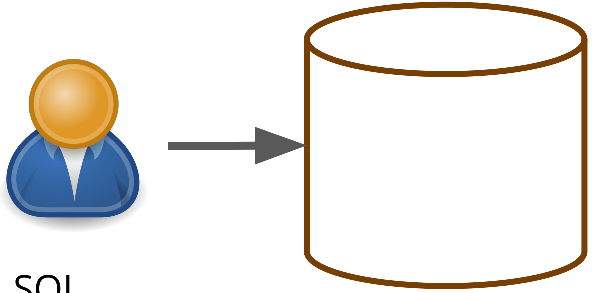
INFOH417 Database System Architectures

Mahmoud SAKR <mahmoud.sakr@ulb.be>

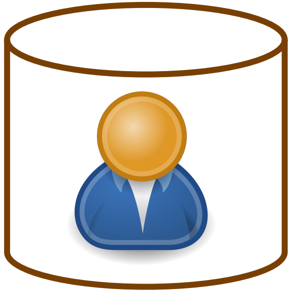
École polytechnique de Bruxelles

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

ULB

Course Topics

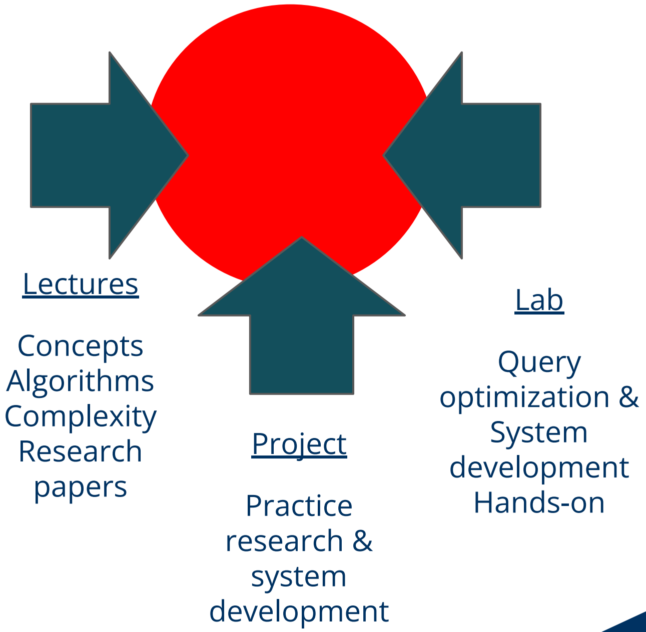
- Query execution
- Refreshing SQL and relational Algebra
- Cost-based query optimization
- Indexes
- Crash recovery
- Concurrency control
- Distributed databases

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]()



ULB

Recommended Readings

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

Lecture 1: 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)`

ULB

select-from-where

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

RA ?

ULB

select-from-where

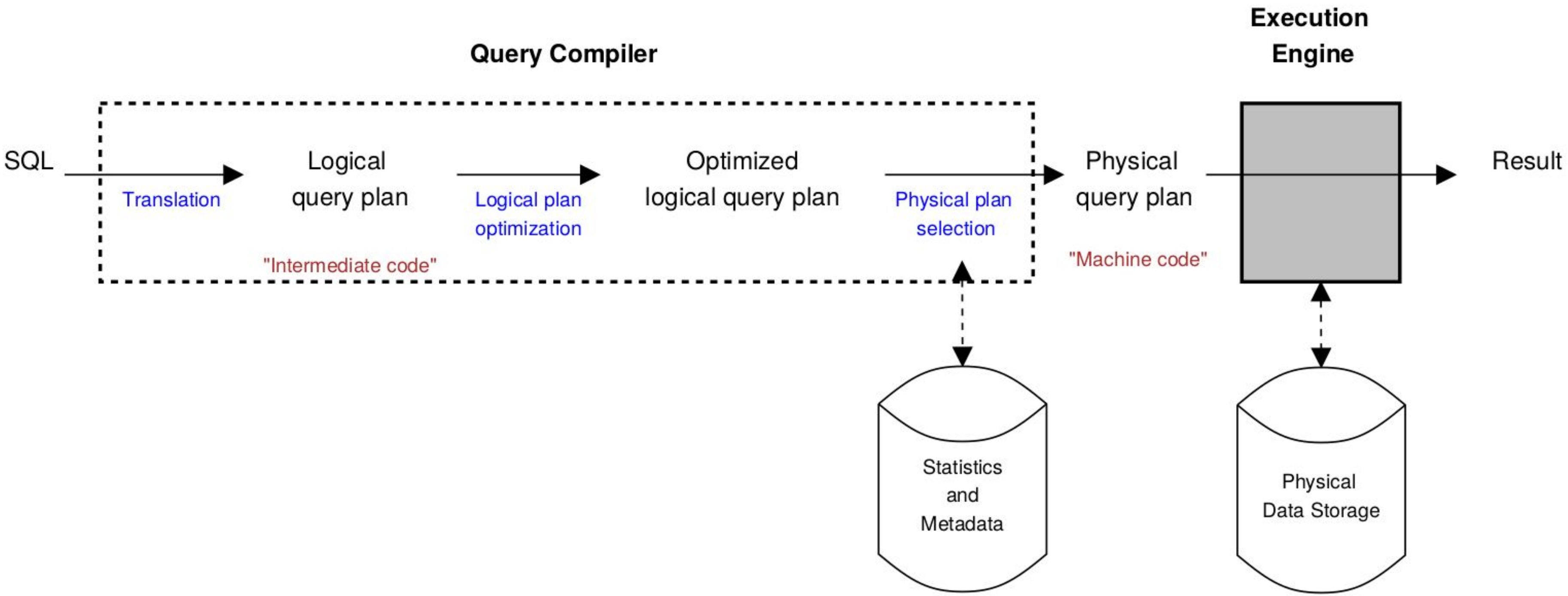
```
SQL:
SELECT movieTitle
FROM   StarsIn S, MovieStar M
WHERE  S.starName = M.name AND M.birthdate = 1960
```

RA ?

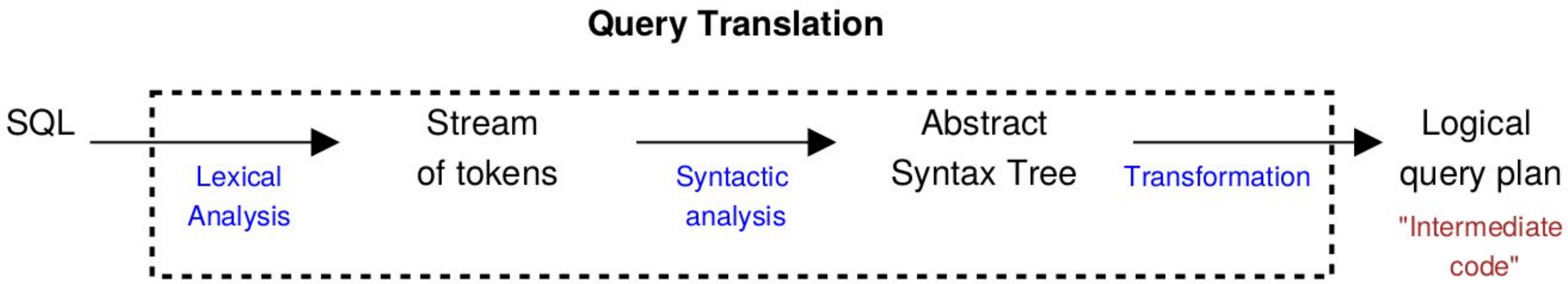
$\pi_{\text{movieTitle}}\ \sigma_{\text{S.starName=M.name and M.birthdate=1960}}(\rho_S(\text{StarsIn}) \times \rho_M(\text{MovieStar}))$

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.

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

ULB

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

ULB

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

SELECT * FROM (SELECT * FROM Movie where length>=100) JOIN StarsIn ON title=movietitle

SELECT * FROM Movie, StarsIn WHERE length>=100 AND title=movietitle

The Extended Relational Algebra

Add more operators

Extended projection

allows renaming

π

A,C->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

ULB

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 ?

$$\gamma_{\texttt{M.movieTitle,count(S.starName)->numStars}}($$

$$\rho_S(\texttt{StarsIn}) \bowtie_{\texttt{S.starName=M.name}} \rho_M(\texttt{MovieStar}))$$

ULB

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 ?

$$\sigma_{\texttt{numStarts>5}}(\gamma_{\texttt{M.movieTitle,count(S.starName)->numStars}}($$

$$\rho_{\texttt{S}}(\texttt{StarsIn}) \bowtie_{\texttt{S.starName=M.name}} \rho_{\texttt{M}}(\texttt{MovieStar})))$$

Subqueries

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

V.S.

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

Which query is better ?

PostgreSQL Source Code git master


```
    /*
     * 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.

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

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

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

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)
```

Complete the expression

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

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

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

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

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)
```

SQL Result ?

```
movieTitle studioName
---------------------------------
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 !

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

RA Result ?

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


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


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

Wait !

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

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
```

Wait !

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

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
```

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

Credits

Many slides are copied from:

- Stijn Vansummeren, Database Systems Architecture course slides.

Lecture 2: Query Optimization

System R paper

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

ULB

System R paper - metadata

- Coming from IBM
- Published in ACM TODS
- Authors including:
    - Jim Gray - Turing award 1998, and others
    - Bruce G. Lindsay - ACM SIGMOD Edgar F. Codd Innovations Award, 2012, and others
    - Patricia G. Selinger - SIGMOD Edgar F. Codd Innovations Award, and others

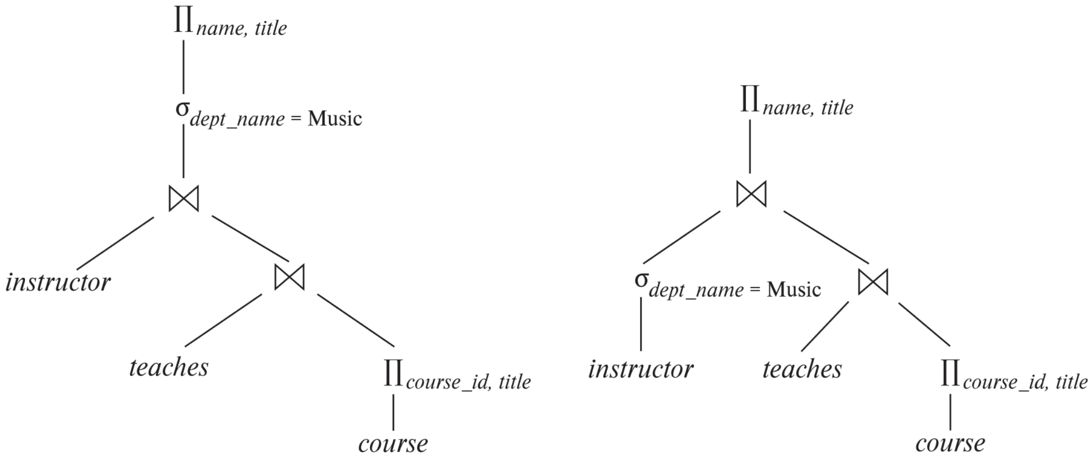
System R paper - Discussion

1. What are the architecture components of system R ?
2. Which language did system R use for querying ?
3. What is a catalogue ?
4. What is a cursor ? Is it still used ?
5. What is a clustering image ?
6. How did the system R optimizer work ?
7. What are the parameters of cost estimation ?
8. What is an access path ?

Query Optimization

Alternative ways of evaluating a given query

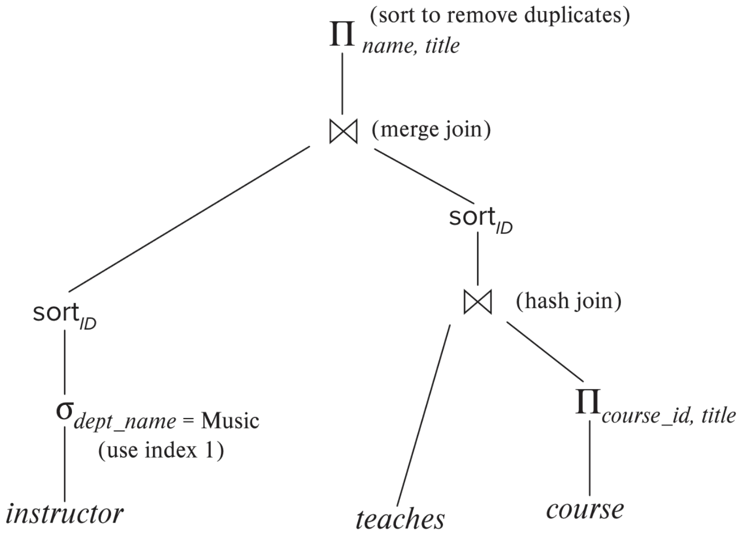
- Equivalent expressions
- Different algorithms for each operation



(a) Initial expression tree (b) Transformed expression tree

Query Plan

An evaluation plan defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.



Find out how to view query execution plans on your favorite database

Cost-based Query Optimization

- Cost difference between evaluation plans for a query can be enormous
  - E.g., seconds vs. days in some cases
- Steps in cost-based query optimization
  a. Generate logically equivalent expressions using equivalence rules
  b. Annotate resultant expressions to get alternative query plans
  c. Choose the cheapest plan based on estimated cost
- Estimation of plan cost based on:
  - Statistical information about relations. Examples:
    - number of tuples, number of distinct values for an attribute
  - Statistics estimation for intermediate results
    - to compute cost of complex expressions
  - Cost formulae for algorithms, computed using statistics

Viewing Query Evaluation Plans

- Most database support explain <query>
  - Displays plan chosen by query optimizer, along with cost estimates
  - Some syntax variations between databases
    - Oracle: **explain plan for <query>** followed by **select * from table** (dbms_xplan.display)
    - SQL Server: set showplan_text on
- Some databases (e.g. PostgreSQL) support explain analyse <query>
  - Shows actual runtime statistics found by running the query, in addition to showing the plan
- Some databases (e.g. PostgreSQL) show cost as *f..l*
  - *f* is the cost of delivering first tuple and *l* is cost of delivering all results

Generating Equivalent Expressions

Transformation of Relational Expressions

- Two relational algebra expressions are said to be equivalent if the two expressions generate the same set of tuples on every legal database instance
    - Note: order of tuples is irrelevant
- In SQL, inputs and outputs are multisets (bag) of tuples
    - Two expressions in the multiset version of the relational algebra are said to be equivalent if the two expressions generate the same multiset of tuples on every legal database instance.
- An equivalence rule says that expressions of two forms are equivalent, so one can replace the other

Equivalence Rules

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.

$$\sigma_{\theta 1 \wedge \theta 2}(E) \equiv \sigma_{\theta 1}(\sigma_{\theta 2}(E))$$

2. Selection operations are commutative.

$$\sigma_{\theta 1}(\sigma_{\theta 2}(E)) \equiv \sigma_{\theta 2}(\sigma_{\theta 1}(E))$$

3. Only the last in a sequence of projection operations is needed, the others can be omitted (where L1 $\subseteq$ L2 … $\subseteq$ Ln)

$$\prod_{L1}(\prod_{L2}(\ldots(\prod_{Ln}(E))\ldots)) \equiv \prod L_1(E)$$

4. Selections can be combined with Cartesian products and theta joins.

$$\sigma_{\theta}(E_1 \text{ x } E_2) \equiv E_1 \bowtie_{\theta} E_2$$

$$\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) \equiv E_1 \bowtie_{\theta 1 \wedge \theta 2} E$$

Equivalence Rules (Cont.)

5. Theta-join operations (and natural joins) are commutative.

$$E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$$

6. (a) Natural join operations are associative:

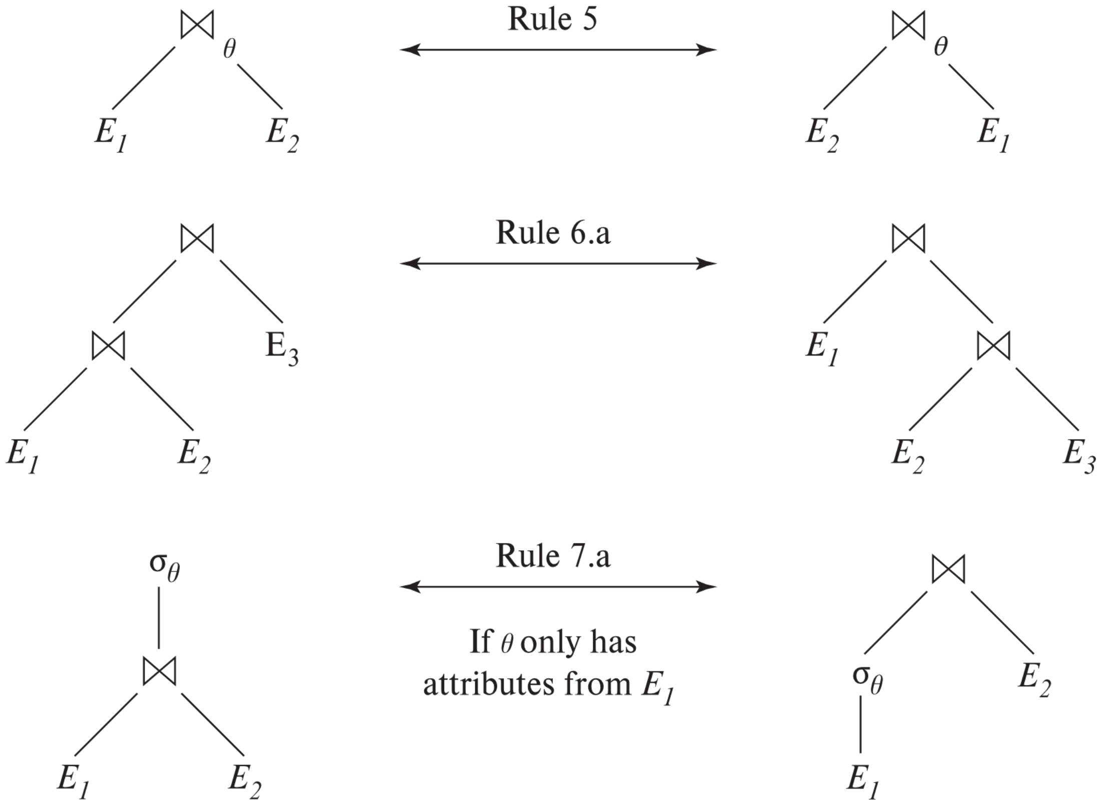
$$(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$$

(b) Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 \equiv E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$$

where θ_2 involves attributes from only E_2 and E_3

Pictorial Depiction of Equivalence Rules



Equivalence Rules (Cont.)

7. The selection operation distributes over the theta join operation under the following two conditions:

   (a) When all the attributes in θ_0 involve only the attributes of one of the expressions (E1) being joined.

$$\sigma_{\theta 0} (E_1 \bowtie_\theta E_2) \equiv (\sigma_{\theta 0}(E_1)) \bowtie_\theta E_2$$

   (b) When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.

$$\sigma_{\theta 1 \land \theta 2}(E_1 \bowtie_\theta E_2) \equiv (\sigma_{\theta 1}(E_1)) \bowtie_\theta (\sigma_{\theta 2}(E2))$$

Equivalence Rules (Cont.)

8. The projection operation distributes over the theta join operation as follows:

   (a) if θ involves only attributes from $L_1 \cup L_2$:

$$\Pi_{L1 \cup L2}(E_1 \bowtie_\theta E_2) \equiv \Pi_{L1}(E_1) \bowtie_\theta \Pi_{L2}(E_2)$$

Similar equivalence hold for outerjoin operations: ⟖, ⟕, and ⟗

Equivalence Rules (Cont.)

9. The set operations union and intersection are commutative

$$E_1 \cup E_2 \equiv E_2 \cup E_1 \qquad E_1 \cap E_2 \equiv E_2 \cap E_1$$

(set difference is not commutative).

10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 \equiv E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 \equiv E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over $\cup$, $\cap$ and –.

$$\sigma_\theta (E_1 \cup E_2) \equiv \sigma_\theta (E_1) \cup \sigma_\theta(E_2)$$
$$\sigma_\theta (E_1 \cap E_2) \equiv \sigma_\theta (E_1) \cap \sigma_\theta(E_2)$$
$$\sigma_\theta (E_1 - E_2) \equiv \sigma_\theta (E_1) - \sigma_\theta(E_2)$$
$$\sigma_\theta (E_1 \cap E_2) \equiv \sigma_\theta(E_1) \cap E_2$$
$$\sigma_\theta (E_1 - E_2) \equiv \sigma_\theta(E_1) - E_2 \text{ (does not hold for } \cup\text{)}$$

12. The projection operation distributes over union

$$\Pi_L(E_1 \cup E_2) \equiv (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

Exercise

- Create equivalence rules involving
  - The group by/aggregation operation
  - Left outer join operation

Transformation Example: Pushing Selections

Query: Find the names of all instructors in the Music department, along with the titles of the courses that they teach

$$\Pi_{name,\ title}(\sigma_{dept\_name=\ \text{'Music'}}(instructor \bowtie (teaches \bowtie \Pi_{course\_id,\ title}\ (course))))$$

Transformation using rule 7a.

$$\Pi_{name,\ title}((\sigma_{dept\_name=\ \text{'Music'}}(instructor)) \bowtie (teaches \bowtie \Pi_{course\_id,\ title}\ (course)))$$

Performing the selection as early as possible reduces the size of the relation to be joined.

Example with Multiple Transformations

Query: Find the names of all instructors in the Music department who have taught a course in 2017, along with the titles of the courses that they taught

$$\Pi_{name,title}(\sigma_{dept\_name= \text{"Music"} \wedge year=2017}(instructor \bowtie (teaches \bowtie \Pi_{course\_id,\ title}\ (course))))$$

Transformation using join associatively (Rule 6a):

$$\Pi_{name,\ title}(\sigma_{dept\_name= \text{"Music"} \wedge year\ =\ 2017}((instructor \bowtie teaches) \bowtie \Pi_{course\_id,\ title}\ (course)))$$

Second form provides an opportunity to apply the "perform selections early" rule, resulting in the subexpression

$$\sigma_{dept\_name\ =\ \text{"Music"}}\ (instructor) \bowtie \sigma_{\ year\ =\ 2017}\ (teaches)$$

Transformation Example: Pushing Projections

Consider:

$$\Pi_{name,title}(\sigma_{dept\_name=\text{"Music"}}(instructor) \bowtie teaches) \bowtie \Pi_{course\_id,title}(course)))$$

When we compute

$$(\sigma_{dept\_name = \text{"Music"}}(instructor \bowtie teaches)$$

we obtain a relation whose schema is:

(ID, name, dept_name, salary, course_id, sec_id, semester, year)

Push projections using equivalence rules 8a and 8b; eliminate unneeded attributes from intermediate results to get:

$$\Pi_{name,title}(\Pi_{name,course\_id}(\sigma_{dept\_name=\text{"Music"}}(instructor) \bowtie teaches)) \bowtie \Pi_{course\_id,\ title}(course))))$$

Performing the projection as early as possible reduces the size of the relation to be joined.

Join Ordering Example

For all relations r_1, r_2, and r_3,

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

(Join Associativity) $\bowtie$

If $r_2 \bowtie r_3$ is quite large and $r_1 \bowtie r_2$ is small, we choose

$$(r_1 \bowtie r_2) \bowtie r_3$$

so that we compute and store a smaller temporary relation.

Join Ordering Example (Cont.)

Consider the expression

$\Pi_{name, title}(\sigma_{dept\_name=\text{“Music”}}(instructor) \bowtie teaches) \bowtie \Pi_{course\_id,\ title}\ (course)))$

Could compute

$teaches \bowtie \Pi_{course\_id,\ title}\ (course)$ first,

and join result with

$\sigma_{dept\_name=\ \text{“Music”}}\ (instructor)$

but the result of the first join is likely to be a large relation.

Only a small fraction of the university's instructors are likely to be from the Music department. It is better to first compute

$\sigma_{dept\_name=\ \text{“Music”}}\ (instructor) \bowtie teaches$

Enumeration of Equivalent Expressions

- Query optimizers use equivalence rules to **systematically** generate expressions equivalent to the given expression
- Can generate all equivalent expressions as follows:

Repeat

apply all applicable equivalence rules on every subexpression of every equivalent expression found so far

add newly generated expressions to the set of equivalent expressions

Until no new equivalent expressions are generated above

Cost Estimation

- Cost of each operator (not detailed here)
  - Need statistics of input relations
    - E.g., number of tuples, sizes of tuples
- Inputs can be results of sub-expressions
  - Need to estimate statistics of expression results
  - To do so, we require additional statistics
    - E.g., number of distinct values for an attribute

Choice of Evaluation Plans

- Must consider the interaction of evaluation techniques when choosing evaluation plans
  - choosing the cheapest algorithm for each operation independently may not yield best overall algorithm. E.g.
    - merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost for an outer level aggregation.
    - nested-loop join may provide opportunity for pipelining
- Practical query optimizers incorporate elements of the following two broad approaches:
  - 1. Search all the plans and choose the best plan in a cost-based fashion.
  - 2. Uses heuristics to choose a plan.

Cost-Based Optimization

- Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie \ . . . \bowtie r_n$.
- There are $(2(n - 1))!/(n - 1)!$ different join orders for above expression. With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!
- No need to generate all the join orders. Using dynamic programming, the least-cost join order for any subset of $\{r_1, r_2, . . . r_n\}$ is computed only once and stored for future use.

Dynamic Programming in Optimization

- To find best join tree for a set of *n* relations:
  - To find best plan for a set *S* of *n* relations, consider all possible plans of the form: $S_1 \bowtie (S - S_1)$ where S_1 is any non-empty subset of *S*.
  - Recursively compute costs for joining subsets of *S* to find the cost of each plan. Choose the cheapest of the $2^n - 2$ alternatives.
  - Base case for recursion: single relation access plan
    - Apply all selections on R_i using best choice of indices on R_i
  - When plan for any subset is computed, store it and reuse it when it is required again, instead of recomputing it
    - Dynamic programming

Join Order Optimization Algorithm

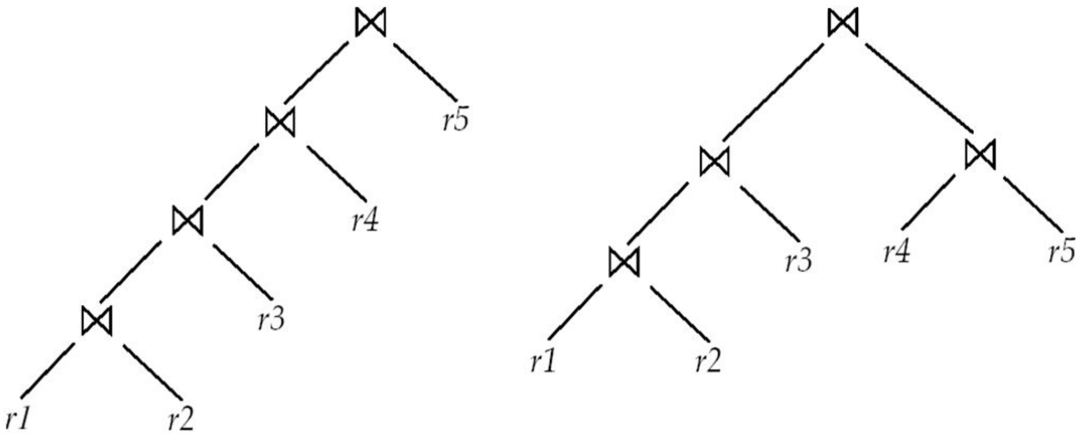
```
procedure findbestplan(S)
   if (bestplan[S].cost ≠ ∞)
    return bestplan[S]
   // else bestplan[S] has not been computed earlier, compute it now
   if (S contains only 1 relation)
       set bestplan[S].plan and bestplan[S].cost based on the best way
       of accessing S  using selections on S and indices (if any) on S
   else for each non-empty subset S1 of S such that S1 ≠ S
    P1= findbestplan(S1)
    P2= findbestplan(S - S1)
    A= Best Algorithm for joining results of P1 and P2
    cost= P1.cost + P2.cost + A.cost

    if cost < bestplan[S].cost
            bestplan[S].cost = cost
            bestplan[S].plan = plan;
    return bestplan[S]
```

Cost of Cost-based Optimization !

- With dynamic programming, the time complexity of join order optimization is $O(3^n)$.
  - With $n = 10$, this number is 59000 instead of 176 billion!
- Space complexity is $O(2^n)$
- System R restricts only to left-deep join trees - good for pipelined execution

Left Deep Join Trees

- In **left-deep join trees,** the right-hand-side input for each join is a relation, not the result of an intermediate join.



(a) Left-deep join tree

(b) Non-left-deep join tree

Left Deep Join Trees - Reduced Cost of Optimization

- For a set of *n* relations:
  - Consider *n* alternatives with one relation as right-hand side input and the other relations as left-hand side input.

- Time complexity of finding best join order is $O(n\ 2^n)$ - compared to (3^n)
  - Space complexity remains at $O(2^n)$
- Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10)

Volcano - recommended reading

The Volcano Optimizer Generator: Extensibility and Efficient Search

Goetz Graefe, Portland State University
William J. McKenna, University of Colorado at Boulder

https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.21.2197&rep=rep1&type=pdf

Heuristic Optimization

- Cost-based optimization is expensive, even with dynamic programming.
- Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion.
- Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Perform most restrictive selection and join operations (i.e., with smallest result size) before other similar operations.
  - Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

Heuristic Optimization - Apache Spark

spark/Optimizer.scala at

https://github.com/apache/spark/blob/master/sql/catalyst/src/main/scala/org/apache/spark/sql/catalyst/optimizer/Opti

143%

```scala
   * Defines the default rule batches in the Optimizer.
   *
   * Implementations of this class should override this method, and [[nonExcludableRules]] if
   * necessary, instead of [[batches]]. The rule batches that eventually run in the Optimizer,
   * i.e., returned by [[batches]], will be (defaultBatches - (excludedRules - nonExcludableRules)).
   */
  def defaultBatches: Seq[Batch] = {
    val operatorOptimizationRuleSet =
      Seq(
        // Operator push down
        PushProjectionThroughUnion,
        ReorderJoin,
        EliminateOuterJoin,
        PushDownPredicates,
        PushDownLeftSemiAntiJoin,
        PushLeftSemiLeftAntiThroughJoin,
        LimitPushDown,
        LimitPushDownThroughWindow,
        ColumnPruning,
        // Operator combine
        CollapseRepartition,
```

Structure of Query Optimizers (Cont.)

- Some query optimizers integrate heuristic selection and the generation of alternative access plans.
  - Frequently used approach
    - heuristic rewriting of nested block structure and aggregation
    - followed by cost-based join-order optimization for each block
- **Optimization cost budget** to stop optimization early (if cost of plan is less than cost of optimization)
- **Plan caching** to reuse previously computed plan if query is resubmitted
    - Even with different constants in query
- Even with the use of heuristics, cost-based query optimization imposes a substantial overhead.
  - But is worth it for expensive queries
  - Optimizers often use simple heuristics for very cheap queries, and perform exhaustive enumeration for more expensive queries

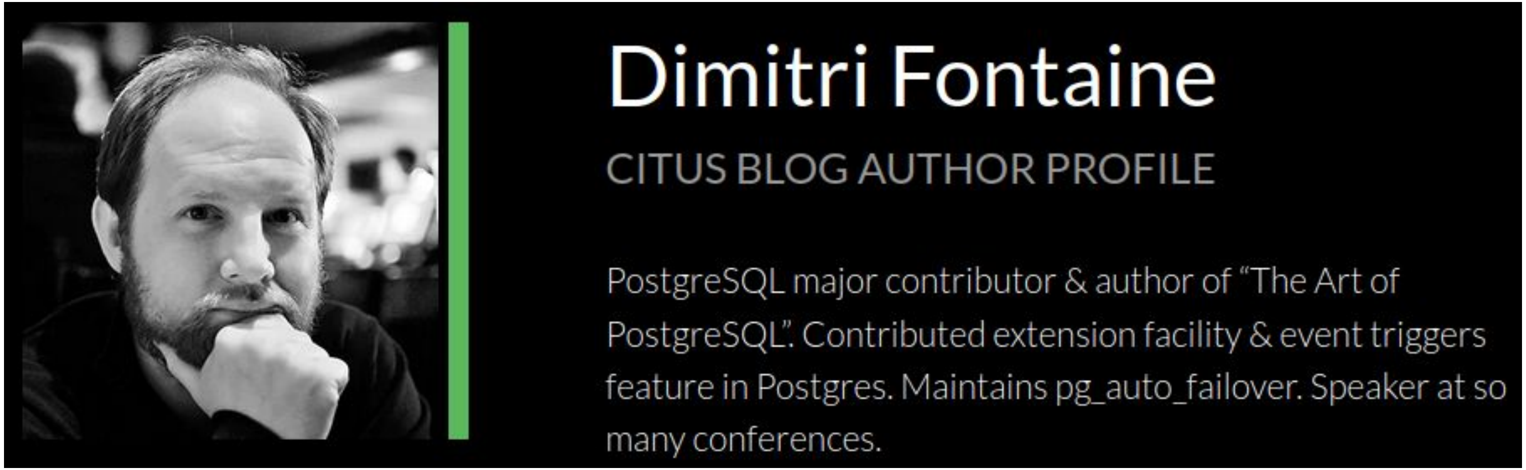
Credits

The slides of this lecture are taken from:

- Avi Silberschatz, Henry F. Korth, S. Sudarshan. Database System Concepts

Lecture 3: Invited Lecture

Extensible Databases



Lecture 5: Statistics for Cost Estimation

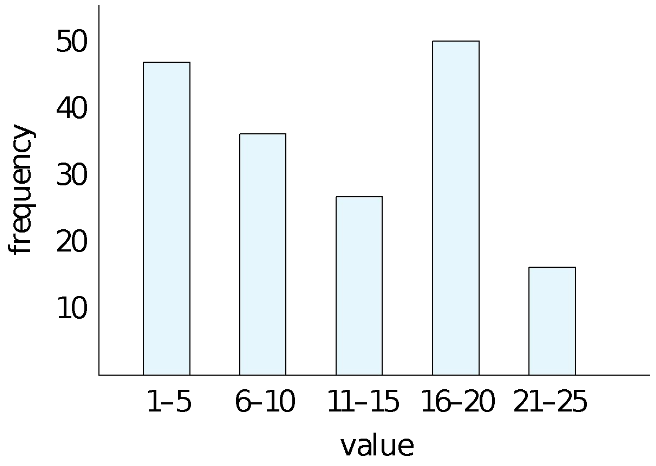
Statistical Information for Cost Estimation

- n_r: number of tuples in a relation *r.*
- b_r: number of blocks containing tuples of *r.*
- l_r: size of a tuple of *r.*
- f_r: blocking factor of *r* — i.e., the number of tuples of *r* that fit into one block.
- *V(A, r):* number of distinct values that appear in *r* for attribute *A;* same as the size of $\prod_A(r)$.
- If tuples of *r* are stored together physically in a file, then:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

Histograms

- Histogram on attribute *age* of relation *person*



- **Equi-width** histograms
- **Equi-depth** histograms break up range such that each range has (approximately) the same number of tuples
    - E.g. (4, 8, 14, 19)
- Many databases also store *n* **most-frequent values** and their counts
    - Histogram is built on remaining values only

Histograms (cont.)

- Histograms and other statistics usually computed based on a **random sample**
- Statistics may be out of date
  - Some database require a **analyze (vacuum)** command to be executed to update statistics
  - Others automatically recompute statistics
    - e.g., when number of tuples in a relation changes by some percentage

Postgres Statistics

Table 51.89. pg_stats Columns

Column Type Description
schemaname name (references pg_namespace.nspname) Name of schema containing table
tablename name (references pg_class.relname) Name of table
attname name (references pg_attribute.attname) Name of the column described by this row
inherited bool If true, this row includes inheritance child columns, not just the values in the specified table
null_frac float4 Fraction of column entries that are null
avg_width int4 Average width in bytes of column's entries
n_distinct float4 If greater than zero, the estimated number of distinct values in the column. If less than zero, the negative of the number of distinct values divided by the number of rows. (The negated form is used when ANALYZE believes that the number of distinct values is likely to increase as the table grows; the positive form is used when the column seems to have a fixed number of possible values.) For example, -1 indicates a unique column in which the number of distinct values is the same as the number of rows.

`most_common_vals anyarray`
: A list of the most common values in the column. (Null if no values seem to be more common than any others.)

`most_common_freqs float4[]`
: A list of the frequencies of the most common values, i.e., number of occurrences of each divided by total number of rows. (Null when `most_common_vals` is.)

`histogram_bounds anyarray`
: A list of values that divide the column's values into groups of approximately equal population. The values in `most_common_vals`, if present, are omitted from this histogram calculation. (This column is null if the column data type does not have a < operator or if the `most_common_vals` list accounts for the entire population.)

`correlation float4`
: Statistical correlation between physical row ordering and logical ordering of the column values. This ranges from -1 to +1. When the value is near -1 or +1, an index scan on the column will be estimated to be cheaper than when it is near zero, due to reduction of random access to the disk. (This column is null if the column data type does not have a < operator.)

`most_common_elems anyarray`
: A list of non-null element values most often appearing within values of the column. (Null for scalar types.)

`most_common_elem_freqs float4[]`
: A list of the frequencies of the most common element values, i.e., the fraction of rows containing at least one instance of the given value. Two or three additional values follow the per-element frequencies; these are the minimum and maximum of the preceding per-element frequencies, and optionally the frequency of null elements. (Null when `most_common_elems` is.)

`elem_count_histogram float4[]`
: A histogram of the counts of distinct non-null element values within the values of the column, followed by the average number of distinct non-null elements. (Null for scalar types.)

Selection Size Estimation

$\sigma_{A=v}(r)$

- $n_r / V(A,r)$: number of records that will satisfy the selection
- Equality condition on a key attribute: *size estimate* = 1

$\sigma_{A \le V}(r)$ (case of $\sigma_{A \ge V}(r)$ is symmetric)

- Let c denote the estimated number of tuples satisfying the condition.
- If min(A,r) and max(A,r) are available in catalog
    - c = 0 if v < min(A,r)
    - c = $n_r . \dfrac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$
    - If histograms available, can refine above estimate
    - In absence of statistical information *c* is assumed to be $n_r / 2$.

Size Estimation of Complex Selections

- The **selectivity** of a condition θ_i is the probability that a tuple in the relation *r* satisfies θ_i .
  - If s_i is the number of satisfying tuples in *r*, the selectivity of θ_i is given by s_i / n_r.
- **Conjunction:** $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}(r)$. *Assuming independence,* estimate of tuples in the result is: $n_r * \dfrac{s_1 * s_2 * \ldots * s_n}{n_r^n}$
- **Disjunction:** $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n}(r)$. Estimated number of tuples:

$$n_r * \left(1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \ldots * (1 - \frac{s_n}{n_r}) \right)$$

- **Negation:** $\sigma_{\neg\theta}(r)$. Estimated number of tuples: $n_r - size(\sigma_\theta(r))$

Join Operation: Running Example

Running example: *student* ⋈ *takes*

Catalog information for join examples:

- $n_{student}$ = 5,000. $f_{student}$ = *50,* which implies that $b_{student}$ =5000/50 = 100.
- n_{takes} = 10000. f_{takes} = 25, which implies that b_{takes} = 10000/25 = 400.
- *V(ID, takes)* = 2500, which implies that on average, each student who has taken a course has taken 4 courses.
  - Attribute *ID* in *takes* is a foreign key referencing *student.*
  - *V*(*ID, student*) = 5000 (*primary key!*)

```
create table student
   (ID              varchar(5),
    name            varchar(20) not null,
    dept_name       varchar(20),
    tot_cred        numeric(3,0) check (tot_cred >= 0),
    primary key (ID),
    foreign key (dept_name) references department (dept_name)
          on delete set null
   );
```

```
create table takes
   (ID              varchar(5),
    course_id       varchar(8),
    sec_id          varchar(8),
    semester        varchar(6),
    year            numeric(4,0),
    grade           varchar(2),
    primary key (ID, course_id, sec_id, semester, year),
    foreign key (course_id, sec_id, semester, year) references section
(course_id, sec_id, semester, year)
          on delete cascade,
    foreign key (ID) references student (ID)
          on delete cascade
   );
```

Estimation of the Size of Joins

- The Cartesian product $r \times s$ contains $n_r . n_s$ tuples; each tuple occupies $s_r + s_s$ bytes.
- If $R \cap S = \varnothing$, then $r \bowtie s$ is the same as $n_r \times n_s$.
- If $R \cap S$ is a key for R, then a tuple of s will join with at most one tuple from r
  - therefore, the number of tuples in $r \bowtie s$ is no greater than the number of tuples in s.
- If $R \cap S$ *in* S is a foreign key in S referencing R, then the number of tuples in $r \bowtie s$ is exactly the same as the number of tuples in s.
    - The case for $R \cap S$ being a foreign key referencing S is symmetric.
- In the example query *student* $\bowtie$ *takes, ID* in *takes* is a foreign key referencing *student*
  - hence, the result has exactly n_{takes} tuples, which is 10000

Estimation of the Size of Joins (Cont.)

- If $R \cap S = \{A\}$ is not a key for R or S.
  If we assume that every tuple t in R produces tuples in $R \bowtie S$, the number of tuples in $R \bowtie S$ is estimated to be:

  $$\frac{n_r * n_s}{V(A,s)}$$

  If the reverse is true, the estimate obtained will be:

  $$\frac{n_r * n_s}{V(A,r)}$$

  The lower of these two estimates is probably the more accurate one.
- Can improve on above if histograms are available
  - Use formula similar to above, for each cell of histograms on the two relations

Estimation of the Size of Joins (Cont.)

- Compute the size estimates for *student* ⋈ *takes* without using information about foreign keys:
  - *V(ID, takes)* = 2500, and
    V(ID, student) = 5000
  - The two estimates are 5000 * 10000/2500 = 20,000 and 5000 * 10000/5000 = 10000
  - We choose the lower estimate, which in this case, is the same as our earlier computation using foreign keys.

The Internals of PostgreSQL

Chapter 3

Query Processing

https://www.interdb.jp/pg/pgsql03.html

Postgres optimizer code snippets

Postgres genetic query optimizer

https://www.postgresql.org/docs/13/geqo-intro.html

https://doxygen.postgresql.org/geqo_8h_source.html

var=const selectivity

https://doxygen.postgresql.org/selfuncs_8h.html#a31ee9824c23028c56ca3d6ca92c39a7e

Range typanalyze

https://doxygen.postgresql.org/rangetypes__typanalyze_8c_source.html

Range overlap

https://github.com/postgres/postgres/blob/cd3f429d9565b2e5caf0980ea7c707e37bc3b317/src/include/catalog/pg_operator.dat#L3110

rangesel

https://doxygen.postgresql.org/rangetypes__selfuncs_8c.html#a632d39f45c72d18cf792fb33014155ee

Selectivity Estimation of Inequality Joins In Databases

Diogo Repas, Zhicheng Luo, Maxime Schoemans, Mahmoud Sakr

https://arxiv.org/abs/2206.07396

Credits

Many slides in this lecture are taken from:

- Avi Silberschatz, Henry F. Korth, S. Sudarshan. Database System Concepts

Recommended reading

- The Internals of PostgreSQL (https://www.interdb.jp/pg/)