Abschlußbericht

Seminar „Advanced Topics in Databases“ im Wintersemester 2008/2009

März 2009

Verantwortlich:
Prof. Felix Naumann
Alexander Albrecht
Jens Bleiholder

Lehrstuhl Informationssysteme
Hasso-Plattner-Institut für Softwaresystemtechnik
Universität Potsdam
Prof.-Dr.-Helmert Str. 2-3
14482 Potsdam

Kurzfassung

Das Seminar behandelt wichtige Themen der Datenbank-Forschung und gibt einen tieferen Einblick, als es in den grundlegenden Datenbank-Vorlesungen möglich ist. Dabei werden nicht nur hochaktuelle sondern auch grundlegende Themen behandelt.

Die „Readings in Database Systems“ von Joseph M. Hellerstein und Michael Stonebraker bieten eine exzellente Sammlung bedeutender Artikel im Bereich Datenbanken („best of“) und bilden die Grundlage für unser Seminar. Es werden die folgenden Themen und Artikel im Seminar behandelt:

Query Processing:

- P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, T. Price. *Access Path Selection in a Relational Database Management System.* Proceedings of SIGMOD Conference, 1979, 23-34.
- L. Shapiro. *Join Processing in Database Systems with Large Main Memories.* ACM Transactions on Database Systems, 11(3), 1986, 239-264.

Data Storage and Access Methods / Transaction Management:

- J. Gray, G. Graefe. *The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb.* SIGMOD Record, 26(4), 1997, 63-68.
- H. Kung, J. Robinson. *On Optimistic Methods for Concurrency Control.* Proceedings of VLDB, 1979, 351.

Extensible Systems / Web Services and Databases:

- J. Hellerstein, J. Naughton, A. Pfeffer. *Generalized Search Trees for Database Systems.* Proceedings of VLDB, 1995, 562-573.
- S. Brin, L. Page. *The Anatomy of a Large-Scale Hypertextual Web Search Engine.* Computer Networks, 30(1-7), 1998, 107-117.

Data Warehousing / Data Mining:

- J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, H. Pirahesh. *Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals.* Data Mining and Knowledge Discovery, 1(1), 1997, 29-53.
- T. Zhang, R. Ramakrishnan, M. Livny. *BIRCH: An Efficient Data Clustering Method for Very Large Databases.* Proceedings of SIGMOD Conference, 1996, 103-114.

Stream-Based Data Management:

- P. Seshadri, M. Livny, R. Ramakrishnan. *The Design and Implementation of a Sequence Database System.* Proceedings of VLDB, 1996, 99-110.

Inhaltsverzeichnis

Teil I

Query Processing

Edgar Näther
edgar.naether@student.hpi.uni-potsdam.de

1

Access Path Selection in a Relational Database Managment System

Inhaltsangabe

1.1 Vorwort

Diese Ausarbeitung soll einen Überblick über die Ermittlung des Ausführungsplans zu einer gegebenen Anfrage mit Hilfe eines kostenbasierten Optimierers in einem relationalen Datenbankmanagementsystem geben. Beispielhaft wird die Anfrageoptimierung in System R erläutert. Anschließend wird der Ansatz mit anderen Optimierungsstrategien verglichen. Schließlich wird eine kurze zeitliche Entwicklung des Ansatzes dargestellt. Die Basis dieser Ausarbeitung ist das Paper „Access path selection in a relational database management system“ von P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, und T. G. Price [50].

1.2 Anfrageoptimierung in System R

System R war das erste experimentelle, relationale Datenbankmanagementsystem(RDBMS). Bis dahin waren hierachische und netzwerkartige Datenbanken verbreitet. Die Anfrage war sehr implementierungsnah, nicht deklarativ. Zugriffswege und Optimierungen mussten vom Programmierer festgelegt werden. Die im Zuge der Forschungsarbeiten an System R entwickelte Abfragesprache SQL war die erste deklarative Abfragesprache. System R war das erste

RDBMS, welches eine solche deklarative Anfragesprache unterstützte. Aufgabe des RDBMS ist es nun einen Ausführungsplan für eine gegebene Anfrage zu ermitteln. Dabei wird die Anfrage in einen Operatorbaum überführt. Anschließend wird der Baum logisch und physisch optimiert. Bei der physischen Optimierung geht es darum, zu den logischen Operatoren(z. B. Join) die günstigsten, physischen Algorithmen(z. B. Sorted-Merge-Join) und so schließlich den günstigsten Ausführungsplan zu finden.

Meist gibt es sehr viele Möglichkeiten Anfragen auszuführen. Um auf Tabellen zuzugreifen hat man 2 prinzipielle Möglichkeiten: Table Scan und Index Scan. Dabei ist ein Index Scan keines Wegs immer schneller. Welcher Zugriffsweg effizienter ist hängt von vielen Faktoren ab: Größe der Tabelle, Selektivität der Anfrage, Art des Indexes (Clustered vs. Nicht-Clustered). Für logische Operatoren existieren meist mehrere physische Operatoren. Beispielsweise für den Join gibt es den Nested Loop Join, Sorted Merge Join, Hash Join etc. . Dadurch ergeben sich viele mögliche Anfragepläne, der Suchraum ist sehr groß.

Der Algorithmus zur Ermittlung des günstigsten Ausführungsplans in System R baut per Bottom-up Strategie unter alleiniger Berücksichtigung von links-tiefen Bäumen einen Suchbaum auf. Die erste Ebene repräsentieren dabei die günstigsten Zugriffswege für jede Tabelle im FROM-Teil der Anfrage. Dabei können zu einer Tabelle durchaus mehrere günstigste Zugriffswege betrachtet werden, da System R sogenannte *Interesting Orders* berücksichtigt (siehe 1.2.3 Interesting Orders), sowie den günstigsten Zugriff ohne Berücksichtigung einer Sortierung. Im Falle von Joins wird nun zu jeder Relation pro ermittelten Zugriffsweg jede andere mögliche Relation gejoint unter Berücksichtigung verschiedener, möglicher Join-Algorithmen. Dabei werden Kreuzprodukte allerdings vermieden. Sind mehrere Joins erforderlich, wiederholt sich dieser Schritt. Für jeden Teilschritt werden die Ergebniskardinalität, die verwendeten Zugriffswege und Algorithmen, sowie die Sortierreihenfolge gespeichert. So entsteht ein Suchbaum, welcher einer nahezu erschöpfenden Suche gleicht. Für alle ermittelten Ausführungspläne werden die Gesamtkosten berechnet und der günstigste wird schließlich ausgeführt.

1.2.1 Kostenbasierte Optimierung

In System R ist eine Optimierungskomponente integriert, welche Zugriffswege auf Relationen und physische Operatoren ermittelt und auf Basis eines Kostenmodells Optimierungen vornimmt. Dabei werden sowohl I/O-Kosten als auch CPU-Kosten berücksichtigt.

Kosten = I/O-Kosten + W * #Tupel

Tabelle 1.1: Allgemeine Formel zur Kostenvorhersage

Die I/O-Kosten werden durch die zu ladenden Speicherseiten bestimmt. W ist ein Gewichtungsfaktor mit dem das Kostenmodell an eine Hardwarekonfiguration angepasst werden kann und der das durchschnittliche Verhältnis des Aufwandes für einen Aufruf des Zugriffssystems zu einem Seitenzugriff auf den externen Speicher angibt. #Tupel ist die geschätzte Anzahl von Tupeln (Kardinalität des Anfrageergebnisses). Dabei spiegelt W * #Tupel die CPU-Kosten wieder. Dabei setzen sich die I/O Kosten je nach Zugriffsart unterschiedlich zusammen. Die genauen Kostenformeln für jede Zugriffsart können im Paper[50] nachgeschlagen werden.

1.2.2 Selektivitätsfaktoren und Statistiken

Um genauere Kostenvorhersagen treffen zu können, wurde das Konzept der Selektivitätsfaktoren ($0 \geq SF \leq 1$) eingeführt. Zu jeder Relation werden Statistiken geführt, z.B. über die Kardinalität der Relation, Kardinalität des Index(Anzahl verschiedener Schlüssel) etc. . Auf Basis dieser Statistiken schätzt der Optimierer mit Hilfe der Selektivitätsfaktoren die Kardinalität des Anfrageergebnisses ab, welche Einfluss auf die Kosten hat. Dabei sind Selektivitätsfaktoren für jeden möglichen Typ von Bedingung im WHERE-BLOCK definiert. Sind keine Statistiken verfügbar wird ein Standardwert verwendet.

Bedingung im WHERE-BLOCK	Selektivitätsfaktor
Feld = Wert	SF = $\frac{1}{\#Schlüssel(Index_{Feld})}$ falls ein Index auf dem Attribut existiert. Annahme: gleichmäßige Wertverteilung der Tupel bezüglich der Indexwerte SF = $\frac{1}{10}$ sonst, wenn kein Index existiert.

Tabelle 1.2: Beispiel Selektivitätsfaktor

1.2.3 Interesting Orders

Ähnlich wie bei dem Konzept der dynamischen Programmierung verfolgt der Optimierungsansatz in System R eine Bottom-up Strategie bei der Ermittlung des günstigstens Zugriffs für Joins. Zuerst werden Zugriffswege für jede einzelne Relation berechnet, anschließend Zugriffskosten für jedes Paar von Relationen (Join) usw. . Schließlich entsteht so ein Suchbaum, aus dem der günstigste Zugriffsweg ermittelt wird. Jedoch wird nicht nur der generell günstigste Zugriffsweg für jede Relation oder für jeden Join berücksichtigt, sondern für jede *Interesting Order* wird der günstigste Zugriffsweg gespeichert. Dabei werden *Interesting Orders* durch GROUP-BY, ORDER-BY und durch jedes Joinattribut bestimmt. Joinattribute definieren auch *Interesting Orders*, da bei bereits sortierten Relationen effizientere Algorithmen, z. B. Sorted-Merge-Join, angewendet werden können. Somit werden nicht nur Kosten für Zugriffswege, sondern auch noch die Sortierreihenfolgen der Ergebnissrelationen gespeichert. Zusätzlich sind *Interesting Orders* von Interesse, da so eventuelle, spätere Sortierungen vermieden werden können, wenn bereits ein Zugriff die gewünschte Sortierung der Ergebnisrelation herbeiführt, z. B. Index-Scan. Des Weiteren werden noch die Kosten für den jeweils generell günstigsten Zugriff ohne Berücksichtigung von Sortierreihenfolgen ermittelt, da eine spätere Sortierung immer noch günstiger sein kann. Um den Suchbaum klein zu halten, werden Heuristiken angewendet. So werden Selektionen so früh wie möglich durchgeführt und frühe Kreuzprodukte bezüglich Joins verworfen.

1.3 Diskussion des Ansatzes

Strategien und Algorithmen, den optimalen Anfrageplan zu ermitteln, gibt es viele. Im nachfolgenden werden einige weitere Algorithmen vorgestellt. Einige schienen damals als nicht praxisrelevant, weil die Anfrageoptimierung an sich zu teuer war. Schließlich wird ein kurzer Vergleich zwischen den alternativen Anfrageoptimierungen und der Anfrageoptimierung in System R gezogen.

1.3.1 Alternative Anfrageoptimierungen

Eine Anfrageoptimierung die sich rein auf Heuristiken beschränkt, generiert nur einen Anfrageplan und kann somit nicht zwischen Alternativen entscheiden. Solch eine Optimierung ist nicht kostenorientiert, da keine geschätzten Kosten für den konkreten Fall berechnet werden und kann somit unter Umständen einen teueren Plan als Ergebnis hervorbringen. In der Praxis erwies sich daher eine solche Anfrageoptimierung als untauglich. Denkbar wäre auch auf Basis eines Kostenmodells einen vollständigen Suchbaum aufzubauen und aus diesem den günstigsten Plan zu ermitteln. Dies wäre zwar kostenorientiert, allerdings wäre der Suchraum zu groß und somit die Optimierung zu teuer. Die Kosten der Optimierungsphase würden die eventuell eingesparten Kosten des günstigsten Zugriffsplans dominieren.

Branch-and-Bound ist eine Optimierungsstrategie bei der heuristisch ein initialer Plan ermittelt und als momentan bester Zugriffsplan gemerkt wird. Anschließend werden für eine festgelegte Zeitspanne weitere mögliche Zugriffspläne ermittelt. Sind die Kosten des ermittelten Plans günstiger als ein Schwellwert(z. b. Kosten des aktuellen Zugriffsplans), ersetzt der neu gefundene Plan den aktuell gespeicherten Zugriffsplan. Dabei können auch nur Teilpläne berücksichtigt werden. Allerdings schien auch diese Strategie damals zu teuer zu sein. Zudem hier die Probleme darin liegen, die Zeitspanne für die Optimierung festzulegen.

Auch die *Hill-Climbing-Optimierungsstrategie* startet mit einem initialen Plan, der durch Heuristiken gewählt wird. Anschließend wird dieser Plan lokal modifiziert,solange die Gesamtkosten reduziert werden können. Wenn keine Modifikationen mehr zu Kostenreduzierungen führen, wird der ermittlete Zugriffsplan ausgeführt. Das Problem bei dieser Strategie ist, dass Kostenreduzierungen nur lokal betrachtet werden. Lokale Optimierungen müssen nicht zu globalen Optimierungen führen.

Die *Dynamische Programmierung* beruht auf dem Prinzip, dass die optimale Lösung aus optimalen Teillösungen besteht(Optimalitätsprinzip von Bellman). In dieser Optimierungsstrategie wird aber nur jeweils der günstigste Zugriffsweg für eine Teillösung bestimmt und gespeichert. Jedoch muss dieses Prinzip nicht immer im Bereich der Anfrageoptimierung von Datenbanken gelten, da möglicherweise ein Teilplan teurer sein kann, aber später Vorteile bringen kann.

Der gewählte Ansatz der Optimierung in System R ist eine Erweiterung der *dynamischen Programmierung*. Heuristiken werden verwendet, um den Suchbaum zu verkleinern und somit Optimierungszeit zu sparen. Kosten werden auf Basis eines Kostenmodells berechnet, um den günstigsten Zugriffsplan zu ermitteln. Durch die Berücksichtigung der *Interesting Orders* werden auch teurere Teilepläne berücksichtigt, die aber zu einem günstigeren Gesamtplan führen können, da z. B. eine spätere Sortierung vermieden wird, bzw. später günstigere Algorithmen eingesetzt werden können. Somit werden nicht nur lokale Optimierungen berücksichtigt. Die *Hill-Climbing-Strategie* neigt dazu oft in lokalen Optimierungen stecken zu bleiben und keine globale Gesamtoptimierung zu finden. Dieses Problem weist die Optimierungsstrategie in

System R nicht auf. Durch das Bottum-up Vorgehen werden im Grunde auch jeweils lokale Optimierungen vorgenommen, indem nur jeweils die günstigsten Zugriffe unter Berücksichtigung aller Algorithmen und *Interesting Orders* gespeichert werden. Da der Anfrageplan aber nach diesen Teilschritten noch nicht vollständig ist, können diese sich auch nicht negativ auf den Gesamtplan auswirken. Weitere Schritte zur Ermittlung vollständiger Anfragepläne bauen schließlich auf diesen Teilschritten auf. Zudem werden viele Anfragepläne miteinander verglichen und nicht nur ein initialer Anfrageplan bestmöglich optimiert.

1.3.2 Weitere Aspekte der Optimierungsstrategie

Der kostenbasierte Optimierer in System R erleichtert Anfragen an das Datenbankmanagementsystem. Der Programmierer muss nur noch anfragen, was er haben möchte und muss sich keine Gedanken mehr um Implementierungsdetails, Zugriffswege und Optimierungen machen. Daher ist die Nutzung des Datenbankmanagementsystems wesentlich einfacher. Anfragen können wesentlich schneller erarbeitet werden. Allerdings ist der Programmierer auch auf den Optimierer uns seine Ergebnisse angewiesen,da er selbst keinen primären Einfluss mehr auf die Optimierung hat. Somit muss der Optimierer verlässlich sein und möglichst performant Abfragen bearbeiten können.

Die Qualität der Kardinalitätsabschätzungen mit Selektivitätsfaktoren ist stark abhängig von der Qualität der Statistiken. Denn ohne abrufbare Statistiken werden Standardselektivitätsfaktoren gewählt, wodurch die Gefahr der Fehlabschätzung wesentlich erhöht wird. System R verwendet sehr einfache und wenige Statistiken, wodurch viele Annahmen bei der Berechnung der Selektivitäten getroffen werden müssen. Dadurch kann es zu sehr ungenauen Kostenberechnungen kommen. Allgemein muss erwähnt werden, dass es sich bei der Berechnung der Kosten nur um Schätzungen handelt und keineswegs exakte Kosten berechnet werden können. Daher ist es durchaus möglich, das der Optimierer in bestimmten Fällen nicht den optimalen Anfrageplan ermittelt.

1.4 Zeitliche Entwicklung

Die Forschungsarbeit an der Anfrageoptimierung in System R hatte großen Einfluss auf die Optimierungen in späteren, kommerziellen, relationalen Datenbanken. Die Kernideen - *Dynamische Programmierung*, *Interesting Orders*, kostenbasierte Optimierung - der Anfrageoptimierung sind heute in vielen Datenbankmanagementsystem verbreitet. Das Kostenmodell in Sytem R berücksichtigt nur I/O Kosten und CPU Kosten. In die Kostenberechnungen heutiger Optimierer fließen noch weitere Ressourcen ein, wie z. B. der zur Verfügung stehende und benötigte Hauptspeicher. In parallelen und verteilten Datenbanksystemen werden auch noch Kommunikationskosten berücksichtigt.

Die in System R verwendete Join-Optimierung ermittelt nur die optimale, lineare Join-Reihenfolge. Auf Änderungen im Suchraum, aufgrund neuer physischer Operatoren(z. B. neue Join-Methode) kann nicht flexibel reagiert werden. Moderne Architekturen von kostenbasierten Optimierern ermöglichen die Anpassung des Kostenberechnungsmodells (erweiterbare Optimierer) durch den Datenbankadministrator. Der erweiterbare Optimierer der Oracle 11.1 Datenbank bietet Möglichkeiten eigene Selektivitätsfaktoren, Statistikinformationen und Kostenfunktionen zu definieren, die bei der Anfrageoptimierung benutzt werden sollen. Dadurch sind heutige Optimierer wesentlich flexibler.

Die in System R verwendeten Statistiken waren sehr einfach. Heutzutage werden mehr Statistiken zusätzlich durch komplizierte Histogramme bereitgestellt, wodurch detailiertere und genauere Statistikinformationen berücksichtigt werden können. Dadurch können genauere Aussagen über Selektivitäten getroffen werden und somit genauere Kostenberechnungen mit geringeren Schätzfehlern. Meistens werden equi-height Histogramme verwendet.

Desweiteren wurden auch neue Optimierungsstrategien erforscht, zum Beispiel Anfrageoptimierungen mit Hilfe materialisierter Views. Neue Probleme in der Anfrageoptimierung sind hinzugekommen, z. B. die Optimierung von fuzzy-Queries. Das Problem einer optimalen und effizienten Anfrageoptimierung mit exakten Kostenmodellen ist nach wie vor schwierig und Forschungsthema.

Matthias Richly
matthias.richly@student.hpi.uni-potsdam.de

2

Join Processing in Database Systems with Large Main Memories

Inhaltsangabe

2.1 Einleitung

Das Paper *„Join Processing in Database Systems with Large Main Memories“* von Leonard Shapiro [55] beschreibt, wie Joins effizient auf Basis von Hashing-Algorithmen in Datenbankmanagementsystemen implementiert werden können. Dabei wird davon ausgegangen, dass eine gewisse, hinreichende Menge an Hauptspeicher zur Verfügung steht. „Hinreichend“ bedeutet in diesem Zusammenhang, dass der von den beschriebenen Algorithmen mindestens benötigte Hauptspeicher etwa der Quadratwurzel der Größe der kleineren der beteiligten Relationen entspricht, gemessen in physischen Blöcken.

In der vorliegenden Ausarbeitung sollen die Erkenntnisse des Papers erläutert werden. Dabei wird insbesondere auf die Funktionsweise der drei vorgestellten Join-Algorithmen und auf deren I/O-Kosten eingegangen. Kapitel 2.5 vergleicht die entstehenden Kosten. Dabei wird sich zeigen, dass einer der vorgestellten Algorithmen, der *Hybrid Hash-Join*, die anderen vorgestellten Hashing-basierten, sowie auch Sort-Merge-basierte Algorithmen dominiert. Im

Anschluss daran wird auf das Phänomen der Partitionsüberläufe eingegangen, welches in den davor liegenden Abschnitten zur Vereinfachung der Betrachtungen ignoriert wird.
Die Ausarbeitung endet mit einem Ausblick auf einige Entwicklungen, die nach der Veröffentlichung des Papers im Jahre 1986 zum Thema Hashing-basiertes Joinen stattgefunden haben.

Nachfolgend werden zunächst einige Grundlagen zu Hashing-basierten Join-Algorithmen geschaffen und die Notation und einige Annahmen der nachfolgenden Abschnitte eingeführt.

2.1.1 Einführung zu Hash-Join Algorithmen

Generell steckt hinter dem Einsatz von Hashing in Join-Algorithmen der Gedanke, die Tupel der beiden zu joinenden Relationen so zu partitionieren, dass mögliche Joinpartner ausschließlich in sich entsprechenden Buckets zu finden sind. Daher muss die Hashfunktion genau auf die Attribute angewendet werden, auf denen der Join ausgeführt werden soll. Tupel, die in den Joinattributen übereinstimmen, bekommen dann gleiche Hashwerte (und landen so in sich entsprechenden Buckets).

Die einfachste Form, Hashing in Join-Algorithmen einzusetzen, das sog. *klassische Hashing*, setzt voraus, dass die kleinere der zu joinenden Relationen in den für den Join zur Verfügung stehenden Hauptspeicher passt. In diesem Fall wird diese Relation eingelesen, wobei im Hauptspeicher eine Hashtabelle erzeugt wird. Dann wird die andere Relation eingelesen. Für jedes Tupel muss der Hashwert berechnet werden. Dann kann in dem entsprechenden Bucket der Hashtabelle nach Joinpartnern für das Tupel gesucht und ggf. der Join der beiden Tupel berechnet und ausgegeben werden. Die Tatsache, dass die Tupel der kleineren Relation im Hauptspeicher in der Hashtabelle nach ihren Hashwerten geordnet sind, ermöglicht einen schnelleren Zugriff auf mögliche Joinpartner und spart so CPU-Zeit. Wären die Tupel im Hauptspeicher unstrukturiert gespeichert, müsste für jedes Tupel der zweiten Relation jedes Tupel der ersten nach einem Match untersucht werden.

In dem häufiger auftretenden Fall, dass keine der zu joinenden Relationen vollständig in den Hauptspeicher passt, wird der schon angeführte Gedanke des Hashings, die Partitionierung, weitergeführt. Dabei werden die Relationen in disjunkte Teilmengen partitioniert, und zwar so, dass das Joinergebnis durch Joinen sich entsprechender Partitionen berechnet werden kann. Die Partitionierung wird dabei durch Teilung des Wertebereichs der verwendeten Hashfunktion h in Teilmengen $H_1, \ldots, H_n$ erreicht. Ein Tupel t landet in Partition P_i, wenn $h(t) \in H_i$. Generell ist das Ziel der Partitionierung, dass die Partitionen der kleineren Relation in den verfügbaren Hauptspeicher passen. Die Anzahl der Partitionen n und die Mengen $H_1, \ldots, H_n$ müssen dazu entsprechend gewählt werden.

2.1.2 Notation/Annahmen

Seien R und S zwei Relationen. Zu berechnen sei der Equijoin aus R und S. $|R|$ bezeichnet die Anzahl der Blöcke, die R belegt (entsprechend für S). R sei die kleinere Relation, also $|R| \leq |S|$. Weiterhin sei M der für den Join zur Verfügung stehende Hauptspeicher, $|M|$ bezeichnet entsprechend seine Größe gemessen in physischen Blöcken. Der Faktor F wird benutzt, um Werte zu bestimmen, die geringe Steigerungen anderer Werte sind. Zum Beispiel wird davon ausgegangen, dass eine Hashtabelle für die Relation R im Hauptspeicher $|R| * F$ Blöcke belegt.
Es gelten folgende Annahmen:

- R und S seien weder geordnet gespeichert, noch indiziert.
  Es kann daher keine dieser Eigenschaften ausgenutzt werden, um die Berechnung des Joins zu optimieren.
- $|M| \geq \sqrt{F * |R|}$ („Large Main Memory“)
  Der zur Verfügung stehende Hauptspeicher reicht somit gerade aus, um einen zweiphasigen Hash-Join Algorithmus anzuwenden.
- Bei der Betrachtung der I/O-Kosten werden das erste Einlesen der Relationen R und S, sowie die Ausgabe des Joinergebnisses nicht mitgezählt.
  Diese Kosten sind für alle Algorithmen gleich.
- Die Partitionierung der Relationen funktioniert perfekt; keine Partition wird größer als erwartet.
  Die Algorithmen erfordern, dass die Partitionierung Partitionen von bestimmter Größe liefert. Die obige Annahme vereinfacht die Betrachtung der Algorithmen. In Kapitel 2.6 wird auf die Behandlung von eventuell auftretenden Partitionsüberläufen eingegangen.
- Ein Tupel aus S joint höchstens mit einem Block von Tupeln aus R.
  Wenn R viele Tupel enthielte, die in den Joinattributen übereinstimmen, dann ist unter Umständen keine passende Partitionierung möglich.

2.2 Einfacher Hash-Join

Dieser Algorithmus ist eine iterative Erweiterung des oben beschriebenen klassischen Hashings, die es erlaubt, Relationen beliebiger Größe zu joinen. In jeder Iteration werden dabei gerade so viele Blöcke von R und S behandelt, wie es der verfügbare Hauptspeicher M zulässt. Der ggf. übrig bleibende Rest wird auf Disk gespeichert und als Eingabe für die nächste Iteration benutzt.

Die folgenden Schritte beschreiben den Algorithmus im Detail:

1. Wähle eine Hashfunktion h und eine Teilmenge H ihres Wertebereichs, so dass alle R-Tupel, die in H gehasht werden, gerade in $|M|/F$ Blöcke passen. M kann somit eine Hashtabelle für alle in dieser Iteration ausgewählten R-Tupel aufnehmen. (Zur Erinnerung: Eine Hashtabelle für $|M|/F$ Tupel-Blöcke braucht $|M|/F * F = |M|$ Blöcke.)
2. Lies R. Für jedes Tupel r: Wenn $h(r) \in H$, füge es in die Hashtabelle in M ein; sonst($h(r) \notin H$) schreibe es in eine extra Datei auf Disk.
3. Lies S. Für jedes Tupel s: Wenn $h(s) \in H$, suche in der Hashtabelle nach Matches, joine ggf. und gib das Ergebnistupel aus; sonst schreibe s in eine extra Datei auf Disk.
4. Wiederhole die Schritte 1 bis 3 für die übrig gebliebenen Tupel aus den beiden extra Dateien, solange bis keine R-Tupel mehr übrig bleiben.

Der Algorithmus benötigt

$$A = \left\lceil \frac{|R| * F}{|M|} \right\rceil$$

Iterationsschritte. In jedem Schritt werden $|M|/F$ R-Blöcke verarbeitet. Somit bleiben nach dem i-ten Iterationsschritt

$$|R| - i * \frac{|M|}{F}$$

R-Blöcke übrig und müssen auf Disk geschrieben werden. Bei Annahme der Gleichverteilung der Join-Attribute in R und S sind das für die Relation S $(|S|/|R|)$-mal so viele. Entsprechend ergeben sich die folgenden I/O-Kosten für den einfachen Hash-Join:

- Schreiben und wieder Einlesen der R-Tupel in/aus extra Datei

$$2 * \left(\sum_{i=1}^{A} |R| - i * \frac{|M|}{F} \right) = 2 * \left((A-1) * |R| - \frac{A * (A-1)}{2} * \frac{|M|}{F} \right)$$

- Schreiben und wieder Einlesen der S-Tupel in/aus extra Datei

$$2 * \left(\sum_{i=1}^{A} |S| - i * \frac{|M|}{F} * \frac{|S|}{|R|} \right) = 2 * \left((A-1) * |S| - \frac{A * (A-1)}{2} * \frac{|M|}{F} * \frac{|S|}{|R|} \right)$$

- insgesamt:

$$\underline{2 * \left((A-1) * (|R| + |S|) - \frac{A * (A-1)}{2} * \frac{|M|}{F} * \left(1 + \frac{|S|}{|R|} \right) \right)}$$

Dieser Algorithmus erzielt gute Ergebnisse, wenn große Teile von R (mehr als $|R| * F/2$) in den verfügbaren Hauptspeicher passen. In diesem Fall werden nur 2 Iterationen benötigt und der Großteil von R (und S) kann gleich in der ersten Iteration verarbeitet werden. Nur ein kleiner Teil muss für die zweite Iteration auf Disk geschrieben werden und erzeugt überhaupt I/O-Operationen (die bei den I/O-Kosten mitgezählt werden).
Wenn M dagegen nur geringe Teile von R aufnehmen kann, werden viele Iterationen benötigt. Große Teile von R und S müssen dann wiederholt auf Disk geschrieben und wieder eingelesen werden. Die I/O-Kosten steigen somit stark an.

2.3 GRACE Hash-Join

Im Gegensatz zu dem im letzten Kapitel vorgestellten Algorithmus, hat der GRACE Hash-Join genau zwei Phasen. In der ersten Phase werden R und S in korrespondierende Partitionen zerlegt, wobei entstehende R-Partitionen etwa die gleiche Größe bekommen. In der zweiten Phase werden dann korrespondierende Partitionen gejoint. Der Algorithmus benötigt mindestens $\sqrt{F * |R|}$ Blöcke Hauptspeicher. Wenn mehr Hauptspeicher für den Join zur Verfügung steht, wird dieser benutzt, um Teile der Partitionen im Speicher zu halten, so dass diese nicht auf Disk geschrieben und wieder eingelesen werden müssen.
Der Algorithmus läuft wie folgt ab:

1. Wähle eine Hashfunktion h und $\sqrt{F * |R|}$ Teilmengen ihres Wertebereichs, so dass alle R-Partitionen etwa gleich groß werden. Lege für jede Partition einen Ausgabepuffer in M an.

2. Lies R. Für jedes Tupel r: Berechne $h(r)$ und füge r in den entsprechenden Ausgabepuffer ein. Wenn ein Puffer voll ist, schreibe ihn auf Disk; am Ende schreibe alle Puffer auf Disk.

3. Lies S. Verfahre wie in 2. mit R.

   [An dieser Stelle ist die erste Phase abgeschlossen. Auf Disk befinden sich jetzt die einzelnen Partitionen von R und S.]

4. Für jede R-Partition R_i:

   (a) Lies R_i und erzeuge eine Hashtabelle davon in M.

   (b) Lies S_i. Für alle Tupel s: Berechne $h(s)$ und prüfe in der Hashtabelle auf Matches, berechne ggf. den Join und gib das Ergebnistupel aus.

Zuerst muss überprüft werden, dass alle R-Partitionen tatsächlich in M passen: Es werden genau $\left\lceil \sqrt{F * |R|} \right\rceil$ Partitionen angelegt (Das Aufrunden wurde bzw. wird an anderer Stelle zur Vereinfachung weggelassen). Alle Partitionen haben (ungefähr) die gleiche Größe. Somit belegt jede Partition

$$\frac{|R|}{\sqrt{F * |R|}} = \sqrt{\frac{|R|}{F}}$$

Blöcke. Eine Hashtabelle benötigt somit

$$F * \sqrt{\frac{|R|}{F}} = \sqrt{F * |R|}$$

Hauptspeicherblöcke. Das entspricht gerade dem in Abschnitt 2.1.2 festgelegtem Minimum.

Für den GRACE Hash-Join ergeben sich folgende I/O-Kosten:

- Schreiben der partitionierten Relationen auf Disk: $|R| + |S|$
- Einlesen der partitionierten Relationen von Disk: $|R| + |S|$
- I/O Einsparungen, wenn mehr als $\sqrt{F * |R|}$ Blöcke Hauptspeicher zur Verfügung stehen:
$$\min\left(|R| + |S|, |M| - \sqrt{F * |R|}\right) * 2$$
- insgesamt:
$$\underline{2 * \left((|R| + |S|) - \min\left(|R| + |S|, |M| - \sqrt{F * |R|}\right)\right)}$$

Der Algorithmus funktioniert gut, wenn M klein ist (nicht viel mehr als $\sqrt{F * |R|}$ Blöcke). Im Gegensatz zum einfachen Hash-Join wird das wiederholte Schreiben und wieder Einlesen von R und S vermieden. Wenn der zur Verfügung stehende Hauptspeicher dagegen groß ist, wird dieser allerdings nicht sehr effizient genutzt. Zum Beispiel ist, um I/O-Kosten von o zu erreichen, viel mehr Hauptspeicher nötig als beim einfachen Hash-Join.

2.4 Hybrid Hash-Join

Rekapituliert man die Stärken der beiden gerade vorgestellten Algorithmen, so stellt man fest, dass der einfache Hash-Join, gut arbeitet, wenn M groß ist und der GRACE Hash-Join, selbiges tut, wenn M relativ klein ist. Vorteilhaft wäre es, wenn man beide Algorithmen miteinander kombinieren könnte. Wie der Name schon vermuten lässt, ist der Hybrid Hash-Join nun diese Kombination.
Die Idee besteht darin, in Phase 1 so wenig wie möglich Blöcke von M für die Partitionierung von R und S zu nutzen, um Partitionen zu erzeugen, die in Phase 2 gerade im M passen. Mit diesen wird in Phase 2 wie beim GRACE Hash-Join verfahren. Die in Phase 1 verbleibenden Hauptspeicherblöcke werden aber nicht zum Cachen genutzt, sondern für eine Hashtabelle, die schon in Phase 1 zum Joinen einiger Tupel verwendet wird, ähnlich wie beim einfachen Hash-Join.

Die folgenden Schritte beschreiben den Algorithmus. Die Konstante B bezeichnet dabei die mindestens benötigte Anzahl an Partitionen, so dass für jede entstehende R-Partition eine Hashtabelle in M passt.

1. Wähle eine Hashfunktion h und Teilmengen $H_0, \ldots, H_B$ ihres Wertebereichs, so dass für R

   (a) eine Partition R_0 mit $(|M| - B)/F$ Blöcken entsteht.
   Somit reichen $|M| - B$ Blöcke für eine Hashtabelle dieser Partition.

   (b) Partitionen $R_1, \ldots, R_B$ gleicher Größe entstehen.
   Die Wahl der Konstante B stellt dabei sicher, dass jede der Partitionen $R_1, \ldots, R_B$ nicht größer als $|M|/F$ Blöcke wird. Somit reichen hier $|M|$ Blöcke für eine Hashtabelle.

   Verwende B Blöcke von M als Ausgabepuffer und den Rest für eine Hashtabelle der Partition R_0.

2. Lies R. Für jedes Tupel r: Berechne $h(r)$. Wenn $h(r) \in H_0$, füge r in die Hashtabelle ein; sonst in den entsprechenden Ausgabepuffer. Wenn ein Puffer voll ist, schreibe ihn auf Disk; am Ende schreibe alle Puffer auf Disk.

3. Lies S. Für jedes Tupel s: Berechne $h(s)$. Wenn $h(s) \in H_0$, prüfe in der Hashtabelle auf Matches, joine ggf. und gib das Ergebnistupel aus; sonst füge s in den entsprechenden Ausgabepuffer ein. Wenn ein Puffer voll ist, schreibe ihn auf Disk; am Ende schreibe alle Puffer auf Disk.

   [Hier endet Phase 1. Auf Disk befinden sich jetzt die Partitionen $R_1, \ldots, R_B$ und $S_1, \ldots, S_B$.]

4. Wiederhole für $i = 1, \ldots, B$

   (a) Lies R_i und erzeuge eine Hashtabelle dafür in M.

   (b) Lies S_i. Für jedes Tupel s: Berechne $h(s)$ und prüfe in der Hashtabelle von R_i auf Matches, joine ggf. und gib das Ergebnistupel aus.

Nun zur Herleitung der Konstante B. Wie oben bereits erläutert, bezeichnet sie die mindestens benötigte Anzahl an Partitionen, so dass jede in Phase 1 entstehende R-Partition nicht mehr als $|M|/F$ Blöcke belegt, ausgehend von der Annahme, dass jede der Partitionen gleich groß wird. Somit ergibt sie sich als Quotient von A, der Anzahl der in Phase 2 zu bearbeitenden

R-Blöcke und Z, der Zielgröße jeder Partition von $|M|/F$ Blöcken. A ist abhängig $|R|$ und $|R_0|$, da die Tupel, die in R_0 fallen, bereits in der ersten Phase verarbeitet werden. Die Größe von R_0 wird wiederum davon bestimmt, wie viel Hauptspeicher für die Hashtabelle von R_0 übrig bleibt, nachdem die B Ausgabepuffer alloziert sind. Die folgenden Gleichungen spiegeln diese Überlegungen wieder.

$$B = \frac{A}{Z} = \frac{|R| - |R_0|}{\frac{|M|}{F}} = \frac{|R| - \left(\frac{|M|-B}{F}\right)}{\frac{|M|}{F}} = \frac{|R| * F - (|M| - B)}{|M|}$$

Nach dem Umstellen nach B und mit Aufrunden auf die nächste Ganzzahl ergibt sich für B:

$$\underline{B = \left\lceil \frac{|R| * F - |M|}{|M| - 1} \right\rceil}$$

Für die Berechnung der I/O-Kosten des Hybrid Hash-Joins bezeichne zunächst q den Anteil von R_0 an R, also $q = |R_0|/|R|$. Unter der Annahme der Gleichverteilung der Join-Attribute in R und S kann die Größe der Partition S_0 auf $q * |S|$ geschätzt werden. Somit ist der Anteil von R und S, der für die Verarbeitung in der zweiten Phase auf Disk geschrieben werden muss (die Partitionen $R_1, \ldots, R_B$ und $S_1, \ldots, S_B$), gerade $1 - q$. Als I/O-Operationen für den Hybrid Hash-Join ergeben sich damit das Schreiben und wieder Einlesen eben jener Partitionen und damit:

$$\underline{2 * (|R| + |S|) * (1 - q)}$$

2.5 I/O-Kosten der Algorithmen im Vergleich

In diesem Abschnitt wird gezeigt, dass der Hybrid Hash-Join die beiden anderen vorgestellten Hashing-basierten Join-Algorithmen bezüglich der I/O-Kosten dominiert. Außerdem wird ein Vergleich mit einem Sort-Merge-basierten Algorithmus angestellt.

2.5.1 Hybrid vs. einfacher Hash-Join

Für den Vergleich mit dem einfachen Hash-Join stellt man zunächst fest, dass sich die beiden Algorithmen identisch verhalten, wenn $|M| \geq (|R| * F)/2$, also mehr als die Hälfte einer Hashtabelle für R in M passt. In diesem Fall ist die Anzahl der Iterationen (A) beim einfachen Hash-Join ≤ 2. Wenn M dagegen klein ist, sind viele Iterationen nötig, weshalb dann einige Teile von R und S mehrfach geschrieben und wieder eingelesen werden müssen. Der Hybrid Hash-Join bleibt hingegen in jedem Fall zweiphasig (unter der Annahme $|M| \geq \sqrt{F * |R|}$) und R und S müssen höchstens einmal geschrieben und wieder eingelesen werden. Dieses intuitive Argument soll an dieser Stelle als Beweis genügen. Der mathematische Beweis kann dem Paper entnommen werden. □

2.5.2 Hybrid vs. GRACE Hash-Join

Nun zum GRACE Hash-Join. Hier noch einmal die I/O-Kosten der beiden Algorithmen:

$$\begin{aligned} &\text{GRACE:} \quad 2 * ((|R| + |S|) - \min(|R| + |S|, |M| - \sqrt{F * |R|})) \\ &\text{Hybrid:} \quad 2 * ((|R| + |S|) - q * (|R| + |S|)) \end{aligned}$$

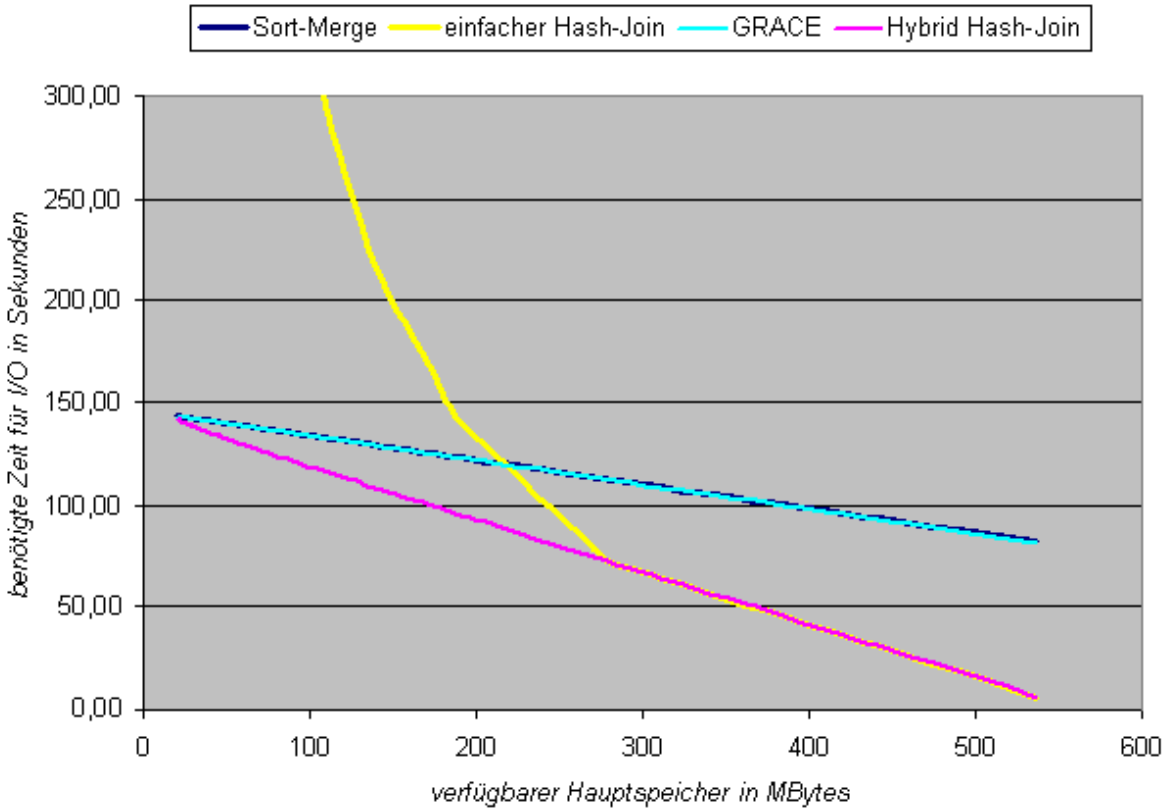


Abbildung 2.1: I/O-Kosten-Vergleich der Algorithmen

Ein Blick auf die Formeln zeigt, dass die Unterschiede in den Einsparungen liegen, die durch die Nutzung von zusätzlichem Hauptspeicher ermöglicht werden. Zu zeigen ist, dass

$$q * (|R| + |S|) \geq |M| - \sqrt{F * |R|}$$

Da $|S| \geq |R|$, kann eine entsprechende Ersetzung vorgenommen werden. Außerdem gilt, dass $q * |R| = |R_0| = (|M| - B)/F$. Es folgt:

$$q * (|R| + |S|) \geq q * (|R| + |R|) = 2 * q * |R| = 2 * (|M| - B)/F \geq |M| - \sqrt{F * |R|}$$

Der Faktor $2/F$ ist für sinnvolle F-Werte (im Paper 1,4) ≥ 1 und kann daher gleich 1 gesetzt werden. Es bleibt zu zeigen dass

$$B \leq \sqrt{F * |R|}$$

B war die mindestens benötigte Anzahl von Partitionen, um $|R| - |R_0|$ Blöcke in Partitionen zu teilen, so dass jede davon in M passt. $\sqrt{F * |R|}$ Partitionen sind jedoch genug, um alle $|R|$ Blöcke auf diese Weise zu partitionieren. Daher kann B nicht größer sein als $\sqrt{F * |R|}$. □

2.5.3 Hybrid vs. Sort-Merge Join

Als Vergleichsalgorithmus auf Sort-Merge-Basis dient ein leicht angepasster *TPMMS (Two Phase Multiway Merge Sort)*. Als Hauptspeicher benötigt dieser mindestens $\sqrt{|S|}$ Blöcke. Die Anpassung bezieht sich darauf, dass alle Hauptspeicherblöcke, die nicht zum eigentlichen

Sortieren bzw. Mergen benötigt werden, wie beim GRACE-Hash-Join zum Cachen verwendet werden. Die I/O-Kosten für diesen Algorithmus sind dann wie folgt

$$2 * \left((|R| + |S|) - \min \left(|R| + |S|, |M| - \sqrt{|S|} \right) \right)$$

Sie ähneln damit sehr denen des GRACE Hash-Join. Daher bietet sich dieser zum Vergleich an.
Wenn beide Algorithmen nur ihr Minimum von Hauptspeicher zur Verfügung haben, brauchen beide $2 * (|R| + |S|)$ I/O-Operationen. Mehr Hauptspeicher resultiert in gleichen Einsparungen. GRACE dominiert den TPMMS demnach genau dann, wenn $\sqrt{F * |R|} < \sqrt{|S|}$. Da $|R| \leq |S|$ und F nur für kleine Erhöhungen steht, ist dies der typische Fall.
Aus den vorigen Betrachtungen folgt, dass dann auch der Hybrid Hash-Join den TPMMS dominiert. □

Abbildung 2.1 zeigt den Verlauf der erwarteten I/O-Kosten der verschiedenen Algorithmen in Abhängigkeit von M. Dabei wurden die folgenden Werte als Berechnungsgrundlage angenommen: $|R| = 400$, $|S| = 800$, Blockgröße: 512 kByte, Zeit zum Lesen/Schreiben eines Blocks: 30 ms und $F = 1,4$.

2.6 Partitionsüberläufe

In den vorausgegangenen Kapiteln wurde davon ausgegangen, dass man die Hashfunktion h bzw. die Teilmengen ihres Wertebereichs $H_1, \ldots, H_n$ von vornherein so wählen kann, dass entstehende Partitionen genau die gewünschte Größe haben. Diese Annahme ist unrealistisch. Sie erfordert sehr genaues Wissen über die Werteverteilung der Joinattribute in den Relationen, welches in der Praxis selten vorhanden ist. Daher werden mit der initialen Wahl von h und $H_1, \ldots, H_n$ wahrscheinlich einige Partitionen zu leer bleiben und andere größer werden als erwartet, und somit nicht mehr in M passen.
Im Folgenden werden Lösungsansätze für Partitionsüberläufe in verschiedenen Situationen aufgeführt. Ein Vorteil bei den Hashing-basierten Algorithmen ist, dass die Größenbeschränkung nur für die Partitionen der kleineren Relation R relevant ist, da nur diese in den Hauptspeicher geladen werden müssen. Stellt sich heraus, dass eine R-Partition zu groß wird, ist eine Anpassung von h bzw. $H_1, \ldots, H_n$ erforderlich. Entsprechende Änderungen passieren aber, bevor S eingelesen wird, so dass beim Einlesen von S schon die angepassten Werte benutzt werden können.

Beim einfachen Hash-Join und beim Hybrid wird eine Hashtabelle für eine Partition im Hauptspeicher erzeugt. Wenn dieser dafür nicht ausreicht, dann müssen einige Buckets der Hashtabelle auf Disk geschrieben werden. Im Fall vom einfachen Hash-Join werden die betroffenen Tupel einfach zu den anderen übrig gebliebenen Tupeln auf Disk geschrieben. Im Fall des Hybrid Hash-Joins entsteht daraus eine neue Partition, die dann wie alle anderen Partitionen auf Disk in der zweiten Phase behandelt werden kann.

Beim GRACE und beim Hybrid Hash-Join werden Partitionen in extra Dateien auf Disk erstellt. Wenn eine dieser Partitionen zu groß geworden ist, dann gibt es zwei Möglichkeiten: Entweder wird die Partition so geteilt, das entstehende Teilpartitionen die Größenbeschränkung erfüllen, oder so, dass eine Partition entsteht, die gerade in M passt und der Rest wird anderen, zu leer gebliebenen, Partitionen zugeordnet.

2.7 Zusammenfassung

Im Paper werden drei Hashing-basierte Join-Algorithmen vorgestellt und deren I/O-Kosten theoretisch untersucht. Einer der drei, der Hybrid Hash-Join, eine Kombination aus den anderen beiden, hat sich dabei als am effizientesten herausgestellt. Es wurde auch gezeigt, dass dieser in typischen Fällen auch Sort-Merge-basierte Join-Algorithmen dominiert.
Bei den Betrachtungen wurden allerdings starke Annahmen bezüglich Partitionierung der Eingaberelationen gemacht. Danach sollten Partitionen niemals größer werden, als vom jeweiligen Algorithmus benötigt. In der Realität kann eine so „perfekte“ Partitionierung nicht ohne zusätzlichen Aufwand erreicht werden. Das Paper hat einfache Konzepte vorgestellt, um die Partitionierung während der Laufzeit an die Erfordernisse anzupassen, wenn nötig. Deren Auswirkungen auf die I/O-Kosten wurden allerdings nicht untersucht. Die Aussagen bezüglich der I/O-Kosten der Hashing-basierten Algorithmen sind somit zu optimistisch.

2.8 Weitere Entwicklungen

Als großes praktisches Problem des Hybrid Hash-Joins erweist sich die Partitionierung von R in wenige gleich große Partitionen. Falls diese nicht gut gelingt, steigen die I/O-Kosten aufgrund der nötigen Umverteilungen schnell an. In [42] wird daher eine flexiblere Hashing-basierte Join-Methode vorgeschlagen, die eine deutlich höhere Anzahl von Partitionen verwendet, als der Hybrid Hash-Join. Durch eine hohe Anzahl kleiner Partitionen treten Partitionsüberläufe, insbesondere solche, bei denen die entstehende Partition nicht mehr in M passt, deutlich seltener auf. Für eine optimale Bearbeitung der einzelnen kleinen Partitionen können diese in der zweiten Phase zu größeren Partitionen zusammengefasst werden (*„Bucket Tuning“*). Zusätzlich wird die Partition R_0, deren Tupel schon bei der Partitionierung von S gejoint werden, dynamisch bei bzw. nach der Partitionierung von R ermittelt. In [32] wird diese Methode, dort *Dynamic Hybrid GRACE Hash Join* genannt, weiter auf ihre Performance bei verschiedenen Werteverteilungen in den Joinattributen untersucht.
Weiterhin wurde bei allen vorgestellten Join-Algorithmen davon ausgegangen, dass dem Join-Prozess eine feste Menge von Hauptspeicher zur Verfügung steht. Dies kann zum Beispiel in Multiuser-Umgebungen nur schwer garantiert werden, insbesondere wenn die benötigte Speichermenge einen bedeutenden Anteil des insgesamt verfügbaren Hauptspeichers ausmacht. In [63] wird darum der Frage nachgegangen, wie Hash-Joins ihren Ressourcenbedarf während der Ausführung des Joins regulieren können, um gleichzeitig mit anderen Anwendungen ausgeführt zu werden. Ein ähnliches Problem stellt sich auch bei Echtzeit-Datenbanksystemen, wo prioritätenbasiertes Scheduling ein wichtiges Features ist. Als Folge des Beginns einer hochprioren Transaktion können andere Transaktionen Speicher verlieren, genauso wie sie Speicher hinzugewinnen können, wenn Transaktionen beendet werden. [45] vergleicht die Performance verschiedener Algorithmen und stellt eine Familie Speicher-adaptiver Algorithmen vor (sog. *Partially Preemptible Hash Joins*), die einen effektiven Umgang mit Schwankungen der Größe des verfügbaren Hauptspeichers ermöglichen.
Mit Anwachsen der Größe der Datenbanken und somit steigenden Kosten für Join-Operationen drängt sich auch die Frage nach einer möglichen Parallelisierung auf. In [48] werden sowohl Sort-Merge als auch Hashing-basierte, parallel arbeitende Algorithmen vorgestellt und verglichen. Dabei hat sich gezeigt, dass Hashing-basierte Methoden auch bei der Parallelisierung Vorteile gegenüber Sort-Merge-basierte Methoden haben, da die Partitionierung eine vollständig parallele Verarbeitung ermöglicht. Bei Sort-Merge-basierten Algorithmen können dagegen nicht alle Schritte parallelisiert werden (zum Beispiel das letzte Mischen). In [37]

werden verschiedene Hashing-basierte Algorithmen für den Einsatz auf einem Mehrprozessorsystem mit gemeinsam genutztem Speicher untersucht. Für das Schreiben in den gemeinsamen Speicher wird ein Locking-Mechanismus verwendet. Dabei stellt sich heraus, dass der Hybrid Hash-Join nicht immer optimal ist, insbesondere wenn die Anzahl der Partitionen klein ist (häufiges Blockieren durch das Locking). Für diese Fälle werden einfachere Hashing-basierte Verfahren vorgeschlagen. Außerdem ist es für eine optimale Überlappung bei der Parallelisierung notwendig, dass die beteiligten Ressourcen gleichmäßig ausgelastet werden. In [66] wird mit dem *Dynamic Balancing Hash Join* ein neuer Algorithmus vorgeschlagen, der dies erreicht, ohne die Werteverteilung in der Eingabe zu kennen oder vorher analysieren zu müssen.

Die vielfältigen Entwicklungen zeigen, dass sich Hash-Joins bewähren. Heute unterstützen nahezu alle modernen DBMS Hashing-basierte Join-Methoden.

Teil II

Data Storage and Access Methods / Transaction Management

Marcel Taeumel
marcel.taeumel@student.hpi.uni-potsdam.de

3

The Five-Minute Rule

Inhaltsangabe

3.1 Hauptspeicher und Festplatte optimal genutzt

Die *Five-Minute Rule* stammt aus dem Jahr 1987. Sie wurde damals erstmalig in einer Publikation von Jim Gray erwähnt und propagiert [22]. Es handelt sich hierbei um einen Kompromiss zwischen Hauptspeicherkosten und Kosten von Festplattenzugriffen. *Wann lohnt es sich finanziell, ein Datum im Hauptspeicher zu halten, als es bei Bedarf von der Festplatte zu laden?*

3.1.1 Herleitung der Five-Minute Rule

Wichtige Größen dieser Thematik sind die Kosten pro Festplatte, wobei die Betriebskosten (CPU, Controller) eingerechnet werden, die Zugriffsgeschwindigkeit, wobei hier vorerst wahlfreier Plattenzugriff gemeint ist, und der Preis pro Megabyte[1] Hauptspeicher. Sofern von einem *Datum* geschrieben wird, ist die Größe des entsprechenden Datensatzes von Bedeutung. Falls ein Datensatz jedoch zu groß ist, liegt er fragmentiert auf der Festplatte vor. Diese Fragmente, im Folgenden als *Datenblöcke* oder *Seiten* bezeichnet, sind dann maßgebend. Die Größe dieser Datenblöcke ist prinzipiell frei wählbar. Sie sollte nur nicht kleiner als die Blockgröße der Festplatte sein, da dies das Minimum an übertragenen Daten vorgibt. Bei den berechneten I/O-Kosten geht es um die Übertragung eines solchen Blocks.

[1] ein Megabyte entsprechen hier 1000 Kilobyte

Betrachtet wird der Zeitraum, welcher zwischen zwei Zugriffen auf das gleiche Datum liegt. Dieses Referenzintervall bestimmt zusammen mit Festplattenpreis und Zugriffsgeschwindigkeit die Höhe der anfallenden I/O-Kosten:

$$\$_{I/O} = \frac{PricePerAccessPerSecond}{ReferenceInterval}$$

Die Zugriffsgeschwindigkeit hängt unter anderem davon ab, wie viele Daten bei einem Zugriff übertragen werden. 1987 waren dies typischerweise 1000 Byte bei etwa 15 Zugriffen pro Sekunde. Auf Grund dieser Tatsache entstanden überhaupt erst die fünf Minuten der *Five-Minute Rule*. Bei anderen Datenmengen würden andere Zahlen resultieren, die grundlegende Intention dieser Regel bliebe jedoch unverändert. Den I/O-Kosten stehen die Kosten für das Halten eines Datums im Hauptspeicher gegenüber:

$$\$_{RAM,Record} = \$_{RAM} \cdot RecordSize$$

Die Kosteneinsparung, welche man durch das Halten eines Datums im Hauptspeicher hätte, berechnet sich aus der Differenz von I/O-Kosten und Hauptspeicherkosten für dieses Datum. Wenn beide Kosten die gleiche Höhe aufweisen, lässt sich das gesuchte Referenzintervall bestimmen. Es gibt an, wieviel Zeit zwischen zwei Zugriffen auf die gleiche Seite maximal vergehen darf, damit es sich noch lohnt, diesen Datenblock im Hauptspeicher zu halten.

$$ReferenceInterval = \frac{PagesPerMBofRAM}{AccessPerSecondPerDisk} \cdot \frac{PricePerDiskDrive}{PricePerMBofRAM}$$

$$ReferenceInterval = \frac{1000}{15\ acc/sec} \cdot \frac{30000\ \$}{5000\ \$} = 400\ sec/acc \approx 5\ min/acc$$

Für eine Seitengröße von 1000 Byte ergibt sich somit die *Five-Minute Rule* von 1987: *„Daten, welche innerhalb von 5 Minuten wiederholt referenziert werden, sollten im Hauptspeicher verbleiben."*

3.1.2 Anwendung bei der Ressourcenplanung

Die *Five-Minute Rule* kann benutzt werden, um zu bestimmen, wie eine Datenbank am kostengünstigsten betrieben werden kann [22]. Dabei benötigt man Umfang (Anzahl der Datensätze) und Auslastungsdaten (Spitzenlast), sowie geforderte Antwortzeiten. Falls sich fast alle Abfragen auf einzelne Datensätze beziehen, ist die Regel direkt anwendbar.

Ausgehend von der benutzten Datenblock- oder Seitengröße, wird mit der Formel das Zeitfenster berechnet. Für Seiten mit 1000 Bytes waren dies 1987 fünf Minuten. Nun sucht man den Anteil der Daten, welche innerhalb von diesen fünf Minuten wiederholt referenziert werden. Diese sollten dann im Hauptspeicher gehalten werden.

Nach statistischen Abschätzungen gehen 80% Zugriffe auf 20% der Daten. Diese Erkenntnis sollte benutzt werden, um die Anzahl der Festplatten zu bestimmen, welche benötigt werden, um die geforderte Antwortzeit einhalten zu können. Ein gewisser Anteil der Daten sollte auf Grund der vorigen Untersuchungen im Hauptspeicher resistent sein und so die Last der Festplatten deutlich reduzieren. Dies impliziert wiederum eine Kosteneinsparung des RAID-Systems, was die Antwortzeiten sicherstellen muss.

Eine hypothetische Datenbank [22] umfasst 500.000 Datensätze zu je 1.000 Bytes. Die Spitzenlast beträgt 600 Transaktionen pro Sekunde. Wenn die Datenbank komplett im Hauptspeicher gehalten wird, werden die Festplatten während des Betriebs nicht benötigt. Diese persistieren lediglich die komplette Datenbank einmalig und speichern Indizes, sowie benötigte Programme. Dieses Design hätte 1987 etwa 12 *Mio* $ gekostet. Unter Berücksichtigung der *Five-Minute Rule* konnte ermittelt werden, dass sechs Prozent der Daten innerhalb von fünf Minuten wiederholt referenziert werden. Auf diese Daten erfolgen, statistisch gesehen, 96% aller Zugriffe. So müssen nur noch vier Prozent der Spitzenlast von den Festplatten bewältigt werden. Dieses Design würde eine Einsparung von etwa 30% ergeben — 3,5 *Mio* $.

3.1.3 Zeitliche Entwicklung der Regel

In den folgenden 10 Jahren entwickelten sich höhere Bandbreiten und schnellere Zugriffszeiten bei Festplatten. Im gleichen Zug fielen die Preise pro Megabyte Haupt- oder Festplattenspeicher drastisch. Diese Entwicklung resultiert im Verwenden von größeren Speicherseiten beim Datentransfer. Entstand die Regel ursprünglich unter Betrachtung von 1kB-Seiten, waren damals bereits 4kB pro Seite realistisch. Jetzt, 1997, erfüllen 8kB-Seiten immernoch die *Five-Minute Rule*. Die beiden Faktoren der Gleichung für das Referenzintervall sind bewusst so gewählt, da sie Technologie und Wirtschaftlichkeit trennen:

$$ReferenceInterval = TechnologyRatio \cdot EconomicRatio$$

Da der technologische Faktor um das 10fache gestiegen, dagegen der wirtschaftliche Faktor um das 10fache gefallen ist, bleibt im Großen und Ganzen das Ergebnis gleich: *5 Minuten pro Zugriff*.

Die jüngste Betrachtung zu dieser Thematik stammt von 2007. Goetz Graefe befasste sich bei HP Labs erneut damit [19] und berücksichtigt aktuellste Technologie: *den Flashspeicher*. Dieser Speicher ordnet sich in der bekannten Speicherhierarchie genau zwischen den Hauptspeicher und der klassischen Festplatte. Dank der enorm hohen Zugriffsgeschwindigkeit und der steigenden Bandbreite ist zu überlegen, ob Flashspeicher eher den Pufferbereich erweitert, und so als flüchtig angesehen werden muss, oder zum persistenten Speicher zählt. Je nach Sichtweise müssen Datenblöcke zwischen den Ebenen verschoben werden. Die alte Regel ist, bezüglich Festplatte und Hauptspeicher, nur noch für 64kB-Seiten gültig. Wenn es um den Transfer von Daten zwischen Flash- und Hauptspeicher geht, so gilt für 4kB-Seiten die 15-Minuten-Regel. Es müssen also mindestens 15 Minuten zwischen zwei Zugriffen auf das gleiche Datum vergehen. Inzwischen (2009) sind *Solid-State-Disks* (Flashspeicherplatten) günstiger geworden und es ergeben sich *zwei neue Regeln*. Für 4kB-Seiten zwischen Haupt- und Flashspeicher, sowie für 256kB-Seiten zwischen Flashspeicher und Festplatte gilt die *Five-Minute Rule* immer noch. Mit dieser Information ist es wieder möglich, mittels gegebener, eigentlich frei wählbarer, Seitengröße zu entscheiden, wie viele Daten jeweils im Haupt- oder Flashspeicher verbleiben sollten, um am kostengünstigsten zu verfahren.

3.2 Weitere Faustregeln

Es gibt weitere Regeln, welche am im Hinblick auf die Speicherverwaltung berücksichtigen sollte. Im Folgenden wird auf die optimale Größe einer Indexseite bei binären Indexbäumen

eingegangen. Danach wird untersucht, wann ein Datum im Hauptspeicher gehalten werden sollte, falls es sich um sequentiellen Zugriff handelt.

3.2.1 Seitengröße von binären Indexbäumen

Die Größe einer Indexseite eines binären Suchbaumes definiert die Bereitstellungskosten, was das Lesen von der Festplatte betrifft, und den *Fan-Out*, also die Anzahl der Einträge bzw. Knoten im Baum, welche bereitgestellt werden. Eine Anfrage, welche gezielt ein Element sucht, geht einen bestimmten Pfad durch den Baum. Dabei wird eine gewisse Anzahl an Indexseiten gelesen. Sofern der binäre Suchbaum balanciert ist, ergibt sich folgende Höhe des Baumes in Indexseiten:

$$IndexHeight \approx \frac{\log_2(N)}{\log_2(EntriesPerPage)}$$

Nun stellt sich die Frage nach der optimalen Größe einer Indexseite. Eine Seite bringt die Suchanfrage eine gewisse Zahl an Schritten näher zum Ziel. Diesen Nutzen, den die Seite hat, bezeichnet man als *IndexPageUtility*. Da dieser Nutzen mittels $\log_2(EntriesPerPage)$ berechnet wird, könnte man ihn beliebig vergrößern. Deswegen teilt man ihn durch die Zugriffskosten (Latenz- und Übertragungszeit) für eine solche Seite. Das resultierende Verhältnis gibt einen Indikator für den wirtschaftlichen Wert der Seite vor. Dieser Indikator besitzt nun das gewünschte Maximum, welches gesucht wird.

$$BenefitCostRatio = \frac{IndexPageUtility}{IndexPageCost}$$

Bei einer Seitengröße von 2000 Bytes mit Einträgen zu je 20 Bytes, welche die Seiten zu 70% füllen, ergibt sich eine optimale Indexseitengröße[2] für klassische Festplatten von 256 Kilobyte. Wenn Flashspeicher genutzt wird, so beträgt sie 2 Kilobyte, da hier von der sehr kurzen Zugriffszeit profitiert werden kann. Daraus folgt, dass eine einheitliche Indexseitengröße eher suboptimal ist, sofern Flashspeicher genutzt wird. *SB-Bäume*[44] berücksichtigen beispielsweise diese Problematik.

Diese optimale Seitengrößen können nun wiederum für die Anwendung der zwei neuen *Five-Minute Rules* [19] genutzt werden.

3.2.2 Sequentieller Datenzugriff berücksichtigt

Wenn Datensätze komplett sequentiell gelesen werden können, weil Fragmentierung nicht vorhanden ist, ist die Transferrate der Festplatte gegenüber wahlfreiem Zugriff etwa zehnmal höher [21]. Sequentielle I/O-Operationen, wie Sort, Cube, Hash-Join oder Rollup, können effektiver ausgeführt werden, wenn bis zu einer gewissen Grenze in Hauptspeicher investiert wird, um mit einem Durchlauf auszukommen. Wann lohnt beispielsweise der Einsatz eines *Two-Phase Multiway Merge Sort* im Vergleich zum *One-Pass Sort*, welcher mit der Hälfte der I/O-Operationen auskäme, aber viel mehr Hauptspeicher benötigt? Es wird wieder die Zeit gesucht, welche zwischen zwei Zugriffen auf das gleiche Datum vergeht. Die Intervallgröße muss verdoppelt werden, da Lese- und Schreiboperationen berücksichtigt werden müssen:

[2] Werte für 2007

$$ReferenceInterval = 2 \cdot \frac{PagesPerMBofRAM}{AccessPerSecondPerDisk} \cdot \frac{PricePerDiskDrive}{PricePerMBofRAM}$$

Bei hohem Datentransfer sind die Zugriffskosten vernachlässigbar und die Bandbreite des Mediums ist entscheidend. So ergibt sich 1997 die *One-Minute Sequential Rule* für 64kB-Datensätze: *Sequentielle Operationen, welche Daten innerhalb von einer Minute wiederholt referenzieren, sollten diese Daten im Hauptspeicher halten.*

Wenn ein One-Pass-Sort-Algorithmus 5 GB pro Minute verarbeiten kann, sollte erst bei Relationen ab 5 GB ein Two-Pass-Sort genutzt werden. Andererseits lohnt es sich, in Hauptspeicher zu investieren.

3.2.3 Ausblick

Weitere Themenbereiche sind hinsichtlich der effizienten Speicherverwaltung interessant. Es wäre durchaus möglich, neben den Anschaffungs- auch die Energiekosten zu berücksichtigen. Dies könnte bei der Entscheidung zwischen SSDs und klassischen Festplatten hilfreich sein. Weiterhin existiert die *10-Byte Rule* aus dem Jahr 1987 [22], welche sich damit befasst, wann es sich lohnt, ein Datum komprimiert im Hauptspeicher zu halten und bei Bedarf CPU-Zyklen zu investieren um es zu entpacken. Bezüglich des Flashspeichers benötigt man wirksame Richtlinien, um Daten sinnvoll in der Speicherhierarchie zu bewegen. Hier stehen manuelle Ansätze, wobei ein Administrator entscheidet, den automatischen gegenüber, welche beispielsweise nach dem LRU-Verfahren arbeiten können. Weiterhin könnte *generative Garbage-Collection* von der dreistufigen Speicherhierarchie profitieren.

3.3 Fazit

Die *Five-Minute Rule* schafft theoretisch eine Berechnungsgrundlage, um für eine gegebene Datenbank zu entscheiden, wieviel Hauptspeicher und wieviel Festplatten benötigt werden. Es wird deutlich, dass es sich zum Beispiel nicht lohnt, alle Datensätze im Hauptspeicher zu halten. Die finanziellen Kosten übersteigen den Nutzen, da immer die tatsächlich benötigte Antwortzeit der Anfragen berücksichtigt wird. In der Praxis wird es womöglich schwierig werden, solche Lastdaten im Vorfeld zuverlässig ermitteln zu können. Sofern Hauptspeicher und Festplatten variabel verringert oder erweitert werden können, könnte die *Five-Minute Rule* dazu benutzt werden, um dynamisch sich verändernde Datenbankauslastungen kostengünstig zu kompensieren. Mit Technologien, wie *Instant Capacity on Demand* (iCAP) von HP, wäre dieses Vorgehen durchaus denkbar, da hier innerhalb kürzester Zeit Ressourcen reguliert werden können.

Leider werden bei den Datenbankanfragen keine Bereichsanfragen berücksichtigt. Inwiefern die *Five-Minute Rule* darauf abzubilden ist, wird nie erwähnt. Die *Fünf*, welche energisch in den Ausarbeitungen propagiert wird, lenkt zu sehr von der eigentlichen Intention des gesuchten Referenzintervalls ab. Auf den ersten Blick erscheinen fünf Minuten als das Maß der Dinge bezüglich Kompromiss zwischen Hauptspeicher- und Festplattenkosten. Es ist sehr leicht, zu jeder Zeit zu behaupten, dass diese Regel noch gilt. Dank diverser Zahlenspielereien wird der praktische Nutzen teilweise zu weit in den Hintergrund gestellt. Dies könnte einer der Gründe sein, weshalb die praktische Anwendung der Regel in realen Systemen keine

Erwähnung findet. Es ist sehr wahrscheinlich, dass sie in zahlreichen Datenbanksystemen inherent Anwendung findet.

Felix Elliger
felix.elliger@student.hpi.uni-potsdam.de

4

Vom optimistischen Umgang mit Nebenläufigkeit

Inhaltsangabe

4.1 Konsistenz trotz Nebenläufigkeit

Die Anforderungen an ein Datenbankmanagementsystem sind vielfältig. 1982 wurden sie von Edgar Codd in [8] beschrieben. Zu ihnen zählen unter anderem die Sicherung der Integrität und der Konsistenz sowie die Fähigkeit mit Nebenläufigkeit umzugehen.

Dabei stellt die Nebenläufigkeit eine inhärente Gefahr für die Konsistenz dar. So würde ein unkontrollierter nebenläufiger Zugriff mehrerer Transaktionen auf die Datenbank schnell zu einer Vielzahl von Problemen führen, wie zum Beispiel Phantomen oder verlorenen Updates. Dennoch ist Nebenläufigkeit zwingend notwendig, wenn es darum geht den Durchsatz zu erhöhen und damit die Wartezeiten der Benutzer zu verringern. Ziel muss es also sein Mechanismen zu entwickeln die Nebenläufigkeit erlauben und dabei trotzdem die Konsistenz der Daten erhalten.

Der heutzutage meist verbreitete Ansatz zur Lösung dieses Problems sind Lockingprotokolle. Einzelne Datenbankelemente werden vor ihrem Zugriff gesperrt,sodass jeweils nur eine Transaktion ein Objekt verändern kann.

Bereits 1981 wurde von Prof. H T Kung und John T. Robinson eine Alternative zu diesem „pessimistischen“ Vorgehen entwickelt. Der in [34] vorgestellte Algorithmus beruht auf der Annahme, dass Konflikte nur selten auftreten werden und ist in diesem Sinne „optimistisch“.

So wird erst unmittelbar vor dem Commit einer Transaktion geprüft, ob sie die Konsistenz verletzt haben könnte.

Abschnitt 4.2 wird zunächst die grundlegenden Schwächen von Lockingprotokollen erläutern bevor in Abschnitt 4.3 der optimistische Ansatz betrachtet wird. Da auch diese Methode nicht frei von Kritik ist, werden die wesentlichen Schwachpunkte in Abschnitt 4.4 vorgestellt. Abschließend fasst Abschnitt 4.5 die Kernpunkte noch einmal zusammen.

4.2 Locking

Der wohl meist verbreitete Ansatz zur Kontrolle von Nebenläufigkeit sind Lockingprotokolle, wie zum Beispiel das 2-Phase-Locking. Diese sind im Kontext dieser Ausarbeitung als pessimistische Herangehensweise zu bezeichnen. Sie bauen auf dem Grundgedanken auf, dass Konflikte von vornherein ausgeschlossen werden müssen. Um auf ein Element in der Datenbank, beispielsweise ein Tupel einer Relation, zugreifen zu können, muss jede Transaktion zunächst eine Sperre(Lock) anfordern und diese im weiteren Verlauf wieder freigeben. Aus diesem Grund muss die Transaktion um weitere Operationen ergänzt werden, die der Einhaltung des Lockingprotokolls dienen. Abbildung 4.1 zeigt dies an einem abstrakten Beispiel.

Abbildung 4.1: Zusätzliche Operation bei Locking

Diese Zusatzoperationen führen zwar zu einer kontrollierten nebenläufigen Ausführung mehrerer Transaktionen, erzeugen aber auch Overhead der in einigen Fällen gar nicht nötig ist. Dies ist zum Beispiel der Fall, wenn nur eine Transaktion aktiv ist oder die Transaktionen in verschieden Bereichen der Datenbank agieren. In [25] wird beispielsweise der resultierende Overhead in System R mit 10% der Gesamtausführungszeit einer Transaktion beziffert.

Eine weitere Schwachstelle der Locking-Algorithmen sind Deadlocks. Tabelle 4.1 zeigt einen Schedule, in dem zwei Transaktionen nebenläufig ausgeführt werden. Da beide auf die Freigabe einer Sperre warten, die von der jeweils anderen Transaktion gehalten wird, kann keine ihre Ausführung fortsetzen. Es muss also weiterer Aufwand darin investiert werden, Deadlocks zu identifizieren und diese über einen Mechanismus aufzulösen.

Genau an diesen Punkten setzt nun der optimistische Algorithmus an, wie er in [34] vorgestellt wird.

4.3 Der optimistische Weg

Der grundlegende Gedanke hinter diesem Ansatz ist die Annahme, dass Konflikte gar nicht oder nur sehr selten zwischen mehreren parallel ausgeführten Transaktionen bestehen. Dies

T1	T2
lock(A)	
A := A + 1	
	lock(B)
	B := B - 10
	lock(A)
lock(B)	

Tabelle 4.1: Nebenläufiger Schedule mit Deadlock

ist vor allem dann der Fall, wenn überwiegend lesende Zugriffe über Anfragen erfolgen, da das Lesen eines Wertes die Konsistenz nicht gefährdet. Das optimistische Prinzip ist es somit, nicht jeden Schritt einer Transaktion zu überwachen, sondern erst einzugreifen, wenn ein Konflikt besteht.

Diese Form der Nebenläufigkeitskontrolle führt zu einem geänderten Transaktionsmodell, welches in Abschnitt 4.3.1 erläutert wird. In Abschnitt 4.3.2 werden dann Validierungskriterien vorgestellt, die in dem im Abschnitt 4.3.3 präsentierten Algorithmus implementiert sind.

4.3.1 Das Transaktionsmodell

Nach [34] setzt sich eine Transaktion nun aus den folgenden drei Phasen zusammen:

1. **read-Phase**
   In dieser Phase werden alle Operationen der Transaktion ausgeführt. Schreiboperationen werden jedoch nicht auf den Original-Datenelementen sondern lokalen Kopien dieser durchgeführt. Zu diesem Zweck erhält jede Transaktion ein privates write-set, in dem die geänderten Werte abgelegt sind. Weiterhin wird vermerkt welche Elemente gelesen wurden (read-set).

2. **validate-Phase**
   Im Anschluss an die read-Phase muss überprüft werden, ob die gemachten Änderungen die Konsistenz gefährden oder gelesene Werte unter Umständen veraltet sind und somit zu einem falschen Ergebnis führen könnten. Schlägt diese Validierung fehl, wird die Transaktion neu gestartet.

3. **write-Phase**
   Ist die Validierung hingegen erfolgreich, können die globalen Elemente durch die bis dahin lokalen Kopien der Transaktion ersetzt werden. Sollte es sich bei der Transaktion um eine Anfrage handeln, wird in dieser Phase das Anfrageergebnis zurückgegeben.

Wie zu erkennen ist, wird die gesamte Transaktion zunächst abgearbeitet, bevor sie validiert wird. Dies ermöglicht in dieser Phase hohe Nebenläufigkeit ohne zusätzlichen Overhead. Entscheidend ist nun die Frage, wie die Validierung erfolgt.

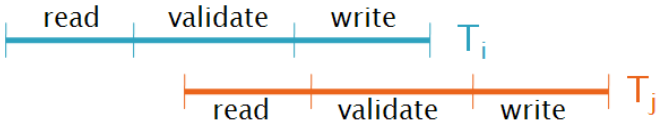
4.3.2 Optimistische Validierungskriterien

Essentieller Bestandteil der Validierung sind Transaktionsnummern. Jeder Transaktion T_n wird bei Beendigung ihrer read-Phase eine solche Nummer $t(n)$ zugewiesen. Folglich spie-

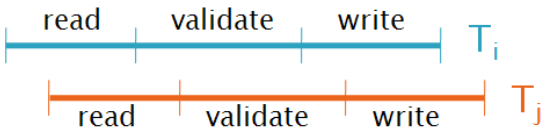
geln die Transaktionsnummern die zeitliche Reihenfolge wider, in der die Transaktionen ihre read-Phasen beendet haben.

Eine nebenläufige Ausführung mehrerer Transaktionen ist dann valide, wenn ihr Effekt zu einem seriellen Schedule äquivalent ist, in dem die Transaktionen in der Reihenfolge ihrer Transaktionsnummern ausgeführt werden. Werden alle Transaktionen ohnehin schon seriell ausgeführt, ist diese Bedingung trivialer Weise erfüllt. Für nebenläufige Transaktionen lassen sich zwei Kriterien definieren, die die Serialisierbarkeit sicherstellen. So ist Serialisierbarkeit gegeben, wenn für alle Transaktionen T_i, T_j mit $t(i) < t(j)$ eine der folgenden zwei Bedingungen erfüllt ist:

1. Die write-Phasen der Transaktionen sind seriell und es gilt $writeset(T_i) \cap readset(T_j) = \emptyset$.
   Diese Bedingung garantiert, dass T_j keine verfälschten oder veralteten Werte gelesen hat. Aufgrund der seriellen write-Phasen ist es weiterhin nicht möglich, dass T_i Ergebnisse von T_j überschreibt.



2. T_j beendet die read-Phase nach der read-Phase von T_i und es gilt $writeset(T_i) \cap (readset(T_j) \cup writeset(T_j) = \emptyset$.
   Da von der zeitlichen Abfolge der read-Phasen nicht auf die Reihenfolge der write-Phasen geschlussfolgert werden kann, muss sichergestellt sein, dass T_j weder veraltete Werte liest, noch Schreibkonflikte zwischen den Transaktionen bestehen. Die Anforderung an die read-Phasen ist über die Vergabe der Transaktionsnummern automatisch garantiert.



Der folgende Abschnitt zeigt, wie diese Kriterien in einem Algorithmus validiert werden.

4.3.3 Der Validierungsalgorithmus

Größtmögliche Parallelität kann im optimistischen Ansatz nur erreicht werden, wenn beide der oben genannten Kriterien implementiert werden.

Für die Validierung dieser Kriterien werden weitere Informationen benötigt. Zu jeder Transaktion T_j muss bekannt sein

- eine Menge $finished\_txn$ von Transaktionen, welche ihre write-Phase seit dem Start von T_j beendet haben, und
- eine Menge $active\_txn$ von Transaktionen, welche seit dem Start von T_j ebenfalls gestartet wurden, ihre write-Phase aber noch nicht beendet haben.

Auf Basis dieser Mengen kann der Algorithmus für eine Transaktion `t` wie folgt beschrieben werden:

```
valid = true;

for (Transaction txn : active_txn) {
   if (t.readset.intersects(txn.writeset) ||
      t.writeset.intersects(txn.writeset))
         valid = false;
}

for (Transaction txn : finished_txn) {
if (t.writeset.intersects(txn.writset))
      valied = false;
}

if (valid)
   write-Phase();
else
   restart();
```

Über diesen Algorithmus ist es möglich Transaktionen nahezu vollständig parallel zu validieren, wodurch ein hohes Maß an Nebenläufigkeit erreicht wird. Lediglich die Zuweisung einer eigenen Transaktionsnummer, sowie die Bestimmung der beiden Transaktionsmengen muss in einer kritischen Sektion erfolgen, wie in [34] vorgestellt.

4.4 Schwachstellen des Optimismus

Obwohl der in [34] eingeführte Algorithmus deadlockfrei ist und den Overhead für Nebenläufigkeitskontrolle verringert, besitzt er einige Schwachstellen, die im Folgenden kurz betrachtet werden.

Aufgrund der Validierung gegen noch nicht abgeschlossene Transaktionen in der Menge $active\_txn$, besteht die Gefahr von unnötigen Neustarts, wie bereits in [34] erwähnt wird. So ist es durchaus möglich, dass eine Transaktion $t \in active\_txn$ die aktuelle Transaktion zum Neustart zwingt, obwohl sie selbst nicht erfolgreich validiert werden kann. Ebenfalls in [34] wird das Problem des Verhungerns von Transaktionen erwähnt, für das eine Lösung gefunden werden muss.

Weitere Kritikpunkte werden elf Jahre nach der Veröffentlichung des Papers in [41] betrachtet. Einer der wesentlichen hier aufgeführten Aspekte ist die Granularität bei der Konfliktüberprüfung. Der optimistische Ansatz prüft Konflikte auf der Ebene von Speicherseiten. Da pro Seite durchaus mehrere Hundert Datensätze gespeichert werden können, erhöht dies die Wahrscheinlichkeit von Neustarts, auch wenn unterschiedliche Datensätze dieser Seite geändert werden. Dies ist ein klarer Nachteil gegenüber den Lockingprotokollen, die wesentlich feingranularer arbeiten.

Obwohl optimistische Nebenläufigkeitskontrolle einige Anwendungsfälle besitzt, ist es wohl die Vielzahl von Schwachstellen, die dazu führt, dass , soweit es dem Autor bekannt ist,

bis heute keine Implementierung dieses Ansatzes in einem Datenbankmanagementsystem existiert. Dies ist wohl auch der Tatsache geschuldet, dass im Allgemeinen im Vorfeld nicht bekannt ist, wie wahrscheinlich Konflikte auftreten werden.

4.5 Zusammenfassung

Ausgehend von den Schwachstellen der Lockingprotokolle, wurde der optimistische Ansatz vorgestellt. Er definiert zunächst ein dreiphasiges Transaktionsmodell, in dem der zentrale Schritt die Validierung der Transaktion ist. Es werden zwei Kriterien definiert, die die Serialisierbarkeit der Transaktionen überprüfen. Diese sind in einem Algorithmus implementiert, der nahezu vollständig parallele Validierung ermöglicht und dabei deadlockfrei ist. Somit ermöglicht die optimistische Nebenläufigkeitskontrolle ein hohes Maß an Parallelität und stellt bei geringem Overhead die Konsistenz der Daten sicher.

Die nebenläufige Validierung führt jedoch zu vielen unnötigen Neustarts von Transaktionen, was durch die grobgranulare Konfliktüberprüfung auf der Ebene von Speicherseiten weiter verstärkt wird.

Damit bietet dieser vielversprechende Ansatz zu wenig Vorteile gegenüber den etablierten Methoden, wie Locking oder auch Multiversion Concurrency Control.

Teil III

Extensible Systems / Web Services and Databases

Johannes Dyck
johannes.dyck@student.hpi.uni-potsdam.de

5

Generalized Search Trees for Database Systems

Inhaltsangabe

5.1 Einleitung

Zur Gewährleistung effizienten und performanten Zugriffs auf Daten stellen Indizes eine wichtige Grundlage eines jeden Datenbanksystems dar, die beim Ausführen von Anfragen das Laden nicht benötigter Tupel überflüssig macht und somit sowohl Bearbeitungszeit als auch Hauptspeicherkapazitäten einspart. Neben Hash-basierten Indexstrukturen haben sich vor allem Bäume als Mittel der Indizierung durchgesetzt.

Diese Baumstrukturen kommen dabei in einer Vielzahl an Varianten. Der B^+-Baum beispielsweise ermöglicht die effiziente Indizierung eines vollständig geordneten Wertebereiches, wie der der ganzen Zahlen, und unterstützt Anfragen, die sich auf einen genauen Wert oder ein Intervall innerhalb dieses Wertebereiches beziehen. Als klassisches Beispiel wäre hier die Indizierung von IDs durch diese Art Baumstruktur denkbar. Handelt es sich allerdings um zweidimensionale Daten, zum Beispiel Punkte oder Flächen innerhalb eines Koordinatensystems, ist eine Abbildung auf B^+-Bäume nicht mehr ohne Weiteres möglich. Abhilfe schaffen hier R-Bäume, mit denen räumliche Daten indiziert werden können, wobei Anfragen nach der Gleichheit, Überlappung oder des Beinhaltens zweier Räume schneller beantwortet werden können.

Darüber hinaus existiert jedoch eine Vielzahl weiterer Datentypen, die sich der Indizierung mittels klassischer Baumstrukturen durch ihre speziellen Eigenschaften entziehen. Der Frage, wie trotzdem mit der Notwendigkeit der Indizierung dieser Daten umgegangen werden kann, widmen sich Joseph M. Hellerstein, Jeffrey F. Naughton und Avi Pfeffer in ihrer Arbeit „Generalized Search Trees for Database Systems“ [29] im Kontext der *Very Large Data Base Conference* 1995. Vorgestellt wird darin eine allgemeine und hochkonfigurierbare Baumstruktur, die die benötigte Flexibilität zur Indizierung nahezu beliebiger Daten und Unterstützung

diverser Anfragearten bereitstellt.

Diese Ausarbeitung stellt im folgenden Abschnitt 5.2 in Kürze eine Motivation für den Inhalt der Arbeit vor, bevor Abschnitt 5.3 die Grundlagen der vorgestellten Struktur präsentiert. Abschnitt 5.4 enthält eine kritische Diskussion der vorgestellten Überlegungen und Schlussfolgerungen und Abschnitt 5.5 fasst schließlich die zuvor präsentierten Überlegungen zusammnen und zieht ein Fazit, das sich auch auf den Wert der Arbeit im aktuellen Kontext bezieht.

5.2 Motivation

Hellerstein, Naughton und Pfeffer sehen zwei konventionelle Ansätze, verschiedenste Datentypen durch Indexstrukturen für den schnellen Zugriff zu indizieren. Der erste basiert auf hochspezialisierten Suchbäumen, wird jedoch aufgrund seiner Unflexibilität kritisiert. Bei einem solchen Ansatz sei die Erweiterung oder Einführung von Datentypen problematisch, da für diese neue Suchbäume entworfen werden müssten. Ebenso führten neue Arten von Anwendungen zu hohem Implementierungsaufwand die Suchbäume betreffend, von neuen Problemdomänen ganz zu schweigen.

Der zweite Ansatz sieht die Erweiterung bereits existierender Suchbäume wie B^+- oder R-Bäume vor, um weitere Datentypen zu unterstützen. So könne ein B^+-Baum beispielsweise Daten beliebiger Wertebereiche unterstützen, die sich in einer bestimmten Form ordnen lassen. Dennoch sei auch dieser Ansatz nicht zufriedenstellend, da zwar die Indizierung anderer Datentypen möglich ist, nicht jedoch eine Erweiterung der unterstützen Anfragearten. Tatsächlich beschränkt sich die Nützlichkeit eines B^+-Baumes auf Anfragen nach konkreten Werten oder Intervallen, während neuartige Datentypen dagegen möglicherweise andere Arten an Anfragen erforderlich machen.

Daher stellen die Autoren einen dritten Ansatz vor, der den genannten Schwierigkeiten entgegenwirkt. Der *GiST – Generalized Search Tree* – ist ein abstrakter Datentyp, der unter Anderem die typischen Operationen eines Suchbaums bereitstellt. Er stellt den verallgemeinerten Rahmen für eine hochkonfigurierbare Baumstruktur bereit, die je nach verwendetem Datentyp durch den Benutzer passend konfiguriert werden kann.

5.3 Der GiST

Im grundlegenden Aufbau (bezüglich Knoten, Zeigern und indizierten Tupeln) ähnlich einem B^+-Baum, liegt die Grundidee der Konfigurierbarkeit in der Implementierung von sechs grundlegenden Methoden, die das Verhalten des Baumes beeinflussen. Dadurch wird die Indizierung verschiedenster Datenstrukturen ermöglicht. Zusätzlich können alle Anfragearten, die dem Datentyp angemessen sind und durch den Benutzer spezifiziert werden, unter Benutzung des Baumes beantwortet werden. Typische Methoden wie *insert*, *search* oder *delete* sind dabei bereits vorgegeben und arbeiten unter Verwendung der durch den Nutzer spezifizierten Methoden.

Schlüssel, also Einträge in einem Knoten, die die darunter liegenden Schlüssel und schließlich Tupel charakterisieren, sind im GiST als Prädikate dargestellt. Alle Tupel, die von einem Knoten aus erreichbar sind, erfüllen das im Eintrag angegebene Prädikat. Selbiges kann dabei beliebige Variablen verwenden, solange diese durch jedes Tupel instanziiert und damit ausgewertet werden können. Wichtig zu vermerken ist an dieser Stelle, dass ein Tupel meh-

rere Prädikate auf der gleichen Ebene erfüllen kann und damit von beiden aus erreichbar sein kann. Als Beispiel aus der vorliegenden Arbeit sei das Prädikat für B^+-Bäume genannt: *Contains*([2,7), q) wird von allen Tupeln erfüllt, deren Wert im gefragten Attribut zwischen einschließlich 2 und ausschließlich 7 liegt. Dieses Prädikat als Suchschlüssel muss durch den Benutzer spezifiziert werden. Die definierten Prädikate entsprechen damit den Anfragen, die gestellt werden können, da alle Anfragen als atomare Prädikate oder ihre Kombinationen formuliert werden. Somit wird der GiST theoretisch für beliebige gewünschte Suchanfragen geöffnet, was genau den Punkt befriedigt, den die Autoren im einem der zwei genannten konventionellen Ansätze bemängeln.

Bei den sechs Schlüsselmethoden, die außerdem der Umsetzung durch den Anwender überlassen sind und das Herzstück der Konfiguration darstellen, handelt es sich um:

Consistent (E, q), wobei E ein Eintrag mitsamt seinem Prädikat ist und q ein Prädikat. Diese Methode muss feststellen, ob beide Prädikate gleichzeitig für ein Tupel der Domäne zutreffen können. Wohlgemerkt hängt dies nicht mit der Existenz eines konkreten Tupels zusammen, sondern bezieht sich lediglich auf die hypothetische Möglichkeit. Diese Methode wird beispielsweise beim Suchen nach Tupeln, die ein bestimmtes Prädikat erfüllen, verwendet. Dabei kann durch Aufruf von Consistent der Baum traversiert werden, bis die Suche bei den Tupeln angekommen ist und deren Werte gegen das Prädikat der Anfrage geprüft werden können.

Union (P). P ist dabei eine Menge an Einträgen, bestehend aus je einem Prädikat und einem Zeiger auf den zugehörigen Knoten der nächsttieferen Ebene. Das Ergebnis soll ein Prädikat sein, dass für alle Tupel, die von den Knoten aus erreichbar sind, erfüllt ist. Um nicht die tatsächlichen Tupel prüfen zu müssen, sollte in der Regel nur mit den in den Knoten aus P enthaltenen Prädikaten gearbeitet werden. Diese Methode findet ihre Anwendung zum Beispiel beim Einfügen von Tupeln, da dieses Einfügen Änderungen für die Gültigkeit von Prädikaten mit sich bringen kann, die eine Anpassung erforderlich macht.

Compress (E). Zur effizienten Speichernutzung kann diese Methode Schlüssel durch eine vorgegebene Methode komprimieren – der Rückgabewert ist ein Eintrag mit komprimierter Darstellung des Prädikates aus E.

Decompress (E), um das im Eintrag E enthaltene komprimierte Prädikat zu dekomprimieren. Dabei darf es sich um eine verlustbehaftete Komprimierung handeln, die durch den Anwender vorgegebene Komprimierung und Dekomprimierung muss keine Wiederherstellung des Originalwertes sicherstellen. Es muss allerdings zugesichert werden, dass alle potenziellen Tupel, die das ursprüngliche Prädikat erfüllten, auch dem dekomprimierten Eintrag genügen.

Penalty (E_1, E_2). Hier muss der Anwender spezifizieren, wie günstig oder ungünstig es ist, den Teilbaum, der am Knoten E_2 beginnt, in denjenigen bei E_2 einzusetzen. Diese Bewertung wird dabei auf Basis der die Knoten E_1 und E_2 repräsentierenden Prädikate gegeben. *Penalty* wird innerhalb des bereits vorgegebenen *Insert*-Algorithmus verwendet, um den optimalen Ort zum Einfügen eines neuen Tupels zu finden. Soll der GiST beispielsweise das Verhalten eines B^+-Baums umsetzen, könnte diese Methode die Differenz zwischen den in den Prädikaten enthaltenen Intervallen zurückgeben.

PickSplit (P), wobei P eine Menge an Baumeinträgen (bestehend aus je einem Prädikat und Zeiger auf einen zugehörigen Knoten) ist. Die Methode muss dabei die Menge in zwei

Teile spalten, was benötigt wird, wenn durch Einfügen eines Wertes ein Knoten zu viele Einträge erhalten wird. Die Implementierung dieser Funktion entscheidet den minimalen Füllwert eines Knoten.

Von der Implementierung dieser Methoden und der Spezifizierung der Anfrageprädikate und ihrer Auswertung für konkrete Tupel abgesehen, muss der Benutzer keine weitere Konfiguration vornehmen. Alle übrigen Methoden, die typischerweise zur Funktionsweise des Baumes nötig sind, sind bereits implementiert und bauen auf den konkreten Umsetzungen der abstrakten sechs Schlüsselmethoden auf.

Eine mögliche Erweiterung des GiST soll an dieser Stelle noch erwähnt werden: Falls es sich bei dem zu indizierenden Wertebereich um eine Menge, die einer Ordnung unterliegt, handelt, kann dies speziell vermerkt werden. In diesem Fall werden Bereichsanfragen effizienter beamtwortet, indem zunächst ein Wert am Anfang des Bereiches gesucht wird und dann bis zum Ende des Intervalls die Blattknoten in aufsteigender Reihenfolge betrachtet werden. Dafür muss gewährleistet sein, dass sich Prädikate innerhalb eines Knotens nicht überlappen. Zusätzlich muss eine *Compare*-Methode zwei gegebene Tupel gemäß der Ordnung vergleichen und die PickSplit-Methode die Eingabemenge so teilen, dass alle Prädikate des einen Teils kleiner sind als jedes Prädikat des anderen Teils.

5.4 Diskussion

Die in der Arbeit präsentierten Überlegungen und Schlussfolgerungen sind größtenteils schlüssig – das erläuterte Problem ist klar ersichtlich. Auch für die Ablehnung der zwei konventionellen Ansätze – spezialisierte Suchbäume beziehungsweise solche für erweiterbare Datentypen – ist anhand der gegebenen Argumente nachvollziehbar. Die Verwendung neuer Datenstrukturen bedeutet tatsächlich einen hohen Implementierungsaufwand, sollten dafür neue, spezielle Suchbäume entwickelt werden. Auf der anderen Seite bringen neue Datentypen andere Anfragearten, die selbst bei erweiterbaren Suchbäumen nicht ohne Weiteres darstellbar sind. Insofern ist der von Hellerstein, Naughton und Pfeffer vorgestellte Ansatz eine praktikable Lösung, die eine Vielzahl an Optionen offenhält. Gleichzeitig bleibt jedoch die allgemeine Struktur als solche klar überschaubar – die Implementierung von lediglich sechs Kernmethoden ist weniger umfangreich als der komplette Entwurf eines neuen Baumtypes.

In diesem Zusammenhang ist insbesondere die Kapselung der Implementierungsdetails zu erwähnen. Ohnehin ein fundamentales Prinzip der Informatik, tritt der Nutzen dieser Abstraktion hier auf das Deutlichste hervor: Alle für einen Baum üblichen Operationen funktionieren im Allgemeinen gleich, völlig unabhängig von der Art der verwendeten Datentypen und Anfragen. Dies gestattet es einem Entwickler, den Fokus auf dem letztgenannten Punkt zu halten, ohne sich anderer Implementierungen bewusst zu sein, was die Robustheit des Systems gegenüber Veränderungen erhöht.

Darüber hinaus gelingt es den Autoren, die Natur des GiST trotz seiner abstrakten Art anschaulich zu illustrieren. Gerade die Beispiele, in denen der GiST wie ein B^+-Baum, ein R-Baum und ein RD-Baum[1] konfiguriert wird, tragen zur Verständlichkeit bei und demonstrieren die Flexibilität.

Allerdings gestehen Hellerstein, Naughton und Pfeffer ein, dass nicht für alle durch einen

[1] RD-Bäume dienen der Indizierung mengenwertiger Wertebereiche, zum Beispiel der Potenzmenge der ganzen Zahlen.

GiST indizierbaren Domänen tatsächlich eine effiziente Anfragebearbeitung möglich ist. Dies hängt unter Anderem mit dem geringen Potenzial für Optimierungen zusammen – hier ist trotz aller Flexibilität die Grundstruktur des GiST bezüglich der nicht durch den Nutzer gegebenen Methoden eher starr.

Weiterhin muss gesagt werden, dass die Nützlichkeit dieser Flexibilität lediglich in der Theorie nachvollziehbar unter Beweis gestellt wird – einen praktischen Beleg bleiben die Autoren an dieser Stelle schuldig. Auch das Argument, die Implementierung von Bäumen durch einen GiST würde weniger Quelltext erfordern als konventionelle Umsetzungen dieser Strukturen, ist wenig ausschlaggebend. Schließlich erlaubt ein spezialisierter B^+-Baum oder R-Baum mehr Optimierungsstrategien als der konfigurierbare GiST, die in mehr Quelltext resultieren könnten. Darüber hinaus ist in Frage zu stellen, inwiefern die Zeilen an Quellcode ein sinvolles Maß für die Güte des Produktes oder auch für den Entwicklungs- und Forschungsaufwand sind.

Zudem umgehen die Autoren die Überlegungen, in wie vielen Fällen ein GiST tatsächlich benötigt würde. Für viele in Relationen typische Attribute sind B^+-Bäume und ihre Abwandlungen ausreichend. Praktische Beispiele ungewöhnlicher Datentypen, bei denen eine Indizierung tatsächlich sinnvoll erschiene, werden nicht vorgestellt. Die präsentierten Umsetzungen von B^+-Bäumen oder R-Bäumen sind somit für das Verständnis hilfreich, stellen aber nicht die Nützlichkeit des ganzen Potenzials eines GiST unter Beweis.

Als vorläufig letzter Punkt muss die Implementierung von *Consistent* erwähnt werden. Tatsächlich trifft es hier zwar zu, dass im Zusammenhang mit Prädikaten beliebige Anfragen unterstützt werden können. Die volle Ausschöpfung dieser Möglichkeiten würde jedoch dazu führen, dass die Verträglichkeit zweier logischer Ausdrücke bestimmt werden muss. An dieser Stelle stößt das Prinzip an seine Grenzen, denn die Auswertung davon führt zu exponentieller Laufzeit, was in keinem üblichen Datenbanksystem tolerierbar ist. Außerdem steckt für neue Datentypen in *Consistent* wie auch in den anderen zu definierenden Methoden ein nicht zu vernachlässigender Aufwand, der durchaus den Gewinn durch die bereits implementierten Methoden wie *Search* nichtig erscheinen lassen mag.

5.5 Fazit

Zusammenfassend ist zu sagen, dass die Arbeit ohne Zweifel eine wichtige Fragestellung untersucht und eine in der Theorie nachvollziehbare Antwort darauf gibt. Trotz der angeführten Kritikpunkte bietet der Report eine Basis für weitere Erörterungen zur Frage der Indizierung komplexer Datentypen.

Weiterhin ist der Nutzen des GiST möglicherweise nicht darin zu sehen, dass er das Verhalten eine B^+-Baums und anderer gebräuchlicher Bäume annehmen kann: Für diese Strukturen gibt es bereits hocheffiziente Implementierungen, die sich in ihrer Verwendung bereits durchgesetzt haben – gerade heute, 14 Jahre später. Hier kann der GiST nicht mithalten, da spezifische Optimierungen zu Gunsten von Flexibilität vernachlässig werden. Das ändert jedoch nichts daran, dass der GiST für aktuelle Fragen wie die effiziente Indizierung multimedialer Daten eine solide Forschugsgrundlage liefern kann. Daraus folgt allerdings nicht notwendigerweise, dass in zukünftigen Datenbanken und Datentypen tatsächlich ein GiST zu diesem Zweck benutzt wird, sobald die Frage der Indizierung hinreichend geklärt ist.

Schlussendlich kann also zusammenfassend gesagt werden, dass die Arbeit interessante und wichtige Fragestellungen berührt und selbt aufwirft und durchaus einen Nutzen aufweist –

wenn auch vielleicht nicht durch größere Verbreitung des GiST in bekannten Datenbanksystemen.

Markus Güntert
markus.guentert@student.hpi.uni-potsdam.de

6

The Anatomy of a Large-Scale Hypertextual Web Search Engine

Inhaltsangabe

6.1 Abstract

1998 wurde am Computer Science Department der Stanford University das Paper "The Anatomy of a Large-Scale Hypertextual Web Search Engine" verfasst und damit der Grundstein des Weltunternehmens *Google* gelegt. Diese Ausarbeitung im Seminarfach Advanced Topics in Databases fasst den Inhalt des Papers – insbesondere die Punkte PageRank und Architektur – zusammen, verfolgt weitere Entwicklungen und diskutiert Schwachstellen der Suchmaschine.

6.2 Autoren

Die Autoren des Papers, *Sergey Brin* und *Lawrence Page*, lernten sich 1997 während des Masterstudiums im Fach Computer Science in Stanford kennen. Brins Fachgebiet war Data Mining [12], während sich Page mit der Exploration der mathematischen Eigenschaften des World Wide Webs beschäftigte [4]. Beide haben ihre Promotion bis heute nicht abgeschlossen, da das erkennbare Potential des gemeinsamen Forschungsprojektes Google die beiden eher in eine gemietete Garage als hinter Bücher lockte [12]. Das folgende Zitat von Lawrence Page über Suchmaschinen soll als Einleitung zu diesem Paper dienen: "The perfect search engine would understand exactly what you mean and give back exactly what you want." [16]

6.3 Initiale Ziele von Google

Google war der Prototyp einer Internet-Suchmaschine, die auf großen Mengen von Daten effizient arbeitet und die zusätzlich zu Dokumentinhalten auch die Dokumentstruktur berücksichtigt. Der Fokus lag dabei von Anfang an auf der Verbesserung der Qualität von Suchergebnissen. In der Architektur von Google wurde das Wachsen des Webs – sowie in Korrelation – der technologische Fortschritt berücksichtigt. Da damals existierende Systeme in erster Linie kommerziell orientiert waren, sollte das Projekt Google zur wissenschaftlichen Arbeit im Bereich der Suchmaschinen beitragen. Der spätere Erfolg von Google ist zudem nicht zuletzt auf die einfache Zugänglichkeit zurückzuführen.
In dem Paper werden hundert Millionen indizierte Webseiten und zehn Millionen Anfragen pro Tag als Kennzahlen genannt. Aktuelle Werte sind eine Billion indizierte Webseiten [2] und 91 Millionen Suchanfragen jeden Tag [59].

6.4 Standpunkt 1998

Heutzutage beginnen die Internetrecherchen der meisten Nutzer bei einer renommierten Suchmaschine, da durch eine hohe Abdeckung des Internets meist ausgereifte Ergebnisse geliefert werden. 1998 bestand eine derartige Infrastruktur noch nicht und zentrale Anlaufstelle war meist eine "human-maintained" Webseite wie *Yahoo!*. Auf solchen wurden Links zu bestimmten Themen von Hand eingepflegt. Diese Vorgehensweise garantiert unter Umständen zuverlässige Referenzen, da die Relevanz von einem Menschen bewertet wird. Bei einer nicht mehr überschaubaren Menge von Informationen erweist sich dieses Modell aber als nicht zuverlässig; es ist unvollständig, subjektiv und reagiert nur langsam auf Veränderungen.
Automatisierte Suchmaschinen, die allein auf Keyword-Matching basieren, liefern zu viele schlechte Ergebnisse. Das Angebot an Informationen steigt zwar kontinuierlich, das Aufnahmevermögen des Nutzers bleibt jedoch gleich. Was fehlt, ist ein Ranking der Ergebnisse nach Relevanz. Außerdem sind diese Systeme anfällig für sog. "Junk Results", die durch gezielte Manipulation das Ergebnis beeinflussen sollen.

6.5 PageRank

Die wesentliche Essenz des Papers ist sicherlich der PageRank-Algorithmus, welcher eine Priorisierung von Suchergebnissen in Form eines Rankings realisiert. Grundlegend neu ist,

dass neben dem Inhalt auch die Struktur von verlinkten Dokumenten bewertet und gewichtet wird. Eine Webseite gilt dem Algorithmus nach als wichtig, wenn viele oder wichtige Webseiten auf sie verlinken. PageRank kann als Adaption des Zitats für das Web betrachtet werden. Dies kommt dem subjektiven Verständnis von Wichtigkeit nahe.

6.5.1 Mathematische Betrachtung

Die Berechnung des PageRanks einer Webseite wird durch folgende vereinfachte Formel beschrieben:

$$r(P_i) = \sum_{P_j \in B_{P_i}} \frac{r(P_j)}{|P_j|}$$

Der PageRank einer Seite P_i ergibt sich aus der Summe aller PageRanks von auf P_i verlinkenden Seiten P_j. Jede der Seiten P_j wird dabei bezüglich ausgehender Links normalisiert eingebracht. Die Berechnung erfolgt rekursiv, um den PageRank von P_i zu berechnen muss der PageRank aller Webseiten P_j bekannt sein. Ein iterativer Lösungsansatz wird an anderer Stelle erläutert [35].

6.5.2 Anschauliche Erklärung

In dem Paper folgt eine anschauliche Erklärung: der PageRank-Algorithmus ist die Nachahmung eines zufällig durch das Netz surfenden Nutzers. Ein solcher Nutzer soll ausgehend von einer zufälligen Startseite fortwährend Links klicken, ohne jemals zurück zu gehen. Das Anfordern einer neuen zufälligen Startseite ist jederzeit erlaubt (unabdingbar, wenn eine erreichte Seite keine ausgehenden Links besitzt). Die Wahrscheinlichkeit für das Finden einer bestimmten Webseite entspricht genau ihrem PageRank.

6.6 Auswertung weiterer Hypertext-Informationen

Google legt neben dem vorgestellten PageRank-Algorithmus Wert auf weitere Hypertext-Informationen, um vielfältige Bewertungskriterien zu berücksichtigen.
Basierend auf der Annahme, dass der *Anchor Text* einer verlinkenden Seite die verlinkte Seite häufig besser beschreibt als diese sich selbst, kommt dem Anchor Text ein besonderer Stellenwert zu. Der Anchor Text bietet darüber hinaus die Möglichkeit, nicht-indizierbare Inhalte wie Bilder, Audio- und Videodateien zu erfassen.
Formatierungseigenschaften innerhalb einer Webseite werden ausgewertet. Wörter, die relativ hervorgehoben erscheinen (durch Schriftgröße, Fett- oder Kursivschreibung), gelten als wichtig. Interessant sind für Google insbesondere auch Wörter, die in der *URL*, dem *Title-* oder dem *Meta-Tag* vorkommen. Außerdem gewinnt ein Wort an Relevanz, je früher und je häufiger es genannt wird.
Eine Webseite als Ergebnis zu einer Suchanfrage gilt demnach als wichtig, wenn viele oder wichtige Seiten auf sie verlinken, wenn Wortvorkommnisse in URL, Title-/Meta-Tag oder dem Anchor Text verlinkender Webseiten gegeben sind und das angefragte Wort auf der Seite relativ hervorgehoben erscheint. Inwiefern die einzelnen Parameter gewichtet werden, wird in dem Paper nicht erläutert und kann als Googles Erfolgsrezept betrachtet werden.

6.7 Architektur

Google erfasst Informationen, indem Webseiten automatisch durchsucht und analysiert werden. Diesen Vorgang bezeichnet man als *Crawlen*. Jede der gecrawlten Webseiten wird in komprimierter Form in einem Repository gespeichert und erhält eine eindeutige *docID*. In der Indizierungsphase werden die Webseiten im Repository geparst und Informationen extrahiert. Zum einen werden neue Links zum Crawlen vorgemerkt und anhand der Linkstruktur wie oben beschrieben der PageRank der Webseite berechnet.
Zum anderen wird für jedes Dokument eine *Hit List* von Wörtern angelegt; jedes Wort, das in einem Dokument vorkommt, ist ein *Hit*. Für jeden Hit werden neben einer eindeutig kennzeichnenden *wordID* ebenfalls Meta-Informationen gespeichert, v.a. an welcher Stelle das Wort steht und wie es relativ formatiert ist. Besonderer Stellenwert kommt dabei Wörtern in URL, Title-, Meta- oder Anchor-Tag zu, man spricht in diesem Zusammenhang von *Fancy Hits*. Alle anderen Hits werden als *Plain Hits* bezeichnet. Da die Hit Lists den größten Teil des gespeicherten Datenvolumens ausmachen, ist eine effiziente Repräsentation unabdingbar. In dem Paper wird eine Datenstruktur vorgestellt, die pro Hit genau 2 Bytes aufwendet.
Die Hit Lists werden als *Forward Index* gespeichert, sortiert nach docID. Der Forward Index kann demnach Anfragen, welche Wörter in einem bestimmten Dokument vorkommen, effizient bearbeiten. Typischere Anfragen sind aber sicherlich in welchen Dokumenten ein bestimmtes Wort bzw. bestimmte Wörter vorkommen. Um diesen Anfragetyp beantworten zu können, wird der Forward Index während einer Sortierungsphase zu einem *Inverted Index* reorganisiert. Die Informationen bleiben unverändert, es liegt lediglich eine Sortierung nach wordID vor.

6.8 Suchanfragen

Bei Suchanfragen ist zwischen Suchen nach genau einem Wort (*One-Word-Queries*) und Suchen nach mehreren Wörtern (*Multi-Word-Queries*) zu unterscheiden.
One-Word-Queries werden bearbeitet, indem die Hit List des entsprechenden Wortes im Inverted Index betrachtet wird. Um das Finden dieser Hit List zu beschleunigen, wird eine Datenstruktur im Speicher gehalten, das *Lexicon*, welches für jedes Wort die wordID und einen Zeiger auf die Hit List im Inverted Index bereitstellt. Parameter für das Ranking der Ergebnisse sind die Informationen in Fancy/Plain Hits, die Häufigkeit des Auftretens des Wortes und der PageRank der Webseite.
Bei Multi-Word-Queries spielt zusätzlich *Proximität* eine entscheidende Rolle. Ein Ergebnis erhält ein besseres Ranking, wenn die Wörter nahe beieinander auftreten bzw. in einem Kontext sind. Die Informationen über Proximität kann allein durch die Hit Lists bestimmt werden. Dabei werden die Hit Lists der entsprechenden Wörter parallel durchsucht.

6.9 Weitere Entwicklungen

In dem Paper werden weitere Ideen genannt, deren Umsetzung für die Zukunft angedacht waren. Die Erkennung von gemeinsamen Wortstämmen in semantisch ähnlichen Wörtern (basierend auf dem sog. *Stemming* [61]) ist eine der wichtigen Weiterentwicklungen der damaligen Suchmaschine. So sollen beispielsweise Anfragen nach 'Suchmaschinenoptimierung' ebenfalls Webseiten mit Vorkommnissen von 'Suchmaschine', 'Optimierung' oder 'optimie-

ren' berücksichtigen. Im Vergleich zu reinem Keyword-Matching sind durch dieses Verfahren wesentlich vielfältigere Ergebnisse möglich, die dem Nutzer entgegen kommen.
Außerdem interessant war das Problem des erneuten Crawlens von Webseiten. Manche Webseiten ändern ihre Inhalte häufiger als andere und sollten aus diesem Grund häufiger recrawlt werden damit der Index aktuell bleibt. Deshalb sollte die Erfassung von statistischen Informationen über die Frequentierung von Updates einer Webseite implementiert werden um den Vorgang des Recrawlens zu kontrollieren.
Wird ein Query ein zweites Mal ausgeführt, bestünde die Möglichkeit, den Query nicht erneut ausführen zu müssen, indem auf ein gecachtes Ergebnis in einem schnelleren Speichermedium zurückgegriffen wird.
Als weiterer wichtiger Aspekt galt das aussagekräftige Zusammenfassen des Inhalts von Suchergebnissen.
Im Jahr 2000 indizierte Google bereits 1 000 000 000 Webseiten [7]. In diesem Jahr wurde erstmals eine Internationalisierung angeboten, um noch mehr Nutzer erreichen zu können [5]. Die Finanzierung des Projektes wurde mit *Google AdWords* realisiert [6]. 2004 folgte der Börsengang des Unternehmens [14].

Der wirtschaftliche Aufstieg des Unternehmens, die zahlreichen weiteren Anwendungen, die im Namen von Google bis heute entwickelt wurden, sowie die Kritik an Google in punkto Datenschutz sollen nicht Thema dieses Papers sein.

6.9.1 Google heute

Um den vielen Suchanfragen standhalten zu können, betreibt Google heute weltweit mehrere Rechenzentren, die jeweils die komplette Funktionalität der Suchmaschine bereitstellen. Eine Suchanfrage wird im Idealfall an das netztopologisch nächste Rechenzentrum geleitet und von diesem beantwortet. Auf diese Weise kann eine Verteilung der Last realisiert und sichergestellt werden, dass der Ausfall eines Knotens kein Ausfall des ganzen Systems bewirkt. Die Last eines ausgefallenen Knotens kann auf andere übertragen werden. Um schnelle Antwortzeiten zu ermöglichen, werden alle Daten redundant in jedem Rechenzentrum gespeichert und können parallel genutzt werden. Ein Richtwert für Antwortzeiten von höchstens einer halben Sekunde ist angestrebt. [3]

Im Web werden viele rechtsverletztende Inhalte angeboten. Aufgrund gesetzlicher Auflagen entfernt Google solche Suchergebnisse. Beispielsweise Webseiten, die Inhalte ohne entsprechendes Urheberrecht anbieten, sollen nicht angezeigt werden. Es besteht die Möglichkeit, eine solche Rechtsverletzung an Google zu melden. [15]
Kritik an Google wird wegen Eingriffen in den automatisierten Ranking-Betrieb laut [28]. Dabei werden Ergebnisse entfernt, deren Inhalte in einem betreffenden Land verboten sind. Google gewährt damit der Politik eines Staates Einfluss auf die Informationsverbreitung.

6.10 Schwachstellen von Google

6.10.1 Suchmaschinenoptimierung

Um das Ranking einer Webseite zu verbessern, werden oft technische Tricks eingesetzt [11]. Durch systematisches Analysieren des Verhaltens von Webcrawlern können diese gezielt ma-

nipuliert werden. Dies wird durch bewusstes Platzieren von Keywords erreicht; so ergeben sich lange Hit Lists (bei Auftreten in URL, Title- oder Meta-Tag insbesondere fancy hits). Tricks sind beispielsweise weiße Schrift auf weißem Hintergrund oder für Menschen unlesbar kleine Schrift.
Auch eine intensive Verlinkung auf die Webseite beschert dieser durch den PageRank-Mechanismus mehr Relevanz. Webseiten, die in diesem Zusammenhang durch *Spamming* auffallen, werden häufig aus dem Suchindex ausgeschlossen.
Google beschreibt, wie das Gefundenwerden von Webseiten erleichtert wird [18] [17].

6.10.2 Google Bombing

Das starke Gewichten von Anchor Text erwies sich im Lauf der Zeit als problematisch, da es für Manipulation von Suchergebnissen empfänglich ist. So wurden in der Vergangenheit Angriffe auf u.a. Microsoft und George W. Bush arrangiert, indem Suchanfragen nach 'more evil than satan himself' [57] oder 'miserable failure' [26] bewusst auf deren Webseiten gelenkt wurden. Möglich wird dies, wenn viele oder wichtige Seiten einen Link auf besagte Webseiten mit eben diesem Anchor Text setzen. Jene Webseiten gewinnen dadurch an Relevanz für die im Anchor Text genannten Wörter. Solche Angriffe werden als *Google Bomb* [38] bezeichnet. Google sieht dieses Problem heute als Zeichen der Demokratie des Internets und arbeitet an algorithmischen Lösungen [9] [10].

6.11 Zusammenfassung

Primärziel von Google war von Anfang an die Qualität von Suchergebnissen zu verbessern. Dies wird dadurch erreicht, dass Ergebnisse nach Relevanz geordnet sind. Zusätzlich zum Inhalt von Webseiten werden hierfür auch Hypertext-Informationen ausgewertet, grundlegend neu ist dabei die Tatsache, dass die Form von Dokumenten berücksichtigt wird. Der PageRank-Algorithmus hat sich in der Praxis bewährt und wurde von vielen Suchmaschinen adaptiert. Google hat dadurch einen wesentlichen Teil zur heutigen Informationsinfrastruktur beigetragen und ist mit über 90% aller Suchanfragen Marktführer unter den Internet-Suchmaschinen [27]. Die Architektur von Google wurde von vornherein auf Skalierbarkeit ausgelegt, da der hohe Stellenwert des Webs und die schnelle Informationsverbreitung richtig erkannt wurden.

Teil IV

Data Warehousing / Data Mining

Jan-Felix Schwarz
jan-felix.schwarz@student.hpi.uni-potsdam.de

7

Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals

Inhaltsangabe

7.1 Abstract

In dieser Seminarausarbeitung wird der wesentliche Inhalt des Papers „Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals“ [20] von Jim Gray et al. zusammengefasst. Der Data Cube ist ein verallgemeinerter Aggregationsoperator, der in der Analyse großer Datenmengen mit vielen Dimensionen zum Einsatz kommt. Neben den Erläuterungen zu dem Konzept und der Berechnung des Data Cubes, gibt diese Arbeit einen Überblick zu praktischen Anwendungen und der heutigen Relevanz des vorgestellten Operators.

7.2 Autoren

Jim Gray (geboren 1944) war ein Spezialist für Datenbanken und Transaktionen bei Microsoft Research. Er war einer der führenden Wissenschaftler der Informationstechnologie und wurde für seine Leistungen und bahnbrechenden Forschungsbeiträge im Jahre 1998 mit dem Turing Award ausgezeichnet. Vor seiner Anstellung bei Microsoft arbeitete Gray für Digital, Tandem, IBM und AT&T. Am 28. Januar 2007 verschwand Gray auf tragische Weise während eines Solo-Segelausflugs auf offener See. Trotz massiver High-tech Unterstützung bliebt die Suche nach ihm erfolglos und wurde schließlich am 31. Mai 2007 eingestellt.

Die Co-Autoren des Papers waren Kollegen Grays bei Microsoft und IBM.

7.3 Hintergrund und Problemstellung

Relationale Datenbanken sind optimiert um möglichst viele Transaktionen in möglichst kurzer Zeit ausführen zu können. Diese Transaktionen setzen sich zu etwa gleichen Teilen aus Einfüge- und Anfrageoperationen zusammen. Für aufwendige Analyseanfragen auf großen Datenmengen sind diese klassischen DBMS allerdings nicht geeignet. Typische Anfragen der Datenanalyse betrachten Aggregationen über viele Dimensionen (Spalten) der Daten um Informationen zu Statistiken, Trends und Kategorisierungen zu liefern. Für diese Aufgaben werden daher spezialisierte Datenbanksysteme entwickelt und eingesetzt.

7.3.1 Data Warehousing

Data Warehouses sind speziell für das Data Mining zugeschnittene Datenbanken. Es handelt sich dabei um zentrale „Datenlager“ , deren Inhalt aus verschiedenen Quellen zusammengeführt wird. Die gesammelten Rohdaten werden gefiltert und aufbereitet und in die strukturierten Datenbestände integriert. Das Data Warehouse ist auf die langfristige Speicherung der Daten ausgelegt. Dabei sind kurze Ausführungszeiten der Transaktionen von geringerer Relevanz.

7.3.2 OLAP

OLAP steht für „On-Line Analytical Processing“ und ist ein Überbegriff für Technologien zum schnellen und detaillierten Analysieren von in Data Warehouses gesammelten Daten. Der Cube ist die OLAP zugrunde liegende Datenstruktur. Der in dem vorgestellten Paper präsentierte Operator liefert somit die Grundlage für solche Systeme.

7.4 Generalisierung von Aggregationsoperationen

Aggregationsfunktionen in SQL erzeugen in Verbindung mit dem klassischen GROUP BY Operator o-dimensionale Ergebniswerte. Das heißt, dass für eine durch GROUP BY spezifizierte Gruppe genau ein Aggregationswert berechnet wird, wie zum Beispiel die Anzahl aller vorhandenen Tupel (COUNT() Aggregationsfunktion) oder der Durchschnittswert einer Spalte über alle Tupel der Gruppe hinweg (AVG() Aggregationsfunktion). Wie oben beschrieben, werden in der Praxis sehr viel komplexere Analyseanfragen benötigt, die eine wesent-

lich höhere Flexibilität der Aggregationsberechnungen erfordern. Aggregationen werden über mehrere Dimensionen betrachtet um aussagekräftigere Analyseergebnisse zu erhalten. Um solche Anfragen geeignet formulieren und effizient ausführen zu können sind eine entsprechende Unterstützung durch das DBMS und passende Syntaxerweiterungen der Anfragesprache erforderlich.

7.4.1 Roll-up

Ein Beispiel hierfür ist der Roll-up Operator, der das Aggregieren von Datensätzen nacheinander über mehrere Dimensionen hinweg erlaubt. Eine Roll-up-Anfrage über die Dimensionen „Tag", „Monat", „Jahr", liefert zunächst Aggregationsergebnisse für Gruppen von Tupeln, die in Monat und Jahr übereinstimmen. Die Dimension „Tag" wird dabei „aufgerollt". Auf der nächsten Ebene liefert die Anfrage über die Monat-Dimension hinweg zusammengefasste Ergebnisse, also einen Aggregationswert pro Jahr. Das Aufrollen der letzten Dimension liefert schließlich ein Super-Aggregat über alle Tupel der Relation hinweg. Dabei ist der Operator allerdings asymmetrisch, was bedeutet, dass die Reihenfolge, in welcher die Dimensionen aufgerollt werden, relevant für das Anfrageergebnis ist.

7.4.2 Cross-tab

Cross-tabs sind die symmetrische Erweiterung eines zweidimensionalen Roll-ups. Sie enthalten somit die Aggregationswerte für beide möglichen Reihenfolgen eines Roll-ups über zwei Dimensionen. Abbildung 7.1 zeigt die graphische Repräsentation einer exemplarischen Cross-tab, die Verkaufszahlen von Chevy-Fahrzeugen über die Dimensionen Farbe und Jahr aggregiert. Ein Roll-up über das Jahr liefert die rechts stehenden zusammengefassten Summen, während ein Roll-up über die Farbe die unten stehenden Summen erzeugt. Der Gesamtwert wird auf dem Ergebnis eines der beiden Roll-ups aufbauend durch ein Roll-up über die jeweils andere Dimension erzeugt.

Chevy	1994	1995	Total (ALL)
Black	50	85	135
White	40	115	155
Total (ALL)	90	200	290

Abbildung 7.1: Graphische Repräsentation einer Cross-tab

7.4.3 Data Cube

Der Data Cube Operator ist wiederum eine Verallgemeinerung der Cross-tab für beliebig viele Dimensionen. Das bedeutet, dass das Ergebnis einer Cube-Anfrage über n Spalten alle möglichen Roll-ups über diese Spalten beinhaltet. Dabei gibt es für jede Reihenfolge der n Dimensionen genau einen separaten Roll-up. Die Ergebnismenge des n-dimensionalen Data Cubes entspricht somit der Vereinigung der Ergebnismengen von n! vielen Roll-ups über n Dimensionen.

Abbildung 7.2 veranschaulicht den dreidimensionalen Fall des Data Cubes. Wie im Beispiel zuvor werden Autoverkaufszahlen betrachtet - nun zusätzlich über die Dimension „Hersteller“ hinweg. Für jeden Hersteller lässt sich eine Cross-tab innerhalb des Cubes ausmachen, die zu der in Abbildung 7.1 exemplarisch aufgeführten Cross-tab für Chevy äquivalent ist. Zusätzlich ergibt sich eine Cross-tab, die alle Hersteller zusammengefasst betrachtet und auf der Vorderseite des abgebildeten Cubes zu sehen ist.

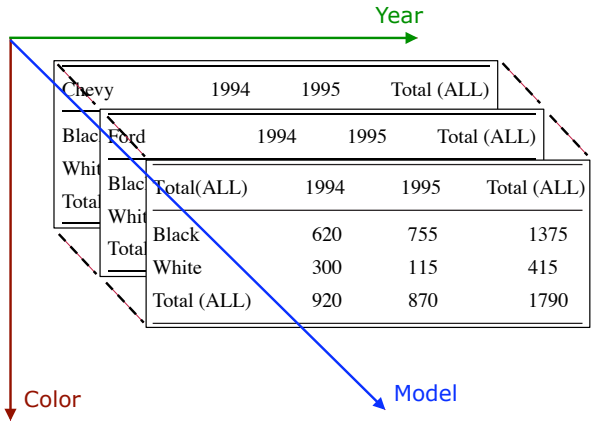


Abbildung 7.2: Graphische Repräsentation eines dreidimensionalen Data Cubes

Anschaulich gesehen liegt jedes einzelne Tupel der Ausgangsrelation innerhalb des abgebildeten Würfels. Die Position wird dabei durch die Koordinaten, also die Wertausprägungen des Datensatzes für die einzelnen Spalten festgelegt. Außen am Würfel hängen die zusätzlich hinzugekommenen Aggregationstupel: Auf der Unterseite befinden sich das Ergebnis des Roll-ups über die Farbe, die rechte Seite enthält aggregierte Tupel über die Jahr-Dimension hinweg und die Vorderseite entsprechend für die Hersteller-Dimension. Entlang der Schnittkanten dieser drei Seiten befinden sich die eindimensionalen Aggregationstupel. Der Schnittpunkt dieser drei Kanten an der unteren vorderen rechten Ecke enthält das o-dimensionale Super-Aggregat, nämlich die Gesamtsumme von 1790 Autoverkäufen.

In dem Paper wird dargestellt, wie sich das Ergebnis eines Data Cubes als relationale Tabelle darstellen lässt, so dass der Operator in DBMS implementiert werden kann. Dazu werden ’ALL’ Werte eingeführt, die die Gesamtmenge aller Werte repräsentieren, die für eine Spalte von den aggregierten Tupeln angenommen werden. Der Vorteil, der sich daraus ergibt ist, dass Data Cubes wie herkömmliche Relationen angefragt werden können. Außerdem eröffnet sich damit die Möglichkeit, in herkömmlichen DBMS durch relativ geringe Eingriffe Data Cubes materialisiert speichern zu können. Zu bemerken ist aber, dass auf Grund der Mengeninterpretation des ’ALL’ Wertes, die Atomarität einzelner Spaltenwerte nicht mehr gegeben ist.

7.4.4 Implementierung des Cubes

Der naive Berechnungsalgorithmus für den Cube lässt sich wie folgt in natürlicher Sprache ausdrücken:

```
(1) Initialisiere für jede Zelle des Cubes die Berechnung der
    Aggregationsfunktion
```

```
(2) Für jedes neue Tupel (x1, x2, ..., xN, v) der eingelesenen Relation:
        (2.1) Für alle Zellen mit Aggregationen über mindestens einer der
              Spalten x1 bis xN:
                (2.1.1) Füge v zum Zwischenergebnis der Aggregation für die
                        Zelle hinzu
(3) Berechne für jede Zelle des Cubes das Endergebnis der Aggregations-
    funktion unter Zuhilfenahme der gespeicherten Zwischenergebnisse
```

Dieser Weg ist jedoch ineffizient und für die Berechnung von Cubes über viele Dimensionen nicht praktikabel. Das Paper präsentiert für einen Großteil der üblichen Aggregationsfunktionen effiziente Berechnungsstrategien. Dabei wird die Wiederverwendbarkeit von vorberechneten Zwischenergebnissen für die Berechnung des Super-Ergebnisses ausgenutzt. Diese Wiederverwendbarkeit ist allerdings nur für bestimmte Klassen von Aggregationsfunktionen gegeben. Dazu wird zwischen algebraischen, distributiven und holistischen Aggregationsfunktionen unterschieden. Während für algebraische und distributive Funktionen die Superaggregate auf Basis der Subaggregate berechnet werden können, ist dies für holistische Funktionen nicht möglich, da für diese die für die Berechnung erforderlichen Zwischenergebnisse potenziell unendlich groß werden können.

7.5 Heutige Relevanz des Data Cubes

Während der letzten 10 Jahre ist die Nachfrage nach sogenannten *Decision Support Systems* drastisch angestiegen. Dabei handelt es sich um Informationssysteme, die große Unternehmen und Organisationen bei strategischen Entscheidungen unterstützen. Um Analysen zur Marktentwicklung durchführen zu können, werden sehr große Datenmengen aus unterschiedlichen Quellen gesammelt und in großen Data Warehouses gespeichert. Diese Rohdaten werden als Ausgangsbasis für Data Mining und für die Aggregation zu betrieblichen Kennzahlen genutzt.

Das vorgestellte Paper war seiner Zeit voraus und liefert die Grundlagen für solche Systeme.

7.5.1 Rolle des Data Cubes in OLAP Anwendungen

Der Data Cube erlaubt die Analyse von Datenmengen in jeder beliebigen Granularität, in dem die interessanten Sub-Cubes, Flächen oder Zellen des Würfels ausgewählt werden. Somit bildet der Cube-Operator die Basis für OLAP Anfragen wie Roll-ups und Drill-downs. Drill-down ist die Umkehroperation des Roll-ups, erlaubt es also die Daten mit feinerer Granularität zu betrachten. Zuvor aggregiert betrachtete Werte werden nach dem Drill-down in der entsprechenden Dimension unterschieden.

Abfragen auf einem Cube werden üblicherweise mit einer Multi-dimensionalen Anfragesprache formuliert. Solche Anfragen liefern Teilmengen des Cubes. Diese können anschaulich gesehen durch einzelne Zellen, zweidimensionale Scheiben (*Slices*) oder mehrdimensionale Sub-Cubes (*Dices*) repräsentiert sein.

In Data Warehouses wird der Data Cube typischerweise materialisiert gespeichert, so dass alle relevanten Analyseanfragen in kürzester Zeit durch eine einfache Selektion von Teilbereichen des Cubes beantwortet werden können. Es gibt sogar Ansätze, die die konzeptionelle multidimensionale Struktur des Cubes auf die physikalische Speicherebene übertragen, indem die Daten in multidimensionalen Arrays organisiert werden. Durch diese sogenannte MOLAP

Architektur können die aggregierten Kennzahlen persistent gespeichert und somit schneller abgefragt werden.

7.6 Zusammenfasung

Durch die schrittweise Generalisierung des klassischen Group-By Operators, über den Roll-up und weitere verallgemeinerte Aggregationsoperatoren, wurde schließlich der vollständig generalisierte Cube Operator entwickelt. Der Cube kann für einen Großteil der gebräuchlichen Aggregationsfunktionen effizient berechnet werden.

Der Data Cube enthält den gesamten Raum aller möglichen Aggregationen über die betrachteten Dimensionen und ermöglicht somit komplexeste Analyseanfragen. Der Cube ist in aktuellen OLAP-Anwendungen implementiert und dient somit als Berechnungsgrundlage für Data Mining Anfragen in großen Unternehmens-Informationssystemen.

Über die in dieser Ausarbeitung zusammengefassten Themen hinaus, wird in dem Paper eine Syntaxerweiterung für SQL vorgeschlagen, die die einfache Formulierung von Anfragen mit Roll-up und Cube Aggregationen unterstützt. Außerdem wurde im Rahmen dieser Zusammenfassung weniger detailliert auf die Charakteristika von Aggregationsfunktionen eingegangen. In dem Paper wird eine Klassifizierung solcher Funktionen definiert und dargestellt, wie Anwender ein Datenbankmanagementsystem durch zusätzliche, nutzerdefinierte Aggregationsfunktionen erweitern können.

Jan Oberst
jan.oberst@student.hpi.uni-potsdam.de

8

Efficient Data Clustering and How to Groom Fast-Growing Trees

Inhaltsangabe

Clustering is one of the most important unsupervised learning problems. It deals with finding a structure in a collection of unlabeled data points.

Pattern recognition uses clustering to filter out faces from pictures or video feeds. In medicine clusters represent different types of tissue and blood in three dimensional images, for example to help diagnose tumors. Biologists find functionally related proteins or genes that are co-regulated by scanning large DNA sequences.

This report focusses on the BIRCH clustering algorithm[64]. The authors of BIRCH focus on a scalable architecture. In this report we will discuss the scalability aspects in the field of clustering algorithms and especially those of the BIRCH algorithm. In the section on *Clustering Feature* (CF) trees we then present the main contribution of BIRCH in more detail. A review on further research based on the BIRCH paper concludes this report.

8.1 The BIRCH Algorithm

BIRCH uses a hierarchical data structure called Clustering Feature tree (CF tree) for partitioning the incoming data points in an incremental and dynamic way. This section is organized as follows. First we give an overview of the four phases in the BIRCH algorithm and the construction of the CF tree. Theoretical basics of Clustering Features are given and it is described how distance metrics can work solely on Clustering Features. We finish this section by revisiting how BIRCH can fit large datasets into RAM using these Clustering Features to compress the data.

8.1.1 BIRCH's Four step process

BIRCH employs four different phases during each clustering process.

1. Linearly scan all data points and insert them in the CF tree as described earlier.
2. Condense the CF tree to a desirable size depending on the clustering algorithm employed in step three. This can involve removing outliers and further merging of clusters.
3. Employ a global clustering algorithm using the CF tree's leaves as input. The Clustering Features allow for effective distance metrics. The authors describe different metrics with time complexity of $O(N^2)$. This is feasible because the CF tree is very densely compressed at this point.
4. Optionally refine the output of step three. All clusters are now stored in memory. If desired the actual data points can be associated with the generated clusters by reading all points from disk again.

8.1.2 B^+ Trees for Clustering

BIRCH's CF tree is a height-balanced tree modeled after the widely used B^+ tree. With such a tree no insert or update mechanisms will influence the balance of the tree. This allows fast lookups even when large datasets have been read. It is based on two parameters. CF nodes can have at maximum B children for nonleaf nodes and a maximum of L entries for leaf nodes. As discussed earlier T is the threshold for the maximum diameter of an entry.

A CF tree is built on the fly as the data is scanned. At every level of the tree a new data data point is inserted to the closest subcluster. Upon reaching a leaf the point is inserted to the closest entry, as long as it is not overcrowded (diameter $D > T$ after the insert). Otherwise a new CF entry is constructed and the data point inserted. Finaly all CF statistics are updated for all nodes from the root to the leaf to represent the changes made to the tree. Since the maximum number of children per node (branching factor) is limited, one or several splits can happen. Splits also result in updated CF values which have to be propagated back to the root of the tree.

8.1.3 Clustering Features

In the BIRCH tree a node is called a Clustering Feature. It is a small representation of an underlying cluster of one or many points. BIRCH builds on the idea that points that are close

enough should always be considered as a group. Clustering Features provide this level of abstraction.

Clustering Features sum up a group of points that are close. The authors name such nodes *Leaf Entries*. Every entry represents of a number of data points that are close. The authors further distinguish between *leaf nodes* and *nonleaf nodes*. A leaf node has many entries and represents a cluster that itself is made up of the subclusters of these entries. A nonleaf node follows this principle and represents a cluster that is made up of its child-node subclusters. As suggested before every node in the CF tree is a Clustering Feature. The whole CF tree consists of clusters that themselves have subclusters. This is a classic example of hierarchical clustering[31].

Clustering Features are stored as a vector of three values: $CF = (N, \vec{LS}, SS)$. The linear sum ($\vec{LS}$), the square sum (SS), and the number of points it encloses (N). All of these metrics can be calculated using only basic math:

$$SS = \sum_{i=1}^{N} \vec{X}_i^2 \qquad\qquad \vec{LS} = \sum_{i=1}^{N} \vec{X}_i$$

If divided by the number of points in the cluster the linear sum marks the *centroid* of the cluster. As the formulas suggest both of these values can be computed iteratively. Any Clustering Feature in the tree can be calculated by adding its child Clustering Features:

$$CF_1 + CF_2 = (N_1 + N_2, \vec{LS}_1 + \vec{LS}_2, SS_1 + SS_2)$$

8.1.4 Distance metrics

Clustering algorithms use distance measure to discover close data points.

Even the highly compressed Clustering Features still allow the use of exhaustive distance metrics. The authors describe how five different metrics, including the Euclidean and Manhattan distances work on Cluster Features to measure the distance between points and clusters, and the intra-cluster distance.

8.1.5 Data compression with CF trees

Every Clustering Feature encloses points that are "close". This closeness criterion is based on the *diameter* of a cluster. Because Clustering Features are vector-based the calculation of diameter D or radius R is straightforward ($\vec{X}_i$ are the vectors in a cluster, $\vec{X}0$ is the cluster's centroid):

$$D = \left(\frac{\sum_{i=1}^{N}\sum_{j=1}^{N}(\vec{X}_i - \vec{X}_j)^2}{N \cdot (N-1)}\right)^{\frac{1}{2}} \qquad\qquad R = \left(\frac{\sum_{t=1}^{N}(\vec{X}_t - \vec{X}0)^2}{N}\right)^{\frac{1}{2}}$$

The construction of Clustering Features immediately compresses all incoming data. Even for hundrets of close points only one Clustering Feature has to be stored in RAM. BIRCH can be set to use more or less extensive data compression by setting the *threshold* value T. New points can only be inserted to en entry if its Clustering Feature is smaller than T after the insertion. If the point does not fit it is inserted to a newly created entry.

Noisy datasets generally pose a problem to clustering algorithms. Outliers will generally not fit in any existing cluster, and therefore bias the algorithm to generate clusters with only one data point. BIRCH will never add outliers to existing clusters, since T already limits the cluster radius. To resolve this problem BIRCH automatically filters outliers during tree rebuilts and writes them to disk. Once it finishes the CF tre it will try to re-fit those outliers. This mechanism improves clustering quality because it avoids clusters with only one entry, and increase performance by removing outliers from the CF tree.

Perfect cluster size It can be shown that there is no absolute best criterion which would be independent of the final aim of the clustering: every clustering problem is different. For the same reason there cannot be a "best" cluster size. The more points can be compacted to one Clustering Feature the better the compression ratio gets. On the other hand more information is lost if more points are represented by a cluster. A balance between better level of detail (smaller clusters) and better compression ratio (larger clusters) is cruical for good overall clustering quality.

BIRCH solves this problem by always trying to maximize RAM usage (which maximizes precision). The algorithm starts with maximum precision at $T = 0$ and as the CF tree grows larger than the available memory it iteratively tries to find suitable cluster sizes. T has to be increased to be larger than the smallest distance between two entries in the current tree. This will cause at least these two entries to be merged into a coarser one, it reduces the amount of clusters produced, and clusters will grow larger in diameter.

Threshold heuristics Rebuilding the CF tree is generally fast enough to allow an iterative approximation to a "best" suitable threshold. All performance analysis in the original paper start with $T = 0$ and the authors note that beginning with a good initial value for T would save about 10% time[65]. The authors also supply basic heuristic to derive a new threshold from an existing CF tree.

Once BIRCH has to rebuild the CF tree it needs to increase T based on some measure of cluster volume. BIRCH maintains a record of leaf radii as a function of the number of points in the cluster. Using least-squares regression it can estimate to future growth of the tree, and extrapolate an "expansion factor", which will generally be higher for high-volume trees. The CF tree is then rebuilt using T multiplied with the expansion factor as the new threshold.

8.2 Test of time

The authors of BIRCH have achieved groundbreaking results for the scalability of clustering algorithms. The following paragraphs discuss the four most important contributions of the BIRCH paper and their impact on the data mining community.

On the other hand, BIRCH is sensitive to the order of the input stream, only functions properly on spherical data, and is limited to low dimensional datasets. To conclude this section we show how these problems have been solved in other systems.

Single-pass Today many datasets are too large to fit into main memory. From this point on the dominating cost of any clustering algorithm is I/O, because seek times on disk are orders of a magnitude higher than RAM access times.

BIRCH can typically find a good clustering with a single scan of the data. After the initial CF tree has been built additional scans can help to improve the clustering quality.

Adaptive compression As described above BIRCH frequently rebuilds the whole CF tree with a different threshold setting and tries to merge as many CF nodes as possible. The rebuild happens sufficiently fast since all needed data is already in RAM. At the same time outliers are removed from the tree and are stored to disk.

BIRCH is one of the first algorithm to effectively use an in memory index-structure for spatial data. The authors of STING[60] employ similar techniques and claim to have outperformed BIRCH by a very large margin (despite not having compared the two algorithms side by side). The CURE algorithm[23] uses a similar approch to BIRCH's CF tree: Each cluster is represented by a certain number of "well scattered" points. CURE relies on drawing random samples from the dataset to derive starting clusters. This preclustering phase is somewhat similar to the first phase in the BIRCH algorithm.

Non-local A new data point is always added to the closest subcluster in the CF tree. The first advantage of this approach is the limited amount of comparisons that are needed when a data point is inserted to the tree. Being based on a B^+-tree insert operations require at most $\log(n)$ operations. Data points are inserted to the *closest* subcluster at all levels of the tree. This means that only those subclusters are checked that have at all a chance to absorb the added data point.

Suspendable, stoppable, resumable Even scalable architectures like BIRCH have runtimes of many hours when large datasets have to be clustered. At this point it is impossible to restart the algorithm every time a customer proceeds to checkout[36]. Once built a CF tree can be used to sequentially add as much data as needed. If the growing CF tree hits the memory limit it is rebuilt and more data can be added.

Order sensitivity BIRCH is sensitive to the order of the input stream. Different orders of the same input data may result in different clusters[24]. This problem becomes even more relevant when the input stream itself is sorted. Each data point is immediately added to the CF tree and all Clustering Features on the path to the inserted leaf have to be changed. If nodes had to be split during the insert even more Clustering Features change.

Two points with the same coordinates could very well end up in different clusters. Both rebuilding the CF tree and the refinement phase 4 of the algorithm try to resolve this problem, but fail to eliminate it completely[23].

Non-Spherical BIRCH uses the concepts of radius and diameter to control the boundary of a cluster. This approach may not work well when clusters are not spherical, because clustering is limited to a vector space. BUBBLE[13] allows single-pass scalable clustering in arbitrary metric spaces based on the BIRCH framework.

Low Dimensionality Agrawal et al.[1] tested three algorithms on a dataset with varying dimensionality from 5 to 50 dimensions. At more than five dimensions BIRCH was unable

to identify the true clusters. BIRCH gives equal importance to all the dimensions when computing the distance between two points. Clustering algorithms with a specific focus on high dimensionality tend perform better than BIRCH [39, 1, 46].

8.3 Conclusion

The *SIGMOD Test of Time* award is a significant recognition that is given to a paper submitted to the conference ten years ago. To the SIGMOD judges BIRCH is the most important contribution from 1996 and that with the most impact today.

BIRCH was the first algorithm to run in $O(N)$ time. Its authors claim that their algorithm performed 15 times better than CLARANS[43], the leading spatial clustering algorithm at the time. While BIRCH was still limited to low-dimensional and vectorized datasets, but sparked lots of research on how to effectively cluster large datasets. The algorithms CURE and CLIQUE[1] showed that higher dimensionality can be clustered effectively. With BUBBLE, CURE, and WaveCluster[56] also irregularly shaped clusters became possible.

The reader is referred to data clustering surveys[33, 30, 62] for a more in-depth comparison of the aforementioned clustering algorithms.

8.4 Appendix: Clustering Bundesliga News

Ligageschichte[1] gathers many different sources around the German soccer-league Bundesliga and displays them in many different ways. One of these so-called "portlets" is a classic example for data clustering: Finding popular news topics and clustering them accordingly. The following sections give a short introduction on how BIRCH can be used to cluster news texts. We test how different settings change the CF tree and analyze the resulting clusters. We also compare BIRCH against the *k*-means clustering Ligageschichte uses today to find out if BIRCH would be the better choice.

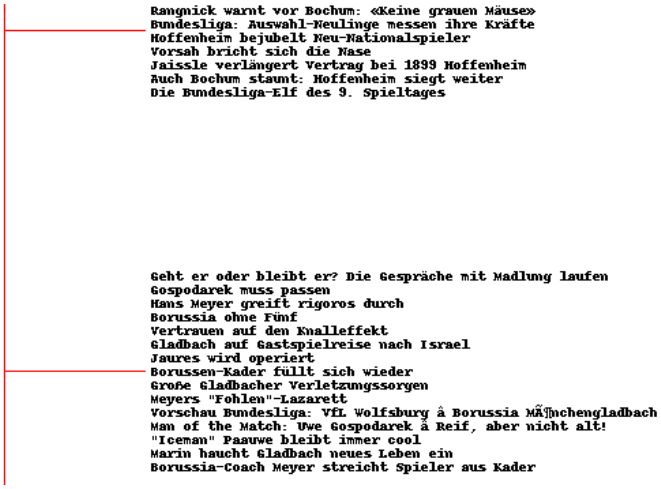


Abbildung 8.1: Two example clusters with news covering Hoffenheim and Mönchengladbach

[1] http://www.ligageschichte.de

8.4.1 Test Implementation & Dataset

BIRCH's authors limited their experiments to two dimensions. In order to test BIRCH on a high-dimensional dataset and to experiment with the different settings for branching factor and threshold we implemented the CF tree in Python. Please note that this section is based completely on phase one of the BIRCH algorithm. Furthermore we exclude automatic tree rebuilds because we're interested in different CF-tree layouts. The complete source code of our implementation can be found online[2]. All tests were run on a 2.5 GHz Intel Core 2 Duo with 4 GB of RAM. As a result even large CF trees with $T = 0$ will fit into RAM. The smaller test data set with 1000 data points also still fit into disk cache.

The large dataset consisted of 58,096 news texts, including sources from Kicker.de, Bundesliga.de, FAZ.net, and Fussball24.net. Duplicate removal was not performed before clustering. However, all testing with k-means had to be limited to a smaller dataset of only 10,000 texts, because its implementation did not use the main memory as efficiently as it could have.

Dimension (Entity)	TF/IDF value
'hingegen'	0.0434
'niederlage'	0.028
Team{VfB Stuttgart}	0.189
Player{Mario Gomez}	0.040
'roter'	0.060
'fans'	0.175
Player{Maik Franz}	1.299
Team{Bayer 04 Leverkusen}	0.642
⋮	⋮
Team{Karlsruher SC}	1.010

Abbildung 8.2: Example vector as prepared for the BIRCH algorithm

To generate a proper input dataset for BIRCH we transformed the texts to high-dimensional vectors. Each text was represented by a single vector and each word represented by a single dimension. Ligageschichte uses a fine-tuned Named Entity Recognition (NER) algorithm[3] The final step in preparing the news texts for clustering is to assign the proper values to each dimension. The TF/IDF value represents the importance of a word based on the number of occourances in the text and the number of occourances in all texts combined. Term weights for named entities are computed accordingly. Figure 8.2 shows the example of a text vector as used in our example clustering.

8.4.2 Impact of different threshold settings

We tested the algorithm with five different threshold settings $T = \{0.5, 1, 2, 4, 8\}$. Our sample dataset were 1000 normalized and cleaned news items from November 2008. As a result we used a total of 18,099 dimensions. The branching factor was set to $B = 7$, which we found to be a good fit for our dataset.

[2] http://www.janoberst.com/_academics/2009_01_27-BIRCH-Python-Implementation/

[3] More details about Ligageschichte can be found at http://www.janoberst.com/_academics/2008_07_16-Ligageschichte-Clustering-LSI-Semantic-SVD.pdf

When the threshold is set as low as $T = 1$ most 'clusters' only consist of two or three items. Accordingly, the amount of outliers is low as BIRCH uses the proposed limit of 25% of the average cluster size to find out which clusters are to be marked as outliers. The overall clustering result is not acceptable.

Large thresholds generally produce fewer and bigger clusters. The "average cluster size" as described in Table 8.1 represents the amount of news items per cluster. The most promising results were created with $T = 40$. At that point the overall quality of the clustering was even better than the already highly tweaked clusters that are currently used by Ligageschichte.

With $T = 10$ almost a third of the clusters would have been marked as outliers. As discussed in Section 8.2 we suspect that the very high dimensionality of our dataset is responsible for this behavior. However, the currently used *k*-means approach does not have any outlier detection at all. Data points that actually are outliers are mixed with proper clusters and worsen the results. Ligageschichte already tries to eliminate this problem by removing the largest and the smallest cluster after the first few passes restarting the algorithm. With BIRCH outliers could be discarded much easier, because they do not influence clustering quality.

8.4.3 Compared to *k*-means

We conducted a second experiment where we compared the runtime of BIRCH to that of the standard *k*-means that is built into Ligageschichte.

We had to limit the amount of news items to cluster to 10,000, because the *k*-means implementation in Ligageschichte already used 930MB of RAM at that point. This caching is done for performance reasons, because the algorithms would need to re-fetch all data from disk for each of the 30 iterations. We suspect that the performance differences will grow with larger test datasets.

k-means was set to use 22 iterations and the number of clusters k was fixed at 30. This amount of clusters has proven to be adequate. Based on the tests described above we chose $T = 40$ as an appropriate threshold for BIRCH.

The clustering output generally consists of 18 clusters for the 18 Bundesliga teams. Another large cluster usually represent news reports on a whole weekend (where all teams appear in the same text). News about player transfers, injured players and the performance of the referees are generally also placed in individual clusters. The remaining (small) clusters are usually outliers which are discarded by Ligageschichte.

T	# Clusters	# Splits	% outliers	Average cluster size	Average cluster radius
1	491	258	0.000	2.020	0.419
2	362	210	0.000	2.740	0.887
4	232	176	28.879	4.276	1.694
8	148	107	17.568	6.703	3.677
16	83	54	18.072	11.952	7.423
32	44	35	22.727	22.545	15.933
40	39	29	20.513	25.436	18.718
64	24	12	20.833	41.333	33.841

Tabelle 8.1: Results of the BIRCH clustering phase 1

Algorithm	Runtime	# Clusters
k-means	209 minutes	30
BIRCH	97 minutes	38

Abbildung 8.3: Comparison of *k*-means and BIRCH phase 1 for 10,000 news items

Most of the time in both algorithm is lost because of the barely optimized Python implementation. Since real-world clustering experiments are only comparable to themselves, these results do not represent the algorithmic performance differences between *k*-means and BIRCH.

8.4.4 Conclusion

After tweaking of both algorithms the results of BIRCH were slightly better than those of the standard *k*-means approach built into Ligageschichte while being two times as fast.

This comparison between BIRCH and *k*-means makes actually sense in the case of Ligageschichte. We are trying to figure out which clustering algorithm can replace the quite ineffective *k*-means implementation. The observed speedup of factor two is certainly worth considering a switch.

We suspect that BIRCH would greatly benefit from using a larger dataset. It would also be interesting to see how BIRCH performs with the automatic generation of T using the heuristics described in Section 8.1.5.

Teil V

Stream-Based Data Management

Benjamin Emde
benjamin.emde@student.hpi.uni-potsdam.de

9

Sequenzdatenbanken

Inhaltsangabe

9.1 Übersicht

Daten aus dem echten Leben unterliegen häufig einer Ordnung. Herkömmliche relationale Datenbanken speichern ihre Tupel jedoch ungeordnet auf der Festplatte. Besondere Anfragetypen, wie der gleitende Durchschnitt der in zahlreichen Finanzapplikationen genutzt wird, sind bislang durch SQL nicht hinreichend abgedeckt oder wenig effizient. Sequenzdatenbanken sind eine neue Art von Datenbanksystemen, die tatsächlich Nutzen aus der Sequentialität der Daten ziehen. Sie speichern ihre Tupel geordnet auf die Festplatte und antworten schneller auf Sequenzanfragen. Genau das erreicht SEQ, eine von Praveen Seshadri, Miron Livny und Raghu Ramakrishnan entworfene Sequenzdatenbank, durch eine deklarative Anfragesprache, die besondere Optimierungen zulässt. Für Bereichsanfragen müssen z.B. fast nur noch die relevanten Tupel gelesen werden. Neue Aggregationsfunktionen, wie Fensterfunktionen die mittlerweile im SQL:1999 Standard definiert sind, werden durch SEQ eingeführt. Sie erlauben dem Nutzer einfach zu erstellende Anfragen für den gleitenden Durchschnitt und können optimiert abgearbeitet werden.

Abbildung 9.1: Ein Candlestick-Chart für den Kurs der Google-Aktie zwischen Juni 2008 und Februar 2009. Der Aktienkurs ist ein typisches Beispiel für Sequenzdaten.

9.2 Einleitung

Um erste Missverständnisse zu beseitigen, befassen wir uns zunächst mit einem typischen Beispiel für Sequenzdaten, danach werden die Begriffe Ordnungsdomäne und Sequenz definiert. Abbildung 9.1 zeigt den Kurs der Google-Aktie über die letzten anderthalb Jahre. Ein Börsenmakler muss in der Lage sein aus dem ständigen Auf und Ab eines Aktienpreises den richtigen Trend zu erkennen. Er wird z.B. versuchen aus diesem Candlestick-Chart abzuleiten, ob die Google-Aktie im Abwärts- oder Aufwärtstrend ist. Dazu behilft er sich mit sog. Indikatoren. Die blaue Linie zeigt den 10-Tages Moving Average (gleitender Durchschnitt), die rote Linie den langsameren 30-Tages Moving Average. Beide Linien zusammen bilden die Grundlage für eine simple Form der technischen Analyse der Aktie. Wie in [58] erläutert, gilt: Kreuzt die blaue Linie die rote Linie von oben, ist das ein Signal die Aktien zu verkaufen. Kreuzt die blaue Linie die rote Linie jedoch von unten, sagt die Theorie die Aktie wird weiter an Wert gewinnen. Die Preisentwicklung nach Mitte Juni 2008 in Abbildung 9.1 zeigt allerdings, dass es keine Garantie für diese Theorie gibt. I.d.R. gibt der gleitende Durchschnitt trotzdem einen guten Anhaltspunkt für die künftige Kursentwicklung, er ist damit zusammen mit anderen Indikatoren fester Bestandteil zahlreicher Finanzapplikationen. Eine *Ordnungsdomäne* ist ein Datentyp, der eine totale Ordnung über seine Elemente definiert hat, diese heißen Positionen. Somit gibt es für jede Position einen eindeutigen Vorgänger und Nachfolger. Gängige Ordnungsdomänen sind demnach die natürlich Zahlen, aber auch Stunden, Tage und Wochen. Eine *Sequenz* ist eine Abbildung von gleichartigen Datensätzen auf Positionen einer Ordnungsdomäne. Jeder Datensatz hat eine Position, u.U. haben mehrere Datensätze dieselbe Position und leere Positionen sind ebenfalls gültig.

9.3 Beispiel einer Sequenz

Die Definitionen werden jetzt mit einem ersten Beispiel veranschaulicht. Darüber hinaus wird musterhaft eine Sequenzanfrage für den gleitenden Durchschnitt gegeben. Die Tabelle 9.1 zeigt einen Ausschnitt aus Abbildung 9.1 für 5 Tage als Relation mit dem Schema {time,

time	high	low	volume
2009/01/27	333.87	320.56	9,133,479
2009/01/28	352.33	336.31	7,680,857
2009/01/29	352.33	320.56	24,516,392
2009/01/30	348.80	336.00	4,672,843
2009/02/02	345.00	332.00	5,188,489

Tabelle 9.1: Ausschnitt aus Abbildung 9.1 für 5 Tage in Tabellenform

high, low, volume}. Die Ordnungsdomäne ist hier der Datentyp Tag. Für jeden Tag gibt es einen eindeutigen Vorgänger und Nachfolger, die Bedingung für Ordnungsdomänen ist damit erfüllt. Jeder Position (time) wird in diesem Fall genau ein Datensatz zugeordnet, der aus dem Tageshöchstwert (high), Tagestiefstwert (low) und dem Volumen der gehandelten Aktien (volume) besteht. In der Tabelle nicht zu sehen sind die leeren Positionen 31. Januar und 01. Februar 2009. An diesem Samstag und Sonntag wurden keine Aktien gehandelt. Jeder Datensatz hat eine Position, diese ist hier Schlüsselattribut. Somit erfüllt die Relation alle Kriterien einer Sequenz. Um für eine Position der Sequenz den n-Tages-Moving Average zu bestimmen kann folgende Formel angewendet werden:

$$n - Tages - GD = \sum_{i=0}^{n-1} \frac{high(today - i)}{n} \tag{9.1}$$

Ein 3-Tages Moving Average der Tageshöchstwerte für den 29. Januar 2009 könnte also wie folgt berechnet werden:

$$\frac{333.87 + 352.33 + 352.33}{3} = 346.18 \tag{9.2}$$

Diese Berechnung wird auch durch die folgende Sequenzanfrage abgefragt:

```
PROJECT avg(S.high) as avghigh
FROM    GOOG S
OVER    $P - 2 TO $P;
```

Die Sprache, in der diese Anfrage gestellt wurde, wird im Kapitel 9.4.2 vorgestellt. Im Folgenden befassen wir uns genauer mit der im Paper vorgestellten Datenbank SEQ.

9.4 Sequenzdatenbank SEQ

Die hier erläuterten Techniken beruhen auf einem 1996 im Rahmen der International Conference on Very Large Databases (VLDB) in Bombay (Indien) veröffentlichten Paper [54], das einen Grundpfeiler für die Sequenz- und Datenstrom-basierte Datenbankentwicklung legte. An der University of Wisconsin in Madison befassten sich die drei Autoren als erste mit der Frage, wie relationale Datenbanksysteme erweitert werden sollten, um auch Anfragen über geordnete Tupel zu unterstützen, anstatt über einfache Mengen oder Multimengen von Tupeln. Sie fragten sich wie SQL, die Standard-Anfragesprache, zusätzlich zu relationalen Anfragen, eine umfassende Auswahl an Sequenzanfragen unterstützen könnte. Die Antwort wurde zunächst mit der Sequenzdatenbank SEQ geliefert. Die von ihr angebotene Anfragesprache SEQUIN unterstützt ein breites Spektrum von Anfragen auf sequentiellen Daten. Die von SEQ eingesetzten Optimierungen ermöglichen zudem eine schnellere Auswertung der

Anfragen, gegenüber bisherigen Ansätzen. Das hier betrachtete Paper baut u.a. auf zwei von den selben Autoren geschriebenen Paper auf. Bereits 1994 wurden in [52] neue Anfragetypen und Optimierungen speziell für sequentielle Daten vorgestellt. 1995 wurde in [53] des Weiteren ein Modell für Sequenzen definiert, welches SEQ zugrunde liegt.

SEQ wurde als eine Komponente der ebenfalls an der University of Wisconsin entwickelten, objekt-relationalen Datenbank PREDATOR entworfen. PREDATOR unterstützt eine neuartige Verwaltung von komplexen Datentypen namens Enhanced-Abstract-Datatype (E-ADT). Diese ermöglicht u.a., dass der komplexe Datentyp Sequenz seine eigene deklarative Anfragesprache zur Verfügung stellt. Besondere Datentyp-spezifische Anfragen müssen nicht mehr prozedural aus einem SQL-Statement heraus gestellt werden. Die Details werden in [51] näher erläutert.

9.4.1 Speicherimplementierungen

Die Autoren vergleichen in ihrer Arbeit vier Varianten der Repräsentation einer Sequenz auf der Festplatte. Dateien, in der die Tupel im ASCII-Format kodiert sind, kommen in der Praxis häufig als Ursprungsformat für sequentielle Daten vor. Es bedarf jedoch einer Dekodierung, bevor die Daten angefragt werden können. Naheliegend ist die Speicherung als Datei direkt im Byte-Code. Aufwendig ist hier weiterhin das Einfügen neuer Tupel in die Mitte der Sequenz. Ein zusätzlicher Index auf die Datei würde Abhilfe leisten. Der in den Versuchen von Seshadri, Livny und Ramakrishnan benutzte Index arbeitete jedoch auf logischen Zeigern. Das kostenintensive Berechnen der physikalischen Speicherposition aus der logischen disqualifizierte jedoch auch diese Variante. Ein komprimiertes Array ermöglicht performanten Zugriff und spart zudem Speicher, indem leere Positionen keinen Platz verschwenden. Diese Art der Speicherung stellte sich, bei einem Versuch, für den Zweck einer Anfrage-lastigen Umgebung als die Vielversprechendste heraus. Näheres dazu ist im Paper nachzulesen. Bei der Betrachtung der Performanzgewinne durch mögliche Optimierungen in Kapitel 9.4.3 ist immer letztere Speicherimplementierung vorauszusetzen.

9.4.2 Deklarative Anfragesprache SEQUIN

In diesem Abschnitt befassen wir uns mit der Anfragesprache die im Paper vorgestellt wird. Das Ergebnis jeder Sequenzanfrage ist wieder eine Sequenz. Die Operatoren von SEQUIN adaptieren die aus SQL bekannten Operatoren und ergänzen sie zusätzlich. Dies ist die Syntax einer SEQUIN-Anfrage:

```
 PROJECT <project-list>
 FROM    <sequences-to-be-merged>
[WHERE   <selection-conditions>]
[OVER    <start> TO <end>]
[ZOOM    <zoom-info>];
```

Die Semantik der Operatoren ist wie folgt. Der PROJECT-Operator gleicht dem SELECT in SQL, wobei im Ergebnis immer das Ordnungsattribut steht, auch wenn es in der Anfrage ausgespart wurde. Wenn im FROM-Satz mehrere Sequenzen angegeben sind, werden diese implizit über die Ordnungsattribute zusammengeführt. WHERE ist genau wie WHERE in SQL. Mit dem OVER-Satz kann die Fenstergröße für eine Fensteranfrage, wie den gleitenden Durchschnitt, angegeben werden. Dazu wird die aktuelle Position eines Datensatzes mit der

Variablen $P referenziert. Die Aggregation im PROJECT-Satz wird dabei zur Fensterfunktion. Der ZOOM-Operator bietet die Möglichkeit zwischen vordefinierten Ordnungsdomänen zu wechseln. Wenn zu einer gröberen Granularität gewechselt wird, z.B. von Tagen zu Wochen, muss gleichzeitig im PROJECT-Satz eine Aggregation vorgenommen werden. Um über die gesamte Sequenz zu aggregieren muss ZOOM ALL angegeben werden.

9.4.3 Optimierungen

In ihrer Publikation erläutern die Autoren, warum die Verwendung einer deklarativen Anfragesprache gegenüber herkömmlichen Ansätzen, Anfragen auf komplexe Datentypen zu stellen, überlegen ist. Seshadri, Livny und Ramakrishnan zeigen die Vorteile einzelner Optimierungen anhand von Performanztests. Für jede Technik werden im Paper entsprechende Anfragen betrachtet und Graphen ausgewertet. An dieser Stelle verzichtet diese Ausarbeitung auf eine ausführliche Beschreibung dessen. Stattdessen werden nur kurz die Techniken dargestellt. Wichtig für die Vergleichbarkeit der Anfragebearbeitungen ist, dass jede Anfrage eine finale Aggregation enthält. Diese vermindert die Zeit für die Ausgabe des Ergebnisses relativ zum Rest.

Bereichs-Propagierung

Bei Bereichsanfragen über Sequenzen kann die Ordnung der Tupel auf der Festplatte in besonderem Maße ausgenutzt werden. Deutlich wird dies an folgenden Anfragen, bei der die Selektivität variabel ist: 1. Fall: Das Selektions-Fenster ist am Anfang der Sequenz.

```
PROJECT count(*)
FROM    GOOG S
WHERE   S.time < <timestamp>
ZOOM    ALL;
```

2. Fall: Das Selektions-Fenster ist am Ende der Sequenz.

```
PROJECT count(*)
FROM    GOOG S
WHERE   S.time > <timestamp>
ZOOM    ALL;
```

Die Datenbank muss im 1. Fall von Position 1 der Sequenz nur solange Tupel einlesen, wie die Selektions-Bedingung zutrifft. Wenn also der *timestamp* erreicht ist, muss kein weiterer Datensatz gelesen werden. Die potentielle Unordnung von Relationen ermöglicht diese optimierte Bearbeitung der Anfrage nicht. Ein Problem ergibt sich im 2. Fall. Wenn der Bereich mitten in der Sequenz startet, muss zunächst die erste, die Selektions-Bedingung erfüllende Position gefunden werden. Auf die von den Autoren gewählte Speicherimplementierung, ein komprimiertes Array, kann nicht index-basiert zugegriffen werden. Daher empfiehlt sich zunächst mittels binärer Suche die gewünschte Position zu finden. Die Suche kann zusätzlich gewichtet werden, ausgehend von Statistiken die von der Datenbank fortlaufend über die Sequenz gepflegt wurden.

Fensteranfragen

In diesem Abschnitt wird die optimierte Berechnung einer Fensteranfrage, wie dem gleitendem Durchschnitt, demonstriert. Die Anfrage hierfür wird mit variierender Fenstergröße und Aggregationsfunktion ausgeführt:

```
CREATE VIEW MovAggr AS
    (PROJECT <aggr_function> (S.high)
     FROM    GOOG S
     OVER    $P - <window_size> TO $P);
PROJECT count(*)
FROM    MovAggr
ZOOM    ALL;
```

Seshadri, Livny und Ramakrishnan haben festgestellt, dass manche Aggregationsfunktionen bei Fensteranfragen gut optimierbar sind. Die von ihnen *Symmetric Property* genannte Eigenschaft trifft z.B. auf AVG, COUNT, SUM und PRODUCT zu. Das folgende Beispiel soll die Optimierung erläutern: Für eine Sequenz mit den Positionen von 1 bis 6 und den Werten $1 \rightarrow 5$, $2 \rightarrow 10$, $3 \rightarrow 7$, $4 \rightarrow 2$, $5 \rightarrow 12$, $6 \rightarrow 21$ soll der gleitende Durchschnitt mit der Fenstergröße 4 berechnet werden.

$$(5+10+7+2):4 = 24:4=6 \tag{9.3}$$
$$(10+7+2+12):4 = 31:4=7.75 \tag{9.4}$$
$$(24-5+12):4 = 31:4=7.75 \tag{9.5}$$
$$(31-10+21):4 = 42:4=10.5 \tag{9.6}$$

Berechnet wird ein Durchschnittswert für Position 4,5 und 6. Während in 1.3 und 1.4 der gleitende Durchschnitt durch den Quotient der immer neu berechneten Summe erschlossen wird, ist in 1.5 und 1.6 die Summe der vorhergehenden Berechnungen genutzt worden. Indem nur der Wert, der aus dem Fenster rausfallenden Position, subtrahiert wird und der Wert, der ins Fenster reinkommenden Position, addiert wird, können zahlreiche Additionen gespart werden. Die Berechnung ist damit nahezu unabhängig von der angegebenen Fenstergröße. Die Laufzeit des gleitenden Durchschnitts für ein größeres Fenster nimmt minimal zu. Verglichen zur Berechnung des Minimums, kann von der Unabhängigkeit von der Fenstergröße gesprochen werden.

Gemeinsame Unterabfragen

Sequenzanfragen, die eine Sequenz mehrfach nutzen, können ebenfalls optimiert abgearbeitet werden. Das Ergebnis auf folgende Anfrage enthält Tage, an denen sich der Wochen-Moving Average des Aktienpreis nur gering gegenüber dem Vortag geändert hat. Der Offset-Operator verschiebt dabei eine Sequenz um die angegebene Anzahl von Postionen.

```
CREATE VIEW MovAvg AS
   (PROJECT avg(S.high) as avghigh
    FROM    GOOG S
    OVER    $P - 6 TO $P);
PROJECT *
FROM    MovAvg T1, Offset(MovAvg, 1) T2
WHERE   T1.avghigh - T2.avghigh < 10;
```

SEQ würde die gemeinsame Unterabfrage erkennen und nur einmal die Sequenz GOOG einlesen. Die von den Autoren *Stream Access* genannte Methode lässt die Datenbank den mehrfach gebrauchten Teil der Sequenz im Cache halten. Dazu ist bekannt, wieviel Speicher jeder Operator maximal benötigt. Für die gezeigte Anfrage müsste nur ein Puffer für die Datensätze von acht Positionen bereitgestellt werden; sieben Positionen aufgrund der Fenstergröße und eine zusätzliche aufgrund der Verschiebung durch den Offset-Operator.

Befehlsverknüpfung

Bei einer Aggregation über die gesamte Sequenz ist das direkte Weiterreichen von Zwischenergebnissen von Vorteil. Es gibt zwei Möglichkeiten der Abarbeitung, entweder wird zwischengespeichert (auch materialized) oder durchgereicht (auch pipelined). Also entweder werden die Zwischenergebnisse während der Abarbeitung auf die Festplatte geschrieben und hinterher wieder eingelesen, oder die einzelnen Befehle werden verknüpft, sodass keine zusätzlichen IO nötig sind. Die folgende Anfrage zählt die Datensätze in der Sequenz GOOG:

```
PROJECT count(*)
FROM    GOOG
ZOOM    ALL;
```

Die im Paper dargestellten Versuchsergebnisse zeigen, dass Sequenzanfragen generell durch eine pipelined Abarbeitung schneller beantwortet werden können.

9.5 Evaluation

Ein sehr entscheidender Punkt für die *Main Contribution* des Paper ist die höhere Effizienz der Anfragebearbeitung durch SEQ mit SEQUIN im Vergleich. In Kapitel 1.1 heißt es: MM-any temporal queries can be expressed in SQL-92 using features like correlated subqueries and aggregation, these are typically very inefficient to execute.Ïn Kapitel 3.4 werden mehrere Optimierungen mit Anfragen in der neuen Sprache SEQUIN vorgestellt, keine Anfrage wird jedoch auch als herkömmliche SQL-Anfrage formuliert. Die Versuchsergebnisse zeigen dementsprechend auch keinen Vergleich zwischen der Laufzeit einer relationalen Datenbank und einer Sequenzdatenbank. Gezeigt wird nur, wielange die Sequenzanfrage von SEQ ohne und wielange mit Optimierung abgearbeitet wird.

Zu kritisieren ist weiterhin, dass die vorgeschlagene Sequenzanfragesprache SEQUIN zwar neue Möglichkeiten bietet, wie z.B. Fensteranfragen, jedoch umständlich verfremdet wurde gegenüber SQL. Implizite Joins und PROJECT statt SELECT verwirren denjenigen, der bislang mit SQL gearbeitet hat. Der ZOOM-Operator, der ähnlich wie der GROUP BY-Operator in SQL die Aggregation steuert, ist zwar intuitiv benannt worden. Die Metaphor des Herein- bzw. Heraus-Zoomens mit einer Linse ist leicht nachvollziehbar. Jedoch muss für eine Aggregation über die gesamte Sequenz, ZOOM ALL angegeben werden. In SQL kommt die Aggregation über eine gesamte Spalte der Relation ohne GROUP BY aus. Des Weiteren ist nicht ersichtlich, warum die Variable $P umständlich für die aktuelle Position eines Datensatzes benutzt wird, statt direkt das Ordnungsattribut der Sequenz anzugeben. Eine Sequenzanfragesprache, die SQL nur um die nötigsten Operatoren für u.a. Fensteranfragen (OVER) und Granularitätswechsel (ZOOM) erweitert, hätte mehrere Vorteile. Bestehende SQL-Anfrage-Parser müssten nur erweitert und nicht umgeschrieben werden, um Sequenzanfragen zu unterstützen. Anwender der Sprache, die SQL bereits kennen, müssten nicht umdenken.

9.6 Test of Time

Die Erkenntnisse von Seshadri, Livny und Ramakrishnan wurden in den Folgejahren mehrfach adaptiert. Im Jahr 1998 veröffentlichte zunächst wieder Ramakrishnan mit Anderen ein Paper, dass Sequenzanfragen näher an SQL formuliert [47]. Die Sorted Relational Query Language (SRQL) erweitert SQL nur um das Nötigste. Folgende Anfrage verlangt das Wochen-Moving Average für den Tageshöchstwert der Google-Aktie.

```
SELECT      time, avg(high) OVER 0 TO 6 as avghigh
FROM        GOOG
SEQUENCE BY time;
```

Im Gegensatz zu SEQUIN, steht bei SRQL das Fenster einer Fensteranfrage direkt im SELECT-Satz. Die Variable $P für die aktuelle Position des Datensatzes wird einfach durch 0 angegeben. Eine Sequenzanfrage ist hier durch den Operator SEQUENCE BY gekennzeichnet, nicht mit PROJECT statt SELECT für die Projektion der Anfrage.

Beachtlich ist das SQL ab dem Standard SQL:1999 ebenfalls Sequenzanfragen, wie den gleitenden Durchschnitt, unterstützt. Das Fenster wird wie bei SRQL im SELECT-Satz angegeben, wobei die Größe durch ROWS PRECEDING oder FOLLOWING angegeben wird. Die folgende Anfrage könnte einer aktuellen Datenbanken von Oracle oder IBM gestellt werden.

```
SELECT time, avg(high) OVER (ORDER BY time ROWS 6 PRECEDING) as avghigh
FROM   GOOG;
```

Die Veröffentlichung eines Papers in 2004 [49], das SQL-TS, eine Anfragesprache zur Suche von komplexen, sich wiederholenden Mustern in Sequenzen, vorstellt, weist darauf hin, dass das Thema Sequenzdatenbanken heute noch aktuell ist.

9.7 Zusammenfassung

In dieser Ausarbeitung wurde zunächst die Thematik Sequenzdaten eingeführt und mit dem Beispiel eines Aktienkurses motiviert. Hiernach wurde, aufbauend auf [54], eine konkrete Sequenzdatenbank mit dazugehöriger deklarativer Sequenzanfragesprache beleuchtet und die durch sie möglichen Optimierungen an Beispielanfragen erläutert. Schließlich wurden Probleme des Papers und speziell der Anfragesprache aufgedeckt, sowie die daran anknüpfenden neueren Ansätze aus [47] und dem SQL:1999 Standard beschrieben.

Literaturverzeichnis

[1] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 94–105. ACM Press, 1998. 67, 68

[2] Jesse Alpert and Nissan Hajaj. We knew the web was big... http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html, 2008. 50

[3] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003. 53

[4] John Battelle. The birth of google. http://www.wired.com/wired/archive/13.08/battelle.html?tw=wn_tophead_4, 2005. 50

[5] Google Press Center. Google goes global with addition of 10 languages. http://www.google.com/press/pressrel/pressrelease22.html, 2000. 53

[6] Google Press Center. Google launches self-service advertising program. http://www.google.com/press/pressrel/pressrelease39.html, 2000. 53

[7] Google Press Center. Google launches world's largest search engine. http://www.google.com/press/pressrel/pressrelease26.html, 2000. 53

[8] E. F. Codd. Relational database: A practical foundation for productivity. *Commun. ACM*, 25(2):109–117, 1982. 35

[9] Matt Cutts. A quick word about googlebombs. http://googlewebmastercentral.blogspot.com/2007/01/quick-word-about-googlebombs.html, 2007. 54

[10] Matt Cutts. Detecting new 'googlebombs'. http://googlepublicpolicy.blogspot.com/2009/01/detecting-new-googlebombs.html, 2009. 54

[11] Stephan Dörner. Wie unternehmen bei google glänzen. http://www.handelsblatt.com/technologie/it-internet/wie-unternehmen-bei-google-glaenzen;1341018, 2007. 53

[12] The Economist. Enlightenment man. http://www.economist.com/science/tq/displaystory.cfm?story_id=12673407, 2008. 50

[13] Venkatesh Ganti, Raghu Ramakrishnan, Johannes Gehrke, and Allison Powell. Clustering large datasets in arbitrary metric spaces. In *ICDE '99: Proceedings of the 15th International Conference on Data Engineering*, page 502. IEEE Computer Society, 1999. 67

[14] Golem.de. Google geht an die börse. http://www.golem.de/0404/31054.html, 2004. 53

[15] Google. Digital millennium copyright act (dmca). http://www.google.com/dmca.html, 2004. 53

[16] Google. Corporate information – our philosophy. http://www.google.com/corporate/tenthings.html, 2009. 50

[17] Google. Search engine optimization (seo). http://www.google.com/support/webmasters/bin/answer.py?hl=en&answer=35291, 2009. 54

[18] Google. Webmaster guidelines. http://www.google.com/support/webmasters/bin/answer.py?hl=en&answer=35769, 2009. 54

[19] Goetz Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. In *DaMoN '07: Proceedings of the 3rd international workshop on Data management on new hardware*, pages 1–9, New York, NY, USA, 2007. ACM. 31, 32

[20] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1:29, 1997. 57

[21] Jim Gray and Goetz Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Rec.*, 26(4):63–68, 1997. 32

[22] Jim Gray and Franco Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for cpu time. *SIGMOD Rec.*, 16(3):395–398, 1987. 29, 30, 31, 33

[23] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. Cure: an efficient clustering algorithm for large databases. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 73–84, New York, NY, USA, 1998. ACM. 67

[24] Maria Halkidi, Yannis Batistakis, and Michalis Vazirgiannis. On clustering validation techniques. *J. Intell. Inf. Syst.*, 17(2-3):107–145, 2001. 67

[25] Theo Härder. Observations on optimistic concurrency control schemes. *Inf. Syst.*, 9(2):111–120, 1984. 36

[26] heise online. Google überrascht bei der suche nach erbärmlichen versagern. http://www.heise.de/newsticker/Google-ueberrascht-bei-der-Suche-nach-erbaermlichen-Versagern--/meldung/42679, 2003. 54

[27] heise online. Google in deutschland über 90 prozent. http://www.heise.de/newsticker/Google-in-Deutschland-ueber-90-Prozent--/meldung/78315, 2006. 54

[28] heise online. Google zensiert seine neue chinesische suchmaschine. http://www.heise.de/newsticker/Google-zensiert-seine-neue-chinesische-Suchmaschine-Update--/meldung/68792, 2006. 53

[29] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21st VLDB Conference*, 1995. 43

[30] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999. 68

[31] Stephen Johnson. Hierarchical clustering schemes. *Psychometrika*, 32(3):241–254, 1967. 65

[32] Masaru Kitsuregawa, Masaya Nakayama, and Mikio Takagi. The effect of bucket size tuning in the dynamic hybrid grace hash join method. In Peter M. G. Apers and Gio Wiederhold, editors, *VLDB*, pages 257–266. Morgan Kaufmann, 1989. 24

[33] Erica Kolatch. Clustering algorithms for spatial databases: A survey, 2001. 68

[34] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981. 35, 36, 37, 39

[35] Amy N. Langville and Carl D. Meyer. *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, Princeton, NJ, USA, 2006. 51

[36] Erendira Rendón Lara and R. Barandela. Scaling clustering algorithm for data with categorical attributes. In *ICCOMP'05: Proceedings of the 9th WSEAS International Conference on Computers*, pages 1–6, Stevens Point, Wisconsin, USA, 2005. World Scientific and Engineering Academy and Society (WSEAS). 67

[37] Hongjun Lu, Kian-Lee Tan, and Ming-Chien Shan. Hash-based join algorithms for multiprocessor computers. In McLeod et al. [40], pages 198–209. 24

[38] Marissa Mayer. Googlebombing 'failure'. http://googleblog.blogspot.com/2005/09/googlebombing-failure.html, 2005. 54

[39] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *KDD '00: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 169–178, New York, NY, USA, 2000. ACM. 68

[40] Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors. *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*. Morgan Kaufmann, 1990. 85, 87

[41] C. Mohan. Less optimism about optimistic concurrency control. In *Second International Workshop on Research Issues on Data Engineering, 1992: Transaction and Query Processing*, pages 199–204, 1992. 39

[42] Masaya Nakayama, Masaru Kitsuregawa, and Mikio Takagi. Hash-partitioned join method using dynamic destaging strategy. In François Bancilhon and David J. DeWitt, editors, *VLDB*, pages 468–478. Morgan Kaufmann, 1988. 24

[43] Raymond T. Ng and Jiawei Han. Clarans: A method for clustering objects for spatial data mining. *IEEE Trans. on Knowl. and Data Eng.*, 14(5):1003–1016, 2002. 68

[44] Patrick E. O'Neil. The sb-tree: An index-sequential structure for high-performance sequential access. *Acta Informatica*, 29(3):241–265, 1992. 32

[45] HweeHwa Pang, Michael J. Carey, and Miron Livny. Partially preemptive hash joins. In Peter Buneman and Sushil Jajodia, editors, *SIGMOD Conference*, pages 59–68. ACM Press, 1993. 24

[46] Cecilia M. Procopiuc, Michael Jones, Pankaj K. Agarwal, and T. M. Murali. A monte carlo algorithm for fast projective clustering. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 418–427, New York, NY, USA, 2002. ACM. 68

[47] Raghu Ramakrishnan, Donko Donjerkovic, Arvind Ranganathan, Kevin S. Beyer, and Muralidhar Krishnaprasad. Srql: Sorted relational query language. In *SSDBM '98: Proceedings of the 10th International Conference on Scientific and Statistical Database Management*, pages 84–95, Washington, DC, USA, 1998. IEEE Computer Society. 82

[48] James P. Richardson, Hongjun Lu, and Krishna P. Mikkilineni. Design and evaluation of parallel pipelined join algorithms. In Umeshwar Dayal and Irving L. Traiger, editors, *SIGMOD Conference*, pages 399–409. ACM Press, 1987. 24

[49] Reza Sadri, Carlo Zaniolo, Amir Zarkesh, and Jafar Adibi. Expressing and optimizing sequence queries in database systems. *ACM Trans. Database Syst.*, 29(2):282–318, 2004. 82

[50] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34, New York, NY, USA, 1979. ACM. 9, 10

[51] Praveen Seshadri. *Management of sequence data*. PhD thesis, 1996. Supervisor-Ramakrishnan, Raghu. 78

[52] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Sequence query processing. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 430–441, New York, NY, USA, 1994. ACM. 78

[53] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Seq: A model for sequence databases. In *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*, pages 232–239, Washington, DC, USA, 1995. IEEE Computer Society. 78

[54] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The design and implementation of a sequence database system. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 99–110, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc. 77, 82

[55] Leonard D. Shapiro. Join processing in database systems with large main memories. *ACM Trans. Database Syst.*, 11(3):239–264, 1986. 15

[56] Gholamhosein Sheikholeslami, Surojit Chatterjee, and Aidong Zhang. Wavecluster: A multi-resolution clustering approach for very large spatial databases. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 428–439, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc. 68

[57] Tom Spring. Search engines gang up on microsoft. `http://www.cnn.com/TECH/computing/9911/15/search.engine.ms.idg/`, 1999. 54

[58] Inc. StockCharts.com. Moving averages, February 2009. `http://stockcharts.com/school/doku.php?id=chart_school:technical_indicators:moving_averages`. 76

[59] Danny Sullivan. Searches per day. `http://searchenginewatch.com/2156461`, 2006. 50

[60] Wei Wang, Jiong Yang, and Richard R. Muntz. Sting: A statistical information grid approach to spatial data mining. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 186–195, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc. 67

[61] Jinxi Xu and W. Bruce Croft. Corpus-based stemming using cooccurrence of word variants. *ACM Trans. Inf. Syst.*, 16(1):61–81, 1998. 52

[62] Rui Xu and D. Wunsch. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678, 2005. 68

[63] Hansjörg Zeller and Jim Gray. An adaptive hash join algorithm for multiuser environments. In McLeod et al. [40], pages 186–197. 24

[64] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: An efficient data clustering method for very large databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 103–114. ACM Press, 1996. 63

[65] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: A new data clustering algorithm and its applications. *Data Mining and Knowledge Discovery*, 1(2):141–182, 1997. 66

[66] X. Zhao, Roger G. Johnson, and Nigel J. Martin. Dbj - a dynamic balancing hash join algorithm in multiprocessor database systems (extented abstract). In Matthias Jarke, Janis A. Bubenko Jr., and Keith G. Jeffery, editors, *EDBT*, volume 779 of *Lecture Notes in Computer Science*, pages 301–308. Springer, 1994. 25

Abbildungsverzeichnis

Tabellenverzeichnis