PD Dr. rer. nat. habil. Sven Groppe

Übungen zur Vorlesung

Semantic Web

WS 2013/2014

Übung 8 – Anfrageverarbeitung

Aufgabe 1:

Widerlegt die folgenden Gleichungen:

$\Pi_m(R1 \cap R2) = \Pi_m(R1) \cap \Pi_m(R2)$

$\Pi_m(R1 - R2) = \Pi_m(R1) - \Pi_m(R2)$

Aufgabe 2:

In dieser Aufgabe soll der *Nested-Loop-Join* zweier Relationen *R* und *S* näher betrachtet werden.

a) Entwickle eine Formel, die die Anzahl der Seitenzugriffe bei einer einfachen Nested-Loop-Join-Implementierung (ohne Optimierung wie z.B. Wiederverwendung von gelesenen Seiten der inneren Schleife) repräsentiert. Benutze dabei die folgenden Variablen:

   b_R: Anzahl Seiten der Relation R
   b_S: Anzahl Seiten der Relation S
   m: Anzahl Seiten, die im Hauptspeicher passen
   k: Anzahl Seiten des Hauptspeichers, die für die innere Schleife benutzt werden, also $1 \leq k \leq m-1$

b) Wie viele Seitenzugriffe werden bei den folgenden Beispieldaten zur Berechnung eines Joins von A und B nötig? Welches k sollte dabei gewählt werden und welche Relation sollte für die äußere und welche für die innere Schleife benutzt werden (also $A \bowtie B$ oder $B \bowtie A$)?

   - Die Relation A enthält 800.000 Tupel, die Relation B enthält 3.000.000 Tupel.
   - Tupel von Relation A haben eine Größe von 100 Bytes und Tupel von Relation B eine Größe von 50 Bytes.
   - Eine Seite umfasst 8 KBytes. Ein einzelnes Tupel muss immer auf eine Seite, d.h. pro Seite bleibt evtl. ein kleiner Rest unbenutzt.
   - Der für den Join zur Verfügung stehende Hauptspeicher hat eine Größe von 2 MByte.

Berechne zuerst die Parameter b_A, b_B und benutze dann die Formel aus Teil a).

Aufgabe 3:

Betrachte nun den *Merge-Join*, der davon ausgeht, dass die beidem Relationen bereits nach den zu verbindenden Attributen sortiert sind. Wie viele Seitenzugriffe sind dann im Beispiel von Aufgabe 3 nötig, falls wir von dem optimalen Fall ausgehen, dass es keine doppelten Werte gibt?

Aufgabe 4:

In dieser Aufgabe soll der Hash-Join näher untersucht werden. Der Hash-Join versucht, einen ähnlichen Vorteil wie den des Merge-Joins gegenüber dem Nested-Loop-Join zu erreichen ohne den gesamten zusätzlichen Aufwand des Sortierens leisten zu müssen.

Die Grundidee ist dabei, die Relationen mit Hilfe von Hash-Funktionen in kleine Partitionen zu zerteilen. Bei dem anschließenden Join muss dann nicht jedes Tupel mit jedem verglichen werden, sondern nur die Tupel in den jeweils entsprechenden Partitionen der beiden Relationen müssen verglichen werden. Indem zumindest einen der beiden zu vergleichenden Partitionen so klein gemacht wird, dass sie in *m-1* von *m* zur Verfügung stehenden Hauptspeicherseiten passt, kann man den Vergleich zweier Partitionen so realisieren, dass dazu jede Partition insgesamt nur einmal vom Sekundärspeicher geladen werden muss.

Der Hash-Join läuft nach dem folgenden Schema ab:

Zuerst wird die kleinere der beiden Relationen partitioniert. Das Partitionieren geschieht in Runden, wobei in jeder Runde aus jeder bestehenden Partition *m-1* entsprechend kleinere Partitionen entstehen. Das Partitionieren einer bestehenden Partition wird ausgeführt, indem eine der m Hauptspeicherseiten jeweils einer neuen Partition entsprechen. Die Tupel der bestehenden Partition werden durch eine Hash-Funktion einer der *m-1* neuen Partitionen zugewiesen und in die entsprechende Hauptspeicherseite kopiert. Läuft eine Hauptspeicherseite leer bzw. voll, wird die nächste Seite nachgeladen bzw. wird die Seite auf den Sekundärspeicher geschrieben. Insgesamt werden die Runden solange wiederholt, bis jede der entstandenen Partitionen in *m-1* Hauptspeicherseiten passt.

- Anschließend wird die zweite Relation partitioniert. Dies geschieht nach demselben Schema unter Benutzung derselben Hash-Funktion. Es werden genauso viele Runden gemacht wie bei der Partitionierung der ersten Relation.
- Schließlich kommt die Join-Phase. Dazu werden die beiden sich entsprechenden Partitionen beider Relationen verglichen. Da eine der beiden Partitionen in *m-1* Hauptspeicherseiten passt, kann die verbliebene Hauptspeicherseite für die andere Partition benutzt werden.

Zur Analyse des Hash-Joins beantworten Sie die folgenden Fragen:

a) Sei x_i die Größe (Anzahl Seiten) einer Partition nach i Partitionierungs-Runden. Wie groß ist x_i bei m Hauptspeicherseiten und einer Relation R mit b_R Seiten? (Dabei soll hier von einer *perfekten* Hash-Funktion ausgegangen werden, d.h. jede der entstehenden Partitionen ist gleich groß. Diese Perfektion wird in der Praxis nie erreicht werden. Aber bei hinreichend großen Relationen und einer guten Streuung des Hash-Attributs ist die Abweichung von der perfekten Hash-Funktion sehr klein.)
b) Wie viele Partitionsrunden sind dann nötig?
c) Wie viele Sekundärspeicherseiten müssen pro Partitionierungsrunde gelesen oder geschrieben werden? Wie groß ist dann der Gesamtaufwand für das Partitionieren einer Relation?
d) Wie groß ist der Gesamtaufwand für das Hash-Join zweier Relationen R und S mit b_R bzw. b_S Seiten und m Hauptspeicherseiten?

Aufgabe 5:

Verwende den hauptspeicherbasierten Indexierungsansatz aus der Vorlesung für Semantic Web Daten, um den Join zwischen den Tripelmustern *(?Z, rdf:type, ub:Course)*, *(?Y, ub:teacherOf, ?Z)* und *(?X, ub:advisor, ?Y)* zu berechnen, wenn der po-Index wie folgt gegeben ist:

Schlüssel *po*	**Tripel**
rdf:type ub:Course	*{(Course1, rdf:type, ub:Course), (Course2, rdf:type, ub:Course), (Course3, rdf:type, ub:Course)}*
ub:teacherOf Course1	*{(Teacher1, ub:teacherOf, Course1)}*
ub:teacherOf Course3	*{(Teacher2, ub:teacherOf, Course 3), (Teacher3, ub:teacherOf, Course3)}*
ub:advisor Teacher1	*{(Student1, ub:advisor, Teacher1), (Student2, ub:advisor, Teacher1)}*
ub:advisor Teacher3	*{(Student3, ub:advisor, Teacher 3), (Student4, ub:advisor, Teacher 3), (Student5, ub:advisor, Teacher 3), (Student6, ub:advisor, Teacher3)}*

Aufgabe 6:

Füge die fehlenden Informationen in dem folgenden Histogramm-Index hinzu und berechne das Histogramm mit zwei Intervallen für die Variable *?v* und dem Schlüssel (5, *?v*, *?o*):