

4. Physische Operatoren einer Algebra

- ❑ In diesem Abschnitt betrachten wir Operatoren und ihre Implementierungen, die auf einer Relation bzw. zwei Relationen arbeiten.
- ❑ Aller Operatoren erfüllen die sogenannte ONC-Schnittstelle eines Iterators:
 - **Open:** Öffnet den Iterator
 - **Next:** Gibt das nächste Element
 - **Close:** Schließt den Iterator

Diese Schnittstelle wird insbesondere bei der Umsetzung logischer in physischer Operatoren sehr wichtig sein.
- ❑ Naive Implementierung der Operatoren durch
 - Relationen-Scan und
 - Index-Scan

⇒ hohe Laufzeiten

73.

Einfache Beispiele für ONC

- ❑ Erste Implementierungen für Projektion und Selektion erfüllen in einfacher Weise die ONC Schnittstelle.
- ❑ Projektion: $\pi_X(R)$

```
next() {
    rec = R.next();
    return new Record(rec, X);
}
```
- ❑ Selektion: $\sigma_F(R)$

```
next() {
    do
        rec = R.next();
    while (!F(rec));
    return rec;
}
```

In einem späteren Kapitel werden wir auf Indexstrukturen eingehen, die insbesondere Selektionen effizient unterstützen können.

75.

Anforderung an ONC

- ❑ Alle Operatoren erfüllen die Schnittstelle Open-Next-Close eines Iterators. Dies hat insbesondere folgende Vorteile:
 - Einfacher Austausch von Implementierungen
 - Keine Zwischenspeicherung von Ergebnissen
 - Unterstützung von Operatorparallelität
- ❑ Damit eine möglichst effiziente Verarbeitung gewährleistet wird, sollten die physischen Operatoren nicht blockieren, sondern erste Ergebnisse bereits liefern, auch wenn die Eingabemenge(n) nicht vollständig vorliegen.
 - Erste Ergebnisse sollten schnell erzeugt werden.
 - Die durchschnittliche Antwortzeit für ein Aufruf von *next* sollte niedrig sein.
- ❑ Beim *open* eines Operators werden die Parameter gesetzt (wie z. B. der benötigte Hauptspeicher).
- ❑ *open* wird rekursiv für die Kinderknoten im Operatorbaum aufgerufen.

74.

Bei den anderen Operatoren ist ONC nicht so einfach zu implementieren.

- ❑ Durchlaufen mehrerer Relationen (z. B. beim Join)
- ❑ Mehrmaliges Durchlaufen einer Relation (z. B. beim Gruppieren)
- ❑ Anlegen von zusätzlichen Datenstrukturen notwendig, die aufgrund ihrer Größe i. A. auf dem Plattenspeicher abgelegt werden müssen.
 - ⇒ Die Kosten bei diesen Operationen sind damit im wesentlichen durch die benötigten I/O Operationen bestimmt.

Einfache Lösungen

- ❑ Basieren auf dem Durchlaufen der Relation(en), wobei ein möglichst großer Anteil im Hauptspeicher gehalten wird, um bei einem erneuten Durchlauf I/O Operationen einzusparen.

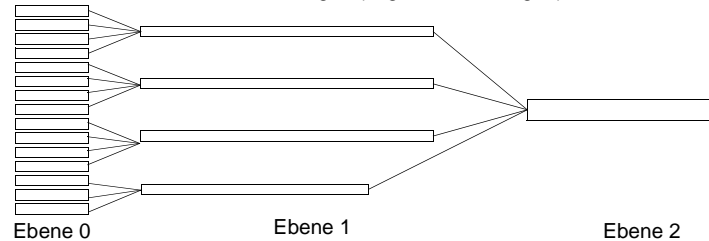
Prinzipielle Techniken für Entwurf effizienterer Lösungen

- ❑ Externes Sortieren
- ❑ Divide-and-conquer (z. B. mit Hashing)

76.

4.1 Externes Sortieren

- ❑ Sortieren ist eine der wichtigsten Operationen in einem DBS
 - Verwendung der order-by Klausel in einer SQL-Anfrage
 - Implementierungen anderer Operatoren basieren auf sortierten Relationen.
- ❑ Die zu sortierende Relation paßt i. a. nicht in den Hauptspeicher.
- ❑ Externes Sortieren wird implementiert durch **Merge-Sort**
 1. Generierung von sortierten Teilfolgen (engl.: runs).
 2. Verschmelzen der sortierten Teilfolgen (in größere Teilfolgen).



77.

Parameter externer Sortierverfahren

Bezeichner	Beschreibung	Größeneinheit
M	verfügbarer Hauptspeicher	Datensatz
N	Größe der Eingabe	Datensatz
C	Größe einer Seite	Datensatz
m	M/C	Seite
n	N/C	Seite

Die Notation aus dem letzten Kapitel wird beibehalten:

Sei R eine Relation. Dann bezeichnet

- $T(R)$ die Anzahl der Tupel,
- $B(R)$ die Anzahl der Blöcke.

I. a. gilt in diesem Kapitel $T(R) = \Theta(C \times B(R))$.

78.

4.1.1 Erzeugen der Ebene 0

Quicksort

- ❑ Algorithmus
 - Load:** Lies die nächsten M Elemente der Eingabe in den Hauptspeicher;
 - Sort:** Sortiere die Daten durch Quicksort im Hauptspeicher;
 - Store:** Gib die sortierte Folge aus;
- ❑ Eigenschaften
 - Erzeugt N/M sortierte Teilfolgen der Größe M.
 - durchschnittliche CPU Kosten: $O(N \log M)$
 - Keine überlappende Verarbeitung von Prozessor und Plattenspeicher
 - Ausnutzung von sequentiellen Zugriff beim Laden und Speichern der Daten
 - Sortierung der Eingabe reduziert i. a. nicht die Laufzeit (im Gegenteil!).
- ❑ Beispiel ($M = 3$)

Beim Sortieren der Folge 5, 12, 9, 7, 8, 14, 11, 6 werden 3 Teilfolgen erzeugt:
 $\langle 5, 9, 12 \rangle$, $\langle 7, 8, 14 \rangle$, $\langle 6, 11 \rangle$

79.

Replacement-Selection (Knuth, 1973)

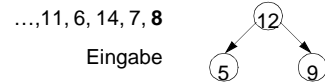
- ❑ basiert auf der Verwendung eines Heaps
- ❑ Algorithmus
 1. Man erzeuge einen Heap im Hauptspeicher und fülle diesen Heap mit den nächsten M Elementen der Eingabe;
 2. LOOP
 - Ersetze das größte Element aus dem Heap (*Max*);
 - Lies das nächste Element (*Next*) von der Eingabe;
 - Falls (*Next* > *Max*)
 - EXIT;
 - SONST
 - Füge *Next* in den Heap ein;
 3. Gib die restlichen M-1 Elemente des Heaps in sortierter Reihenfolge aus.

80.

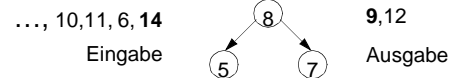
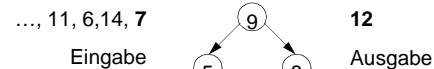
Beispiel : $M = 3$

- Sortieren der Folge 5, 12, 9, 7, 8, 14, 6, 11, 10, ...

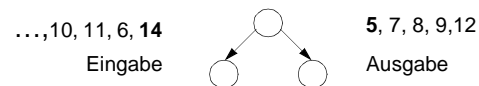
1. Schritt:



2. Schritt:



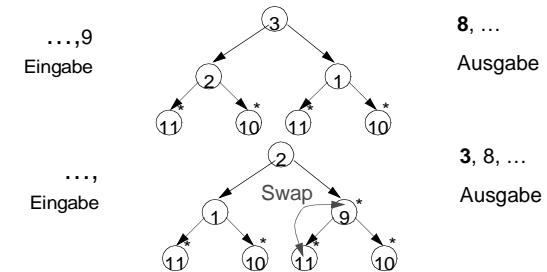
3. Schritt:



81.

Verfeinerung des 1. und des 3. Schritts

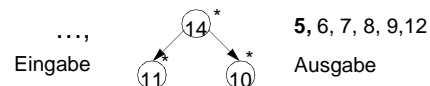
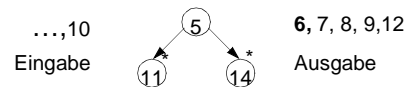
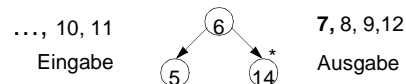
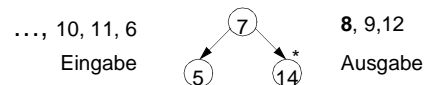
- Die Ausgabe der Elemente wird mit dem Aufbau des Heaps für die nächste sortierte Teilfolge kombiniert werden.
- Falls ein neues Element aus der Eingabe größer ist als das kleinste Element in der Ausgabe, wird es der neuen Teilfolge zugeordnet (und mit einem * markiert). Andernfalls wird das Element noch der alten Teilfolge zugeordnet.
- Die mit * gekennzeichneten Elemente werden am Ende des aktuellen Heaps abgespeichert. Beim Einfügen eines solchen Elements wird dann der nächste Heap bereits (bottom-up) aufgebaut.



82.

Beispiel:

3. Schritt:



- Nach diesem Schritt wird die nächste Teilfolge erzeugt.
- Man beachte: die Aufbauphase des nächsten Heaps ist bereits abgeschlossen.

83.

Eigenschaften von Replacement-Selection Verfahren

- Die sortierten Teilfolgen sind i. a. wesentlich länger als M .
 - Es kann gezeigt werden, daß bei Gleichverteilung der Daten, der Erwartungswert für die Länge einer Teilfolge $2M$ beträgt.
 - Sind die Daten bereits sortiert, so wird genau eine sortierte Teilfolge der Ebene 0 erzeugt.
- Durch die Heap-Datenstruktur wird garantiert, daß M Datensätze in $O(M \log M)$ Zeit im Hauptspeicher sortiert werden.
- Die Ein- und Ausgabe erfolgt **seitenweise**, wobei alternierend jeweils eine Seite aus der Eingabe gelesen und eine Seite in die Ausgabe geschrieben wird.
- Werden jeweils zwei Pufferseiten für die Ein- und Ausgabe verwendet, ist eine **überlappende** Verarbeitung zwischen Prozessor und Plattenspeicher möglich.
- Eingabedaten müssen aus den ursprünglichen Seiten in einen separaten Speicherbereich geschrieben werden.
 - Verwaltung des Speicherbereichs ist relativ komplex, wenn die Länge der Datensätze variabel ist.

84.

4.1.2 Verschmelzen sortierter Teilfolgen

- Beim Verschmelzen werden F Teilfolgen zu einer Teilfolge verschmolzen. Da für jede Teilfolge eine Pufferseite reserviert werden muß, gilt

$$F = \left\lfloor \frac{M}{C} \right\rfloor - 1 = \Theta(m)$$

- Die Anzahl der Ebenen beim Verschmelzen ist dann wie folgt gegeben:

$$\lceil \log_F n \rceil$$

- Der gesamte I/O Aufwand für das Verschmelzen beträgt dann also

$$O(n \log_m n)$$

Dies ist auch der asymptotische Aufwand für das gesamte Sortieren. Es kann gezeigt werden, daß dies auch die untere Schranke für externes Sortieren ist, d. h. **asymptotisch geht es nicht besser** (Aggrawal & Vitter, 1988).

85.

Speicheradaptivität

- Darunter verstehen wir, daß Algorithmen in einem DBS zur Laufzeit Speicherplatz abgeben können bzw. freien Speicherplatz zusätzlich reservieren können.
- Speicheradaptivität ist also nicht nur ein wichtiger Punkt bei Sortierverfahren!

Beispiel:

- In einem Operatorbaum einer Anfrage sind oft zur gleichen Zeit zwei oder mehrere Sortierprozesse aktiv.
 - Wie soll die Speicherzuteilung an die Sortieroperatoren erfolgen?
- In einem DBS sind verschiedene Anfragen zur gleichen Zeit aktiv.
 - Wie soll die Speicherzuteilung an die Anfragen erfolgen?
 - Was passiert, wenn eine neue Anfrage gestartet wird und kein Speicherplatz mehr verfügbar ist?

Bemerkung

- Es ist damit zu rechnen, daß kommerzielle DBMS diese Probleme nicht sehr effizient lösen. Erst in den letzten Jahren (ab 1988) wurde das Problem der Speicheradaptivität in der Forschung diskutiert.

87.

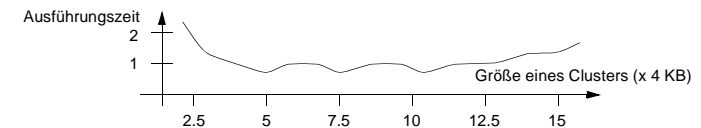
Vorschläge zur Leistungsverbesserung

Doppelte Pufferung

- Verwendung von "doppelter Pufferung", d. h. jede Teilfolge erhält 2 Pufferseiten. Dies ermöglicht eine überlappende Verarbeitung von Prozessor und Platte.
 - Nachteil ist hierbei aber, daß F halbiert wird.
 - I. a. ist es aber nicht notwendig für alle Teilfolgen eine zweite Pufferseite zu reservieren, sondern nur für die Teilfolge, die als nächste wieder eine Seite vom Plattenspeicher benötigt ("**Forecasting**").

Ausnutzung von großen Seiten

- Statt einer Seite kann eine physisch zusammenhängende Menge von Seiten, ein sogenannter **Cluster**, für jede Teilfolge betrachtet werden.
 - Nachteil: F wird kleiner
 - Vorteil: I/O-Kosten werden reduziert.



86.

ONC

- Sortieren ist (leider) ein blockierender Operator, der erst eine Ausgabe produzieren kann, wenn alle Datensätze der Eingabe vorliegen.
- Um möglichst frühzeitig erste Ergebnisse beim Sortieren zu produzieren, sollten nicht bereits bei der Open-Phase die sortierte Folge produziert, sondern der letzte Merge-Schritt im Hauptspeicher durchgeführt werden.
 - Lesen und Schreiben der Datensätze kann somit einmal eingespart,
 - Sortieroperator benötigt möglichst viel Hauptspeicher während der next-Phase.
- Im letzten Merge-Schritt sollten möglichst viele Teilfolgen verschmolzen werden (ansonsten könnte es im schlimmsten Fall passieren, dass nur zwei Teilfolgen zu verschmelzen sind).

88.

4.2 Hashing

- ❑ Wenn die Ordnung der Daten nicht zwingend notwendig ist (wie z. B. beim Natural Join), können Hashverfahren benutzt werden, um die Daten zunächst zu partitionieren und dann den Operator auf die einzelnen Partitionen anzuwenden.
- ❑ Vorteil von Hashverfahren im Vergleich zu Bäumen
 - erwartete Kosten für eine Operation (Einfügen, Suchen u. Löschen) ist konstant
 - einfache Implementierung

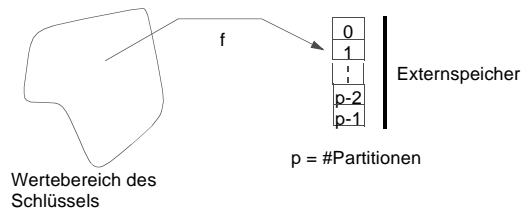
Beachte:

- ❑ Hashverfahren werden zum einen benutzt, um Daten zu partitionieren, so dass Partitionen vollständig in den Hauptspeicher passen (**Partitionierende Verfahren**).
- ❑ Zum anderen werden die Verfahren verwendet, um die Verarbeitung von Operationen zu beschleunigen, bei denen die Datenmengen bereits vollständig im Hauptspeicher liegen (**Hashverfahren**).

89.

4.2.1 Partitionierende Verfahren

- ❑ Verwendung einer Partitionierungsfunktion f :
 - Abbildung der Datensätze auf $\{0, \dots, p-1\}$.



- ❑ Für jede Partition wird eine Pufferseite im Hauptspeicher reserviert.
- ❑ Datensätze werden gelesen und mittels der Funktion f einer Partition zugeordnet, d. h. Datensätze werden in die zugehörige Pufferseite geschrieben.
- ❑ Falls eine Pufferseite voll ist, wird deren Inhalt auf eine Seite im Externspeicher übertragen.

91.

Problem bei Hashverfahren

- ❑ Was tun, wenn die Hashtabelle in Gefahr ist, nicht komplett in den Hauptspeicher zu passen?
 - optimistische Strategie:
Es wird zunächst versucht, die Hashtabelle im Hauptspeicher zu erzeugen. Bei Bedarf (bei einem Überlauf) wird eine Partitionierung vorgenommen.
 - pessimistische Strategie:
Eingabe wird (mittels einer Partitionierungsfunktion) in verschiedene Partitionen unterteilt.
 - hybride Strategie

90.

- ❑ Nach der vollständigen Partitionierung der Daten, wird iterativ für alle Partitionen folgendermaßen vorgegangen:
 - a) Alle Daten werden in den Hauptspeicher gelesen,
 - b) und das entsprechende Problem (z. B. Duplikatbeseitigung) wird gelöst.
- ❑ Sollte eine Partition immer noch nicht vollständig in den Hauptspeicher passen, wendet man obiges Prinzip rekursiv an oder verwendet ggf. Nested-Loops.

Leistung des Verfahrens

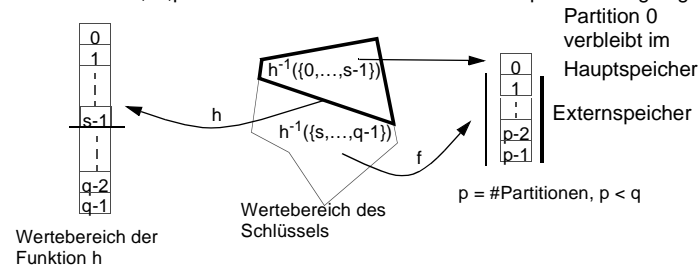
- ❑ Annahmen
 - Gleichmäßige Verteilung der Daten über die Partitionen, d. h. jede Partition besitzt etwa N/p Datensätze.
 - ❑ Für p Partitionen wird insgesamt $p \cdot C$ Hauptspeicher benötigt.
 - ❑ Um das Verfahren in zwei Phasen ablaufen zu lassen, muß für den verfügbaren Hauptspeicher M folgende Bedingungen erfüllt sein:
 - $M \geq N/p$
 - $M \geq p \cdot C + C$
- $p \approx \sqrt{N/C}$ ist die optimale Wahl für p , d. h. es gilt $M \approx \sqrt{CN}$.
- ❑ Anzahl der I/O Operationen: $2 \cdot N/C$

92.

4.2.2 Hybrides Hashing

Idee

- Möglichst viele Daten sollen im Hauptspeicher verarbeitet werden und nur wenige in Partitionen ausgelagert werden.
 - Partition 0 verbleibt im Hauptspeicher
 - Partitionen 1,...,p-1 werden bei Bedarf auf den Externspeicher ausgelagert.



93.

Wieviel externe Partitionen werden benötigt?

- Sei p die Anzahl der Partitionen. Wir nehmen an, daß jede der Partitionen aus m Seiten besteht.
- Jede der Partitionen benötigt einen Ausgabepuffer. Weiterhin wird noch ein Eingabepuffer benötigt. Somit verbleiben

$$m - (p + 1)$$

Seiten für die Partition 0 im Hauptspeicher.

- Berechne nun p so, daß p dem kleinsten Wert entspricht, so daß die Ungleichung

$$m - (p + 1) + mp \geq n$$

- Nach Auflösung bezgl. p ergibt sich dann

$$p = \lceil (n - m + 1) / (m - 1) \rceil$$

Aus dieser Gleichung lassen sich dann auch sofort die Kosten für das hybride Hashverfahren abschätzen.

95.

Parameter des Verfahrens

- Größe der Partition 0
 - Durch Verkleinerung des Grenzwertes s ist eine dynamische Anpassung der Größe der Partition 0 möglich.
- p = Anzahl der "externen" Partitionen
- Datenstruktur der Partition 0
 - Falls die Partition 0 sehr groß ist, ist eine effiziente Datenstruktur zur Verwaltung der Datensätze vorteilhaft.
 - In der Literatur wird i. a. vorgeschlagen, die Partition 0 wiederum durch eine Hashtabelle zu verwalten.

Bemerkung

- Hybrides Hashing lohnt, falls die Größe der Eingaberelation R zwischen M und M^2 Datensätze liegt, d. h. ein (ideales) Hashverfahren benötigt einen Partitionierungsschritt.

94.

4.3 Aggregate und Duplikatenbeseitigung

- Implementierung von Aggregation und Duplikatbeseitigung durch einen Operator
- Aggregatberechnung
 - Berechnung eines einzigen Werts für die gesamte Relation (Skalares Aggregat)
`select count(*)
from Emp
where Gehalt > 20000`
 - Berechnung eines Werts für jede Klasse einer Relation (Aggregatfunktionen)
`select AbtNr, count(*)
from Emp
where Gehalt > 20000
group by AbtNr`
- Duplikatelimination
 - ist beispielsweise nach einer Projektion mit *distinct* erforderlich
`select distinct Abteilung
from Angestellten
where Gehalt > 20000`
- Unterschied zwischen Duplikatelimination und Aggregatfunktionen?

96.

Duplikatbeseitigung durch Nested-Loops

- ❑ Algorithmus DupElim(Eingabe Relation R, Ausgabe Relation S)


```

      S.insert(R[0])
      out: FOR (int i = 1; i < R.length(); i++) {
            FOR (int j = 0; j < S.length(); j++)
              IF (R[i] == S[j])
                CONTINUE out;
            S.insert(R[i]);
          }
      
```
- ❑ Algorithmus hat im schlimmsten Fall quadratische Laufzeit.

Was tun, wenn R und S nicht in den Hauptspeicher passen?

- ❑ Wir betrachten jedes Tupel der Relation R und testen, ob das Tupel bereits im Hauptspeicherteil von S liegt.
 - Wenn ja, wird dieses Tupel verworfen.
 - Andernfalls wird überprüft, ob wir das Tupel noch im Hauptspeicher ablegen können.
 - Wenn ja, fügen wir das Tupel in den Hauptspeicherteil von S ein.
 - Andernfalls, speichern wir es auf dem Externspeicher in einer Datei R' ab.

97.

Bitfilter

Vorgehensweise bei einem Bitfilter

- ❑ Wertebereich wird disjunkt unterteilt. Jeder disjunkten Teilmenge wird ein Bit eines Vektors zugeordnet.
- ❑ Bit ist gesetzt, falls ein Datensatz aus S in der entsprechenden Teilmenge liegt.
- ❑ Wenn ein Datensatz gelesen wird, bei dem noch nicht ein Bit gesetzt ist, kann dieser als aktueller Satz frühzeitig zum nächsten Operator hochgereicht werden (auch wenn der Hauptspeicher bereits voll ist).
- ❑ Hochgereichte Datensätze, die nicht im Hauptspeicher Platz finden, werden nun (statt in der Relation R') in einer Relation R'' verwaltet.
- ❑ Nachdem die Relation R vollständig verarbeitet wurde, muß die Operation R' - R'' ausgeführt werden. Die daraus resultierende Menge wird dann weiterverarbeitet.

Eigenschaften:

- ❑ Durch Verwendung eines Bitfilters werden in einem Durchlauf mehr Datensätze ausgegeben (Ann.: Speicherplatz für den Bitfilter kann vernachlässigt werden).
- ❑ Zusätzlich muß jetzt nach einer Schleife eine Mengendifferenz berechnet werden.

99.

- ❑ Danach wird der Hauptspeicher geleert und der Algorithmus wird rekursiv für R' aufgerufen.

Wie kann ONC implementiert werden?

- ❑ In der Open-Phase wird im wesentlichen der Hauptspeicher reserviert und das erste Tupel eingefügt (und als aktuelles Tupel markiert).
- ❑ In der Next-Phase wird
 - das aktuelle Tupel ausgegeben,
 - und ein neues Tupel in den Hauptspeicher geholt (dies kann insbesondere dazu führen, dass die Relation R' erzeugt werden muß).

Vorschläge zur Leistungsverbesserung

- ❑ Die im Hauptspeicher liegenden Daten von S können durch einen Index organisiert werden, der Suchen und Einfügen effizient unterstützt.
- ❑ Verwendung eines **Bitfilters**

98.

Duplikatbeseitigung durch Sortieren und Hashing

Sortieren

- ❑ direkte Anwendung externer Sortierverfahren
- ❑ Vorschlag zur Leistungsverbesserung
 - Duplikatbeseitigung sollte so früh wie möglich durchgeführt werden, d. h. in den frühen Phasen des Verschmelzens sollten bereits Duplikate beseitigt werden.
 - Beobachtung von Bitton & DeWitt:
Durch eine frühe Duplikatbeseitigung werden die sortierten Teilfolgen nicht verkürzt. Erst beim letzten Verschmelzen werden viele Duplikate beseitigt.

Hashing

- ❑ Aufteilen der Daten in Partitionen, die komplett in den Hauptspeicher passen.
- ❑ Eliminierung von Duplikaten aus den Partitionen

100.

Besonderheiten bei Aggregatfunktionen

- ❑ Statt der Datensätze der Eingaberelation werden im Hauptspeicher Datensätze verwaltet, die aus den Gruppierungsattributen und den Werten der Aggregatfunktionen bestehen.
- ❑ Wird ein Duplikat erkannt, so führt dies zu einer Änderung der aggregierten Werte.

Problem:

- ❑ Als Problemfall erweisen sich Aggregatfunktionen, die eine Duplikateliminierung auf dem entsprechenden Attribut verlangen:
`select A, avg(distinct B1), ..., avg(distinct Bm) from ...`

101.

4.4 Joinverarbeitung

- ❑ Verknüpfung der Datensätze zweier Relationen, die ein (relationales) Prädikat erfüllen (relationale Joins)
- ❑ Varianten von Joins
 - **Semijoin** zwischen R und S (bzgl. eines Prädikats)
Berechnet alle Tupel aus R, die zur Joinberechnung mit S beitragen.
 - **Anti-Semijoin** zwischen R und S (bzgl. eines Prädikats)
Berechnet alle Tupel aus R, die nicht zur Joinberechnung mit S beitragen.
 - **Left-Outer Join** zwischen Relationen R und S (bzgl. eines Prädikats)
Vereinigung des Joins zwischen R und S mit dem Semijoin (R, S), wobei die Werte der "fehlenden Attribute von S" auf NULL gesetzt werden.
 - **Right-Outer Join** ...
 - ...
 - ...
- ❑ Anforderung an Joinverfahren
 Einsatz im Operatorbaum erfordert eine iterator-konforme Verarbeitung (datenflußorientierte Verarbeitung)
 ⇒ beim Start des Verfahrens wird noch nicht die gesamte Eingabe benötigt.

103.

Differenz

- ❑ Wir betrachten hier die Differenz zwischen zwei Multimengen R und S.
 - Als Ergebnis werden alle Tupel aus R geliefert, die nicht in S vorkommen.
 - Sei x die Anzahl der Instanzen des Tupel t in R und sei t nicht in S. Dann wird t genau x-mal als Ergebnis geliefert.

Nested-Loops

DO

Lies möglichst viele Tupel aus R in den Hauptspeicher;

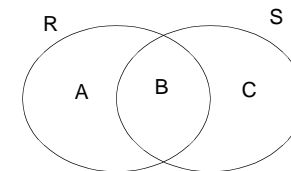
Für alle Tupel s aus S:

Falls es ein Tupel r aus dem Hauptspeicherteil von R mit $r = s$ gibt:
 Markiere r als gelöscht;

Gib alle als nicht gelöscht markierten Tupel an den darüberliegenden Operator;
 WHILE (noch nicht alle Tupel aus R sind verarbeitet);

102.

Implementierung von Mengenoperationen durch Joins



A	berechnet sich aus	Anti-SemiJoin(R,S)
B	berechnet sich aus	SemiJoin(R,S) bzw. Join(R,S)
C	berechnet sich aus	Anti-SemiJoin(S,R)
$A \cup B$	berechnet sich aus	Left-OuterJoin(R,S)
$A \cup C$	berechnet sich aus	Anti-Join(R,S)
$B \cup C$	berechnet sich aus	Right-OuterJoin(R,S)
$A \cup B \cup C$	berechnet sich aus	OuterJoin(R,S)

104.

4.4.1 Nested-Loops Join

Algorithmus

```
FOR (int i = 0; i < R.length(); i++) {
  FOR (int j = 0; j < S.length(); j++)
    IF (R[i] θ S[j]) (R[i], S[j]) ist ein Ergebnis;
```

❑ Bemerkungen

- R wird als *äußere* und S als *innere* Relation bezeichnet.
- S wird mehrmals durchlaufen und muß deshalb explizit in einer Datei abgespeichert bzw. zwischengespeichert werden.
- Die Leistung des Algorithmus beträgt $O(T(R) * T(S))$

❑ Leistungsverbesserungen

- Statt über Datensätze kann der Nested-Loops Join auch über Seiten iterieren. Mit einer Seitenkapazität C ist der Aufwand dann $O(T(R)/C * T(S)/C)$.
- Durch Lesen von D hintereinanderliegender Seiten kann die Leistung nochmals erheblich verbessert werden.
- Innere Relation kann abwechselnd von vorne und hinten eingelesen werden.

105.

Index Nested-Loop Join

- ❑ Falls auf S (der inneren Relation) ein entsprechender Index definiert ist, kann dieser bei der Joinverarbeitung genutzt werden.
- ❑ Index Nested-Loops Join sind effizienter als alle anderen Joinverfahren, falls S (innere Relation) erheblich größer ist als R.
- ❑ Benötigt man den kompletten Datensatz der inneren Relationen, müssen die TID noch entsprechend umgesetzt werden.
- ❑ Teilweise ist es sogar günstig, durch Massenaufbau einen Index für den Join zu erzeugen.
 - Der Index sollte dann auch die Datensätze direkt verwalten (und nicht wie bisher die Datensätze über TID ansprechen).

TID-Join

- ❑ Voraussetzung: auf beiden Relationen gibt es einen Index (auf dem Joinattribut)
- ❑ Zweistufiges Verfahren:
 1. Berechne die Paare von TIDs, die sich für den Join qualifizieren, indem die Indizes synchron sequentiell durchlaufen werden.
 2. Umsetzung aller TIDs in die entsprechenden Datensätze

107.

Leistung

❑ Annahmen

- innere Relation wird jeweils abwechselnd von vorne nach hinten gelesen
- m sei der zur Verfügung stehende Hauptspeicherplatz (in Seiten)
- k sei die Anzahl der Pufferseiten, die der Relation S zur Verfügung stehen.

❑ I/O Kosten zur Verarbeitung der inneren Schleife

$$\left\lceil \frac{B(R)}{m-k} \right\rceil \times (B(S) - k) + k$$

Dieser Ausdruck ist minimal, wenn R die größere Relation ist und $k=1$.

❑ Weitere Annahme

- Es werden immer Cluster von D Seiten eingelesen.

❑ Damit ergeben sich dann folgende I/O Kosten

$$\left\lceil \frac{B(R)}{m-D} \right\rceil \times (B(S)/D) + 1$$

Dieser Ausdruck ist minimal für $D = m/2$, d. h. es ist günstig ein Cluster halb so groß wie den verfügbaren Hauptspeicher zu wählen.

106.

4.4.2 Sort-Merge Join

Algorithmus

1. Sortiere (extern) Relationen R und S
2. Verschmelze die sortierte Teilfolgen und erzeuge dabei die qualifizierenden Paare von Datensätzen aus R und S.

Aufwand

$O(B(R) * \log_{M/C} B(R) + B(S) * \log_{M/C} B(S))$ I/Os (entspricht dem Aufwand von externen Sortieren)

Variante: Hybrid-Join (implementiert in DB2)

- ❑ Annahmen: Index (B+-Baum) auf der inneren Relation
- ❑ Skizze des Verfahrens
 - Sortieren der äußeren Relation
 - Verschmelzen mit der Blattebene des B+-Baums
 - Sortiere die Ergebnisse bzgl. den TIDs
 - Setze die TIDs in Datensätze um.

108.

4.4.3 Hash Join

Verarbeitung in zwei Phasen:

Build Phase: Aufbau einer (internen) Hashtabelle über der kleineren Relation.

Probing Phase: Testen der Tupel der größeren Relation gegen die Hashtabelle.

Falls die kleinere Relation nicht in den Hauptspeicher paßt,

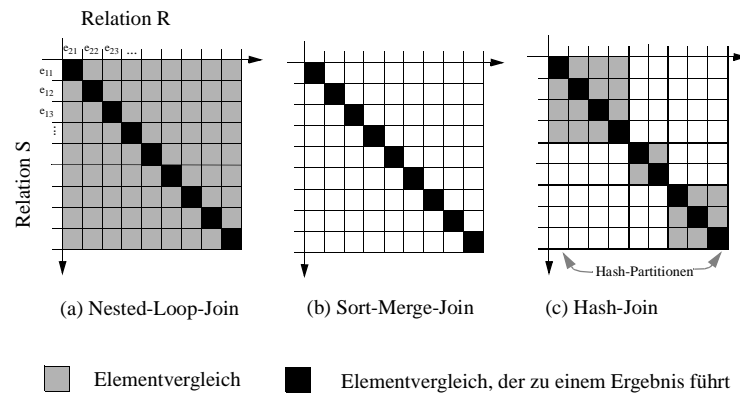
- ❑ Kleinere Relation wird in p Partitionen P_1, \dots, P_p mit $T(P_i) < M$.
- ❑ Führe die zweiphasige Verarbeitung für jede Partition durch.

Aufwand ($T(R) < T(S)$)

- ❑ Die Anzahl der I/Os beträgt $O(p \cdot B(S))$.

109.

Vergleich von verschiedenen Joins



111.

Leistungsverbesserung (Grace-Join und Hybrid-Hash-Join)

- ❑ Es wird nun sowohl die Relation R als auch die Relation S unter der Verwendung der gleichen Hashfunktion (Partitionierungsfunktion) in jeweils p Partitionen R_0, \dots, R_{p-1} und S_0, \dots, S_{p-1} aufgeteilt.
- ❑ Partitionen R_i und S_i , $0 \leq i < p$, werden in Dateien zwischengespeichert.
- ❑ FOR ($i = 0$; $i < p$; $i++$)
 - Aufbau einer Hashtabelle über R_i
 - Testen der Tupel aus S_i gegen die Hashtabelle

Hybrid-Hash-Join

- ❑ Optimiert das Verfahren dadurch, daß während der Partitionsbildung Aufbau und Testen von einer Hashtabelle erfolgt.
- ❑ Die Hybrid-Technik verbessert die Leistung immer dann erheblich, wenn die kleinere Relation nur wenig größer als der verfügbare Hauptspeicher ist.

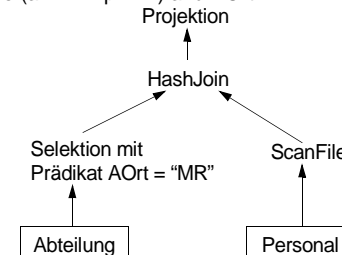
Kosten des Hybrid Hash Join

110.

4.5 Komplexe Anfragen

- ❑ Zusammensetzung einer Anfrage durch einen Operatorbaum
 - Blätter entsprechen den Basisrelationen
 - jedem internen Knoten wird ein Operator zugeordnet, der die ONC-Schrittstelle erfüllt. Man spricht dann auch von **Iteratoren**.

Beispiel
 select distinct AName, PName
 from Abteilung a, Personal p
 where (a.ANr = p.ANr) and AOrt = "MR"



112.

Effekt von Open, Next und Close für verschiedene Operatoren

- Exemplarisch für einige wichtige Operatoren

Iterator	Open	Next	Close
ScanFile (file)	file.open();	return file.next();	file.close();
Selektion(file, P)	file.open();	do t = file.next(); while(!(t erfüllt Prädikat P)) return t;	file.close();
Hash Join(left, right, P)	h = new HashTable(); left.open(); do t = left.next(); h.insert(t); while (left.eof()) left.close(); right.open();	do t = right.next(); s = h.probe(t); while (!P(s,t)); return (s,t);	right.close(); h = null; // Löschen von h
Sort(input)	input.open(); Erzeuge sortierte Teilfolgen der Ebene 0; input.close(); Verschmelze alle Teilfolgen bis nur noch ein Merge- schritt verbleibt;	Bestimme nächstes Ausga- beelement; Lies neues Tupel aus der entsprechenden Teilfolge;	Lösche alle Teilfolgen;

113.

Zusammenfassung

- Techniken zur Implementierung der Operatoren
 - externes Sortieren
 - Hashing
- Physische Operatoren zur Duplikatbeseitigung
- Physische Operatoren der Joinverarbeitung
 - Nested-Loops Join
 - Hash Join
 - SortMerge Join
- Komplexe Anfragen
 - physischer Operatorbaum
 - Knoten im Baum repräsentieren physische Operatoren, welche die ONC-Schnittstelle erfüllen (Iteratoren).

115.

Zusammenspiel der einzelnen Operatoren

- Ziel: möglichst hoher Datenfluß im Operatorbaum
- Vermeidung von "Stoppunkten"
 - Operatoren, die zur Initialisierung alle Daten benötigen.
- Bei binären Operatoren stellt sich die Frage, in welcher Reihenfolge Elemente der Eingaberelationen angefordert werden.
- Speicherzuteilung
 - Jeder der Operatoren hat einen gewissen Bedarf an Hauptspeicher. Werden nun Operatoren gleichzeitig verarbeitet, muß der Hauptspeicher entsprechend aufgeteilt werden.
 - I. a. wird es in einem Operatorbaum z. B. mehrere Sortiertoperatoren geben.

114.