Philipp Breitbach Matrikelnummer: 344712

Inhaltsverzeichnis

Kapitel 1: Einleitung

In den letzten Jahren nahm die Bedeutung von *Online-Analytical-Processing*(**OLAP**)-Anwendungen stetig zu. OLAP wird zur Analyse sehr großer Datenmengen genutzt, also typischerweise zur Entscheidungsunterstützung in großen Unternehmen. Dabei werden eine *multidimensionale Datensicht*, eine *interaktive Anfrageverarbeitung* und eine *Aggregatbildung entlang der Dimensionen* unterstützt. Für nähere Informationen über die Konzepte, die OLAP zugrunde liegen, wird hier auf [BG01] verwiesen.

Interaktivität bedeutet, dass Anfragen im Bereich von wenigen Sekunden verarbeitet werden. Bedenkt man, dass OLAP-Anfragen in der Regel auf enorme Datenmengen zugreifen, so wird die sehr hohe Leistungsanforderung deutlich, die durch die Interaktivität gestellt wird.

Eine *multidimensionale Datensicht* projiziert die zu analysierenden Daten auf einen so genannten **Datenwürfel**[1], der durch Dimensionen aufgespannt wird. Eine *Zelle* des Datenwürfels wird durch die Werte der *Dimensionsgrößen* adressiert. Diese Werte liegen in einer bestimmten *Granularität* vor. Eine typische Dimensionsgröße ist beispielsweise die Zeit mit möglichen Granularitäten Tag, Monat und Jahr. Dagegen stellen die sogenannten *Messgrößen* den tatsächlichen Inhalt einer Zelle des Würfels dar und sind die für die Analyse relevantesten Daten. Ein Beispiel für eine Messgröße ist der Umsatz eines bestimmten Zeitraums.

Die *Aggregatbildung* entlang einiger Dimensionen (z.B. „Summe der Umsätze pro Jahr und Produkt“) ist die dominierende Anfrageform, weil sehr große Datenmengen in einer übersichtlichen Form dargestellt werden müssen. Um den Leistungsanforderungen an OLAP-Anwendungen gerecht zu werden, müssen solche Aggregate häufig *vorberechnet* werden.

Die Anfrageevaluierung und Optimierung wird hier in diesem Zusammenhang für die Berechnung eines Datenwürfels sowie für die Berechnung von Bereichsanfragen geschildert.

Die Gliederung dieser Ausarbeitung gestaltet sich wie folgt:

Kapitel 2 beschäftigt sich mit der Berechnung des Datenwürfels. Dazu wird der **CUBE**-Operator, der einen Datenwürfel berechnet und bestimmte vorberechnete Aggregate in diesen integriert, als OLAP-Schnittstelle vorgestellt. Weiterhin wird die interne Darstellung dieses

[1] Eigentlich wird ein Datenquader gebildet, der Begriff des Datenwürfels hat sich jedoch in der Literatur eingebürgert.

Datenwürfels als Suchgitter erläutert und ein effizienter Algorithmus zur Berechnung beschrieben.

In Kapitel 3 wird die Verarbeitung von Bereichsanfragen, die hier als Spezialisierung einer Aggregationsanfrage betrachtet werden, behandelt. Es werden die Eigenschaften von Bereichsanfragen dargelegt sowie ein schneller Algorithmus zur Berechnung vorgestellt.

In Kapitel 4 schließlich erfolgt eine Zusammenfassung, welche die Ergebnisse dieser Ausarbeitung darstellt.

Kapitel 2: Berechnung des Datenwürfels

Die Bildung von Aggregaten entlang mehrerer Dimensionen ist **die** charakteristische Anfrageform im OLAP-Bereich. Sie resultiert im Allgemeinen in der Bildung eines *Datenwürfels*, der von den Dimensionen aufgespannt wird. Betrachtet man die Anfrage „Summe der Umsätze pro Produkt, Jahr und Kunde“, so wird eine *Zelle* des Datenwürfels, welche die Summe der Umsätze enthält, durch die einzelnen Dimensionswerte adressiert, also z.B. durch die Koordinaten (Waschmaschine, 2003, Herr Müller). Aus der Analysesicht ist es weiterhin sinnvoll, auch die Aggregatsbildung über Teilmengen der angegeben Dimensionen in den Datenwürfel zu integrieren, also z.B. die „Summe der Umsätze pro Produkt und Jahr“ oder die „Gesamtsumme der Verkäufe“. Zur Bildung eines solchen Datenwürfels wurde der CUBE-Operator entwickelt, der in Abschnitt 2.1 beschrieben wird. Abschnitt 2.2 befasst sich einem speziellen Graphen, dem *Suchgitter*, welches in Abschnitt 2.3 als Grundlage des *PipeSort*-Algorithmus zur Berechnung des Datenwürfels dient.

2.1 Der CUBE-Operator:

Der CUBE-Operator ist die Schnittstelle für OLAP-Anwendungen zur Erzeugung eines Datenwürfels. Es ist ein deskriptiver Operator, d.h. man *beschreibt* mit seiner Hilfe den gewünschten Datenwürfel. Prinzipiell ist er eine n-dimensionale Verallgemeinerung des Group-By-Operators. Er gruppiert nicht nur über die Menge der Gruppierungsattribute, sondern auch über all deren Teilmengen.

Um das Ergebnis des CUBE-Operators zu definieren wird zunächst eine abkürzende Notation für Group-by-Anfragen definiert:

Select $D_1, D_2, \ldots, D_d$, Sum(M)
From Tabelle $\Leftrightarrow$ $(D_1, D_2, \ldots, D_d)$
Group-by $D_1, D_2, \ldots, D_d$

Dabei sind $D_1, D_2, \ldots, D_d$ die Gruppierungsattribute und M ist dasjenige Attribut, über welches das Aggregat gebildet wird. M wird in der abkürzenden Schreibweise nicht angegeben, da es für die weiteren Betrachtungen uninteressant ist, über welchem Attribut Aggregate gebildet werden.

Der CUBE-Operator hat folgende Aufrufsemantik:

Select $D_1, D_2, \ldots, D_d$, Sum(M)
From Tabelle
Cube-by $D_1, D_2, \ldots, D_d$

Das Ergebnis seines Aufrufs ist: $\Upsilon(T)$.
$\forall T \subseteq \{D_1, D_2, \ldots, D_d\}$

Beispiel 1: Ein Einzelhandelsunternehmen verwaltet seine Umsatzzahlen in einer Tabelle Umsätze(Produkt, Jahr, Kunde, Umsatz). Auf diese Tabelle kann der CUBE-Operator wie folgt angewendet werden:

Select Produkt, Jahr, Kunde, Sum(Umsatz)
From Verkäufe
Cube-by Produkt, Jahr, Kunde

Diese Anfrage besteht aus der Berechnung von $2^3 = 8$ Gruppierungen: (Produkt, Jahr, Kunde), (Produkt, Kunde), (Produkt, Jahr), (Kunde, Jahr), (Produkt), (Jahr), (Kunde) und (), wobei die Gruppierung () die leere Gruppierung (Gesamtsumme aller Umsatzzahlen) ist. Das Ergebnis des CUBE-Operators ist die Vereinigung der einzelnen Gruppierungsergebnisse:

(Produkt, Jahr, Kunde) **UNION** (Produkt, Kunde) **UNION** (Produkt, Jahr) **UNION** (Kunde, Jahr) **UNION** (Produkt) **UNION** (Jahr) **UNION** (Kunde) **UNION** ().

2.2 Suchgitter für Datenwürfel:

Der gerade Weg eine Anfrage wie in Beispiel 1 auszuwerten ist jede Gruppierung isoliert zu berechnen und die Vereinigung der Einzelergebnisse zu bilden. Das ist jedoch bei großen Datenmengen eine sehr langsame Methode. Es sind verschieden Optimierungen nötig, um die Interaktivität zu gewährleisten. Man kann bei jedem Datenwürfel Verwandtschaften der Gruppierungen berücksichtigen. So kann beispielsweise die Gruppierung (Produkt) aus der Gruppierung (Produkt, Kunde) berechnet werden. Diese Verwandtschaft lässt sich allgemein durch ein sogenanntes *Suchgitter* beschreiben:

Definition: Seien D_1, …, D_d die Gruppierungsattribute eines Datenwürfels. Ein **Suchgitter** ist ein gerichteter Graph, dessen Knoten eine Gruppierung des Würfels repräsentieren. Zwei Gruppierungen i und j werden durch eine gerichtete Kante e_{ij} verbunden, falls j aus i berechnet werden kann und falls j genau ein Attribut weniger als i hat. Dabei kann j aus i berechnet werden, falls die Gruppierungsattribute von j eine Teilmenge der Gruppierungsattribute von i sind. Es können alle Gruppierungen mit k Attributen in einer *Ebene* k zusammengefasst werden. Bild 1 gibt ein Beispiel eines Suchgitters.

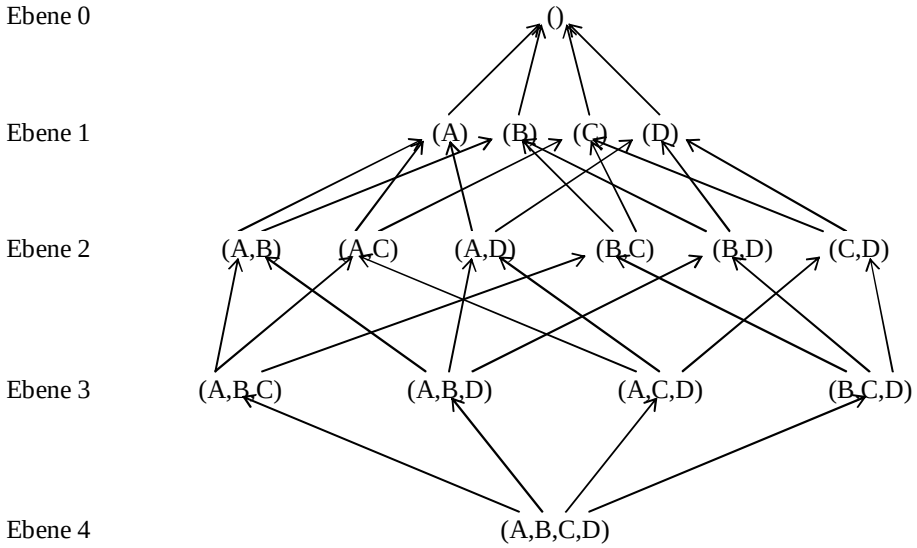


Bild 1: Suchgitter für einen Würfel mit vier Dimensionen

Dieses Suchgitter lässt sich verwenden, um den Datenwürfel effizienter zu berechnen, denn jeder Knoten kann aus seinem Vater berechnet werden[2], anstatt aus der (größeren) Basistabelle. Falls ein Knoten A mehrere Väter hat, muss einer ausgewählt werden, aus dessen Daten A berechnet werden soll. Diese alternativen Berechnungsmöglichkeiten können sehr unterschiedliche Qualitäten in Bezug auf die Berechnungskosten aufweisen. Ein Algorithmus zur Berechnung des Datenwürfels sollte also den kostengünstigsten Plan, der angibt welche Gruppierungen wie berechnet werden sollen, zur Berechnung heranziehen.

Zur Berechnung *einer* Gruppierung gibt es zwei grundsätzliche Ansätze: den Sortierungsbasierten sowie den Hashbasierten Ansatz. Der folgende Abschnitt 2.3 wird den Sortierungsbasierten *PipeSort*-Algorithmus vorstellen.

2.3 *PipeSort*-Algorithmus:

Der PipeSort-Algorithmus verwendet, wie der Name schon sagt, ein sortierungsbasiertes Verfahren zur Berechnung der einzelnen Gruppierungen. Er zieht seinen Vorteil daraus, dass Gruppierungen, deren Sortierungsreihenfolge ein Präfix der Sortierungsreihenfolge eines Vaters ist, ohne zusätzlichen Sortierungsaufwand berechnet werden können. Sei beispielsweise die Basisrelation in der Attributreihenfolge (A,B,C,D) sortiert. Dann können die Gruppierungen (A,B,C,D), (A,B,C), (A,B), (A) und () ohne zusätzliche Sortierungskosten berechnet werden. Man sagt, die einzelnen Gruppierungen teilen sich die Kosten für das Sortieren und nennt diese Optimierungsstrategie deswegen „*share-sorts*“. Außerdem verwendet PipeSort die Optimierungsstrategie „*smallest-parent*“, der zufolge zur Berechnung einer Gruppierung der kleinste Vater herangezogen wird. Es ist offensichtlich, dass diese beiden Strategien gegensätzlich sein können: Angenommen der kleinste Vater von (A,B) ist (B,D,A), wohingegen (A,B) nach dem Share-Sort-Prinzip aus (A,B,C) berechnet werden sollte. Zudem werden die Optimierungen „*cache-results*“ und „*amortize-scans*“ in folgender Weise eingegliedert:

Man betrachte eine Basisrelation, die wieder in der Reihenfolge (A,B,C,D) sortiert ist und aus der die Gruppierungen (A,B,C,D), (A,B,C), (A,B) und (A) berechnet werden sollen. Anstatt jede dieser Gruppierungen getrennt zu berechnen, kann man sie alle in einer Art Pipeline wie folgt berechnen: Jedes mal wenn ein Tupel von (A,B,C,D) berechnet wird, wird es in der Pipeline weitergegeben um (A,B,C) zu berechnen; Jedes mal wenn ein Tupel von (A,B,C) berechnet wird,

[2] Die Wurzel muss selbstverständlich aus der Basistabelle berechnet werden, weil sie keinen Vater hat.

wird es in der Pipeline weitergegeben um (A,B) zu berechnen, usw. Also ist jede Pipeline eine Liste von Gruppierungen, die alle in einem Durchlauf berechnet werden, wobei während der Berechnung dieser Gruppierungen für jede Gruppierung nur ein Tupel gespeichert werden muss.

Im Folgenden wird der PipeSort Algorithmus detailliert beschrieben:

PipeSort:

Der Algorithmus erhält als Eingabeparameter das Suchgitter des Datenwürfels. Zunächst werden nun für jede Kante e_{ij} im Gitter zwei Kosten $S(e_{ij})$ und $A(e_{ij})$ hinzugefügt[3]. $A(e_{ij})$ sind die Kosten, um j aus i zu berechnen, falls i schon in der richtigen Reihenfolge sortiert ist. Konträr dazu beschreibt $S(e_{ij})$ die Kosten j aus i zu berechnen, falls i nicht in der richtigen Reihenfolge sortiert ist.

Der Algorithmus iteriert nun beginnend von Ebene k=0 des Suchgitters bis Ebene k=N-1 des Suchgitters, wobei N die Anzahl der Dimensionen des Würfels darstellt. Er findet für jede Ebene k den kostengünstigsten Weg Ebene k aus Ebene k+1 zu berechnen, indem er das Problem auf ein *weighted bipartite matching problem* (WBMP) reduziert.

Das WBMP ist folgendermaßen definiert: Gegeben ist ein Graph mit zwei disjunkten Knotenmengen V_1 und V_2 und eine Menge von Kanten E, die Knoten in V_1 mit Knoten in V_2 verbindet. Jeder Kante ist ein festes Gewicht zugeordnet. Das WBMP wählt die Teilmenge von Kanten aus E mit einem maximalen Gewicht aus, so dass in dem ausgewählten Untergraph jeder Knoten in V_1 mit höchstens einem Knoten aus V_2 verbunden ist und umgekehrt.

Es wird in jeder Iteration Ebene k+1 dahingehend verändert, dass für jede Gruppierung dieser Ebene k zusätzliche Kopien gemacht werden. Damit gibt es für jede Gruppierung in Ebene k+1 nun k+1 Knoten, so wie jede Gruppierung in Ebene k+1 auch k+1 Kinder hat. Jeder replizierte Knoten wird mit derselben Menge von Knoten verbunden wie sein Original. Die Kosten einer Kante e_{ij} von dem Originalknoten werden auf $A(e_{ij})$ gesetzt, wohingegen allen replizierten Knoten Kantenkosten $S(e_{ij})$ zugeordnet werden. Auf den so erzeugten zweigeteilten Graphen wird nun der Algorithmus aus [PS82] für das WBMP angewendet[4]. In dessen Ausgabe ist jeder Knoten A aus Ebene k mit genau einem Knoten B aus Ebene k+1 verbunden. Falls A und B durch eine A-Kante verbunden sind, so bestimmt A die Attributreihenfolge, in welcher B während seiner Berechnung sortiert wird. Andernfalls wird B umsortiert um A zu berechnen.

Es folgt eine Darstellung des Verfahrens in Pseudocode [SAG+96]:

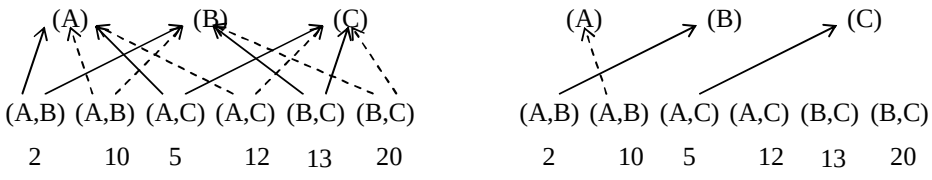
[3] Diese Kosten werden als gegeben vorrausgesetzt.
[4] Hierbei wird allerdings nach dem Teilgraphen mit minimalen Kosten gesucht.

(Input: Suchgitter mit A- und S-Kantenkosten)
FOR Ebene k=0 TO Ebene k=N-1
Generate-Plan(k+1 → k);
FOREACH Gruppierung g in Ebene k+1
IF eine A-Kante e_{gh} zu einem Knoten h aus Ebene k existiert
THEN setze die Attributreihenfolge von g auf die von h;

Generate-Plan(k+1 → k)
Lege k zusätzliche Kopien für jeden Ebene-k+1-Knoten an;
Verbinde jeden kopierten Knoten mit denselben Knoten ,
mit denen das Original verbunden ist;
Ordne der Kante e_{ij} Kosten $A(e_{ij})$ für das Original und $S(e_{ij})$ für die Kopien zu;
Finde die minimale Kosten mittels Algorithmus für WBMP;

Nachdem so ein optimaler Plan zur Berechnung des Würfels gefunden worden ist können nun nacheinander die einzelnen Pipelines in der oben beschriebenen Art ausgeführt werden, wodurch der Datenwürfel sukzessive aufgebaut wird.

Zur Verdeutlichung von Generate-Plan(k+1 → k) wird in Beispiel 2 erläutert, wie ein Plan zur Berechnung von Ebene-1-Gruppierungen aus den Ebene-2-Gruppierungen für ein Suchgitter mit drei Gruppierungsattributen erstellt wird:



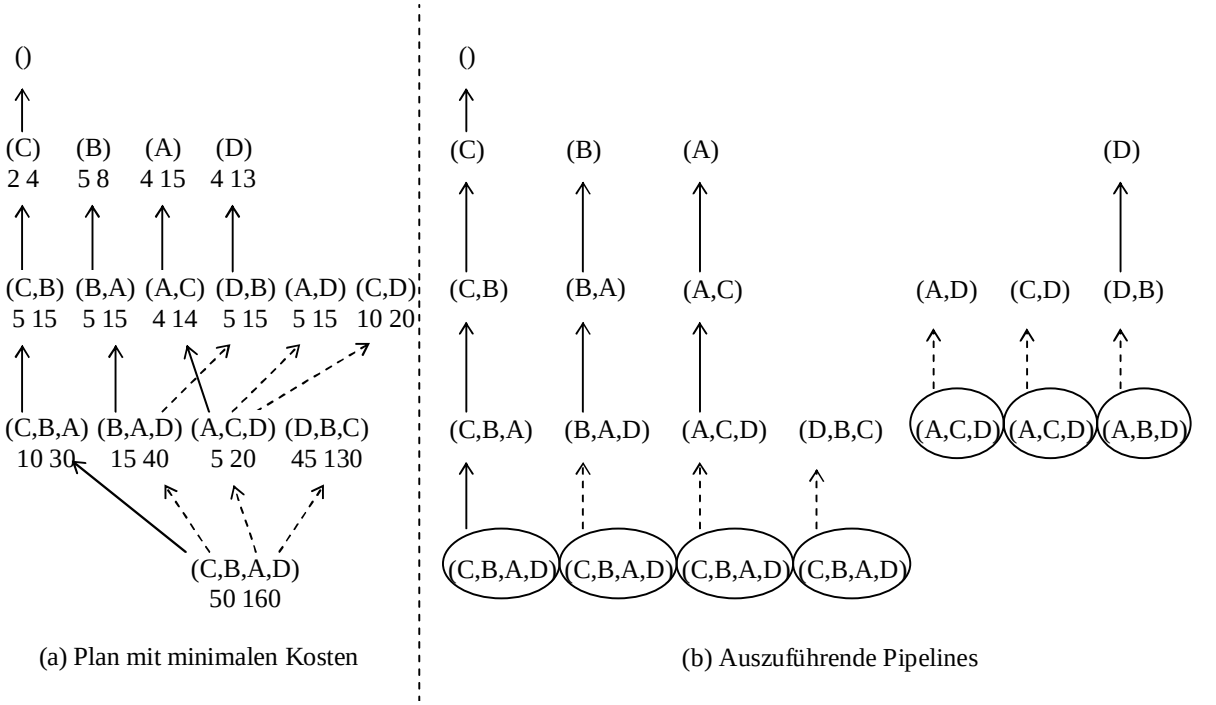
(a) Geändertes Suchgitter (b) Plan für minimale Kosten

Bild 2: Berechnen von Ebene-1-Gruppierungen aus Ebene-2-Gruppierungen in einem Würfel mit drei Dimensionen.

Beispiel 2: Wie Bild 2(a) aus [SAG+96] zeigt, wird zunächst für jede Gruppierung in Ebene 2 eine zusätzliche Kopie hinzugefügt. Die gestrichelten Kanten deuten auf S()-Kosten hin, während die durchgezogenen Kanten A-Kosten indizieren. Die Zahl

unter jedem Ebene-2-Knoten repräsentiert die Kosten für jede einzelne ausgehende Kante. In dem Rückgabegraph des Algorithmus für das WBMP (Bild 2(b)), der den Berechnungsplan mit minimalen Kosten darstellt, ist (A) mit (A,B) durch eine S-Kante verbunden, während (B) mit (A,B) durch eine A-Kante verbunden ist. Also wird in Ebene 2 die Gruppierung (A,B) in Attributreihenfolge (B,A) berechnet, so dass (B) ohne zusätzliche Sortierungskosten aus (B,A) berechnet werden kann, während (B,A) für die Berechnung von (A) wieder umsortiert werden muss. Dadurch dass (C) mit (A,C) von einer A-Kante verbunden wird, wird (A,C) in der Reihenfolge (C,A) berechnet während die Reihenfolge von (B,C) beliebig gewählt werden kann, da (B,C) mit keinem Ebene 1 Knoten verbunden ist.

In Bild 3 wird dargestellt, welches Ergebnis der PipeSort-Algorithmus für das Suchgitter aus Bild 1 ausgibt(Bild 3(a)) und welche Pipelines ausgeführt werden(Bild 3(b)). Der Einfachheit halber wird angenommen, dass die A- und S-Kosten für alle aus einer Gruppierung g berechenbaren Gruppierungen gleich sind. In Bild 3(a) wird dieses Kostenpaar unter der Gruppierung g eingetragen. Durchgezogene Kanten stehen für A-Kosten, wohingegen gestrichelte Kanten für S-Kosten stehen. In Bild 3(b) werden durchzuführende Sortierungen durch Ellipsen angedeutet.



(a) Plan mit minimalen Kosten (b) Auszuführende Pipelines

Bild 3: Berechnen eines Würfels mit vier Dimensionen durch PipeSort

Die Laufzeit des *PipeSort*-Algorithmus wurde in [SAG+96] empirisch untersucht. Dabei wurde festgestellt, dass der *PipeSort*-Algorithmus nicht nur wesentlich schneller ist als die isolierte Berechnung der einzelnen Gruppierungen sondern sogar sehr nahe an die geschätzten unteren Schranken für die Berechnung des Datenwürfels kommt. Es wurde also durch die geschickte Kombination altbekannter Optimierungsstrategien eine nahezu ideale Laufzeit erreicht. Dazu kommt, dass im so berechneten Datenwürfel vorberechnete Aggregate gespeichert sind, die wiederum die Laufzeit von Aggregationsanfragen auf dem Würfel optimieren.

Dieses Kapitel hat damit den CUBE-Operator als externe Schnittstelle vorgestellt, seine interne Darstellung als Suchgitter geschildert und seine effiziente Berechnung durch den *PipeSort*-Algorithmus illustriert. Das folgende Kapitel 3 wird sich nun mit der Auswertung von Bereichsanfragen auf einem so berechneten Datenwürfel befassen.

Kapitel 3: Bereichsanfragen

Dieses Kapitel beschäftigt sich mit der Berechnung von *Bereichsanfragen*. Folglich ist es zunächst notwendig, den Begriff der Bereichsanfrage zu definieren:

Definition: Eine **Bereichsanfrage** ist eine Anfrage auf einem Datenwürfel, die eine Aggregatfunktion auf ausgewählte Zellen des Würfels anwendet, wobei die ausgewählten Zellen durch zusammenhängende Bereiche in den Wertemengen der Dimensionen bestimmt sind.

Die im Datenwürfel vorberechneten Aggregate können hier bei der Auswertung von Bereichsanfragen nicht genutzt werden, da hier davon ausgegangen wird, dass in jeder Dimension eine Selektion stattfindet.

Die Bereichsanfragen sind eine sehr häufig benötigte OLAP-Anfrage, da sie für jede Problemstellung der Art „Wie hat sich der Umsatz unserer Firma in den letzten drei Quartalen entwickelt?“ benötigt wird. Es lohnt sich also, sie einer detaillierten Betrachtung zu unterziehen.

Hier wird nur eine spezielle Art der Bereichsanfrage betrachtet, nämlich die Bereichssummenanfrage, welche die Summe aller Werte in einem bestimmten Bereich des Würfels berechnet. Um diese Summenberechnung möglichst effizient zu gestalten, müssen einzelne Summen schon vorberechnet werden.

Die Gliederung dieses Kapitels ist nun so aufgebaut, dass in Abschnitt 3.1 das in den folgenden Abschnitten verwendete Modell erläutert wird. Die Abschnitte 3.2 bzw. 3.3 werden dieses Modell zur Beschreibung des Range-Sum-Algorithmus ohne bzw. mit Blockbildung nutzen. Die beiden Algorithmen verwenden zur Beschleunigung ein vorberechnetes Präfixsummenarray, welches je zu Anfang des jeweiligen Abschnitts erläutert wird. Darauf folgt dann die Darstellung der Algorithmen.

3.1 Vorstellung des verwendeten Modells:

Sei D = {1, 2, ..., d} die Menge der Dimensionen eines Datenwürfels. Der d-dimensionale Datenwürfel kann dann durch ein d-Dimensionales Array A der Größe $n_1*n_2*...*n_d$ ($n_j>1$, $j \in D$) beschrieben werden. Dann ist Gesamtgröße von A festgelegt durch $N = \prod_{j=1}^{d} n_j$. In dieser Ausarbeitung werden alle Bereichsanfragen auf einem Datenwürfel auf Array A bezogen.

Die Berechnung einer Bereichssummenanfrage kann somit durch folgende Gleichung ausgedrückt werden:

$$Sum(l_1 : h_1, ..., l_d : h_d) = \sum_{i_1=l_1}^{h_1} ... \sum_{i_d=l_d}^{h_d} A[i_1,...,i_d].$$

Weiterhin wird eine *Region*($l_1 : h_1, ..., l_d : h_d$) R einen d-dimensionalen Raum bezeichnen, der für alle Dimensionen $j \in D$ begrenzt ist durch $l_j \le i_j \le h_j$. Das *Volumen* V einer Region ist definiert durch die Anzahl ihrer Zellen, also $V(R) = \prod_{j=1}^{d} (h_j - l_j + 1)$. Das Volumen einer Bereichsanfrage ist bestimmt durch das Volumen der Region, durch die die Bereichsanfrage definiert ist. Die Bereichsvariablen l_j und h_j sind benutzerspezifisch und im Allgemeinen nicht im vorhinein bekannt, währenddessen Array A als bekannt vorrausgesetzt wird.

Zuletzt muss erwähnt werden, dass ein dicht besetzter Datenwürfel vorrausgesetzt wird.

3.2 Range-Sum-Algorithmus ohne Blockbildung [HAMS97]:

Die Grundlage des Range-Sum-Algorithmus ohne Blockbildung ist das Präfixsummenarray ohne Blockbildung, das nun definiert werden soll:

Definition: Das **Präfixsummenarray** P ist ein d-dimensionales Array der Größe

N = $n_1 * n_2 * ... * n_d$, womit es dieselbe Kardinalität wie A besitzt.

P wird genutzt um Präfixsummen aus A zu speichern. In P werden folgende Werte gespeichert:

$$\forall(0 \le x_j < n_j) \wedge (j \in D): \qquad P[x_1,...,x_d] = Sum(0:x_1,...,0:x_d) = \sum_{i_1=0}^{x_1} ... \sum_{i_d=0}^{x_d} A[i_1,...,i_d].$$

Bild 4 gibt ein Beispiel für ein zweidimensionales Array A[x, y] und sein entsprechendes Präfixsummenarray P[x, y] für $n_1 = 6$ und $n_2 = 3$.

Array A						
Index	0	1	2	3	4	5
0	3	5	1	2	2	3
1	7	3	2	6	8	2
2	2	4	2	3	3	5

Array P						
Index	0	1	2	3	4	5
0	3	8	9	11	13	16
1	10	18	21	29	39	44
2	12	24	29	40	53	63

Bild 4: Zweidimensionales Array A und das zugehörige Präfixsummenarray P

Die effiziente Berechnung von P aus A kann in [HAMS97] nachgeschlagen werden.

Der Range-Sum-Algorithmus ohne Blockbildung ermöglicht es, jede Bereichssumme über A aus bis zu 2^d geeigneten Elementen von P zu berechnen. Er wird durch eine einfache Gleichung spezifiziert:

$$\forall\, j \in D: \quad Sum(l_1 : h_1, ..., l_d : h_d) = \sum_{\forall x_j \in \{l_j - 1, h_j\}} \{(\prod_{i=1}^{d} s(i)) * P[x_1,...,x_d]\},$$

$$\text{wobei } s(j) = \begin{cases} 1, \textit{falls} & x_j = h_j \\ -1, \textit{falls} & x_j = l_j - 1 \end{cases}.$$

Der korrekten Notation willen sei $P[x_1, ..., x_d] = 0$, falls $x_j = -1$für ein $j \in D$.

Beispiel: Für d = 2 kann die Bereichssumme $Sum(l_1 : h_1, l_2 : h_2)$ in drei Schritten berechnet werden:

$Sum(l_1 : h_1, l_2 : h_2) = P[h_1, h_2] - P[h_1, l_2 - 1] - P[l_1 - 1, h_2] + P[l_1 - 1, l_2 - 1]$.

Die Bereichssumme $Sum(2 : 3, 1 : 2)$ angewendet auf das Beispiel in Bild 4 kann berechnet werden als

$Sum(2 : 3, 1 : 2) = P[3,2] - P[3,0] - P[1,2] + P[1,0] = 40 - 11 - 24 + 8 = 13$.

Bild 5 verdeutlicht den Range-Sum-Algorithmus ohne Blockbildung für d = 2 geometrisch.

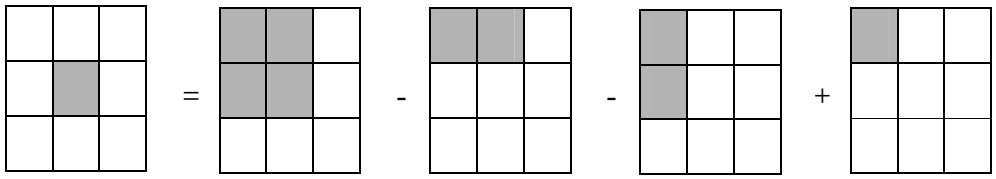


Bild 5: Eine geometrische Verdeutlichung für d = 2:
Sum(Region_E) = *Sum*(Region_A) - *Sum*(Region_B) - *Sum*(Region _C) + *Sum*(Region _D).

Dadurch, dass eine ebenso große Menge an vorberechneten Aggregaten wie Originaldaten existiert, ist dieser Algorithmus sehr schnell. Allerdings möchte man Array A und Array P in aller Regel nicht beide vollständig im Speicher haben, um diesen nicht zu stark zu belasten.

Sobald jedoch Array P berechnet ist, bietet sich die Möglichkeit Array A zu löschen, da jedes Element von A als eine spezielle Bereichssumme aufgefasst werden kann:

$A[x_1, ..., x_d] = Sum(x_1 : x_1, ..., x_d : x_d)$. Dadurch wird der zur Anfrageverarbeitung benötigte Speicher drastisch reduziert. Allerdings resultiert ein Zugriff auf $A[x_1, ..., x_d]$ in einem Zugriff auf bis zu 2^d Elemente von P. Diese Strategie bietet einen interessanten Kompromiss zwischen Laufzeit und benötigtem Speicher, insbesondere falls d klein ist.

3.3 Range-Sum-Algorithmus mit Blockbildung [HAMS97]:

Eine andere Variation Speicher zulasten der Laufzeit zu sparen ist es, die Präfixsummen auf einer gröberen Ebene zu speichern, d.h. eine Präfixsumme wird nur gespeichert, falls ihr Index entweder ein Dekrement eines Vielfachen eines *Blockfaktors* b ist oder der höchste Index einer Dimension. Formal ausgedrückt bedeutet dies:

$P[i_1, \ldots, i_d]$ wird gespeichert, falls für alle $j \in D$ $(i_j+1) \bmod b = 0$ <u>oder</u>
für beliebiges $j \in D$ $i_j = n_j - 1$.

Also wird für jeden d-dimensionalen Block der Größe b^d nur annähernd [5] eine Präfixsumme gespeichert. Daher wird ein solches Präfixsummenarray als *Präfixsummenarray mit Blockbildung* bezeichnet. Bild 6 zeigt ein Beispiel für ein Präfixsummenarray mit Blockbildung.

Array P						
Index	0	1	2	3	4	5
0	-	-	-	-	-	-
1	-	18	-	29	-	44
2	-	24	-	40	-	63

Bild 6: Beispiel für ein Präfixsummenarray mit Blockbildung für b = 2, d = 2.

In einer praktischen Implementierung würde das dünn besetzte Array P der Größe N und Dichte $\approx 1/b^d$ in ein dicht besetztes Array der ungefähren Größe N/b^d komprimiert werden. Hier wird jedoch der Überschaubarkeit wegen auf eine solche Komprimierung verzichtet. In Unterabschnitt 3.2.5 wird das Problem der richtigen Wahl des Blockfaktors b vertieft. Außerdem ist es offensichtlich, dass bei Speicherung eines Präfixsummenarrays mit Blockbildung das Originalarray A nicht gelöscht werden darf.

Beschrieben wird nun der Range-Sum-Algorithmus mit Blockbildung, der das Präfixsummenarray P mit Blockbildung und das Originalarray A benutzt. Als gegeben vorrausgesetzt werden die Dimensionalität d des Datenwürfels, der Blockfaktor b sowie die Bereichssummenanfrage $Sum(l_1 : h_1, \ldots, l_d : h_d)$.

Zunächst müssen einige Notationen festgelegt werden:

Für alle $j \in D$ seien $l''_j = b \lfloor l_j/b \rfloor$, $l'_j = b \lceil l_j/b \rceil$, $h'_j = b \lfloor h_j/b \rfloor$, und

$h''_j = \min(b \lceil h_j/b \rceil, n_j)$.

Offensichtlich gilt $l''_j \leq l_j \leq l'_j$ und $h'_j \leq h_j \leq h''_j$. Dahingegen impliziert $l_j < h_j$ nicht notwendigerweise $l'_j < h'_j$. Deshalb unterscheidet der Algorithmus folgende zwei Fälle:

Fall 1 tritt ein, falls für alle $j \in D$ $l'_j < h'_j$ gilt, während es bei Fall 2 mindestens ein $j \in D$ gibt, so dass $l'_j \geq h'_j$.

[5] Im Schnitt wird aufgrund der Speicherung im Falle des höchsten Indexes einer Dimension etwas mehr als eine Präfixsumme gespeichert.

Fall 1: Um die Bereichssumme *Sum*($l_1 : h_1$, …, $l_d : h_d$) zu berechnen, wird ihre zugehörige Region in 3^d disjunkte Unterregionen aufgeteilt wie folgt: Sei

$R_j = \{l_j : l'_j\text{-}1, l'_j : h'_j\text{-}1, h'_j : h_j\}$ eine Menge von drei aneinander angrenzenden Bereichen.

Dann ist *Sum*($l_1 : h_1$, …, $l_d : h_d$) = $\sum_{\forall r_j \in R_j} Sum(r_1,...,r_d)$. Intuitiv wird hier der Bereich jeder Region in drei angrenzende Unterbereiche aufgeteilt, so dass der mittlere Unterbereich mit der Blockstruktur übereinstimmt. Somit wird das kartesische Produkt dieser d Dimensionen 3^d disjunkte Unterregionen bilden. Einige dieser3^d Regionen können auch leer sein. Bild 7(a) gibt ein bildhaftes Beispiel für eine Anfrageregion *Region*(50 : 350, 50 : 350) (die schraffierte Fläche im Bild).

Bild 7(b) zeigt die $3^2 = 9$ zerlegten Regionen für diese Anfrage.

Weiterhin heißt diejenige Unterregion *Region*(r_1, …, r_d) eine *interne Region*, für die gilt:

$r_j = l'_j : h'_j\text{-}1$ für alle $j \in D$.

Alle anderen Unterregionen heißen *Grenzregionen*. In Bild 7(b) ist A5 die interne Region.

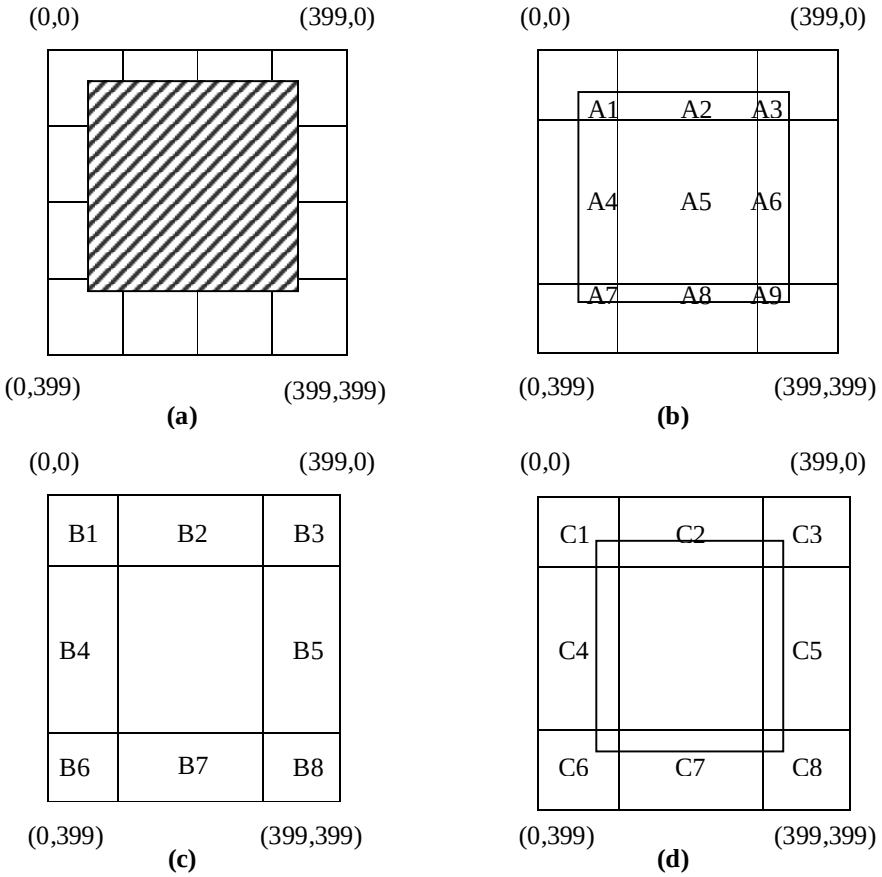


Bild 7: Berechnung von *Sum*(50 : 349, 50 : 349) durch den Range-Sum-Algorithmus mit Blockbildung mit b = 100.

Um eine Bereichssumme zu berechnen, wird nun zunächst die zugehörige interne Region berechnet. Diese stimmt mit der Blockstruktur überein, und kann daher allein aus bis zu 2^d Elementen des Präfixsummenarrays mit Blockbildung berechnet werden. Um die verbleibenden 3^d - 1 Unterbereichssummen aufzuaddieren wird Array A benötigt und möglicherweise auch Array P. Für jede Grenzregion R = *Region*(r_1, …, r_d) gibt es eine
Superblock-Region B = *Region*(B_1, …, B_d), wobei für alle $j \in D$

$$B_j = \begin{cases} l_j'' : l_j', \textit{falls} \quad r_j = l_j : l_j' - 1, \\ r_j, \textit{falls} \quad r_j = l_j' : h_j' - 1, \\ h_j' : h_j'', \textit{falls} \quad r_j = h_j' : h_j \end{cases}.$$

Weiterhin ist für jede Grenzregion ihre Komplement-Region durch *Region*(B_1, …, B_d) - *Region*(r_1, …, r_d) definiert. Also ist die Superblock-Region einer Grenzregion R die kleinste Region, die R enthält und mit der Blockstruktur übereinstimmt. Bild 7(c) und (d) zeigen die zu Bild 7(b) zugehörigen Superblock- und Komplementregionen.

Für jede Grenzregion R gibt es nun zwei Möglichkeiten ihre Bereichssumme zu berechnen. Erstens kann man einfach alle Elemente aus A die zu R gehören aufsummieren. Andererseits besteht auch die Möglichkeit, die Summe der Elemente aus A die zur Komplementregion von R gehören von der Bereichssumme der Superblock-Region zu subtrahieren. Um die Zeitkomplexität zu minimieren wählt der Range-Sum-Algorithmus die erste Möglichkeit, falls das Volumen von R kleiner oder gleich dem Volumen seiner Komplementregion plus $2^d - 1$ ist, ansonsten die zweite. Diese Auswahl findet für jede Grenzregion unabhängig statt.

Fall 2: Durch eine einfache Modifikation kann der Algorithmus für Fall 1 direkt auf Fall 2 angewendet werden. Sei

$R_j = \{l_j : l_j' \text{-} 1, l_j' : h_j' \text{-} 1, h_j' : h_j\}$(wie gehabt) falls $l_j' < h_j'$, *ansonsten* sei $R_j = \{l_j : h_j\}$. Zudem muss in ähnlicher Weise die Definition einer Superblock-Region angepasst werden, nämlich für alle $j \in D$:

$$B_j = \begin{cases} l_j'' : l_j' - 1, \textit{falls} \quad r_j = l_j : l_j', \\ r_j, \textit{falls} \quad r_j = l_j' : h_j' - 1, \\ h_j' : h_j'' - 1, \textit{falls} \quad r_j = h_j' : h_j, \\ l_j'' : h_j'' - 1, \textit{falls} \quad r_j = l_j : h_j. \end{cases}.$$

Mit diesen zwei Änderungen kann der Algorithmus aus Fall 1 angewendet werden.

Durch die geblockte Speicherung von Präfixsummen wird es möglich, den Blockfaktor b derart zu wählen, dass das Präfixsummenarray vollständig in den Speicher passt und die Zugriffszeiten somit optimiert werden. Dazu muss der Blockfaktor nur groß genug gewählt werden. Demgegenüber steht die Laufzeitverbesserung desto kleiner b wird, weil dann nur auf wenige Elemente von Array A zugegriffen werden muss. In [HAMS97] wird mithilfe einer Nutzen/Speicher-Funktion vorgestellt, wie der Blockfaktor optimal zu wählen ist.

Zusammenfassend bleibt zu sagen, dass der Range-Sum-Algorithmus mit oder ohne Blockbildung Bereichsanfragen aufgrund der Vorberechnung des Präfixsummenarrays sehr effizient berechnet. Durch die mögliche Blockbildung kann er sehr flexibel auf den vorhandenen Speicher eingestellt werden. Beim Range-Sum-Algorithmus ohne Blockbildung kommt hinzu, dass die Möglichkeit besteht das Originalarray A zu löschen, und so den Speicheraufwand um die Hälfte zu reduzieren. Jede Anfrage die auf ein einzelnes Element des gelöschten Arrays A zugreifen will, wird nun als Bereichssummenanfrage auf P formuliert. Da in OLAP-Anfragen jedoch relativ selten ein einzelnes Element ausgelesen wird ist der Mehraufwand zur Berechnung eines einzelnen Element durch den Nutzen des weniger genutzten Speichers gerechtfertigt.

Kapitel 4: Zusammenfassung

In dieser Ausarbeitung wurden schnelle Algorithmen zur Berechnung des Datenwürfels und zur Berechnung von Bereichssummenanfragen präsentiert, um die Anfrageevaluierung sowie deren Optimierung zu veranschaulichen.

Die hauptsächliche Idee die Berechnung des Datenwürfels zu beschleunigen war es, nicht alle Gruppierungen isoliert zu berechnen, sondern mehrere Gruppierungen parallel in einer Art Pipeline zu berechnen und dabei darauf zu achten, dass jede Gruppierung in einer bestimmten Sortierungsreihenfolge, die für die nachfolgenden Berechnungen vorteilhaft ist, vorliegt. Der *PipeSort*-Algorithmus benutzt diese Strategie, wodurch seine Laufzeit nahe an die geschätzten unteren Schranken zur Bildung eines Datenwürfels kommt.

Um hingegen die Berechnung einer Bereichssumme zu beschleunigen, wurden Präfixsummenarrays des Datenwürfels vorberechnet. Dadurch kann jede Bereichssummenanfrage durch Auslesen von 2^d geeigneten Präfixsummen beantwortet werden. Der Range-Sum-

Algorithmus bietet dazu flexible Möglichkeiten, den benötigten Speicherplatz zu verringern. Dies geschieht durch die Blockbildung, bei welcher Präfixsummen auf einer gröberen Ebene gespeichert werden. Zur Berechnung der Bereichssumme müssen nun 2^d Präfixsummen sowie einige Elemente des Datenwürfels ausgelesen werden.

Referenzliteratur

[HAMS97] Ho, C.-T.; Agrawal, R.; Megiddo, N.; Srikant, R.: “Range Queries in OLAP Data Cubes”. In Proceedings of the ACM SIGMOD Conference on Management of Data, S. 73-88, Tucson, Arizona, USA, 1997.

[SAG+96] Agrawal S.; Agrawal R.; Deshpande, P. M.; Gupta, A; Naughton, J. F.; Ramakrishnan, R.; Sarawagi, S.: “On the Computation of Multidimensional Aggregates”. In Proceedings of the VLDB Conference, S. 506-521, Bombay, India, 1996.

[PS82] Papadimitriou, C. H.; Steiglitz, K.: “Cobinatorial Optimization: Algorithms and Complexity”. Englewood Cliffs, New Jersey, Prentice Hall, 1982.

[BG01] Bauer, A.; Günzel, H.: „Data-Warehouse-Systeme: Architektur, Entwicklung, Anwendung“. Erlangen, Deutschland, dpunkt.verlag, 2001.