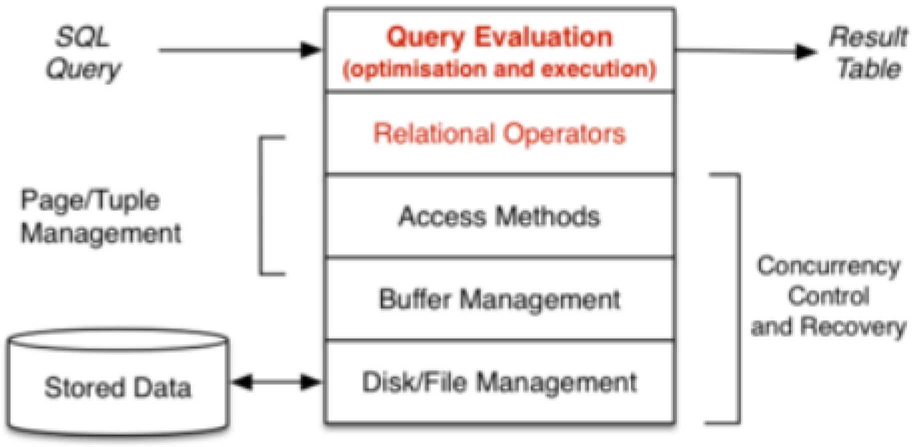
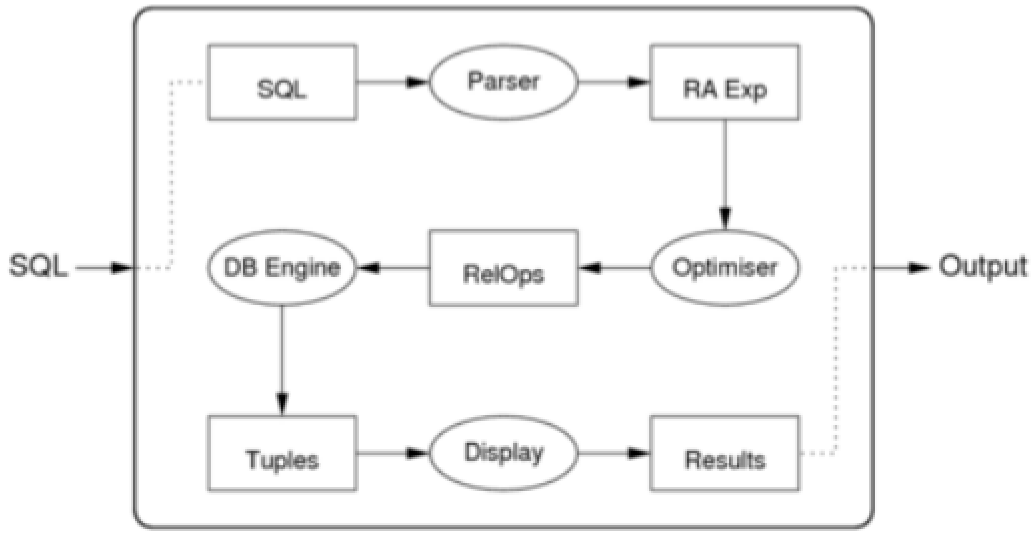
course 9 query processing part 1-query Parser and Optimiser

1. query processing overview



1.1 query evaluation



there are 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

1.2 intermediate representations

SQL → Relational Algebra (RA)

let’s talk about some basic RA in DBMS.

The standard set of RA operations in a DBMS includes:

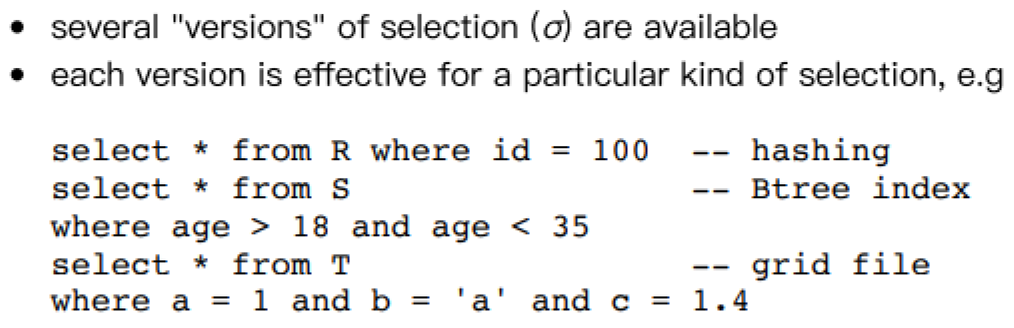
- filtering and combining (*select*, *project*, *join* (inner,outer))
- set operations (*union*, *intersection*, *difference*)

others:

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

in practice, DBMS provide several versions of each **RA operations**

for example, in selection operation:

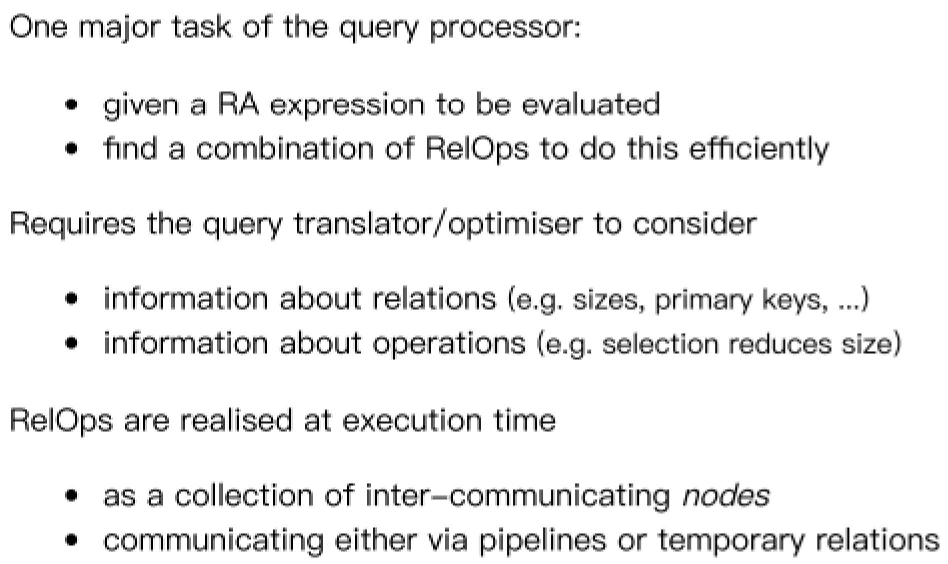


- several "versions" of selection (σ) are available
- each version is effective for a particular kind of selection, e.g

```
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
```

Note: similarly, π and $\bowtie$ (natural join) have versions to match specific query type.

and we call these specialise RA operations as RelOps



One major task of the query processor:

- given a RA expression to be evaluated
- 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. selection reduces size)

RelOps are realised at execution time

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

execution plan as collection of RelOps:

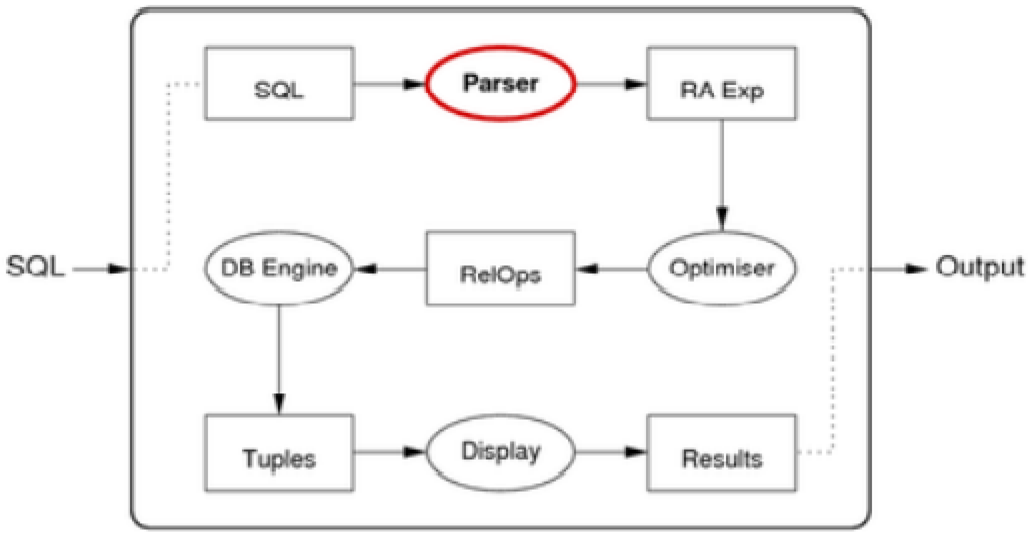
- query evaluation plan
- query execution plan
- physical query plan

and there are some representations of RA operators and expressions:

- *σ = Select = Sel,* *π = Project = Proj*
- *R ⋈ S = R Join S = Join(R,S),* *∧ = &,* *∨ = |*

2. Query Translation

query translations: ***SQL statement text*** → ***RA expression***



parser’s main job is mapping from SQL to RA, and the mapping processes includes:

- 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)
```

2.1 parsing SQL

parsing SQL is similar to compile programe languages

SQL 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'`

there are only one difference between parsing SQL and PL-compile

- PL define objects as part of program defination
- SQL references tables/types/functions stored in the DBMS

2.2 catalog and parsing

we use a school examples to show how catalog work

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

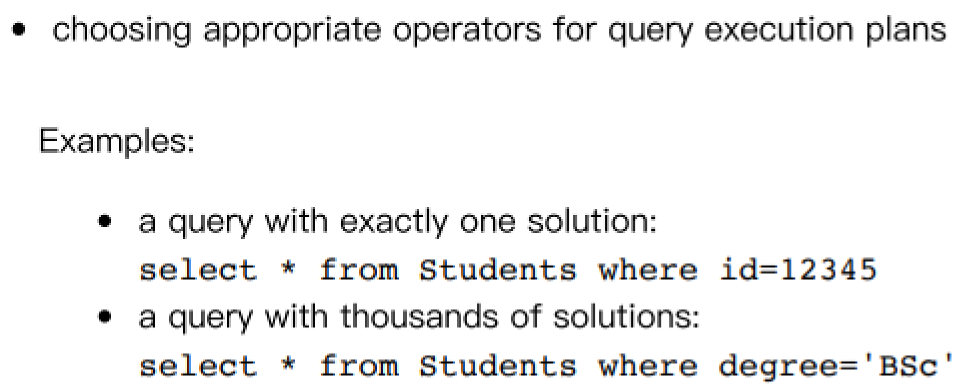
DBMS catalog stores following kinds of information:

- `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`(?)

there are two kinds of information stored in catalog:

- schema/type information
    - 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'`)

- statistical information

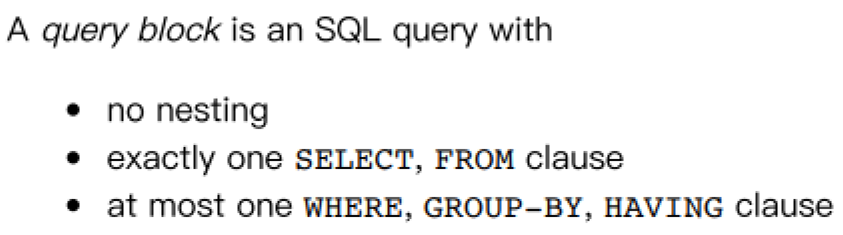


- choosing appropriate operators for query execution plans

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'`

2.3 query blocks

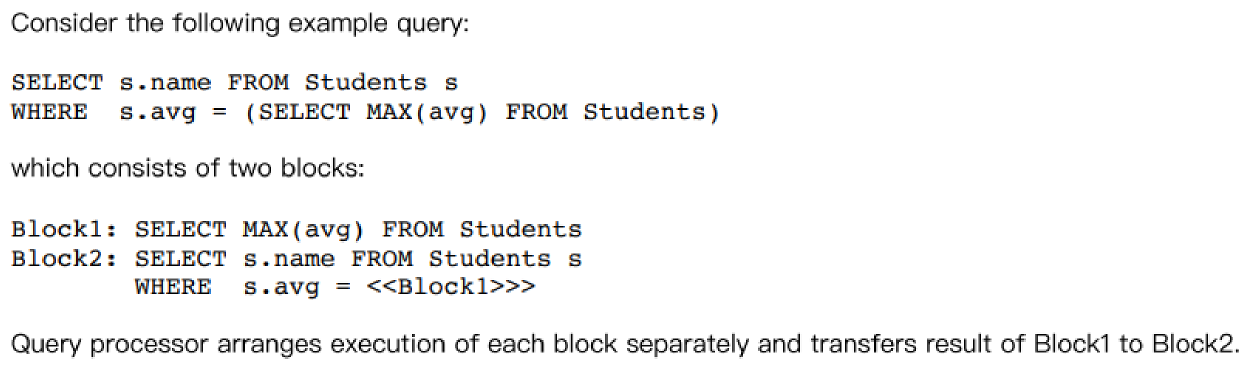


A *query block* is an SQL query with

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

query optimisers typically deal with one query at a time

so query compliers need to decompose query into 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.

2.4 mapping SQL to relational algebra

there are several translation from SQL → RA for a ***naive query compiler***, it may use the following translation scheme:

- `SELECT` clause $\rightarrow$ projection
- `FROM` clause $\rightarrow$ product
- `WHERE` clause $\rightarrow$ selection

example:

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

is translated to

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

for a ***better query compiler***:

- `SELECT` clause $\rightarrow$ projection
- `WHERE` clause on single relation $\rightarrow$ selection
- `WHERE` clause on two relations $\rightarrow$ join

examples:

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

is translated to

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

in fact, many query compiler will first produce a intermediate result, and then ***rewrite*** it through RA laws

aims of rewrite:

- use RA laws to reduce complexity

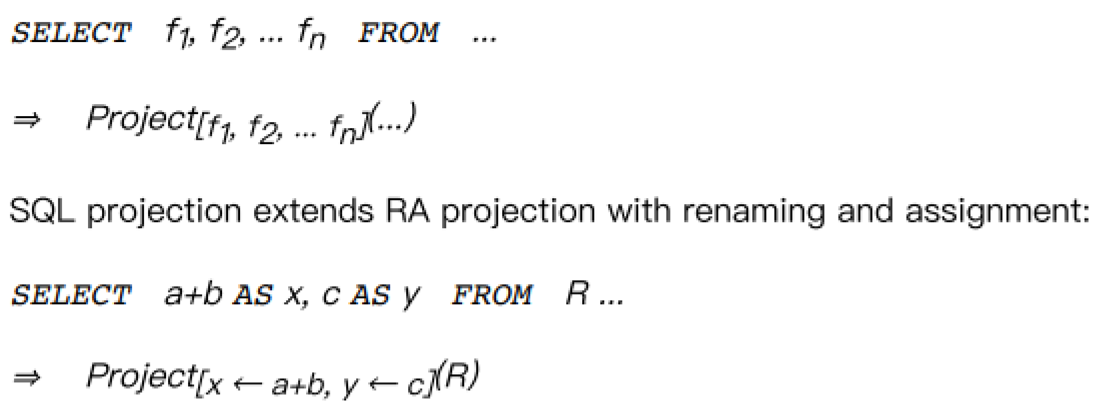
2.5 mapping rules

- a collection of templates for particular kinds of queris
- the mapping process includes:
    - choose a template according to the query
    - bind components of actual query to slots in the template
- and then covert the matched query into relational algebra

sometimes we need to use multiple templates to map whole SQL statement

and then we will talk about query → RA

- 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 ...

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

- Join

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*** ...

⇒ $Join_{[R.a\ op\ S.e]}(R,S)$

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

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

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

- Selection

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

  ⇒ $Select_{[R.f\ op\ val]}(R)$

  SELECT ... ***FROM*** ... *R* ... ***WHERE*** ... $Cond_{1,R}$ ***AND*** $Cond_{2,R}$ ...

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

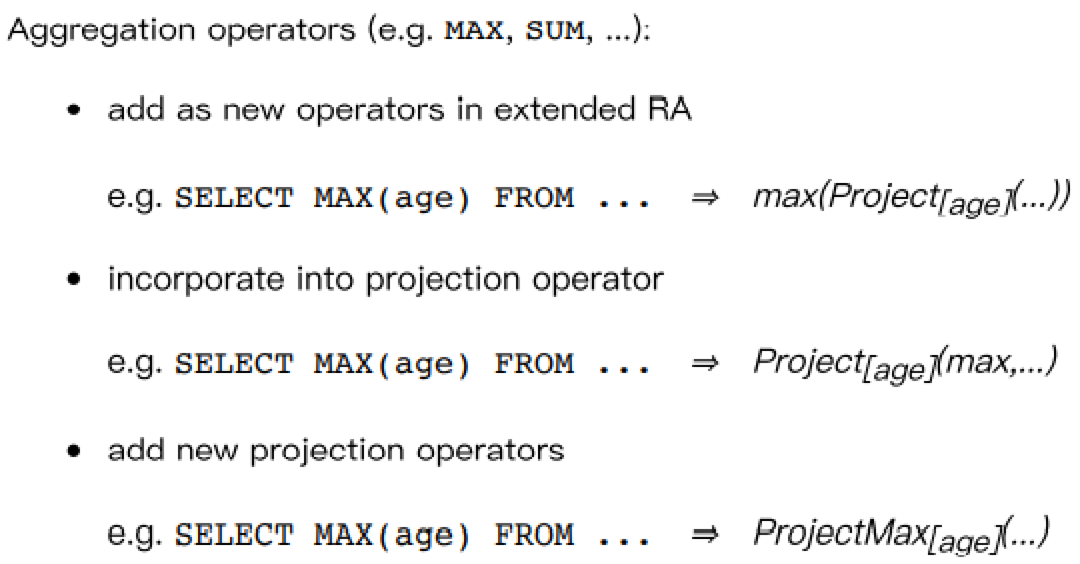
  or

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

  or

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

- Aggregation operators


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

- add as new operators in extended RA

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

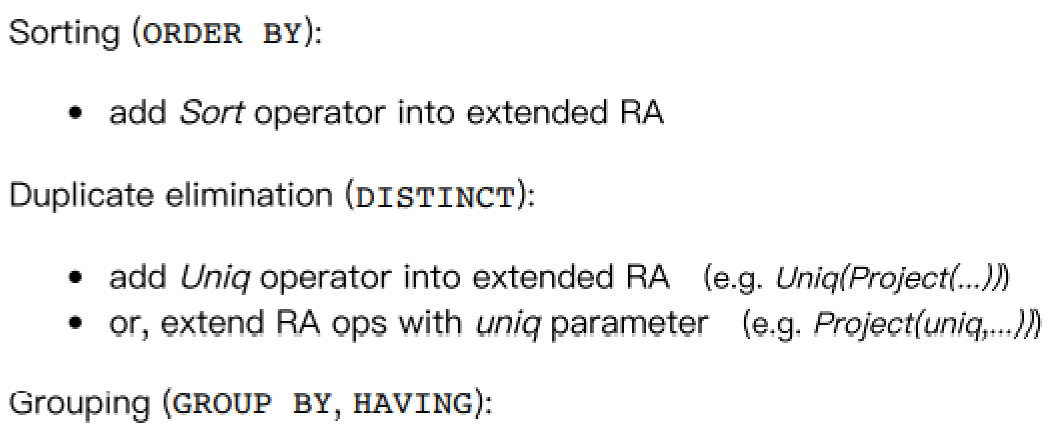
- incorporate into projection operator

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

- add new projection operators

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

- sorting && Duplicate elimination && Grouping

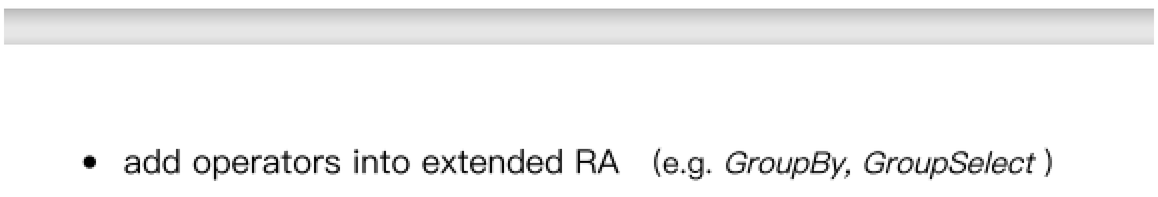

Sorting (`ORDER BY`):

- 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*)

there are a example about how a query → RA

List the names of all subjects with more than 100 students in them

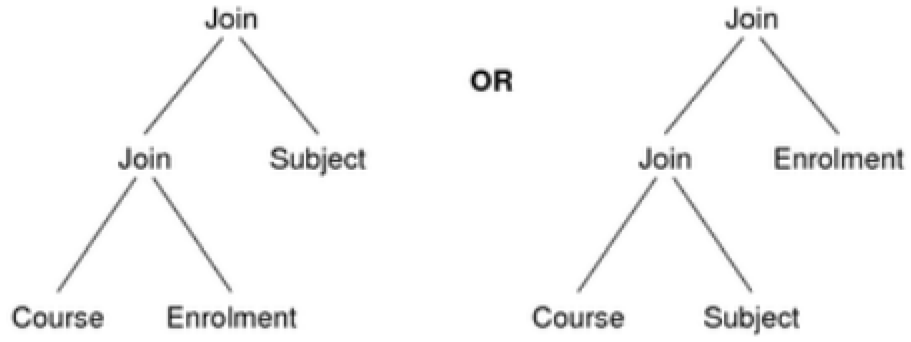
In the SQL compiler, the query

```
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 ⋈ Course ⋈ Subjects
))))
```

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.

2.6 expression rewriting rules

since RA is a well-define mathematical system:

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

Expressions transformation based on such rules can be used:

- to simplify SQL → RA mapping results
- to generate new plan variations to check in query optimisation

2.7 relational algebra laws

Commutative and Associative Laws:

- $R \bowtie S \leftrightarrow S \bowtie R, \quad (R \bowtie S) \bowtie T \leftrightarrow R \bowtie (S \bowtie T)$ (natural join)
- $R \cup S \leftrightarrow S \cup R, \quad (R \cup S) \cup T \leftrightarrow R \cup (S \cup T)$
- $R \bowtie_{Cond} S \leftrightarrow S \bowtie_{Cond} R$ (theta join)
- $\sigma_c (\sigma_d (R)) \leftrightarrow \sigma_d (\sigma_c (R))$

Selection splitting (where c and d are conditions):

- $\sigma_{c \wedge d}(R) \leftrightarrow \sigma_c (\sigma_d (R))$
- $\sigma_{c \vee d}(R) \leftrightarrow \sigma_c(R) \cup \sigma_d(R)$

Selection pushing:

Selection pushing ($\sigma_c(R \cup S)$ and $\sigma_c(R \cup S)$):

- $\sigma_c(R \cup S) \leftrightarrow \sigma_c R \cup \sigma_c S, \quad \sigma_c(R \cap S) \leftrightarrow \sigma_c R \cap \sigma_c S$

Selection pushing with join:

- $\sigma_c (R \bowtie S) \leftrightarrow \sigma_c(R) \bowtie S$ (if c refers only to attributes from R)
- $\sigma_c (R \bowtie S) \leftrightarrow R \bowtie \sigma_c(S)$ (if c refers only to attributes from S)

if conditino contains attributes from both R and S:

- $\sigma_{c' \wedge c''} (R \bowtie S) \leftrightarrow \sigma_{c'}(R) \bowtie \sigma_{c''}(S)$
- c' contains only R attributes, c'' contains only S attributes

rules for projections:

- all but last projection can be ignored

  - $\pi_{L1} (\pi_{L2} (\ldots \pi_{Ln} (R))) \rightarrow \pi_{L1} (R)$

- projections ca 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 $(L \subset M \cup N)$

2.8 Query Rewriting

Subqueries → convert to a join

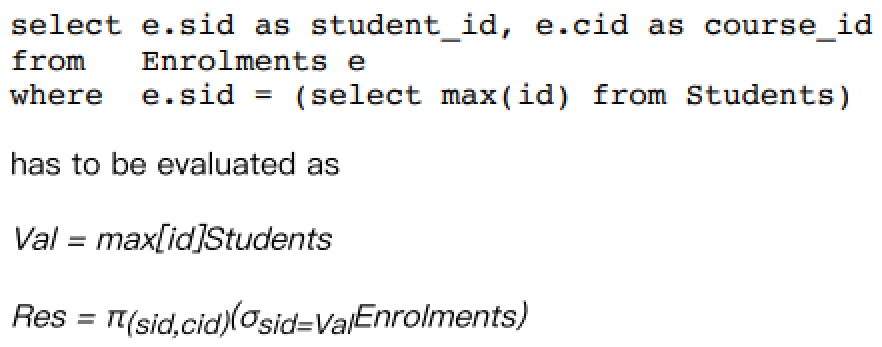
Example: (on schema Courses(id,code,...), Enrolments(cid,sid,...), Students(id,name,...)

```
select c.code, count(*)
from   Courses c
where  c.id in (select cid from Enrolments)
group  by c.code
```

becomes

```
select c.code, count(*)
from   Courses c join Enrolments e on c.id = e.cid
group  by c.code
```

but not all subqueries can be converted to join, like:



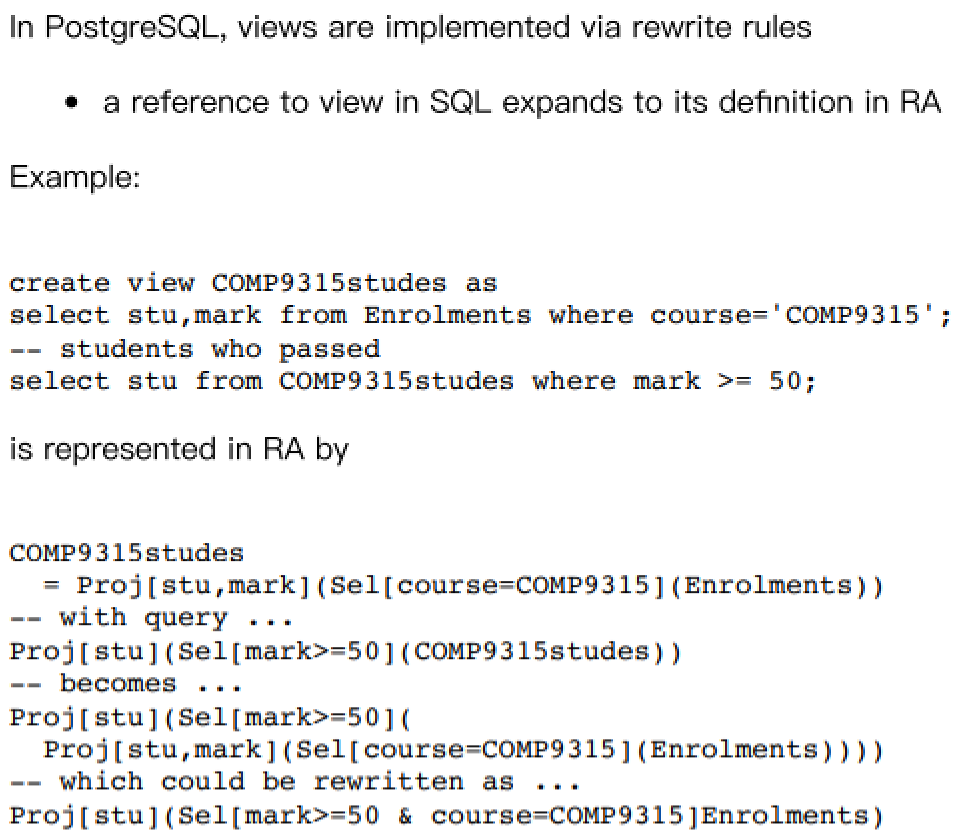
```
select e.sid as student_id, e.cid as course_id
from   Enrolments e
where  e.sid = (select max(id) from Students)
```

has to be evaluated as

Val = max[id]Students

$Res = \pi_{(sid,cid)}(\sigma_{sid=Val}Enrolments)$

some extra content about views in PostgreSQL:



In PostgreSQL, views are implemented via rewrite rules

- a reference to view in SQL expands to its definition in RA

Example:

```
create view COMP9315studes as
select stu,mark from Enrolments where course='COMP9315';
-- students who passed
select stu from COMP9315studes where mark >= 50;
```

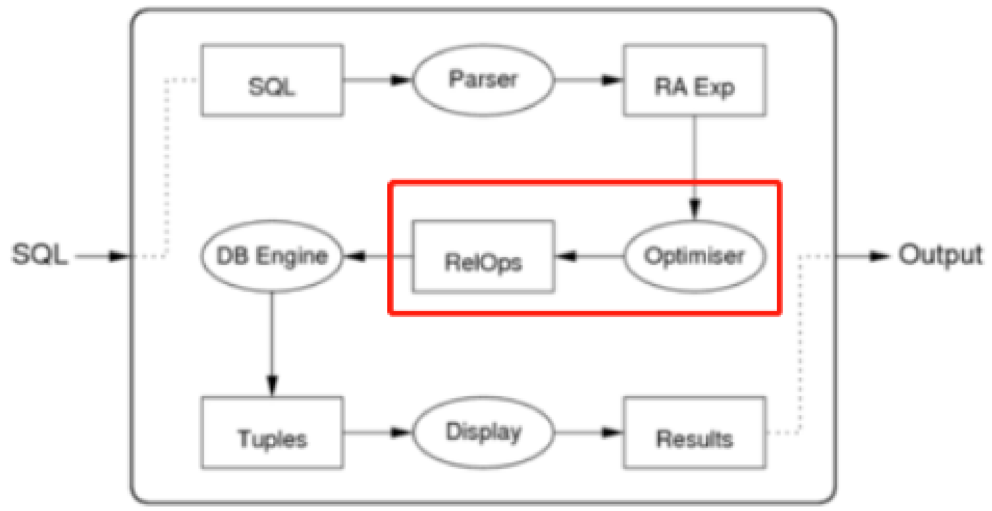
is represented in RA by

```
COMP9315studes
  = Proj[stu,mark](Sel[course=COMP9315](Enrolments))
-- with query ...
Proj[stu](Sel[mark>=50](COMP9315studes))
-- becomes ...
Proj[stu](Sel[mark>=50](
  Proj[stu,mark](Sel[course=COMP9315](Enrolments))))
-- which could be rewritten as ...
Proj[stu](Sel[mark>=50 & course=COMP9315]Enrolments)
```

3. Query Optimisation

Query optimiser: ***RA expression*** → ***efficient evaluation plan***

query optimisation is a crtical step in query evaluation：



query evaluation

query optimiser’s tasks is:

- receive RA expression from SQL compile
- produces sequence of RelOps to evaluate the expression
- query execution plan should provide efficient evaluation

actually, in Query Optimisation phase, we did not find the optimal query execution plans, since the search space is very large.

so we can only compromise:

- do limited search of query plan space (guide by heuristics)
- and quickly choose a reasonably efficient execution plan

but query optimisation is really helpful, it reduce tons of time cost, for example:

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

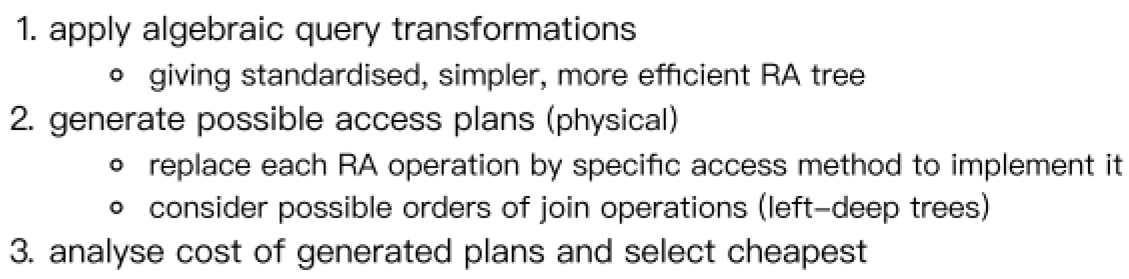
3.1 Approaches to Optimisation

three main classes of techniques developed:

- algebric (equivalances, rewriting, heuristics)
- physical (execution cost, search-based)
- ~~semantic (application properties, heuritics) (too expensive)~~

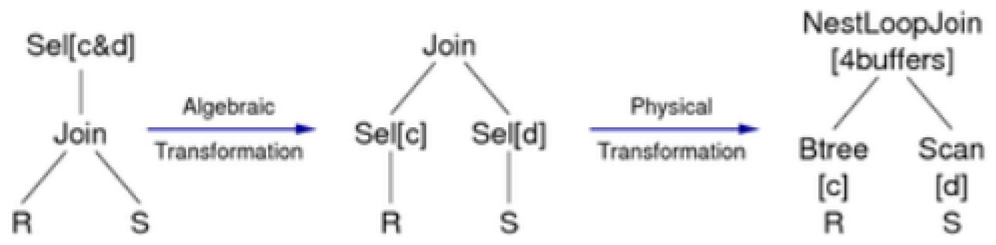
3.1.1 optimisation process

the optimisation process 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

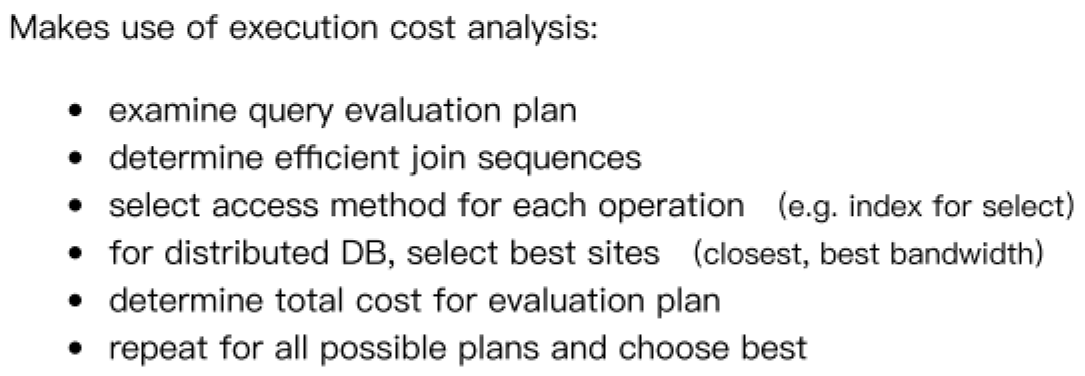
Example of optimisation transformations:



there are so much opstimize strategies:

in algebric optimization stage:

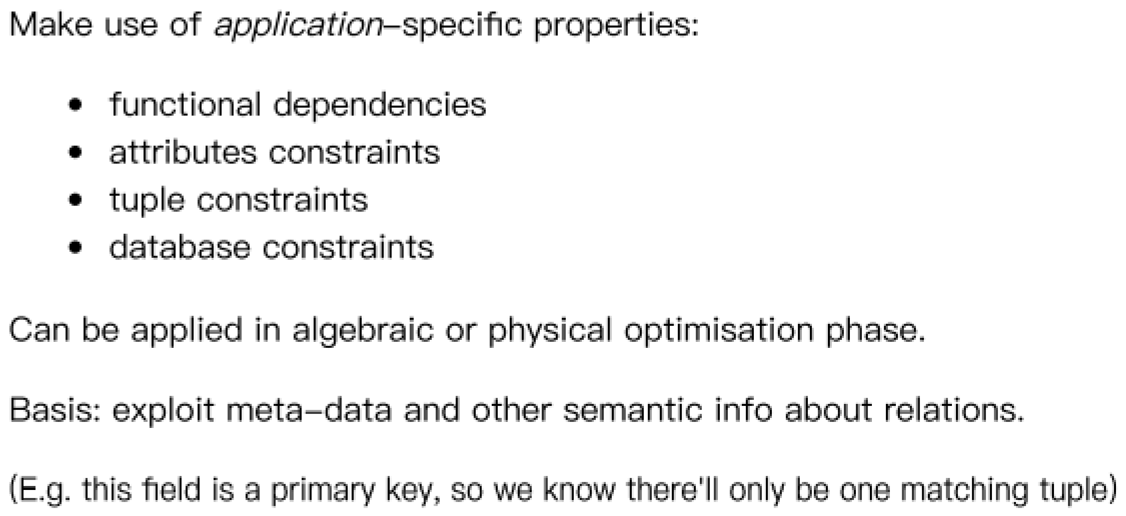
- make use of algebric equivalence, like apply selelct and project before join
- physical optimisation



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

- 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.

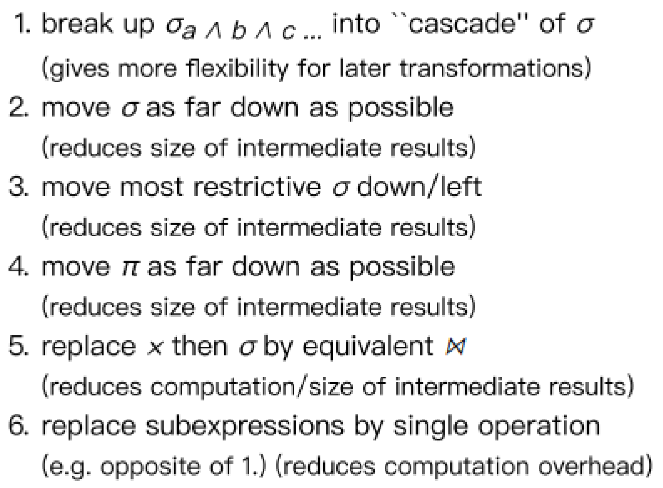
(E.g. this field is a primary key, so we know there'll only be one matching tuple)

3.1.2 Stages in Algebraic optimisation

start with RA expression:

- standardise
  - construct normal form of expression
- simplify
  - transform to eliminate redundancy (like eliminate redundancy in discrete math)
- ameliorate (improvement)
  - transform to improve effciency

in ameliorate stage, we use RA laws to promote efficiency:



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

example:

for a SQL like:

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

its RA expression is:

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

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

we can do optimization shown below:

there are two types of optimize operation:

- push selection
- split selection

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

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:

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

3.1.3 Physical (Cost-based) optimisation

in the phase of physical(cost-based) optimsation, we aim to find the cheapest **execution plan**

we need to do a series of determination for all possible execution plan:

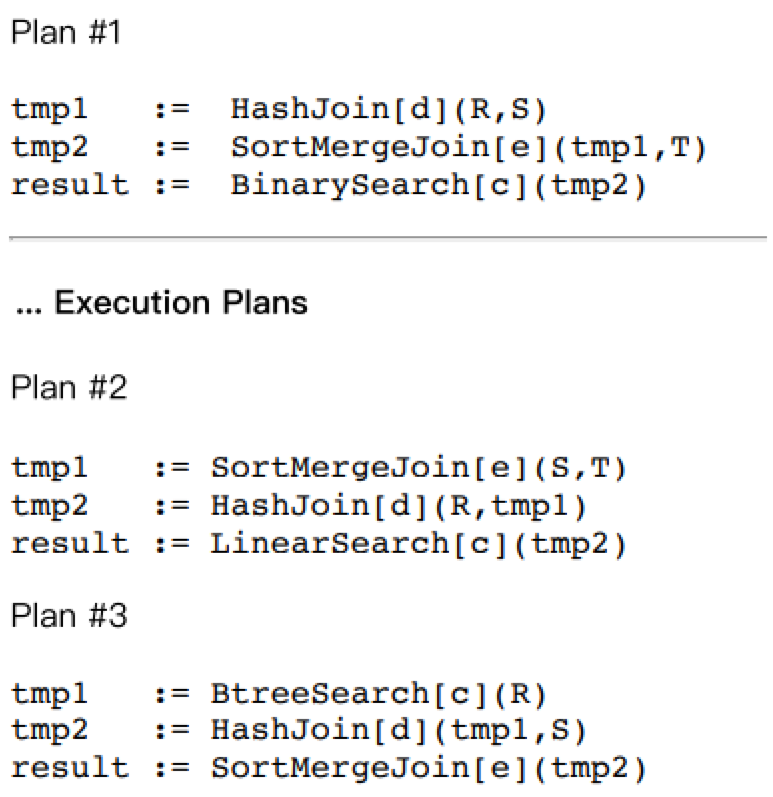
- order in which operations applied (execution plan)
- precisely how each operation done (map to DBMS ops)
- size of intermediate results (need to estimate these)

generally, there are multiple types of access for RA to conduct:

for examples, Join has: HashJoin, SortMergeJoin, etc

let's 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 the plans above produce same result, but have quite different costs

3.2 Cost-based Query Optimiser

the whole process of query optimiser is:

- receive RA tree from query complier
- use RA laws to rewrite rules to generate a new RA tree
- for each node in **RA tree**, **choose specific access method**

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
}   }
```

3.3 Cost Models and Analysis

the cost of evaluating a query is determined by:

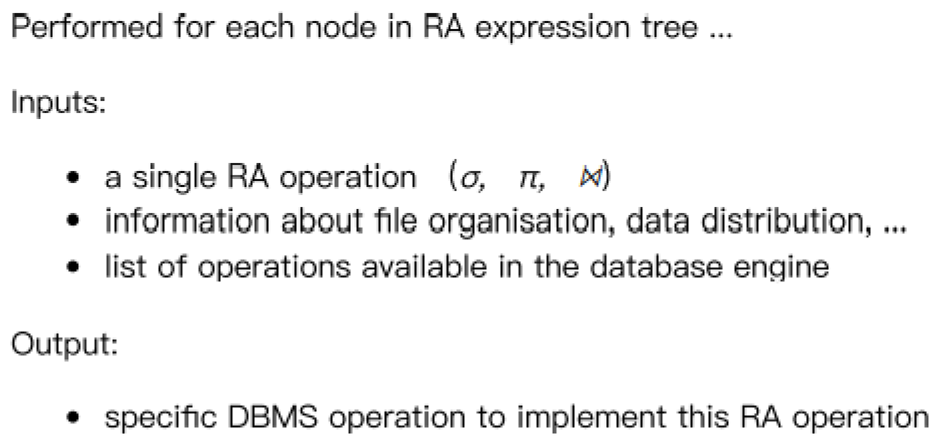
- size of relations
- access mechanisms
- size/number of main memory buffers

Analysis of costs involves estimating:

- size of intermediate results
- number of **secondary storage accesses**

3.4 choosing access methods (RelOps)

Performed for each node in RA expression tree


Performed for each node in RA expression tree ...

Inputs:

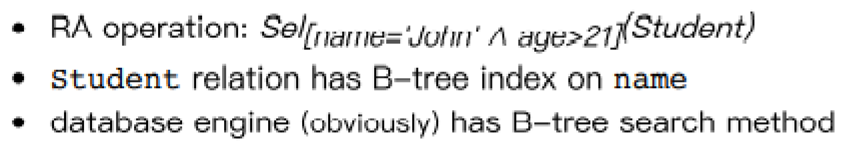
- a single RA operation $(\sigma, \quad \pi, \quad \bowtie)$
- information about file organisation, data distribution, ...
- list of operations available in the database engine

Output:

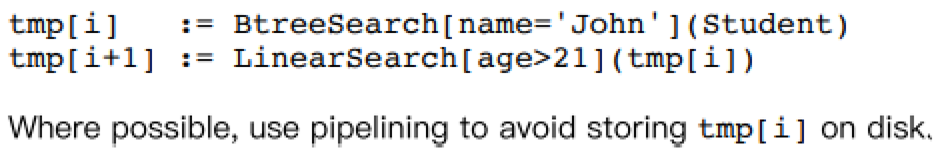
- specific DBMS operation to implement this RA operation

for example:

there are a RA operations : ***select * from Students where name=’John’ and age>21***


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

DBMS will use a pipeline to conduct this RA operations


```
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.

some Rules for some access methods in DBMS:

$\sigma_A = c^{(R)}$ means that select attribute A in relation R with condition c

such as:

```
σ Place = 'Mumbai' or Salary >= 1000000 (Citizen)
σ Department = 'Analytics'(σLocation = 'NewYork'(Manager))
```

Rules for choosing σ 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)`
- $\sigma_{A \geqslant c}(R)$ and `R` has clustered index on `A` $\Rightarrow$ `indexSearch[A=c+1](R)` then scan
- $\sigma_{A \geqslant c}(R)$ and `R` is hashed on `A` $\Rightarrow$ `linearSearch[A>=c](R)`

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)`

Notes:

- bnlJoin: block nested loop join
- inlJoin: index nested loop join
- smJoin: sort-merge join

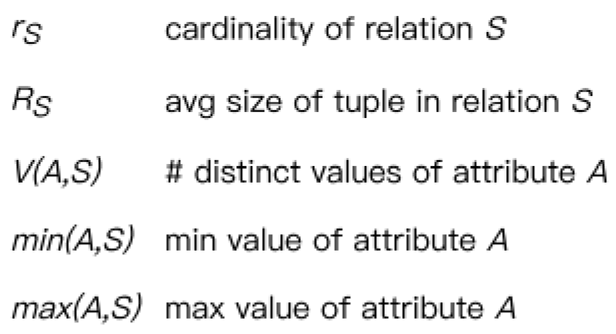
we can draw a conclusion that for different types of RA node, RA optimiser will generate special conduction plan.

3.5 cost estimation

query optimiser can't know the precise time of a plan, so that query optimiser estimate cost via

- cost of performing operation
- size of result

some statistical measures the optmiser will use:

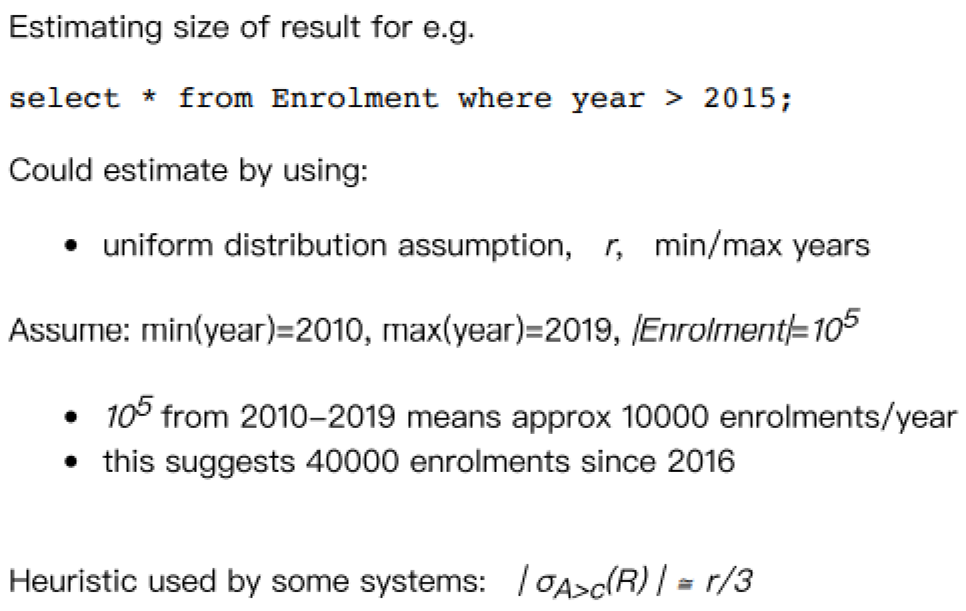


r_S	cardinality of relation S
R_S	avg size of tuple in relation S
$V(A,S)$	# distinct values of attribute A
$min(A,S)$	min value of attribute A
$max(A,S)$	max value of attribute A

3.6 Estimating selection result size

we **can use uniform distribution assumption** to estimate size of result

there are two examples:



Estimating size of result for e.g.

```
select * from Enrolment where year > 2015;
```

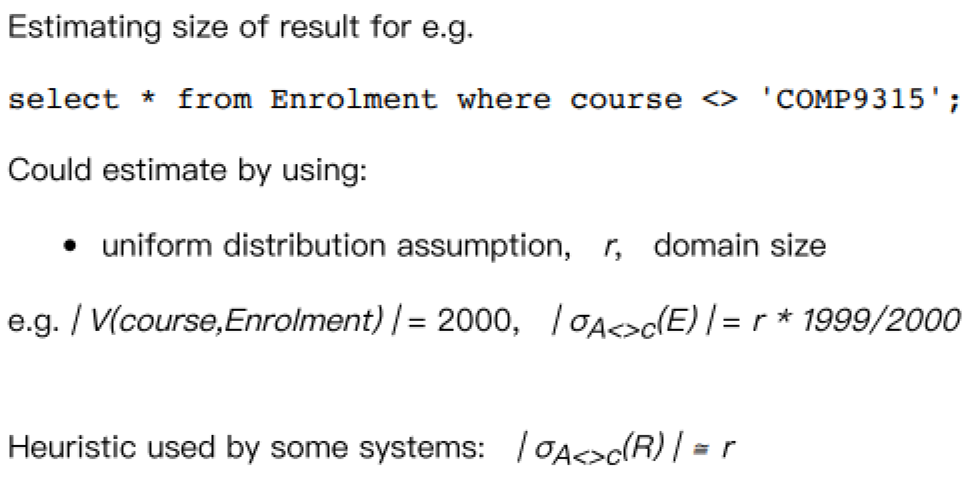
Could estimate by using:

- uniform distribution assumption, r, min/max years

Assume: min(year)=2010, max(year)=2019, $|Enrolment|=10^5$

- 10^5 from 2010–2019 means approx 10000 enrolments/year
- this suggests 40000 enrolments since 2016

Heuristic used by some systems: $|\sigma_{A>c}(R)| \approx r/3$


Estimating size of result for e.g.

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

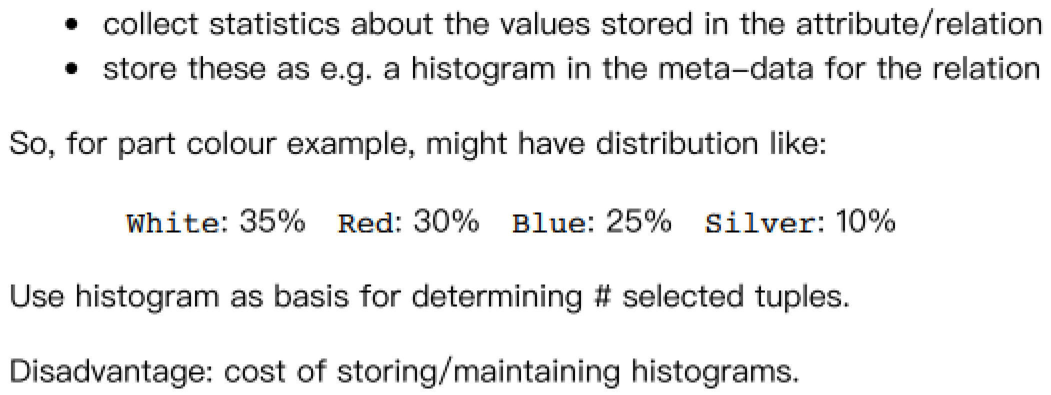
Could estimate by using:

- uniform distribution assumption, *r*, domain size

e.g. $| V(course,Enrolment) | = 2000$, $| \sigma_{A<>c}(E) | = r * 1999/2000$

Heuristic used by some systems: $| \sigma_{A<>c}(R) | \approx r$

so 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

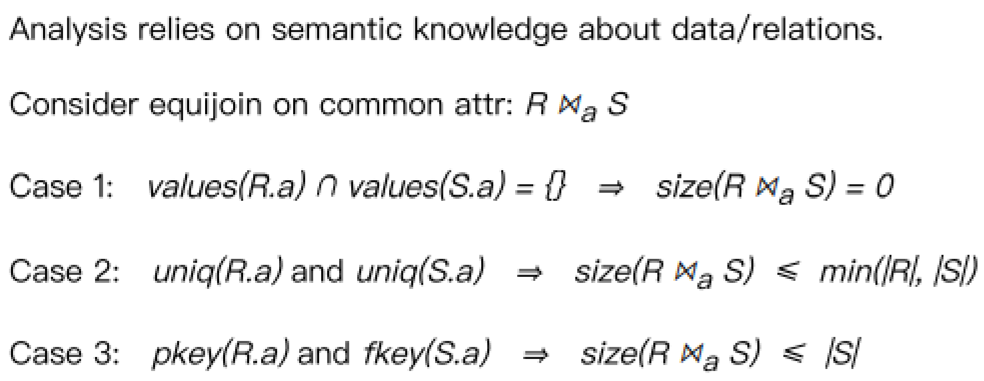
So, for part colour example, might have distribution like:

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

Use histogram as basis for determining # selected tuples.

Disadvantage: cost of storing/maintaining histograms.

3.7 Estimating Join result size


Analysis relies on semantic knowledge about data/relations.

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

Case 1: $values(R.a) \cap values(S.a) = \{\}$ $\Rightarrow$ $size(R \bowtie_a S) = 0$

Case 2: $uniq(R.a)$ and $uniq(S.a)$ $\Rightarrow$ $size(R \bowtie_a S) \leq min(|R|, |S|)$

Case 3: $pkey(R.a)$ and $fkey(S.a)$ $\Rightarrow$ $size(R \bowtie_a S) \leq |S|$

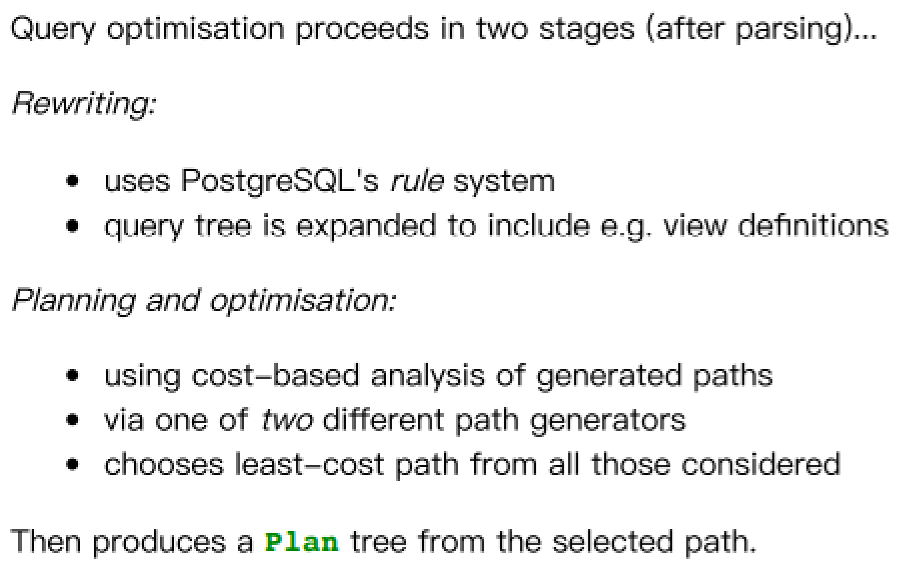
3.8 PostgreSQL query optimization

input: ***tree of query nodes*** returned by parser

output: ***tree of plan nodes*** used by query executor

intermediate data structures are ***trees of Path nodes***

- one path tree represents one evaluation order (a feasible conduct plan) for a query



Query optimisation proceeds in two stages (after parsing)...

Rewriting:

- uses PostgreSQL's *rule* system
- query tree is expanded to include e.g. view definitions

Planning and optimisation:

- using cost–based analysis of generated paths
- via one of *two* different path generators
- chooses least–cost path from all those considered

Then produces a `Plan` tree from the selected path.