Notizen zur Diskussion des Papers: Optimizing Queries Using Materialized Views, von Goldstein und Larson

Note Taker: Patrick Rappensberger

13.05.2015

1 Was ist das view-matching Problem, welches in der Arbeit Beachtung findet? Was sagt es aus?

Ein grundlegendes Problem aller Datenbanksysteme ist die effiziente Handhabung von Queries. Diese sollen möglichst effizient verarbeitet werden. Dieses Paper beschreibt die Idee Anfragen mit eigenen vordefinierten Views zu beantworten. Im Prinzip ist diese Vorgehensweise eine Art von Cashing, weil weniger Daten vom System angefordert werden müssen. Besonders effizient ist diese Technik wenn wenige Ergebnistuppel erwartet werden.
In diesem Paper wurde auf folgende Themen eingegangen (Zusätzliche Frage während der Diskussion im Zusammenhang mit Frage 1):

- Der Algorithmus zum Auswählen der View im Allgemeinen
- Verwendete Datenstrukturen
- Welche Tests verwendet werden
- Wie mit Aggregated Queries umgegangen wird
- Experimente / Ergebnisse

2 In der Arbeit werden die drei verschiedenen Tests besprochen, um zu zeigen, das alle Reihen die von der Query benötigt werden in der View enthalten sind. Welche waren die drei Tests? Warum vereinfacht die Abwesenheit von Disjunktionen bei einem der Tests den Algorithmus bzw. warum könnte es den Algorithmus komplexer machen? (Betrifft konkret den Range subsumption Test und die Aussage, dass es ohne Disjunktionen einfacher ist. Leider geht nicht hervor, was das Problem hierbei war.)

Folgende drei Tests wurden in diesem Paper beschrieben:

- Equijoin subsumption test
- Range subsumption test
- Residul subsumption test

Die Funktion der Tests ist in Frage 7 genauer beschrieben und die mathematische Herleitung der Tests ist in Frage 4 beantwortet.

Bei der Diskussion zum Thema Disjunktion beim range subsumption Test, wurden folgendes ermittelt: Das Problem bei Disjunktionen ist die Tatsache, das die Range nicht einfach festgestellt werden kann, weil sich Bereiche der einzelnen Terme leicht überschneiden können und somit eine einfache Implementierung nicht mehr ausreicht. Es gibt kein globales Minimum, oder Maximum bei den Ranges (z.B.: bei der Abfrage $x > 5 \wedge x < 4$), somit funktioniert die beschriebene Technik zum Überprüfen der Range nicht mehr. Im Paper wurde folgende Methode vorgeschlagen:
Sich von beiden Seiten $(-\infty, +\infty)$ nähern und den Intervall durch die Konjunktionsterme eingrenzen. Das bedeutet das der Bereich mit der kleinsten Range das Minimum darstellt und der Bereich mit der größten Range das Maximum. Zum Beispiel $x < 5 \wedge x > 3$. Dann ergibt sich als Bereich den die View erfüllen muss das Intervall [3,5].

3 In der Arbeit von Goldstein und Larson wird das Fast Filtering von Views mittels "filter trees" und "lattice index" vorgestellt. Für die späteren Experimente und deren Auswertung haben sie einen Baum verwendet, welcher 8 Levels/Ebenen hat und sich wie folgt aus diesen Partitioning Conditions zusammensetzt:" From top to bottom, the levels are: hubs, source tables, output expressions, output columns, residual constraints, and range constraints. For aggregation views, there are two additional levels: grouping expressions and grouping columns." Dieser Filter Tree wirkt sich natürlich in seiner Zusammensetzung auf die Ergebnisse aus, ist dies der am ehesten optimale Baum oder gibt es womögliche bessere Anordnungen? Oder folgt der Aufbau logisch aus den verschiedenen Partitioning Conditions? Wieso kommen dann hubs vor source tables? Warum immer expressions vor columns?

Die optimale Anordnung wird es nicht geben allerdings, schränkt die Hub-Condition die Suchmenge der Views stärker ein als die source tables.Dadurch müssen in späteren Schritten weniger Views durchsucht werden und das ist logischerweise performanter.

Source table condition: Bei dieser Condition wird überprüft ob ein View alle nötigen Tabellen besitzt um die Anfrage beantworten zu können (Minimum).

Hub condition: Ein Hub einer View ist die kleinstmögliche Menge an Sets (Columns/Tables) die verwendetet werden kann um die Anfrage beantworten zu können (Maximum).

4 Zeigen Sie dass $W_q \rightarrow W_v$ gilt, falls das Ergebnis von SELECT Anweisung SELECT * FROM $T_1, T_2, ..., T_n$ WHERE W_q erzeugt eine Untermenge des SELECT * FROM $T_1, T_2, ..., T_n$ WHERE W_v Anweisung.

Eine verbale Erklärung befindet sich in Frage 7. Hier befinden sich die Formeln/Umformungen um formal auf die Test schließen zu können:

W_q sind die Prädikate der WHERE-Clause in der Query, W_v die Prädikate in der View. Gesucht ist also ein Algorithmus um $W_q \rightarrow W_v$ festzustellen.

Wir können W_q folgendermaßen schreiben: $W_q = P_(q,1) \wedge P_(q,2) ... \wedge P_(q,m)$. Selbiges gilt für W_v, also $W_v = P_(v,1) \wedge P_(v,2) ... \wedge P_(v,n)$.

W_q und W_v werden in drei Teile aufgespalten und es wird in die Implikation eingesetzt.

$PE_q \wedge PR_q \wedge PU_q \rightarrow PE_v \wedge PR_v \wedge PU_v$

Die Implikation kann aufgeteilt werden zu
$PE_q \wedge PR_q \wedge PU_q \rightarrow PE_v \wedge$; $PE_q \wedge PR_q \wedge PU_q \rightarrow PR_v \wedge$; $PE_q \wedge PR_q \wedge PU_q \rightarrow PU_v$
Die drei Tests (Equijoin subsumption test, Range subsumption test und Residual subsumption test) folgen aus der weiteren Aufteilung der Formel und sind ebenfalls in Frage 7 genauer beschrieben.

- $PE_q \rightarrow PE_v$
- $PE_q \wedge PR_q \rightarrow PR_v$
- $PE_q \wedge PU_q \rightarrow PU_v$

5 Welche Bedingungen muss ein View erfüllen, um Aggregat Query von View zu berechnen?

Alle Group-By Expressions müssen auch in der Ausgabe sein um einen eindeutigen Schlüssel für jede Zeile zu erhalten. Außerdem ist die einzige Aggregationsfunktion die erlaubt ist in materialized views die Funktion sum(). Zusätzlich müssen folgende Bedingungen zutreffen:

- Der SPJ Teil der View muss alle nötigen Zeilen produzieren die vom SPJ-Teil der Query verlangt werden (inklusive dem richtigen Duplikationsfaktor).
- All Spalten die von compenstating predicates verlangt werden müssen vom View ausgegeben werden können.
- Die View besitzt keine Aggregation, oder ist weniger aggregiert als die Query.
- Alle Spalten die für Grouping Operationen benötigt werden muss die View liefern können.
- Alle Spalten die verwendet werden um den Output Expressions zu berechnen müssen vom View ausgegeben werden können.

Im Paper wurden Compensating Predicates vorgestellt. Compensating Predicates sind Prädikate die während der View Erstellung erstellt werden. Im folgenden Beispiel ist also *cnt* und *revenue* Compensating Predicates.

```
CREATE VIEW v4 WITH SCHEMABINDING AS
SELECT o_custkey, count_big(*) AS cnt
SUM(l_quantity*l_extendedprice) AS revenue
FROM dbo.lineitem,dbo.orders
WHERE l_orderkey = o_orderkey
GROUP BY o_custkey
```

6 Was ist ein filter tree?

Ein filter tree ist ein multiway search tree bei dem alle Blätter sich in der selben Ebene befindet. Ein Knoten des Baumes enthält Schlüssel,Pointer-Paare. Ein Schlüssel ist kein einzelner Wert, sondern immer eine Menge aus Werten/Variablen. Ein filter tree wird verwendet um effizient passende Views für eine Query zu finden. In diesem Paper wird vor allem auf **Lattice indizes** eingegangen. Dieser ist ein Frage 8 genauer beschrieben. Ein filter tree setzt sich aus mehreren lattice indizes zusammen, die verschachtelt sind und weil nur jene Subsets weiter verfolgt werden, die dem Test/Anfrage entsprechen wird ein Baum aufgebaut.

In Grafik 1 wird ein Beispiel eines filter tree dargestellt. Die Suche beginnt in Ebene 1, wobei die weiteren Ebenen nur Verschachtelungen einzelner Sets sind. Zum Beispiel erfüllt in Ebene 1 nur s3 und s4 die gesuchte Bedingung und innerhalb dieser Sets wird weiter gesucht. In Ebene 2 erfüllen im Set 4, s3 die gesuchte Bedingung und im Set 3, s1 die Bedingung. Das Ergebnis sind dann Mengen die die Anfrage erfüllen. In unserem Fall sind diese Mengen Views die auf die Anfrage getestet werden.

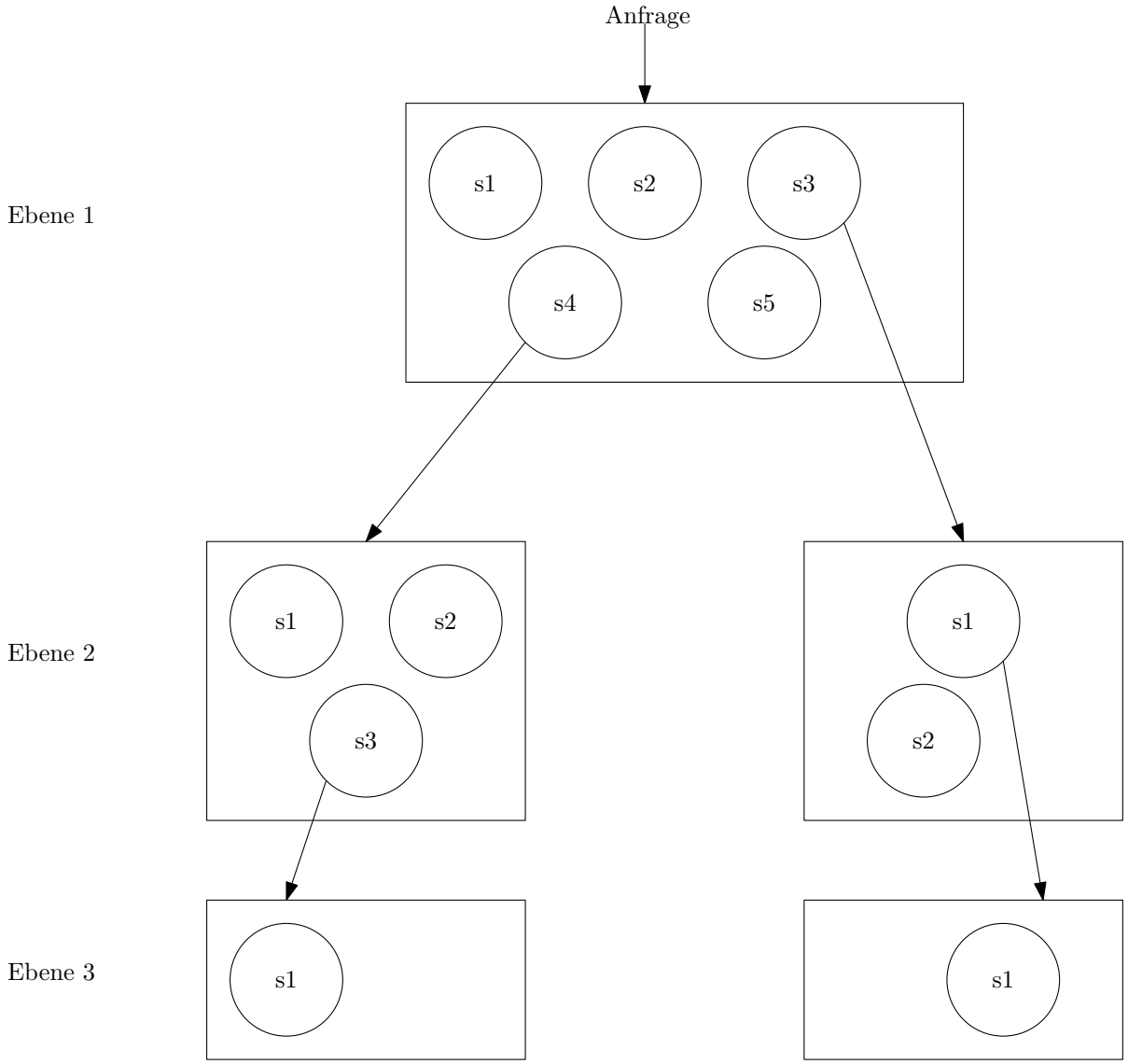


Figure 1: Beispiel filter tree

7 How does the decision algorithm (to determine if all required rows exist in a view) work in principle and what are the three tests applied?

Folgende drei Tests werden verwendet:

- Equijoin subsumption test
- Range subsumption test
- Residul subsumption test

7.1 Equijoin subsumption test

Im Prinzip wird bei diesem Test darauf geachtet, das alle notwendigen Spalten, um die Anfrage beantworten zu können, vorhanden sind. Dazu werden **Äquivalenzklassen** für jede Spalte berechnet (für die Query und für Views).

Eine Äquivalenzklasse ist eine Menge von Spalten die Gleich sind. Definiert/Berechnet werden diese wie folgt:
Sei $W = P_1 \wedge P_2 ... \wedge P_n$ ein Selektionsprädikate in CNF (conjunctive normal form) eines SPJ (join-select-project) Ausdrucks. Durch das Zusammenfassen von Konjunktionen kann W als $W = PE \wedge PNE$ geschrieben werden, wobei PE Spalten Gleichheitsprädikate der Form $T_i.C_p = T_j.C_q$ enthält. T_j und T_i sind Tabellen und C_p und C_q Referenzen auf Spalten. PNE enthält alle anderen Konjunktionen. Um nun Äquivalenzklassen zu berechnen geht man wie folgt vor:
Jede Spalte einer Tabelle stellt eine eigene Menge dar. Durchsuche alle column equavility predicates in beliebiger Reihenfolge. Falls zwei gleiche Spalten gefunden werden ($T_i.C_p = T_j.C_q$), vereinige die beiden Mengen, sonst tu nichts. Nachdem alle column equavility predicates durchgegangen wurden stellen die verbleibenden Mengen die Äquivalenzklassen dar.

Danach wird überprüft ob jede nicht triviale Äquivalenzklasse eines Views ein Subset einer Query Äquivalenzklasse ist. Falls diese Bedingung stimmt , ist garantiert das es keine Konflikte zwischen Spalten des Views und Spalten der Query gibt. Weil alle Spalten von View und Query überprüft werden, werden auch bereits einfache Transitive Abhängigkeiten innerhalb der Spalten berücksichtigt.

Nachdem der Test durchgeführt wurde erhält man somit eine Unterschiedliche Anzahl an Äquivalenzklassen (E1,E2,E3) und dazu passende Views (V1,V2,V3) in der jeweiligen Klasse.

7.2 Range subsumption test

Der range subsumption test wird verwendet um festzustellen ob die Grenzen des Views größer sind als die der Query. Das muss überprüft werden damit beim Beantworten der Anfrage nicht weniger Ergebnistupple zurückgegeben werden als wenn keine Views verwendet werden würden.

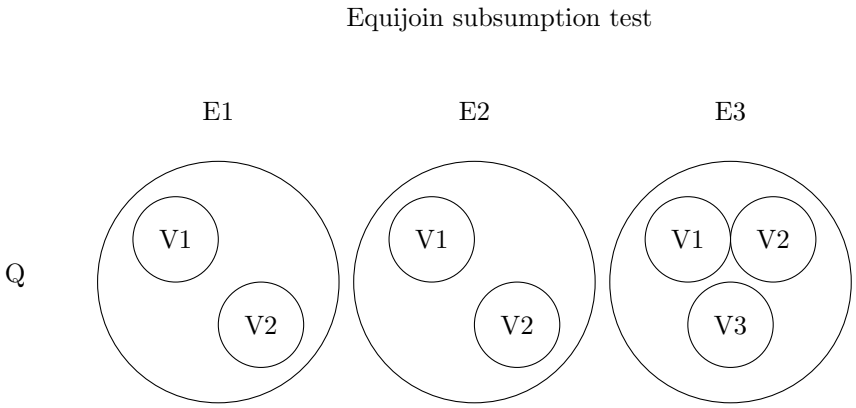


Figure 2: Schema des Equijoin subsumption test

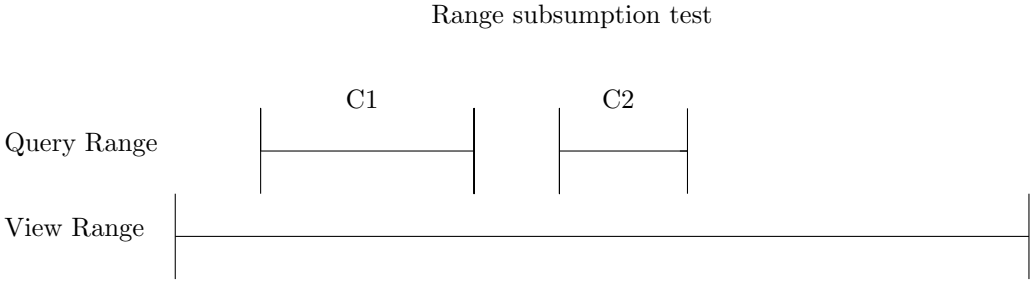


Figure 3: Schema des Range subsumption test

Wenn in der Selektion keine logischen Oder-Verknüpfungen vorkommen wird in diesem Paper ein simpler Algorithmus vorgestellt wie die Grenzen überprüft werden können.

Die folgende Grafik ist ein Beispiel für solch einen Test. Es wird überprüft ob die Range des Views ausreicht um die Query richtig und vollständig beantworten zu können. C1 und C2 stehen Beispielhaft für die Ranges der einzelnen Konjunktionsterme.

7.3 Residual subsumption test

Für Conjuncts die weder column equavility Prädikate noch range Prädikate besitzten, wird dieser Test angewandt. In diesem Test wird verglichen ob jede überbleibende Conjunction im View-Residual Prädikat ein gleichwertiges in der gleichen Query Äquivalenzklasse besitzt. Falls nicht wird der View verworfen, weil der View zusätzliche Prädikate enthält die nicht in der Query vorkommen und somit nicht benötigt werden.Bei diesem Test wird allerdings nicht mehr auf Grenzen in der View oder in der Anfrage geachtet.

7.4 Algorithmus

In dieser Sektion ist der Algorithmus zusammengefasst um eine passende View für eine Abfrage zu verwenden.

- Ermittle die Äquivalenzklassen für die View und die Query.
- Für jede View überprüfe ob jede Äquivalenzklasse der View eine Untermenge einer Äquivalenzklasse Query ist.
- Berechne die Grenzen/Intervalle für die Query und die übergebliebenen Views.
- Überprüfe die Grenzen der Query und der Views. Falls die Intervallgrenzen einer View nicht passen, verwerfe die View.
- Überprüfe ob alle übrigen Conjunctions der restlichen Views eine passende in der Query besitzten. Falls nicht entferne die Queue.

8 Why is it not sufficient to just keep view information in the memory to speed up view matching? Describe briefly the data structure used to solve this problem.

Alle Views im Speicher zu halten und linear zu durchsuchen würde zu viel Zeit in Anspruch nehmen. Um das View-Matching zu beschleunigen werden unter anderem *hub conditions* und *source tree conditions*verwendet (siehe Frage 3).

In diesem Paper wurden die Datenstrukturen Lattice-Index und darauf aufbauend der Filter-Tree diskutiert. Der Filter-Tree ist in Frage 6 ausführlicher Beschrieben.

Ein Lattice Index ist ein Graph und wird von Halbordungsrelationen aufgebaut. Die Suche beginnt immer von der obersten Ebene und reduziert sich dann auf Untermengen dieser Menge. Ein Lattice Index ist doppelt verlinkt, allerdings gibt es keine Verbindungen innerhalb einer Ebene. Die Knoten selbst sind Mengen und ihre Kinder Untermengen dieser Menge. Es darf nur auf die nächst kleinste Übermenge verlinkt werden (also keine Ebene übersprungen werden, wenn es dazwischen eine passende Übermenge gibt).
In Grafik 4 wird der Lattice Index des Papers dargestellt:

Falls nun zum Beispiel (siehe Grafik 4) nach der Menge A gesucht wird, traversiert man den Baum von oben nach unten. Also wird in der obersten Ebene überprüft welche Mengen, B enthalten. In unserem Fall sind das ABC und ABF. Von diesen Mengen aus werden die Pointer zu den Kindern verfolgt und diese Überprüft ob sie A enthalten. In diesem Beispiel ist gibt es nur ein Kind das auch die Bedingung erfüllt (AB). Von dort wird wieder nach A gesucht und auch gefunden. Mithilfe eines Lattice Index erhält man immer die kleinste Menge, die das gesuchte Element/Menge enthält.
In diesem Paper werden anstatt Mengen Views verwendet und der Input ist die Anfrage. Dadurch wird effizient nach einer (oder mehreren) passenden Views gesucht.

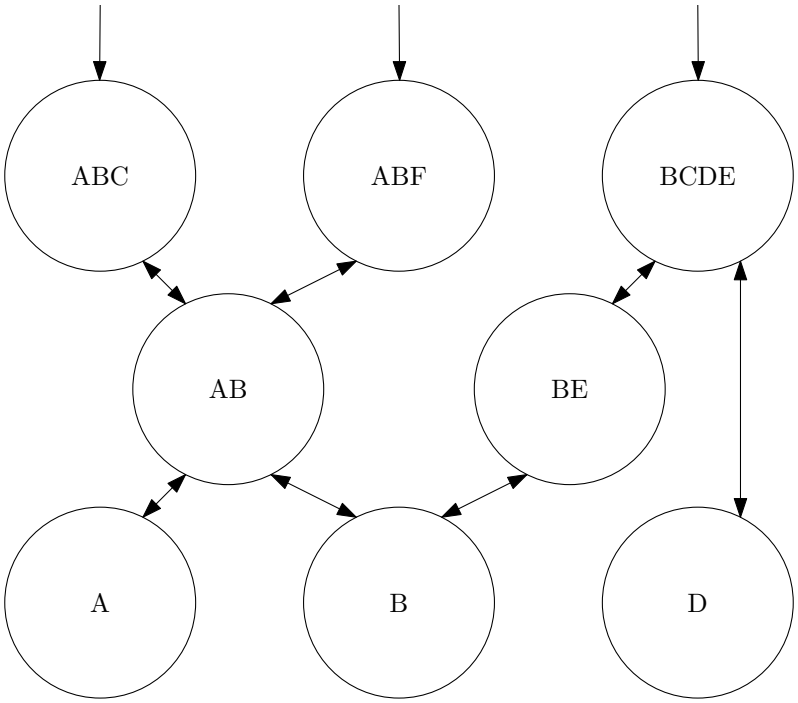


Figure 4: Beispiel eines Lattice Index

9 What experiments have been done and what do the results tell us about the performance and the scalability of the approach introduced by this paper?

Die Graphen wurden aus dem Paper entnommen.

In Figure 5 wird dargestellt wie sich die Optimierungszeit von Filter-Trees und den gefunden Alternativen verhält. Es ist klar zu erkennen das die Optimierungszeit mit Filter-Trees sinkt. Am längsten dauert es falls keine Filter Trees verwendet werden und Alternativen gesucht werden. Die Alternativen werden durch lineare Suche ermittelt wodurch natürlich die Suchzeit bei mehreren Views stark ansteigt.

In Figure 6 wird dargestellt, wie sich die Optimierungszeit im Vergleich zum View-Matching verhält. Die beiden Graphen verhalten sich linear zueinander und ungefähr die Hälfte der Optimierungszeit wird für das suchen von passenden Views verbraucht.

In Figure 7 wird gezeigt das der Optimizer auch tatsächlich die vorher ermittelten Views verwendet um die Anfrage zu beantworten. Zu erkennen ist das bereits bei 200 Views, ca. 60 Prozent der Anfragen mit Views beantwortet werden. Bei 1000 Views sind es bereits 87 Prozent (lt. Paper).

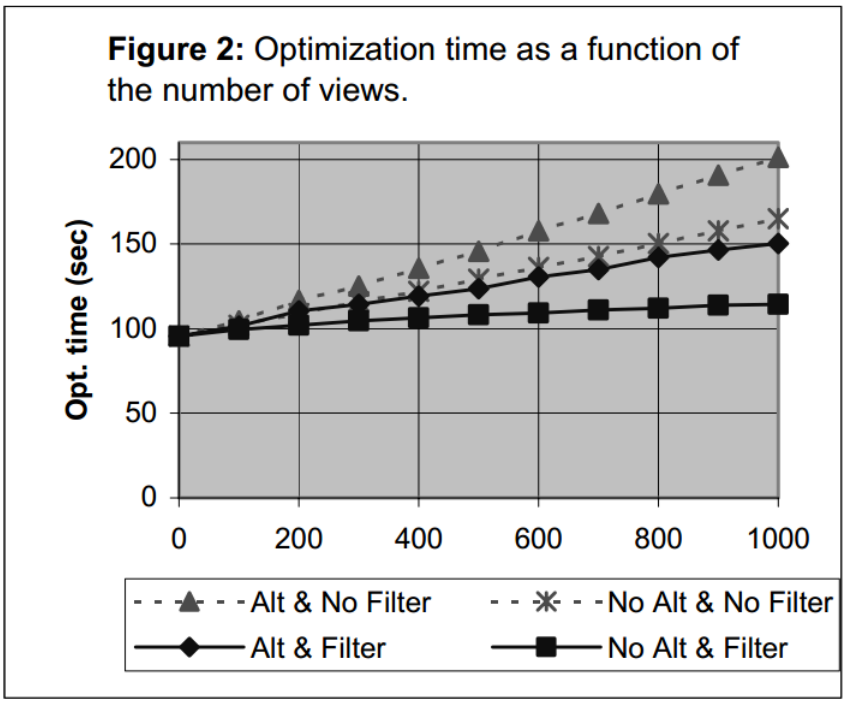


Figure 5

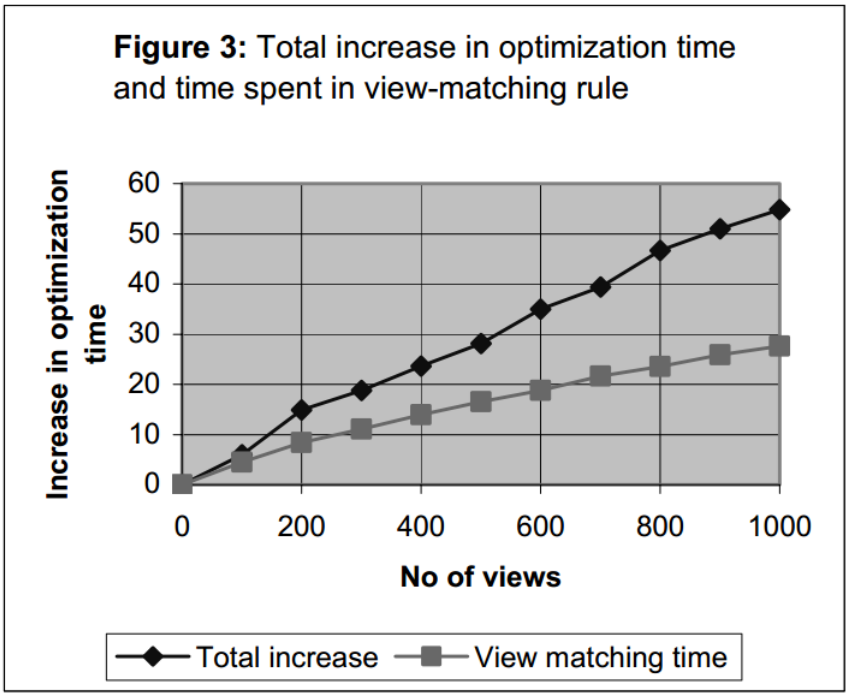


Figure 6

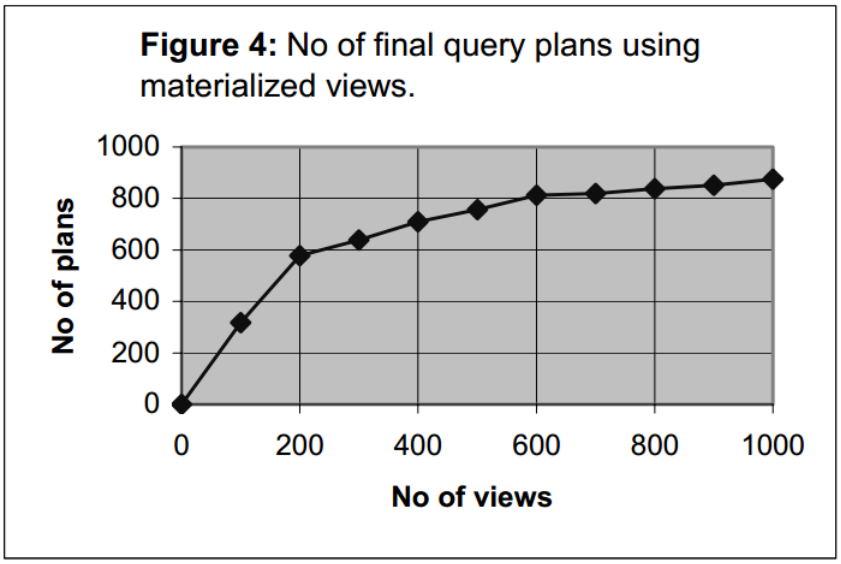


Figure 4: No of final query plans using materialized views.
No of plans
1000
800
600
400
200
0
0
200
400
600
800
1000
No of views

Figure 7

Diskussionsprotokoll von *Optimizing queries using materialized views: a practical, scalable solution* von Jonathan Goldstein and Per-Åke Larson

Note Taker: Alexander Miller

Perlen der Datenbankliteratur

Fachbereich für Computerwissenschaften

Universität Salzburg

15. Mai 2015

1

Frage

Was ist das view-matching Problem, welches in der Arbeit Beachtung findet? Was sagt es aus?

Antwort

Die Subexpressions der Query, die es zu beantworten gilt, mit den Views abzugleichen. Es wird quasi ein sehr informiertes Caching benutzt: Das Ergebnis wird gelesen anstatt berechnet.

2

Frage

Struktur des Papers

Antwort

Das Paper beschreibt zwei Teile:

1. Algorithmus (Wie darf ich die Views verwenden?). Beschreibt die Anforderungen an die Views um Subexpressions zu beantworten: Columns, Tupel, Aggregation.

2. Fast access to views (Wie finde ich die Views schnell?)

3

Frage

How does the decision algorithm (to determine if all required rows exist in a view) work in principle and what are the three tests applied?

Antwort

Der Algorithmus umfasst drei Tests:

1. Equijoin subsumption test: Sind alle Spalten der Query in der View? Äquivalenzprädikate in Query ⇒ Äquivalenzprädikate in View. Umgesetzt mit Äquivalenzklassen.

2. Range subsumption test: Werden die Grenzen der Query nicht durch die Grenzen der View eingeschränkt?

3. Residual subsumption test: Weitere Subexpressions werden auf Stringgleichheit überprüft. Evtl. wird das Prädikat auf die View angewandt.

Ein Beispiel wird im Paper unter 3.1.2 in Example 2 angeführt.

4

Frage

In der Arbeit werden die drei verschiedenen Tests besprochen, um zu zeigen, das alle Reihen die von der Query bentigt werden in der View enthalten

sind. Welche waren die drei Tests? Warum vereinfacht die Abwesenheit von Disjunktionen bei einem der Tests den Algorithmus bzw. warum knnte es den Algorithmus komplexer machen? (Betrifft konkret den Range subsumption Test und die Aussage, dass es ohne Disjunktionen einfacher ist. Leider geht nicht hervor, was das Problem hierbei war.)

Antwort

Die Autoren haben sich aus Platzgründen entschieden, die Beschreibung fr disjunkte Ranges weg zu lassen. Der Algorithmus wird komplizierter, da bei konjunktiven Ranges stets nur ein Interval überprft werden muss, bei disjunktiven Ranges steigt die Zahl der Intervalle jedoch. Bei der Besprechung des Papers wurde über Lösungsmöglichkeit diskutiert: Es wäre zB möglich beim Anlegen einer View auf ORs zu prüfen und Teilviews zu erzeugen. Das führt aber zu Problemen beim Zusammenfügen der Teilviews. Wenn die Informationen im Filter Tree gespeichert werden sollen, steigt dessen Platzbedarf.

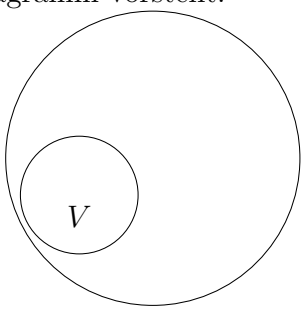
5

Frage

Zeigen Sie dass $W_q => W_v$ gilt, falls das Ergebnis von SELECT Anweisung SELECT * FROM T1,T2,...,Tn WHERE Wq erzeugt eine Untermenge des SELECT * FROM T1,T2,...,Tn WHERE Wv Anweisung

Antwort

Die Antwort folgt direkt, wenn man sich die Menge der Tupel in einem Venn-Diagramm vorstellt:



6

Frage

Welche Bedingungen muss ein View erfllen, um Aggregat Query von View zu berechnen?

Antwort

Die Antwort in 3.3 beschrieben: Eine Aggregation Query kann aus einer View berechnet werden, wenn sie folgende Anforderungen erfüllt:

- Der SPJ-Teil der View produziert alle Rows die vom SPJ-Teil der Query gebraucht werden und im richtigen Duplication factor.
- Alle Spalten die vo nden Compensating predicates gebraucht werden sind in der Ausgabe der View
- Die View enthält keine Aggregation oder ist weniger aggregiert als die Query.
- Alle Spalten die zur Gruppierung benötigt werden sind in der View enthalten
- Alle Spalten die zur Berechnung von Output expression benötigt werden sind in der View enthalten.

7

Frage

Was ist ein filter tree?

Antwort

Sektion 4 beschreibt die Datenstruktur: Der Filter Tree ist ein In-Memory Index auf Views. Ein Knoten im Tree besteht aus (Key, Pointer)-Paaren, wobei ein Key eine Menge von Werten ist. Pointer in Blattknoten zeigen auf Listen von View-Deskriptoren. Bei der Suche im Baum werden die Keys in den Knoten mit einem Lattice Index auf Existenz des Suchschlüssel überprüft.

Ein Filter Tree unterteilt die Menge an Views in von Level zu Level kleinere disjunkte Partition. In jedem Level wird eine andere Bedingung zur Partitionierung verwendet (siehe Frage 9).

8

Frage

Why is it not sufficient to just keep view information in the memory to speed up view matching? Describe briefly the data structure used to solve this problem

Antwort

Die sequentielle Suche kann bei vielen Views sehr lange dauern. Mit dem Filter Tree soll die Suche beschleunigt werden. Eine Beschreibung des Trees ist bei den Fragen 7 und 9 zu lesen.

9

Frage

In der Arbeit von Goldstein und Larson wird das Fast Filtering von Views mittels filter trees und lattice index vorgestellt. Fr die spteren Experimente und deren Auswertung haben sie einen Baum verwendet, welcher 8 Levels/Ebenen hat und sich wie folgt aus diesen Partitioning Conditions zusammensetzt:" From top to bottom, the levels are: hubs, source tables, output expressions, output columns, residual constraints, and range constraints. For aggregation views, there are two additional levels: grouping expressions and grouping columns." Dieser Filter Tree wirkt sich natrlich in seiner Zusammensetzung auf die Ergebnisse aus, ist dies der am ehesten optimale Baum oder gibt es womgliche bessere Anordnungen? Oder folgt der Aufbau logisch aus den verschiedenen Partitioning Conditions? Wieso kommen dann hubs vor source tables? Warum immer expressions vor columns?

Antwort

Die 8 Levels im Filter Tree werden von folgenden Bedingungen bestimmt:

1. Hub condition: Extra tables werden durch den Algorithmus von 3.2 eliminiert, der Rest wird Hub genannt.
2. Source table condition: Die Tables, aus denen die View errechnet wurde.
3. Output expression condition
4. Output column condition: Die View muss alle output columns der Query enthalten
5. Residual predicate condition: Die Residual predicates der View müssen eine Teilmenge der predicates der Query sein.
6. Range constrained columns: Die View darf nicht mehr eingeschränkt sein als die Query
7. Grouping expression condition
8. Grouping columns: Aus den Spalten der View muss eine Gruppierung der Query möglich sein

Die Reihenfolge wurde so gewählt, dass die stärkste Partitionierungen oben sind.

10

Frage

What experiments have been done and what do the results tell us about the performance and the scalability of the approach introduced by this paper?

Antwort

Die Experimente zeigen eine Verkürzung der Optimierungszeit. Schon einer Verwendung des Filters ohne Substitutes kann die Optimierungszeit verkürzen, da mit guten Pfaden sehr früh viele andere Pfade geprunet werden können.