Wintersemester 2014/15

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Suche in Dateien

"If you don't find it in the index,
look very carefully through the entire catalog"
— Sears, Roebuck, and Co., Consumers' Guide, 1897

(aus dem Buch von Ramakrishnan & Gehrke)

Lineare Suche

Binäre Suche in sortierten Datien

Suche via Indexen

Bekannt zum Teil aus der Vorlesung Informationssysteme.

Klassifizierung von Indexen

Primäre, Sekundäre und Clustering Indexe

- Jede Datei kann nur in einer Art und Weise sortiert sein.
- **Primär Index**: Index über gegebener Ordnung der Datei - via Primärschlüssel. Ein Index-Eintrag pro Block.
- **Clustering Index**: Ordnung wie in Datei, aber keine distinct Werte pro Attribut nach dem sortiert wird.
- **Sekundär Index**: Index über anderen Attributen (nicht wie Ordnung der Datei). Es kann mehrere dieser sek. Indexe geben

Single-Level und Multi-Level Indexe

- **Single-Level**: Einfacher Index, Verweise auf Sätze oder Blöcke
- **Multi-Level**: Verweise auch auf andere Index-Einträge (siehe B+ Baum)

Sparse vs. Dense

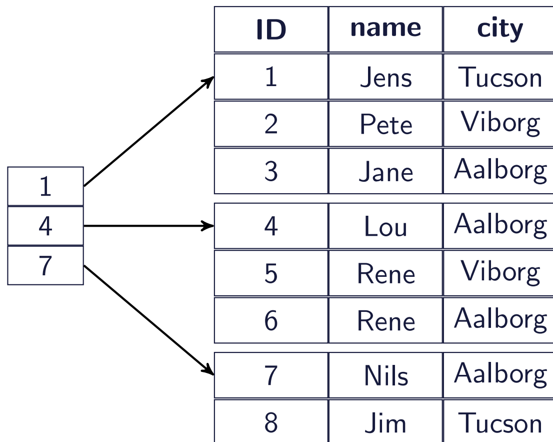
- **Dense(=Dicht)**: Index hat einen Eintrag für jeden (Daten)satz
- **Sparse (=Dünnbesetzt)**: Index verweist nur auf (Anfang eines) Blocks. D.h. primär Index ist sparse.

Beispiel

- Gegeben ein sekundärer Index auf einem Attribut, das für jeden Satz einen eigenen (distinct) Wert hat.
- Ist dieser Index sparse oder dense?

Literatur: Das Buch von Elmasri & Navathe: "Fundamentals of Database Systems" (Addison Wesley) ist hier sehr ausführlich!

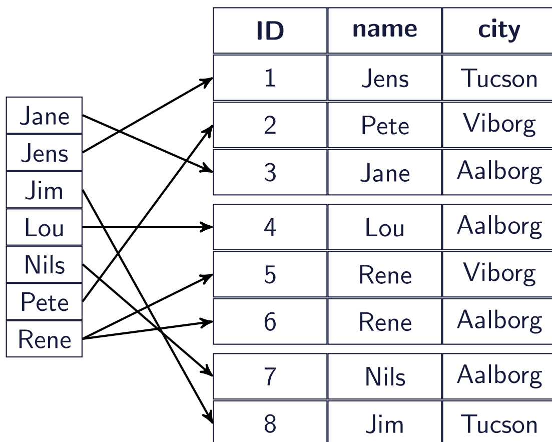
Beispiel: Primärer Index, Sparse



ID	name	city
1	Jens	Tucson
2	Pete	Viborg
3	Jane	Aalborg
4	Lou	Aalborg
5	Rene	Viborg
6	Rene	Aalborg
7	Nils	Aalborg
8	Jim	Tucson

- Angelegt über einer Datei, die nach dem Such-Schlüssel sortiert ist
- Ein Index-Eintrag für jeden Block

Beispiel: Sekundärer Index, Dense



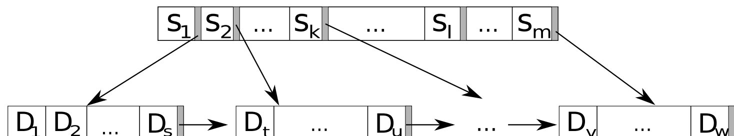
ID	name	city
1	Jens	Tucson
2	Pete	Viborg
3	Jane	Aalborg
4	Lou	Aalborg
5	Rene	Viborg
6	Rene	Aalborg
7	Nils	Aalborg
8	Jim	Tucson

- Angelegt über einer Datei, die nicht nach dem Such-Schlüssel geordnet ist
- Ein Index-Eintrag pro Tupel

ISAM

Index-Sequential Access Method (ISAM). Statisch.

Besteht aus Schlüsseln S_j und Datensätzen D_i.



- Sowohl Schlüssel als auch Daten werden geordnet abgespeichert.
- **Suche:**
  - Binäre Suche in Schlüsseln zur gewünschten Position
  - Sequentielles Lesen in Datenseiten
- **Einfügen:**
  - Auffinden der Einfügeposition (wie bei Suche)
  - Was passiert wenn die Seite, in die eingefügt werden soll, voll ist? Nicht gut ...
- **Löschen:**
  - Auffinden der Löschposition (wie bei Suche und Einfügen)
  - Löschen des Datensatzes. Was passiert wenn die Seite leer wird?

Indexstrukturen

Je schneller die Anfrage berechnet wird, desto besser.

Beobachtung

- Daten passen nicht in den Hauptspeicher.
- Es liegen Größenordnungen zwischen Performanz des Hauptspeichers und der Festplatte.
- Zugriffe sind unglaublich teuer!
- Insbesondere die wahlfreien (random access).

Was muss getan werden?

- Effiziente Speicherung auf Festplatte
- Wie findet man die gesuchten Daten möglichst schnell?
- Welche Garantien können bzgl. "Laufzeit" gegeben werden?

Bäume

Beobachtung

- Binäre Bäume nicht optimal für Festplatten
- Wichtig: Anpassung der Kapazität der Knoten an Größe der Seiten.
- Warum?

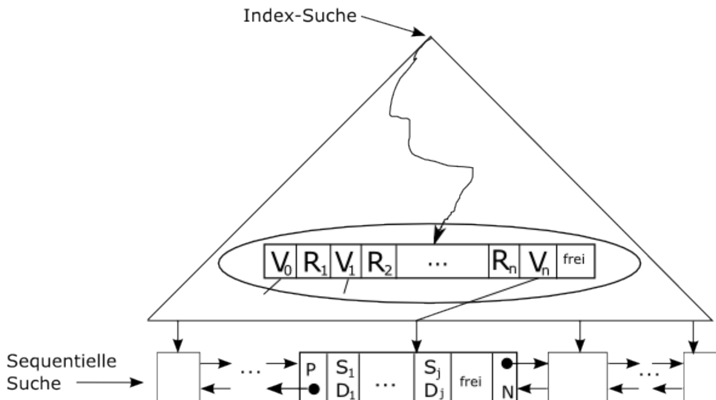
Fanout

- Je breiter der Baum desto weniger tief.
- Je flacher desto weniger “Sprünge” zwischen Knoten

⇒ Weniger Zugriffe auf Seiten auf der Festplatte.

B+ Baum (Bekannt aus VL Informationssysteme)

- "Hohler" Baum: Daten nur in den Blättern
- Suche muss also immer bis zu den Blättern laufen
- Aufbau: Referenzschlüssel R_j, Schlüssel S_k, Daten D_i, Zeiger V_m
- Blattknoten sequentiell verbunden!



B+ Baum Übersicht

- Jeder Knoten eines B+ Baums hat die Göße eines Blocks (Seite).
- Jeder Knoten ist mindestens 50% gefüllt
- Ein B+ Baum hat eine relativ geringe Anzahl von Stufen (Levels)
- Die ersten Ebenen (ein oder zwei) des Baumes werden im Hauptspeicher gehalten!
- “Logisch” nahe Knoten im Baum bedeutet nicht unbedingt auch physisch nahe. D.h. Zugriff auf einen Knoten kostet einen wahlfreien Zugriff.
- Die inneren Knoten sind eine Hierarchie von sparse Indexen.

Statischer Hash Index

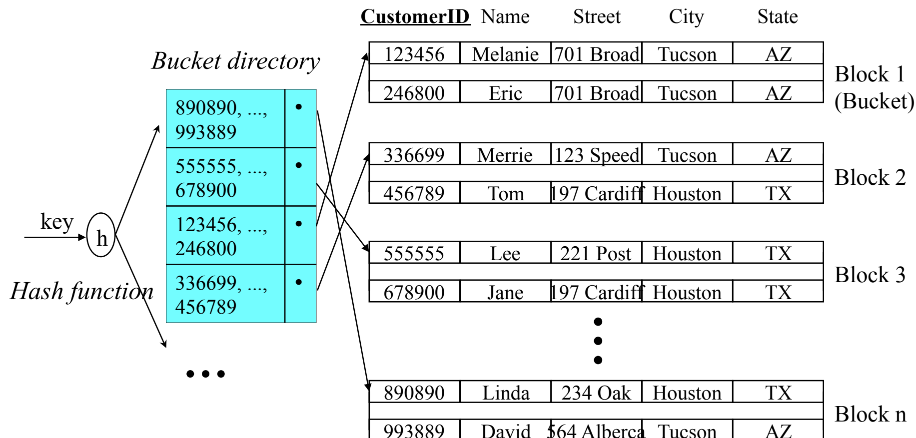
Idee

Erzeuge Index basierend auf Gruppierung von Tupel anhand Hash-Funktion.

Vorgehensweise

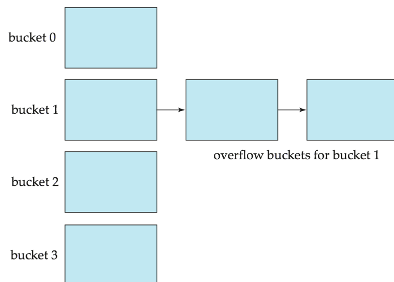
- Suche geeignete **Hash-Function** h
- Wende Hash-Funktion h an auf Wert k des Such-Schlüssel eines Tupels $\rightarrow h(k)$
- Erzeuge ein **Bucket** (Eimer) für jeden Wert von $h(k)$
- Hier, ein Block für jeden Bucket
- Einfaches Beispiel: $h(x) := x \textbf{ mod } 5$. Bildet z.B. Matrikelnummern ab auf Buckets 0 bis 4.

Statischer Hash Index: Beispiel



Statischer Hash Index

- Suche (Lookup)
  - Ein Zugriff auf auf das Verzeichnis (directory)
  - Ein Zugriff auf die eigentliche Datei
- Performanz hängt von Wahl der Hash-Funktion ab
- Überlauf von Buckets
  - Zu viele verschiedene Schlüssel-Werte werden auf gleichen Bucket abgebildet
  - Lösung: Überlaufbehandlung mittels Verkettung von Überlauf-Buckets



Statischer Hash Index: Probleme

Hash-Funktion und Anzahl der Buckets ist gegeben bei Initialisierung

Aber Daten verändern sich im Laufe der Zeit:

- Initiale Anzahl Buckets wird zu klein
  $\rightarrow$ viele Überlaufe, schlechte Performanz
- Initiale Anzahl Buckets zu groß gewählt
  $\rightarrow$ Verschwendung von Speicherplatz (leere Blöcke)

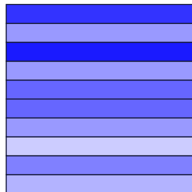
Lösung:

- Periodische Neuorganisation
- Dynamische Hashverfahren:
  erlauben dynamische Anpassung der Anzahl der Buckets

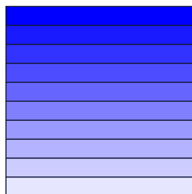
Wir kommen im Kapitel über Indexstrukturen ausführlich darauf zurück.

Überblick: Anordnung von Tupeln in Dateien

Haufen



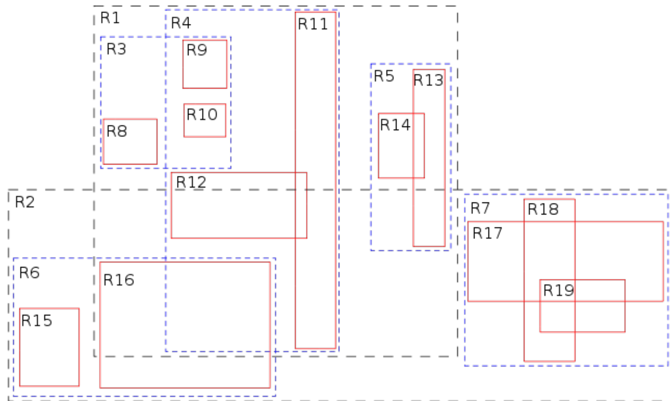
Sequenziell (Geordnet)



Hashing

Kurzer Ausblick: Kommende Vorlesungen zum Thema Indexstrukturen

- Werden zu späterem Zeitpunkt einige andere Index-Strukturen kennenlernen
- z.B. R Baum für räumlich ausgedehnte Objekte



Indexe in Postgres: B+ Baum

create table MeineTabelle (
  ID int,
  major int,
  minor int,
  name varchar
);

B+ Baum mit Schlüssel ID
create index index1 **on** MeineTabelle ID;

B+ Baum mit Schlüssel (major, minor)
create index index2 **on** MeineTabelle (major, minor);

Indexe über mehrere Spalten

Sogenannte Composite Indexe.
Angabe einer Reihe (Achtung! Reihenfolge ist wichtig!) von Spalten.

create index indexName **on** MeineTabelle (att1 , att2, att3);

Tupel werden dann im Index sortiert anhand dieser Attribute, "Lexikographische Ordnung": att1 ist primäres Kriterium, gefolgt von att2, etc.

Optional mit Ordnung ASC oder DESC der einzelnen Attribute. Default ist ASC. Zum Beispiel:

create index indexName **on** MeineTabelle (att1 **DESC** , att2, att3);

Indexe in Postgres: Hash Index

create index index3 **on** MeineTabelle **using hash** (column);

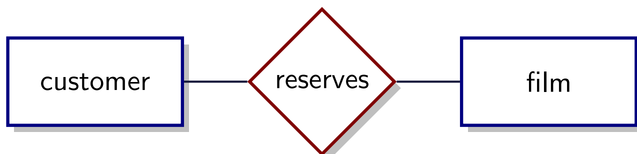
Eignung von Hash Indexen allgemein

- Verwendung bei Punktanfragen (d.h. vom Typ A=x)
- Hash Index über mehrere Attribute (Spalten) werden von Postgresql nicht unterstützt
- Nicht geeignet für Anfragen mit Operatoren$<$, $\leq$, $\geq$, $>$
- ebenso wie für **Bereichsanfragen**, z.B., $20 <$ Alter < 50

Fragen zum Performance-Tuning

- Soll auf diesem Attribut ein Index angelegt werden?
- Gibt es Attribute, für die gemeinsam ein Index angelegt werden soll?
- Für welche Anfragen wäre dieser geeignet? (Reihenfolge, z.B. (A,B) oder besser (B,A)?)
- Dazu, welche Anfragen werden im System erwartet?
- Was ist mit Attributen zu Primär- und Fremdschlüsseln?

Beispiel Datenbank



- **customer** (customerID, name, street, city, state)
- **reserved** (customerID, filmID, resDate)
- **film** (filmID, title, kind, rentalPrice)

Verschiedene Selektionen

- Primärschlüssel, Punktanfrage
  $$\sigma_{filmID=2}(film)$$
- Punktanfrage
  $$\sigma_{title='Terminator'}(film)$$
- Bereichsanfrage
  $$\sigma_{1<rentalPrice<4}(film)$$
- Konjunktion (d.h. logisches und)
  $$\sigma_{kind='F' \ \wedge \ rentalPrice=4}(film)$$
- Disjunktion (d.h. logisches oder)
  $$\sigma_{rentalPrice<2 \ \vee \ kind='D'}(film)$$

Ziel

Ersetze die Blätter des Anfrageplans durch spezifische Zugriffs-Methoden.

Strategien für konjunktive Anfragen

select *
from customer
where name = 'Jensen' **and** street = 'Elm'
and state = 'Arizona'

- Können die Indexe auf (name) und (street) benutzt werden? Ja
- Kann der Index auf (name, street, state) benutzt werden? Ja
- Kann der Index auf (name, street) benutzt werden? Ja
- Kann der Index auf (name, street, city) benutzt werden? Ja
- Kann der Index auf (city, name, street) benutzt? Nein

Optimierung für konjunktive Anfragen: Indexe anschauen!

Strategien für Punkt- und Bereichsanfragen

Lineare Suche

- Teuer, aber immer anwendbar

Binäre Suche

- Nur, wenn Daten passend sortiert sind

Suche mit Hash-Index

- Gut für Punktanfragen, aber nicht geeignet für Bereichsanfragen

Suche mit Primär/Clustering

- Für jeden Index Eintrag ein Block/Seite mit mehreren Einträgen (sparse)
- Verweis durch einen einzelnen Verweis (Pointer) auf den ersten Eintrag dort

Suche mit Sekundärindex

- Implementiert mittels Zeigern auf einzelne Einträge (dense)
- Kann teuer werden

Implementierung von Join Operatoren

Bislang nur Bedeutung (Ergebnis) eines Joins

- Aber nicht wie dieser berechnet wird.

Eingabe/Ausgabe

- Einganbe: Tabelle R und S
- Ausgabe: Alle Tupel in $R \bowtie S$
- Mit bekannter Semantik

Join Implementierungen

- Nested Loop Join
- Index-based Nested Loop Join
- Block-based (Index-Based) Nested Loop Join
- Sort-merge Join
- Hash Join

Größe der Eingabe-Relationen sowie Selektivität beeinflusst Kosten eines Joins

- Selektivität: $sel = \frac{\#\text{Ergebnistupel}}{\#\text{Kandidaten}}$
- Für Join, #Kandidaten ist die Größe des kartesischen Produkts

Nested Loop Join

"Brute Force" algorithm

for each $r \in R$
  for each $s \in S$
    if $s.B = r.A$
      then $Res := Res \cup (r \circ s)$

- Idee: bilde Kreuzprodukt
- Suche Tupel heraus, die das Joinprädikat erfüllen
- Problem: quadratischer Aufwand
- Vorteil: funktioniert für jedes Prädikat

Notation: $r \circ s$ bedeutet Konkatenation der Tupel

Nested Loop Join (NLJ)

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

emp

$\bowtie$

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

phone

$=$

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160
21	Dave	170
23	Anne	100
23	Anne	130
23	Anne	140

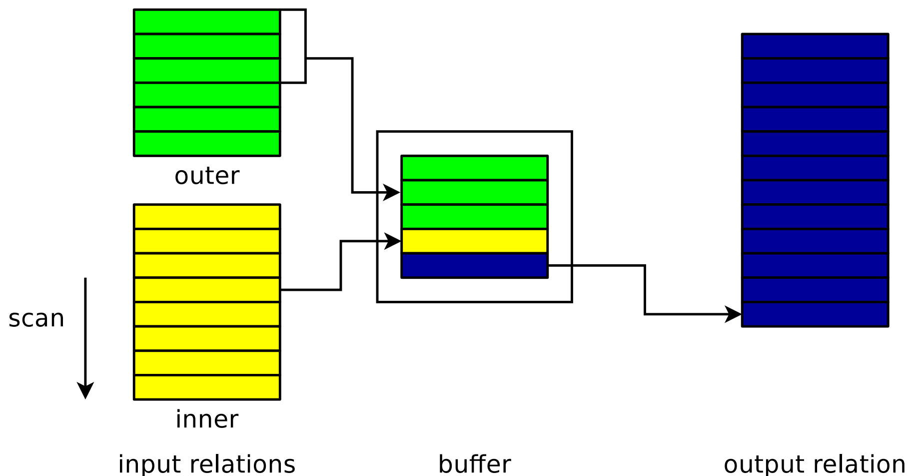
result

Als Iterator Implementierung

iterator NestedLoop

- **open**
  - Öffne die linke Eingabe
- **next**
  - Rechte Eingabe geschlossen?
    - Öffne sie
  - Fordere rechts solange Tupel an, bis Bedingung p erfüllt ist
  - Sollte zwischendurch rechte Eingabe erschöpft sein
    - Schließe rechte Eingabe
    - Fordere nächtes Tupel der linken Eingabe an
    - Starte **next** neu
  - Return Verbund von aktuellem linken und aktuellem rechten Tupel
- **close**
  - Schließe beide Eingabequellen

Block Nested Loop Join



Block Nested Loop Join

Nicht alle Blöcke passen in Hauptspeicher

repeat
  lese $n_B - 2$ Blöcke aus der äußeren Relation
  repeat
    lese 1 Block der inneren Relation
    vergleiche Tupel
  until Komplette innere Rel. gelesen
until Komplette äußere Rel. gelesen

Parameter:

- b_{inner}, b_{outer}: Anzahl Blöcke
- n_B: Größe des Puffers im Hauptspeicher

Kostenschätzung (als Anzahl Zugriffe auf Blöcke)

$$b_{outer} + (\lceil b_{outer}/(n_B - 2) \rceil) * b_{inner}$$

Block Nested Loop Join

Example ($reserved \bowtie customer$)

- Größe der Relationen in Anzahl Blöcke
  $b_{reserved} = 2000$, $b_{customer} = 10$
- Größe des Puffers im Hauptspeicher
  $n_B = 6$
- Kostenschätzung (Anzahl transferierte Blöcke)
  $b_{outer} + (\lceil b_{outer}/(n_B - 2) \rceil) * b_{inner}$

Kosten

- **Relation** `reserved` **als äußere Relation:**
  $2000 + \lceil (2.000/4) \rceil * 10 = 7000$
- **Relation** `customer` **als äußere Relation:**
  $10 + \lceil (10/4) \rceil * 2000 = 6010$

Index-based Nested Loop Join

for each $r \in R$
    for each $s \in S$ **where** $[s.B = r.A]$
        then $Res := Res \cup (r \circ s)$

Gleiches Prinzip wie Nested Loop Join

- Äußere Relation
- Innere Relation
- Suche anhand Index kann (teure) naive Suche auf innerer Relation ersetzen

Merge Join

Ausnutzen von sortierten Relationen

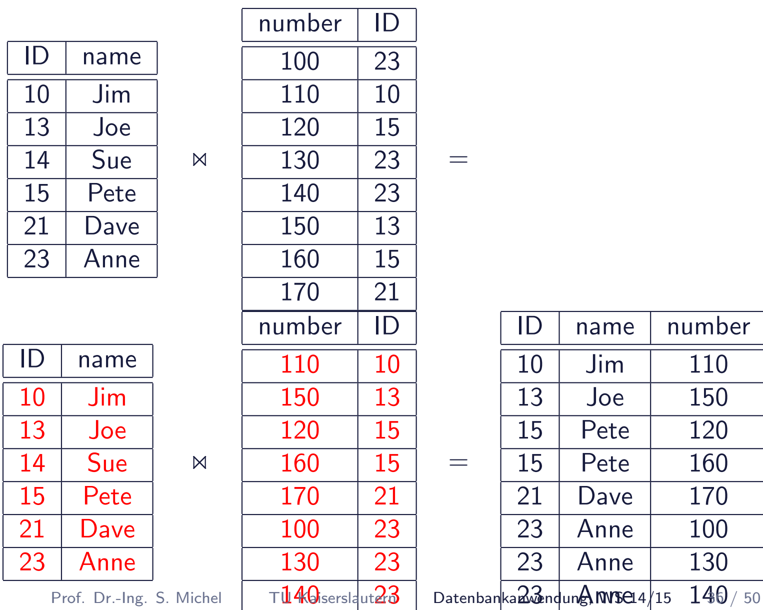
R	
	A
...	0
...	7
...	7
...	8
...	8
...	10
...	...

$\leftarrow$ $\rightarrow$

S	
B	
5	...
6	...
7	...
8	...
8	...
11	...
...	...

- Beide Eingaben sind sortiert (oder werden vorher sortiert)
- Dann genügt ein einfacher Merge über die Eingaben.
- Achtung! Bei “Zeilen” mit gleichen Werten muss zurückgesprungen werden.

Merge Join



ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

⋈

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

=

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

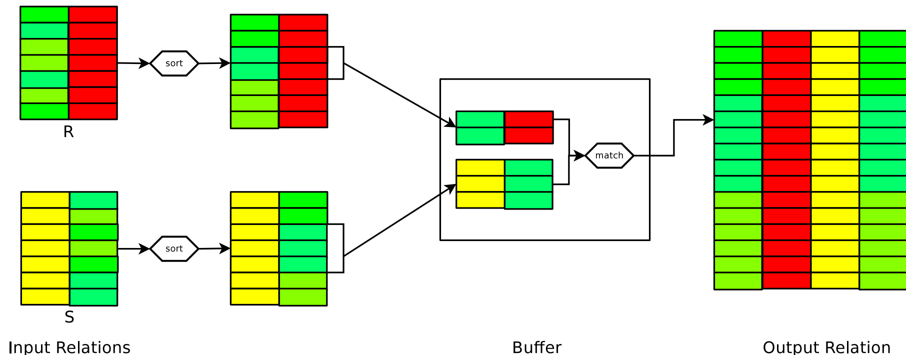
⋈

number	ID
110	10
150	13
120	15
160	15
170	21
100	23
130	23
140	23

=

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160
21	Dave	170
23	Anne	100
23	Anne	130
23	Anne	140

Merge Join: Veranschaulichung



Merge Join – Kosten

Parameter

- b_1, b_2: Anzahl Blöcke

Kostenschätzung (Anzahl Blöcke-Transfers)

$$b_1 + b_2$$

Für den Fall, dass eine Relation nicht mehrfach gescannt werden muss.

Was passiert, wenn die Tupel in beiden Seiten (d.h. alle) den gleichen Wert für das Join Attribut haben?

Erweiterung

- Kombination mit Sortieren falls Eingabe-Relationen nicht bereits sortiert.
- Was ist wenn Daten zu groß für Hauptspeicher sind?

Externes Sortieren

Wenn eine Datei (Relation) nicht als Ganzes in den Hauptspeicher passt kann sie dennoch sortiert werden:

Idee

- Spalte Datei in (viele) kleinere Teile (Runs)
- Sortiere jedes Teil.
- Merge diese inkrementell.
- Anzahl dieser Schritte hängt von Größe des zur Verfügung stehenden Hauptspeicher Puffers ab

Erzeugen von Runs

Wie viele solcher Runs gibt es?

- Anzahl von Blöcken der Datei: b (d.h. Größe der Datein in Anzahl Blöcke)
- Größe des zum sortieren nutzbaren Hauptspeichers (Puffer) in Blöcke: n_B
- Anzahl Runs gegeben durch

$$n_R = \lceil \frac{b}{n_B} \rceil$$

- Beispiel: $n_B = 5$ Blöcke, $b = 1024$ Blöcke, dann sind 205 (initiale) Runs erzeugt worden
  - Jeder dieser Runs ist 5 Blöcke groß, bis auf den letzten, der 4 Blöcke belegt.
  - D.h. nach diesem Schritt liegen 205 sortierte Runs als temporäre Dateien auf der Festplatte.

Externes Sortieren

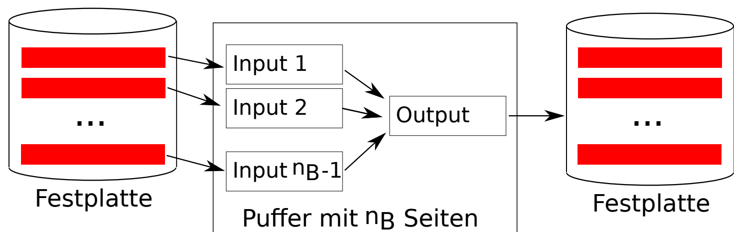
Sortier-Phase

- Lese die nächsten n_B Blocks der Datei in den Puffer
- Sortiere die Tupel im Puffer und schreibe Ergebnis in temporäre Teil-Datei (Run)

Merge-Phase

- Anzahl Merge-Phasen ist $p = \lceil log_{n_B}(n_R) \rceil$
- solange noch mehr als ein Run übrig:
  - Lese die nächsten $n_B - 1$ Runs, je ein Block nach dem anderen
  - Merge Tupel und schreibe Ergebnis (Run) auf Festplatte, ein Block nach dem anderen.
  - hierbei werden längere Runs erzeugt

Externes Sortieren: Illustration Mischen von Runs



- (n_B-1) – way merge

Beispiel

Anzahl Durchläufe (Passes), in Abhängigkeit von Größe der Datei in b Seiten (Blöcke) und Anzahl Puffer Seiten n_B.

b	n_B=3	n_B=5	n_B=9	n_B=17	n_B=129	n_B=257
100	7	4	3	2	1	1
1000	10	5	4	3	2	2
10000	13	7	5	4	2	2
100000	17	9	6	5	3	3
1000000	20	10	7	5	3	3
10000000	23	12	8	6	4	3
100000000	26	14	9	7	4	4
1000000000	30	15	10	8	5	4

Sortieren: Anwendung

Wann muss ein Datenbanksystem Daten sortieren?

- Wie oben gesehen, zur **Realisierung von Sort-Merge Joins**
- Oder zur **Eliminierung von Duplikaten** (select distinct ...) (wie?)
- Falls Benutzer **Antworten in einer bestimmen Reihenfolge** haben möchten.
- Wenn z.B. ein B+ Baum über grossen bestehenden Daten angelegt werden soll: Sogenanntes **Bulk-Loading**. Wieso ist das geschickt?

Hash Join

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

Angestellte

$\bowtie$

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

Telefon

Wende Hashfunktion an auf Join-Attribut(e)
$\rightarrow$ Partitioniert Tupel in Buckets

Hash Join

ID	name
15	Pete
21	Dave

Angestellte_0 $\bowtie$

number	ID
120	15
160	15
170	21

Telefon_0 $=$

ID	name	number
15	Pete	120
15	Pete	160
21	Dave	170

Ergebnis_0

ID	name
10	Jim
13	Joe

Angestellte_1 $\bowtie$

number	ID
110	10
150	13

Telefon_1 $=$

ID	name	number
10	Jim	110
13	Joe	150

Ergebnis_1

ID	name
14	Sue
23	Anne

Angestellte_2 $\bowtie$

number	ID
100	23
130	23
140	23

Telefon_2 $=$

ID	name	number
23	Anne	100
23	Anne	130
23	Anne	140

Ergebnis_2

Hash Join

- Hashing jeder Relation basierend auf Join Attribut(en)
- Jeder Bucket muss klein genug für Hauptspeicher sein
- Tupel in korrespondierenden Buckets der beiden Relationen werden dann verbunden

R

h

S

h

⋈

⋈

⋈

Input Relations

Output Relation

Hash Join: Pseudocode

Aka. Grace Hash Join (in Literatur)

Partitioniere R in k Partitionen R_i

for each tuple $r \in R$ **do**
    lese r und füge es Puffer-Seite (Block) $h(r)$ hinzu

Partitioniere S in k Partitionen S_i

for each tuple $s \in S$ **do**
    lese s und füge es Puffer-Seite (Block) $h(s)$ hinzu

Hash Join: Pseudocode (2)

Sondierung (engl. Probing) Phase

for $l = 1, \ldots, k$ **do** {

//Erzeuge im Hauptspeicher Hash-Tabelle für R_l mit Hash-Funktion h_2
for each tuple $r \in$ partition R_l **do**
lese r und füge es in Hash-Tabelle anhand $h_2(r)$ ein

//Lese S_l und teste nach passenden Tupeln aus R_l
for each tuple $s \in$ partition S_l **do** {
lese s und teste Hash-Tabelle unter Verwendung von $h_2(s)$
für passende Tupel $r \in R$: $Res := Res \cup (r \circ s)$ }
leere Hash-Tabelle und betrachte nächste Partition.

}

Kosten und Anwendbarkeit der verschiedenen Join-Strategien

Nested Loop Join

- Kann für alle Arten von Joins benutzt werden
- Kann aber sehr teuer sein

Merge Join

- Eingaben müssen bereits sortiert vorliegen
- Oder für den Join extra sortiert werden
- Kann ggf. Indexe ausnutzen

Hash join

- Gute Hashfunktionen sind elementar
- Performanz am besten, wenn kleinere Relation direkt in Hauptspeicher passt

Was ist mit Joins über mehrere Attribute? Was mit Joins, die nicht auf Gleichheit (=) testen? Welche Implementierungen sind anwendbar?