Database management II.

Join Execution
Database optimisation

Gergely Lukács

Pázmány Péter Catholic University

Faculty of Information Technology and Bionics

Budapest, Hungary

lukacs@itk.ppke.hu

Index Usage - Query Selectivity

Check the following query with different constant values in the `WHERE` condition (i.e., with different selectivities)!

```
SELECT Count(updated_at)
FROM   ad18___db.historyitems_large
WHERE  user_id < 50;
```

1. When (at which selectivities) does the DBMS use the index? How does the execution cost changes? (Create a notice in form of a table: paramater value, query selectivity, index usage (yes/no), execution cost.)
2. What changes, when querying COUNT(duration)? When querying SUM(duration)? Why?
   (In both cases, please use the `/*+ NO_REWRITE */` optimizer hint!)

AD18___DB.HISTORYITEMS_LARGE

		Column	Type
P	*	ID	NUMBER (38)
	*	USER_ID	NUMBER (*,0)
	*	AUDIO_ID	NUMBER (*,0)
	*	STARTED_AT	TIMESTAMP WITH TIME ZONE
	*	STARTED_AT_ZONE	NUMBER (*,0)
	*	DURATION	NUMBER (*,0)
	*	RATING	NUMBER (*,0)
	*	CREATED_AT	TIMESTAMP WITH TIME ZONE
		UPDATED_AT	TIMESTAMP WITH TIME ZONE
	*	CREATED_AT_ZONE	NUMBER (*,0)
	*	UPDATED_AT_ZONE	NUMBER (*,0)
	*	FLAGS	NUMBER (*,0)

HISTORYITEMSLARGE_PKEY (ID)

- HISTORYITEMS_LARGE_AUDIO_II (AUDIO_ID)
- HISTORYITEMSLARGE_PKEY (ID)
- HISTORYITEMS_LARGE_STARTED_I (SYS_EXTRACT_UTC("STARTED_AT"))
- HISTORYITEMS_LARGE_STARTEDID_I (SYS_EXTRACT_UTC("STARTED_AT"), ID)
- HISTORYITEMS_LARGE_USRSTRTD_I (USER_ID, SYS_EXTRACT_UTC("STARTED_AT"))

Data Access Structures, Indexes

4

```sql
select user_id, count(*) from lukacs.historyitems_large
 group by user_id;
```

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		9983	2382
HASH (GROUP BY)		9983	2382
INDEX (FAST FULL SCAN)	HISTORYITEMS_LARGE_USRSTRTD_I	2971500	1886

```sql
select user_id, count(*) from lukacs.historyitems_large
 group by user_id order by user_id;
```

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		9983	2382
SORT (GROUP BY)		9983	2382
INDEX (FAST FULL SCAN)	HISTORYITEMS_LARGE_USRSTRTD_I	2971500	1886

Data warehouse queries

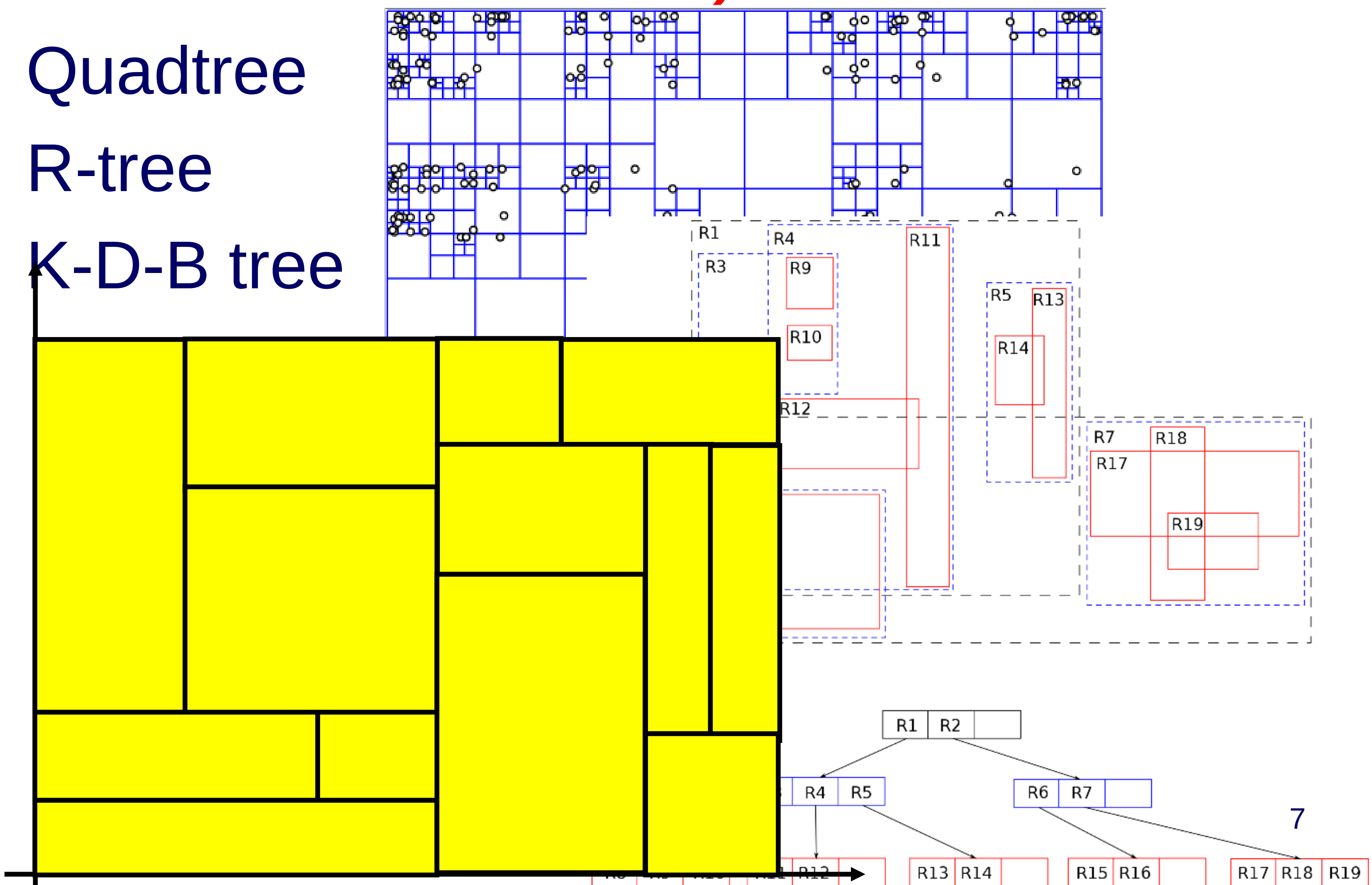
- Large number of records
- Ad-hoc queries, multiple dimensions
- Aggregations

- Read operations almost exclusively

- Bitmap index
- Materialized view + query rewrite

Geographic databases (> 1D data)

- Quadtree
- R-tree
- K-D-B tree



Join execution

8

Join execution

- Relational databases, normalisation => large number of joins
- Join – very expensive
- Several ways to execute

Nested loop join

10

Notes

- We are again considering “IO aware” algorithms: ***care about disk IO***

- Given a relation R, let:
  - T(R) = # of tuples in R
  - P(R) = # of pages in R

Recall that we read / write entire pages with disk IO

Nested Loop Join (NLJ)

```
Compute R ⋈ S on A:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Nested Loop Join (NLJ)

```
Compute R ⋈ S on A:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Cost:

P(R)

1. **Loop over the tuples in R**

Note that our IO cost is based on the number of ***pages*** loaded, not the number of tuples!

Nested Loop Join (NLJ)

```
Compute R ⋈ S on A:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Cost:

$P(R) + T(R)*P(S)$

1. Loop over the tuples in R
2. **For every tuple in R, loop over all the tuples in S**

Have to read ***all of S*** from disk for ***every tuple in R!***

Nested Loop Join (NLJ)

Compute $R \bowtie S$ *on* A:

```
for r in R:
  for s in S:
    if r[A] == s[A]:
      yield (r,s)
```

Cost:

$$P(R) + T(R)*P(S)$$

1. Loop over the tuples in R
2. For every tuple in R, loop over all the tuples in S
3. **Check against join conditions**

Note that NLJ can handle things other than equality constraints… just check in the *if* statement!

Nested Loop Join (NLJ)

```
Compute R ⋈ S on A:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

What would ***OUT*** be if our join condition is trivial (*if TRUE)?*

OUT could be bigger than P(R)*P(S)… but usually not that bad

Cost:

P(R) + T(R)*P(S) + OUT

1. Loop over the tuples in R
2. For every tuple in R, loop over all the tuples in S
3. Check against join conditions
4. **Write out (to page, then when page full, to disk)**

Nested Loop Join (NLJ)

Compute $\mathrm{R} \bowtie S \text{ on } A$:

```
for r in R:
  for s in S:
    if r[A] == s[A]:
      yield (r,s)
```

<u>Cost:</u>

P(R) + T(R)*P(S) + OUT

What if R ("outer") and S ("inner") switched?

P(***S***) + T(***S***)*P(***R***) + OUT

Outer vs. inner selection makes a huge difference- DBMS needs to know which relation is smaller!

Block Nested Loop Join
(IO-Aware variant)

Block Nested Loop Join (BNLJ)

```
Compute R ⋈ S on A:
  for each B-1 pages pr of R:
    for page ps of S:
      for each tuple r in pr:
        for each tuple s in ps:
          if r[A] == s[A]:
            yield (r,s)
```

Given ***B+1*** pages of memory

Cost:

$P(R)$

1. **Load in B-1 pages of R at a time (leaving 1 page each free for S & output)**

Note: There could be some speedup here due to the fact that we're reading in multiple pages sequentially however we'll ignore this here!

Block Nested Loop Join (BNLJ)

Given ***B+1*** pages of memory

Cost:

$$P(R) + \frac{P(R)}{B-1} P(S)$$

```
Compute R ⋈ S on A:
  for each B-1 pages pr of R:
    for page ps of S:
      for each tuple r in pr:
        for each tuple s in ps:
          if r[A] == s[A]:
            yield (r,s)
```

1. Load in B-1 pages of R at a time (leaving 1 page each free for S & output)

2. **For each (B-1)-page segment of R, load each page of S**

Block Nested Loop Join (BNLJ)

Given ***B+1*** pages of memory

Cost:

$$\mathrm{P}(R) + \frac{P(R)}{B-1} P(S)$$

```
Compute R ⋈ S on A:
  for each B-1 pages pr of R:
    for page ps of S:
      for each tuple r in pr:
        for each tuple s in ps:
          if r[A] == s[A]:
            yield (r,s)
```

1. Load in B-1 pages of R at a time (leaving 1 page each free for S & output)
2. For each (B-1)-page segment of R, load each page of S
3. **Check against the join conditions**

BNLJ can also handle non-equality constraints

Block Nested Loop Join (BNLJ)

Given ***B+1*** pages of memory

Cost:

$$P(R) + \frac{P(R)}{B-1} P(S) + \text{OUT}$$

```
Compute R ⋈ S on A:
  for each B-1 pages pr of R:
    for page ps of S:
      for each tuple r in pr:
        for each tuple s in ps:
          if r[A] == s[A]:
            yield (r,s)
```

1. Load in B-1 pages of R at a time (leaving 1 page each free for S & output)
2. For each (B-1)-page segment of R, load each page of S
3. Check against the join conditions
4. **Write out**

Again, ***OUT*** could be bigger than P(R)*P(S)… but usually not that bad

BNLJ vs. NLJ: Benefits of IO Aware

- In BNLJ, by loading larger chunks of R, we minimize the number of full *disk reads* of S
  - We only read all of S from disk for ***every (B-1)-page segment of R***!
  - Still the full cross-product, but more done only *in memory*

NLJ

P(R) + T(R)*P(S) + OUT

BNLJ

$P(R) + \frac{P(R)}{B-1} P(S)$ + OUT

BNLJ is faster by roughly $\frac{(B-1)T(R)}{P(R)}$!

BNLJ vs. NLJ: Benefits of IO Aware

- Example:
  - R: 500 pages
  - S: 1000 pages
  - 100 tuples / page
  - We have 12 pages of memory (B = 11)

Ignoring OUT here…

- NLJ: Cost = 500 + **50,000*1000** = **50 Million IOs ~= <u>140 hours</u>**

- BNLJ: Cost = 500 + $\frac{500*1000}{10}$ = **50 *Thousand* IOs ~= <u>0.14 hours</u>**

A very real difference from a small change (IO-awareness) in the algorithm!

Nested Loops Join cont.

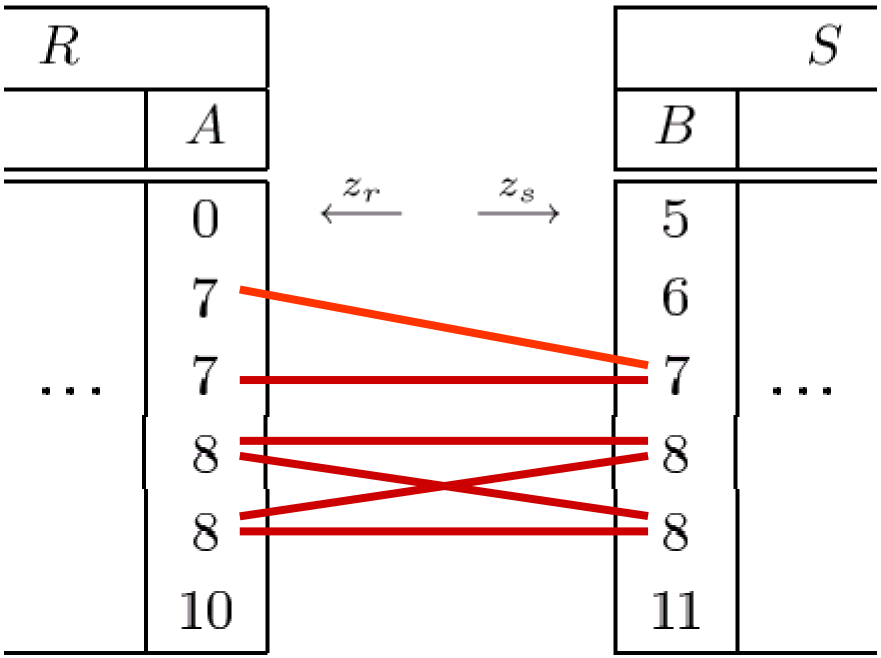
- **index nested loops join** …
- **temporary index nested loops join** …

Nested Loops Join

- Pros: any join condition
- Cons: expensive
  - Better, if
    - Tables fit in memory
    - Smaller table fits in memory (-> inner table)

Sort-Merge-Join

- Sort relations by join attribute,
  Interleaved linear scan



Sort-merge join

```
function sortMerge(relation left, relation
  right, attribute a)
     var relation output
     var list left_sorted := sort(left, a)
     var list right_sorted := sort(right, a)
     var left_key
     var right_key
     var set left_subset
     var set right_subset
…
```

Sort-merge join 2

```
…
advance(left_subset, left_sorted, left_key, a)
     advance(right_subset, right_sorted, right_key, a)
     while not empty(left_subset) and not empty(right_subset)
         if left_key = right_key
             add cross product of left_subset
                    and right_subset to output
             advance(left_subset, left_sorted, left_key, a)
             advance(right_subset,right_sorted, right_key, a)
         else if left_key < right_key
            advance(left_subset, left_sorted, left_key, a)
         else // left_key > right_key
            advance(right_subset, right_sorted, right_key, a)
     return output
…
```

Sort-merge join 3

```
…
function advance(subset, sorted, key, a)
     key = sorted[1].a
     subset = emptySet
     while not empty(sorted) and
                                 sorted[1].a = key
         insert(subset, sorted[1])
         remove first element from sorted
```

Sort-merge join

- Join operators
  - Also <, <=, >, >=

Simple Hash Join (1)

- Partition relation R in R_1, R_2, ..., R_p with Hash-function h so that for all tuples $r \in R_i$ h($r.A$) $\in H_i$.
- Scan relation S, apply for each tuple $s \in S$ the hash function h. Is $h(s.B)$ in H_i, search there for fitting r

Build-phase

```
repeat
    begin
    while memory not exhausted do
      insert next(R) into partition[h(next(R).A)];
    foreach s ∈ S do
      foreach r ∈ partition[h(s.B)] do
            if r.A = s.B Res := Res ∪ (r ∘ s)
    end
until R exhausted
```

Probe-phase

Hash Join

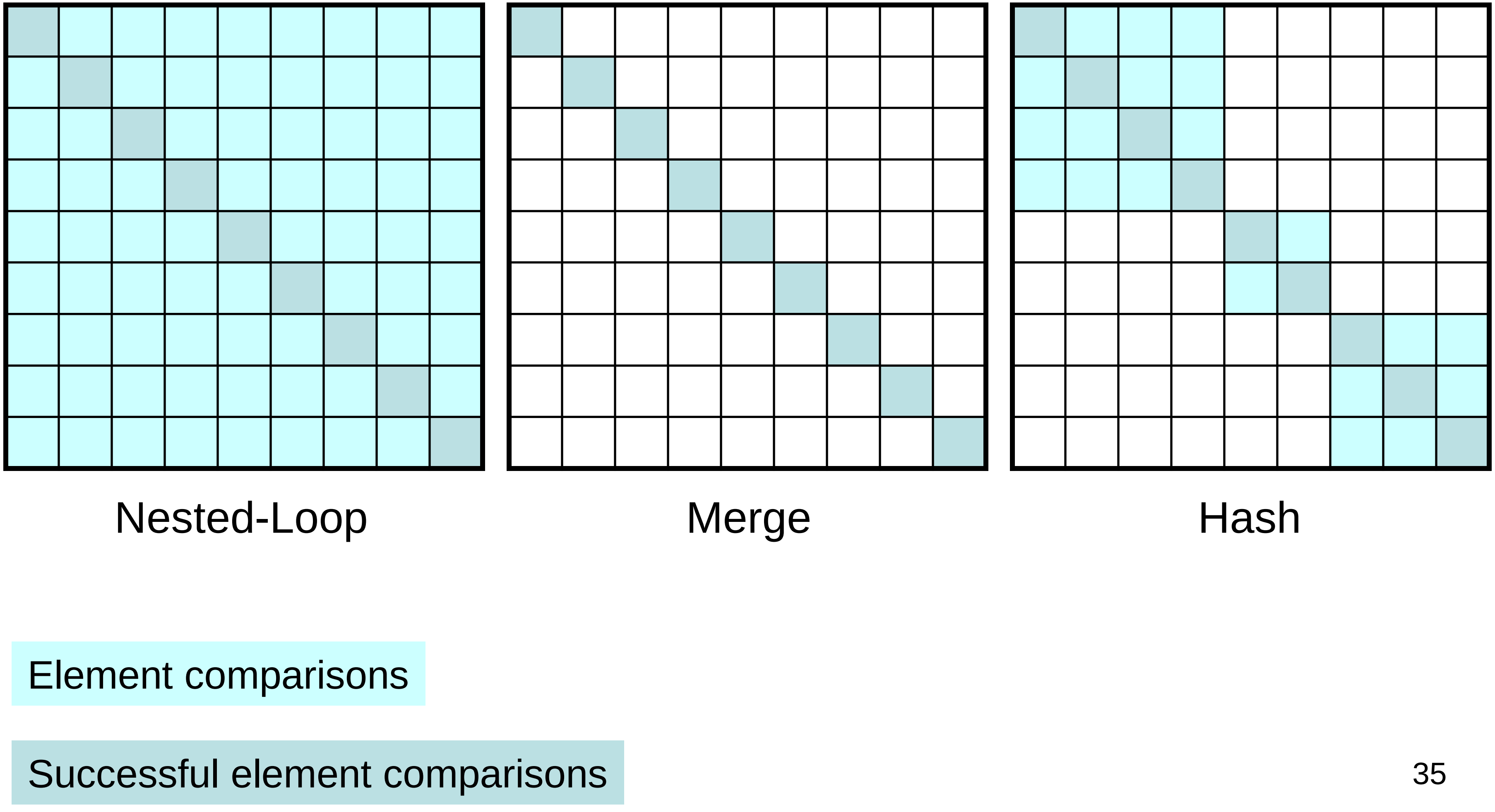
- Size of hash-table > available RAM
  - Multiple scans! (Very) slow!
  - (Small increase of data can cause large increase in run time!)

- (Grace hash join)
  - Partitioning R and S via a hash function
  - Join only within partitions
  - avoids multiple rescanning of entire *S* relation

Hash Join

- Pros
  - quick (if memory sufficient)!
- Cons
  - Memory requirement
  - Join operator: only „="

Resource requirement, complexity (comparison step)



Comparison

- Size and schema of relations in join
- Available indices
- Other operations (presorted intermediate results)
- Available memory
- Join-type (natural join, theta-join, equijoin; outer join?)

Cost based query optimization

37

Query execution

- *SQL: very high level, declarative*
  - *What* to retrieve, not *how*!
- SQL query is translated by the *query processor* into a low level program – the *execution plan*
- An execution plan is a program that can be executed to get the answer to the query.

Key Idea: Algebraic Optimization

- N = ((n*5)+(n*2)+0)/1
- Algebraic laws:
  1. (+) identity: x+0=x
  2. (/) identity: x/1=x
  3. (*) distributes: (y*x)+(z*x)=(y+z)*x
  4. (*) commutes: y*x=x*y
- Rules 1, 3, 4,2:
  - N=(5+2)*n

Relational algebra Equivalences

1. Selection $\sigma_{c_1 \wedge c_2 \wedge \ldots \wedge c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))\ \ldots))$

2. σ is commutative $\sigma_{c_1}(\sigma_{c_2}((R)) \equiv \sigma_{c_2}(\sigma_{c_1}((R))$

3. π -cascades: If $L_1 \subseteq L_2 \subseteq \ldots \subseteq L_n$, then

$$\pi_{L_1}(\pi_{L_2}(\ldots(\pi_{L_n}(R))\ \ldots)) \equiv \pi_{L_1}(R)$$

4. Changing σ and π

   If the selection only refers to the projected attributes $A_1, \ldots, A_n$,

   selection and projection can be exchanged

$$\sigma_c(\pi_{A_1, \ldots, A_n}(R)) \equiv \pi_{A_1, \ldots, A_n}(\sigma_c(R))$$

Equivalences 2

5. $\times$, $\bowtie$, $\cup$ and $\cap$ are commutative

6. A Cartesic product, followed by a selection referring to both operands can be replaced by a join.

$$\sigma_c(R \times S) \equiv R \bowtie_c S$$

....

Equivalences…

$\sigma_{c_1 \wedge c_2 \wedge \ldots \wedge c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R)) \ldots))$

$\sigma_{c_1}(\sigma_{c_2}((R)) \equiv \sigma_{c_2}(\sigma_{c_1}((R))$

$\pi_{L_1}(\pi_{L_2}(\ldots(\pi_{L_n}(R)) \ldots)) \equiv \pi_{L_1}(R)$

$\sigma_c(\pi_{A_1, \ldots, A_n}(R)) \equiv \pi_{A_1, \ldots, A_n}(\sigma_c(R))$

$R \,\Phi\, S \equiv S \,\Phi\, R \;(\bowtie, \times, \cup, \cap)$

$\sigma_c(R \times S) \equiv R \bowtie_c S$

$\sigma_c(R \,\Phi\, S) \equiv \sigma_c(R) \,\Phi\, S \;(\bowtie, \times)$

$\sigma_c(R \,\Phi\, S) \equiv \sigma_{c_1}(R) \,\Phi\, \sigma_{c_2}(S) \;(\bowtie, \times)$

$\pi_L(R \bowtie_c S) \equiv (\pi_{A_1, \ldots, A_n}(R)) \bowtie_c (\pi_{B_1, \ldots, B_n}(S))$

$\pi_L(R \bowtie_c S) \equiv \pi_L((\pi_{A_1, \ldots, A_n, A_1', \ldots, A_n'}(R)) \bowtie_c (\pi_{B_1, \ldots, B_n, B_1', \ldots, B_n'}(S)))$

$(R \,\Phi S) \,\Phi T \equiv R \,\Phi (S \,\Phi T) \;(\bowtie, \times, \cup, \cap)$

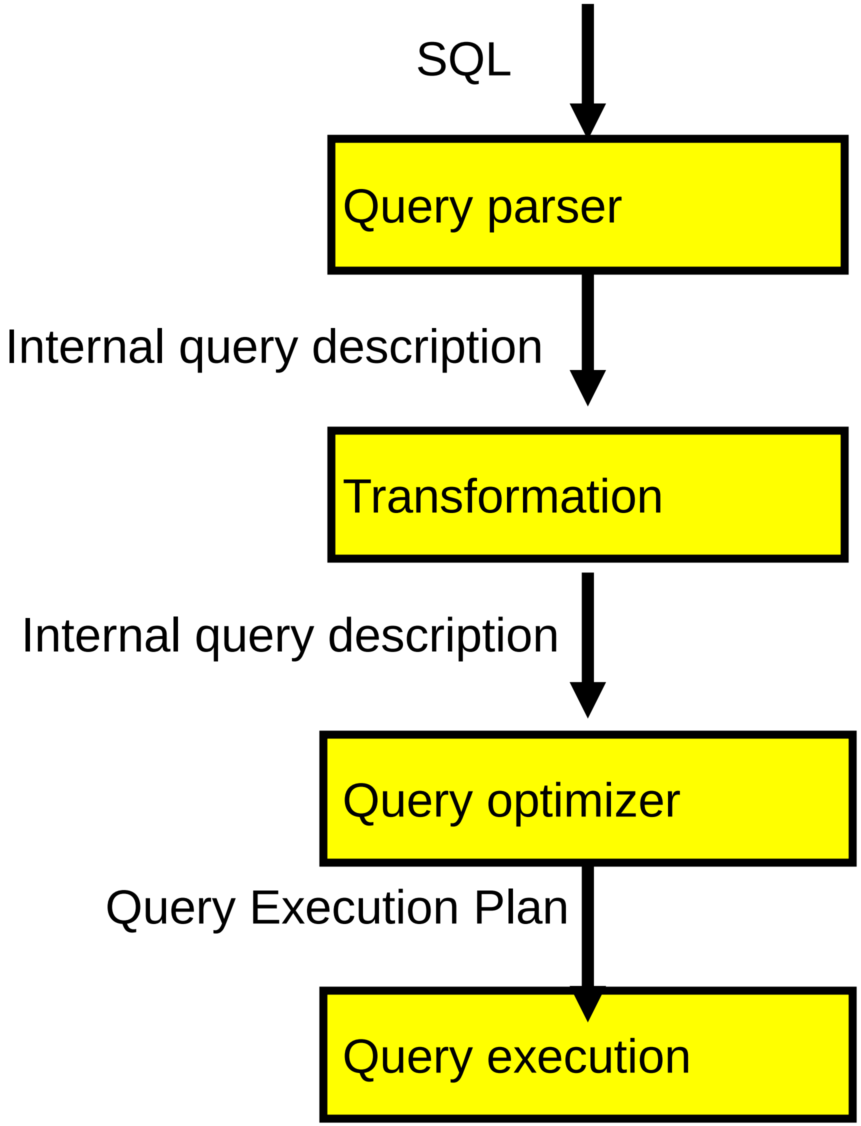
$\sigma_c(R \,\Phi\, S) \equiv (\sigma_c(R)) \,\Phi\, (\sigma_c(S)) \;(\cup, \cap, -)$

$\pi_L(R \,\Phi\, S) \equiv (\pi_L(R)) \,\Phi\, (\pi_L(S)) \;(\cup, \cap, -)$

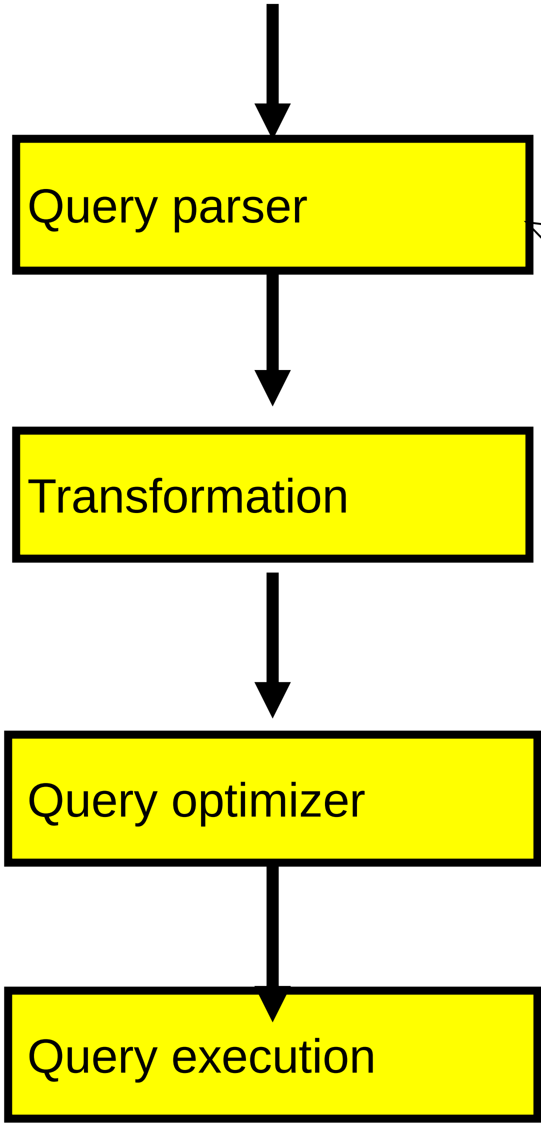
Query optimisation

- One SQL query – many (!) different execution plans, execution alternatives
  - Index – using, not using
    - tendence: selective query – using
    - Multiple indices, which index to use?
  - Join execution?
    - Size of tables, available memory…
  - Join order ?
- Dramatically different costs!
- Query optimizer: choosing a relatively good execution plan

Query execution

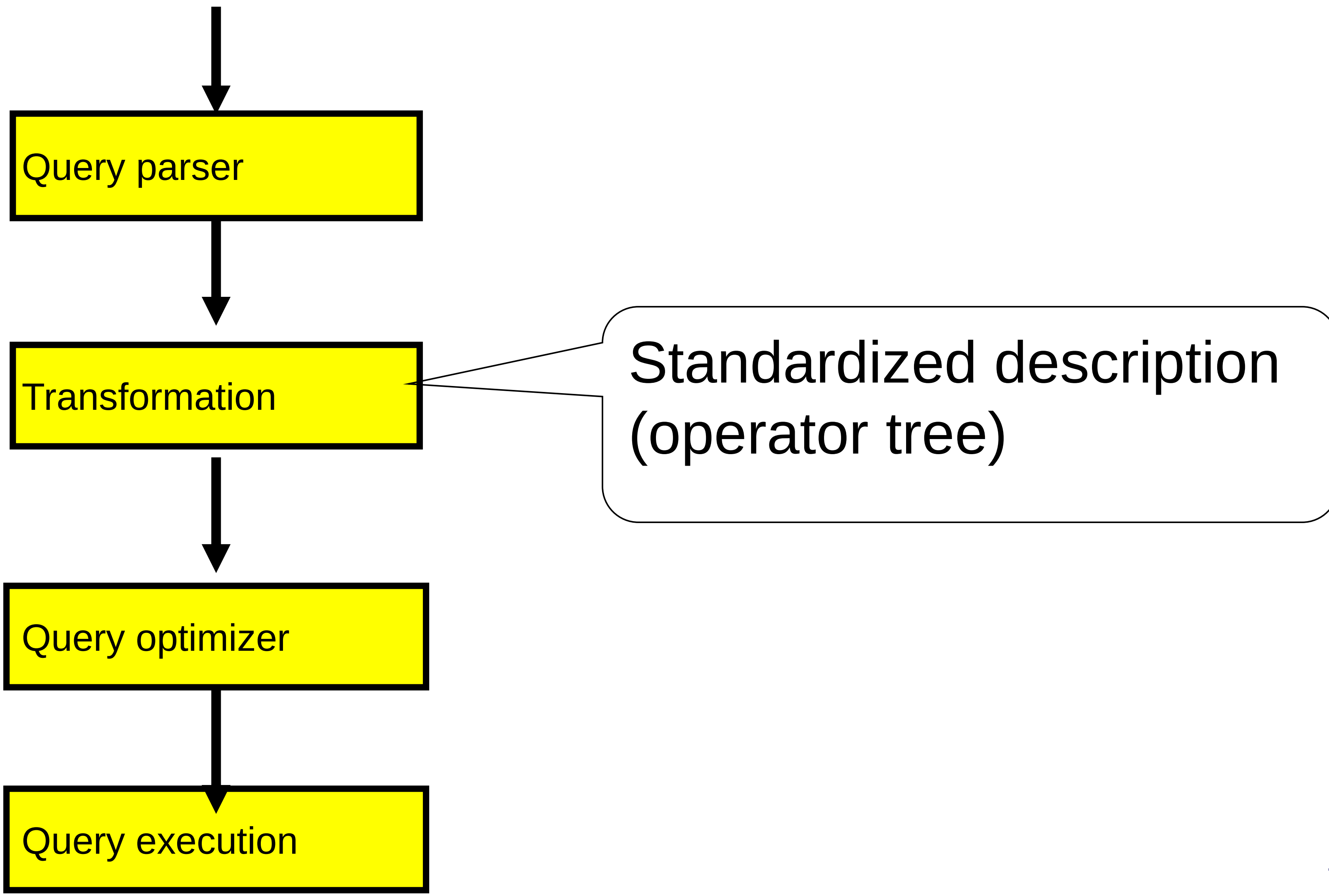


Query execution (detailed)

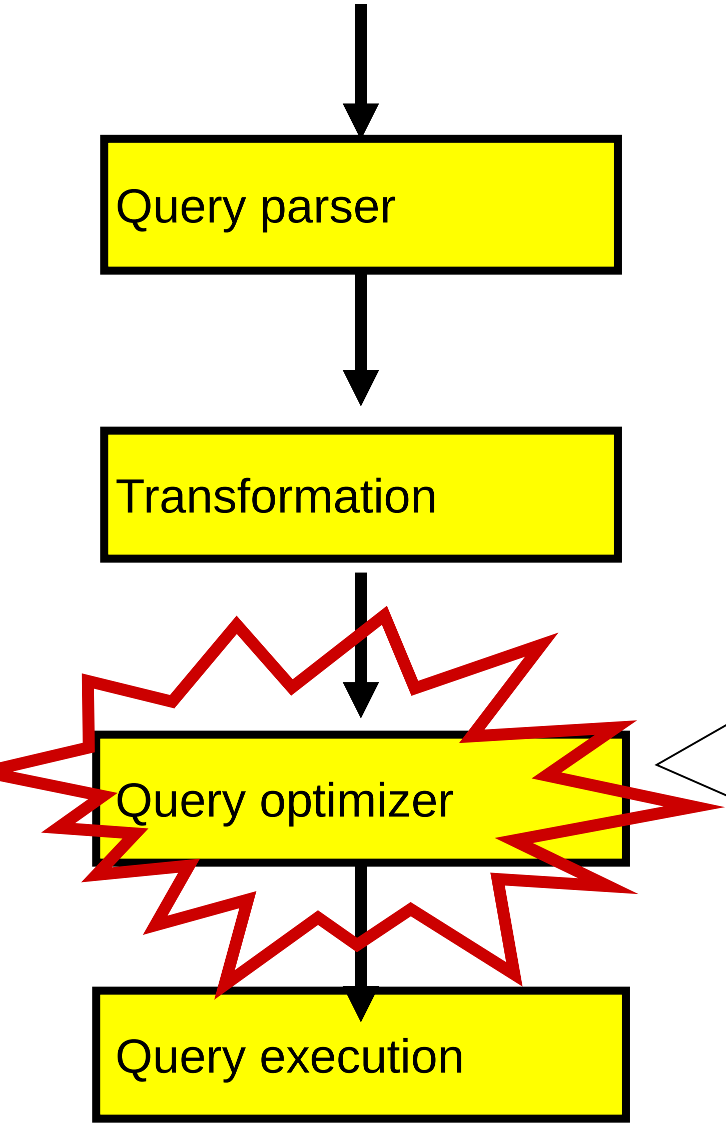


Parsing
Semantic check (tables, attributes – Data Dictionary)
View expansion

Query execution (detailed)



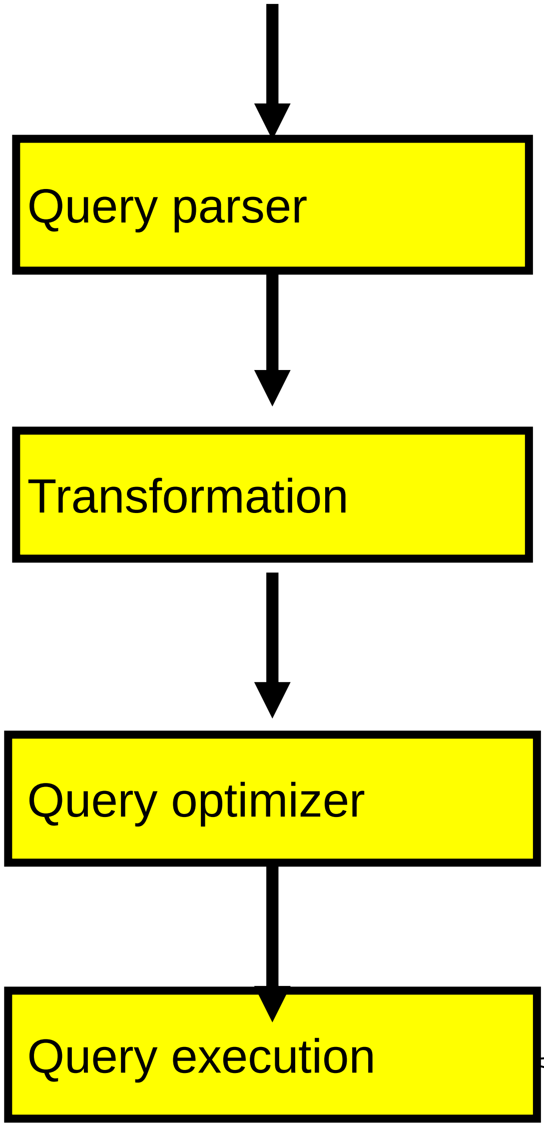
Query execution (detailed)



Optimisation (cost-based):

- Setting up execution plans
- Estimating their costs
- Selecting a cheap execution plan

Query execution (detailed)

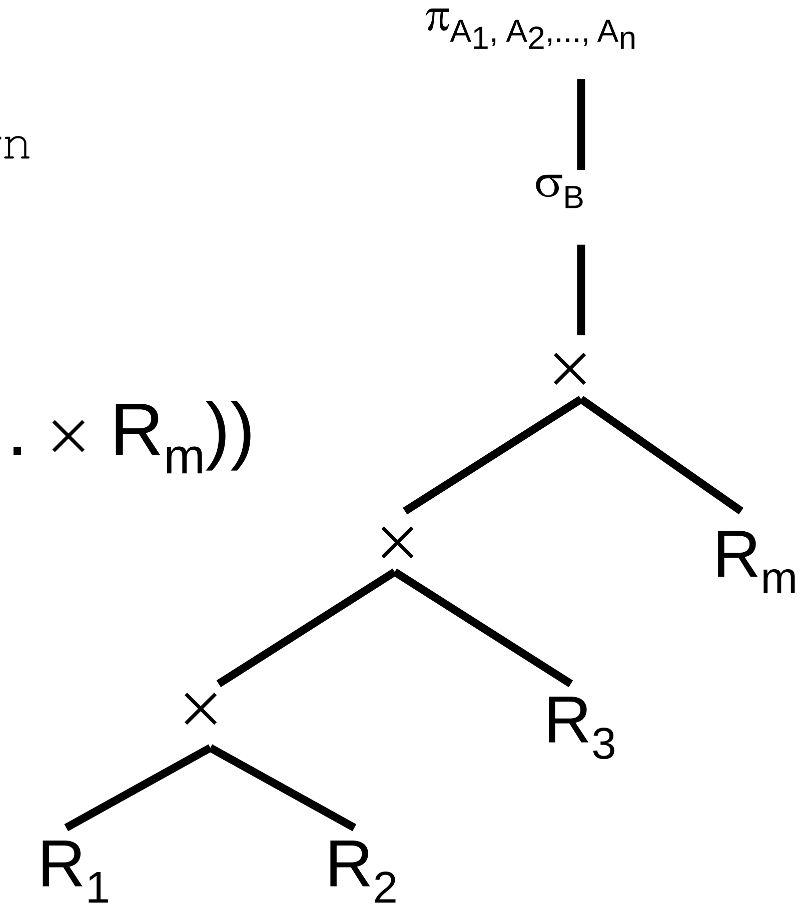


Execution of selected execution plan

Transformation, operator tree

```
select   A1, A2,..., An
from R1, R2,..., Rm
where    B
```

$\Rightarrow \pi_{A_1, A_2,\ldots, A_n} (\sigma_B (R_1 \times R_2 \times \ldots \times R_m))$

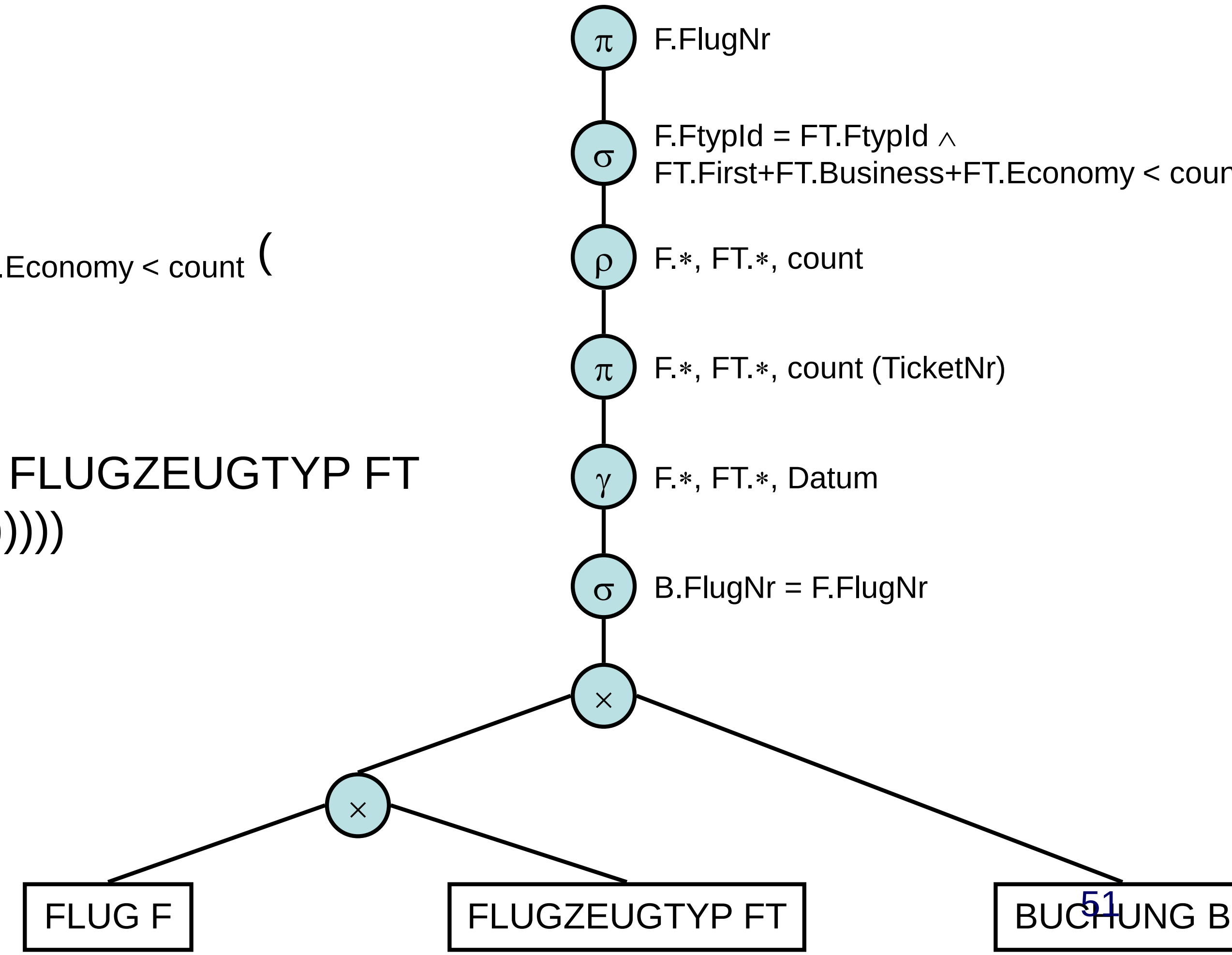


Example

```
select FlugNr
from   (select F.*, FT.*, count (TicketNr)
        from    FLUG F, FLUGZEUGTYP FT,
                BUCHUNG B
        where   B.FlugNr = F.FlugNr
        group by            F.*, FT.*, Datum)
       as DFT(F.*,FT.*,count)
where  F.FtypId = FT.FtypId
and    FT.First+FT.Business+FT.Economy
                        < DFT.count
```

Example

$\pi_{\text{F.FlugNr}}\ ($
$\sigma_{\text{F.FtypId = FT.FtypId} \wedge \text{FT.First+FT.Business+FT.Economy < count}}\ ($
$\rho_{\text{F.}*\text{,FT.}*\text{,count}}\ ($
$\pi_{\text{F.}*\text{, FT.}*\text{, count (TicketNr)}}\ ($
$\gamma_{\text{F.}*\text{, FT.}*\text{, Datum}}\ ($
$\sigma_{\text{B.FlugNr = F.FlugNr}}\ (\text{FLUG F} \times \text{FLUGZEUGTYP FT BUCHUNG B}))))))$



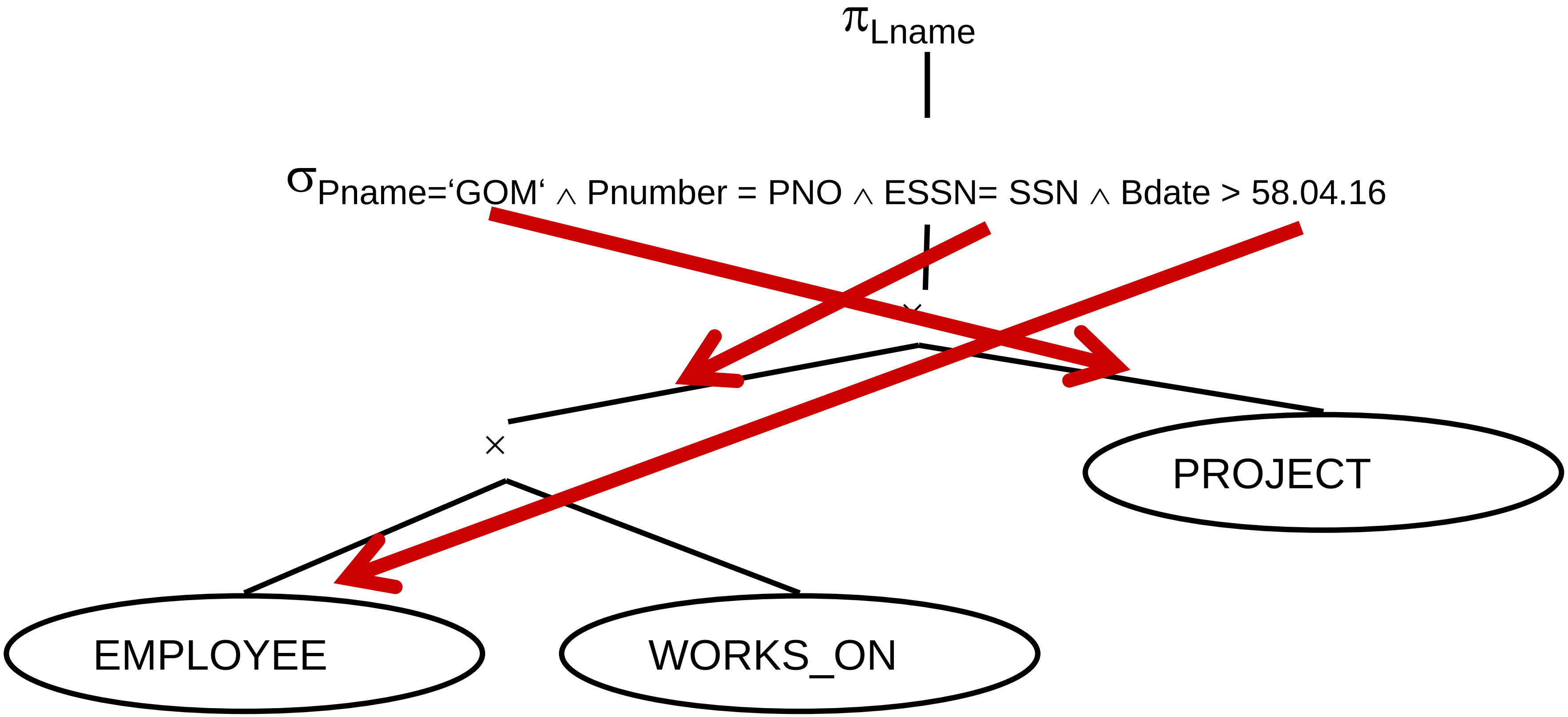
Basic idea

- Some transformations always reasonable
- Some transformations depend on data
  - Optimizing the average case
  - Keep intermediate results as small as possible
- Executing σ, π early, $\bowtie$, ⍰, $\cup$,… late, as
  - σ and π reduces the volume of data,
  - $\bowtie$, … result often in large intermediate results.

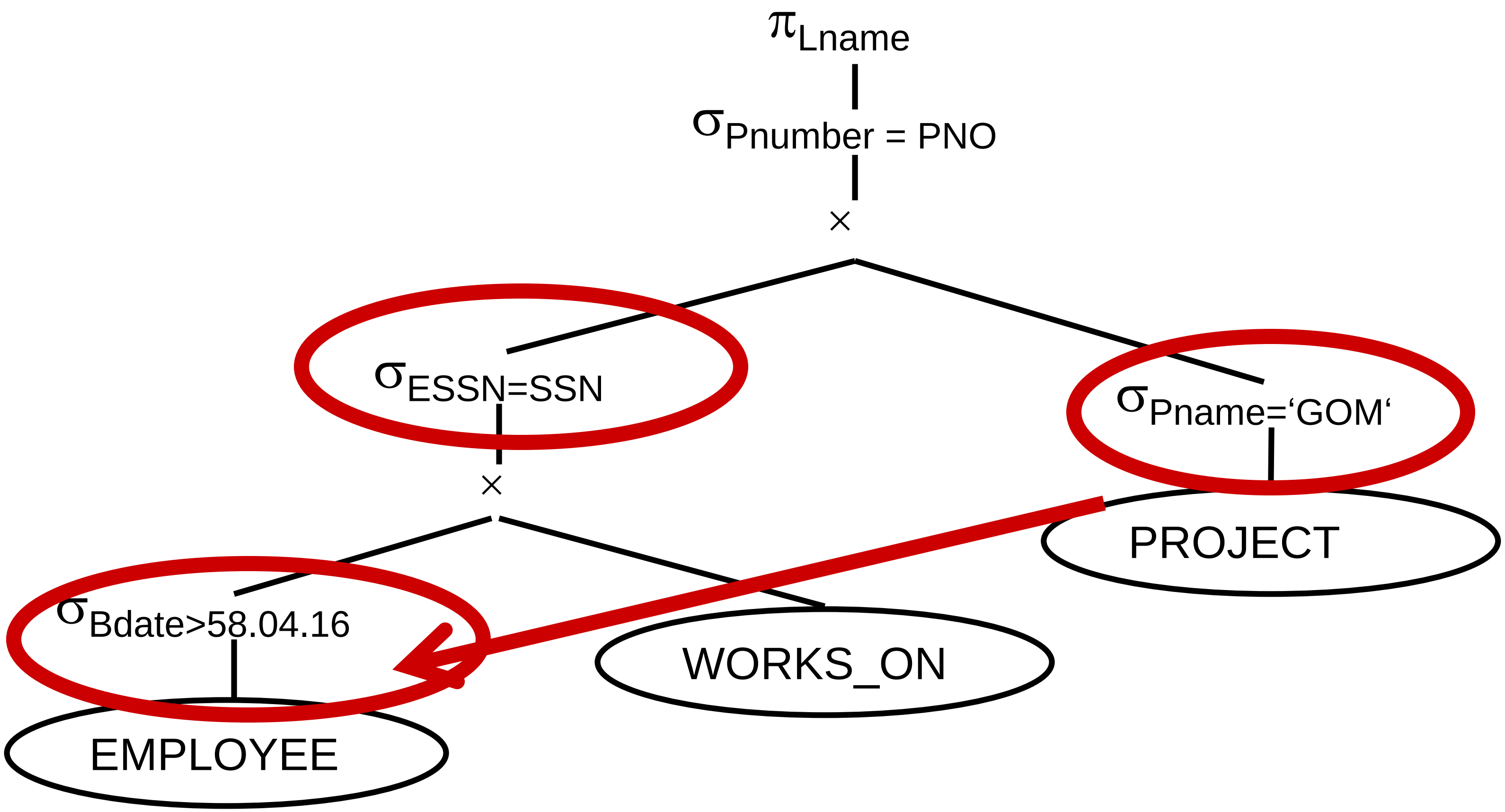
Example

- **select** Lname
  from Employee, WorksOn, Project
  where Pname = 'GOM'
  and Pnumber = PNO
  and ESSN = SSN
  and Bdate > 58.04.16

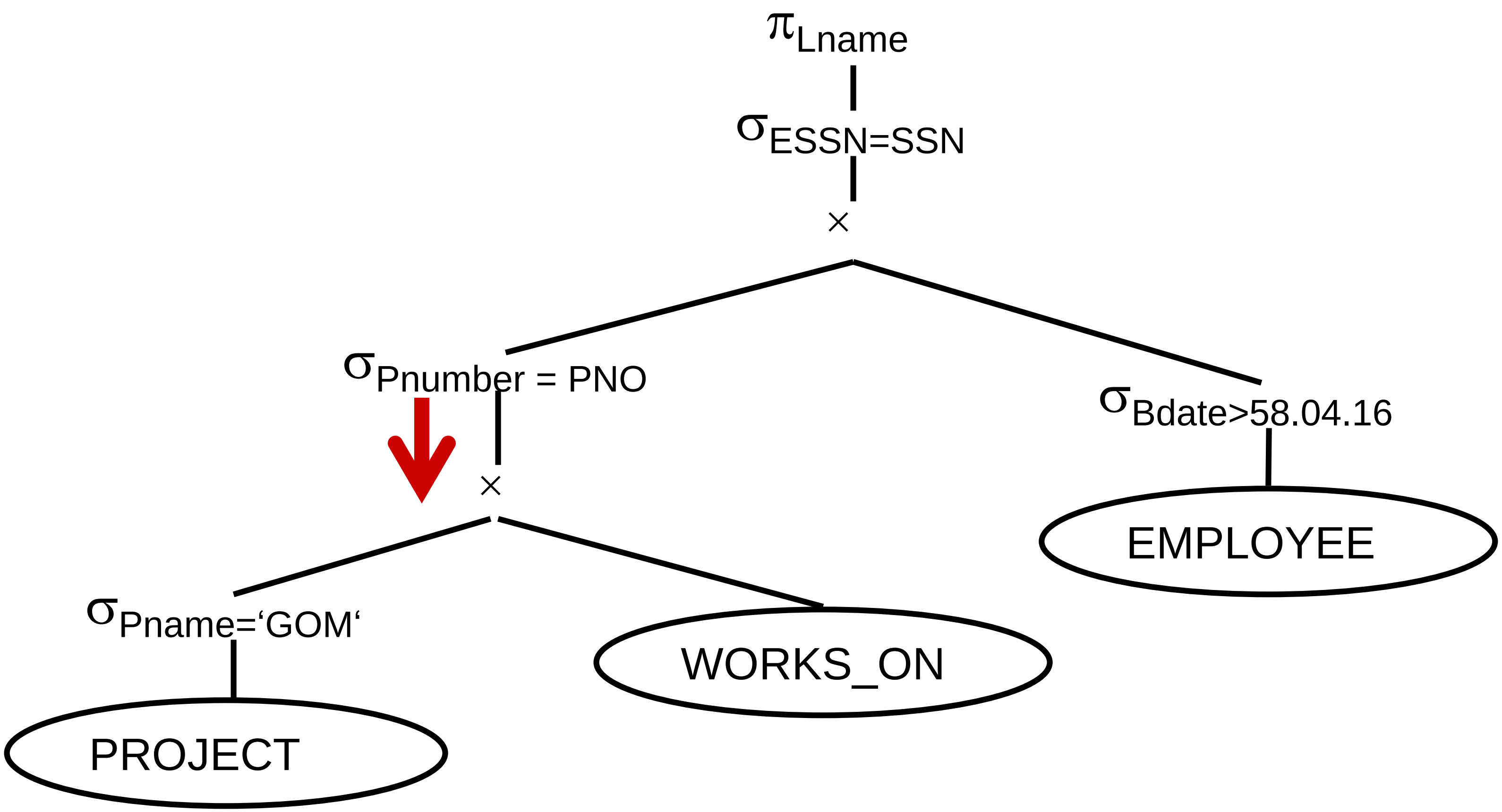
(Select the lastname of an employee born after 16.04.58 and working on the project „GOM”)



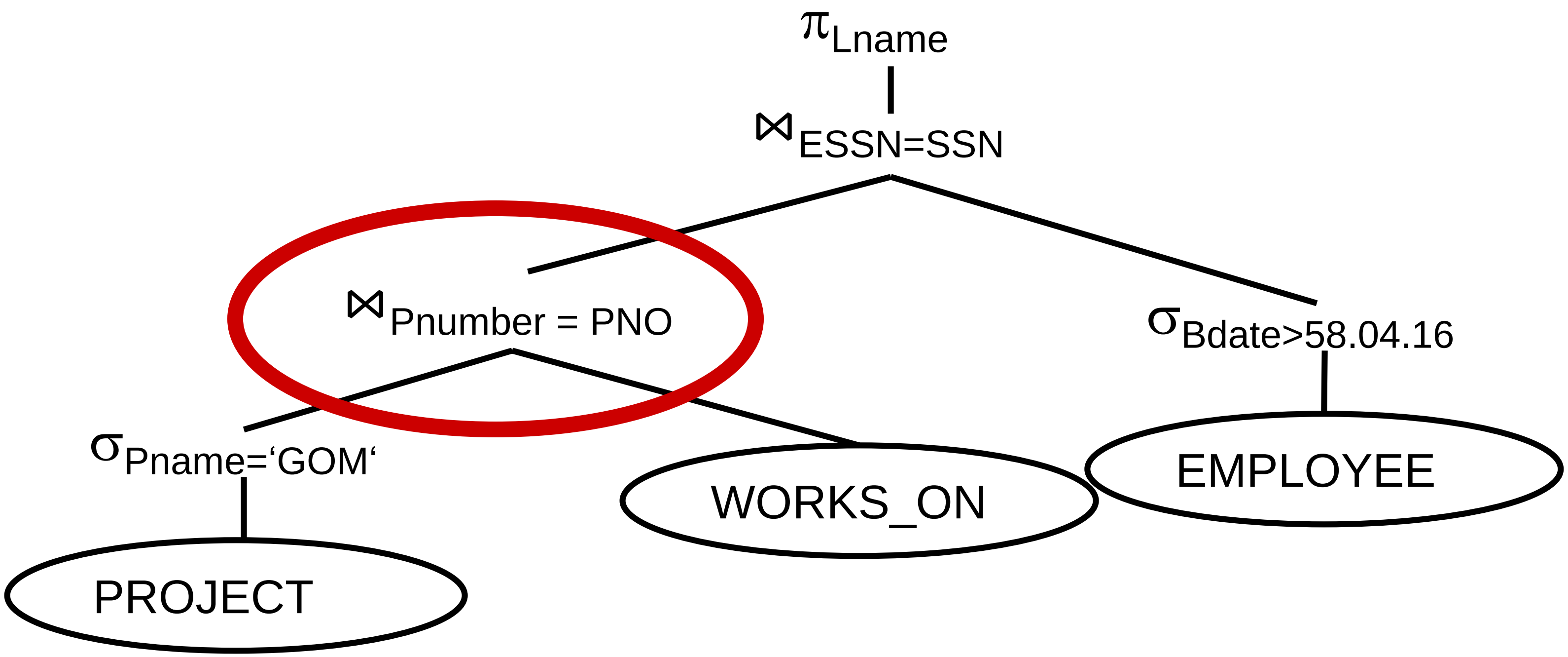
Selection as early, as possible



Restrictive joins early

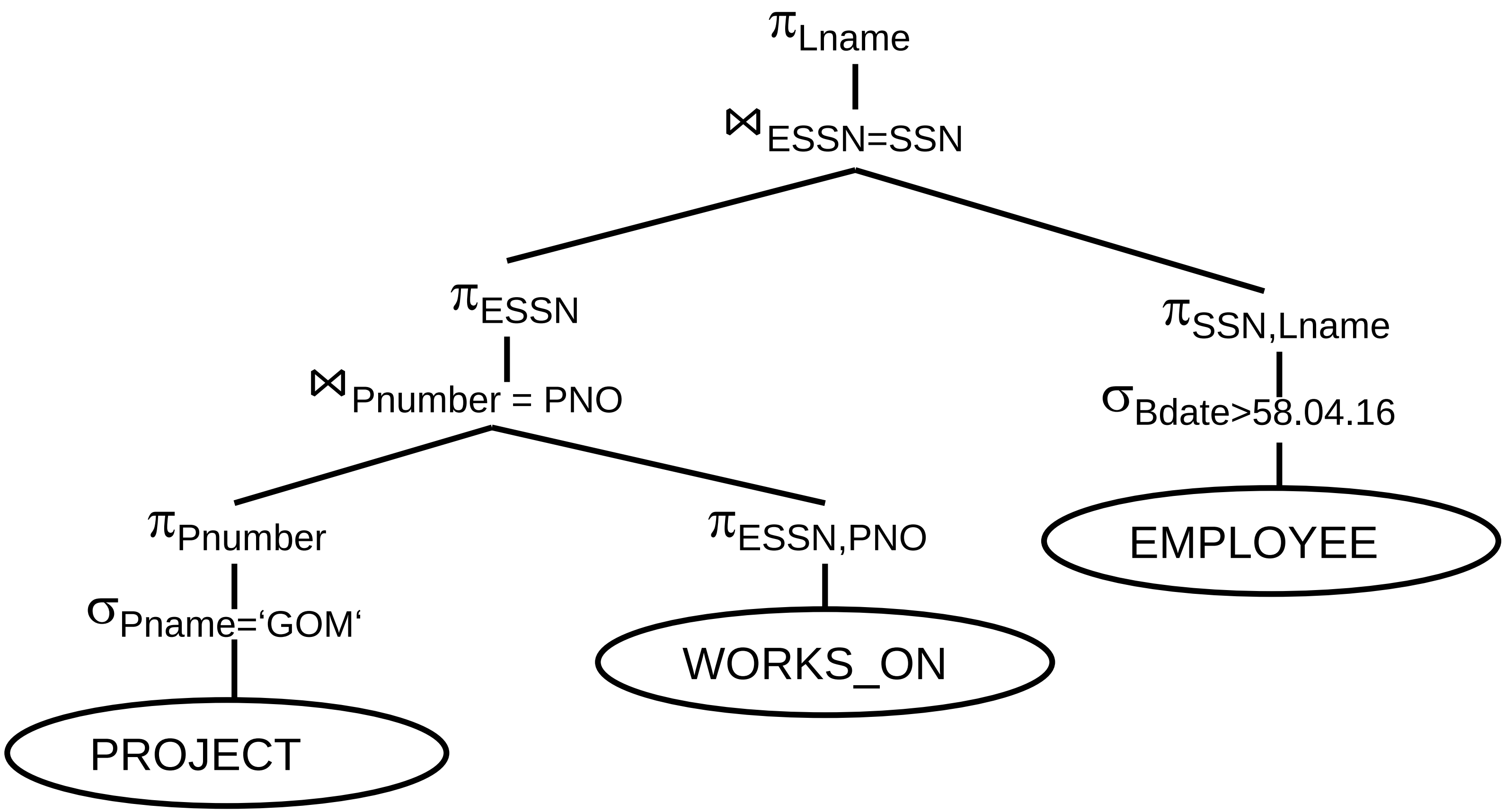


Cross product and selection => join



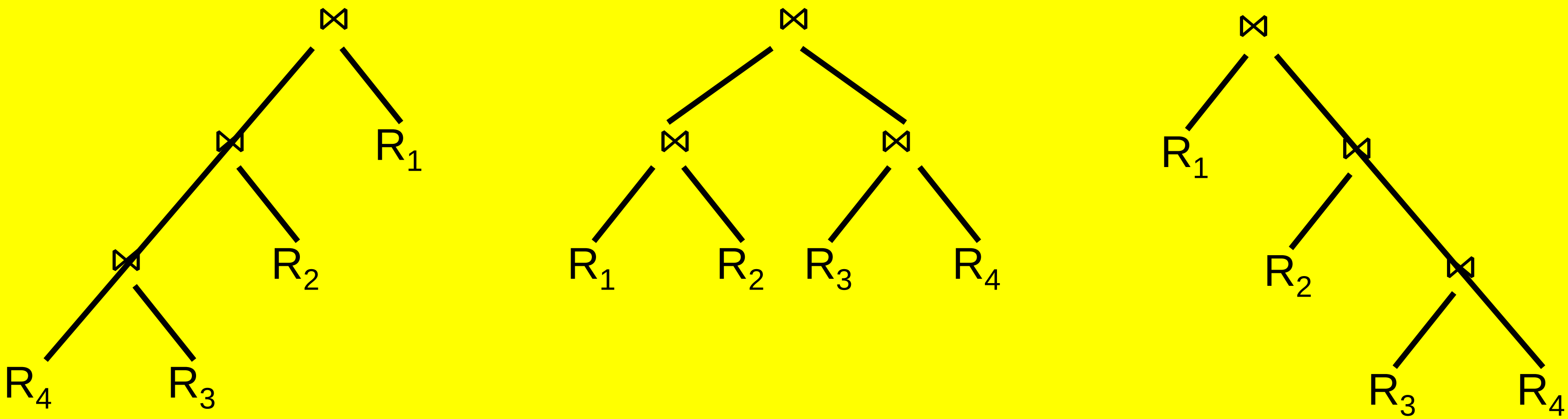
Projections as early, as possible

(attributes for result and intermediate results kept)


π Lname
⋈ ESSN=SSN
π ESSN
⋈ Pnumber = PNO
π SSN,Lname
σ Bdate>58.04.16
π Pnumber
π ESSN,PNO
EMPLOYEE
σ Pname=‘GOM‘
WORKS_ON
PROJECT

Join-order

- Many joins
- Joins expensive

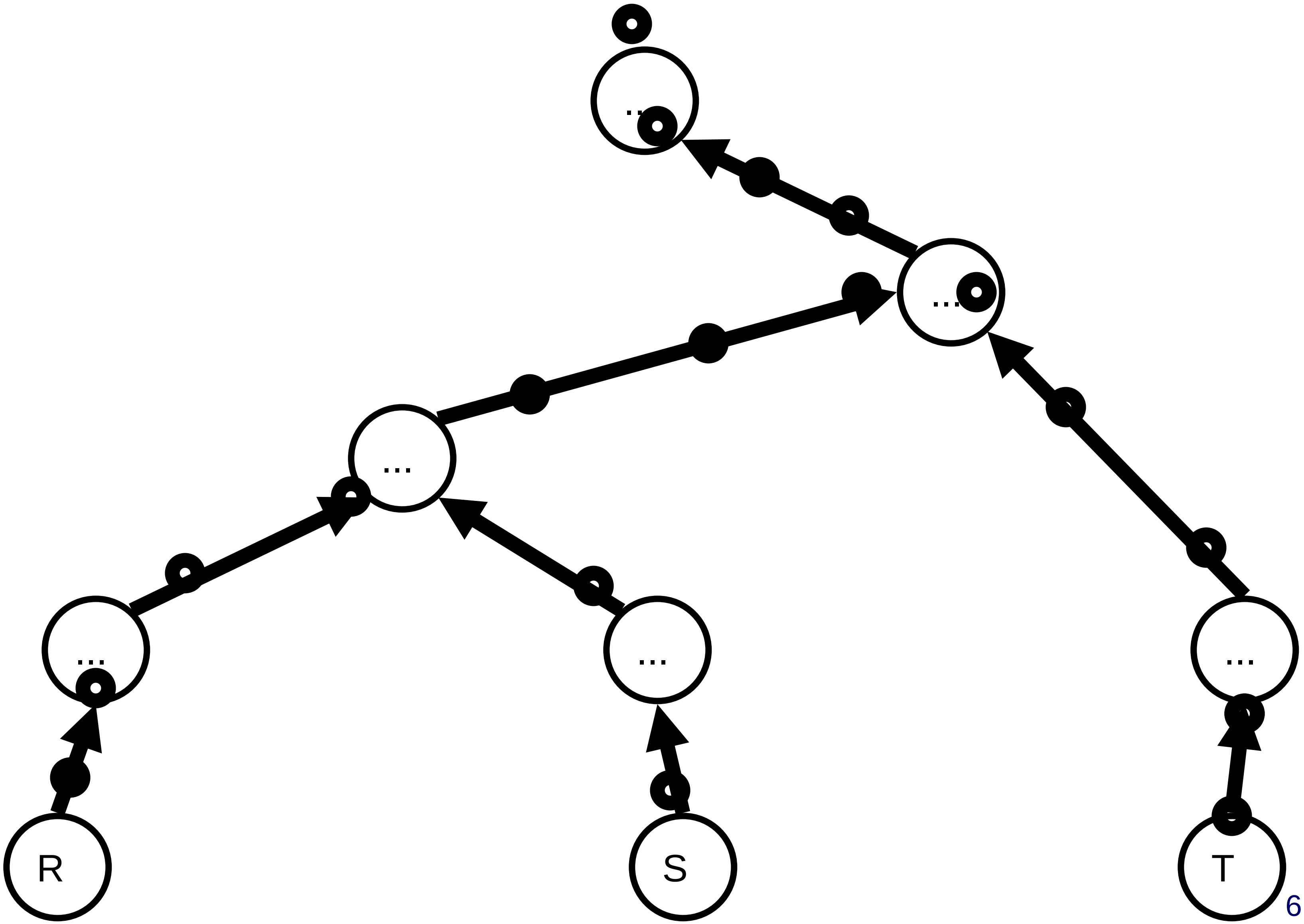


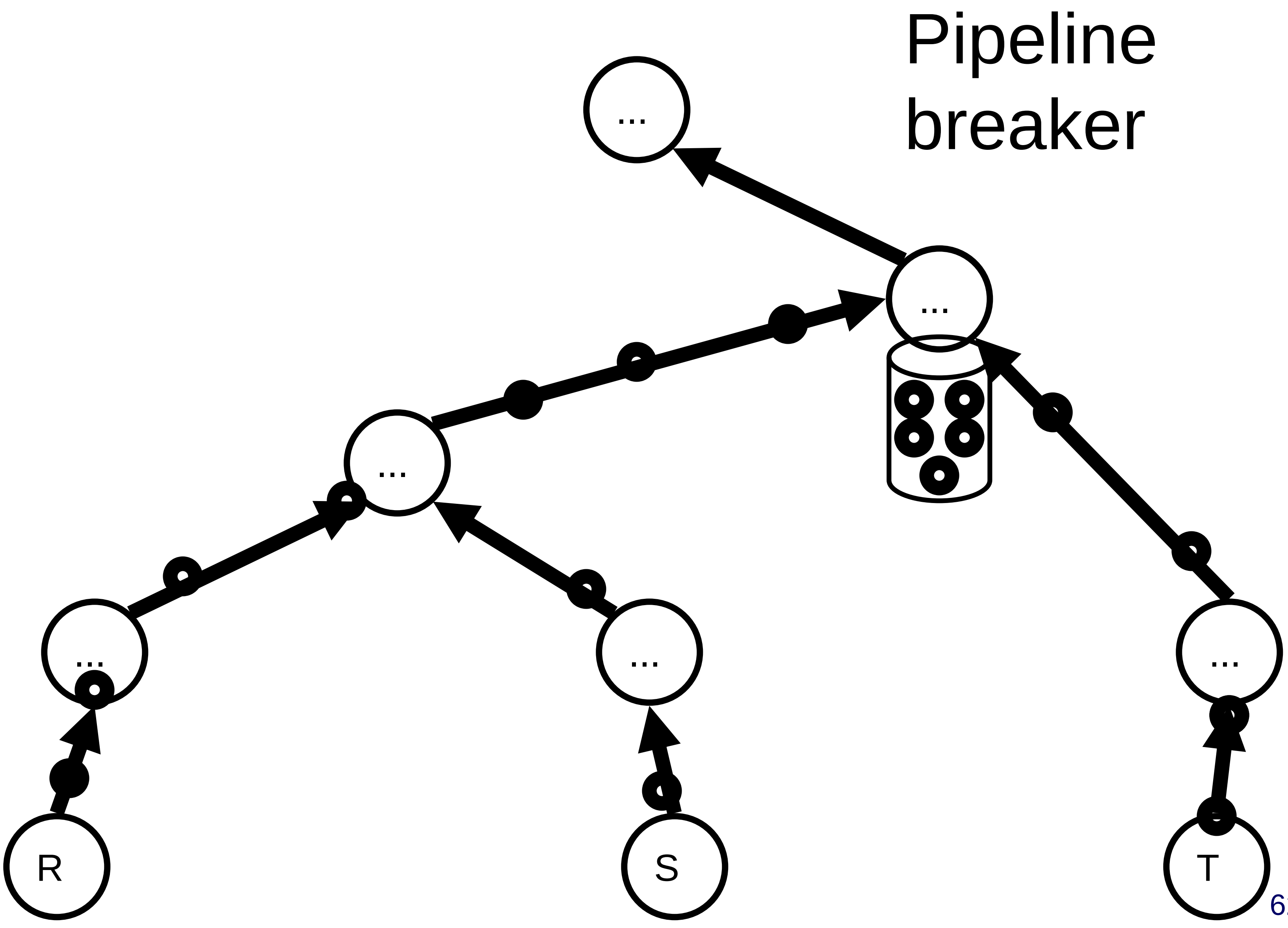
Left oriented join trees, greedy search...

Execution mode

- Full calculation
  - Node fully calculated before next operator
- Pipeline
  - Tuple calculated at one node sent immediately to next node

Execution modes (2)



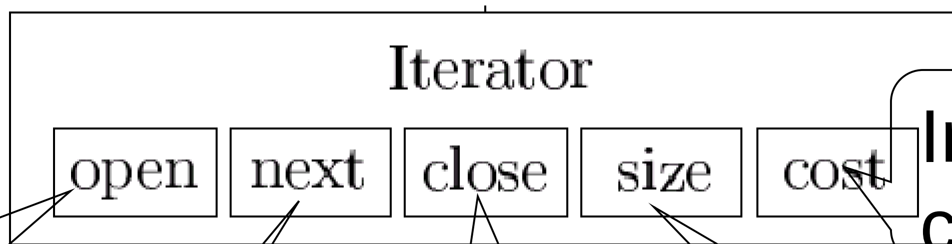


Pipeline
breaker
...
...
...
...
...
...
R
S
T

Pipeline-Breaker

- Unary operations
  - Sort
  - Duplicate elimination (unique distinct)
  - Aggregation (min, max, sum)
- Binary operations
  - Set difference
- Depending on the implementation
  - Join
  - Union

Unified description of operators



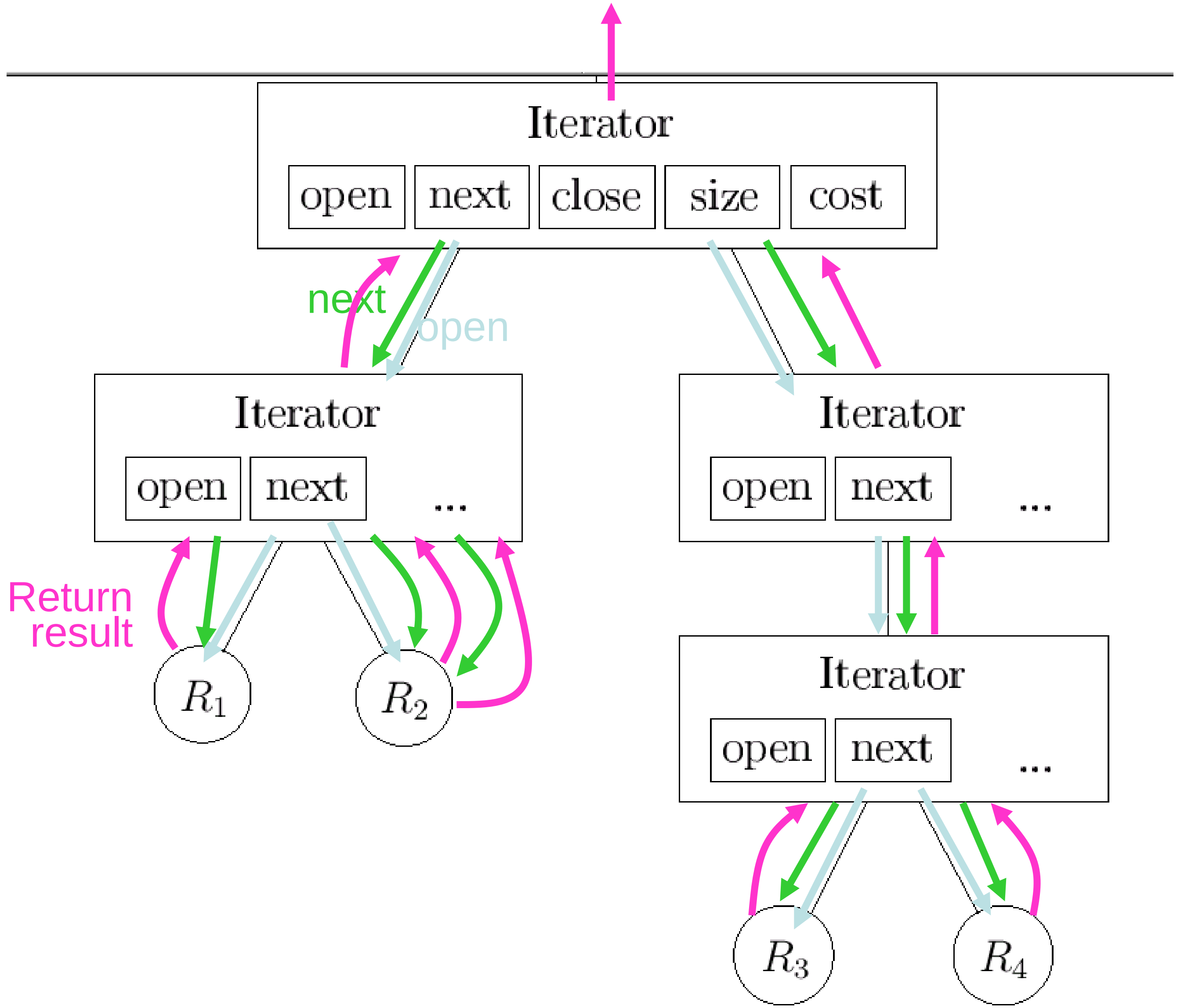
Information on costs

Initialisation

Information on selectivity

Reading next tuple

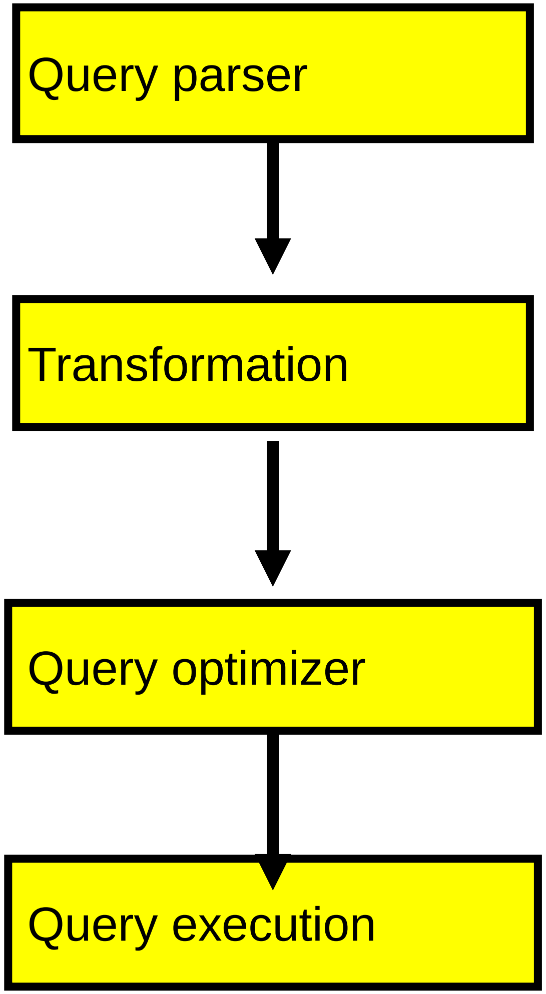
Clean up (closing iterators, freeing memory, stop blocking if full result required)


Iterator
open
next
close
size
cost
next
open
Iterator
open
next
...
Iterator
open
next
...
Iterator
open
next
...
Return result
R_1
R_2
R_3
R_4

TOP-N Query

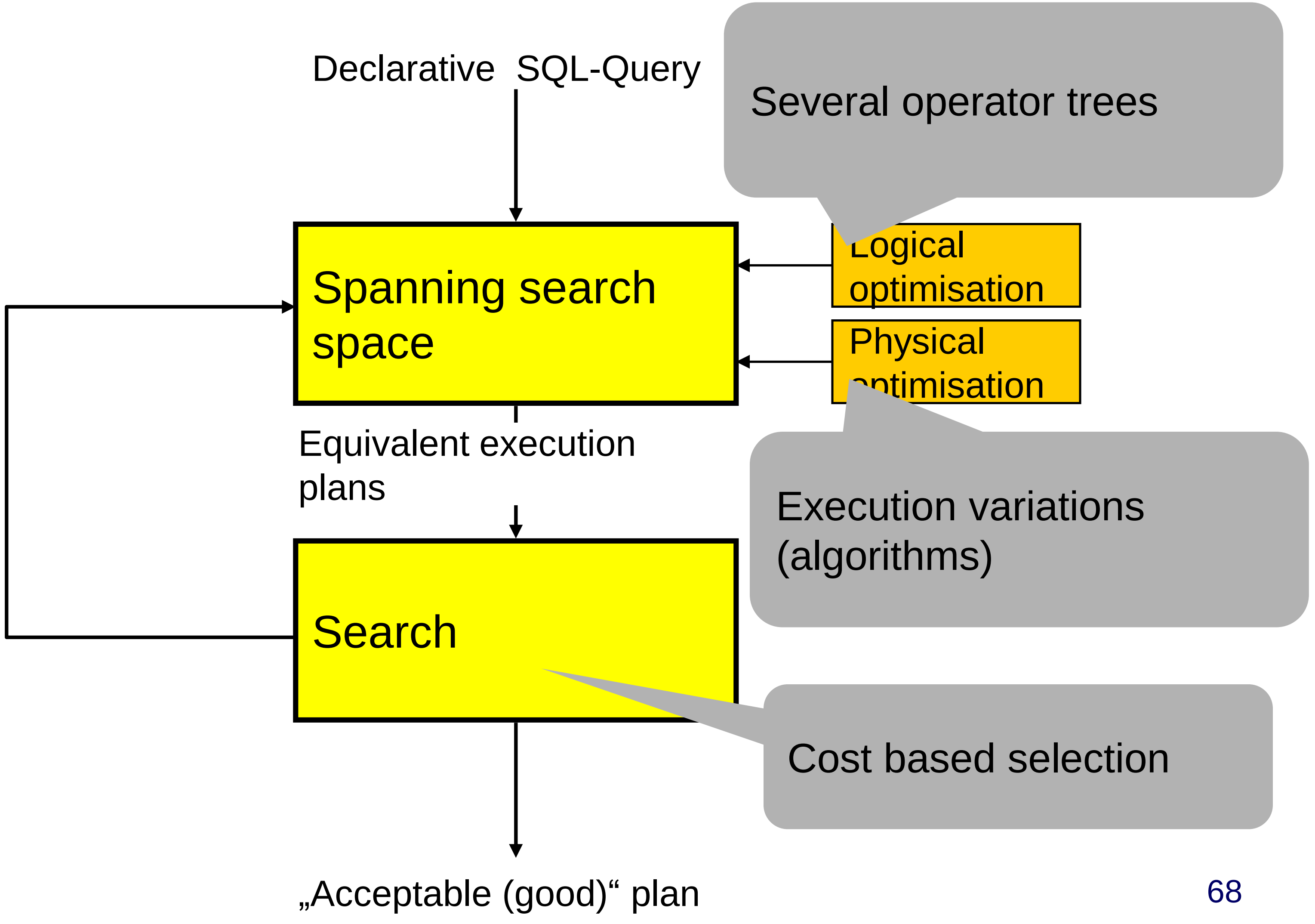
- Only the first N rows of the result are required
- Execution plan optimized specifically for this
  - Hard decisions
  - Soft decisions (internal restart of query possible with a low probability)

Cost based selection



Optimisation (cost-based):

- Setting up execution plans
- Estimating their costs
- Selecting a cheap execution plan


Declarative SQL-Query
Several operator trees
Spanning search space
Logical optimisation
Physical optimisation
Equivalent execution plans
Execution variations (algorithms)
Search
Cost based selection
„Acceptable (good)“ plan

Cost function

- Cost of execution
  - One major factor: disk access, number of blocks
- Size of intermediate result of fundamental importance
  - Estimation on selectivities required
- DBMS calculates and stores different statistics on the data

Oracle

70

Database SQL Tuning Guide (Oracle 12c, Database Administration)

Home / Database / Oracle Database Online Documentation 12c Release 1 (12.1) / Database Administration

Database SQL Tuning Guide

Page 1 of 40

Contents

Expand All · Collapse All

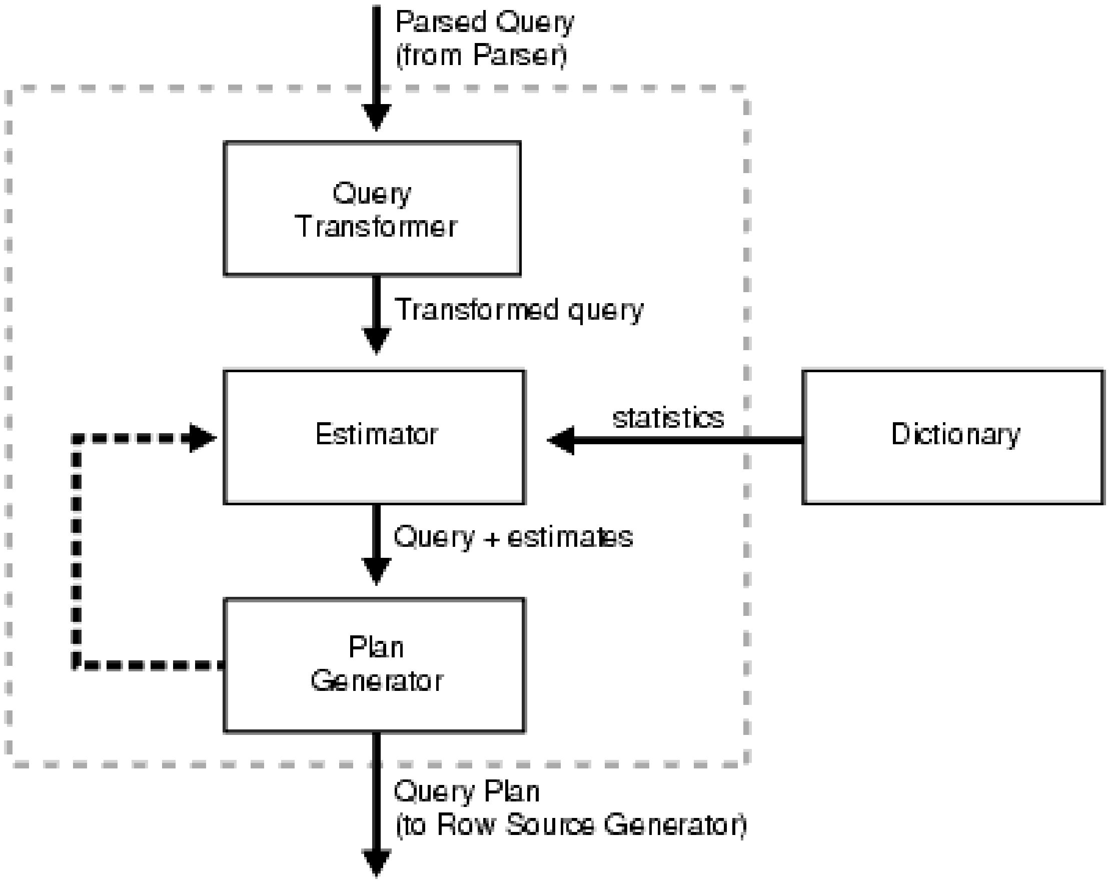
Optimizer Operations

Table 11-1 Optimizer Operations

Operation	**Description**
Evaluation of expressions and conditions	The optimizer first evaluates expressions and conditions containing constants as fully as possible.
Statement transformation	For complex statements involving, for example, correlated subqueries or views, the optimizer might transform the original statement into an equivalent join statement.
Choice of optimizer goals	The optimizer determines the goal of optimization. See "Choosing an Optimizer Goal".
Choice of access paths	For each table accessed by the statement, the optimizer chooses one or more of the available access paths to obtain table data. See "Overview of Optimizer Access Paths".
Choice of join orders	For a join statement that joins more than two tables, the optimizer chooses which pair of tables is joined first, and then which table is joined to the result, and so on. See "How the Query Optimizer Chooses Execution Plans for Joins".

- Best throughput
- Best response time

Components of the Query Optimizer



Costs

The cost is an estimated value proportional to the expected resource use needed to execute the statement with a particular plan. The optimizer calculates the cost of access paths and join orders based on the estimated computer resources, which includes **I/O, CPU, and memory.**

Statistics in databases

- **Oracle**
  - **TABLES**:
    - **num_rows**,
    - **num_blocks**
    - **avg_row_len**
  - **TAB_COL_STATISTICS**
    - **num_distinct**
    - **num_nulls**
    - **num_buckets**
- **INDEXES**
  - **leaf_blocks**
  - **blevel**

Calculation of statistics

Task of database administrator (expensive task!)

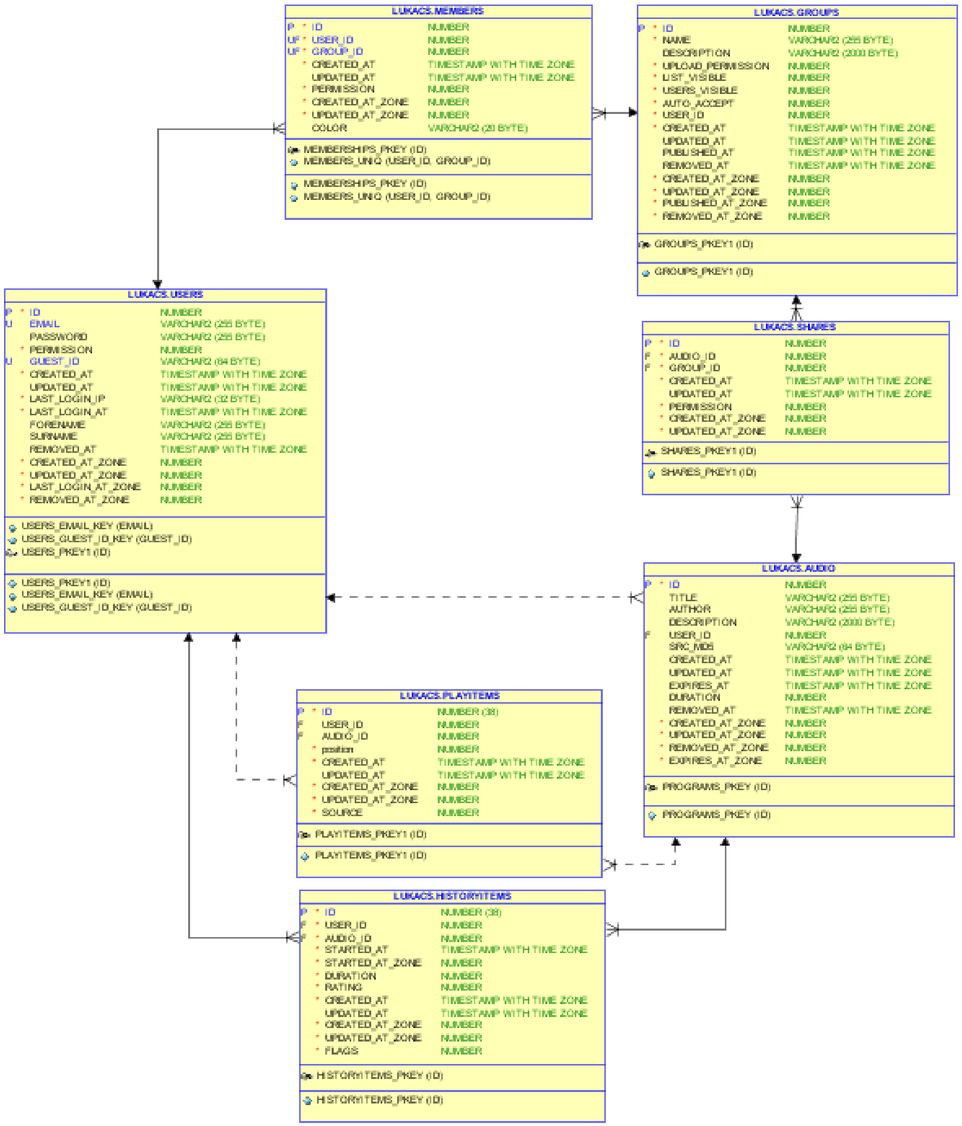
analyze table *relation*
compute statistics for columns *attribute*,..., *attribute*
size *value*

analyze table *relation*
estimate statistics sample *value* **percent**

Tasks

77

Grapefon - schema



Groups – statistics of shared audio recordings

0.1: Create a statistics showing the shared audio content (count only) per group

- Handle groups not having any shared audio content correctly
- Use column alias names
- Format the SQL command

NAME	SHARES_COUNT
PPKE ITK	24
Spirit - Demo	13
teknős6	0
teszt x	0
teknős5	0
[DEL] 3	0
teknős7	0
teszt	429
ía	0
6	0

Groups – statistics of shared audio recordings

0.2. Create a statistics showing the shared audio content (count + average length, rounded to seconds) per group

- Handle groups not having any shared audio content correctly
- Use column alias names
- Format the SQL command

NAME	SHARES_COUNT	SHARES_TOTALLENGTH	SHARES_AVGLENGTH
PPKE ITK	24	7933	331
Spirit - Demo	13	532	41
teknős6	0	(null)	(null)
teszt x	0	(null)	(null)
teknős5	0	(null)	(null)
[DEL] 3	0	(null)	(null)
teknős7	0	(null)	(null)
teszt	429	164076	382
ía	0	(null)	(null)
[illegible]	0	(null)	(null)

http://docs.oracle.com/cd/E11882_01/server.112/e26088/functions002.htm#SQLRF51178

Oracle® Database SQL Language Reference 11*g* Release 2 (11.2) Single-Row Functions

Groups, statistics of shared audio recordings and members

0.3. Extend the previous query with an additional column showing the number of the members in the group!

NAME	SHARES_COUNT	SHARES_TOTALLENGTH	MEMBERS_COUNT
teknős6	0	(null)	0
PPKE ITK Énekkar	1	130	5
Emmánuel	56	35442	18
ía	0	(null)	0
[DEL] teknős8	0	(null)	0
[DEL] 3	0	(null)	0
5	0	(null)	0
teknős4	0	(null)	0
Boardport	0	(null)	3

Check the execution plan of the query (number of members, shared audio).

Oracle Documentation: Oracle 12 g Database SQL Tuning Guide, https://docs.oracle.com/en/database/oracle/oracle-database/12.2/tgsql/reading-execution-plans.html#GUID-5AE1939F-F654-42FF-B0C5-706507CD12A2

Check the execution plan of the following for queries! Can you find any differences? What are the reasons for the differences?

```
SELECT COUNT(*)
FROM audio_large
INNER JOIN historyitems_large
ON audio_large.id = historyitems_large.audio_id;

SELECT COUNT(historyitems_large.rating)
FROM audio_large
INNER JOIN historyitems_large
ON audio_large.id = historyitems_large.audio_id;

SELECT AVG(historyitems_large.rating)
FROM audio_large
INNER JOIN historyitems_large
ON audio_large.id = historyitems_large.audio_id;

SELECT COUNT(historyitems_large.updated_at)
FROM audio_large
INNER JOIN historyitems_large
ON audio_large.id = historyitems_large.audio_id;
```

Check the execution plan of the following queries! (For the 2nd and 3rd queries, please only check the execution plans, without starting the queries.) What are the differences? Why?

```
SELECT COUNT(audio_large.user_id)
FROM audio_large
INNER JOIN historyitems_large
ON audio_large.id = historyitems_large.audio_id;

SELECT COUNT(audio_large.user_id)
FROM audio_large
INNER JOIN historyitems_large
ON audio_large.id BETWEEN historyitems_large.audio_id - 0.5 AND historyitems_large.audio_id + 0.5 ;


SELECT COUNT(audio_large.user_id)
FROM audio_large
INNER JOIN historyitems_large
ON audio_large.id - historyitems_large.audio_id = 0;
```

Check changes in the execution plan as the selectivity of the condition in the WHERE clause changes! (Change the condition on the user_id first!)

```
SELECT *
FROM   audio_large
       inner join historyitems_large
              ON audio_large.id = historyitems_large.audio_id
WHERE  To_char(historyitems_large.started_at, 'yyyymmdd') = '20160302'
       AND audio_large.user_id < 500;
```