenster aufgenommenen Tupel, die außerhalb der Fenstergrenzen liegen, verwendet werden. Sollte kein solches Tupel existieren, was in Ausnahmefällen vorkommen kann, z. B. wenn eine Punktierung mit einer Fensterobergrenze zusammenfällt, dann muss erst noch bis zur nächsten Punktierung weitergelesen werden.

4.4.3 Diskussion

Der größte Vorteil der Ausnutzung punktierter Datenströme liegt in der deutlichen Erweiterung der möglichen Einsatzbereiche des fensterbasierten Algorithmus zur Berechnung des Left-Outer-Bestmatch-Joins mit Einschränkungen. Da die Eingabedaten nicht mehr nach einem bestimmten Attribut streng aufsteigend sortiert sein müssen, vergrößert sich der Kreis der verwendbaren Datenquellen. Ein weiterer Vorteil besteht in der Möglichkeit, die Eingabedaten bei entsprechender Unterstützung durch die Datenströme bereits nach Epsilon-Grid-Ordnung sortiert anzuliefern und dadurch die Umsortierung der Daten bei der Materialisierung einzusparen. Intelligenten Datenquellen teilt man einfach mit, nach welcher Ordnung die gelieferten Datenelemente sortiert sein sollen. Der eigentliche Berechnungsalgorithmus muss sich dann nicht mehr darum kümmern. Bei der Sortierung nach EGO ist allerdings darauf zu achten, dass der Sortierung in den Datenströmen dieselbe ϵ-Umgebung zugrunde liegen muss wie dem Join-Algorithmus. Auch dies lässt sich intelligenten Datenquellen vor Beginn der Berechnung mitteilen. Es bleibt anzumerken, dass die angeführten weitreichenden Verbesserungen der Einsatzmöglichkeiten des Algorithmus mit relativ geringfügigen Modifikationen an dem ursprünglichen Verfahren und daher mit verhältnismäßig wenig Zusatzaufwand erreicht werden können.

Kapitel 5

Entwurf und Implementierung eines Prototyps

Während in den letzten beiden Kapiteln die Theorie des Bestmatch-Joins auf Datenströmen und der Implementierung des ϵ-LOBMJ behandelt wurde, geht es in diesem und dem nächsten Kapitel nun um die Praxis. Konkret wird in diesem Kapitel der im Rahmen dieser Arbeit durchgeführte Entwurf und die Implementierung eines auf den vorgestellten Algorithmen basierenden Prototyps zur Berechnung des ϵ-LOBMJ beschrieben. Es handelt sich dabei im Wesentlichen um einen Überblick über den Entwurfs- und Implementierungsvorgang mit den wichtigsten dort getroffenen Entscheidungen und möglichen Alternativen sowie um eine Zusammenstellung interessanter Erfahrungen, die im Laufe der Arbeit an der Implementierung gesammelt wurden. Zunächst folgen einige allgemeine Überlegungen zu Entwurf und Implementierung. Es schließen sich auf den Prototyp bezogene Betrachtungen zur verwendeten Programmiersprache Java an. Weiterhin folgen Ausführungen zur Realisierung der Join-Algorithmen und des Datengenerators, ehe das Kapitel mit einer Zusammenfassung schließt. Kapitel 6 analysiert den Prototyp anschließend unter Leistungsgesichtspunkten.

5.1 Allgemeines

Zunächst soll auf einige allgemeine Punkte, die während der Entwurfs- und Implementierungsphase eine Rolle gespielt haben, eingegangen werden. Dies beinhaltet neben der Zielsetzung der Prototyp-Implementierung auch das allgegenwärtige Iterator-Konzept und einige weitere Überlegungen.

5.1.1 Zielsetzung

Mit der Implementierung des Prototyps wurden zwei Hauptziele verfolgt:

- Zum einen sollten die entwickelten Algorithmen auf ihre Implementierbarkeit und praktische Einsetzbarkeit hin untersucht, gegebenenfalls korrigiert sowie an sich aus der Realisierung in einer Programmiersprache ergebende Erfordernisse angepasst werden.

- Zum anderen sollte anhand des Prototyps die Leistungsfähigkeit der Methoden im praktischen Einsatz an repräsentativen Testfällen gemessen, untersucht und miteinander verglichen werden. Dabei sollte insbesondere die erwartete Überlegenheit des fensterbasierten Verfahrens gegenüber dem naiven Ansatz bestätigt und größenordnungsmäßig erfasst werden.

Dieses und das Kapitel über die Leistungsmessungen zeigen, dass das Erreichen beider Ziele gelungen ist.

Bei der Implementierung selbst lag besonderes Augenmerk auf den Zielen Effizienz und Modularität.

Effizienz: Um aussagekräftige Ergebnisse bei den Leistungsmessungen zu erhalten, war es wichtig, die Algorithmen möglichst effizient zu realisieren. Dabei sind bei Verwendung der Programmiersprache Java einige Punkte zu beachten, die im weiteren Verlauf dieses Kapitels noch angesprochen werden.

Modularität: Die Zielsetzung einer modularen Implementierung dient vor allem dem Zweck der einfachen späteren Erweiterbarkeit des Prototyps, z. B. um neue Join-Algorithmen, Vergleichsordnungen, Materialisierungsvarianten, Sortieralgorithmen, Auswertungspläne und Benutzerschnittstellen.

5.1.2 Implementierungsumgebung

Die Implementierung erfolgte in der Programmiersprache Java unter Verwendung des Editors XEmacs ausschließlich auf Rechnern unter den Betriebssystemen UNIX und Linux. Für die Realisierung des Prototyps kam die zum damaligen Zeitpunkt aktuellste Version 1.4.0 des Sun Java Development Kit (JDK) zum Einsatz. Diese Version hat gegenüber der Vorgängerversion mit der Versionsnummer 1.3.1 zahlreiche Erweiterungen und Verbesserungen erfahren. Davon hat insbesondere auch die API profitiert, die neben vielen neuen Klassen und Funktionen auch in manchen Bereichen, insbesondere z. B. bei der Reflection-API `java.lang.reflect`, starke Effizienzverbesserungen aufweisen kann. Da die Prototyp-Implementierung auf mehrere der neuen API-Klassen und Funktionen zurückgreift, ist sie mit älteren Versionen als 1.4.0 nicht kompatibel.

Für die Erstellung objektorientierter Entwurfsdiagramme fand das Entwurfswerkzeug Together 6.0 von TogetherSoft Verwendung.

5.1.3 Paket-Übersicht des Prototyps

Der Prototyp besteht aus insgesamt fünf Paketen, die nachfolgend kurz beschrieben sind. Anhang B gibt einen Überblick über die Entwurfsdiagramme aller Pakete.

- Das Paket `bmj` bildet die Basis der Implementierung und enthält alle allgemein relevanten Teile des Prototyps. Dazu gehören neben der Benutzerschnittstelle auch global für den gesamten Prototyp verfügbare Konstanten und Methoden sowie das zentrale Iterator-Interface.
- Der Inhalt des Pakets `bmj.datatypes` besteht aus sämtlichen eigens für den Prototyp entwickelten Datentypen, z. B. für die Tupel, die Datenfenster und die Speicherseiten.
- In dem Paket `bmj.io` sind alle Klassen enthalten, die den Datenaustausch mit dem Hintergrundspeicher regeln. Dazu gehört das Einlesen von Eingabedaten ebenso, wie das Schreiben und Lesen von Hintergrundspeicherseiten mit materialisierten Tupeln.
- Das Paket `bmj.join` beinhaltet die einzelnen Implementierungen der eigentlichen Join-Algorithmen. Implementiert wurde neben dem naiven und dem fensterbasierten Algorithmus auch der Scheduling-Algorithmus zur weiteren Leistungsverbesserung der fensterbasierten Methode. Außerdem enthält dieses Paket auch die Klassen für die Auswertungspläne und die Vergleichsordnungen, anhand derer bei der Join-Berechnung bestimmt wird, ob ein Tupelpaar ein anderes dominiert.
- Die Bestandteile des Pakets `bmj.test` sind ein Datengenerator zur Erzeugung von Testdaten für den Prototyp sowie verschiedene Verteilungen für die zufällig generierten Attributwerte.

5.1.4 Iterator-Konzept

Das Iterator-Konzept ist das grundlegende Prinzip der Prototyp-Implementierung, was das Lesen und Weitergeben von Tupeln betrifft. Es kommt sowohl beim Einlesen von Tupeln aus den beiden Eingaben eines Joins, als auch bei der Entnahme von Ergebnistupeln aus dem Join-Ergebnis zur Anwendung. Das Interface `BMJIterator` im Paket `bmj` gibt die Signaturen der drei Methoden `open`, `next` und `close` eines Iterators vor. Es wird u. a. von der Klasse `Filescan` im Paket `bmj.io`, die das Einlesen der Eingabedateien vornimmt, als auch von den Klassen `NLJoin` und `BMJoin` im Paket `bmj.join`, welche die Implementierung des naiven bzw. fensterbasierten Algorithmus enthalten, implementiert. Die Aufgaben und das Verhalten der drei Methoden des Iterator-Interface sind wie folgt:

open: Initialisiert den Iterator für die weitere Benutzung. Dies sollte immer die erste Methode sein, die auf einem neu instanziierten Iterator aufgerufen wird. Insbesondere muss ein Aufruf dieser Methode erfolgt sein, bevor die Methode `next` erfolgreich ausgeführt werden kann. Wiederholte Aufrufe der `open` Methode sind jederzeit möglich und versetzen den Iterator jeweils wieder zurück in den initialisierten Anfangszustand. Im Prototyp ist eine solche Reinitialisierung zwar grundsätzlich immer möglich, wird aber an den Stellen, an denen der Iterator einen Datenstrom simuliert, nie vorgenommen. Damit bleibt die charakteristische Eigenschaft eines Datenstroms, dass jedes im Strom enthaltene Tupel genau einmal aus dem Strom gelesen werden kann, erhalten. Bei jeder Ausführung gibt die Methode die Metadaten der in der

Iteration enthaltenen Tupel zurück. Diese Metadaten bestehen aus den Namen und Typen der Attribute der Relation, über die iteriert wird.

next: Liefert das nächste Element der Iteration. Falls der Iterator keine weiteren Elemente enthält, gibt diese Methode `null` zurück. Damit ist `null` auch das vereinbarte Zeichen für das Ende eines Datenstroms. Ein Aufruf von `next` ist nur dann möglich, wenn bereits mindestens ein Aufruf von `open` und seit dem letzten Aufruf von `open` kein Aufruf von `close` erfolgt ist. Wird `next` hingegen auf einem nicht initialisierten Iterator aufgerufen, so wirft die Methode eine `IllegalStateException`.

close: Schließt den Iterator und nimmt alle nötigen Aufräumarbeiten vor. Ein Aufruf dieser Methode ist jederzeit möglich. Ist seit der Instanziierung des Iterators oder seit dem letzten Aufruf von `close` kein Aufruf von `open` erfolgt, so bleibt die Ausführung von `close` ohne Wirkung.

Abbildung 5.1 zeigt die möglichen Abfolgen von Aufrufen der drei Methoden eines solchen Iterators.

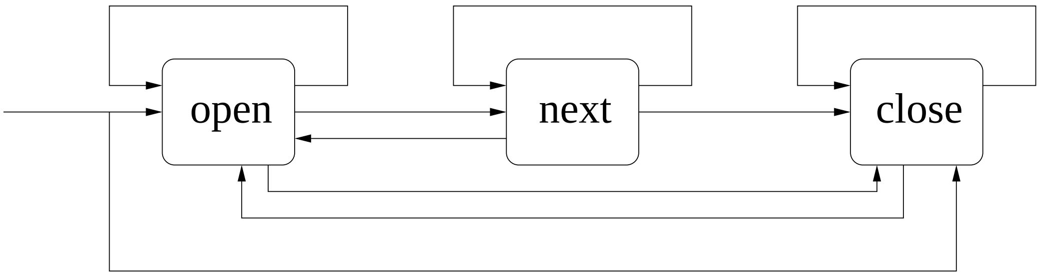


Abb. 5.1: Mögliche Aufrufreihenfolgen der Iterator-Methoden

5.1.5 Sonstige allgemeine Überlegungen

Die implementierte Benutzerschnittstelle des Prototyps und auch die des Datengenerators ist kommandozeilenorientiert und besitzt eine Hilfefunktion, welche die Aufrufsyntax und die möglichen Optionen erklärt. Anhang A gibt einen Überblick über die Bedienung sämtlicher Komponenten des Prototyps. Das Benutzerinterface wurde modular realisiert, so dass bei Beachtung der benutzten Schnittstellen weitere Interfaces, z. B. mit anderen Optionen oder einer graphischen Benutzeroberfläche, einfach implementiert und hinzugefügt werden können. Die verschiedenen Benutzerschnittstellen können dabei nebeneinander existieren, so dass jederzeit auf die gerade gewünschte Alternative zurückgegriffen werden kann.

Im Basis-Paket `bmj` befindet sich die Klasse `ToolBox`. Sie enthält für den gesamten Prototyp allgemein verfügbare Konstanten und statische Methoden. In dieser Klasse befinden sich u. a. die Konstanten mit den Schlüsseln für die Konfigurationsdateien, die anstelle der Kommandozeilen-Optionen zur Konfiguration des Programms benutzt werden können, Versionsinformationen, die Methode für die Rundung von Double-Werten und Implementierungen verschiedener Sortieralgorithmen. Außerdem wird jedem existierenden Attributtyp eine eindeutige Integer-Konstante zugewiesen, so dass effiziente Typprüfungen auf der Basis eines Vergleichs von Integer-Werten möglich sind. Erlaubte Attributtypen sind Integer, Double und String. Im Rahmen dieser Arbeit werden aber nur Attribute vom Typ Double als Join-Attribute herangezogen. Die Klasse `ToolBox` wurde mit dem Modifier `final` versehen, so dass es nicht möglich ist, Unterklassen von ihr zu bilden und ihre Methoden zu überschreiben. Auch alle Attribute der Klasse wurden mit diesem Modifier versehen und stellen somit Konstanten dar.

Die Funktion zur Rundung von Double-Werten rundet jede übergebene Zahl vom Typ Double standardmäßig auf maximal fünf Nachkommastellen und gibt die gerundete Zahl zurück. Sie muss jedesmal angewandt werden, wenn Berechnungen auf Werten vom Typ Double vorgenommen wurden, um das Ergebnis der Berechnung zu runden. Das gilt für alle Arithmetik-Operationen, also neben der Multiplikation und der Division z. B. auch für die Addition und die Subtraktion. Der Grund hierfür liegt darin, dass die Standardarithmetik von Java auf Double-Werten sehr ungenau ist, da viele Fließkommazahlen nicht korrekt dargestellt werden können. So liefert die Operation $0.5265 - 0.5$ nicht etwa das erwartete Ergebnis 0.0265, sondern 0.026499999999999968. Wenn nun aber der so berechnete Wert beispielsweise die neue Untergrenze eines Datenfensters darstellt, dann läge ein Tupel mit einem Attributwert von 0.0265 für das sortierte Attribut noch innerhalb des Fensters anstatt genau auf der Untergrenze. Das würde im Falle des linken Datenfensters bei der nächsten Aktualisierung von Fensterinhalt und -grenzen zu Problemen führen, da hierbei gerade die Tupel aus dem Fenster entfernt werden, die genau auf der Untergrenze liegen. Damit müsste der Algorithmus das Beispieltupel eigentlich aus dem Fenster entfernen, übersieht es jedoch aufgrund des Darstellungsfehlers des Wertes für die Fensteruntergrenze. Natürlich kann es noch eine Reihe ähnlicher Probleme geben, da die Korrektheit der Algorithmen von exakten Double-Werten abhängig ist. Deshalb muss jeder Double-Wert, der das Ergebnis einer arithmetischen Berechnung ist, mittels der Rundungsfunktion aus der Klasse `ToolBox` gerundet werden.

Alternativ zu diesem Rundungsansatz ist auch die Verwendung der Klassen `BigDecimal` für Double-Werte und `BigInteger` für Integer-Werte denkbar. Diese sind in dem Paket `java.math` der Java API enthalten und ermöglichen eine genauere Zahlendarstellung und eine gewisse Kontrolle des Rundungsverhaltens. Sie enthalten außerdem Methoden, mit denen sich verschiedene arithmetische Operationen auf Objekten dieser beiden Klassen durchführen lassen, ohne dass die oben dargestellten Ungenauigkeiten auftreten. Allerdings ist der Einsatz dieser genaueren Zahlendarstellungen und ihrer Methoden mit zusätzlichem Aufwand verbunden, was sich negativ auf die Effizienz des gesamten Programms auswirken würde. Zudem sind die bekannten Infix-Operatoren für arithmetische Operationen hier nicht verwendbar. Stattdessen muss auf die entsprechenden Methoden der Klasse `BigDecimal` bzw. `BigInteger` zurückgegriffen werden. Daher wurde für den Prototyp die einfachere und effizientere Lösung der Rundung von Double-Werten, die das Ergebnis einer arithmetischen Operation sind, auf eine feste Anzahl an Nachkommastellen vorgezogen.

Die implementierten Sortieralgorithmen sind generisch, d. h. ihnen wird beim Aufruf neben der zu sortierenden Menge von Tupeln auch die Ordnung übergeben, nach der die Sortierung vorgenommen werden soll. Die Algorithmen dienen zum Sortieren von Hauptspeichertupeln vor der Materialisierung. Daher ist die übergebene Ordnung immer die Epsilon-Grid-Ordnung, lässt sich bei Bedarf aber auch durch eine andere Ordnung ersetzen. Die übergebene Datenstruktur, welche die zu sortierenden Tupel enthält, ist eine Liste, besitzt also den Typ `List` aus dem API-Paket `java.util`. Das entspricht der Datenstruktur, in der die Hauptspeichertupel gehalten werden. Diese verwaltet der Algorithmus in einer `ArrayList`, also einer Liste mit direktem Indexzugriff auf die einzelnen Listenelemente in konstanter Zeit. Um den Aufwand für eine Konversion der übergebenen Liste von Hauptspeichertupeln in ein Array zu sparen, arbeiten alle Sortieralgorithmen direkt auf der Liste unter Verwendung der im Interface `List` definierten Methoden `get` und `set`. Diese arbeiten auf einer `ArrayList` ebenso wie der normale Indexzugriff auf Arrays in konstanter Zeit. Auf einer `LinkedList` aber laufen sie in linearer Zeit, bezogen auf die Länge der Liste, da bei jedem Zugriff die Datenstruktur vom Beginn oder vom Ende aus bis zu dem zu lesenden oder zu schreibenden Element durchlaufen werden muss. Daher ist aus Effizienzgründen darauf zu achten, dass die den Sortieralgorithmen übergebenen Listen möglichst auf der Basis von Arrays implementiert sind und ihre `get` und `set` Methoden konstante Laufzeiten haben. In der gegenwärtigen Implementierung ist dies durch die ausschließliche Verwendung des Datentyps `ArrayList` der Fall.

Insgesamt besitzt der Prototyp drei verschiedene Sortieralgorithmen. Wichtig ist dabei, dass jeder dieser Algorithmen eine in-place Sortierung der Tupel im Hauptspeicher durchführt. Schließlich steht für den Sortiervorgang kein zusätzlicher Platz im Hauptspeicher zur Verfügung. Neben dieser in-place Anforderung ist ein zweiter wichtiger Punkt die Effizienz der Sortierverfahren. Damit scheidet z. B. der Mergesort-Algorithmus aus, da dieser in der Standardimplementierung nicht in-place sortiert und die existierenden Varianten, die eine in-place Sortierung mittels Mergesort erlauben, quadratische Komplexität in der Anzahl der zu sortierenden Elemente haben. Stattdessen wurde der Quicksort-Algorithmus implementiert, der zwar im schlimmsten Fall ebenfalls quadratische Komplexität besitzt, in der Praxis aber ohnehin in der Regel effizienter ist als das normale Mergesort-Verfahren. Außerdem ist der Quicksort-Algorithmus von sich aus ein in-place Verfahren. Die beiden anderen Algorithmen, die außerdem noch implementiert wurden, sind der Insertionsort- und der Bubblesort-Algorithmus. Sie sollten nur zur Steigerung der Effizienz des Quicksort-Algorithmus dienen, indem dieser ab einer bestimmten geringen Anzahl zu sortierender Elemente auf eines dieser einfachen Verfahren umschaltet. Messungen haben allerdings ergeben, dass die Quicksort-Implementierung am schnellsten ist, wenn eine derartige Umschaltung nicht erfolgt. Sie wurde deshalb wieder deaktiviert. Mittels entsprechender Konstanten im Quellcode lässt sich aber nach wie vor einstellen, ob eine Umschaltung erfolgen soll, ab welcher Anzahl zu sortierender Elemente dann tatsächlich umgeschaltet wird und welcher Algorithmus dabei zum Einsatz kommt.

Ebenso wie der Prototyp mehrere verschiedene Benutzerschnittstellen und Sortieralgorithmen unterstützt, können auch mehrere Auswertungspläne und Vergleichsordnungen implementiert und eingesetzt werden. In der ursprünglichen Version enthält die Implementierung je einen Plan und eine Ordnung. Der vorhandene Standard-Auswertungsplan, der im Folgenden auch stets zugrunde gelegt wird, berechnet den Left-Outer-Bestmatch-

Join mit Einschränkungen auf den beiden Eingabedatenströmen R und S. Es ist aber durchaus denkbar, auch kompliziertere Pläne mit mehreren Joins einzuführen. Dies ist einfach durch die Implementierung einer weiteren Unterklasse der abstrakten Klasse `Plan` in dem Paket `bmj.join` möglich. Die Standard-Vergleichsordnung entspricht der in Kapitel 3.1.2 vorgestellten Ordnung *minAttrDist*. Entsprechend sind auch hier zusätzliche Ordnungen durch Implementierung weiterer Unterklassen der abstrakten Klasse `Order` in `bmj.join` hinzufügbar.

Wie Pläne und Ordnungen werden auch verschiedene Joins als Unterklassen einer abstrakten Basisklasse implementiert. Es handelt sich dabei um die Klasse `Join` im Paket `bmj.join`. Hier sind zwei Varianten bereits realisiert, nämlich der naive Algorithmus aus Kapitel 4.1 in der Klasse `NLJoin` und der fensterbasierte Algorithmus inklusive der Erweiterungen um die Epsilon-Grid-Ordnung und den Scheduling-Algorithmus aus den Kapiteln 4.2 und 4.3 in der Klasse `BMJoin`. Dabei kann der fensterbasierte Algorithmus sowohl mit als auch ohne die Erweiterungen ausgeführt werden. Dies dient später zum Vergleich, inwiefern der Einsatz der Epsilon-Grid-Ordnung und des Scheduling-Algorithmus zur Verbesserung der Leistung des Verfahrens beiträgt. Bei Einsatz der Modifikationen erfolgt die Berechnung wie in Kapitel 4.3 beschrieben. Ohne die Erweiterungen entfällt die Sortierung der Tupel im Hauptspeicher nach EGO bei der Materialisierung. Stattdessen werden die Tupel sequenziell gemäß der Reihenfolge im Speicher materialisiert. Außerdem kommt dann auch der auf der Epsilon-Grid-Ordnung aufbauende Scheduling-Algorithmus zum Laden von Seiten des Hintergrundspeichers nicht mehr zum Einsatz. Vielmehr wird der Reihe nach für jede Seite der linken Eingabe jede benötigte Seite der rechten Eingabe bei Bedarf, und damit mitunter mehrmals, geladen. Weiterhin können für den fensterbasierten Algorithmus beide in Kapitel 4.3.3 eingeführten Materialisierungsvarianten benutzt werden. Auch für den naiven Algorithmus sind diese Varianten wählbar. Allerdings ist bei diesem Verfahren in keinem Fall eine explizite Sortierung nach EGO erforderlich. Der naive Algorithmus materialisiert vor Beginn der Join-Berechnung die komplette rechte Eingaberelation tupelweise auf dem Hintergrundspeicher. Bei der Standard-Materialisierung wird dabei einfach eine Liste von Seiten verwaltet, wobei die Tupel sequenziell in die jeweils letzte Seite der Liste geschrieben werden, bis diese voll ist. Danach legt das Verfahren eine neue Seite an, fügt sie am Ende der Liste ein und schreibt in dieser Seite weiter. Bei der Grid-Materialisierung werden ebenfalls alle Tupel der rechten Eingabe sequenziell materialisiert. Bei dieser Variante muss aber der Semantik der Datenstruktur entsprechend für jedes einzelne Datenelement zunächst die Seite bestimmt werden, in die dieses Datenelement gehört. Danach muss das Verfahren diese Seite laden, das Tupel hineinschreiben und die Seite wieder auf den Hintergrundspeicher zurückschreiben. Die Grid-Materialisierung ist für den naiven Algorithmus zwar von geringer Relevanz, aber dennoch einsetzbar.

Die Eingabedaten des Prototyps stammen aus Textdateien, die im Dateisystem abgelegt sind. Sie genügen einem bestimmten Format, das in Abschnitt A.3 von Anhang A beschrieben ist. Um einen Datenstrom zu simulieren, liest der Prototyp eine solche Eingabedatei mittels einer Instanz der Klasse `FileScan` aus dem Paket `bmj.io` ein. Diese Klasse implementiert das Interface `BMJIterator`. Die Methode `open` öffnet den Datenstrom, mit `next` wird das nächste Tupel des Stroms geliefert und ein Aufruf von `close` schließt den Strom wieder.

5.2 Besonderheiten der Implementierung in Java

Nachdem zuvor allgemeine Aspekte des Entwurfs und der Implementierung des Prototyps beleuchtet wurden, beschäftigt sich dieser Abschnitt nun mit einigen Besonderheiten der verwendeten Programmiersprache Java, die es bei der Entwicklung des Prototyps zu beachten galt. Java ist grundsätzlich eine sehr komfortable Sprache, die dem Programmierer neben viel syntaktischem Zucker auch eine große Klassenbibliothek, die Java API [Sun02a], an die Hand gibt. Leider führen einige der komfortablen Syntaxkonstrukte, wie z. B. der binäre Operator + zur Konkatenation von Strings, zu zum Teil äußerst ineffizientem Programmcode. Das liegt in diesem konkreten Beispiel daran, dass der Compiler jede Anwendung des Konkatenationsoperators in einen Aufruf der Methode `append` eines Objekts der Klasse `StringBuffer` aus dem Paket `java.lang` umwandelt. Daher muss auch für jedes Statement, das mindestens einen solchen Operator enthält, zunächst ein Objekt dieser Klasse instanziiert werden, das unmittelbar nach der Ausführung des betreffenden Statements wieder verworfen und dem Garbage Collector überlassen wird. Da gerade die Instanziierung von Objekten und die Garbage Collection zu den aufwendigen Vorgängen zählen, die relativ viel Rechenzeit in Anspruch nehmen, kann dies bei exzessivem Gebrauch des Konkatenationsoperators durchaus zu spürbaren Leistungseinbußen führen. Stattdessen wäre es besser, einmal ein solches `StringBuffer`-Objekt zu instanziieren und die einzelnen Teilstrings durch explizite Aufrufe von `append` zu konkatenieren. Auch die API enthält viele Klassen und Methoden, die zwar komfortabel anzuwenden, aber oftmals sehr ineffizient sind. Dies hängt allerdings mitunter auch von der konkreten Anwendungssituation ab. Zu dieser Thematik findet sich in [SMFH01] eine Reihe weiterer interessanter Beobachtungen. Da die Effizienz neben der Modularität eines der Hauptziele bei der Implementierung des Prototyps war, waren genauere Betrachtungen der Arbeitsweise der verwendeten API-Teile und Syntaxkonstrukte notwendig, um potenzielle Leistungsbremsen erkennen und eine möglichst effiziente Implementierung realisieren zu können. Die wichtigsten damit zusammenhängenden Punkte, Entscheidungen und Alternativen werden nachfolgend genauer beschrieben.

5.2.1 Wahl der Datenstrukturen

Wie bereits in Abschnitt 5.1.5 im Rahmen der Ausführungen über die Sortieralgorithmen angedeutet wurde, kann die Wahl der Datenstrukturen entscheidenden Einfluss auf die Effizienz des Programms haben. Insbesondere die Art und Weise, wie Tupel im Hauptspeicher abgelegt und wie auf sie zugegriffen wird, ist hier von Bedeutung, da es sich dabei um eine sehr häufige Aktion handelt. Grundsätzlich wäre es dabei wünschenswert, zwei schwer vereinbare Eigenschaften in einer Datenstruktur zu kombinieren.

- Zum einen wäre eine dynamische Datenstruktur vorteilhaft, da die Anzahl der zu speichernden Datenelemente meistens vorab nicht bekannt ist. Eine dynamische Struktur kann dann mit dem Datenaufkommen wachsen und schrumpfen, ohne Speicherplatz zu verschwenden. Außerdem sind bei einer solchen Datenstruktur meist effiziente Methoden zur Aktualisierung der Daten, etwa zum Einfügen und Löschen einzelner Datenelemente, realisierbar.

- Zum anderen hätte man im Allgemeinen gern die Möglichkeit, auf jedes beliebige Element der Datenstruktur in konstanter Zeit zugreifen zu können. Dies setzt eine Art Indexstruktur voraus, so dass jedes Datenelement über einen eindeutigen Index effizient angesprungen werden kann.

Diese beiden Ziele sind in der Java API in dem Paket `java.util` erreicht worden, allerdings in zwei unterschiedlichen Datenstrukturen. Die Klasse `LinkedList` ist eine Implementierung des `List`-Interface auf der Basis einer doppelt verketteten Liste. Sie erfüllt das erste Ziel, das zweite hingegen nicht. Für den Zugriff auf ein beliebiges Element der Datenstruktur muss die Liste beginnend beim ersten oder letzten Element bis zu dem gesuchten Datum durchlaufen werden. Im schlimmsten Fall ist also das Traversieren der halben Liste nötig. Mit der Klasse `ArrayList` verhält es sich genau umgekehrt. Hierbei handelt es sich um eine Implementierung des `List`-Interface auf der Basis eines dynamischen Arrays. Durch das Array ist ein direkter Indexzugriff auf jedes Element der Liste in konstanter Zeit möglich. Dafür muss bei der Initialisierung dieser Datenstruktur eine Initialkapazität angegeben werden, die im Speicher auf jeden Fall belegt wird, auch wenn letztlich weit weniger Datenelemente in der Liste abgelegt werden. Es kommt hier also unter Umständen zu einem höheren Speicherverbrauch. Außerdem muss der komplette Inhalt des Arrays in ein neues, größeres oder kleineres Array umkopiert werden, wenn die Größe der Datenstruktur mit dem Datenaufkommen wachsen oder schrumpfen soll. Hinzu kommt auch, dass das Einfügen und das Löschen eines Datenelements an einer anderen Stelle als am Ende relativ hohe Kosten für das Umkopieren anderer Daten verursacht. Daher muss an jeder Stelle individuell entschieden werden, ob eine `LinkedList` oder eine `ArrayList` die geringeren Kosten verursacht. Dies hängt natürlich davon ab, welche Operationen auf der Datenstruktur an der betreffenden Stelle vor allem ausgeführt werden.

Bei der Implementierung des Prototyps wurde diese Entscheidung entsprechend für alle benötigten Listen getroffen. So sind z. B. die Liste der im Hauptspeicher enthaltenen Tupel eines Datenfensters und die Liste der Hintergrundspeicherseiten bei der Standard-Materialisierung jeweils als `ArrayList` realisiert. Dies ist effizienter, weil die Liste der Hauptspeichertupel gegebenenfalls nach Epsilon-Grid-Ordnung sortiert werden muss und auf die Seiten des Hintergrundspeichers schneller und direkter Indexzugriff nötig ist. Die Liste der Tupelpaare im Puffer O für potenzielle Ergebnispaare, der in der Klasse `OutputBuffer` in dem Paket `bmj.datatypes` implementiert ist, wurde hingegen als `LinkedList` realisiert, da hier kein direkter Zugriff auf die Paare nötig ist. Vielmehr wird über den gesamten Pufferinhalt iteriert. Daneben muss bei dieser Datenstruktur auch effizientes Entfernen von Einträgen, oft vom Anfang der Liste, möglich sein, wenn Tupelpaare als Teil des endgültigen Ergebnisses ausgegeben oder von einer besseren Tupelkombination invalidiert werden.

Eine Alternative wäre die Verwendung der Datenstruktur `HashMap` aus `java.util`. Dies ist aber nicht praktikabel, da diese Klasse keine Iterationsreihenfolge garantieren kann und damit bei einer Iteration jegliche Sortierung der Datenelemente hinfällig ist. Seit der Version 1.4.0 des SUN JDK gibt es jedoch auch die Klasse `LinkedHashMap`, welche auf einer `LinkedList` aufgebaut ist und somit eine Iterationsreihenfolge gemäß der Einfügereihenfolge der Datenelemente in der Datenstruktur einhält. Allerdings haben derartige, aus Paaren von Schlüsseln und Datenelementen bestehende Datenstrukturen noch weitere Nachteile, die sie u. a. für die oben aufgeführten Anwendungsbereiche uninteressant machen. So ist eine Sortierung des Inhalts nach einer bestimmten Ordnung nicht ohne weiteres möglich, auch für eine Iteration muss erst eine sogenannte Collection View erzeugt werden, über die dann iteriert wird, was jedesmal eine teure Objektinstanziierung mit sich bringt und schließlich muss für jedes einzufügende Element ein passender Schlüssel existieren, über den sich das Element später wieder finden lässt. In vielen Fällen ist deshalb ein solcher Ansatz nicht möglich. Die Objekte der Klasse `TupleMetaData` in `bmj.datatypes`, welche die Metadaten eines Tupels repräsentieren, speichern die in ihnen enthaltenen Objekte der Klasse `AttributeMetaData`, also die Metadaten der einzelnen Attribute des jeweiligen Tupels, für schnellen, direkten Zugriff tatsächlich in einer `HashMap`. Als Schlüssel wird der jeweilige Attributname verwendet.

Bei der Entscheidung für vorgefertigte Datentypen aus der API ist auch zu beachten, ob die betreffende Klasse für den gleichzeitigen Zugriff durch mehrere Threads synchronisiert ist oder nicht. Untersuchungen, die in [SMFH01] beschrieben sind, haben gezeigt, dass synchronisierte Datenstrukturen im Vergleich zu ihren nicht synchronisierten Äquivalenten einen deutlichen Mehraufwand und damit merklich erhöhte Ausführungszeiten mit sich bringen. Daher sollte nur dann auf synchronisierte Datenstrukturen zurückgegriffen werden, wenn die Synchronisation auch wirklich benötigt wird. Die Implementierung des Prototyps benutzt deshalb soweit möglich nicht synchronisierte Klassen der API. Beispielsweise werden die nicht synchronisierten Klassen `ArrayList` und `HashMap` den weitgehend funktionsgleichen, aber synchronisierten Implementierungen `Vector` und `Hashtable` vorgezogen.

5.2.2 Sortieralgorithmen

Sortieralgorithmen werden im Prototyp an zwei Stellen benötigt. Einerseits bei der Sortierung von Tupeln im Hauptspeicher nach Epsilon-Grid-Ordnung vor der Materialisierung und andererseits im Datengenerator, wenn eine Relation erzeugt werden soll, deren Tupel nach den Werten eines Attributs aufsteigend sortiert sind. Die Java API bietet in der Klasse `Arrays` des Pakets `java.util` statische Sortiermethoden für Arrays jeden Typs an. Der Datengenerator verwendet eine solche Methode, um ein Array vom Typ Double, das die zufällig generierten Werte der zu sortierenden Dimension enthält, zu sortieren. Dabei kommt wie bei allen Arrays, die einen primitiven Datentyp haben, ein modifizierter Quicksort-Algorithmus zum Einsatz. Für die Sortierung von Tupeln im Hauptspeicher gemäß EGO kann aber keine dieser Methoden benutzt werden. Da hier ein Array von Objekten zu sortieren ist, müssten entweder alle diese Objekte das Interface `Comparable` aus dem Paket `java.lang` implementieren, oder der Sortiermethode müsste eine entsprechende Implementierung des Interface `Comparator` aus `java.util` übergeben werden. Die Methoden zur Sortierung von Arrays, die Objekte enthalten, basieren aber im Gegensatz zu denen für die Sortierung von Arrays mit primitiven Datentypen nicht auf einem veränderten Quicksort-Algorithmus, sondern auf einem leicht modifizierten Mergesort-Algorithmus. Dieser sortiert die Elemente nicht in-place und würde somit die Bedingung der Hauptspeicherbeschränkung verletzen. Daher wurde für diesen Anwendungsfall, wie bereits in Abschnitt 5.1.5 dargestellt, eigens ein Quicksort-Algorithmus implementiert, der eine effiziente in-place Sortierung durchführt.

5.2.3 Speicherverwaltung

Die Speicherverwaltung ist in Java aus der Sicht des Programmierers im Prinzip eine einfache Aufgabe. Die Allokation von Speicher für ein neues Objekt erfolgt bei der Instanziierung des Objektes mittels des Befehls `new`. Die Deallokation wird implizit durch das Löschen aller Referenzen auf ein Objekt eingeleitet und letztendlich vom Garbage Collector vollzogen. Solange man keine besondere Kontrolle über die Speicherverwaltung benötigt, z. B. bei wenig speicherintensiven Programmen, erweist sich diese Verfahrensweise als vorteilhaft, da sie einfach und komfortabel ist. Benötigt ein Programm aber mehr Kontrolle über den Speicher, etwa darüber, welche Speicherbereiche wann deallokiert werden sollen, so ist dies nicht ohne weiteres machbar. Vielmehr erfolgt die Ausführung des Garbage Collectors zu nicht vorhersagbaren Zeiten. Zwar kann der Garbage Collector explizit durch einen Aufruf der Methode `System.gc` gestartet werden, es ist dann aber nicht garantiert, dass er alle nicht mehr referenzierten Objekte bei diesem Lauf auch tatsächlich aus dem Speicher entfernt. Unter Umständen kann gerade der Speicherbereich, den man eigentlich deallokieren wollte, im Speicher verbleiben.

Im Prototyp übernimmt der Garbage Collector neben der Deallokation von Objekten im Hauptspeicher auch das Aufräumen auf dem Hintergrundspeicher. Nicht mehr benötigte Seiten mit materialisierten Tupeln werden von ihm gelöscht. Dazu existiert für jede Seite auf dem Hintergrundspeicher ein Objekt mit Metadaten. Dieses Objekt, eine Instanz einer Unterklasse der abstrakten Klasse `bmj.datatypes.MaterializedPage`, puffert zur Effizienzsteigerung einige für Berechnungen relevante Werte wie den größten und den kleinsten Wert des sortierten Attributs aller Tupel in der Seite und enthält die Größe des aktuellen Seiteninhalts in Byte sowie die Kapazität der Seite in Byte und in Kilobyte. Außerdem besitzt es eine Instanz der Klasse `java.io.File`, die einen Verweis auf die Datei im Dateisystem darstellt, welche die zugehörige Seite repräsentiert und die Daten all ihrer Tupel enthält. Dies ist in Abbildung 5.2 dargestellt. Die Ovale repräsentieren dabei die Referenzobjekte und die grauen Rechtecke die Dateien auf dem Hintergrundspeicher. Das Objekt einer bestimmten Seite ist über eine Referenz, dargestellt als Pfeil, mit der Datei verbunden, welche die materialisierten Tupel dieser Seite enthält. Über das Referenzobjekt kann die zugehörige Seite aus der Datei in den Hauptspeicher eingelesen und auch wieder auf den Hintergrundspeicher geschrieben werden. Die Klasse der Referenzobjekte überschreibt die `finalize`-Methode der Klasse `Object`. Diese Methode führt der Garbage Collector aus, unmittelbar bevor das zugehörige Objekt aus dem Speicher entfernt wird. Der Aufruf von `finalize` schließt alle auf der Datei mit den materialisierten Tupeln der Seite dieses Objekts noch geöffneten Channels und Streams und löscht die Datei vom Hintergrundspeicher.

Ein größeres Problem entstand in einer frühen Phase der Implementierung des Prototyps durch die Tatsache, dass es in Java keine Möglichkeit gibt, die Größe eines Objekts im Hauptspeicher festzustellen. Außerdem ist es auch nicht möglich, bei Verwendung der Objekt-Serialisierung zur Speicherung von Objekten auf dem Hintergrundspeicher

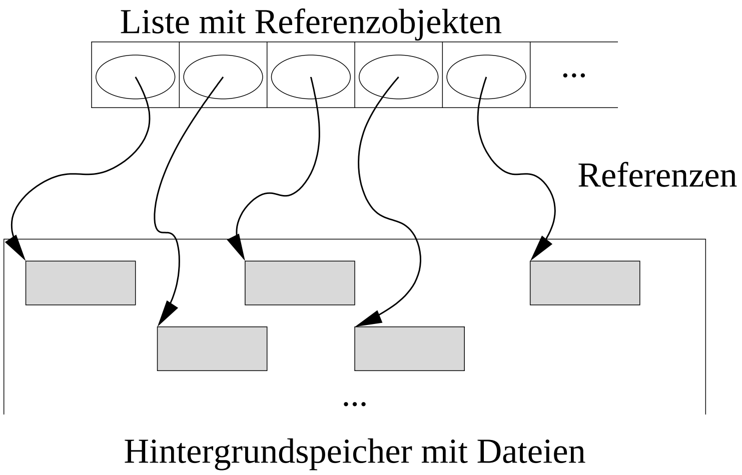


Abb. 5.2: Beziehung zwischen Referenzobjekten und Dateien

schon im Voraus in Erfahrung zu bringen, wie groß das serialisierte Objekt auf dem Hintergrundspeicher letztlich sein wird. Da zu Beginn die Materialisierung von Tupeln noch auf der Basis der Objekt-Serialisierung arbeitete und man z. B. schon vor der Speicherung eines Tupels wissen musste, ob es überhaupt noch in die aktuelle Seite passt, kam es hier zu Schwierigkeiten. Diese konnten endgültig erst beseitigt werden, als die Objekt-Serialisierung aus Effizienzgründen durch eine eigene Implementierung der I/O-Operationen ersetzt wurde. Dazu folgen genauere Ausführungen in Abschnitt 5.2.4.

Insgesamt lässt sich festhalten, dass man sich bei der Programmierung in Java weniger mit der größtenteils automatisierten Speicherverwaltung aufhalten muss, gleichzeitig über diese aber auch weit weniger Kontrolle hat als beispielsweise in C, das keinen integrierten Garbage Collector besitzt und bei dem deutlich mehr Augenmerk auf die Allokation und Deallokation von Speicher gelegt werden muss. Das führt zwar in der Regel zu komplizierterem Code, erlaubt aber auch die für speicherintensive Programme oft wichtige Einflussnahme auf das Speichermanagement. Außerdem bietet C noch weitere Möglichkeiten, die Java nicht besitzt, wie z. B. das `sizeof`-Konstrukt, mit dem sich die Menge des benutzten physikalischen Speichers bestimmen lässt. In [SMFH01] finden sich weitere Ausführungen zum Speichermanagement in Java und zum Java Garbage Collector.

5.2.4 Implementierung der I/O-Operationen

Größte Aufmerksamkeit kam während der Implementierung des Prototyps der Realisierung der I/O-Operationen zu, insbesondere dem Schreiben und Lesen von Tupeln im Rahmen der Materialisierung. Da die Zugriffe auf den Hintergrundspeicher mit Abstand die teuersten Operationen in der gesamten Implementierung sind, war hier durch effiziente Programmierung der größte Leistungsgewinn zu erreichen. Andererseits bestand die Gefahr, bei Zurückgreifen auf ineffiziente Techniken massive Leistungseinbußen hinnehmen zu müssen. Daher wurden drei Alternativen untersucht und verglichen, die im Folgenden genauer beschrieben werden. Des Weiteren enthält dieser Abschnitt Ausführungen über die Implementierung der Standard- und der Grid-Materialisierungsvariante.

Implementierungsansätze

Nachfolgend werden die drei verschiedenen in Erwägung gezogenen Implementierungsansätze im Einzelnen vorgestellt. Im Anschluss daran folgt die Beschreibung der Untersuchung und der Vergleich der drei Varianten sowie die Begründung der Entscheidung für den schließlich verwendeten Ansatz.

Objekt-Serialisierung: Java bietet mittels des Konzepts der Objekt-Serialisierung die Möglichkeit, jedes serialisierbare Objekt eines Java-Programms auf den Hintergrundspeicher zu schreiben und wieder einzulesen. Da die Tupel im Prototyp Objekte der Klasse `bmj.datatypes.Tuple` sind, ist dieser Ansatz grundsätzlich auch für die Materialisierung von Tupeln im Rahmen der Prototyp-Implementierung einsetzbar. Dazu müssen nur die Klasse `bmj.datatypes.Tuple` und die Klassen aller Objekte, die direkt oder indirekt in `Tuple` vorkommen, das Interface `Serializable` aus dem Paket `java.io` der Java API implementieren. Die Klasse `Tuple` enthält beispielsweise ein Objekt der Klasse `TupleMetaData` und diese wiederum Objekte der Klasse `AttributeMetaData`. Somit müssen diese beiden Klassen ebenfalls das Interface `Serializable` implementieren. Dieses Interface enthält keinerlei Signaturen von Methoden. Die implementierenden Klassen kennzeichnen durch die Implementierung des Interfaces nur, dass sie serialisierbar sind. Nun ist es möglich, ein `Tuple`-Objekt mit einem `ObjectOutputStream` mittels der Methode `writeObject` auf den Hintergrundspeicher zu schreiben. Entsprechend kann dieses Objekt durch die Methode `readObject` mit einem `ObjectInputStream` später wieder eingelesen werden. Diese Streams arbeiten auf einem darunterliegenden `FileOutputStream` bzw. `FileInputStream` um die Tupel in eine Datei zu schreiben oder aus einer Datei zu lesen. Zur Effizienzsteigerung kann noch ein `BufferedOutputStream` bzw. `BufferedInputStream` dazwischengesetzt werden.

Java NIO: Seit der Version 1.4.0 des Sun JDK gibt es für die Implementierung von I/O-Operationen das neue API-Paket `java.nio`. Es ergänzt das alte Paket `java.io` um neue und zusätzliche Funktionen. Dabei bietet es nicht nur erweiterte Kontrolle und Funktionalität, sondern ist auch insgesamt effizienter als die alten Klassen. Näheres zu diesem neuen Paket findet sich z. B. in [Sun02c] und [Sun02b]. Die Implementierung der Materialisierung von Tupeln im Prototyp auf der Basis von `java.nio` sieht wie folgt aus. Zunächst wird ein ByteBuffer allokiert, der die zu schreibenden bzw. die gelesenen Daten im Byte-Format aufnimmt. Dabei handelt es sich um eine Instanz der Klasse `java.nio.ByteBuffer`. Bei der Instanziierung sind zwei Alternativen zu unterscheiden. Ein ByteBuffer kann sowohl als direkter als auch als indirekter ByteBuffer instanziiert werden. Ein direkter ByteBuffer hat den Vorteil, dass die Java Virtual Machine versucht, die betriebssystemspezifischen I/O-Operationen direkt auf dem Puffer anzuwenden, ohne den Pufferinhalt vor oder nach dem Aufruf einer solchen Operation in einen Zwischenpuffer zu kopieren. Deshalb können I/O-Operationen auf einem direkten Puffer im Allgemeinen schneller ausgeführt werden als auf einem indirekten. Der Nachteil ist, das die Allokations- und Deallokationskosten für einen direkten ByteBuffer in der Regel um einiges höher sind als für einen indirekten ByteBuffer. Bei Versuchen auf einem Sun Enterprise 450 Server mit vier 400 Mhz Prozessoren, von denen aber nur einer verwendet wurde, und 4 GB RAM unter Sun Solaris 7 betrug die Zeit für die Allokation eines acht Kilobyte großen ByteBuffers etwa das Drei- bis Vierfache der für einen gleich großen, indirekten ByteBuffer benötigten Zeit, bewegte sich aber dennoch nur im Bereich zwischen vier und zwölf Millisekunden. Die Java API empfiehlt, direkte ByteBuffer nur für große, langlebige Puffer zu verwenden, auf denen betriebssystemspezifische I/O-Operationen auszuführen sind. Untersuchungen am Prototyp haben ergeben, dass dieser insgesamt bei Verwendung von direkten Puffern immer zumindest geringfügig schneller ist als bei Einsatz indirekter Puffer. Bei keinem Testfall erreichte die Variante mit indirekten ByteBuffern die Geschwindigkeit der Version mit direkten ByteBuffern. Deshalb verwendet der Prototyp nun ausschließlich direkte ByteBuffer. Jedes Referenzobjekt einer Seite auf dem Hintergrundspeicher besitzt so einen direkten Puffer, der genau die Größe der entsprechenden Seite in Bytes hat. Dieser Puffer wird aus Effizienzgründen bei der Initialisierung auch auf die Byte-Ordnung der zugrunde liegenden Systemarchitektur eingestellt. So haben beispielsweise die Prozessoren der Ultra-Baureihe von Sun die Byte-Ordnung Big-Endian[1], die Prozessoren von Intel hingegen Little-Endian[2]. Durch dieselbe Byte-Ordnung von System und ByteBuffern können plattformunabhängig teure Umwandlungsoperationen beim Schreiben und Lesen von Daten mittels betriebssystemeigener I/O-Operationen entfallen. Die statische Methode `nativeOrder` der Klasse `java.nio.ByteOrder` ermöglicht die Bestimmung der Byte-Ordnung des Systems, die anschließend mit der Methode `order` eines ByteBuffers für den jeweiligen Puffer eingestellt werden kann. Soll nun ein Tupel materialisiert werden, so können seine Attributwerte mittels entsprechender `put`-Methoden des Puffers in Form von Bytes in den ByteBuffer und von dort mit einem FileChannel in eine Datei auf dem Hintergrundspeicher geschrieben werden. Für ein String-Attribut wird dabei zunächst ein Integer-Wert geschrieben, der die Länge des Strings in Zeichen angibt, und anschließend der String selbst. Die Werte von Double- und Integer-Attributen werden hingegen direkt geschrieben. Entsprechend lässt sich der Inhalt einer solchen Datei auch wieder mit einem FileChannel in den ByteBuffer einlesen. Bei einem String-Wert gibt dabei der vorausgehende Integer-Wert an, wieviele Zeichen und damit Bytes gelesen werden müssen, damit der gesamte String vorliegt. Ein Zeichen benötigt in dem verwendeten Zeichensatz ISO-8859-1 genau ein Byte. Ein Double-Wert belegt acht Byte, ein Integer-Wert vier Byte. Aus dem Inhalt des Puffers können dann durch entsprechende `get`-Methoden die gespeicherten Attributwerte wiederhergestellt werden. Dieses Verfahren materialisiert nicht die Metadaten der Tupel, sondern nur die Attributwerte. Bezogen auf ein Datenfenster, das ja nur Tupel der gleichen Relation enthält, merkt sich das Verfahren die Metadaten des ersten geschriebenen Tupels. Diese gelten dann zugleich auch für alle anderen zu schreibenden Tupel. Beim Einlesen erhält schließlich jedes gelesene Tupel seine Metadaten wieder zurück. Eine genauere Beschreibung von ByteBuffern und allen anderen Klassen und Funktionen des Pakets `java.nio` ist in der Java API Spezifikation [Sun02a] enthalten.

Java IO: Statt des neuen Pakets `java.nio` kann man sich für die Implementierung der Materialisierung von Tupeln im Prototyp auch auf die altbekannten Klassen in `java.io`

[1] Das höchstwertige Byte kommt zuerst.
[2] Das niedrigstwertige Byte kommt zuerst.

beschränken. Anstelle von ByteBuffern und FileChannels greift man dann auf herkömmliche Stream-Klassen zurück. Ansonsten ist die Vorgehensweise bei diesem Verfahren aber mit der vorherigen Variante identisch. Zum Schreiben von Tupeln wird nunmehr ein `DataOutputStream` benutzt, mit dessen Hilfe die primitiven Datentypen Double und Integer durch die speziellen Methoden `writeDouble` und `writeInteger` direkt als Bytes geschrieben werden können. Strings lassen sich mit der Methode `writeBytes` als Folge von Bytes schreiben. Auch hier geht jedem String ein Integer-Wert voraus, der die Anzahl an Zeichen des Strings angibt, so dass dieser später wieder korrekt eingelesen werden kann. Der `DataOutputStream` schreibt seine Daten in einen darunter liegenden `ByteArrayOutputStream`. Dessen gesamter Inhalt kann dann geschlossen durch den Aufruf der Methode `writeTo` des Streams auf einen `FileOutputStream` und damit auf den Hintergrundspeicher geschrieben werden. Entsprechend ist das Einlesen der materialisierten Tupel über einen `DataInputStream` und einen darunter liegenden `FileInputStream` möglich. Die Methoden `readDouble` und `readInt` erlauben das direkte Einlesen von Double- und Integer-Werten. Bei einem String wird zunächst die Längenangabe gelesen und auf deren Basis ein Byte-Array von dieser Länge angelegt. Dann kann dieses Array mittels einer speziellen `read`-Methode des Streams mit gelesenen Bytes aufgefüllt werden, so dass sein Inhalt letztlich der Folge von Bytes entspricht, die den geschriebenen String repräsentiert. Ein Konstruktor aus der Klasse `java.lang.String` kann aus dem Byte-Array dann den entsprechenden String rekonstruieren.

Vergleich der Ansätze

Bei allen drei obigen Varianten erfolgt die Speicherung der materialisierten Tupel in temporären Dateien im Dateisystem, die im Standardverzeichnis für temporäre Dateien des zugrunde liegenden Betriebssystems abgelegt werden. Einzelne Dateien werden wieder entfernt, wenn der Garbage Collector das zugehörige Referenzobjekt verwirft. Außerdem löscht die Java Virtual Machine bei ihrer Terminierung automatisch alle dann noch verbliebenen derartigen Dateien aus dem Dateisystem, da im Prototyp auf jedem `File`-Objekt unmittelbar nach seiner Instanziierung die Methode `deleteOnExit` aufgerufen wird.

Die Varianten Java NIO und Java IO speichern Tupel als Folgen von Bytes. Entsprechend werden auch solche Bytefolgen später wieder vom Hintergrundspeicher eingelesen. Innerhalb des Programms liegen die Tupel und deren Attribute aber als Objekte vor. Um die materialisierten Daten im Programm nutzen zu können, muss daher sowohl beim Schreiben als auch beim Einlesen eine Konversion der Objekte in Bytes bzw. umgekehrt erfolgen. Der Prototyp nimmt diese Konversion jeweils unmittelbar vor dem Schreiben bzw. unmittelbar nach dem Einlesen der Daten eines Tupels vor. Das hat den Vorteil, dass die Tupel innerhalb des Prototyps als Objekte vorliegen und jeder Zugriff auf die Werte der einzelnen Attribute eines Tupels ohne zusätzliche Konversionsoperation und damit relativ schnell und effizient erfolgen kann. Dafür sind die genannten Konversionen bei der Materialisierung und beim Einlesen erforderlich. Es ist auch denkbar, die Attributwerte eines Tupels nicht als Objekte in einem Objekt-Array, sondern direkt als Byte-Folgen in einem Byte-Array zu speichern. In diesem Fall entfallen die Konversionen beim Schreiben und Lesen eines Tupels. Das Byte-Array, das die Attributwerte des Tupels enthält, kann ohne weitere Veränderung direkt geschrieben bzw. gelesen werden. Dafür muss nun bei jedem

Zugriff auf ein Attribut der entsprechende Attributwert aus der gespeicherten Byte-Folge rekonstruiert werden. Es ist also bei jedem Zugriff auf das Tupel eine Konversionsoperation auszuführen, die oft mit einer teuren Objektinstanziierung verbunden ist. Zwar spart man sich auf diese Weise die Konversion von Attributwerten, auf die nie zugegriffen wird, z. B. weil das entsprechende Attribut kein Join-Attribut und deshalb für die Join-Berechnung nicht weiter interessant ist, aber durch die relativ hohe Anzahl an Zugriffen auf die Werte verschiedener Attribute im Rahmen der Join-Berechnung steigt die Zahl der auszuführenden Konversionen stark an. Insgesamt ist dieser zweite Ansatz daher in der Regel deutlich ineffizienter als der erstgenannte und wird deshalb nicht verwendet.

Ein großer Vorteil der Implementierung der Materialisierungsoperationen mittels Java NIO und Java IO gegenüber der Objekt-Serialisierung liegt in der genauen Kontrollierbarkeit des Speicherverbrauchs. Während sich bei der Objekt-Serialisierung vorab keinerlei Aussagen darüber treffen lassen, wie viel Platz das serialisierte Objekt tatsächlich auf dem Hintergrundspeicher belegen wird, ist dies bei den beiden anderen Varianten durchaus möglich. Mit dem Wissen, dass ein Integer-Wert vier Byte, ein Double-Wert acht Byte und ein String ein Byte pro Zeichen und zusätzlich vier Byte für die Längenangabe benötigt, ist hier der belegte Speicherplatz schon im voraus exakt bestimmbar. Dies ist z. B. wichtig, um festzustellen, ob ein zu materialisierendes Tupel noch in die aktuelle Seite passt oder ob eine neue Seite angelegt werden muss. Bei der Objekt-Serialisierung führen Header-Daten und eine undurchsichtige Verfahrensweise bei der Speicherung der Objekte zu unvorhersagbaren Dateigrößen. Insgesamt braucht die Objekt-Serialisierung außerdem für die Materialisierung des gleichen Tupelbestandes im Allgemeinen deutlich mehr Platz auf dem Hintergrundspeicher als die anderen beiden Verfahren.

Letztendlich entscheidend ist aber die Effizienz der Methoden. Um sie vergleichen zu können, wurde ein I/O-Testprogramm implementiert, das die Geschwindigkeit aller drei Varianten misst. Es ist in Abschnitt A.5 von Anhang A beschrieben. Tabelle 5.1 zeigt die Laufzeiten der Varianten in Sekunden bei einem Testlauf mit einer zu materialisierenden Eingaberelation bestehend aus 5000 Tupeln auf einem Sun Enterprise 450 Server mit vier 400 Mhz Sun UltraSPARC-II Prozessoren und 4 GB RAM unter Sun Solaris 7. Erwartungsgemäß schneidet die Objekt-Serialisierung dabei relativ schlecht ab. Die mangelnde Effizienz dieses Ansatzes resultiert vermutlich hauptsächlich aus dem generischen Ansatz. Da mit der Objekt-Serialisierung jegliches Objekt, dessen Klasse das Interface `Serializable` implementiert, auf den Hintergrundspeicher geschrieben werden kann, muss beim Einlesen eines serialisierten Objektes sehr viel Aufwand für die Wiederherstellung dieses Objektes betrieben werden. Dabei greift das Verfahren auf die Reflection API von Java zurück, die aber nicht besonders effizient ist. Die Implementierung auf der Basis des neuen Pakets `java.nio` ist der Objekt-Serialisierung in allen Belangen, sowohl beim Schreiben als auch beim Lesen, deutlich überlegen. Das ist darauf zurückzuführen, dass dieses Verfahren nicht generisch, sondern speziell auf die Materialisierung von Tupeln des Prototyps ausgerichtet ist. Damit lässt sich jeglicher unnötige Zusatzaufwand vermeiden. Gleiches gilt für die Implementierung auf der Basis des herkömmlichen Pakets `java.io`. Hier fällt auf, dass diese Variante beim Schreiben sogar noch geringfügig besser abschneidet als Java NIO, dafür aber beim Lesen deutlich abfällt und in diesem Bereich sogar noch schlechtere Werte als die Objekt-Serialisierung erzielt. Woher diese

Laufzeiten [s]	Objekt-Serialisierung	Java NIO	Java IO
Schreiben	8,11	1,23	0,88
Lesen	6,87	0,56	7,33

Tab. 5.1: Laufzeiten verschiedener I/O-Implementierungen

deutliche Diskrepanz zwischen Schreib- und Lesegeschwindigkeit bei Java IO kommt, ist allerdings nicht nachvollziehbar. In der Gesamtheit aus Schreib- und Lesegeschwindigkeit zeigt die Variante Java NIO die beste Leistung. Sie wurde daher für die Implementierung der Materialisierungsoperationen des Prototyps herangezogen.

Zur Objekt-Serialisierung bleibt noch anzumerken, dass ihre Leistung mitunter stark von der Art der zu serialisierenden Daten abhängen kann. Materialisiert man mit dieser Variante beispielsweise eine Relation bestehend aus 5000 identischen Tupeln, so läuft die Materialisierung deutlich schneller ab und verbraucht deutlich weniger Platz auf dem Hintergrundspeicher als bei einer Relation mit 5000 unterschiedlichen Tupeln, wie sie bei obigem Test benutzt wurde. Bei der Materialisierung der 5000 identischen Tupel war die Objekt-Serialisierung im Test, sowohl was Laufzeit als auch was Speicherplatzverbrauch betrifft, besser als Java NIO und Java IO. Da das Szenario einer Relation mit ausschließlich identischen Tupeln aber schon allein aufgrund eines im Allgemeinen vorhandenen Primärschlüssels in der Praxis eher unwahrscheinlich ist, hat diese Beobachtung keine praktische Bedeutung für die Implementierung des Prototyps.

Standard-Materialisierungsvariante

Die Implementierung der Standard-Materialisierungsvariante aus Abschnitt 4.3.3 ist relativ einfach. Sie besteht im Wesentlichen aus einer Liste, die wegen des effizienten Zugriffs auf einen beliebigen Eintrag als `ArrayList` realisiert ist. Diese Liste enthält alle Referenzobjekte von Seiten auf dem Hintergrundspeicher. Wenn der Inhalt einer Seite nicht mehr benötigt wird, so entfernt diese Variante das betreffende Referenzobjekt aus der Liste und löscht damit die einzige Referenz auf dieses Objekt. Es wird dann bei nächster Gelegenheit vom Garbage Collector verworfen, der dabei auch die zugehörige Datei vom Hintergrundspeicher löscht. Kommt eine neue Seite hinzu, so wird ein neues Referenzobjekt für die Seite erzeugt und an das Ende der Liste angehängt. Alle Materialisierungsvarianten haben in der abstrakten Basisklasse `bmj.io.PageManager` ein gemeinsames Interface, das u. a. von der Implementierung des Scheduling-Algorithmus beim Laden von Seiten des Hintergrundspeichers verwendet wird. Für diesen Algorithmus ist es auch wichtig, dass jede beliebige Seite und damit auch jedes Referenzobjekt effizient auffindbar ist, weshalb hier eine `ArrayList` einer `LinkedList` vorgezogen wurde.

Grid-Materialisierungsvariante

Wie die Standard-Materialisierung baut auch die Grid-Materialisierungsvariante aus Abschnitt 4.3.3 auf dem Interface von `bmj.io.PageManager` auf. Sie verwendet als Datenstruktur für die Verwaltung der Referenzobjekte statt einer Liste eine d-dimensionale

Matrix, wobei d die Anzahl der Join-Dimensionen ist. Die Matrix ist als d-dimensionales Array implementiert, das mittels der Methode `newInstance` aus der in der Reflection API enthaltenen Klasse `java.lang.reflect.Array` instanziiert wird, da die Anzahl der Dimensionen erst zur Laufzeit bekannt ist. Die Anzahl e_i der Einträge des Arrays in der Dimension i berechnet sich nach folgender Formel:

$$e_i = \left\lceil \frac{1}{\epsilon_i} \right\rceil$$

Die Zahl 1 im Zähler des Bruches ergibt sich aus der Normierung der Werte der Join-Attribute auf das Intervall $[0; 1]$. Jeder Eintrag des Arrays enthält eine Liste, die hier zugunsten geringeren Speicherbedarfs und besserer Leistung beim Einfügen und Löschen von Elementen als `LinkedList` realisiert ist. Jede Arrayzelle repräsentiert eine Gitterzelle der Epsilon-Grid-Ordnung. Die Liste in einer Zelle des Arrays enthält alle Referenzobjekte von Seiten auf dem Hintergrundspeicher, deren Tupel in die entsprechende Gitterzelle gehören. Bei zu vielen Dimensionen oder zu kleinen Werten für die ϵ_i kann es passieren, dass das Array zu viele Einträge hat und nicht mehr in den Speicher passt. Dann kommt es zu einem `OutOfMemoryError`.

Während es bei der Standard-Materialisierungsvariante mittels eines Indexzugriffs möglich ist, effizient auf das Referenzobjekt einer beliebigen Seite zuzugreifen, ist dies bei der Grid-Materialisierungsvariante nicht so einfach. Hier muss im Prinzip über die enthaltenen Seiten iteriert werden, um z. B. die zehnte Seite in der Datenstruktur zu finden. In der Implementierung des Prototyps wird versucht, diesen Vorgang dadurch ein wenig zu optimieren, dass die Iteration in Abhängigkeit von der gesuchten Seite entweder am Anfang oder am Ende der Datenstruktur startet, je nachdem von welchem Startpunkt aus mit einem schnelleren Auffinden der Seite gerechnet werden kann. Jedoch wird das Finden einer beliebigen Seite, wie es im Scheduling-Algorithmus benötigt wird, bei einer großen Anzahl an Arrayzellen immer noch sehr ineffizient. Weitere Überlegungen zu möglichen Optimierungsansätzen, die jedoch im Rahmen dieser Arbeit nicht mehr realisiert wurden, umfassen die folgenden Punkte:

- Zunächst ist denkbar, den Zugriff auf eine beliebige Seite im Scheduling-Algorithmus durch Puffern von Referenzobjekten zu umgehen. Im Crabstep-Modus muss in Zeile 28 von Algorithmus 4.5 auf Seite 61 zu einer früher bereits geladenen Seite der Relation R zurückgesprungen werden. Dies ist die zentrale Stelle für das Erfordernis, beliebige Seiten und damit auch Referenzobjekte laden bzw. auffinden zu können. Falls der Algorithmus in Zeile 21 nicht nur den Index der Seite, sondern auch deren Referenzobjekt zwischenspeichert, kann später in Zeile 28 mittels dieses Objektes die zugehörige Seite sofort geladen werden, ohne dass das Referenzobjekt erneut in der Datenstruktur gesucht werden muss. Natürlich müssen dann auch alle weiteren Referenzobjekte von Seiten der Relation R, die der Crabstep-Modus bei seiner Ausführung lädt, gepuffert werden. Dann kann die Datenstruktur für das erstmalige Laden einer jeden Seite linear durchlaufen werden und jedes Referenzobjekt, dessen zugehörige Seite später noch einmal geladen werden muss, wird außerhalb der Matrix gepuffert und ist dadurch schnell zugreifbar. Diese Optimierung ist mit relativ wenig

Aufwand realisierbar, verursacht wenig Zusatzaufwand bei der Ausführung der Algorithmen und verspricht eine merkliche Beschleunigung des Scheduling-Verfahrens gegenüber der normalen Grid-Materialisierungsvariante.

- Unabhängig vom Scheduling-Algorithmus kann auch die Datenstruktur des mehrdimensionalen Arrays selbst optimiert werden. Beispielsweise ist denkbar, eine beliebige Seite mittels binärer Suche zu suchen. Dazu müssen aber zusätzliche Metadaten über das Array und seinen Inhalt gespeichert und verwaltet werden. Für jede Zelle einer jeden Dimension des Arrays speichert man die akkumulierte Anzahl der in dieser und allen lexikographisch davor liegenden Zellen enthaltenen Seiten. Dann kann bei der Suche nach der n-ten Seite für jede Dimension eine binäre Suche durchgeführt werden, die schließlich in der letzten Dimension bei der gesuchten Arrayzelle endet, deren Liste das benötigte Referenzobjekt enthält. Welches Objekt aus der Liste gesucht wird, kann dann einfach aus der Seitennummer und den Metadaten berechnet werden. Der Nachteil dieses Verfahrens ist der hohe Aufwand zur Aktualisierung der Metadaten beim Löschen und Hinzufügen von Seiten. Da akkumulierte Daten gespeichert werden, muss für jede gelöschte und hinzugefügte Seite der Seitenzähler aller vor der Einfügestelle liegenden Arrayzellen dekrementiert bzw. inkrementiert werden.
- Um diesen Aktualisierungsaufwand zu verringern, lässt sich statt der binären Suche eine lineare Suche verwenden. Hier enthalten die Metadaten einer Arrayzelle nicht die akkumulierte Seitenanzahl für alle Zellen, sondern nur für die Zellen der aktuellen Dimension. Bei der Suche nach der n-ten Seite wird jede Dimension linear bei der ersten Zelle beginnend durchlaufen und der Wert des Seitenzählers der dabei angetroffenen Zellen so lange aufaddiert, bis die Summe größer oder gleich der Nummer der gesuchten Seite ist. Dann wird in die Dimension der Zelle, deren Wert als letzter addiert wurde, abgestiegen und entsprechend in der nächsten Dimension fortgefahren. In der letzten Dimension gelangt man so letztendlich zu der Arrayzelle, in deren Liste sich das gesuchte Referenzobjekt befindet. Das richtige Objekt der Liste kann wiederum aus der Seitennummer und den Metadaten berechnet werden. Bei diesem Verfahren ist der Aufwand für die Aktualisierung der Seitenzähler beim Löschen und Einfügen von Seiten weitaus geringer als bei dem vorhergehenden. Es müssen immer genau so viele Zähler dekrementiert bzw. inkrementiert werden, wie das Array Dimensionen hat. Dafür ist der eigentliche Suchvorgang wegen der linearen Suche im Allgemeinen etwas langsamer.

5.3 Join-Algorithmen

Dieser Abschnitt gibt einen kurzen Überblick über die Architektur und die Implementierung des naiven und des fensterbasierten Join-Algorithmus. Die Algorithmen implementieren dabei grundsätzlich das Interface `BMJIterator` aus dem Paket `bmj`. Der Aufruf von `open` initialisiert den entsprechenden Join und gibt die Metadaten der Tupel des Join-Ergebnisses zurück, `next` liefert das nächste Ergebnistupel der Join-Berechnung und `close` beendet die Berechnung und führt eventuell notwendige Aufräumarbeiten durch.

5.3.1 Implementierung des naiven Algorithmus

Die Implementierung des naiven Algorithmus ist relativ einfach und liegt nah an dem in Algorithmus 4.2 auf Seite 35 gezeigten Pseudocode. Die `open`-Methode des naiven Algorithmus materialisiert zunächst die komplette rechte Eingaberelation. Dazu liest sie die Eingabe tupelweise ein und schreibt sie mit Hilfe der oben dargestellten Datenstrukturen für die Standard- oder die Grid-Materialisierungsvariante auf den Hintergrundspeicher. Wie bereits erwähnt, ist die Grid-Materialisierungsvariante für den naiven Algorithmus eigentlich nicht relevant, da hier keine Sortierung nach Epsilon-Grid-Ordnung benötigt wird. Aus Gründen der Vollständigkeit und zum Vergleich der Effizienz der Datenstrukturen ist sie aber dennoch verwendbar.

Die `next`-Methode enthält den eigentlichen Algorithmus. Dieser läuft in der äußeren Schleife mittels Aufrufen von `next` auf dem entsprechenden `FileScan`-Objekt über die linke Eingabe und in den beiden inneren Schleifen mittels zweier Iteratoren über den materialisierten Datenbestand der rechten Eingabe. Die Iteratoren laden bei Bedarf die nächste Seite der materialisierten Relation vom Hintergrundspeicher in den Hauptspeicher und iterieren über die darin enthaltenen Tupel. Wenn alle Tupel einer Seite abgearbeitet wurden, laden die Iteratoren beim nächsten Aufruf von `next` die nächste Seite und arbeiten auf dieser weiter. Standardmäßig lädt die `open`-Methode der Iteratoren über die materialisierten Daten die erste Seite, allerdings nur, wenn sie nicht schon bei einem früheren Aufruf der Methode geladen wurde und sich immer noch im Hauptspeicher befindet. Dies kann z. B. der Fall sein, wenn sich die gesamten Daten einer Relation in einer einzigen Seite befinden oder wenn die innere Schleife des Join-Algorithmus vorzeitig abgebrochen wurde und wieder von vorne beginnt, bevor die zweite Seite geladen werden musste. Dieses optimierte Verhalten zur Verringerung der Anzahl an Seitenzugriffen lässt sich im Quellcode durch das Ändern einer Konstante auch abschalten. Dann lädt die `open`-Methode bei ihrer Ausführung stets die erste Seite, egal ob sich diese schon im Speicher befindet oder nicht. Die innere Schleife des Join-Algorithmus bricht ab, sobald das aktuelle Tupelpaar von einem Vergleichspaar dominiert wird und deshalb nicht mehr Teil des Ergebnisses werden kann.

Die `close` Methode des naiven Algorithmus schließt alle Iteratoren und beendet damit die Join-Berechnung.

5.3.2 Implementierung des fensterbasierten Algorithmus

Für die Implementierung des fensterbasierten Algorithmus muss im Vergleich zum naiven Verfahren deutlich mehr Aufwand betrieben werden. Abbildung 5.3 zeigt die allgemeine Architektur der Join-Implementierungen. Die Klasse `bmj.Main` stellt das kommandozeilenbasierte Benutzerinterface dar, das mit den Eingabeparametern für den jeweils aktuellen Lauf versorgt wird. Sie instanziiert den gewählten Auswertungsplan, eine Unterklasse der abstrakten Basisklasse `bmj.join.Plan`, und führt diesen aus. Der Plan wiederum instanziiert den gewählten Join, seinerseits eine Unterklasse der abstrakten Basisklasse `bmj.join.Join`, und stößt dessen Berechnung an. Dafür holt der Join die nötigen Daten aus den `FileScan`-Objekten der beiden Eingaberelationen, welche die Eingabetu-

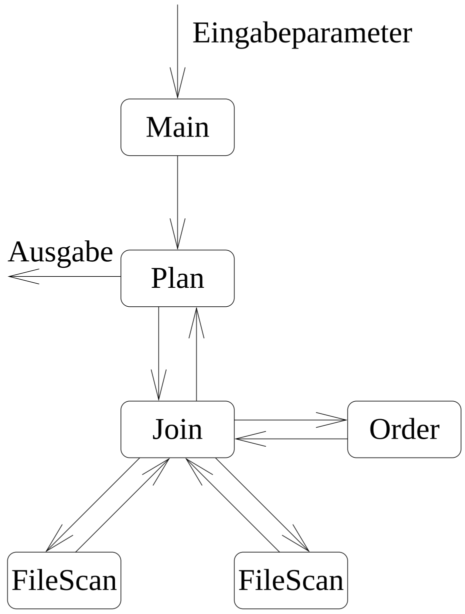


Abb. 5.3: Architektur der Join-Implementierungen

pel mittels des Iterator-Interface aus den Dateien auf dem Hintergrundspeicher einlesen und an den Join liefern. Der Vergleich zweier Tupelpaare während der Berechnung des Join-Ergebnisses erfolgt anhand einer Vergleichsordnung, die wiederum eine Instanz einer Unterklasse der abstrakten Basisklasse `bmj.join.Order` ist. Die Ergebnistupel gelangen schließlich über das Iterator-Interface des Joins zurück zum Plan, der sie tupelweise ausgibt oder, je nach Art des Plans, auch für weitere Berechnungen verwenden kann.

Damit die Join-Berechnung nicht jedesmal neu angestoßen werden muss, wenn der Plan durch einen Aufruf der `next`-Methode des Joins das nächste Ergebnistupel anfordert, sondern der Join seine Ergebnistupel größtenteils unabhängig von den Anforderungen des Plans berechnen kann, wird ein Multithreading-Ansatz verfolgt. Die eigentliche Berechnung des Left-Outer-Bestmatch-Joins mit Einschränkungen durch den fensterbasierten Algorithmus wird in einem eigenen Thread gestartet, der kontinuierlich Ergebnistupel produziert und in einem Ergebnispuffer ablegt. Aus diesem Puffer bezieht der Plan bei jedem Aufruf von `next` auf dem Join das nächste Ergebnistupel. Nur wenn der Puffer voll ist, muss die Join-Berechnung vorübergehend angehalten werden, d h. der Thread blockiert, bis der Plan das nächste Tupel aus dem Ergebnispuffer entnommen und dadurch wieder Platz freigemacht hat. Entsprechend muss bei leerem Puffer auch der Aufruf der `next`-Methode des Joins blockieren, bis der Join das nächste Ergebnistupel in den Pufferspeicher geschrieben hat. Abbildung 5.4 zeigt eine schematische Übersicht über die Implementierungsarchitektur des fensterbasierten Verfahrens. Die Klasse `bmj.join.BMJoin` startet zu Beginn der Berechnung einen eigenen Thread. Diese Instanz der Klasse `ProducerThread`, einer inneren Klasse von `BMJoin`, berechnet die Ergebnistupel des Joins durch Einbeziehung der Eingaberelationen, repräsentiert durch die beiden `FileScan`-Objekte, und der Vergleichsordnung und schreibt sie in eine Instanz der Klasse `bmj.datatypes.ResultBuffer`. Diese Klasse ist synchronisiert, so dass mehrere Threads

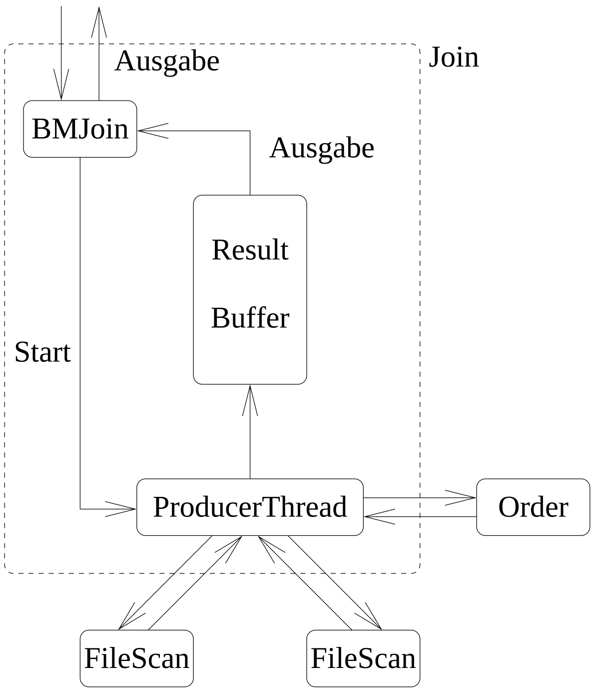


Abb. 5.4: Architektur der Implementierung des fensterbasierten Algorithmus

sicher auf ein Objekt dieser Klasse zugreifen können. Im vorliegenden Fall handelt es sich dabei um Schreibzugriffe durch den ProducerThread und um Lesezugriffe durch den Thread, in dem der restliche Prototyp und damit auch der Auswertungsplan ausgeführt wird. Die `next`-Methode des Joins kann dann bei jedem Aufruf das jeweils nächste Ergebnistupel aus dem Puffer auslesen und an den Plan zurückliefern. Abbildung 5.5 ordnet das Schema aus Bild 5.4 in die Gesamtübersicht aus Abbildung 5.3 ein.

Die Aufgaben der drei Iterator-Methoden der fensterbasierten Implementierung sind wie folgt. Die `open`-Methode öffnet die beiden Eingaberelationen, instanziiert den ResultBuffer und den ProducerThread und startet schließlich den Thread über dessen `start`-Methode. Damit beginnt der Thread mit der Ausführung seiner `run`-Methode, welche die Implementierung des fensterbasierten Algorithmus enthält. Endgültige Ergebnispaare, die aus dem Puffer für potenzielle Ergebnispaare des Algorithmus entfernt und zur Ausgabe oder zu nachfolgenden Operatoren propagiert werden sollen, schreibt der Thread mittels der synchronisierten `put`-Methode des ResultBuffers in den Ergebnispuffer. Dieser Ablauf ist in Abbildung 5.6 dargestellt.

Ein Aufruf der `next`-Methode des Joins holt das nächste Ergebnistupel durch die Ausführung der ebenfalls synchronisierten `get`-Methode des ResultBuffers aus dem Puffer und gibt es an den Aufrufer zurück. Währenddessen fährt der ProducerThread mit der Join-Berechnung fort. Die Aufrufe der Methoden `put` und `get` des ResultBuffers erfolgen aufgrund der Synchronisation unter gegenseitigem Ausschluss. Der Vorgang ist in Abbildung 5.7 gezeigt.

Der Aufruf der Methode `close` beendet schließlich die Join-Berechnung und führt die nötigen Aufräumarbeiten durch. Zunächst teilt die Methode dem ProducerThread durch

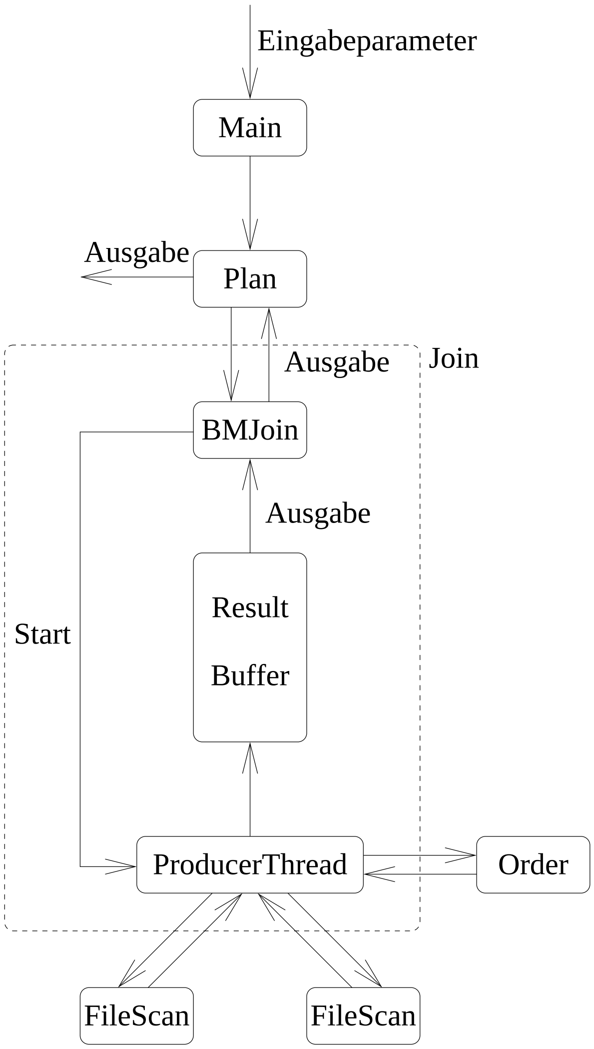


Abb. 5.5: Gesamtübersicht der Implementierung des fensterbasierten Algorithmus

Setzen eines Flags, das der Thread jedesmal vor dem Schreiben in den Ergebnispuffer abfragt, mit, dass er sich selbst beenden soll. Trifft der Thread auf ein gesetztes Flag, so terminiert sofort seine `run`-Methode und damit auch der gesamte Thread. Durch den Aufruf von `clear` auf dem ResultBuffer wird sichergestellt, dass der Puffer nicht voll ist und der Thread, falls er blockiert ist, weiterlaufen kann. Durch den Aufruf von `interrupt` wird die Ausführung des ProducerThread anschließend fortgeführt. Der Aufruf von `join` auf dem ProducerThread lässt die `close`-Methode bis zur Terminierung des Threads warten. Danach werden die Variablen für den ProducerThread und den ResultBuffer auf `null` gesetzt, was die entsprechenden Objekte dem Garbage Collector überlässt, und die `FileScan`-Objekte der beiden Eingaberelationen werden geschlossen. Eine schematische Übersicht über den Aufruf von `close` zeigt Abbildung 5.8.

Die Implementierungen der eigentlichen Algorithmen, sowohl des fensterbasierten als auch des Scheduling-Algorithmus, halten sich weitgehend an die in Kapitel 4.2 vorgestellten Verfahren. An manchen Stellen wurden aus Gründen der besseren Implementierbarkeit

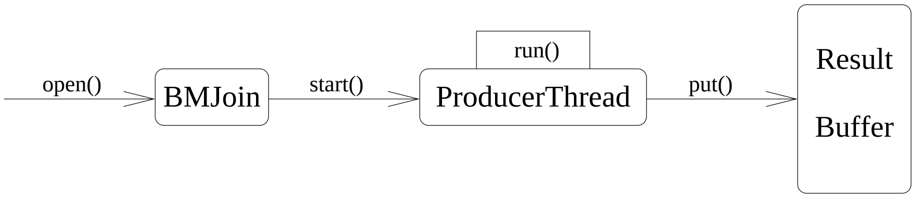


Abb. 5.6: Aufruf von `open` der fensterbasierten Implementierung

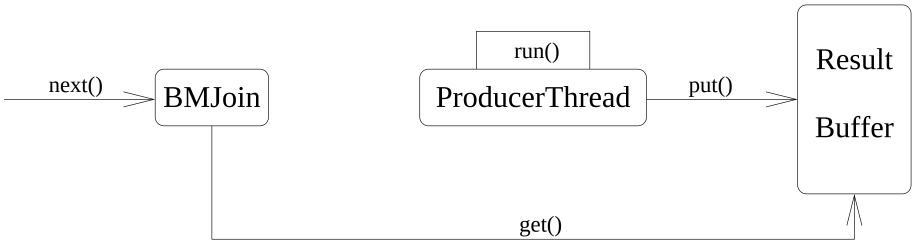


Abb. 5.7: Aufruf von `next` der fensterbasierten Implementierung

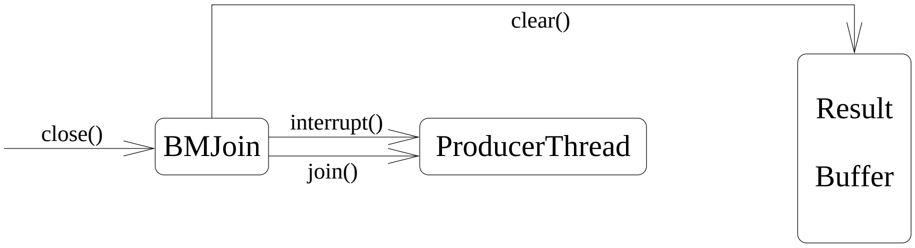


Abb. 5.8: Aufruf von `close` der fensterbasierten Implementierung

oder der Effizienz kleinere Veränderungen vorgenommen, die aber keinen Einfluss auf das Verhalten der Algorithmen haben. Die Implementierung von Algorithmus 4.6 auf Seite 63 verdient noch besondere Erwähnung, da es hier zu einem kleinen Problem kam. In Zeile 5 des Algorithmus wird überprüft, ob das linke Tupel des aktuell zu untersuchenden Tupelpaares mit dem linken Tupel des gegenwärtigen Vergleichspaares aus dem Puffer O für Zwischenergebnisse identisch ist. Die ursprüngliche Implementierung, noch ohne Materialisierung von Tupeln, verglich hier auf Objektidentität, was auch einwandfrei funktionierte. Nach dem Hinzufügen der Tupelmaterialisierung war dies aber nicht mehr der Fall. Das liegt daran, dass bei der Materialisierung die Attributwerte der Tupel auf dem Hintergrundspeicher abgelegt und die Objekte, welche die gespeicherten Tupel repräsentieren, verworfen werden. Beim Einlesen eines Tupels vom Hintergrundspeicher wird das Tupelobjekt rekonstruiert und dabei erneut instanziiert. Damit ist es aber nicht mehr dasselbe Objekt wie vor der Materialisierung und der Test auf Objektidentität schlägt fehl. Um dieses Problem zu beheben, wurde in der Klasse `bmj.datatypes.Tuple` eine eigene `equals`-Methode für `Tuple`-Objekte implementiert. Diese vergleicht die Attributwerte zweier Tupel und geht davon aus, dass die Tupel gleich sind, wenn sie für alle Attribute identische Attributwerte besitzen. Damit funktioniert obige Abfrage, nun mit einem Vergleich auf Wertgleichheit statt auf Objektidentität, wieder so wie gewünscht. Es wird darauf hingewiesen, dass beide Vergleichsvarianten ohne die Materialisierung von

Tupeln semantisch äquivalent sind, da es zum einen zu jedem Zeitpunkt von jedem Tupel einer Relation höchstens ein `Tuple`-Objekt im Prototyp geben kann und da zum anderen in dem verwendeten Modell jedes Tupel einer Relation einen Primärschlüssel besitzt. Damit kann keine Relation zwei wertgleiche Tupel enthalten.

5.4 Datengenerator

Um mit der Prototyp-Implementierung der Join-Algorithmen Leistungsmessungen durchführen zu können, war es nötig, einen Datengenerator zu implementieren, der die dafür nötigen Eingabedaten generieren kann. Der Generator besitzt mehrere Optionen, mit denen gesteuert werden kann, wie die erzeugten Relationen aussehen. Das Programm generiert Daten, die bestimmten relationalen Schemata genügen und speichert sie in Dateien im Dateisystem ab, die den Spezifikationen in Anhang A.3 entsprechen. Die Bedienung des Datengenerators ist in Anhang A.4 beschrieben.

5.4.1 Implementierung des Datengenerators

Der Datengenerator wird wie der Prototyp über ein Kommandozeilen-Interface gesteuert. Beim Aufruf werden neben den Optionen auch die Anzahl der zu erzeugenden Attributwerte pro Tupel und die Anzahl der Tupel in der generierten Relation angegeben. Der Generator erzeugt dann für jedes Tupel die angegebene Anzahl an Double-Attributen und hängt als letztes Attribut zusätzlich noch bei jedem Tupel ein String-Attribut an, dessen Länge ebenfalls einstellbar ist. Dieses String-Attribut erfüllt zwei Aufgaben. Zum einen dient es als Füllattribut, das die Tupel mit einer gewissen Menge an Daten füllt. Es repräsentiert somit die Nutzdaten. Zum anderen endet der String bei jedem Tupel mit einer fortlaufenden Nummer beginnend bei 0 für das erste Tupel. Damit stellt das String-Attribut den Primärschlüssel der erzeugten Relationen dar und ermöglicht die eindeutige Identifikation aller Tupel einer Relation. Die Tupel können in den Relationen unsortiert oder nach den Werten eines Double-Attributs aufsteigend sortiert generiert werden, um die Bedingung der Vorsortierung für den fensterbasierten Algorithmus erfüllen zu können. Um die Sortierung durchzuführen, legt der Datengenerator zunächst ein Double-Array an, das so viele Einträge hat, wie die zu erzeugende Relation Tupel enthalten soll. Dieses Array füllt er dann mit zufällig generierten Double-Werten. Schließlich sortiert er das Array unter Verwendung der entsprechenden Methode `sort` zur Sortierung von Double-Arrays aus der Klasse `java.util.Arrays`. Diese Methode verwendet einen modifizierten Quicksort-Algorithmus, um ein Double-Array nach aufsteigenden Double-Werten zu sortieren. Bei der Generierung der Tupel erzeugt der Datengenerator nun für jedes Double-Attribut zufällig einen Double-Wert, außer für das sortierte Attribut. Dieses erhält den entsprechenden Wert aus dem sortierten Array. Auf diese Weise sind alle Tupel der Relation am Ende aufsteigend nach den Werten des betreffenden Attributs sortiert. Für die zufällige Generierung von Double-Werten unterstützt der Datengenerator vier verschiedene Verteilungen, die im nächsten Abschnitt vorgestellt werden.

5.4.2 Verteilungen

Sämtliche Verteilungen, die mit dem Datengenerator verwendet werden können, implementieren das Interface `bmj.test.Distribution` und damit auch die einzige darin enthaltene Methode `generate`. Der Aufruf dieser Methode gibt einen zufällig generierten Double-Wert zurück. Die vier vorhandenen Verteilungen umfassen die uniforme Verteilung, die auch Gleichverteilung heißt, die Normalverteilung sowie eine korrelierte und eine anti-korrelierte Verteilung. Nachfolgend werden diese Verteilungen und ihre Implementierungen beschrieben.

Uniforme Verteilung

Bei der uniformen Verteilung wird jeder Attributwert im Intervall $[0;1]$ mit der gleichen Wahrscheinlichkeit generiert. Abbildung 5.9 zeigt dies am Beispiel eines zweidimensionalen Datenraums. Alle erzeugten Datenpunkte verteilen sich gleichmäßig auf den Bereich zwischen 0 und 1 in beiden Dimensionen y_1 und y_2. Die Generierung der Double-Werte erfolgt durch die Methode `nextDouble` der Klasse `java.util.Random`. Dazu wird bei der Instanziierung einer uniformen Verteilung auch ein Objekt der Klasse `Random` erzeugt. Die von `nextDouble` gelieferten Werte werden noch mit der Rundungsmethode `round` aus `bmj.ToolBox` standardmäßig auf fünf Nachkommastellen gerundet und anschließend an den Aufrufer der Methode `generate` zurückgegeben. Da die Rückgabewerte der Methode `nextDouble` im Intervall $[0;1[$ liegen, befinden sich die gerundeten Rückgabewerte von `generate` damit im gewünschten Intervall $[0;1]$.

Normalverteilung

Die Normalverteilung benutzt die Methode `nextGaussian` der Klasse `java.util.Random` zur Erzeugung zufälliger Double-Werte. Bei dieser Verteilung können bei der Instanziierung zusätzlich ein Mittelwert und eine Standardabweichung angegeben werden. Standardmäßig ist der Mittelwert der Normalverteilung in der Klasse `Random` auf 0.0 und die Standardabweichung auf 1.0 gesetzt. Der Datengenerator besitzt allerdings seine eigene Standardeinstellung. Sie liegt bei 0.5 für den Mittelwert und 0.15 für die Standardabweichung. Bei der Generierung der Double-Werte wird der von `nextGaussian` gelieferte Wert mit der Standardabweichung multipliziert und anschließend wird zu dem so erhaltenen Wert der Mittelwert hinzuaddiert. Danach erfolgt noch die Rundung mittels `bmj.ToolBox.round`. Da im Gegensatz zur uniformen Verteilung hier nicht garantiert ist, dass die erzeugten Werte im Intervall $[0;1]$ liegen, verwirft die Implementierung alle zufälligen Double-Werte, die sich außerhalb dieses Bereichs befinden. Bei einem Aufruf von `generate` werden also so lange Double-Werte gemäß oben beschriebener Vorgehensweise generiert, bis der erste Wert angetroffen wird, der innerhalb des gewünschten Intervalls liegt. Dieser wird dann zurückgegeben. Es ist offensichtlich, dass es bei entsprechend gewählten Werten für den Mittelwert und die Standardabweichung mitunter sehr lange dauern kann, bis ein passender Wert generiert wird. Setzt man den Mittelwert auf eine Zahl außerhalb des vorgegebenen Intervalls $[0;1]$ und die Standardabweichung auf 0, dann

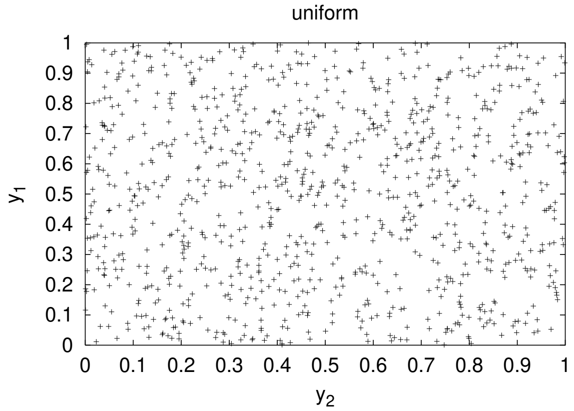


Abb. 5.9: Uniforme Verteilung

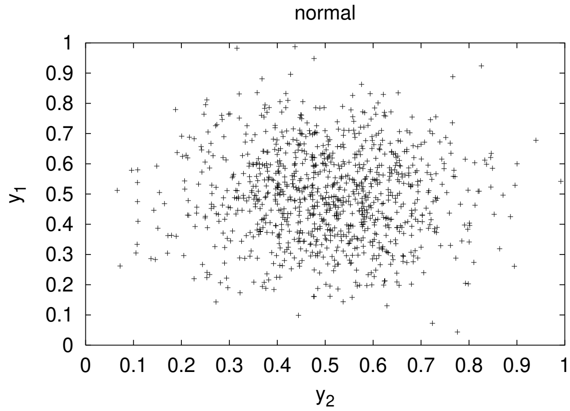


Abb. 5.10: Normalverteilung

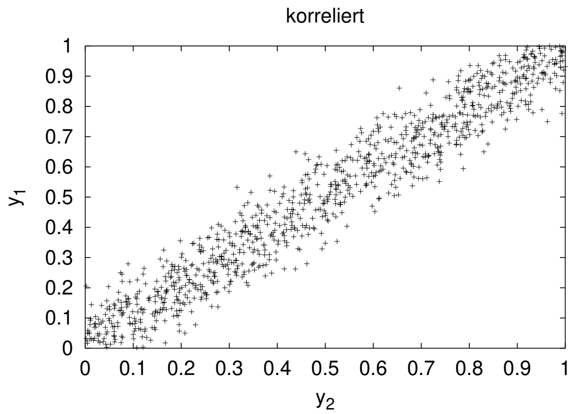


Abb. 5.11: Korrelierte Verteilung

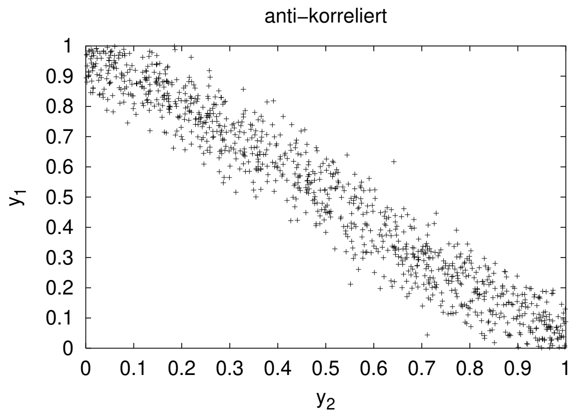


Abb. 5.12: Anti-korrelierte Verteilung

wird es sogar nie einen passenden Wert geben, da jede generierte Zahl gleich dem Mittelwert ist. In diesem Fall würde es zu einer Endlosschleife kommen. Um dies zu vermeiden, lässt der Datengenerator grundsätzlich nur Mittelwerte im Bereich $[0;1]$ zu. Die Standardabweichung ist beliebig, darf aber nicht negativ sein. Insbesondere ist es möglich, mit der Normalverteilung mit einem Mittelwert aus dem Intervall $[0;1]$ und einer Standardabweichung von 0 eine Relation zu erzeugen, in der alle Double-Attribute sämtlicher Tupel den gleichen Wert, nämlich den des eingestellten Mittelwerts, haben. Die Normalverteilung ist in Abbildung 5.10 für das bekannte Beispiel eines zweidimensionalen Datenraums dargestellt. Der Mittelwert beträgt 0.5, die Standardabweichung 0.15. Deutlich ist die Ballung der Datenpunkte in der Mitte des Datenraums erkennbar.

Korrelierte Verteilung

Die Idee hinter der korrelierten Verteilung basiert darauf, dass Attributwerte von den Werten anderer Attribute abhängig sind, wie das in praktischen Anwendungen häufig der Fall ist. Beispielsweise kostet eine Lebensversicherung oder eine private Krankenversicherung umso mehr, je älter der Versicherungsnehmer ist. Die Generierung einer korrelierten Relation funktioniert im Datengenerator wie folgt. Als Bezugsdimension dient die sortierte Dimension, falls eine solche vorhanden ist. Ansonsten wird dafür einfach die erste Dimension der Relation herangezogen. Die Werte aller Tupel für diese Dimension werden mit einer uniformen Verteilung erzeugt. Die Werte für alle anderen Dimensionen eines Tupels ergeben sich dann dadurch, dass zu dem Wert der Bezugsdimension dieses Tupels ein Differenzwert hinzuaddiert wird. Dieser Wert kann positiv oder negativ sein. Die korrelierten Attributwerte schwanken dann alle um den Bezugswert. Damit die Wahrscheinlichkeit für nahe am Bezugswert liegende korrelierte Werte höher ist als für weiter entfernte, benutzt der Datengenerator für die Erzeugung der Differenzwerte eine Normalverteilung. Diese hat immer den Mittelwert 0 und eine voreingestellte Standardabweichung von 0.075. Diese lässt sich aber wiederum durch eine Kommandozeilenoption im Bereich nicht negativer Zahlen ändern. Eine Option zur Veränderung des verwendeten Mittelwertes ist zwar vorhanden, lässt aber bisher keine anderen Werte als 0 zu, da sonst auch hier keine Garantie dafür gegeben werden kann, dass der Datengenerator terminiert. Das liegt daran, dass auch die generierten korrelierten Werte in dem Intervall $[0; 1]$ liegen müssen. Fällt die Summe aus Bezugswert und Differenzwert aus diesem Bereich heraus, wird sie verworfen und ein neuer Differenzwert wird generiert. Dies geschieht so lange, bis die Summe aus Bezugswert und Differenzwert innerhalb des Intervalls liegt und somit als gültiger Attributwert verwendbar ist. Wenn der Mittelwert nicht 0 ist, so ist es möglich, dass bei entsprechender Wahl der Standardabweichung stets Differenzwerte generiert werden, die zusammen mit einem bestimmten Bezugswert nie einen gültigen Attributwert ergeben. Dies würde wie schon bei der Normalverteilung wieder zu einer Endlosschleife führen. Setzt man z. B. den Mittelwert auf 0.1 und die Standardabweichung auf 0, so haben alle erzeugten Differenzwerte den Wert 0.1, entsprechen also dem Mittelwert. Bei einem Bezugswert von 0.95 würde dies stets zu einem Attributwert von 1.05 führen, der außerhalb des vorgegebenen Intervalls $[0; 1]$ liegt. In Abbildung 5.11 ist ein Beispiel für eine korrelierte Verteilung von Datenpunkten auf den beiden Dimensionen y_1 und y_2 dargestellt.

Anti-korrelierte Verteilung

Die anti-korrelierte Verteilung ist der korrelierten sehr ähnlich. Allerdings verhalten sich hier die übrigen Attributwerte eher gegensätzlich zum Bezugswert. Ein großer Bezugswert führt also nicht zu großen Werten für die abhängigen Attribute, wie bei einer korrelierten Verteilung, sondern zu kleinen. Umgekehrt führt ein kleiner Bezugswert auch nicht zu kleinen Werten für die abhängigen Attribute, sondern entsprechend zu großen. In der Praxis kommt das beispielsweise bei KFZ-Versicherungen vor. Je jünger der Versicherungsnehmer ist, desto teurer ist, wegen des statistisch gesehen höheren Unfallrisikos, die Versicherung. Der Datengenerator benutzt für die Erzeugung anti-korrelierter Relationen denselben Code wie für korrelierte Relationen. Der einzige Unterschied ist, dass nun ein in einer if-Abfrage enthaltener Codeteil zusätzlich ausgeführt wird. Diese Abfrage testet, ob anti-korrelierte Daten erzeugt werden sollen. Wenn das der Fall ist, verändert der Datengenerator jeden erzeugten korrelierten Wert k, indem er ihn unmittelbar nach seiner Generierung auf $1 - k$ setzt. Das Ergebnis ist für das bekannte Beispiel mit zwei Dimensionen in Abbildung 5.12 gezeigt.

5.5 Zusammenfassung

In diesem Kapitel wurden die wichtigsten Punkte des Entwurfs und der Implementierung eines Prototyps zur Berechnung des Left-Outer-Bestmatch-Joins mit Einschränkungen mittels des naiven und des fensterbasierten Algorithmus beschrieben. Die wichtigsten Ziele waren dabei der Nachweis der Funktionsfähigkeit der fensterbasierten Methode und die Möglichkeit zur Leistungsmessung und zum Leistungsvergleich zwischen dem naiven und dem fensterbasierten Ansatz. Außerdem wurde bei der Implementierung Wert auf Effizienz und Modularität gelegt, um einen möglichst schnellen Prototyp zu erhalten, der bei den Leistungsmessungen repräsentative Ergebnisse liefert und der zudem leicht erweiterbar und um neue Funktionalität ergänzbar ist. Das Kapitel beschrieb auch einige Implementierungsalternativen und Optimierungsmöglichkeiten und beleuchtete sie unter den Aspekten ihrer Anwendbarkeit für den Prototyp und ihres Nutzens insbesondere bezüglich der Effizienz. Im folgenden Kapitel werden nun die Ergebnisse der mit dem Prototyp durchgeführten Leistungsmessungen vorgestellt.

Kapitel 6

Leistungsmessung

Dieses Kapitel gibt einen Überblick über die wichtigsten Leistungsmessungen, die mit der im Vorangegangenen beschriebenen Prototyp-Implementierung durchgeführt worden sind, und über deren Ergebnisse. Zunächst wird eine Übersicht über die verwendete Messumgebung und die verschiedenen, im Rahmen der Messungen variierten Parameter gegeben. Anschließend folgen die wichtigsten Ergebnisse der Leistungsmessungen und deren Auswertung. Eine Zusammenfassung hält schließlich die zentralen Erkenntnisse noch einmal fest.

6.1 Umgebung und Parameter

Die Messumgebung für die Leistungstests besteht aus den verwendeten Eingabedaten und der Hardware des Testrechners, auf dem die Messungen durchgeführt wurden. Sämtliche Eingabedaten des Prototyps, die bei den Leistungsmessungen zum Einsatz kamen, stammen aus dem Datengenerator und entsprechen dem in Anhang A.3 beschriebenen Format. Jedes Tupel enthält zehn Attribute vom Typ Double und ein Attribut vom Typ String. Die Double-Werte sind auf fünf Nachkommastellen gerundet und das String-Attribut ist jeweils 16 Zeichen lang. Jedes Tupel ist somit materialisiert genau 100 Byte groß. Alle Eingaberelationen sind nach dem ersten Double-Attributwert aufsteigend sortiert. Die Messungen wurden auf einem Sun Enterprise 450 Server mit 4 GB Hauptspeicher und vier 400 Mhz UltraSPARC-II Prozessoren, von denen die Prototyp-Implementierung aber nur jeweils einen benutzt hat, durchgeführt. Als Betriebssystem wurde Sun Solaris 7 eingesetzt.

Einen Überblick über die einzelnen Messparameter gibt Tabelle 6.1. Insgesamt wurden bei den Messungen acht verschiedene Parameter variiert. Die Tabelle führt diese Parameter, ihre Standardeinstellungen und eine kurze Beschreibung des jeweiligen Parameters auf. Zur Join-Berechnung wurden sowohl der naive als auch der fensterbasierte Algorithmus verwendet. Auch die Materialisierungsvariante[1], Standard oder Grid, wurde variiert. Jede der vier in Abschnitt 5.4.2 beschriebenen Verteilungen wurde für die Erstellung von

[1] Siehe hierzu die Abschnitte 4.3.3 auf Seite 65 und 5.2.4 auf Seite 82.

Parameter	Standardeinstellung	Beschreibung
Join	fensterbasiert	Veranlasst den Prototyp zur Benutzung des naiven oder des fensterbasierten Joins.
Materialisierungs-variante	Standard	Veranlasst den Prototyp zur Benutzung der Standard- oder der Grid-Materialisierungsvariante.
Verteilung	uniform	Die Verteilung der Eingabedaten (uniform, normal, korreliert oder antikorreliert).
Eingabegröße	1000	Die Anzahl an Tupeln pro Eingaberelation.
Dimensionen	2	Die Anzahl der verwendeten Join-Dimensionen.
ϵ-Umgebung	0.1	Der Wert der ϵ_i für jede Join-Dimension i.
Seitenkapazität	8 KB	Die Kapazität einer Seite auf dem Hintergrundspeicher in Kilobyte.
Hauptspeicher-kapazität	5 Seiten	Die Kapazität des Hauptspeicherpuffers je Eingaberelation in Seiten.

Tab. 6.1: Parameter der Leistungsmessungen

Eingaberelationen für die Testläufe benutzt. Die bevorzugte Verteilung ist die uniforme Verteilung. Ein weiterer Parameter ist die Größe der Eingaberelationen, d. h. die Anzahl der in den Relationen enthaltenen Tupel. Für die Messungen enthalten beide Eingaberelationen eines Joins stets gleich viele Tupel. Standardmäßig sind das 1000 Tupel pro Relation. Als Join-Dimensionen werden jeweils Attribute vom Typ Double herangezogen. Bei beispielsweise zwei Join-Dimensionen stellen somit die Werte der restlichen acht Double-Attribute Nutzdaten dar, die keinen Einfluss auf die Berechnung des Join-Ergebnisses haben. Die ϵ-Umgebung für den eingeschränkten Left-Outer-Bestmatch-Join, repräsentiert durch die Werte der ϵ_i für jede Join-Dimension i, beträgt standardmäßig 0.1 für alle ϵ_i. Die Prototyp-Implementierung ermöglicht selbstverständlich auch unterschiedliche Werte für die einzelnen ϵ_i einer ϵ-Umgebung. Im Rahmen der Leistungsmessungen wurden aber ausschließlich Umgebungen mit gleichen Werten für alle Dimensionen verwendet. Auch die Kapazität der Seiten auf dem Hintergrundspeicher, in denen die materialisierten Tupel gespeichert werden, ist ein veränderbarer Parameter. Die Kapazität wird in Kilobyte angegeben. Der Standardwert ist 8 KB. Der letzte Parameter ist die Hauptspeicherkapazität. Sie wird in der Anzahl an Seiten angegeben, die gleichzeitig im zur Verfügung stehenden Hauptspeicherpuffer des Prototyps Platz finden. Aus der Seitenanzahl und der Seitenkapazität ergibt sich so die Kapazität des Hauptspeichers in Kilobyte. Die Angabe der Hauptspeicherkapazität erfolgt pro Eingaberelation, d. h. standardmäßig bietet der Hauptspeicher Platz für fünf Seiten der linken und für fünf Seiten der rechten Eingabe, insgesamt also für zehn Seiten. Auch hier sind seitens der Implementierung des Prototyps verschiedene Werte für die linke und die rechte Eingabe möglich. Die Leistungsmessungen beschränken sich aber auf jeweils gleiche Werte für beide Eingaben.

Soweit nicht anders angegeben, besitzen im Folgenden alle Parameter die in Tabelle 6.1 aufgeführte Standardeinstellung. Abweichungen von diesen Einstellungen werden ausdrücklich erwähnt oder sind aus den Diagrammen ersichtlich.

6.2 Messungen und Auswertung

Die durchgeführten Messungen lassen sich in zwei Arten klassifizieren. Die Messungen von Laufzeit und Seitenzugriffen geben die gesamte Laufzeit der Verfahren zur Verarbeitung einer endlichen Eingaberelation mit einer bestimmten Größe sowie die dabei erfolgte Anzahl an Lesezugriffen auf Seiten des Hintergrundspeichers wieder. Die hierbei gemessenen Laufzeiten dienen vor allem zur objektiven Einschätzung der praktischen Verarbeitungsgeschwindigkeiten der getesteten Verfahren unter verschiedenen Bedingungen. Die Veränderung der Bedingungen wird dabei durch die Variation der oben aufgeführten Parameter erreicht. Wegen der Abhängigkeit der Laufzeit von der zugrunde liegenden Hardware wird zusätzlich die Anzahl an lesenden Seitenzugriffen auf den Hintergrundspeicher gemessen. Sie dient als abstraktes Maß für die Leistung der Verfahren. Da ein Zugriff auf eine Seite des Hintergrundspeichers vergleichsweise sehr viel Zeit kostet, sind möglichst wenig Seitenzugriffe wünschenswert. Die Anzahl der lesenden Seitenzugriffe auf den Hintergrundspeicher korreliert nicht notwendigerweise immer mit der Laufzeit, da auch schreibende Zugriffe auf den Hintergrundspeicher und die eigentliche Join-Berechnung maßgeblich zur Gesamtlaufzeit beitragen können. Die Messungen der Ergebnistupel pro Zeit sollen zeigen, wie kontinuierlich die Verfahren Ergebnistupel propagieren und wie lange es dauert, bis eine bestimmte Menge an solchen Tupeln ausgegeben worden ist. Sie zielen somit auf die Untersuchung der Pipeline-Fähigkeit der Verfahren auf mitunter unendlichen Datenströmen ab.

6.2.1 Laufzeit und Seitenzugriffe

Die folgenden Tabellen und Diagramme enthalten bzw. visualisieren eine Auswahl der wichtigsten durchgeführten Messungen und Messergebnisse von Laufzeiten und Seitenzugriffen der Prototyp-Implementierung. Dabei liegt das Hauptaugenmerk auf dem Verhalten des fensterbasierten Algorithmus im Vergleich zum naiven Verfahren und auf den Unterschieden zwischen der Standard- und der Grid-Materialisierungsvariante. Diese werden bei verschiedenen Verteilungen der Eingabedaten, variierenden Eingabegrößen, unterschiedlicher Anzahl an Join-Dimensionen, unterschiedlich großer ϵ-Umgebung, wechselnder Seitenkapazität und verschiedenen Hauptspeicherkapazitäten miteinander verglichen. Außerdem wird untersucht, wie sich die fensterbasierte Methode mit und ohne die Erweiterung durch die Epsilon-Grid-Ordnung und den Scheduling-Algorithmus verhält. Insgesamt gibt es Messungen mit sechs verschiedenen Verfahren:

fensterbasiert/Standard/EGO: Fensterbasierter Algorithmus mit Standard-Materialisierung unter Benutzung von Epsilon-Grid-Ordnung und Scheduling-Algorithmus.

fensterbasiert/Standard: Fensterbasierter Algorithmus mit Standard-Materialisierung ohne Epsilon-Grid-Ordnung und Scheduling-Algorithmus.

fensterbasiert/Grid/EGO: Fensterbasierter Algorithmus mit Grid-Materialisierung unter Benutzung von Epsilon-Grid-Ordnung und Scheduling-Algorithmus.

fensterbasiert/Grid: Fensterbasierter Algorithmus mit Grid-Materialisierung ohne Epsilon-Grid-Ordnung und Scheduling-Algorithmus. Dieses Verfahren dient der Ermittlung des Verhaltens der Grid-Materialisierungsvariante bei fehlender Sortierung der zu materialisierenden Daten nach Epsilon-Grid-Ordnung und abgeschaltetem Scheduling-Algorithmus.

naiv/Standard: Naiver Algorithmus mit Standard-Materialisierung.

naiv/Grid: Naiver Algorithmus mit Grid-Materialisierung. Auch dieses Verfahren dient der Beobachtung des Verhaltens der Grid-Materialisierungsvariante ohne Verwendung der Epsilon-Grid-Ordnung und des Scheduling-Algorithmus.

Variation der Eingabegrößen

Zunächst sollen die Gesamtlaufzeit und die Anzahl an Seitenzugriffen der Verfahren bei Eingaberelationen unterschiedlicher Größe betrachtet werden. Die Tupelanzahl pro Relation wurde dabei in Schritten von 100 Tupeln zwischen 100 und 1000 Tupeln variiert. Tabelle 6.2 zeigt die Laufzeiten in Sekunden und die Anzahl der Seitenzugriffe der sechs verschiedenen Verfahren bei 1000 Tupeln pro Eingaberelation für alle vier Verteilungen. Wie man sieht, verursacht die Variante fensterbasiert/Standard/EGO bei jeder der vier Verteilungen der Eingabedaten mit Abstand die geringste Anzahl an Seitenzugriffen. Die Laufzeiten sind bei fensterbasiert/Grid/EGO am kürzesten. Dies erklärt sich daraus, dass

	uniform		normal		korreliert		anti-korreliert	
	Zeit [s]	# Seiten	Zeit [s]	# Seiten	Zeit [s]	# Seiten	Zeit [s]	# Seiten
fensterbasiert Standard EGO	60,6	13.599	195,6	18.172	100,0	14.457	102,7	17.530
fensterbasiert Standard	98,8	66.980	283,1	130.764	142,7	71.445	146,0	70.666
fensterbasiert Grid EGO	48,3	47.331	171,6	40.508	88,7	20.620	84,5	20.758
fensterbasiert Grid	1.144,8	3.429.327	2.991,5	5.801.817	816,3	1.516.224	787,8	1.407.399
naiv Standard	588,1	259.073	1.649,2	758.102	1.674,1	756.627	1.672,6	758.908
naiv Grid	868,7	1.983.216	2.149,7	3.893.838	1.938,8	2.709.166	1.831,2	2.549.399

Tab. 6.2: Laufzeiten und Seitenzugriffe bei 1000 Tupeln pro Eingaberelation

die Grid-Materialisierungsvariante im Vergleich zur Standard-Variante das Mischen bereits materialisierter Daten mit neuen Tupeln einspart. Dadurch ist bei der Materialisierung der Datenfenster weniger Schreiben und Lesen auf dem Hintergrundspeicher erforderlich. Die höhere Anzahl an Seitenzugriffen bei der Grid-Variante liegt daran, dass aufgrund der Verteilung der Tupel auf die Seiten des Gitters die einzelnen Seiten in der Regel wesentlich dünner besetzt sind als beim Standard-Verfahren. Das führt insgesamt dazu, dass mehr Seiten vorhanden sind, auf die sich die Tupel verteilen. Damit müssen entsprechend auch mehr Seiten geladen werden, um den gleichen Tupelbestand zu verarbeiten. Die Varianten fensterbasiert/Standard und fensterbasiert/Grid, die ohne Epsilon-Grid-Ordnung und Scheduling-Algorithmus arbeiten, verlieren sowohl bei der Laufzeit als auch bei der Anzahl an Seitenzugriffen klar gegen die Verfahren mit EGO. Insbesondere bei fensterbasiert/Grid steigen Laufzeit und Seitenzugriffe im Vergleich zu fensterbasiert/Grid/EGO massiv an. Schließlich ist die Grid-Materialisierungsvariante speziell auf die Verwendung der Epsilon-Grid-Ordnung ausgerichtet. Wird diese nicht eingesetzt, verliert die Grid-Variante ihre Vorteile und der Mehraufwand für die Verwaltung der Datenstruktur mit den Referenzobjekten macht sich stark bemerkbar.

Beim naiven Algorithmus ist die Standard-Materialisierung dem Grid-Verfahren überlegen, was angesichts der fehlenden Sortierung nach Epsilon-Grid-Ordnung nicht überrascht. Insgesamt ist die Leistung der naiven Methode was Laufzeit und Seitenzugriffe betrifft deutlich schlechter als die der ersten drei fensterbasierten Verfahren. Die fensterbasierten Implementierungen unter Verwendung der Epsilon-Grid-Ordnung und des Scheduling-Algorithmus zeigen sowohl bei Standard- als auch bei Grid-Materialisierung wesentlich bessere Leistungswerte als alle anderen Verfahren. Bezogen auf die Ausführungszeit gewinnt die Methode fensterbasiert/Grid/EGO diesen Vergleich.

Die Abbildungen 6.1 bis 6.4 zeigen das Verhalten der vier interessantesten Verfahren für alle gängigen Verteilungen der Eingabedaten bei einer Variation der Tupelanzahl in jeder der beiden Eingaberelationen R und S zwischen 100 und 1000 Tupeln. Auf den logarithmisch skalierten Hochwertachsen der Diagramme ist jeweils die Ausführungszeit in Minuten bzw. die Anzahl an lesenden Seitenzugriffen auf Seiten des Hintergrundspeichers aufgetragen. Der Verlauf der Kurven ist für alle Verteilungen in etwa gleich. Allerdings liegen sie bei der Normalverteilung sowie der korrelierten und der anti-korrelierten Vertei-

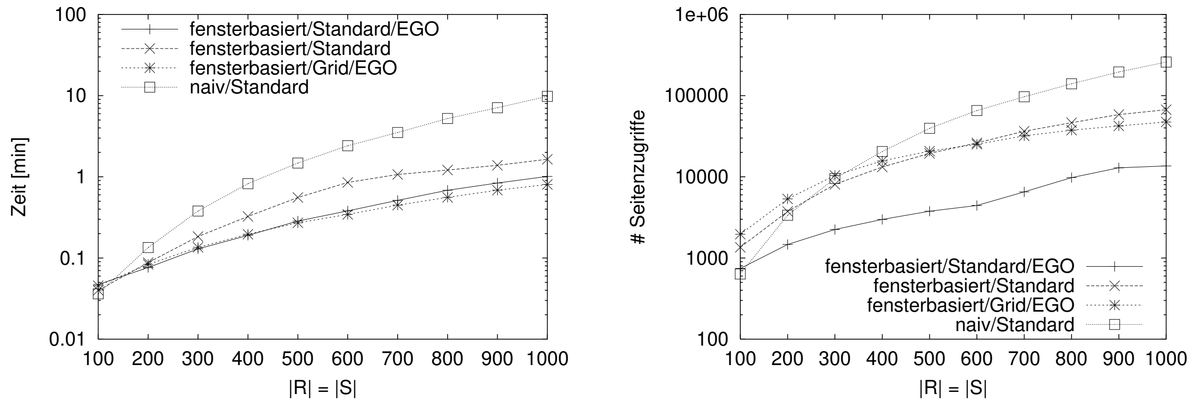


Abb. 6.1: Variation der Eingabegrößen bei uniformer Verteilung

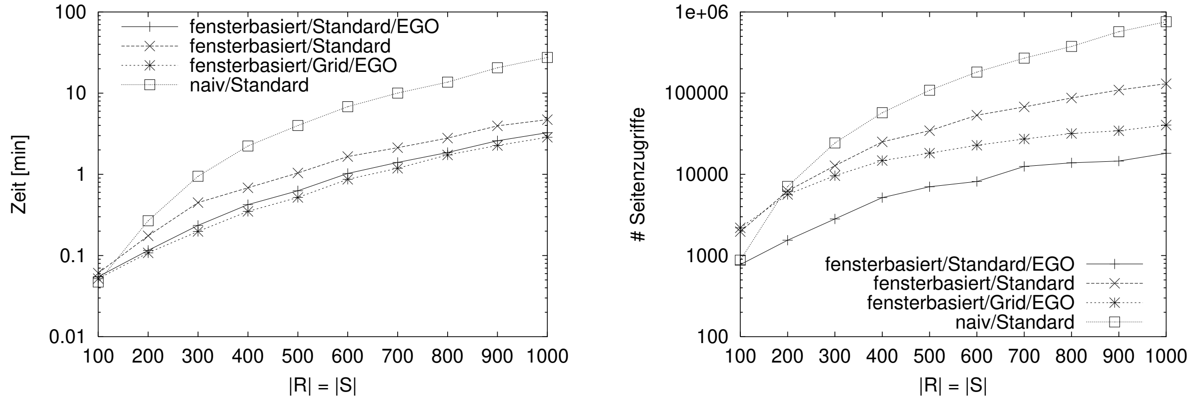


Abb. 6.2: Variation der Eingabegrößen bei Normalverteilung

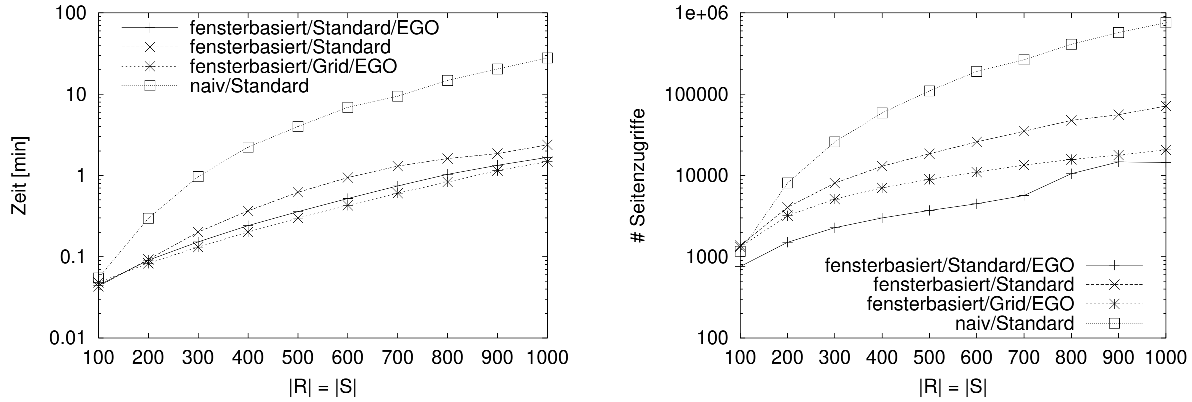


Abb. 6.3: Variation der Eingabegrößen bei korrelierter Verteilung

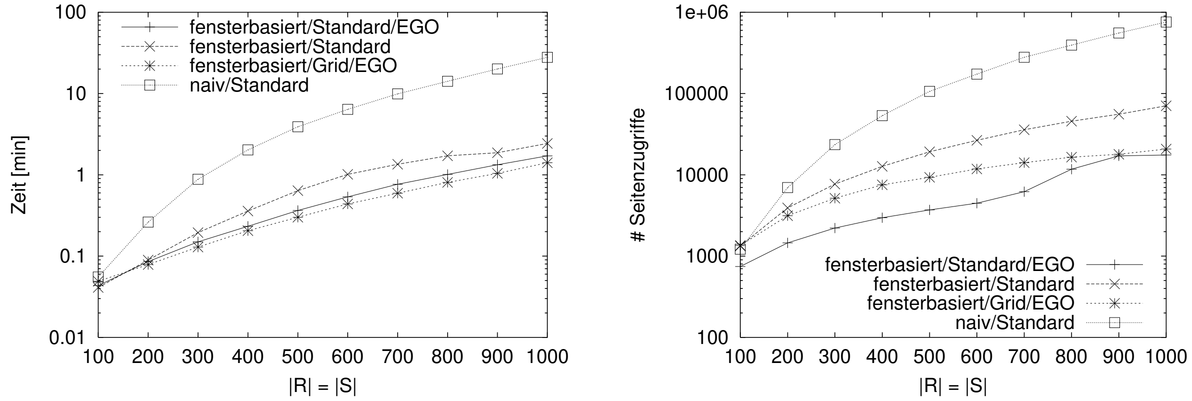


Abb. 6.4: Variation der Eingabegrößen bei anti-korrelierter Verteilung

lung auf einem höheren Niveau als bei der uniformen Verteilung. Das hat seine Ursache darin, dass bei diesen Verteilungen die Tupel in einem gewissen Bereich des Datenraums dicht gedrängt liegen, wie aus den Abbildungen 5.10 bis 5.12 ersichtlich ist. Damit finden sich für ein Tupelpaar mehr relevante Vergleichspaare innerhalb der entsprechenden ϵ-Umgebung, so dass mehr Vergleiche stattfinden und weniger Paare eliminiert werden können. Dies führt zu längeren Laufzeiten und zu mehr Seitenzugriffen. Weiterhin ist aus den Diagrammen ersichtlich, dass das naive Verfahren sowohl in Bezug auf die Laufzeit als auch in Bezug auf die Anzahl an Seitenzugriffen den fensterbasierten Verfahren im Allgemeinen weit unterlegen ist. Insbesondere wird der Abstand zwischen den gemessenen Werten der naiven und der fensterbasierten Implementierung mit größer werdenden Eingaberelation ebenfalls sehr schnell immer größer. Die fensterbasierten Verfahren sind also im Vergleich zur naiven Variante umso besser, je größer die Eingaberelationen sind. Das liegt daran, dass der reine Nested-Loops Ansatz des naiven Algorithmus bei größer werdenden Datenbeständen rasch sehr ineffizient wird. Lediglich bei kleinen Eingaberelationen mit wenigen hundert Tupeln kann das naive Verfahren mithalten und ist mitunter sogar, wenn auch nur geringfügig, besser als der fensterbasierte Ansatz, da es hier von seiner wenig Zusatzaufwand verursachenden Einfachheit profitieren kann. Allerdings sind derart kleine Eingaben in der Praxis vermutlich eher selten anzutreffen, insbesondere nicht bei der Verarbeitung von unendlichen Datenströmen, für die der naive Algorithmus wegen seiner blockierenden Architektur ohnehin nicht einsetzbar ist.

Offensichtlich verursacht das Verfahren fensterbasiert/Standard/EGO bei jeder Verteilung die wenigsten Seitenzugriffe, was am Einsatz des Scheduling-Algorithmus und der im Vergleich zur Grid-Materialisierung geringeren Anzahl an Seiten, die dafür entsprechend besser gefüllt sind, liegt. Es folgen die Verfahren fensterbasiert/Grid/EGO und fensterbasiert/Standard. Bei den Laufzeiten ist fensterbasiert/Grid/EGO am schnellsten, dicht gefolgt von fensterbasiert/Standard/EGO. Die Variante fensterbasiert/Standard ohne Epsilon-Grid-Ordnung und Scheduling-Algorithmus fällt hingegen schon merklich ab. Dies zeigt, dass sich der Mehraufwand für die Sortierung nach Epsilon-Grid-Ordnung und den Scheduling-Algorithmus nicht nur bei den Seitenzugriffen, sondern auch bei der Laufzeit deutlich auszahlt.

Abbildung 6.5 zeigt Laufzeiten und Seitenzugriffe bei uniformer Verteilung und variierenden Eingabegrößen noch einmal für alle sechs Verfahren. Daraus geht hervor, dass die beiden Verfahren fensterbasiert/Grid und naiv/Grid, bei denen der Einsatz der Grid-Materialisierungsvariante aufgrund des fehlenden Scheduling-Algorithmus eigentlich nicht sinnvoll ist, deutlich schlechter abschneiden als alle anderen Methoden. Hier schlägt sich der Mehraufwand für die Verwaltung der Referenzobjekte in der Grid-Datenstruktur nieder, ohne dass dieser Ansatz an anderer Stelle Vorteile bringt.

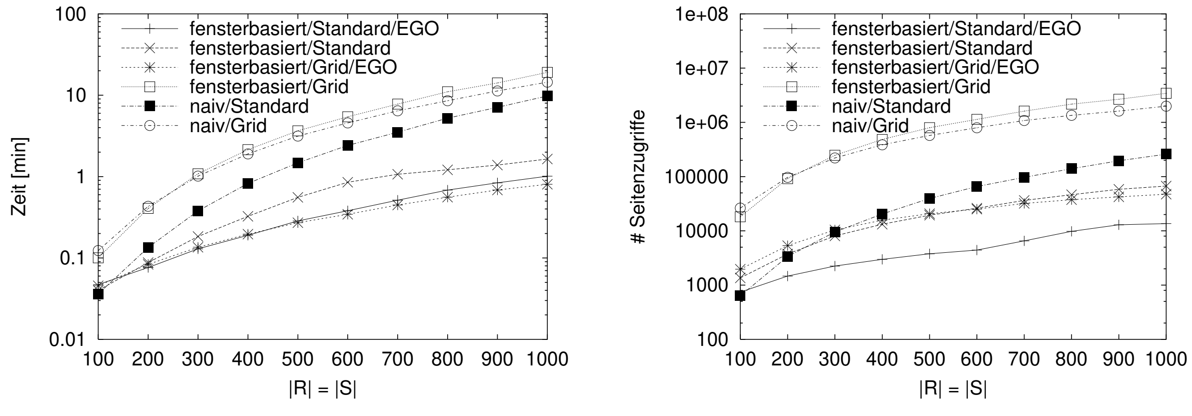


Abb. 6.5: Vergleich aller Verfahren bei variierenden Eingabegrößen und uniformer Verteilung

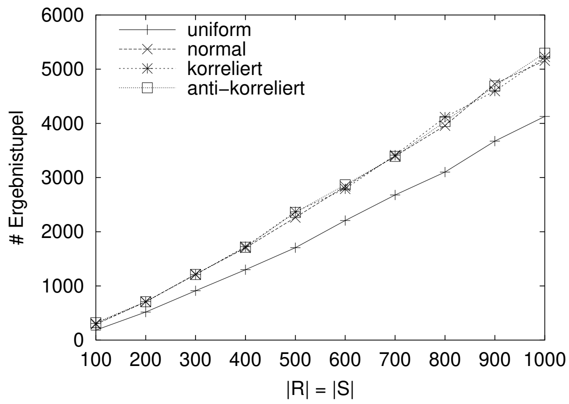


Abb. 6.6: Anzahl der Ergebnistupel bei variierenden Eingabegrößen

Schließlich zeigt Abbildung 6.6 noch die Anzahl der erzeugten Ergebnistupel für alle vier Verteilungen bei unterschiedlichen Eingabegrößen. Daraus ist ersichtlich, dass bei Normalverteilung sowie korrelierter und anti-korrelierter Verteilung durchgehend mehr Ergebnistupel erzeugt werden als bei uniformer Verteilung. Bei 1000 Tupeln pro Eingaberelation enthält das Ergebnis für diese Verteilungen beispielsweise über 1000 Tupel mehr als für die uniforme Verteilung. Dies erklärt sich wiederum aus der dichteren Konzentration von Tupeln in bestimmten Bereichen des Datenraums, so dass hier mehr relevante Tupelkombinationen innerhalb der vorgegebenen ϵ-Umgebung gefunden werden.

Variation der Anzahl an Join-Dimensionen

Ein weiterer interessanter Parameter für die Leistungsmessung ist die Anzahl an Join-Dimensionen. In Tabelle 6.3 sind die Laufzeiten und die Anzahl an Seitenzugriffen für alle sechs Verfahren und die vier Verteilungen der Eingabedaten bei fünf Join-Dimensionen aufgeführt. Auch hier weist das Verfahren fensterbasiert/Standard/EGO die geringste Anzahl an Seitenzugriffen auf. Bei der Laufzeit ist es ebenfalls wieder eines der besten. Nur bei der uniformen Verteilung ist das naive Verfahren mit Standard-Materialisierung noch

	uniform		normal		korreliert		anti-korreliert	
	Zeit [s]	# Seiten	Zeit [s]	# Seiten	Zeit [s]	# Seiten	Zeit [s]	# Seiten
fensterbasiert Standard EGO	47,1	12.583	79,4	16.858	110,5	11.889	117,5	13.721
fensterbasiert Standard	82,1	66.980	165,1	130.764	149,4	71.445	155,0	70.666
fensterbasiert Grid EGO	41.294,6	546.257	127.463,3	1.781.490	21.789,7	246.402	28.063,7	355.736
fensterbasiert Grid	12.429,7	18.963.916	31.485,2	72.070.173	11.050,5	12.762.166	10.646,4	12.444.182
naiv Standard	37,7	16.232	199,2	80.852	772,2	335.731	799,7	347.916
naiv Grid	380,0	1.239.685	1.752,9	5.730.645	5.622,1	14.525.209	5.590,2	15.115.335

Tab. 6.3: Laufzeiten und Seitenzugriffe bei fünf Join-Dimensionen

schneller. Das liegt daran, dass bei einer großen Anzahl an Dimensionen und uniformer Verteilung der Daten relativ wenige Tupel innerhalb der sich in den Join-Dimensionen erstreckenden ϵ-Umgebung eines Referenztupels liegen. Somit werden die meisten Tupel bei der Join-Berechnung schon vor der Bildung eines relevanten Tupelpaares aussortiert bzw. übersprungen, wodurch sich die Ausführungszeit natürlich stark reduziert, da weit weniger Tupelpaare miteinander zu vergleichen sind. Bei den anderen Verteilungen, bei denen die Tupel dichter gedrängt liegen und sich daher auch bei einer höheren Anzahl an Join-Dimensionen mit größerer Wahrscheinlichkeit innerhalb der ϵ-Umgebung eines Referenztupels aufhalten, ist dies nicht in diesem Ausmaß der Fall. Hier ist das naive Verfahren deshalb auch deutlich langsamer als die fensterbasierte Methode mit Standard-Materialisierung und Epsilon-Grid-Ordnung.

Auffallend ist das sehr schlechte Abschneiden der Variante fensterbasiert/Grid/EGO, die im vorangegangenen Abschnitt bei der Standardeinstellung von zwei Join-Dimensionen noch die kürzesten Laufzeiten hatte. Bei fünf Dimensionen ist sie hingegen mit Abstand die langsamste. Dies ist auf den stark ansteigenden Aufwand für die Grid-Datenstruktur bei zunehmender Dimensionszahl zurückzuführen. Der Zugriff auf eine beliebige Seite der Datenstruktur, wie er im Scheduling-Algorithmus erfolgt, dauert hier umso länger, je mehr Zellen das mehrdimensionale Array besitzt, in dem die Referenzobjekte enthalten sind. Die Zahl der Zellen wächst mit kleiner werdenden Werten für die ϵ_i, insbesondere aber mit steigender Anzahl an Join-Dimensionen sehr stark an. Daher ist dieses Verfahren bei mehr als zwei Join-Dimensionen nicht mehr zu empfehlen. Untermauert wird die Feststellung, dass der Leistungseinbruch vor allem aufgrund des Zugriffs auf beliebige Seiten im Scheduling-Algorithmus zustande kommt, durch die Tatsache, dass dasselbe Verfahren ohne Epsilon-Grid-Ordnung und ohne diesen Algorithmus zwar weitaus mehr Seitenzugriffe benötigt, bezogen auf die Laufzeit aber dennoch um das Zwei- bis Vierfache schneller ist als mit EGO und Scheduling-Algorithmus. Eine Verbesserung der Situation und damit eine Erweiterung des Anwendbarkeitsbereichs des Ansatzes fensterbasiert/Grid/EGO kann eventuell mit den in Abschnitt 5.2.4 beschriebenen Optimierungsansätzen für die Grid-Materialisierungsvariante erreicht werden.

Die Abbildungen 6.7 bis 6.10 zeigen das Verhalten der vier interessantesten Verfahren für die gängigen Verteilungen der Eingabedaten und eine zwischen eins und fünf variie-

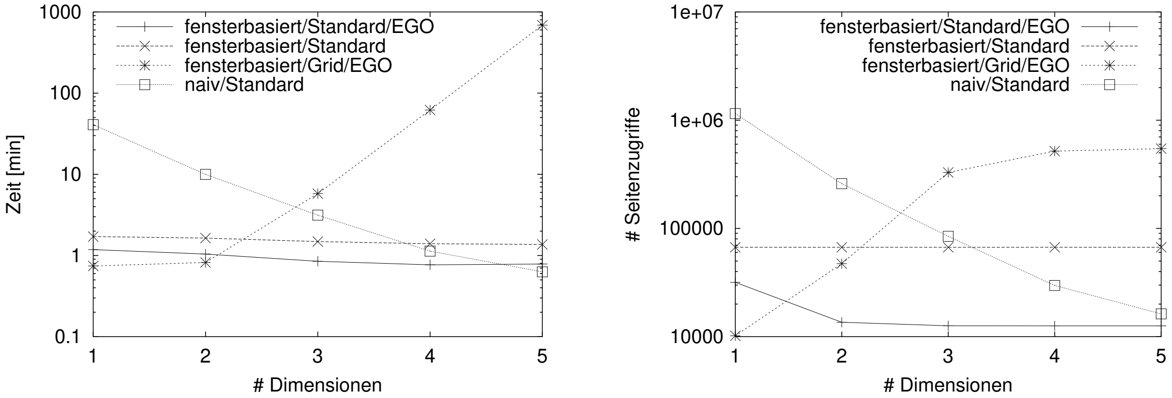


Abb. 6.7: Variation der Anzahl an Join-Dimensionen bei uniformer Verteilung

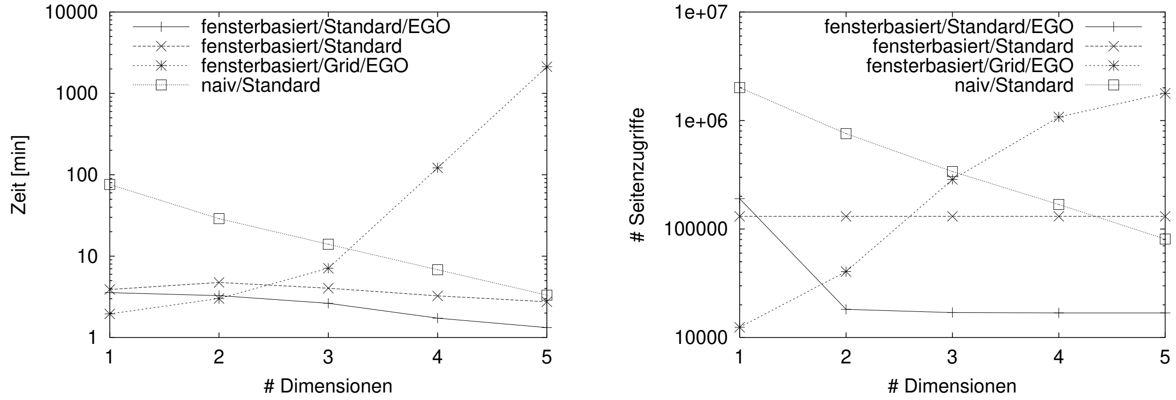


Abb. 6.8: Variation der Anzahl an Join-Dimensionen bei Normalverteilung

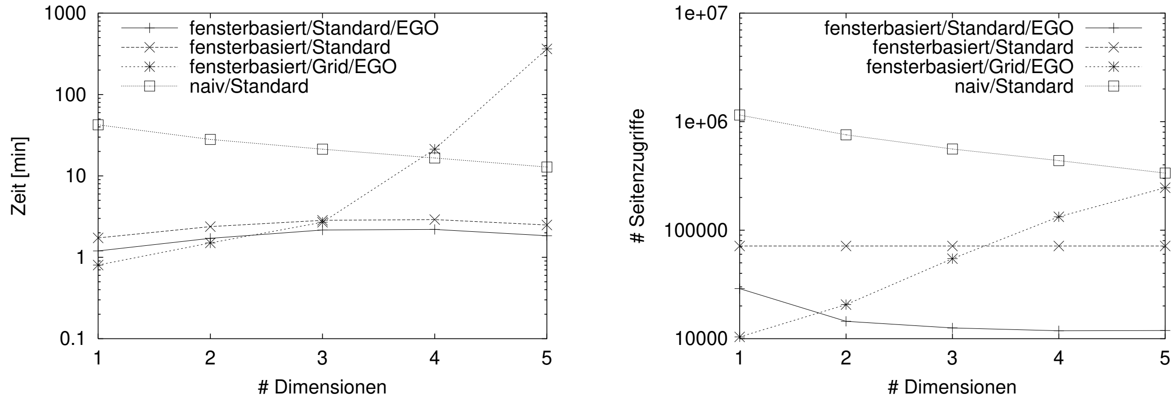


Abb. 6.9: Variation der Anzahl an Join-Dimensionen bei korrelierter Verteilung

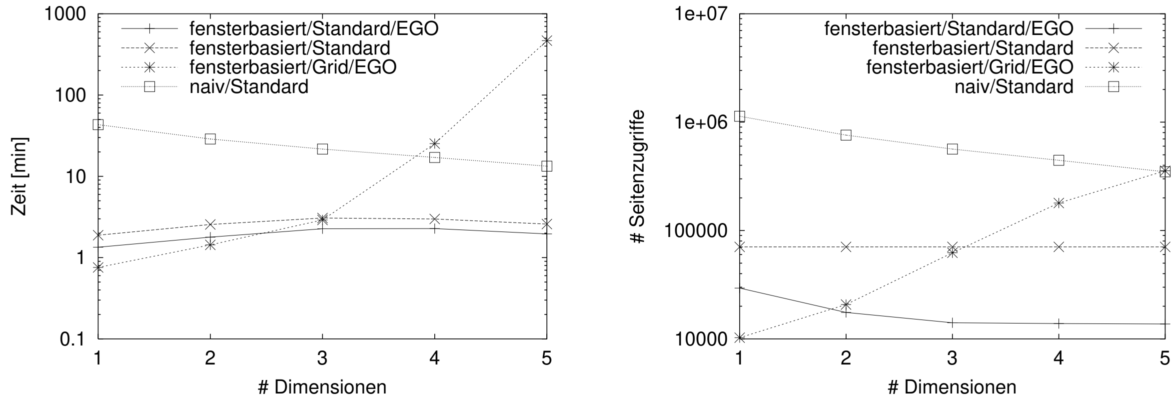


Abb. 6.10: Variation der Anzahl an Join-Dimensionen bei anti-korrelierter Verteilung

rende Anzahl an Join-Dimensionen. Die Hochwertachsen der Diagramme sind wiederum logarithmisch skaliert. Offensichtlich wird das Verfahren naiv/Standard mit steigender Anzahl an Join-Dimensionen immer besser. Sowohl die Laufzeit als auch die Anzahl an Seitenzugriffen nehmen kontinuierlich ab und zwar bei allen Verteilungen. Das hat seinen Grund wie oben beschrieben darin, dass bei steigender Dimensionszahl immer mehr Tupel schon vorab übersprungen werden können, da sie außerhalb der gerade gültigen ϵ-

Umgebung liegen. Deshalb fallen die Kurven mit wachsender Anzahl an Join-Dimensionen auch bei der uniformen Verteilung am stärksten ab, während der Kurvenverlauf bei der Normalverteilung schon flacher und bei der korrelierten und anti-korrelierten Verteilung am flachsten ist, da hier wegen der dichter gedrängten Tupel insgesamt weniger Datenelemente übersprungen werden können. Bei der Methode fensterbasiert/Grid/EGO kommt hingegen der Mehraufwand für die verwendete Datenstruktur zum Tragen, weshalb bei diesem Verfahren ein gegenteiliger Effekt zu beobachten ist. Sowohl die Laufzeit als auch die Anzahl an Seitenzugriffen nimmt mit steigender Anzahl an Join-Dimensionen stark zu. Die Laufzeit steigt aus den zu Tabelle 6.3 erläuterten Gründen sogar in etwa exponentiell mit der Dimensionszahl an. Da die verwendete mehrdimensionale Matrix bei mehr Join-Dimensionen weitaus mehr Zellen besitzt und jede Zelle die Tupel enthält, die in dem ϵ-Bereich der betreffenden Zelle enthalten sind, gibt es insgesamt auch mehr und dafür dünner besetzte Seiten. Dies erklärt die Zunahme der Seitenzugriffe, denn nun müssen zur Verarbeitung der gleichen Tupel mehr Seiten geladen werden.

Am gleichmäßigsten verhalten sich Laufzeit und Seitenzugriffe bei den Verfahren fensterbasiert/Standard/EGO und fensterbasiert/Standard. Sie lassen sich durch eine Veränderung der Anzahl an Join-Dimensionen kaum beeinflussen. Bei fensterbasiert/Standard ist die Anzahl an Seitenzugriffen sogar unabhängig von der Zahl der Join-Dimensionen, denn hier werden die Seiten der zu kombinierenden Teile der Datenfenster immer der Reihe nach geladen und verarbeitet. Darauf haben die Dimensionen keinen Einfluss. Mit Epsilon-Grid-Ordnung und Scheduling-Algorithmus ist bei allen Verteilungen eine merkliche Verringerung der Seitenzugriffe beim Übergang von einer auf zwei Dimensionen feststellbar. Bei einer weiteren Erhöhung der Anzahl an Join-Dimensionen ergibt sich aber keine große Veränderung mehr. Insgesamt ist die Variante mit Epsilon-Grid-Ordnung und Scheduling-Algorithmus der Variante ohne diese Erweiterung wiederum so gut wie immer überlegen, so dass auch hier festgehalten werden kann, dass sich der Einsatz der Ordnung und des Scheduling-Algorithmus durchaus lohnt. Zusammenfassend lässt sich feststellen, dass für ein bis zwei Join-Dimensionen das Verfahren fensterbasiert/Grid/EGO die beste Wahl darstellt. Bei mehr als zwei Join-Dimensionen ist hingegen fensterbasiert/Standard/EGO vorzuziehen.

In Abbildung 6.11 sind zum Vergleich noch einmal alle sechs Verfahren bei uniformer Verteilung und variierender Anzahl an Join-Dimensionen dargestellt. Auch hier zeigt sich, dass die beiden Methoden fensterbasiert/Grid und naiv/Grid sehr schlecht abschneiden. Dies bestätigt, dass die Grid-Materialisierungsvariante nur dann Sinn macht, wenn die zu materialisierenden Tupel vor der Einfügung in die Gitter-Datenstruktur nach Epsilon-Grid-Ordnung sortiert werden und die von der Grid-Variante implizit durchgeführte Sortierung des gesamten materialisierten Datenbestands gemäß Epsilon-Grid-Ordnung erwünscht bzw. nötig ist sowie im Weiteren auch entsprechend ausgenutzt werden kann, z. B. durch den Einsatz des Scheduling-Algorithmus.

Abbildung 6.12 stellt den Verlauf der Ergebnisgröße in Abhängigkeit von der Anzahl an Join-Dimensionen dar. Dabei sind zwei gegenläufige Entwicklungen zu berücksichtigen. Einerseits steigt bei einer größeren Zahl an Join-Dimensionen die Wahrscheinlichkeit, dass zwei Tupelpaare unvergleichbar sind. Dies führt zu einer Erhöhung der Ergebnisgröße, da es insgesamt seltener vorkommt, dass ein Tupelpaar ein anderes dominiert und damit aus

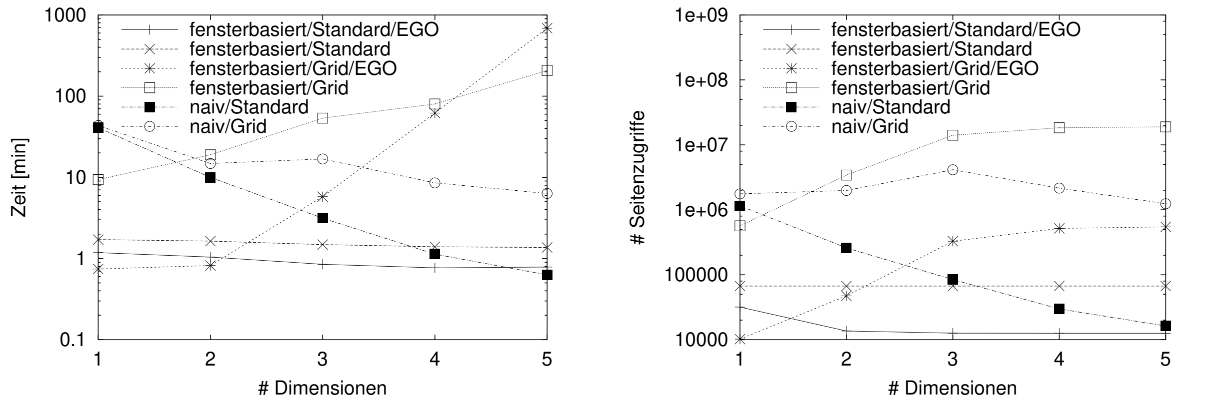


Abb. 6.11: Vergleich aller Verfahren bei variierender Anzahl an Join-Dimensionen und uniformer Verteilung

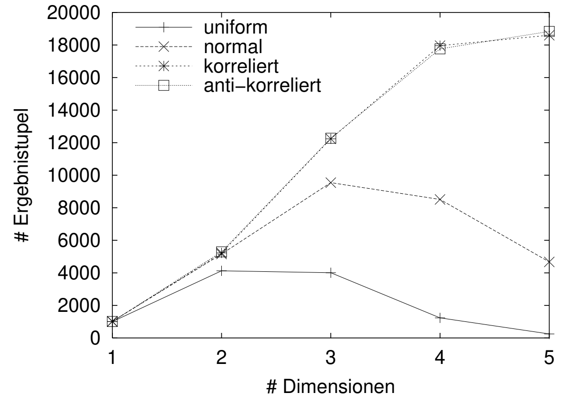


Abb. 6.12: Anzahl der Ergebnistupel bei variierender Anzahl an Join-Dimensionen

dem Ergebnis eliminiert. Andererseits wächst mit der Zahl der Join-Dimensionen auch die Wahrscheinlichkeit, dass ein Tupel außerhalb der auf die Join-Dimensionen bezogenen ϵ-Umgebung eines Referenztupels, mit dem es ein Paar bilden soll, liegt. Schließlich genügt es bereits, wenn der vorgegebene Maximalabstand ϵ_i bei zwei Tupeln in nur einer einzigen Join-Dimension i überschritten wird, um dieses Tupelpaar für das Ergebnis uninteressant zu machen. Damit ergeben sich insgesamt weniger Ergebnispaare und die Ergebnisgröße sinkt. Wie die Abbildung zeigt, ist bei korrelierter und anti-korrelierter Verteilung der Eingabedaten der erste Effekt dominant und die Ergebnisgröße wächst durchgehend mit der Anzahl an Join-Dimensionen. Die Ursache hierfür liegt in der Tatsache, dass aufeinander folgende Eingabetupel bei diesen Verteilungen in allen relevanten Join-Dimensionen relativ nah beieinander liegen, wodurch sich der zweite, die Eingabegröße reduzierende Effekt stark in Grenzen hält. Bei der uniformen und der Normalverteilung liegen die Tupel hingegen in den einzelnen Dimensionen wegen der fehlenden Korrelation bzw. Anti-Korrelation im Allgemeinen weiter auseinander, so dass hier der die Eingabegröße verringernde Effekt mehr Gewicht hat. Bei bis zu zwei bzw. drei Join-Dimensionen überwiegt hier der erste Effekt, die Ergebnisgröße wächst. Danach nimmt der zweite Effekt überhand und die Ergebnisgröße schrumpft wieder. Dabei ergibt die uniforme Verteilung insgesamt weniger Ergebnispaare als die Normalverteilung, weil sich bei ihr die Eingabetupel weniger dicht drängen und daher innerhalb der jeweils geltenden ϵ-Umgebung eines Tupels in der Regel weniger Tupel vorliegen, mit denen sich ein Ergebnispaar bilden lässt.

Variation der ϵ-Umgebung

Im Folgenden wird das Verhalten der Verfahren bei variierender ϵ-Umgebung betrachtet. Die restlichen Untersuchungen dieses Abschnitts beschränken sich dabei auf die uniforme Verteilung. Für die anderen Verteilungen ergibt sich jeweils ein sehr ähnliches Bild. Tabelle 6.4 zeigt die Laufzeiten und die Anzahl an Seitenzugriffen für alle sechs Verfahren bei uniformer Verteilung und $\epsilon_i = 0.5$ für alle Join-Dimensionen. Da hier wieder die standardmäßig eingestellten zwei Join-Dimensionen verwendet wurden, erweist sich die fensterbasierte Methode mit Grid-Materialisierung und Epsilon-Grid-Ordnung als die beste Wahl. Sowohl die Laufzeit als auch die Anzahl an Seitenzugriffen sind bei diesem Verfahren im Vergleich zu den anderen am geringsten. Auf Platz zwei folgt dasselbe Verfahren mit Standard-Materialisierung. Diese Methode zeigt selbst ohne Epsilon-Grid-Ordnung und Scheduling-Algorithmus noch akzeptable Werte. Der Ansatz fensterbasiert/Grid und die naiven Algorithmen fallen hingegen deutlich ab. Das naive Verfahren ist hier mit Grid-Materialisierung sogar etwas besser als mit Standard-Materialisierung, was auf die relativ große ϵ-Umgebung zurückzuführen ist.

Abbildung 6.13 stellt den Verlauf der Laufzeiten und Seitenzugriffe bei uniformer Verteilung und variierender ϵ-Umgebung dar. Hier ist die Hochwertachse nur bei dem Diagramm für die Seitenzugriffe logarithmisch skaliert. Bei allen gezeigten Verfahren nimmt die Laufzeit bei größer werdenden ϵ_i zu Beginn immer stärker, später immer weniger stark zu. Die Zunahme erklärt sich bei den fensterbasierten Methoden aus größer werdenden Fenstern, deren Verarbeitung mehr Zeit in Anspruch nimmt, und bei der naiven Methode aus der Tatsache, dass bei der Join-Berechnung immer weniger Tupel übersprungen werden können, da immer mehr von ihnen innerhalb der relevanten ϵ-Umgebung des jeweiligen Referenztupels liegen. Die Verringerung der Zunahme bei größeren Werten für die ϵ_i ist darauf zurückzuführen, dass viele Tupelpaare bei derart großen ϵ-Umgebungen bereits früh von besseren Paaren dominiert und eliminiert werden. Dadurch kann die

	uniform	
	Zeit [s]	# Seiten
fensterbasiert Standard EGO	1.525,9	60.070
fensterbasiert Standard	1.818,7	169.769
fensterbasiert Grid EGO	1.481,4	13.603
fensterbasiert Grid	9.726,1	5.137.364
naiv Standard	5.567,3	2.362.500
naiv Grid	4.995,7	2.301.864

Tab. 6.4: Laufzeiten und Seitenzugriffe bei $\epsilon_i = 0.5$

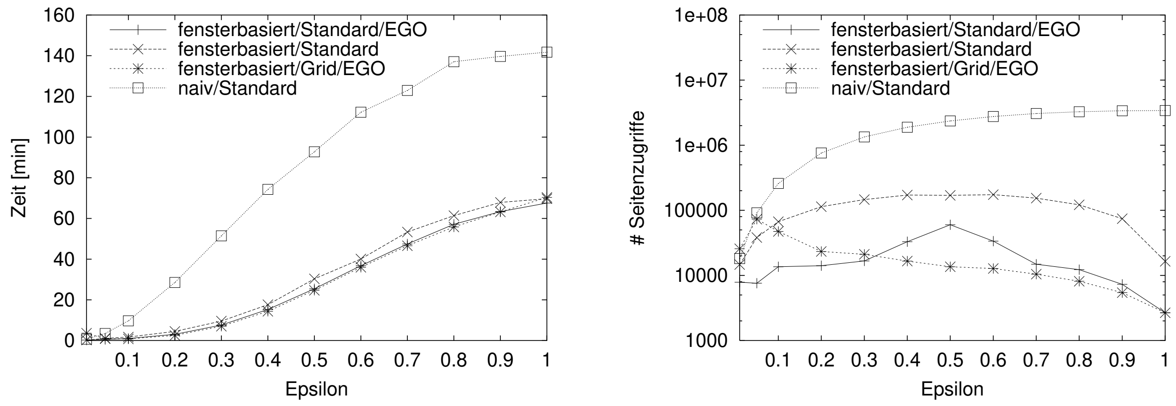


Abb. 6.13: Variation der ϵ-Umgebung bei uniformer Verteilung

naive Methode die innere Schleife des Algorithmus früher abbrechen und beim fensterbasierten Verfahren enthält der Puffer O für potenzielle Ergebnispaare im Mittel weniger Paare, so dass auch hier die innere Schleife des Algorithmus weniger Durchläufe benötigt. Insgesamt bestätigt das Diagramm mit den Laufzeiten den Eindruck aus Tabelle 6.4 für alle verwendeten ϵ-Werte. Die fensterbasierten Methoden liegen hier durchgehend dicht beisammen und sind der naiven Methode weit überlegen. Während die fensterbasierten Verfahren mit Epsilon-Grid-Ordnung und Scheduling-Algorithmus unabhängig von der verwendeten Materialisierungsvariante fast gleichauf liegen, befinden sich die Laufzeiten der Variante mit Standard-Materialisierung und ohne die Erweiterung um die Ordnung und den Scheduling-Algorithmus bereits auf einem merklich höheren Niveau.

Auch bei den Seitenzugriffen weist fensterbasiert/Grid/EGO zumindest ab einer gewissen Größe der ϵ-Umgebung die besten Werte auf. Die Anzahl der Seitenzugriffe nimmt bei diesem Verfahren mit größer werdenden ϵ_i ab, da dann die Grid-Datenstruktur immer weniger Zellen und damit auch weniger Seiten enthält, die dafür besser gefüllt sind. Somit müssen zur Verarbeitung der gleichen Menge an Tupeln insgesamt weniger Seiten geladen werden. Die beiden fensterbasierten Verfahren mit Standard-Materialisierung laden bei wachsenden Werten für die ϵ_i zunächst mehr, im weiteren Verlauf wieder weniger Seiten. Der Zuwachs ist dabei auf die Vergrößerung der Datenfenster bei wachsenden ϵ_i zurückzuführen, so dass bei der mitunter wiederholten Verarbeitung von Teilen der Fenster insgesamt mehr Seiten geladen werden müssen. Die Verringerung im weiteren Verlauf ist eine Folge davon, dass die Datenfenster hier schon so groß sind, dass ältere Teile der Fenster nur noch selten mehrmals geladen werden müssen. Bei $\epsilon_i = 1.0$ enthalten die initialen Datenfenster schließlich bereits den gesamten Datenbestand und müssen nur noch einmal komplett bearbeitet werden. Dadurch entfällt das mehrfache Laden einzelner alter Fensterteile komplett. Beim naiven Verfahren steigt die Anzahl an Seitenzugriffen hingegen kontinuierlich an.

Abbildung 6.14 zeigt den Verlauf der Anzahl an Ergebnistupeln bei uniformer Verteilung und verschiedenen Werten für die ϵ_i. Die Ergebnisgröße steigt mit wachsenden ϵ_i zunächst sehr stark und später immer schwächer an. Eine Vergrößerung der ϵ-Werte und damit der ϵ-Umgebung bedeutet natürlich eine Erweiterung des Suchraums, so dass sich dann insgesamt mehr Paare von Tupeln finden lassen, die bezüglich der verwendeten Vergleichsord-

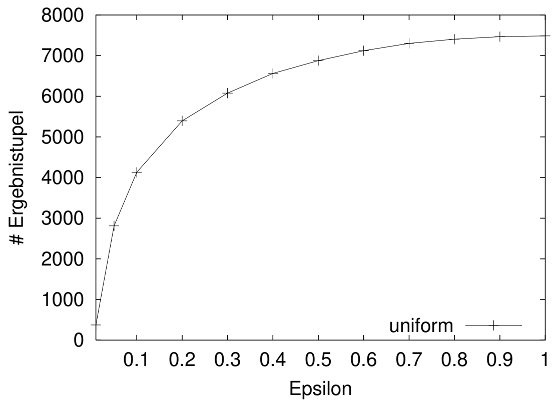


Abb. 6.14: Anzahl der Ergebnistupel bei variierender ϵ-Umgebung

nung am besten zusammenpassen. Je größer die ϵ-Umgebung wird, desto wahrscheinlicher ist es aber, dass die nun zusätzlich in Betracht zu ziehenden Paare von bereits bei kleineren Umgebungen relevanten Tupelpaaren dominiert werden. Deshalb nimmt der Zuwachs an Ergebnistupeln mit größer werdenden ϵ-Werten ab.

Variation der Seitenkapazität

In diesem Abschnitt wird der Einfluss unterschiedlicher Kapazitäten für die Seiten des Hintergrundspeichers auf die einzelnen Verfahren zur Berechnung von besten Zuordnungen untersucht und bewertet. Dazu zeigt Tabelle 6.5 die Laufzeiten und die Anzahl an Seitenzugriffen der Berechnungsverfahren bei uniformer Verteilung der Eingabedaten und einer Seitenkapazität von 64 Kilobyte. Am schnellsten ist unter diesen Bedingungen wiederum das Verfahren fensterbasiert/Grid/EGO, gefolgt von fensterbasiert/Standard/EGO. Bei der Anzahl der Seitenzugriffe tauschen diese beiden Methoden die Plätze. Das drittbeste Verfahren ist fensterbasiert/Standard, das zwar ein achtbares Ergebnis erzielt, aber an die beiden erstgenannten Ansätze nicht heranreicht. Auch hier zeigt der Einsatz der Epsilon-Grid-Ordnung und des Scheduling-Algorithmus also deutliche Wirkung. Das Ver-

	uniform	
	Zeit [s]	# Seiten
fensterbasiert Standard EGO	67,8	7.336
fensterbasiert Standard	196,8	64.599
fensterbasiert Grid EGO	52,6	47.331
fensterbasiert Grid	1214,6	3.429.327
naiv Standard	556,4	28.610
naiv Grid	871,9	1.983.216

Tab. 6.5: Laufzeiten und Seitenzugriffe bei einer Seitenkapazität von 64 KB

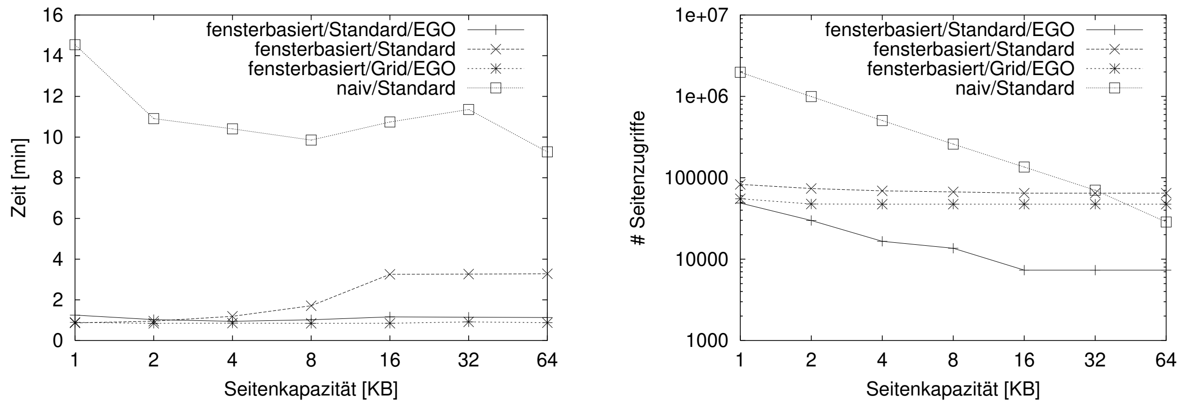


Abb. 6.15: Variation der Seitenkapazität bei uniformer Verteilung

fahren fensterbasiert/Grid und der naive Algorithmus sind im Vergleich zu den anderen Berechnungsmethoden weit abgeschlagen.

In Abbildung 6.15 ist der Verlauf der Laufzeiten in Minuten und der Anzahl an Seitenzugriffen für die vier wichtigsten Methoden bei uniformer Verteilung der Eingabedaten und variierender Seitenkapazität dargestellt. Hier ist nur die Hochwertachse des Diagramms für die Seitenzugriffe logarithmisch skaliert. Die Seitenkapazität wird sukzessive von einem auf 64 Kilobyte erhöht und dabei schrittweise verdoppelt. Zur besseren Übersicht ist daher bei beiden Diagrammen die Rechtswertachse logarithmisch skaliert. Die beiden fensterbasierten Varianten mit Epsilon-Grid-Ordnung sind insgesamt die schnellsten, wobei fensterbasiert/Grid/EGO noch ein wenig schneller ist als fensterbasiert/Standard/EGO. Diese beiden Verfahren lassen sich durch die Veränderung der Seitenkapazität auch nicht beeinträchtigen. Ihre Laufzeit ist davon weitgehend unabhängig. Anders bei fensterbasiert/Standard. Dieses Verfahren wird bei zu großen Seiten langsamer, weil es wegen der Standard-Materialisierung die Hintergrundspeicherseiten immer ganz voll schreibt und dann die ganze Seite laden muss, selbst wenn nur ein kleiner Teil ihres Inhalts benötigt wird. Bei großen Seiten werden deshalb oft viele Tupel unnötig geladen. Ohne die effizientere Strategie des Scheduling-Algorithmus schlägt sich dies in den Laufzeiten nieder. Das naive Verfahren kommt an die fensterbasierten nicht annähernd heran, kann aber mitunter von größeren Seiten profitieren. Zwar müssen auch hier mehr Daten pro Seite geladen werden, die eventuell gar nicht relevant sind, dafür ist aber die Wahrscheinlichkeit größer, dass für die innere Schleife des Algorithmus nur die erste Seite einmal geladen und für mehrere Schleifendurchläufe wiederverwendet werden kann. Dies ist dann möglich, wenn entweder alle Tupel einer Relation in einer einzigen Seite Platz finden, dann muss diese Seite nur einmal geladen werden, oder wenn die innere Schleife jeweils so frühzeitig abgebrochen werden kann, dass die zweite Seite noch nicht geladen werden musste. Dann lässt sich die bereits geladene erste Seite für den nächsten Schleifendurchlauf wieder verwenden.

Aus demselben Grund zeigt das naive Verfahren auch eine sinkende Zahl von Seitenzugriffen bei steigender Seitenkapazität. Die Methoden fensterbasiert/Standard und fensterbasiert/Grid/EGO zeigen sich bezogen auf die Seitenzugriffe durch veränderte Seitenkapazitäten hingegen kaum beeinflusst. Bei fensterbasiert/Standard/EGO können die Seitenzugriffe allerdings durch größere Seiten reduziert werden. Hier profitiert der Scheduling-

Algorithmus merklich von der verringerten Gesamtzahl der Seiten bei höherer Seitenkapazität. Bei Verwendung der Grid-Materialisierung kommt dieser Effekt wegen der Verteilung der materialisierten Tupel auf die Seiten der zugehörigen Gitterzellen dagegen nicht zum Tragen.

Variation der Hauptspeicherkapazität

Als letzter Parameter wird nun noch die Kapazität des Hauptspeicherpuffers betrachtet. Diese Kapazität wird in Seiten pro Eingaberelation angegeben. Damit die fensterbasierten Verfahren unter Benutzung der Epsilon-Grid-Ordnung funktionieren können, müssen für jede Eingaberelation mindestens zwei Seiten im Hauptspeicher Platz finden. Eine Seite muss bei der Materialisierung die zu materialisierenden und nach Epsilon-Grid-Ordnung sortierten Tupel aufnehmen. Die andere ist nötig, um beim Mischen mit bereits materialisierten Datenbeständen bei der Standard-Materialisierung die aktuelle Seite der schon in einem früheren Schritt auf den Hintergrundspeicher geschriebenen Tupel für den Mischvorgang laden zu können bzw. bei der Grid-Materialisierung die Seite in den Speicher zu laden, in welche die nächsten Tupel zu schreiben sind. Die folgenden Betrachtungen beschränken sich wiederum auf uniform verteilte Eingabedaten und außerdem auf die fensterbasierten Berechnungsverfahren. Auf die naiven Verfahren hat eine Veränderung der Hauptspeicherkapazität keinen Einfluss, da diese Methoden immer die gleiche Anzahl von zwei Seiten im Hauptspeicher halten, eine Seite für jede der beiden inneren Schleifen des Algorithmus. Tabelle 6.6 gibt einen Überblick über Laufzeiten und Seitenzugriffe der fensterbasierten Verfahren bei uniformer Verteilung und einer Hauptspeicherkapazität von acht Seiten pro Eingaberelation. Es ergibt sich ein ähnliches Bild wie im vorangegangenen Abschnitt zur Seitenkapazität. Wieder ist fensterbasiert/Grid/EGO am schnellsten, gefolgt von fensterbasiert/Standard/EGO. Letztgenannte Methode weist auch wieder die geringste Zahl an Seitenzugriffen auf. Die fensterbasierte Variante mit Standard-Materialisierung aber ohne Epsilon-Grid-Ordnung und Scheduling-Algorithmus kann mit diesen beiden Ansätzen nicht mithalten, erreicht aber dennoch akzeptable Werte. Dasselbe Verfahren mit Grid-Materialisierung fällt dagegen deutlich ab.

Abbildung 6.16 zeigt die Laufzeiten in Sekunden und die Anzahl an Seitenzugriffen für die fensterbasierten Verfahren mit Epsilon-Grid-Ordnung und Scheduling-Algorithmus bei uniformer Verteilung der Eingabedaten und unterschiedlichen Hauptspeicherkapazitäten.

	uniform	
	Zeit [s]	# Seiten
fensterbasiert Standard EGO	60,9	13.470
fensterbasiert Standard	99,7	66.980
fensterbasiert Grid EGO	47,9	42.848
fensterbasiert Grid	1195,4	3.429.327

Tab. 6.6: Laufzeiten und Seitenzugriffe bei einer Hauptspeicherkapazität von acht Seiten

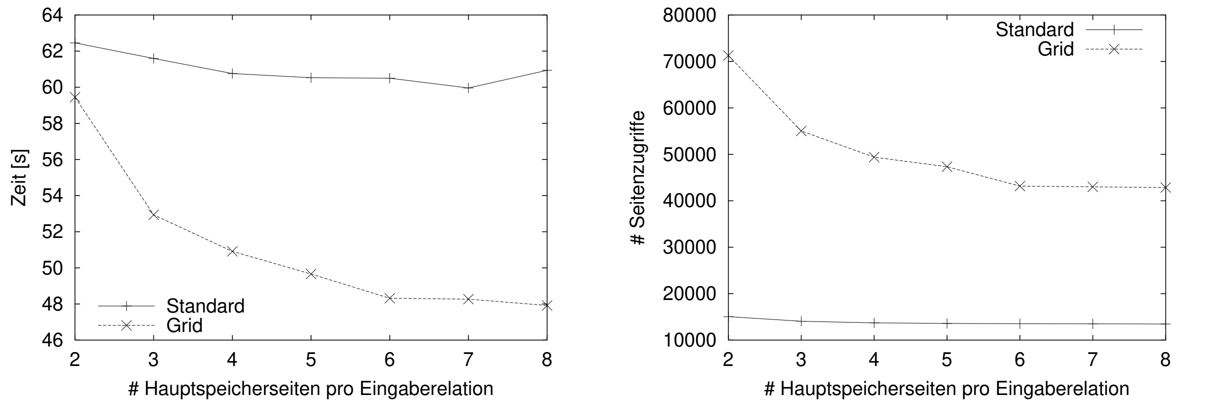


Abb. 6.16: Variation der Hauptspeicherkapazität für das fensterbasierte Verfahren mit Epsilon-Grid-Ordnung bei uniformer Verteilung

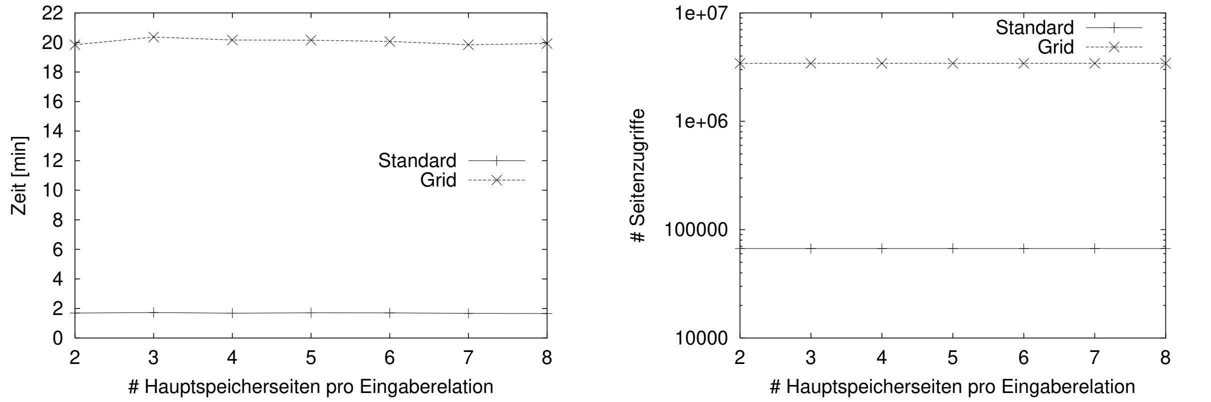


Abb. 6.17: Variation der Hauptspeicherkapazität für das fensterbasierte Verfahren ohne Epsilon-Grid-Ordnung bei uniformer Verteilung

Die Grid-Materialisierungsvariante ist bei allen Pufferkapazitäten die schnellere Variante. Während sich die Methode bei Standard-Materialisierung kaum von einer Veränderung der Kapazität des Hauptspeicherpuffers beeinflussen lässt, können die Ausführungszeit und die Seitenzugriffe bei Grid-Materialisierung durch eine Vergrößerung des Puffers reduziert werden. Da die Grid-Variante im Allgemeinen wegen der dünneren Besetzung mit Tupeln insgesamt mehr Seiten verwalten muss als die Standard-Variante, profitiert sie relativ stark von der Möglichkeit, mehr Seiten gleichzeitig im Hauptspeicher halten zu können. Dadurch muss das Verfahren seltener auf den Crabstep-Modus umschalten. Das verringert die Anzahl mehrfach zu ladender Seiten der linken Eingabe und damit die Seitenzugriffe insgesamt. Weiterhin reduziert sich dadurch auch die Laufzeit, da im Scheduling-Algorithmus seltener der bei der Grid-Variante aufwendigere Zugriff auf eine beliebige Seite der Datenstruktur nötig ist. Dieser erfolgt immer am Ende der Ausführung des Crabstep-Modus beim Rücksprung zu der als letztes vor der Umschaltung auf diesen Modus bearbeiteten Seite der linken Eingaberelation.

In Abbildung 6.17 ist dieselbe Situation für die fensterbasierten Verfahren ohne Epsilon-Grid-Ordnung und Scheduling-Algorithmus dargestellt. Hier ist die Grid-Variante wegen der fehlenden Sortierung der zu materialisierenden Tupel gemäß Epsilon-Grid-Ordnung und des deaktivierten Scheduling-Algorithmus für alle untersuchten Hauptspeicherkapazitäten deutlich langsamer als die Standard-Variante, die mit diesen Bedingungen weitaus besser zurecht kommt. Die Laufzeiten sind bei beiden Materialisierungsvarianten von der verfügbaren Hauptspeicherkapazität weitgehend unabhängig, da diese Methoden ohnehin immer nur die jeweils gerade benötigte Seite einer jeden Eingaberelation im Hauptspeicher halten. Aus diesem Grund bleiben auch die Seitenzugriffe beider Verfahren vollkommen unbeeinflusst von der Kapazität des Hauptspeicherpuffers. Wegen der größeren Zahl der zu verwaltenden Seiten liegen sie aber bei der Grid-Materialisierung auf einem deutlich höheren Niveau als bei der Standard-Materialisierung.

6.2.2 Ergebnistupel pro Zeit

Neben Gesamtlaufzeiten und Seitenzugriffen ist vor allem auch interessant, wie sich die Methoden zur Berechnung von besten Zuordnungen von Objekten zweier Datenströme bei der kontinuierlichen Generierung von Ergebnistupeln verhalten. Vor dem Hintergrund der Anwendung auf Datenströmen und der gewünschten Pipeline-Fähigkeit ist dabei insbesondere die Geschwindigkeit, mit der die einzelnen Ergebnistupel auf den Ausgabedatenstrom geschickt werden können, wichtig. Die nachfolgenden Untersuchungen beleuchten diesen Aspekt näher. Dabei sei vorab noch einmal darauf hingewiesen, dass alle hier behandelten naiven Verfahren grundsätzlich blockierend und damit nicht pipelinefähig sind. Sie müssen vor Beginn der Berechnung die komplette rechte Eingaberelation materialisieren. Daher kann auch das erste Ergebnistupel erst erzeugt und weitergegeben werden, wenn die rechte Eingabe bereits vollständig gelesen worden ist. Damit ist klar, dass diese Verfahren bei unendlichen Datenströmen nicht einsetzbar sind. Für die mit endlichen Datenbeständen durchgeführten Messungen dieses Abschnitts können sie allerdings verwendet werden. Sie dienen jedoch lediglich als Vergleichsinstanz, um die Leistungswerte der fensterbasierten Verfahren besser einordnen und beurteilen zu können, und stellen für die praktische Anwendung bei Berechnungen auf Datenströmen keine brauchbaren Alternativen dar.

Die Abbildungen 6.18 und 6.19 zeigen das Verhalten der fensterbasierten Methode mit Epsilon-Grid-Ordnung und Scheduling-Algorithmus bzw. das Verhalten der naiven Methode für verschiedene Verteilungen der Eingabedaten bei Standard- und bei Grid-Materialisierung. Dabei ist auf der Rechtswertachse jeweils die Anzahl der bereits propagierten Ergebnistupel und auf der Hochwertachse die bis dahin vergangene Zeit seit dem Start der Berechnung aufgetragen. Es ist zu beachten, dass die Zeitachse beim fensterbasierten Verfahren die Zeit in Sekunden angibt, während sie beim naiven Verfahren in Minuten angegeben ist. Da die Verteilungen Einfluss auf die Ergebnisgröße haben, weisen die Kurven bezogen auf die Rechtswertachse unterschiedliche Längen auf, je nachdem, wie viele Ergebnistupel insgesamt erzeugt wurden. Aus den Abbildungen ist ersichtlich, dass bei der korrelierten und der anti-korrelierten Verteilung die meisten Ergebnistupel entstehen, da hier alle Attributwerte benachbarter Tupel relativ dicht beisammenliegen, wodurch mehr unvergleichbare beste Paare entstehen. Bei der Normalverteilung ist die Streuung bereits etwas größer, so dass hier die Ergebnisgröße kleiner wird. Dies setzt sich bei der uniformen Verteilung weiter fort. Bei allen Verfahren zeigt sich, dass sie die Ergebnistupel

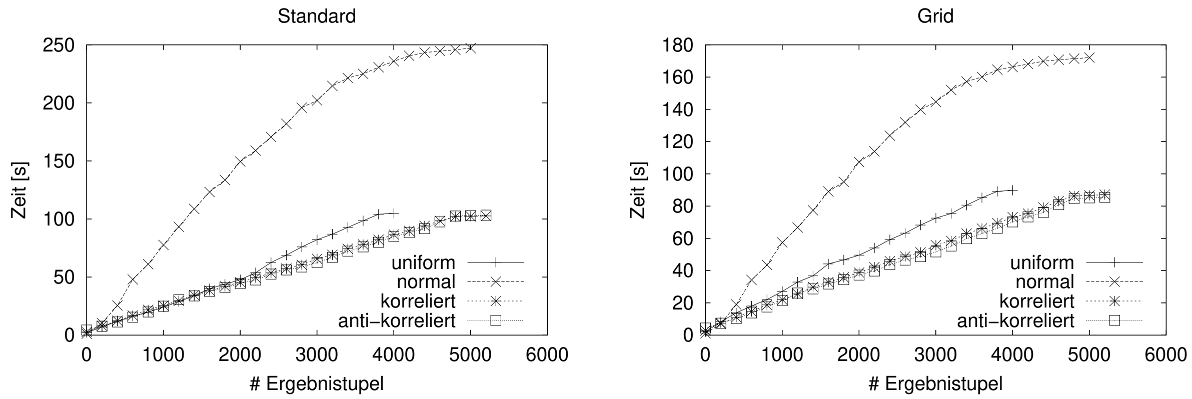


Abb. 6.18: Ergebnistupel pro Zeit beim fensterbasierten Verfahren mit EGO und verschiedenen Verteilungen

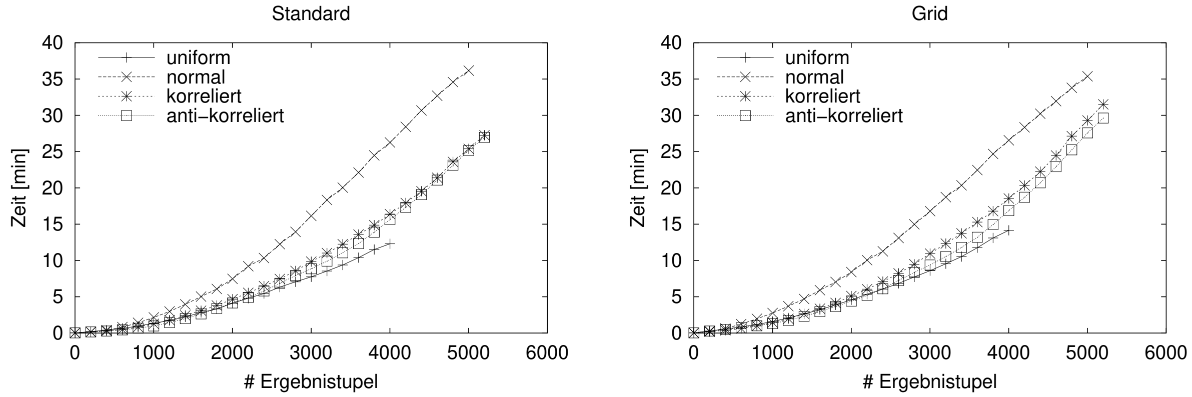


Abb. 6.19: Ergebnistupel pro Zeit beim naiven Verfahren und verschiedenen Verteilungen

bei Normalverteilung der Eingabedaten langsamer liefern als bei den ähnliche Kurvenverläufe aufweisenden anderen Verteilungen. Dies ist mit der hohen Konzentration der Attributwerte der Double-Attribute in der Mitte des Wertebereichs bei der verwendeten Normalverteilung zu erklären. Dadurch füllen sich im Falle des fensterbasierten Algorithmus die Datenfenster an dieser Stelle sehr stark, weshalb ihre Verarbeitung mehr Zeit in Anspruch nimmt. Weiterhin fällt auf, dass bei der fensterbasierten Methode die Grid-Variante unter den herrschenden Bedingungen von z. B. zwei Join-Dimensionen deutlich schneller ist als die Standard-Variante. Insgesamt ist aus Abbildung 6.18 ersichtlich, dass die fensterbasierten Verfahren die Ergebnistupel kontinuierlich zum folgenden Operator oder zur Ausgabe propagieren. Die Pipeline-Fähigkeit ist bei diesen Methoden also über die Theorie hinaus auch praktisch gegeben. Zwar erzeugen die naiven Verfahren bei den durchgeführten Messungen auch einen kontinuierlichen Ausgabedatenstrom, unterliegen aber aufgrund ihrer blockierenden Architektur den zu Beginn dieses Kapitels beschriebenen Einschränkungen.

Weiterhin zeigen die Messungen, dass der fensterbasierte Algorithmus mit Epsilon-Grid-Ordnung und Standard-Materialisierung bei ansonsten den Standardeinstellungen aus Tabelle 6.1 entsprechenden Bedingungen im Mittel einen Durchsatz von etwa 39,2 Tupeln pro Sekunde bei uniformer Verteilung der Eingabedaten erreicht. Bei korrelierter und anti-korrelierter Verteilung sind es sogar 51,0, bei Normalverteilung 20,8 Tupel pro Sekunde. Bei Verwendung der Grid-Materialisierung ergeben sich entsprechend Werte von 45,8, 61,0 und 29,9 Tupeln pro Sekunde. Der naive Algorithmus kommt bei Standard-Materialisierung und uniform verteilten Eingabedaten im Mittel auf einen Durchsatz von ungefähr 5,4 Tupeln pro Sekunde. Bei korreliert bzw. anti-korreliert verteilten Daten erreicht er nur knapp 3,2, bei normalverteilten Eingaben 2,3 Tupel pro Sekunde. Verwendet man die Grid-Materialisierung, liegt das naive Verfahren bei Werten von 4,7, 2,8 und 2,4 Tupeln pro Sekunde.

Aus den Abbildungen ist noch ein anderes interessantes Detail ersichtlich. Bei den fensterbasierten Verfahren weisen alle Kurvenverläufe eine Rechtskrümmung auf, die ausdrückt, dass die Berechnung zum Ende hin schneller wird. Dies erklärt sich dadurch, dass dann von den erschöpften Eingaben keine neuen Vergleichstupel mehr nachgeliefert werden und deshalb nur noch der Inhalt des Puffers O für potenzielle Ergebnistupel auszugeben ist. Da kein darin enthaltenes Tupelpaar mehr eliminiert werden kann – schließlich kommen von den Eingaben keine neuen Tupel mehr nach – ist der gesamte verbleibende Inhalt des Puffers Teil des Endergebnisses. Die Kurvenverläufe der naiven Verfahren zeigen hingegen alle eine Linkskrümmung, d. h. diese Methoden sind am Anfang relativ schnell, werden im Laufe der Berechnung aber immer langsamer und die Zeit zwischen der Propagierung zweier Ergebnistupel wird immer länger. Das liegt an den drei verschachtelten Schleifen über die Eingaberelationen. Je weiter die äußere Schleife über die linke Eingaberelation fortgeschritten ist, desto länger müssen die beiden inneren Schleifen aufgrund der Vorsortierung der Relationen nach aufsteigenden Werten eines Attributs über die rechte Relation laufen, bevor eventuell eine dominierende Tupelkombination gefunden und die innere Schleife vorzeitig abgebrochen werden kann. Daher sind die naiven Verfahren zwar bei kleinen Eingabegrößen durchaus schnell, verlieren aber stark an Effizienz und Geschwindigkeit sobald die zu verarbeitenden Relationen größer werden.

Als nächstes sollen die Verfahren bei unterschiedlicher Granularität der Eingabedaten untersucht werden. Dazu wurden die Messungen der Ergebnistupel pro Zeit mit Eingabedaten unterschiedlicher Genauigkeit durchgeführt. Die Werte der Double-Attribute der verwendeten Relationen besitzen zwischen einer und fünf Nachkommastellen (NKS). Abbildung 6.20 zeigt die Ergebnisse des fensterbasierten Verfahrens mit Epsilon-Grid-Ordnung und Scheduling-Algorithmus bei uniformer Verteilung der Eingabedaten für die Standard- und die Grid-Materialisierungsvariante. Offensichtlich beeinträchtigt die Granularität der Eingabedaten zumindest ab einer Genauigkeit von zwei Nachkommastellen die kontinuierliche Erzeugung von Ergebnistupeln nicht. Nur bei der geringsten Granularität von einer Nachkommastelle ist eine Treppenbildung erkennbar. Diese erklärt sich aus der Tatsache, dass hier die Genauigkeit der Double-Werte der Eingaben mit den verwendeten ϵ-Werten von $\epsilon_i = 0.1$ zusammenfällt. Damit entspricht der Wert 0.1 für die Höhe der eingesetzten Datenfenster genau dem kleinstmöglichen Abstand zweier Eingabewerte. Dies führt dazu, dass alle Datenfenster zu jeder Zeit nur solche Tupel enthalten, die entweder genau auf der Obergrenze oder der Untergrenze des Fensters liegen. Dazwischen existiert nichts. Bei der Aktualisierung der Datenfenster wird daher immer die Obergrenze

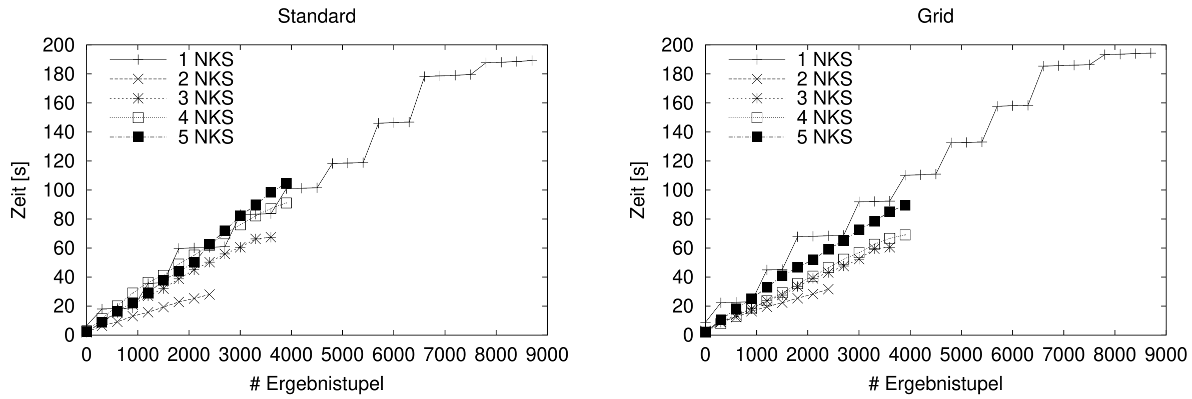


Abb. 6.20: Ergebnistupel pro Zeit beim fensterbasierten Verfahren mit EGO und unterschiedlicher Granularität der Eingabedaten bei uniformer Verteilung

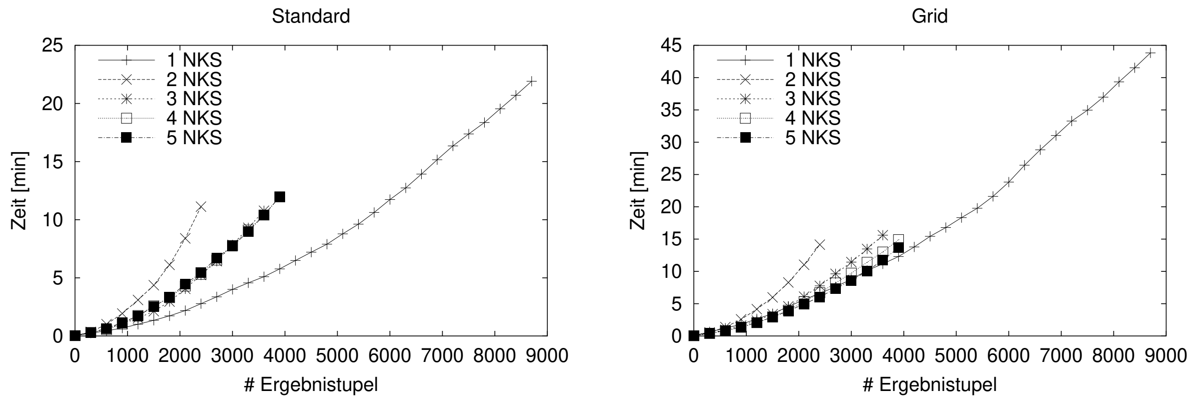


Abb. 6.21: Ergebnistupel pro Zeit beim naiven Verfahren und unterschiedlicher Granularität der Eingabedaten bei uniformer Verteilung

des alten Fensters zur Untergrenze des neuen Fensters, außer es kommt zu einem Springen der Fenster wie in Abbildung 4.7 auf Seite 49 dargestellt. Die Fenster werden also im Normalfall immer um die gesamte Fensterhöhe weitergezogen, denn bei der Berechnung der neuen Grenzen entfernt der Algorithmus zunächst alle Tupel aus dem linken Datenfenster, die genau auf der aktuellen Untergrenze dieses Fensters liegen. Damit bleiben innerhalb des Fensters nur die Tupel auf der aktuellen Obergrenze übrig, die von der Untergrenze alle den Abstand 0.1 haben. Aber auch das nächste einzulesende Tupel hat seinerseits von der Obergrenze den Abstand 0.1. Damit wandern sowohl die Obergrenze als auch die Untergrenze bei jeder Aktualisierung um diesen Wert nach oben. Die Tupel, die im alten Fenster auf der Obergrenze lagen, liegen im neuen Fenster nun auf der Untergrenze. Die neue Obergrenze wird dafür von den neu eingelesenen Tupeln besetzt. Damit können bei jedem Aktualisierungsschritt alle Tupelpaare des Ergebnispuffers O, deren linkes Tupel eines der entfernten Datenelemente der alten Fensteruntergrenze des linken Datenfensters ist, als Teil des Ergebnisses ausgegeben werden. Dies entspricht den Plateaus der Kurven in Abbildung 6.20. Dazwischen muss jeweils der gesamte Fensterinhalt verarbeitet werden, was jedes Mal einige Zeit dauert und die einzelnen Stufen erklärt. Bild 6.21 zeigt das Verhalten der naiven Verfahren.

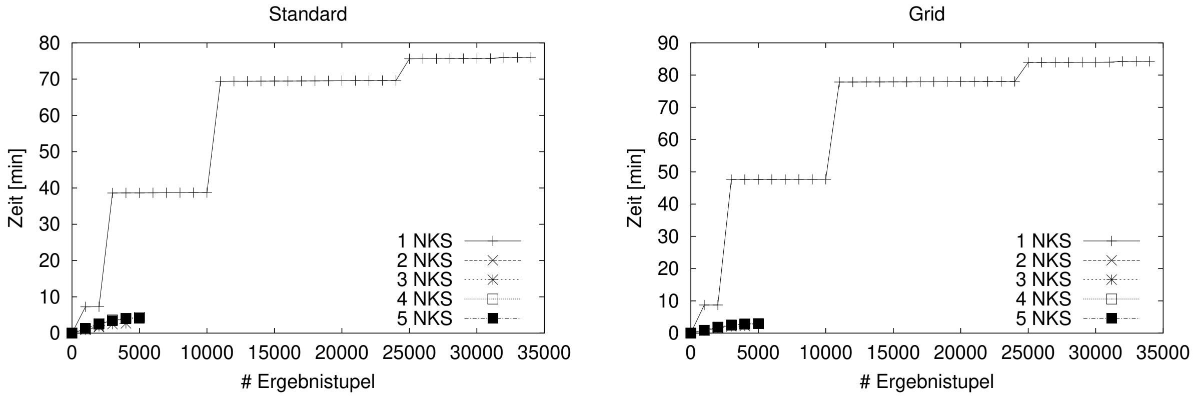


Abb. 6.22: Ergebnistupel pro Zeit beim fensterbasierten Verfahren mit EGO und unterschiedlicher Granularität der Eingabedaten bei Normalverteilung

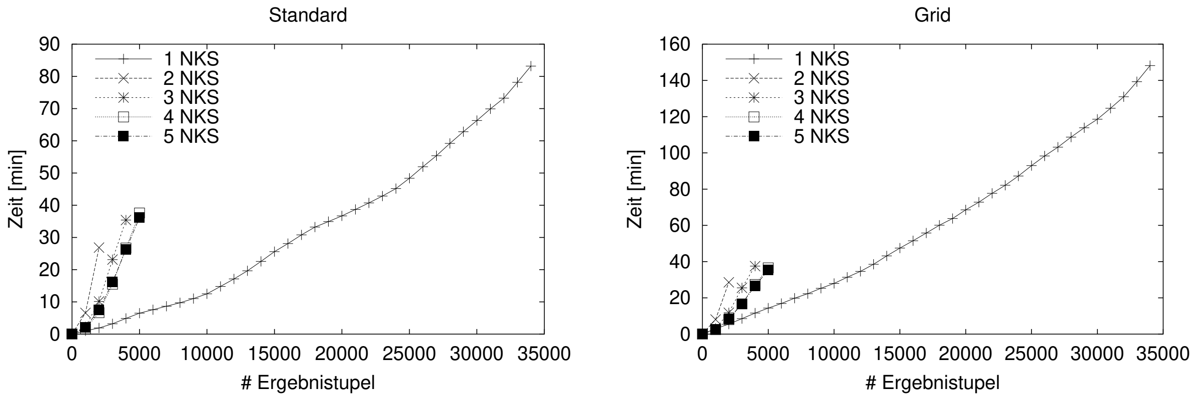


Abb. 6.23: Ergebnistupel pro Zeit beim naiven Verfahren und unterschiedlicher Granularität der Eingabedaten bei Normalverteilung

In den Abbildungen 6.22 und 6.23 sind die Ergebnisse der gleichen Messungen bei Normalverteilung der Eingabedaten dargestellt. Neben der stark erhöhten Anzahl an Ergebnistupeln ist dabei insbesondere der veränderte Verlauf der Treppenkurve bei einer Nachkommastelle in Bild 6.22 interessant. Wegen der Konzentration der Tupel in der Mitte des Datenraumes[1] sind die Treppenstufen in der Mitte der Berechnung deutlich höher als am Anfang und am Ende, weil die Datenfenster hier mit deutlich mehr Tupeln gefüllt sind. Damit benötigt auch die Verarbeitung der Fenster an dieser Stelle mehr Zeit.

6.3 Zusammenfassung

Insgesamt haben die Leistungsmessungen gezeigt, dass die fensterbasierten Methoden zur Berechnung von besten Zuordnungen von Objekten zweier Datenströme den naiven Methoden in zweierlei Hinsicht überlegen sind. Zum einen sind die fensterbasierten Verfahren pipelinefähig, während die naiven Verfahren blockierend sind. Zum anderen übertrifft der

[1] Vergleiche hierzu Abbildung 5.10 auf Seite 97.

fensterbasierte Ansatz den naiven auch in Bezug auf die Geschwindigkeit bei weitem. Sowohl die Laufzeit als auch die Anzahl an Seitenzugriffen lässt sich meist deutlich reduzieren, in vielen Fällen etwa um den Faktor 10.

Auch haben die Messungen ergeben, dass sich der Mehraufwand für die Umsortierung der Datenelemente gemäß Epsilon-Grid-Ordnung und den Einsatz des Scheduling-Algorithmus in jedem Fall lohnt. Diese Erweiterung ermöglicht eine messbare Verringerung der Ausführungszeiten und der Zahl der Seitenzugriffe.

Damit geht die fensterbasierte Methode mit Epsilon-Grid-Ordnung und Scheduling-Algorithmus als bestes aller untersuchten Verfahren aus der Evaluation hervor. Lediglich auf die Frage nach der zu verwendenden Materialisierungsvariante kann keine eindeutige Antwort gegeben werden. Hier hängt die richtige Entscheidung von den Rahmenbedingungen ab. Für bis zu zwei Join-Dimensionen hat sich die Grid-Materialisierung als die schnellere Alternative herausgestellt. Bei mehr als zwei Join-Dimensionen verliert sie aber aus den genannten Gründen stark an Effizienz, weshalb in diesem Fall unbedingt die Standard-Materialisierung vorzuziehen ist. Die Grid-Variante bietet allerdings noch Optimierungspotenzial, wodurch sich der Bereich ihrer Anwendbarkeit eventuell noch verbessern lässt.

Kapitel 7

Zusammenfassung und Ausblick

In dieser Arbeit wurden Methoden zur Berechnung von besten Zuordnungen von Objekten zweier Datenströme behandelt. Der zentrale Operator zur Bestimmung solcher bester Zuordnungen war dabei der Bestmatch-Join. Aus Gründen der höheren praktischen Relevanz beschränkten sich die Betrachtungen im Wesentlichen auf die Variante des Left-Outer-Bestmatch-Joins. Neben einem kurzen Überblick über verwandte Probleme wurde sowohl eine Übersicht über den Bestmatch-Join und seine Varianten als auch über mögliche algorithmische Ansätze zu seiner Berechnung gegeben. Dies geschah stets vor dem Hintergrund der Anwendbarkeit der Verfahren auf möglicherweise unendlichen Datenströmen. Neben einem kurzen Einblick in den approximativen Bestmatch-Join auf Datenströmen wurde genauer auf den Left-Outer-Bestmatch-Join mit Einschränkungen eingegangen. Dabei handelt es sich um einen Ansatz, der unter den Voraussetzungen vorsortierter Eingabedaten und eines eingeschränkten Suchraums die Ermittlung korrekter Join-Ergebnisse auf Datenströmen ermöglicht. Zur Berechnung dieser Ergebnisse wurden ein naiver Algorithmus, der lediglich als Vergleichsinstanz für die Leistungsmessungen dient, und ein spezieller fensterbasierter Algorithmus vorgestellt. Weiterhin wurde eine effizienzsteigernde Erweiterung dieses Verfahrens mittels einer Sortierung nach Epsilon-Grid-Ordnung und eines auf dieser Sortierung aufbauenden Scheduling-Algorithmus erläutert.

Darüber hinaus gibt die Arbeit einen Überblick über die Java-Implementierung eines Prototyps, der die vorgestellten naiven und fensterbasierten Methoden enthält und als Grundlage für die Leistungsmessungen und den Leistungsvergleich dient. Dabei wurde besonderer Wert auf die Effizienz und die Modularität der Implementierung gelegt.

Die Leistungsmessungen haben schließlich ergeben, dass die fensterbasierte Methode mit Epsilon-Grid-Ordnung und Scheduling-Algorithmus ein effizientes Verfahren ist, um beste Zuordnungen auf Datenströmen zu berechnen, und dabei aus zwei Eingabedatenströmen einen kontinuierlichen Strom von Ausgabedaten erzeugen kann. Damit ist bei diesem Verfahren die gewünschte Pipeline-Fähigkeit gegeben. Der naive Ansatz ist im Vergleich dazu deutlich unterlegen, da er blockierend und damit nicht pipelinefähig ist. Insbesondere ist er auf unendlichen Datenströmen nicht anwendbar. Auch was seine Leistungswerte anbetrifft, kann er im Allgemeinen nicht mit der fensterbasierten Methode mithalten.

Es gibt in dem behandelten Gebiet noch eine Reihe interessanter Punkte, die weitergehend untersucht werden könnten, im Rahmen dieser Arbeit aber keinen Platz mehr fanden. Dazu gehören auch die folgenden Überlegungen:

- Bezogen auf die Theorie des Bestmatch-Joins ist die Erweiterung der Operatoren auf mengenwertige Attribute wünschenswert, um die Beschränkung auf zahlenwertige Attribute aufzubrechen. Dabei sind auch Verfahren zur effizienten Berechnung von Ordnungen auf solchen mengenwertigen Attributen zu entwickeln. Ebenfalls denkbar sind weitere Betrachtungen zur Aufweichung der Voraussetzung einer strengen Vorsortierung der Eingabedaten, wie sie mit dem Konzept der Punktierungen in Kapitel 4.4 bereits kurz angedeutet wurden. Außerdem kann man sich mit der Zeit als vorsortiertem Attribut, sowohl in der Rolle eines Join-Attributs als auch in der Rolle eines nicht direkt an der Join-Berechnung teilnehmenden Nutzdaten-Attributs, noch näher beschäftigen. Zu all diesen Bereichen gibt es bereits Erkenntnisse, die z. B. in [KS02a] dargestellt sind.

- Auch die Prototyp-Implementierung kann aufgrund ihres modularen Aufbaus noch um viele zusätzliche Teile erweitert werden. Möglich und sinnvoll wären u. a. weitere Auswertungspläne, Vergleichsordnungen oder auch Join-Varianten. Denkbar ist weiter die Umsetzung der in Abschnitt 5.2.4 aufgeführten Optimierungsansätze für die Grid-Materialisierungsvariante und die Untersuchung ihrer Auswirkungen. Außerdem gibt es Überlegungen, die Algorithmen durch eine Hauptspeicherumschaltung zu optimieren, welche die Materialisierung der Eingabedaten abschaltet und die Ausführung der Join-Berechnung ausschließlich im Hauptspeicher vollzieht, falls die zu verarbeitenden Datenmengen vollständig in den Hauptspeicherpuffer passen.

- Zuletzt kann noch eine mögliche Ergänzung der Leistungsmessungen angeführt werden. Hier ließe sich der Speicherplatzbedarf der verschiedenen Verfahren auf dem Hintergrundspeicher messen und vergleichen. Von besonderem Interesse wäre in diesem Zusammenhang der maximale Speicherverbrauch der Verfahren im Verlauf einer Berechnung.

Abschließend bleibt zu sagen, dass insbesondere das Gebiet der Verarbeitung von Datenströmen ganz allgemein auch in Zukunft noch Raum für weitere Untersuchungen bieten wird.

Anhang A

Bedienung des Prototyps

Die Bedienung und Konfiguration des Prototyps zur Berechnung des Left-Outer-Bestmatch-Joins mit Einschränkungen erfolgt über Kommandozeilenparameter. Alternativ können die Konfigurationseinstellungen und Aufrufparameter auch in eine Konfigurationsdatei geschrieben und von dort vom Programm eingelesen werden. Die Bedienung des Datengenerators und des I/O-Testprogramms erfolgt ebenfalls über die Kommandozeile.

A.1 Kommandozeilen-Interface des Prototyps

Der Prototyp wird auf der Kommandozeile mittels folgender Syntax gestartet:

```
java bmj.Main [OPTIONS] file1 attr1,attr2,attr3,...
                        file2 attr1,attr2,attr3,...
                        ...
```

An den Programmaufruf `java bmj.Main` schließen sich zunächst die Optionen an. Diese werden ab Seite 128 im Detail beschrieben. Danach folgen, gegebenenfalls mit absoluter oder relativer Pfadangabe versehen, die Namen der Eingabedateien, welche die zu verarbeitenden Daten enthalten. Jede Datei wird unmittelbar von einem Leerzeichen und der Liste ihrer Join-Attribute gefolgt. Die Namen der Join-Attribute sind durch Kommata getrennt. Die Attribut-Liste darf keine Leerzeichen enthalten. Wieviele Dateien sinnvoll angegeben werden können, hängt von den restlichen Parametern ab. Standardmäßig verarbeitet das Programm nur die ersten beiden Eingabedateien, wobei die erste Datei als linke und die zweite Datei als rechte Eingabe des Joins verwendet wird. In der Regel muss die Anzahl der Join-Attribute der beiden Eingabedateien eines Joins gleich sein, jedoch ist auch dies eventuell von anderen Parametern abhängig.

Ist der Aufruf fehlerhaft, so gibt das Programm eine entsprechende Fehlermeldung und einen Hilfetext aus und beendet die Programmausführung.

Optionen:

`-h, --help`
Gibt einen Hilfetext aus.

`-vs, --version`
Gibt die Programmversion aus.

`-v, --verbose`
Gibt während des Programmlaufs zusätzliche Informationen aus.

`-nm, --no-mat`
Deaktiviert die Materialisierung. Alle Algorithmen laufen im Hauptspeicher ab. Diese Option muss mit einem nicht-materialisierenden Datenfenster kombiniert werden.

`-no, --no-order`
Deaktiviert die Verwendung einer besonderen Sortierordnung bei der Materialisierung. Durch den Einsatz dieser Option kann der fensterbasierte Algorithmus ohne Verwendung der Epsilon-Grid-Ordnung ausgeführt werden. Es wird also sowohl die Sortierung der Hauptspeichertupel vor der Materialisierung als auch der Scheduling-Algorithmus für eine effizientere Strategie beim Laden von Hintergrundspeicherseiten abgeschaltet.

`-ps, --pagesize <int>`
Legt die Seitengröße für Speicherseiten in Kilobyte fest.

Mögliche Einstellungen: $n \in \mathbb{N}^+$

Standardeinstellung: `8`

`-s, --sorted-by <int>`
Gibt an, nach welchem Join-Attribut die Eingabedaten aufsteigend sortiert sind. Dabei steht `0` für das erste und `Anzahl Join-Attribute` $-$ `1` für das letzte Join-Attribut. Der Wert $-$`1` bedeutet, dass die Eingabedaten nach keinem Join-Attribut aufsteigend sortiert sind.

Mögliche Einstellungen: $-$`1, 0, ..., Anzahl Join-Attribute` $-$ `1`

Standardeinstellung: $-$`1`

`-c, --config-file <filename>`
Lädt die Konfigurationseinstellungen für den aktuellen Programmlauf aus der angegebenen Konfigurationsdatei. Beim Laden einer Konfigurationsdatei werden zunächst alle Einstellungen auf Standardwerte zurückgesetzt, d.h. durch andere Kommandozeilenparameter vorgenommene Einstellungen werden ignoriert, und anschließend die Werte aus der Datei eingelesen und übernommen. Optionen, die in der Datei nicht vorhanden oder nicht gesetzt sind, werden auf der Standardeinstellung belassen. Das Aussehen und die möglichen Einträge einer Konfigurationsdatei werden in Abschnitt A.2 beschrieben.

`-w, --write-to-file <filename>`
Schreibt die Ausgabe des Programms in die angegebene Datei. Existiert die Datei bereits, wird nachgefragt, ob sie überschrieben oder der Programmlauf abgebrochen werden soll.

`-tt, --tuples-time <filename>`
Schreibt während des Programmlaufs für jedes erzeugte Ergebnistupel die Nummer des Tupels sowie die vergangene Zeit seit dem Start des Programmlaufs bis zur Ausgabe des jeweiligen Ergebnistupels in die angegebene Datei. Existiert die Datei bereits, wird nachgefragt, ob sie überschrieben oder der Programmlauf abgebrochen werden soll.

`-ds, --detail-stats <filename>`
Schreibt am Ende des Programmlaufs die Statistik des Laufs in die angegebene Datei. Existiert die Datei noch nicht, so wird sie neu angelegt. Ansonsten wird das Ergebnis dieses Laufs an das Ende der existierenden Datei angehängt. Wird diese Option mit der Option `-rp, --replace` kombiniert, dann wird die Datei in jedem Fall neu erzeugt. Falls sie bereits existiert, wird sie überschrieben.

`-rp, --replace`
Weist die Option `-ds, --detail-stats` an, die angegebene Datei durch eine neu angelegte Datei zu überschreiben, falls sie bereits existiert, anstatt das Ergebnis des Programmlaufs an das Ende der existierenden Datei anzuhängen.

`-j, --join <name>`
Weist das Programm an, den angegebenen Join-Algorithmus zu verwenden.

Mögliche Einstellungen:

- `NLJ` naiver Algorithmus
- `BMJ` fensterbasierter Algorithmus

Standardeinstellung: `BMJ`

`-o, --order <name>`
Veranlasst das Programm, die angegebene Vergleichsordnung zu benutzen.

Mögliche Einstellungen: `defaultorder` Standard-Ordnung (*minAttrDist*)

Standardeinstellung: `defaultorder`

`-p, --plan <name>`
Veranlasst das Programm, den angegebenen Auswertungsplan zu benutzen.

Mögliche Einstellungen: `defaultplan` Standard-Plan

Standardeinstellung: `defaultplan`

`-pm, --page-manager <name>`
Weist das Programm an, die angegebene Materialisierungsvariante zu verwenden.

Mögliche Einstellungen:

- `defpm` Standard-Variante mit Objekt-Serialisierung
- `moddefpm` Standard-Variante mit Objekt-Serialisierung und verbessertem Iterator über die materialisierten Seiten
- `defbmjpm` Standard-Variante mit Materialisierung auf Basis von Java NIO
- `moddefbmjpm` Standard-Variante mit Materialisierung auf Basis von Java NIO und verbessertem Iterator über die materialisierten Seiten
- `gridpm` Grid-Variante

Standardeinstellung: `moddefbmjpm`

`-dw, --data-window <name>`
Weist das Programm an, das angegebene Datenfenster zu benutzen.

Mögliche Einstellungen:

- `defmat` Standard-Datenfenster für materialisierende Algorithmen
- `defnonmat` Standard-Datenfenster für nicht-materialisierende Algorithmen

Standardeinstellung: `defmat`

`-so, --sort-order <name>`
Aktiviert die angegebene Sortierordnung.

Mögliche Einstellungen:

- `epsilongrid` Epsilon-Grid-Ordnung
- `initial` Ordnung, nach der die Eingabedaten vorsortiert sind

Standardeinstellung: `epsilongrid`

`-sa, --sort-algorithm <name>`
Wählt den angegebenen Sortieralgorithmus aus.

Mögliche Einstellungen: `quicksort` Quicksort-Algorithmus

Standardeinstellung: `quicksort`

`-e, --epsilons <e1,e2,e3,...>`
Setzt die vom Programm zu verwendenden ϵ-Werte für die ϵ-Umgebung. Dabei steht `e1` für die ϵ-Umgebung des ersten Join-Attributs, `e2` für die ϵ-Umgebung des zweiten Join-Attributs, usw. Für jedes Join-Attribut muss ein ϵ-Wert angegeben werden. Die einzelnen ϵ-Werte werden durch Kommata getrennt. Die Liste der ϵ-Werte darf keine Leerzeichen enthalten.

Mögliche Einstellungen: $\forall i : \epsilon_i \in]0;1]$

Standardwert: $\forall i : \epsilon_i =$ `0.1`

`-lb, --left-buffersize <int>`

Legt den verfügbaren Hauptspeicher in Anzahl Speicherseiten für die linke Eingaberelation fest.

Mögliche Einstellungen: $n \in \mathbb{N} \wedge n \geq 2$

Standardeinstellung: `5`

`-rb, --right-buffersize <int>`

Legt den verfügbaren Hauptspeicher in Anzahl Speicherseiten für die rechte Eingaberelation fest.

Mögliche Einstellungen: $n \in \mathbb{N} \wedge n \geq 2$

Standardeinstellung: `5`

A.2 Konfigurationsdateien

Mit Hilfe von Konfigurationsdateien ist es möglich, einen Programmlauf mit denselben Konfigurationseinstellungen mehrmals zu starten, ohne jedesmal eine mitunter große Anzahl an Kommandozeilenparametern neu eintippen zu müssen. Auch sind mehrere Läufe mit jeweils geringfügig veränderten Einstellungen durch kleine Modifikationen an einer bestehenden Konfigurationsdatei mit relativ wenig Aufwand möglich. Die meisten in Abschnitt A.1 beschriebenen Optionen können auch in einer Konfigurationsdatei gesetzt werden. Dazu gibt es für jede Option einen Konfigurationsschlüssel, der zusammen mit dem gewünschten Wert für die jeweilige Option in die Konfigurationsdatei eingetragen wird. Teilt man dem Programm beim Aufruf mittels der `-c, --config-file` Option mit, dass die Konfiguration aus der angegebenen Datei gelesen werden soll, so werden zunächst alle Parameter auf Standardwerte zurückgesetzt und alle durch andere Kommandozeilenparameter getätigten Einstellungen ignoriert. Anschließend lädt das Programm die Einstellungen aus der Datei. Fehlt ein Konfigurationsschlüssel in der angegebenen Konfigurationsdatei oder ist ihm in der Datei kein Wert zugewiesen, so nimmt die zugehörige Option beim Laden der Datei ihre Standardeinstellung an.

Die Konfigurationsdatei ist eine Java Properties Datei und muss deren Format, wie in der Java API Spezifikation der Klasse `java.util.Properties` in [Sun02a] beschrieben, genügen. Dieses Format ist sehr einfach. Jede Zeile enthält ein Paar bestehend aus Konfigurationsschlüssel und dem diesem Schlüssel zugeordneten Wert, verbunden durch ein Gleichheitszeichen. Leerzeilen oder Schlüsseleinträge, die das Programm nicht kennt, sind erlaubt. Sie werden einfach ignoriert. Alle Zeilen, die mit dem Zeichen `#` oder `!` beginnen, sind Kommentare und werden ebenfalls ignoriert.

Die Konfigurationsschlüssel und ihre möglichen Werte sind im Anschluss aufgeführt.

Konfigurationsschlüssel:

`BMJVERBOSE`

Gibt an, ob während des Programmlaufs zusätzliche Informationen ausgegeben werden sollen.

Mögliche Einstellungen:

- `true` zusätzliche Informationen ausgeben
- `false` keine zusätzlichen Informationen ausgeben

Standardeinstellung: `false`

`BMJMATERIALIZE`

Aktiviert oder deaktiviert die Materialisierung. Bei deaktivierter Materialisierung laufen alle Algorithmen im Hauptspeicher ab. Diese Option muss bei aktivierter Materialisierung mit einem materialisierenden Datenfenster, bei deaktivierter Materialisierung mit einem nicht-materialisierenden Datenfenster kombiniert werden.

Mögliche Einstellungen:

- `true` Materialisierung aktiviert
- `false` Materialisierung deaktiviert

Standardeinstellung: `true`

`BMJUSEORDER`

Aktiviert oder deaktiviert die Verwendung einer besonderen Sortierordnung bei der Materialisierung. Durch Deaktivieren der Ordnung kann der fensterbasierte Algorithmus ohne Verwendung der Epsilon-Grid-Ordnung ausgeführt werden. Es wird also sowohl die Sortierung der Hauptspeichertupel vor der Materialisierung als auch der Scheduling-Algorithmus für eine effizientere Strategie beim Laden von Hintergrundspeicherseiten abgeschaltet.

Mögliche Einstellungen:

- `true` Sortierordnung aktiviert
- `false` Sortierordnung deaktiviert

Standardeinstellung: `true`

`BMJPAGESIZE`

Legt die Seitengröße für Speicherseiten in Kilobyte fest.

Mögliche Einstellungen: $n \in \mathbb{N}^+$

Standardeinstellung: `8`

`BMJSORTEDBY`

Gibt an, nach welchem Join-Attribut die Eingabedaten aufsteigend sortiert sind. Dabei steht `0` für das erste und `Anzahl Join-Attribute − 1` für das letzte Join-Attribut. Der Wert `−1` bedeutet, dass die Eingabedaten nach keinem Join-Attribut aufsteigend sortiert sind.

Mögliche Einstellungen: `−1, 0, ..., Anzahl Join-Attribute − 1`

Standardeinstellung: `−1`

`BMJOUTPUTFILE`

Schreibt die Ausgabe des Programms in die angegebene Datei. Existiert die Datei bereits, wird nachgefragt, ob sie überschrieben oder der Programmlauf abgebrochen werden soll. Fehlt dieser Schlüssel in der Konfigurationsdatei oder ist ihm kein Wert zugewiesen, wird die Ausgabe des Programms nicht in eine Datei geschrieben, sondern auf die Standardausgabe.

`BMJTUPLESPERTIMEFILE`

Schreibt während des Programmlaufs für jedes erzeugte Ergebnistupel die Nummer des Tupels sowie die vergangene Zeit seit dem Start des Programmlaufs bis zur Ausgabe des jeweiligen Ergebnistupels in die angegebene Datei. Existiert die Datei bereits, wird nachgefragt, ob sie überschrieben oder der Programmlauf abgebrochen werden soll. Fehlt dieser Schlüssel in der Konfigurationsdatei oder ist ihm kein Wert zugewiesen, werden die Informationen dieser Option nicht geschrieben und nicht ausgegeben.

`BMJDETAILSTATFILE`

Schreibt am Ende des Programmlaufs die Statistik des Laufs in die angegebene Datei. Existiert die Datei noch nicht, so wird sie neu angelegt. Ansonsten wird das Ergebnis dieses Laufs an das Ende der existierenden Datei angehängt. Wird diese Option mit der Option `BMJREPLACEDETAILSTATFILE` kombiniert, dann wird die Datei in jedem Fall neu erzeugt. Falls sie bereits existiert, wird sie überschrieben. Fehlt dieser Schlüssel in der Konfigurationsdatei oder ist ihm kein Wert zugewiesen, werden die Informationen dieser Option nicht geschrieben und nicht ausgegeben.

`BMJREPLACEDETAILSTATFILE`

Weist die Option `BMJDETAILSTATFILE` an, die angegebene Datei durch eine neu angelegte Datei zu überschreiben, falls sie bereits existiert, anstatt das Ergebnis des Programmlaufs an das Ende der existierenden Datei anzuhängen.

`BMJJOIN`

Weist das Programm an, den angegebenen Join-Algorithmus zu verwenden.

Mögliche Einstellungen:

- `NLJ` naiver Algorithmus
- `BMJ` fensterbasierter Algorithmus

Standardeinstellung: `BMJ`

`BMJORDER`

Veranlasst das Programm, die angegebene Vergleichsordnung zu benutzen.

Mögliche Einstellungen: `defaultorder` Standard-Ordnung (*minAttrDist*)

Standardeinstellung: `defaultorder`

`BMJPLAN`

Veranlasst das Programm, den angegebenen Auswertungsplan zu benutzen.

Mögliche Einstellungen: `defaultplan` Standard-Plan

Standardeinstellung: `defaultplan`

`BMJPAGEMANAGER`

Weist das Programm an, die angegebene Materialisierungsvariante zu verwenden.

Mögliche Einstellungen:

- `defpm` Standard-Variante mit Objekt-Serialisierung
- `moddefpm` Standard-Variante mit Objekt-Serialisierung und verbessertem Iterator über die materialisierten Seiten
- `defbmjpm` Standard-Variante mit Materialisierung auf Basis von Java NIO
- `moddefbmjpm` Standard-Variante mit Materialisierung auf Basis von Java NIO und verbessertem Iterator über die materialisierten Seiten
- `gridpm` Grid-Variante

Standardeinstellung: `moddefbmjpm`

`BMJDATAWINDOW`

Weist das Programm an, das angegebene Datenfenster zu benutzen.

Mögliche Einstellungen:

- `defmat` Standard-Datenfenster für materialisierende Algorithmen
- `defnonmat` Standard-Datenfenster für nicht-materialisierende Algorithmen

Standardeinstellung: `defmat`

`BMJSORTORDER`

Aktiviert die angegebene Sortierordnung.

Mögliche Einstellungen:

- `epsilongrid` Epsilon-Grid-Ordnung
- `initial` Ordnung, nach der die Eingabedaten vorsortiert sind

Standardeinstellung: `epsilongrid`

`BMJSORTALGORITHM`
Wählt den angegebenen Sortieralgorithmus aus.

Mögliche Einstellungen: `quicksort` Quicksort-Algorithmus

Standardeinstellung: `quicksort`

`BMJEPSILONS`
Setzt die vom Programm zu verwendenden ϵ-Werte für die ϵ-Umgebung. Für jedes Join-Attribut muss ein ϵ-Wert angegeben werden. Die einzelnen ϵ-Werte werden durch Kommata getrennt. Die Liste der ϵ-Werte darf keine Leerzeichen enthalten.

Mögliche Einstellungen: $\forall i : \epsilon_i \in]0;1]$.

Standardeinstellung: $\forall i : \epsilon_i =$ `0.1`

`BMJLEFTBS`
Legt den verfügbaren Hauptspeicher in Anzahl Speicherseiten für die linke Eingaberelation fest.

Mögliche Einstellungen: $n \in \mathbb{N} \ \wedge \ n \geq 2$

Standardeinstellung: `5`

`BMJRIGHTBS`
Legt den verfügbaren Hauptspeicher in Anzahl Speicherseiten für die rechte Eingaberelation fest.

Mögliche Einstellungen: $n \in \mathbb{N} \ \wedge \ n \geq 2$

Standardeinstellung: `5`

`BMJFILES`
Spezifiziert die Eingabedateien und deren Join-Attribute. Der Eintrag für diesen Schlüssel besteht für jede Eingabedatei aus dem Dateinamen, gegebenenfalls mit absoluter oder relativer Pfadangabe versehen, und den durch Kommata getrennten Namen der Join-Attribute der jeweiligen Datei. Die Liste der Attributnamen darf keine Leerzeichen enthalten. Der Dateiname und die Liste der Namen der Join-Attribute dieser Datei werden ebenso mittels eines Leerzeichens getrennt, wie die Einträge für verschiedene Eingabedateien.

Abbildung A.1 auf der nächsten Seite zeigt eine exemplarische Konfigurationsdatei.

A.3 Format der Eingabedateien

Die Eingabedateien sind normale Textdateien, die im Dateisystem abgespeichert werden. Von dort liest das Programm sie ein und verarbeitet ihren Inhalt. Die Dateien müssen ein bestimmtes Format einhalten, damit der Prototyp sie korrekt einlesen und verarbeiten kann. Die erste Zeile der Datei enthält sämtliche Attributnamen in der richtigen Reihenfolge. Die Namen der einzelnen Attribute werden durch Tabulatorzeichen getrennt. Die

```
# Beispiel-Konfigurationsdatei

BMJMATERIALIZE=true
BMJUSEORDER=true
BMJVERBOSE=false
BMJPAGESIZE=4
BMJSORTEDBY=0
BMJOUTPUTFILE=
BMJTUPLESPERTIMEFILE=tpt.stat
BMJDETAILSTATFILE=detail.stat
BMJREPLACEDETAILSTATFILE=true
BMJJOIN=BMJ
BMJORDER=
BMJPLAN=
BMJPAGEMANAGER=gridpm
BMJDATAWINDOW=defmat
BMJSORTORDER=epsilongrid
BMJSORTALGORITHM=
BMJEPSILONS=0.2,0.3
BMJLEFTBS=10
BMJRIGHTBS=10
BMJFILES=../../tests/test1.data D0,D1 ../../tests/test2.data D0,D1
```

Abb. A.1: Beispiel für eine Konfigurationsdatei

zweite Zeile enthält die entsprechenden Typen der einzelnen Attribute, ebenfalls durch Tabulatorzeichen getrennt. Mögliche Attribut-Typen sind Integer, Double und String. Die restlichen Zeilen enthalten die Attributwerte der einzelnen Tupel der Relation. Jede Zeile entspricht einem Tupel, die einzelnen Attributwerte sind wiederum durch Tabulatorzeichen getrennt.

Abbildung A.2 zeigt ein Beispiel für eine kleine Eingabedatei.

```
D0      D1      D2      D3      D4      DATA
Double  Double  Double  Double  Integer String
0.90683 0.19013 0.43587 0.79811 12      XXXXXXXXXXXXXXXXXXX0
0.38014 0.43757 0.69444 0.01    8       XXXXXXXXXXXXXXXXXXX1
0.60981 0.64523 0.39352 0.29459 123     XXXXXXXXXXXXXXXXXXX2
0.07932 0.707   0.64832 0.51674 0       XXXXXXXXXXXXXXXXXXX3
0.02481 0.72145 0.73998 0.18397 42      XXXXXXXXXXXXXXXXXXX4
```

Abb. A.2: Beispiel für eine Eingabedatei

A.4 Kommandozeilen-Interface des Datengenerators

Der Datengenerator wird auf der Kommandozeile mittels folgender Syntax gestartet:

```
java bmj.test.DataGenerator  <number of attributes> <number of tuples>
                             <outputfilename> [OPTIONS]
```

Dem Programmaufruf `java bmj.test.DataGenerator` folgen zwei Integer-Werte und ein String-Wert. Der erste Integer-Wert gibt die Anzahl an Attributen vom Typ Double an, die jedes Tupel in der generierten Relation besitzen soll. Tatsächlich besitzen die Tupel insgesamt ein Attribut mehr, da als letztes Attribut der Relation immer zusätzlich ein Füllattribut vom Typ String generiert wird. Der zweite Integer-Wert gibt an, wie viele Tupel die generierte Relation enthalten soll. Der String-Wert schließlich spezifiziert den Dateinamen, in den die erzeugten Daten zu schreiben sind. Existiert die hier angegebene Datei bereits, fragt das Programm nach, ob die existierende Datei überschrieben oder der Programmlauf abgebrochen werden soll. Am Ende des Aufrufs können noch einige Optionen angegeben werden, die nachfolgend genauer beschrieben sind.

Bei einem fehlerhaften Aufruf gibt auch der Datengenerator eine entsprechende Fehlermeldung und einen Hilfetext aus und beendet die Programmausführung.

Optionen:

`-h, --help`
Gibt einen Hilfetext aus.

`-vs, --version`
Gibt die Programmversion aus.

`-p, --prefix <prefix>`
Benutzt das angegebene Präfix für die Namen der Attribute vom Typ Double.

Standardeinstellung: `D`

`-bn, --bulk-name <name>`
Benutzt den angegebenen Namen für das Füllattribut vom Typ String.

Standardeinstellung: `DATA`

`-bl, --bulk-length <int>`
Erzeugt für das Füllattribut Strings von der angegebenen Länge.

Standardeinstellung: `20`

`-d, --distribution <name>`
Gibt an, welche Verteilung zur Erzeugung der zufällig generierten Double-Werte verwendet werden soll.

Mögliche Einstellungen:

- `uniform` uniforme Verteilung
- `normal` Normalverteilung
- `corr` korrelierte Verteilung
- `anticorr` anti-korrelierte Verteilung

Standardeinstellung: `uniform`

`-s, --sort-by <int>`
Generiert eine Relation, deren Tupel nach dem Double-Attributwert mit dem angegebenen Index aufsteigend sortiert sind. Dabei ist `0` der Index des ersten und `Anzahl Double-Attribute` $-$ `1` der Index des letzten Double-Attributs. Wird diese Option nicht angegeben, erzeugt der Datengenerator eine unsortierte Relation.

Mögliche Einstellungen: `0, ..., Anzahl Double-Attribute` $-$ `1`

Standardeinstellung: keine Sortierung

`-m, --mean <double>`
Benutzt den angegebenen Erwartungswert für die Normalverteilung zur Generierung zufälliger Double-Werte.

Mögliche Einstellungen: $n \in [0; 1]$

Standardeinstellung: `0.5`

`-sd, --std-dev <double>`
Benutzt die angegebene Standardabweichung für die Normalverteilung zur Generierung zufälliger Double-Werte.

Mögliche Einstellungen: $n \in \mathbb{R}_0^+$

Standardeinstellung: `0.15`

`-dm, --diff-mean <double>`
Benutzt den angegebenen Erwartungswert für die Generierung von zufälligen Differenzwerten für die korrelierte und anti-korrelierte Verteilung. Um sicherzustellen, dass der Datengenerator terminiert, ist hier nur `0` als Wert zugelassen. Die Option ist dennoch vorhanden, um die Einstellung des Wertes erlauben zu können, falls dies einmal nötig werden sollte.

Mögliche Einstellungen: `0`

Standardeinstellung: `0`

-dsd, --diff-std-dev <double>

Benutzt die angegebene Standardabweichung für die Generierung von zufälligen Differenzwerten für die korrelierte und anti-korrelierte Verteilung.

Mögliche Einstellungen: $n \in \mathbb{R}_0^+$

Standardeinstellung: `0.075`

A.5 I/O-Testprogramm

Zur Entscheidungsunterstützung bei der Suche nach einem geeigneten Implementierungsansatz für die I/O-Operationen der Tupel-Materialisierung des Prototyps wurde ein Testprogramm implementiert, das den Vergleich der Leistung dreier verschiedener Ansätze ermöglicht. Diese drei Varianten sind im einzelnen:

- Einsatz der Java Objekt-Serialisierung zum Schreiben und Lesen der Tupel auf den und vom Hintergrundspeicher.
- Implementierung eines eigenen Verfahrens zum Schreiben und Lesen der Tupel auf der Basis des im JDK 1.4.0 von Sun Microsystems, Inc. neu hinzugekommenen Java NIO Pakets, das unter anderem in [Sun02a], [Sun02c] und dem Specification Request [Sun02b] beschrieben wird.
- Implementierung eines eigenen Verfahrens zum Schreiben und Lesen der Tupel auf der Basis des herkömmlichen Java IO Pakets.

Um eine Entscheidung für eine dieser Alternativen treffen zu können, war es angebracht, eine Testimplementierung aller drei Möglichkeiten zu erstellen und im Praxistest zu vergleichen.

Das I/O-Testprogramm wird auf der Kommandozeile mittels folgender Syntax gestartet:

```
java bmj.IOTest <input-filename>
```

Auf den Programmaufruf `java bmj.IOTest` folgt als einziges Argument der Dateiname der Eingabedatei, deren Tupel zu Testzwecken materialisiert und wieder eingelesen werden sollen. Diese Datei besitzt dasselbe Format wie die in Abschnitt A.3 beschriebenen Eingabedateien des Prototyps.

Das Testprogramm führt nacheinander mit jeder der drei oben genannten Implementierungsvarianten dieselben Operationen aus. Dabei werden zunächst alle Tupel der angegebenen Relation sequenziell mit dem entsprechenden Verfahren auf die Platte geschrieben. Das Programm misst die hierfür benötigte Zeit und gibt den Messwert anschließend aus. Danach werden alle Tupel ebenfalls sequenziell wieder eingelesen. Auch die hierfür benötigte Zeit wird gemessen und ausgegeben. Das Testergebnis besteht schließlich aus insgesamt sechs Werten. Dies sind für jede Implementierungsalternative jeweils die benötigte Zeit in Millisekunden für die Materialisierung der kompletten Eingaberelation sowie für das vollständige Einlesen der materialisierten Tupel. Eine mögliche Ausgabe des I/O-Testprogramms für eine exemplarische Eingabedatei bestehend aus 5000 Tupeln auf einem Sun Enterprise 450 Server zeigt Abbildung A.3.

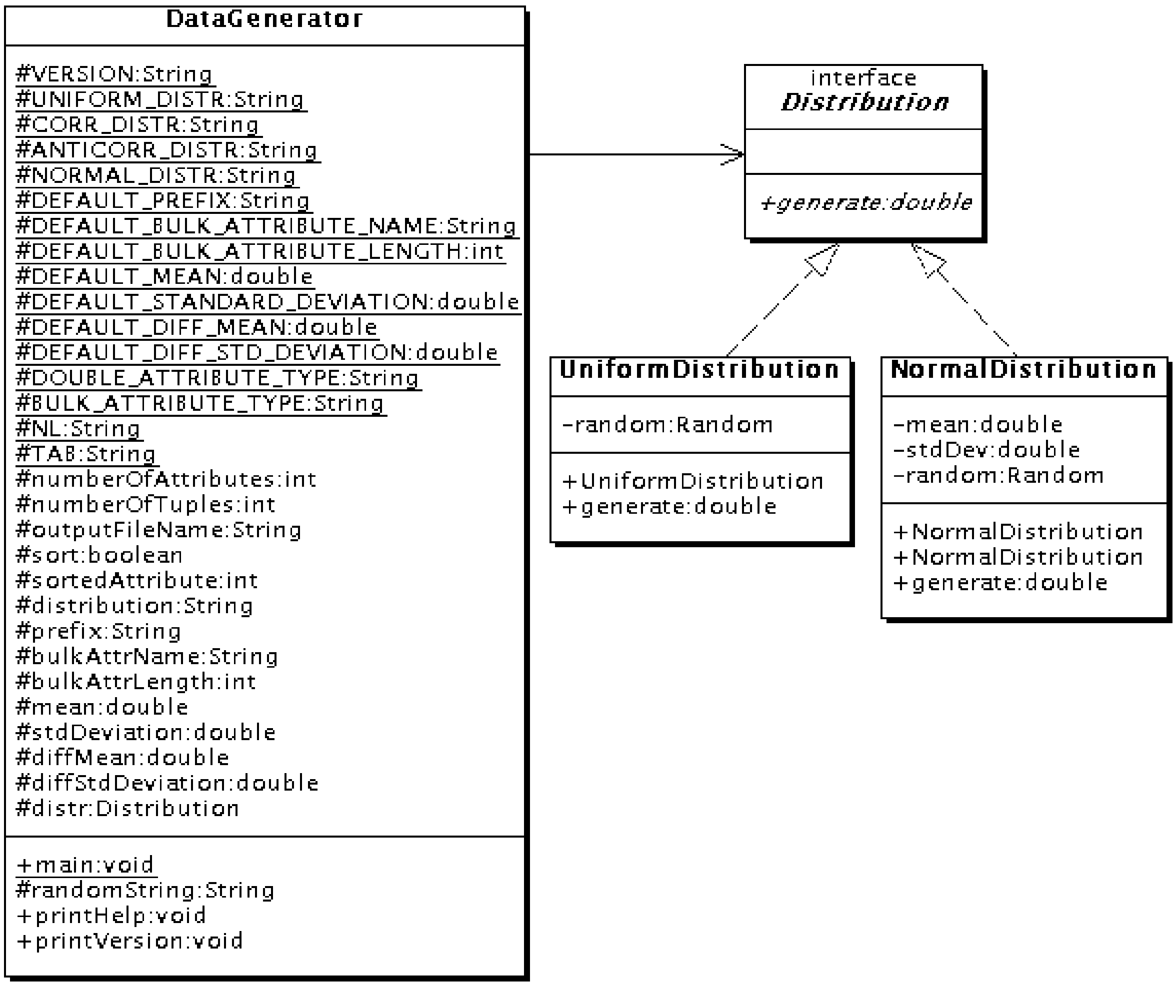
```
Object Serialization (Write): 8112 ms
Object Serialization (Read): 6869 ms
NIO (Write): 1225 ms
NIO (Read): 563 ms
IO (Write): 877 ms
IO (Read): 7333 ms
```

Abb. A.3: Ausgabe des I/O-Testprogramms

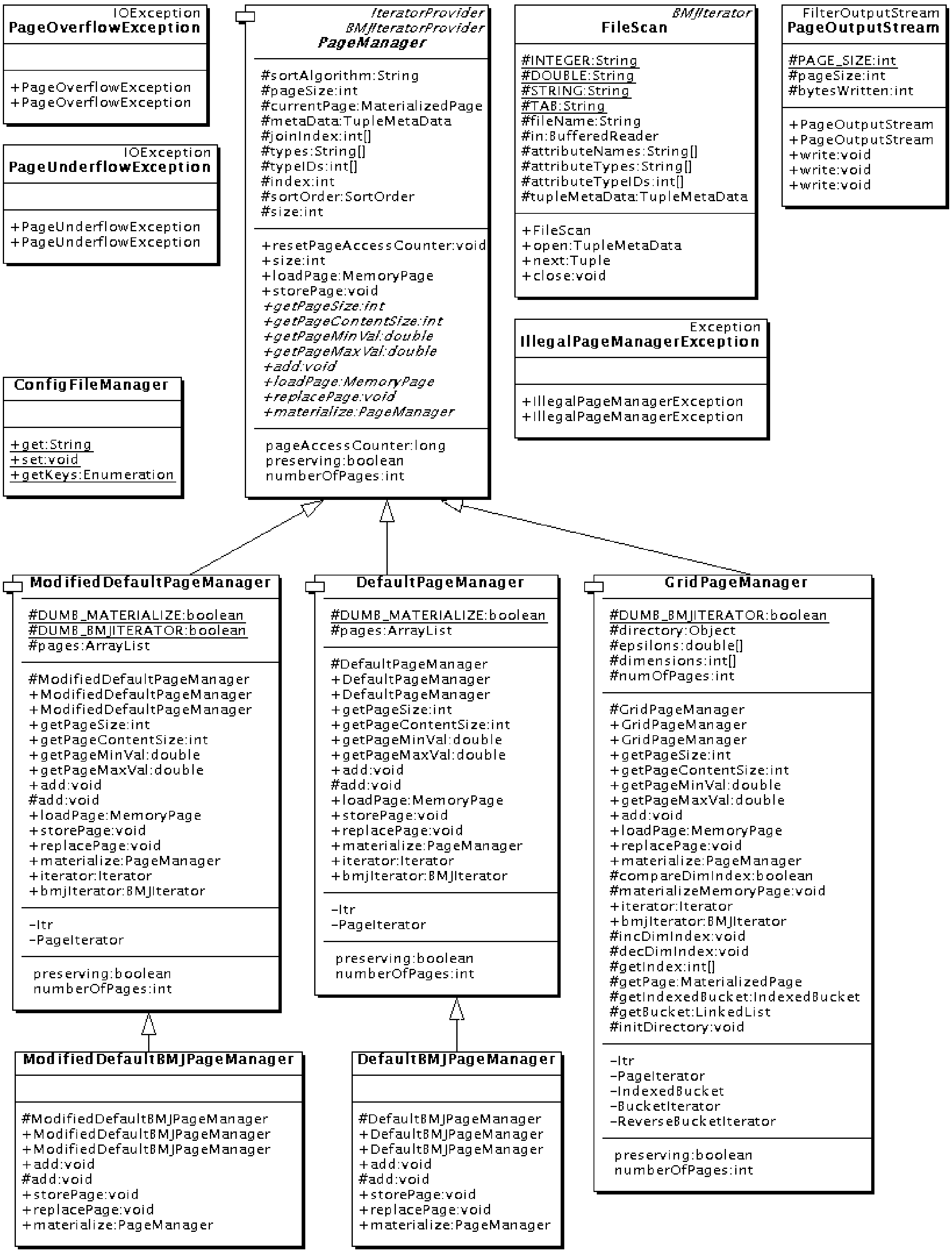
Anhang B

Entwurfsdiagramme

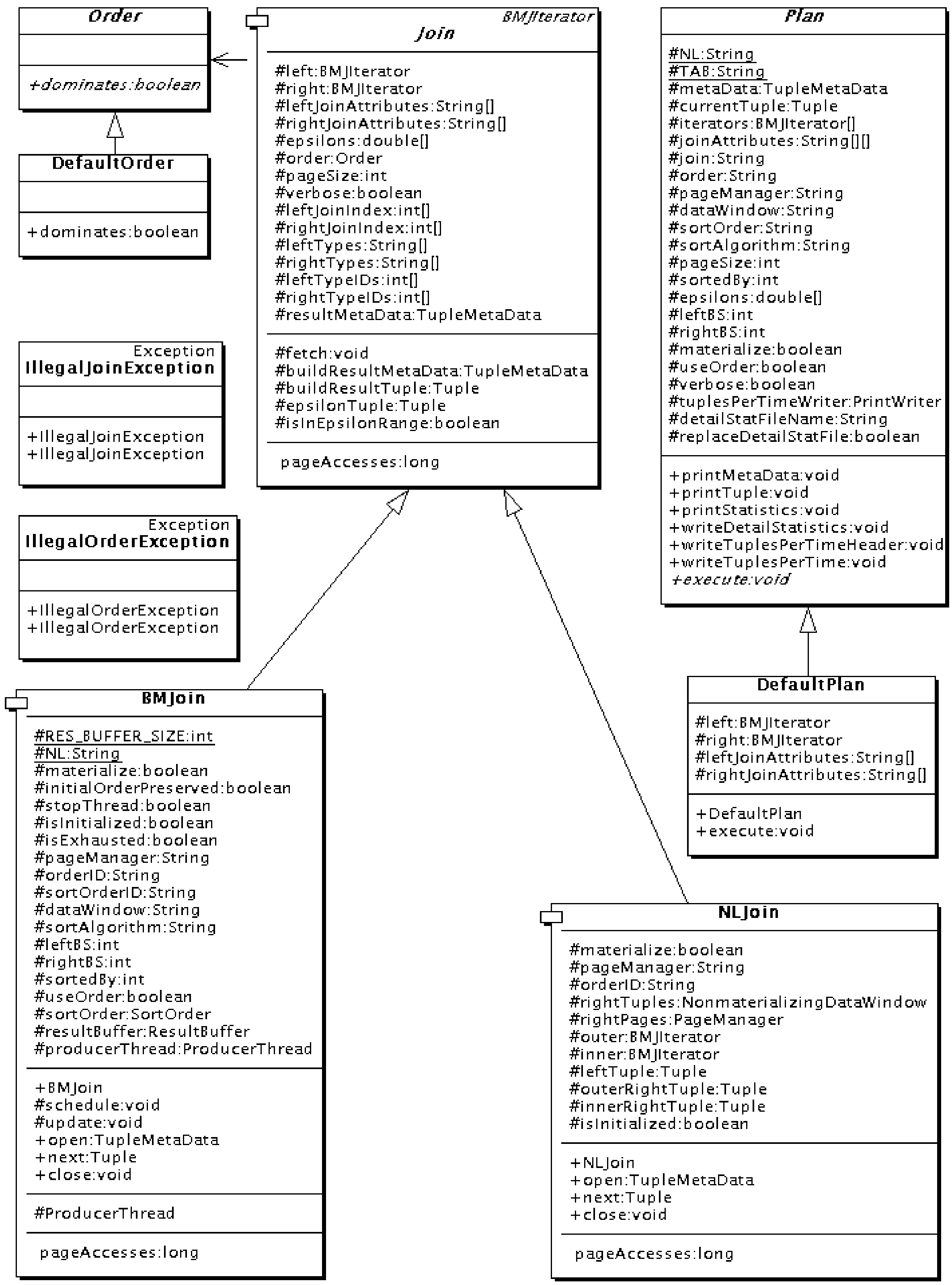
B.1 Paket bmj.test



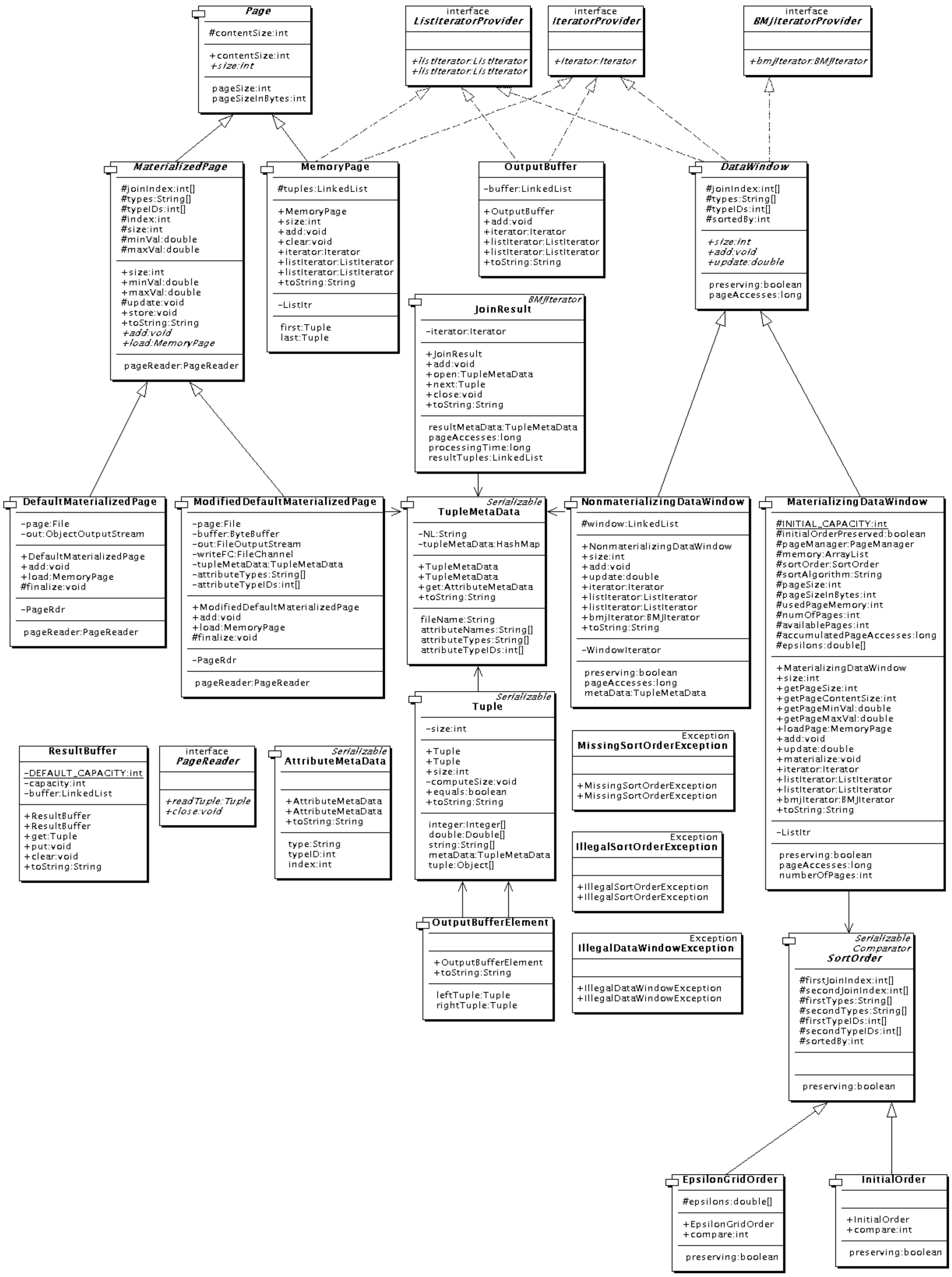
B.2 Paket `bmj.io`



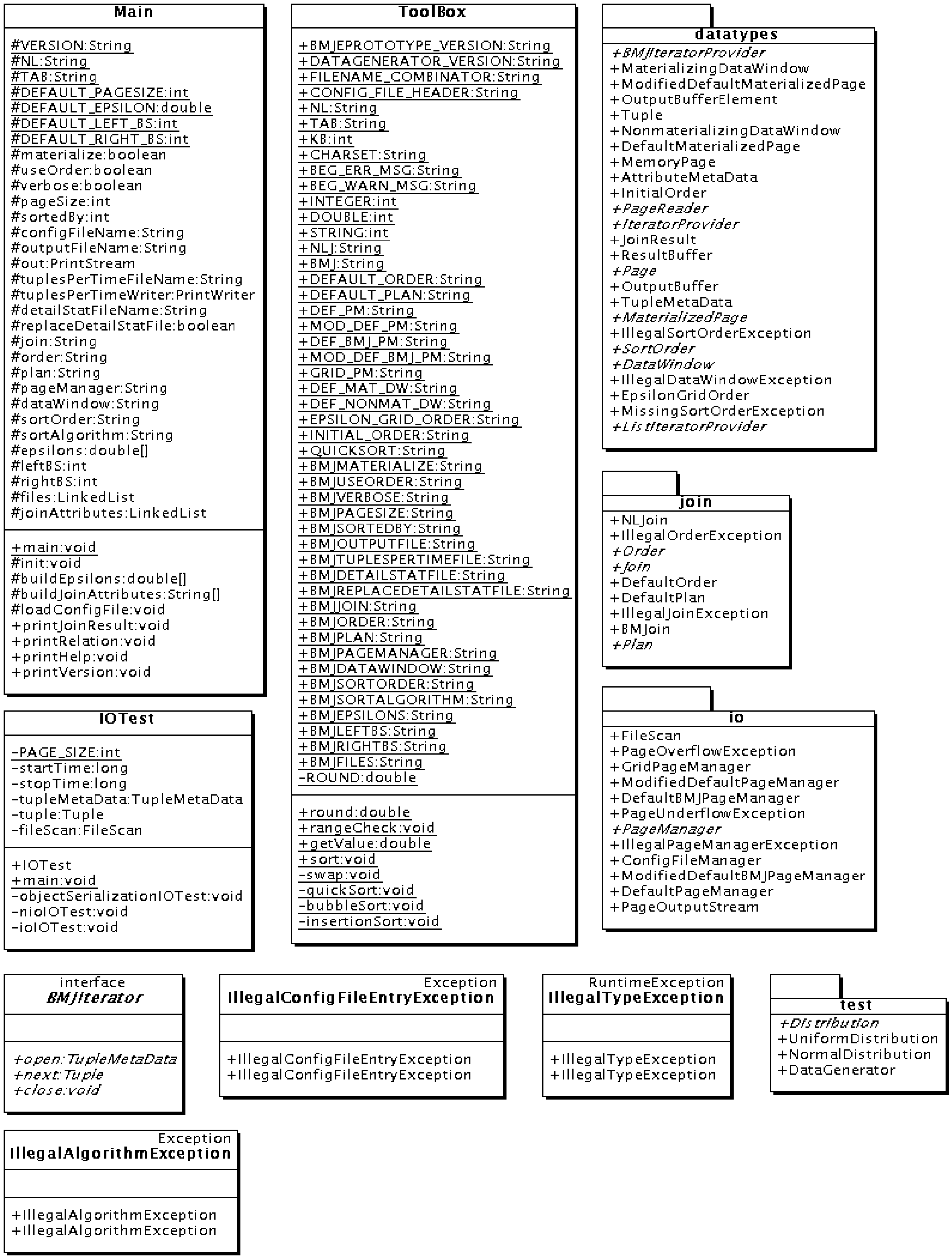
B.3 Paket bmj.join



B.4 Paket `bmj.datatypes`



B.5 Paket bmj


Main
#VERSION:String
#NL:String
#TAB:String
#DEFAULT_PAGESIZE:int
#DEFAULT_EPSILON:double
#DEFAULT_LEFT_BS:int
#DEFAULT_RIGHT_BS:int
#materialize:boolean
#useOrder:boolean
#verbose:boolean
#pageSize:int
#sortedBy:int
#configFileName:String
#outputFileName:String
#out:PrintStream
#tuplesPerTimeFileName:String
#tuplesPerTimeWriter:PrintWriter
#detailStatFileName:String
#replaceDetailStatFile:boolean
#join:String
#order:String
#plan:String
#pageManager:String
#dataWindow:String
#sortOrder:String
#sortAlgorithm:String
#epsilons:double[]
#leftBS:int
#rightBS:int
#files:LinkedList
#joinAttributes:LinkedList
+main:void
#init:void
#buildEpsilons:double[]
#buildJoinAttributes:String[]
#loadConfigFile:void
+printJoinResult:void
+printRelation:void
+printHelp:void
+printVersion:void
ToolBox
+BMJEPROTOTYPE_VERSION:String
+DATAGENERATOR_VERSION:String
+FILENAME_COMBINATOR:String
+CONFIG_FILE_HEADER:String
+NL:String
+TAB:String
+KB:int
+CHARSET:String
+BEG_ERR_MSG:String
+BEG_WARN_MSG:String
+INTEGER:int
+DOUBLE:int
+STRING:int
+NLJ:String
+BMJ:String
+DEFAULT_ORDER:String
+DEFAULT_PLAN:String
+DEF_PM:String
+MOD_DEF_PM:String
+DEF_BMJ_PM:String
+MOD_DEF_BMJ_PM:String
+GRID_PM:String
+DEF_MAT_DW:String
+DEF_NONMAT_DW:String
+EPSILON_GRID_ORDER:String
+INITIAL_ORDER:String
+QUICKSORT:String
+BMJMATERIALIZE:String
+BMJUSEORDER:String
+BMJVERBOSE:String
+BMJPAGESIZE:String
+BMJSORTEDBY:String
+BMJOUTPUTFILE:String
+BMJTUPLESPERTIMEFILE:String
+BMJDETAILSTATFILE:String
+BMJREPLACEDETAILSTATFILE:String
+BMJJOIN:String
+BMJORDER:String
+BMJPLAN:String
+BMJPAGEMANAGER:String
+BMJDATAWINDOW:String
+BMJSORTORDER:String
+BMJSORTALGORITHM:String
+BMJEPSILONS:String
+BMJLEFTBS:String
+BMJRIGHTBS:String
+BMJFILES:String
-ROUND:double
+round:double
+rangeCheck:void
+getValue:double
+sort:void
-swap:void
-quickSort:void
-bubbleSort:void
-insertionSort:void
datatypes
+BMJIteratorProvider
+MaterializingDataWindow
+ModifiedDefaultMaterializedPage
+OutputBufferElement
+Tuple
+NonmaterializingDataWindow
+DefaultMaterializedPage
+MemoryPage
+AttributeMetaData
+InitialOrder
+PageReader
+IteratorProvider
+JoinResult
+ResultBuffer
+Page
+OutputBuffer
+TupleMetaData
+MaterializedPage
+IllegalSortOrderException
+SortOrder
+DataWindow
+IllegalDataWindowException
+EpsilonGridOrder
+MissingSortOrderException
+ListIteratorProvider
join
+NLJoin
+IllegalOrderException
+Order
+Join
+DefaultOrder
+DefaultPlan
+IllegalJoinException
+BMJoin
+Plan
IOTest
-PAGE_SIZE:int
-startTime:long
-stopTime:long
-tupleMetaData:TupleMetaData
-tuple:Tuple
-fileScan:FileScan
+IOTest
+main:void
-objectSerializationIOTest:void
-nioIOTest:void
-ioIOTest:void
io
+FileScan
+PageOverflowException
+GridPageManager
+ModifiedDefaultPageManager
+DefaultBMJPageManager
+PageUnderflowException
+PageManager
+IllegalPageManagerException
+ConfigFileManager
+ModifiedDefaultBMJPageManager
+DefaultPageManager
+PageOutputStream
interface
BMJIterator
+open:TupleMetaData
+next:Tuple
+close:void
Exception
IllegalConfigFileEntryException
+IllegalConfigFileEntryException
+IllegalConfigFileEntryException
RuntimeException
IllegalTypeException
+IllegalTypeException
+IllegalTypeException
test
+Distribution
+UniformDistribution
+NormalDistribution
+DataGenerator
Exception
IllegalAlgorithmException
+IllegalAlgorithmException
+IllegalAlgorithmException

Literaturverzeichnis

[AMO93] AHUJA, R. K. ; MAGNANTI, T. L. ; ORLIN, J. B.: *Network Flows: Theory, Algorithms, and Applications.* New Jersey : Prentice Hall, Inc., 1993. – ISBN 0–13–617549–X

[BBD+02] BABCOCK, B. ; BABU, S. ; DATAR, M. ; MOTWANI, R. ; WIDOM, J.: Models and Issues in Data Stream Systems. In: *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '02).* Madison, WI, USA, Juni 2002. – ISBN 1–58113–507–6, S. 1–16

[BBKK01] BÖHM, C. ; BRAUNMÜLLER, B. ; KREBS, F. ; KRIEGEL, H.-P.: Epsilon Grid Order: An Algorithm for the Similarity Join on Massive High-Dimensional Data. In: *ACM SIGMOD Record (ACM Special Interest Group on Management of Data)* 30 (2001), Juni, Nr. 2, S. 379–388. – ISSN 0163–5808

[BK01] BÖHM, C. ; KRIEGEL, H.-P.: A Cost Model and Index Architecture for the Similarity Join. In: *Proceedings of the 17th IEEE International Conference on Data Engineering (ICDE '01).* Heidelberg, Germany, April 2001. – ISBN 0–7695–1001–9, S. 411–420

[BKS01] BÖRZSÖNYI, S. ; KOSSMANN, D. ; STOCKER, K.: The Skyline Operator. In: *Proceedings of the 17th IEEE International Conference on Data Engineering (ICDE '01).* Heidelberg, Germany, April 2001. – ISBN 0–7695–1001–9, S. 421–432

[CCC+02] CARNEY, D. ; CETINTEMEL, U. ; CHERNIACK, M. ; CONVEY, C. ; LEE, S. ; SEIDMAN, G. ; STONEBREAKER, M. ; TATBUL, N. ; ZDONIK, S.: Monitoring Streams - A New Class of Data Management Applications. In: *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02).* Hong Kong, China, August 2002. – ISBN 1–55860–869–9, S. 215–226

[CF02] CHANDRASEKARAN, S. ; FRANKLIN, M. J.: Streaming Queries over Streaming Data. In: *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02).* Hong Kong, China, August 2002. – ISBN 1–55860–869–9, S. 203–214

[CK97a] CAREY, M. J. ; KOSSMANN, D.: On saying "Enough already!" in SQL. In: *ACM SIGMOD Record (ACM Special Interest Group on Management of Data)* 26 (1997), Juni, Nr. 2, S. 219–230. – ISSN 0163–5808

[CK97b] Carey, M. J. ; Kossmann, D.: Processing Top N and Bottom N Queries. In: *IEEE Data Engineering Bulletin* 20 (1997), September, Nr. 3, S. 12–19

[CK98] Carey, M. J. ; Kossmann, D.: Reducing the Braking Distance of an SQL Query Engine. In: *Proceedings of the 24th International Conference on Very Large Databases (VLDB '98).* New York, NY, USA, August 1998. – ISBN 1–55860–566–5, S. 158–169

[CP00] Ciaccia, P. ; Patella, M.: PAC Nearest Neighbor Queries: Approximate and Controlled Search in High-Dimensional and Metric Spaces. In: *Proceedings of the 16th IEEE International Conference on Data Engineering (ICDE '00).* San Diego, CA, USA, März 2000. – ISBN 0–7695–0506–6, S. 244–255

[Dad96] Dadam, P.: *Verteilte Datenbanken und Client/Server-Systeme: Grundlagen, Konzepte, Realisierungsformen.* New York, Berlin, etc. : Springer-Verlag, 1996. – ISBN 3–540–61399–4

[Edm65] Edmonds, J.: Paths, Trees and Flowers. In: *Canadian Journal of Mathematics* 17 (1965), S. 449–467. – ISSN 0008–414X

[Epp98] Eppstein, D.: Fast Hierarchical Clustering and Other Applications of Dynamic Closest Pairs. In: *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms.* San Francisco, CA, USA, Januar 1998. – ISBN 0–89871–410–9, S. 619–628

[FTTF01] Filho, R. ; Traina, A. ; Traina Jr., C. ; Faloutsos, C.: Similarity Search without Tears: The OMNI-Family of All-Purpose Access Methods. In: *Proceedings of the 17th IEEE International Conference on Data Engineering (ICDE '01).* Heidelberg, Germany, April 2001. – ISBN 0–7695–1001–9, S. 623–632

[HAK00] Hinneburg, A. ; Aggarwal, C. C. ; Keim, D. A.: What is the Nearest Neighbor in High Dimensional Spaces? In: *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00).* Cairo, Egypt, September 2000. – ISBN 1–55860–715–3, S. 506–515

[Har02] Harrison, J. *Sorting Algorithms.* `http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html`. September 2002

[Hel97] Hellerstein, J. M.: Online processing redux. In: *IEEE Data Engineering Bulletin* 20 (1997), September, Nr. 3, S. 20–29

[ILW+00] Ives, Z. G. ; Levy, A. Y. ; Weld, D. S. ; Florescu, D. ; Friedman, M.: Adaptive Query Processing for Internet Applications. In: *IEEE Data Engineering Bulletin* 23 (2000), Juni, Nr. 2, S. 19–26

[Jun99] Jungnickel, D.: *Graphs, Networks and Algorithms.* New York, Berlin, etc. : Springer-Verlag, 1999. – ISBN 3–540–63760–5

[KE01] KEMPER, A. ; EICKLER, A.: *Datenbanksysteme – Eine Einführung*. 4., überarbeitete und erweiterte Auflage. München, Wien : R. Oldenbourg Verlag, 2001. – ISBN 3–486–25706–4

[Kie02] KIESSLING, W.: Foundations of Preferences in Database Systems. In: *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*. Hong Kong, China, August 2002. – ISBN 1–55860–869–9, S. 311–322

[KLP75] KUNG, H. T. ; LUCCIO, F. ; PREPARATA, F. P.: On Finding the Maxima of a Set of Vectors. In: *Journal of the ACM* 22 (1975), Oktober, Nr. 4, S. 469–476. – ISSN 0004–5411

[KRR02] KOSSMANN, D. ; RAMSAK, F. ; ROST, S.: Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In: *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*. Hong Kong, China, August 2002. – ISBN 1–55860–869–9, S. 275–286

[KS02a] KEMPER, A. ; STEGMAIER, B.: Evaluating Bestmatch-Joins on Streaming Data / Universität Passau. 94030 Passau, Germany, Dezember 2002 (MIP-0204). – Technical Report

[KS02b] KEMPER, A. ; STEGMAIER, B.: *The Bestmatch-Join Operator*. 2002. – Unveröffentlicht

[LLL01] LIAO, S. ; LOPEZ, M. ; LEUTENEGGER, S.: High Dimensional Similarity Search with Space Filling Curves. In: *Proceedings of the 17th IEEE International Conference on Data Engineering (ICDE '01)*. Heidelberg, Germany, April 2001. – ISBN 0–7695–1001–9, S. 615–622

[MF02] MADDEN, S. ; FRANKLIN, M. J.: Fjording the Stream: An Architecture for Queries over Streaming Sensor Data. In: *Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE '02)*. San José, CA, USA, März 2002. – ISBN 0–7695–1531–2, S. 555–566

[NHS84] NIEVERGELT, J. ; HINTERBERGER, H. ; SEVCIK, K. C.: The Grid File: An Adaptable Symmetric Multikey File Structure. In: *ACM Transactions on Database Systems* 9 (1984), März, Nr. 1, S. 38–71. – ISSN 0362–5915

[OW02] OTTMANN, T. ; WIDMAYER, P.: *Algorithmen und Datenstrukturen*. 4. Auflage. Heidelberg, Berlin : Spektrum Akademischer Verlag, 2002. – ISBN 3–8274–1029–0

[PS85] PREPARATA, F. P. ; SHAMOS, M. I.: *Computational Geometry: An Introduction*. New York, Berlin, etc. : Springer-Verlag, 1985. – ISBN 0–387–96131–3

[RKV95] ROUSSOPOULOS, N. ; KELLEY, S. ; VINCENT, F.: Nearest Neighbor Queries. In: *ACM SIGMOD Record (ACM Special Interest Group on Management of Data)* 24 (1995), Juni, Nr. 2, S. 71–79. – ISSN 0163–5808

[Sed88] Sedgewick, R.: *Algorithms.* 2. Auflage. Reading, Massachusetts : Addison-Wesley Publishing Company, 1988. – ISBN 0–201–06673–4

[SMFH01] Shah, M. A. ; Madden, S. ; Franklin, M. J. ; Hellerstein, J. M.: Java Support for Data-Intensive Systems: Experiences Building the Telegraph Dataflow System. In: *ACM SIGMOD Record (ACM Special Interest Group on Management of Data)* 30 (2001), Dezember, Nr. 4, S. 103–114. – ISSN 0163–5808

[SU00] Sack, J.-R. (Hrsg.) ; Urrutia, J. (Hrsg.): *Handbook of Computational Geometry.* Amsterdam, Lausanne, New York, Oxford, Shannon, Singapore, Tokyo : ELSEVIER, 2000

[Sun02a] Sun Microsystems, Inc. *Java 2 Platform, Standard Edition, v1.4.0 API Specification.* `http://java.sun.com/j2se/1.4/docs/api/index.html`. Oktober 2002

[Sun02b] Sun Microsystems, Inc. *JSR 51: New I/O APIs for the Java Platform.* `http://jcp.org/jsr/detail/051.jsp`. Oktober 2002

[Sun02c] Sun Microsystems, Inc. *New I/O APIs.* `http://java.sun.com/j2se/1.4/docs/guide/nio/index.html`. Oktober 2002

[TEO01] Tan, K.-L. ; Eng, P.-K. ; Ooi, B. C.: Efficient Progressive Skyline Computation. In: *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01).* Rome, Italy, September 2001. – ISBN 1–55860–804–4, S. 301–310

[TM02] Tucker, P. ; Maier, D.: Exploiting Punctuation Semantics in Data Streams. In: *Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE '02).* San José, CA, USA, März 2002. – ISBN 0–7695–1531–2, S. 279

Glossar

ϵ-BMJ Bestmatch-Join mit Einschränkungen

ϵ-FOBMJ Full-Outer-Bestmatch-Join mit Einschränkungen

ϵ-LOBMJ Left-Outer-Bestmatch-Join mit Einschränkungen

ϵ-ROBMJ Right-Outer-Bestmatch-Join mit Einschränkungen

anti-korrelierte Verteilung Verteilung, bei der sich die Werte bestimmter Attribute bzw. Dimensionen entgegengesetzt zu den Werten bestimmter anderer Dimensionen entwickeln.

BMJ Bestmatch-Join

Closest-Point Problem aus der algorithmischen Geometrie, bei dem zu einem gegebenen Punkt derjenige Punkt gesucht wird, der vom Ausgangspunkt den geringsten Abstand hat. Siehe auch Nearest-Neighbor.

EGO Epsilon-Grid-Ordnung. Spezielle Ordnung, die in einer leicht modifizierten Variante bei der fensterbasierten Methode zur Berechnung von besten Zuordnungen von Objekten zweier Datenströme eingesetzt wird, um die Anzahl an Ladevorgängen von Seiten des Hintergrundspeichers möglichst gering zu halten.

FOBMJ Full-Outer-Bestmatch-Join

Grid-File Eine dynamische Datenstruktur.

Grid-Materialisierungsvariante Verfahren zur Materialisierung von Tupeln im Rahmen der Methoden zur Berechnung von besten Zuordnungen auf Datenströmen, das ein mehrdimensionales Array zur Verwaltung von Referenzobjekten verwendet und im Vergleich zur Standard-Variante das Mischen neuer Datenelemente mit bereits materialisierten Datenbeständen einspart. Dadurch kann die Anzahl an Lese- und Schreibzugriffen auf Seiten des Hintergrundspeichers bei der Materialisierung reduziert werden.

Java NIO Neues API-Paket im JDK 1.4.0 von Sun Microsystems, Inc. zur Programmierung von I/O-Operationen in Java. Zeichnet sich durch verbesserte Effizienz und erhöhten Funktionsumfang gegenüber dem herkömmlichen Paket `java.io` aus.

korrelierte Verteilung Verteilung, bei der die Werte bestimmter Attribute bzw. Dimensionen mit den Werten bestimmter anderer Dimensionen korrelieren, also ähnliche Ausprägungen besitzen.

LOBMJ Left-Outer-Bestmatch-Join

LRU Least Recently Used. Seitenverdrängungsstrategie, bei der bei Bedarf stets die Seite aus dem Hauptspeicher verdrängt wird, deren letzter Zugriff am längsten zurückliegt.

Matching Teilmenge der Kantenmenge eines Graphen, so dass keine zwei Kanten dieser Teilmenge mit demselben Knoten inzident sind.

MRU Most Recently Used. Seitenverdrängungsstrategie, bei der bei Bedarf stets die Seite aus dem Hauptspeicher verdrängt wird, deren letzter Zugriff am kürzesten zurückliegt.

Nearest-Neighbor Problem der Bestimmung nächster Nachbarn, z. B. von Punkten in der algorithmischen Geometrie. Siehe auch Closest-Point.

Normalverteilung Verteilung gemäß der gaußschen Glockenkurve. Die Verteilung ist durch die Parameter Mittelwert und Standardabweichung beeinflussbar.

Objekt-Serialisierung Mechanismus der Java API zum Speichern und Laden von Objekten auf dem bzw. vom Hintergrundspeicher.

Punktierungen Spezielle Metadaten in einem Datenstrom, die verwertbare Aussagen über den Zustand des Stroms enthalten.

Quicksort In der Praxis effizientes in-place Sortierverfahren. Wird für die Sortierung von Tupeln gemäß EGO eingesetzt.

ROBMJ Right-Outer-Bestmatch-Join

Skyline Die Teilmenge aller Punkte einer Menge von Punkten, die gemäß einer vorgegebenen Vergleichsordnung von keinem anderen Punkt der Menge dominiert werden. Die Skyline auf dem Kreuzprodukt zweier Punktmengen entspricht dem Ergebnis des Bestmatch-Joins auf diesen beiden Mengen.

Standard-Materialisierungsvariante Einfaches Verfahren zur Materialisierung von Tupeln im Rahmen der Methoden zur Berechnung von besten Zuordnungen auf Datenströmen, das auf dem Ansatz des externen Sortierens basiert.

Top N Anfragen Datenbankanfragen, die nur die N besten Ergebnistupel liefern. Die Einschränkung kann zur Effizienzsteigerung der Berechnung ausgenutzt werden.

Uniforme Verteilung Auch Gleichverteilung genannt. Verteilung, bei der jeder mögliche Wert mit der gleichen Wahrscheinlichkeit auftritt.

Index

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich diese Diplomarbeit selbständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle wörtlich oder sinngemäß übernommenen Ausführungen wurden als solche gekennzeichnet. Weiterhin erkläre ich, dass ich diese Arbeit in gleicher oder ähnlicher Form nicht bereits einer anderen Prüfungsbehörde vorgelegt habe.

Passau, den 15. Januar 2003 ....

Richard Kuntschke