



॥ त्वं ज्ञानमयो विद्वानमयोऽसि ॥

Query Optimization

Saptarshi Pyne

Assistant Professor

Department of Computer Science and Engineering
Indian Institute of Technology Jodhpur, Rajasthan, India 342030

CSL4030 Data Engineering Lectures 31, 32, 33, 34, 35
October 30th, November 1st, 6rd, 8th, 10th, 2023

What we discussed in the last class

Distributed query processing

- Factors: Disk cost, network cost, parallelizability
- Strategies for
 - Multiple replicas
 - Multiple fragments
 - Joins (such as semijoin)

What is a query evaluation plan?

Given a query, a query evaluation plan defines:

- what operations need to be performed,
- in which sequence the operations needs to be performed,
- and which algorithms or indices need to be used for executing each operation.

Example of a query evaluation plan

User request:

- Print the names of the instructors in the 'Music' department.
- Additionally, for each instructor, print the course titles that he/she is teaching.

Query: (in the relational-algebra expression)

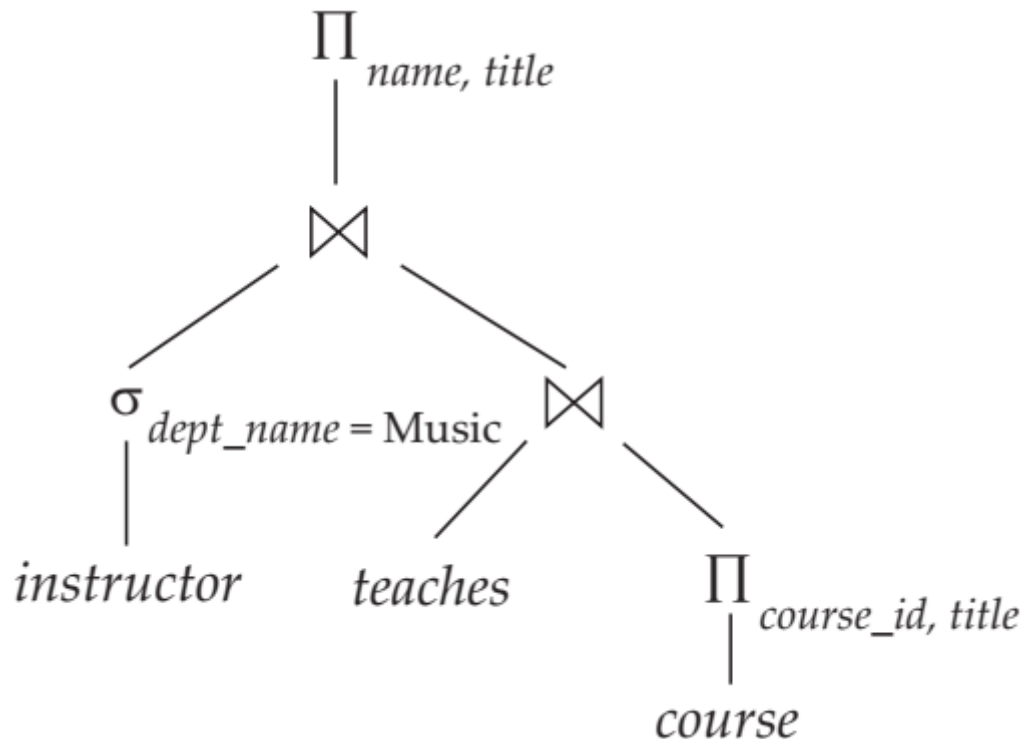
$$\Pi_{name, title} ((\sigma_{dept_name = \text{"Music"}} (instructor)) \bowtie (teaches \bowtie \Pi_{course_id, title}(course)))$$

Example of a query evaluation plan (contd.)

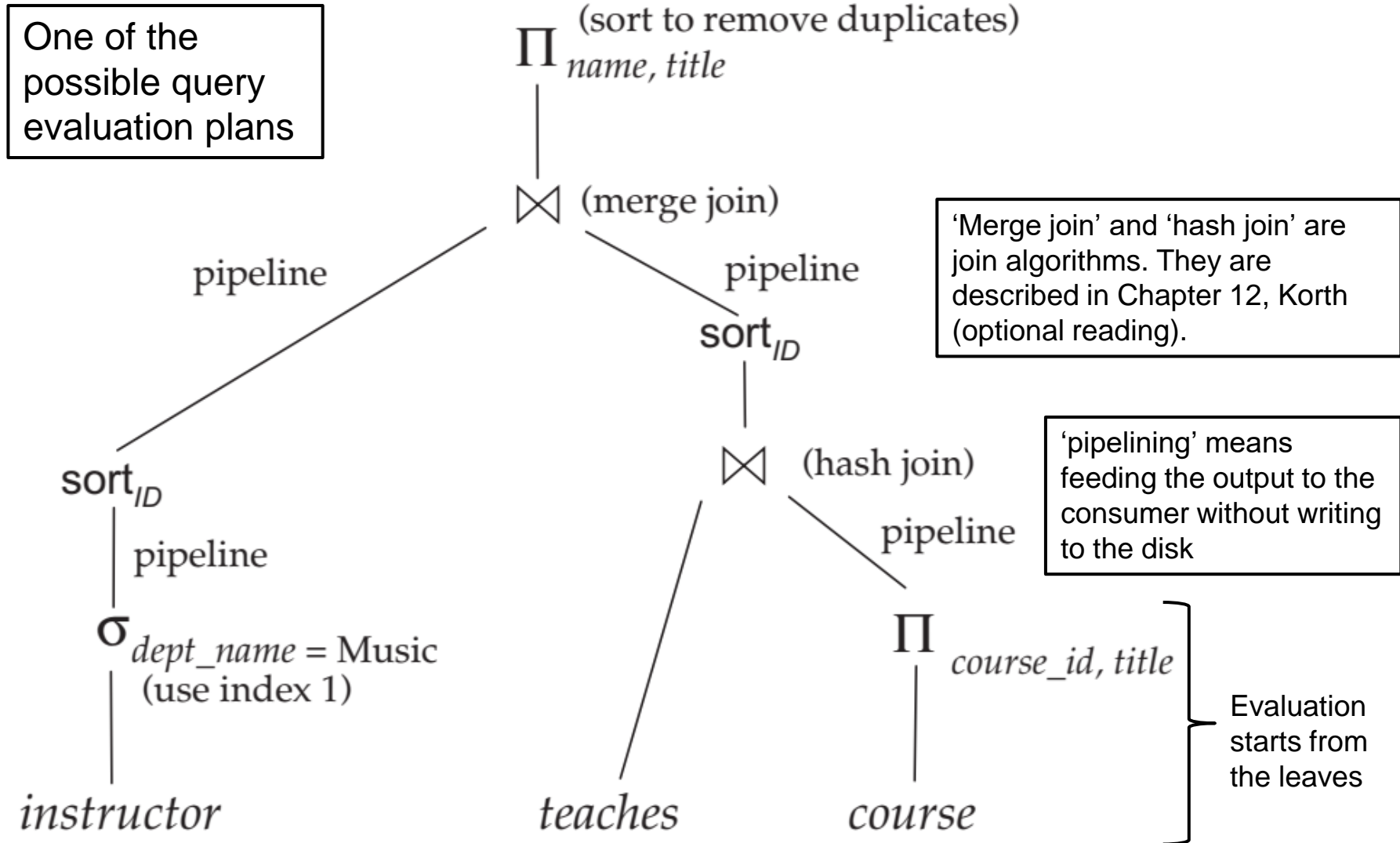
The relational-algebra expression:

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

The relational-algebra expression tree:



Example of a query evaluation plan (contd.)

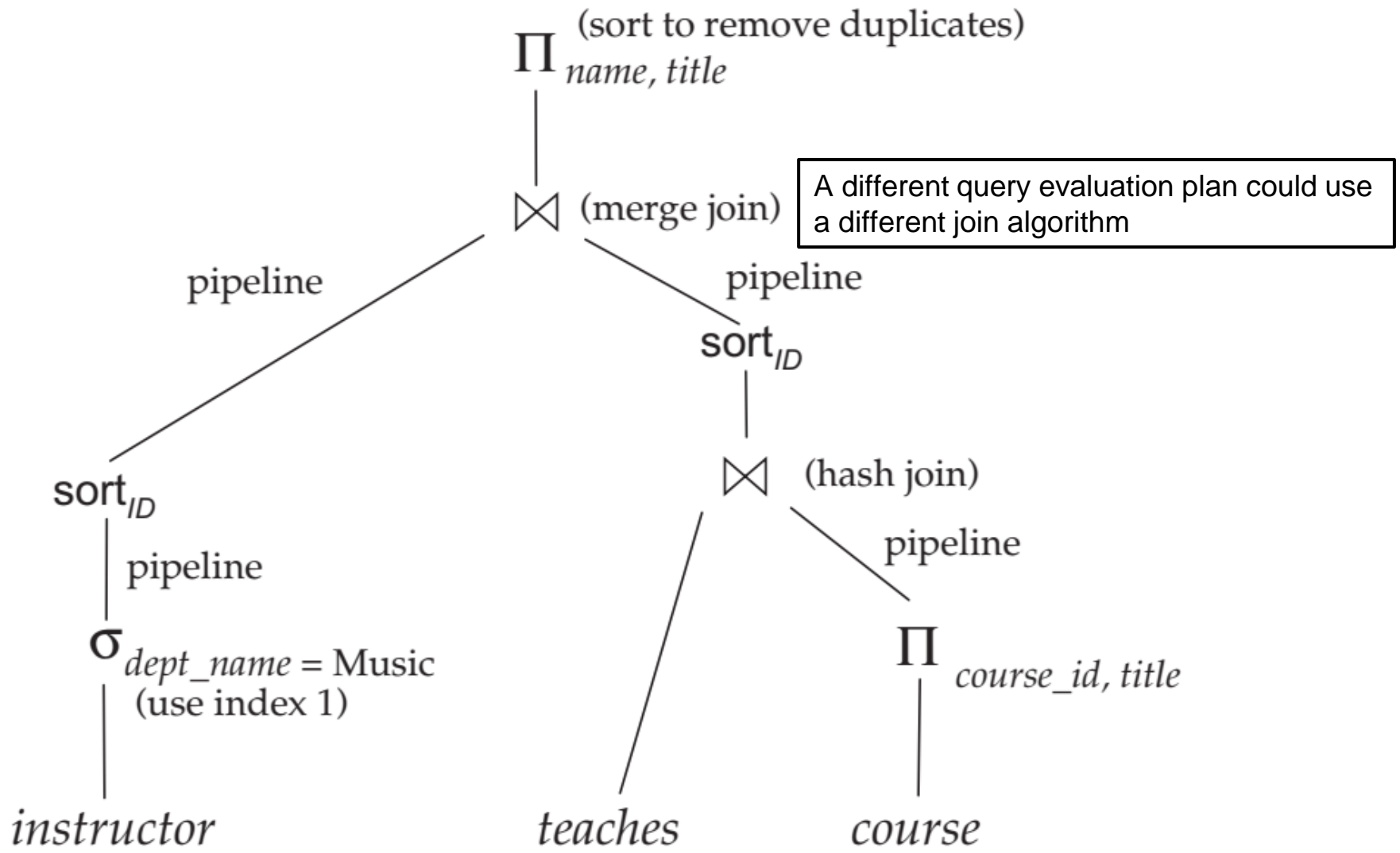


There could be multiple query evaluation plans for a query

Two query evaluation plans can differ in:

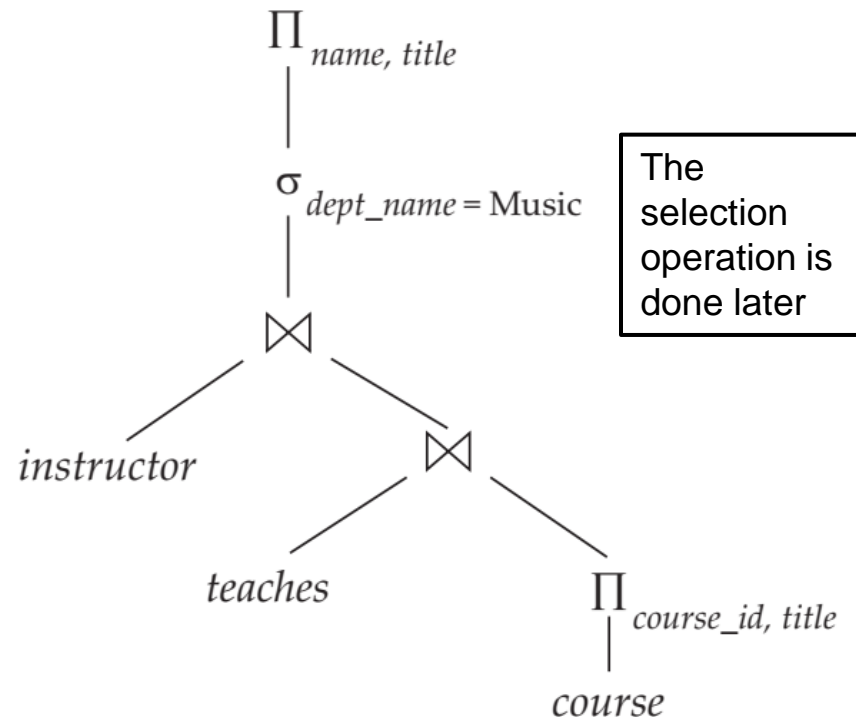
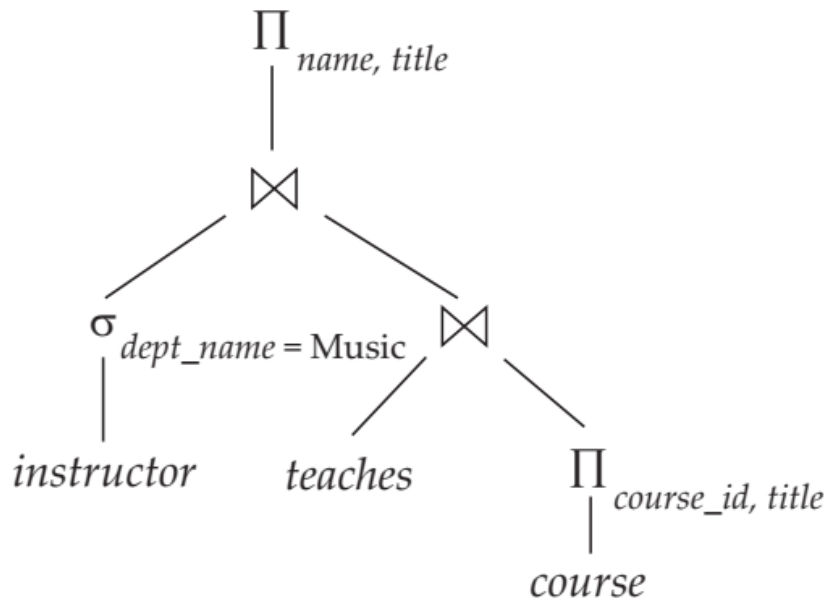
- what operations need to be performed,
- in which sequence the operations needs to be performed,
- and which algorithms or indices need to be used for executing each operation.

There could be multiple query evaluation plans for a query (contd.)



There could be multiple query evaluation plans for a query (contd.)

A different query evaluation plan could use a different set and sequence of operations i.e. a **different relational-algebra expression tree**



There could be multiple query evaluation plans for a query (contd.)

A different query evaluation plan could use a different set and sequence of operations i.e. a **different relational-algebra expression**

The
selection
operation is
done earlier

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

The
selection
operation is
done later

What is query optimization?

Query optimization is the process by which DBMS selects the most **cost**-effective query evaluation plan for a given query.

'Cost' depends on the **cost model** of a particular DBMS. Usually, it refers to **evaluation time** (which depends on the number of disk accesses, network cost, parallelizability, etc.).

How to display the chosen query evaluation plan?

Most DBMSes provide a command that displays the chosen evaluation plan for a query.

PostgreSQL uses the following command:

```
EXPLAIN <query>;
```

Example:

```
EXPLAIN SELECT * FROM film;
```

QUERY PLAN
▶ Seq Scan on film (cost=0.00..64.00 rows=1000 width=384)

Sequentially
scan the rows

Whose responsibility is it to perform query optimization?

It is the responsibility of **DBMS**, not of the user.

In other words, query optimization must be automatically done by a DBMS. Therefore, it is the **developer's** responsibility to write the query optimizer module.

How to write a query optimizer?

Pseudocode:

Enumerate all possible query evaluation plans;

For each plan

 Estimate the 'cost';

Choose a plan with the minimal cost;

How to enumerate all possible query evaluation plans?

Step 1: Generate the set of all possible relational-algebra expressions.

Step 2.1: For each algorithmic operation, generate the set of all possible algorithms (join algorithms for join operations, sorting algorithms for sorting operations, etc.)

Step 2.2: For each indexing operation, generate the set of all possible indices that can be utilised.

And so on...

How to generate the set of all possible relational-algebra expressions?

Two relational-algebra expressions are called **equivalent expressions** if they produce the same result.

Given a relational-algebra expression, there are **equivalence rules** which helps us to generate its equivalent expressions.

These rules are usually **equations**. If any subexpression of the given expression matches any side of the equation, we can replace that subexpression with the other side of the equation to generate an equivalent expression. Example of an equivalence rule is given below: here, thetas are predicates (i.e. selection criteria) and E is any expression.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

How to generate the set of all possible relational-algebra expressions? (contd.)

Pseudocode:

Define E = any one expression that produces the desired result;
The set of all equivalent expressions $EQ = \text{genAllEquiv}(\{E\})$;

function $\text{genAllEquiv}(\text{set of expressions } S)$

Initialize $S' = S$;

For each expression E in S

 For each equivalence rule R

 If any subexpression in E matches one side of R , replace it with the other side of R , thus, creating a new expression E' ;
 $S' = S' \cup E'$;

If $(|S'| > |S|)$

$S' = \text{genAllEquiv}(S')$; // Recursion

Return S' ;

A few equivalence rules

Rule 1: (serial numbers are just for the slides. There are no universally accepted ordering of the rules.)

Transform a conjunctive selection operation into a **cascade of σ 's**.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

Notations:

E's = Relational-algebra expressions.

θ 's = Predicates or selection criteria.

L's = List of attributes.

A few equivalence rules (contd.)

Rule 2: Selection is commutative.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

Rule 3: In a cascade of Π 's, retain only the final Π .

$$\Pi_{L_1}(\Pi_{L_2}(\dots(\Pi_{L_n}(E))\dots)) = \Pi_{L_1}(E)$$

A few equivalence rules (contd.)

Rule 4:

$$\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$$

where \bowtie_{θ} denotes a conditional join or a 'theta join'.

Examples:

STUDENT $\bowtie_{\text{STUDENT.ROLLNO=SUBJECT.RNO}}$ SUBJECT;

CUSTOMER $\bowtie_{\text{CUSTOMER.AGE BETWEEN AGEGRP.MIN AND AGEGRP.MAX}}$ AGEGRP;

A few equivalence rules (contd.)

Rule 5: Theta join is commutative.

$$E_1 \bowtie_{\theta} E_2 = E_2 \bowtie_{\theta} E_1$$

The ordering of attributes in the result of the RHS may differ from that of the LHS. We can simply use a **projection** operation on the result of the RHS to make it exactly the same as that of the LHS.

Since, the natural join is a special case of the theta join, **natural join is also commutative.**

A few equivalence rules (contd.)

Rule 6(a): Natural join is associative.

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

Rule 6(b): Theta join is associative if θ_2 does not involve attributes from E_1 .

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

A few equivalence rules (contd.)

Rule 7(a): Selection is distributive over theta join if θ_0 does not involve attributes from E_2 .

$$\sigma_{\theta_0}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_{\theta} E_2$$

Rule 7(b): Selection is distributive over theta join if θ_1 involves attributes from E_1 only and θ_2 involves attributes from E_2 only.

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_{\theta} (\sigma_{\theta_2}(E_2))$$

A few equivalence rules (contd.)

Rule 8: Projection is distributive over theta join for the following cases.

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_{\theta} E_2) = (\Pi_{L_1}(E_1)) \bowtie_{\theta} (\Pi_{L_2}(E_2))$$

If $\{L_1, L_3\}$ are attributes of E_1 ,
 $\{L_2, L_4\}$ are attributes of E_2 , and
 $\{L_3, L_4\}$ are involved in θ , then

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_{\theta} E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_{\theta} (\Pi_{L_2 \cup L_4}(E_2)))$$

A few equivalence rules (contd.)

Rule 9: Union and intersection are commutative.

$$E_1 \cup E_2 = E_2 \cup E_1$$

$$E_1 \cap E_2 = E_2 \cap E_1$$

Note: Difference is not commutative.

$$(E_1 - E_2) \neq (E_2 - E_1)$$

A few equivalence rules (contd.)

Rule 10: Union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$

$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

A few equivalence rules (contd.)

Rule 11: Selection is distributive over union, intersection, and difference.

$$\sigma_{\theta}(E1 \cup E2) = \sigma_{\theta}(E1) \cup \sigma_{\theta}(E2)$$

$$\sigma_{\theta}(E1 \cap E2) = \sigma_{\theta}(E1) \cap \sigma_{\theta}(E2)$$

$$\sigma_{\theta}(E1 - E2) = \sigma_{\theta}(E1) - \sigma_{\theta}(E2)$$

Rule 12: Projection is distributive over union.

$$\Pi_L(E1 \cup E2) = (\Pi_L(E1)) \cup (\Pi_L(E2))$$

... there are many such rules.

An application of equivalence rules: Join ordering

A good join order can reduce the size of the intermediate results, which in turn saves memory and evaluation time.

Consider the following relations.

instructor: instrID, instrName, dept_name, ...

teaches: instrID, courseID, ...

course: courseID, courseTitle, ...

User request: Print the names of the music instructors along with the titles of the courses they are teaching.

An application of equivalence rules: Join ordering (contd.)

Query:

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

Join order 1:

$$\sigma_{dept_name = \text{“Music”}} (instructor) \bowtie (teaches \bowtie \Pi_{course_id, title} (course))$$

Join order 2:

$$(\sigma_{dept_name = \text{“Music”}} (instructor) \bowtie teaches) \bowtie \Pi_{course_id, title} (course)$$

Join order 3:

$$(\sigma_{dept_name = \text{“Music”}} (instructor) \bowtie \Pi_{course_id, title} (course)) \bowtie teaches$$

All three join orders are equivalent since **natural joins are associative (rule 6a) and commutative (rule 5).**

An application of equivalence rules: Join ordering (contd.)

Join order 1:

$\sigma_{\text{dept_name}=\text{"Music"}}(\text{instructor}) \bowtie (\text{teaches} \bowtie \Pi_{\text{course_id, title}}(\text{course}))$

Join order 2:

$(\sigma_{\text{dept_name}=\text{"Music"}}(\text{instructor}) \bowtie \text{teaches}) \bowtie \Pi_{\text{course_id, title}}(\text{course})$

Join order 3:

$(\sigma_{\text{dept_name}=\text{"Music"}}(\text{instructor}) \bowtie \Pi_{\text{course_id, title}}(\text{course})) \bowtie \text{teaches}$

However, do they take the same amount of memory and evaluation time?

An application of equivalence rules: Join ordering (contd.)

Join order 1:

$\sigma_{\text{dept_name}=\text{"Music"}}(\text{instructor}) \bowtie (\text{teaches} \bowtie \Pi_{\text{course_id, title}}(\text{course}))$

Join order 2:

$(\sigma_{\text{dept_name}=\text{"Music"}}(\text{instructor}) \bowtie \text{teaches}) \bowtie \Pi_{\text{course_id, title}}(\text{course})$

Join order 3: (No common attribs, join transforms into a Cartesian product)

$(\sigma_{\text{dept_name}=\text{"Music"}}(\text{instructor}) \times \Pi_{\text{course_id, title}}(\text{course})) \bowtie \text{teaches}$

However, do they take the same amount of memory and evaluation time?

They do not. Join order 2 requires the least amount of memory and time.

Minimal set of query equivalence rules

When developing a query optimizer module, we should use a **minimal set** of equivalence rules.

In a minimal set of equivalence rules, no rules can be derived by combining a subset of the other rules.

How to enumerate all possible query evaluation plans? (recap)

Step 1: Generate the set of all possible relational-algebra expressions.

Step 2.1: For each algorithmic operation, generate the set of all possible algorithms (join algorithms for join operations, sorting algorithms for sorting operations, etc.)

Step 2.2: For each indexing operation, generate the set of all possible indices that can be utilised.

And so on...

How to write a query optimizer? (recap)

Pseudocode:

Enumerate all possible query evaluation plans;

For each plan

 Estimate the 'cost';

Choose a plan with the minimal cost;

How to write a query optimizer? (recap)

Pseudocode:

Enumerate all possible query evaluation plans;

For each plan

Estimate the 'cost';

Choose a plan with the minimal cost;

How to estimate the cost of a query evaluation plan?

The cost of a plan depends on the costs of the operations involved.

The cost of each operation in turn depends on the sizes of its operators (which are usually relations).

Hence, a query optimizer stores metadata information about the relations in **database-system catalogs**.

Catalog information

For each relation r ,

- The number of tuples (n_r)
- The length of each tuple in bytes (l_r)
- The number of distinct values of each attribute.

$V(A, r)$ = The number of distinct values of attribute A in $r = |\Pi_A(r)|$ since projection returns distinct values.

For a set of attributes that are accessed together, say S , DBMS can catalog $V(S, r)$.

Catalog information (contd.)

For each relation r (contd.),

- The number of tuples that fit into one disk block aka. the blocking factor of r (f_r)
- The number of blocks that r occupies (b_r)

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

Catalog information (contd.)

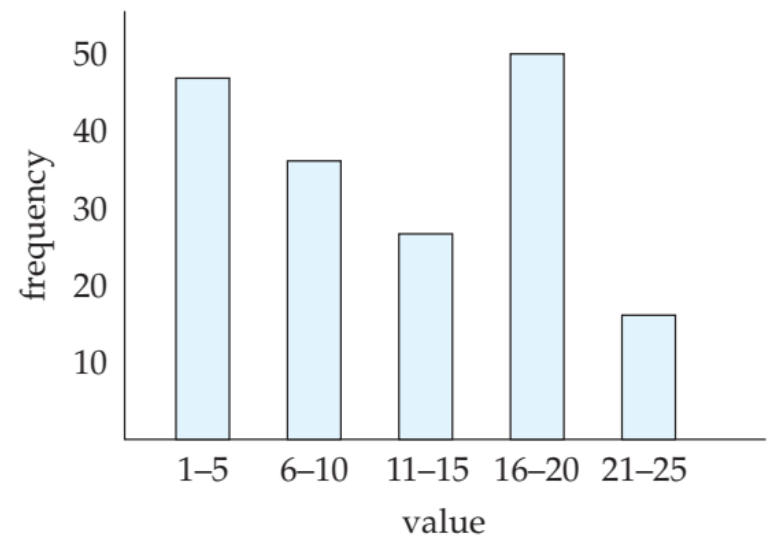
For each index structure (suppose, a B⁺ tree),

- The height of the tree
- The number of leaves

Catalog information (contd.)

A query optimizer usually stores additional complex metadata in catalogs, such as a histogram for the values of an attribute.

The shown histogram is an equi-width histogram i.e. each value range has the same width.



The query optimizer can also store an equi-depth histogram where the range widths vary but the number of values in each range is the same.

Selection size estimation

Metadata like histograms are very helpful in estimating the number of tuples that will match a selection criteria.

Generating a histogram for millions of tuples will be costly. Hence, a query optimizer tends to **randomly sample** a few thousand tuples to generate the histogram.

Selection size estimation (contd.)

Based on the histogram, the query optimizer estimates the number of tuples that matches a selection criteria. Depending on the estimation, the optimizer chooses the best query evaluation plan.

When the plan is executed, the actual number of matching tuples is revealed. If the actual number is far from the estimated number, the optimizer redo the random sampling or use all the tuples to generate the histogram. This is a **self-correction mechanism** of the optimizer.

Selection size estimation (contd.)

- For a conjunctive selection
- For a disjunctive selection
- For a negative selection (e.g., `SELECT * FROM employee WHERE dependent IS NOT NULL`)

Please study Section 13.3.2 from Korth.

Keeping catalog information up-to-date

Updating catalog information is resource intensive. Hence, it is usually not updated every time a relation or index structure is modified.

It is usually done when the DBMS has a light workload i.e. resources (disk, network, etc.) are available.

As a result, the cost estimated using the catalogs is not expected to be accurate but close to accurate.

Join size estimation

Estimate the output size of:

$r \bowtie s$

No common attributes.

$n_r * n_s * (l_r + l_s)$ bytes

The common attributes are part of r 's primary key.
Each tuple of s can join with at most one tuple of r .

$\leq n_s * (l_r + l_s)$ bytes

Join size estimation (contd.)

The primary key of r is referenced by a foreign key in s .
 $= n_s * (l_r + l_s)$ bytes

Join size estimation (contd.)

The set of common attributes $\{A\}$ contain the primary key of neither r nor s .

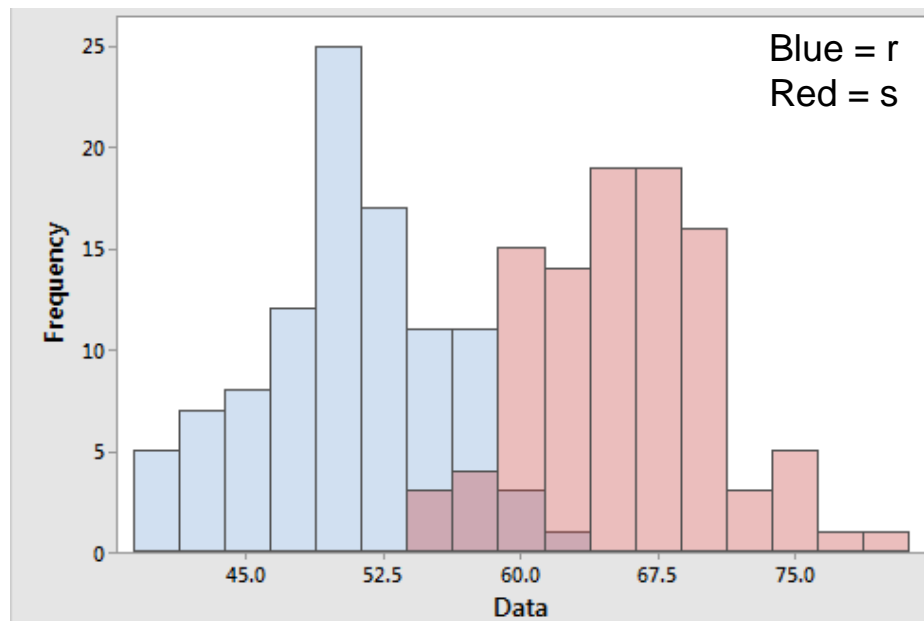
Case 1: If r and s do not have histograms on $\{A\}$, assume each value is equally likely.

$$= \min \left(\frac{(nr * ns)}{V(A, r)}, \frac{(nr * ns)}{V(A, s)} \right) * (lr + ls) \text{ bytes}$$

Join size estimation (contd.)

Case 2(a): If

- r and s have histograms on $\{A\}$ and
- the histograms are on the same ranges.



Join size estimation (contd.)

For each attribute A_i in $\{A\}$

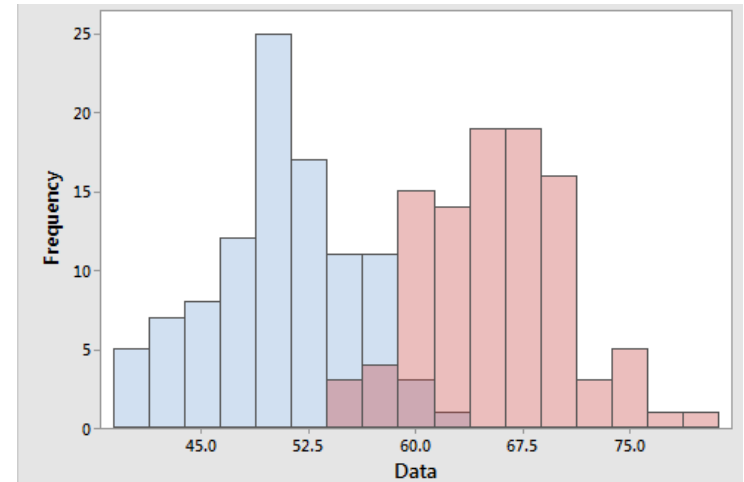
For each histogram range/bin

Use the equation from Case 1;

Replace n_r with the actual frequency in r ;

Replace n_s with the actual frequency in s ;

Replace $V(A, r)$ and $V(A, s)$ with the number of distinct values in the range;

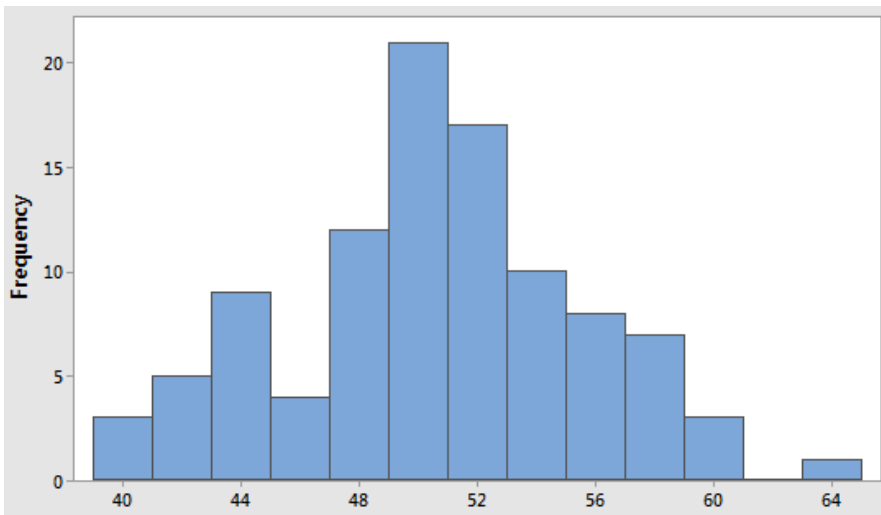


Join size estimation (contd.)

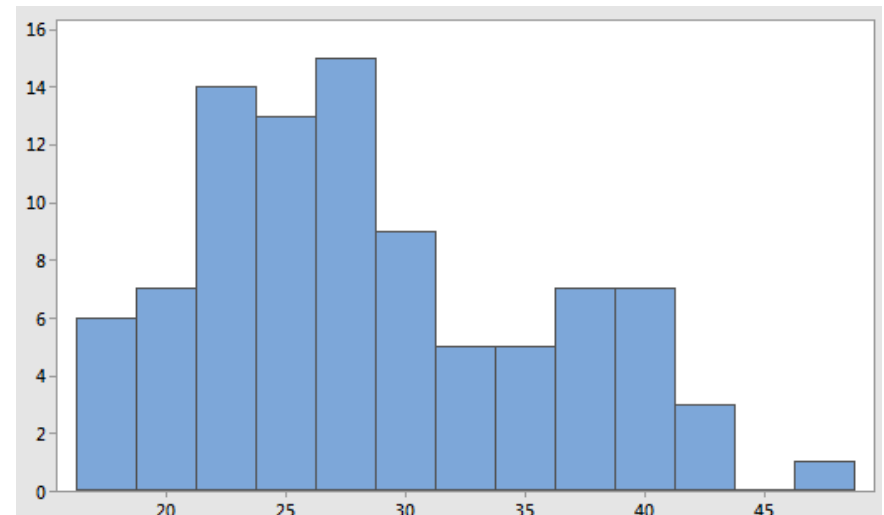
Case 2(b): If

- r and s have histograms on $\{A\}$ and
- the histograms are NOT on the same ranges.

Relation r , attribute A_i



Relation s , attribute A_i

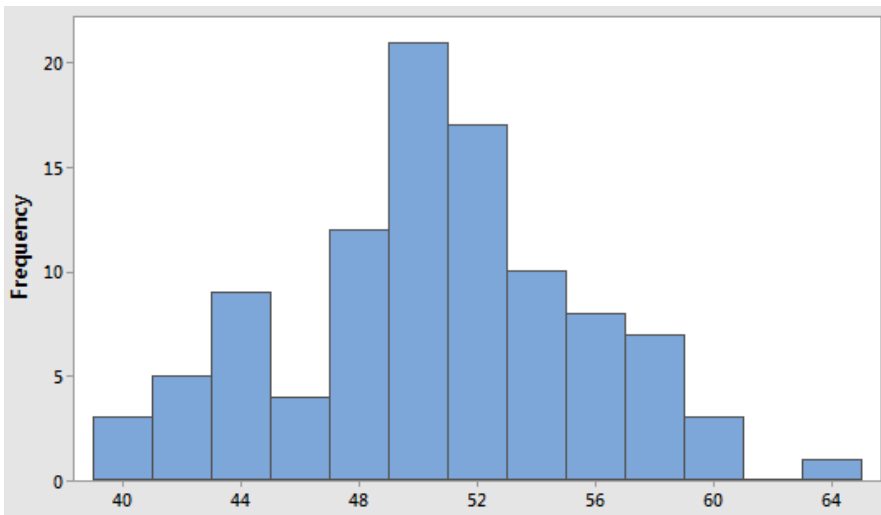


Join size estimation (contd.)

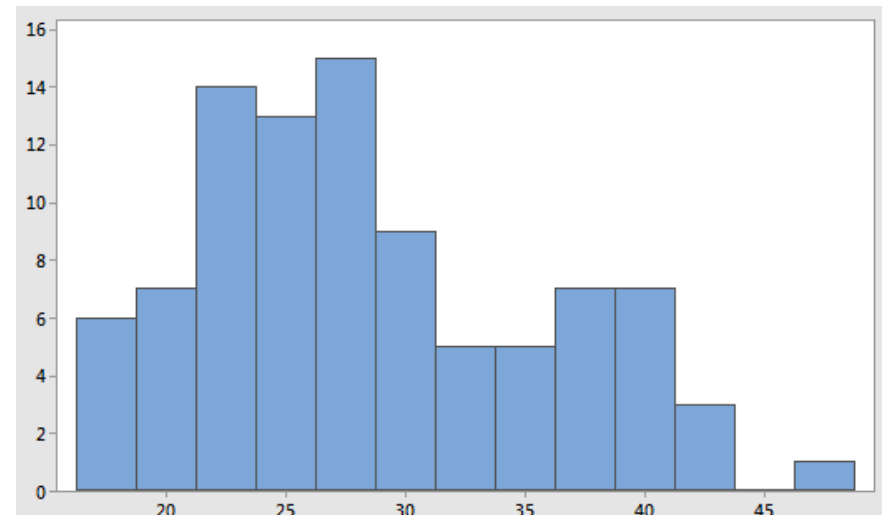
Solution hint:

For each range in each histogram, assume each value is equally frequent.

Relation r, attribute A_i



Relation s, attribute A_i



Join size estimation (contd.)

To illustrate all these ways of estimating join sizes, consider the expression:

$$student \bowtie takes$$

Assume the following catalog information about the two relations:

- $n_{student} = 5000$.
- $f_{student} = 50$, 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 only half the students have taken any course (this is unrealistic, but we use it to show that our size estimates are correct even in this case), and on average, each student who has taken a course has taken four courses.

Please see the solution given in Korth.

Size estimation for other operations

Please see Korth Section 13.3.4 to learn how to perform size estimations for the following operations:

- projection
- aggregation: count, sum, average, etc.
- set operations
- outer join

Estimating the number of unique values

Estimation of $V(A, r \bowtie s)$

Suppose, A is made up of $A1$ from r and $A2$ from s .

$$\min(V(A1, r) * V(A2 - A1, s), V(A1 - A2, r) * V(A2, s), n_{r \bowtie s})$$

In case, the result of the join does not have all possible values of A

How to write a query optimizer? (recap)

Pseudocode:

Enumerate all possible query evaluation plans;

For each plan

Estimate the 'cost';

Choose a plan with the minimal cost;

How to write a query optimizer? (recap)

Pseudocode:

Enumerate all possible query evaluation plans;

For each plan

 Estimate the 'cost';

Choose a plan with the minimal cost;

Such exhaustive search-based query optimization is also known as the **cost-based query optimization**.

The optimization cost

The cost of finding an optimal query evaluation plan is called the **optimization cost**.

For some type of queries, the optimization cost could be too high to consider optimization at all.

Example:

For a cascade of $(n-1)$ natural join operations with 'n' operands, the optimization cost for finding an optimal join order can be very high.

Finding an optimal join order

For a cascade of $(n-1)$ natural join operations with 'n' operands, there are $(2(n-1))!/(n-1)!$ possible join orders.

Example: $(n=3)$
 $r_1 \bowtie r_2 \bowtie r_3$

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

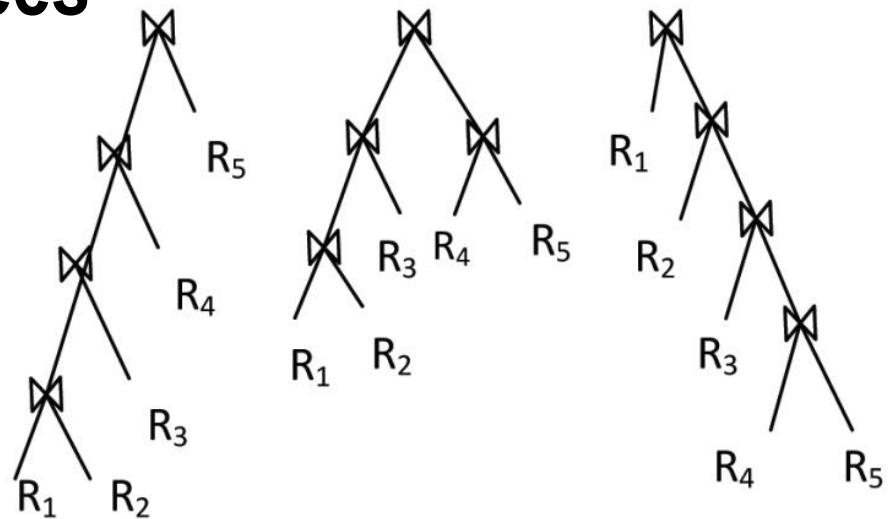
For $n=10$, billions of join orders to consider.

Finding an optimal join order (contd.)

For a cascade of $(n-1)$ natural join operations with 'n' operands, there are $(2(n-1))!/(n-1)!$ possible join orders.

Proof sketch:

There are that many **join trees** (three possible join trees shown for $n=5$).



Finding an optimal join order (contd.)

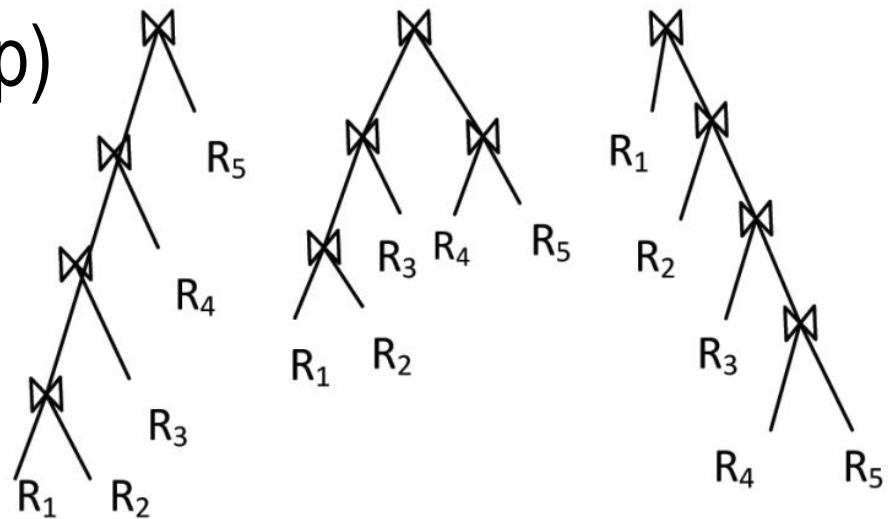
A join tree is a **full binary tree** where **leaves** are relations and **internal nodes** are join operators.

Three main types of join trees (from L to R):

- **Left-deep** = Each right operand is a fresh relation. The left operand could be an intermediate stored relation.

- **Bushy** (left- plus right-deep)

- **Right-deep**



Finding an optimal join order (contd.)

The number of full binary trees with 'n' **unlabelled** leaves

= The $(n-1)^{\text{th}}$ Catalan number

$$= (1/n)^* \binom{2(n-1)}{(n-1)}$$

The number of ways to label 'n' leaves = n!

Therefore, the number of full binary trees with 'n' **labelled** leaves

$$= (1/n)^* \binom{2(n-1)}{(n-1)} * n!$$

$$= (2(n-1))! / (n-1)!$$

What if the join order is partially specified

Q. How many join orders are possible for the following query? $(r1 \bowtie r2 \bowtie r3) \bowtie r4 \bowtie r5$

Ans. 144

12 join orders are possible for $(r1 \bowtie r2 \bowtie r3)$.

12 join orders are possible for $(\dots) \bowtie r4 \bowtie r5$.

$12 * 12 = 144$.

What if the join order is partially specified (contd.)

However, we can find an optimal join order by examining only 24 join orders.

1. Examine 12 possible join orders for $(r1 \bowtie r2 \bowtie r3)$.
2. Join $(r1 \bowtie r2 \bowtie r3)$ and save the result in $r1'$.
3. Examine 12 possible join orders for $(r1' \bowtie r4 \bowtie r5)$.

Number of join orders examined = $(12 + 12) = 24$

Thus, when applicable, we can divide the expression into subexpressions and find an optimal join order for each subexpression. **(Divide and conquer)**

What if the join order is partially specified (contd.)

However, we can find an optimal join order by examining only 24 join orders.

1. Examine 12 possible join orders for $(r1 \bowtie r2 \bowtie r3)$.
2. Join $(r1 \bowtie r2 \bowtie r3)$ and save the result in $r1'$.
3. Examine 12 possible join orders for $(r1' \bowtie r4 \bowtie r5)$.

Number of join orders examined = $(12 + 12) = 24$

Thus, when applicable, we can divide the expression into subexpressions and find an optimal join order for each subexpression. **(Divide and conquer)**

Can we utilize parallelization?

$(r1 \bowtie r2 \bowtie r3) \bowtie (r4 \bowtie r5 \bowtie r6)$

Yes, when possible, we can divide the expression into subexpressions and find an optimal join order for each subexpression. (**Divide and conquer**)

How should we handle duplicate subexpressions?

$(r1 \bowtie r2 \bowtie r3) \bowtie (r4 \bowtie r5 \bowtie r6) \bowtie (r1 \bowtie r2 \bowtie r3)$

- First, we should design an algorithm for **efficiently detecting duplicate subexpressions**.
- Then we can find an optimal join order for a subexpression only once and **memoize** it for the duplicate occurrences. (**Dynamic programming**)

We should also utilize heuristics

- Perform selection operations as early as possible
- Perform projections early
- Avoid Cartesian products, etc.

Demerit:

Heuristics do **not always** lead to an **optimal** query evaluation plan.

$\sigma_{\theta}(r1 \bowtie r2)$: Suppose, we have an index on the join attributes but not on the 'θ' attributes. Then performing the selection after the join might be optimal.

Combining the best of both worlds – exhaustive and heuristic

Most commercial optimizers set an **optimization cost budget (say, 5 sec in time)** before performing optimization.

The goal is:

(Optimization cost budget + cost of executing the chosen query evaluation plan) should be far less than the cost of executing a random query evaluation plan.

Combining the best of both worlds – exhaustive and heuristic (contd.)

Most commercial optimizers, first, use some cost-efficient heuristics to generate some plans.

Using these initial plans, the optimizer makes a prediction about the execution cost of an optimal plan.

Utilizing the prediction, the optimizer sets its optimization cost budget.

Combining the best of both worlds – exhaustive and heuristic (contd.)

Subsequently, the optimizer initiates an **exhaustive search**.

If the optimization cost budget is reached before the search ends, the optimizer terminates the search and chooses **the best plan found up to that point**.

The Halloween problem (IBM, 1976)

“Pat (Patricia Selinger) and Morton (Morton Astrahan) discovered this problem on Halloween...

I remember they came into my office and said, ‘Chamberlin, look at this. We have to make sure that **when the optimizer is making a plan for processing an update**, it doesn't use an index that is based on the field that is being updated. How are we going to do that?’

It happened to be on a Friday, and we said, ‘Listen, we are not going to be able to solve this problem this afternoon. Let's just give it a name. We'll call it the Halloween Problem and we'll work on it next week.’ And it turns out it has been called that ever since.”

~ Oral history interview with Donald D. Chamberlin, co-designer of SQL

The Halloween problem (contd.)

Pat, Morton, and Don were trying to write a query optimizer function for UPDATE queries that give raise to the employees whose salaries are less than a certain amount.

(This happens to be the exact query they were tinkering with)

```
UPDATE employee
```

```
SET salary = salary * 1.1
```

```
WHERE salary < $25,000;
```


The Halloween problem (contd.)

Now, without query optimization, the DBMS first executes the following query (given in the MySQL syntax) to retrieve the list of target tuples.

```
SELECT @target := empid FROM employee WHERE salary < $25,000;
```

Then update the target tuples.

```
UPDATE employee  
SET salary = salary * 1.1  
WHERE empid = @id;
```

The Halloween problem (contd.)

However, such sequential execution is extremely time consuming for large organizations with thousands of employees.

Hence, Pat and Morton were attempting for parallelization: there was a **scanner** that was finding the target tuples and there was an **updater** that was updating the already found tuples. Both were running in parallel.

The Halloween problem (contd.)

The problem was that the 'employee' table was indexed (e.g., column-oriented RLE bitmap indexed) as large tables should be.

Hence, as soon as a tuple was updated, it was reindexed in the index structure. The bug was encountered when the index of the updated tuple was placed ahead of the scanner.

Suppose, the scanner is scanning in the ascending order of salaries. It scanned an employee with a salary of \$20,000 and marked it as a target. The updater updated the salary to be \$22,000. At this point, suppose, the scanner is still scanning the employees with salary \$21,000. Therefore, the scanner will see this employee again and mark it as a target. It may keep happening in a loop until the employee's salary reaches \$25,000 or above.

The Halloween problem (contd.)

Pat and Morton observed exactly that. When the query completed, all the employees had salaries of \$25,000 or above.

One of the solutions:

- If the updater increases the salary, then the scanner needs to scan in the descending order of salaries.
- On the other hand, if the updater decreases the salary, then the scanner needs to scan in the ascending order of salaries.

The Halloween problem (contd.)

If for some database conditions, we can not ensure that the updater will never get ahead of the scanner i.e. **it is not guaranteed that the Halloween problem will not occur, query optimization must not be performed.**

Because data integrity is more important than query optimization for such data.

In general, the Halloween problem corresponds to any large indexed table where update operations take place periodically, e.g., on Diwali, Amazon wants to apply a flat 50% discount on all products with prices less than ₹1,999.

Materialized views

- How to keep a materialized view up-to-date a.k.a. ‘how to maintain a materialized view’?
 - Incremental ‘materialized view maintenance’: Compute differential in the result using only the **differential in the input** (the **insert** and **delete** operations).
- How to select which views should be materialized?
 - Not in the syllabus but please remember the question since it is the first question to answer when designing a materialized view.

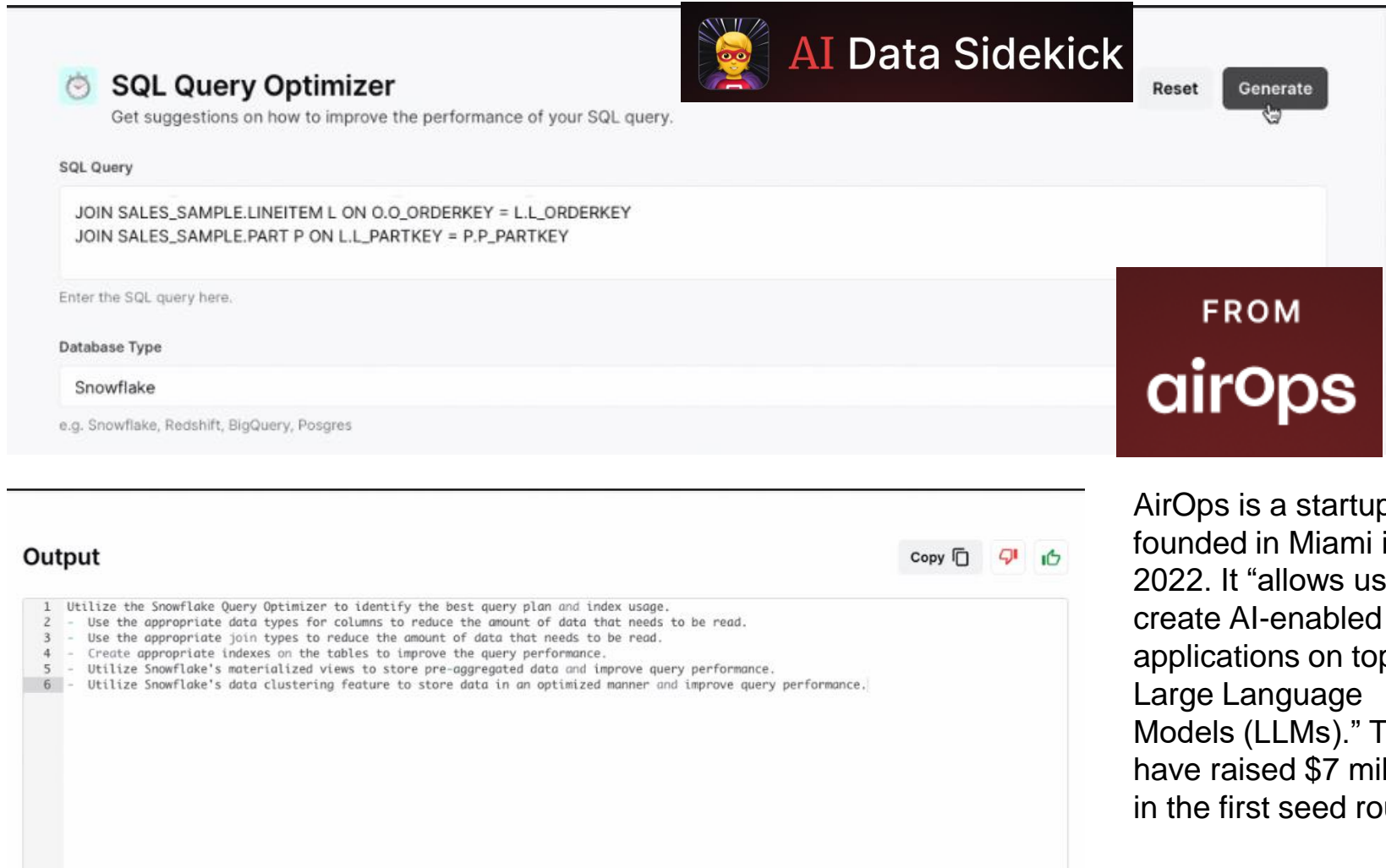
Examples of commercial query optimizers

1979: **The System R Optimizer** was developed for IBM System R (1974), which was the first implementation of SQL.

1989: **Starburst**. It was the successor of the System R Optimizer and used in the initial versions of IBM DB2.

1993-95: **Volcano** and **Cascades** were developed for Microsoft SQL server.

Data Sidekick: AI based SQL query optimizer



SQL Query Optimizer
Get suggestions on how to improve the performance of your SQL query.

SQL Query

```
JOIN SALES_SAMPLE.LINEITEM L ON O.O_ORDERKEY = L.L_ORDERKEY  
JOIN SALES_SAMPLE.PART P ON L.L_PARTKEY = P.P_PARTKEY
```

Enter the SQL query here.

Database Type

Snowflake

e.g. Snowflake, Redshift, BigQuery, Posgres

Output

- 1 Utilize the Snowflake Query Optimizer to identify the best query plan and index usage.
- 2 - Use the appropriate data types for columns to reduce the amount of data that needs to be read.
- 3 - Use the appropriate join types to reduce the amount of data that needs to be read.
- 4 - Create appropriate indexes on the tables to improve the query performance.
- 5 - Utilize Snowflake's materialized views to store pre-aggregated data and improve query performance.
- 6 - Utilize Snowflake's data clustering feature to store data in an optimized manner and improve query performance.

FROM
aiOps

AirOps is a startup founded in Miami in 2022. It “allows users to create AI-enabled applications on top of Large Language Models (LLMs).” They have raised \$7 million in the first seed round.

References

- Chapter 13 'Query Optimization', A. SILBERSCHATZ, H.F. KORTH, S. SUDARSHAN (2011), Database System Concepts, McGraw Hill Publications, 6th Edition.

Thank you