COMP9311:
Database Systems

Term 3 2021

Week 8 Data Storage, Indices and Query Processing

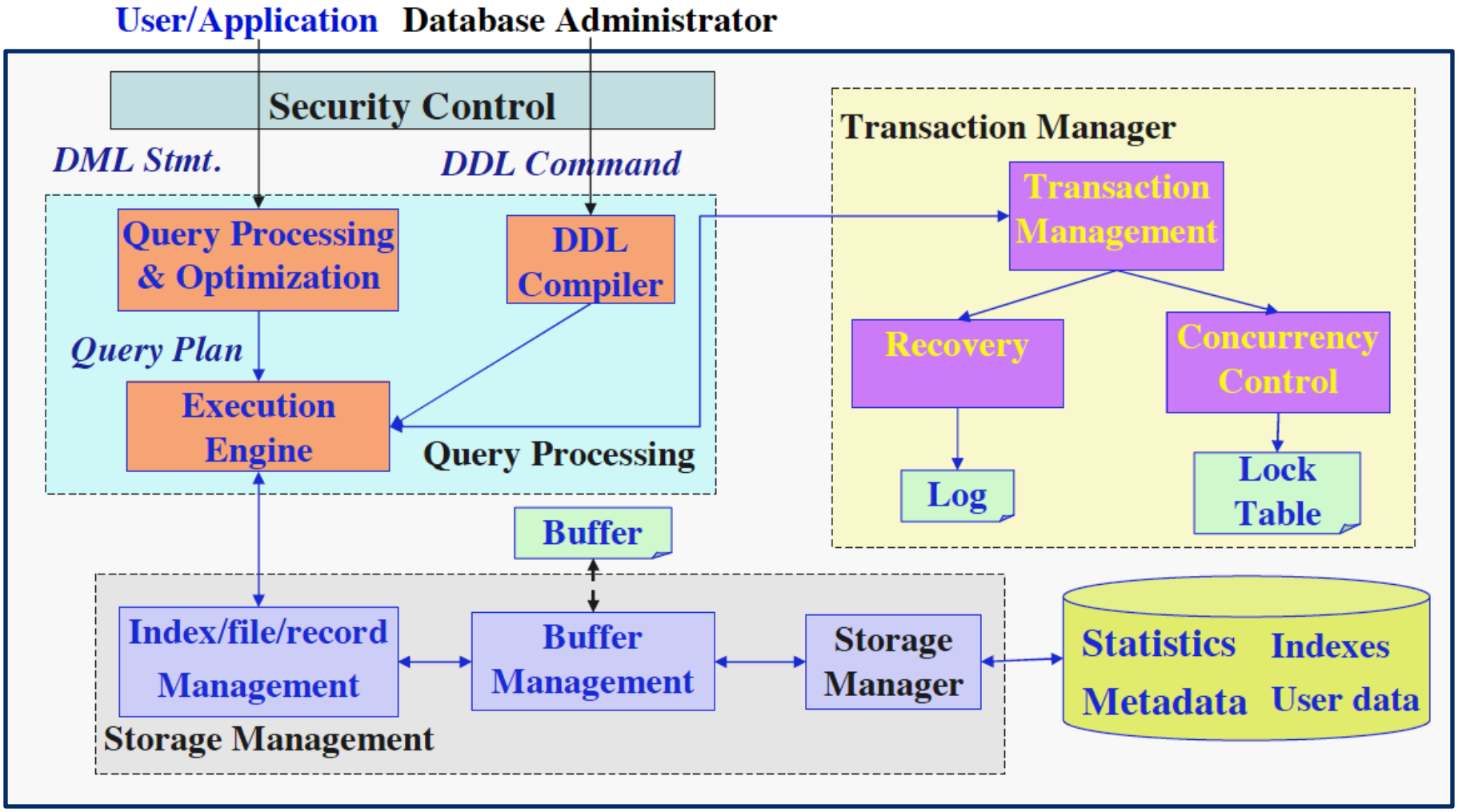
By Helen Paik, CSE UNSW

Textbook: Chapters 16, 17, 18 and 19

Disclaimer: the course materials are sourced from

- previous offerings of COMP9311 and COMP3311
- Prof. Werner Nutt on Introduction to Database Systems (http://www.inf.unibz.it/~nutt/Teaching/IDBs1011/)

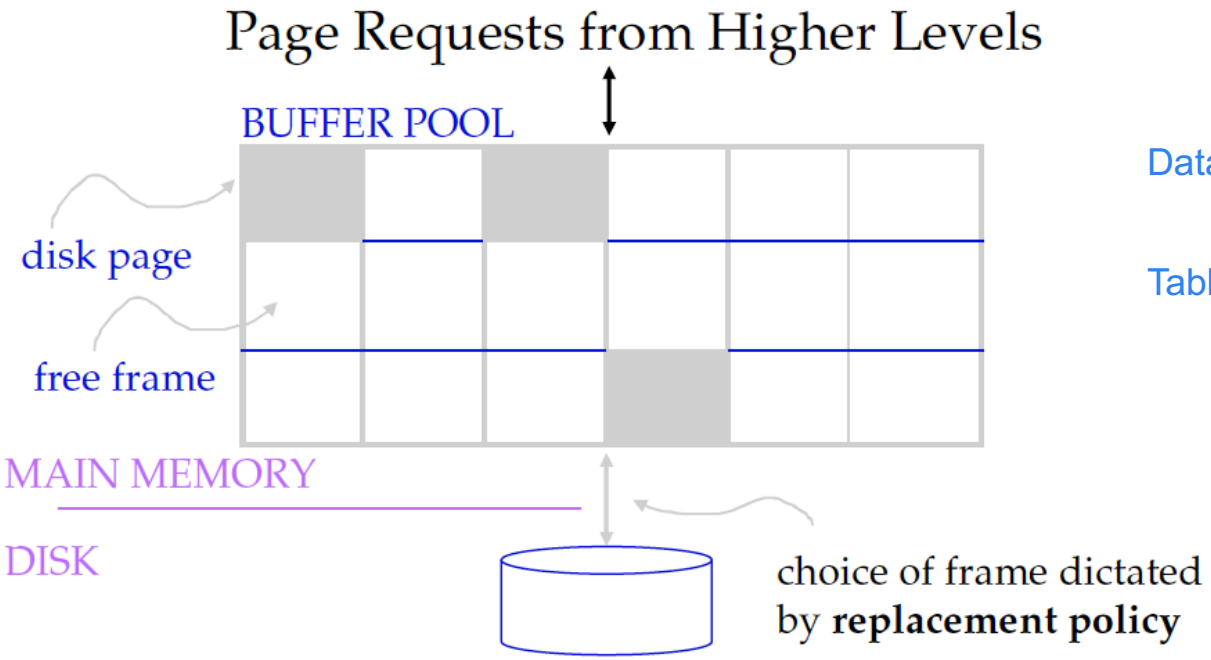
Functional Components of DBMS



Data Storage Principles

- Database relations are implemented as **files of records.**
- This is still an abstraction: the real storage medium are disks, which consist of pages (size about 0.5–5 kbytes)
- Pages are read from disk and written to disk → high cost operations!
- Mapping: each record has a record identity (*rid*), which identifies the page where it is stored and its offset on that page
- The DBMS reads (and writes) entire pages and stores a number of them in a buffer pool
- The buffer manager decides which pages to load into the buffer (Replacement policy: e.g., “least recently used” or “clock”)

Buffer Management in DBMS



Data must be in main memory for DBMS to operate on it …

Table of <frame#, pageid> pairs is maintained in the buffer pool …

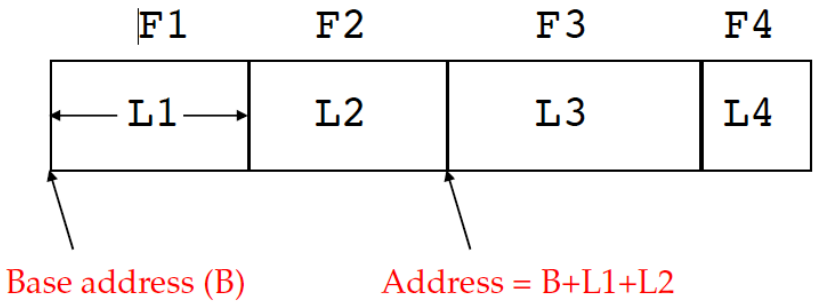
When a page is requested ... and if requested page is not in pool:

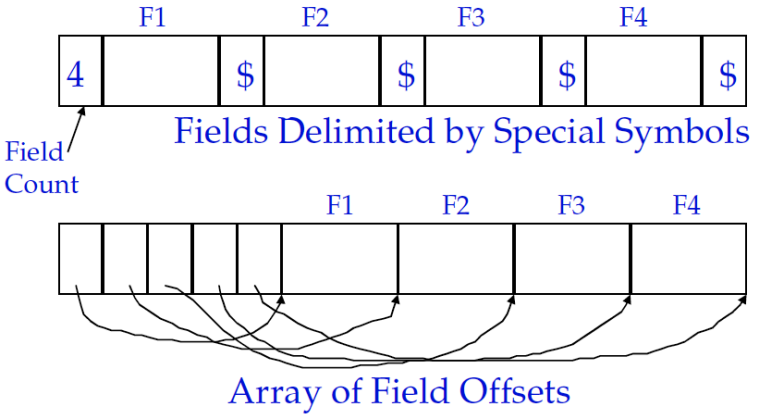
- *Choose a frame for replacement (replacement policy – Least Recently Used, Clock, etc.)*
- *If frame is dirty, write it to disk*
- *Read requested page into chosen frame*
- *Pin the page and return its address.*

If requests can be predicted (e.g., sequential scans) pages can be pre-fetched several pages at a time!

File and Record organisation

So … table data are stored as **files of records.**





Record format: Fixed length

Information about field types same for all records in a file; stored in system catalogs. Finding the i^{th} field requires scan of record.

Record format: Variable length

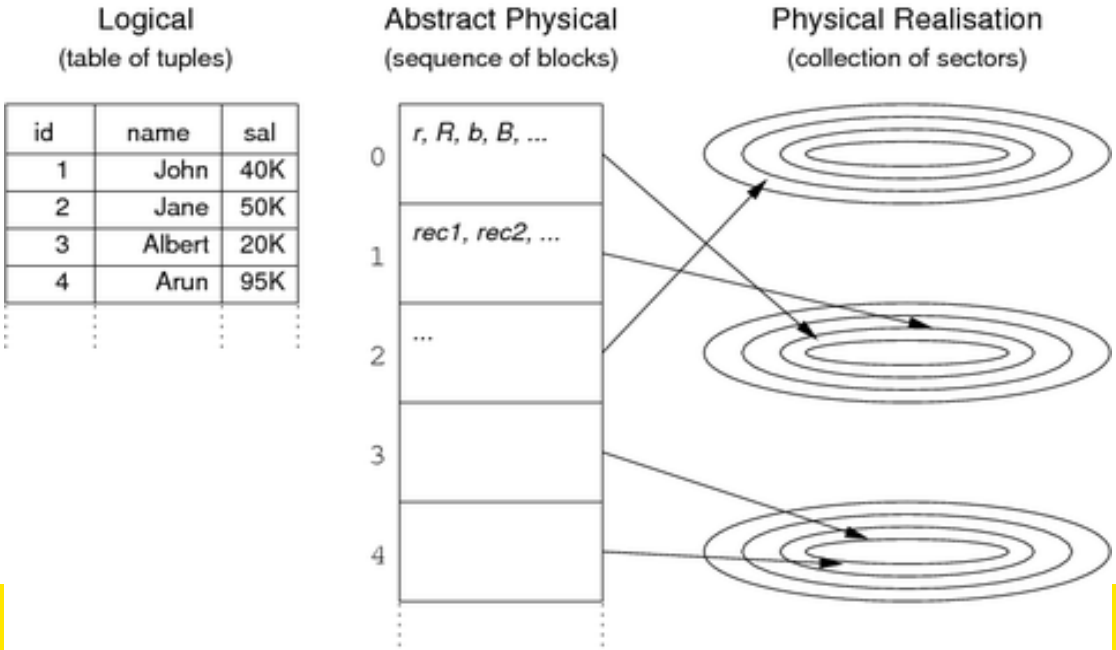
- Two alternative formats (# fields is fixed)
- Second option offers direct access to i^{th} field, efficient storage
- small directory overhead

Files of Records

Ultimately, data is stored into disk as **disk pages**, but higher levels components of DBMS operate on **records and files of records** (abstract physical)**, the record blocks in a file are stored in different disk pages in different sectors of a disk**

A file: is a collection of pages, each containing a collection of records. The storage management component, through the abstraction of "files", support:

- insert/delete/modify record
- read a particular record (specified using record id)
- scan all records (possibly with some conditions on the records to be retrieved)



Alternative File Organisations

Alternatives are *ideal* for some situation, and *not so good in others*:

Heap Files: No order on records. Suitable when typical access is a file scan retrieving all records.

Sorted Files: Sorted by a specific record field (key). Best if records must be retrieved in some order, or only a "range" of records is needed.

Hashed Files:

- File is a collection of buckets, each bucket containing some records.
- Uses a hashing function *h. e.g.,* h(search-fields from r) computes the buckets that record r belongs (looks at only some of the fields of r, the search fields.)
- Good for equality selections.

Indexes (basic concept)

Find all subcode belonging to the Law faculty (i.e., subcode = LAWS)

Basic strategy = scan …, test, select

Not efficient …

An "idea" of an index on a file on the search key 'subcode' may look like …

```
LAND, {1}
ANAT, {2,19}
BENV, {…}
LAWS, {4,7, …}
```

An index gives a short cut to the tuples that match the search key

An added cost for building/maintaining it …

subcode	code	name	uoc	career
LAND	LAND1170	Design 1	4	UG
ANAT	ANAT2211	Histology 1	0	UG
CHEN	CHEN2062	Intro to Process Chemistry 2	3	UG
LAWS	LAWS2332	Law and Social Theory	8	UG
ECON	ECON4321	Economic History 4 Honours	48	UG
AHIS	AHIS1602	History 1	12	
LAWS	LAWS2425	Research Thesis	4	UG
ENVS	ENVS4503	Env Sci Hons (Geog) 18uoc	18	UG
SOLA	SOLA5058	Special Topic in PV	6	PG
BENV	BENV2704	Advanced Construction Systems	3	UG
ANAT	ANAT3141	Functional Anatomy 2	6	UG
BENV	BENV2205	Classical Architecture	3	UG
BENV	BENV2402	Design Modelling - Time Based	6	UG
BIOT	BIOT3081	Environmental Biotech	6	UG
BIOC	BIOC4103	Genetics 4 Honours Full-Time	24	UG
ECON	ECON4127	Thesis (Economics)	12	UG
ENVS	ENVS4404	Environmental Science 4 Chemis	24	UG
LAWS	LAWS2423	Research Thesis	8	UG
MUSC	MUSC2402	Professional Practices D	6	UG
REGS	REGS3756	Negotiation	8	UG
HPSC	HPSC5020	Supervised Reading Program	8	PG
CRIM	CRIM4000	Crim Honours (Research) F/T	24	UG
BIOC	BIOC4109	Genetics Honours (PT)	12	UG
CEIC	CEIC4105	Professional Electives	3	UG

Indexes

An **index** on a file speeds up selections on the **search key fields** for the index

- *Any subset of the fields* of a relation can be the *search* key for an index on the relation
- *Search key* is not the same as *key*
  (minimal set of fields that uniquely identify a record in a relation)

An index contains a collection of **data entries**, and supports efficient retrieval of all data entries K^* with a given key value K.

Data Entries in Indexes

```
LAND,{(LAND, LAND1170, Design 1, 6, ..) }
ANAT,  {(ANAT, ANAT2211, Histology 1, …)}
ANAT,  {(ANAT, ANAT3141, Functional Anatomy, …)}
Etc.
```

Three alternatives: *(is actual data stored or pointers to the data stored?)*

- Data record with key value K
- $\langle K, r \rangle$, where r is rid of a record with search key value K
- $\langle K, [r_1, \ldots, r_n] \rangle$, where $[r_1, \ldots, r_n]$ is a list or rid's of records with search key value K

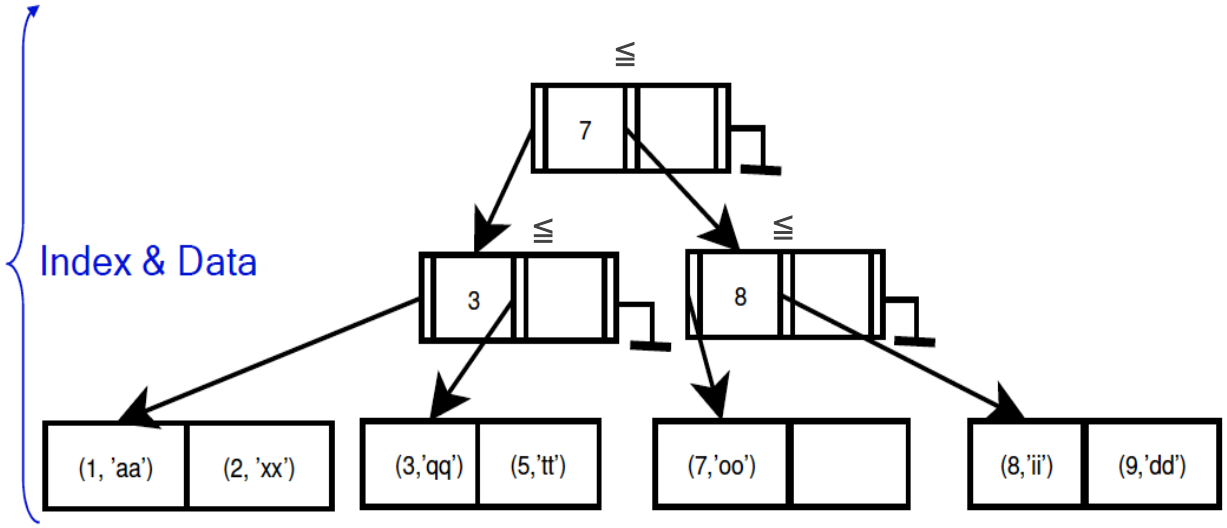
```
LAND, 1
ANAT, 2
ANAT, 19
Etc.
```

```
LAND, 1
ANAT,  [2, 19]
Etc.
```

Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value K

- Examples: B+-trees, hash-based structures
- Index may contain auxiliary information that directs searches

e.g., Alt1 (Data record with K) in B+ Tree

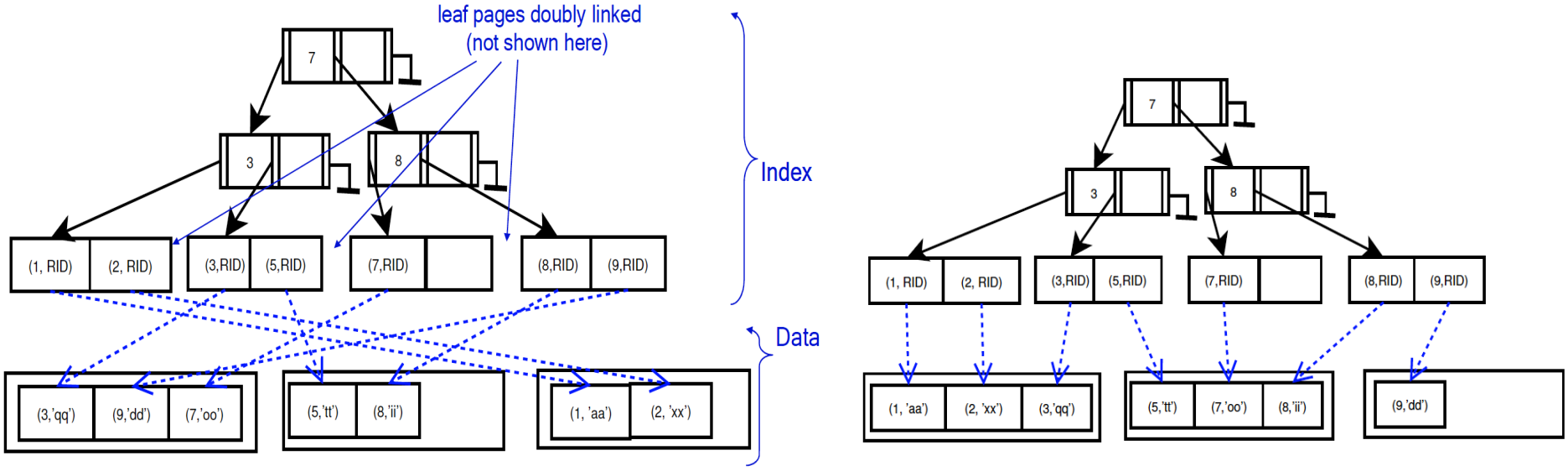


A	B
1	aa
2	xx
3	qq
5	tt
7	oo
8	ii
9	dd

The records (data) are organised along with the index in one.

- No need to follow pointers. The leaves of the tree contains the data
- That is, the index contains the data file itself … this is the full copy of the data. Typically you can only build one index in this manner as having another index on a different search key will duplicate the entire data …

e.g., Alt2 (K with r) in B+ Tree – (un)clustered



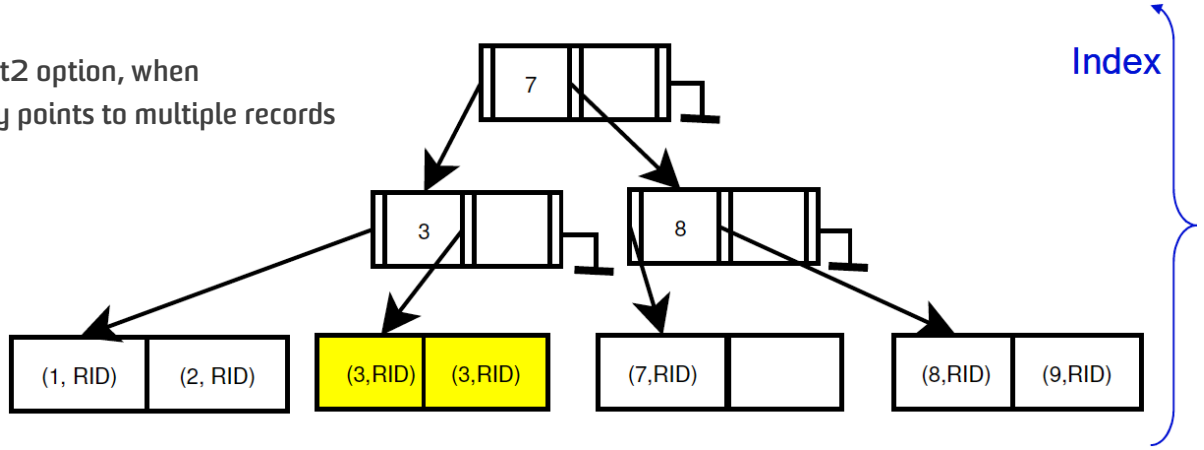
The leaves of the index contains a pointer to the data (single record)

You can build many such indexes on a file (different search keys) as the index is separated from the data

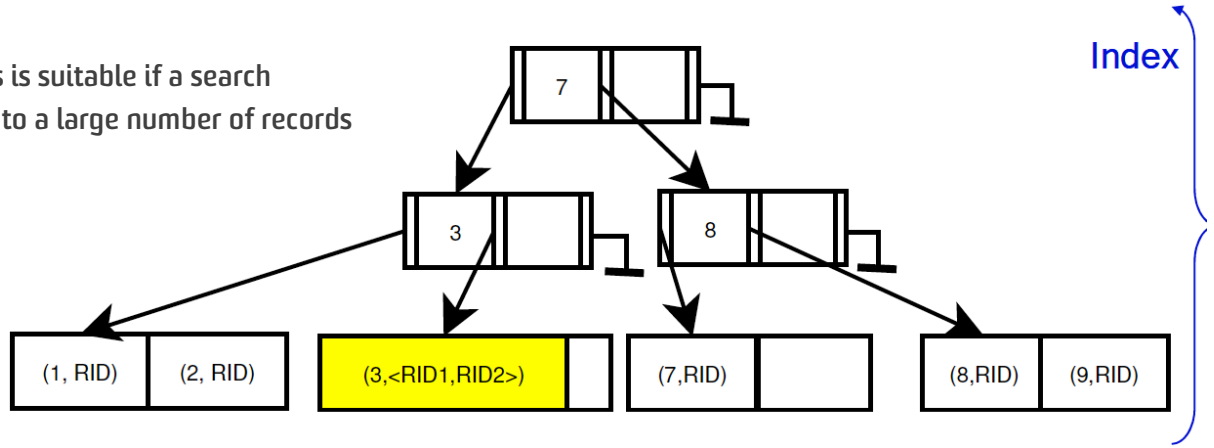
The underlying file that contains the records may or not be sorted … when unsorted, the arrows (i.e., the pointers to the data) 'cross' each other, this is referred to as 'unclustered' index option (cf. clustered, on the right)

e.g., Alt2 and Alt3 in B+ Tree

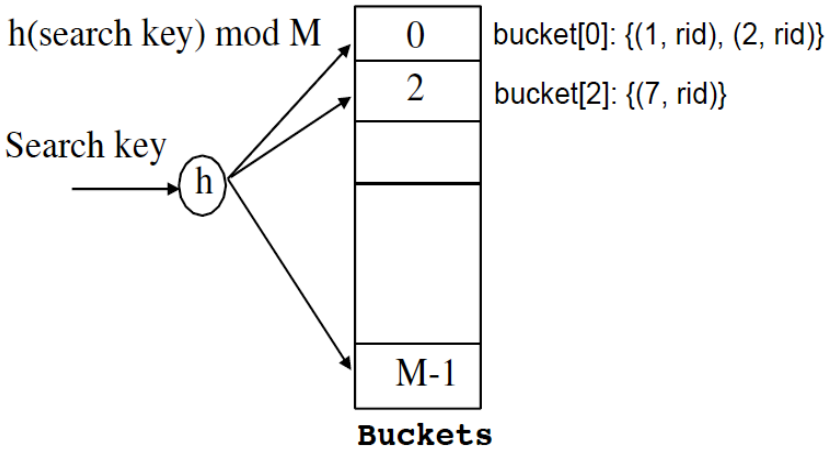
In Alt2 option, when
a key points to multiple records



Alt3 option: this is suitable if a search
key could point to a large number of records



e.g., Indexes in Hash



(an approximate diagram of Hash Index)

Index contains “buckets”, each bucket contains the index data entries …

A hash function works on the search key and produces a number over the range of 0 … M-1 (M is the number of buckets).

e.g., h(K) = (a * K + b), where a, b are constant ... K is the search key.

Fast to search (i.e., no traversing of tree nodes)

Best for equality searches, cannot support range searches.

Index can be dense or sparse

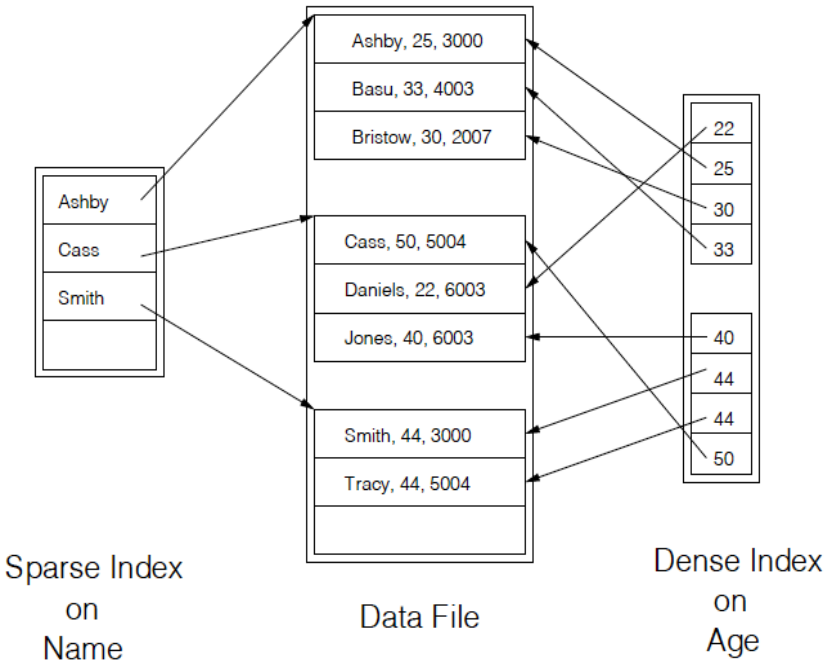
Regardless of the index structure … An index can be dense or sparse

Dense, if the index contains at least one data entry per every value of the search key

- faster to search for a particular record
- high cost

Sparse, if only some values of the search key have data entries

- low cost
- slower to search (i.e., some scan required)



Summary on Storage Management

- Many alternative file organisations exist .. Heap, Sorted and Hash files … each appropriate in some situation.
- Data entries can be (1) actual data records, (2) key, rid-pairs, (3) key, rid-list-pairs
- Choice orthogonal to indexing technique used to locate data entries with a a given key value.
- There may be several indexes on a given file of data records, each with a different search key.
- Indexes can be classified as
  - clustered vs. unclustered
  - dense vs. sparse.
- Differences have important consequences for utility/performance

DB Application Performance

In order to make DB applications efficient, we need to know:

- what operations on the data does the application require (which queries, updates, inserts and how frequently is each one performed)
- how these operations might be implemented in the DBMS (data structures and algorithms for select, project, join, sort, ...)
- how much each implementation will cost (in terms of the amount of data transferred between memory and disk)

and then, as much as the DBMS allows, "encourage" it to use the most efficient methods

DB Application Performance

Application programmer choices that affect the cost of executing a query …

how queries are expressed is important … Generally speaking:

- a join is faster than subquery, especially if subquery is correlated
- avoid producing large intermediate tables *then* filtering
- avoid applying functions in where/group-by clauses

We could create *indexes* on tables

- index will speed-up filtering based on the search key attributes
- indexes generally only effective for equality or greater than/less than type search
- indexes have update-time and storage overheads (i.e., indexes could be costly)
  - only useful if filtering (i.e., reading) is much more frequent operations on that table than than updates

Database Query Processing

Whatever you do as a DB application programmer

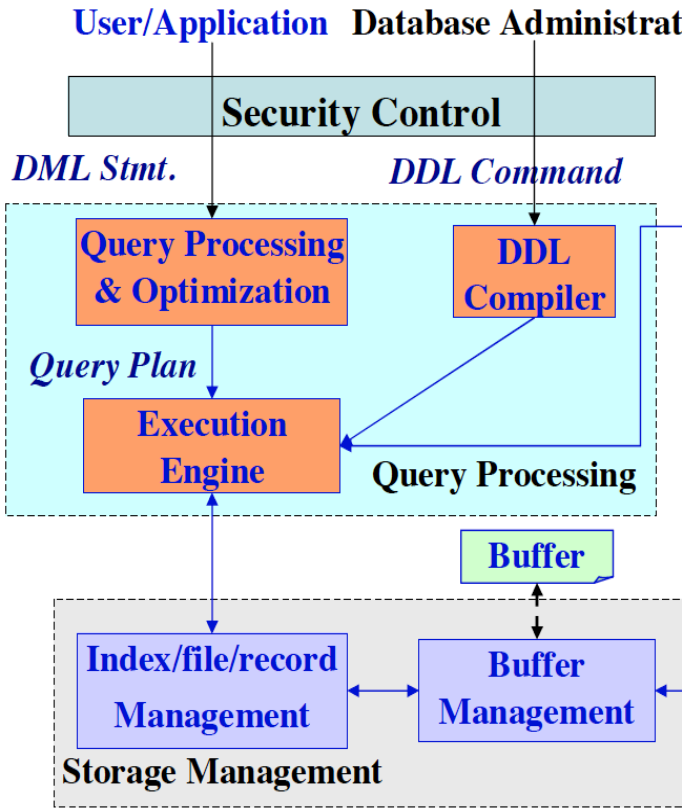
- the DBMS will transform your query to make it execute as efficiently as possible

Transformation is carried out by *query optimiser*

- which assesses possible query execution approaches
- evaluates the likely cost of each approach, chooses cheapest

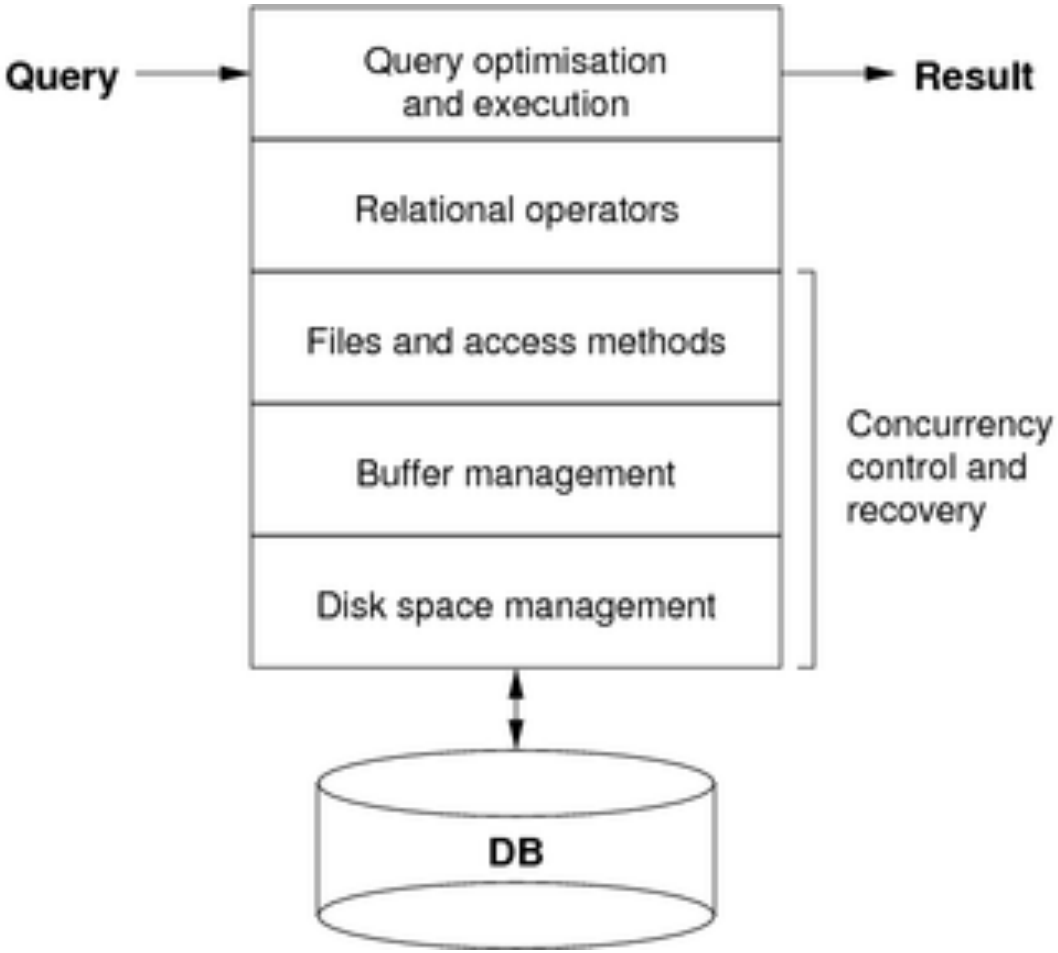
You have no control over the optimisation process

- but choices you make can block certain options, limiting the query optimiser's chance to improve



DB Architecture

Layers in a DB Engine (Ramakrishnan's View)



DB Components

- *File Manager: manages allocation of disk space and data structures*
- *Buffer Manager: manages data transfer between disk and main memory*
- ***Query Optimiser:***
  - ***translates queries into efficient sequence of relational operations***
- Recovery Manager: ensures consistent database state after system failures
- Concurrency Manager: controls concurrent access to database
- Integrity Manager: verifies integrity constraints and user privileges

Database Query Processing

Example: query to find “sales” people earning more than $50K

```
select name from Employee
where  salary > 50000 and
       empid in (select empid from WorksIn
                 where  dept = 'Sales')
```

A query optimiser might use the strategy (roughly …)

```
SalesEmps = (select empid from WorksIn where dept='Sales')
foreach e in Employee {
    if (e.empid in SalesEmps && e.salary > 50000)
        add e to result set
}
```

Needs to examine *all* employees, even if not in Sales

Database Query Processing

A different expression of the same query:

```
select name
from   Employee join WorksIn using (empid)
where  Employee.salary > 50000 and
       WorksIn.dept = 'Sales'
```

A different expression of the same query:

```
SalesEmps = (select * from WorksIn where dept='Sales')
foreach e in (Employee join SalesEmps) {// join on (empid) -> smaller #tuples
    if (e.salary > 50000)
        add e to result set
}
```

Only examines Sales employees, and uses a simpler test

Database Query Processing

A subquery … especially correlated subquery:

```
select name from Employee e
where  salary > 50000 and
       'Sales' in (select dept from WorksIn where empid=e.id)
```

A query optimiser would be forced to use the strategy:

```
foreach e in Employee {
    Depts = (select dept from WorksIn where empid=e.empid)
    if ('Sales' in Depts && e.salary > 50000)
        add e to result set
}
```

Needs to run a query for *every* employee ...

Query Processing

A *query* in SQL:

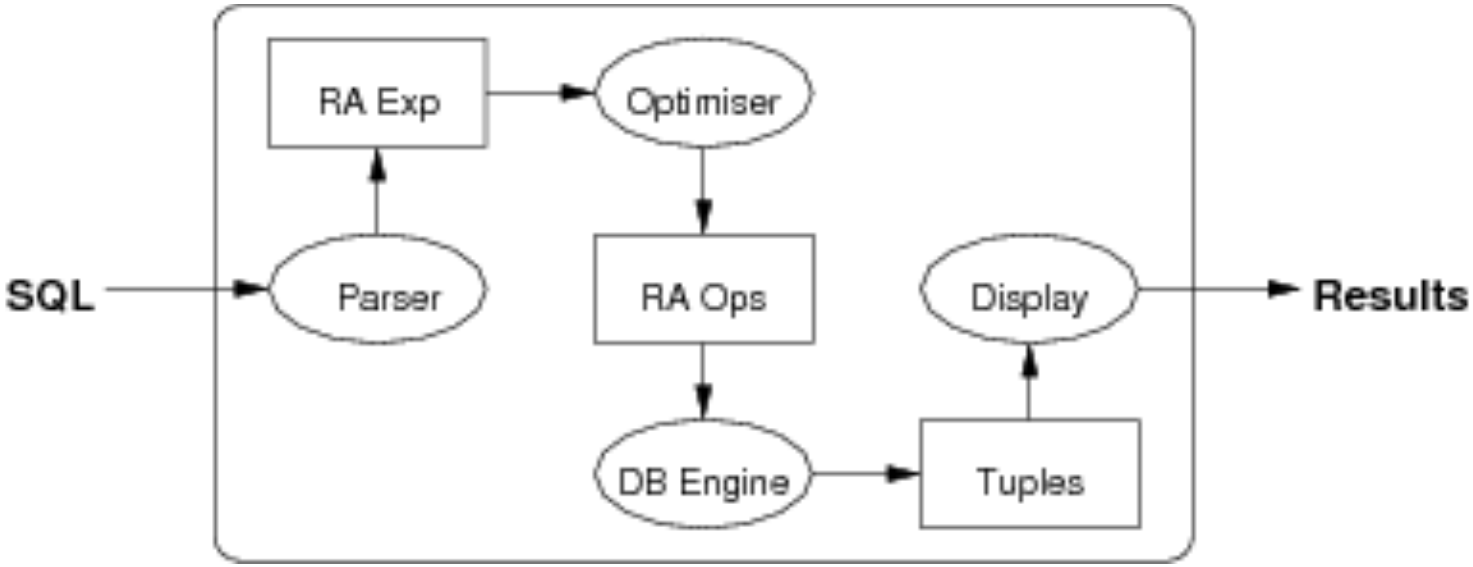
- states *what* answers are required
- says little about *how* they should be computed

A *query evaluator & optimiser* :

- takes a declarative description of the *query* in SQL
- parses the query into a *relational algebra* expression
- determines a *plan* for answering the query
- *executes* the plan via the database engine

Query Processing

mapping SQL to relational algebra (RA)



RA Expressions → Optimiser → concrete RA operations
(e.g., JOIN on empid) (e.g., HASH JOIN on empid)

Mapping SQL to RA expression

A naive translation scheme from SQL to relational algebra:

- `SELECT` clause → projection
- `FROM` clause → cross-product
- `WHERE` clause → selection

Example:

```
select s.name, e.course
from   Student s, Enrolment e
where  s.id = e.student and e.mark > 50;
```

is translated to

Project [name,course] (Select [id=student ∧ mark>50] (Student × Enrolment))

Mapping SQL to RA Expression

A better translation scheme would be something like:

- `SELECT` clause → projection
- `WHERE` clause on single reln → selection
- `WHERE` clause on two relns *R* → join

Example:

```
select s.name, e.course
from   Student s, Enrolment e
where  s.id = e.student and e.mark > 50;
```

is translated to

Project [name,course] (Select [mark>50] (Join [id=student] (Student, Enrolment)))

Mapping SQL to RA Expression

- Mapping other SQL syntax to (extended) RA operations ...
- Aggregation operators (e.g. MAX, SUM, ...):
  - add new operators to extend RA (e.g. *max(Project[age](..))*)
- Duplicate elimination (DISTINCT)
  - incorporate into projection operator (e.g. *Project'*)
- Grouping (GROUP-BY, HAVING)
  - add new operators to extend RA (e.g. *GroupBy, GroupSelect*)
- Sorting (ORDER-BY)
  - add *sort* operator to extend RA

Mapping Example

The query: *Courses with more than 100 students in them?*

Can be expressed in SQL as

```
select   distinct s.code
from     Course c, Subject s, Enrolment e
where    c.id = e.course and c.subject = s.id
group by s.id
having   count(*) > 100;
```

and might be compiled to

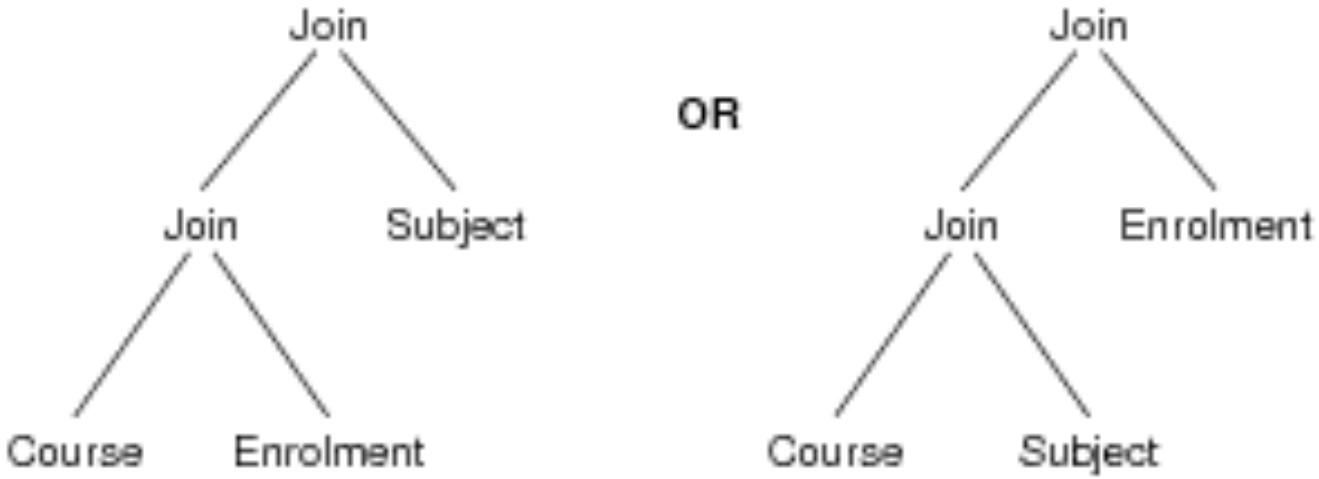
Result = Project'[s.code](GroupSelect[size>100](GroupBy[id] (JoinRes)))

where

JoinRes = Join[s.id=c.subject] (Subject, Join[id=course](Course, Enrolment))

Query Evaluation

The *Join* operations could be done (at least) two different ways:



Which is better? ... The query optimiser works this out.

Note: for a join involving *N* tables, there are *O(N!)* possible trees to consider ...

Query Evaluation

- The order of operations is important.
  - e.g., foreach employee vs. foreach (employee join SalesEmp)
- Equally important is the choice of concrete operations:
  - each RA operator in an RA expression has several implementation methods
    - *Note: understanding the choices are out of the scope of the course …*
  - DBMSs typically provide a range of choices
  - each implementation is effective under certain conditions, not in others …
- The DBMS query optimiser needs to
  - choose concrete operations for each RA operation in query
  - by analysing the cost of potential concrete operations

Database Engine Operations

One view of DB engine - "relational algebra virtual machine":

selection (σ)	projection (π)	join ($\bowtie$, $\times$)
union ($\cup$)	intersection ($\cap$)	difference (-)
sort	insert	delete

For each of these operations:

- various data structures and algorithms are available
- DBMSs may provide only one, or may provide a choice
- we need to be able to estimate the cost of each method

Cost analysis requires a model of DBMS internal implementation details ...

Query Optimisation Problem

Given:

a query *Q*, a database *D*, a database "engine" *E*

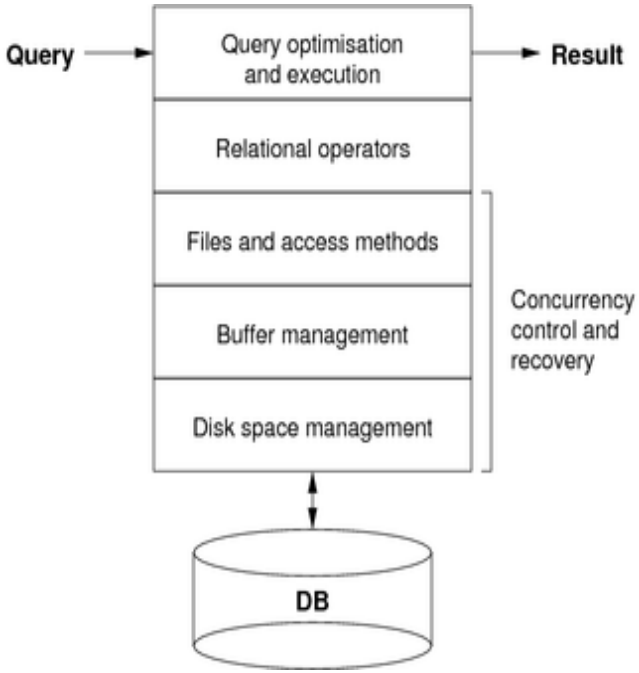
Determine a sequence of relational algebra operations that:

- produces the answer to *Q* in *D*
- executes *Q* efficiently on *E* (minimal I/O)

The term "query optimisation" is a little misleading:

- not just for queries (e.g. also updates)
- not necessarily optimal ("reasonably efficient")

(Finding the *optimal* query is NP-hard; the cost of finding it may be higher than the query cost).



Query Optimisation Problem

The query optimiser start with an RA expression, then

- generates a set of equivalent expressions
- generates possible execution plans for each
- estimates cost of each plan, chooses cheapest

The cost of evaluating a query is determined by:

- size of relations (database relations and temporary relations)
- access mechanisms (indexing, hashing, sorting, join algorithms)
- size/number of main memory buffers (and replacement strategy)

Analysis of costs involves *estimating*:

- the size of intermediate results
- then, based on this, cost of disk storage accesses (i.e., I/O - page read/write)

Query Optimisation Problem

An *execution plan* is a sequence of relational operations (see last two slides for examples)

Consider execution plans for: $\sigma_c (R \bowtie_d S \bowtie_e T)$

```
tmp1   :=  hash_join[d](R,S)
tmp2   :=  sort_merge_join[e](tmp1,T)
result :=  binary_search[c](tmp2)
```

or

```
tmp1   :=  sort_merge_join[e](S,T)
tmp2   :=  hash_join[d](R,tmp1)
result :=  linear_search[c](tmp2)
```

or

```
tmp1   :=  btree_search[c](R)
tmp2   :=  hash_join[d](tmp1,S)
result :=  sort_merge_join[e](tmp2)
```

All produce same result, but have different costs.

Performance Tuning

Schema design:

- devise data structures to *represent application information*

Performance tuning:

- devise data structures to *achieve good performance*

Good performance may involve any/all of:

- making applications run faster
- lowering response time of queries/transactions
- improving overall transaction throughput

Performance Tuning

Tuning requires us to consider the following:

- **which queries and transactions will be used?**

  (e.g. check balance for payment, display recent transaction history)

- **how frequently does each query/transaction occur?**

  (e.g. 99% of transactions are EFTPOS payments; 1% are print balance)

- **are there time constraints on queries/transactions?**

  (e.g. payment at EFTPOS terminals must be approved within 7 seconds)

- **are there uniqueness constraints on any attributes?**

  (therefore, define index on attributes to speed up insertion uniqueness check)

- **how frequently do updates occur?**

  (indexes slow down updates, because must update table *and* index)

Performance Tuning

Performance can be considered at two times:

during schema design

- typically towards the end of schema design process
- requires schema transformations such as denormalisation

after schema design

- requires adding extra data structures such as indexes

Denormalisation

Normalisation structures data to minimise storage redundancy.

- achieves this by "breaking up" the data into logical chunks
- requires minimal "maintenance" to ensure data consistency
  - (i.e., removes update/insert anomalies)

Problem: queries that need to put data back together.

- need to use a (potentially expensive) join operation
- if an expensive join is frequent, system performance suffers

Solution: store some data redundantly

- benefit: queries needing expensive join are now cheap
- trade-off: extra maintenance effort to maintain consistency
- worthwhile if joins are frequent and updates are rare

Denormalisation

```
ass2=# \d courses;
                Table "public.courses"
  Column  |   Type    | Collation | Nullable | Default
----------+-----------+-----------+----------+---------
 id       | integer   |           | not null |
 subject  | integer   |           | not null |
 term     | integer   |           | not null |
 homepage | urlstring |           |          |
```

Example: Courses = Course ⋈ Subject ⋈ Term

Say we frequently need to refer to course "standard" name as e.g., COMP93112020T2

- add extra courseName column into Course table
- cost: trigger before insert on Course to construct name
- trade-off likely to be worthwhile: Course insertions infrequent

```
-- can now replace a query like:
select s.code||t.year||t.sess, e.grade, e.mark
from   Course c, CourseEnrolment e, Subject s, Term t
where  e.course = c.id and c.subject = s.id and c.term = t.id


-- by a query like:
select c.courseName, e.grade, e.mark
from   Course c, CourseEnrolment e
where  e.course = c.id
```

Indexes

Indexes provide efficient content-based access to tuples (i.e., through search keys).

Can build indexes on any (combination of) attributes.

Defining indexes (syntax):

CREATE INDEX index_*name* ON *table_name* ($attr_1$, $attr_2$, ...)

```
e.g., CREATE INDEX idx_address_phone ON address(phone);
```

CREATE INDEX also allows us to specify

- that the index is on UNIQUE values
- an access method (USING btree, hash, rtree, or gist)

```
e.g., CREATE INDEX idx_address_phone ON address USING hash (phone);
```

Indexes

Indexes can make a huge difference to query processing cost.

On the other hand, they introduce overheads (storage, updates).

Creating indexes to maximise performance benefits:

- apply to attributes used in equality, greater/less-than conditions, e.g.

```
select * from Employee where id = 12345

select * from Employee where age > 60

select * from Employee where salary between 10000 and 20000
```

- but only in queries that are frequently used
- and on tables that are not updated frequently

Indexes

Considerations in applying indexes:

- is an attribute used in frequent or expensive queries? (i.e., is it worth it?)
- should we create an index on a collection of attributes?
  (yes, if the collection is used in a frequent/expensive query)
- can we exploit a clustered index? (only one per table)
- should we use B-tree or Hash index?

```
-- use hashing for (unique) attributes in equality tests, e.g.
select * from Employee where id = 12345


-- use B-tree for attributes in range tests, e.g.
select * from Employee where age > 60
```

Query Tuning

Sometimes, a query can be re-phrased to affect performance:

- by helping the optimiser to make use of indexes
- by avoiding (unnecessary) operations that are expensive

Examples which *may* prevent optimiser from using indexes:

```
select name from Employee where salary/365 > 10.0
       -- fix by re-phrasing condition to (salary > 3650)
select name from Employee where name like '%ith%'
select name from Employee where birthday is null
       -- above two are difficult to "fix"
select name from Employee
where  dept in (select id from Dept where ...)
       -- fix by using Employee join Dept on (e.dept=d.id)
```

Query Tuning

Other factors to consider in query tuning:

- select distinct requires a sort; is distinct necessary?
- if multiple join conditions are available ...
  choose join attributes that are indexed, avoid joins on strings

```
select ... Employee join Customer on (s.name = p.name)
vs
select ... Employee join Customer on (s.ssn = p.ssn)
```

- sometimes "OR" in condition prevents index from being used ...
  replace the or condition by a union of non-or clauses

```
select name from Employee where dept=1 or dept=2
vs
(select name from Employee where dept=1)
union
(select name from Employee where dept=2)
```

PostgreSQL Query Tuning

PostgreSQL provides the **explain** statement to

- give a representation of the query execution plan
- with information that may help to tune query performance

Usage:

`EXPLAIN [ANALYZE]` *`Query`*

Without `ANALYZE`, `EXPLAIN` shows plan with estimated costs.

With `ANALYZE`, `EXPLAIN` executes query and prints real costs.

- Note that runtimes may show considerable variation due to buffering.

EXPLAIN Examples

Example: Select on indexed attribute

```
ass2=# explain select * from student where id=100250;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using student_pkey on student  (cost=0.00..5.94 rows=1 width=17)
   Index Cond: (id = 100250)

ass2=# explain analyze select * from student where id=100250;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using student_pkey on student  (cost=0.00..5.94 rows=1 width=17)
                                  (actual time=31.209..31.212 rows=1 loops=1)
   Index Cond: (id = 100250)
 Total runtime: 31.252 ms
```

EXPLAIN Examples

Example: Select on non-indexed attribute

```
ass2=# explain select * from student where stype='local';
                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on student  (cost=0.00..70.33 rows=18 width=17)
   Filter: ((stype)::text = 'local'::text)

ass2=# explain analyze select * from student where stype='local';
                                QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on student  (cost=0.00..70.33 rows=18 width=17)
             (actual time=0.061..4.784 rows=2512 loops=1)
   Filter: ((stype)::text = 'local'::text)
 Total runtime: 7.554 ms
```

EXPLAIN Examples

Example: Join on a primary key (indexed) attribute

```
ass2=# explain
ass2-# select s.sid,p.name from Student s, Person p where s.id=p.id;
                                QUERY PLAN
-------------------------------------------------------------------------
 Hash Join  (cost=70.33..305.86 rows=3626 width=52)
   Hash Cond: ("outer".id = "inner".id)
   ->  Seq Scan on person p  (cost=0.00..153.01 rows=3701 width=52)
   ->  Hash  (cost=61.26..61.26 rows=3626 width=8)
         ->  Seq Scan on student s  (cost=0.00..61.26 rows=3626 width=8)
```

EXPLAIN Examples

Join on a primary key (indexed) attribute:

```
ass2=# explain anaylze
ass2-# select s.sid,p.name from Student s, Person p where s.id=p.id;
                                QUERY PLAN
-------------------------------------------------------------------------
 Hash Join  (cost=70.33..305.86 rows=3626 width=52)
            (actual time=11.680..28.242 rows=3626 loops=1)
   Hash Cond: ("outer".id = "inner".id)
   ->  Seq Scan on person p  (cost=0.00..153.01 rows=3701 width=52)
                        (actual time=0.039..5.976 rows=3701 loops=1)
   ->  Hash  (cost=61.26..61.26 rows=3626 width=8)
             (actual time=11.615..11.615 rows=3626 loops=1)
         ->  Seq Scan on student s  (cost=0.00..61.26 rows=3626 width=8)
                               (actual time=0.005..5.731 rows=3626 loops=1)
 Total runtime: 32.374 ms
```