

IN3020&4020 – Database Systems (2022)

Part 1: Query Processing

Lectures 11, 12: Query Compilation and Optimisation

21, 24 February

Egor V. Kostylev

IFI, University of Oslo

egork@ifi.uio.no



Big Picture (reminder)

Syllabus of the course:

- (Extended) Intro and SQL recap 

Part 1. Query processing in relational databases

Part 2. Transaction management in relational databases

Part 3. NoSQL DBMS

Big Picture (reminder)

Syllabus of the course:

- (Extended) Intro and SQL recap ✓

Part 1. Query processing in relational databases

Part 2. Transaction management in relational databases

Part 3. NoSQL DBMS

SQL query processing relies on two key ingredients:

Relational Algebra (✓) and Indexes (✓)

We have studied these ingredients (almost in isolation)
and now put all things together

SQL (DQL) query processor in action

```
SELECT DISTINCT title FROM StarsIn WHERE starName IN  
(SELECT name FROM MovieStar WHERE birthDate LIKE '%1960');
```

↓ parsing

(parse tree)

↓ compilation

$$\pi_{title}(\sigma_{starName=name}(\mathbf{StarsIn} \times \pi_{name}(\sigma_{birthDate \text{ LIKE } '%1960'}(\mathbf{MovieStar}))))$$

↓ optimisation

$$\pi_{title}(\mathbf{StarsIn} \bowtie_{starName=name} \pi_{name}(\sigma_{birthDate \text{ LIKE } '%1960'}(\mathbf{MovieStar})))$$

↓ evaluation over tables

(Result)

SQL (DQL) query processor in action

```
SELECT DISTINCT title FROM StarsIn WHERE starName IN  
(SELECT name FROM MovieStar WHERE birthDate LIKE '%1960');
```

⌋ parsing

(parse tree)

⌋ compilation

$$\pi_{title}(\sigma_{starName=name}(\mathbf{StarsIn} \times \pi_{name}(\sigma_{birthDate \text{ LIKE } '%1960'}(\mathbf{MovieStar}))))$$

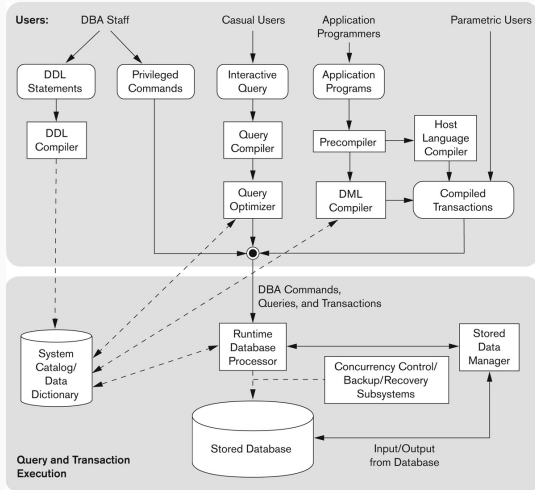
⌋ optimisation

$$\pi_{title}(\mathbf{StarsIn} \bowtie_{starName=name} \pi_{name}(\sigma_{birthDate \text{ LIKE } '%1960'}(\mathbf{MovieStar})))$$

⌋ evaluation over tables (next time)

(Result)

The place of parsing and optimisation



Materials to read about query compilation and optimisation

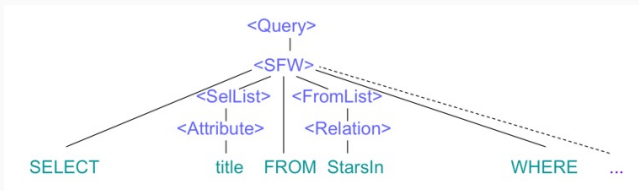
1. *Fundamentals of Database Systems* by R. Elmasri and S.B. Navathe (7th edition): Chapter 18 (beginning) and 19
(my lectures have different accents)
2. *Database Systems: the Complete Book* by H. Garcia-Molina, J. Ullman & J. Widom (2nd edition): Section 16
(closer to my presentation)
3. etc.

1. Parsing
2. Compilation
3. Optimisation: Overview
4. Optimisation: Cost Function
5. Optimisation: Heuristics

1. Parsing

Parsing: overview

Parsing transforms an SQL (**SELECT**) query to a **parse tree**—
that is, from **SELECT title FROM StarsIn WHERE ...** to

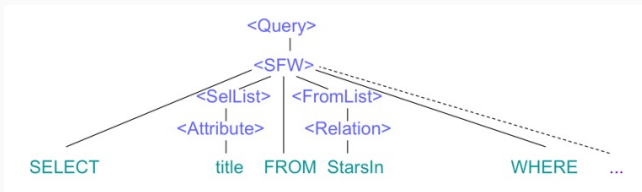


Parse tree:

- **inner nodes** are **syntactic categories** for parts of the query
- **leaves** are **primitive atoms** representing keywords, names, constants, parentheses, operators, etc.

Parsing: overview

Parsing transforms an SQL (**SELECT**) query to a **parse tree**—that is, from `SELECT title FROM StarsIn WHERE ...` to



Parse tree:

- **inner nodes** are **syntactic categories** for parts of the query
- **leaves** are **primitive atoms** representing keywords, names, constants, parentheses, operators, etc.

Parsing is done by standard techniques using a **(context-free) grammar** for SQL (see details in a **Compiler Book**)

Excerpt of SQL (DQL) grammar in BNF

`<Query> ::= <SFW> | "(" <Query> ")" | <Query> "UNION" <Query> | ...`

`<SFW> ::= "SELECT" (<SelList>|"*") "FROM" <FromList> ["WHERE" <Condition>]
["GROUP BY" ... ["HAVING" ...]] ["ORDER BY" ...]`

`<SelList> ::= <Attribute> | <Attribute> "," <SelList> | ...`

`<FromList> ::= <Relation> | <Relation> "," <FromList> | ...`

`<Condition> ::= <Condition> "AND" <Condition> | "NOT" <Cond> |
<Attribute> "IN" <Query> | <Attribute> "=" <Attribute> | ...`

...

`<Attribute>, <Relation> ::= ... (parse to strings of symbols)`

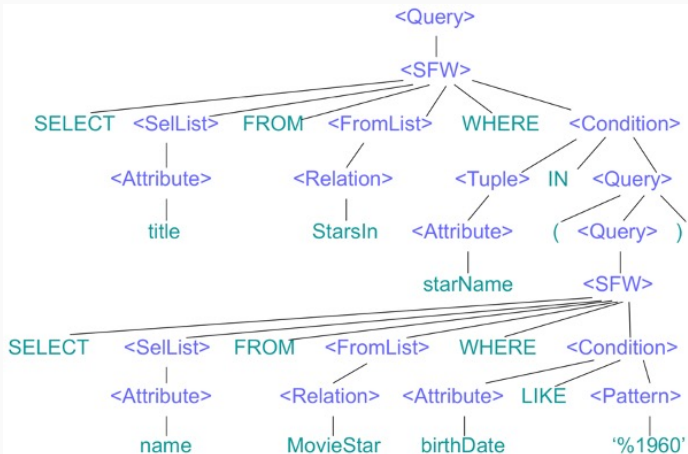
...

Parsing: Example

```
SELECT title FROM StarsIn WHERE starName IN
```

```
( SELECT name FROM MovieStar WHERE birthDate LIKE '%1960' );
```

parses to



Resolving names and type checking

Besides building the tree, several **checks** is done using the **System Catalog**:

- **resolving names**: checking that all the tables and attributes **exist** (and, if successful, linking them)
- **type checking**: all operations in expressions are **well-typed**

Resolving names and type checking

Besides building the tree, several **checks** is done using the **System Catalog**:

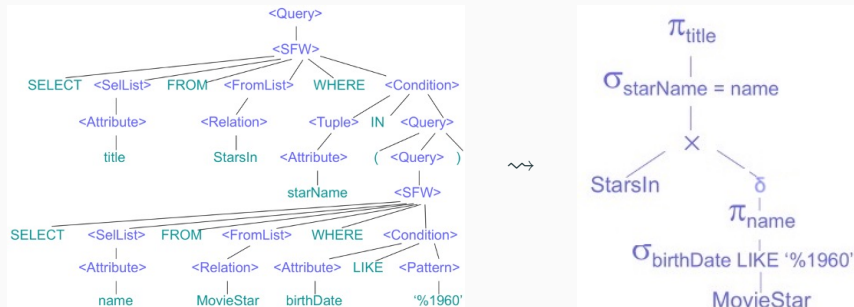
- **resolving names**: checking that all the tables and attributes **exist** (and, if successful, linking them)
- **type checking**: all operations in expressions are **well-typed**

Both building the tree and the checks can **fail**

2. Compilation

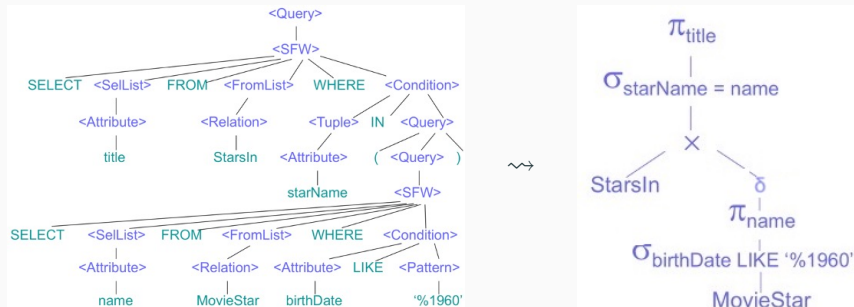
Query compilation: Idea

Converts (i.e., 'compiles') the parse tree to an **RA expression**, called initial (logical) query plan:



Query compilation: Idea

Converts (i.e., 'compiles') the parse tree to an **RA expression**, called initial (logical) query plan:



Note 1: I use **tree-like representation** for the RA expression (natural and intuitive)

Note 2: I use the **relation names** instead of full relations (for brevity)

Query compilation: Subquery-free case

Compilation without subqueries is direct

For **example**, the parse tree for

`SELECT L FROM Table1, ..., TableM WHERE C`

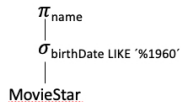
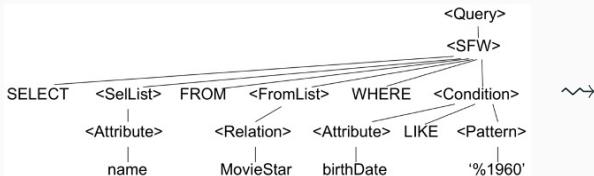
compiles as follows:

- **replace** `Table1, ..., TableM` with
 $(\dots (Table1 \times Table2) \times \dots) \times TableM$ in the tree
- apply selection $\sigma_C(\dots)$ to the product
- apply projection $\pi_L(\dots)$ to the selection

Note: I assumed here that `Table1, ..., TableM` do not have common attributes; if they do, some renaming is needed

Query compilation: Subquery-free example

```
SELECT name FROM MovieStar WHERE birthDate LIKE '%1960'
```



Compiling queries with subqueries

Recall: conditions in **WHERE** clause can have subqueries:

```
SELECT ... FROM ...  
    WHERE ... EXISTS(SELECT ... ) ... ;
```

```
SELECT ... FROM ...  
    WHERE ... (Att1, ... , AttN) op SOME(SELECT ... ) ... ;
```

```
SELECT ... FROM ...  
    WHERE ... (Att1, ... , AttN) op ALL(SELECT ... ) ... ;
```

here, **op** is one of **=**, **>=**, **>**, ...

SOME has synonym **ANY**

= SOME(SELECT ...) has synonym **IN(SELECT ...)**

Can be **correlated**: subquery can use attributes from the outside

Compiling queries with subqueries

Recall: conditions in **WHERE** clause can have subqueries:

```
SELECT ... FROM ...  
    WHERE ... EXISTS(SELECT ... ) ... ;
```

```
SELECT ... FROM ...  
    WHERE ... (Att1, ... , AttN) op SOME(SELECT ... ) ... ;
```

```
SELECT ... FROM ...  
    WHERE ... (Att1, ... , AttN) op ALL(SELECT ... ) ... ;
```

here, **op** is one of **=**, **>=**, **>**, ...

SOME has synonym **ANY**

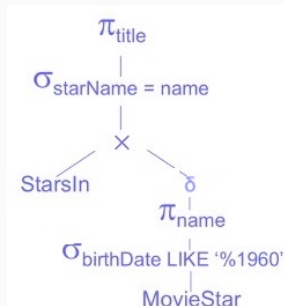
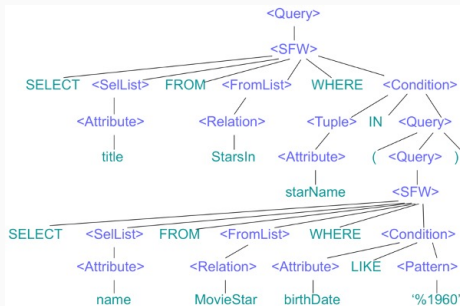
= SOME(SELECT ...) has synonym **IN(SELECT ...)**

Can be **correlated**: subquery can use attributes from the outside

Compiling subqueries is more **challenging**,
especially with **correlated attributes** (but always possible)

Non-correlated IN-subqueries

```
SELECT title FROM StarsIn WHERE starName IN  
( SELECT name FROM MovieStar WHERE birthDate LIKE '%1960' );
```

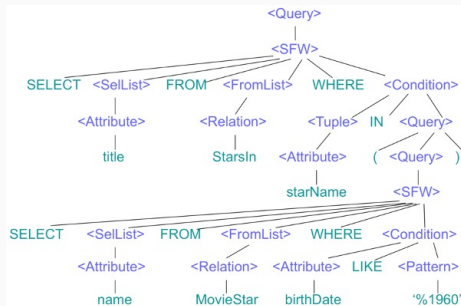


Alternatively:

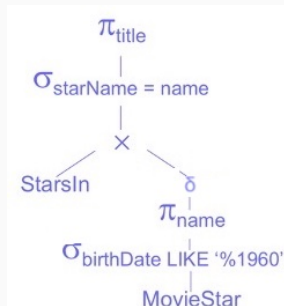
Non-correlated IN-subqueries

```
SELECT title FROM StarsIn WHERE starName IN
( SELECT name FROM MovieStar WHERE birthDate LIKE '%1960' );
```

↴



↗



Alternatively:

$$\pi_{title}(StarsIn \bowtie (\pi_{name \rightarrow starName}(\sigma_{birthDate LIKE '%1960'}(MovieStar))))$$

Other non-correlated subqueries

Subqueries with NOT IN are **symmetric**:

```
SELECT title FROM StarsIn WHERE starName NOT IN  
  (SELECT name FROM MovieStar WHERE birthDate LIKE '%1960');
```

↓

$$\pi_{title}(StarsIn \triangleright (\pi_{name \rightarrow starName}(\sigma_{birthDate LIKE '%1960'}(MovieStar))))$$

Other non-correlated subqueries

Subqueries with NOT IN are **symmetric**:

```
SELECT title FROM StarsIn WHERE starName NOT IN  
  (SELECT name FROM MovieStar WHERE birthDate LIKE '%1960');
```

↕

$$\pi_{title}(StarsIn \triangleright (\pi_{name \rightarrow starName}(\sigma_{birthDate LIKE '%1960'}(MovieStar))))$$

If a condition with a subquery is a **sub-condition** of a complex Boolean expression, it can be **isolated**:

```
SELECT title FROM StarsIn WHERE (starName NOT IN  
  (SELECT name FROM MovieStar WHERE birthDate LIKE '%1960')) AND Cond;
```

↕

$$\pi_{title}(\sigma_{Cond}(StarsIn) \triangleright (\pi_{name \rightarrow starName}(\sigma_{birthDate LIKE '%1960'}(MovieStar))))$$

Other non-correlated subqueries

Subqueries with NOT IN are **symmetric**:

```
SELECT title FROM StarsIn WHERE starName NOT IN  
  (SELECT name FROM MovieStar WHERE birthDate LIKE '%1960');
```

↕

$$\pi_{\text{title}}(\text{StarsIn} \triangleright (\pi_{\text{name} \rightarrow \text{starName}}(\sigma_{\text{birthDate LIKE '%1960'}}(\text{MovieStar}))))$$

If a condition with a subquery is a **sub-condition** of a complex Boolean expression, it can be **isolated**:

```
SELECT title FROM StarsIn WHERE (starName NOT IN  
  (SELECT name FROM MovieStar WHERE birthDate LIKE '%1960')) AND Cond;
```

↕

$$\pi_{\text{title}}(\sigma_{\text{Cond}}(\text{StarsIn}) \triangleright (\pi_{\text{name} \rightarrow \text{starName}}(\sigma_{\text{birthDate LIKE '%1960'}}(\text{MovieStar}))))$$

Other (non-correlated) sub-queries are **similar**

Correlated subqueries by copying

For **correlated subqueries**, we may create a **copy** of the outer table in the subquery:

```
SELECT Person.name FROM Person
    WHERE EXISTS(SELECT * FROM Address
                WHERE Person.name = Address.name);
```

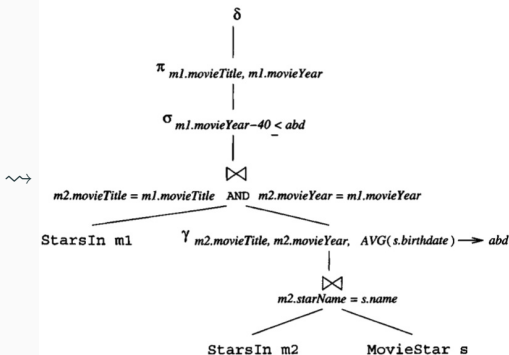
$\Downarrow$

$$\pi_{Person.name}(Person \bowtie (\sigma_{Person.name=Address.name}(Person \times Address)))$$

Correlated subqueries by postponing selection

Alternatively, we may postpone the selection:

```
SELECT DISTINCT m1.movieTitle, m1.movieYear
FROM StarsIn m1
WHERE m1.movieYear - 40 <= ANY (
    SELECT AVG(birthdate)
    FROM StarsIn m2, MovieStar s
    WHERE m2.starName = s.name AND
          m1.movieTitle = m2.movieTitle AND
          m1.movieYear = m2.movieYear
);
```



(Need to be careful with multiplicities with this technique)

3. Optimisation: Overview

Optimisation: Idealistic goal

Given a query plan (RA expression) $Expr$, our **idealistic** goal is to find **another** query plan $Expr'$ such that

Optimisation: Idealistic goal

Given a query plan (RA expression) $Expr$, our **idealistic** goal is to find **another** query plan $Expr'$ such that

- $Expr$ and $Expr'$ give the **same answer**
- $Expr'$ can be evaluated as **quick as possible**
- finding $Expr'$ and **ensuring** the first two conditions is also quick

Optimisation: Idealistic goal

Given a query plan (RA expression) $Expr$, our **idealistic** goal is to find **another** query plan $Expr'$ such that

- $Expr$ and $Expr'$ give the **same answer**
- $Expr'$ can be evaluated as **quick as possible**
- finding $Expr'$ and **ensuring** the first two conditions is also quick

There are few **serious** (in fact, usually **crucial**) obstacles

- search space for $Expr'$ is huge (actually, infinite)
- in general, the **only way to guarantee** that some $Expr'$ can be evaluated quicker than some other $Expr$ is to **evaluate both** and measure time

Optimisation: Pragmatic approach

Given a (logical) query plan (RA expression) *Expr*, DBMSs *pragmatically* find *another* query plan *Expr'* such that

- *Expr* and *Expr'* give the *same answer*
- *Expr'* is *estimated* to be *quicker* than other candidates
- finding candidate *Expr'* and *estimation* is quick

Optimisation: Pragmatic approach

Given a (logical) query plan (RA expression) $Expr$, DBMSs **pragmatically** find **another** query plan $Expr'$ such that

- $Expr$ and $Expr'$ give the **same answer**
- $Expr'$ is **estimated** to be **quicker** than other candidates
- finding candidate $Expr'$ and **estimation** is quick

This approach is based on **two components**:

1. good **cost function** f for query plans that is quick to compute and **reflects** evaluation time (i.e., $f(Expr') \leq f(Expr)$ in most cases means that $Expr'$ is quicker to evaluate than $Expr$)
2. good **heuristic** (usually based on **RA algebraic laws**) that allows to quickly construct (one or several) candidate $Expr'$ with (much) better cost

Optimisation: Example

SELECT B, C, Y FROM R, S WHERE W=X AND A=3 AND Z='a'

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

Candidate 1: Initial query plan

SELECT B, C, Y FROM R, S WHERE W=X AND A=3 AND Z='a'

compiles to initial plan

$\pi_{B,C,Y}(\sigma_{W=X \ \& \ A=3 \ \& \ Z='a'}(R \times S))$

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

$\times$
 $\rightsquigarrow$

A	B	C	...	W	X	Y	Z
1	z	1	...	4	1	a	a
...	...	...	...	...	2	f	c
2	c	6	...	2	1	a	a
...	...	...	...	...	2	f	c
3	r	8	...	7	1	a	a
...	...	...	...	...	2	f	c
3	r	8	...	7	7	k	a
4	n	9	...	4	1	a	a
...	...	...	...	...	2	f	c
...	...	...	...	...	...	...	v
2	i	0	...	3	1	a	a
...	...	...	...	...	2	c	c
3	t	5	...	9	1	a	a
...	...	...	...	...	2	f	c
7	e	3	...	3	1	a	a
...	...	...	...	...	2	f	c
...	...	...	...	...	...	...	v

σ, π
 $\rightsquigarrow$

B	C	Y
r	8	k

Candidate 1: Initial query plan

SELECT B, C, Y FROM R, S WHERE W=X AND A=3 AND Z='a'

compiles to initial plan

$\pi_{B,C,Y}(\sigma_{W=X \ \& \ A=3 \ \& \ Z='a'}(R \times S))$

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

$\times$
 $\rightsquigarrow$

A	B	C	...	W	X	Y	Z
1	z	1	...	4	1	a	a
...	...	...	...	...	2	f	c
2	c	6	...	2	1	a	a
...	...	...	...	...	2	f	c
3	r	8	...	7	1	a	a
...	...	...	...	...	2	f	c
3	r	8	...	7	7	k	a
4	n	9	...	4	1	a	a
...	...	...	...	...	2	f	c
...	...	...	...	...	...	...	v
2	i	0	...	3	1	a	a
...	...	...	...	...	2	c	c
3	t	5	...	9	1	a	a
...	...	...	...	...	2	f	c
7	e	3	...	3	1	a	a
...	...	...	...	...	2	f	c
...	...	...	...	...	...	...	v

σ, π
 $\rightsquigarrow$

B	C	Y
r	8	k

Intuitively, cannot be the most efficient

Candidate 2: Pushing selection down

Initial plan $\pi_{B,C,Y} (\sigma_{W=X \ \& \ A=3 \ \& \ Z='a'}(R \times S))$
 transforms, by **pushing selection** and **theta-join definition** laws, to
 another plan $\pi_{B,C,Y} ((\sigma_{A=3}(R) \bowtie_{W=X} \sigma_{Z='a'}(S)))$

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

σ, σ
 $\rightsquigarrow$

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

$\bowtie_{W=X}$
 $\rightsquigarrow$

A	B	C	...	W	X	Y	Z
3	r	8	...	7	7	k	a

π
 $\rightsquigarrow$

B	C	Y
r	8	k

Candidate 2: Pushing selection down

Initial plan $\pi_{B,C,Y} (\sigma_{W=X \ \& \ A=3 \ \& \ Z='a'}(R \times S))$
transforms, by **pushing selection** and **theta-join definition** laws, to
another plan $\pi_{B,C,Y} ((\sigma_{A=3}(R) \bowtie_{W=X} \sigma_{Z='a'}(S)))$

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

σ, σ
 $\rightsquigarrow$

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

$\bowtie_{W=X}$
 $\rightsquigarrow$

A	B	C	...	W	X	Y	Z
3	r	8	...	7	7	k	a

π
 $\rightsquigarrow$

B	C	Y
r	8	k

Looks **better**

Candidate 2: Pushing selection down

Initial plan $\pi_{B,C,Y} (\sigma_{W=X \ \& \ A=3 \ \& \ Z='a'}(R \times S))$
transforms, by **pushing selection** and **theta-join definition** laws, to
another plan $\pi_{B,C,Y} ((\sigma_{A=3}(R) \bowtie_{W=X} \sigma_{Z='a'}(S)))$

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

σ, σ
 $\rightsquigarrow$

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

$\bowtie_{W=X}$
 $\rightsquigarrow$

A	B	C	...	W	X	Y	Z
3	r	8	...	7	7	k	a

π
 $\rightsquigarrow$

B	C	Y
r	8	k

Looks **better**, especially, if we use **indexes** on A in R and Z in S

Candidate 3: More fine-grained

Candidate 1 (initial): $\pi_{B,C,Y} (\sigma_{W=X} \ \& \ A=3 \ \& \ Z='a' (R \times S))$

Candidate 2: $\pi_{B,C,Y} ((\sigma_{A=3}(R) \bowtie_{W=X} \sigma_{Z='a'}(S)))$

Candidate 3: $\pi_{B,C,Y} ((\sigma_{A=3}(R) \bowtie_{W=X} \sigma_{Z='a'}(S \ltimes_{W=X} \sigma_{A=3}(R))))$

(here, $\ltimes_{W=X}$ is 'equi-semi-join', naturally expressible via ρ and $\ltimes$)

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

$\sigma, \ltimes$
 $\rightsquigarrow$

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

$\sigma, \bowtie_{W=X}$
 $\rightsquigarrow$

A	B	C	...	W	X	Y	Z
3	r	8	...	7	7	k	a

π
 $\rightsquigarrow$

B	C	Y
r	8	k

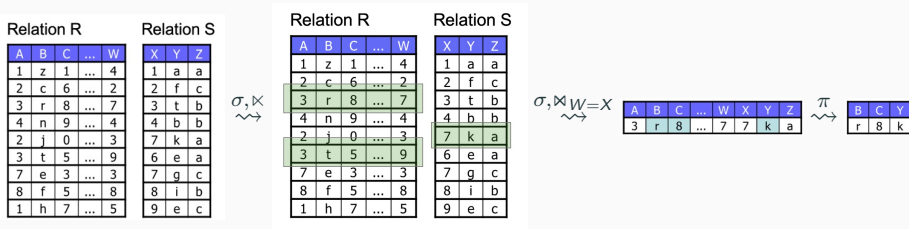
Candidate 3: More fine-grained

Candidate 1 (initial): $\pi_{B,C,Y} (\sigma_{W=X \ \& \ A=3 \ \& \ Z='a'}(R \times S))$

Candidate 2: $\pi_{B,C,Y} ((\sigma_{A=3}(R) \bowtie_{W=X} \sigma_{Z='a'}(S)))$

Candidate 3: $\pi_{B,C,Y} ((\sigma_{A=3}(R) \bowtie_{W=X} \sigma_{Z='a'}(S \ltimes_{W=X} \sigma_{A=3}(R))))$

(here, $\ltimes_{W=X}$ is 'equi-semi-join', naturally expressible via ρ and $\ltimes$)



More operations, but **quicker**

Candidate 3: More fine-grained

Candidate 1 (initial): $\pi_{B,C,Y} (\sigma_{W=X \ \& \ A=3 \ \& \ Z='a'}(R \times S))$

Candidate 2: $\pi_{B,C,Y} ((\sigma_{A=3}(R) \bowtie_{W=X} \sigma_{Z='a'}(S)))$

Candidate 3: $\pi_{B,C,Y} ((\sigma_{A=3}(R) \bowtie_{W=X} \sigma_{Z='a'}(S \ltimes_{W=X} \sigma_{A=3}(R))))$

(here, $\ltimes_{W=X}$ is 'equi-semi-join', naturally expressible via ρ and $\ltimes$)

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

$\sigma, \ltimes$
 $\rightsquigarrow$

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

$\sigma, \bowtie_{W=X}$
 $\rightsquigarrow$

A	B	C	...	W	X	Y	Z
3	r	8	...	7	7	k	a

π
 $\rightsquigarrow$

B	C	Y
r	8	k

More operations, but **quicker**,
especially, if we use **indexes** on A in R and X (not Z !) in S

4. Optimisation: Cost function

Cost function requirements

Recall: We need cost function f such that

$f(\text{Expr}') \leq f(\text{Expr})$ in most cases means that
plan Expr' is quicker to evaluate than plan Expr

Evaluation time is spent for

- reading and writing to the secondary memory (e.g., disk I/O)
- evaluating RA operations in the main memory

Cost function requirements

Recall: We need **cost function** f such that

$f(\text{Expr}') \leq f(\text{Expr})$ in most cases means that
plan Expr' is **quicker** to evaluate than plan Expr

Evaluation time is spent for

- **reading** and **writing** to the secondary memory (e.g., disk I/O)
- **evaluating** RA operations in the main memory

Real DBMS (advanced) cost functions **take into account**:

- (potentially different) **evaluation time** for operations (using **algorithms** discussed in the next lecture)
- **indexes** that decrease the number of reads (**we know**)
- **reusing** intermediate results (e.g., as in Candidate 3 above)
- **pipelining** intermediate results (tuples) from one operation to another without writing back to the disk

Our cost function

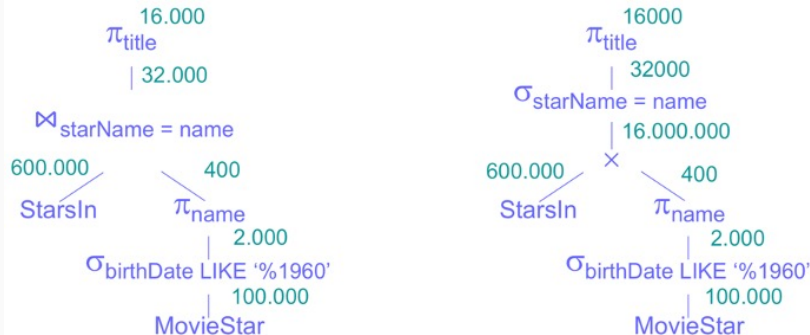
We take as our cost function:

an **estimate** of the **sum** of **sizes** (in bytes) of the **intermediate** results of the plan (i.e., the RA expression)

- **estimate**, because exact may be difficult to compute (our **estimate** is computed as described below)
- **intermediate**, because the inputs and the outputs are the same
- **reflects** the most important thing above: the number of disk I/O (number of used blocks is roughly proportional to the size of a relation)
- **compositional**: knowing estimates for immediate sub-plans, we can quickly compute the estimate for the plan (e.g., if we know the size estimates of **Expr** and **Expr'**, we can estimate the size of **Expr** $\bowtie$ **Expr'**)
- **quick** to compute

Essentially, we assume that each operation is executed in negligible time and in isolation, and all the time is spent in reading all arguments (in whole) of each operation from the disk and writing the result back

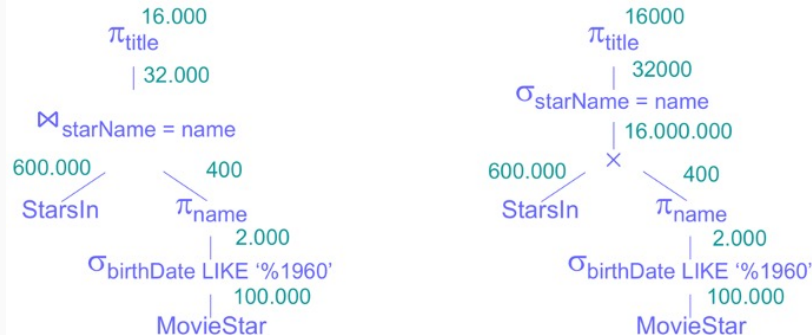
Before we go: Example



Our cost of the left plan: $2,000 + 400 + 32,000 = 34,400$

Our cost of the right plan: $2,000 + 400 + 16,000,000 + 32,000 = 16,034,400$

Before we go: Example



Our cost of the left plan: $2,000 + 400 + 32,000 = 34,400$

Our cost of the right plan: $2,000 + 400 + 16,000,000 + 32,000 = 16,034,400$

Reflects our intuition: **good cost function** (at least for this case)

Thus, our **main** metric:

$Size(R)$ – (estimation of) the size of relation R in bytes
(i.e., our **cost**)

Notation

Thus, our **main** metric:

$Size(R)$ – (estimation of) the size of relation R in bytes
(i.e., our **cost**)

To estimate $Size(R)$, we will also maintain **secondary** metrics:

$Tups(R)$ – (estimation of) the **number of tuples** in relation R

$TupSize(R)$ – the (average) **size of a tuple** in R

as a result, we have $Size(R) = Tups(R) * TupSize(R)$

$Vals(R.A)$ – (estimation of) the number of **different values** of attribute A in R

real DBMSs use **histograms** instead (we will discuss them later)

Cost for the base case: Statistics

To compute costs for complex plans by induction, we need to start with **statistics**—that is, $Size(R)$, $Tups(R)$, $TupSize(R)$, $Vals(R.A)$, etc., for the base database relations R

Computing statistics may be **expensive**, and doing it after each update may be **prohibitively costly**

However, we do **not need the exact** values every time we estimate the cost of a plan, some **inaccuracies** is not a problem

System can recompute statistics only **time to time**
(automatically or manually)

Cost of bag projection

Assuming **uniform** size of tuples (i.e., no VARCHAR, etc.), the size and almost all secondary metrics for **bag (generalised) projection** can be calculated accurately, (due to **bag semantics**, one result tuple for each argument tuple):

- $Tups(\pi_L^b(R)) = Tups(R)$
- $TupSize(\pi_L^b(R))$ is defined by L
- $Size(\pi_L^b(R)) = Tups(\pi_L^b(R)) * TupSize(\pi_L^b(R))$

Cost of bag projection

Assuming **uniform** size of tuples (i.e., no VARCHAR, etc.), the size and almost all secondary metrics for **bag (generalised) projection** can be calculated accurately, (due to **bag semantics**, one result tuple for each argument tuple):

- $Tups(\pi_L^b(R)) = Tups(R)$
- $TupSize(\pi_L^b(R))$ is defined by L
- $Size(\pi_L^b(R)) = Tups(\pi_L^b(R)) * TupSize(\pi_L^b(R))$
- $Vals((\pi_L^b(R)).A) = Vals(R.A)$ if A is an attribute in R
- $Vals((\pi_{Expr \rightarrow A}^b(R)).A)$ depends on $Expr$;
by default, can be estimated as
 $\max(Vals(R.A_1), \dots, Vals(R.A_n))$ where $A_1, \dots, A_n$ are all different attributes of R mentioned in $Expr$

Cost of bag projection

Assuming **uniform** size of tuples (i.e., no VARCHAR, etc.), the size and almost all secondary metrics for **bag (generalised) projection** can be calculated accurately, (due to **bag semantics**, one result tuple for each argument tuple):

- $Tups(\pi_L^b(R)) = Tups(R)$
- $TupSize(\pi_L^b(R))$ is defined by L
- $Size(\pi_L^b(R)) = Tups(\pi_L^b(R)) * TupSize(\pi_L^b(R))$
- $Vals((\pi_L^b(R)).A) = Vals(R.A)$ if A is an attribute in R
- $Vals((\pi_{Expr \rightarrow A}^b(R)).A)$ depends on $Expr$;
by default, can be estimated as
 $\max(Vals(R.A_1), \dots, Vals(R.A_n))$ where $A_1, \dots, A_n$ are all different attributes of R mentioned in $Expr$

(We will cover **set projection** with duplicate elimination)

Cost of bag projection: Example

Relation R :

A	B	C	D
1	cat	1999	a
2	cat	2002	b
3	dog	2002	c
4	cat	1998	a
5	dog	2000	c

Statistics of R :

- $Tups(R) = 5$
- $TupSize(R) = 4 + 20 + 4 + 10 = 38$
(INT(4) + CHAR(20) + INT(4) + CHAR(10))
- $Size(R) = 5 * 38 = 190$
- $Vals(R.A) = 5$, $Vals(R.B) = 2$,
 $Vals(R.C) = 4$, $Vals(R.D) = 3$

Cost of bag projection: Example

Relation R :

A	B	C	D
1	cat	1999	a
2	cat	2002	b
3	dog	2002	c
4	cat	1998	a
5	dog	2000	c

Statistics of R :

- $Tups(R) = 5$
- $TupSize(R) = 4 + 20 + 4 + 10 = 38$
(INT(4) + CHAR(20) + INT(4) + CHAR(10))
- $Size(R) = 5 * 38 = 190$
- $Vals(R.A) = 5$, $Vals(R.B) = 2$,
 $Vals(R.C) = 4$, $Vals(R.D) = 3$

$S = \pi_{A+10 \rightarrow Aplus, B, A+C \rightarrow AC}^b(R)$:

- $Tups(S) = 5$
- $TupSize(S) = 4 + 20 + 4 = 28$
- $Size(S) = 5 * 28 = 140$
- $Vals(S.Aplus) = 5$, $Vals(S.B) = 2$ (both exact),
 $Vals(S.AC) = 5$ (estimation, 4 in reality)

Cost of selection: Overview

The size of **selection** $\sigma_{Cond}(R)$ is much more difficult to estimate:

- $TupSize(\sigma_{Cond}(R)) = TupSize(R)$ (easy)
- so, $Size(\sigma_{Cond}(R)) = Tups(\sigma_{Cond}(R)) * TupSize(\sigma_{Cond}(R))$
relies on estimation of $Tups(\sigma_{Cond}(R))$
- we consider different types of **Cond** separately
- by default, we **naively** assume **uniform** and **independent**
distribution of $Vals(R.A)$ for relevant $R.A$
- real DMBSs use **histograms** for much better estimations
- estimation of all $Vals((\sigma_{Cond}(R)).A)$ with A mentioned in
Cond can be done **similarly** (try it by **yourself**);
for other attributes A we can take
$$Vals((\sigma_{Cond}(R)).A) = Vals(R.A) * Tups(\sigma_{Cond}(R)) / Tups(R)$$

Cost of selection: Atomic comparisons

Estimating $Tups(\sigma_{Cond}(R))$ for **atomic Cond**:

- equality to a constant:

$$Tups(\sigma_{A=c}(R)) = Tups(R) / Vals(R.A) \text{ (may be not integer!)}$$

- inequality to a constant:

$$Tups(\sigma_{A \neq c}(R)) = Tups(R) * (1 - 1 / Vals(R.A)) \text{ (or just } Tups(R))$$

- comparison to a constant:

$$Tups(\sigma_{A < c}(R)) =$$

Cost of selection: Atomic comparisons

Estimating $Tups(\sigma_{Cond}(R))$ for **atomic Cond**:

- equality to a constant:

$$Tups(\sigma_{A=c}(R)) = Tups(R) / Vals(R.A) \text{ (may be not integer!)}$$

- inequality to a constant:

$$Tups(\sigma_{A \neq c}(R)) = Tups(R) * (1 - 1 / Vals(R.A)) \text{ (or just } Tups(R))$$

- comparison to a constant:

$$Tups(\sigma_{A < c}(R)) = Tups(R) / 3 \text{ (or just } Tups(R) / 2)$$

- equality of attributes:

$$Tups(\sigma_{A=B}(R)) = (Tups(R) / \max(Vals(R.A), Vals(R.B)))$$

- inequality of attributes:

$$Tups(\sigma_{A \neq B}(R)) = Tups(R) * (1 - 1 / \max(Vals(R.A), Vals(R.B))) \text{ (or ...)}$$

- comparison of attributes:

$$Tups(\sigma_{A < B}(R)) = Tups(R) / 3 \text{ (or just } Tups(R) / 2)$$

Cost of selection: Atomic comparisons

Estimating $Tups(\sigma_{Cond}(R))$ for **atomic** *Cond*:

- equality to a constant:

$$Tups(\sigma_{A=c}(R)) = Tups(R) / Vals(R.A) \text{ (may be not integer!)}$$

- inequality to a constant:

$$Tups(\sigma_{A \neq c}(R)) = Tups(R) * (1 - 1 / Vals(R.A)) \text{ (or just } Tups(R))$$

- comparison to a constant:

$$Tups(\sigma_{A < c}(R)) = Tups(R) / 3 \text{ (or just } Tups(R) / 2)$$

- equality of attributes:

$$Tups(\sigma_{A=B}(R)) = (Tups(R) / \max(Vals(R.A), Vals(R.B)))$$

- inequality of attributes:

$$Tups(\sigma_{A \neq B}(R)) = Tups(R) * (1 - 1 / \max(Vals(R.A), Vals(R.B))) \text{ (or ...)}$$

- comparison of attributes:

$$Tups(\sigma_{A < B}(R)) = Tups(R) / 3 \text{ (or just } Tups(R) / 2)$$

- comparisons with functions:

$$Tups(\sigma_{f(A_1, \dots, A_n) \text{ op } g(B_1, \dots, B_n)}(R)) = \dots$$

Cost of atomic selection: Examples

Relation R :

A	B	C	D
1	cat	1999	a
2	cat	2002	b
3	dog	2002	c
4	cat	1998	a
5	dog	2000	c

Statistics of R :

- $Tups(R) = 5$
- $TupSize(R) = \dots$
- $Size(R) = \dots$
- $Vals(R.A) = 5$, $Vals(R.B) = 2$,
 $Vals(R.C) = 4$, $Vals(R.D) = 3$

Estimates of the number of tuples:

- $Tups(\sigma_{A=4}(R)) = 5/5 = 1$ (real: 1),
 $Tups(\sigma_{A \neq 4}(R)) = 5 * (1 - 1/5) = 4$ (4),
 $Tups(\sigma_{A < 4}(R)) = 5/3 = 1.66$ (3)
- $Tups(\sigma_{B='cat'}(R)) = 5/2 = 2.5$ (3),
 $Tups(\sigma_{B \neq 'cat'}(R)) = 5 * (1 - 1/2) = 2.5$ (2)
- $Tups(\sigma_{A=C}(R)) = 5/5 = 1$ (0),
 $Tups(\sigma_{A \neq C}(R)) = 5 * (1 - 1/5) = 4$ (5),
 $Tups(\sigma_{A < C}(R)) = 5/3 = 1.66$ (5)

Cost of selection: Boolean combinations

Estimating $Tups(\sigma_{Cond}(R))$ for Boolean combinations of conditions:

- **AND**: $Tups(\sigma_{Cond_1 \& Cond_2}(R)) = Tups(\sigma_{Cond_1}(\sigma_{Cond_2}(R))) = Tups(\sigma_{Cond_2}(\sigma_{Cond_1}(R))) = Tups(R) * factor_{Cond_1} * factor_{Cond_2}$
(assumes independence)
- **NOT**: $Tups(\sigma_{\neg Cond'}(R)) = Tups(R) - Tups(\sigma_{Cond'}(R))$
- **OR**: $Tups(\sigma_{Cond_1 \vee Cond_2}(R)) = Tups(\sigma_{\neg(\neg Cond_1 \& \neg Cond_2)}(R)) = Tups(R) * (1 - (1 - factor_{Cond_1}) * (1 - factor_{Cond_2}))$
(assumes independence)

Cost of selections with Boolean combinations: Examples

Relation R :

A	B	C	D
1	cat	1999	a
2	cat	2002	b
3	dog	2002	c
4	cat	1998	a
5	dog	2000	c

Statistics of R :

- $Tups(R) = 5$
- $TupSize(R) = \dots$
- $Size(R) = \dots$
- $Vals(R.A) = 5$, $Vals(R.B) = 2$,
 $Vals(R.C) = 4$, $Vals(R.D) = 3$

Estimates of the number of tuples:

- $Tups(\sigma_{A=4 \ \& \ B='cat'}(R)) = 5 * 1/5 * 1/2 = 0.5$ (real: 1),
 $Tups(\sigma_{A \neq 4 \ \& \ B \neq 'cat'}(R)) = 5 * 4/5 * 1/2 = 2$ (2)
- $Tups(\sigma_{A=4 \vee B='cat'}(R)) = 5 * (1 - (1 - 1/5) * (1 - 1/2)) = 3$ (3),
 $Tups(\sigma_{A \neq 4 \vee B \neq 'cat'}(R)) = 5 * (1 - (1 - 4/5) * (1 - 1/2)) = 4.5$ (4)

Cost of natural join: Overview

The size of **natural join** $R \bowtie S$ (and other joins) is also difficult:

- $TupSize(R \bowtie S)$ is the sum of sizes of distinct attributes in R and S
- so, $Size(R \bowtie S) = Tups(R \bowtie S) * TupSize(R \bowtie S)$ relies on estimation of $Tups(R \bowtie S)$
- to estimate $Tups(R \bowtie S)$ and $Vals((R \bowtie S).A)$ for join attributes A we rely on **value containment** assumption:
If A is an attribute in R and S such that $Vals(R.A) \leq Vals(S.A)$ then every value of A in R appears in A of S (cf. **foreign key**)

Cost of natural join: Overview

The size of **natural join** $R \bowtie S$ (and other joins) is also difficult:

- $TupSize(R \bowtie S)$ is the sum of sizes of distinct attributes in R and S
- so, $Size(R \bowtie S) = Tups(R \bowtie S) * TupSize(R \bowtie S)$ relies on estimation of $Tups(R \bowtie S)$
- to estimate $Tups(R \bowtie S)$ and $Vals((R \bowtie S).A)$ for join attributes A we rely on **value containment** assumption:
If A is an attribute in R and S such that $Vals(R.A) \leq Vals(S.A)$ then **every value** of A in R **appears** in A of S (cf. **foreign key**)
- to estimate $Vals((R \bowtie S).A)$ for non-join attributes A we rely on **value preservation** assumption:
If A is in R but not in S then $Vals((R \bowtie S).A) = Vals(R.A)$ (and symmetric for A in S but not in R)

Cost of natural join: Overview

The size of **natural join** $R \bowtie S$ (and other joins) is also difficult:

- $TupSize(R \bowtie S)$ is the sum of sizes of distinct attributes in R and S
- so, $Size(R \bowtie S) = Tups(R \bowtie S) * TupSize(R \bowtie S)$ relies on estimation of $Tups(R \bowtie S)$
- to estimate $Tups(R \bowtie S)$ and $Vals((R \bowtie S).A)$ for join attributes A we rely on **value containment** assumption:
If A is an attribute in R and S such that $Vals(R.A) \leq Vals(S.A)$ then **every value** of A in R **appears** in A of S (cf. **foreign key**)
- to estimate $Vals((R \bowtie S).A)$ for non-join attributes A we rely on **value preservation** assumption:
If A is in R but not in S then $Vals((R \bowtie S).A) = Vals(R.A)$ (and symmetric for A in S but not in R)
- real DMBSs use **histograms** for much better estimations
- **other joins** are expressible, and we can use **the defining laws** for estimation

Cost of natural join: One join attribute case

Estimating $R \bowtie S$ with **one** join (i.e., common) attribute A
(using these two assumptions):

- if $Vals(R.A) \leq Vals(S.A)$ then each tuple in R will match approximately $Tups(S) / Vals(S.A)$ tuples in S
- thus, in general, $Tups(R \bowtie S) = Tups(R) * Tups(S) / \max(Vals(R.A), Vals(S.A))$
- $Vals((R \bowtie S).A) = \min(Vals(R.A), Vals(S.A))$
- $Vals((R \bowtie S).A') = Vals(R.A')$ for each non-join A' in R
- $Vals((R \bowtie S).A') = Vals(S.A')$ for each non-join A' in S

Cost of natural join: Example 1

Relations R and S :

Relation R

A	B	C	...	X
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

Statistics of R and S :

- $Tups(R) = 9$, $Tups(S) = 9$
- $TupSize(R) = \dots$, $TupSize(S) = \dots$
- $Size(R) = \dots$, $Size(S) = \dots$
- $Vals(R.A) = 6$, $Vals(R.X) = 7$,
 $Vals(S.X) = 9$, $Vals(S.Z) = 3$

Estimates (all correct, since essentially primary & foreign keys):

- $Tups(R \bowtie S) = Tups(R) * Tups(S) / \max(Vals(R.X), Vals(S.X)) = 9$
(real: 9)
- $Vals((R \bowtie S).X) = \min(Vals(R.X), Vals(S.X)) = 7$ (7)
- $Vals((R \bowtie S).A) = Vals(R.A) = 6$ (6)
- $Vals((R \bowtie S).Z) = Vals(S.Z) = 3$ (3) (these may be different even with the keys)

Cost of natural join: Example 2

Relations R and S :

Relation R

A	B	C	...	X
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	0

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

Statistics of R and S :

- $Tups(R) = 9$, $Tups(S) = 9$
- $TupSize(R) = \dots$, $TupSize(S) = \dots$
- $Size(R) = \dots$, $Size(S) = \dots$
- $Vals(R.A) = 6$, $Vals(R.X) = 7$,
 $Vals(S.X) = 9$, $Vals(S.Z) = 3$

Estimates (no foreign key any more):

- $Tups(R \bowtie S) = Tups(R) * Tups(S) / \max(Vals(R.X), Vals(S.X)) = 9$
(real: 8)
- $Vals((R \bowtie S).X) = \min(Vals(R.X), Vals(S.X)) = 7$ (6)
- $Vals((R \bowtie S).A) = Vals(R.A) = 6$ (6)
- $Vals((R \bowtie S).Z) = Vals(S.Z) = 3$ (3)

Cost of natural join: Many join attribute case

Estimating $R \bowtie S$ with **several** join attributes $A_1, \dots, A_n$
(using these two assumptions):

- applying the same logic, we have $Tups(R \bowtie S) =$

$$\frac{Tups(R) * Tups(S)}{\max(Vals(R.A_1), Vals(S.A_1)) * \dots * \max(Vals(R.A_n), Vals(S.A_n))}$$

Cost of natural join: Many join attribute case

Estimating $R \bowtie S$ with **several** join attributes $A_1, \dots, A_n$
(using these two assumptions):

- applying the same logic, we have $Tups(R \bowtie S) =$

$$\frac{Tups(R) * Tups(S)}{\max(Vals(R.A_1), Vals(S.A_1)) * \dots * \max(Vals(R.A_n), Vals(S.A_n))}$$

- for $n = 0$ we have **Cartesian product**:

$$Tups(R \bowtie S) = Tups(R) * Tups(S)$$

Cost of natural join: Many join attribute case

Estimating $R \bowtie S$ with **several** join attributes $A_1, \dots, A_n$
(using these two assumptions):

- applying the same logic, we have $Tups(R \bowtie S) =$

$$\frac{Tups(R) * Tups(S)}{\max(Vals(R.A_1), Vals(S.A_1)) * \dots * \max(Vals(R.A_n), Vals(S.A_n))}$$

- for $n = 0$ we have **Cartesian product**:

$$Tups(R \bowtie S) = Tups(R) * Tups(S)$$

- $Vals((R \bowtie S).A)$ is as before:

$\min(Vals(R.A_i), Vals(S.A_i))$ for each join A' ,

$Vals((R \bowtie S).A') = Vals(R.A')$ for non-join A' (in R)

Cost of bag-theoretic operations

Estimating metrics for bag union, intersection and difference:

- $TupSize(R \cup^b S)$, $TupSize(R \cap S)$, $TupSize(R \setminus S)$ are **inherited** from the arguments
- $Tups(R \cup^b S) = Tups(R) + Tups(S)$ (always),
 $Vals((R \cup^b S).A) = \max(Vals(R.A), Vals(S.A)) + \min(Vals(R.A), Vals(S.A))/2$ (a possibility for estimation)
- $Tups(R \cap S) = \min(Tups(R), Tups(S))/2$,
 $Vals((R \cap S).A) = \min(Vals(R.A), Vals(S.A))/2$
(possible estimates)
- $Tups(R \setminus S) = Tups(R) - Tups(S)/2$,
 $Vals((R \setminus S).A) = Vals(R.A) - Vals(S.A)/2$
(possible estimates)

Cost of bag-theoretic operations

Estimating metrics for bag union, intersection and difference:

- $TupSize(R \cup^b S)$, $TupSize(R \cap S)$, $TupSize(R \setminus S)$ are **inherited** from the arguments
- $Tups(R \cup^b S) = Tups(R) + Tups(S)$ (always),
 $Vals((R \cup^b S).A) = \max(Vals(R.A), Vals(S.A)) + \min(Vals(R.A), Vals(S.A))/2$ (a possibility for estimation)
- $Tups(R \cap S) = \min(Tups(R), Tups(S))/2$,
 $Vals((R \cap S).A) = \min(Vals(R.A), Vals(S.A))/2$
(possible estimates)
- $Tups(R \setminus S) = Tups(R) - Tups(S)/2$,
 $Vals((R \setminus S).A) = Vals(R.A) - Vals(S.A)/2$
(possible estimates)
- set-theoretic union can be estimated via **duplicate elimination** (see below)

Cost of duplicate elimination and grouping

Estimating metrics for **duplicate elimination**:

- $TupSize(\delta(R))$ and all $Vals((\delta(R)).A)$ are **inherited** from R
- $Tups(\delta(R))$ is somewhere in between 1 and $Tups(R)$, so we can estimate as $Tups(R)/2$

Cost of duplicate elimination and grouping

Estimating metrics for **duplicate elimination**:

- $TupSize(\delta(R))$ and all $Vals((\delta(R)).A)$ are **inherited** from R
- $Tups(\delta(R))$ is somewhere in between 1 and $Tups(R)$, so we can estimate as $Tups(R)/2$
- however, $Tups(\delta(R))$ is also bounded by $Vals(R.A_1) * \dots * Vals(R.A_n)$ for $A_1, \dots, A_n$ all attributes of R
- so, we can estimate
$$Tups(\delta(R)) = \min(Tups(R)/2, Vals(R.A_1) * \dots * Vals(R.A_n))$$

Cost of duplicate elimination and grouping

Estimating metrics for **duplicate elimination**:

- $TupSize(\delta(R))$ and all $Vals((\delta(R)).A)$ are **inherited** from R
- $Tups(\delta(R))$ is somewhere in between 1 and $Tups(R)$, so we can estimate as $Tups(R)/2$
- however, $Tups(\delta(R))$ is also bounded by $Vals(R.A_1) * \dots * Vals(R.A_n)$ for $A_1, \dots, A_n$ all attributes of R
- so, we can estimate
$$Tups(\delta(R)) = \min(Tups(R)/2, Vals(R.A_1) * \dots * Vals(R.A_n))$$

Estimating metrics for **grouping with aggregation**:

- similar to duplicate elimination (write it down by yourself)
- the only new case is $Vals((\gamma_{Agg(A) \rightarrow A'}(R)).A')$:
we can take $Tups(\gamma_{Agg(A) \rightarrow A'}(R))/2$ (i.e., every aggregate value appears **twice** on average)

Histograms

We maintained the number $Vals(R.A)$ of distinct values of each attribute A of each R to estimate the size of selection and joins

Assumed **uniform** distribution of these values across the tuples in R

Histograms

We maintained the number $Vals(R.A)$ of distinct values of each attribute A of each R to estimate the size of selection and joins

Assumed **uniform** distribution of these values across the tuples in R

Real DBMSs uses more fine-grained **histograms**:

- **simple histogram**: number of tuples in R for each value of A
- **advanced** (when $Vals(R.A)$ is large): number of tuples in R for intervals of values of A (many techniques to pick the intervals)

Histograms

We maintained the number $Vals(R.A)$ of distinct values of each attribute A of each R to estimate the size of selection and joins

Assumed **uniform** distribution of these values across the tuples in R

Real DBMSs uses more fine-grained **histograms**:

- **simple histogram**: number of tuples in R for each value of A
- **advanced** (when $Vals(R.A)$ is large): number of tuples in R for intervals of values of A (many techniques to pick the intervals)

Histograms allow us to compute the metrics (in particular, $Size(R)$) more accurately

However, they are difficult to propagate estimations of histograms for complex RA expressions (i.e., not in the the statistics of tables)

Histograms

We maintained the number $Vals(R.A)$ of distinct values of each attribute A of each R to estimate the size of selection and joins

Assumed **uniform** distribution of these values across the tuples in R

Real DBMSs uses more fine-grained **histograms**:

- **simple histogram**: number of tuples in R for each value of A
- **advanced** (when $Vals(R.A)$ is large): number of tuples in R for intervals of values of A (many techniques to pick the intervals)

Histograms allow us to compute the metrics (in particular, $Size(R)$) more accurately

However, they are difficult to propagate estimations of histograms for complex RA expressions (i.e., not in the the statistics of tables)

Often, a **hybrid** approach is used: histograms in the statistics, $Vals(R.A)$ with uniform distribution assumption for sub-plans

Plan costs: Come back to the example

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

Plan 1 (initial): $\pi_{B,C,Y} (\sigma_{W=X \ \& \ A=3 \ \& \ Z='a'}(R \times S))$

Plan 2: $\pi_{B,C,Y} ((\sigma_{A=3}(R) \bowtie_{W=X} \sigma_{Z='a'}(S)))$

The cost of Plan 2 is **obviously better** than the cost of Plan 1

Plan costs: Come back to the example

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

$\sigma, \bowtie$
 $\rightsquigarrow$

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

$\sigma, \bowtie_{W=X}$
 $\rightsquigarrow$

A	B	C	...	W	X	Y	Z
3	r	8	...	7	7	k	a

π
 $\rightsquigarrow$

B	C	Y
r	8	k

Plan 1 (initial): $\pi_{B,C,Y} (\sigma_{W=X \ \& \ A=3 \ \& \ Z='a'}(R \times S))$

Plan 2: $\pi_{B,C,Y} ((\sigma_{A=3}(R) \bowtie_{W=X} \sigma_{Z='a'}(S)))$

Plan 3: $\pi_{B,C,Y} ((\sigma_{A=3}(R) \bowtie_{W=X} \sigma_{Z='a'}(S \bowtie_{W=X} \sigma_{A=3}(R))))$

(here, $\bowtie_{W=X}$ is 'equi-semi-join', naturally expressible via ρ and $\bowtie$)

The cost of Plan 2 is **obviously better** than the cost of Plan 1

Homework: compute and compare the costs of Plans 2 and 3

5. Optimisation: Heuristics

Searching for a plan with good cost

As we discussed, given an initial query plan (RA expression) $Expr$, we have a huge search space of plans to compare with

DBMSs pragmatically find another query plan $Expr'$ such that

- $Expr$ and $Expr'$ are equivalent (i.e., give the same answer on all possible data instances)
- $Expr'$ is estimated to be quicker than other candidates using the cost function (for us: sum of sizes of intermediate sub-queries in the plan)

Searching for a plan with good cost

As we discussed, given an initial query plan (RA expression) *Expr*, we have a huge search space of plans to compare with

DBMSs pragmatically find another query plan *Expr'* such that

- *Expr* and *Expr'* are equivalent (i.e., give the same answer on all possible data instances)
- *Expr'* is estimated to be quicker than other candidates using the cost function (for us: sum of sizes of intermediate sub-queries in the plan)

They use greedy heuristic:

- starting from the initial *Expr*, they apply the algebraic laws one by one, trying to decrease the cost on (almost) each step as much as possible
- stops with a selected plan when there are no more cost improvements

Common rules

The **algebraic laws** that (almost always) decrease our cost:

- **Push selections** as far down as possible
- If the selection condition consists of several parts (AND or OR), **split into multiple selections** and **push** each one as far down the tree as possible
- **Push projections** as far down as possible
- **Combine selections and joins** (especially, Cartesian products) to appropriate theta-joins
- Apply **idempotence** as much as possible

Common rules

The **algebraic laws** that (almost always) decrease our cost:

- **Push selections** as far down as possible
- If the selection condition consists of several parts (AND or OR), **split into multiple selections** and **push** each one as far down the tree as possible
- **Push projections** as far down as possible
- **Combine selections and joins** (especially, Cartesian products) to appropriate theta-joins
- Apply **idempotence** as much as possible

Real DBMS cost functions are more fine-grained, and these rules may be less universal (e.g., due to indexes)

Order of joins

One of the most important practical question for optimisation is the order to join—that is, how to evaluate

$$(\dots (R_1 \bowtie R_2) \bowtie \dots \bowtie R_{n-1}) \bowtie R_n$$

Good estimates of intermediate results here are **crucial** for efficient evaluation (and the difference may be huge)

In general, the search space (all possible orders of ()) is large

Luckily, such joins are usually **not arbitrary**: the **join graph** has **bounded (hyper-)tree width** (**check it out!**)

Following the structure of the corresponding **hyper-tree** gives us an efficient join order (good cost)

Order of joins

One of the most important practical question for optimisation is the order to join—that is, how to evaluate

$$(\dots (R_1 \bowtie R_2) \bowtie \dots \bowtie R_{n-1}) \bowtie R_n$$

Good estimates of intermediate results here are **crucial** for efficient evaluation (and the difference may be huge)

In general, the search space (all possible orders of ()) is large

Luckily, such joins are usually **not arbitrary**: the **join graph** has **bounded (hyper-)tree width** (**check it out!**)

Following the structure of the corresponding **hyper-tree** gives us an efficient join order (good cost)

Even the choice between $R_1 \bowtie R_2$ and $R_2 \bowtie R_1$ is **not trivial**: the join algorithms are **not symmetric** (but our cost function does not distinguish these cases)

What we learned?

How DBMSs **compile** queries to RA plans and **optimise** to
presumably better plans

Next time: how plans are evaluated