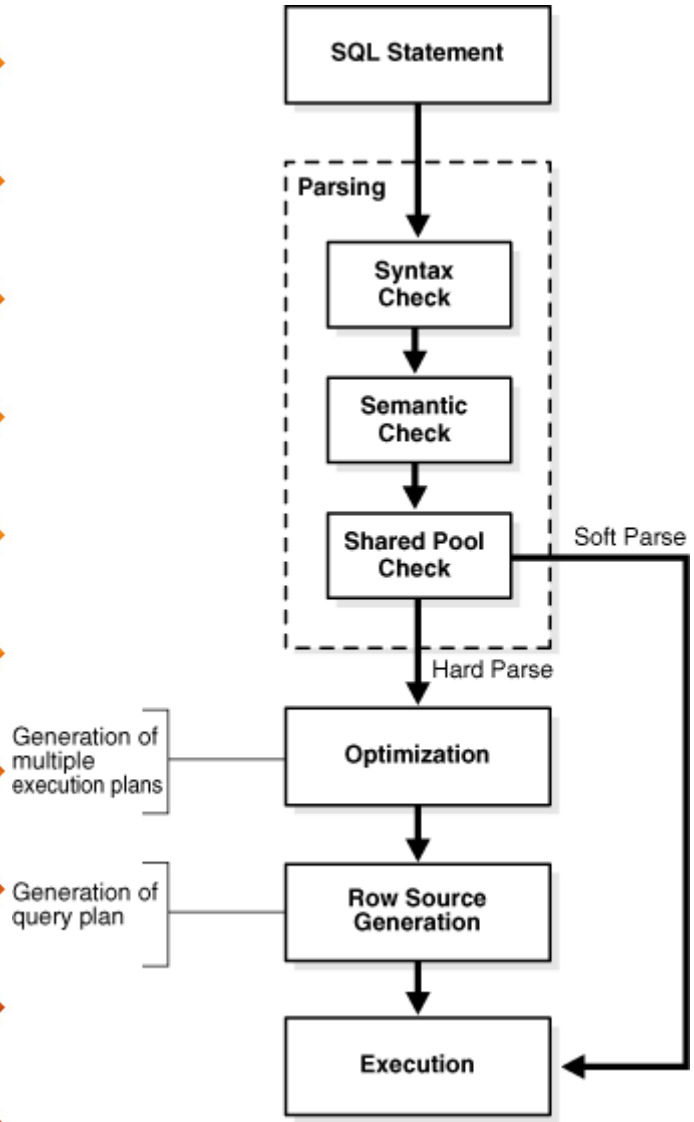
Query Optimisation

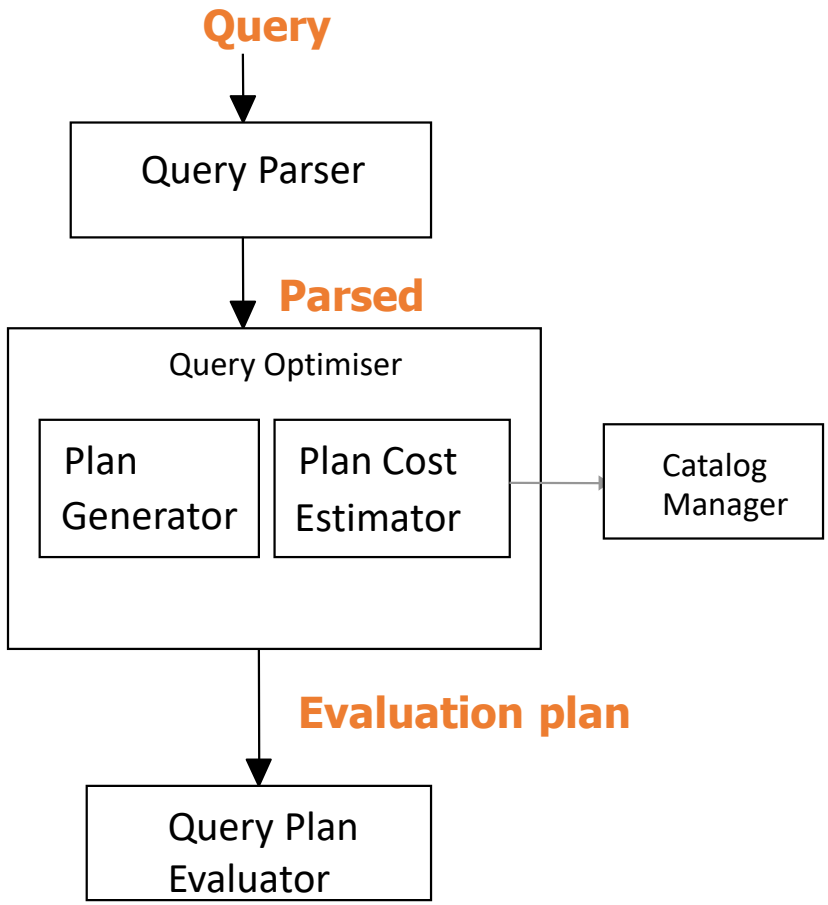
INFS602 Physical Database Design

Learning Outcomes

- Examine Oracle's Query Optimiser
- Role of Query Optimisation
- Identify basic query execution strategies
- Appreciate the role played by access paths in query optimisation

Query Execution





Oracle's Query Optimiser…

Evaluates different plans and chooses the one with lowest cost

- Makes use of statistics like no. of rows (cardinality), data distribution, selectivity, cost, etc., to choose an optimal plan
  - Access paths: full table scan, index scan, etc.
- Generates plans based on
  - Available access paths
  - Estimated cost of executing the statement using each access path or combination of paths
- Chooses the plan with the lowest estimated cost

Oracle's Query Optimiser

Oracle's optimiser works in Cost-based mode

- Available Options
  - FIRST_ROWS (for backward compatibility)
  - FIRST_ROWS_n
  - ALL_ROWS
- FIRST_ROWS - Get the first row fastest (Forms/Online queries)
- FIRST_ROWS_n -Get the first n rows fastest (Forms/Online queries)
- ALL_ROWS - Get all rows as fast as possible (Batch /Reports)

OPTIMIZER_MODE Parameter Values

Value	Description
CHOOSE	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 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. •If the data dictionary contains no statistics for any of the accessed tables, then the optimizer uses a rule-based approach.
ALL_ROWS	The optimizer uses a cost-based approach for all SQL statements in the session regardless of the presence of statistics and optimizes with a goal of best throughput (minimum resource use to complete the entire statement).
FIRST_ROWS_*n*	The optimizer uses a cost-based approach, regardless of the presence of statistics, and 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. Note: Using heuristics sometimes leads the CBO to generate a plan with a cost that is significantly larger than the cost of a plan without applying the heuristic. FIRST_ROWS is available for backward compatibility and plan stability.
RULE	The optimizer chooses a rule-based approach for all SQL statements regardless of the presence of statistics.

SQL Trace Statistics for Parses, Executes, and Fetches.

- **COUNT** - Number of times a statement was parsed, executed, or fetched.
- **CPU** - Total CPU time in seconds for all parse, execute, or fetch calls for the statement.
- **ELAPSED** - Total elapsed time in seconds for all parse, execute, or fetch calls for the statement.
- **DISK** - Total number of data blocks physically read from the datafiles on disk for all parse, execute, or fetch calls.
- **QUERY** - Total number of buffers retrieved in consistent mode for all parse, execute, or fetch calls. Usually, buffers are retrieved in consistent mode for queries.
- **CURRENT** - Total number of buffers retrieved in current mode. Buffers are retrieved in current mode for statements such as INSERT, UPDATE, and DELETE.

Interpreting the Results

Look for statements with

1. **A high CPU, elapsed time compared with overall totals**
2. **# disk, query, current much higher than rows returned**
3. **High parse statistics (cpu and elapsed time)**

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.04	0.06	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	3	4.55	7.00	3280	11330	1166	2
Total	5	4.59	7.06	3280	11330	1166	2

EXPLAIN PLAN statement

- displays execution plans chosen by the Oracle optimizer for SELECT, UPDATE, INSERT, and DELETE statements.
  - it is the sequence of operations Oracle performs to run the statement.
  - the 'trace' facility produces the execution plan as part of its output
- In the 'explain plan' output, look for
  1. Steps where the #rows processed is high
  2. Inefficient retrieval methods
  3. Inefficient driving table / join orders

Explain Plan Output – example 1

```
SELECT phone_number  FROM emp
WHERE phone_number LIKE '650%';
```

Rows	Row Source Operation
10	Filter
10220	TABLE ACCESS FULL EMP

Note: 1) Every row in the table employees is accessed, and the WHERE clause is evaluated for every row.
2) The filter operation returns rows satisfying the SELECT statement (that satisfy WHERE clause conditions).

Explain Plan Output – example 2

```
SELECT ename  FROM emp
WHERE ename LIKE 'Pe%';
```

Id	Row Source Operation
117	Filter
10220	INDEX RANGE SCAN Emp_ename_idx

Note:
1) Index on ename column of Employee table is used in a range scan operation to evaluate the WHERE clause criteria.
2) The filter operation returns rows satisfying the SELECT statement (that satisfy WHERE clause conditions).

Using Hints

- Oracle allows users to tune queries by providing hints
- These hints are used to override the strategies used by Oracle in optimising queries

```
SELECT /*+ index (table_name, index_name) */ col1, col2
  FROM …..
```

- This hint forces Oracle to use the index specified by *index_name*
  - For example, if Optimiser has NOT used the index on ename column, you could specify:
  - SELECT /*+ index (emp, emp_ename_idx) */ ename
    FROM emp WHERE ename LIKE ‘Pe%';

Query Evaluation Plans

- Let's look at an example
- Suppose we have the following database on Sailors

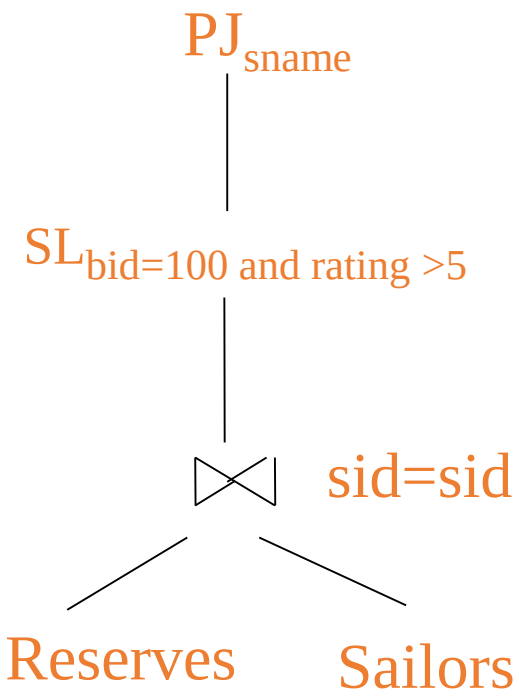
```
Sailors(sid, sname, rating, dob)
Reserves(sid, bid, day, rname)
```

- Suppose that we have the following query:

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

Execution Strategy

- A query execution strategy is generally expressed as a tree structure



Strategies for Joining

- Three basic strategies exist:
  1. Nested Loops
  2. Sort Merge
  3. Hash Join
- No one strategy is universally the best
- DBMS optimisers usually estimate the costs involved for each strategy and pick the one that yields the lowest cost

Nested Loops Join

- One of the tables to be joined is designated as the *outer* and the other is called the *inner*
- The join is carried out by scanning all the *inner* tables rows for EACH row of the *outer* table
- This is a viable strategy if an index exists on the joining column of the inner table
  - Useful when small subsets of data are being joined and if the join condition is an efficient way of accessing the second table
  - Inner table is driven from (dependent on) the outer table
  - oracle hint: USE_NL(table1, table2)

Nested Loops Join example

```
SELECT e.employee_id, j.job_title, e.salary
FROM employees e, jobs j
WHERE e.job_id = j.job_id
AND e.employee_id < 103;
```

Id	Operation	Name
4	NESTED LOOPS	
1	TABLE ACCESS FULL	EMPLOYEES
3	TABLE ACCESS BY INDEX ROWID	JOBS
2	INDEX UNIQUE SCAN	JOB_ID_PK

Sort-Merge Join

- This is a two-phase algorithm
  - Phase 1, the two relations are sorted on the join column
  - Phase 2 – the sorted lists are merged together
- If the join is on a primary -> foreign key combination, then the merging phase can be very efficient
- Optimizer may choose a sort merge when
  - joining large sets of data
  - the join condition between two tables is an inequality condition, (i.e. not an equi-join) like <. <=, >, >=
  - Oracle hint (USE_MERGE)

Hash Join

- Conceptually similar to the Sort-Merge Join method
- The main difference is that *hashing*, instead of *sorting* is used to ensure efficient merging
- The basic idea is to hash both relations on the join attribute using the *same* hash function
- Thus each relation will be partitioned into (say, k) buckets
- The join can now be carried out efficiently, since we only need to merge rows from corresponding buckets

Hash Join example

```
SELECT o.customer_id, i.unit_price, i.quantity
FROM
  orders o, order_items i
WHERE
  l.order_id = o.order_id;
```

Id	Operation	Name
0	SELECT STATEMENT	
3	HASH JOIN	
1	TABLE ACCESS FULL	ORDERS
2	TABLE ACCESS FULL	ORDER_ITEMS

Hash Join

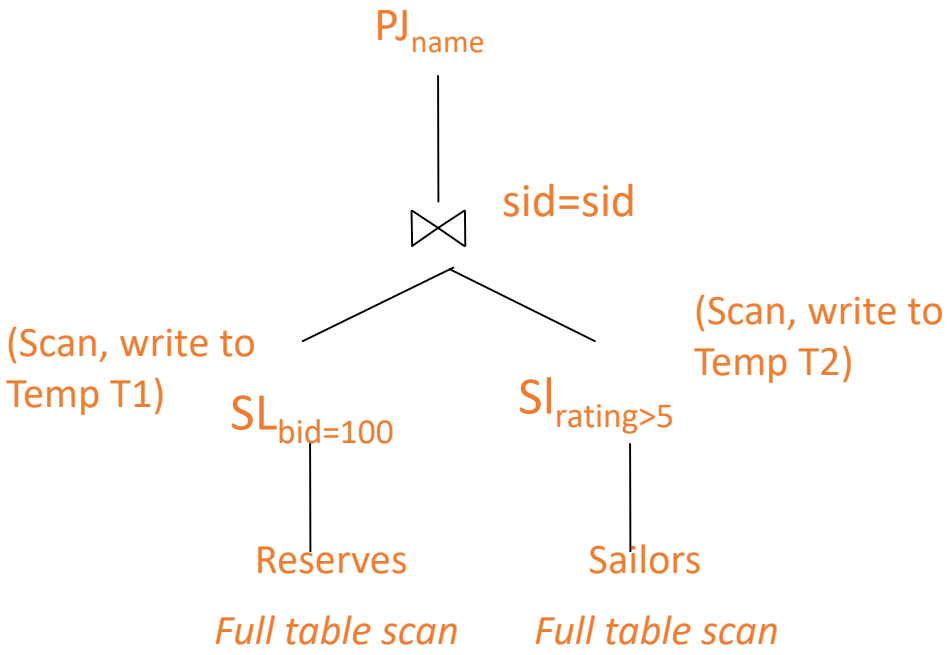
Optimizer may choose a hash join when

- two tables are joined using an equijoin
- a large amount of data needs to be joined
- a large fraction of a small table needs to be joined

- Oracle hint: USE_HASH

Alternative Query Plans

- We use the Sailor database as an example
- Consider the following query plan



Estimation of Result Sizes

- Result sizes play a crucial part in determining the best plan to be used
- Consider a general query:

SELECT attribute list

FROM relation list

WHERE $term_1$ and $term_2$ and ..... $term_n$

- The maximum number of rows returned by the query is the product of the cardinalities of the relations in the FROM clause
- However, each term in the WHERE clause reduces the ultimate result size by a factor

Reduction Factors

- Thus, optimisers require formulae for estimating the values of these *reduction factors*

Predicate Type	Reduction Factor
column = value	1/NKeys(I)
column1 = column2	1/Max(NKeys(I1),NKeys(I2))
column>value	(High(I)-value)/(High(I)-Low(I)+1)

Query Plan Cost – Reduction factors

- Suppose that we have the following statistics
  1. Number of boats: 100
  2. Sailor ratings are in the range 1 to 10
  3. Number of blocks for Sailors: 500
  4. Number of blocks for Reserves: 1000
  5. Number of buffer pages for sorting: 5
- Predicates in query:

  R.bid =100 S.rating > 5
- Reduction factor =

  1/100 10-5/10+1-1 = 1/2

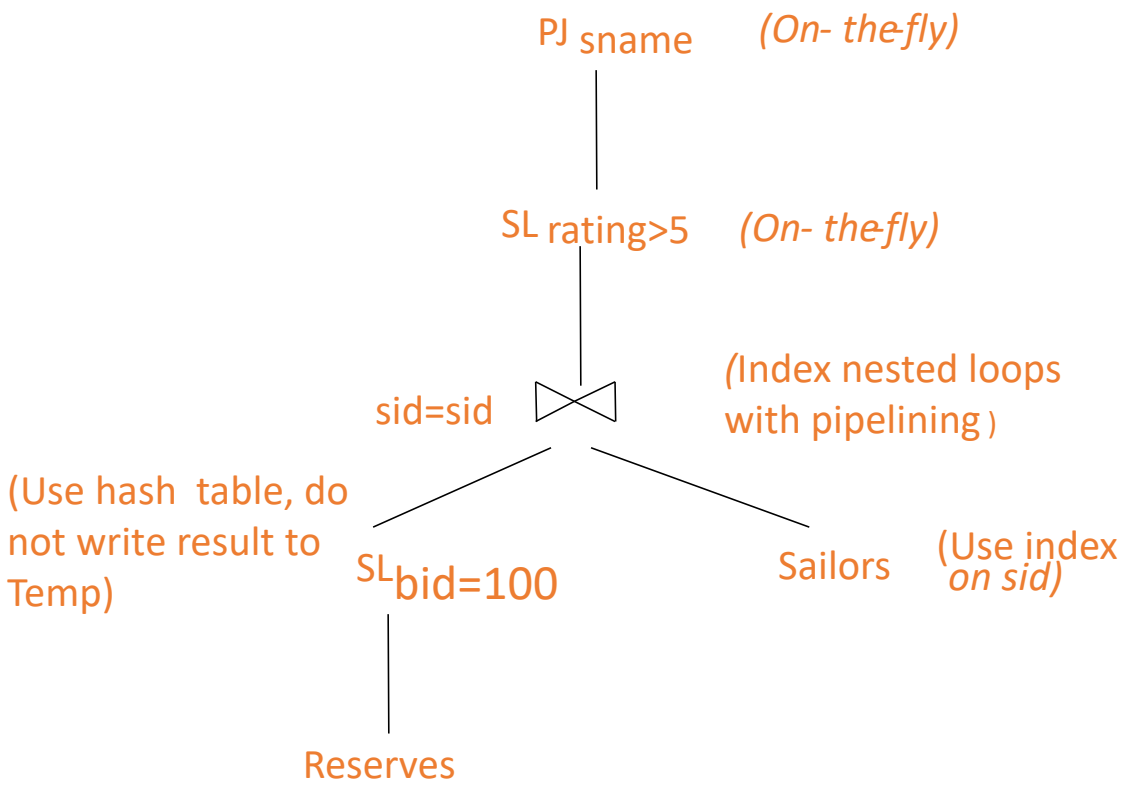
Result size=1/100*1000 = 1/2 * 500

= 10 blocks = 250 blocks

Query Plan Cost

- The cost of the select operation on *Reserves* = cost of scanning *Reserve blocks*+cost of writing out T1
- Thus cost = 1000+10 (assuming that reservations are uniformly distributed amongst boats)
- Similarly, cost of the SELECT operation on Sailors = 500+250 (assuming that there are 10 ratings)
- Cost of sorting T1 = 2*2*10 =40 (since two passes are required)
- Similarly, the cost of sorting T2 = 4*2*250=2000 (4 passes are required)
- Cost of the Merge phase is 10+250
- Thus total cost = 1010+750+40+2000+260 = 4060

Alternative Query Plan



Alternative Query Plan

- With a hash table on the *bid* column of *Reserves* and an index on the *sid* column of *Sailors* a more efficient plan can be formulated
- In this plan, we do not perform the selections before the join in order to take advantage of the access paths
- This illustrates an important point: *Optimisers that rely on heuristics (rules) instead of cost estimates can end up choosing inefficient plans*

Improving Accuracy of Size Estimates

- All the estimates so far are based on uniform distribution of data
- In practice, data can be quite skewed
- Commercial DBMSs provide methods of gathering statistics on Tables and Columns
- Data distribution across a Column is modelled by a Histogram

Use of Histograms

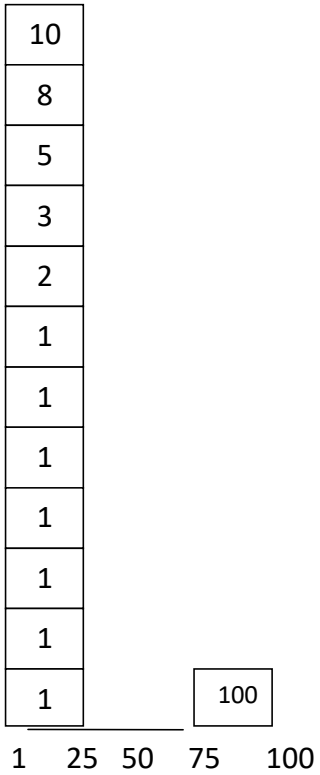
- Histograms are of two types:
  1. Width balanced
  2. Height balanced
- Width-balanced histograms divide the data into a fixed number of equal-width ranges and then count the number of values falling into each range
- Height-balanced histograms place approximately the same number of values into each range so that the endpoints of the range are determined by how many values are in that range

Height Balanced Histograms

- Height balanced histograms are far more accurate when data is skewed
- Histograms should be created on indexed columns which are known to have a high degree of data skew
- Since histograms are persistent objects they should not be created when:
    1. The column data is uniformly distributed
    2. The column is not used in WHERE clauses of queries.
    3. The column is unique and is used only with equality predicates.

Height-Balanced vs. Width-Balanced Histograms

Width-Balanced



Height-Balanced

1			
1	1	5	100
1	1	3	10
1	1	2	8
1	1	5	100

Comparative Example

Without Histograms

Predicate	Selectivity
column =1 (or any other particular value)	0.01
column <=3	0.03 (the odds that it is a 1, 2, or 3)
column >3	0.97 (the odds that it is a 4, 5, 6, 7, ……)

With Histograms

Predicate	Selectivity	Explanation
column =1	0.5	2 buckets
column <=3	0.675	2.5 buckets
column >3	0.375	1.5 buckets

Using the CBO – Oracle10g

- To use CBO, it is essential to collect statistics
  - EXECUTE *dbms_stats.gather_table_stats (‘*SHOBA*’,* 'S_SALES*’);*
- To generate BOTH table statistics and a histogram use:
  - EXECUTE dbms_statS.Gather_table_STATS (OWNNAME =>'SHOBA', TABNAME =>'S_SALES', METHOD_OPT =>'FOR ALL COLUMNS SIZE AUTO')

Viewing Histogram Statistics – 10g

```
SELECT column_name, num_distinct,
  num_buckets, histogram
FROM USER_TAB_COL_STATISTICS
WHERE table_name = 'S_SALES'
```

COLUMN_NAME	NUM_DISTINCT	NUM_BUCKETS	HISTOGRAM
SALE_ID	3269	254	HEIGHT BALANCED
CUST_ID	6	6	FREQUENCY
PROD_ID	181	181	FREQUENCY
QTY	6	1	NONE
PRICE	105	1	NONE
TRANS_DATE	36	1	NONE

Viewing Histogram Statistics – 10g

- Query the appropriate data dictionary views to view statistics on tables, indexes and columns in the data dictionary.
- Refer to the Database Performance Tuning Guide (Chapter 14-Page 16) for a listing of more of these views.

References

- Fundamentals of Database Systems, Elmasri & Navathe, 4th Ed., Ch 15
- Oracle 10g Database Performance Tuning Guide, Chapters 11, 14, 19, and 20
- Oracle 10g Concepts
- Oracle Database Online Documentation 12c Release 1 (12.1)

Review Questions

- Explain the role of the query optimiser in improving performance in a DBMS
- In what situations would you consider using query optimisation hints in the Oracle environment?