Politecnico di Torino

Database Management Systems

Oracle Optimizer

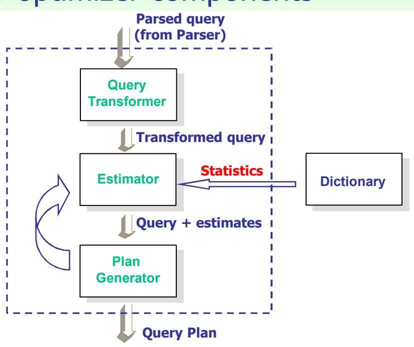
Tania Cerquitelli
tania.cerquitelli@polito.it

A.A. 2012-2013

Optimizer objective

- A SQL statement can be executed in many different ways
- The query optimizer determines the most efficient way to execute a SQL statement after considering many factors (e.g., objects referenced, conditions specified in the query)
- The output from the optimizer is a plan that describes an optimum method of execution (i.e., minimum execution cost)
- The cost is an estimated value proportional to the expected resource use (i.e., I/O, CPU, and memory) needed to execute the statement with a particular plan

Query optimizer components



Query Transformer

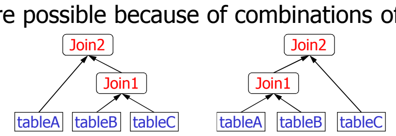
- The input is a parsed query, represented by a set of query blocks
- The query blocks are nested or interrelated to each other (sub-queries)
  - the innermost query block is optimized first and a sub-plan is generated for it (bottom-up approach)
- The objective is to determine if it is advantageous to change the form of the query so that it enables generation of a better query plan

Estimator

- The goal is to estimate the overall cost of a given plan by exploiting three different types of measures
  - **Selectivity** represents a fraction of rows from a row set
  - **Cardinality** represents the number of rows in a row set
  - **Cost** represents units of work or resource used. The query optimizer uses disk I/O (access path), CPU usage, and memory usage as units of work
- Estimator uses statistics from the dictionary to compute the measures which improve the degree of accuracy of the evaluation
  - histogram of different values in a table column

Plan Generator

- Its main function is to
  - try out different possible plans for a given query
  - and pick the one that has the lowest cost
- Many plans are possible because of combinations of different
  - access paths
  - join methods
  - join orders



- It uses an internal cutoff to reduce the number of plans explored
  - the cutoff is based on the cost of the current best plan
  - if cutoff is high, more plans are explored, and vice versa

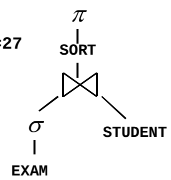
Optimizer operations

Operation	*Description*
Evaluation of expressions and conditions	The optimizer first evaluates expressions and conditions containing constants as fully as possible
Statement transformation	For complex statements involving, for example, correlated sub-queries or views, the optimizer might transform the original statement into an equivalent join statement
Choice of optimizer goals	The optimizer determines the goal of optimization
Choice of access paths	For each table accessed by the statement, the optimizer chooses one or more of available access paths to obtain data
Choice of join orders	For a join statement that joins more than two tables, the optimizer chooses which pair of tables is joined first, and then which table is joined to the result, and so on
Choice of join methods	For a join statement that joins more than two tables, the optimizer chooses which join method is exploited to perform the required operation

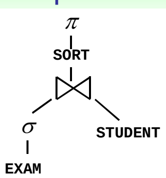
Example

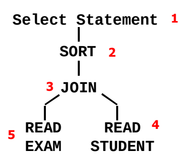
```
STUDENT (SId, SSurname, SName)
COURSE (CCode, PId, Year, Semester)
EXAM (CCode, SId, Date, Score)
```

```
Query: SELECT SName, S.Sid
       FROM EXAM E, STUDENT S
       WHERE S.Sid=E.Sid and Score>=27
       ORDER BY SName
```



Example





Id	Pid	Operation	Cost
1	/	Select Statement	100
2	1	Sort	90
3	2	Join	70
4	3	Read STUDENT	40
5	3	Read EXAM + Selection	20

Overview of EXPLAIN PLAN

- It is possible to examine the execution plan chosen by the optimizer for a SQL statement by using the **EXPLAIN PLAN** statement
- When the statement is issued, the optimizer chooses an execution plan and then inserts data describing the plan into a database table
- Simply issue the **EXPLAIN PLAN** statement and then query the output table

Understanding EXPLAIN PLAN

- The **EXPLAIN PLAN** statement displays execution plans chosen by the Oracle optimizer for **SELECT**, **UPDATE**, **INSERT**, and **DELETE** statements
- A statement's execution plan is the sequence of operations Oracle performs to run the statement
- The raw source tree shows the following information
  - An ordering of the tables referenced by the statement
  - An access method for each table mentioned in the statement
  - A join method for tables affected by join operations in the statement
  - Data operations like filter, sort, or aggregation
- The plan table also contains information about the following
  - Optimization, such as the cost and cardinality of each operation
  - Partitioning, such as the set of accessed partitions
  - Parallel execution, such as the distribution method of join input order

Understanding the Query Optimizer

- The query optimizer determines, for a given SQL statement, which execution plan is most efficient (i.e., has the lowest cost)
  - by considering available access paths
  - by changing execution join orders
  - by evaluating different join methods
  - by analyzing statistics from the data dictionary for the schema objects (tables or indexes) accessed by the SQL statement

Access Paths for the Query Optimizer

- Access paths allow the retrieval of data from the database
  - Index access paths should be used for statements that retrieve a small subset of table rows
  - Full scans are more efficient when accessing a large portion of the table
- Data can be retrieved in any table by means of the following access paths
  - Full Table Scans
  - Index Scans
  - Rowid Scans

Full Table Scans

- This type of scan reads all rows from a table and filters out those that do not meet the selection criteria
- Each row is examined to determine whether it satisfies the statement's **WHERE** clause
- Physical blocks are adjacent and they are read sequentially
- Larger I/O calls are allowed, i.e., many blocks (multiblock) are read in a single I/O call
- Multiblock reads can be used to speed up the process
- The size of multiblock is initialized by the parameter **DB_FILE_MULTIBLOCK_READ_COUNT**

Full Table Scans: Example

```
STUDENT (SId, SSurname, SName)
COURSE (CCode, PCode, Year, Semester)
EXAM (CCode, SId, Date, Score)

Query: SELECT SId, CCode, Score
       FROM EXAM
       WHERE Score>=20;
```

π
|
σ
|
EXAM

Select Statement Cost = 5
↑
Table Access Full Cost = 2

Assessing I/O for Blocks

- Oracle does I/O by blocks
  - Generally multiple rows are stored in each block. The total number of rows could be clustered together in a few blocks, or they could be spread out over a larger number of blocks.
- The optimizer decision to use full table scans is influenced by the percentage of blocks accessed, not rows. This is called the **index clustering factor**
- Although the clustering factor is a property of the index, the clustering factor actually relates to the spread of similar indexed column values within data blocks in the table
  - Low clustering factor: individual rows are concentrated within fewer blocks in the table.
  - High clustering factor: individual rows are scattered more randomly across blocks in the table. It costs more to use a range scan to fetch rows by rowid, because more blocks in the table need to be visited to return the data.

Effects of Clustering Factor on Cost

- Assume the following situation
  - There is a table with 9 rows
  - There is a non-unique index on column 1
  - Column 1 currently stores the values A, B, and C
  - Oracle stores the table using only 3 blocks
- Case 1. The **index clustering factor** is low for the rows as they are arranged in the following diagram

Block 1	Block 2	Block 3
A A A	B B B	C C C

Effects of Clustering Factor on Cost

- Case 2. If the same rows in the table are rearranged so that the index values are scattered across the table blocks (rather than clustered together), then the **index clustering factor** is higher, as in the following schema.

Block 1	Block 2	Block 3
A B C	A B C	A B C

When the Optimizer Uses Full Table Scans

- Lack of index
- Retrieval of a large amount of data stored in the target table
  - If the query will access most of the blocks in the table, the optimizer uses a full table scan, even though indexes might be available
  - Full table scans can use larger I/O calls, and making fewer large I/O calls is cheaper than making many smaller calls
- Small table
  - If a table has less than **DB_FILE_MULTIBLOCK_READ_COUNT** blocks it can be read in a single I/O call, then a full table scan might be cheaper than an index range scan

Oracle indexes

- System indexes (secondary indexes) created automatically on the primary key attributes
  - SYS_#
- Primary indexes
  - Clustered Btree (physical sort)
  - Hash (bucket)
- Secondary indexes
  - Btree
  - Bitmap
  - Hash

```
CREATE INDEX IndexName ON Table (Column, …);

DROP INDEX IndexName;
```


Index Scans

- The index contains the indexed value and the rowids of rows in the table having that value
- An index scan retrieves data from an index based on the value of one or more columns in the index
  - Oracle searches the index for the indexed column values accessed by the statement
  - If the statement accesses only columns of the index, the indexed column values are read directly from the index, otherwise the rows in the table are accessed by means of the rowid
- An index scan can be one of the following types
  - Index Unique Scans
  - Index Range Scans
  - Index Full Scans
  - Fast Full Index Scans
  - Bitmap Indexes

Index Unique Scans

- This scan returns at most a single rowid for each indexed value
- Oracle performs a unique scan if a statement contains a **UNIQUE** or a **PRIMARY KEY** constraint that guarantees that only a single row is accessed
- It is used when all columns of a unique (e.g., B-tree) index or an index created as a result of a primary key constraint are specified with equality conditions

Index Range Scans

- An index range scan is a common operation for accessing selective data
- Data is returned in the ascending order of index columns. Multiple rows with identical values are sorted in ascending order by rowid
- The optimizer uses a range scan when it finds one or more leading columns of an index specified in conditions
  - col1 = :b1
  - col1 <= :b1
  - col1 > =:b1
  - and combinations of the preceding conditions for leading columns in the index
- Range scans can use unique or non-unique indexes
- Range scans avoid sorting when index columns constitute the **ORDER BY/GROUP BY** clause

Index Full Scans

- A index full scan is available if a predicate references one of the columns in the index. The predicate does not need to be an index driver.
- It is also available when there is no predicate, if both the following conditions are met
  - all of the columns in the table referenced in the query are included in the index
  - at least one of the index columns is not null
- A full scan can be used to eliminate a sort operation (required by **GROUP BY**, **ORDER BY**, **MERGE JOIN**), because the data is ordered by the index key
- It reads the blocks singly (one by one)

Fast Full Index Scans

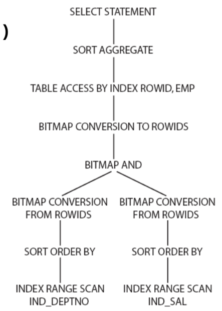
- Fast full index scans are an alternative to a full table scan when the index contains all the columns that are needed for the query, and at least one column in the index key has the **NOT NULL** constraint
- A fast full scan accesses the data in the index itself, without accessing the table
- It cannot be used to eliminate a sort operation, because the data is not ordered by the index key
- A fast full scan is faster than a normal full index scan
  - It reads the entire index using multiblock reads

Bitmap Indexes

- Bitmap indexes are most effective for queries that contain multiple conditions in the WHERE clause
- They are usually easier to destroy and re-create than to maintain
- A bitmap join uses a bitmap for key values and a mapping function that converts each bit position to a rowid

Bitmap Indexes: Example

```
EMP(Empno, Ename, Job, Mgr,
  Hiredate, Sal, Grade, Deptno)
DEPT(Deptno, Dname, Loc)
SALGRADE(Grade, Losal, Hisal)

SELECT AVG(e.sal)
FROM EMP E
WHERE E.Depno < 10 and
E.Sal > 100 and E.Sal < 200;

CREATE INDEX Ind_Deptno
On EMP(Deptno);

CREATE INDEX Ind_Sal
On EMP(Sal);
```



Bitmap Indexes

- They are most effective for queries that contain multiple conditions in the WHERE clause
- They are usually easier to destroy and re-create than to maintain
- A bitmap join uses a bitmap for key values and a mapping function that converts each bit position to a rowid. Bitmaps can efficiently merge indexes that correspond to several conditions in a WHERE clause, using Boolean operations to resolve AND and OR conditions.

Rowid Scans

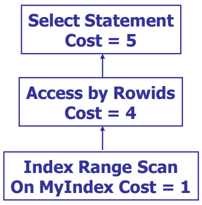
- The **rowid** of a row specifies the data file and data block (i.e., physical address) containing the row and the location of the row in that block
- Locating a row by its rowid is the fastest way to retrieve a single row
- To access a table by rowid (in Oracle)
  - Rowids of the selected rows are obtained through an index scan of one or more of the table's indexes
  - Each selected row is accessed in the table based on the physical address obtained by its rowid

Index Unique Scans: Example

```
EXPLAIN PLAN FOR
SELECT e.employee_id, j.job_title, e.salary, d.department_name
    FROM employees e, jobs j, departments d
    WHERE  e.employee_id < 103
       AND e.job_id = j.job_id
       AND e.department_id = d.department_id;

| Id  | Operation                     | Name        | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |             |     3 |   189 |    10  (10)|
|   1 |  NESTED LOOPS                 |             |     3 |   189 |    10  (10)|
|   2 |   NESTED LOOPS                |             |     3 |   141 |     7  (15)|
|*  3 |    TABLE ACCESS FULL          | EMPLOYEES   |     3 |    60 |     4  (25)|
|   4 |    TABLE ACCESS BY INDEX ROWID| JOBS        |    19 |   513 |     2  (50)|
|*  5 |     INDEX UNIQUE SCAN         | JOB_ID_PK   |     1 |       |            |
|   6 |   TABLE ACCESS BY INDEX ROWID | DEPARTMENTS |    27 |   432 |     2  (50)|
|*  7 |    INDEX UNIQUE SCAN          | DEPT_ID_PK  |     1 |       |            |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   3 - filter("E"."EMPLOYEE_ID"<103)
   5 - access("E"."JOB_ID"="J"."JOB_ID")
   7 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
```

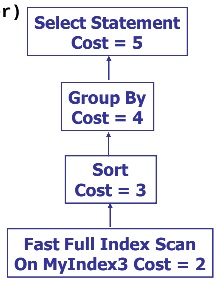
Index Range Scans: Example

```
STUDENT (SId, SSurname, SName)
COURSE (CCode, PCode, Year, Semester)
EXAM (CCode, SId, Date, Score)

Query: SELECT SId, CCode, Score
       FROM EXAM
       WHERE Score>=27;

CREATE INDEX MyIndex On EXAM(Score);
```



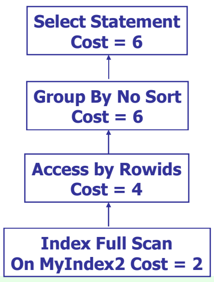
Fast Full Index Scans: Example

```
STUDENT (SId, SSurname, SName)
COURSE (CCode, PCode, Year, Semester)
EXAM (CCode, SId, Date, Score)

Query: SELECT CCode, AVG(Score)
       FROM EXAM
       GROUP BY CCode;

CREATE INDEX MyIndex3
On EXAM(CCode,Score);
```



Index Full Scans: Example

```
STUDENT (SId, SSurname, SName)
COURSE (CCode, PCode, Year, Semester)
EXAM (CCode, SId, Date, Score)

Query: SELECT SId, AVG(Score)
       FROM EXAM
       GROUP BY SId;

CREATE INDEX MyIndex2
On EXAM(Sid);
```

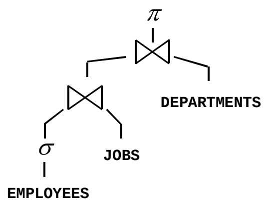


Rowid Scans: Example

```
EMPLOYEES(
        employee_id,
        department_id,
        job_id, name, birth_date, salary)
JOBS( job_id,
        grade, job_title, name)
DEPARTMENTS(
        department_id,
        department_name, city)

EXPLAIN PLAN FOR
SELECT e.employee_id, j.job_title, e.salary, d.department_name
    FROM employees e, jobs j, departments d
    WHERE  e.employee_id < 103
       AND e.job_id = j.job_id
       AND e.department_id = d.department_id;
```


Rowid Scans: Example

```
EXPLAIN PLAN FOR
SELECT e.employee_id, j.job_title, e.salary, d.department_name
    FROM employees e, jobs j, departments d
    WHERE  e.employee_id < 103
       AND e.job_id = j.job_id
       AND e.department_id = d.department_id;
```



Rowid Scans: Example

```
EXPLAIN PLAN FOR
SELECT e.employee_id, j.job_title, e.salary, d.department_name
    FROM employees e, jobs j, departments d
    WHERE  e.employee_id < 103
       AND e.job_id = j.job_id
       AND e.department_id = d.department_id;
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)
0	SELECT STATEMENT		3	189	10 (10)
1	NESTED LOOPS		3	189	10 (10)
2	NESTED LOOPS		3	141	7 (15)
* 3	TABLE ACCESS FULL	EMPLOYEES	3	60	4 (25)
4	TABLE ACCESS BY INDEX ROWID	JOBS	19	513	2 (50)
* 5	INDEX UNIQUE SCAN	JOB_ID_PK	1		
6	TABLE ACCESS BY INDEX ROWID	DEPARTMENTS	27	432	2 (50)
* 7	INDEX UNIQUE SCAN	DEPT_ID_PK	1		

```
Predicate Information (identified by operation id):
---------------------------------------------------
   3 - filter("E"."EMPLOYEE_ID"<103)
   5 - access("E"."JOB_ID"="J"."JOB_ID")
   7 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
```


Tania Cerquitelli
Politecnico di Torino

JOIN

- Join Method
  - To join each pair of row sources, Oracle must perform a join operation
  - Join methods include
    - nested loop
    - sort merge
    - hash joins
- Join Order
  - To execute a statement that joins more than two tables, Oracle joins two of the tables and then joins the resulting row source to the next table
  - This process is continued until all tables are joined into the result.

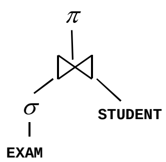
Nested Loop Joins

- Nested loop joins are useful when small subsets of data are being joined and if the join condition is an efficient way of accessing the second table
- A nested loop join involves the following steps
  - The optimizer determines the driving table and designates it as the **outer** table
  - The other table is designated as the **inner** table
  - For every row in the **outer** table, Oracle accesses all the rows in the **inner** table.
  - The outer loop is for every row in outer table and the inner loop is for every row in the inner table. The outer loop appears before the inner loop in the execution plan.

Nested Loop Joins: Example

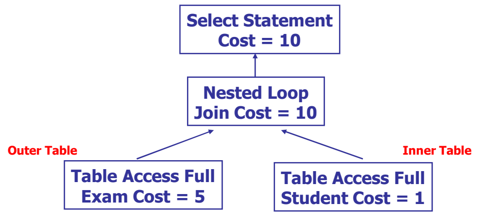
```
STUDENT (SId, SSurname, SName)
COURSE (CCode, PId, Year, Semester)
EXAM (CCode, SId, Date, Score)
```

```
Query: SELECT Surname, CCode, Score
       FROM EXAM E, STUDENT S
       WHERE S.Sid=E.Sid and
             Score>=18
```



Nested Loop Joins: Example

```
Query: SELECT Surname,CCode,Score
       FROM EXAM E, STUDENT S
       WHERE S.Sid=E.Sid and Score>=18
```



When the Optimizer Uses Nested Loop Joins

- The optimizer uses nested loop joins when joining small number of rows, with a good driving condition between the two tables.
- The outer loop is the driving row source. It produces a set of rows for driving the join condition. The row source can be a table accessed using an index scan or a full table scan.
- The inner loop is iterated for every row returned from the outer loop, ideally by an index scan.

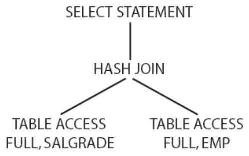
Hash Joins

- Hash joins are used for joining large data sets. The optimizer uses the smaller of two tables or data sources to build a hash table on the join key in memory. It then scans the larger table, probing the hash table to find the joined rows.
- This method is best used when the smaller table fits in available memory. The cost is then limited to a single read pass over the data for the two tables.
- The optimizer uses a hash join to join two tables if they are joined using an equijoin and if either of the following conditions are true
  - A large amount of data needs to be joined
  - A large fraction of a small table needs to be joined

Hash Joins: Example

```
EMP( Empno, Ename, Job, Mgr,
  Hiredate, Sal, Comm, Deptno)
DEPT( Deptno, Dname, Loc)
SALGRADE( Grade, Losal, Hisal)
```

```
SELECT *
FROM EMP E, SALGRADE S
WHERE E.Sal = S.Losal
      AND E.Job = 'RESEARCHER';
```



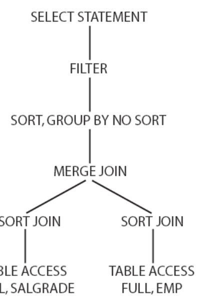
Sort Merge Joins

- Sort merge joins can be used to join rows from two independent sources
- Sort merge joins can perform better than **hash joins** if all of the following conditions exist
  - The row sources are sorted already
  - A sort operation does not have to be done (e.g., after the **GROUP BY**)
  - A sort operation can be performed for the next operation (e.g., before the **GROUP BY**)
- Sort merge joins are useful when the join condition between two tables is an inequality condition (but not a non-equality like <>) like <, <=, >, or >=
- Sort merge joins perform better than **nested loop joins** for large data sets
- The join consists of two steps
  - Sort join operation: both the inputs are sorted on the join key
  - Merge join operation: the sorted lists are merged together

Sort Merge Joins: Example

```
EMP( Empno, Ename, Job, Mgr,
  Hiredate, Sal, Comm, Deptno)
DEPT( Deptno, Dname, Loc)
SALGRADE( Grade, Losal, Hisal)
```

```
SELECT E.Sal,count(*)
FROM EMP E, SALGRADE S
WHERE E.Sal < 200 and
      E.Sal = S.Losal
GROUP BY E.Sal
HAVING COUNT(*) >2;
```



Understanding Statistics

- Optimizer statistics are a collection of data that describe more details about the database and the objects in the database
- Optimizer statistics, stored in the data dictionary, include the following:
  - Table statistics
    - Number of rows
    - Number of blocks
    - Average row length
  - Column statistics
    - Number of distinct values (NDV) in columns
    - Number of nulls in columns
    - Data distribution (histogram)
  - Index statistics
    - Number of leaf blocks
    - Levels
    - Clustering factor
  - System statistics
    - I/O performance and utilization
    - CPU performance and utilization

Statistics on Tables, Indexes and Columns

- To view statistics in the data dictionary, query the appropriate data dictionary view (USER, ALL, or DBA). These DBA_* views include the following:

  - DBA_**TABLES**
  - DBA_OBJECT_TABLES
  - DBA_**TAB_STATISTICS**
  - DBA_**TAB_COL_STATISTICS**
  - DBA_**TAB_HISTOGRAMS**
  - DBA_**INDEXES**
  - DBA_IND_STATISTICS
  - DBA_CLUSTERS
  - DBA_TAB_PARTITIONS
  - DBA_TAB_SUBPARTITIONS
  - DBA_IND_PARTITIONS
  - DBA_IND_SUBPARTITIONS
  - DBA_PART_COL_STATISTICS
  - DBA_PART_HISTOGRAMS
  - DBA_SUBPART_COL_STATISTICS
  - DBA_SUBPART_HISTOGRAMS

- **`describe table_name`** allows to view the table schema

Automatic Statistics Gathering

- Optimizer statistics are automatically gathered with the job **GATHER_STATS_JOB**
- This job is created automatically at database creation time
- By default, the job is run every night from 10 P.M. to 6 A.M. and all day on weekends
- Automatic statistics gathering should be sufficient for most
- If database objects are modified at a moderate speed automatic statistics gathering is the best approach, otherwise it may not be adequate

Manual Statistics Gathering

- If the data in database changes regularly, it is necessary to regularly gather statistics (manually) to ensure that the measures accurately represent characteristics of your database objects
- Statistics on tables, indexes, individual columns and partitions of tables are gathered using the **DBMS_STATS** package (i.e., PL/SQL package) which is also used to modify, view, export, import, and delete statistics
- When statistics are generated for a table, column, or index, if the data dictionary already contains statistics for the object, Oracle updates the existing statistics
- When statistics are updated for a database object, Oracle invalidates any currently parsed SQL statement accessing that object. The next time such a statement executes, the statement is re-parsed and the optimizer automatically chooses a new execution plan based on the new statistics

When to Gather Statistics

- For an application in which tables are being incrementally modified, new statistics need to be gather every week or every month
- For tables which are being substantially modified in batch operations, such as with bulk loads, statistics should be gathered on those tables as part of the batch operation
- The frequency of collection intervals should balance the task of providing accurate statistics for the optimizer against the processing overhead incurred by the statistics collection process.

Column Statistics and Histograms

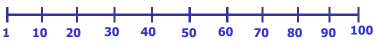
- When gathering statistics on a table, **DBMS_STATS** gathers information about the data distribution of the columns within the table (e.g., the maximum value and minimum value of the column)
- For skewed data distributions, histograms can also be created as part of the column statistics to describe the data distribution of a given column

Histograms

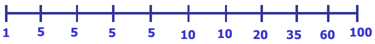
- Column statistics may be stored as histograms which provide accurate estimates of the distribution of column data.
- Histograms provide improved selectivity estimates in the presence of data skew, resulting in optimal execution plans with non-uniform data distributions
- Oracle uses two types of histograms for column statistics
  - Height-balanced histograms
  - Frequency histograms
- The type of histogram is stored in the **HISTOGRAM** column of the **USER/DBA_TAB_COL_STATISTICS** views

Height-Balanced Histograms

- In a height-balanced histogram, the column values are divided into bands so that each band contains approximately the same number of rows.
- The useful information that the histogram provides is where in the range of values the endpoints fall.
- Consider a column C with values between 1 and 100 and a histogram with 10 buckets



- If the data is not uniformly distributed, then the histogram might look similar to



Height-Balanced Histograms

```
SELECT column_name, num_distinct, num_buckets, histogram
  FROM USER_TAB_COL_STATISTICS
 WHERE table_name = 'INVENTORIES' AND column_name = 'QUANTITY_ON_HAND';

COLUMN_NAME                    NUM_DISTINCT NUM_BUCKETS HISTOGRAM
------------------------------ ------------ ----------- ---------------
QUANTITY_ON_HAND                        237          10 HEIGHT BALANCED
```

```
SELECT endpoint_number, endpoint_value
  FROM USER_HISTOGRAMS
 WHERE table_name = 'INVENTORIES' and column_name = 'QUANTITY_ON_HAND'
  ORDER BY endpoint_number;

ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
              0              0
              1             27
              2             42
              3             57
              4             74
              5             98
              6            123
              7            149
              8            175
              9            202
             10            353
```

Height-Balanced Histograms

Frequency Histograms

- In a frequency histogram, each value of the column corresponds to a single bucket of the histogram
- Each bucket contains the number of occurrences of that single value.
- Frequency histograms are automatically created instead of height-balanced histograms when the number of distinct values is less than or equal to the number of histogram buckets specified
- Frequency histograms can be viewed using the **`*USER_HISTOGRAMS`** tables

Frequency Histograms

```
SELECT column_name, num_distinct, num_buckets, histogram
  FROM USER_TAB_COL_STATISTICS
 WHERE table_name = 'INVENTORIES' AND column_name = 'WAREHOUSE_ID';

COLUMN_NAME                    NUM_DISTINCT NUM_BUCKETS HISTOGRAM
------------------------------ ------------ ----------- -----------
WAREHOUSE_ID                              9           9 FREQUENCY

SELECT endpoint_number, endpoint_value
  FROM USER_HISTOGRAMS
 WHERE table_name = 'INVENTORIES' and column_name = 'WAREHOUSE_ID'
  ORDER BY endpoint_number;

ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
             36              1
            213              2
            261              3
            370              4
            484              5
            692              6
            798              7
            984              8
           1112              9
```


Choosing an Optimizer Goal

- Optimization for best **throughput**
  - Optimizer chooses the least amount of resources necessary to process all rows accessed by the statement
  - Throughput is more important in batch applications (e.g., Oracle Reports applications) because the user is only concerned with the time necessary for the application to complete
- Optimization for best **response time**
  - Optimizer uses the least amount of resources necessary to process the first row accessed by a SQL statement.
  - Response time is important in interactive applications (e.g., SQL*Plus queries)

OPTIMIZER_MODE Parameter Values

Value	Description
`ALL_ROWS`	The optimizer uses a cost-based approach for all SQL statements in the session. It optimizes with a goal of best **throughput** (minimum resource use to complete the entire statement). Default.
`FIRST_ROWS_n`	The optimizer uses a cost-based approach, 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

- The following SQL statement changes the goal of the query optimizer for the current session to **best response time**

```
ALTER SESSION SET OPTIMIZER_MODE = FIRST_ROWS_1;
```


Looking Beyond Execution Plans

- The execution plan operation alone cannot differentiate between well-tuned statements and those that perform poorly
- For example, an **`EXPLAIN PLAN`** output that shows that a statement uses an index does not necessarily mean that the statement runs efficiently. In this case, you should examine
  - the columns of the index being used
  - their selectivity (fraction of table being accessed)
- It is best to use **`EXPLAIN PLAN`** to determine an access plan, and then later prove that it is the optimal plan through testing.

Tania Cerquitelli
Politecnico di Torino