

Optimierung

Lehrbuch Kapitel 7.1 bis 7.5

1 L	Einführung
4 L	Datenorganisation Speicherung
4 L	Optimierung
2 L	Transaktionen, Recovery
2 L	Non-Standard Datenbanken
1 L	Repetition, Abschluss

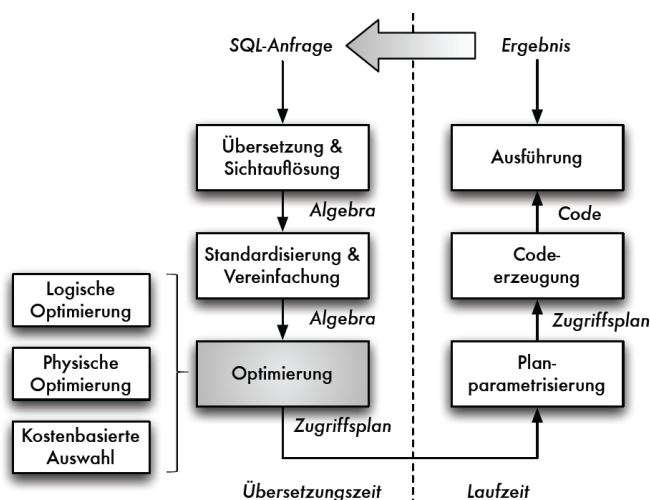
← "You are here"

Verschiedene Zugriffsstrukturen (Dateiorganisationsform und/oder Zugriffspfad) verstehen:

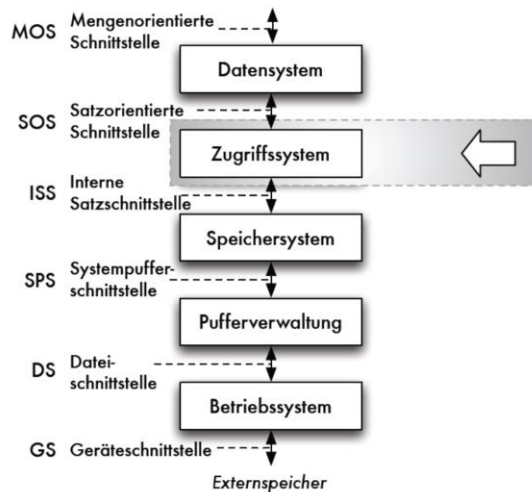
- Statische Verfahren:
 - Heap
 - Sequentielle Datei
 - Indexsequentielle Dateien
 - Indexiert-nichtsequentielle Datei
 - Statisches Hashing
- Dynamische Verfahren:
 - B-Bäume
 - Dynamisches Hashing

- Verständnis gewinnen für die 'Kosten' eines Verfahrens
- Einige in RDBMS eingesetzten Algorithmen kennen und 'Kosten' beurteilen können:
 1. Hauptspeicheralgorithmen
 2. Zugriffe auf Datensätze
 3. Scans
 4. Sortierung
 5. Joins / Verbund (1. Teil)

Zur Orientierung im Abschnitt «Optimierung» dient uns das Bild unten. Wir werden dieses später detaillierter betrachten. Wir werden zunächst die Grundlagen schaffen und anschliessend den gesamten Prozess von der SQL-Abfrage bis zur Rückgabe des Ergebnisses, in welchem die Optimierung eine zentrale Rolle spielt, betrachten. In der heutigen Lektion werden die Basisalgorithmen (z.B. wie führt das DBMS einen Join durch) betrachten, welches das DBMS nutzt, um eine SQL-Abfrage durchzuführen, sowie überlegen, welche Kosten die einzelnen Basisalgorithmen verursachen, so dass das Datenbank jeweils den «günstigsten» Basisalgorithmus auswählen kann (physische Optimierung).



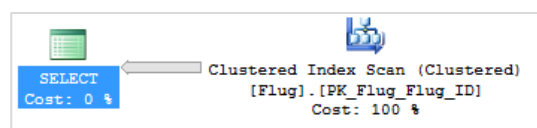
- Das Zugriffssystem baut auf den Funktionen des Speichersystems auf und stellt **Basisalgorithmen** dem Datensystem zur Verfügung.
- Die Auswahl der geeigneten Algorithmen, etwa die Auswahl eines Zugriffspfades, übernimmt die **physische Optimierung**.




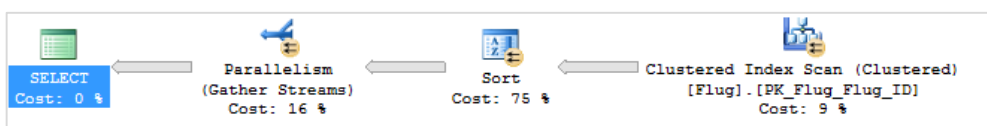
Es ist üblich, die Operatoren der Relationalalgebra (z.B. Projektion, Selektion, Join), sowie weitere Operationen (z.B. Sortierung) als Funktionen (Methoden) des Zugriffssystems zu implementieren. Für einzelne Operationen (z.B. Join) werden dabei mehrere Basisalgorithmen implementiert, um für unterschiedliche Situationen jeweils den geeigneten Algorithmus auswählen zu können. Die Auswahl des geeigneten Basisalgorithmus und der besten Reihenfolge übernimmt der Optimizer (Komponente des Zugriffssystems).

Die vom DBMS ausgewählten Basisalgorithmen und Reihenfolge wird als Ausführungsplan (Execution Plan) bezeichnet. Diese Auswahl erfolgt aufgrund einer Aufwand-Kostenabschätzung. Im SQL-Server kann dieser Ausführungsplan auch grafisch dargestellt werden, z.B. für (Ausführungspläne werden von rechts nach links gelesen):

`SELECT * FROM FlughafenDB.dbo.Flug;`



Wird zusätzlich noch nach Flugnr sortiert (`SELECT * FROM FlughafenDB.dbo.Flug ORDER BY Flugnr`), ändert sich auch der Ausführungsplan. Wer genau hinschaut stellt fest, auch der Clustered Index Scan hat sich geändert (), er wird jetzt parallel ausgeführt (abhängig von Hardware):



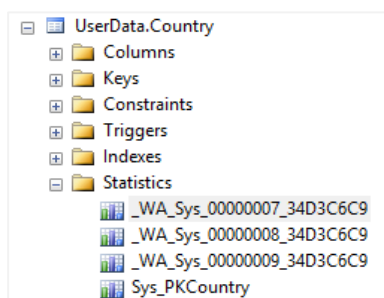
Aufwandabschätzungen: Basialgorithmen

- Für **Aufwandsabschätzungen** (→ zur Auswahl eines Verfahrens bei der Optimierung) sind zusätzliche Informationen notwendig:
 - $|r|$ Anzahl Tupel in der Relation r
 - b_r Anzahl von Blöcken die Tupel der Relation r beinhalten
 - mem Puffergrösse in Anzahl der Blöcke
 - $val_{A,r}$ Anzahl verschiedener Werte für das Attribut A in der Relation r («Cardinality»)
 - $lev_{I(R(A))}$ Anzahl der Indexebenen eines B⁺-Baums für den Index $I(R(A))$ für das Attribut A des Relationenschemas R
 - $bsize$ Blockgrösse
 - $size_r$ (mittlere) Grösse von Tupeln aus r
 - f_r Blockungsfaktor – wieviel Tupel aus r können in einem Block gespeichert werden: $f_r = bsize / size_r$

6

Zürcher Fachhochschule

Auch der SQL Server führt eine Vielzahl dieser Statistiken im Systemkatalog. Diese Zahlen können auch abgefragt werden, entweder im Management-Studio (siehe Bild unten), oder mittels SQL-Befehlen (**DBCC SHOW_STATISTICS** ('UserData.Country', '_WA_Sys_00000007_34D3C6C9') **WITH HISTOGRAM**). Im Beispiel wird für die Tabelle Country das Histogramm (Verteilung der Attributswerte) angezeigt (rechts das Resultat der Abfrage).



	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
4	AL	1	1	1	1
5	AM	0	1	0	1
6	AN	0	1	0	1
7	AR	1	1	1	1
8	AT	0	1	0	1
9	AW	1	1	1	1
10	AZ	0	1	0	1
11	BA	0	1	0	1
12	BB	0	1	0	1
13	BE	1	1	1	1
14	BG	1	1	1	1
15	BI	1	1	1	1
16	BJ	0	1	0	1
17	BN	1	1	1	1

Die auf der Folie aufgeführten statistischen Kennzahlen werden wir bei der Kosten-Betrachtung der nachfolgend eingeführten Basialgorithmen verwenden.

- Werte ändern sich **während der Benutzung** und müssen nachgeführt werden (z.B. $val_{A,r}$).
- Grundlegende Aufwandabschätzung geht von folgenden Annahmen aus:
 - **Dominierender Kostenfaktor** sind Seitenzugriffe
 - Indexe als B⁺-Bäume realisiert
 - Für Zwischenergebnisse sind Seitenzugriffe nötig (z.B. Sortierung)
 - Zwischenrelationen werden im Hintergrundspeicher (nicht Hauptspeicher) angelegt
 - Hauptspeicheralgorithmen (RAM) werden vernachlässigt

Im den folgenden Kostenberechnungen vernachlässigen wir Hauptspeicheralgorithmen (die im RAM ohne externe Speichermedien durchgeführt werden). Allerdings sind diese für den Gesamtdurchsatz des DBMS auch wichtig, da diese SEHR häufig genutzt werden. Typische Hauptspeicheralgorithmen:

- Sortierung: Iterativ durch Vergleich der Einzelattribute, wobei Attribute mit grosser Selektivität (zum Beispiel Schlüsselattribute) zuerst verglichen/getestet werden
- Tupelvergleich (z.B. für Distinct-Operation) (analog Sortierung)
- TID-Zugriff: Innerhalb einer bereits in den Hauptspeicher transferierten Seite erfolgt der Zugriff auf Tupel mittels des Tupelidentifikators TID
- Binäre Suche innerhalb von Seiten im Puffer

Da sich die Werte (vorherige Folie) im laufenden Betrieb ändern, muss in regelmässigen Abständen eine Art 'Nabelschau' betrieben werden, um diese Werte im Systemkatalog zu aktualisieren (Art und Zeitpunkt kann im DBMS konfiguriert werden). Im SQL Server werden diese Statistiken in der Regel (Default) automatisch nachgeführt («The query optimizer determines when statistics might be out-of-date and then updates them when they are needed for a query plan.»).

Möchte man die Statistik öfter aktualisieren, kann dies z.B. innerhalb eines Maintenance-Jobs erfolgen. Aber Achtung: Bei geänderten Statistiken werden bestehende Ausführungspläne überprüft und gegebenenfalls neu erstellt, was wiederum zusätzliche Ressourcen zur Laufzeit benötigt.

- Für relationenalgebraische Operationen (resp. Bag-Algebra):

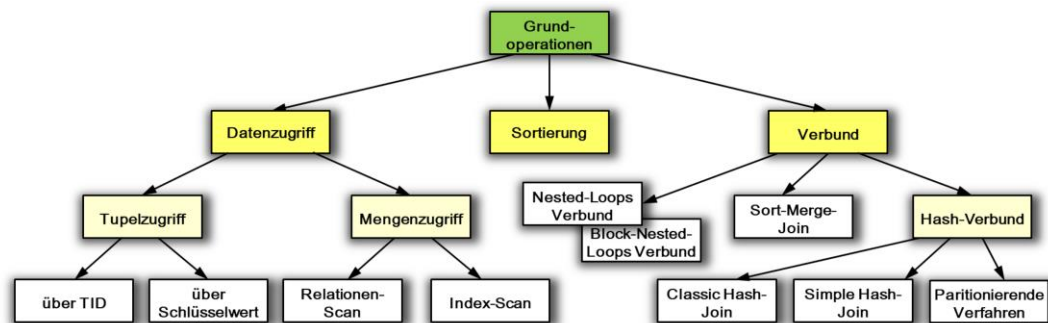
- Selektion
- Projektion
- Aggregation
- Gruppierung
- Vereinigung
- Durchschnitt
- Differenz

gelangen **Kombinationen** der Verfahren zum Einsatz.

Die Kostenabschätzung ist oft anspruchsvoll!

- Achtung: **Duplikateliminationen** (DISTINCT) sind **teure** Operationen, wenn die Daten nicht sortiert vorliegen!
- Details hierzu sind im Lehrbuch in den Abschnitten 7.3 & 7.4 zu finden
→ Selbststudium!

- Klassifikation von **Zugriffs-Grundoperationen**:



Das Bild auf der Folie bildet einen Rahmen um die verschiedenen Algorithmen des Zugriffssystems zu klassifizieren, welche wir im Folgenden betrachten werden (ohne Hauptspeicheralgorithmen):

1. Beim Datenzugriff werden Daten gelesen/verändert. Beim Tupelzugriff wird maximal ein Datensatz anhand eines Wertes (TID oder Schlüsselwert) gesucht, während beim Mengenzugriff (SCAN, Durchlaufen einer Relation), alle oder eine Menge von Datensätzen gelesen wird.
2. Bei der Sortierung werden grosse Datenmengen ausserhalb des Hauptspeichers sortiert.
3. Bei den Verbundoperationen werden mehrere Tabellen miteinander verbunden (Join).

Die nachfolgend eingeführten Algorithmen zeigen die Grundidee dieser Algorithmen. Die konkrete Implementation im DBMS weicht davon ab. Des weiteren wird jedes DBMS noch weitere, andere Grundoperationen implementieren.

Das Bild rechts zeigt einen kleinen Ausschnitt der Operationen, welche im Ausführungsplan des SQL Servers dargestellt werden ([https://technet.microsoft.com/en-us/library/ms175913\(v=sql.105\).aspx](https://technet.microsoft.com/en-us/library/ms175913(v=sql.105).aspx)).

	Bitmap
	Bookmark Lookup
	Clustered Index Delete
	Clustered Index Insert
	Clustered Index Scan
	Clustered Index Seek
	Clustered Index Update
	Collapse
	Compute Scalar
	Concatenation
	Constant Scan
	Delete
	Deleted Scan
	Eager Spool
	Filter
	Hash Match
	Hash Match Root
	Hash Match Team

- Zugriff auf Tupel mittels RelationenID (Identifikator der Relation) und TID. Tupel wird im Tupelpuffer abgelegt.
- Operation:
`fetch-tuple(RelationenID; TID) → Tupelpuffer`
- Aufwand: $O(1)$



Zur Erinnerung, die TID besteht aus der Seitennummer und einem Offset, wodurch direkt auf den im Hintergrundspeicher liegenden Datensatz zugegriffen werden kann.

- Zwei Arten von Indexen:
 1. **Primärindex**, z.B. Index(KUNDE(KNr)): liefert max. ein Tupel
 2. **Sekundärindex**, z.B. Index(BESTELLUNG(KNr)): liefert ev. mehrere Tupel, siehe Index-Scan
- Zugriff auf Index mittels IndexID (Identifikator des Index)
- Operation Primärindex:
`fetch-TID(IndexID; Attributwert) → TID`
- Aufwand: $O(\text{lev}_{I(R(A))}) + O(1) \approx O(\log_m(|r|))$
 - $\text{lev}_{I(R(A))}$ = Anzahl Indexebenen
 - m = Ordnung des Baumes
 - r = Anzahl Tupel



11

Zürcher Fachhochschule

Beim Zugriff mittels Schlüsselwert wird maximal ein Datensatz gelesen. In der Regel wird hierbei über den Primärschlüssel zugegriffen.

Sekundärindexe können NICHT mittels fetch-TID abgehandelt werden, diese werden im Mengenzugriff verwendet. Bei Sekundärindexten ist das Resultat nicht eine einzelne TID sondern eine TID-Liste (siehe Index-Scan).

Die Anzahl der Indexebenen des Baumes $\text{lev}_{I(R(A))}$ lässt sich als $\log_m(|r|)$ berechnen, daher ist $O(\text{lev}_{I(R(A))}) = O(\log_m(|r|))$. Da $O(1)$ (Aufwand zum Lesen des Tupels mittels TID) bei zunehmenden Mengen im Verhältnis zu $O(\log_m(|r|))$ an Gewichtung verliert, kann der Aufwand als $O(\log_m(|r|))$ gesehen werden.

Beispiel (Primärindex über KNr):

```
SELECT * FROM Kunde WHERE KNr = 4711
```

Kann wie folgt umgesetzt werden:

```
1 currTID ← fetch-TID(KUNDE-KNR-Index, 4711);  
2 currRec ← fetch-tuple(KUNDE-RelationID, currTID);  
3 put(currRec);
```

Anzeige des Ergebnisses



Der Zugriff mittels Schlüsselwert wird also durch die zwei Operationen fetch-TID und fetch-tuple implementiert.

Relationen-Scan (full table scan)

- **Scan:** Durchlaufen von Tupeln einer Relation (z.B. bei Selektionen / Projektionen).
- **Relationen-Scan:** Durchlaufen *aller* Tupel einer Relation in beliebiger Reihenfolge.
- Operationen:
 - `open-rel-scan(RelationenID) → ScanID`
 - `next-TID(ScanID)` liefert nächsten TID
 - `end-of-scan(ScanID)` true, falls kein TID mehr abzuarbeiten
 - `close-scan(ScanID)` schliesst Scan
- Aufwand (cost_{REL}): $O(b_r)$
 - b_r = Anzahl von Blöcke dieTupel beinhalten



13

`SELECT * FROM Kunde WHERE Nachname BETWEEN 'Heuer' AND 'Jagellowsk'`

Umgesetzt mittels Operationen:

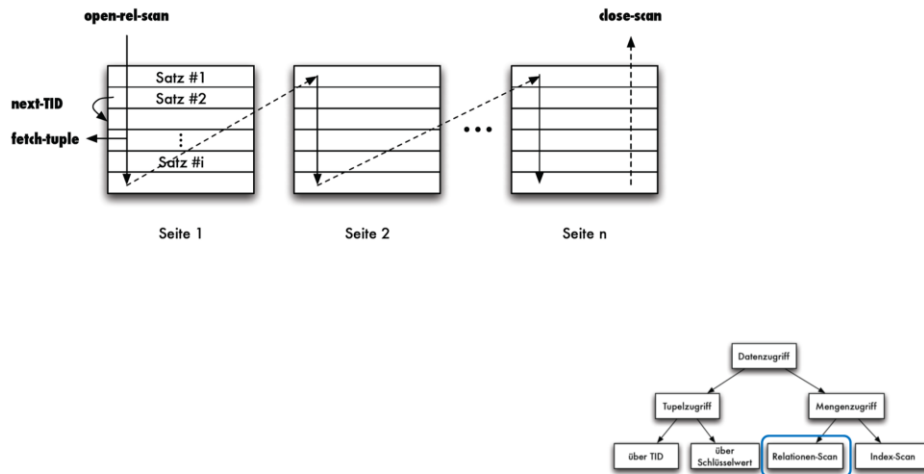
```
1 currScanID ← open-rel-scan(KUNDE-RelationID);
2 currTID ← next-TID(currScanID);
3 while ¬ end-of-scan(currScanID) do
4   currRec ← fetch-tuple(KUNDE-RelationID, currTID);
5   if currRec.Nachname ≥ 'Heuer' and currRec.Nachname ≤ 'Jagellowsk'
6     then
7       put(currRec);
8     end
9   currTID ← next-TID(currScanID);
10 end
11 close-scan(currScanID);
```

In der Programmierung mittels Stored Procedures und Functions spielt der Relationen-Scan ebenfalls eine wichtige Rolle. Dort wird mittels Cursor-Konzept über die Tabellen gescannt. Dieser Scan kann zusätzlich mit Bedingungen, Sortierung, Verbund, usw. ergänzt werden (dann wird es aber ev. nicht mehr als Relationen-Scan ausgeführt). Bsp.:

```
DECLARE kunden_Cursor CURSOR FOR SELECT Name, Vorname FROM Kunde WHERE KNr > 100
OPEN kunden_Cursor
FETCH NEXT FROM kunden_Cursor INTO @name, @vorname
WHILE @@FETCH_STATUS = 0
BEGIN
  PRINT @name + ' ' + @vorname
  ...
  FETCH NEXT FROM kunden_Cursor INTO @name, @vorname
END
CLOSE kunden_Cursor;
DEALLOCATE kunden_Cursor;
```

Relationen-Scan (full table scan)

Prinzip:



Zürcher Fachhochschule

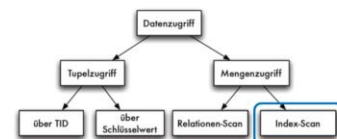
14

Damit die Daten einer einzelnen Relation sequentiell gelesen werden können, werden die Seiten einer bestimmten Relation mittels verketteter Liste (linked list) verknüpft. Es muss daher so lange gelesen werden, bis die letzte Seite erreicht ist.

Das sieht zunächst einfacher aus, als es in Realität ist. Im SQL-Server wird für die Relationen ein gemeinsames File verwendet (Default), ausser die Relation überschreitet eine gewisse Grösse.

Um den Zugriff zu beschleunigen, führt der SQL Server ausserdem eine sogenannte Index Allocation Map (IAM). In dieser sind die Seiten einer Relation aufgelistet. Beim Zugriff auf die Disk optimiert er nun den Zugriff auf die Seiten in Abhängigkeit ihrer physischen Position auf den Disks.

- **Index-Scan** nutzt Index zum Auslesen der Tupel in Sortierreihenfolge (Ausnahme: Hash-Index).
- **Zusätzliche Operation:**
 - `open-index-scan(IndexID, Min, Max) → ScanID`
Min und Max bestimmen den Bereich einer Bereichsanfrage
- **Aufwand für einen bestimmten Wert des Attributes A:**
 $O(\text{lev}_{I(R(A))} + (|r| / \text{val}_{A,r}))$
 - $\text{lev}_{I(R(A))}$ = Anzahl Indexebenen (Blätter des Index sind verkettet)
 - $\text{val}_{A,r}$ = Anzahl verschiedener Werte für das Attribut A in der Relation r



`SELECT * FROM Kunde WHERE Nachname BETWEEN 'Heuer' AND 'Jagellowsk'`

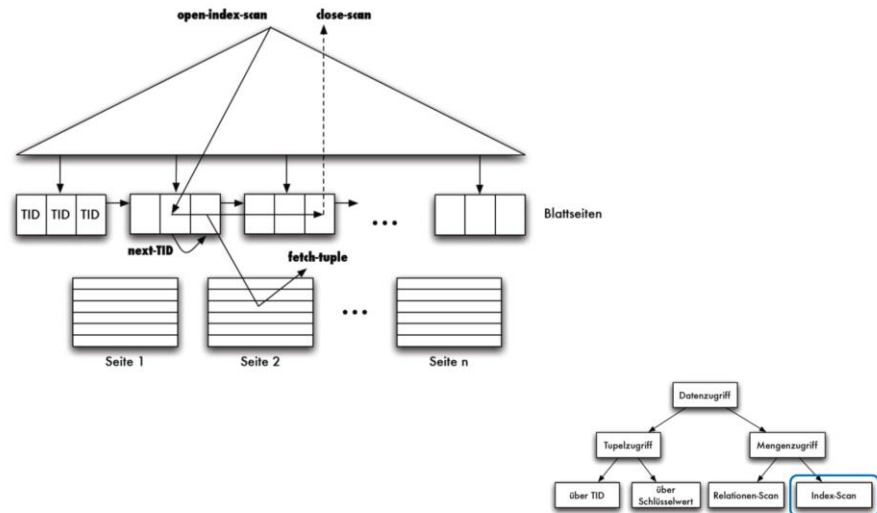
Umgesetzt mittels Operationen:

```

1 currScanID ← open-index-scan(KUNDE-Nachname-IndexID,
  'Heuer','Jagellowsk');
2 currTID ← next-TID(currScanID);
3 while ¬ end-of-scan(currScanID) do
4   currRec ← fetch-tuple(KUNDE-RelationID, currTID);
5   put(currRec);
6   currTID ← next-TID(currScanID);
7 end
8 close-scan(currScanID);
  
```

Ob das Lesen mittels Index oder mittels Full-Table-Scan schneller ist, ist unter anderem davon abhängig, wie gross die Tabelle (ergibt die Anzahl Indexebenen) ist und wie viele verschiedene Werte zum verwendeten Attribut existieren. Wird z.B. nach Geschlecht gesucht und es wird die Mehrzahl der Tupel gelesen, dann ist die Suche über den Index sehr wahrscheinlich ineffizient. In diesem Falle sollte sich der Optimizer des DMBS für den Full-Table-Scan entscheiden.

Der Index-Scan ist besser, falls wenige Daten benötigt werden (Bereichsanfragen, exact-match), aber schlechter beim Auslesen vieler Tupel.



Falls die Daten nach dem Index geclustert vorliegen, kann natürlich, nach dem erstmaligen Einsatz des Indexes, bei weiteren Zugriffen auf den Index verzichtet werden.

(Externes) Sortieren

- Problem: Datenmengen die nicht in Hauptspeicher passen.
- **Mergesort-Verfahren**, Externes Sortieren durch Mischen.
- Ablauf:
 - Relation in gleich grosse Teile teilen (sog. runs), die im Hauptspeicher sortiert werden können. Sortierung mit bekannten Sortieralgorithmen (z.B. Heapsort).
 - n Mischläufe mit Merge-Operation, die zwei Zwischenergebnisse mischt. So lange wiederholen, bis nur noch eine Relation verbleibt.



17

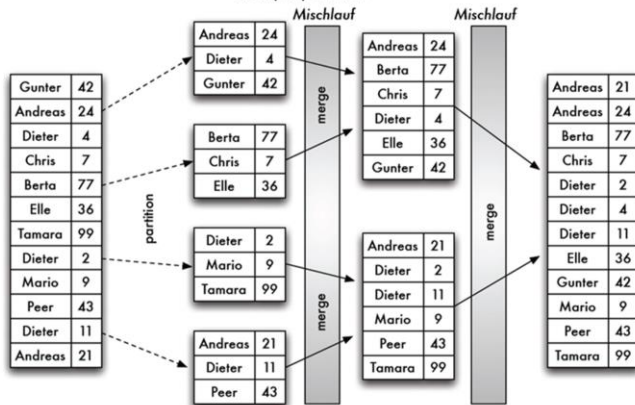
Zürcher Fachhochschule

In der Theorie sind sehr viele Sortieralgorithmen bekannt. Was aber, wenn der Hauptspeicherplatz zur Sortierung nicht genügend gross ist? Wie kann effizient sortiert werden, wenn dies auf externen Medien erfolgen muss?

Herleitung Aufwand siehe übernächste Folie.

(Externes) Sortieren

Sortierung im Hauptspeicher



(Externes) Sortieren

- Initialer Schritt: Jeden Block lesen, sortieren & auf Partition schreiben = $2b_r$
- Kombination von 2 Zwischenergebnissen: n Läufe kombinieren 2^n Partitionen:
 - Initial mindestens $\frac{b_r}{mem}$ Partitionen, daher: $\left\lceil \log_2 \left(\frac{b_r}{mem} \right) \right\rceil$ Mischläufe
 - Jeder Mischlauf liest (b_r) und schreibt (b_r) alle Blöcke (zusammen $2b_r$):

$$2b_r \left(1 + \left\lceil \log_2 \left(\frac{b_r}{mem} \right) \right\rceil \right)$$
 - Mehr als 2 Teilrelationen (\log_2) gleichzeitig mischbar, genau $mem-1$ (\log_{mem-1}):

$$cost_{SORT} = 2b_r \left(1 + \left\lceil \log_{mem-1} \left(\frac{b_r}{mem} \right) \right\rceil \right)$$
 - Bem.: Effizienzsteigerungen und Parallelisierung möglich
 - b_r = Anzahl Blöcke die Tupel beinhalten
 - mem = Puffergrösse in Anzahl der Blöcke



Die Herleitung ist nicht ganz trivial. Grundsätzlich interessieren wir uns nur für das Lesen und Schreiben der Daten, das Sortieren selbst erfolgt im RAM und wird vernachlässigt.

Für den ersten Schritt (das Partitionieren und anschliessende Sortieren) müssen alle Tupel einmal gelesen und einmal geschrieben werden. Dies benötigt $2b_r$ Operationen.

Werden die Daten wie im Bild auf der vorhergehenden Folie wieder zusammengeführt, gibt es $\log_2(b_r/mem)$ Mischläufe (bei 8 Partitionen 3 Mischläufe, bei 9 Partitionen 4 Mischläufe, etc.). Jeder Mischlauf liest und schreibt $2b_r$ Blöcke. Daher werden beim Mischen $2b_r \cdot (\log_2(b_r/mem))$ Seiten gelesen und geschrieben. Berücksichtigen wir noch das Partitionieren und Sortieren sind es $2b_r \cdot (1 + \log_2(b_r/mem))$.

Bei der Berechnung sind wir davon ausgegangen, dass die Schreib- und Leseoperationen des Mischlaufs nacheinander erfolgen. Datenbanksysteme versuchen aber, diese Operationen parallel auszuführen (falls die Hardware dies erlaubt). Im besten Fall kann sogar schon während die Tabelle noch erstellt wird (die ersten Datensätze sind schon gemerged, aber noch nicht alle), bereits wieder mit dem nächsten Mischlauf für diese neue Tabelle begonnen werden. Im theoretisch besten Fall werden ALLE Mischläufe parallel ausgeführt, dann wäre der Aufwand (als Mass für die Zeit) für die Mischläufe nur noch $2b_r$. Der Gesamtaufwand wäre dann Partitionieren/Sortieren + Mischen: $4b_r$.

Das funktioniert natürlich nicht wirklich (es bestehen Latenzzeiten beim Warten auf den Output des vorhergehenden Mischlaufs), ausserdem ist die Parallelität durch die Grösse des Speichers begrenzt, es können nicht alle Tabellen gleichzeitig in den Puffer geladen werden (das war ja das Ausgangs-problem). Es können höchstens mem Tabellen geladen werden, es können also höchstens $mem-1$ Tabellen parallel gemischt werden. Im besten Fall werden nicht \log_2 (je 2 Tabellen werden 'parallel' gemischt), sondern \log_{mem-1} Mischläufe notwendig. In diesem Fall ergibt sich ein Aufwand von $2b_r \cdot (1 + \log_{mem-1}(b_r/mem))$.

- Der Verbund (Join) ist eine der wichtigsten Basisoperationen
- Es gibt viele verschiedene Varianten/Algorithmen:
 - Nested-Loops-Join (NLJ)
 - Block-Nested-Loops-Join
 - Sort-Merge-Join (SMJ) (auch Merge-Join)
 - Hash-Join (→ Lektion 7)
 - ...
- Kostenabschätzung schwierig! Je nach JOIN-Art ('natural-', 'theta-', 'outer-', 'self-', ...) sind unterschiedliche Verfahren anwendbar bzw. optimal.



20

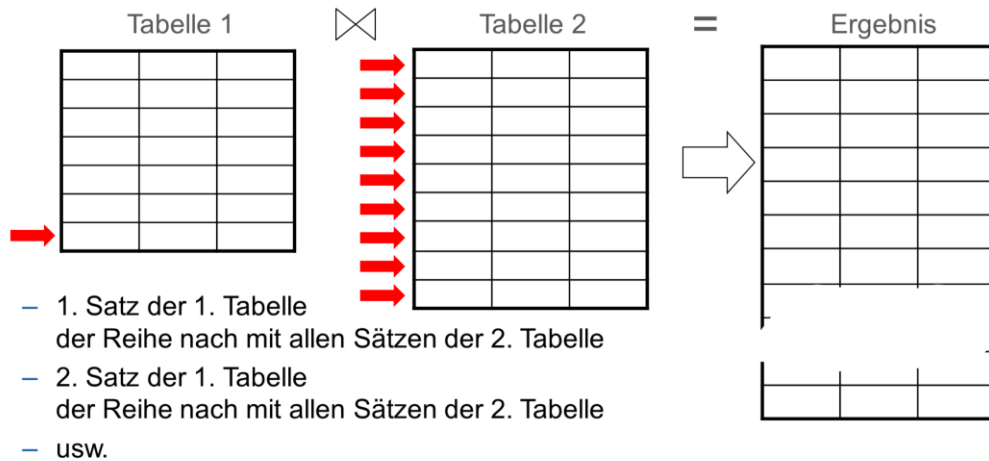
Zürcher Fachhochschule

Der Join ist die einzige Operation, die Tupel aus unterschiedlichen Tabellen verknüpft. Die Join-Operation ist auch eine sehr häufige Operation. Eine effiziente Ausführung der Join-Operation ist daher zentral. Es verwundert damit auch nicht, dass hierfür eine Vielzahl von Algorithmen entwickelt wurden, so dass Datenbanksysteme mehrere Basisalgorithmen für Joins anbieten. Der SQL Server bietet hier z.B. 5 Varianten an (Loop Join = Nested-Loop-Join, Merge-Join und 3 Varianten des Hash-Join).

Der Optimizer versucht aufgrund statistischer Daten die optimale Join-Operation zu wählen. Im SQL-Server ist es trotzdem möglich, die Anwendung eines bestimmten Algorithmus zu erzwingen (z.B. `SELECT * FROM A INNER HASH JOIN B ON A.x = B.y`), davon muss in der Regel aber abgeraten werden.

Verbundoperationen: Nested-Loop-Join (NLJ)

- **Nested-Loop-Join** ('geschachtelte Schleifen'): Prinzip



Verbundoperationen: Nested-Loop-Join (NLJ)

- Verbesserungen:
 - «Geblocktes» Lesen der Sätze beider Tabellen (statt über Tupel über Blöcke iterieren)
 - Auf innere Tabelle s wird über einen Index zugegriffen
- Kosten:
 - ohne Indexunterstützung (geblocktes Lesen):

$$cost_{LOOP} = b_r + \left\lceil \frac{b_r}{mem - 1} \right\rceil \cdot b_s$$

Möglichst viele Blöcke von r im Speicher
 - mit Primärindexunterstützung in s:

$$cost_{LOOP} = b_r + |r| \cdot (lev_{I(S(B))} + 1)$$
 - mit Sekundärindexunterstützung in s:

$$cost_{LOOP} = b_r + |r| \cdot (lev_{I(S(B))} + \frac{|s|}{val_{B,s}})$$



Zürcher Fachhochschule

22

Kosten mit geblockten Lesen:

Die Kosten mit geblocktem Lesen (es werden möglichst viele Blöcke von r in den Speicher gelesen und dann mit den einzelnen Blöcken von s gejoint) lassen sich wie folgt verstehen:

- es müssen alle Blöcke von r genau ein Mal gelesen werden: b_r Lese-Operationen
- ein einzelner Block von s muss so oft gelesen werden, wie der Speicher mit den Blöcken von r gefüllt wird ($b_r/mem-1$). Insgesamt wird also $b_s \cdot (b_r/mem-1)$ lesend auf s zugegriffen.
- Total (schlechtester Fall): $b_r + b_s \cdot (b_r/mem-1)$ Lese-Operationen

Passen sämtliche Blöcke von r und s in den Speicher sind nur $b_r + b_s$ Lese-Operationen notwendig (bester Fall). Schreib-Operationen haben wir hier vernachlässigt, diese müssen gegebenenfalls auch noch einberechnet werden.

Kosten mit Primärindex:

Besteht für das Verbundattribut B zum Relationschema S der Relation s ein B⁺-Baum als Primärindex I(S(B)) (der Index ist eindeutig), dann muss für jedes Tupel in r das zugehörige Tupel in s gesucht und gelesen werden. D.h. es muss $|r| \cdot (lev_{I(S(B))} + 1)$ Mal in s gelesen werden, um alle relevanten Tupel in s zu lesen. Insgesamt muss also $b_r + |r| \cdot (lev_{I(S(B))} + 1)$ gelesen werden; + 1 um die eigentlichen Daten zu lesen.

Kosten mit Sekundärindex:

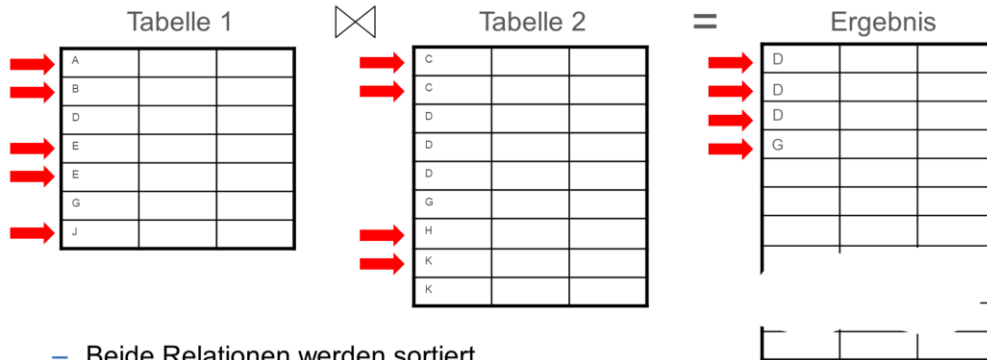
Die Überlegung ist analog zum Primärindex. Nur muss jetzt nicht + 1 gerechnet werden um die eigentlichen Daten zu lesen, es gibt ja mehrere Datensätze je Wert, nämlich $|s| / val_{B,s}$.

Wann lohnt sich der Einsatz eines Primär-, resp. Sekundärindexes? Das lässt sich jetzt anhand dieser Kostenabschätzungen berechnen. Dafür müssen aber b_r , b_s , mem, $|r|$, $|s|$, $lev_{I(S(B))}$ respektive $val_{B,s}$ bekannt sein. Damit der Optimizer diesen Entscheid fällen kann, werden exakt diese statistischen Daten durch das DBMS geführt.

Wann lohnt es sich einen Index einzuführen: Diese Frage kann nicht allein von Joins abhängig gemacht werden – es spielen noch viele weitere Faktoren eine Rolle...

Verbundoperationen: Sort-Merge-Join (SMJ)

- **Sort-Merge-Join: Prinzip**



- Beide Relationen werden sortiert.
- In der Tabelle mit dem niedrigeren Wert wird solange vorgerückt, bis das Join-Kriterium erfüllt ist oder der Attributwert grösser als aktuell in der anderen Tabelle.
- Jetzt dasselbe Vorgehen in der anderen Tabelle usw. bis eine von beiden Tabellen vollständig durchlaufen ist.

23

- Mehrfachwerte:
 - Der Algorithmus muss berücksichtigen, dass in beiden Relationen Werte der Join-Attribute mehrfach auftreten können.
- Kosten falls ein Attribut Schlüsselattribut ist (die Werte sind eindeutig, jeder Datensatz muss höchstens ein Mal gelesen werden):
 - Sortieren plus Lesen je Tabelle:
$$cost_{r_i} = b_{r_i} \log_{mem} b_{r_i} + b_{r_i} \text{ für Relation } r_i \in \{r, s\}$$
 - Kosten über beide Tabellen: $cost_{MERGE} = cost_r + cost_s$
- Kosten falls kein Attribut Schlüsselattribut ist: Kommen den Kosten des Nested-Loop-Join nahe.

Die Kostenabschätzung auf der Folie gilt für den Fall, dass ein Attribut Schlüsselattribut ist (die Werte sind eindeutig) und die Tabellen noch nicht sortiert sind. Dann muss die Tabelle zunächst sortiert werden, der Aufwand hierfür ist $b_r \cdot \log_{mem}(b_r)$. Anschliessend müssen die Datensätze mit einem Aufwand von b_r in der Sortierreihenfolge gelesen werden. Total ist der Aufwand für eine Tabelle daher: $b_r \cdot \log_{mem}(b_r) + b_r$. Für beide Tabellen damit: $b_r \cdot \log_{mem}(b_r) + b_r + b_s \cdot \log_{mem}(b_s) + b_s$. Allfällige Schreib-Operationen haben wir wieder vernachlässigt.

Im schlechtesten Fall, wenn die Werte der Verbundattribute in beiden Tabellen immer den selben Wert hat, wird das kartesische Produkt ermittelt (es muss über die innere Tabelle iteriert werden). D.h. im schlechtesten Fall ergeben sich $(b_r \cdot \log_{mem}(b_r)) + (b_s \cdot \log_{mem}(b_s) + b_r + b_s \cdot (b_r / mem - 1))$ Operationen (2 mal Sortierung plus Aufwand analog Blocked Nested Loop).

Bei der Wahl des geeigneten Basisalgorithmus ist das Datenbanksystem daher dringend auf statistische Daten angewiesen.

- Das nächste Mal: Hash-Join, Optimierung
- Lesen: Lehrbuch Kapitel 8.1 bis 8.4