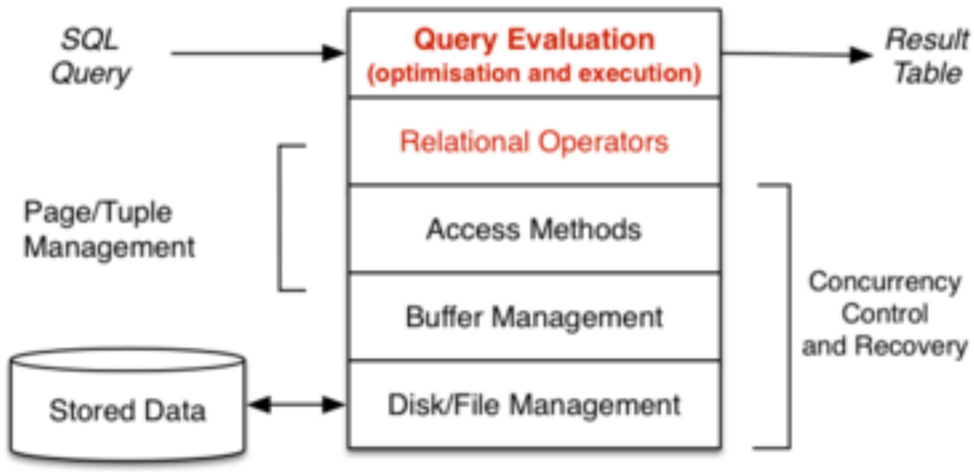
# Query Processing

## Query Processing Overview

(Translation, Optimisation, Execution)

## Query Evaluation






### ... Query Evaluation



The most common form of interaction with a DBMS involves:

- supplying some input in the form of an SQL query
- getting back the results as a set of tuples


Overall approach clearly applies to `select` ... but also to `delete` and `update`.

### ... Query Evaluation



A *query* in SQL:

- states *what* answers are required (declaratively)
- does not say *how* they should be computed

A *query evaluator/processor* :

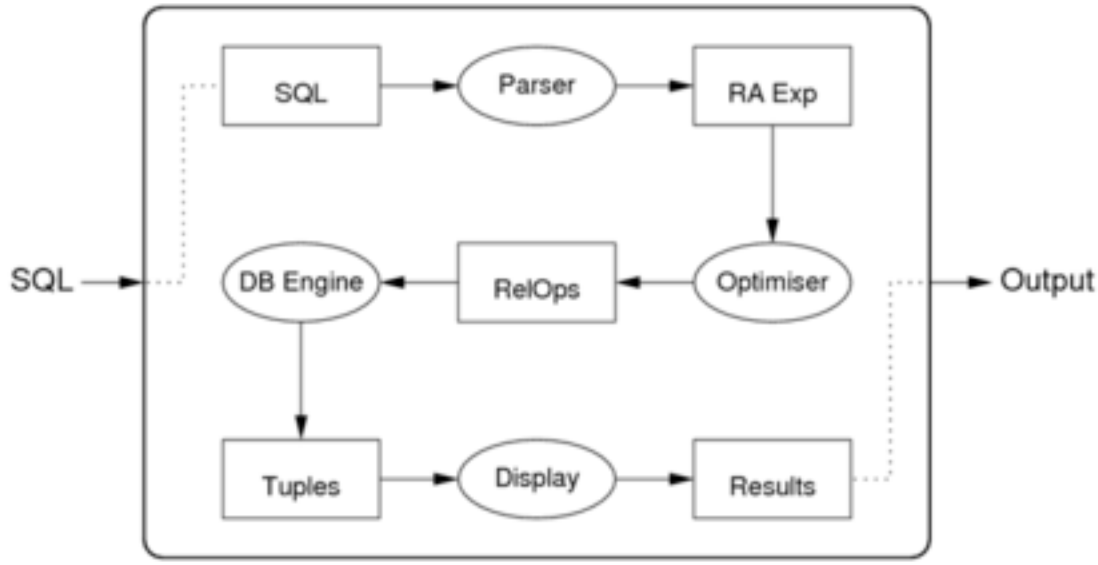
- takes declarative description of query (in SQL)
- determines plan for answering query (expressed as DBMS ops)
- executes plan via DBMS engine (to produce result tuples)

Typically: translate → plan → execute ... in a pipeline.

Some DBMSs can save query plans for later re-use
(because the planning step is potentially quite expensive).

### ... Query Evaluation

Internals of the query evaluation "black-box":




---



### ... Query Evaluation

Three phases of query evaluation:

1. parsing/compilation
   - input: SQL query, catalog
   - using: parsing techniques, mapping rules
   - output: relational algebra (RA) expression
2. query optimisation
   - input: RA expression, DB statistics
   - using: cost models, search strategy
   - output: query execution plan (DB engine ops)
3. query execution
   - input: query execution plan
   - using: database engine
   - output: tuples that satisfy query

We ignore the "display" step; simply maps result tuples into appropriate format

---



## Intermediate Representations

SQL query text is not easy to manipulate/transform.

Need a query representation formalism that ...

- is powerful enough to express all queries
- has a well-defined, formal basis
- simplifies the query transformation process

*Relational algebra* (RA) expressions:

- are easy to transform and have a procedural interpretation

Thus, RA typically used as "target language" for SQL compilation.

---



### ... Intermediate Representations

In principle, could base RA engine on $\{ \sigma, \pi, \cup, -, \times \}$ (completeness).

In practice, having only these operators makes execution inefficient.

The standard set of RA operations in a DBMS includes:

- filtering and combining (*select*, *project*, *join* (inner,outer))
- set operations (*union*, *intersection*, *difference*)

Other operations typically provided in extended RA engines:

- grouping (*group by*) and group-based selection (*having*)
- aggregates (*count*, *sum*, *avg*, *max*, *min*)
- sorting (*order by*), *uniq* (*distinct*, sets)

RA *rename* operator is "hidden" in table/tuple representation.

---

### ... Intermediate Representations



In practice, DBMSs provide several versions of each RA operation.

For example:

- several "versions" of selection ($\sigma$) are available
- each version is effective for a particular kind of selection, e.g

```
select * from R where id = 100  -- hashing
select * from S                 -- Btree index
where age > 18 and age < 35
select * from T                 -- grid file
where a = 1 and b = 'a' and c = 1.4
```

Similarly, $\pi$ and $\bowtie$ have versions to match specific query types.

---

### ... Intermediate Representations



We call these specialised version of RA operations *RelOps*.

One major task of the query processor:

- given a set of RA operations to be executed
- find a combination of RelOps to do this efficiently

Requires the query translator/optimiser to consider

- information about relations (e.g. sizes, primary keys, ...)
- information about operations (e.g. select reduces size)

RelOps are realised at execution time

- as a collection of inter-communicating *nodes*
- communicating either via pipelines or temporary relations

---

### ... Intermediate Representations



Terminology variations ...

Relational algebra expression of SQL query

- intermediate query representation
- logical query plan

Execution plan as collection of RelOps

- query evaluation plan

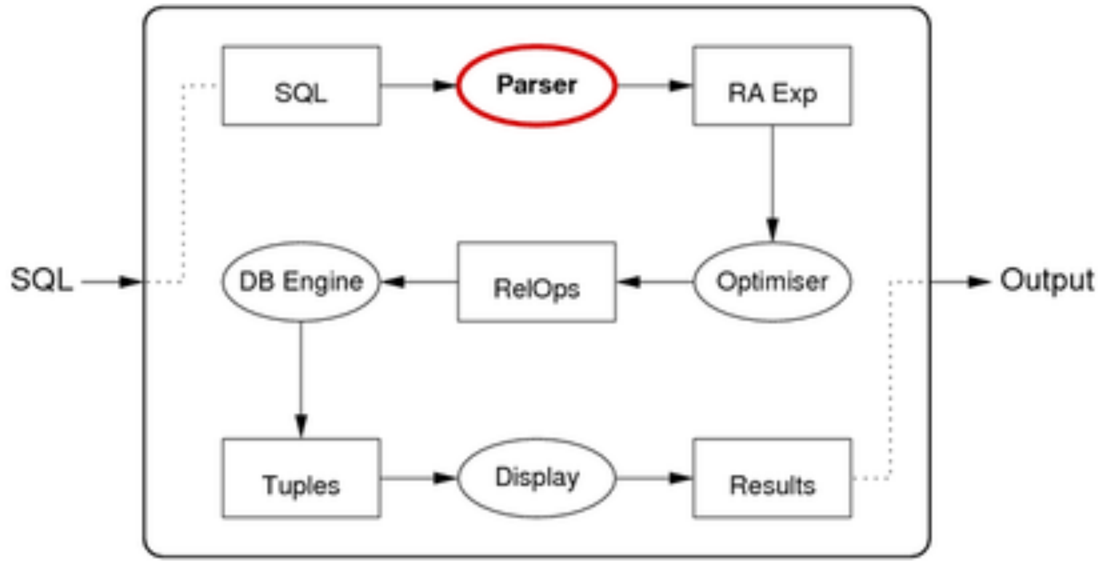
* query execution plan
* physical query plan

# Query Translation

### ... Intermediate Representations



Query translation: SQL statement text → RA expression




## Query Translation



Translation step is a mapping

* from the text of an SQL statement
* to a useful internal representation for optimisation

Standard internal representation is relational algebra (RA).

Mapping from SQL to RA may including some optimisations.

Mapping processes: lexer/parser, mapping rules, rewriting rules.

Example:

```
SQL: select name from Students where id=7654321;
-- is translated to
RA:  Proj[name](Sel[id=7654321]Students)
```

## Parsing SQL



Parsing task is similar to that for programming languages.

Language elements:

* keywords: `create, select, from, where,` ...
* identifiers: `Students, name, id, CoursCode,` ...
* operators: `+, -, =, <, >, AND, OR, NOT, IN,` ...
* constants: `'a string', 123, 19.99, '01-jan-1970'`

One difference to parsing PLs:

* PLs define objects as part of program definition (e.g. in `*.h`)
* SQL references tables/types/functions stored in the DBMS

Therefore, SQL parser needs access to catalog for definitions.

---

### ... Parsing SQL

16/154

PostgreSQL dialect of SQL ...

- large set of keywords (> 700 of them) (e.g. `select`)
- parser is implemented via lex/yacc (**`src/backend/parser`**)
- handles multiple languages (i.e. Unicode strings)
- maps all identifiers to lower-case (A-Z → a-z)
- needs to handle user-extendable operator set
- makes extensive use of catalog (**`src/backend/catalog`**)
  - **`pg_class`** hold pre- and user-defined tables
  - **`pg_type`** hold pre- and user-defined types
  - **`pg_proc`** hold pre- and user-defined functions

---

## Catalog and Parsing

17/154

Consider the following simple University schema:

```
Staff(id, name, position, office, phone, ...)
Students(id, name, birthday, degree, avg, ...)
Subjects(id, title, prereqs, syllabus, ...)
Enrolments(sid, subj, session, mark, ...)
```

DBMS catalog stores following kinds of information:

- `Staff`, `Students`, ... are relations owned by some user
- `id`, `name` ... are fields of the `Staff` relation
- `id` has type `INTEGER` and contains a unique value
- there are 1000 tuples in `Staff`, 20000 tuples in `Students`, ...
- `Enrolments.sid` is a foreign key referencing `Students.id`
- a `Student` is associated to *<40* `Subjects` via `Enrolments`(?)

---

### ... Catalog and Parsing

18/154

The schema/type information is used for

- checking that the named tables and attributes exist
- resolving attribute references (e.g. is `id` from `Students` or `Subjects`?)
- checking that attrs are used appropriately (e.g. not `id='John'`)

The statistical information is used for

- choosing appropriate operators for query execution plans

Examples:

- a query with exactly one solution:
  `select * from Students where id=12345`
- a query with thousands of solutions:
  `select * from Students where degree='BSc'`

---

## Query Blocks

19/154

A *query block* is an SQL query with

- no nesting
- exactly one `SELECT`, `FROM` clause
- at most one `WHERE`, `GROUP-BY`, `HAVING` clause

Query optimisers typically deal with one query block at a time ...

⇒ SQL compilers need to decompose queries into blocks

Interesting kinds of blocks:

- non-correlated query returning a set of tuples/values
- non-correlated query returning a single result (treat as constant)
- correlated sub-query (query changes for each outer tuple)

---

### ... Query Blocks



Consider the following example query:

```
SELECT s.name FROM Students s
WHERE  s.avg = (SELECT MAX(avg) FROM Students)
```

which consists of two blocks:

```
Block1: SELECT MAX(avg) FROM Students
Block2: SELECT s.name FROM Students s
        WHERE  s.avg = <<Block1>>>
```

Query processor arranges execution of each block separately and transfers result of Block1 to Block2.

---

## Mapping SQL to Relational Algebra



A naive query compiler might use the following translation scheme:

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

Example:

```
SELECT s.name, e.subj
FROM   Students s, Enrolments e
WHERE  s.id = e.sid AND e.mark > 50;
```

is translated to

$$\pi_{s.name,e.subj}( \sigma_{s.id=e.sid \wedge e.mark>50} ( Students \times Enrolments ) )$$

---

### ... Mapping SQL to Relational Algebra



A better translation scheme would be something like:

- `SELECT` clause → projection
- `WHERE` clause on single relation → selection
- `WHERE` clause on two relations → join

Example:

```
SELECT s.name, e.subj
FROM   Students s, Enrolments e
WHERE  s.id = e.sid AND e.mark > 50;
```

is translated to

$\pi_{s.name,e.subj}( \sigma_{e.mark>50} ( Students \Join_{s.id=e.sid} Enrolments ) ) )$

---

### ... Mapping SQL to Relational Algebra

23/154

In fact, many SQL compilers ...

- produce the cross-product version as an intermediate result
- map it into the join version by applying rewriting rules

This is one instance of a general query translation process

- expression rewriting via algebraic laws on RA expressions

Aim of rewriting: convert a given RA expression

- into an *equivalent* RA expression
- that is guaranteed/likely to be *more efficient*

(Rewriting is discussed later)

---

## Mapping Rules

24/154

Mapping from an SQL query to an RA expression requires:

- a collection of *templates* for particular kinds of queries
- a matching process to ...
  - determe what kind of query we have (i.e. choose a template)
  - bind components of actual query to slots in the template
- mapping rules to ...
  - convert the matched query into relational algebra
  - filling slots in RA expression from matched components

May need to use multiple templates to map whole SQL statement.

---

### ... Mapping Rules

25/154

**Projection:**

`SELECT` $f_1, f_2, \ldots f_n$ `FROM` ...

$\Rightarrow$ $Project_{[f_1,\, f_2,\, \ldots\, f_n]}(\ldots)$

SQL projection extends RA projection with renaming and assignment:

`SELECT` *a+b* `AS` *x*, *c* `AS` *y* `FROM` *R* ...

$\Rightarrow$ $Project_{[x \leftarrow a+b,\ y \leftarrow c]}(R)$

---

### ... Mapping Rules

26/154

**Join:** (e.g. on *R(a,b,c,d)* and *S(c,d,e)*)

`SELECT` ... `FROM` ... *R, S* ... `WHERE` ... *R.a op S.e* ... , or

`SELECT` ... `FROM` ... *R* `JOIN` *S* `ON` *(R.a op S.e)* ... `WHERE` ...

$\Rightarrow$ $Join_{[R.a\ op\ S.e]}(R,S)$

`SELECT ... FROM ... R NATURAL JOIN S`, or

`SELECT ... FROM ... R JOIN S USING (c,d) ... WHERE ...`

⇒ $Proj_{[a,b,e]}(Join_{[R.c=S.c \wedge R.d=S.d]}(R,S))$

---

### ... Mapping Rules



**Selection:**

`SELECT ... FROM ... R ... WHERE ... R.f op val ...`

⇒ $Select_{[R.f\ op\ val]}(R)$

`SELECT ... FROM ... R ... WHERE ... Cond1,R AND Cond2,R ...`

⇒ $Select_{[Cond_{1,R} \wedge Cond_{2,R}]}(R)$
or
⇒ $Select_{[Cond_{1,R}]}(Select_{[Cond_{2,R}]}(R))$
or
⇒ $Select_{[Cond_{2,R}]}(Select_{[Cond_{1,R}]}(R))$

---

### ... Mapping Rules



Aggregation operators (e.g. `MAX`, `SUM`, ...):

- add as new operators in extended RA

  e.g. `SELECT MAX(age) FROM ...` ⇒ $max(Project_{[age]}(...))$

- incorporate into projection operator

  e.g. `SELECT MAX(age) FROM ...` ⇒ $Project_{[age]}(max,...)$

- add new projection operators

  e.g. `SELECT MAX(age) FROM ...` ⇒ $ProjectMax_{[age]}(...)$

---

### ... Mapping Rules



Sorting (`ORDER BY`):

- add *Sort* operator into extended RA

Duplicate elimination (`DISTINCT`):

- add *Uniq* operator into extended RA (e.g. *Uniq(Project(...))*)
- or, extend RA ops with *uniq* parameter (e.g. *Project(uniq,...)*)

Grouping (`GROUP BY`, `HAVING`):

- add operators into extended RA (e.g. *GroupBy, GroupSelect* )

---

### ... Mapping Rules



View definitions produce:

- mapping of view statement to RA expression
- association between view name and RA expression

Example: assuming *Employee(id,name,birthdate,salary)*

```
create view OldEmps as
select * from Employees
where birthdate < '01-01-1960';
```

yields

$OldEmps = Select_{[birthdate<'01-01-1960']}(Employees)$

---

### ... Mapping Rules



General case:

`CREATE VIEW` *V* `AS` *SQLstatement*

$\Rightarrow \quad V = mappingOf(SQLstatement)$

Special case: views with attribute renaming:

`CREATE VIEW` *V(a,b,c)* `AS SELECT` *h,j,k* `FROM` *R* `WHERE` *C ...*

$\Rightarrow \quad V = Proj_{[a \leftarrow h,\ b \leftarrow j,\ c \leftarrow k]}(Select_{[C]}(R))$

---

### ... Mapping Rules



Views used in queries:

- references to views are replaced by the view definition
- introduces a new subexpression into the RA expression
- may require renaming of some attributes

Example:

```
select name from OldEmps;  -- using OldEmps as defined above
```

$\Rightarrow \quad Proj_{name}(mappingOf(\texttt{OldEmps}))$

$\Rightarrow \quad Proj_{name}(Select_{[birthdate<'01-01-1960']}(Employees))$

---

## Mapping Example



The query

> *List the names of all subjects 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;
```

---

### ... Mapping Example



In the SQL compiler, the query

```
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;
```

might be translated to the relational algebra expression

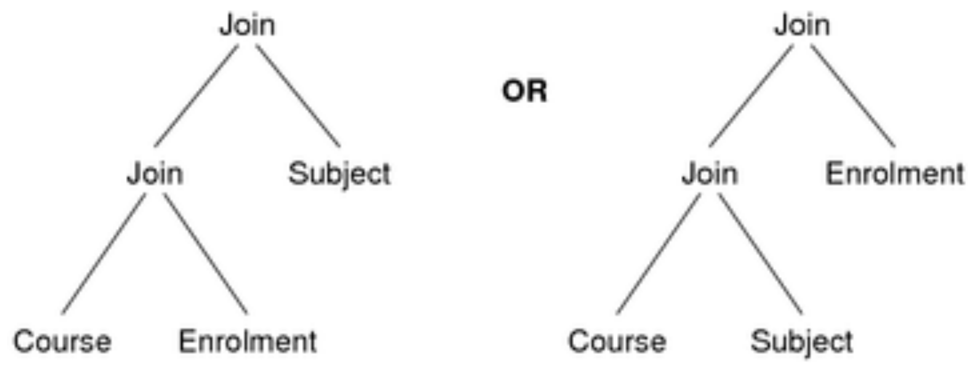
$$Uniq(Project_{[code]}(\ GroupSelect_{[groupSize>100]}(\ GroupBy_{[id]}\ (\ Enrolment \bowtie Course \bowtie Subjects\ ))))$$

---

### ... Mapping Example



The join operations could be done in two different ways:




The query optimiser determines which has lower cost.

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

---

## Expression Rewriting Rules



Since RA is a well-defined formal system

- there exist many algebraic laws on RA expressions
- which can be used as a basis for expression rewriting
- in order to produce equivalent (more-efficient) expressions

Expression transformation based on such rules can be used

- to simplify/improve SQL→RA mapping results
- to generate new plan variations during query optimisation

---

## Relational Algebra Laws



Commutative and Associative Laws:

- $R \times S \leftrightarrow S \times R$, $(R \times S) \times T \leftrightarrow R \times (S \times T)$
- $R \bowtie S \leftrightarrow S \bowtie R$, $(R \bowtie S) \bowtie T \leftrightarrow R \bowtie (S \bowtie T)$ (natural join)
- $R \cup S \leftrightarrow S \cup R$, $(R \cup S) \cup T \leftrightarrow R \cup (S \cup T)$
- $R \cap S \leftrightarrow S \cap R$, $(R \cap S) \cap T \leftrightarrow R \cap (S \cap T)$
- $R \bowtie_{Cond} S \leftrightarrow S \bowtie_{Cond} R$ (theta join)

But it is **not** true in general that

- $(R \bowtie_{Cond1} S) \bowtie_{Cond2} T \leftrightarrow R \bowtie_{Cond1} (S \bowtie_{Cond2} T)$

Example: *R(a,b)*, *S(b,c)*, *T(c,d)*, $(R\ Join_{[R.b>S.b]}\ S)\ Join_{[a<d]}\ T$

Cannot rewrite as $R\ Join_{[R.b>S.b]}\ (S\ Join_{[a<d]}\ T)$ because neither *S.a* nor *T.a* exists.

---

### ... Relational Algebra Laws



Selection commutativity (where *c* is a condition):

- $\sigma_c ( \sigma_d (R)) \leftrightarrow \sigma_d ( \sigma_c (R))$

Selection splitting (where *c* and *d* are conditions):

- $\sigma_{c \wedge d}(R) \leftrightarrow \sigma_c ( \sigma_d (R))$
- $\sigma_{c \vee d}(R) \leftrightarrow \sigma_c(R) \cup \sigma_d(R)$ (but only if *R* is a set)

Selection pushing ( $\sigma_c(R\ op\ S)$ ):

- $\sigma_c(R \cup S) \leftrightarrow \sigma_c R \cup \sigma_c S$
  (must be pushed into both arguments of union)
- $\sigma_c(R - S) \leftrightarrow \sigma_c R - S$, $\sigma_c(R - S) \leftrightarrow \sigma_c R - \sigma_c S$
  (must be pushed into left branch of difference, may be pushed to right branch)

---

### ... Relational Algebra Laws



Selection pushing with join, cross-product and intersection ...

If *c* refers only to attributes from *R*:

- $\sigma_c (R \bowtie S) \leftrightarrow \sigma_c(R) \bowtie S$ (similarly for $\times$ and $\cap$)

If *c* refers only to attributes from *S*:

- $\sigma_c (R \bowtie S) \leftrightarrow R \bowtie \sigma_c(S)$ (similarly for $\times$ and $\cap$)

If *c* refers to attributes from both *R* and *S*:

- $\sigma_c (R \bowtie S) \leftrightarrow \sigma_c(R) \bowtie \sigma_c(S)$
- above is always true for $\cap$ (union-compatible)

---

### ... Relational Algebra Laws



Rewrite rules for projection ...

All but last projection can be ignored:

- $\pi_{L1} ( \pi_{L2} ( ... \pi_{Ln} (R))) \leftrightarrow \pi_{L1} (R)$

Projections can be pushed into joins:

- $\pi_L (R \bowtie_c S) \leftrightarrow \pi_L ( \pi_M(R) \bowtie_c \pi_N(S) )$

where

- *M* and *N* must contain all attributes needed for *c*
- *M* and *N* must contain all attributes used in *L* ($L \subset M \cup N$)

---

# Mapping Subqueries



Two varieties of sub-query: (sample schema *R(a,b)*, *S(c,d)*)

- independent: sub-query makes no reference to outer query

```
select * from R
where a in (select c from S where d>5)

select * from R
where a = (select max(c) from S)
```

- correlated: sub-query depends on data from outer query

```
select * from R
where a in (select c from S where d=R.b)
```

Standard strategy: convert to a join.

---

### ... Mapping Subqueries



Example of mapping independent subquery ...

```
select * from R
where a in (select c from S where d>5)
```

is mapped as

$\Rightarrow$ $Sel_{[a \textbf{ in } Proj_{[c]}(Sel_{[d>5]}S)]}(R)$

$\Rightarrow$ $Sel_{[a=c]}(R \times Proj_{[c]}(Sel_{[d>5]}(S)))$

$\Rightarrow$ $R\ \ Join_{[a=c]}\ \ Proj_{[c]}(Sel_{[d>5]}(S))$

---

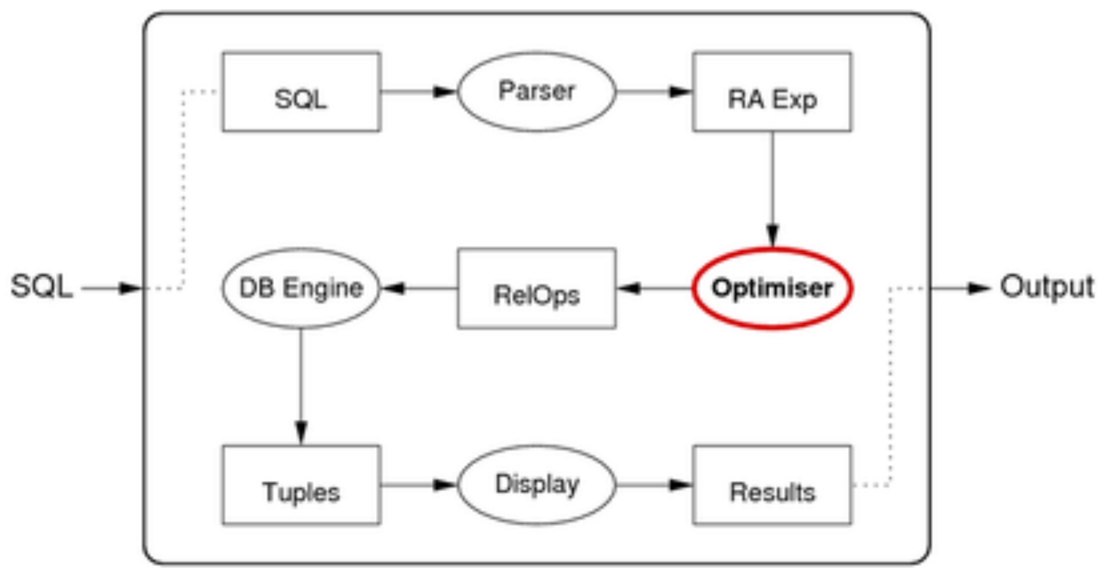
# Query Optimisation

---

# Query Optimisation



Query optimiser: RA expression → efficient evaluation plan




---

### ... Query Optimisation

*Query optimisation* is a critical step in query evaluation.

The query optimiser

- takes a relational algebra expression from SQL compiler
- produces a sequence of RelOps to evaluate the expression
- the query execution plan yields an efficient evaluation

"Optimisation" is necessary because there can be enormous differences in costs between different plans for evaluating a given RA expression.

E.g. $tmp \leftarrow A \times B;\ \ res \leftarrow \sigma_x(tmp)$ vs $res \leftarrow A \Join_x B$

---

# Query Evaluation Example



Example database schema (multi-campus university):

```
Employee(eid, ename, status, city)
Department(dname, city, address)
Subject(sid, sname, syllabus)
Lecture(subj, dept, empl, time)
```

Example database instance statistics:

| **Relation** | *r* | *R* | *C* | *b* |
|---|---|---|---|---|
| `Employee` | 1000 | 100 | 10 | 100 |
| `Department` | 100 | 200 | 5 | 20 |
| `Subject` | 500 | 95 | 10 | 100 |
| `Lecture` | 2000 | 100 | 10 | 200 |

---

### ... Query Evaluation Example



Query: *Which depts in Sydney offer database subjects?*

```
select dname
from   Department D, Subject S, Lecture L
where  D.city = 'Sydney' and S.sname like '%Database%'
       and D.dname = L.dept and S.sid = L.subj
```

Additional information (needed to determine query costs):

- 5 departments are located in Sydney
- 80 subjects are about databases
- 300 lectures are on databases
- 100 lectures are in Sydney
- 3 of these are on databases

---

### ... Query Evaluation Example



Consider total I/O costs for five evaluation strategies.

Assumptions in computing costs:

- database tables are unsorted and have no indexes
- intermediate tables are written to disk and then re-read

These are worst-case scenarios and could be improved by

- having indexes on database tables
- writing intermediate results as e.g. sorted/hashed

---

### ... Query Evaluation Example



Strategy #1 for answering query:

1. *TMP1* ← *Subject × Lecture*
2. *TMP2* ← *TMP1 × Department*
3. *TMP3* ← *Select[check](TMP2)*

   *check = (city='Sydney' & sname='Databases' & dname=dept & sid=subj)*

4. *RESULT* ← *Project[dname](TMP3)*

---

### ... Query Evaluation Example



Costs involved in using strategy #1:

| **Reln** | *r* | *R* | *C* | *b* |
|---|---|---|---|---|
| `TMP1` | 1000000 | 195 | 5 | 20000 |
| `TMP2` | 100000000 | 395 | 2 | 50000000 |
| `TMP3` | 3 | 395 | 2 | 2 |
| `RESULT` | 3 | 100 | 10 | 1 |

Total I/Os
= $Cost_{Step1} + Cost_{Step2} + Cost_{Step3} + Cost_{Step4}$
= $(100+100*200+20000) + (20+20*20000+5*10^6) + (5*10^6+2) + 2$
= 100,440,124

---

### ... Query Evaluation Example



Strategy #2 for answering query:

1. *TMP1* ← *Join[sid=subj](Subject,Lecture)*
2. *TMP2* ← *Join[dept=dname](TMP1,Department)*
3. *TMP3* ← *Select[city='Sydney' & sname='Databases'](TMP2)*
4. *RESULT* ← *Project[dname](TMP3)*

---

### ... Query Evaluation Example



Costs involved in using strategy #2:

| **Reln** | *r* | *R* | *C* | *b* |
|---|---|---|---|---|
| `TMP1` | 2000 | 195 | 5 | 400 |
| `TMP2` | 2000 | 395 | 2 | 1000 |
| `TMP3` | 3 | 395 | 2 | 2 |
| `RESULT` | 3 | 100 | 10 | 1 |

Total I/Os
= $Cost_{Step1} + Cost_{Step2} + Cost_{Step3} + Cost_{Step4}$
= $(100+100*200+400) + (20+20*400+1000) + (1000+2) + 2$
= 30,524

## ... Query Evaluation Example



Strategy #3 for answering query:

1. *TMP1* ← *Join[sid=subj](Subject,Lecture)*
2. *TMP2* ← *Select[sname='Databases'](TMP1)*
3. *TMP3* ← *Join[dept=dname](TMP2,Department)*
4. *TMP4* ← *Select[city='Sydney'](TMP3)*
5. *RESULT* ← *Project[dname](TMP4)*

---

## ... Query Evaluation Example



Costs involved in using strategy #3:

| **Reln** | *r* | *R* | *C* | *b* |
|---|---|---|---|---|
| TMP1 | 2000 | 195 | 5 | 400 |
| TMP2 | 20 | 195 | 5 | 4 |
| TMP3 | 20 | 395 | 2 | 10 |
| TMP4 | 3 | 395 | 2 | 2 |
| RESULT | 3 | 100 | 10 | 1 |

Total I/Os
= $Cost_{Step1} + Cost_{Step2} + Cost_{Step3} + Cost_{Step4} + Cost_{Step5}$
= (100+100*200+400) + (400+4) + (4+4*20+10) + (10+2) + 1
= 21,011

---

## ... Query Evaluation Example



Strategy #4 for answering query:

1. *TMP1* ← *Select[sname='Databases'](Subject)*
2. *TMP2* ← *Select[city='Sydney'](Department)*
3. *TMP3* ← *Join[sid=subj](TMP1,Lecture)*
4. *TMP4* ← *Join[dept=dname](TMP3,TMP2)*
5. *RESULT* ← *project[dname](TMP4)*

---

## ... Query Evaluation Example



Costs involved in using strategy #4:

| **Reln** | *r* | *R* | *C* | *b* |
|---|---|---|---|---|
| TMP1 | 80 | 95 | 5 | 16 |
| TMP2 | 5 | 200 | 5 | 1 |
| TMP3 | 300 | 195 | 5 | 60 |
| TMP4 | 3 | 395 | 2 | 2 |
| RESULT | 3 | 100 | 10 | 1 |

Total I/Os
= $Cost_{Step1} + Cost_{Step2} + Cost_{Step3} + Cost_{Step4} + Cost_{Step5}$
= (100+16) + (20+1) + (16+16*200+60) + (1+1*60+2) + 2
= 3478

### ... Query Evaluation Example



Strategy #5 for answering query:

1. *TMP1* ← *Select[sname='Databases'](Subject)*
2. *TMP2* ← *Select[city='Sydney'](Department)*
3. *TMP3* ← *Join[dept=dname](TMP2,Lecture)*
4. *TMP4* ← *Join[sid=subj](TMP3,TMP1)*
5. *RESULT* ← *project[dname](TMP4)*

---

### ... Query Evaluation Example



Costs involved in using strategy #5:

| **Reln** | *r* | *R* | *C* | *b* |
|---|---|---|---|---|
| TMP1 | 80 | 95 | 5 | 16 |
| TMP2 | 5 | 200 | 5 | 1 |
| TMP3 | 100 | 295 | 3 | 34 |
| TMP4 | 3 | 395 | 2 | 2 |
| RESULT | 3 | 100 | 10 | 1 |

Total I/Os
= $Cost_{Step1} + Cost_{Step2} + Cost_{Step3} + Cost_{Step4} + Cost_{Step5}$
= (100+16) + (20+1) + (1+1*200+34) + (16+16*34+2) + 2
= 936

---

## 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 misnomer:

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

---

### ... Query Optimisation Problem



Why do we not generate optimal query execution plans?

Finding an optimal query plan ...

- requires an exhaustive search of a space of possible plans
- for each possible plan, need to estimate cost (not cheap)

Even for a small query (e.g. 4-5 joins, 4-5 selects)

- the space of possible query plans is very large
  (choices: order of operations, access methods, intermediate results, etc.)
- cost of searching plan space ≅ cost of executing query

Compromise:

- do limited searching of query plan space (guided by heuristics)
- *quickly* choose a *reasonably efficient* execution plan

---

## Cost Models and Analysis

61/154

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*:

- size of intermediate results
- number of secondary storage accesses

---

## Approaches to Optimisation

62/154

Three main classes of techniques developed:

- algebraic (equivalences, rewriting, heuristics)
- physical (execution costs, search-based)
- semantic (application properties, heuristics)

All driven by aim of minimising (or at least reducing) "cost".

Real query optimisers use a combination of algrebraic+physical.

Semantic QO is good idea, but expensive/difficult to implement.

---

## Optimisation Process

63/154

Start with RA tree representation of query, then ...

1. apply algebraic query transformations
   - giving standardised, simpler, more efficient RA tree
2. generate possible access plans (physical)
   - replace each RA operation by specific access method to implement it
   - consider possible orders of join operations (left-deep trees)
3. analyse cost of generated plans and select cheapest

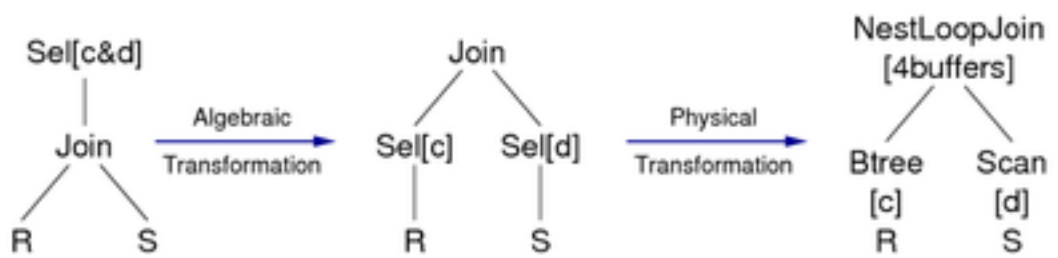
Result: tree of DBMS operations to answer query efficiently.

---

### ... Optimisation Process

64/154

Example of optimisation transformations:




For join, may also consider sort/merge join and hash join.

---

## Algebraic Optimisation

65/154

Make use of algebraic equivalences:

- examine query expression
- search for applicable transformation rules (heuristics)
- generate equivalent (and "better") algebraic expressions

Most commonly used heuristic:

> Apply *Select* and *Project* before *Join*

Rationale: minimises size of intermediate relations.

Can potentially be done during the SQL→RA mapping phase.

Algebraic optimisation cannot assist with finding good join order.

---



## Physical Optimisation

Makes use of execution cost analysis:

- examine query evaluation plan
- determine efficient join sequences
- select access method for each operation (e.g. index for select)
- for distributed DB, select best sites (closest, best bandwidth)
- determine total cost for evaluation plan
- repeat for all possible plans and choose best

Physical optimisation is also called *query evaluation plan generation*.

---



## Semantic Optimisation

Make use of *application*-specific properties:

- functional dependencies
- attributes constraints
- tuple constraints
- database constraints

Can be applied in algebraic or physical optimisation phase.

Basis: exploit meta-data and other semantic info about relations.

(E.g. this field is a primary key, so we know there'll only be one matching tuple)

---



## Stages in Algebraic Optimisation

Start with relational algebra (RA) expression.

1. Standardise
   - construct normal form of expression (boolean algebra laws)
2. Simplify
   - transform to eliminate redundancy (boolean algebra and RA laws)
3. Ameliorate
   - transform to improve efficiency (RA laws and heuristics)

Result: an RA expression equivalent to the input, but more efficient.

---

## Standardisation

Many query optimisers assume RA expression is in a "standard form".

E.g. select conditions may be stated in

- conjunctive normal form
  `(A1 AND ... AND An) OR ... OR (Z1 AND ... AND Zm)`
- disjunctive normal form
  `(A1 OR ... OR An) AND ... AND (Z1 OR ... OR Zm)`

RA expression is first *transformed* into one of these forms.

---



## Simplification

Main aim of simplification is to reduce redundancy.

Makes use of tranformation rules based on laws of boolean algebra.

```
A AND B           ↔  B AND A

A OR (B AND C)    ↔  (A OR B) AND (A OR C)

A AND A           ↔  A

A OR NOT(A)       ↔  True

A AND NOT(A)      ↔  False

A OR (A AND B)    ↔  A

NOT(A AND B)      ↔  NOT(A) OR NOT(B)

NOT(NOT(A))       ↔  A
```

(Programmers don't produce redundant expressions; but much SQL is produced by programs)

---



## Simplification Example

Consider the query:

```
select Title from Employee
where  (not (Title = 'Programmer')
        and (Title = 'Programmer'
             or Title = 'Analyst')
        and not (Title = 'Analyst'))
       or Name = 'Joe'
```

Denote the atomic conditions as follows:

$P$ = `(Title = 'Programmer')`
$A$ = `(Title = 'Analyst')`
$J$ = `(Name = 'Joe')`

---



### ... Simplification Example

Selection condition can then be simplified via:

$$
\begin{aligned}
\text{Cond} &= (P \wedge (P \vee A) \wedge A) \vee J \\
&= ((P \wedge A) \wedge (P \vee A)) \vee J \\
&= ((P \vee A) \wedge (P \vee A)) \vee J \\
&= False \vee J \\
&= J
\end{aligned}
$$

Giving the equivalent simplified query:

```
select Title from Employee where Name = "Joe"
```

---



## Amelioration

Aim of amelioration is to improve efficiency.

Transform query to equivalent form which is known to be more efficient.

Achieve this via:

- relational algebra laws/transformations
- heuristics to choose which to apply

Often describe RA expression as a tree and RA transformations as tree transformations.

---



## Amelioration Process

1. break up $\sigma_{a \wedge b \wedge c \ldots}$ into ``cascade'' of $\sigma$
   (gives more flexibility for later transformations)
2. move $\sigma$ as far down as possible
   (reduces size of intermediate results)
3. move most restrictive $\sigma$ down/left
   (reduces size of intermediate results)
4. move $\pi$ as far down as possible
   (reduces size of intermediate results)
5. replace $\times$ then $\sigma$ by equivalent $\bowtie$
   (reduces computation/size of intermediate results)
6. replace subexpressions by single operation
   (e.g. opposite of 1.) (reduces computation overhead)

---



## Amelioration Example

Consider school information database:

```
CSG(Course, StudentId, grade)
SNAP(StudentId, name, address, phone)
CDH(Course, Day, Hour)
CR(Course, Room)
```

And the query:

*Where is Brown at 9am on Monday morning?*

---



### ... Amelioration Example

An obvious translation of the query into SQL

```
select Room from CSG,SNAP,CDH,CR
where  name='Brown' and day='Mon' and hour='9am'
```

and an obvious mapping of the SQL into relational algebra gives

$$\pi_{Room} ( \sigma_{N\&D\&H} ( CSG \bowtie SNAP \bowtie CDH \bowtie CR))$$

*N* is `name='Brown'`, *D* is `Day='Mon'`, *H* is `Hour='9am'`

---

### ... Amelioration Example



$$\pi_{Room} ( \sigma_{N\&D\&H} ( CSG \bowtie SNAP \bowtie CDH \bowtie CR))$$

Push selection:

$$\pi_{Room} ( \sigma_{N\&D\&H} ( CSG \bowtie SNAP \bowtie CDH) \bowtie CR)$$

Split selection:

$$\pi_{Room} ( \sigma_{N} ( \sigma_{D\&H} ( CSG \bowtie SNAP \bowtie CDH)) \bowtie CR)$$

Push two selections:

$$\pi_{Room} ( ( \sigma_{N} (CSG \bowtie SNAP) \bowtie \sigma_{D\&H} (CDH)) \bowtie CR)$$

Push selection:

$$\pi_{Room} ( ((CSG \bowtie \sigma_{N} (SNAP)) \bowtie \sigma_{D\&H} (CDH)) \bowtie CR)$$

Can we show that the final version is more efficient?

---

### ... Amelioration Example



Could use an argument based on size of intermediate results, as above.

But run into problems with expressions like:

$$CSG \bowtie SNAP \bowtie CDH \bowtie CR$$

Order of joins can make a significant difference?

How to decide "best" order?

Need understanding of physical characteristics of relations
(for example, selectivity and likelihood of join matches across tables)

---

## Physical (Cost-Based) Optimisation



Need to determine:

- order in which operations applied (execution plan)
- precisely how each operation done (map to DBMS ops)
- size of intermediate results (need to estimate these)

From these, estimate overall evaluation cost.

Do this for all possible execution plans.

Choose cheapest plan.

---

# Execution Plans



Consider query execution plans for the RA expression:

$$\sigma_c (R \bowtie_d S \bowtie_e T)$$

Plan #1

```
tmp1   :=  HashJoin[d](R,S)
tmp2   :=  SortMergeJoin[e](tmp1,T)
result :=  BinarySearch[c](tmp2)
```

---

### ... Execution Plans



Plan #2

```
tmp1   := SortMergeJoin[e](S,T)
tmp2   := HashJoin[d](R,tmp1)
result := LinearSearch[c](tmp2)
```

Plan #3

```
tmp1   := BtreeSearch[c](R)
tmp2   := HashJoin[d](tmp1,S)
result := SortMergeJoin[e](tmp2)
```

All plans produce same result, but have quite different costs.

---

# Cost-based Query Optimiser



Overview of cost-based query optimisation process:

- start with RA tree from compilation of SQL query
- use RA laws as rewrite rules to generate new RA trees
- for each node in RA tree, choose specific access method

For a given SQL query, there are

- very many *possible* RA trees   (large choice of re-write rules)
- many *possible* execution plans   (choice of access methods, params)

Too many combinations to enumerate all; prune via heuristics.

---

### ... Cost-based Query Optimiser



Approximate algorithm for cost-based optimisation:

```
translate SQL query to RAexp
for all transformations RA' of RAexp {
    for each node e of RA' (recursively) {
        select access method for e
        plan[i++] = access method for e
    }
    cost = 0
    for each op in plan[]
        { cost += Cost(op) }
    if (cost < MinCost) {
        MinCost = cost
        BestPlan = plan
}   }
```

As noted above, we do not consider *all* transformations.
Heuristics: push selections down, consider only left-deep join trees.

---

# Choosing Access Methods



Inputs:

- a single RA operation ($\sigma$, $\pi$, $\bowtie$)
- information about file organisation, data distribution, ...
- list of operations available in the database engine

Output:

- calls to database operations to implement this RA operation

---

### ... Choosing Access Methods



**Example:**

- RA operation: $\sigma_{name='John' \land age>21}(Student)$
- `Student` relation has B-tree index on `name`
- database engine (obviously) has B-tree search method

giving

```
tmp[i]   := BtreeSearch[name='John'](Student)
tmp[i+1] := LinearSearch[age>21](tmp[i])
```

Where possible, use pipelining to avoid storing `tmp[i]` on disk.

---

### ... Choosing Access Methods



Rules for choosing $\sigma$ access methods:

- $\sigma_{A=c}(R)$ and `R` has index on `A` $\Rightarrow$ `indexSearch[A=c](R)`
- $\sigma_{A=c}(R)$ and `R` is hashed on `A` $\Rightarrow$ `hashSearch[A=c](R)`
- $\sigma_{A=c}(R)$ and `R` is sorted on `A` $\Rightarrow$ `binarySearch[A=c](R)`
- $\sigma_{A \geq c}(R)$ and `R` has clustered index on `A` $\Rightarrow$ `indexSearch[A=c+1](R)` then scan
- $\sigma_{A \geq c}(R)$ and `R` is hashed on `A` $\Rightarrow$ `linearSearch[A>=c](R)`

---

### ... Choosing Access Methods



Rules for choosing $\bowtie$ access methods:

- $R \bowtie S$ and `R` fits in memory buffers $\Rightarrow$ `bnlJoin(R,S)`
- $R \bowtie S$ and `R`,`S` sorted on join attr $\Rightarrow$ `smJoin(R,S)` (merge only)
- $R \bowtie S$ and `R` has index on join attr $\Rightarrow$ `inlJoin(S,R)`
- $R \bowtie S$ and no indexes, no sorting, $|R| \ll |S|$ $\Rightarrow$ `hashJoin(R,S)`

(`bnl` = block nested loop; `inl` = index nested loop; `sm` = sort merge)

---

# Example Plan Enumeration



Consider the query:

```
select name
from   Student s, Enrol e
where  s.sid = e.sid AND e.cid = 'COMP9315';
```

where

| **Relation** | *r* | *b* | **PrimKey** | **Clustered** | **Indexes** |
|---|---|---|---|---|---|
| `Student` | 10000 | 500 | `sid` | No | B-tree on `sid` |
| `Enrol` | 40000 | 400 | `cid` | Yes | B-tree on `cid` |
| `S ⋈ E` | 40000 | 800 | `sid,cid` | No | - |

---

### ... Example Plan Enumeration



First step is to map SQL to RA expression

$\sigma_{9315}\,(\,\texttt{Student}\ \bowtie_{sid}\ \texttt{Enrol}\,)$

then devise an execution plan based on this expression

- index on `Student.sid` ⇒ index nested loop join
- join result not sorted on `Enrol.cid` ⇒ linear scan

giving

```
tmp1   := inlJoin[s.sid=e.sid](Enrol,Student)
result := LinearSearch[e.id='COMP9315'](tmp1)
```

---

### ... Example Plan Enumeration



Estimated cost for this plan:

$$\begin{aligned}
\text{Cost} &= \text{inl-join on } (\texttt{Enrol,Student}) + \text{linear scan of } \texttt{tmp1} \\
&= T_r(b_S + r_S.btree_E) + T_w b_{tmp1} + T_r b_{tmp1} \\
&= 500 + 10{,}000.4 + 800 + 800 \\
&= 42{,}100
\end{aligned}$$

Notes:

- if we assume pipelining on `tmp1`, can remove (800+800) term
- if we make `Enrol` the outer relation, join cost is *400 + 40,000.3*

---

### ... Example Plan Enumeration



Apply rule for pushing select into join:

$\sigma_{9315}\,(\,Student\ \bowtie_{sid}\ Enrol\,)$

giving

$Student\ \bowtie_{sid}\ \sigma_{9315}\,(\,Enrol\,)$

and devise a plan based on

- index on `Enrol.cid` ⇒ B-tree lookup
- $\sigma_{9315}$ output small ⇒ buffered nested loop join

giving

```
tmp1   := BtreeSearch[e.cid='COMP9315'](Enrol)
result := bnlJoin[s.sid=e.sid](tmp1,Student)
```

---

### ... Example Plan Enumeration



Estimated cost for this plan:

| | | |
|---|---|---|
| Cost | = | btree-search on `Enrol` + bnl-join of (`tmp1,Student`) |
| | = | search+scan on `Enrol` + $T_r(b_{tmp1} + b_S)$ |
| | = | $3T_r + 3T_r + T_r(1 + 500)$ |
| | = | $3 + 3 + 1 + 500 \;=\; 507$ |

Notes:

- assumes 200 students enrolled in COMP9315
- with pipelining $1T_r$ would vanish
- if `Enrol` no clustered, $3T_r$ scan would increase

---

## Cost Estimation



Without actually executing it, cannot always know the cost of plan precisely.

(E.g. how big is the select result that feeds into the join?)

Thus, query optimisers need to *estimate* costs.

Two aspects to cost estimation:

- cost of performing operation (dealt with extensively in earlier lectures)
- size of result (which affects cost of performing next operation)

---

## Cost Estimation Information



Cost estimation uses statistical information about relations:

| | |
|---|---|
| $r_S$ | cardinality of relation *S* |
| $R_S$ | avg size of tuple in relation *S* |
| $V(A,S)$ | # distinct values of attribute *A* |
| $min(A,S)$ | min value of attribute *A* |
| $max(A,S)$ | max value of attribute *A* |

---

## Estimating Projection Result Size



Straightforward, since we know:

- all tuples from input table are included in result
- all required attributes (and their sizes) ⇒ $R_{out}$

#bytes in result = $r_S \times R_{out}$

If pipelining through buffers of size *B*, then

#buffers-full = $r_S \times \lceil B/R_{out} \rceil$

---

## Estimating Selection Result Size



Selectivity factor = fraction of tuples expected to satisfy a condition.

Common assumption: attribute values uniformly distributed.

**Example:** Consider the query

```
select * from Parts where colour='Red'
```

and assume that there are only four possible colours.

If we have 1000 `Parts`, we would expect 250 of them to be `Red`.

In other words,

$$V(colour,R)=4, \quad r_R=1000 \;\Rightarrow\; |\sigma_{colour=K}(R)|=250$$

In general, $|\sigma_{A=c}(R)| \;\cong\; r_R/V(A,R)$

---

### ... Estimating Selection Result Size



Estimating size of result for e.g.

```
select * from Enrolment
where  year > 2003;
```

Could estimate by using:

- uniform distribution assumption, *r*, min/max years

Assume: min(year)=1996, max(year)=2005, $|Enrolment|=10^5$

- $10^5$ from 1996-2005 means approx 10000 enrolments/year
- this suggests 20000 enrolments since 2003

Of course, we know that enrolments grow continually, so this underestimates.

Simpler heuristic used by some systems: selected tuples = *r/3*

---

### ... Estimating Selection Result Size



Estimating size of result for e.g.

```
select * from Enrolment
where  course <> 'COMP9315';
```

Could estimate by using:

- uniform distribution assumption, *r*, #courses

Alternative, simpler way to estimate:

- assume that most tuples are not equal to chosen value
- # selected tuples = *r*

---

### ... Estimating Selection Result Size



Uniform distribution assumption is convenient, but often not realistic.

How to handle non-uniform attribute value distributions?

- collect statistics about the values stored in the attribute/relation
- store these as e.g. a histogram in the meta-data for the relation

So, for the part colour example, we might have a distribution like:

`White`: 35% `Red`: 30% `Blue`: 25% `Silver`: 10%

Use histogram as basis for determining # selected tuples.

Disadvantage: cost of storing/maintaining histograms.

---

## Estimating Join Result Size



Analysis is not as simple as select.

Relies on semantic knowledge about data/relations.

Consider equijoin on common attr: $R \bowtie_A S$

Case 1: $|R \bowtie_A S| = 0$

Useful if we know that $dom(A,R) \cap dom(A,S) = \{\}$

Case 2: $A$ is unique key in both `R` and `S`
⇒ can't be more tuples in join than in smaller of `R` and `S`.

$$max(|R \bowtie_A S|) = min(|R|, |S|)$$

---

### ... Estimating Join Result Size



Case 3: $A$ is primary key in `R`, foreign key in `S`
⇒ every `S` tuple has at most one matching `R` tuple.

$$max(|R \bowtie_A S|) = |S|$$

**Example:**

```
select name from Students s, Enrol e
where  e.cid = 'COMP9315' AND e.sid = s.sid
```

$$|Students \bowtie \sigma_{9315}(Enrol)| = |\sigma_{9315}(Enrol)|$$

---

## Cost Estimation: Postscript



Inaccurate cost estimation can lead to poor evaluation plans.

Above methods can (sometimes) give inaccurate estimates.

To get more accurate estimates costs:

- more time ... complex computation of selectivity
- more space ... storage for histograms of data values

Either way, optimisation process costs more (more than query?)

Trade-off between optimiser performance and query performance.

---



## Semantic Query Optimisation (SQO)

Makes use of semantics of data to assist query optimisation process.

The discussion of join cost estimation above gives an example of this.

Kinds of information used:

- knowledge about relations
- nature of data
- constraints within/between attributes/relations

SQO uses this to simplify queries and reduce search space.

Two examples of semantic transformation operations:

- restriction elimination,   index introduction

---



## Semantic Equivalence

Semantic equivalence does not require syntactic equivalence.

Consider the two queries:

```
select * from Emp where sal > 80K
```

```
select * from Emp where sal > 80K and job = 'Prof'
```

They are equivalent under the semantic rule:

`Emp.job = 'Prof' ↔ Emp.sal > 80K`

(i.e. Profs are the only people earning more than $80,000)

Could use this to transform query to exploit index on `salary`.

---



## Restriction Elimination

Query:

*List all the departments that store benzene in quantities of more than 400*

```
select dept from  Storage
where material = 'Benzene' and qty > 400
```

Use rule: `material = 'Benzene' ⇒ qty > 500`

```
select dept from Storage where material = 'Benzene'
```

Result: Unnecessary restriction on the attribute `qty` is eliminated.

---



## Index Introduction

Query: Find all the employees who make more than 42K

```
select name from Employees where salary > 42K
```

Use rule: `salary > 42K` ⇒ `job = 'manager'`

```
select name from Employees
where salary > 42K and job = 'manager'
```

Result: A new constraint is obtained on the indexed attribute `job`.

---

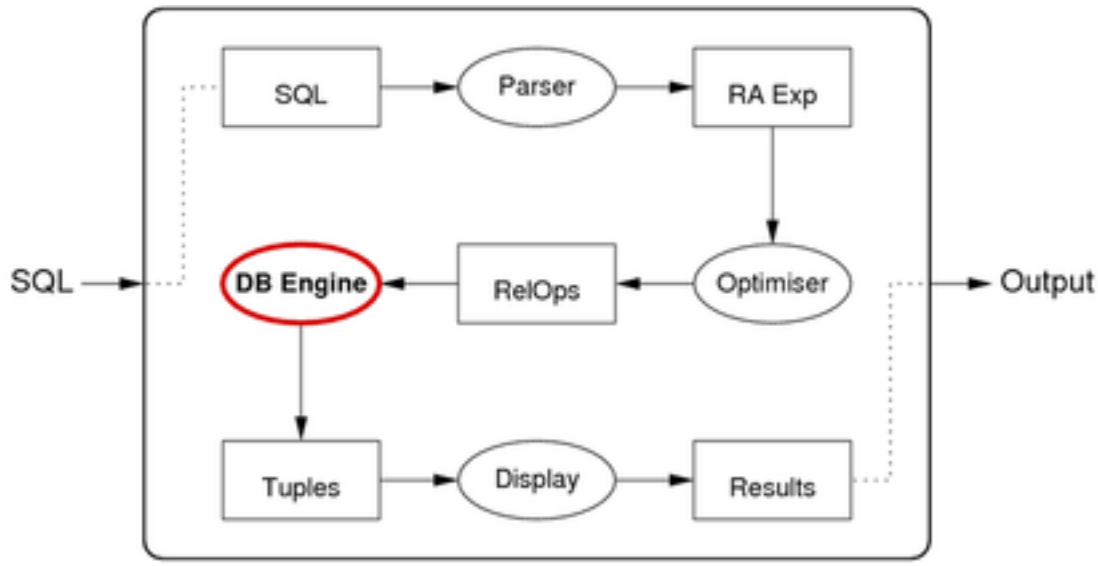
# Query Execution

---

## Query Execution



Query execution: applies evaluation plan → set of result tuples




---

### ... Query Execution



Query execution

- applies a query execution plan
- and produces a set of result tuples

---

### ... Query Execution



Example of query translation:

```
select s.name, s.id, e.course, e.mark
from   Student s, Enrolment e
where  e.student = s.id and e.semester = '05s2';
```

maps to

$$\pi_{name,id,course,mark}(Stu \bowtie_{e.student=s.id} (\sigma_{semester=05s2}Enr))$$

maps to

```
Temp1  = BtreeSelect[semester=05s2](Enr)
Temp2  = HashJoin[e.student=s.id](Stu,Temp1)
Result = Project[name,id,course,mark](Temp2)
```

---

### ... Query Execution



A query execution plan:

* consists of a *sequence of operations*
* each operation is a relational algebra operator
  (a specific implementation with particular performance characteristics)

Results may be passed from one operator to the next:

* *materialization* ... writing results to disk and reading them back
* *pipelining* ... generating and passing results one-at-a-time

---

### ... Query Execution



Two ways of communicating results between query blocks ...

*Materialization*

* first block writes all results to disk
* next block reads tuples from disk to process
* advantage: can exploit file structures (e.g. hashing)

*Pipelining*

* blocks execute "concurrently" as producer/consumer pairs
* structured as interacting iterators (open; while(next); close)
* advantage: no requirement for disk access

---

## Materialization



Steps in *materialization* between two operators

* first operator reads input(s) and writes results to disk
* next operator treats tuples results on disk as its input

Advantage:

* intermediate results can be placed in a file structure
  (which can be chosen to speed up execution of subsequent operators)

Disadvantage:

* requires disk space/writes for intermediate results
* requires disk access to read intermediate results

---

### ... Materialization



Example:

```
select s.name, s.id, e.course, e.mark
from   Student s, Enrolment e
where  e.student = s.id and
       e.semester = '05s2' and s.name = 'John';
```

might be executed as

```
Temp1  = BtreeSelect[semester=05s2](Enrolment)
Temp2  = BtreeSelect[name=John](Student)
         -- indexes on name and semester
         -- produce sorted Temp1 and Temp2
Temp3  = SortMergeJoin[e.student=s.id](Temp1,Temp2)
         -- SMJoin especially effective, since
         -- Temp1 and Temp2 are already sorted
Result = Project[name,id,course,mark](Temp3)
```

---

# Pipelining



How *pipelining* is organised between two operators:

- blocks execute "concurrently" as producer/consumer pairs
- first operator acts as producer; second as consumer
- structured as interacting iterators (open; while(next); close)

Advantage:

- no requirement for disk access (results passed via memory buffers)

Disadvantage:

- each operator accesses inputs via linear scan

---

# Iterators (reminder)



Iterators provide a "stream" of results:

- **`iter = open(`*params*`)`**
  - set up data structures for iterator (create state, open files, ...)
  - *params* are specific to operator (e.g. reln, condition, #buffers, ...)
- **`tuple = next(iter)`**
  - get the next tuple in the iteration; return null if no more
- **`close(iter)`**
  - clean up data structures for iterator

Other possible operations: reset to specific point, restart, ...

---

### ... Iterators (reminder)



Implementation of single-relation selection iterator:

```
typedef struct {
    File   inf;  // input file
    Cond   cond; // selection condition
    Buffer buf;  // buffer holding current page
    int    curp; // current page during scan
    int    curr; // index of current record in page
} Iterator;
```

Iterator structure contains information:

- related to operation being performed (e.g. `cond`)
- information giving current execution state (e.g. `curp, curr`)

---

### ... Iterators (reminder)



Implementation of single-relation selection iterator (cont):

```
Iterator *open(char *relName, Condition cond) {
    Iterator *iter = malloc(sizeof(Iterator));
    iter->inf  = openFile(fileName(relName),READ);
    iter->cond = cond;
    iter->curp = 0;
    iter->curr = -1;
    readBlock(iter->inf, iter->curp, iter->buf);
    return iter;
}
void close(Iterator *iter) {
```

```
    closeFile(iter->inf);
    free(iter);
}
```

---

### ... Iterators (reminder)



Implementation of single-relation selection iterator (cont):

```
Tuple next(Iterator *iter) {
    Tuple rec;
    do {
        // check if reached end of current page
        if (iter->curr == nRecs(iter->buf)-1) {
            // check if reached end of data file
            if (iter->curp == nBlocks(iter->inf)-1)
                return NULL;
            iter->curp++;
            iter->buf = readBlock(iter->inf, iter->curp);
            iter->curr = -1;
        }
        iter->curr++;
        rec = getRec(iter->buf, iter->curr);
    } while (!matches(rec, iter->cond));
    return rec;
}
// curp and curr hold indexes of most recently read page/record
```

---

## Pipelining Example



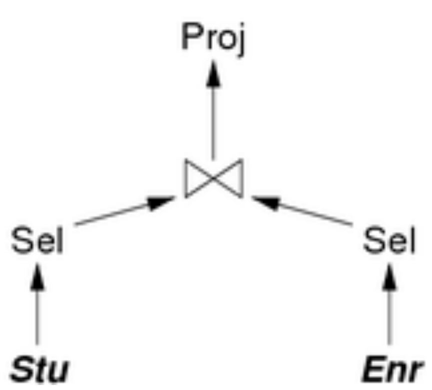
Consider the query:

```
select s.id, e.course, e.mark
from   Student s, Enrolment e
where  e.student = s.id and
       e.semester = '05s2' and s.name = 'John';
```

which maps to the RA expression

*Proj[id,course,mark](Join[student=id](Sel[05s2](Enr),Sel[John](Stu)))*

which could represented by the RA expression tree

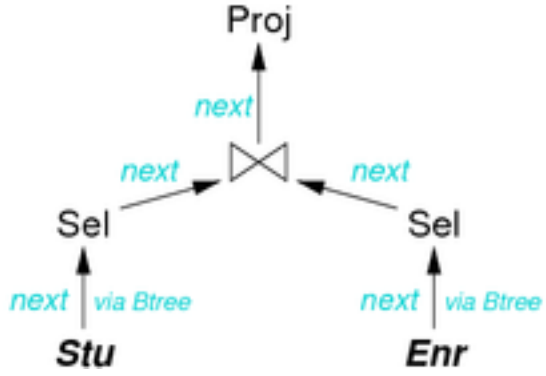



---

### ... Pipelining Example



Modelled as communication between RA tree nodes:




## ... Pipelining Example



This query might be executed as

```
System:
    iter0 = open(Result)
    while (Tup = next(iter0)) { display Tup }
    close(iter0)
Result:
    iter1 = open(Join)
    while (T = next(iter1))
        { T' = project(T); return T' }
    close(iter1)
Sel1:
    iter4 = open(Btree(Enrolment,'semester=05s2'))
    while (A = next(iter4)) { return A }
    close(iter4)
...
```

## ... Pipelining Example



```
...
Join: -- nested-loop join
    iter2 = open(Sel1)
    while (R = next(iter2) {
        iter3 = open(Sel2)
        while (S = next(iter3))
            { if (matches(R,S) return (RS) }
        close(iter3) // better to reset(iter3)
    }
    close(iter2)
Sel2:
    iter5 = open(Btree(Student,'name=John'))
    while (B = next(iter5)) { return B }
    close(iter5)
```

# Pipeline Execution



Piplines can be executed as ...

*Demand-driven*

- producers wait until consumers request tuples

*Producer-driven*

- producers generate tuples until output buffer full, then wait

In both cases, top-level driver is request for result tuples.

In parallel-processing systems, iterators could run concurrently.

# Disk Accesses



Pipelining cannot avoid all disk accesses.

Some operations use multiple passes (e.g. merge-sort, hash-join).

- data is written by one pass, read by subsequent passes

Thus ...

- *within* an operation, disk reads/writes are possible
- *between* operations, no disk reads/writes are needed

---

### ... Disk Accesses



Sophisticated query optimisers might realise e.g.

> *if operation X writes its results to a file with structure S,*
> *the subsequent operation Y will proceed much faster*
> *than if Y reads X's output tuple-at-a-time*

In this case, it could materialize *X*'s output in an *S*-file.

Produces a pipeline/materialization hybrid query execution.

Example:

- selection writes output into an indexed file (Btree)
- later join can then be implemented as efficient index-join

---

### ... Disk Accesses



Example: (pipeline/materialization hybrid)

```
select s.id, e.course, e.mark
from   Student s, Enrolment e
where  e.student = s.id and
       e.semester = '05s2' and s.name = 'John';
```

might be executed as

```
System:
    exec(Sel2)  -- creates Temp1
    iter0 = open(Result)
    while (Tup = next(iter0)) { display Tup }
    close(iter0)
Result:
    iter1 = open(Join)
    while (T = next(iter1))
        { T' = project(T); return T' }
    close(iter1)
...
```

---

### ... Disk Accesses



```
...
Join: -- index join
    iter2 = open(Sel1)
    while (R = next(iter2) {
        iter3 = open(Btree(Temp1,'id=R.student'))
        while (S = next(iter3)) { return (RS) }
```

```
        close(iter3)
    }
    close(iter2)
Sel1:
    iter4 = open(Btree(Enrolment,'semester=05s2'))
    while (A = next(iter4)) { return A }
    close(iter4)
Sel2:
    iter5 = open(Btree(Student,'name=John'))
    createBtree(Temp1,'id')
    while (B = next(iter5)) { insert(B,Temp1) }
    close(iter5)
```

---

# PostgreSQL Execution



Defs: `src/include/executor` and `src/include/nodes`

Code: `src/backend/executor`

PostgreSQL uses pipelining ...

- query plan is a tree of `Plan` nodes
- each type of node implements one kind of RA operation
  (node implements specific access methods and provides iterator interface)
- node types e.g. `Scan`, `Group`, `Indexscan`, `Sort`, `HashJoin`
- execution is managed via a tree of `PlanState` nodes
  (mirrors the structure of the tree of Plan nodes; holds execution state)

---

## ... PostgreSQL Execution



Modules in `src/backend/executor` fall into two groups:

`execXXX` (e.g. execMain, execProcnode, execScan)

- implement generic control of plan evaluation (execution)
- provide overall plan execution and dispatch to node iterators

`nodeXXX` (e.g. nodeSeqscan, nodeNestloop, nodeGroup)

- implement iterators for specific types of RA operators
- typically contains `ExecInitXXX`, `ExecXXX`, `ExecEndXXX`

The "style" is OO (e.g. specialisations of Nodes), but implementation in C masks this

---

## ... PostgreSQL Execution



Top-level data/functions for executor ...

`QueryDesc`

- contains plan and state information (e.g. pointer to root of plan tree)

`ExecutorStart(QueryDesc *, ...)`

- initialises all dynamic state information (e.g. iterators)

`ExecutorRun(QueryDesc *, ScanDirection, ...)`

- implements "get next result tuple" (via iterator tree)

`ExecutorEnd(QueryDesc *)`

* cleans up all iterator and state information

---

## ... PostgreSQL Execution



Overview of query processing:

```
CreateQueryDesc

ExecutorStart
    CreateExecutorState -- creates per-query context
    switch to per-query context to run ExecInitNode
    ExecInitNode -- recursively scans plan tree
        CreateExprContext -- creates per-tuple context
        ExecInitExpr

ExecutorRun
    ExecutePlan -- invoke iterators from root
        ExecProcNode -- recursively called in per-query context
            ExecEvalExpr -- called in per-tuple context
            ResetExprContext -- to free memory

ExecutorEnd
    ExecEndNode -- recursively releases resources
    FreeExecutorState -- frees per-query and child contexts

FreeQueryDesc
```

---

## ... PostgreSQL Execution



More detailed view of plan execution (but still much simplified)

```
ExecutePlan(execState, planStateNode, ...) {
    process "before each statement" triggers
    for (;;) {
        tuple = ExecProcNode(planStateNode)
        check tuple validity // MVCC
        if (got a tuple) break
    }
    process "after each statement" triggers
    return tuple
}
ExecProcNode(node) {
    switch (nodeType(node)) {
    case SeqScan:
        result = ExecSeqScan(node); break;
    case NestLoop:
        result = ExecNestLoop(node); break;
    ...
    }
    return result;
}
```

---

## ... PostgreSQL Execution



Generic iterator interface is provided by ...

**`ExecInitNode`**

* initialize a plan node and its subplans

**`ExecProcNode`**

- get a tuple by executing the plan node

**ExecEndNode**

- shut down a plan node and its subplans

Each calls corresponding function for specific node type
(e.g. for nested loop join ExecInitNestLoop(), ExecNestLoop(), ExecEndNestLoop())

---

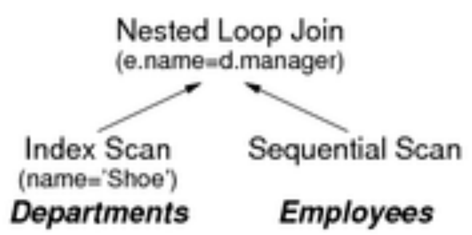
# Example PostgreSQL Execution



Consider the query:

```
-- get manager's age and # employees in Shoe department
select e.age, d.nemps
from   Departments d, Employees e
where  e.name = d.manager and d.name ='Shoe'
```

and its execution plan tree




---

### ... Example PostgreSQL Execution



This produces a tree with three nodes:

- `NestedLoop` with join condition (Outer.manager = Inner.name)
- `IndexScan` on Departments with selection (name = 'Shoe')
- `SeqScan` on Employees

We ignore the top-level node here (it handles the projection via attrList)

---

### ... Example PostgreSQL Execution



Initially InitPlan() invokes ExecInitNode() on plan tree root.

```
ExecInitNode() sees a NestedLoop node ...
    so dispatches to ExecInitNestLoop() to set up iterator
    and then invokes ExecInitNode() on left and right sub-plans
        in left subPlan, ExecInitNode() sees an IndexScan node
            so dispatches to ExecInitIndexScan() to set up iterator
        in right sub-plan, ExecInitNode() sees aSeqScan node
            so dispatches to ExecInitSeqScan() to set up iterator
```

Result: a plan state tree with same structure as plan tree.

---

### ... Example PostgreSQL Execution



Execution: ExecutePlan() repeatedly invokes ExecProcNode().

```
ExecProcNode() sees a NestedLoop node ...
    so dispatches to ExecNestedLoop() to get next tuple
    which invokes ExecProcNode() on its sub-plans
        in the left sub-plan, ExecProcNode() sees an IndexScan node
```

```
        so dispatches to ExecIndexScan() to get next tuple
        if no more tuples, return END
        for this tuple, invoke ExecProcNode() on right sub-plan
            ExecProcNode() sees a SeqScan node
                so dispatches to ExecSeqScan() to get next tuple
            check for match and return joined tuples if found
        reset right sub-plan iterator
```

Result: stream of result tuples returned via ExecutePlan()

---

# Performance Tuning

---

# 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. 90% withdrawals; 10% deposits; 50% balance check)
- are there time constraints on queries/transactions?
  (e.g. EFTPOS payments must be approved within 7 seconds)
- are there uniqueness constraints on any attributes?
  (define indexes 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*
- *outside* schema design
  - typically after application has been deployed/used
  - requires adding/modifying data structures such as *indexes*

---

# Denormalisation



*Normalisation* minimises storage redundancy.

- achieves this by "breaking up" data into logical chunks
- requires minimal "maintenance" to ensure consistency

Problem: queries that need to put data back together.

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

Solution: store some data redundantly

- benefit: queries no longer need expensive joins
- trade-off: extra maintenance effort to keep consistency
- worthwhile if joins are frequent and updates are rare

---

### ... Denormalisation



Example ... consider the following normalised schema:

```
create table Subject (
   id         serial primary key,
   code       char(8), -- e.g. COMP9315
   title      varchar(60),
   syllabus   text, ... );
create table Term (
   id         serial primary key,
   name       char(4), -- e.g. 09s2
   starting   date, ... );
create table Course (
   subject    integer references Subject(id),
   term       integer references Term(id),
   lic        integer references Staff(id), ... );
```

---

### ... Denormalisation



Example: Courses = Course ⋈ Subject ⋈ Term

If we often need to refer to "standard" name (e.g. COMP9315 09s2)

- 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.name, avg(e.mark)
from   Course c, Subject s, Term t
where  c.subject = s.id and c.term = t.id
       and s.code='COMP9315' and t.name='09s2'
-- by a query like:
select c.courseName, e.grade, e.mark
from   Course c
where  c.courseName = 'COMP9315 09s2'
```

---

## Indexes



Indexes provide efficient content-based access to tuples.

Can build indexes on any (combination of) attributes.

Defininining indexes:

`CREATE INDEX` *name* `ON` *table* `(` $attr_1$`,` $attr_2$`, ... )`

$attr_i$ can be an arbitrary expression (e.g. `upper(name)`).

`CREATE INDEX` also allows us to specify

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

---

### ... Indexes



Indexes can significantly improve query costs.

Considerations in applying indexes:

- is an attribute used in frequent/expensive queries?
  (note that some kinds of queries can be answered from index alone)
- should we create an index on a collection of attributes?
  (yes, if the collection is used in a frequent/expensive query)
- is the table containing attribute frequently updated?
- 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/expensive operations

Examples which *may* prevent optimiser from using indexes:

```
select name from Employee where salary/365 > 100
       -- fix by re-phrasing condition to (salary > 36500)
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 Students 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 Students 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 Students 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 Students
ass2-#                           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
ass2-# from Students s, People 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



Example: Join on a non-indexed attribute

```
ass3=> explain select s1.code, s2.code
ass2-# from Subjects s1, Subjects s2 where s1.offerer=s2.offerer;
                        QUERY PLAN
----------------------------------------------------------------
 Merge Join  (cost=2744.12..18397.14 rows=1100342 width=18)
   Merge Cond: (s1.offerer = s2.offerer)
   ->  Sort  (cost=1372.06..1398.33 rows=10509 width=13)
         Sort Key: s1.offerer
         ->  Seq Scan on subjects s1
                      (cost=0.00..670.09 rows=10509 width=13)
   ->  Sort  (cost=1372.06..1398.33 rows=10509 width=13)
         Sort Key: s2.offerer
         ->  Seq Scan on subjects s2
                      (cost=0.00..670.09 rows=10509 width=13)
(8 rows)
```