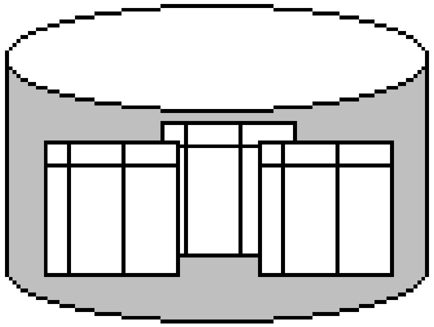
ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

Συστήματα Διαχείρισης Βάσεων Δεδομένων

Φροντιστήριο 8: Query Optimization and Tuning in Oracle

Δημήτρης Πλεξουσάκης
Τμήμα Επιστήμης Υπολογιστών

QUERY OPTIMIZATION & TUNING IN ORACLE DATABASES



Part I:
Query Optimization & Tuning Hints

Query Tuning Hints

- *Avoid redundant DISTINCT*
- *Avoid HAVING when WHERE is enough*
- *Avoid using intermediate relations (tables)*
- *Optimize set difference queries*
- *Change nested queries (subqueries) to joins*
- *Avoid complicated correlation subqueries*
- *Join on clustering and integer attributes*
- *Avoid views (pseudotables) with unnecessary joins*
- *Maintain frequently used aggregates*
- *Avoid external loops*
- *Avoid cursors*
- *Retrieve needed columns only*
- *Use direct path for bulk loading*

Avoid Redundant DISTINCT

```
SELECT DISTINCT <expression>
FROM Employee
WHERE dept = 'information systems'
```

where **<expression>** *== ssnum or ssnum1, ssnum2, …*

- *DISTINCT usually entails a* **<u>sort</u>** *operation**
  - *Slows down query optimization because one more "interesting" order to consider.*
- *Remove if you know the result has no duplicates (or duplicates are acceptable) or if answer contains a (primary or foreign) key.*
- *Intuitively, when multiple expressions are provided in the DISTINCT clause then the query will retrieve all unique combinations for the expressions listed.*

**Depends on many factors (i.e. database version, query expression etc).*

Newer versions (>= Oracle DB 10g) use **<u>hash</u>** *instead of* **<u>sorting</u>**.

Avoid HAVING when WHERE is enough

```
SELECT MIN(E.age)
FROM Employee E
GROUP BY E.dno
HAVING E.dno = 102
```

```
SELECT MIN(E.age)
FROM Employee E
WHERE E.dno = 102
GROUP BY E.dno
```

The **WHERE** *clause will filter or limit rows as they are selected from the table, but* ***before*** *grouping is done. The* **HAVING** *clause will filter rows* ***after*** *the grouping.*

HINT: *Consider DBMS's use of index when writing arithmetic expressions: E.age = 2*D.age will benefit from index on E.age, but might not benefit from index on D.age!*

Avoid Using Intermediate Relations (Tables) - I

```
SELECT * INTO Temp
FROM Emp E, Dept D
WHERE E.dno = D.dno
  AND D.mgrname = ‘Joe’
```

and

```
SELECT T.dno, AVG(T.sal)
FROM Temp T
GROUP BY T.dno
```

vs.

```
SELECT E.dno, AVG(E.sal)
FROM Emp E, Dept D
WHERE E.dno = D.dno
  AND D.mgrname = ‘Joe’
GROUP BY E.dno
```

Avoid Using Intermediate Relations (Tables) - II

- *Creating the Temp table causes frequent updates to catalog (i.e. database).*
  - *The columns of the newly created table inherit the column names, their data types, whether columns can contain null values or not, and any associated IDENTITY property from the source table. However, the* **SELECT INTO** *clause does have some restrictions: it will* **not** *copy any constraints, indexes, or triggers from the source table*[*].
- *Materialization of the intermediate relation Temp consumes resources (even if it takes place in-memory).*

[*] *[**Source:** Beginning C# 2008 Databases: From Novice to Professional By Vidya Vrat Agarwal, James Huddleston, Ranga Raghuram]*

Optimizing Set Difference Queries (I)

Suppose you have to select all of the employee's that are not account representatives:

Table1 (s_emp)

soc_number	last_name	first_name	salary

Table2 (s_account_rep)

soc_number	last_name	first_name	region

```
SELECT soc_number FROM s_emp
   MINUS
SELECT soc_number FROM s_account_rep;
```

The above query is slow because the minus has to select distinct values from both tables.

Optimizing Set Difference Queries (II)

```
SELECT soc_number
FROM s_emp
WHERE soc_number NOT IN
           (SELECT soc_number
            FROM s_account_rep);
```

The above query is faster but we are not joining and are not taking advantage of indexes.

Optimizing Set Difference Queries (III)

```
SELECT /*+ index(t1) */ soc_number
FROM s_emp t1
WHERE NOT EXISTS
            (SELECT /*+ index(t1) index(t2) */  *
             FROM s_account_rep t2
             WHERE t1.soc_number = t2.soc_number);
```

Regardless of indexes, the above query is generally preferred when having to choose between [NOT] IN and [NOT] EXISTS since [NOT] EXISTS does not take into account any null values present.

Challenge time: *Make it faster!!!*

More challenge time: *If both NOT IN and NOT EXISTS produce same execution plans which one will you prefer and why, why, WHY???* ☺

Optimizing Set Difference Queries (IV)

```
SELECT /*+ index(t1) */ soc_number
FROM s_emp t1
LEFT OUTER JOIN s_account_rep t2
  ON t1.soc_number = t2.soc_number
  WHERE t2.soc number IS NULL
```

We should always default to NOT EXISTS.

The execution plans may be the same at the moment but if either column is altered in the future to allow NULLs the NOT IN version will need to do more work (even if no NULLs are actually present in the data) and the semantics of NOT IN if NULLs are present are unlikely to be the ones you want anyway.

Change Nested Queries (subqueries) to Joins

```
SELECT ssnum
FROM Employee E
WHERE E.dept IN
  (SELECT E.dept FROM Techdept)
```

An index may or may not be applied on E.dept.

```
SELECT [DISTINCT] ssnum
FROM Employee E, Techdept TD
WHERE E.dept = TD.dept
```

In most cases **JOINs** *are faster than sub-queries and it is very rare for a sub-query to be faster.*

Tradeoff: *Readability vs Speed!*

Avoid Complicated Correlation Subqueries (I)

```
SELECT ssnum
FROM Employee e1
WHERE salary =
    (SELECT MAX(salary)
     FROM Employee e2
     WHERE e2.dept = e1.dept
```

The subquery references the dept in the outer query. The value of the dept changes by row of the outer query, so the DB must rerun the subquery for each row comparison. This has a significant performance impact on the execution time of the query, and for that reason, correlated subqueries should be avoided if possible.

Avoid Complicated Correlation Subqueries (II)

```
SELECT MAX(salary) AS bigsalary, dept
INTO Temp FROM Employee
GROUP BY dept

SELECT ssnum FROM Employee E, Temp T
WHERE salary = bigsalary
AND E.dept = T.dept
```

Nowadays, most RDBMS makes this efficient although legacy ones made it expensive to use it (i.e. most DBs used the naive implementation of nested loops between query blocks whereas MS SQL 2000 used hashed joins)[*].

[*] *[**Source:** Orthogonal Optimization of Subqueries and Aggregates" by C.Galindo-Legaria and M.Joshi, SIGMOD 2001]*

Join on Clustering and Integer Attributes

```
SELECT Employee.ssnum
FROM Employee E,
     Student S
WHERE E.name = S.name
```

⇨

```
SELECT Employee.ssnum
FROM Employee E,
     Student S
WHERE E.ssnum = S.ssnum
```

Table ***Employee*** *is clustered on ssnum where ssnum is an integer.*

Avoid Views (pseudotables) with Unnecessary Joins

```
CREATE VIEW Techlocation AS
   (SELECT ssnum, Techdept.dept,
           location
    FROM Employee E, Techdept TD
    WHERE E.dept = TD.dept)

SELECT dept FROM Techlocation TD
WHERE ssnum = 4444
```

```
SELECT dept
FROM Employee E
WHERE E.ssnum = 4444
```

Join with **Techdept** *is unnecessary.*

Maintain frequently used Aggregates

```
CREATE OR REPLACE TRIGGER updateVendorOutstanding
  [BEFORE|AFTER INSERT|UPDATE|DELETE AND/OR [I|U|P] ]
  ON orders [FOR EACH ROW] [DECLARE <vars>] [WHERE <cond>]
  BEGIN
      UPDATE vendorOutstanding
      SET :NEW.amount =
        SELECT :OLD.amount+SUM(INS.quantity*IT.price)
        FROM inserted INS, item IT
        WHERE INS.itemnum = IT.itemnum
      WHERE vendor = (SELECT vendor FROM inserted);
  END
```

Useful if the aggregate is needed frequently.

Avoid External Loops (I)

No loop:

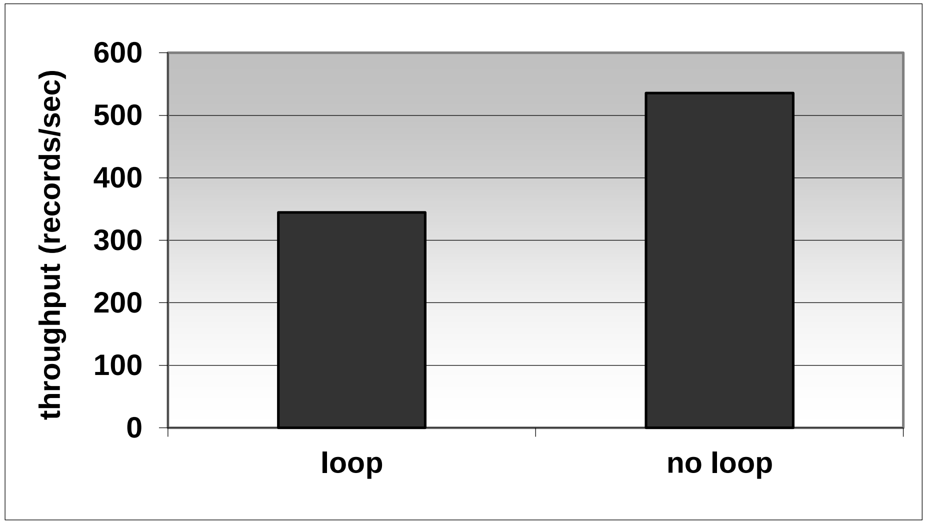
```
sqlStmt = “select * from lineitem where l_partkey <= 200;”
odbc->prepareStmt(sqlStmt);
odbc->execPrepared(sqlStmt);
```

Loop:

```
sqlStmt=“select * from lineitem where l_partkey = ?;”
odbc->prepareStmt(sqlStmt);
for (int i=1; i<200; i++) {
  odbc->bindParameter(1, SQL_INTEGER, i);
  odbc->execPrepared(sqlStmt);
}
```

Avoid External Loops (II)

Graphical impact of loops vs no-loops in MS SQL Server 2000 on Windows 2000



Let the DBMS optimize set operations since crossing the application interface has a significant impact on performance.

Avoid Cursors[*]

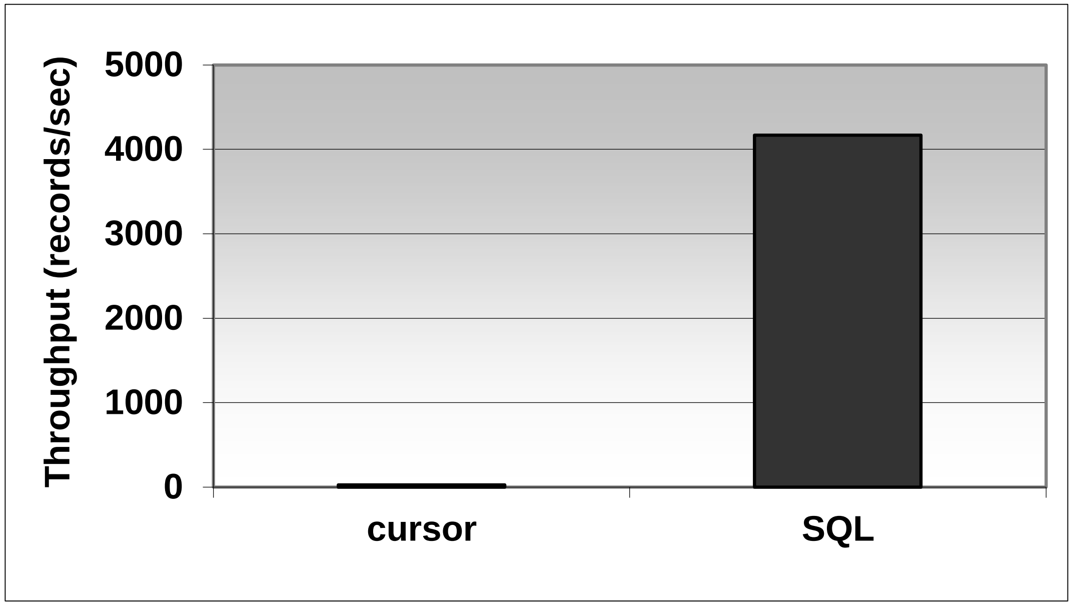
No cursor

```
SELECT * FROM employees;
```

Cursor

```
DECLARE
  v_emp employees%ROWTYPE;
  CURSOR crs IS
    SELECT * FROM
    employees;
BEGIN
  OPEN crs;
  LOOP
    FETCH crs INTO v_emp;
    EXIT WHEN crs%NOTFOUND;
  END LOOP;
  CLOSE crs;
END;
```

Graphical response time of cursor vs no-cursor in MS SQL Server 2000 on Windows 2000 (>1h vs some seconds)



[*]A cursor is a pointer to a private SQL area that stores information about the processing of a SELECT or data manipulation language (DML) statement (INSERT, UPDATE, DELETE, or MERGE). Cursor management of DML statements is handled by Oracle Database, but PL/SQL offers several ways to define and manipulate cursors to execute SELECT statements.

Retrieve Needed Columns Only

All

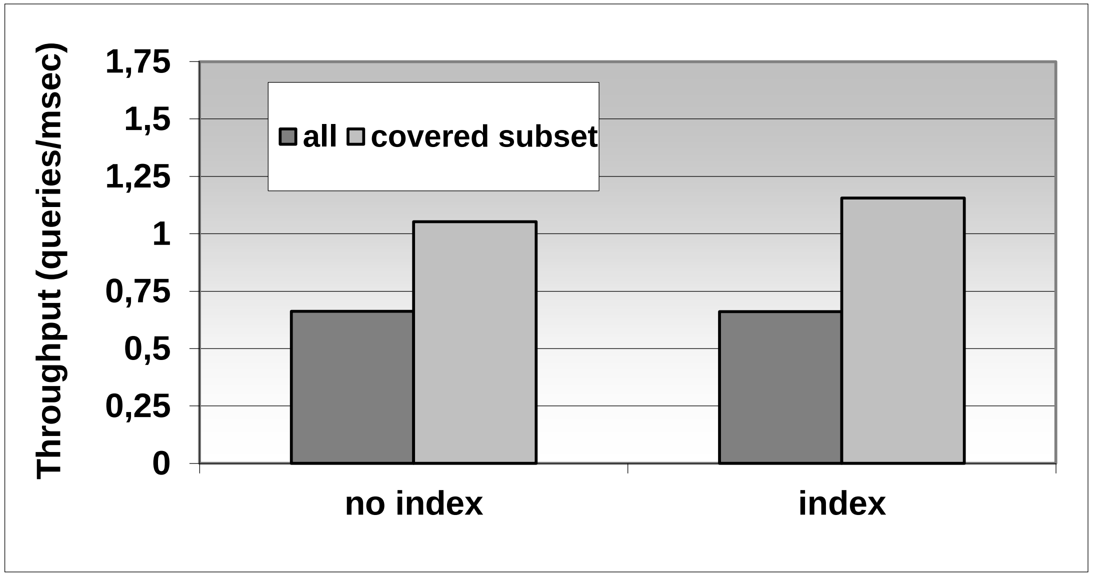
```
SELECT * FROM lineitem;
```

Covered subset

```
SELECT l_orderkey,
       l_partkey,
       l_suppkey,
       l_shipdate,
        l_commitdate
   FROM lineitem;
```

Avoid transferring unnecessary data

Use of a covering index may enhance performance.*



An index* ***covers* *the* ***query*** *if all the columns specified in the query are part of the index.*

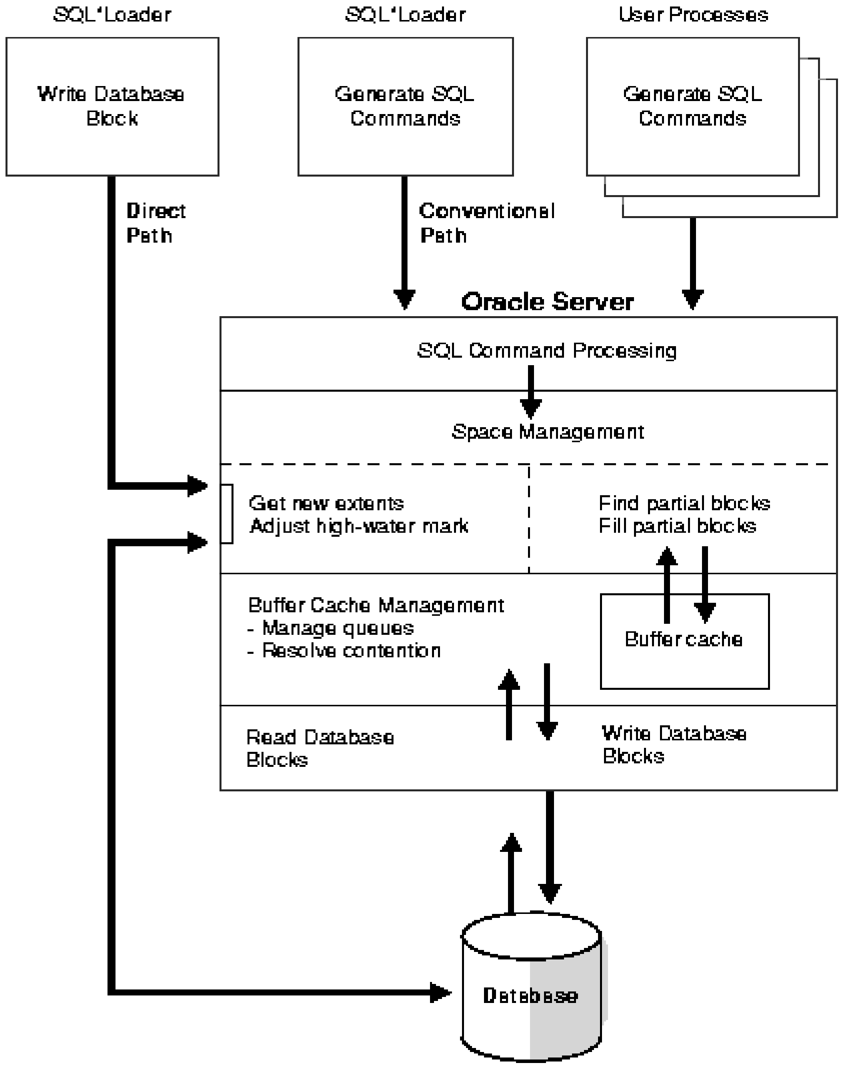
Use Direct Path for Bulk Loading (I)

```
sqlldr         ->  Oracle Bulk Loader
directpath=true ------------------->
control=load_lineitem.ctl ---------
--------------> see next slide ;)
log=           -> <log file>
bad=           -> <bad file>
```

If no log file and/or bad file are specified the **sqlldr** *will use the name of the control file with the* **.log** *and* **.bad** *extensions, respectively.*

.log -> *Relevant information about the bulk load operation, such as the number of tuples loaded, and a description of errors that may have occurred.*

.bad -> *Bad tuples (load fails) are recorded (if any).*



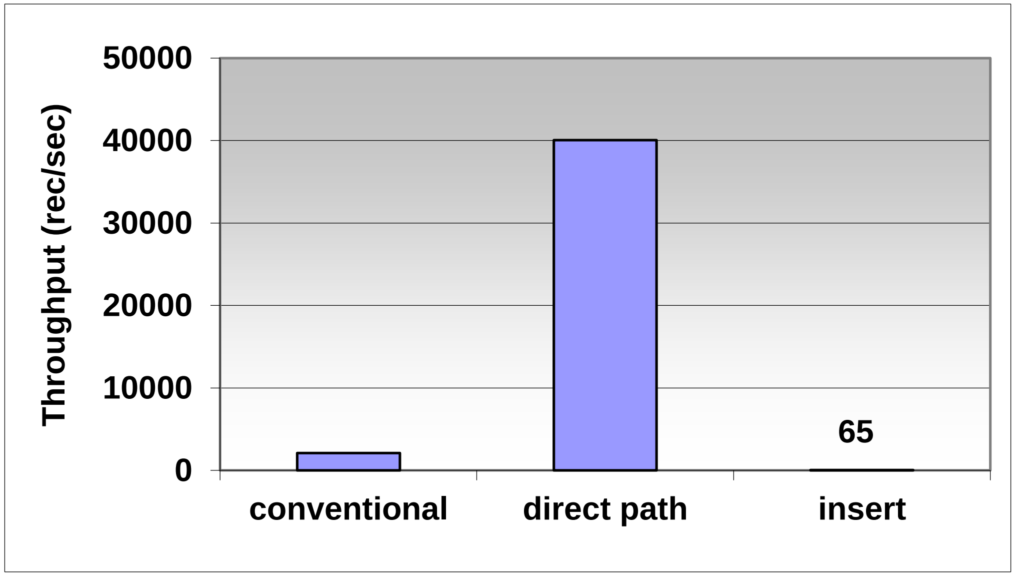
Use Direct Path for Bulk Loading (II)

```
LOAD DATA
INFILE "lineitem.tbl“     -> <datafile>
APPEND INTO TABLE lineitem-> <tableName>
FIEDLS TERMINATED BY '|'  -> <separator>
(
  L_ORDERKEY, L_PARTKEY, L_SUPPKEY, L_LINENUMBER, L_QUANTITY,
  L_EXTENDEDPRICE, L_DISCOUNT, L_TAX, L_RETURNFLAG,
  L_LINESTATUS, L_SHIPDATE DATE "YYYY-MM-DD", L_COMMITDATE DATE
  "YYYY-MM-DD", L_RECEIPTDATE DATE "YYYY-MM-DD", L_SHIPINSTRUCT,
  L_SHIPMODE, L_COMMENT
)                         -> (<list of attribute names to load>)
```

Use Direct Path for Bulk Loading (III)

Graphical illustration of

conventional path load vs direct path load vs common insert methods.



Direct path loading bypasses the query engine and the storage manager. As such, it is orders of magnitude faster than for conventional bulk load (commit every 100 records) and inserts (commit for each record).

Part II:
Rule-based & Cost-based Optimization Techniques

ORACLE Query Optimization Approaches (I)

Oracle supports two approaches for query optimization: Rule-based and Cost-based, which was introduced in Oracle 7i in order to improve query optimization.*

- **Rule-based:** *The optimizer ignores statistics.*
- **Cost-based:** *Three different goals.*
  - **All_Rows:** *The optimizer optimizes with a goal of best throughput (minimum resource use to complete the entire statement.*
  - **First_Rows_n:** *The optimizer optimizes with a goal of best response time to return the first n number of rows; n can equal 1, 10, 100 or 1000.*
  - **First_Rows:** *The optimizer uses a mix of cost and heuristics to find a best plan for fast delivery of the first few rows.*

*<u>*Deprecated as of Oracle 10g but still honored when used in Oracle 11g (if applicable)*</u>

ORACLE Query Optimization Approaches (II)

The **CHOOSE** *mode states that:*

- *The optimizer chooses between a cost-based approach and a rule-based approach, depending on whether statistics are available. This is the default value.*
- *If the data dictionary (i.e. DBA_, USER_ & ALL_ views) contains statistics for at least one of the accessed tables, then the optimizer uses a cost-based approach and optimizes with a goal of best throughput.*
- *If the data dictionary contains only some statistics, then the cost-based approach is still used, but the optimizer must guess the statistics for the subjects without any statistics. This can result in suboptimal execution plans.*

ORACLE Query Optimization Approaches (III)

To specify the optimizer's goal for an entire session, use the following statement:

```
alter session set optimizer_mode = <MODE_VALUE> where
MODE_VALUE = {rule, choose, all_rows,
              first_rows, first_rows_n}
```

For a specific statement the goal to be used by the optimizer can be stated using a **hint**.

For **RBO** *&* **CBO**, *a hint is nothing more than a comment with a specific format inside a SQL statement (/*+ **<HINT>(<param>** */). Hints come with their own set of problems. A hint looks just like a comment (more in the idiosyncrasies section)!*

```
SELECT /*+ INDEX(EMP_IDX) */ LASTNAME, FIRSTNAME, PHONE
FROM EMP
```

ORACLE Query Optimization Approaches (IV)

Hints can be categorized as follows:

- *Optimizer SQL hints for changing the query optimizer goal.*
- *Full table scan hints.*
- *Index unique scan hints.*
- *Index range scan descending hints.*
- *Fast full index scan hints.*
- *Join hints, including index joins, nested loop joins, hash joins, sort merge joins, Cartesian joins, and join order.*
- *Other optimizer hints, including access paths, query transformations, and parallel execution.*

Rule-Based Approach (I)

When ignoring statistics & heuristics, there should be a way to choose between possible access paths suggested by different execution plans.

Thus, 15 rules were ranked in order of efficiency. An **access path** *for a table is chosen if the statement contains a predicate or other construct that makes that access path available.*

Score assigned to each execution strategy (plan) using these rankings and strategy with best (lowest) score selected.

When two strategies produce the same score, the "tie-break" rule is used by making a decision based **on order in which tables occur in the SQL statement***.*

Challenge time: *Does this rule impact query optimization???*

Rule-Based Approach (II)

Table 20.4 Rule-based optimization rankings.

Rank	Access path
1	Single row by ROWID (row identifier)
2	Single row by cluster join
3	Single row by hash cluster key with unique or primary key
4	Single row by unique or primary key
5	Cluster join
6	Hash cluster key
7	Indexed cluster key
8	Composite key
9	Single-column indexes
10	Bounded range search on indexed columns
11	Unbounded range search on indexed columns
12	Sort–merge join
13	MAX or MIN of indexed column
14	ORDER BY on indexed columns
15	Full table scan

Understanding the RBO : An Example (I)

*Suppose there is a table "***PropertyForRent***" with* **indexed** *attributes: propertyNo, rooms and city. Consider the query:*

```
SELECT propertyNo
  FROM PropertyForRent
  WHERE rooms > 7 AND city = 'Sydney'
```

- *Single-column access path using index on city from WHERE condition (city = 'Sydney'):* **rank 9**
- *Unbounded range scan using index on rooms from WHERE condition (rooms > 7):* **rank 11**.
- *Full table scan:* **rank 15**
- *Although there is an index on propertyNo, the column does not appear in the* **WHERE** *clause and so is not considered by the optimizer.*

Based on these paths, rule-based optimizer will choose to use the index on the "city" column.

Understanding the RBO (II)

Disadvantages

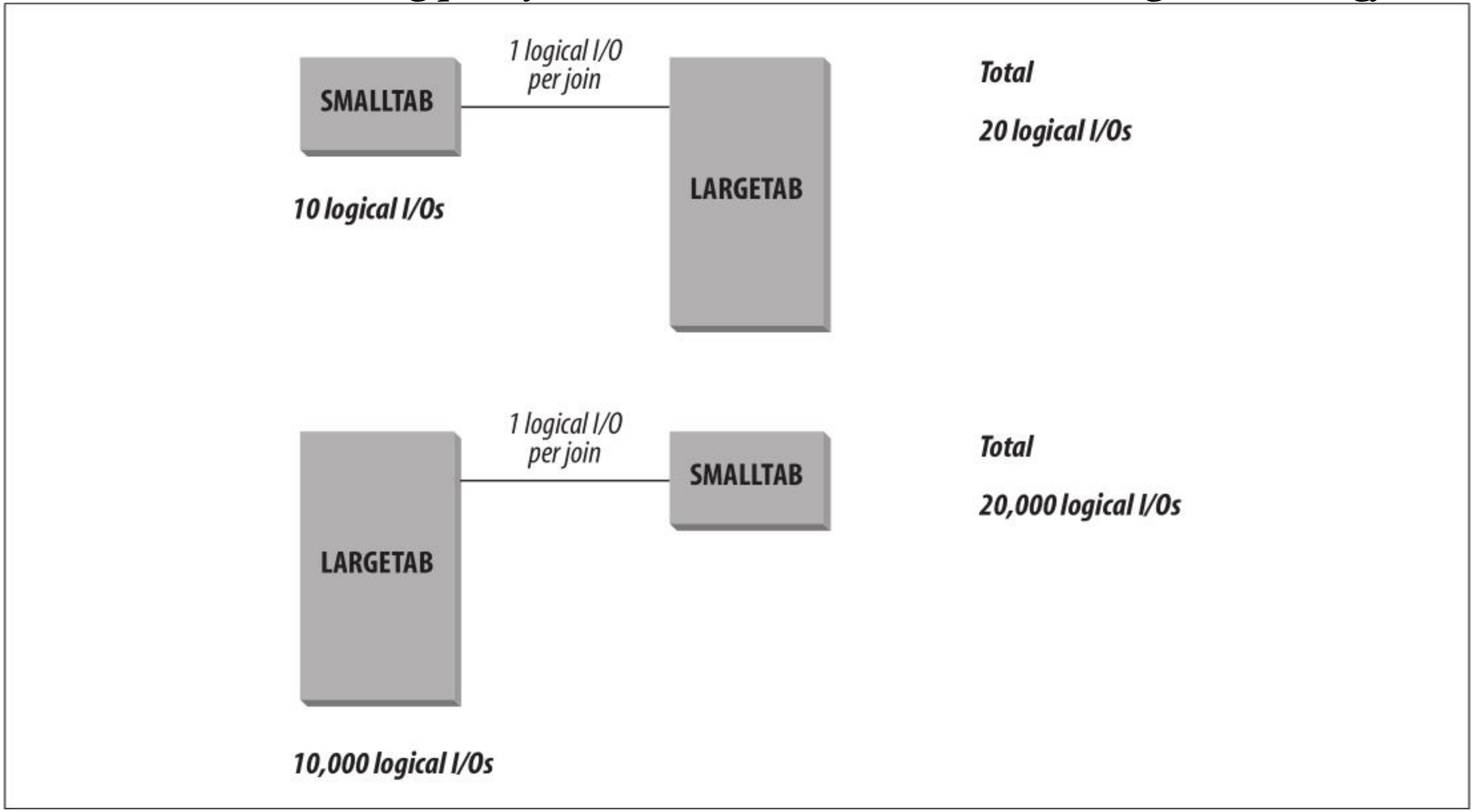
1. *Simplistic set of rules:*

"In a complex database, a query can easily involve several tables, each with several indexes and complex selection conditions and ordering. This complexity means that there were a lot of options, and the simple set of rules used by the rule-based optimizer might not differentiate the choices well enough to make the best choice".

Source: *Oracle Essentials (4th edition)*

Understanding the RBO (III)

2. Weak resolving policy in “tie-break” rule when choosing best strategy



Understanding the RBO (IV)

If you feel like crying, you may do so right now!

Still, the **RBO** *may be useful in cases such as in recursive operations (i.e. nested loops within nested loops in execution plans).*

That's why it has been de-supported and its use is discouraged but it has not been removed! ;)

Cost-Based Approach (I)

- **Cost-based optimizer** **depends on statistics** *for all tables, clusters, and indexes accessed by query. As such, it's the user's responsibility (yes you!) to generate statistics and keep them up-to-date.*

- **Two ways** *for generating and managing statistics:*

  - *By using package* **DBMS_STATS**, *for example (***strongly*** recommended):*
    - `EXECUTE DBMS_STATS.GATHER_SCHEMA_STATS('schema_name');`

  - *By issuing the* **ANALYZE** *statement, for example (avoid if possible):*
    - `ANALYZE TABLE <table> COMPUTE/ESTIMATE STATISTICS;`
    - `ANALYZE TABLE <table> COMPUTE/ESTIMATE STATISTICS FOR TABLE;`
    - `ANALYZE TABLE <table> COMPUTE/ESTIMATE STATISTICS FOR ALL INDEXES;`

Cost-Based Approach (II)

Do not use the COMPUTE and ESTIMATE clauses of ANALYZE to collect optimizer statistics. These clauses are supported for backward compatibility. Instead, use the **DBMS_STATS** *package, which lets you collect statistics in parallel, collect global statistics for partitioned objects, and fine tune your statistics collection in other ways. The cost-based optimizer, which depends upon statistics, will eventually use only statistics that have been collected by DBMS_STATS. See Oracle Database PL/SQL Packages and Types Reference for more information on the DBMS_STATS package.*

You must use the ANALYZE statement (rather than DBMS_STATS) for statistics collection not related to the cost-based optimizer, such as:

- *To use the VALIDATE or LIST CHAINED ROWS clauses (more on this later).*
- *To collect information on freelist blocks (structure where Oracle maintains a list of all free available blocks – critical for INSERT performance – more on this later).*

Cost-Based Approach (III)

Data Dictionary views

Data dictionary view	Type of information
ALL_TABLES	Information about the object and relational tables
TABLES	Information about the relational tables
TAB_COMMENTS	Comments about the table structures
TAB_HISTOGRAMS	Statistics about the use of tables
TAB_PARTITIONS	Information about the partitions in a partitioned table
TAB_PRIVS*	Different views detailing all the privileges on a table, the privileges granted by the user, and the privileges granted to the user
TAB_COLUMNS	Information about the columns in tables and views
COL_COMMENTS	Comments about individual columns
COL_PRIVS*	Different views detailing all the privileges on a column, the privileges granted by the user, and the privileges granted to the user
LOBS	Information about large object (LOB) datatype columns
VIEWS	Information about views
INDEXES	Information about the indexes on tables
IND_COLUMNS	Information about the columns in each index
IND_PARTITIONS	Information about each partition in a partitioned index
PART_*	Different views detailing the composition and usage patterns for partitioned tables and indexes
CONS_COLUMNS	Information about the columns in each constraint
CONSTRAINTS	Information about constraints on tables
SEQUENCES	Information about sequence objects
SYNONYMS	Information about synonyms
TAB_COL_STATISTICS	Statistics used by the cost-based analyzer
TRIGGERS	Information about the triggers on tables
TRIGGER_COLS	Information about the columns in triggers

Understanding the CBO (I)

1) **Functionality**
   - I. *Parse the statement.*
   - II. *Generate a list of all potential execution plans.*
   - III. *Calculate (estimate) the cost of each execution plan using internal rules and exploiting already known parameters (see below) about the relevant tables.*
   - IV. *Select the plan with the lowest cost.*

2) **Parameters**
   - I. *Primary Key – Unique Index*
   - II. *Non – Unique Index*
   - III. *Range evaluation (with bind variables)*
   - IV. *Histograms*
   - V. *System Resource Usage (CPU & I/O – Data Dictionary tables)*
   - VI. *Current Stats*

Understanding the CBO (II)

Data structure	Type of statistics
Table	Number of rows
	Number of blocks
	Number of unused blocks
	Average available free space per block
	Number of chained rows
	Average row length
Column	Number of distinct values per column
	Second-lowest column value
	Second-highest column value
	Column density factor
Index	Depth of index B*-tree structure
	Number of leaf blocks
	Number of distinct values
	Average number of leaf blocks per key
	Average number of data blocks per key
	Clustering factor

Understanding the CBO (III)

Unfortunately, since the CBO is using **heuristics***, it generates execution plans that attempt to execute the query as efficiently as possible but for may reasons it will often choose a sub-optimal plan! :/*

In addition, Oracle 10g comes with **Dynamic Sampling** *support which means that even with the same parameter settings, an SQL query will not necessarily give the same plan in two or more different parses (great :P)!!!*

Still, it's the future and when used properly & correctly it will give the best plans when considerable effort has been put in place by the user (i.e. collecting statistics etc).

Using ORACLE Optimization Modes

1. **When will the RBO be used?**
   1. *OPTIMIZER_MODE = RULE.*
   2. *=CHOOSE & statistics are not present for all tables in SQL statement.*
   3. *Alter session has been issued.*
   4. *RULE hint is present.*

2. **When will the CBO be used?**
   1. *OPTIMIZER_MODE = CHOOSE.*
   2. *=CHOOSE & statistics are not present for any tables in SQL statement.*
   3. *Alter session set optimizer_mode = (choose, first_rows or all_rows).*
   4. *CHOOSE, ALL_ROWS or FIRST_ROWS hint is present.*

Make sure to see: *https://blogs.oracle.com/optimizer/entry/why_was_the_rule_hint*

Part III:
Access Paths of Indexes

Access Path "Criterion"

DEFINITION:

$$\text{Selective Columns}^{*} = \frac{\text{\# of Unique Values for the Column}}{\text{\# Rows in the Table}}$$

SQL DEFINITION:

```
SELECT COUNT(DISTINCT ColumnName) FROM TableName) /
  SELECT COUNT(*) FROM TableName
```

*[*] **Column selectivity** is usually referred to, interchangeably, as **column cardinality**.*

B–Tree Indexes (I)*

1. **Excellent** *performance for highly selective columns, thus perfect for* **OLTP***s*.
   - **NOT** *effective for low selectivity columns.*
   - *Null values "break" performance.*
2. *Unique scan is most efficient when equality predicate on unique index is involved.*
3. *Range scan can be quite efficient but be careful of the size of the range specified.*
4. *Excellent for FIRST_ROWS access, particularly with queries returning a small number of results.*

Index access paths

- *INDEX UNIQUE SCAN/RANGE SCAN*
- *INDEX FULL SCAN/FAST FULL SCAN*
- *INDEX SKIP SCAN*
- *INDEX JOIN SCAN*

B–Tree Indexes (II)*

(Non-)Unique Index creation:

```
CREATE [UNIQUE] INDEX table_idx ON table(column_idx_key);
```

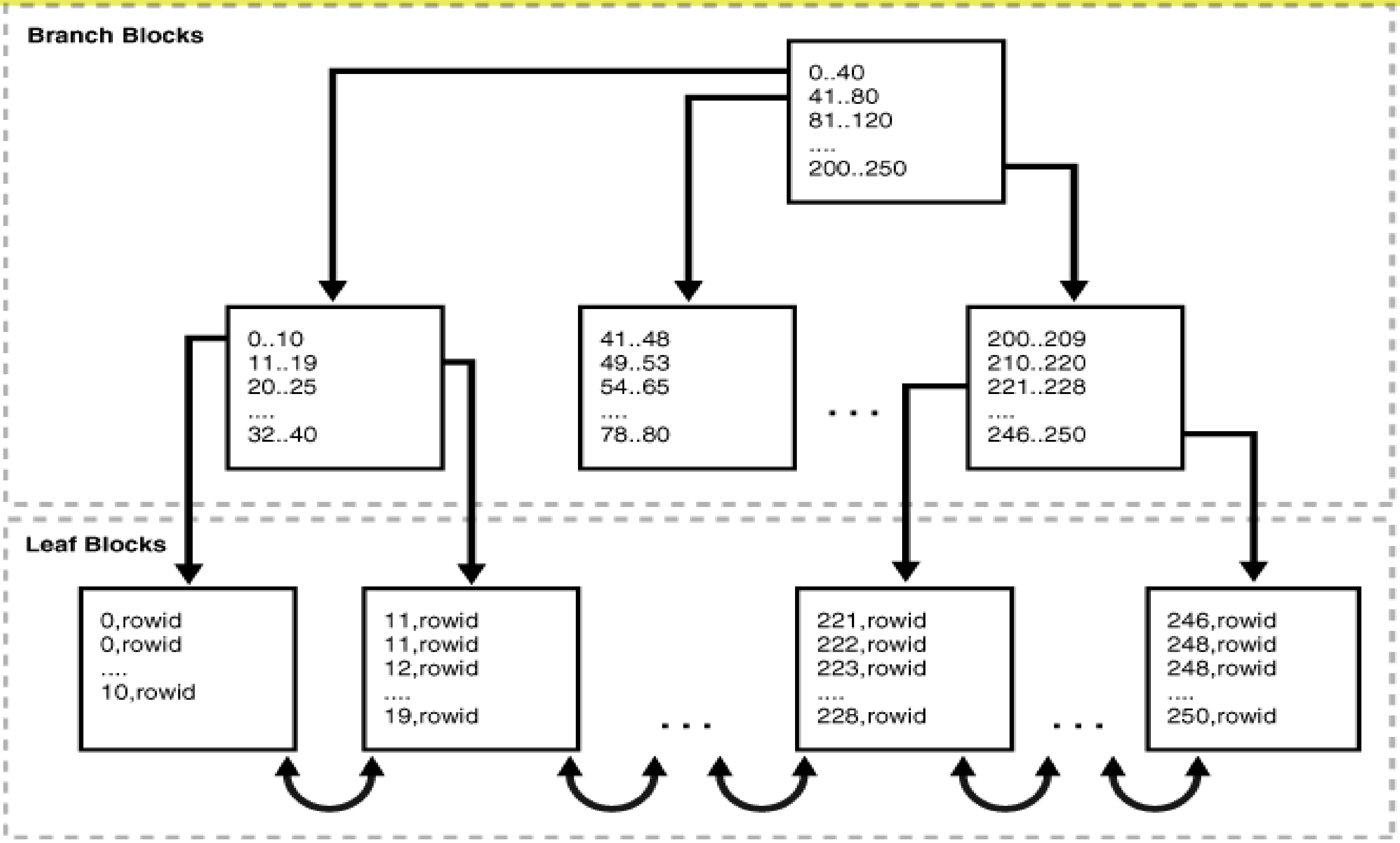
B–Tree indexes are* **pairs of (key, rowid)**.

It can also be specified using **INDEX(alias index_name)**, *if not created, or* **INDEX**. *However, the latter form allows the optimizer to freely choose any column as index of the table or use multiple columns as index if that access path provides the lowest cost.*

Providing multiple independent index hints is legal but it is recommended to use **INDEX_COMBINE** *for a combination of multiple indexes since it is more versatile hint.*

INDEX_DESC *hint uses a descending index.*

B–Tree (non – unique) Index Structure - Example*



B–Tree Index Access Paths (I)*

I. **INDEX UNIQUE SCAN**

- *Equality predicate on unique or primary key column(s).*
- *Generally considered most efficient access path.*
- *Usually, no more than 3 – 4 buffer gets.*
- *If table is "small",* **FULL TABLE SCAN** *could be cheaper.*

II. **INDEX RANGE SCAN**

- *Equality predicate on non-unique index, incompletely specified unique index, or range predicate on unique index.*
- *Be careful of the size of the range:*
  - *Large ranges could amount to huge # of buffer gets.*
  - *If so, consider a* **FAST FULL INDEX SCAN** *or* **FULL TABLE SCAN**.

B–Tree Index Access Paths (II)*

III. INDEX FULL SCAN

- *Will scan entire index by walking tree in index order.*
- *Provides ordered output, can be used to avoid sorts for ORDER BY clauses that specify index column order.*
- *Slower than INDEX FAST FULL SCAN if there is no ORDER BY requirement.*

IV. INDEX FAST FULL SCAN

- *Attempt to read index, in disk block order, and discard root and branch blocks.*
- *Attempt a DB file scattered read*[*]*, reading* ***db_file_multiblock_read_count*** *blocks at a time.*
- *Equivalent to FULL TABLE SCAN for an index.*
- *Fastest way to read entire contents of an index.*

[*] *In case of a "cold" buffer cache Oracle may choose to read ahead and thus opt to reading multiple physically disk blocks adjacent to the cache (aka 'db file scattered read').*

B–Tree Index Access Path (III)*

V. INDEX SKIP SCAN

- *Allows some benefits of multi-column index even without specifying the leading edge.*
- *Oracle will "skip scan" starting with root block skipping through B*-tree structure masking sections of tree that cannot have applicable data.*
- *Could be costly, depending on size of index, distribution of data and bind variable values.*

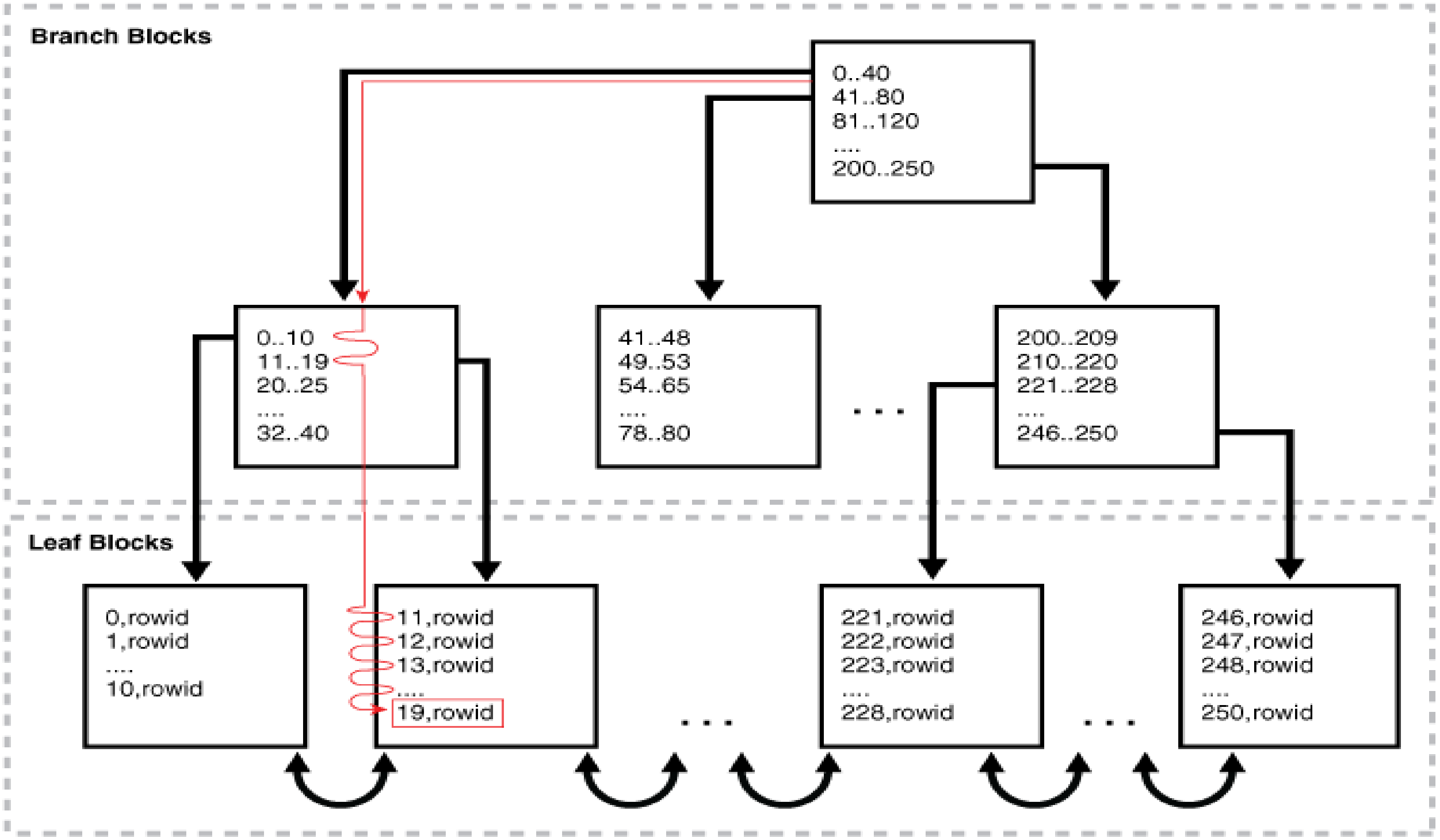
VI. INDEX JOIN SCAN

- *INDEX JOIN SCAN = HASH JOIN (more on this later) followed by INDEX RANGE + INDEX FAST FULL SCAN in the explain plan generation*

B*–Tree Index Access Path – Examples (I)

INDEX UNIQUE SCAN

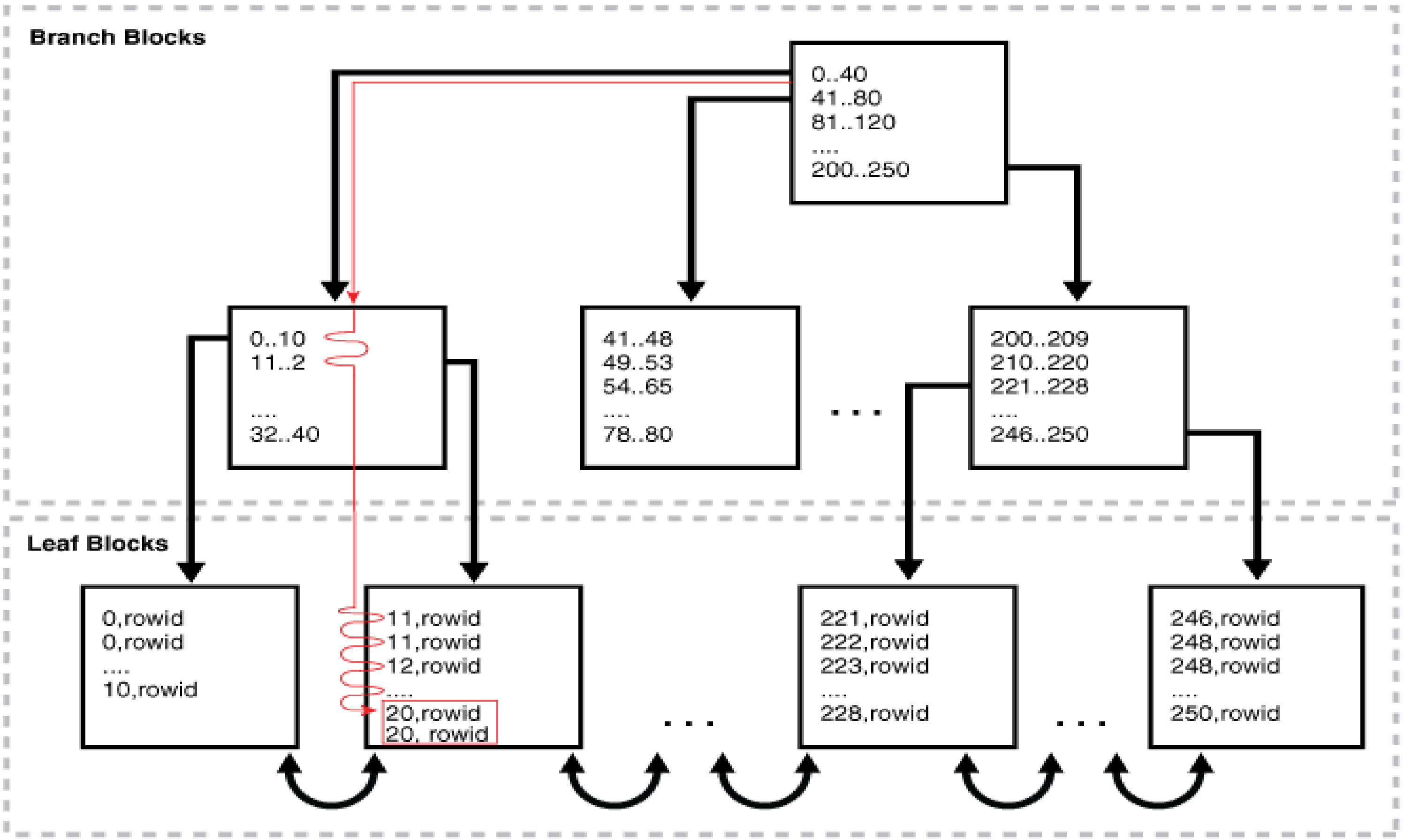
*SELECT * FROM products* **WHERE prod_id=19***;*



B–Tree Index Access Path – Examples (IIa)*

INDEX RANGE SCAN

*SELECT * FROM employees* **WHERE department_id = 20 AND salary > 1000***;*



B*–Tree Index Access Path – Examples (IIb)

INDEX RANGE SCAN DESCENDING

```
SELECT * FROM employees
WHERE department_id < 20
ORDER BY department_id DESC;
```

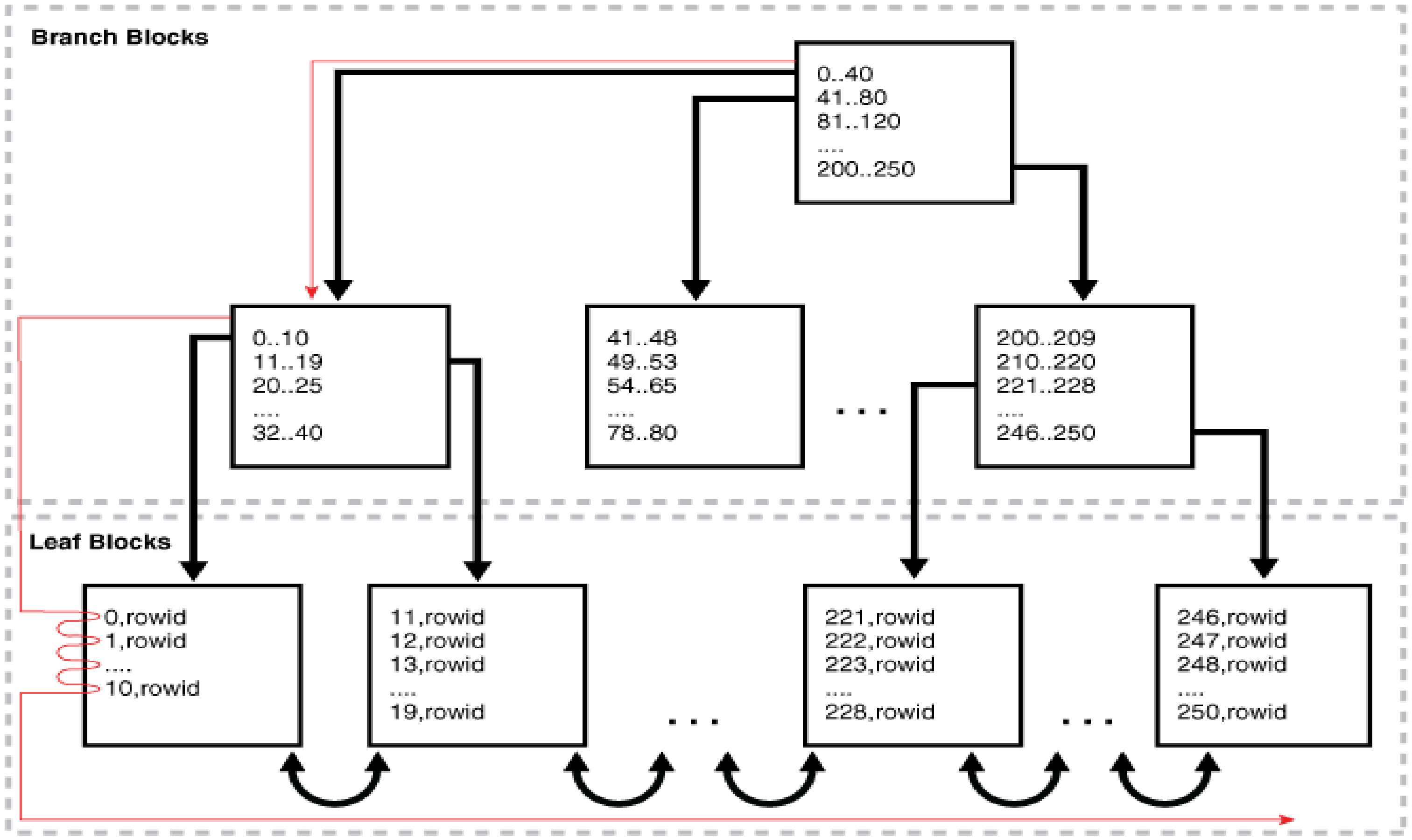
No need for illustration here.

You can figure it out yourself considering the previous example! ;)

B–Tree Index Access Path – Examples (III)*

INDEX FULL SCAN

SELECT dpt_id, dpt_name FROM departments **ORDER BY dpt_id***;*



B–Tree Index Access Path – Examples (IVa)*

INDEX FAST FULL SCAN

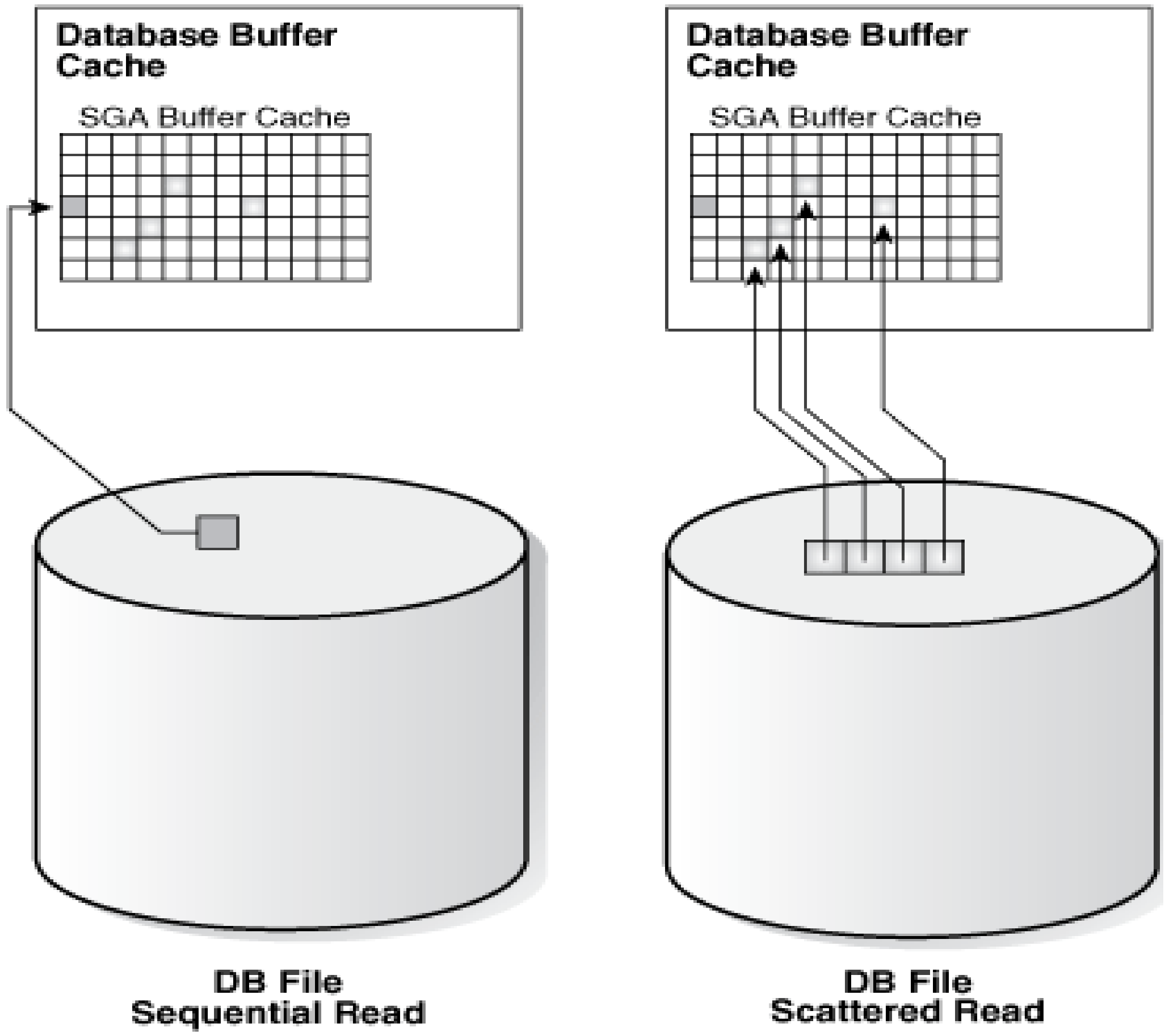
SELECT **/*+ INDEX_FFS(departments dept_id_pk) */** *COUNT(*)*
FROM departments;

The DB uses **multiblock I/O** *to read the root block, branch & leaf blocks by ignoring the former two (root block & branch blocks) and reading directly the latter ones (the leaf blocks). In essence,* **the index itself is used as the table***!*

Do note, that unlike an IFS, an IFFS cannot eliminate a sort operation because it does not read the index in order.

Also, the hint **IS** *mandatory for an IFFS to take place!*

B*–Tree Index Access Path – Examples (IVb)



B–Tree Index Access Path – Examples (Va)*

INDEX SKIP SCAN

I. *CREATE INDEX customers_idx ON customers* **(gender, email)**[*]*;*

II. *SELECT * FROM customers C*
WHERE C.email = 'Abbey@company.example.com'*;*

If below conditions are **both** *met then an ISS occurs:*

- *The leading column (i.e. gender) of the composite index is not part of the predicate condition.*
- *The leading column of the composite index has low selectivity but the following key of the composite index has high selectivity.*

[*]*Composite key index*

B*–Tree Index Access Path – Examples (Vb)

Transformed query

```
( SELECT * FROM customers C
  WHERE C.gender = 'F'
  AND C.email = 'Abbey@company.com' )
UNION ALL
( SELECT * FROM customers C
  WHERE C.gender = 'M'
  AND C.email = 'Abbey@company.com' )
```

B–Tree Index Access Path – Examples (VI)*

INDEX_JOIN SCAN

SELECT **/*+ INDEX_JOIN(employees) */** *last_name, email*

FROM employees E

WHERE E.last_name like 'A%';

An index join involves scanning multiple indexes, and then using a hash join on the ROWIDs obtained from these scans to return the rows. Table access is always avoided. For example, the process for joining two indexes on a single table is as follows:

1. *Scan the first index to retrieve ROWIDs.*
2. *Scan the second index to retrieve ROWIDs.*
3. *Perform a hash join by ROWID to obtain the rows.*

Bitmap Indexes (I)

1. *Most often implemented in a Data Warehouse environment (reporting and data analysis aka* **OLAP**) *by using BITMAP INDEXES.*
2. *Useful for columns which:*
   - *have relatively* **low cardinality** *where B*-Tree indexes will fail to provide any benefit.*
   - *Contain null values (does not affect performance).*
   - *are often specified along with other columns in WHERE clauses of SQL statements, optimizer will BITMAP AND the results of many single column bitmap indexes together.*
3. *Most efficient when doing COUNT(*) operations, where optimizer can utilize the BITMAP CONVERSION COUNT access path.*

Index Access Paths

- *BITMAP INDEX SINGLE VALUE*
- *BITMAP INDEX RANGE SCAN/FULL SCAN*
- *BITMAP AND/OR/NOT*
- *BITMAP CONVERSION COUNT/TO ROWIDs*
- *BITMAP MERGE*

Bitmap Indexes (II)

Bitmap Index creation:

```
CREATE BITMAP INDEX table_idx ON table(column_idx_key);
```

Bitmap Join Index creation:

```
CREATE BITMAP INDEX table_idx ON table1(column_idx_key)
  FROM table1 t1, table2 t2
  WHERE t1.id [=, [!=|<>], <, <=, >, >=] t2.id;
```

A bitmap index is a quadruple of **(key, low-rowid, high-rowid, series of 0 & 1)**

Bitmap Index (Logical) Structure – Example

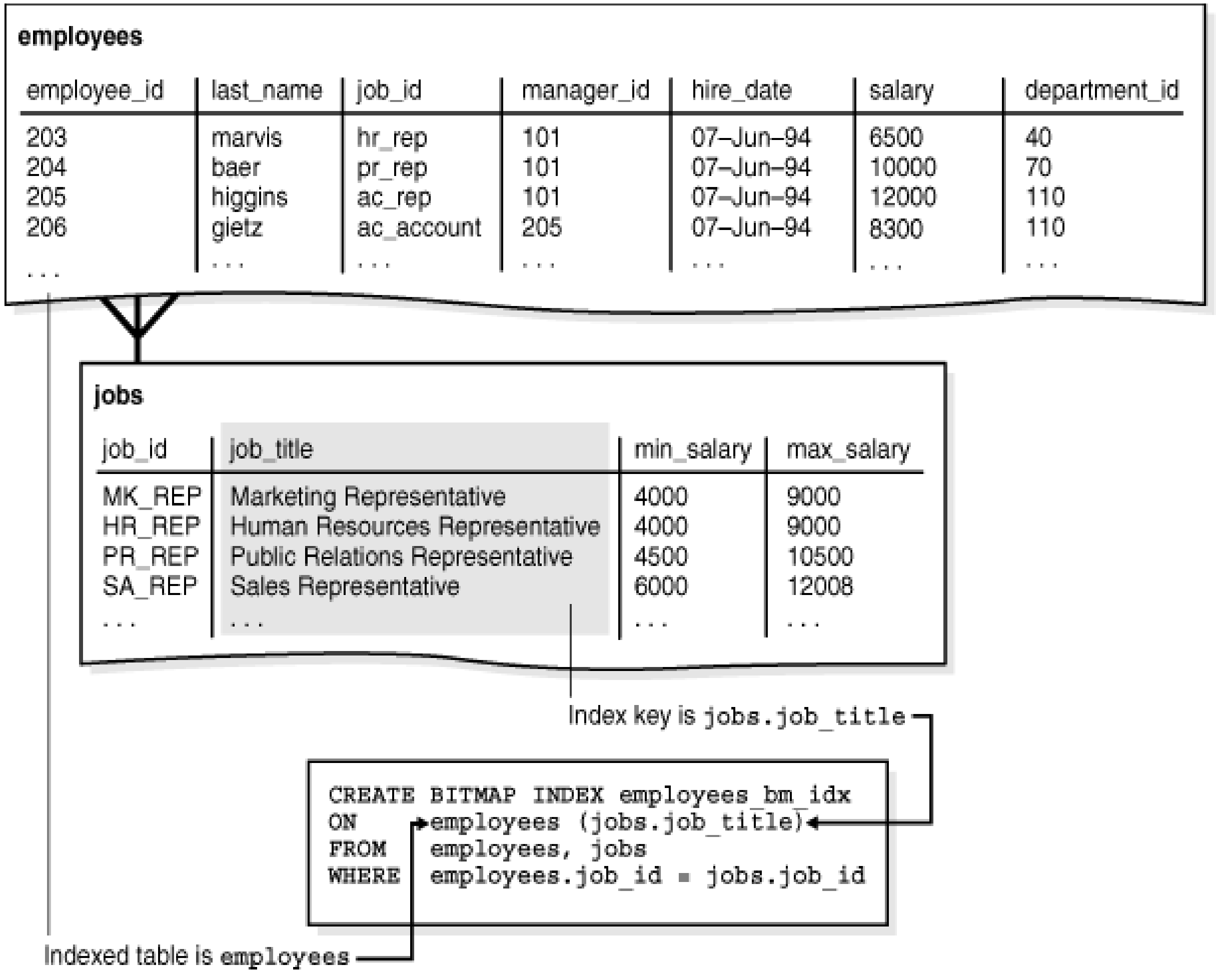
PARTS table

partno	color	size	weight
1	GREEN	MED	98.1
2	RED	MED	1241
3	RED	SMALL	100.1
4	BLUE	LARGE	54.9
5	RED	MED	124.1
6	GREEN	SMALL	60.1
...	...	...	...

Bitmap index on 'color'

color =	'BLUE'	0	0	0	1	0	0	...
color =	'RED'	0	1	1	0	1	0	...
color =	'GREEN'	1	0	0	0	0	1	...
Part number		1	2	3	4	5	6	

Bitmap Join Index – Example (I)



employees

employee_id	last_name	job_id	manager_id	hire_date	salary	department_id
203	marvis	hr_rep	101	07–Jun–94	6500	40
204	baer	pr_rep	101	07–Jun–94	10000	70
205	higgins	ac_rep	101	07–Jun–94	12000	110
206	gietz	ac_account	205	07–Jun–94	8300	110
. . .	. . .	. . .	. . .	. . .	. . .	. . .

jobs

job_id	job_title	min_salary	max_salary
MK_REP	Marketing Representative	4000	9000
HR_REP	Human Resources Representative	4000	9000
PR_REP	Public Relations Representative	4500	10500
SA_REP	Sales Representative	6000	12008
. . .	. . .	. . .	. . .

Index key is `jobs.job_title`

```
CREATE BITMAP INDEX employees_bm_idx
ON     employees (jobs.job_title)
FROM   employees, jobs
WHERE  employees.job_id = jobs.job_id
```

Indexed table is `employees`

In a data warehouse, the join condition is an equijoin between the primary key columns of the dimension tables and the foreign key columns in the fact table. Bitmap join indexes are sometimes much more efficient in storage than materialized join views, an alternative for materializing joins in advance.

Bitmap Join Index – Example (II)

*In reality, Oracle DB uses a B-tree index structure to store bitmaps for each indexed key. For example, if **jobs.job_title** is the key column of a bitmap index, then the index data is stored in one B-tree. The individual bitmaps are stored in the leaf blocks.*

A (conceptual) Bitmap Index leaf block:

```
Shipping Clerk,AAAPzRAAFAAAABSABQ,AAAPzRAAFAAAABSABZ,0010000100
Shipping Clerk,AAAPzRAAFAAAABSABa,AAAPzRAAFAAAABSABh,010010
Stock Clerk,AAAPzRAAFAAAABSAAa,AAAPzRAAFAAAABSAAc,1001001100
Stock Clerk,AAAPzRAAFAAAABSAAd,AAAPzRAAFAAAABSAAt,0101001001
Stock Clerk,AAAPzRAAFAAAABSAAu,AAAPzRAAFAAAABSABz,100001
.
.
.
```

Bitmap Index Access Paths

I. **BITMAP INDEX SINGLE VALUE (BISV)**
- *Used to satisfy equality predicate.*

II. **BITMAP INDEX RANGE SCAN (BIRS)**
- *Used to satisfy range operations such as BETWEEN.*
- *Unlike range scans on B*-Tree, BIRS is very efficient even for very large ranges.*

III. **BITMAP INDEX FULL SCAN (BIFS)**
- *Used to satisfy NOT predicate.*
- *Scan of entire index to identify rows NOT matching.*

IV. **BITMAP AND, OR, NOT**
- *Used for bitwise combinations of multiple bitmap indexes.*
- *A* **BITMAP MERGE** *takes place between conditions before bitwise operations.*

Bitmap Conversions

V. **BITMAP CONVERSION COUNT (BITCOC)**

- *Used to evaluate COUNT(*) operation for queries whose where clause predicates only specify columns having bitmap indexes.*
- *Very fast & very efficient.*

VI. **BITMAP CONVERSION TO ROWIDS (BITCOROW)**

- *Used in cases where row source produced by bitmap index operations needs to be joined to other row sources, i.e., join to another table, group by operation, to satisfy a TABLE ACCESS BY ROWID operation.*
- *More resource intensive than BITMAP CONVERSION COUNT.*
- *Can be quite expensive if number of ROWIDs is large.*

Bitmap Index Access Path – Examples

BITMAP INDEX SINGLE VALUE

*SELECT * FROM customers C* **WHERE C.maritalStatus = 'Widowed‘***;*

BITMAP INDEX RANGE SCAN (involves BITCOROW)

SELECT C.fname, C.lname FROM customers C **WHERE C.yearOfBirth < 1918***;*

BITMAP AND

(with BITMAP MERGE for intermediate BISV, BIRS)

SELECT C.fname, C.lname FROM customers **WHERE C.gender = 'F' AND C.yearOfBirth < 1918*;***

BITMAP CONVERSION COUNT (involves BISV)

SELECT COUNT (*) FROM sales S, customers C **WHERE C.cid = S.cid AND C.city = 'Smithville';**

Other Miscellaneous Access Paths (I)

I. **UNION, UNION [ALL], MINUS, INTERSECTION**

- *Directly correspond to the SQL set operators.*
- *UNION ALL is the cheapest of all since no SORT (UNIQUE) is required.*

II. **TABLE FULL SCAN**

- *Reads all blocks allocated to table.*
- *Can be most efficient access path for "small" tables.*
- *Can cause significant physical I/O, especially on larger tables.*
  - *Consider* **ALTER TABLE table_name CACHE** *(more on this later).*
  - *Consider putting table into KEEP buffer pool (more on this later).*

Other Miscellaneous Access Paths (II)

III. TABLE ACCESS BY INDEX ROWID [BATCHED]

- *Generally used in conjunction with an index access path, where ROWID has been identified, but Oracle needs access to a column not in the index.*
- *Consider whether adding a column to an existing index will provide substantial benefit.*
- *Cost is directly proportional to number of ROWID lookups that are required.*

IV. TABLE ACCESS HASH (TAH)

- *Ideal in situations where equality predicates prevail.*
- *More efficient than indexed access (queried more, modified less).*
- *Requires creation of hash cluster (same as index cluster but uses a hash) and they impose administrative/maintenance overhead (tables require frequent FTS and truncating).*

Other Miscellaneous Access Paths (III)

V. **TABLE ACCESS CLUSTER (TAC)**

- *Improves I/Os & access times on joins of clustered tables.*
- *Avoids data redundancy.*
- *Requires creation of indexed cluster and they impose administrative/maintenance overhead (same as in HAC).*

Index/Hash Cluster (I)

A **table cluster** *is a group of tables that share common columns and store related data in the same blocks. When tables are clustered, a single data block can contain rows from multiple tables. For example, a block can store rows from both the employees and departments tables rather than from only a single table.*

An **index cluster** *is a table cluster that uses an index to locate data. The cluster index is a B-tree index on the cluster key. A cluster scan retrieves all rows that have the same cluster key value from a table stored in an indexed cluster.*

A **hash cluster** *is like an indexed cluster, except the index key is replaced with a hash function. No separate cluster index exists. In a hash cluster, the data is the index. The database uses a hash scan to locate rows in a hash cluster based on a hash value.*

Index/Hash Cluster (II)

Index cluster creation:

```
CREATE CLUSTER emp_depts_cluster
  (dept_id NUMBER(4)) SIZE 512;
```

```
CREATE INDEX emp_depts_cluster_idx
  ON CLUSTER emp_depts_cluster;
```

Hash cluster creation:

```
CREATE CLUSTER emp_depts_cluster
  (dept_id NUMBER(4)) SIZE 8192 HASHKEYS 100;
```

Index/Hash Cluster Access Paths – Example (I)

Consider these tables …

```
CREATE TABLE emp2 CLUSTER emp_depts_cluster (dept_id)
  AS SELECT * FROM employees;
```

```
CREATE TABLE depts2 CLUSTER emp_depts_cluster (dept_id)
  AS SELECT * FROM departments;
```

… and this query.

```
SELECT * FROM emp2 WHERE dept_id = 30;
```

Index/Hash Cluster Access Paths – Example (II)

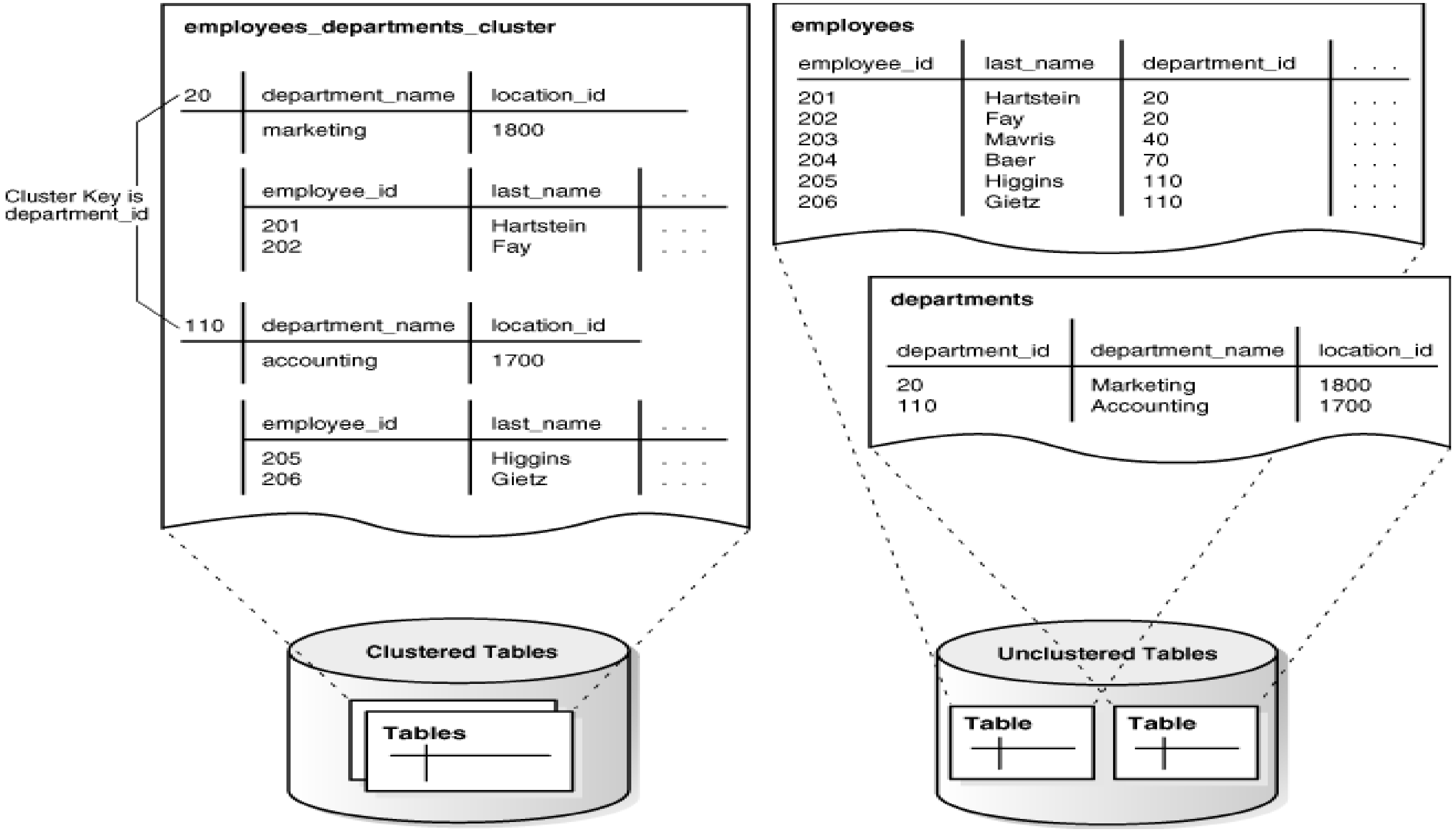
Table Access Cluster (TAC) scenario:

*To perform the scan, Oracle DB first obtains the ROWID of the row describing department 30 by scanning the cluster index (***IUS***). Then, Oracle DB locates the rows in employees using this ROWID (***TAC***)*

Table Access Hash (TAH) scenario:

*To perform a hash scan, Oracle DB first obtains the hash value by applying a hash function to the key value 30 and then uses this hash value to scan the data blocks and retrieve the rows (***TAH***).*

Structure of Clustered vs Unclustered Tables – Example


employees_departments_cluster
Cluster Key is department_id
20 | department_name | location_id
marketing | 1800
employee_id | last_name | . . .
201 | Hartstein
202 | Fay
110 | department_name | location_id
accounting | 1700
employee_id | last_name | . . .
205 | Higgins
206 | Gietz
Clustered Tables
Tables
employees
employee_id | last_name | department_id | . . .
201 Hartstein 20
202 Fay 20
203 Mavris 40
204 Baer 70
205 Higgins 110
206 Gietz 110
departments
department_id | department_name | location_id
20 Marketing 1800
110 Accounting 1700
Unclustered Tables
Table
Table

Access Paths of Join Methods

1. **Nested Loops**
   - *Generally geared towards FIRST_ROWS access.*
   - *Ideal for B*-Tree index driven access, small row sources.*
   - *When this is the case, always best for first row response time.*
   - *Can get very costly very quickly if no index path exists or index path is inefficient.*
2. **Sort Merge**
   - *Generally geared towards ALL_ROWS access.*
   - *Can be useful for joining small to medium size row sources, particularly if viable index path is not available or if Cartesian join is desired.*
   - *Be wary of* **sort_area_size**, *if it's too small, sorts will write to disk, performance will plummet.*
3. **Hash Join**
   - *Most controversial (and misunderstood) join method.*
   - *Can be very powerful, when applied correctly.*
   - *Useful for joining small to medium sized to a large row source.*
   - *Can be sensitive to instance parameters such as* **hash_area_size**, **hash_multiblock_io_count**, **db_block_size**.

Part IV:
Tuning Tools

Tuning Tools

- A significant portion of SQL that performs poorly in production was originally crafted against empty or nearly empty tables
- Make sure you establish a reasonable sub-set of production data that is used during development and tuning of SQL
- In order to monitor execution plans and tune queries, Oracle 9i (and higher) provides the following three tools:
  - `Explain Plan` command
  - `TkProf` trace file formatter
  - The `SQLTrace` (or `AutoTrace)` facility
- These tools, mainly, allow the user to the verify which access paths are considered by an execution plan
  - Some of them provide, also, information about the number of buffers used, physical reads from buffers, rows returned from each step, etc

Explain Plan

- The EXPLAIN PLAN reveals the execution plan for an SQL statement
  - The execution plan reveals the exact sequence of steps that the Oracle optimizer has chosen to employ to process the SQL
- The execution plan is stored in an Oracle table called the PLAN_TABLE
  - Suitably formatted queries can be used to extract the execution plan from the PLAN_TABLE
  - Create PLAN_TABLE command:
    ```
    @$ORACLE_HOME/rdbms/admin/utlxplan.sql
    ```
  - Issue explain plan command:
    ```
        Explain plan set statement_id = ‘MJB’ for
                    select * from dual;
    ```
  - Issue query to retrieve execution plan:
    ```
    @$ORACLE_HOME/rdbms/admin/utlxpls.sql
    ```
- The more heavily indented an access path is, the earlier it is executed
  - If two steps are indented at the same level, the uppermost statement is executed first
  - Some access paths are “joined” – such as an index access that is followed by a table lookup

Plan_Table

```
create table PLAN_TABLE
(
   statement_id varchar2(30),
   timestamp    date,
   remarks      varchar2(80),
   operation    varchar2(30),
   options      varchar2(30),
   object_node  varchar2(128),
   object_owner varchar2(30),
   object_name  varchar2(30),
   object_instance     numeric,
   object_type  varchar2(30),
   optimizer    varchar2(255),
   search_columns      number,
   id                  numeric,
   parent_id           numeric,
   position            numeric,
   cost                numeric,
   cardinality         numeric,
   bytes               numeric,
   other_tag           varchar2(255),
   partition_start     varchar2(255),
   partition_stop      varchar2(255),
   partition_id        numeric,
   other               long,
   distribution        varchar2(30)
);
```

Explain Plan

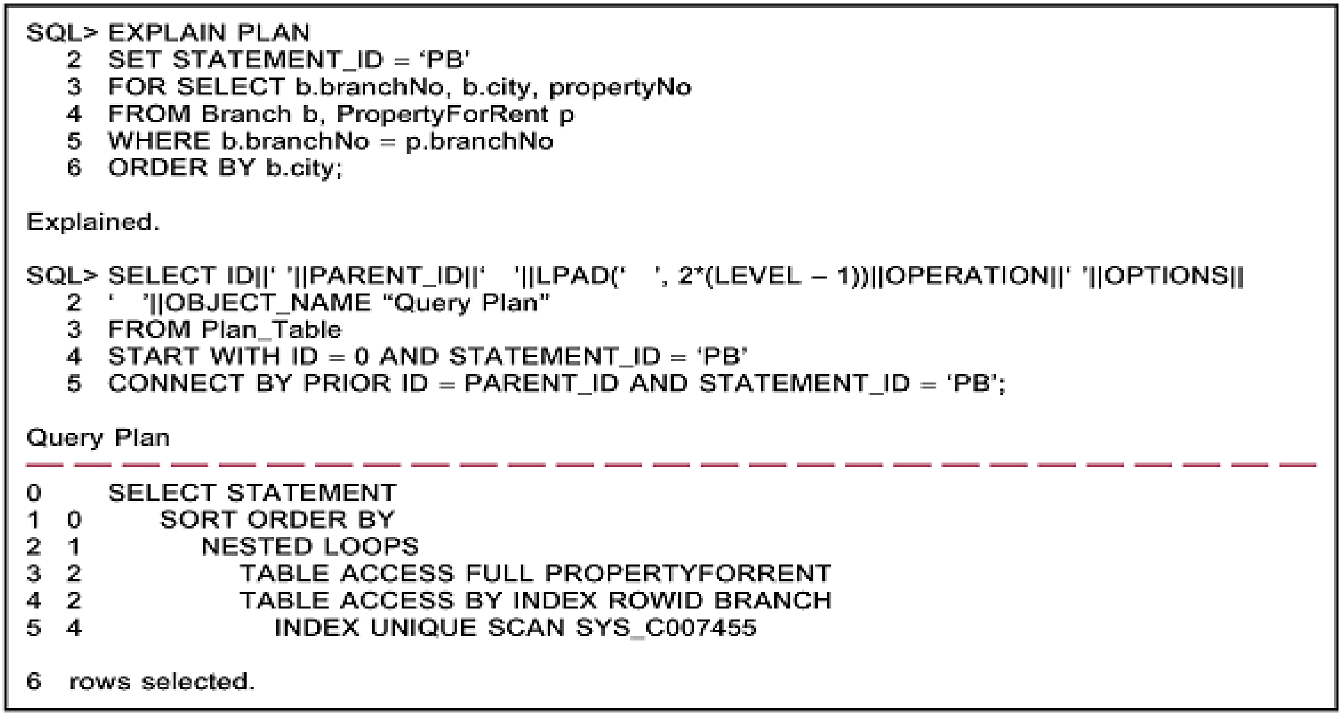
- Sample Query: Explain plan set statement_id = ‘MJB’ for

```
select doc_title
from documents doc, page_collections pc
where pc.pc_issue_date = ‘01-JAN-2002’
      and pc.pc_id = doc.pc_id;
```

- Sample Explain plan output

```
-------------------------------------------------------------------------------------------
| Operation                  |  Object_Name     |  Rows | Bytes|  Cardinality  | Pstart| Pstop |
-------------------------------------------------------------------------------------------
| SELECT STATEMENT           |                  |   61K|    3M|           328 |       |       |
|  NESTED LOOPS              |                  |   61K|    3M|           328 |       |       |
|   TABLE ACCESS BY INDEX RO|PAGE_COLL          |  834 |    9K|            78 |       |       |
|    INDEX RANGE SCAN        |PC_PC2_UK         |  834 |      |             6 |       |       |
|   INDEX RANGE SCAN         |DOC_DOC2_         |   86M|    4G|             3 |       |       |
-------------------------------------------------------------------------------------------
```

Viewing the Execution Plan of a Query in Oracle



```
SQL> EXPLAIN PLAN
  2  SET STATEMENT_ID = 'PB'
  3  FOR SELECT b.branchNo, b.city, propertyNo
  4  FROM Branch b, PropertyForRent p
  5  WHERE b.branchNo = p.branchNo
  6  ORDER BY b.city;

Explained.

SQL> SELECT ID||' '||PARENT_ID||'   '||LPAD('   ', 2*(LEVEL – 1))||OPERATION||' '||OPTIONS||
  2  '   '||OBJECT_NAME "Query Plan"
  3  FROM Plan_Table
  4  START WITH ID = 0 AND STATEMENT_ID = 'PB'
  5  CONNECT BY PRIOR ID = PARENT_ID AND STATEMENT_ID = 'PB';

Query Plan
— — — — — — — — — — — — — — — — — — — — — — — — — — —
0      SELECT STATEMENT
1  0      SORT ORDER BY
2  1        NESTED LOOPS
3  2          TABLE ACCESS FULL PROPERTYFORRENT
4  2          TABLE ACCESS BY INDEX ROWID BRANCH
5  4            INDEX UNIQUE SCAN SYS_C007455

6  rows selected.
```

Plan_Table – an SQL Table

Statement_id – plan identifier

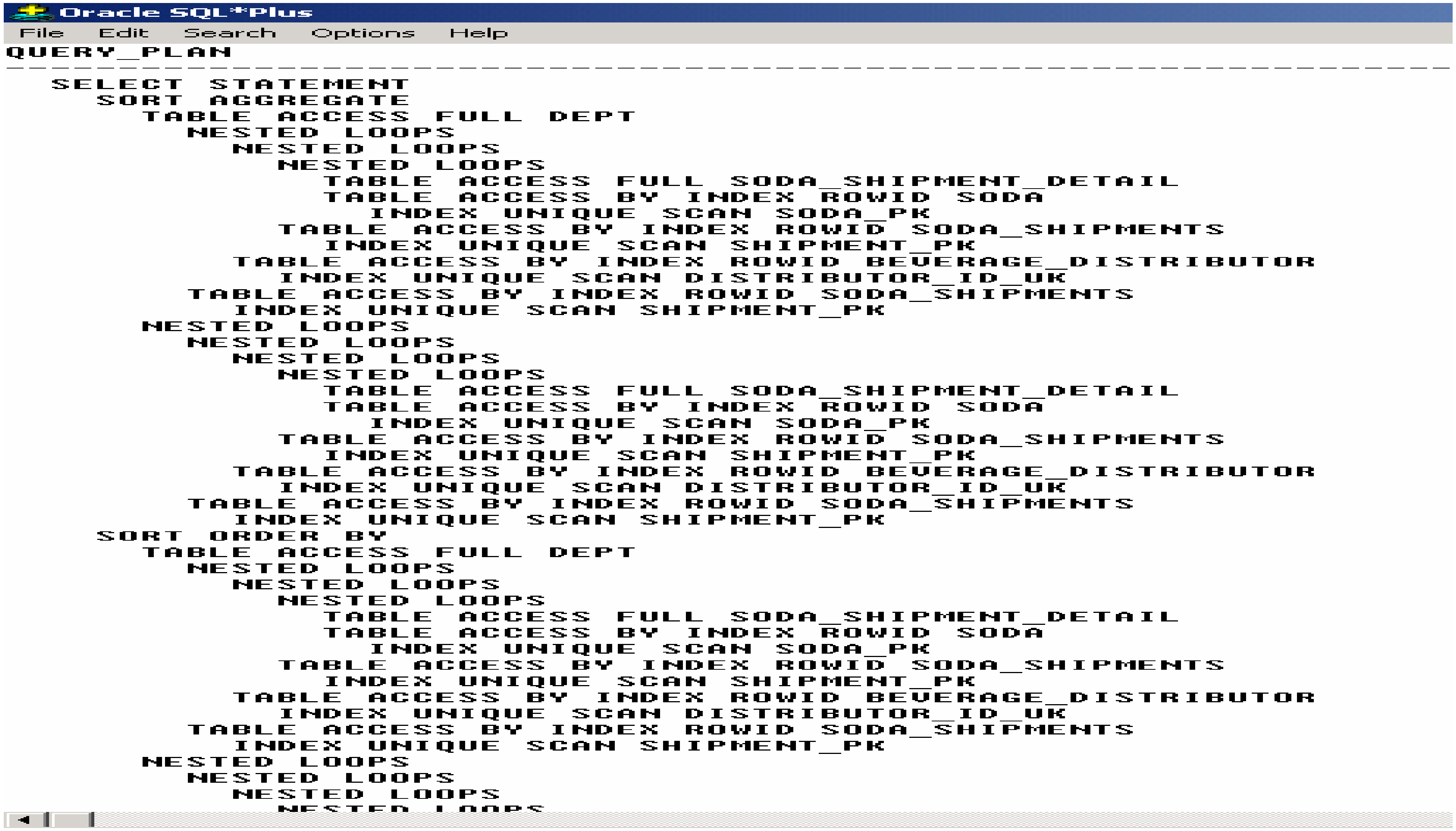
Id – a number assigned to each step

Parent_id – id of next step which operates on output of this step

Operation – eg internal operation select, insert etc

Options – name of internal operation

A more complex EXPLAIN PLAN



```
QUERY_PLAN
--------------------------------------------------------------------------------
   SELECT STATEMENT
      SORT AGGREGATE
         TABLE ACCESS FULL DEPT
            NESTED LOOPS
               NESTED LOOPS
                  NESTED LOOPS
                     TABLE ACCESS FULL SODA_SHIPMENT_DETAIL
                     TABLE ACCESS BY INDEX ROWID SODA
                        INDEX UNIQUE SCAN SODA_PK
                  TABLE ACCESS BY INDEX ROWID SODA_SHIPMENTS
                     INDEX UNIQUE SCAN SHIPMENT_PK
               TABLE ACCESS BY INDEX ROWID BEVERAGE_DISTRIBUTOR
                  INDEX UNIQUE SCAN DISTRIBUTOR_ID_UK
            TABLE ACCESS BY INDEX ROWID SODA_SHIPMENTS
               INDEX UNIQUE SCAN SHIPMENT_PK
         NESTED LOOPS
            NESTED LOOPS
               NESTED LOOPS
                  NESTED LOOPS
                     TABLE ACCESS FULL SODA_SHIPMENT_DETAIL
                     TABLE ACCESS BY INDEX ROWID SODA
                        INDEX UNIQUE SCAN SODA_PK
                  TABLE ACCESS BY INDEX ROWID SODA_SHIPMENTS
                     INDEX UNIQUE SCAN SHIPMENT_PK
               TABLE ACCESS BY INDEX ROWID BEVERAGE_DISTRIBUTOR
                  INDEX UNIQUE SCAN DISTRIBUTOR_ID_UK
            TABLE ACCESS BY INDEX ROWID SODA_SHIPMENTS
               INDEX UNIQUE SCAN SHIPMENT_PK
      SORT ORDER BY
         TABLE ACCESS FULL DEPT
            NESTED LOOPS
               NESTED LOOPS
                  NESTED LOOPS
                     TABLE ACCESS FULL SODA_SHIPMENT_DETAIL
                     TABLE ACCESS BY INDEX ROWID SODA
                        INDEX UNIQUE SCAN SODA_PK
                  TABLE ACCESS BY INDEX ROWID SODA_SHIPMENTS
                     INDEX UNIQUE SCAN SHIPMENT_PK
               TABLE ACCESS BY INDEX ROWID BEVERAGE_DISTRIBUTOR
                  INDEX UNIQUE SCAN DISTRIBUTOR_ID_UK
            TABLE ACCESS BY INDEX ROWID SODA_SHIPMENTS
               INDEX UNIQUE SCAN SHIPMENT_PK
         NESTED LOOPS
            NESTED LOOPS
               NESTED LOOPS
                  NESTED LOOPS
```

TkProf

- More details provided than `Autotrace` or `Explain Plan`
- For more useful information:

```
alter session set timed_statistics = true;
```

- To enable tracing:

```
alter session set sql_trace = true;
```

- Trace file written to *user_dump_dest*
- Use:

```
tkprof <trace_file> <output_file>
```

TkProf Sample Output

```
********************************************************************************
count    = number of times OCI procedure was executed
cpu      = cpu time in seconds executing
elapsed  = elapsed time in seconds executing
disk     = number of physical reads of buffers from disk
query    = number of buffers gotten for consistent read
current  = number of buffers gotten in current mode (usually for update)
rows     = number of rows processed by the fetch or execute call
********************************************************************************
<some text deleted>
select doc_title
  from documents doc,
       page_collections pc
 where pc.pc_issue_date = '01-JAN-2002'
   and pc.pc_id = doc.pc_id

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch     1415      0.07       0.09          0       1853          0       21206
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total     1417      0.07       0.10          0       1853          0       21206
```

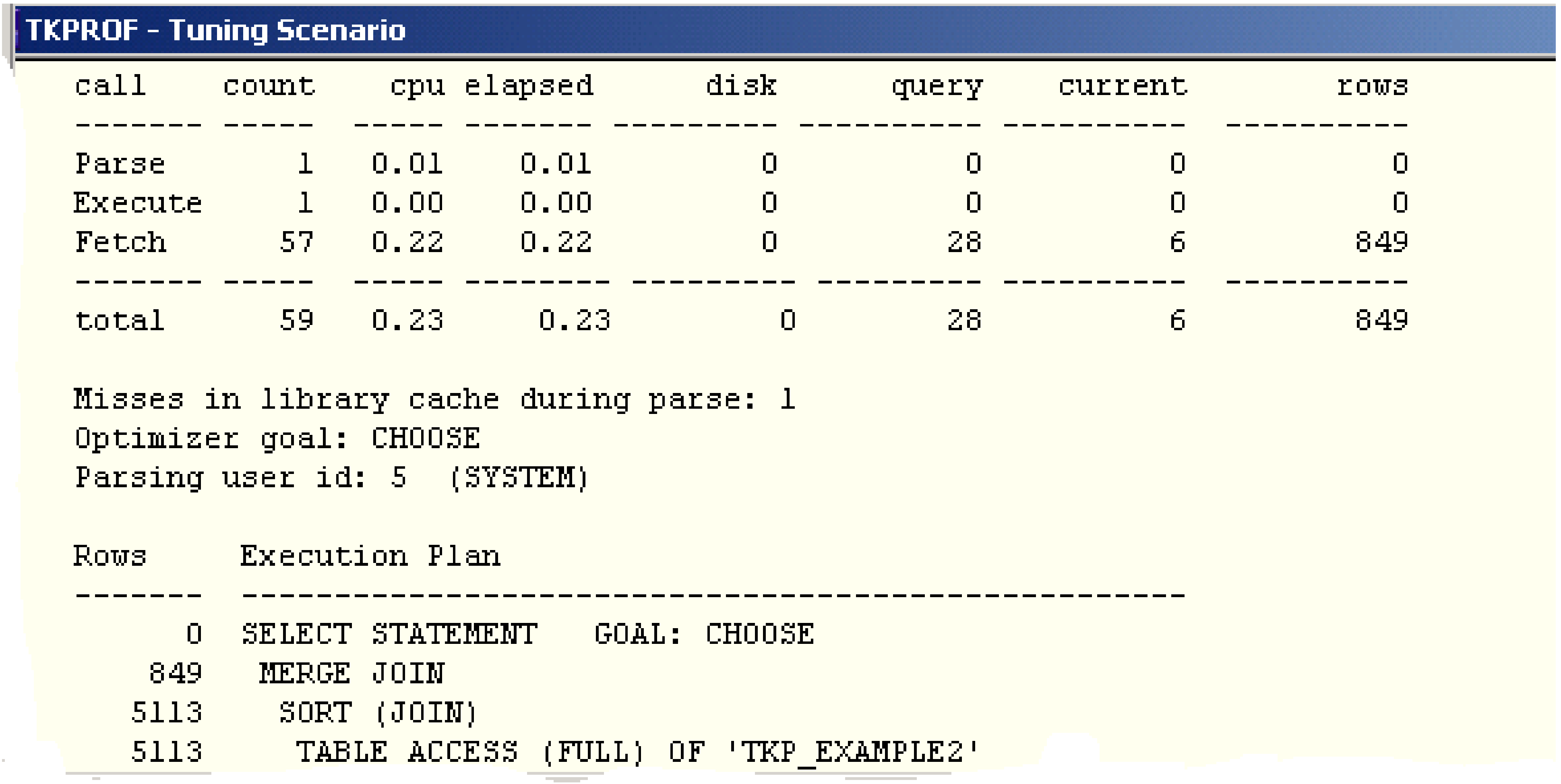
TkProf Sample Output

```
Rows     Row Source Operation
-------  ---------------------------------------------------
  21206  NESTED LOOPS
     31   TABLE ACCESS BY INDEX ROWID PAGE_COLLECTIONS
     31    INDEX RANGE SCAN (object id 22993)
  21206   INDEX RANGE SCAN (object id 22873)


Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
  21206   NESTED LOOPS
     31    TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID) OF 'PAGE_COLLECTIONS'
     31     INDEX   GOAL: ANALYZED (RANGE SCAN) OF 'PC_PC2_UK' (UNIQUE)
  21206    INDEX   GOAL: ANALYZED (RANGE SCAN) OF 'DOC_DOC2_UK' (UNIQUE)
```

SQL_TRACE and tkprof

- `ALTER SESSION SET SQL_TRACE TRUE` causes a trace of SQL execution to be generated
- The `TKPROF` utility formats the resulting output
- `Tkprof` output contains breakdown of execution statistics execution plan and rows returned for each step
  - These stats are not available from any other source
- `Tkprof` is the most powerful tool, but requires a significant learning curve

Tkprof output



TKPROF - Tuning Scenario

```
call     count    cpu  elapsed       disk      query    current        rows
------- -----  ----- ------- --------- ---------- ----------  ----------
Parse        1   0.01     0.01         0          0          0           0
Execute      1   0.00     0.00         0          0          0           0
Fetch       57   0.22     0.22         0         28          6         849
------- -----  ----- -------- --------- --------- ----------  ----------
total       59   0.23      0.23         0         28          6         849

Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 5  (SYSTEM)

Rows     Execution Plan
-------  ----------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
    849   MERGE JOIN
   5113    SORT (JOIN)
   5113     TABLE ACCESS (FULL) OF 'TKP_EXAMPLE2'
```

Part V:
Practical Optimization/Tuning

Query Tuning – What to do?

- Problematic SQL statements usually have:
  - ◆Excessive number of buffer gets
  - ◆Excessive number of physical reads
- So, if we consume less resources, we save time
  - ◆Reduce buffer gets (more efficient access paths)
    - Avoid (most) full table scans
    - Check selectivity of index access paths
    - Stay away from Nested Loop joins on large row sources
  - ◆Avoid physical I/O
    - Avoid (most) full table scans
    - Try to avoid sorts that write to disk, such as order by, group by, merge joins (set adequate sort_area_size)
    - Try to avoid hash joins writing to disk (hash_area_size)

Optimize Joins

- Pick the best join method
  - Nested loops joins are best for indexed joins of subsets
  - Hash joins are usually the best choice for “big” joins
  - Hash Join can only be used with equality
  - Merge joins work on inequality
  - If all index columns are in the where clause a merge join will be faster
- Pick the best join order
  - Pick the best “driving” table
  - Eliminate rows as early as possible in the join order
- Optimize “special” joins when appropriate
  - STAR joins for data-warehousing applications
  - STAR_TRANSFORMATION if you have bitmap indexes
  - ANTI-JOIN methods for NOT IN sub-queries
  - SEMI-JOIN methods for EXISTS sub-queries

Choosing a Driving Table

- The driving table is the table that is first used by Oracle in processing the query
    - Choosing the correct driving table is critical
- Driving table should be the table that returns the smallest number of rows and do the smallest number of buffer gets
    - Driving table should not necessarily be the table with the smallest number of rows
- In the case of cost-based optimization, the driving table is first after the FROM clause. Thus, place smallest table first after FROM, and list tables from smallest to largest
    - The table order still makes a difference in execution time, even when using the cost-based optimizer

Choosing a Driving Table

- Example:
```
select doc_title
from documents doc, page_collections pc
where pc.pc_issue_date = '01-JAN-2002‘
      and pc.pc_id = doc.pc_id
```

- Which table should be driving?
  - DOCUMENTS has 110+ million rows
    - No filtering predicates in where clause, all rows will be in row source
  - PAGE_COLLECTIONS has 1.4+ million rows
    - PC_ISSUE_DATE predicate will filter down to 30 rows

Using Hints

- Hints are used to convey your tuning suggestions to the optimizer
  - Misspelled or malformed hints are quietly ignored
- Commonly used hints include:
  - ORDERED
  - INDEX(*table_alias index_name*)
  - FULL(*table_alias*)
  - INDEX_FFS(*table_alias index_name*)
  - INDEX_COMBINE(*table_alias index_name1 .. index_name_n*)
  - And_EQUAL(*table_alias index_name1 index_name2 .. Index_name5*)
  - USE_NL(*table_alias*)
  - USE_MERGE(*table_alias*)
  - USE_HASH(*table_alias*)
- Hints should be specified as: `/*+ hint */`
  - Hints should immediately follow the ‘SELECT’ keyword
  - The space following the ‘+’can be significant inside of PL/SQL, due to bug in Oracle parser (see bug #697121)
- Driving table will never have a join method hint, since there is no row source to join it to

Simple Example of Tuning with Hints

- Initial SQL

```
select doc_id, doc_title, pc_issue_date
from documents doc, page_collections  pc
where doc.pc_id = pc.pc_id and doc.doc_id = 9572422;
```

- Initial Execution Plan

```
Execution Plan
----------------------------------------------------------
0       SELECT STATEMENT Optimizer=CHOOSE (Cost=2575 Card=50 Bytes=3600)
1    0    MERGE JOIN (Cost=2575 Card=50 Bytes=3600)
2    1       TABLE ACCESS (BY INDEX ROWID) OF 'PAGE_COLLECTIONS' (Cost=2571
                                                    Card=1442348)
3    2           INDEX (FULL SCAN) OF 'PC_PK' (UNIQUE) (Cost=3084 Card=1442348)
4    1       SORT (JOIN) (Cost=3 Card=1 Bytes=60)
5    4           TABLE ACCESS (BY INDEX ROWID) OF 'DOCUMENTS' (Cost=1 Card=1
                                                            Bytes=60)
6    5              INDEX (UNIQUE SCAN) OF 'DOC_PK' (UNIQUE) (Cost=2 Card=1)
```

- Initial number of buffer gets: 444

Simple Example of Tuning with Hints

- First Tuning attempt

```
select /*+ FULL(pc)*/ doc_id, doc_title, pc_issue_date
from documents doc, page_collections  pc
where doc.pc_id = pc.pc_id and doc.doc_id = 9572422;
```

- Tuned Execution Plan

```
Execution Plan
----------------------------------------------------------
0       SELECT STATEMENT Optimizer=CHOOSE (Cost=3281 Card=50 Bytes=3600)
1    0    HASH JOIN (Cost=3281 Card=50 Bytes=3600)
2    1       TABLE ACCESS (FULL) OF 'PAGE_COLLECTIONS' (Cost=1675 Card=1442348
                                                         Bytes=17308176)
3    1       TABLE ACCESS (BY INDEX ROWID) OF 'DOCUMENTS' (Cost=1 Card=1
                                                         Bytes=60)
4    3          INDEX (UNIQUE SCAN) OF 'DOC_PK' (UNIQUE) (Cost=2 Card=1)
```

- Number of buffer gets: 364

Simple Example of Tuning with Hints

- Second Tuning attempt

```
select /*+ ORDERED USE_NL(pc)*/ doc_id, doc_title,
pc_issue_date
from documents doc, page_collections  pc
where doc.pc_id = pc.pc_id and doc.doc_id = 9572422;
```

- Second Tuned Execution Plan

```
Execution Plan
----------------------------------------------------------
0       SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=50 Bytes=3600)
1    0    NESTED LOOPS (Cost=2 Card=50 Bytes=3600)
2    1       TABLE ACCESS (BY INDEX ROWID) OF 'DOCUMENTS' (Cost=1 Card=1
                                                          Bytes=60)
3    2          INDEX (UNIQUE SCAN) OF 'DOC_PK' (UNIQUE) (Cost=2 Card=2)
4    1       TABLE ACCESS (BY INDEX ROWID) OF 'PAGE_COLLECTIONS' (Cost=1
                                                             Card=1442348)
5    4          INDEX (UNIQUE SCAN) OF 'PC_PK' (UNIQUE) (Cost=1 Card=1442348)
```

- Number of buffer gets: 7

Considerations and Cautions

- Fundamental changes to the query structure allow the optimizer different options
- Using select in the select list allowed for a GROUP BY result without a GROUP BY operation, thus avoiding costly BITMAP CONVERSION TO ROWIDS
- Other places where re-writing query can have benefits:
  - Rewrite sub-select as join, allows optimizer more options
  - consider EXISTS/NOT EXISTS and IN/NOT IN operations
- Adding hints to a large number of your SQL statements?
  - Take a step back, consider whether you need to tune your CBO params
  - Hand tuning a majority of SQL in an application will complicate code, and add a lot of time to development effort
  - As new access paths are introduced in Oracle, statements that use hints will not utilize them, and continue using the old access paths
- When individual statement tuning is necessary, a solid understanding of access paths, join order and join methods is the key to success

Considerations and Cautions

- Use hints sparingly
  - If you have the opportunity, tune via CBO parameters first
  - Don't over-specify hints
  - SQL Tuning is as important as ever:
    - Need to understand the access paths, join orders, and join methods, even if only to evaluate what the CBO is doing
    - CBO gets better with each release, but it will never know as much about the application and data model as a well-trained developer

Myths

- SQL tuned for RBO will run well in the CBO
- SQL developers do not need to be retrained to write SQL for the CBO
- 10g, 11g & 12c do not support the RBO
- You can’t run RULE and COST together
- Oracle says the CBO is unreliable and you should use RULE
- Hints can’t be used in RULE

Top 9 Oracle SQL Tuning Tips

1. Design and develop with performance in mind
2. Index wisely
3. Reduce parsing
4. Take advantage of Cost Based Optimizer
5. Avoid accidental table scans
6. Optimize necessary table scans
7. Optimize joins
8. Use array processing
9. Consider PL/SQL for “tricky” SQL

Design and Develop with Performance in Mind

- Explicitly identify performance targets
- Focus on critical transactions
  - Test the SQL for these transactions against simulations of production data
- Measure performance as early as possible
- Consider prototyping critical portions of the applications
- Consider de-normalization and other performance by design features early on

De-Normalization

- If normalizing your OLTP database forces you to create queries with many multiple joins (4 or more)
- De-normalization is the process of selectively taking normalized tables and re-combining the data in them in order to reduce the number of joins needed them to produce the necessary query results
- Sometimes the addition of a single column of redundant data to a table from another table can reduce a 4-way join into a 2-way join, significantly boosting performance by reducing the time it takes to perform the join

De-Normalization

- Example: We have the following schema:

Similarities:

user1	user2	similarity

Averages:

user	average

- Similarities table contains the similarity measure for all the possible pairs of users and Averages table the average ratings of all users in Database
- In order to update all similarity measures we need the average value for each user
- Suppose we have over 1.000.000 users stored in our Database (about 500 billions of user-pairs!)
- To avoid joining we should consider of the following schema:

Similarities:

user1	user2	similarity	average1	average2

De-Normalization

- While de-normalization can boost join performance, it can also have negative effects. For example, by adding redundant data to tables, you risk the following problems:
  - More data means reading more data pages than otherwise needed, hurting performance
  - Redundant data can lead to data anomalies and bad data
  - In many cases, extra code will have to be written to keep redundant data in separate tables in synch, which adds to database overhead
  - As you consider whether to de-normalize a database to speed joins, be sure you first consider if you have the proper indexes on the tables to be joined
    - It is possible that your join performance problem is more of a problem with a lack of appropriate indexes than it is of joining too many tables

Index Wisely

- Index to support selective WHERE clauses and join conditions
- Use concatenated indexes where appropriate
- Consider over-indexing to avoid table lookups
- Consider advanced indexing options
    - Hash Clusters
        - When a table is queried frequently with equality queries
        - You can avoid using the ORDER BY clause, as well as sort operations
        - More administrative overhead
    - Bit mapped indexes
        - Can use large amounts of memory
        - Use sparingly
    - Index only tables

Index Wisely

- Do not index columns that are modified frequently
  - UPDATE statements that modify indexed columns and INSERT and DELETE statements that modify indexed tables take longer than if there were no index
    - must modify data in indexes as well as data in tables
- Do not index keys that appear only with functions or operators
  - A WHERE clause that uses a function (other than MIN or MAX) or an operator with an indexed key does not make available the access path that uses the index (except with function-based indexes)
- When choosing to index a key, consider whether the performance gain for queries is worth the performance loss for INSERTs, UPDATEs, and DELETEs and the use of the space required to store the index
  - You might want to experiment by comparing the processing times of the SQL statements with and without indexes
    - You can measure processing time with the `SQL trace` facility

Reduce Parsing

- Use Bind variables
  - Bind variables are key to application scalability
  - If necessary set cursor CURSOR_SHARING to FORCE
- Reuse cursors in your application code
  - How to do this depends on your development languages
- Use a cursor cache
  - Setting SESSION_CACHED_CURSORS can help applications that are not re-using cursors

Bind Values

- Use bind variables rather than literals in SQL statements whenever possible
- For example, the following two statements cannot use the same shared area because they do not match character for character:

```
SELECT employee_id FROM employees
WHERE department_id = 10;
SELECT employee_id FROM employees
WHERE department_id = 20;
```

- By replacing the literals with a bind variable, only one SQL statement would result, which could be executed twice:

```
SELECT employee_id FROM employees
WHERE department_id = :dept_id;
```

Bind Values

- In SQL*Plus you can use bind variables as follows:

```
SQL> variable dept_id number
SQL> exec :dept_id := 10
SQL> SELECT employee_id FROM employees
           WHERE department_id = :dept_id;
```

- What we've done to the SELECT statement now is take the literal value out of it, and replace it with a placeholder (our bind variable), with SQL*Plus passing the value of the bind variable to Oracle when the statement is processed.

Cursors

Instead of:

```
select count(*) into tot from s_emp
where emp_id = v_emp_id;
```

Declare a cursor for the count:

```
cursor cnt_emp_cur(v_emp_id number)  is
select count(*) emp_total from s_emp
emp_id = v_emp_id;
cnt_emp_rec cnt_emp%rowtype;
```

Or if just checking for existence

```
cursor cnt_emp_cur(v_emp_id
number)  is
select emp_id from s_emp where
where
emp_id= v_emp_id and rownum = 1;
```

And then do the fetch from this cursor:

```
…
open cnt_emp(v_emp_id);
fetch cnt_emp into cnt_emp_rec;
…
close cnt_emp;
```

Take Advantage of the Cost Based Optimizer

- The older rule based optimizer is inferior in almost every respect to the modern cost based optimizer; basic RBO problems

◆Incorrect driving table	40%
◆Incorrect index	40%
◆Incorrect driving index	10%

- Using the cost based optimizer effectively involves:
  - ◆Regular collection of table statistics using the ANALYZE or DBMS_STATS command
  - ◆Understand hints and how they can be used to influence SQL statement execution
  - ◆Choose the appropriate optimizer mode;
    - FIRST_ROWS is best for OLTP applications
    - ALL_ROWS suits reporting and OLAP jobs

Analyze – Wrong Data

- Tables were analyzed with incorrect data volumes

- When does this occur?
  - Table rebuilt
  - Index added
  - Migrate schema to production
  - Analyze before a bulk load

- Missing Stats:
  - Oracle will estimate the stats for you
  - These stats are for this execution only
  - Stats on Indexes

Avoid Accidental Table Scans

- Table scans that occur unintentionally are a major source of poorly performing SQL

- Causes include:
  - Missing Index
  - Using “!=“ , “<>” or NOT
    - Use inclusive range conditions or IN lists
  - Looking for values that are NULL
    - Use NOT NULL values with a default value
  - Using function on indexed columns

Factors that can Cause an Index not to be Used

1) Using a function on the left side

```
SELECT * FROM s_emp
WHERE substr(title,1,3) = ‘Man’;
SELECT * FROM s_emp
WHERE
trunc(hire_date)=trunc(sysdate);
```

→Since there is a function around this column the index will not be used. This includes Oracle functions to_char, to_number, ltrim, rtrim, instr, trunc, rpad , lpad.

Solution:

Use ‘like’ :

```
     SELECT * FROM s_emp
     WHERE title LIKE ‘Man%’;
```

Use >, < :

```
     SELECT * FROM s_emp
     WHERE hire_date >= sysdate
     AND hire_date < sysdate + 1;
```

Factors that can Cause an Index not to be Used

2) Comparing incompatible data types

```
SELECT * FROM s_emp
WHERE employee_number = ‘3’;
SELECT * FROM s_emp
WHERE hire_date = ’12-jan-01’;
```

→There will be an implicit to_char conversion used

→ There will be an implicit to_date conversion used

Solution:

```
SELECT * FROM s_emp
WHERE employee_number = 3;
SELECT * FROM s_emp
WHERE hire_date = to_date(’12-jan-01’);
```

Factors that can Cause an Index not to be Used

3) Using null and not null

```
SELECT * FROM s_emp
WHERE title IS NOT NULL;
SELECT * FROM s_emp
WHERE title IS NULL;
```

→ Since the column title has null values, and is compared to a null value, the index can not be used

Solution:

```
SELECT * FROM s_emp
WHERE title >= ‘ ‘;
```

Use an Oracle hint:

```
SELECT /*+ index (s_emp) */ *
FROM s_emp WHERE title IS NULL;
```

→Oracle hints are always enclosed in /*+ */ and must come directly after the select clause
The index hint causes indexes to be used

Factors that can Cause an Index not to be Used

4) Adding additional criteria in the where clause for a column name that is of a different index

```
SELECT * FROM s_emp
WHERE title= ‘Manager’
AND salary = 100000;
```

→ Column title and salary have separate indexes on these columns

Solution:

(Use an Oracle hint)

```
SELECT /*+ index (s_emp) */
FROM s_emp
WHERE title= ‘Manager’
AND salary = 100000;
```

→ Oracle hints are always enclosed in /*+ */ and must come directly after the select clause
The index hint causes indexes to be used
S_EMP is the Oracle table

Make sure most Restrictive Indexes are being Used by using Oracle hints

```
SELECT COUNT(*) FROM vehicle
WHERE
assembly_location_code = 'AP24A'
AND production_date = '06-apr-01';

COUNT(*)
----------
       787
Elapsed: 00:00:10.00
```

→ This does not use an index

→Notice it is 10 seconds

Make sure most Restrictive Indexes are being Used by using Oracle hints

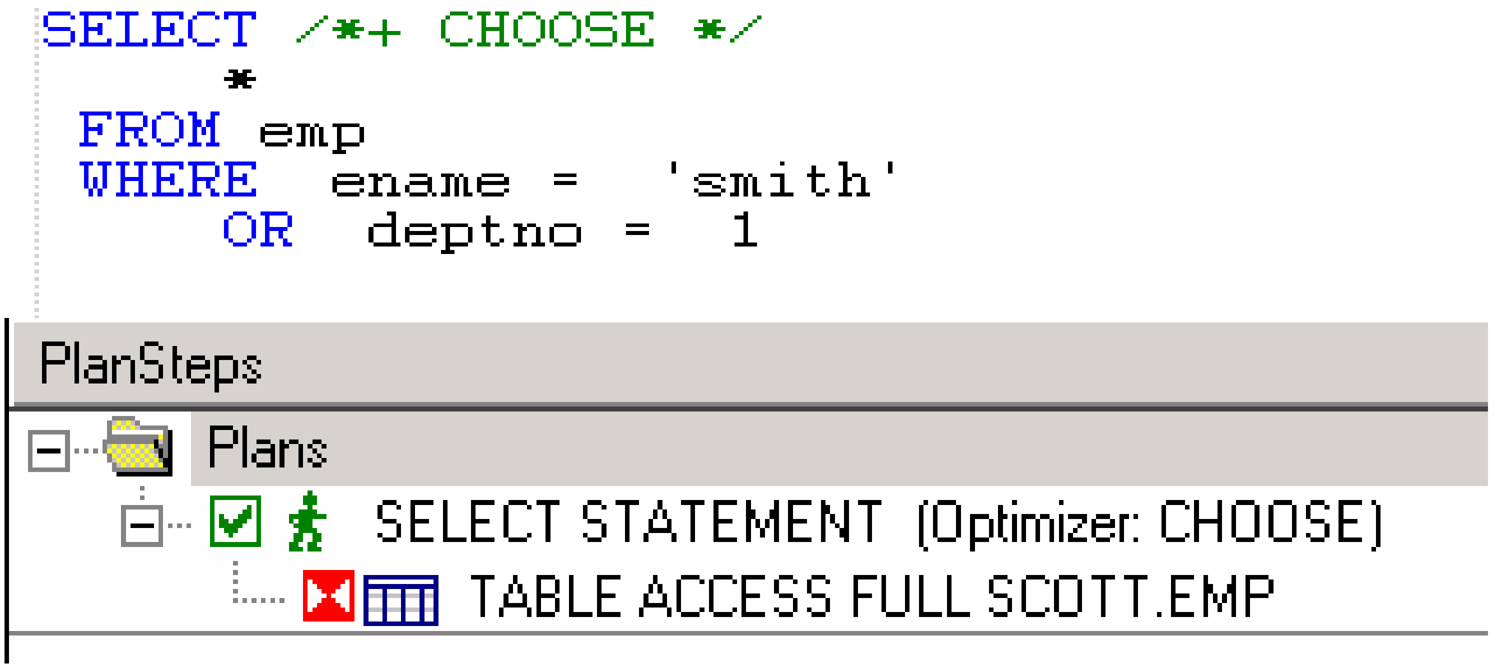
```
SELECT
/*+ index (vehicle FKI_VEHICLE_1)
*/ COUNT(*)
FROM vehicle
WHERE
assembly_location_code = 'AP24A'
AND production_date = '06-apr-01';

COUNT(*)
----------
       787

Elapsed: 00:00:00.88
```

→ This does use an index

→ Notice it is less than 1 second. USE THE MOST SELECTIVE INDEX that will return the fewest records

Some Idiosyncrasies

- Condition Order:
  - The order of your where clause will effect the performance
- OR may stop the index being used
  - break the query and use UNION

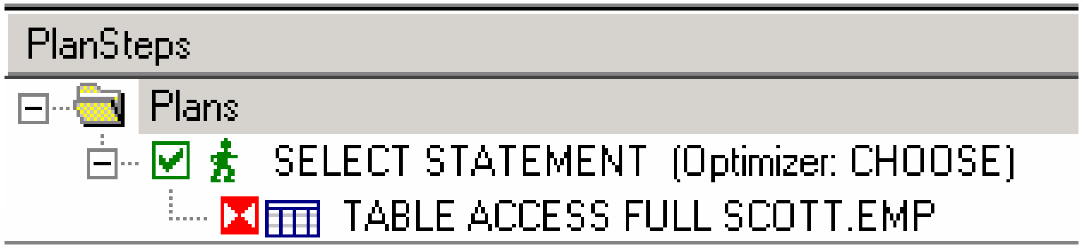


```
SELECT /*+ CHOOSE */
     *
 FROM emp
 WHERE  ename =  'smith'
     OR  deptno =  1
```

```
PlanSteps
Plans
  SELECT STATEMENT (Optimizer: CHOOSE)
    TABLE ACCESS FULL SCOTT.EMP
```

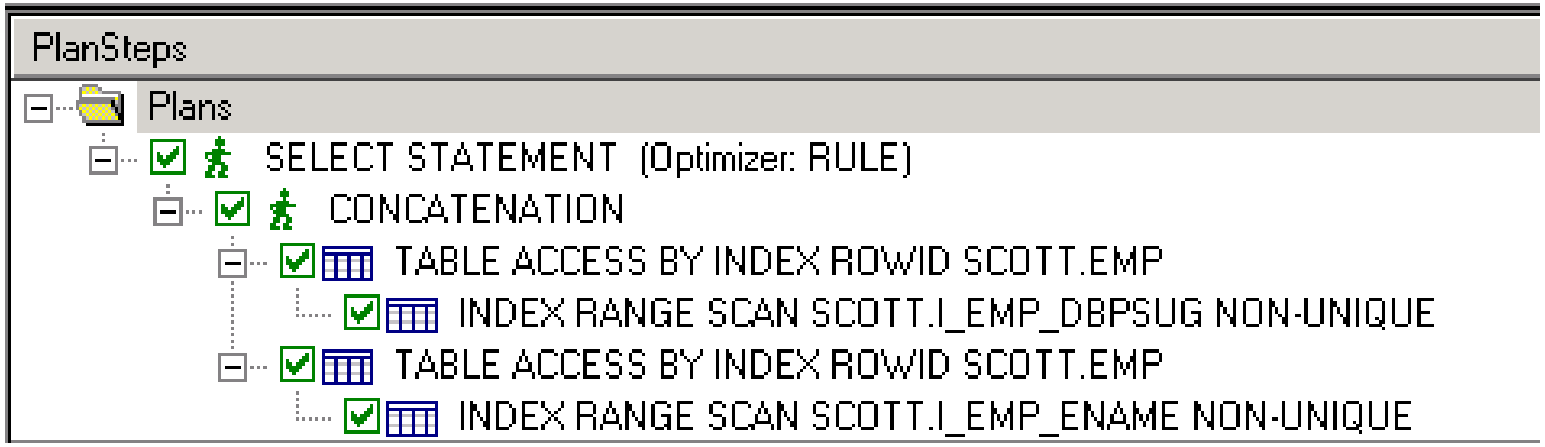
OR

```
SELECT /*+ INDEX(emp, I_EMP_DBPSUG) INDEX(emp, I_EMP_ENAME) */
     *
 FROM emp
 WHERE  ename =  'smith'
     OR  deptno =  1
```



```
PlanSteps
⊟ Plans
  ⊟ SELECT STATEMENT  (Optimizer: CHOOSE)
      TABLE ACCESS FULL SCOTT.EMP
```

```
SELECT   *
 FROM emp
 WHERE  ename =  'smith'
     OR  deptno =  1
```



```
PlanSteps
⊟ Plans
  ⊟ SELECT STATEMENT  (Optimizer: RULE)
    ⊟ CONCATENATION
      ⊟ TABLE ACCESS BY INDEX ROWID SCOTT.EMP
          INDEX RANGE SCAN SCOTT.I_EMP_DBPSUG NON-UNIQUE
      ⊟ TABLE ACCESS BY INDEX ROWID SCOTT.EMP
          INDEX RANGE SCAN SCOTT.I_EMP_ENAME NON-UNIQUE
```

Optimize Necessary Table scans

- There are many occasions where a table scan is the only option, If so:
  - Consider parallel query option
- Try to reduce size of the table
  - Adjust PCTFREE and PCTUSED
  - Relocate infrequently used long columns
- Improve the caching of the table
  - Use the CACHE hint or table property
  - Implement KEEP and RECYCLE pools
- Partition the table
- Consider the fast full index scan

IN Lists

```
SELECT empno FROM emp WHERE
deptno IN (10,20,30)
```

- Rewritten as:

```
SELECT empno FROM emp
WHERE deptno = 10
UNION ALL
SELECT empno FROM emp
WHERE deptno = 20
UNION ALL
SELECT empno FROM emp
WHERE deptno = 30
```

Data Partitioning

- If you are designing a database that potentially could be very large, holding millions or billions of rows, consider the option of horizontally partitioning your large tables
  - Horizontal partitioning divides what would typically be a single table into multiple tables, creating many smaller tables instead of a single, large table
  - The advantage of this is that is generally is much faster to query a single small table than a single large table
- For example, if you expect to add 10 million rows a year to a transaction table, after five years it will contain 50 million rows
  - In most cases, you may find that most queries (although not all) queries on the table will be for data from a single year
  - If this is the case, if you partition the table into a separate table for each year of transactions, then you can significantly reduce the overhead of the most common of queries

When Joining…

- Make sure everything that can be joined is joined (for 3 or more tables)

  Instead of:

```
SELECT * FROM t1, t2, t3
WHERE t1.emp_id = t2.emp_id
AND t2.emp_id = t3.emp_id
```

  add:

```
SELECT * FROM t1, t2, t3
WHERE t1.emp_id = t2.emp_id
AND t2.emp_id = t3.emp_id
AND t1.emp_id = t3.temp_id;
```

- Make sure smaller table is first in the from clause

Joining too Many Tables

- The more tables the more work for the optimizer
- Best plan may not be achievable

Tables	Permutations
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880

Tables	Permutations
10	3628800
11	39916800
12	479001600
13	6226020800
14	87178291200
15	1307674368000

Use ARRAY Processing

- Retrieve or insert rows in batches, rather than one at a time
- Methods of doing this are language specific
- Suppose that a new user registers in our Database
- We have to create an entry in Similarities table for each pair of new user and the existing users
- Instead of selecting all the existing users and make the insertion individually we should use the following statement:

```
insert into
similarities (user1, user2, similarity)
‘new_user_id’ as user1,
select user_id from users as user2,
0 as similarity;
```

Consider PL/SQL for “Tricky” SQL

- With SQL you specify the data you want, not how to get it
  - Sometime you need to specifically dictate your retrieval algorithms
- For example:
  - Getting the second highest value
  - Correlated updates
  - SQL with multiple complex correlated subqueries
  - SQL that seems to hard to optimize unless it is broken into multiple queries linked in PL/SQL
- Using explicit instead of implicit cursors
  - Implicit cursors always take longer than explicit cursors because they are doing an extra to make sure that there is no more data
- Eliminating cursors where ever possible

When your SQL is Tuned, Look to your Oracle Configuration

- When SQL is inefficient there is limited benefit in investing in Oracle server or operating system tuning
- However, once SQL is tuned, the limiting factor for performance will be Oracle and operating system configuration
- In particular, check for internal Oracle contention that typically shows up as latch contention or unusual wait conditions (buffer busy, free buffer, etc)

Other Parameters

- OPTIMIZER_MAX_PERMUTATIONS
  - Remember the too many joins?
  - Default is 80,000
  - Can lead to large Parse times
  - Altering can lead to non optimal plan selection

- OPTIMIZER_INDEX_CACHING
  - Represents # of blocks that can be found in the cache
  - Range 0 - 99
  - Default is 0 – implies that index access will require a *physical read*
  - Should be set to 90

Other Parameters

- OPTIMIZER_INDEX_COST_ADJ
  - Represents cost of index access to full table scans
  - Range 1 – 10000
  - Default is 100 – Means index access is as costly as Full Table Scan
  - Should be between 10 – 50 for OLTP and approx 50 for DSS
- DB_FILE_MULTIBLOCK_READ_COUNT
  - Setting too high can cause Full Table Scans
  - Can adjust for this by setting OPTIMIZER_INDEX_COST_ADJ
- DB_KEEP_CACHE_SIZE
- DB_RECYCLE_CACHE_SIZE
- DB_BLOCK_HASH_BUCKETS

References

- Dennis Shasha and Phillipe Bonnet *Database Tuning : Principles Experiments and Troubleshooting Techniques*, Morgan Kaufmann Publishers 2002.
- Mark Levis *Common tuning pitfalls of Oracle's optimizers,* Compuware
- Duane Spencer *TOP tips for ORACLE SQL tuning*, Quest Software, Inc.

Τέλος Ενότητας

Χρηματοδότηση

•Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.

•Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Κρήτης**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.

•Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο

ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ

ΕΣΠΑ 2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Σημειώματα

Σημειώματα

Σημείωμα αδειοδότησης

Σημείωμα Αναφοράς