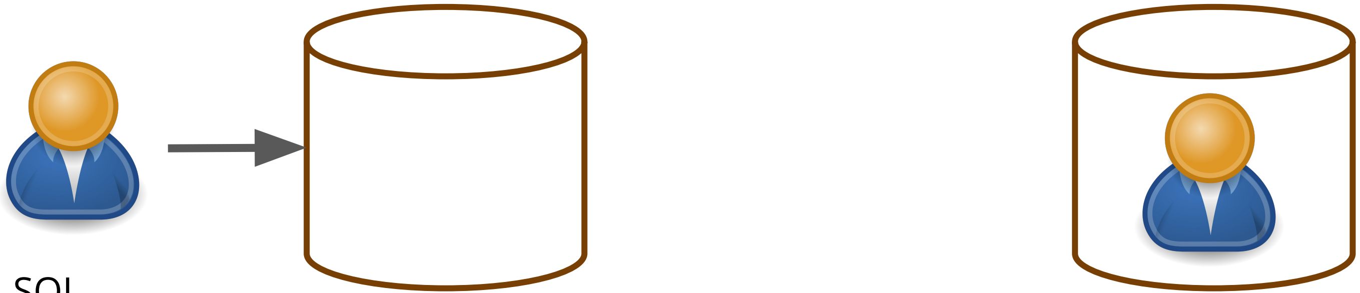
INFOH417 Database System Architectures

Mahmoud SAKR <mahmoud.sakr@ulb.be>

École polytechnique de Bruxelles

2023/24

What is this course for ?



SQL
Python
Web
…

Storage
Access control
Optimization
Distribution
…

Course Goals

- Understanding the query optimization and execution cycle
- Improving slow queries
- Describing the common index structures, knowing their capabilities and shortcomings
- Understanding cost based optimization, and the associated statistics and estimation methods
- Describing and being able to implement Abstract Data Types in extensible database systems
- Describing data and query distribution mechanisms, and being able to configure and run a distributed database system

Course Topics

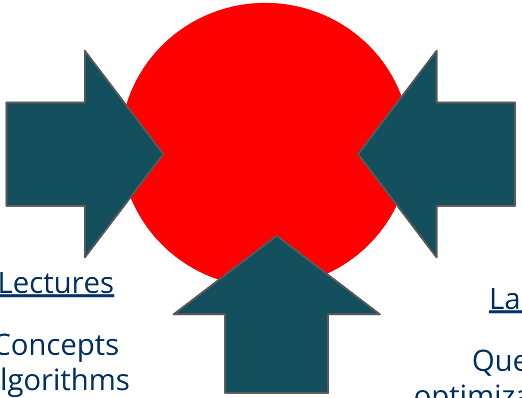
- Query execution
- Refreshing SQL and relational Algebra
- Cost-based query optimization
- Indexes
- Crash recovery
- Concurrency control
- Distributed databases

Prerequisites

- Relational databases
- SQL
- Relational Algebra
- General programming skills

Course Organization

- Lecture schedule and room on timedit: Check regularly for schedule updates
- Lectures by mahmoud.sakr@ulb.be
- Lab sessions by: Maxime Schoemans maxime.schoemans@ulb.be
  - Install PostgreSQL, any version, use same as the advanced DB course
  - Only 6 lab sessions, see UV for the schedule
  - Some of the remaining lab slots will be used for supporting you in the project as needed
- Grading
  - Group project, 4 members, 40% of total
  - Written exam, 60%
- Course notes, please enroll in Université virtuelle



Lectures

Concepts
Algorithms
Complexity
Research papers

Project

Practice research & system development

Lab

Query optimization & System development
Hands-on

Recommended Readings

- A mixture of book chapters and research papers, which will be identified per lecture

Query Planning:
Translating SQL into Relational Algebra

Refreshing the Relational Algebra

- Relations are tables whose columns have names, called attributes
- The set of all attributes of a relation is called the schema of the relation
- The rows in a relation are called tuples
- A relation is set-based if it does not contain duplicate tuples.
- It is called bag-based otherwise.
- A Relational Algebra (RA) operator takes as input 1 or more relations

  and produces as output a new relation

A	B	C	D
1	2	3	4
1	2	3	5
3	4	5	6
5	6	3	4

Translating SQL into Relational Algebra

In the examples that follow, we will use the following database:

- `Movie(title: string, year: int, length: int, genre: string, studioName: string, producerC#: int)`
- `MovieStar(name: string, address: string, gender: char, birthdate: date)`
- `StarsIn(movieTitle: string, movieYear: string, starName: string)`
- `MovieExec(name: string, address: string, CERT#: int, netWorth: int)`
- `Studio(name: string, address: string, presC#: int)`

select-from-where

```
SQL:
SELECT movieTitle
FROM   StarsIn S, MovieStar M
WHERE  S.starName = M.name AND M.birthdate = 1960
```

RA ?

select-from-where

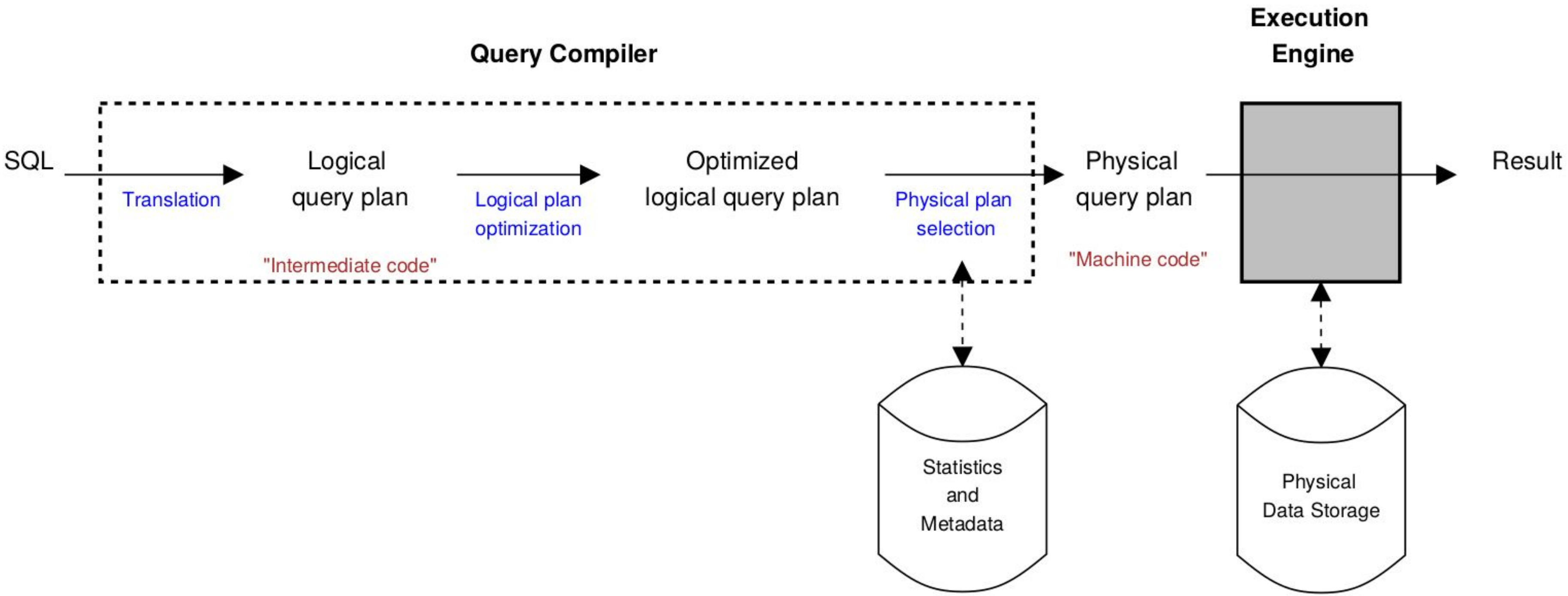
```
SQL:
SELECT movieTitle
FROM   StarsIn S, MovieStar M
WHERE  S.starName = M.name AND M.birthdate = 1960
```

RA ?

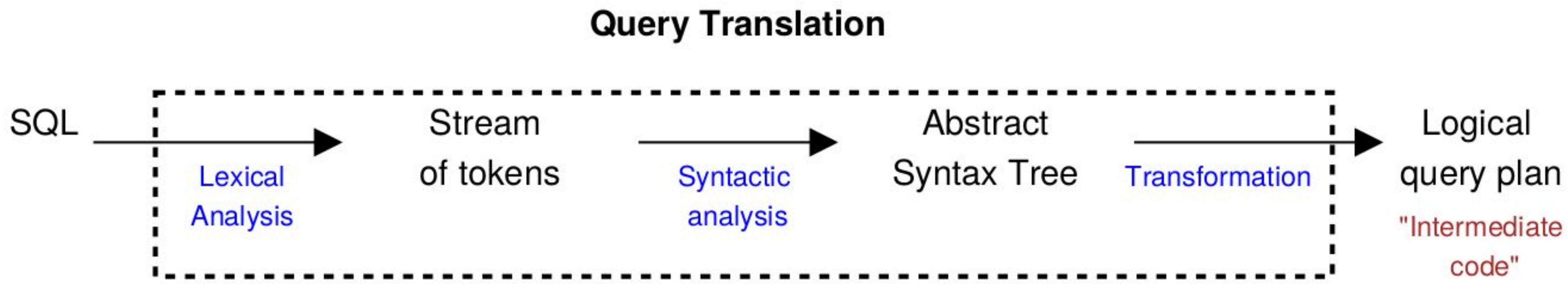
$\pi_{\text{movieTitle}}\ \sigma_{\text{S.starName=M.name and M.birthdate=1960}}(\rho_S(\text{StarsIn}) \times \rho_M(\text{MovieStar}))$

Other translations ?

Query Planning



Query Translation



The Extended Relational Algebra

Operator		Operator	
$\cup$	Union	$\bowtie$	Natural join
$\cap$	Intersection	$\bowtie_{B=C}$	Theta join
-	Difference	$⟕_{B=C}$	Left outer join
$\sigma_{A>=3}$	Selection	$⟖_{B=C}$	Right outer join
$\pi_{A,C}$	Projection	$⟗_{B=C}$	Full outer join
$\times$	Cartesian product	$\gamma_{A,min(B)->D}$	Aggregation
ρ	Rename	$\leftarrow$	Assignment

Union ∪ / Intersection ∩ / Difference -

A	B
1	2
3	4
5	6

∪

A	B
3	4
1	5

=

Set-based

A	B
1	2
3	4
5	6
1	5

Bag-based

A	B
1	2
3	4
5	6
1	5
3	4

- Input relations must have the same schema (same set of attributes)
- Historically speaking, relations are defined to be sets of tuples: duplicate tuples cannot occur in a relation.
- In practical systems, however, it is more efficient to allow duplicates to occur in relations, and only remove duplicates when requested. In this case relations are bags.

ULB

Selection

$\sigma_{A>=3}$

A	B
1	2
3	4
5	6

=

A	B
3	4
5	6

Projection

$$\pi_{A,C}$$

A	B	C	D
1	2	3	5
3	4	3	6
5	6	5	9
1	6	3	5

=

Set-based

A	C
1	3
3	3
5	5

Cartesian Product

A	B
1	2
3	4

×

C	D
2	6
3	7
4	9

=

A	B	C	D
1	2	2	6
1	2	3	7
1	2	4	9
3	4	2	6
3	4	3	7
3	4	4	9

Input relations must have disjoint schema (disjoint set of attributes), otherwise rename first

Natural Join

A	B
1	2
3	4

$\bowtie$

B	D
2	6
3	7
4	9

$=$

A	B	D
1	2	6
3	4	9

Natural Join

A	B
1	2
3	4

$\bowtie$

C	D
2	6
3	7
4	9

=

A	B	C	D
1	2	2	6
1	2	3	7
1	2	4	9
3	4	2	6
3	4	3	7
3	4	4	9

Same as cartesian product

Theta Join

A	B
1	2
3	4

$\bowtie_{B=C}$

C	D
2	6
3	7
4	9

=

A	B	C	D
1	2	2	6
3	4	4	9

Renaming

ρ_T

A	B
1	2
3	4
5	6

=

T.A	T.B
1	2
3	4
5	6

Renaming specifies that the input relation (and its attributes) should be given a new name.

Relational Algebra Expressions

Built using relation variable, AND

RA operators

$$\sigma_{\texttt{length>=100}}(\texttt{Movie}) \bowtie_{\texttt{title=movietitle}} \texttt{StarsIn}$$

Write the equivalent SQL

The Extended Relational Algebra

Add more operators

Extended projection

allows renaming

$\pi_{A, C \to D}$

A	B	C	D
1	2	3	5
3	4	3	6
5	6	5	9
1	6	3	5

=

Set-based

A	D
1	3
3	3
5	5

The Extended Relational Algebra

Add more operators

Grouping

$$\gamma_{\texttt{A,min(B)->D}}$$

A	B	C
1	2	a
1	3	b
2	3	c
2	4	a
2	5	a

=

A	D
1	2
2	3

ULB

select-from-where-groupby

```
SQL:
SELECT movieTitle, count(S.startName) AS numStars
FROM   StarsIn S, MovieStar M
WHERE  S.starName = M.name
GROUP BY movieTitle
```

RA ?

select-from-where-groupby

```
SQL:
SELECT movieTitle, count(S.startName) AS numStars
FROM   StarsIn S, MovieStar M
WHERE  S.starName = M.name
GROUP BY movieTitle
```

RA ?

$$\gamma_{\texttt{M.movieTitle,count(S.starName)->numStars}}(\rho_S(\texttt{StarsIn}) \bowtie_{\texttt{S.starName=M.name}} \rho_M(\texttt{MovieStar}))$$

select-from-where-groupby-having

```
SELECT movieTitle, count(S.startName) AS numStars
FROM   StarsIn S, MovieStar M
WHERE  S.starName = M.name
GROUP BY movieTitle
HAVING count(S.startName) > 5
```

RA ?

select-from-where-groupby-having

```
SELECT movieTitle, count(S.startName) AS numStars
FROM   StarsIn S, MovieStar M
WHERE  S.starName = M.name
GROUP BY movieTitle
HAVING count(S.startName) > 5
```

RA ?

$$\sigma_{numStarts>5}(\gamma_{M.movieTitle,count(S.starName)\text{->}numStars}(\rho_S(StarsIn) \bowtie_{S.starName=M.name} \rho_M(MovieStar)))$$

Subqueries

```
SELECT *
FROM huge
WHERE c1 IN
   (SELECT c1 FROM tiny)
```

V.S.

```
SELECT *
FROM huge h, tiny t
WHERE h.c1=t.c1
```

Which query is better ?

PostgreSQL Source Code git master

Main Page | Namespaces ▾ | Data Structures ▾ | Files ▾

- ► foreign
- ► jit
- ► lib
- ► libpq
- ► main
- ► nodes
- ▼ optimizer
  - ► geqo
  - ► path
  - ▼ plan
    - ► analyzejoins.c
    - ► createplan.c
    - ► initsplan.c
    - ► planagg.c
    - ► planmain.c
    - ► planner.c
    - ► setrefs.c
    - ► subselect.c
  - ► prep
  - ► util
- ► parser

```c
/*
 * If there is a WITH list, process each WITH query and either convert it
 * to RTE_SUBQUERY RTE(s) or build an initplan SubPlan structure for it.
 */
if (parse->cteList)
    SS_process_ctes(root);

/*
 * If the FROM clause is empty, replace it with a dummy RTE_RESULT RTE, so
 * that we don't need so many special cases to deal with that situation.
 */
replace_empty_jointree(parse);

/*
 * Look for ANY and EXISTS SubLinks in WHERE and JOIN/ON clauses, and try
 * to transform them into joins.  Note that this step does not descend
 * into subqueries; if we pull up any subqueries below, their SubLinks are
 * processed just before pulling them up.
 */
if (parse->hasSubLinks)
    pull_up_sublinks(root);

/*
 * Scan the rangetable for function RTEs, do const-simplification on them,
 * and then inline them if possible (producing subqueries that might get
 * pulled up next).  Recursion issues here are handled in the same way as
 * for SubLinks.
 */
preprocess_function_rtes(root);

/*
 * Check to see if any subqueries in the jointree can be merged into this
 * query.
 */
pull_up_subqueries(root);
```

Subquery processing and transformations

Subqueries are notoriously expensive to evaluate. This section describes some of the transformations that Derby makes internally to reduce the cost of evaluating them.

Materialization
Materialization means that a subquery is evaluated only once. There are several types of subqueries that can be materialized.

Flattening a subquery into a normal join

Flattening a subquery into an EXISTS join

Flattening VALUES subqueries

DISTINCT elimination in IN, ANY, and EXISTS subqueries

IN/ANY subquery transformation

Parent topic: Internal language transformations

Related concepts
Predicate transformations
Transitive closure

Subqueries

We can always normalize subqueries to use only EXISTS and NOT EXISTS [Van den Bussche, Vansummeren] [1,2]

```
SELECT movieTitle FROM StarsIn
WHERE starName IN (SELECT name
                   FROM MovieStar
                   WHERE birthdate=1960)
```

```
⇒   SELECT movieTitle FROM StarsIn
    WHERE EXISTS (SELECT name
                   FROM MovieStar
                   WHERE birthdate=1960 AND name=starName)
```

1 Only valid for set-based Relations
2 https://cs.ulb.ac.be/public/_media/teaching/infoh417/sql2alg_eng.pdf

Subqueries

We can always normalize subqueries to use only EXISTS and NOT EXISTS [Van den Bussche, Vansummeren] [1,2]

```
SELECT name FROM MovieExec
WHERE netWorth >= ALL (SELECT E.netWorth
                        FROM MovieExec E)
```

```
⇒   SELECT name FROM MovieExec
    WHERE NOT EXISTS(SELECT E.netWorth
                     FROM MovieExec E
                     WHERE netWorth < E.netWorth)
```

1 Only valid for set-based Relations
2 https://cs.ulb.ac.be/public/_media/teaching/infoh417/sql2alg_eng.pdf

Subqueries

We can always normalize subqueries to use only EXISTS and NOT EXISTS [Van den Bussche, Vansummeren] [1,2]

```
SELECT C FROM S
WHERE C IN (SELECT SUM(B) FROM R
              GROUP BY A)
```

⇒   ?

1 Only valid for set-based Relations
2 https://cs.ulb.ac.be/public/_media/teaching/infoh417/sql2alg_eng.pdf

Subqueries

We can always normalize subqueries to use only EXISTS and NOT EXISTS [Van den Bussche, Vansummeren] [1,2]

```
SELECT C FROM S
WHERE C IN (SELECT SUM(B) FROM R
              GROUP BY A)
```

```
⇒   SELECT C FROM S
    WHERE EXISTS (SELECT SUM(B) FROM R
                  GROUP BY A
                  HAVING SUM(B) = C)
```

1 Only valid for set-based Relations
2 https://cs.ulb.ac.be/public/_media/teaching/infoh417/sql2alg_eng.pdf

Normalization

- Before translating a query we first normalize it such that all of the subqueries that occur in a WHERE condition are of the form EXISTS or NOT EXISTS.

Correlated Subqueries

A subquery can refer to attributes of relations that are introduced in an outer query.

```
SELECT movieTitle
FROM StarsIn S
WHERE EXISTS (SELECT name
              FROM MovieStar
              WHERE birthdate=1960 AND name=S.starName)
```

- **The “outer” relations are called the context relations of the subquery.**
- **The set of all attributes of all context relations of a subquery are called the parameters of the subquery.**

Translation of correlated select-from-where subqueries

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND EXISTS (SELECT name
              FROM MovieStar
              WHERE birthdate=1960 AND name= S.starName)
```

Translation of correlated select-from-where subqueries

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND EXISTS (SELECT name
              FROM MovieStar
              WHERE birthdate=1960 AND name= S.starName)
```

First translate the subquery

$$\pi_{name}\ \sigma_{birthdate=1960 \wedge name=S.starName}(MovieStar)$$

Translation of correlated select-from-where subqueries

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND EXISTS (SELECT name
            FROM MovieStar
            WHERE birthdate=1960 AND name= S.starName)
```

Fix: add the context relation and parameters

$$\pi_{\text{name},\text{S.movieTitle},\text{S.movieYear},\text{S.starName}} \; \sigma_{\text{birthdate}=1960 \wedge \text{name}=\text{S.starName}}(\text{MovieStar} \times \rho_S\,(\text{StarsIn}))$$

Translation of correlated select-from-where subqueries

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND EXISTS (SELECT name
               FROM MovieStar
               WHERE birthdate=1960 AND name= S.starName)
```

Next, translate the FROM clause of the outer query

$$\rho_S(\text{StarsIn}) \times \rho_M(\text{Movie})$$

Translation of correlated select-from-where subqueries

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND EXISTS (SELECT name
              FROM MovieStar
              WHERE birthdate=1960 AND name= S.starName)
```

Synchronize both expressions by means of a join.

$$\rho_S(\mathrm{StarsIn}) \times \rho_M(\mathrm{Movie}) \bowtie$$
$$(\pi_{name, S.movieTitle, S.movieYear, S.starName}$$
$$\sigma_{birthdate=1960 \wedge name=S.starName}(\mathrm{MovieStar} \times \rho_S(\mathrm{StarsIn})))$$

Translation of correlated select-from-where subqueries

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND EXISTS (SELECT name
              FROM MovieStar
              WHERE birthdate=1960 AND name= S.starName)
```

Simplify

$$\rho_M(\text{Movie}) \bowtie \left(\pi_{S.movieTitle,S.movieYear,S.starName}\ \sigma_{birthdate=1960 \wedge name=S.starName}(\text{MovieStar} \times \rho_S(\text{StarsIn}))\right)$$

Translation of correlated select-from-where subqueries

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND EXISTS (SELECT name
              FROM MovieStar
              WHERE birthdate=1960 AND name= S.starName)
```

Complete the expression

$$\pi_{S.movieTitle,M.studioName}\ \sigma_{S.movieYear>=2000 \land S.movieTitle=M.title}$$

$$(\rho_M(\text{Movie}) \bowtie$$

$$\pi_{S.movieTitle,S.movieYear,S.starName}$$

$$\sigma_{birthdate=1960 \land name=S.starName}(\text{MovieStar} \times \rho_S(\text{StarsIn})))$$

Wait !

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND EXISTS (SELECT name
              FROM MovieStar
              WHERE birthdate=1960 AND name= S.starName)
```

SQL Result ?

```
Movie
======
title studioName  movieYear
------------------------
DBSA      ULB       2005


StartsIn
========
starName  movieTitle
------------------------
Foo           DBSA


MovieStar
=========
name firstname birthdate
-------------------------
Foo   Bar       1960
Foo   Baz       1960
```

Wait !

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND EXISTS (SELECT name
              FROM MovieStar
              WHERE birthdate=1960 AND name= S.starName)
```

```
Movie
=====
title studioName  movieYear
-----------------------
DBSA      ULB       2005


StartsIn
========
starName  movieTitle
-----------------------
Foo            DBSA


MovieStar
=========
name firstname birthdate
--------------------------
Foo   Bar        1960
Foo   Baz        1960
```

SQL Result ?

```
movieTitle studioName
------------------------------------
DBSA       ULB
```

ULB

Wait !

$$\pi_{S.movieTitle,M.studioName}$$
$$\sigma_{S.movieYear>=2000 \wedge S.movieTitle=M.title}$$
$$(\rho_M(Movie) \bowtie$$
$$\pi_{S.movieTitle,S.movieYear,S.starName}$$
$$\sigma_{birthdate=1960 \wedge name=S.starName}($$
$$MovieStar \times \rho_S(StarsIn)))$$

RA Result ?

```
Movie
=====
title studioName  movieYear
-----------------------
DBSA     ULB      2005


StartsIn
========
starName  movieTitle
-----------------------
Foo           DBSA


MovieStar
=========
name firstname birthdate
--------------------------
Foo   Bar      1960
Foo   Baz      1960
```

ULB

Wait !

$$\pi_{S.movieTitle,M.studioName}$$
$$\sigma_{S.movieYear>=2000 \wedge S.movieTitle=M.title}$$
$$(\rho_M(Movie) \bowtie$$
$$\pi_{S.movieTitle,S.movieYear,S.starName}$$
$$\sigma_{birthdate=1960 \wedge name=S.starName}($$
$$MovieStar \times \rho_S(StarsIn)))$$

RA Result ?

movieTitle	studioName
DBSA	ULB
DBSA	ULB

```
Movie
=====
title studioName  movieYear
------------------------
DBSA     ULB       2005


StartsIn
========
starName  movieTitle
------------------------
Foo              DBSA


MovieStar
=========
name firstname birthdate
---------------------------
Foo    Bar       1960
Foo    Baz       1960
```

ULB

Wait !

$$\pi_{S.movieTitle,M.studioName}\ \sigma_{S.movieYear>=2000 \wedge S.movieTitle=M.title}\ \rho_M(\text{Movie}) \bowtie \pi_{S.movieTitle,S.movieYear,S.starName}\ \sigma_{birthdate=1960 \wedge name=S.starName}(\text{MovieStar} \times \rho_S(\text{StarsIn}))$$

Only set-based Relations

RA Result ?

```
movieTitle studioName
---------------------------------
DBSA        ULB
DBSA        ULB
```

```
Movie
=====
title studioName  movieYear
------------------------
DBSA      ULB       2005


StartsIn
========
starName  movieTitle
------------------------
Foo            DBSA


MovieStar
=========
name firstname birthdate
--------------------------
Foo   Bar       1960
Foo   Baz       1960
```

ULB

Flattening Subqueries in Bag-based Relations (probably all vendor implementations)

The requirements for flattening into a normal join are:

- There is a uniqueness condition that ensures that the subquery does not introduce any duplicates if it is flattened into the outer query block.
- Each table in the subquery's FROM list (after any view, derived table, or subquery flattening) must be a base table.
- The subquery is not under an OR.
- The subquery is not in the SELECT list of the outer query block.
- The subquery type is EXISTS, IN, or ANY, or it is an expression subquery on the right side of a comparison operator.

https://docs.oracle.com/javadb/10.8.3.0/tuning/ctuntransform36368.html

Flattening Subqueries in Bag-based Relations (probably all vendor implementations)

- There are no aggregates in the SELECT list of the subquery.
- The subquery does not have a GROUP BY clause.
- The subquery does not have an ORDER BY, result offset, or fetch first clause.
- If there is a WHERE clause in the subquery, there is at least one table in the subquery whose columns are in equality predicates with expressions that do not include any column references from the subquery block. These columns must be a superset of the key columns for any unique index on the table. For all other tables in the subquery, the columns in equality predicates with expressions that do not include columns from the same table are a superset of the unique columns for any unique index on the table.

https://docs.oracle.com/javadb/10.8.3.0/tuning/ctuntransform36368.html

System R: Relational Approach to Database Management

M. M. ASTRAHAN, M. W. BLASGEN, D. D. CHAMBERLIN,
K. P. ESWARAN, J. N. GRAY, P. P. GRIFFITHS,
W. F. KING, R. A. LORIE, P. R. McJONES, J. W. MEHL,
G. R. PUTZOLU, I. L. TRAIGER, B. W. WADE, AND V. WATSON

IBM Research Laboratory

To read before the next lecture. We will discuss it in the lecture. Only read until end of The Optimizer section (unless you fall in love with it)

https://www.seas.upenn.edu/~zives/cis650/papers/System-R.PDF

ULB

Credits

Many slides are copied from:

- Stijn Vansummeren, Database Systems Architecture course slides.