Query-Optimierung

Einführung Anfrageoptimierung

- Von der Anfrage (WAS?) zur Auswertung (WIE?)
  - Ziel: kostengünstiger Auswertungsweg
- Einsatz einer großen Anzahl von Techniken und Strategien
  - Logische Transformation von Anfragen
  - Auswahl von Zugriffspfaden
  - Optimierte Speicherung von Daten auf Externspeichern
- Schlüsselproblem:
  - Genaue Optimierung ist i.allg. “nicht berechenbar“
  - Fehlen von genauer statistischer Information
  - Breiter Einsatz von Heuristiken (Daumenregeln)
- Optimierungsziel “möglichst schnelle Anfragebearbeitung“
  - Minimierung der Ressourcen-Nutzung für gegebenen Output (d.h. wenig Seitenzugriffe, wenig Tupel in Zwischenrelationen)

Zu berücksichtigende Kosten

- Kommunikationskosten
  - Anzahl der Nachrichten
  - Menge der zu übertragenden Daten (wichtig für verteilte DBS)
- Berechnungskosten
  - CPU-Kosten
  - Pfadlängen
- I/O-Kosten
  - Seitenzugriffe
- Speicherungskosten
  - Temporäre Speicherbelegung im DB-Puffer und auf Externspeichern
- Kostenarten sind nicht unabhängig voneinander

Phasen der Optimierung

1. *Transformation*:
   Finde geeignete Intern-Darstellung für die Anfrage
2. *Logische Optimierung*:
   Umformung des Anfrageterms aufgrund von Heuristiken
3. *Interne Optimierung*:
   Erzeugung von einem oder mehreren Ausführungsplänen, in denen die abstrakten Algebra-Operatoren durch konkrete Algorithmen ersetzt werden (nutze hierfür den Systemkatalog mit Informationen über vorliegende Index-Strukturen)
4. *Auswahl des günstigsten Plans*
   Basierend auf statistischen Informationen aus dem Katalog Berechnung von Kostenvoranschlägen für jeden möglichen Ausführungsplan und Auswahl des billigsten

Teilziele der Optimierung (Heuristiken)

- Selektionen so früh wie möglich, um Zwischenergebnisse klein zu halten
- Basisoperationen (die wie Selektion und Projektion zusammen-gefaßt werden können) sollten ohne Zwischenspeicherung von Zwischenrelationen als ein Berechnungsschritt realisiert werden
- Nur Berechnungen ausführen, die auch einen Beitrag zum Gesamtergebnis liefern
  - Redundante Operationen, Idempotenzen (z.B. Vereinigung einer Relation mit sich selbst), offenkundig leere Zwischenrelationen aus Ausführungsplänen entfernen
- Zusammenfassen gleicher Teilausdrücke - ermöglicht Wiederverwendung von Zwischenergebnissen

Optimierer von System R (IBM)

- Einfluß:
  - Erste kommerzielle Implementierung eines Query-Optimierers
  - Vorbild für viele heutige Systeme, meistverbreiteter Optimierer
  - Gut geeignet bis zu 10 Joins
- Kostenschätzung:
  - Statistik, die in Systemkatalogen geführt wird, ist Grundlage für die Schätzung von Operationskosten und der Ergebnismengen
  - Berücksichtigt Kombination von CPU- and I/O-Kosten
- Pipelining:
  - *Pipelining* des Outputs von einem Operator zum nächsten Operator ohne Zwischenspeicherung in einer temporären Relation
  - Beispiel: Kopplung einer Selektion mit einer anschließenden Projektion

Beispiel-Schema

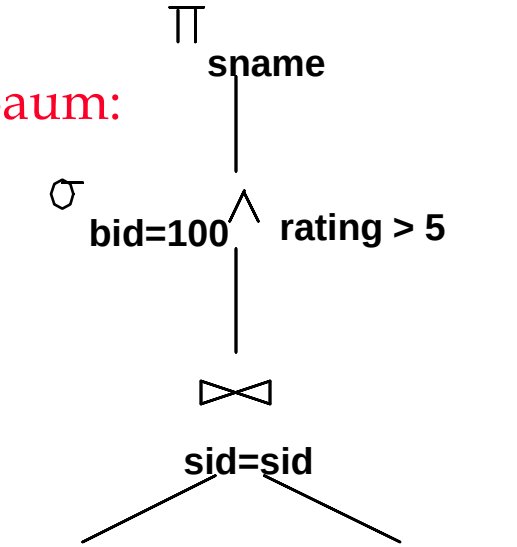
Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- Relationen über Segler und die Reserviert-Beziehung zwischen Seglern und Booten (*rname* hinzugefügt, Name der Reservierung)
- Reserves:
  - Jedes Tupel ist 40 Bytes lang, 100 Tupel pro Seite, 1000 Seiten
- Sailors:
  - Jedes Tupel ist 50 Bytes lang, 80 Tupel pro Seite, 500 Seiten

Motivierendes Beispiel

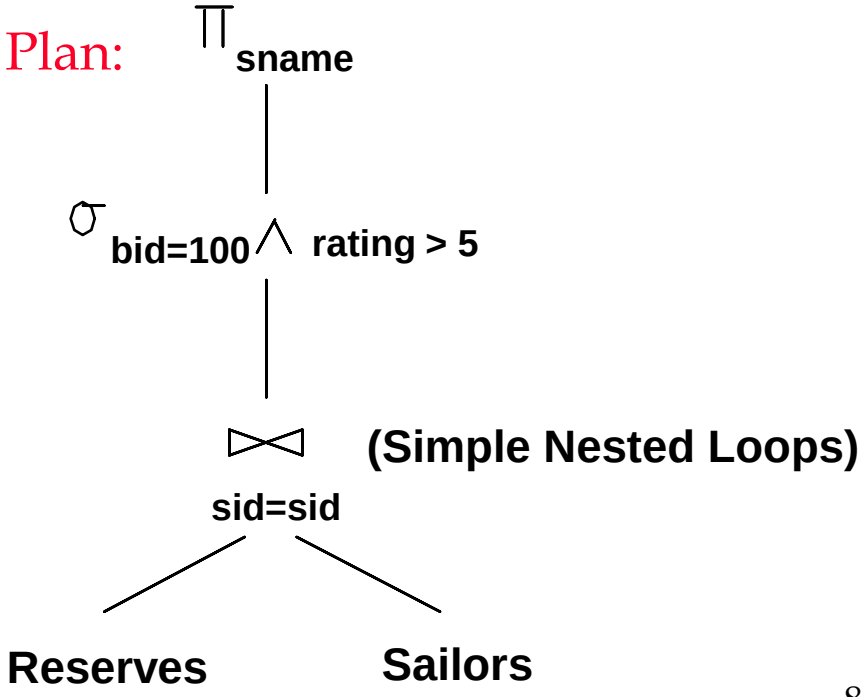
```
SELECT  S.sname
FROM  Reserves R, Sailors S
WHERE  R.sid=S.sid AND
   R.bid=100 AND S.rating>5
```

RA Baum:

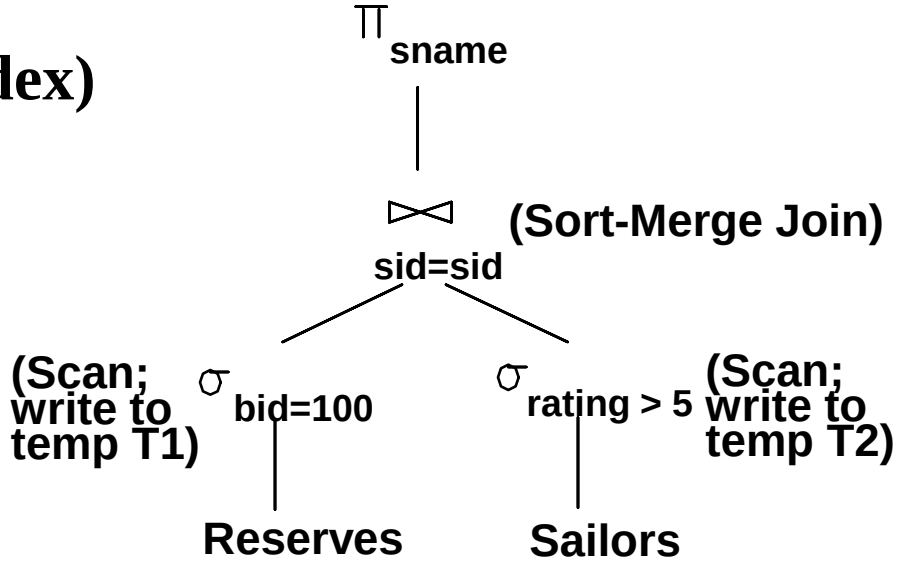


- Kosten: 500+500*1000 I/Os = 500500
- Direkte Auswertung = schlechtester Plan!
- Verzichtet auf verschiedene Möglichkeiten: Selektion früher ausführen, kein Gebrauch von Indexen etc.
- *Optimierungsziel:* Finde effizientere Pläne, die die gleiche Antwort berechnen

Plan:



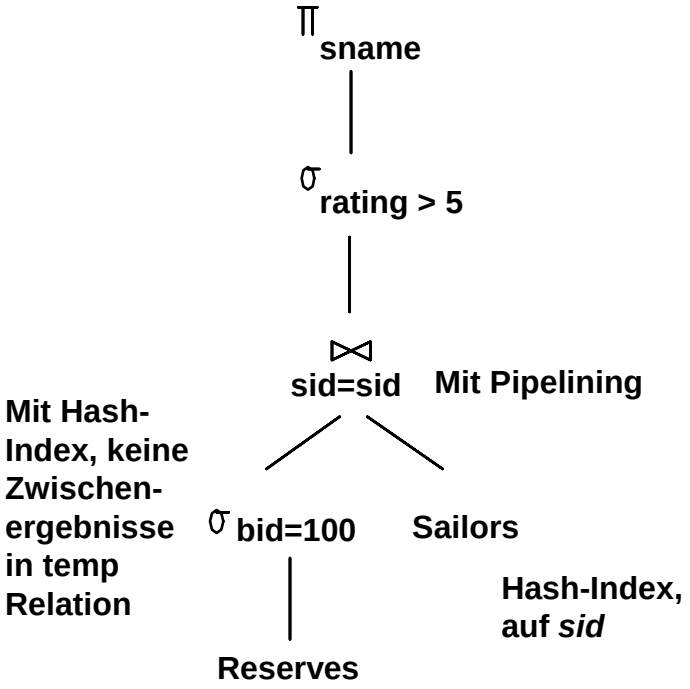
Alternativer Plan 1 (Ohne Index)



- ***Hauptunterschied:*** *Selektion vorziehen*
- Mit 5 Puffer-Seiten, sind die Kosten für diesen Plan:
  - Scan Reserves (1000) + Write temp T1 (10 Seiten, wenn wir 100 Boote haben, gleichmäßige Streuung der Reservierungen über alle Boote)
  - Scan Sailors (500) + Write temp T2 (250 Seiten, wenn wir 10 Ratings haben)
  - Sortieren T1 in 2 Pässen (2*2*10), Sortieren T2 in 4 Pässen (2*4*250), Merge (10+250)
  - Gesamt: 4060 Page I/Os.
- Mit Block Nested Loop Join (BNL):
  - Join-Kosten = Scan T1 (10) + Scan T2 (4*250); mit Blockgröße 3
  - Selektionskosten = 1010 + 750 (siehe oben)
  - Gesamt: kosten = 2770.
- Wenn wir Projektion vorziehen, T1 hat nur *sid*, T2 nur *sid* und *sname*:
  - T1 paßt auf 3 Seiten, Kosten vom BNL fallen unter 250 Seiten, insgesamt < 2000.

Alternativer Plan 2 (Mit Index)

- Mit geclustertem Index auf *bid* in Reserves, erhalten wir 100,000/100 = 1000 (geordnete) Tupel auf 1000/100 = 10 Seiten
- Index Nested Loop (INL) mit *Pipelining* (äußere Relation ist nicht materialisiert)
  - Ausprojizieren der überflüssigen Felder von der äußeren Relation hilft nicht (da Join nicht materialisiert)
- Join-Spalte *sid* ist ein Schlüssel für Sailors
  - Höchstens ein matchendes Tupel, ungeclusterter Index auf *sid* somit OK



- Entscheidung, die Prüfung der Selektionsbedingung rating > 5 erst nach dem Join auszuführen, wegen der Verfügbarkeit des Index *sid* von Sailors
- Kosten: Selektion der Tupel in Reserves (10 I/Os); für jedes dieser Tupel Suche nach matchenden Tupeln in Sailors (1000*1.2) 1.2 = Retrieval-Kosten über Hash-Index, insgesamt 1210 I/Os

Query-Blöcke: Einheiten der Optimierung

- Eine SQL-Query wird übersetzt in eine Menge von *Query-Blöcken*, und diese werden jeweils optimiert zu einem Zeitpunkt
- Geschachtelte Blöcke (nested block) werden i.allg. als Aufruf einer Subroutine behandelt, ausgeführt pro äußerem Tupel (vereinfacht dargestellt!)
- Für jeden Block sind die untersuchten Pläne:
  - Alle verfügbaren Zugriffsmethoden für jede Relation in der FROM-Klausel
  - Alle Join Trees (d.h. alle Möglichkeiten, die Relationen gleichzeitig zu verbinden, mit der inneren Relation in der FROM-Klausel und allen möglichen Permutationen der Relationen und Join-Methoden)

```
SELECT S.sname
FROM Sailors S
WHERE S.age IN
    (SELECT MAX (S2.age)
     FROM Sailors S2
     GROUP BY S2.rating)
```

Äußerer Block Nested Block

Geschachtelte Anfragen

- Geschachtelter Block wird unabhängig optimiert, das äußere Tupel liefert die Selektionsbedingung
- Äußerer Block wird optimiert, geschachtelter Block wird mehrfach aufgerufen (um *sid* zu testen)
- Implizite Ordnung dieser Blöcke bedeutet, daß gute Strategien nicht untersucht werden
- Optimierer findet äquivalente effizientere Queries nicht!
  Die nicht-geschachtelte Version der Query wird typischerweise besser optimiert.
- Optimierer untersucht den Systemkatalog, um Informationen zu erhalten über Typen und Länge der Felder, Statistik der Relationen und Zugriffspfade (Indexe).

```
SELECT S.sname
FROM Sailors S
WHERE EXISTS
  (SELECT *
   FROM Reserves R
   WHERE R.bid=103
   AND R.sid=S.sid)
```

Geschachtelter Block:

```
SELECT *
FROM Reserves R
WHERE R.bid=103
  AND R.sid= outer value
```

Äquivalente nichtgeschachtelte Query:

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid
  AND R.bid=103
```

Abschätzung der Kosten

- Für jeden Ausführungsplan Kostenschätzung erforderlich:
  - Schätzung der Kosten jeder Operation im Ausführungsplan erforderlich:
    - Hängt von der Kardinalität des Inputs ab (Anzahl Tupel)
    - Abschätzung der Kosten von Operationen wurde diskutiert (z.B. Sequential Scan, Index Scan, Joins, etc.)
  - Schätzung der Größe des Ergebnisses für jede Operation im Baum:
    - Verwende Informationen über die Input-Relationen
    - Bei Selektionen und Joins nehme die Unabhängigkeit der Prädikate an
- System R Ansatz der Kostenschätzung
  - Ungenau, aber pragmatisch und praxistauglich
  - Es gibt kompliziertere Algorithmen heute

Statistik und Kataloge

- Informationen über Relationen und Indexe, die an Query beteiligt sind. *System-Katalog (Data Dictionary)* enthält typischerweise:
  - Kardinalität
    - Anzahl Tupel NTuples(R) für jede Relation
  - Größe
    - Anzahl Seiten NPages(R) für jede Relation
  - Index-Größe
    - Anzahl Seiten NPages(I) für jeden Index
  - Index-Kardinalität
    - Anzahl unterschiedlicher Schlüsselwerte Nkeys(I)
  - Index-Höhe
    - Anzahl der Nicht-Blatt-Stufen für jeden Baum-Index I Height(I)
  - Index-Bereich
    - unterer / oberer Schlüsselwert Low(I) / High(I) für jeden Index
- Kataloge werden periodisch aktualisiert
  - Anpassung des Katalogs bei jeder Datenänderung zu teuer; Näherungswerte reichen, leichte Inkonsistenz somit OK
- Manchmal noch detailliertere Informationen erfaßt (z.B. Histogramme von Feldwerten)

Größenschätzung und Reduktionsfaktoren

```
SELECT  attribute list
FROM  relation list
WHERE  term1 AND ... AND termk
```

- Nimm einen Query-Block:
- Maximale # Tupel im Ergebnis ist Produkt der Kardinalitäten der Relationen in der FROM Klausel
- *Reduktionsfaktor (RF)* gehört zu jedem *Term*, drückt aus, inwieweit durch diesen Term die Größe des Ergebnisses verringert wird, manchmal auch als *Selektivität* bezeichnet
  Kardinalität des Resultats = Max # Tupel * Product aller RFs.
  - Implizite Annahme daß die *Terme unabhängig* sind
  - Term *col=value* hat RF *1/NKeys(I),* mit Index I auf *col*
  - Term *col1=col2* hat RF *1/MAX(NKeys(I1), NKeys(I2))*
  - Term *col>value* hat RF *(High(I)-value)/(High(I)-Low(I)),* mit Index I auf col (approximiert)

Logische Optimierung

- Phase der Optimierung ohne Zugriff auf das interne Schema und die statistischen Daten des Systemkatalogs
- Keine Berücksichtigung der Größe der Relationen
- Keine Berücksichtigung von Indexstrukturen
- Algebraische Optimierung
  - Basiert auf Termersetzung von Termen der Relationenalgebra anhand von Algebraäquivalenzen
  - Äquivalenzen als Ersetzungsregel eingesetzt, um anhand von heuristischen Vorgaben die Anfrage in die gewünschte Form zu bringen (siehe nachfolgende Folien)
  - Auch als regelbasierte Optimierung in kommerziellen Systemen bezeichnet (Regeln = Ersetzungsregeln)
- Tableau-Optimierung
  - Exakte Methode zur Minimierung der Anzahl Joins in Anfrage-ausdrücken

Algebraische Äquivalenzen

- Erlaubt uns, die Auswahl verschiedener Reihenfolgen beim Join und die bevorzugte Ausführung von Selektion und Projektion vor einem Join

- *Selektion*: $\sigma_{c1 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}(\ldots \sigma_{cn}(R))$ (*kaskadiert*)

  $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$ (*kommutativ*)

- *Projektion*: $\pi_{a1}(R) \equiv \pi_{a1}(\ldots(\pi_{an}(R)))$ *(kaskadiert)*

- *Join*: $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$ *(assoziativ)*

  $(R \bowtie S) \equiv (S \bowtie R)$ *(kommutativ)*

Es gilt: $R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S$

Weitere Äquivalenzen

- Projektion ist kommutativ mit einer Selektion, die nur Attribute nutzt, die von der Projektion beibehalten werden
- Selektion zwischen Attributen der zwei Argumente eines Kreuzprodukts macht daraus einen Join
- Eine Selektion auf Attributen nur von R kommutiert mit:

$$R \bowtie S \text{ (d.h. } \sigma\ (R \bowtie S) \equiv\ \sigma\,(R)\ \ \bowtie S\ )$$

- Auch wenn eine Projektion einem Join von R und S folgt, können wir diese vorziehen durch Beibehalten der Attribute von R (und S), die vom Join benötigt werden

Bestimmung alternativer Pläne

- Es gibt 2 Fälle:
  - Einzel-Relationen-Pläne
  - Mehr-Relationen-Pläne
- Bei Anfragen auf einzelnen Relationen bestehen die Anfragen aus einer Kombination von Selektion, Projektion und Aggregationsoperatoren:
  - Jeder verfügbare Zugriffspfad (File Scan / Index) wird untersucht und derjenige mit den geschätzten geringsten Kosten ausgewählt
  - Die unterschiedlichen Operationen werden im wesentlichen zusammen ausgeführt, zum Beispiel:
    - Nutzung eines Index für eine Selektion
    - Projektion auf den erhaltenen Tupeln
    - Weitersenden der resultierenden Tupel an eine Aggregations-funktion ohne Materialisierung der Zwischenrelation (Pipelining)

Kostenschätzungen für Einzel-Relationen-Pläne

- Index I auf dem Primärschlüssel erfüllt Selektionsbedingung:
  - *Kosten: Height(I)+1 für B+ Baum, ca. 1.2 für Hash-Index*
- Geclusterter Index I, der einer oder mehreren Selektions-bedingungen entspricht:
  - *(NPages(I)+NPages(R)) * Produkt der RF's der passenden Selects*
- Nicht-geclusterter Index I, der eine oder mehrere Selektions-bedingungen erfüllt:
  - *(NPages(I)+NTuples(R)) * Produkt der RF's der passenden Selects*
- Sequentieller Scan der Datei:
  - *NPages(R).*

Bemerkung: Typischerweisen erfolgt keine Eliminierung von Duplikaten bei Projektion! (Ausnahme: Wenn durch DISTINCT-Klausel explizit durch den Benutzer gewünscht)

Beispiel

```
SELECT S.sid
FROM Sailors S
WHERE S.rating=8
```

- Mit einem Index auf *rating*:
  - (1/NKeys(I)) * NTuples(R) = (1/10) * 40000 Tupel gelesen
  - Geclusterter Index: (1/NKeys(I)) * (NPages(I)+NPages(R)) = (1/10) * (50+500) Seiten gelesen (= Kosten)
  - Ungeclusterter Index: (1/NKeys(I)) * (NPages(I)+NTuples(R)) = (1/10) * (50+40000) Seiten gelesen
- Mit einem Index auf *sid*:
  - Müßten alle Tupel/Seiten lesen. Mit einem geclusterten Index: Kosten 50+500, mit einem ungeclusterten Index: 50+40000.
- Mit einem File Scan:
  - Alle Seiten lesen (500).

Mehr-Relationen-Pläne*

- Mehrere Relationen in der FROM-Klausel
- Untersuchen aller Einzel-Relationen-Pläne
  - Selektion und Projektion so früh wie möglich
  - Alle möglichen Zugriffsmethoden untersuchen (File Scan, Index etc.)
  - Auswahl des billigsten Ausführungsplanes für jede mögliche Ergebnisreihenfolge (z.B. File Scan für unsortierte Ausgabe von Tupeln, Index für sortierte Ausgabe)
- Für jeden 1-Relation-Plan: untersuche alle Möglichkeiten, eine weitere Relation (als innere Relation) zu joinen. Ermittle den besten Plan für jede mögliche Ergebnisreihenfolge.
- Für jeden 2-Relationen-Plan, der im vorigen Paß erstellt wurde, untersuche alle Möglichkeiten, eine andere Relation zu joinen etc.
- Analoges Vorgehen für n-Relationen-Plan

Query-Optimierung in Oracle

- Regelbasierte Optimierung (logische Optimierung):
  - keine kostenbasierte Auswahl
  - Auswahl des genutzten internen Zugriffsplans erfolgt anhand von Prioritäten zwischen Operationen, die auf Heuristiken beruhen
- Kostenbasierte Optimierung
  - Auswahl des Zugriffsplans erfolgt anhand von Statistiken über geeignete Tabellen und Indexe, die mittels analyze-Kommandos angelegt werden können
- Anlegen von Statistiken und Histogrammen
  - Hoher Aufwand, der den Betrieb laufender Anwendungen stören kann (Verantwortung des DB-Admin)
    - Exakte Statistik (Lesen der gesamten Tabelle)
    - Schätzung durch zufällige Auswahl von Tupeln

Query-Optimierung in Oracle (Forts.)

Ermittlung einer Statistik für Mitarbeiter Tabelle auf der Basis von 20 Prozent der Tupel

```
ANALYZE TABLE Mitarbeiter
 ESTIMATE STATISTICS
 SAMPLE 20 PERCENT;
```

Berechnung eines Histogrammes für Gehaltswerte mit 10 Buckets

```
ANALYZE TABLE T_Mitarbeiter
COMPUTE STATISTICS FOR COLUMNS Gehalt
SIZE 10;
```

Zugriff auf Ausführungspläne

```
EXPLAIN PLAN
SET statement_id = ‘test‘
FOR
SELECT * FROM t_teil
WHERE pk_teil_id > 10;
```

Anlegen einer Tabelle *Plan_Table* erforderlich, Zugriff auf diese Tabelle mit statement_id = ´‘test‘

Beeinflussung des Oracle-Optimierers

- Steuerung des Optimierungsmodus
  - Bei der Konfiguration des DBMS (Initialisierungsdatei init.ora)
  - Durch Setzen von Parametern für einzelne Sitzungen mittels alter session
  - Durch *Hints* für Einzelanfragen
- Optimierungsmodi
  - choose: Auswahl zwischen regelbasierter und kostenbasierter Optimierung erfolgt durch das System
  - all_rows: Optimiert wird bezüglich der Bereitstellung des Gesamtergebnisses
  - first_rows: Optimiert wird bezüglich der Zeit der Bereitstellung des ersten Tupels
  - rule: Beschränkung auf regelbasierte Optimierung
- Beispiel:

```
ALTER SESSION
  SET optimizer_goal = first_rows
```

Hints in Oracle

- Hints beeinflussen die Optimierung einzelner Query-Blöcke:
  - SELECT, UPDATE, DELETE
  - Subquery einer komplexen Anfrage
- Hints werden in SQL-Kommentaren eingefügt
- Fehlerfreie Definition der Hints notwendig, ansonsten Interpretation als Kommentar
- Beispiel: Einsatz eines full-Hints
  - Erzwingt einen vollen Table-Scan, ein vorhandener Index wird somit nicht ausgenutzt

```
SELECT  /*+ full(a) Index auf ProdNr
                wird nicht verwendet!  */
          ProdNr, ProduktName
FROM T_Prod a
WHERE ProdNr = 47110815;
```

Zusammenfassung

- Query-Optimierung ist bedeutende Aufgabe eines DBMS
- Grundverständnis der Optimierer notwendig für DB-Anwendungsprogrammierer und DB-Administrator
  - Einfluß des Datenbank-Entwurfs (Relationen, Indexe) auf die Performance einer bestimmten Menge von Anfragen wird klar!
  - DBMS bietet Eingriffsmöglichkeiten in die Query-Optimierung auf verschiedenen Ebenen (Optimierungsziel, Optimierungsstrategie)
- Query-Optimierung hat 2 Hauptbestandteile:
  - Untersuche eine Menge von alternativen Ausführungsplänen
    - Suchraum verkleinern durch Heuristiken
  - Abschätzung der Kosten jedes Planes, der untersucht wird:
    - Abschätzung der Größe des Ergebnisses und der Kosten für jeden Knoten im Ausführungsbaum
    - Issues: Statistik, Indexe, Implementation der Operatoren