Seminar : Algorithmen für große Datenmengen

Thorsten Siffrin

5. August 2001

Projektaufgabe

Funktionsweise von relationalen Datenbanken

- Machen Sie sich mit den Grundlagen relationaler Datenbanken vertraut.
- Skizzieren Sie, wie die wichtigsten Operationen eines relationalen Datenbanksystems I/O effizient implementiert werden können.
  - Wie funktionieren Join-Operationen?
  - Wozu braucht man Indexstrukturen?
  - Welche Rolle spielt Sortieren?

1 Speicherungsstrukturen von Datensätzen

1.1 Abbildung von Datensätzen

Datensätze in relationalen Datenbanken werden in Blöcke(Seiten) fester Größe gespeichert. Ein Block bezeichnet die kleinste Einheit für den Transport zwischen Sekundärspeicher und Hauptspeicher. Eine Seite beginnt mit einem Header, der Angaben zur Größe des gesamten freien Platzes im zugehörigen Block, zur Größe des zusammenhangenden freien Platzes zwischen Datensätzen und Slot-Array, sowie zur Größe des Slot-Arrays enthält. Das Slot-Array (Row Offset Table) befindet sich am Ende einer Seite. Für jeden Datensatz ist hier die Startadresse auf der Seite als Offset-Wert (bezogen auf den Seitenanfang) angegeben. Der durch Datensätze belegte Bereich in einer Seite wächst von vorne nach hinten, das Slot-Array von hinten nach vorne (Überblick dazu in [2],[4]).

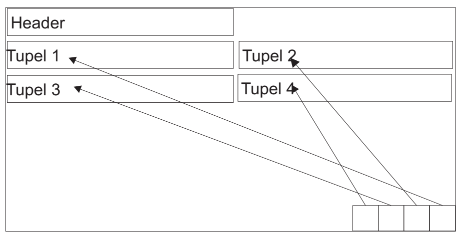

Abbildung 1: Abbildung von Datensätzen

1.2 Abbildung von Seiten auf Externspeicher

Eine oder mehrere Tabellen und/oder Indizes werden in einem sog. Tablespace gespeichert. Tablespace bezeichnet einen logisch zusammenhängenden Seitenadreßraum. Logische Seitennummern werden mit Hilfe einer Seitenadressierungstabelle (Extent-Tabelle) in physische Blocknummern auf die Platte übersetzt, die idealerweise im internen Speicher gehalten werden. In diesem Falle werden keine extra I/O-Operationen beim Zugriff benötigt (Überblick dazu in [2],[4]).

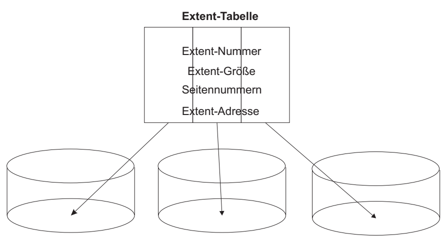

Abbildung 2: Abbildung von Seiten auf Externspeicher

2 Indexstrukturen

Innerhalb von Relationen wird, falls erwünscht, ein schneller Zugriff mittels Indizes erreicht. Ein Index ist eine Gruppierung von Tupeln einer Relation in Bezug auf die verschiedenen vorkommendn Werte eines Attributs oder der Kombination mehrerer Attribute (Attributgruppe).
Zur Realisierung einer solchen Indexstruktur, verwendet man B*-Bäume (siehe auch hierzu [2],[6]).

Ein Mehrwegbaum heißt B*-Baum der Ordnung $(m, m^*)(m \geq 1, m^* \geq 1)$, wenn gilt :

1. Jeder Nichtblattknoten außer der Wurzel enthält mindestens m und höchstens 2m Schlüssel.
2. Ein Nichtblattknoten mit k Schlüsseln $x_1, \ldots, x_k$ hat genau k+1 Söhne $t_1, \ldots, t_k$, so daß
   - für alle Schlüssel s im Teilbaum $t_i (2 \leq i \leq k)$ gilt $x_{i-1} < s \leq x_i$ und
   - für alle Schlüssel s im Teilbaum t_l gilt $s < x_l$
   - für alle Schlüssel s im Teilbaum t_{k+1} gilt $x_k < s$.
3. Alle Blätter haben dasselbe Niveau.
4. Jedes Blatt enthält mindestens m* und höchstens 2m* Schlüssel.

Jeder Eintrag eines Blatts seinerseits hat einen Verweis auf eine sog. TID-Liste, die alle Tupel der Relation mit dem passenden Suchschlüssel enthält. Die TID-Liste sollte nach Seitennummern sortiert sein.
Die Operationen Search, Insert und Delete entsprechen den Operationen, wie sie im Seminar vorgestellt wurden.

Die Speicher- und Zugriffskosten eines B*-Baum-Index hängt von mehreren Faktoren ab. Diese sind:

1. Größe eines Blocks (Seite)
2. Länge eines (Such-)Schlüssels
3. Länge eines Sohnzeigers innerhalb des Baums
4. Länge einer Tupel-ID (TID)

Je nach Wahl der Faktoren 1-3 ergibt sich die Größe und Füllung der Nichblattknoten. Dies wiederum beeinflußt die Höhe und den Fanout des Suchbaumes. Die Faktoren 1 und 4 bestimmen die Anzahl der I/O's auf Blattebene, um die TID-Liste eines Suchschlüssels einzulesen. Entweder sind mehrere TID-Listen in einem Block gespeichert, womit ein I/O zum Einlesen reicht oder eine TID-Liste ist in mehreren Blöcken gespeichert, was entsprechend mehr I/O's benötigt.

Der Vorteil von Indexstrukturen zeigt sich im Vergleich zu einem Relationen-Scan (Sequentielles Lesen aller Tupel einer Relation). Wenn man davon ausgeht, daß die Größe der TID-Listen im allgemeinen bedeutend kleiner als die ganze Relation ist, hat man bei der Verwendung von Indexstrukturen bedeutend weniger I/O-Zugriffe, als wenn man jeden einzelnen Tupel der Relation einlesen muß.

3 Elementare Zugriffsstrukturen

Die elementaren Zugriffsstrukturen in relationalen Datenbanksystemen sind SELECT, INSERT, DELETE UND UPDATE (siehe [4]). Neben diesen Strukturen hat jedes Datenbanksystem einen Optimierer, der entscheidet welche Operationen bei der Ausführung angewendet werden müssen. Bei jeder Abfrage, wird überprüft, ob auf einem Attribut ein Index liegt und mit Hilfe eines B*-Baums, die Ausführung der Operation beschleunigt werden kann. SELECT ist die Anfrageoperation und hat folgende Syntax :
SELECT Projektion($a_1, ..., a_n$) **FROM** Tabelle(n) **WHERE** Filterbedingung(en)
Zunächst werden die einzelnen Tupel der Tabelle(n) in den internen Speicher geladen. Diese werden dann entsprechend der WHERE - Klausel gefiltert und die gesuchten Attribute a_1 bis a_n ausgegeben. Die Zugriffskosten einer solchen Operation belaufen sich im schlechtesten Fall auf einen kompletten Relationen-Scan, wenn kein Index auf der Relation besteht. Die Anzahl der I/O's ist damit die Anzahl der Seiten, in denen alle Tupel der Relation gespeichert sind. Gibt es jedoch eine Indexstruktur für diese Relation, so kann die Anzahl der I/O-Zugriffe stark verringert werden.
Die Einfügeoperation ist INSERT :
INSERT INTO Tabelle(n) **VALUES** Tupel
Im internen Speicher sammelt man mehrere Einfügeoperationen, bis man eine Seite voll hat. Diese wird dann in einem I/O in den externen Speicher geschrieben. Zusätzlich ist gegebenenfalls eine Einfügeoperation in einen Indexbaum notwendig. Amortisiert gesehen sind die I/O-Kosten für das Schreiben $O(\frac{1}{B})$. Das Einfügen in den B*-Baum ist aber logarithmisch ($O(\log_B \frac{N}{B})$), womit auch INSERT INTO logarithmisch ist.
In relationalen Datenbanken wird mit der Operation DELETE gelöscht.
DELETE FROM Tabelle **WHERE** Filterbedingung(en)
Wie bei SELECT muß durch den Optimierer des Datenbanksystems überprüft werden, welche Form des Einlesens verwendet werden kann. Also ob man per Index auf die Tupel zugreifen kann oder einen Relationen-Scan durchführen muß. Hat man die Tupel in den entsprechenden Seiten gelöscht, so müssen diese Seiten wieder zurück in den Externspeicher geschrieben werden. Im schlechtesten Fall, muß man alle gelesenen Seiten wieder zurückschreiben. Sonst können teilweise leere Seiten kombiniert werden, womit der Schreibzugriff der Löschoperation deutlich verringert werden kann.
Mit UPDATE werden bestimmte Tupel und ihre Attribute auf neue Werte gesetzt.
UPDATE Tabelle **SET** Wert(e) **WHERE** Filterbedingung(en)

Wie bei der Operation DELETE müssen alle entsprechenden Tupel geladen werden und auf den neuen Wert gesetzt werden. Im Vergleich zu DELETE müssen bei dieser Operation alle Seiten wieder zurückgeschrieben werden.

4 Joins in relationalen Datenbanken

Joins sind Verbindungen zweier Relationen über gleiche Attributnamen und gleiche Attributwerte der Tupel (siehe hierzu auch [2], Abbildung 3).

Relation Person P : KNr, Name, Vorname
Relation Bestellung B : BNr, KNr, Artikel
Select P.Name, P.Vorname, B.Artikel
From Person P, Bestellung B
Where P.KNr = B.KNr

Abbildung 3: Join zwischen 2 Relationen

Die einfachste Form eines Joins zwischen zwei Relationen ist der Nested-Loop Join, bei dem mittels einer Doppelschleife zwischen den zwei Tupelmengen der Join berechnet wird.

Nested-Loop-Join
  Festlegung der äußeren(**outer**) und der inneren(**inner**) Relation
  for each Tupel $r \in$ outer
    for each Tupel $s \in$ inner
      if r.Joinattribut = s.Joinattribut
      then Füge Tupel (r, s) in die Resultatmenge

Der Nested-Loop-Join ist immer anwendbar, jedoch ist dabei stets das vollständige Durchsuchen des gesamten Suchraums in Kauf zu nehmen. Kann man auf ein Joinattribut per Index zugreifen, so kann die Laufzeit reduziert werden, da nur noch eine Relation gescannt werden muß. Diese Form des Joins wird durch den Nested-Loop-Join mit Indexzugriff realisiert.

Nested-Loop-Join mit Indexzugriff
  Festlegung der äußeren(**outer**) und der inneren(**inner**) Relation
  for each Tupel $r \in$ outer
    Finde durch Index-Scan die TIDs aller Tupel s mit s.Joinattribut = r.Joinattribut
    for each TID
      Füge Tupel (r, s) in die Resultatmenge

Will man nun aber lineare Laufzeiten für einen Join realisieren, so müssen andere Algorithmen angewendet werden. Einer dieser Algorithmen ist Merge-Join. Bei vorliegender Sortierung der Tupel nach den Joinattributen erhält man lineare Laufzeit. Ist die Sortierung nicht gegeben, müssen vorab beide Relationen sortiert werden, wodurch der Laufzeitvorteil in erheblichem Maße reduziert wird. Der Merge-Join-Algorithmus hat folgendes Aussehen:

MERGE-JOIN

Festlegung der äußeren(**outer**) und der inneren(**inner**) Relation
r ist erstes Tupel von outer (nach Sortierung von outer)
s ist erstes Tupel von inner (nach Sortierung von inner)
while in outer oder inner noch Tupel zu bearbeiten sind
  if r.Joinattribut = s.Joinattribut then
    s' = s
    while r.Joinattribut = s.Joinattribut
      Füge (r, s) in die Resultatmenge ein
      s = nächstes Tupel von inner
    s = s'
    r = nächstes Tupel von outer
  else if r.Joinattribut < s.Joinattribut
    then r = nächstes Tupel von outer
    else s = nächstes Tupel von inner

Ein weiterer Join-Algorithmus, der in linearer Laufzeit realisiert werden kann, ist der Hash-Join. Die Berechnung erfolgt durch Abbilden beider Mengen auf eine Hash-Tabelle, so daß Tupel mit gleichem Attributwert auf denselben Eintrag(Bucket) der Hash-Tabelle abgebildet werden. Um ihn jedoch zu realisieren wird vorrausgesetzt, daß die kleinere(innere) Relation vollständig in den internen Hauptspeicher paßt. Der Algorithmus wird folgendermaßen implementiert:

HASH-JOIN

Festlegung der äußeren(**outer) und der inneren(inner) Relation**
Allokation der Hashtabelle
Sei h eine Hashfunktion
for each Tupel $s \in$ inner
  Speichere s in Bucket h(s.Joinattribut)
for each Tupel $r \in outer$
  Sei Bucket B = h(r.Joinattribut)
    for each Bucketelement $b \in B$
      if r.Joinattribut = b.Joinattribut
        then Füge Tupel (r, b) in Resultatmenge ein

Paßt die kleinere Relation nun aber nicht in den Hauptspeicher, so muß man den Hash-Join Algorithmus ein wenig abwandeln. Man führt einfach eine Art Nested-Block-Hash-Join aus. Man partitioniert die kleinere Relation so, daß die einzelnen Teile in den internen Hauptspeicher passen und so auf jede Partition ein Hash-Join ausgeführt werden kann. Sei M die Größe des internen Hauptspeichers und P die Größe der kleineren Relation, dann müssen $\lceil \frac{P}{M} \rceil$ Hash-Joins ausgeführt werden.

5 Sortieren in relationalen Datenbanken

Sortieren ist dann notwendig, wenn man die Tupelmenge nach einem Attribut bzw. einer Attributkombination ordnen will. Dies ist bei Operationen, wie Durchschnitt, Vereinigung und Differenz notwendig. Bevor diese Mengenoperationen von den Query-Optimierern der Datenbanksystemen ausgeführt werden, wird zunächst eine Sortierung auf die Tupelmenge vorgenommen um die Laufzeiten der Operationen zu optimieren. Vor allem wird Sortieren aber bei Merge-Join und bei Anfragen verwendet, die eine Gruppierung oder eine wirkliche Sortierung verlangen. Anfragen der Form

SELECT Attribute $A_{11}, ..., A_{1n}, ..., A_{21}, ..., A_{2n}, ..., A_{mn}$
FROM Tabelle(n) $A_1, ..., A_m$
WHERE Filterbedingung(en)
ORDER BY Attribut A_{xy}

sortieren die Ergebnisse nach dem Wert A_{xy}. Die Operation **ORDER BY** stellt damit die eigentliche Sortierungsmöglichkeit bereit. Dem gegenüber stehen Anfragen, die zur Gruppierung von Datensätzen benötigt werden. Eine solche Anfrage hat die Form

SELECT Attribute $A_{11}, ..., A_{1n}, ..., A_{21}, ..., A_{2n}, ..., A_{mn}$
FROM Tabelle(n) $A_1, ..., A_m$
WHERE Filterbedingung(en)
GROUP BY Attribut A_{xy}
HAVING Filterbedingung.

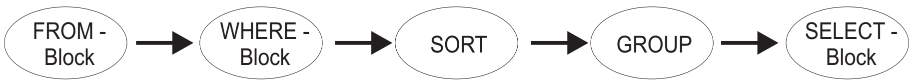

Abbildung 4: Operationen einer GROUP BY - Anfrage

Diese wird nun folgendermaßen von einem Datenbanksystem ausgewertet. Zunächst werden die Tabellen $A_1, ..., A_m$ geladen und nach den Bedingungen aus der WHERE-Klausel gefiltert. Bevor diese Tupel nun nach dem Attribut A_{xy} gruppiert werden, werden sie zunächst sortiert (siehe auch Abbildung 3). Sollte bei all diesen Operationen eine Sortierung auf ein Attribut vorgenommen werden, welches indiziert ist so müssen einfach nur alle Blätterelemente des B*-Baums von links nach rechts ausgegeben werden. All diese Sortierungen werden mit Algorithem für den Externspeicher ausgeführt. Diese externe Sortieralgorithmen arbeiten, wie im Seminar bereits vorgestellt ([1],[5]), in mehrerern Phasen.

- *Bilden von sortierten Teilsequenzen (Runs)*
- *Mischen der Runs*

- *Iterieren der Mischphasen*

I/O-Kosten

S(R) bezeichne die Anzahl der Seiten (Blöcke), die für die Relation R im Externspeicher notwendig ist.

1. Bilden der Runs:
   Es werden $S(R)/M$ Runs der Größe M gebildet. Dazu werden alle Blöcke von R je einmal gelesen und geschrieben. Es sind also 2*S(R) Seitenzugriffe notwendig.

2. Mischen von jeweils M Runs der Größe M:
   Es werden $S(R)/M^2$ Runs der Größe M^2 gebildet. Dies erfordert wiederum $2 * S(R)$ Seitenzugriffe.

3. Iterieren der Mischphasen, bis die Anzahl der Runs gleich 1 ist, also $S(R)/M^i \leq 1$ ist, also $i \geq \log_M S(R)$ gilt. Bis dahin sind $i-1$ Mischphasen notwendig.

Damit betragen die Gesamtkosten $(\lceil(\log_M S(R))\rceil - 1) * 2 * S(R) + 2 * S(R)$ Seitenzugriffe.

Literatur

[1] Allgemeines zu External Memory : J. S. Vitter. External Memory Algorithms. In 6th European Symposium on Algorithms, number 1461 in LNCS, pages 1-25. Springer, 1998.

[2] G.Weikum. Vorlesung Datenbanken WS 1999/2000. http://www-dbs.cs.uni-sb.de.

[3] J.Melton, Alan R. Simon. Understanding The New SQL: A complete guide. Morgan Kaufmann Publishers. Seite 43-60, 1992

[4] Patrick O'Neil. Database-Principles, Programming, Performance. Morgan Kaufmann Publishers, 1994

[5] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. Communications of the ACM, 31(9):1116-1127, 1988.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to Algorithms. McGraw-Hill, 1990.