"If I had only one hour to save the world, I would spend fifty-five minutes defining the problem, and only five minutes finding the solution."

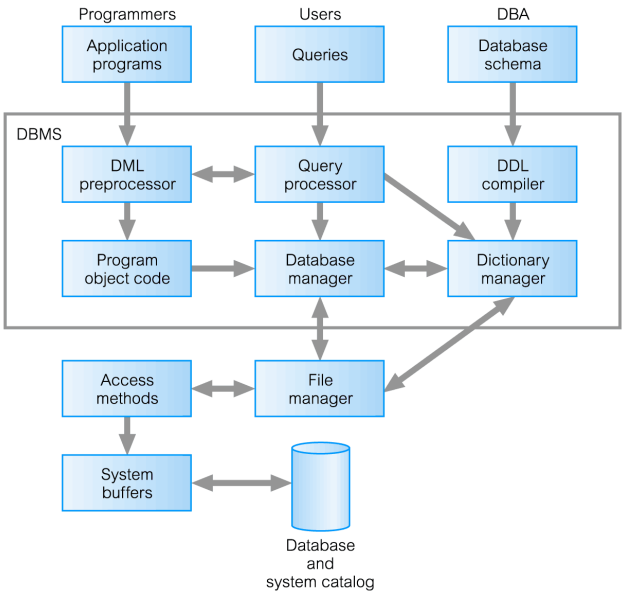
— Albert Einstein

Query Optimisation
Part 1

Brendan Tierney

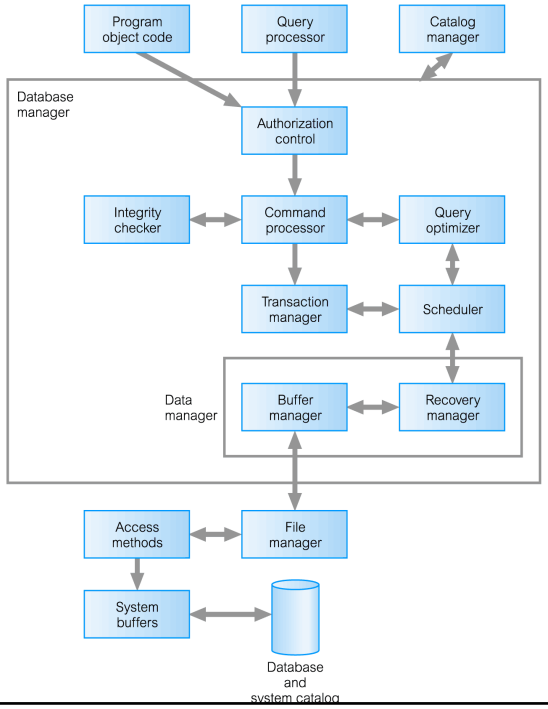
1

Components of a DMBS



2

2



3

3

Introduction

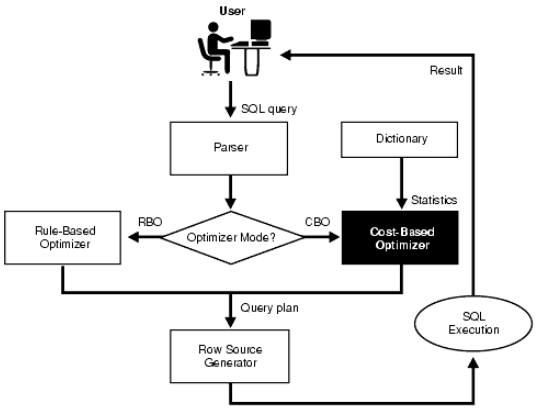
- In network and hierarchical DBMSs, low-level procedural query language is generally embedded in high-level programming language.
  - This is still a very common problem today.
  - Developers do not use the power of the database. Instead they code it themselves
- Programmer's responsibility to select most appropriate execution strategy.
- With declarative languages such as SQL, user specifies what data is required rather than how it is to be retrieved.
- Relieves user of knowing what constitutes good execution strategy.

4

4

Introduction

- Also gives DBMS more control over system performance.
- Two main techniques for query optimization:
  - heuristic rules that order operations in a query;
  - comparing different strategies based on relative costs, and selecting one that minimizes resource usage.
- Disk access tends to be dominant cost in query processing for centralized DBMS.



5

5

Introduction

- "I've been waiting 30 minutes for a response to my query – what's going on ?"
- "My system administrator says our system is I/O bound – what can I do ?"
- " Our application ran fine in testing, but the response is terrible now that we are in production – help!"
- "Our backups take too long – how can we speed them up ?"
- "Our database is fully normalised, but response time is bad – why?"
- …….

6

6

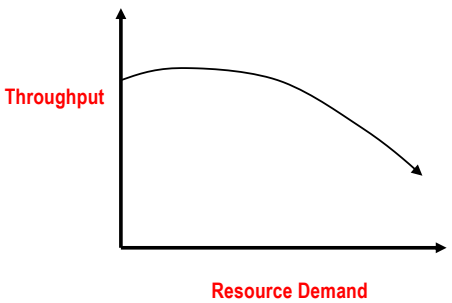
Database Tuning

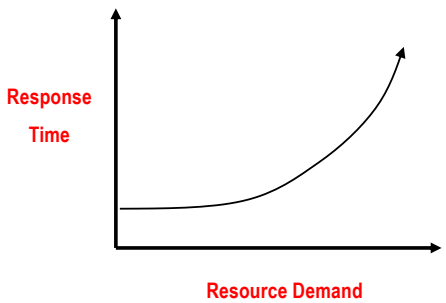
- Effects of Response Time on the Performance
  - The response time of a query or an application is equal to the sum of the service time and the wait time of an application.
    - Service time is the duration that a query or an application requires for its completion.
    - Wait time is the duration for which a transaction must wait before it is allocated the resource to start execution.
  - To optimise the performance of database query or an application, you need to reduce the service or wait time of a query or an application.
    - As a result, the response time of the query or application decreases, which leads to optimised performance of the query or application.
- Effects of the Throughput on the Performance
  - Throughput is the number of transactions that the database server completes in a specified period of time.
  - To optimise the performance of the database server, you must increase the throughput.
    - To do this, you reduce the service time or the overall response time

7

Database Tuning

- Effects of Critical Resources on the Performance
  - The performance of a database also depends on the resources, such as the CPU, memory, and network bandwidth.
  - When the number of transactions increases, the contention for the resources also increases, which results in a reduced throughput and an increased response time



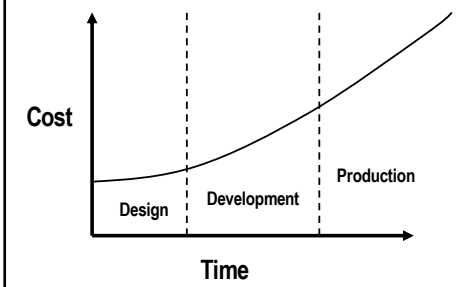


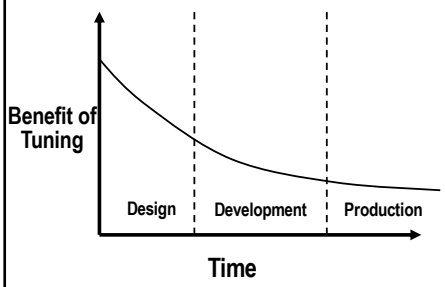
8

Roles of Tuning

- System Planner
  - This person oversees the entire tuning process from planning to production and ongoing monitoring
  - Must identify the type of business rules and procedures that affect the performance of an application or database.
- Designer/Analyst
  - Designs the architecture and data model, as well as the overall application, for performance
  - Must identify the performance related problems of an application and provide a solution for them.
  - An application designer is also responsible for documenting the system design to help other application/database end users to understand the data flow of the application.
- Programmer
  - Develops the application and tunes all program code for performance
  - Tuning is often ignored
  - Must identify the implementation strategies that determine the SQL statements to optimise the database server performance.
- DBA
  - Involved in the design of the database and takes responsibility for tuning the database for performance as well as ongoing database tuning once the application is in production
  - Must monitor the performance of an Oracle database to identify the use of resources and performance problems in the database
- Systems Administrator
  - Responsible for the OS maintenance and monitors systems resources needed by the databases and applications

Cost of Tuning





- The cost of tuning increases during the lifecycle of an application
- ***Proactive Tuning*** normally occurs on a regularly scheduled interval.
  - you need to examine several performance statistics, such as memory and CPU usage, to identify whether or not the system behavior and resource usage has modified.
  - Proactive tuning adjusts the database environment just before the performance degradation becomes significant enough to be noticed.
- The effective time for tuning is during the designing phase, where you obtain the maximum benefit for tuning at the lowest administrative cost.
- You should avoid tuning a system when there is no apparent performance degradation because it can lead to a decrease in the performance of the application.
- ***Reactive Tuning*** occurs when you need to immediately rectify a reported database performance problem.
  - You may need to completely redesign the application to optimize its performance.
  - Otherwise, you may only improve the performance marginally if you reallocate the memory and tune the I/O. To use reactive tuning, you start from the lowest level of the production system and detect and fix any issues.
- The performance of well designed systems can decline with constant use. As a result, you need to conduct tuning on a regular basis for appropriate system maintenance.

Process of Tuning

- To tune a database, you should follow a systematic stepwise approach.
  - The steps are prioritised in the order of diminishing returns.
  - You first need to perform the steps with the largest effect on performance and then the steps with the least effect on the performance.

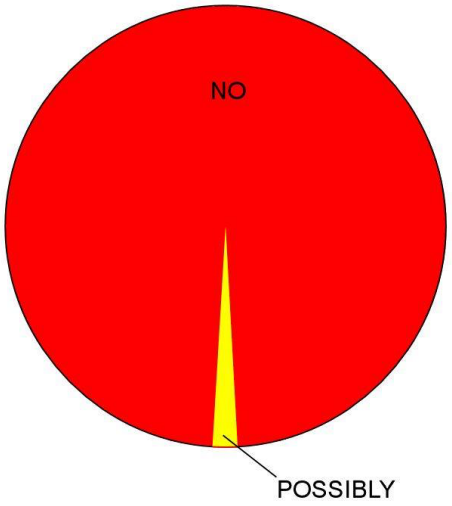
1. Tune the business rules of the application.
2. Tune the data design of the application.
3. Tune the application design to secure optimum performance.
4. Tune the logical structure of the database to secure optimum performance.
5. Tune the database operations to increase the performance of the application.
6. Tune the access paths for efficient data access by the database server.
7. Tune the allocation of memory structures to improve the performance of the application and the database server.
8. Tune the disk I/O to enhance the performance of the application and the database server.
9. Tune the resource contention by multiple database end users.
10. Tune the operating system on which database is installed.

90% of all performance issues

11

11

IS IT A PROBLEM WITH THE DATABASE?



Only if the Database had been designed properly.

Good fully completed ER.

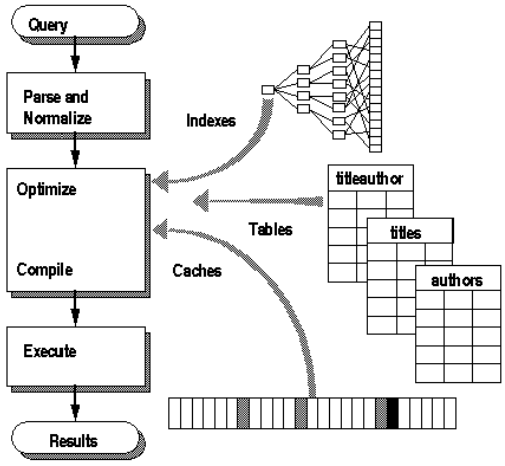
12

12

Query Processing

Activities involved in retrieving data from the database.

- Aims of QP:
  - transform query written in high-level language (e.g. SQL), into correct and efficient execution strategy expressed in low-level language (implementing RA);
  - execute strategy to retrieve required data



13

13

Query Optimisation

Activity of choosing an efficient execution strategy for processing query.

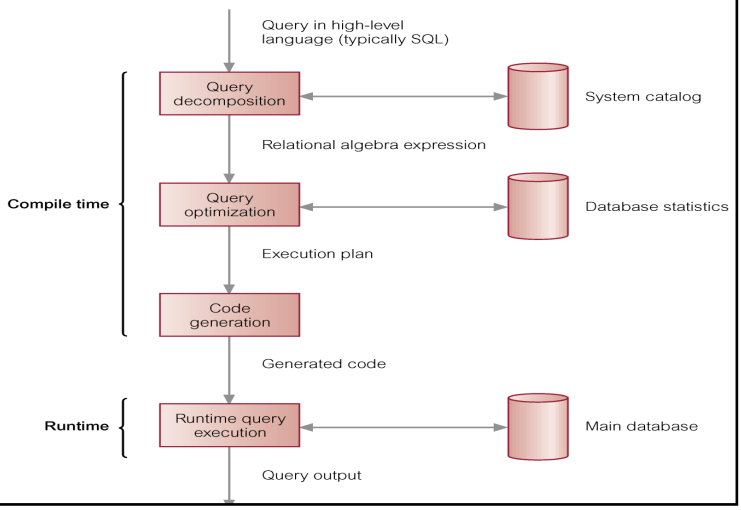
- As there are many equivalent transformations of same high-level query, aim of QO is to choose one that minimizes resource usage.
- Generally, reduce total execution time of query.
- May also reduce response time of query.
- Problem computationally intractable with large number of relations, so strategy adopted is reduced to finding near optimum solution.

14

14

Query Processing Phases

- QP has four main phases:
  - decomposition (consisting of parsing and validation);
  - optimization;
  - code generation;
  - execution.



15

Query Optimisation Types

- **Heuristic Rules** – uses heuristic rules to order the operations of the query execution plan. Generally works, but not in every case.

- **Cost based optimisation** – calculates the cost of different execution strategies and chooses the execution path with the lowest cost estimate.

16

16

Heuristic Based Optimisation

- Query parser generates an initial internal representation which is then optimised using the heuristic rules
  - Use the meta-data / data dictionary
  - A good database design with advanced planning can help
- Internal representation is then used to generate a query execution plan to execute groups of operations based on the access paths available on the tables involved in the query.
    - Multiple query trees generated
    - Differently structured queries can have the same query tree
    - the query trees are used to optimise the processing of the query.

17

17

Heuristic Optimisation Rules

- Break the SELECT statement with conjunctive conditions into a cascade of SELECT operations. This allows the freedom to move these separate SELECT operations around different parts of the query tree
- Move the SELECT operations as far down the query tree as is permitted by the attributes involved in the select condition
- Re-arrange the tree nodes so that:
    - The SELECT operations which are the most restrictive are positioned so that they are executed first.
    - Ensure that the ordering of nodes does not cause a Cartesian Product operation.
- Combine a Cartesian Product operation with a subsequent SELECT operation into a JOIN operation, only if the condition represents a join condition. i.e. remove the Cartesian product.
- Only those attributes needed in the query result and in subsequent operations in the query tree should be kept after each operation.

18

18

Example

```
SELECT   lname              -- employee
FROM     EMPLOYEE, WORKS_ON,  PROJECT
WHERE    pname = 'Aquarius'      -- project
AND      pnumber = pno           -- project.pnumber = works_on.pno
AND      essn = ssn         -- works_on.essn = employee.ssn
AND      bdate > '31/12/1957'    -- employee
```

Draw an ER diagram that represents the 3 tables

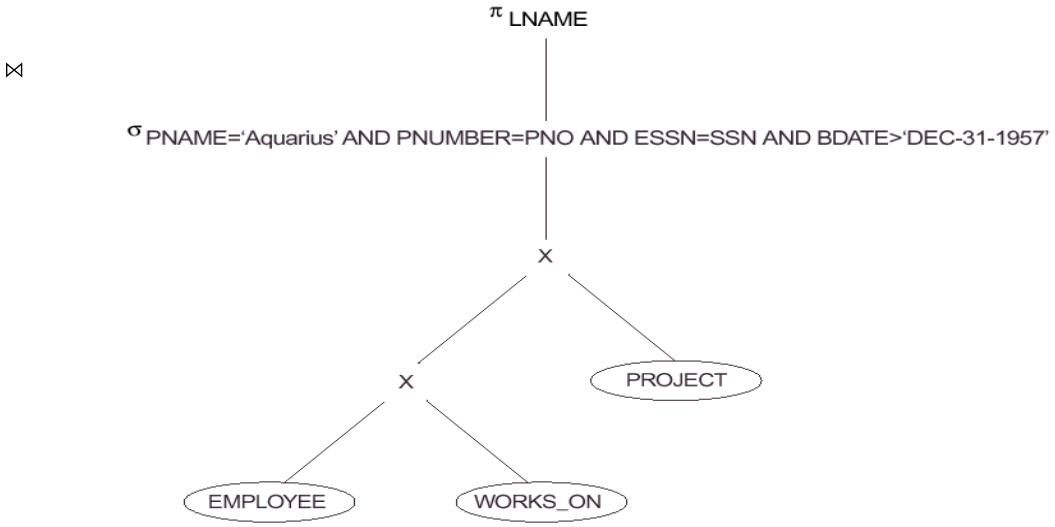
19

19

Example

```
SELECT   lname
FROM     EMPLOYEE, WORKS_ON,  PROJECT
WHERE    pname = 'Aquarius'
AND      pnumber = pno
AND      essn = ssn
AND      bdate > '31/12/1957'
```

- Draw the query tree based on heuristics



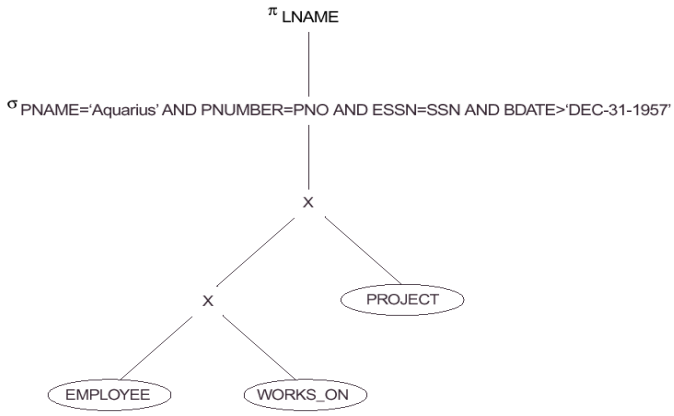
20

20

Example

```
SELECT   lname
FROM     EMPLOYEE, WORKS_ON, PROJECT
WHERE    pname = 'Aquarius'
AND      pnumber = pno
AND      essn = ssn
AND      bdate > '31/12/1957'
```

- This is a very basic structure of the query tree and produces a Cartesian product.
- But the query only needs 1 record from the Project table and only those records from Employee where BDATE > '31/12/1957'.
- Based on this knowledge we can rearrange the query tree to incorporate this



21

21

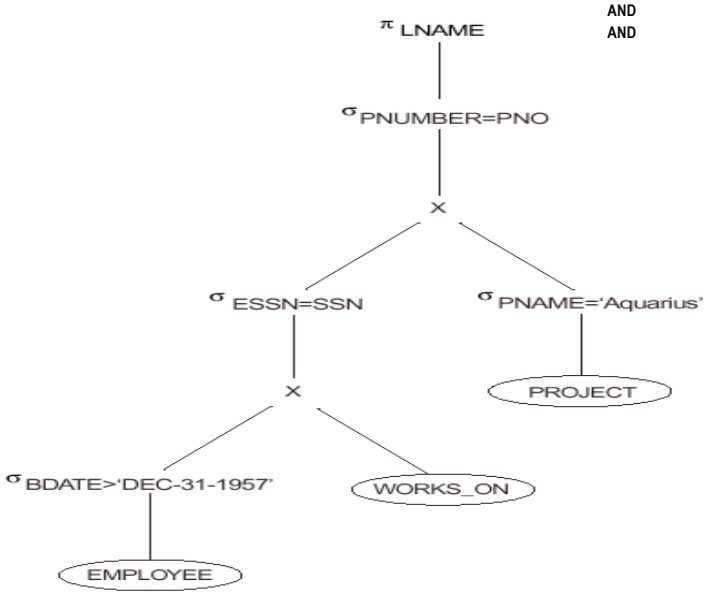
σ Sigma – Select Operator

π Pie – Project – attribute list

⋈ Join

Example

```
SELECT   lname
FROM     EMPLOYEE, WORKS_ON, PROJECT
WHERE    pname = 'Aquarius'
AND      pnumber = pno
AND      essn = ssn
AND      bdate > '31/12/1957'
```

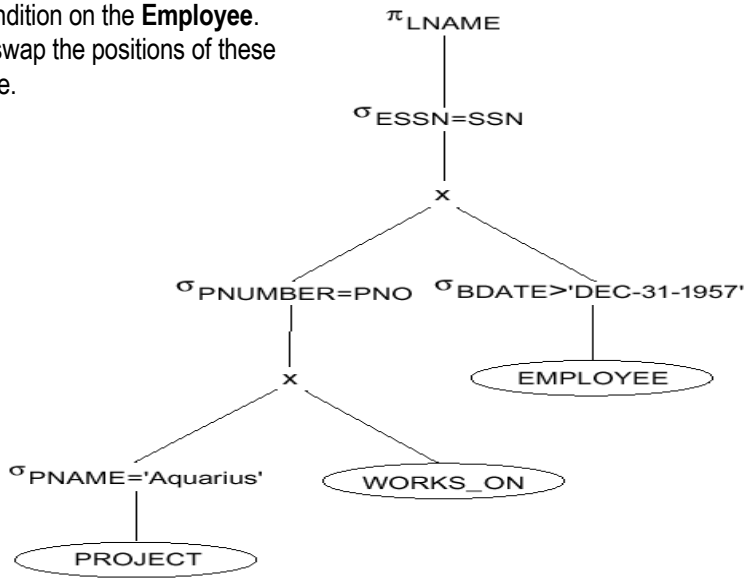


22

22

Example

- The condition on the **Project** table is more restrictive than the condition on the **Employee**. Therefore we should swap the positions of these nodes in the query tree.

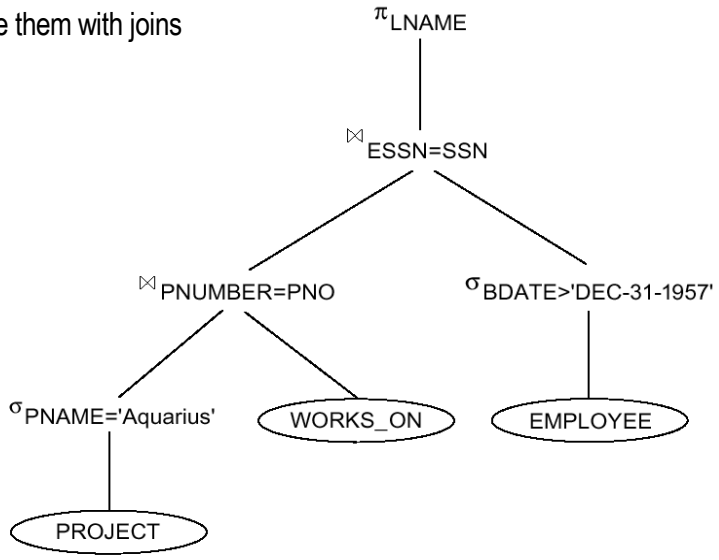


23

23

Example

- Now we need to remove the Cartesian products and replace them with joins

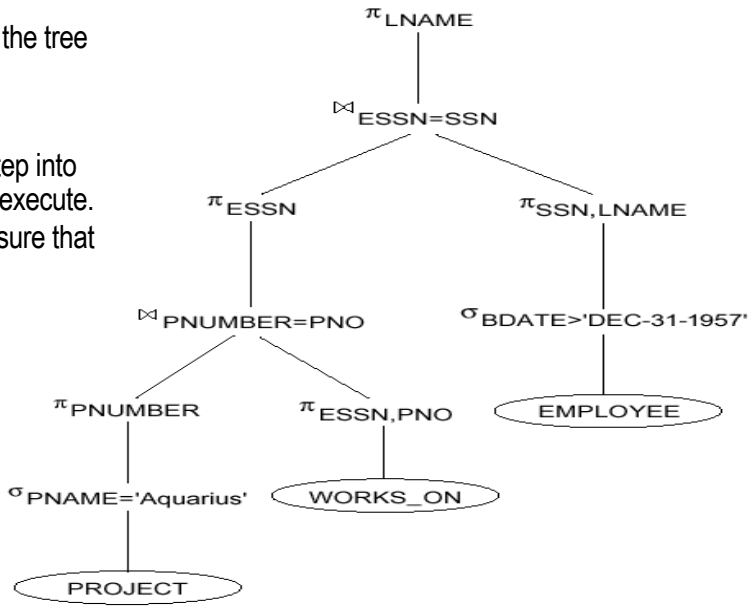


24

24

Example

- Finally we need to push the projects down the tree to the relevant nodes.
- A query tree can be transformed step by step into another query tree that is more efficient to execute. However, the query optimiser must make sure that the transformation steps always lead to an equivalent query tree.



25

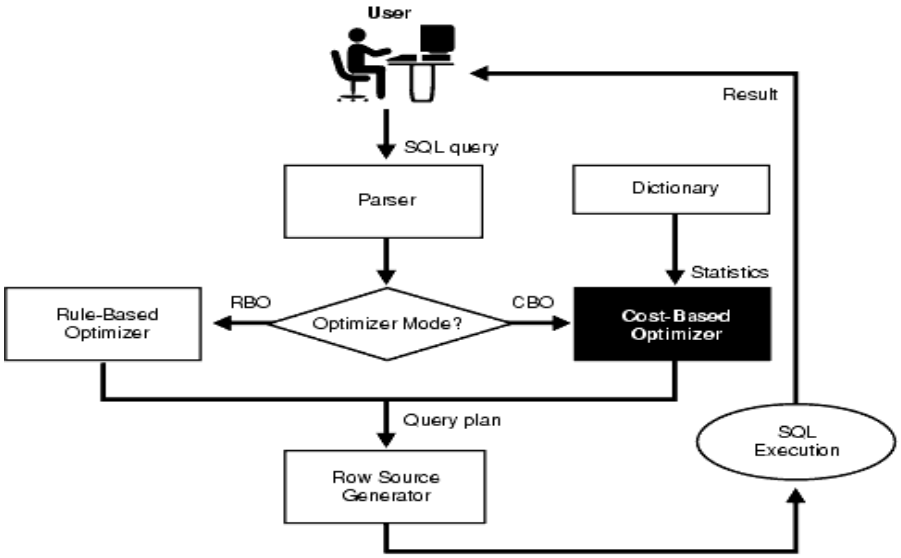
Summary Guidelines

- Basic heuristic optimisation guidelines
  - Reduce the size of the intermediate results
    - Perform SELECT operations to reduce the number of tupes and PROJECT operations to reduce the number of attributes
    - Move SELECT & PROJECT operations as far down the tree as possible
  - The operations that are the most restrictive should be performed first
    - Move to bottom of tree
    - Should be executed before other similar operations
    - May require reordering of leaf nodes
  - Avoid Cartesian products

26

26

Query Optimisation in Oracle



27

27

Query Optimization in Oracle

- Oracle supports two approaches to query optimization: rule-based and cost-based.

Rule-based

- 15 rules, ranked in order of efficiency. Particular access path for a table only chosen if statement contains a predicate or other construct that makes that access path available.
- Score assigned to each execution strategy using these rankings and strategy with best (lowest) score selected.
- When 2 strategies produce same score, tie-break resolved by making decision based on order in which tables occur in the SQL statement.

28

28

QO in Oracle – Rule-based: Example

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

Indexes exist on PropertyNo and City

- Single-column access path using index on city from WHERE condition (city = 'London'). **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, column does not appear in WHERE clause and so is not considered by optimizer.
- Based on these paths, rule-based optimizer will choose to use index based on city column.

QO in Oracle – Viewing Execution Plan

```
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.
```

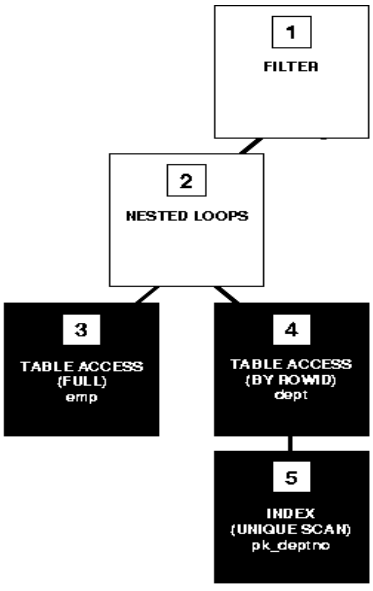
Query Optimisation in Oracle

```
SET AUTOTRACE ON;

SELECT  empno, job, sal, dname
FROM    EMP, DEPT
WHERE   EMP.deptno = DEPT.deptno;

Execution Plan
----------------------------------------------------------------------------
  0         SELECT STATEMENT Optimizer=CHOOSE
  1    0        NESTED LOOPS
  2    1            TABLE ACCESS (FULL) OF 'EMP'
  3    1            TABLE ACCESS (BY INDEX ROWID) OF 'DEPT'
  4    3                INDEX (UNIQUE SCAN) OF 'PK_DEPT' (UNIQUE)
```

31

31

Query Optimisation in Oracle



32

32

How does the database retrieve the data

- The important part of ROWID is the Data Block
  - The data retrieves records in units of blocks
  - How does the data process the following query

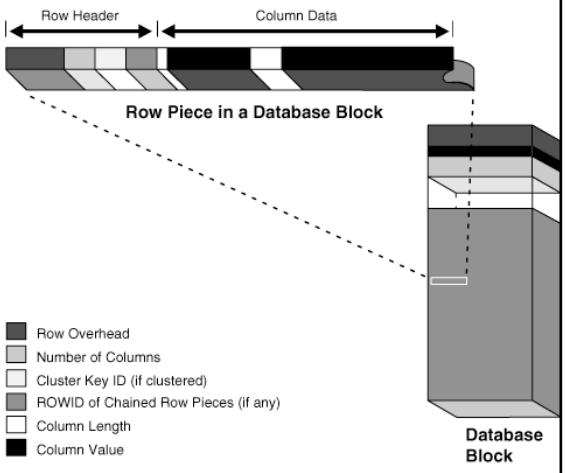
    SELECT * FROM Customers

  - What about

    SELECT * FROM Customers

    WHERE County = 'DUBLIN'

  - Which is the most efficient / How many blocks does the queries return?



33

33

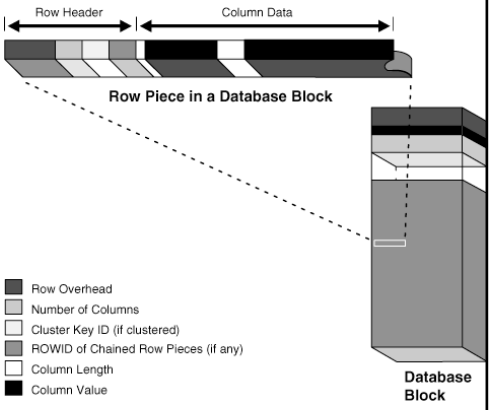
How does the database retrieve the data

- Need to be able to locate the data
  - Need an address (location on disk)
  - Reach record has a ROWID

    SELECT ROWID, <*column name list*> FROM <*table name*>

```
ROWID              LAST_NAME
------------------ ----------
AAAAaoAATAAABrXAAA BORTINS
AAAAaoAATAAABrXAAE RUGGLES
AAAAaoAATAAABrXAAG CHEN
AAAAaoAATAAABrXAAN BLUMBERG
```

  - A ROWID has a four-piece format, OOOOOOFFFBBBBBBRRR:
    - **OOOOOO**: The **data object number** that identifies the database segment (AAAAao in the example). Schema objects in the same segment, such as a cluster of tables, have the same data object number.
    - **FFF**: The tablespace-relative **datafile number** of the datafile that contains the row (file AAT in the example).
    - **BBBBBB**: The **data block** that contains the row (block AAABrX in the example). Block numbers are relative to their datafile, **not** tablespace. Therefore, two rows with identical block numbers could reside in two different datafiles of the same tablespace.
    - **RRR**: The **row** in the block.



34

34

Supporting Database Structures

- Need to avoid full table scans
  - Are there situations when you need to do full scans
- Methods to improve performance
  - Clustering
  - Indexing
    - B-tree
    - Function
    - Index Organised
    - Partitioning
    - Bit Mapped
- Are there situations when you need to do full scans
  - Situations when using an index is not efficient
  - If so you will need to by-pass indexes

Uses the ROWID to build various indexing structures

35

35

Indexing

- SELECT name
- FROM student
- WHERE student_id = 's123456789'

- Need to look for an load every data block into memory
  - Then check to see if each record matches the WHERE clause
- What if the table has 10, 100, 10000, 1000000 records
- Need some way to easily identify the location of records for STUDENT_ID => ROWIDs needed
  - This mapping is called an Index

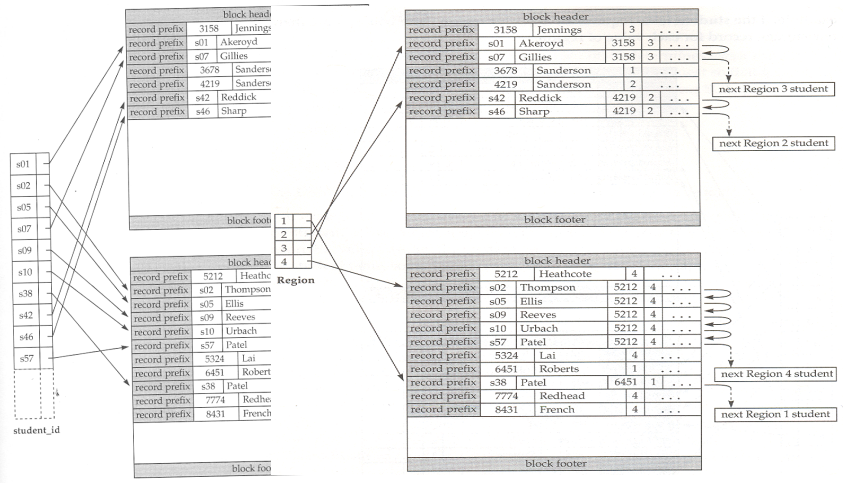
    CREATE INDEX <*Index Name*> on <*Table Name*> (<Attribute List>)
  - Works in the same was as an index to a book
- They also support referential integrity
  - When inserting a row into a table only need check the index on the table (PK Index)
  - To ensure FK to PK integrity
  - Much quicker than checking the table

36

36

Indexing – How are they created

- Index consists of Attribute Value and ROWID



Sequential Indexing

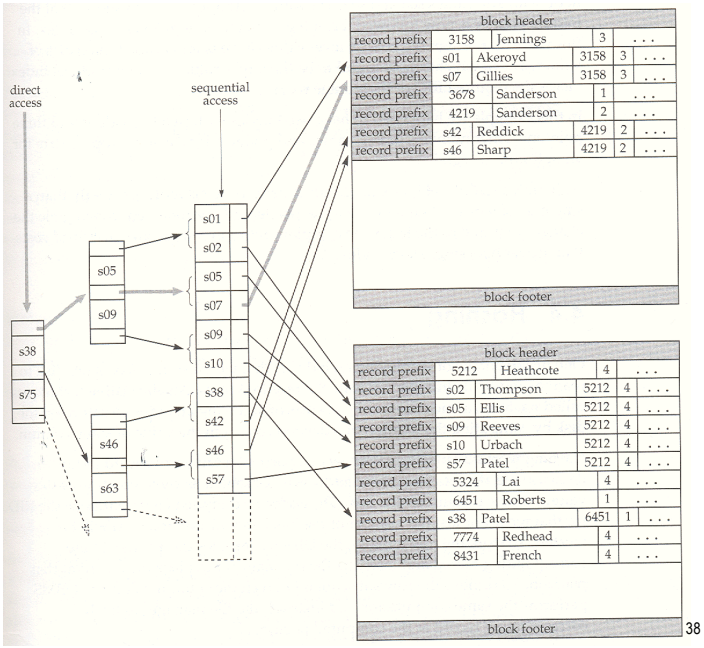
37

37

Indexes – How are they searched

B+ Tree Index

Facilitates sequential and direct access to individual records



- What if the data block already exists in the database memory/buffer
- Are indexes always required ?
  - What situations should you not have an index

38

38

Party Preparation: An Index Maintenance Story

https://www.youtube.com/watch?v=f3U9F_wbo1I

4 minutes

39

Finding All the Red Sweets: A Story of Indexes and Full Table Scans

The Magic of SQL

https://www.youtube.com/watch?v=RIgb7LwOiHk

8min 42 sec

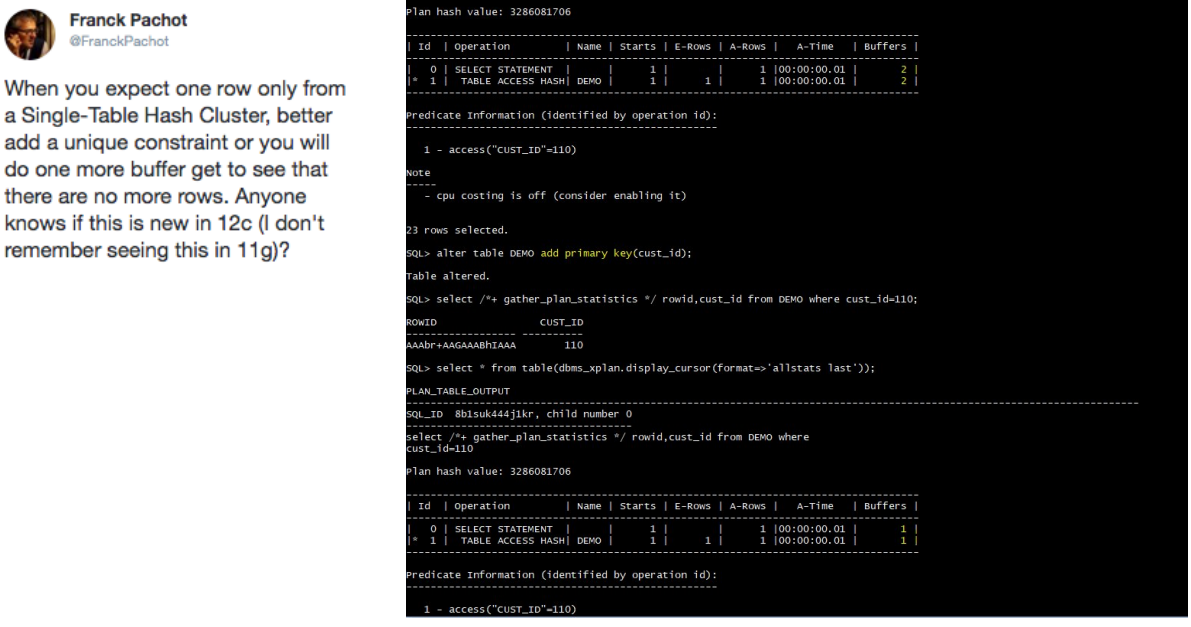
40

https://www.youtube.com/watch?v=eEhvQ-7gaFI

9min 24sec

41

41



42

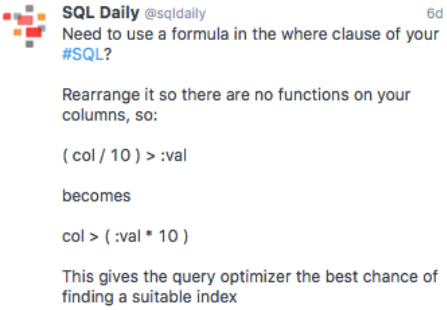
Indexing – Function Based Indexes

```
SELECT *
FROM   employees
WHERE  first_name = 'Brendan'
```

- But you could have Brendan, brendan, BRENDAN, etc
- Index is on FIRST_NAME
- Index will not be used

- Need to use function UPPER(first_name)
- Creating a function based index

```
SELECT * FROM employees
WHERE UPPER(first_name) = 'RICHARD';

SELECT * FROM EMPLOYEES
WHERE SALARY*1.2 >= 10,000;

CREATE INDEX EMP_SAL_INX ON EMPLOYEE (SALARY);
```



- Can only be used in Cost-Based Optimisation (not in Rule)
- The function used for building the index can be an
  - arithmetic expression or
  - an expression that contains a PL/SQL function, package function, C callout, or SQL function.
  - The expression cannot contain any aggregate functions,
    - Why ?
  - it must be executable by the index schema owner

43

Indexing – Function Based Indexes

```
CREATE INDEX emp_lastname ON EMPLOYEES (UPPER(LAST_NAME));
```

Use index expression in query:

```
SELECT first_name, last_name
FROM EMPLOYEES
WHERE UPPER(LAST_NAME) LIKE 'J%S_N';
Result:

FIRST_NAME           LAST_NAME
-------------------- -------------------------
Charles              Johnson

1 row selected.
```

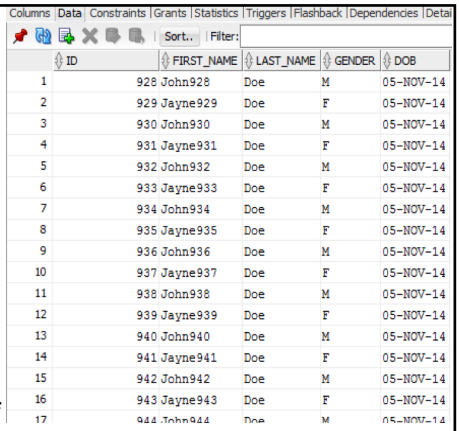
The function used in the Query is the same as the Index.

So the functional based index will be used

44

Setup a Demo Table and some Test data

```
CREATE TABLE user_data (
 id           NUMBER(10)     NOT NULL,
 first_name   VARCHAR2(40)   NOT NULL,
 last_name    VARCHAR2(40)   NOT NULL,
 gender       VARCHAR2(1),
 dob          DATE
);

BEGIN
  FOR cur_rec IN 1 .. 2000 LOOP
    IF MOD(cur_rec, 2) = 0 THEN
      INSERT INTO user_data
      VALUES (cur_rec, 'John' || cur_rec, 'Doe', 'M', SYSDATE);
    ELSE
      INSERT INTO user_data
      VALUES (cur_rec, 'Jayne' || cur_rec, 'Doe', 'F', SYSDATE);
    END IF;
    COMMIT;
  END LOOP;
END;
/

EXEC DBMS_STATS.gather_table_stats(USER, 'user_data', cascade => TRUE);
```



	ID	FIRST_NAME	LAST_NAME	GENDER	DOB
1	928	John928	Doe	M	05-NOV-14
2	929	Jayne929	Doe	F	05-NOV-14
3	930	John930	Doe	M	05-NOV-14
4	931	Jayne931	Doe	F	05-NOV-14
5	932	John932	Doe	M	05-NOV-14
6	933	Jayne933	Doe	F	05-NOV-14
7	934	John934	Doe	M	05-NOV-14
8	935	Jayne935	Doe	F	05-NOV-14
9	936	John936	Doe	M	05-NOV-14
10	937	Jayne937	Doe	F	05-NOV-14
11	938	John938	Doe	M	05-NOV-14
12	939	Jayne939	Doe	F	05-NOV-14
13	940	John940	Doe	M	05-NOV-14
14	941	Jayne941	Doe	F	05-NOV-14
15	942	John942	Doe	M	05-NOV-14
16	943	Jayne943	Doe	F	05-NOV-14
17	944	John944	Doe	M	05-NOV-14

Example by Tim Hall, http://oracle-base.com/articles/8i/function-based-indexes.php

45

At this point the table is not indexed so we would expect a full table scan for any query.

```
SET AUTOTRACE ON
SELECT *
FROM   user_data
WHERE  UPPER(first_name) = 'JOHN2';

Execution Plan
----------------------------------------------------------
Plan hash value: 2489064024

--------------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |    20 |   540 |     5   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| USER_DATA |    20 |   540 |     5   (0)| 00:00:01 |
--------------------------------------------------------------------------------
```

Example by Tim Hall, http://oracle-base.com/articles/8i/function-based-indexes.php

46

Build Regular Index

If we now create a regular index on the FIRST_NAME column we see that the index is not used.

```
CREATE INDEX first_name_idx ON user_data (first_name);

EXEC DBMS_STATS.gather_table_stats(USER, 'user_data', cascade => TRUE);

SET AUTOTRACE ON
SELECT *
FROM   user_data
WHERE  UPPER(first_name) = 'JOHN2';

Execution Plan
----------------------------------------------------------
Plan hash value: 2489064024

--------------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |    20 |   540 |     5   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| USER_DATA |    20 |   540 |     5   (0)| 00:00:01 |
--------------------------------------------------------------------------------
```

Example by Tim Hall, http://oracle-base.com/articles/8i/function-based-indexes.php

47

Build Function Based Index

If we now replace the regular index with a function based index on the FIRST_NAME column we see that the index is used.

```
DROP INDEX first_name_idx;
CREATE INDEX first_name_idx ON user_data (UPPER(first_name));

EXEC DBMS_STATS.gather_table_stats(USER, 'user_data', cascade => TRUE);

SET AUTOTRACE ON
SELECT *
FROM   user_data
WHERE  UPPER(first_name) = 'JOHN2';

Execution Plan
----------------------------------------------------------
Plan hash value: 1309354431

-----------------------------------------------------------------------------------------------
| Id  | Operation                   | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                |     1 |    36 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| USER_DATA      |     1 |    36 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | FIRST_NAME_IDX |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------
```

Example by Tim Hall, http://oracle-base.com/articles/8i/function-based-indexes.php

48

Concatenated Columns

This method works for concatenated indexes also.

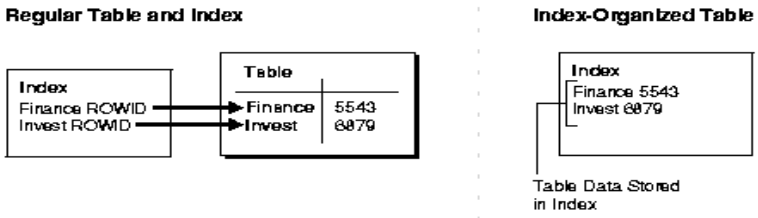
```
DROP INDEX first_name_idx;
CREATE INDEX first_name_idx ON user_data (gender, UPPER(first_name), dob);
EXEC DBMS_STATS.gather_table_stats(USER, 'user_data', cascade => TRUE);

SET AUTOTRACE ON
SELECT *
FROM   user_data
WHERE  gender = 'M'
AND    UPPER(first_name) = 'JOHN2';

Execution Plan
----------------------------------------------------------
Plan hash value: 1309354431

-------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                |     1 |    36 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| USER_DATA      |     1 |    36 |     3   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | FIRST_NAME_IDX |     1 |       |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------
```

Example by Tim Hall, http://oracle-base.com/articles/8i/function-based-indexes.php

49

Indexing – Index Organised Tables

- Storage organization is a variant of a primary B-tree.
  - Unlike an ordinary table whose data is stored as an unordered collection, data for an index-organized table is stored in a B-tree index structure in a primary key sorted manner.
  - Besides storing the primary key column values of an index-organized table row, each index entry in the B-tree stores the non-key column values as well.



  - An index-organized table configuration is similar to an ordinary table and an index on one or more of the table columns
  - but instead of maintaining two separate storage structures
    - one for the table and one for the B-tree index
  - The database system maintains only a single B-tree index.
  - The ROWID is not stored in the index entry, the non-key column values are stored.
  - An Index Organised Table consists of an index that contains
    - <primary_key_value, **non_primary_key_column_values**>.

50

50

Indexing – Index Organised Tables

```
SQL> CREATE TABLE my_iot (id       INTEGER PRIMARY KEY,
                          value    VARCHAR2(50),
                          comments varchar2(1000))
     ORGANIZATION INDEX;
```

Ordinary Table	Index-Organised Table
ROWID uniquely identifies a row.	Primary Key uniquely identifies rows
Access based on ROWID	Access based on logical ROWID
Sequential scan returns all rows	Full index scan returns all rows

- Only suited for static type of data, where there is very little if any Updates, etc

51

51

Indexes – What Columns should you index?

- "I don't know what the users will query on so let's index everything"
- But you do know how they will search!! How?
- Scenarios when you don't know?
- Indexing is part of the (physical) Database Design process
  - Will also be part of the Testing phase
  - Every day task for the DBA/DB Designer.

52

Indexes - Basics

- Identifying candidate key(s) for an entity and then selecting one to be the primary key.
- Candidate keys can never be null.
- Select Primary Key
- Remaining candidate keys are called alternate keys.
- Document candidate, primary, and alternate keys

- Choosing the Primary Key
  - Select the candidate key
    - the minimal set of attributes;
    - that is less likely to have its values changed;
    - that is less likely to lose uniqueness in the future;
    - with fewest characters (for those with textual attribute(s));
    - with the smallest maximum value (for numerical attributes);
    - that is easiest to use from the users' point of view.
- Candidate Keys
  - An attribute or combination of attributes (***K***), which preserve *uniqueness* and *minimality*.
    - Uniqueness : No 2 tuples of the relation can have the same values for ***K***
    - Minimality : If a combination of attributes is used, then no attribute can be discarded from ***K*** without destroying the uniqueness.
- Primary Keys
  - Always have one
  - The PK of a relation is one particular key chosen from the list of candidate keys.
    - Typically only have 1. If more than 1 then you need to select which one to use
- Alternate Keys
  - Candidate keys minus Primary key
- Foreign Keys
  - Used to link to a related table
  - A FK is always a PK in another relation

53

53

Indexes - Basics

- Have to balance overhead in maintenance and use of secondary indexes against performance improvement gained when retrieving data.
- This includes:
  - adding an index record to every secondary index whenever record is inserted;
  - updating a secondary index when corresponding record is updated;
  - increase in disk space needed to store the secondary index;
  - possible performance degradation during query optimization to consider all secondary indexes.

Be aware of Index overload

- General guidelines
  - (1) Do not index small tables.
  - (2) Index PK & FKs of a table
  - (3) Add secondary index to any column that is heavily used as a secondary key.
  - (4) Add secondary index on columns that are involved in: selection or join criteria; ORDER BY; GROUP BY; and other operations involving sorting (such as UNION or DISTINCT).

54

54

Query Optimisation in Oracle

- Query Hints
  - the database developer/administrator (DBA) has a detailed understanding of the data and its structure
  - they might be able to give hints to the optimiser to perform the query in a certain way, which is different to the way the optimiser would choose.
  - Hints can include:
    - What optimisation approach to use
    - What access part to use
    - What indexes to use
    - The join order

- `SELECT /*+ FULL(x) */ FROM tab1 x WHERE col1 = 10;`
- `SELECT /*+ INDEX(x emp_idx1) */ ... FROM scott.emp x…`
- `SELECT /*+ NO_INDEX(x emp_idx1) */ ... FROM scott.emp x...`

55

55

56

Exercises

- For a table with no index give a description, as a list of steps, of how a record is retrieved from the database.
- Assume that an index exists for the table above. Give description, as a list of steps, of how the required data would be retrieved.
- For a very small table (assume less than one block) is it more or less efficient to use an indexed or non-indexed storage for data access. Explain your answer.
- Discuss the reasons you would use the different types of indexing discuss. Give diagrams and examples.
- Discuss how the logical structure of the ROWID differs to the physical address of the record. In particular, discuss how the logical structure facilitates the relocation of data files.

- For a table with no index give a description, as a list of steps, of how a record is retrieved from the database

repeat for each block that contains records containing rows of the table
    retrieve the next block from disk and place it in the buffer
        repeat for each record in the newly buffered block
            check to see if it contains the row required – if it does then stop searching
        continue with the next record
continue with the next block

- Assume that an index exists for the table above. Give description, as a list of steps, of how the required data would be retrieved

retrieve the index block from disk and place it in the buffer
search the index block to find the physical location of the required record
retrieve the required block from disk and place it in the buffer
access the required record in the block just retrieved

59

59

- For a very small table (assume less than one block) is it more or less efficient to use an indexed or non-indexed storage for data access. Explain your answer

In this case it is more efficient to use non-indexed storage.

The use of an index requires that at least one index block is retrieved before any blocks containing table rows are retrieved. If the table is smaller than one block then all the rows of that table can be accessed by loading that block immediately. Use of the index adds an additional, in this case unnecessary, block retrieval to the table access.

- If the table had an index how could you make the database not use the index

60

60

- Discuss how the logical structure of the ROWID differs to the physical address of the record. In particular, discuss how the logical structure facilitates the relocation of data files.
    - A ROWID has a four-piece format, OOOOOOFFFBBBBBBRRR:
        - **OOOOOO**: The **data object number** that identifies the database segment (AAAAao in the example). Schema objects in the same segment, such as a cluster of tables, have the same data object number.
        - **FFF**: The tablespace-relative **datafile number** of the datafile that contains the row (file AAT in the example).
        - **BBBBBB**: The **data block** that contains the row (block AAABrX in the example). Block numbers are relative to their datafile, **not** tablespace. Therefore, two rows with identical block numbers could reside in two different datafiles of the same tablespace.
        - **RRR**: The **row** in the block.

61

61