COMP3311 Week 7 Lecture

DBMS Architecture

DBMS Architecture and Implementation

Aims:

- examine techniques used in implementation of DBMSs:
  - query processing (QP), transaction processing (TxP)
- use QP knowledge to make DB applications *efficient*
- use TxP knowledge to make DB applications *safe*

We also examine foundations:

- *relational algebra* ... basis for QP
- *transactions/serializability* ... basis for TxP

COMP3311: overview of the above; how to use them in app development
COMP9315: explore implementation details; modify PostgreSQL core

Database Application Performance

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

- what operations on the data does the application require

  (which queries, updates, inserts and how frequently is each one performed)

- how these operations might be implemented in the DBMS

  (data structures and algorithms for select, project, join, sort, ...)

- how much each implementation will cost

  (in terms of the amount of data transferred between memory and disk)

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

... Database Application Performance

Application programmer choices that affect query cost:

- how queries are expressed
  - generally join is faster than subquery
  - especially if subquery is correlated
  - avoid producing large intermediate tables *then* filtering
  - avoid applying functions in where/group-by clasues
- creating *indexes* on tables
  - index will speed-up filtering based on indexed attributes
  - indexes generally only effective for equality, gt/lt
  - indexes have update-time and storage overheads
  - only useful if filtering much more frequent than update

Database Query Processing

Whatever you do as a DB application programmer

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

Transformation is carried out by *query optimiser*

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

You have no control over the optimisation process

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

... Database Query Processing

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

```
select name from Employee
```

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

A query optimiser might use the strategy

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

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

... Database Query Processing

A different expression of the same query:

```
select name
from   Employee join WorksIn using (empid)
where  Employee.salary > 5000 and
       WorksIn.dept = 'Sales'
```

Query optimiser might use the strategy

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

Only examines Sales employees, and uses a simpler test

... Database Query Processing

A very poor expression of the query (correlated subquery):

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

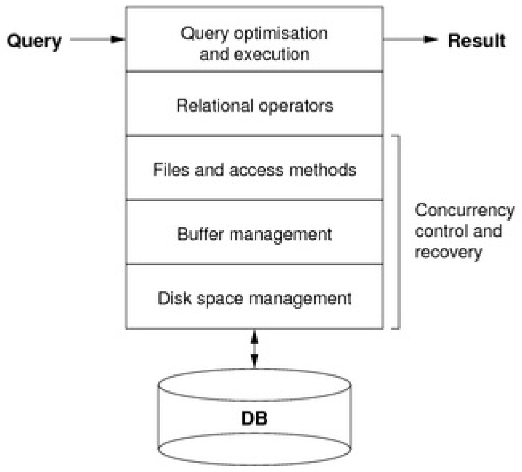
A query optimiser would be forced to use the strategy:

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

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

DBMS Architecture

Layers in a DB Engine (Ramakrishnan's View)



DBMS Components

File manager	manages allocation of disk space and data structures
Buffer manager	manages data transfer between disk and main memory

Query optimiser	translates queries into efficient sequence of relational ops
Recovery manager	ensures consistent database state after system failures
Concurrency manager	controls concurrent access to database
Integrity manager	verifies integrity constraints and user privileges

Relational Algebra

Relational Algebra

Relational algebra (RA) can be viewed as ...

- mathematical system for manipulating relations, or
- data manipulation language (DML) for the relational model

Relational algebra consists of:

- *operands*: relations, or variables representing relations
- *operators* that map relations to relations
- rules for combining operands/operators into expressions
- rules for evaluating such expressions

... Relational Algebra

Core relational algebra operations:

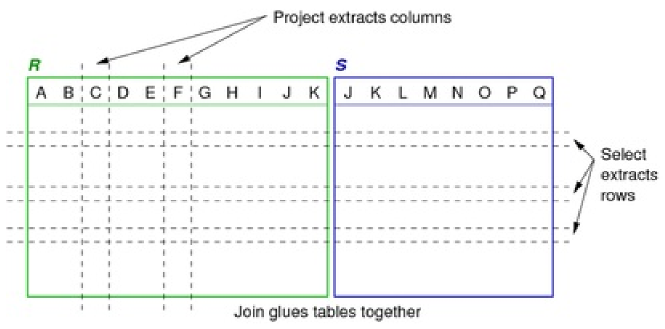
- *selection*: choosing a subset of rows
- *projection*: choosing a subset of columns
- *product*, *join*: combining relations
- *union*, *intersection*, *difference*: combining relations
- *rename*: change names of relations/attributes

Common extensions include:

- *aggregation*, *projection++*, *division*

... Relational Algebra

Select, project, join provide a powerful set of operations for building relations and extracting interesting data from them.



Adding set operations and renaming makes RA *complete*.

Notation

Standard treatments of relational algebra use Greek symbols.

We use the following notation (because it is easier to reproduce):

Operation	**Standard Notation**	**Our Notation**
Selection	$\sigma_{expr}(Rel)$	*Sel[expr](Rel)*
Projection	$\pi_{A,B,C}(Rel)$	*Proj[A,B,C](Rel)*
Join	$Rel_1 \bowtie_{expr} Rel_2$	*Rel*$_1$ *Join[expr]* *Rel*$_2$
Rename	$\rho_{schema} Rel$	*Rename[schema](Rel)*

For other operations (e.g. set operations) we adopt the standard notation.
Except when typing in a text file, where * = intersection, + = union

Describing RA Operations

We define the semantics of RA operations using

- "conditional set" expressions   e.g. *{ X | condition on X }*
- tuple notations:
    - *t[AB]*   (extracts attributes *A* and *B* from tuple *t*)
    - *(x,y,z)*   (enumerated tuples; specify attribute values)
- quantifiers, set operations, boolean operators

For each operation, we also describe it operationally:

- give an algorithm to compute the result, tuple-by-tuple

... Describing RA Operations

All RA operators return a result of type *relation*.

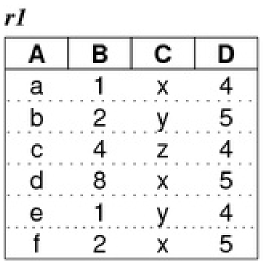
For convenience, we can name a result and use it later.

E.g.

```
Temp = R op1 S op2 T
Res  = Temp op3 Z
-- which is equivalent to
Res  = (R op1 S op2 T) op3 Z
```

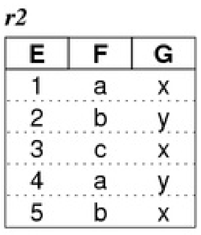
Each "intermediate result" has a well-defined schema.

Example Database #1



r1

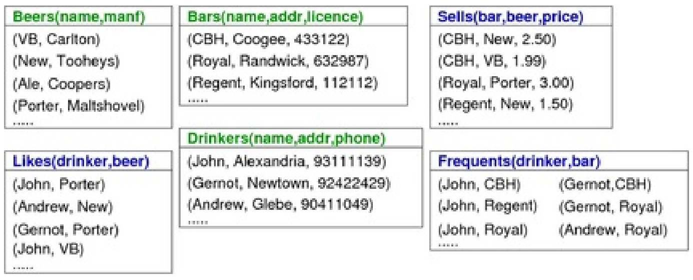
A	B	C	D
a	1	x	4
b	2	y	5
c	4	z	4
d	8	x	5
e	1	y	4
f	2	x	5



r2

E	F	G
1	a	x
2	b	y
3	c	x
4	a	y
5	b	x

Example Database #2



Beers(name,manf)
(VB, Carlton)
(New, Tooheys)
(Ale, Coopers)
(Porter, Maltshovel)
.....

Bars(name,addr,licence)
(CBH, Coogee, 433122)
(Royal, Randwick, 632987)
(Regent, Kingsford, 112112)
.....

Sells(bar,beer,price)
(CBH, New, 2.50)
(CBH, VB, 1.99)
(Royal, Porter, 3.00)
(Regent, New, 1.50)
.....

Likes(drinker,beer)
(John, Porter)
(Andrew, New)
(Gernot, Porter)
(John, VB)
.....

Drinkers(name,addr,phone)
(John, Alexandria, 93111139)
(Gernot, Newtown, 92422429)
(Andrew, Glebe, 90411049)
.....

Frequents(drinker,bar)
(John, CBH)   (Gernot,CBH)
(John, Regent)   (Gernot, Royal)
(John, Royal)   (Andrew, Royal)
.....

Rename

Rename provides "schema mapping".

If expression *E* returns a relation $R(A_1, A_2, \ldots A_n)$, then

$$Rename[S(B_1, B_2, \ldots B_n)](E)$$

gives a relation called *S*

- containing the same set of tuples as *E*
- but with the name of each attribute changed from A_i to B_i

Rename is like the identity function on the *contents* of a relation; it changes only the schema.

Rename can be viewed as a "technical" apparatus of RA.

Selection

Selection returns a subset of the tuples in a relation *r(R)* that satisfy a specified condition *C*.

$$\sigma_C(r) \quad = \quad Sel[C](r) \quad = \quad \{\ t \ \mid\ t \in r \wedge C(t)\ \}$$

C is a boolean expression on attributes in *R*.

Result size: $|\sigma_C(r)| \leq |r|$

Result schema: same as the schema of *r* (i.e. *R*)

Algorithmic view:

```
result = {}
for each tuple t in relation r
    if (C(t)) { result = result ∪ {t} }
```

... Selection

Example selections:

Sel [B = 1] (r1)

A	B	C	D
a	1	x	4
e	1	y	4

Sel [A=b ∨ A=c] (r1)

A	B	C	D
b	2	y	5
c	4	z	4

Sel [B ≥ D] (r1)

A	B	C	D
c	4	z	4
d	8	x	5

Sel [A = C] (r1)

A	B	C	D

Projection

Projection returns a set of tuples containing a subset of the attributes in the original relation.

$$\pi_X(r) \quad = \quad Proj[X](r) \quad = \quad \{\ t[X] \ \mid\ t \in r\ \}, \qquad \text{where } r(R)$$

X specifies a subset of the attributes of *R*.

Note that removing key attributes can produce duplicates.

In RA, duplicates are removed from the result *set*.
(In RDBMS's, duplicates are retained (i.e. they use bags, not sets))

Result size: $|\pi_X(r)| \leq |r|$ Result schema: *R'(X)*

Algorithmic view:

```
result = {}
for each tuple t in relation r
    result = result ∪ {t[X]}
```

... Projection

Example projections:

Proj [A,B,D] (r1)

A	B	D
a	1	4
b	2	5
c	4	4
d	8	5
e	1	4
f	2	5

Proj [B,D] (r1)

B	D
1	4
2	5
4	4
8	5

Proj [D] (r1)

D
4
5

Union

Union combines two *compatible* relations into a single relation via set union of sets of tuples.

$$r_1 \cup r_2 \;=\; \{\, t \mid t \in r_1 \lor t \in r_2 \,\}, \quad \text{where } r_1(R),\ r_2(R)$$

Compatibility = both relations have the same schema

Result size: $|r_1 \cup r_2| \;\le\; |r_1| + |r_2|$ Result schema: *R*

Algorithmic view:

```
$result$ = $r_1$
for each tuple $t$ in relation $r_2$
    $result$ = $result \cup \{t\}$
```

Intersection

Intersection combines two *compatible* relations into a single relation via set intersection of sets of tuples.

$$r_1 \cap r_2 \;=\; \{\, t \mid t \in r_1 \land t \in r_2 \,\}, \quad \text{where } r_1(R),\ r_2(R)$$

Uses same notion of relation compatibility as union.

Result size: $|r_1 \cup r_2| \;\le\; \min(|r_1|,|r_2|)$ Result schema: *R*

Algorithmic view:

```
$result$ = {}
for each tuple $t$ in relation $r_1$
    if ($t \in r_2$) { $result$ = $result \cup \{t\}$ }
```

Difference

Difference finds the set of tuples that exist in one relation but do not occur in a second *compatible* relation.

$$r_1 - r_2 \;=\; \{\, t \mid t \in r_1 \land \neg\, t \in r_2 \,\}, \quad \text{where } r_1(R),\ r_2(R)$$

Uses same notion of relation compatibility as union.

Note: tuples in r_2 but not r_1 do not appear in the result

- i.e. set difference != complement of set intersection

Algorithmic view:

```
$result$ = {}
for each tuple $t$ in relation $r_1$
    if (!($t \in r_2$)) { $result$ = $result \cup \{t\}$ }
```

... Difference

Example difference:

s1 = Sel [B = 1] (r1)

A	B	C	D
a	1	x	4
e	1	y	4

s2 = Sel [C = x] (r1)

A	B	C	D
a	1	x	4
d	8	x	5

s1 - s2

A	B	C	D
e	1	y	4

s2 - s1

A	B	C	D
d	8	x	5

Clearly, difference is not symmetric.

Product

Product (Cartesian product) combines information from two relations pairwise on tuples.

$r \times s \quad = \quad \{\ (t_1 : t_2)\ \mid\ t_1 \in r \wedge t_2 \in s\ \},$ where $r(R), s(S)$

Each tuple in the result contains all attributes from *r* and *s*, possibly with some fields renamed to avoid ambiguity.

If $t_1 = (A_1...A_n)$ and $t_2 = (B_1...B_n)$ then $(t_1{:}t_2) = (A_1...A_n,B_1...B_n)$

Note: relations do not have to be union-compatible.

Result size is *large*: $|r \times s| = |r|.|s|$ Schema: $R \cup S$

Algorithmic view:

```
result = {}
for each tuple $t_1$ in relation r
    for each tuple $t_2$ in relation s
        result = result ∪ {($t_1$:$t_2$)} }
```

Natural Join

Natural join is a specialised product:

- containing only pairs that match on *common* attributes
- with one of each pair of common attributes eliminated

Consider relation schemas *R(ABC..JKLM)*, *S(KLMN..XYZ)*.

The natural join of relations *r(R)* and *s(S)* is defined as:

$r \bowtie s \quad = \quad r\ Join\ s \quad =$
$\{\ (t_1[ABC..J] : t_2[K..XYZ])\ \mid\ t_1 \in r \wedge t_2 \in s \wedge match\ \}$

where $match \quad = \quad t_1[K] = t_2[K] \wedge t_1[L] = t_2[L] \wedge t_1[M] = t_2[M]$

Algorithmic view:

```
result = {}
for each tuple $t_1$ in relation r
   for each tuple $t_2$ in relation s
      if (matches($t_1$,$t_2$))
         result = result ∪ {combine($t_1$,$t_2$)}
```

... Natural Join

Natural join can be defined in terms of other RA operations:

r Join s = *Proj[R ∪ S] (Sel[match] (r × s))*

We assume that the union on attributes eliminates duplicates.

If we wish to join relations, where the common attributes have different names, we rename the attributes first.

E.g. *R(ABC)* and *S(DEF)* can be joined by

R Join Rename[S(DCF)](S)

Note: $|r \bowtie s| \ll |r \times s|$, so *join* not implemented via *product*.

... Natural Join

Example (assuming that *A* and *F* are the common attributes):

r1 Join r2

A	B	C	D	E	G
a	1	x	4	1	x
a	1	x	4	4	y
b	2	y	5	2	y
b	2	y	5	5	x
c	4	z	4	3	x

Strictly the above was: *r1 Join Rename[r2(E,A,G)](r2)*

Theta Join

The *theta join* is a specialised product containing only pairs that match on a supplied condition *C*.

$r \bowtie_C s \;=\; \{ (t_1 : t_2) \mid t_1 \in r \wedge t_2 \in s \wedge C(t_1 : t_2) \}$,
where *r(R),s(S)*

Examples: *(r1 Join[B>E] r2)* ... *(r1 Join[E<D∧C=G] r2)*

All attribute names are required to be distinct (cf natural join)

Can be defined in terms of other RA operations:

$r \bowtie_C s \;=\; r\ Join[C]\ s \;=\; Sel[C]\ (\ r \times s\ )$

Note that $r \bowtie_{true} s \;=\; r \times s$.

... Theta Join

Example theta join:

r1 Join[D < E] r2

A	B	C	D	E	F	G
a	1	x	4	5	b	x
c	4	z	4	5	b	x
e	1	y	4	5	b	x

r1 Join[B > 1 ∧ D < E] r2

A	B	C	D	E	F	G
c	4	z	4	5	b	x

(Theta join is the most frequently-used join in SQL queries)

Division

Consider two relation schemas *R* and *S* where $S \subset R$.

The *division* operation is defined on instances *r(R)*, *s(S)* as:

$r\,/\,s \;=\; r\ Div\ s \;=\; \{\ t \mid t \in r[R\text{-}S] \wedge satisfy\ \}$

where $satisfy \;=\; \forall\, t_s \in S\ (\ \exists\, t_r \in R\ (\ t_r[S] = t_s \wedge t_r[R\text{-}S] = t\ )\ )$

Operationally:

- consider each subset of tuples in *R* that match on *t[R-S]*
- for this subset of tuples, take the *t[S]* values from each
- if this covers all tuples in *S*, then include *t[R-S]* in the result

... Division

Example:

R

A	B
4	x
4	y
4	z
5	x
5	y
5	z

R'

A	B
4	x
4	y
4	z
5	x
5	z

S

B
x
y

R / S

A
4
5

R' / S

A
4

... Division

Division handles queries that include the notion "for all".

E.g. Which beers are sold in all bars?

We can answer this as follows:

- generate a relation of beers and bars where they are sold
- generate a relation of all bars
- find which beers appear in tuples with *every* bar

... Division

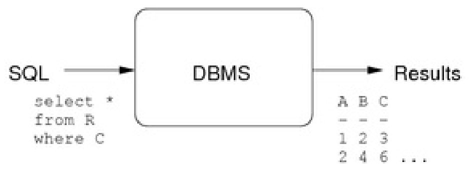
RA operations for answering the query:

- beers and bars where they are sold (ignore prices)
  - *r1 = Proj[beer,bar](Sold)*
- bar names
  - *r2 = Rename[r2(bar)](Proj[name](Bars))*
- beers that appear in tuples with every bar
  - *res   = r1 Div r2*

Query Processing

Query Processing

So far, have viewed query evaluation as a black box:



Query processing is the study of techniques for query evaluation.

... Query Processing

Goal of query processing:

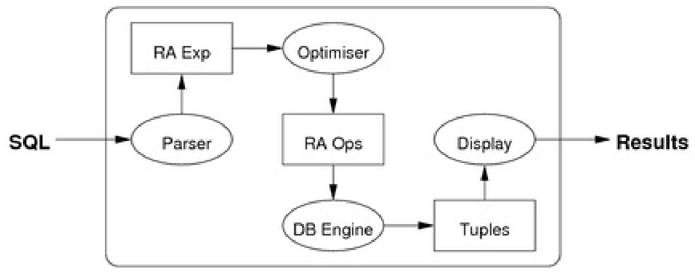
- find/build a set of tuples satisfying some condition
- tuples may be constructed using values from multiple tables

We talk about *query* processing ...

- but the considerations also apply to DB update operations
- INSERT affects only one tuple (and, maybe, indexes)
- DELETE does *find*-then-remove, so is query-like
- UPDATE does *find*-then-change, so is query-like
- one difference: update operations act on a single table

... Query Processing

Inside the query evaluation box:



... Query Processing

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

selection (σ)	projection (π)	join (⋈, ×)
union (∪)	intersection (∩)	difference (-)
sort	insert	delete

For each of these operations:

- various data structures and algorithms are available
- DBMSs may provide only one, or may provide a choice

Query optimiser chooses best methods and order to apply them.

... Query Processing

What is the "best" method for evaluating a query?

Generally, *best* = lowest cost = fastest evaluation time

Cost is measured in terms of pages read/written

- data is stored in fixed-size blocks (e.g. 4KB)
- data transferred disk↔memory in whole blocks
- cost of disk↔memory transfer is highest cost in system
- processing in memory is very fast compared to I/O

Mapping SQL to RA

Mapping SQL to relational algebra, e.g.

```
-- schema: R(a,b) S(c,d)
select a as x
from   R join S on (b=c)
where  d = 100
-- mapped to
Tmp = Sel[d=100] (R join[b=c] S)
Res = Rename[a->x](Proj[a] Ttmp)
```

In general:

- SELECT clause becomes *projection*
- WHERE condition becomes *selection* or *join*
- FROM clause becomes *join*

Mapping Example

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

Can be expressed in SQL as

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

and might be compiled to

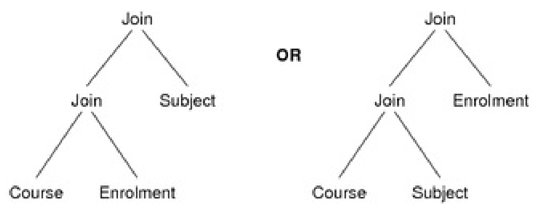
Result = Project'$_{[s.code]}$*(GroupSelect*$_{[size>100]}$*(GroupBy*$_{[id]}$ *(JoinRes)))*

where

JoinRes = Subject Join$_{[s.id=c.subject]}$ *(Course Join*$_{[c.id=e.course]}$ *Enrolment)*

... Mapping Example

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

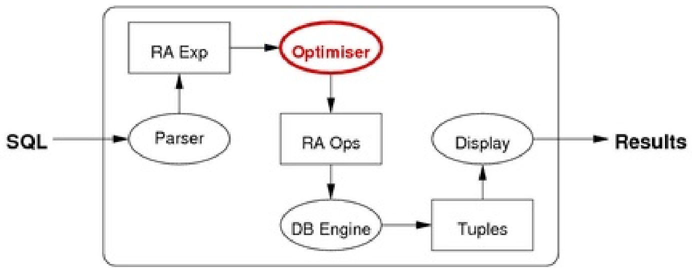


Which is better? ... The query optimiser works this out.

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

Query Optimisation

Phase 2: mapping RA expression to (efficient) query plan



... Query Optimisation

The query optimiser start with an RA expression, then

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

The cost of evaluating a query is determined by:

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

Analysis of costs involves *estimating*:

- the size of intermediate results
- then, based on this, cost of secondary storage accesses

... Query Optimisation

An *execution plan* is a sequence of relational operations.

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

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

or

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

or

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

All produce same result, but have different costs.

Implementations of RA Ops

Sorting (quicksort, etc. are not applicable)

- external merge sort (cost $O(N\log_B N)$ with *B* memory buffers)

Selection (different techniques developed for different query types)

- sequential scan (worst case, cost *O(N)*)
- index-based (e.g. B-trees, cost *O(logN)*, tree nodes are pages)
- hash-based (*O(1)* best case, only works for equality tests)

Join (fast joins are critical to success to erlational DBMSs)

- nested-loop join (cost *O(N.M)*, buffering can reduce to *O(N+M)*)
- sort-merge join (cost *O(NlogN+MlogM)*)
- hash-join (best case cost *O(N+M.N/B)*, with *B* memory buffers)