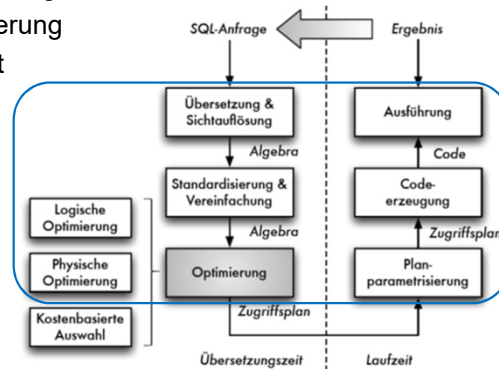


Optimierung

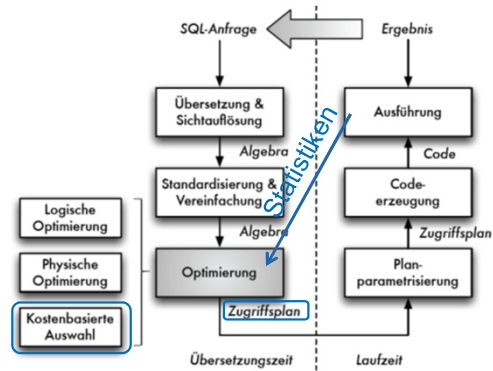
| | |
|-----|----------------------------------|
| 1 L | Einführung |
| 4 L | Datenorganisation Speicherung |
| 4 L | Optimierung |
| 2 L | Transaktionen, Recovery |
| 2 L | Non-Standard Datenbanken |
| 1 L | Repetition, Abschluss |

← "You are here"

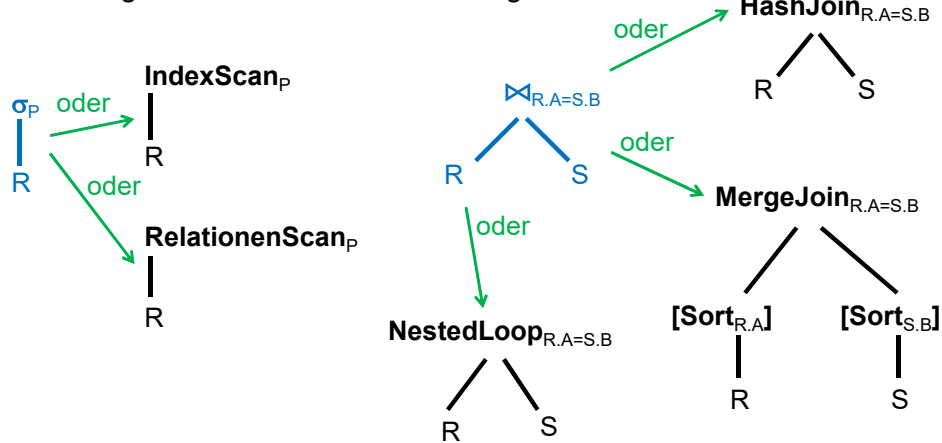
- Grundüberlegungen: Von der SQL-Anfrage zum Ausführungsplan
 - Übersetzung in Parse- und Operator-Baum
 - Standardisierung und Vereinfachung
 - Logische und physische Optimierung
 - Anfragebearbeitung zur Laufzeit



- Weitere Aspekte der Optimierung kennen:
 - Ausführungspläne / QEP («Query Execution Plan»)
 - Kostenbasierte Auswahl
 - Statistiken
 - Performanceaspekte in der Praxis



Umsetzung eines QEP hat mehrere Möglichkeiten:



Es braucht eine **Aufwand-/Kostenabschätzungen**

5

Beim Aufbau des Ausführungsplans bestehen verschiedene Freiheitsgrade:

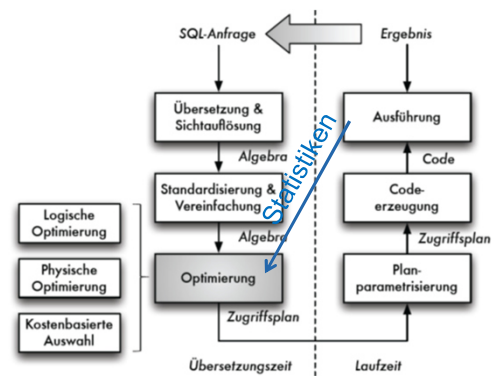
- Reihenfolge und Gruppierung von assoziativen und kommutativen Operatoren: Joins, Vereinigung, Schnittmenge, ... (siehe Regeln aus Lektion 8)
- Wahl eines Algorithmus für jeden Operator
- Zusätzliche Operatoren einbauen, die im logischen Plan nicht auftauchen: Scan, Sort
- Modus des «Datentransports» zwischen Operatoren festlegen: Temporäre Tabelle, Pipeline
- Etc.

Der Optimizer muss eine Variante davon «auswählen». Dazu wird eine Aufwand- resp. Kosten-Abschätzung durchgeführt.

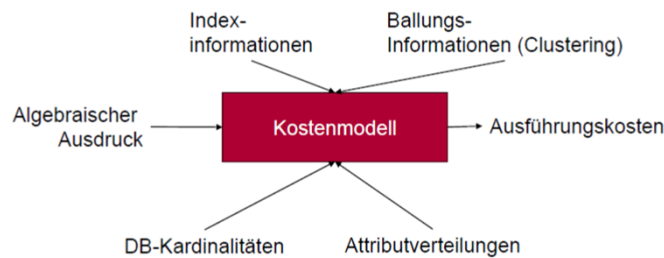
- Ziel: «Billigsten» Plan ausführen
- Idee: Generierung aller denkbaren Pläne und billigsten Plan ausführen.
Bewertung anhand verschiedener Daten:

- Häufigkeiten, Verteilungen, ...
- Verfügbare Ressourcen:
 - Speicher
 - RAM
 - ...

Wie finde ich den «billigsten» Plan?



- RDBMS rechnet für einige Pläne basierend auf einem Kostenmodell die Ausführungskosten aus und wählt dann den «billigsten».
- Kostenmodelle sind von verschiedenen Faktoren abhängig:



7

Es können nicht alle möglichen Anfrageauswertungspläne untersucht werden, der **Suchraum** wird für praktische Probleme sehr rasch viel zu gross («kombinatorische Explosion»).

Die Kosten unterschiedlicher Pläne können sich aber um Größenordnungen unterscheiden, es kommt also auf eine **gute Wahl** an.

Es werden deshalb – je nach RDBMS-Produkt – unterschiedliche Verfahren angewandt, um einen genügend «guten» Plan zu finden. Die Planauswahl erfolgt aufgrund eines **Kostenmodelles**.

In aller Regel funktioniert diese Auswahl ganz gut, in einzelnen Fällen **muss aber manuell eingegriffen werden**, um ein befriedigendes Resultat zu erreichen.

- Zur Kostenberechnung werden statistischen Daten gesammelt:
 - Zu jeder Basisrelation:
 - Anzahl der Tupel ($|r|$ = **Kardinalität der Relation R**), Tupelgrösse, ...
 - Zu (jedem) Attribut:
 - Min / Max, $val_{A,r}$: Anzahl verschiedener Werte für das Attribut A in der Relation r (= **Kardinalität des Attributes**); bei ungleichmässiger Verteilung der Werte → **Werte-verteilung** nötig (**Histogramm**), ...
 - Zum System:
 - Speichergrösse, Bandbreite, I/O Zeiten, CPU Zeiten, ...
- Erstellen und Nachführen der Statistiken ist «teuer»

Um die Kosten verschiedener Pläne ausrechnen zu können benötigt das DBMS eine Fülle von statistischen Angaben (siehe auch Lektion 6). Diese Daten werden in Data Dictionary der DBMS gespeichert.

Das Erstellen und Nachführen dieser Statistiken ist sehr «teuer» und wird deshalb nur sporadisch zu geeigneten Zeiten initiiert, in der Regel in sogenannten «Wartungsfenstern» (→ Aufgabe eines DBA).

Im SQL-Server werden diese statistischen Daten per Default automatisch nach Bedarf aktualisiert (was immer das genau heisst). Falls notwendig kann der Update dieser statistischen Daten auch ausgelöst werden (von Hand oder im Wartungs-Job). Man muss dabei aber bedenken, dass nach einem Update dieser statistischen Daten die Ausführungspläne aller Querys bei der nächsten Ausführung erneuert werden. Dies hat wieder nachteilige Folgen für die Performance. Es gilt also das richtige Mass zu finden.

Begriff Histogramm: Bei einem Histogramm wird für jeden auftretenden Attributwert festgehalten, wie oft dieser Wert in der Tabelle auftritt. Es ist also eine 'Tabelle' aller auftretenden Werte und deren Häufigkeit.

Wesentliches Kostenmerkmal ist **Anzahl der Tupel** im Input:

- Wichtige Frage: Passt die Relation in den Hauptspeicher?
- Kostenmodelle schätzen für jede Operation die Anzahl Ausgabebetupel:
 - **Selektivität**: Anzahl **qualifizierende** Tupel relativ zur Gesamtanzahl
 - Anzahl Ausgabebetupel = Anzahl Eingabetupel x Selektivität

Beispiele:

- Selektivität für das Schlüsselattribut der Relation R ist $\frac{1}{|r|}$
- Für das Attribut A mit $val_{A,r}$ verschiedenen Werten, kann die Selektivität als $\frac{1}{val_{A,r}}$ abgeschätzt werden.

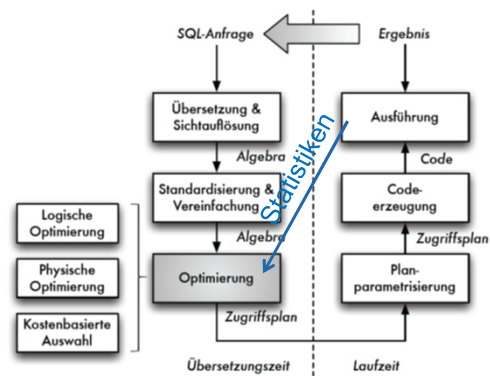
9

Die Selektivität wird auch „Selektivitätsfaktor“ (selectivity factor) genannt, in gewissen Systemen auch einfach Selektivität.

$val_{A,r}$ entspricht der Anzahl Einträge im Histogramm. Ist der zu verwendende Wert von A bekannt, kann mittels Histogramm auch die exakte Selektivität bestimmt werden.

- Ziel: «Billigsten» Plan ausführen
- Idee: Generierung ~~aller~~ denkbarer Pläne und ~~billigsten~~ guten Plan ausführen. Bewertung anhand verschiedener Daten:
 - Häufigkeiten, Verteilungen, ...
 - I/O-Kosten
 - Grössen von Zwischen Ergebnissen

Potentielle Pläne entwickeln und mittels **Kostenmodell** guten Plan auswählen



Was kann/muss man selber tun?

- Ausführungspläne (Interpretation)
- Indexierung:
 - Wann macht ein Index Sinn?
 - Best Practices / Hinweise
 - Indexpflege
- Partitionierung
- Skalierung
- Beeinflussung Optimizer
- Tools zur Analyse

11

Die bisherigen Überlegungen haben vor allem gezeigt, wie das DBMS versucht die Anfragen zu optimieren. Auf den folgenden Folien soll gezeigt werden, mittels welchen Mitteln wir selbst die Anfragen optimieren können.

- Voraussetzungen für Optimierer:
 - **Indexierung**: Erstellen sinnvoller Indexe
 - **Statistiken**: Führen und nachführen der Statistiken
- Was wenn Performanz nicht genügt?
→ Ausführungsplan interpretieren!

Damit der Optimierer eines RDBMS vernünftige Resultate liefert, müssen verschiedene Grundvoraussetzungen durch die Entwickler/Betreiber eines DBMS sichergestellt werden:

- Indexierung:
Das RDBMS erstellt selbständig (zur Laufzeit) keine Indexe, diese müssen manuell erstellt werden. Wir werden anschliessend an die Ausführungsplan diskutieren, wann ein Index sinnvoll ist.
- Statistiken: Das RDBMS muss über adäquate Berechnungsgrundlagen für die Kostenschätzungen verfügen, d.h. diese statistischen Daten müssen periodisch nachgeführt werden.

Was aber, wenn die Performanz noch nicht genügt?

→ Aktuelle Ausführungsplan interpretieren! Und Verbesserungen vornehmen.

Es lohnt sich immer auch, das betroffene, langsame SQL-Statement zu studieren. Häufig sind bei langsamen Abfragen die SQL-Statements ungünstig aufgebaut, so dass überflüssige und redundante Code-Teile enthalten sind. Diese müssen unbedingt erkannt und eliminiert werden.

Informationen in Ausführungsplänen:

- Abfolge der einzelnen Operationen
- angewandten (Basis-)Algorithmen
- statistische Daten (genutzte Indexe, gelesene Seiten u.a.).

Ausführungspläne anzeigen:

- SQL-Anweisungen zur Anzeige statistischer Daten zu SQL-Anweisungen
- Oft bieten Verwaltungswerkzeuge Möglichkeit Pläne grafisch darzustellen

Achtung: braucht Erfahrung QEP zu lesen

13

- Im SQL Server gibt es mehrere Befehle um statistische Informationen zu ausgeführten SQL-Anweisungen zu erhalten, z.B. 'SET STATISTICS IO ON' (zeigt die generierten IO Aufwände auf der Disk an), oder 'SET STATISTICS TIME ON' um Informationen zur Parse-, Compile- und Ausführungszeiten zur erhalten). Mit 'SET STATISTICS XML ON' werden detaillierte Informationen über die ausgeführten SQL-Anweisungen ausgegeben (es handelt sich um den grafischen Ausführungsplan in XML).
- Natürlich kann der Ausführungsplan auch grafisch dargestellt werden – siehe nächste Folie.

Folgende SQL-Anweisung erzeugt nachfolgende Meldungen:

```
SET STATISTICS IO ON;  
SELECT Partner.NamePartner, CustomerOffer.OrderDate FROM UserData.Partner  
JOIN UserData.ProjectObject ON ProjectObject.FK_Client = Partner.PK_Partner  
JOIN UserData.CustomerOffer ON CustomerOffer.FK_ProjectObject =  
    ProjectObject.PK_ProjectObject  
SET STATISTICS IO OFF;
```

Meldungen (wir verzichten auf eine ausführliche Interpretation):

CustomerOffer-Tabelle. Scananzahl 1, logische Lesevorgänge 212, physische Lesevorgänge 6, Read-Ahead-Lesevorgänge 193, logische LOB-Lesevorgänge 0, physische LOB-Lesevorgänge 0, Read-Ahead-LOB-Lesevorgänge 0.

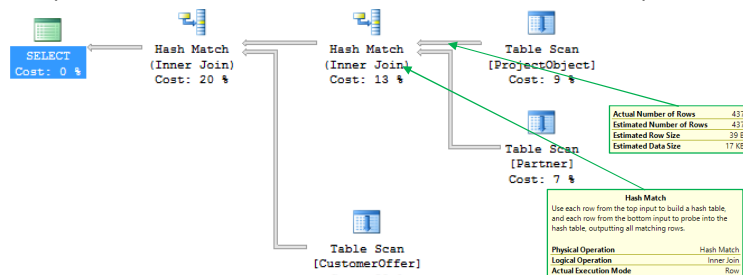
Partner-Tabelle. Scananzahl 1, logische Lesevorgänge 47, physische Lesevorgänge 0, Read-Ahead-Lesevorgänge 24, logische LOB-Lesevorgänge 0, physische LOB-Lesevorgänge 0, Read-Ahead-LOB-Lesevorgänge 0.

ProjectObject-Tabelle. Scananzahl 1, logische Lesevorgänge 70, physische Lesevorgänge 0, Read-Ahead-Lesevorgänge 29, logische LOB-Lesevorgänge 0, physische LOB-Lesevorgänge 0, Read-Ahead-LOB-Lesevorgänge 0.

Begriffe aus Meldungen: Logische Lesevorgänge: aus dem Puffer gelesen; physische Lesevorgänge: vom Datenträger gelesen; Read-Ahead-Lesevorgang: von der Disk in den Puffer geladen; LOB: Large Object (z.B. Image, Text, etc.)

- Beispiele von QEP (lesen von «rechts unten» nach «links oben»)

– SQL Server:



– Oracle:

| Id | Operation | Rows | Bytes | Cost (%CPU) | Time |
|----|-----------------------------|------|-------|-------------|----------|
| 0 | SELECT STATEMENT | 1 | 141 | 3704 (1) | 00:00:45 |
| 1 | SORT GROUP BY | 1 | 141 | 3704 (1) | 00:00:45 |
| 2 | NESTED-LOOPS | 1 | 141 | 3703 (1) | 00:00:45 |
| 3 | NESTED-LOOPS | 1 | 141 | 3703 (1) | 00:00:45 |
| 4 | NESTED-LOOPS | 4 | 396 | 10 (10) | 00:00:01 |
| 5 | TABLE ACCESS BY INDEX ROWID | 1 | 18 | 1 (0) | 00:00:01 |
| 6 | INDEX UNIQUE SCAN | 1 | 0 | 0 (0) | 00:00:01 |
| 7 | VIEW | 4 | 324 | 9 (12) | 00:00:01 |
| 8 | SORT GROUP BY | 4 | 228 | 9 (12) | 00:00:01 |
| 9 | NESTED-LOOPS | 31 | 1767 | 8 (0) | 00:00:01 |
| 10 | MERGE JOIN-CARTESIAN | 819 | 27027 | 8 (0) | 00:00:01 |
| 11 | INDEX RANGE SCAN | 1 | 10 | 2 (0) | 00:00:01 |
| 12 | BUFFER SORT | 819 | 18837 | 6 (0) | 00:00:01 |
| 13 | TABLE ACCESS STORAGE FULL | 819 | 18837 | 6 (0) | 00:00:01 |
| 14 | INDEX UNIQUE SCAN | 1 | 24 | 0 (0) | 00:00:01 |
| 15 | INDEX RANGE SCAN | 1745 | 288 | 0 (0) | 00:00:04 |
| 16 | TABLE ACCESS BY INDEX ROWID | 1 | 42 | 1825 (1) | 00:00:22 |

Hash Match
Use each row from the top input to build a hash table, and each row from the bottom input to probe into the hash table, outputting all matching rows.

| Physical Operation | Hash Match |
|--------------------------------|-----------------|
| Logical Operation | Inner Join |
| Actual Execution Mode | Row |
| Estimated Execution Mode | Row |
| Actual Number of Rows | 437 |
| Actual Number of Batches | 0 |
| Estimated VIO Cost | 0 |
| Estimated Operator Cost | 0.0355691 (13%) |
| Estimated CPU Cost | 0.035569 |
| Estimated Subtree Cost | 0.0809173 |
| Estimated Number of Executions | 1 |
| Number of Executions | 1 |
| Estimated Number of Rows | 355,973 |
| Estimated Row Size | 127 B |
| Actual Rowids | 0 |
| Actual Rewinds | 0 |
| Node ID | 1 |

Output List
[SolvilleERP_Production].[UserData]
[Partner].[NamePartner].[SolvilleERP_Production].[UserData].[ProjectObject].[PK_ProjectObject]

Probe Residual
[SolvilleERP_Production].[UserData].[Partner].[PK_Partner].[SolvilleERP_Production].[UserData].[ProjectObject].[PK_Client]

Hash Keys Probe
[SolvilleERP_Production].[UserData].[Partner].[PK_Partner]

14

Der Ausführungsplan des SQL Server zeigt wie das SQL-Statement auf der vorhergehenden Folie ausgeführt wird:

- Zunächst werden aus den beiden Tabellen ProjectObject (enthält 437 Tupel) und Partner (enthält 478 Tupel) die gesuchten Werte mittels Table Scan (sequentielles Lesen des Files) ermittelt. Obwohl für beide Tabellen geeignete Indexe bestehen, ist der Table Scan effizienter.
- Im 2. Schritt werden die beiden gebildeten Tabellen mittels Hash-Match (Hash Join) zusammengeführt. Da die ProjectObject Tabelle kleiner ist und auch kürzere Datensätze enthält, wird aus dieser Tabelle (obere Tabelle der Hash-Match-Operation) die Hashtabelle gebildet, welche im Memory zwischengespeichert wird.
- Usw.

Für jede Operation und jeden Pfeil können mittels Tool-Tip (grün umrandete Bilder) die Details, inkl. kurzer Erklärung angezeigt werden.

Interessant sind natürlich vor allem Operationen die sehr teuer sind. Lassen sich diese irgendwie vermeiden? Kann ein Index die Abfrage beschleunigen? Kann geclustert werden? Kann ein anderer Basisalgorithmus eingesetzt werden? Ist die Reihenfolge der Basisoperationen ungünstig? Kann ...?

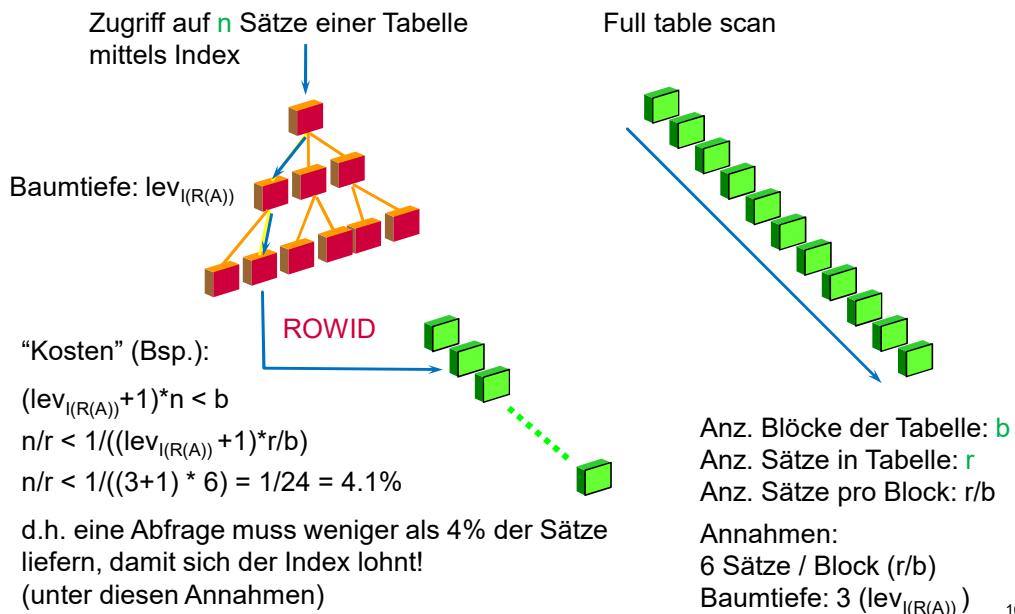
- **Indexe** sind das **wichtigste Hilfsmittel** zur Beschleunigung von Abfragen
- Index enthält nur Daten der Tabelle selbst (redundant)
- Bei DML-Anweisungen müssen diese Redundanzen **konsistent** gehalten werden: Update ↔ Query-Tradeoff (Kompromiss)
- Achtung, DML-Anweisungen profitieren nur eingeschränkt von Indexen:
 - Insert: Gar nicht
 - Update, Delete: Eventuell, wenn nutzbar zur Auswertung der WHERE-Klausel

Indexe sind das weitaus wichtigste Hilfsmittel um eine Abfrageausführung zu beschleunigen, allerdings müssen sie richtig eingesetzt werden.

Bei der Erstellung von Indexen muss berücksichtigt werden, dass diese einen nicht unerheblichen Overhead bei DML-Anweisungen erzeugen. So müssen bei DML-Anweisungen die Index-Daten selbst auch wie normale Daten beim Locking und beim Logging berücksichtigt werden. Dies kann nebst den dadurch verursachten Mehraufwänden auch zu Sperrkonflikten führen (z.B. wenn ein einzelner Schlüsseleintrag in einem Knoten des Baumes geändert wird).

Für Primärschlüssel und Unique-Constraints werden zur Definitionszeit automatisch Indexe erstellt. Für diese müssen daher keine zusätzlichen Index festgelegt werden.

Wann «lohnt» sich ein Index?



16

Im Beispiel sollen Datensätze aus einer Tabelle gesucht werden. Wir gehen dabei davon aus, dass n Datensätze dem Selektionskriterium entsprechen und der Index das Suchkriterium unterstützt.

Beim Zugriff mittels Index müssen daher $(\text{lev}_{I(R(A))} + 1) * n$ Seiten gelesen werden. Beim Full-Table-Scan b Seiten. Der Zugriff mittels Index lohnt sich also, wenn $(\text{lev}_{I(R(A))} + 1) * n < b$ ist.

Wir formen den Ausdruck nun so um, dass auf der linken Seite der Gleichung n/r steht. Damit erreichen wir, dass links die relative Anzahl der Treffer steht (Prozent).

$$n < b / (\text{lev}_{I(R(A))} + 1)$$

$$n/r < b / ((\text{lev}_{I(R(A))} + 1) * r)$$

Indem wir b aus dem Zähler in den Nenner verschieben, erhalten wir

$$n/r < 1 / ((\text{lev}_{I(R(A))} + 1) * r/b)$$

Jetzt können wir die angenommenen Zahlen einsetzen. Daraus ergibt sich, dass sich der Index nur lohnt, wenn weniger als ca. 4% der Sätze gesucht werden. Wären es 100 Datensätzen pro Block, dann sogar erst wenn es weniger als 0.25% sind. Das ist zunächst erstaunlich wenig, berücksichtigt aber nicht, dass der Index vermutlich teilweise oder ganz im Puffer liegt.

Vorsicht: Bei der Berechnung haben wir eine Zugriffsvariante vernachlässigt (dritter Basisalgorithmus, nebst Zugriff mittels Index und Full Table Scan), dass allenfalls ein Index-Scan mit anschließendem Direktzugriff ausgeführt wird.

Wann «lohnt» sich ein Index?

Empfehlungen:

1. Fremdschlüssel indexieren
2. Attribute über die oft gejoint wird indexieren; wird über mehrere Attribute gejoint zusammengesetzten Index verwenden
3. Attribute mit niedriger Kardinalität nicht indexieren
4. Zusätzliche Felder (nicht Index-Felder) in den Index einbinden
5. Eventuell gefilterten Index verwenden
6. Spezialindexe wenn möglich verwenden: räumlich, XML
7. Index eventuell clustern
8. Index eventuell komprimieren
9. Vorsicht bei Änderungen bestehender Indexe: bestehende Abfragen können betroffen sein
10. Vorsicht: Überindexierung kostet Ausführungszeit und Speicherplatz

17

1 – 3.: Fremdschlüssel sollten indexiert werden, insbesondere dann, wenn über Master-Detail gejoint wird. Attribute mit niedriger Kardinalität (Extrembeispiele: Geschlecht, Ja/Nein-Flags, etc.) sollten hingegen nicht indexiert werden (es gibt dafür spezielle Indexstrukturen, hier aber nicht behandelt).

4.: Werden häufig mittels eines bestimmten Attributes Daten gesucht (z.B. Partner-Id) und dabei häufig ein (oder wenige) zusätzliches Attribut verwendet, z.B. der Name des Partner, kann dieses Attribut in den Index integriert werden (Included Column, ohne Teil des Indexes selbst zu sein). Der Optimizer erkennt dann, dass der Zugriff auf die Partner-Id und den Partnername allein durch den Index ausgeführt werden kann.

5: In manchen Systemen können Indexe auch gefiltert werden, d.h. es werden nur Tupel in den Index eingetragen, welche die Filterkriterien erfüllen. Z.B. könnte bei einem Index nach Geburtsdatum ein Filter hinzugefügt werden, so dass Personen ohne Eintrag (Null-Wert) erst gar nicht in den Index aufgenommen werden.

7.: Wird über einen Index sehr häufig zugegriffen (auch häufiger als über den Primärschlüssel), kann die Tabelle zusätzlich nach diesem Index geclustert werden.

8.: Beim Komprimieren können mehr Baum-Knoten pro Block gespeichert werden, dafür steigt der Rechenaufwand... Wie Indexe, können natürlich auch die normalen Daten komprimiert werden.

Der SQL Server gibt auch Tipps, wann sich für eine Tabelle ein Index eignen könnte. Diese Information kann über `sys.dm_db_missing_index_group_stats` abgefragt werden (inkl. Tipps für included Columns). Über `sys.dm_db_index_usage_stats` kann überprüft werden, wie oft ein Index tatsächlich angewendet wurde.

- Zusammengesetzter Primärschlüssel: **Spaltenreihenfolge** wichtig, wenn nur nach einem Teilschlüssel gesucht wird, wird der Index eventuell gar nicht verwendet.

```

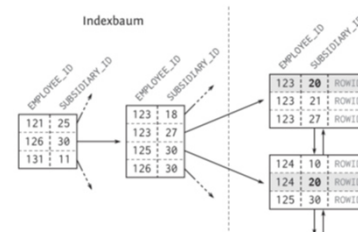
CREATE TABLE employees(
  employee_id NUMBER NOT NULL,
  subsidiary_id NUMBER NOT NULL,
  first_name NVARCHAR(200) NOT NULL,
  last_name NVARCHAR(200) NOT NULL)
    
```

```

CREATE CLUSTERED INDEX employees_pk
  ON employees (employee_id, subsidiary_id);
    
```

```

SELECT first_name, last_name FROM employees
  WHERE subsidiary_id = 20;
    
```



→ Der geclusterte Index nützt in diesem Fall eventuell nichts. Entweder alle Blätter des Index werden durchsucht (Index-Scan), oder es wird direkt ein Table-Scan ausgeführt.

18

Der Index muss im Beispiel nicht zwingend völlig nutzlos sein, denn:

Falls im Beispiel die Anzahl Seiten auf Blatt-Ebene sehr viel kleiner als die Anzahl Seiten der Tabelle selbst ist und die Kardinalität des Attributes `subsidiary_id` hoch ist, kann es sein, dass der Optimizer beschliesst, zunächst die Blattebene des Index nach den gesuchten Tupeln zu durchforsten und anschliessend die Tupel mittels 'Direktzugriff' zu lesen (Ausführungsplan konsultieren).

(Subsidiary = Tochtergesellschaft)

- Gross- und Kleinschreibung bei Vergleichen ist zu beachten:

```
SELECT * FROM employees WHERE last_name = 'WINAND';  
SELECT * FROM employees WHERE last_name = 'Winand';  
SELECT * FROM employees WHERE UPPER(last_name) = UPPER('winand');
```

- Ein Index auf last_name würde im dritten Beispiel nicht benutzt.
Stattdessen benutzen:

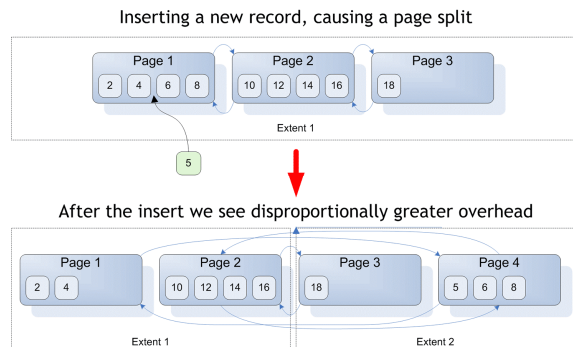
```
CREATE INDEX emp_up_name ON employees (UPPER(last_name));
```

Im SQL Server abhängig von der verwendeten Collation

Beim SQL Server wird der Zeichensatz mittels Collation festgelegt. Dies kann für die ganze Datenbank, aber z.B. auch für einzelne Vergleiche, einzelne Indexe festgelegt werden. Die Collation legt auch fest, wie Gross/Kleinschreibung, Sonderzeichen (ist 'e' = 'é' true?), Sortierung und spezielle Fremdsprachen (z.B. Varianten im Griechisch, Japanisch) behandelt werden sollen. Beim Anlegen einer Datenbank muss daher die zu verwendende Standard-Collation vorsichtig ausgewählt werden!

- Besondere Vorsicht ist auch geboten bei Mustervergleichen.
`SELECT * FROM employees WHERE UPPER(last_name) LIKE 'WIN%D';`
- Mustersuchen können nur dann von Indexen profitieren, wenn das Wildcard-Zeichen **nicht** an **erster Stelle** steht:
`SELECT * FROM employees WHERE UPPER(last_name) LIKE '%IN%';`
- Weitere Empfehlungen und Hinweise zu Indizes sind in der empfohlenen Zusatzliteratur zu finden.

- Im Laufe der Benutzung einer Datenbank werden durch DML-Anweisungen die Daten (und Indexe) **fragmentiert**, das heisst auf (zu) viele Seiten verteilt:



- Einen analogen Effekt gibt es auch bei Dateisystemen zu beobachten.

- schwach besetzte Seiten beeinträchtigen die **Performanz** (mehr Seiten lesen/schreiben)
- Indexe (evtl. auch Daten) regelmässig auf Fragmentierung prüfen
- Sind Schwellwerte (z.B. 30%) überschritten, Indexe **reorganisieren** oder komplett **neu aufbauen**
- Um Betrieb nicht zu stören in Wartungsfenstern durchführen (→ Aufgabe DBA)

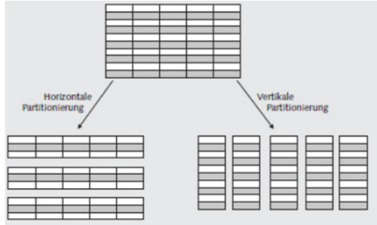
Demo

22

Im SQL-Server gibt der Befehl `dm_db_index_physical_stats` Grössen- und Fragmentierungsinformationen für die Daten und Indizes der angegebenen Tabelle oder Sicht zurück. Es wird dabei empfohlen, einen Index mit einer Fragmentierung zwischen 5 und 30% zu reorganisieren (REORGANIZE) und einen Index mit einer Fragmentierung über 30% neu zu erstellen (REBUILD).

```
SELECT CONVERT (varchar, getdate(), 126) AS runtime,
    mig.index_group_handle, mid.index_handle,
    CONVERT (decimal (28,1), migs.avg_total_user_cost * migs.avg_user_impact *
        (migs.user_seeks + migs.user_scans)) AS 'improvement_measure',
    'CREATE INDEX missing_index_' + CONVERT (varchar, mig.index_group_handle) + '_' +
        CONVERT (varchar, mid.index_handle) + ' ON ' + mid.statement + '
        (' + ISNULL (mid.equality_columns,'')
        + CASE WHEN mid.equality_columns IS NOT NULL
            AND mid.inequality_columns IS NOT NULL
            THEN ',' ELSE '' END + ISNULL (mid.inequality_columns, '')
        + ')'
    + ISNULL (' INCLUDE (' + mid.included_columns + ')', '') AS create_index_statement,
    migs.*,
    mid.database_id,
    mid.object_id
FROM sys.dm_db_missing_index_groups AS mig
INNER JOIN sys.dm_db_missing_index_group_stats AS migs
    ON migs.group_handle = mig.index_group_handle
INNER JOIN sys.dm_db_missing_index_details AS mid
    ON mig.index_handle = mid.index_handle
ORDER BY migs.avg_total_user_cost * migs.avg_user_impact * (migs.user_seeks + migs.user_scans) DESC
```

Beispiel zur Evaluierung, welche Indexe sinnvoll wären.

- Was bestehen für Möglichkeiten, wenn die Performanz immer noch nicht genügt?
 - **Partitionierung:** Daten (grosse Tabellen) aufteilen auf verschiedene Tabellen (v.a. im Bereich data-warehousing oft angewandt), ev. auch Indexe partitionieren
- 
- **Skalierung:**
 - Horizontal (**scale out**): Zusätzliche Rechner einsetzen
 - Vertikal (**scale up**): «Stärkeren» Rechner einsetzen, mehr RAM verwenden, schnellere Speichermedien (SSD, RAID-Verbunde, ...)

Im SQL Server können Tabellen und zugehörige Indexe mittels Filegroups auf mehrere horizontale Partitionen verteilt werden. Die Partitionierung erfolgt aufgrund einer Partitionierungsfunktion. Diese Funktion nutzt der Optimizer dann z.B. auch um einen Equi-Join performanter auszuführen.

Bei der vertikalen Partitionierung werden Attribute einer Tabelle auf zwei oder mehr Tabellen verteilt. Diese Art der Partitionierung wird im SQL Server nicht direkt unterstützt (siehe hierzu Columnstore Index Technologie weiter unten).

Nebst horizontaler und vertikaler Partitionierung, wird im SQL Server auch der Begriff der Hardwarepartitionierung verwendet. Dabei wird die Parallelisierung der Verarbeitung durch Ausnützen der Hardware angesprochen. Z.B. durch Nutzung von mehreren Prozessoren(kernen), oder parallelen Zugriff auf Daten in RAID-Systemen.

Als Spezialfall der vertikalen Partitionierung kann im SQL Server der Columnstore-Index verstanden werden. Dabei werden die Daten nach den Attributen gruppiert (auf einer Seite gespeichert). Diese Speicherform eignet sich vor allem für Data Warehouse-Anwendungen.

Eine «einfache» Art der Skalierung ist natürlich die Steigerung der Performance durch bessere oder schnellere Hardware. Dabei kann im einfachsten Fall die Anzahl der Prozessorkerne, oder der verfügbare Speicher erhöht werden, oder im komplexesten Fall, ein Cluster von Rechnern aufgesetzt werden.

- Beeinflussung Optimizer: Wählt der Optimizer einen schlechten Ausführungsplan, kann dieser beeinflusst werden (Optimizer Hints).

```
SELECT * FROM Customer
INNER MERGE JOIN CustomerAddress
ON Customer.CustomerID = CustomerAddress.CustomerID
WHERE TerritoryID = 5;
```

Beispiel zum SQL Server

- Tools: DBMS haben Werkzeuge um Datenbanken zu analysieren

25

Vorsicht bei Optimizer-Hints: Ändert sich die Datenmenge oder deren Zusammensetzung kann der Hint auch nachteilig sein.

Microsoft stellt den „Database Engine Tuning Advisor“ zur Verfügung. Dieser ist im SQL Server Management Studio zu finden. Mit diesem Tool kann die Datenbank ausgewählt werden, welche für einen bestimmten Workload analysiert werden soll. Nach der Analyse macht das Tool Vorschläge zur Verbesserung der Performance, inkl. der entsprechenden SQL-Statements.

Grundsätze zur Optimierung:

1. Beim Optimieren immer Performance messen – ansonsten ist es schwierig die Verbesserung abzuschätzen.
2. Am besten die Optimierung auf dem produktiven System ausführen. Auf Testsystemen ist ein Vergleich meist schwierig.
3. Bei der Optimierung immer das Gesamtsystem im Auge behalten – sonst ist es an anderer Stelle plötzlich langsam
4. Optimierungsschritte einen nach dem anderen durchführen, ansonsten weiss man nicht, was wie gewirkt hat.

- Das nächste Mal: Transaktionen
- Lesen zur Vorbereitung: Lehrbuch Kapitel 9.1-9.3