Topic 08: Tuning the physical design

ICT285 Databases

Dr Danny Toohey

About this topic

- In this topic we look more closely at how the data in the database is physically held on secondary storage. One of the responsibilities of the DBA is to determine the optimal file organisations and indexes that are required to achieve acceptable performance for the important transactions. These work together with the inbuilt query optimisation techniques of the DBMS to provide efficient processing.

Topic learning outcomes

After completing this topic you should be able to:

- Describe the activities in physical database design
- Describe some different types of file organisation used in commercial DBMSs, and when each is appropriate
- Determine when secondary indexes are appropriate, and when they are not
- Briefly describe how query optimisation works
- Describe in general terms how the hash, sort-merge and nested loops join strategies work

Resources for this topic

READING

- My Unit Readings: Reading from Connolly & Begg (2004) Database Solutions, Appendix D: File Organisations and Indexes
- Text, Chapter 7: SQL for Database Construction and Application Processing, p351 (333 in 13th ed) CREATE INDEX

Oracle

- Indexes: https://docs.oracle.com/database/121/CNCPT/indexiot.htm#CNCPT721
- Query optimisation: https://docs.oracle.com/database/121/TGSQL/tgsql_optcncpt.htm#TGSQL192

Lab 8 –
Indexes and query optimisation in Oracle

- Indexes are used to provide improved access to database records. Indexes are created automatically on columns defined as primary key, and secondary indexes can also be created by the user on other columns, including foreign keys. In SQL, creating a secondary index is done using the CREATE INDEX statement.
- In this topic's lab we'll look at secondary indexes, and also how to view and interpret Oracle's query execution plans.

Topic outline

1. Tuning the physical design
2. File organisations and indexes
3. Query processing and query optimisation

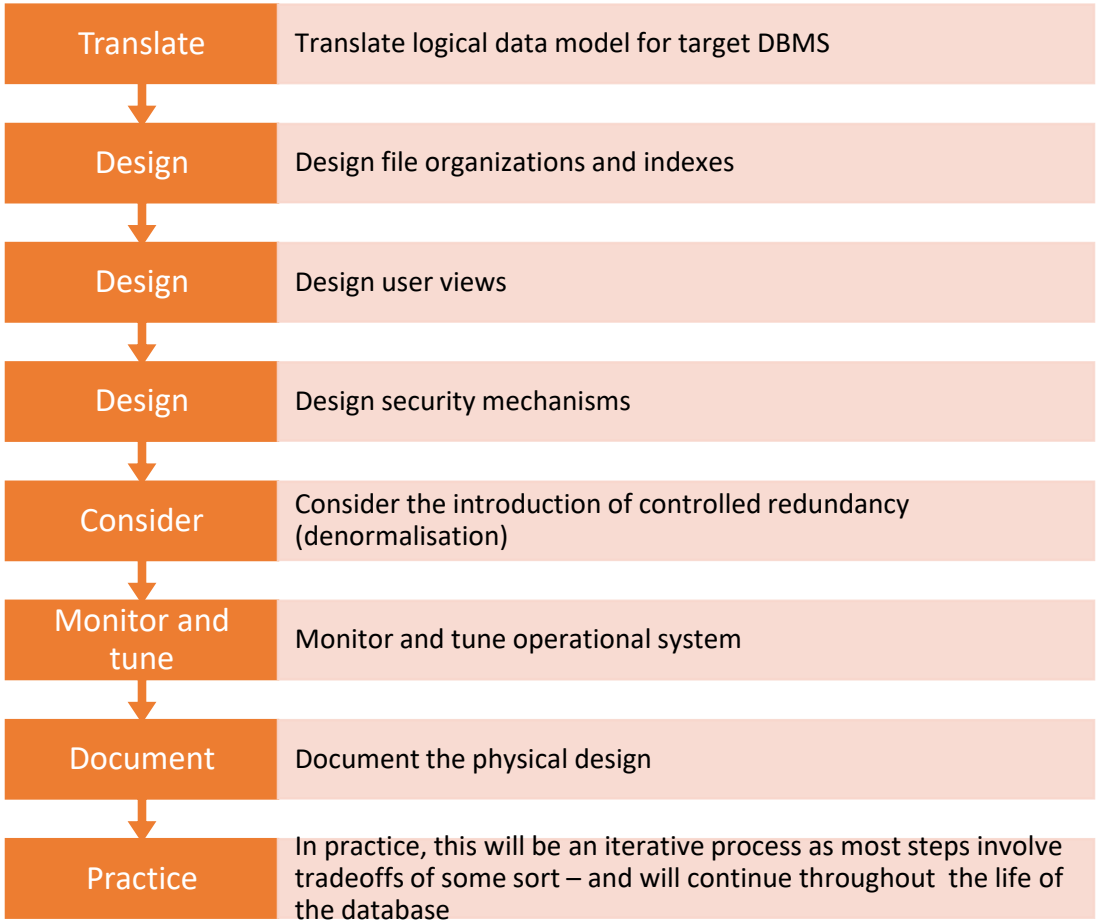
1. Tuning the physical design

Reminder:
Physical database design

- Process of producing a description of the database implementation in secondary storage
- Describes base relations, file organisations, and indexes used to achieve efficient access to data. Also describes any associated integrity constraints and security measures
- Tailored to a specific DBMS system
- The physical design process is where we concentrate on the **efficient implementation of our logical design**
- It is important to note that efficiency for one operation often comes at the expense of another (e.g., retrieval vs update)

A physical database design methodology - steps

- *Steps from Connolly & Begg*



A physical database design methodology - steps

1. Translate logical data model for target DBMS
2. Design file organizations and indexes
3. Design user views
4. Design security mechanisms
5. Consider the introduction of controlled redundancy (denormalisation)
6. Monitor and tune the operational system
7. Document the physical design

- *Steps from Connolly & Begg*

Topic 7

This topic

2. File organisations and indexes

File organisations
Indexes

Choose file organisations and indexes

- This step deals with the method in which the data will be physically stored on secondary storage
- Need to determine how to best organise the data in the files and the indexes that we need to best be able to achieve the performance metric for the important transactions such as:

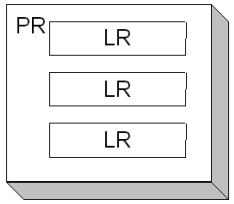
    - Throughput (how many transaction we get through), response time (how long it takes for a transaction to be executed), amount of storage required

- Transaction usage analysis (Topic 7) plus ongoing performance monitoring

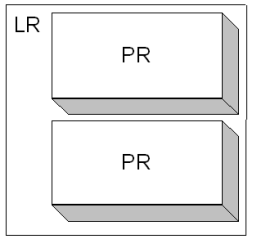
Logical and physical records

The tables we have designed are not *physically* stored as tables

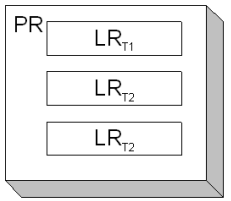
- **Logical** records are how we understand the records are stored
- **Physical** records are the way in which records are actually stored electronically, as a collection of bytes
- Both physical and logical records can be arranged in several ways



Related LRs on same PR



LR split across several PRs



PR containing LRs from different tables

File organisations and indexes

- One vital part of physical database design is selecting the file structures that are going to make our system work the best it can
- The **base tables** are able to be stored in a number of different file storage structures:
    - Heap
    - Sequential
    - Hash
    - B-tree
- **Indexes** may be also assigned to select columns to aid processing requirements
    - As indexes are themselves files, they too can be stored in different structures

File organisations

- Heap
- Sequential
- Hash
- B-tree
- Cluster

Heap (Unordered) Files

Simplest kind of file organisation

New record comes in, goes into the file (physical record), in the order they are inserted.

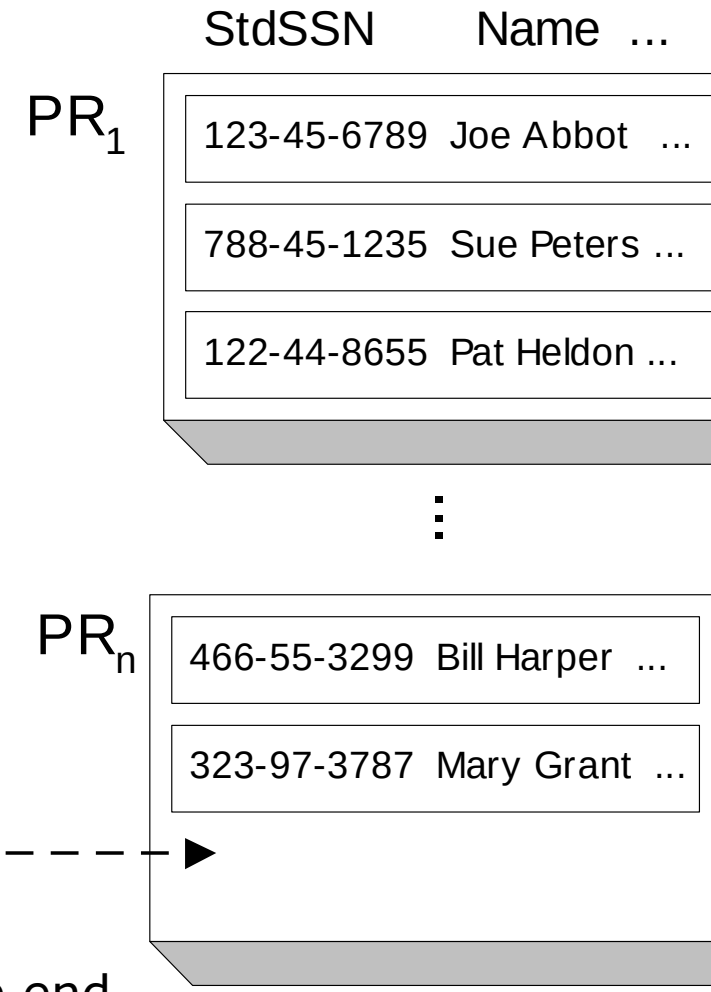
Advantages:

- **Insertion** is going to be efficient because the record gets heaped on top of what's already there Thus
  - Good at loading records in bulk because insertion is more efficient

Disadvantages:

- **Searching** becomes problematic because we need to search through all records until we find the one we want. Thus at worst we might need to go through every single record we have to find the record we want, known as a linear search
- **Deletion** of records also causes issues for various reasons. Firstly
  - The record will be marked as deleted but the space is not deleted and not reused
  - Performance will hinder since the system will search through each of the records even the deleted records with an empty space. Thus continually deletion will cause a build up of empty spaces

Heap (Unordered) Files



Insert a new logical record in the last physical record .

New record comes in gets shoved in at the end

Sequential (Ordered) Files

- Records are stored based on the value of a particular (*ordering*) field

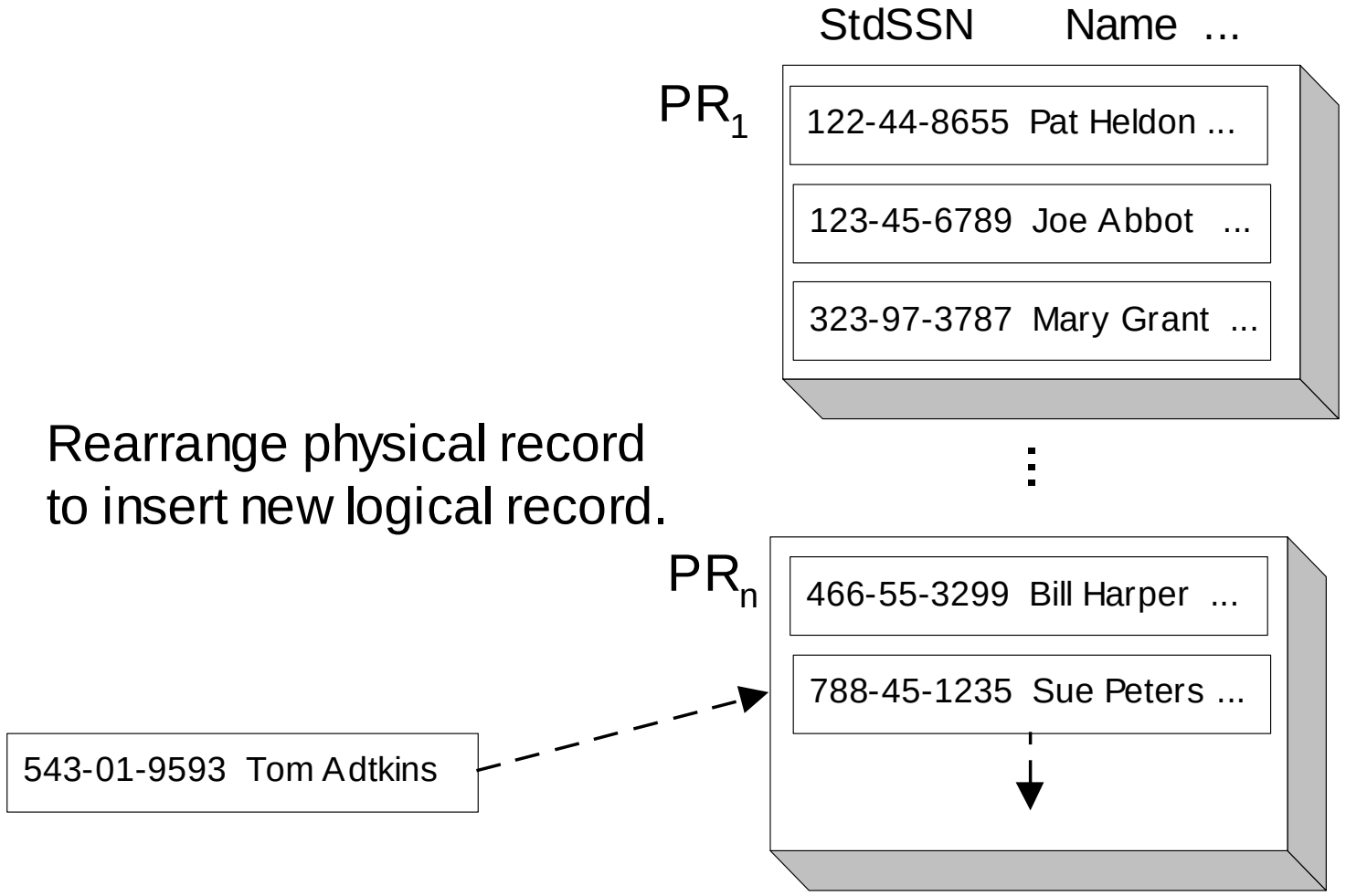
Advantages-

- **Searching** (on the ordering field) is easier because it allows a binary search which is much more efficient than a linear search. Since with binary search you go to middle of file and look for the record you want and you either go up or down and keep dividing it in half until you find record

Disadvantage-

- **Insertion and deletion** can be an issue since every insertion of a record may require the moving of other records to make space. Thus, the order of the records must be continuously maintained
- Usually used with an index – Indexed Sequential– see later slide

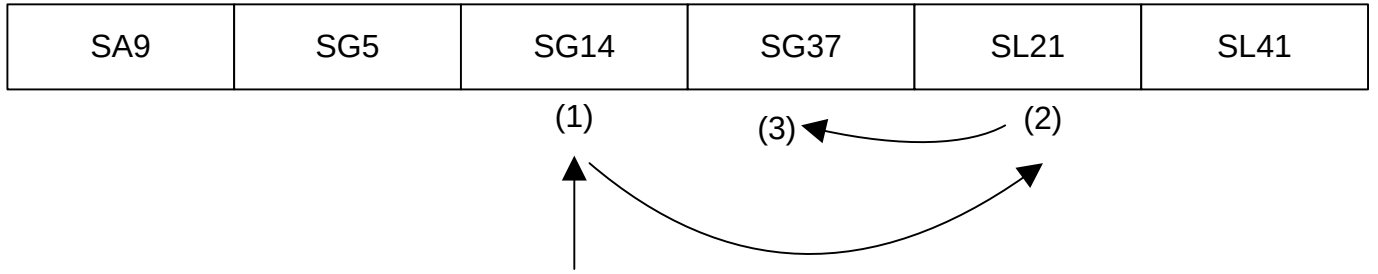
Ordered File – Record Insertion



New record comes other records may get rearranged

Ordered file – binary search

SELECT * FROM Staff WHERE staffNo = 'SG37';



- Go to the middle record (1)
- Use the upper or lower half depending on search value
  - SG37 is > than SG14 so use upper half (2)
- Repeat until value is found (3)

Hash Files

- In a hash file, records are not stored in any particular order on the disk
- Hash file is a keep value look up, which when given a key associates the key to a *physical* record address by some mathematical function
  - This promotes an even distribution of records throughout the file
  - The most common type is a *division-remainder* hashing
- The hash file organisation is commonly referred to as random or direct

Hash files Disadvantages- Collisions

Collisions are where there are more than one record going to a single physical record address. This is a result of hashing not guaranteeing a generation of a unique physical record address. The process explaining collision is the following-

- Addresses that are created through hashing are called *buckets* with *slots* for many records
- In a bucket, records are slotted in order of arrival. And
- *Collisions* occur when a record is trying to be slotted in a bucket that is filled
- Collisions can often deteriorate performance, so collision management techniques are usually adopted with hashing

- Not ideal for retrieving ranges or pattern matching. Since difficult to calculate all the additional hashes we are going need
- Less good when the hash column is updated often

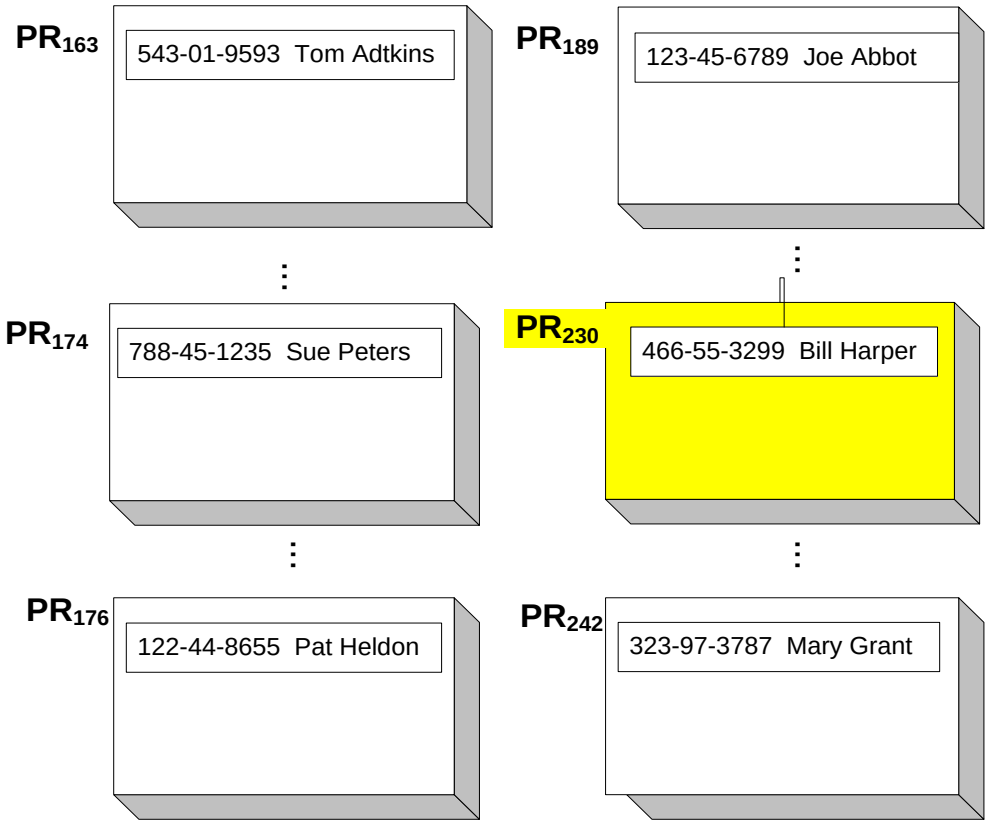
Hash files Advantages

- Ideal for retrieving on an exact match because we are going to that exact bucket

Hash File after Insertions

StdSSN	StdSSN Mod 97	PR Number
122448655	26	176
123456789	39	189
323973787	92	242
466553299	80	230
788451235	24	174
543019593	13	163

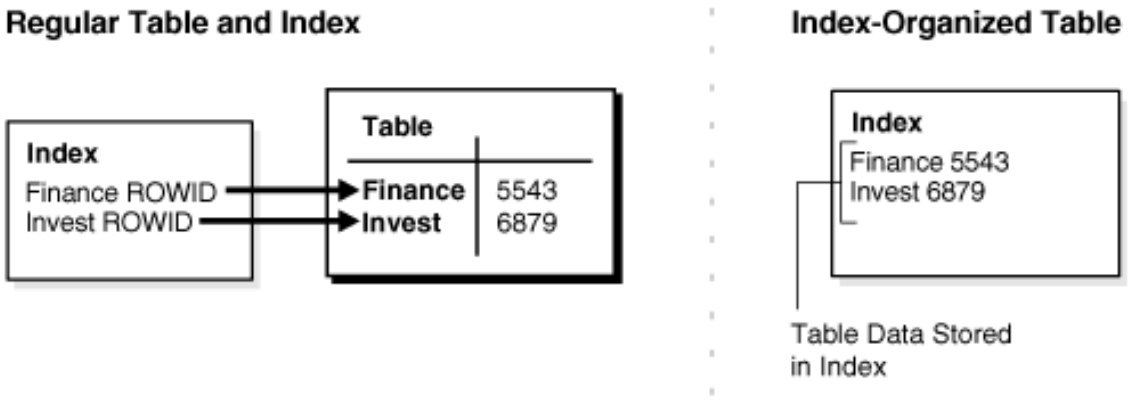
Example shows
Physical Record (PR) =
starting number (150) +
StdSSN Mod 97



Oracle: Indexed-organised tables

- Oracle (and others) index-organised tables are organised as a **b-tree** structure
  (more on b-trees in a later slide)
- There is thus no separate step to retrieve the record after going through the index

Figure 5-9 Structure of a Regular Table Compared with an Index-Organized Table



http://docs.oracle.com/cd/B28359_01/server.111/b28318/schema.htm#i23877

Clusters

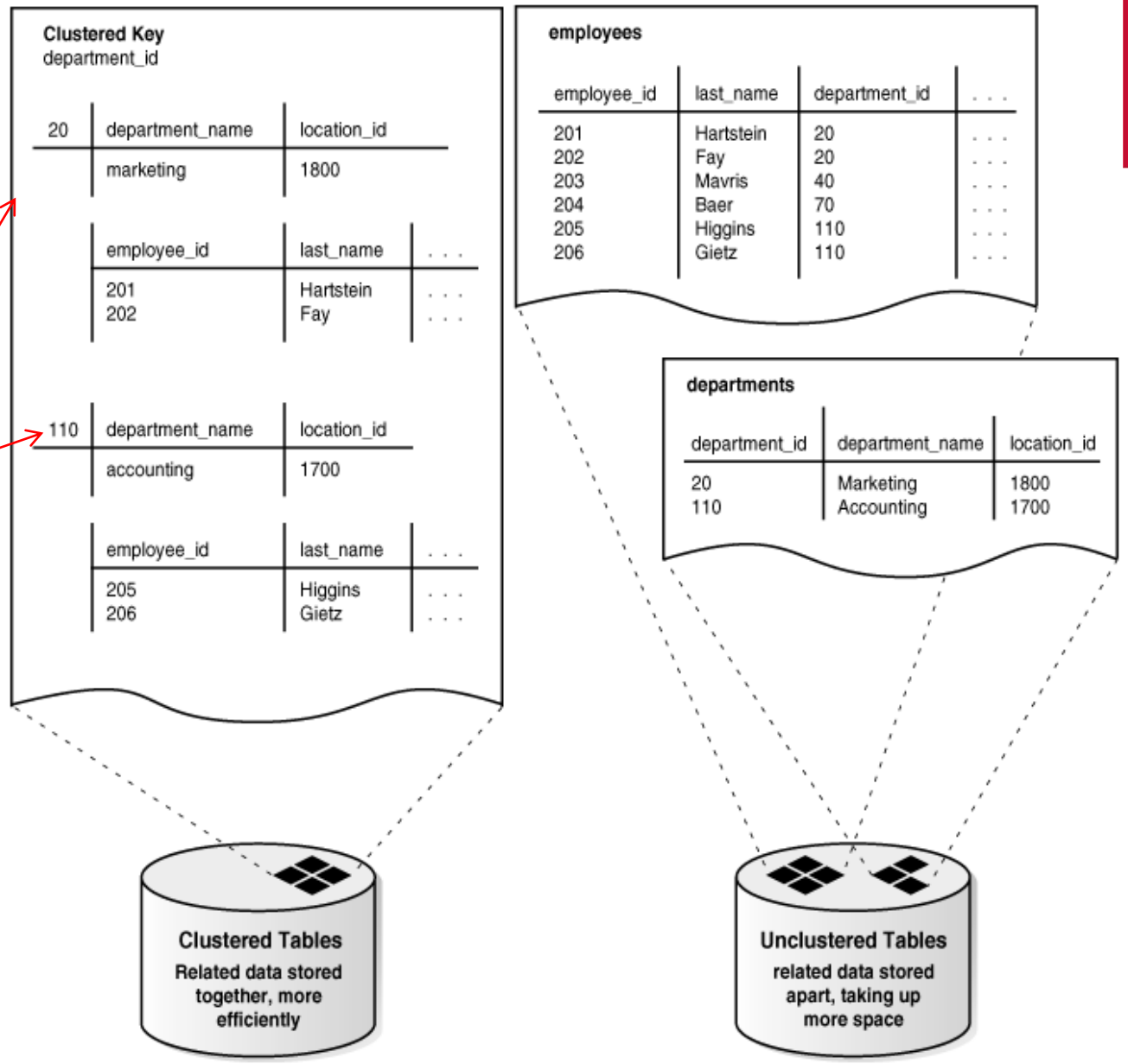
- Groups of relations are stored together in the nearby or same data blocks because the relations share common columns and are often utilised together (e.g., in a join)
- For example, if we were clustering the ARTIST and WORK tables the cluster key would be ArtistNo

Advantage-

- Improves the disk access time since the related records are held physically closer together. Thus reducing number of time we access the disk

Clustered table data

cluster key is department_id – stored only once



Example from http://docs.oracle.com/cd/B28359_01/server.111/b28318/schema.htm#CNCPT608

Indexes

- Indexes are *secondary additional* data structures that enable the DBMS to find particular records in a file more quickly.

- **Avoid using indexes for-**
  - Small relations because more efficient to search through entire relation
  - Indexes must be constantly rebuilt from attributes frequently updated thus processing overhead
  - Attributes consisting of long character strings
  - An attribute where the search will result in a lot of rows of the attribute being retrieved. EG index gender for men football team

- The index file contains:
  - The search key value
  - The address to the corresponding record in the data file
- The index file is organised by the search key value, in a particular structure (e.g. b-tree, bitmap)

Advantages + Disadvantages of Indexes

Advantages-

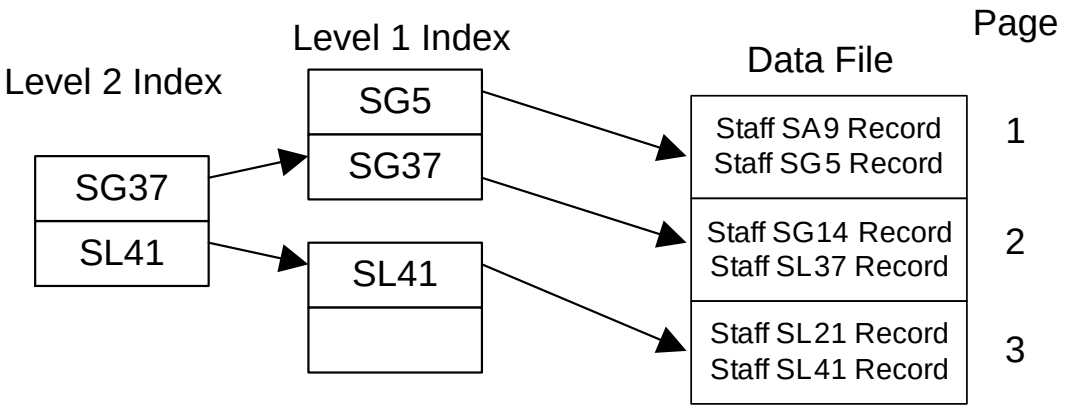
- Improve performance for searches by reducing the need to do a linear search for the record

Disadvantages-

- Insertion is performed worse since the addition of new records require index to be updates thus indexes need to be regularly maintained which comes at a processing cost

Indexes

- The index file contains:
  - The search key value
  - The address to the corresponding record in the data file
- The index file is organised by the search key value, in a particular structure (e.g. b-tree, bitmap)



SQL: CREATE, DROP INDEX

```
CREATE INDEX indexname ON tablename(columnname);

CREATE INDEX idxCustName ON CUSTOMER(CustName);

CREATE BITMAP INDEX idxCustName ON CUSTOMER(CustName);

DROP INDEX idxCustName;
```

Types of Indexes

- **Primary index**
  - Applies to data file that is sequentially ordered by a unique ordering key field
- **Clustering index**
  - Applies to data file that is sequentially ordered, but ordering key field is not unique

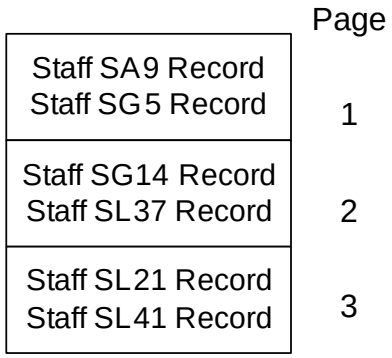
  Secondary index
  - Applies to **non-ordering field** of the data file

•A file can have at most one primary index or one clustering index, but can have multiple secondary indexes

Indexed Sequential Files

Sorted data file with a **primary index**

- Records are able to be processed sequentially or accessed through the search key value via the index



Advantages-

- Ideal for pattern match, exact matching, ranges, part key matches. Because we are able to locate through index

Disadvantages0

- Since the index is static, many insertions and updates cause performance to suffer over time
- Need to reindex from time to time

Secondary indexes

- Index is established on a non-ordering fields
- Used for when-
  - The foreign keys need to be accessed often
  - Any attribute that is heavily used as a secondary key
  - Attributes that are often in- WHERE clauses, Order By Group by or aggregate functions

  Avoid

- Secondary indexes can be in various file structures such as **b-tree** or bitmap

Advantages + Disadvantages of Secondary Index

Advantage-

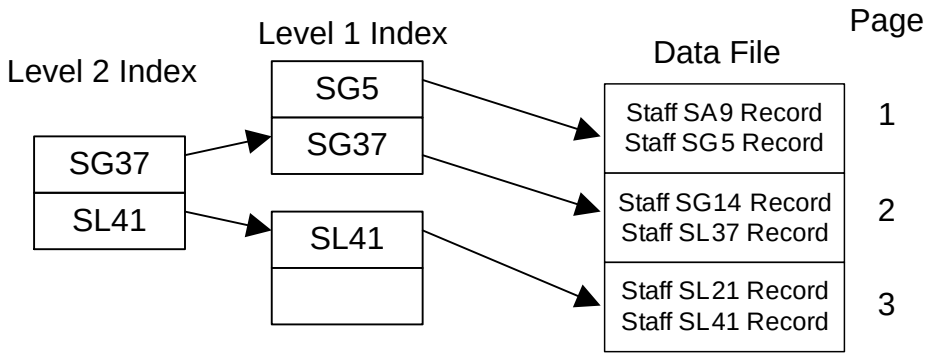
- The performance of queries that utilise attributes other than the primary key are improved

Disadvantage-

- The improved performance of query is countered by increased overhead in maintaining the index to keep it regularly updated with the data file

Multilevel indexes

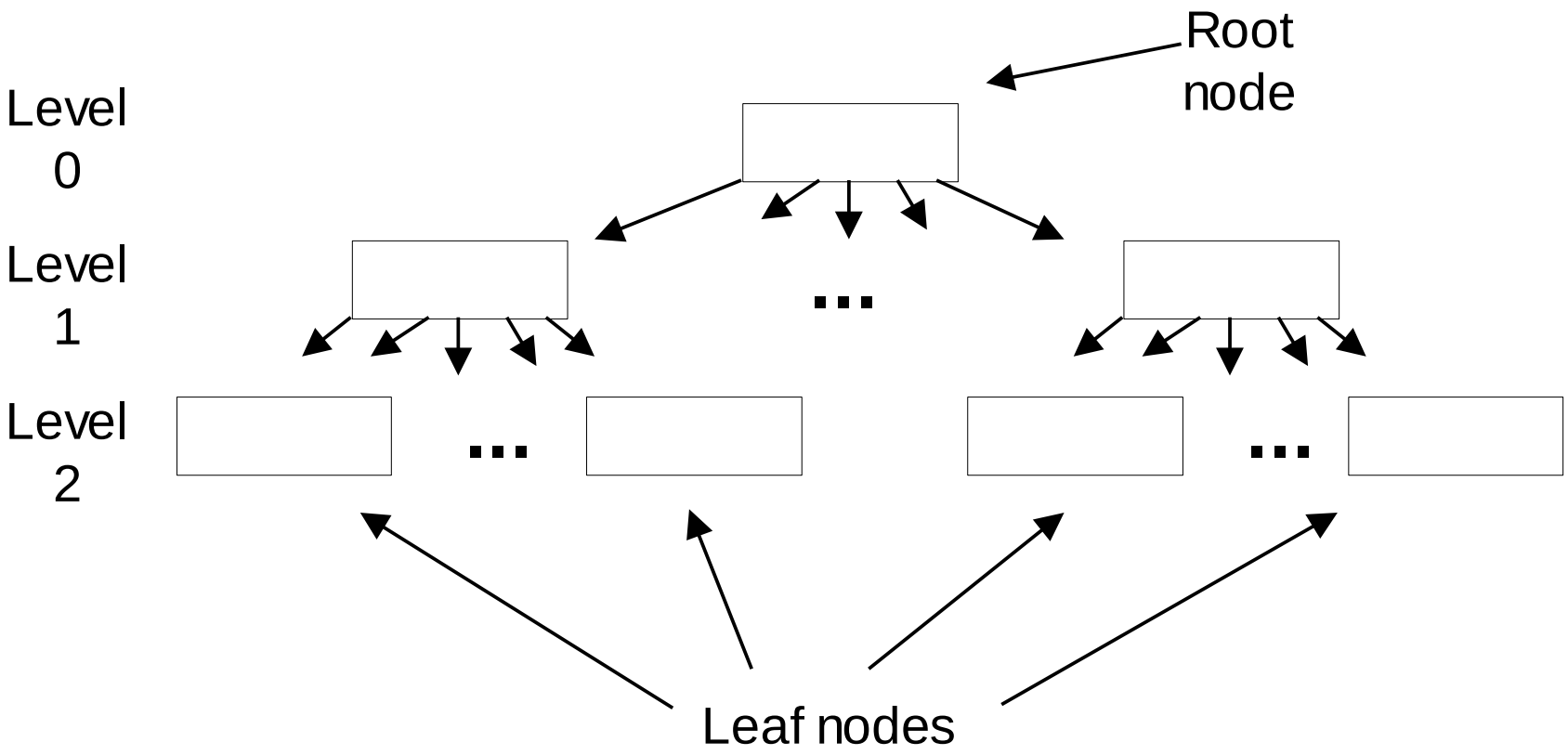
- When an index becomes very large, there is reduced advantage in searching it
- **Multilevel indexes** such as **b-trees** reduce search time/effort by splitting the index into smaller indexes
- This can lead to very efficient retrieval



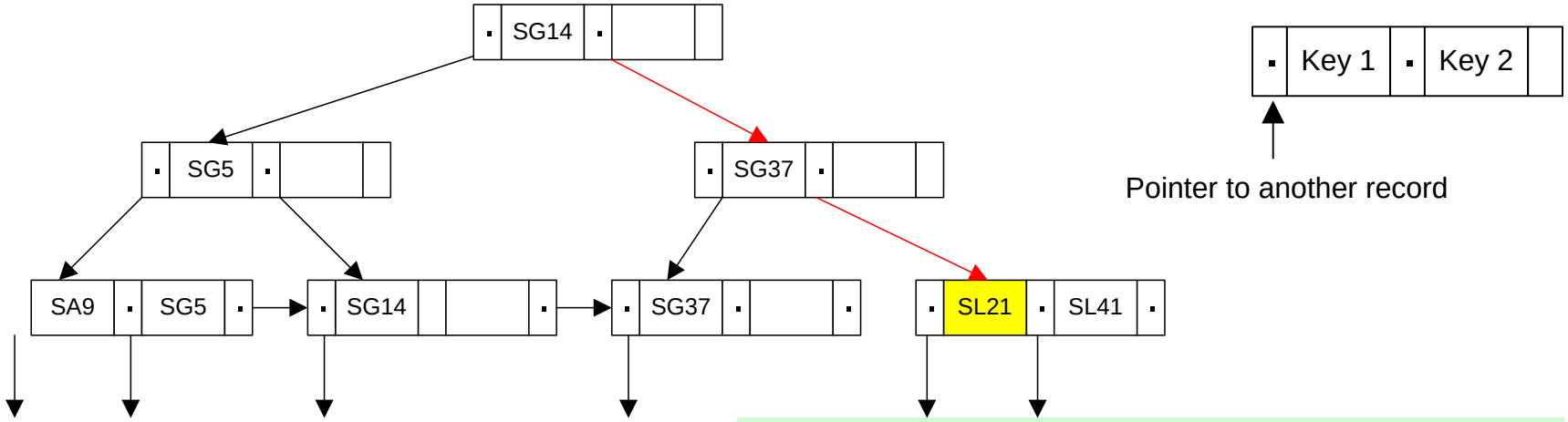
B-tree index

- Is a commonly used index structure supported by most DBMSs
- B-tree provides good performance on sequential search and good performance on key search as well
- A B-tree file is a balanced, multilevel index
  - Each **node** in the tree, except the **root** node, has one parent node and zero or more **child** nodes
  - Root node has no parents
  - Node without children is called a leaf node
  - The depth of the tree is the maximum number of levels between the root and leaf nodes

Structure of a B-tree of depth 3



B-tree example



- If the search value is <= to the Key value the pointer to the left of the Key value is used to find the next node to be searched
- Otherwise, the pointer to the right is used

- To find SL21, we start from the root node.
- SL21 is > SG14 alphabetically so we follow the pointer to the right which leads to SG37
- SL21 is > SG37 so we follow it to the right and arrive at the leaf node containing the address of record SL21

B-tree: when to use (Advantages)

- Supports retrieval based on exact key match, range matching, pattern matching (through links at leaf level), and partial key specification
- Ideal with dynamic tables that change their size often. So they can rebuilding their table quickly when data changes

Indexes in Oracle

Oracle supports the following types of indexes:

- B-tree indexes (default)
- B-tree cluster indexes
- Hash cluster indexes
- Reverse key indexes
- Bitmap indexes
- Bitmap join indexes

See http://docs.oracle.com/cd/B28359_01/server.111/b28318/schema.htm#i5671

The take aways…

- Designing the physical implementation of the database on the chosen DBMS is a complex process
- The DBMS will have available particular file organisations for both the data files and any index files
- Each file organisation has advantages and disadvantages for particular circumstances
- Secondary indexes on non-key fields can be created to improve search performance – again there are various general principles that apply

3. Query processing and query optimisation

Monitoring and tuning

- During the physical design phase we can’t determine how database is running and how systems that’s access it are running. So we collect performance statistics of the running system and database. Then we come back and tune the database and the system if required
- One of the most important of the ongoing responsibilities of the DBA
- Sources of data for tuning include:
    - reports by users
    - the data dictionary (information about available indexes etc)
    - Specific tools e.g. Oracle EXPLAIN PLAN

Query optimisation and processing

Query optimisation-

- The input such as SQL and statistics on the relations go into the optimiser
- The query optimiser generates multiple execution plans for the query based on storage structure, indexes of tables and join strategies allowed for the DBMS
- The query optimiser compares and evaluate the different generated plans. The evaluation is usually a cost-based evaluation
- The final plan is selected, which is usually the plan with the lowest cost, is outputted and therefore eventually executed

Query processing-

- The DBMS retrieves our query and sends it to the *query processor*
- The *query processor* converts the SQL to a series of steps that are executed in a sequence.
- Therefore, the *query processor* is the component of DBMS that deals with executing a query input by the user

Query processing and optimisation

The **query optimiser** selects the best one to execute the query making use of the existing tables and indexes

Optimisation techniques differ according to the DBMS, but can be:

- **Rule-based, RBO** (legacy) - query plans are judged according to a set of rankings of different plans (e.g. if there is an index, use it)
- **Cost-based, CBO** (most common) – The query plans are compared based of the evaluation of the different costs of each plan calculated using data distribution statistics. Cost may be things such as:
  - How many disk accesses are they going to be
  - How much CPU time is used for (sorting, searching)
  - What is the storage costs of intermediate files
  - Network communication time (if database is distributed)

Cost-based

Cost estimation for a query will also depend on what's in the DBMS data dictionary such as:

- Amount of records in each base relation
- Amount of disk blocks needed to hold the table
- For each attribute the amount of unique values in it
- The amount of levels in each index
- Selection cardinality of each attribute

Query execution plan

- Recall from the relational algebra that there are many different ways of writing the same statement that imply different processing sequences:

π FamilyName, GivenName
($\sigma_{\text{NumberOfRooms} > 3}$ (PROPERTY $*_{\text{P.OwnerNo=O.OwnerNo}}$ OWNER))

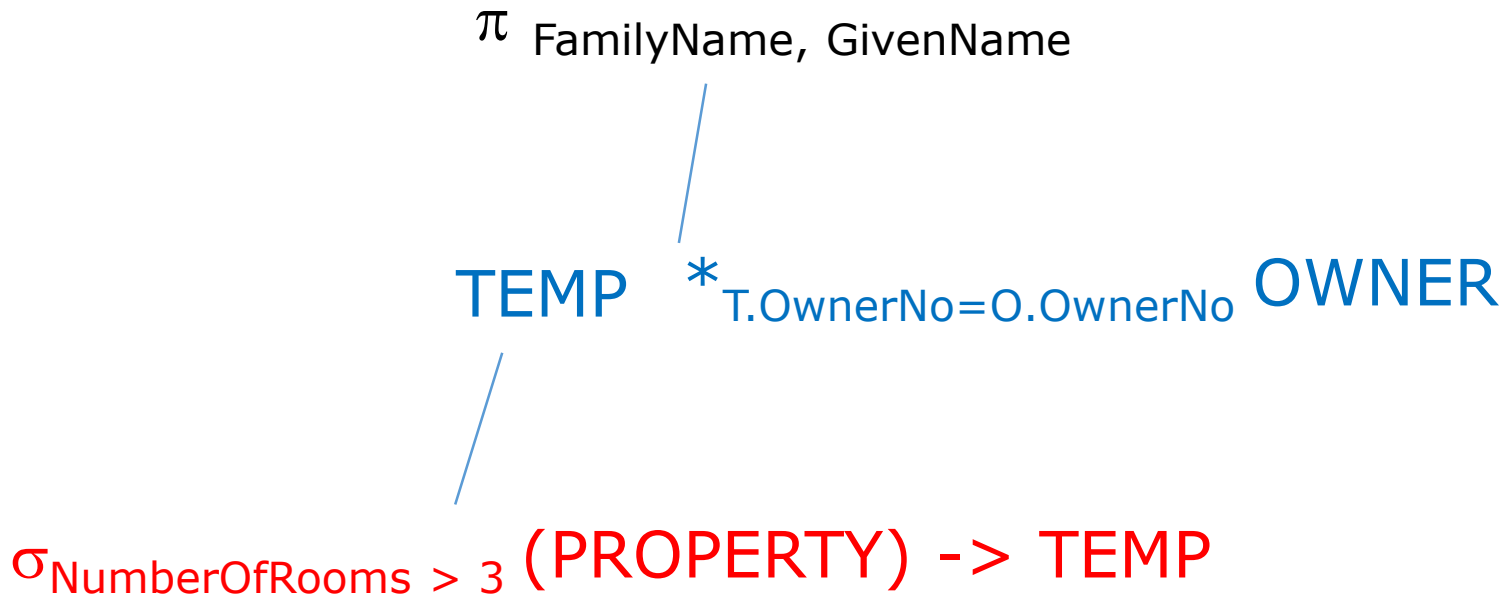
OR

π FamilyName, GivenName (OWNER$*_{\text{O.OwnerNo=P.OwnerNo}}$ ($\sigma_{\text{NumberOfRooms} > 3}$ (PROPERTY)))

(red statement is done first, then blue, then black)

Query execution plan

We can show this in the form of a tree, read from the bottom up:



Query execution plan

- The query optimiser generates possible **query execution plans** based on the storage structures, indexes, etc, of the tables, and the access routines such as join strategies available in the DBMS
- A plan is chosen based on some costing algorithm that also includes statistics about the table data

General rules for query optimisation

Generate most efficient expressions for query processing by (using relational algebra!)

Some general rules are:

- Use restrict operations as soon as possible because reducing number of rows we need to join/search
- Select most restrictive restricts first
- Try to use projects early but only if possible
- Compute common expressions once and reuse
- Transform Cartesian product followed by restrict into join

Join methods

- For a DBMS there are various different strategies by which two relations are joined and thus processed
    - Nested loop
    - Merge
    - Hash
- The optimiser will consider the query, the table structures and the data in the table to determine the optimal strategy

See Oracle documentation at https://docs.oracle.com/database/121/TGSQL/tgsql_join.htm#TGSQL244

Nested loops join

- The relations are searched through by getting every record in the ‘inner’ table with all the records in the outer table and determining whether the join condition is fulfilled
  - (think first record in first table and match associated record in the second table. Then go to the second record in the first table…)

Disadvantages-

- Inefficient solution when tables are not sorted and there is no indexes

Advantages-

- Ideal for small relations with indexes on the join conditions
- Better efficiency when the inner table has an index which is then used to find matching records in the outer table

Sort-merge join

- **Sort-merge** joins are where both relations are stored in order on the join fields thus the tables need to be only be scanned once each (in parallel) to get matching records
- An **index-merge** join is the same as sort-merge but for each join attribute there is an index assigned to it. Thus indexes are scanned through in parallel

Advantages-

- Ideal for inequality conditions (<, >, etc) and equality conditions

Hash join

- The optimiser will select the smaller of the two relations to create a hash table on the join key in memory.
- Once the hash table is created, the larger relation is scanned and the optimiser performs the same hashing algorithm then probes the hash table to retrieve the rows that satisfy the join condition

Advantages-

- Ideal when a lot of the table will be joined + the join is on equality

Oracle Query Optimiser

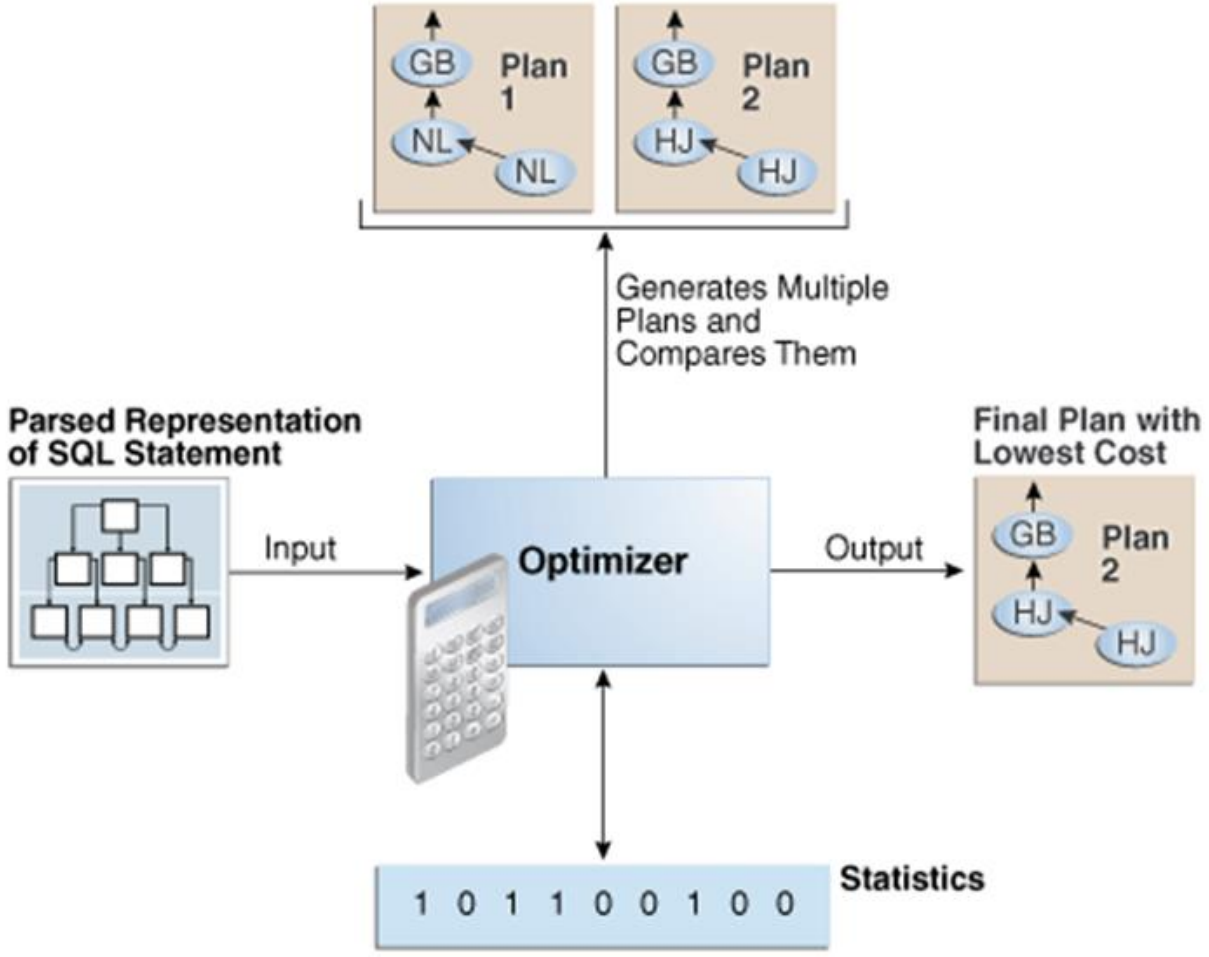


Image and discussion from 'Query Optimiser Concepts' at https://docs.oracle.com/database/121/TGSQL/tgsql_optcncpt.htm#TGSQL94982

Oracle Query Optimiser components

- Query Transformer
- Estimator
- Plan Generator

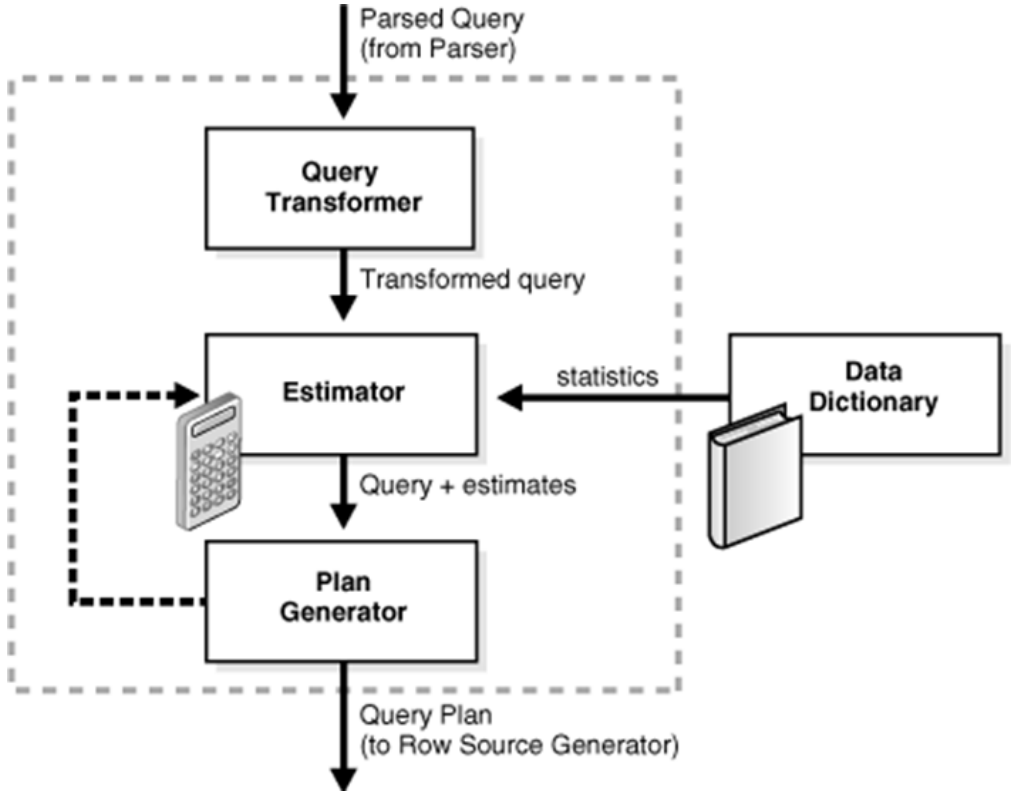


Image and discussion from 'Query Optimiser Concepts' at
https://docs.oracle.com/database/121/TGSQL/tgsql_optcncpt.htm#TGSQL94982

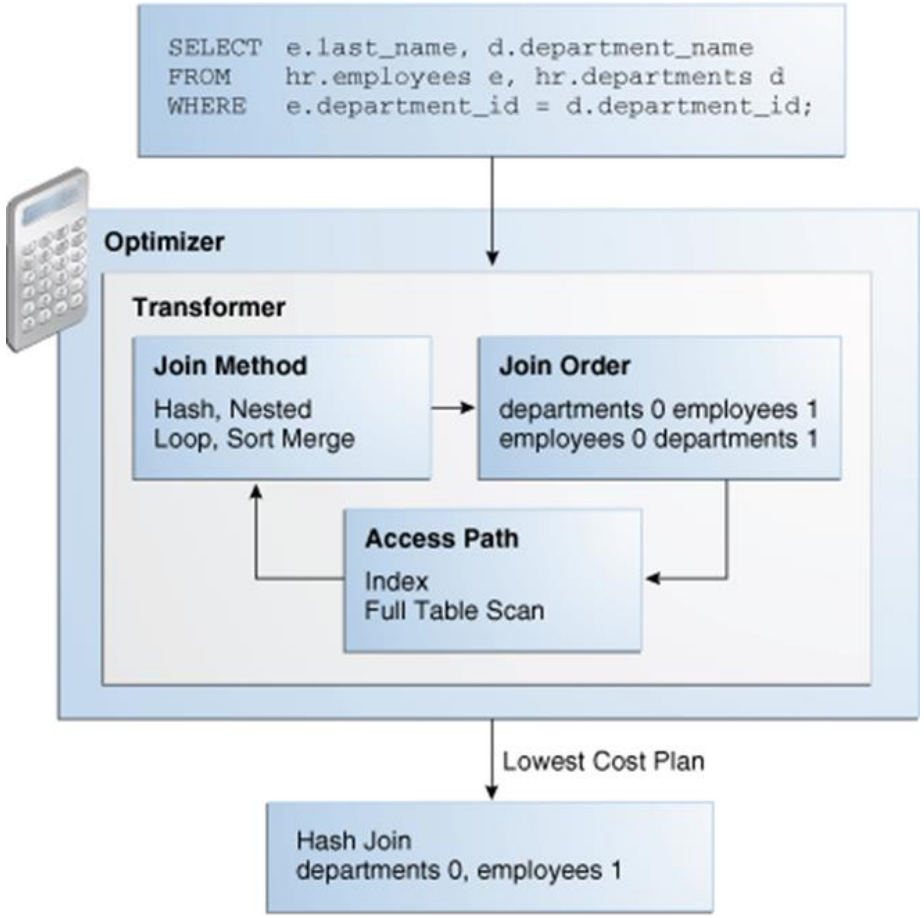
Oracle Query Optimiser components

- **Query transformer** works out whether the query can be converted into a more efficient one
- **Estimator** works out the overall cost of a given execution plan, based on
    - Cardinality (how many rows the query returns)
    - Selectivity (percentage of rows the query returns)
    - Cost (disk I/O, CPU, memory usage)
- **Plan generator** looks at various different plans for the query by trying various access paths, join methods and join orders, then selects the lowest cost plan

Image and discussion from 'Query Optimiser Concepts' at
https://docs.oracle.com/database/121/TGSQL/tgsql_optcncpt.htm#TGSQL94982

Oracle: Example of plan generator

- Image and discussion from ‘Query Optimiser Concepts’ at https://docs.oracle.com/database/121/TGSQL/tgsql_optcncpt.htm#TGSQL94982



Choosing a plan

- The **lowest cost plan** is chosen using various heuristics – e.g. the best plan in a given time; stop looking whenever a higher cost plan is found; if a low cost plan is found, stop looking as improvements will only be marginal
- Sometimes the plan selected will be suboptimal as it is chosen under these constraints
- **Adaptive query optimisation** enables the optimiser to make run-time adjustments to execution plans by taking into account additional information that let it build a better plan

Oracle EXPLAIN PLAN

- The Oracle EXPLAIN PLAN utility allows the DBA to view the execution plans the optimiser generates, in the form of a table showing the objects, the operations on them, and the sequence
- The SQL to create the plan is:

```
EXPLAIN PLAN FOR [SQL statement];
```

- And to display the plan:

```
SELECT PLAN_TABLE_OUTPUT FROM
TABLE(DBMS_XPLAN.DISPLAY());
```

EXPLAIN PLAN FOR
SELECT * from ARTIST A, WORK W
WHERE A.ArtistID = W.ArtistID;

```
PLAN_TABLE_OUTPUT
Plan hash value: 395247666

-------------------------------------------------------------------------------
| Id  | Operation          | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |        |    35 |  7700 |     6   (0)| 00:00:01 |
|*  1 |  HASH JOIN         |        |    35 |  7700 |     6   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| ARTIST |    11 |  1034 |     3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| WORK   |    35 |  4410 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("A"."ARTISTID"="W"."ARTISTID")

Note
-----
   - dynamic statistics used: dynamic sampling (level=2)
```

Execution plan with no PK or FK indexes defined

EXPLAIN PLAN FOR
SELECT * from ARTIST A, WORK W
WHERE A.ArtistID = W.ArtistID;

Murdoch UNIVERSITY

```
PLAN_TABLE_OUTPUT
Plan hash value: 2647660467

-------------------------------------------------------------------------------------
| Id  | Operation                     | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |          |    35 |  7700 |     6  (17)| 00:00:01 |
|   1 |  MERGE JOIN                   |          |    35 |  7700 |     6  (17)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| ARTIST   |    11 |  1034 |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN            | ARTISTPK |    11 |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                   |          |    35 |  4410 |     4  (25)| 00:00:01 |
|   5 |    TABLE ACCESS FULL          | WORK     |    35 |  4410 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("A"."ARTISTID"="W"."ARTISTID")
       filter("A"."ARTISTID"="W"."ARTISTID")
```

Execution plan with PK indexes on both tables and FK index on Work

The take aways…

- Monitoring and tuning the operational database continues throughout its life
- Tuning involves the deliberate choice of physical design by the DBA plus the inbuilt query optimisation features of the DBMS
- The **query optimiser** operates by creating alternative query plans and evaluating them by their relative cost, calculated using current data distribution statistics
- The query plan is available to the DBA through utilities such as EXPLAIN PLAN

Topic learning outcomes revisited

Murdoch UNIVERSITY

After completing this topic you should be able to:

- Describe the activities in physical database design
- Describe some different types of file organisation used in commercial DBMSs, and when each is appropriate
- Determine when secondary indexes are appropriate, and when they are not
- Briefly describe how query optimisation works
- Describe in general terms how the hash, sort-merge and nested loops join strategies work

What's next?

Databases are designed to be a shared resource, and it is essential to maintain the integrity of the database when multiple users are accessing it, or if the system crashes. In the next topic we look at how the DBMS handles concurrency management and recovery.

Central to both concurrency management and recovery is the idea of the *transaction* as a single logical unit of work.