Dario Della Monica

Chapter 16: Query Optimization

These slides are a modified version of the slides provided with the book:

Database System Concepts, 6th Ed.

(however, chapter numeration refers to 7th Ed.)

The original version of the slides is available at: https://www.db-book.com/

Chapter 16: Query Optimization

- Introduction
- Generating Equivalent Expressions
  - Equivalence rules
  - How to generate (all) equivalent expressions
- Estimating Statistics of Expression Results
  - The Catalog
  - Size estimation
    - Selection
    - Join
    - Other operations (projection, aggregation, set operations, outer join)
  - Estimation of number of distinct values
- Choice of Evaluation Plans
  - Dynamic Programming for Choosing Evaluation Plans

Introduction

- **Query optimization**: finding the "best" **query execution plan** (**QEP**) among the many possible ones
  - User is not expected to write queries efficiently (DBMS optimizer takes care of that)
- Alternative ways to execute a given query – 2 levels
  - Equivalent relational algebra expressions
  - Different implementation choices for each relational algebra operation
    - Algorithms, indices, coordination between successive operations, …

Introduction

- **Query optimization**: finding the "best" **query execution plan** (**QEP**) among the many possible ones
  - User is not expected to write queries efficiently (DBMS optimizer takes care of that)
- Alternative ways to execute a given query – 2 levels
  - Equivalent relational algebra expressions
  - Different implementation choices for each relational algebra operation
    - Algorithms, indices, coordination between successive operations, …

INSTR(i_id, name, dept_name, ...)
COURSE(c_id, title, ...)
TEACHES(i_id, c_id, ...)

The name of all instructors in the department of Music together with the titles of all courses they teach

Introduction

- **Query optimization**: finding the "best" **query execution plan** (**QEP**) among the many possible ones
  - User is not expected to write queries efficiently (DBMS optimizer takes care of that)
- Alternative ways to execute a given query – 2 levels
  - Equivalent relational algebra expressions
  - Different implementation choices for each relational algebra operation
    - Algorithms, indices, coordination between successive operations, …

INSTR(i_id, name, dept_name, ...)
COURSE(c_id, title, ...)
TEACHES(i_id, c_id, ...)

The name of all instructors in the department of Music together with the titles of all courses they teach

```
SELECT  I.name, C.title
FROM INSTR I, COURSE C, TEACHES T
WHERE I.i_id = T.i_id
 AND T.c_id = C.c_id
AND dept_name="Music"
```

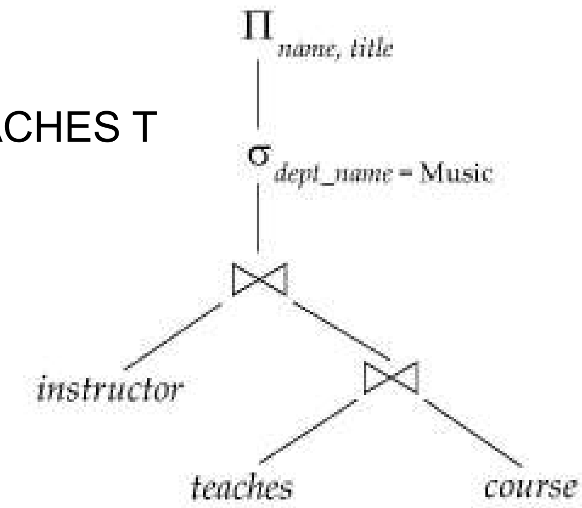
Introduction

- **Query optimization**: finding the "best" **query execution plan** (**QEP**) among the many possible ones
  - User is not expected to write queries efficiently (DBMS optimizer takes care of that)
- Alternative ways to execute a given query – 2 levels
  - Equivalent relational algebra expressions
  - Different implementation choices for each relational algebra operation
    - Algorithms, indices, coordination between successive operations, …

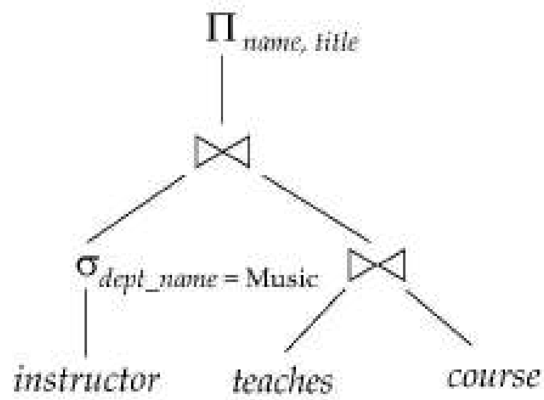
INSTR(<u>i_id</u>, name, dept_name, ...)
COURSE(<u>c_id</u>, title, ...)
TEACHES(<u>i_id</u>, <u>c_id</u>, ...)

The name of all instructors in the department of Music together with the titles of all courses they teach

```
SELECT  I.name, C.title
FROM INSTR I, COURSE C, TEACHES T
WHERE I.i_id = T.i_id
 AND T.c_id = C.c_id
AND dept_name="Music"
```



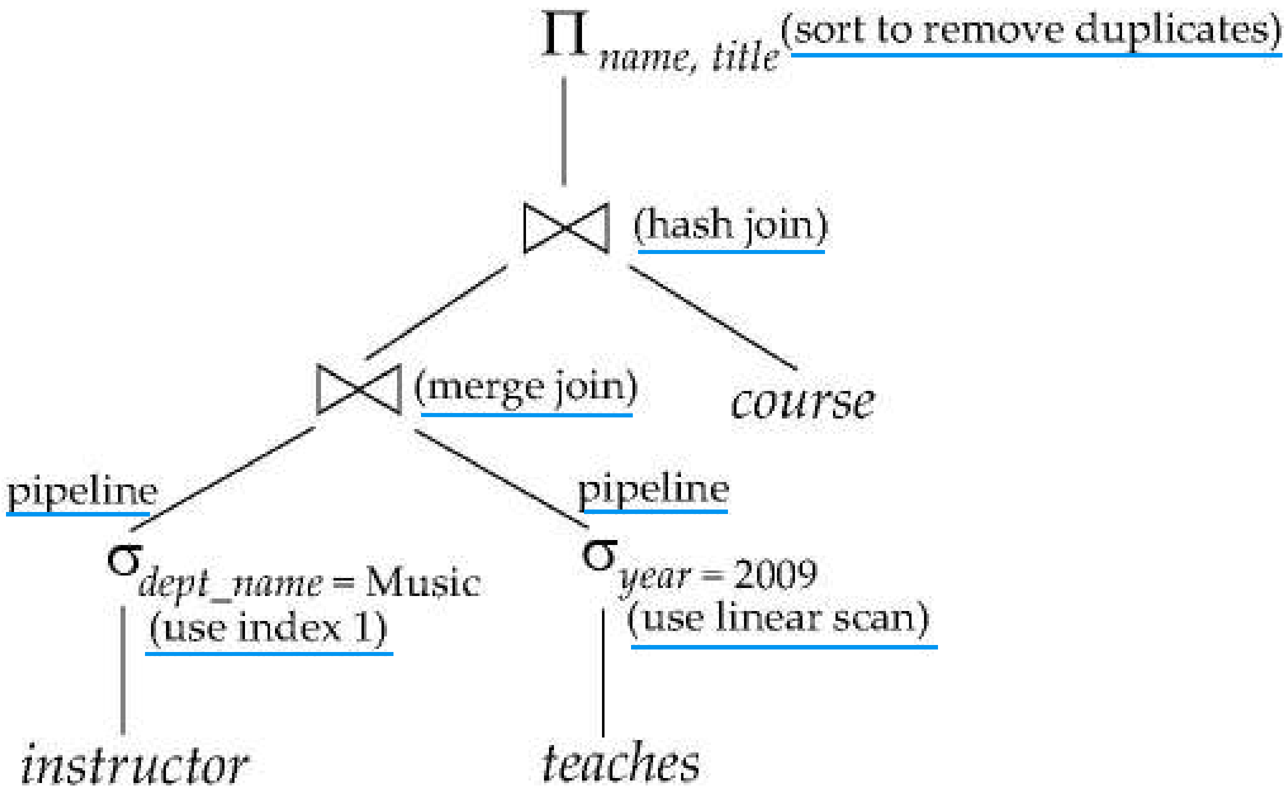
$\prod(\sigma(INSTR \bowtie (TEACHES \bowtie COURSE)))$



$\prod(\sigma(INSTR) \bowtie (TEACHES \bowtie COURSE))$

Introduction (Cont.)

- A **query evaluation plan** (**QEP**) 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

Introduction (Cont.)

- Cost difference between query evaluation plans can be enormous
  - E.g. seconds vs. days in some cases
  - It is worth spending time in finding "best" QEP
- Steps in **cost-based query optimization**
  1. Generate logically equivalent expressions using **equivalence rules**
  2. Annotate in all possible ways resulting expressions to get alternative QEP
  3. Evaluate/estimate the cost (execution time) of each QEP
  4. Choose the cheapest QEP based on **estimated cost**
- Estimation of QEP cost based on:
  - Statistical information about relations (stored in the **Catalog**)
    - 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

Generating Equivalent Expressions

- Equivalence rules
- How to generate (all) equivalent expressions

These slides are a modified version of the slides provided with the book:

Database System Concepts, 6th Ed.

(however, chapter numeration refers to 7th Ed.)

The original version of the slides is available at: https://www.db-book.com/

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 (and also order of attributes)
  - We don't care if they generate different results on databases that violate integrity constraints (e.g., uniqueness of keys)
- In SQL, inputs and outputs are multisets 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
  - We focus on relational algebra and treat relations as sets
- An **equivalence rule** states that expressions of two forms are equivalent
  - One can replace an expression of first form by one of the second form, or vice versa

Equivalence Rules

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

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

Equivalence Rules

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

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

2. Selection operations are commutative.

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

Equivalence Rules

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

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

2. Selection operations are commutative.

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

3. Only the last in a sequence of projection operations is needed, the others can be omitted

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

where $L_1 \subseteq L_2 \subseteq \ldots \subseteq L_n$

Equivalence Rules

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

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

2. Selection operations are commutative.

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

3. Only the last in a sequence of projection operations is needed, the others can be omitted

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

where $L_1 \subseteq L_2 \subseteq \ldots \subseteq L_n$

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

   a. $\sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie_{\theta} E_2$

   b. $\sigma_{\theta1}(E_1 \bowtie_{\theta2} E_2) = E_1 \bowtie_{\theta1 \wedge \theta2} E_2$

Equivalence Rules (Cont.)

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

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

(but the order is important for efficiency)

Equivalence Rules (Cont.)

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

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

(but the order is important for efficiency)

6. (a) Natural join operations are associative:

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

(again, the order is important for efficiency)

Equivalence Rules (Cont.)

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

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

(but the order is important for efficiency)

6. (a) Natural join operations are associative:

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

(again, the order is important for efficiency)

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

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

where θ_1 involves attributes from only E_1 and E_2
and θ_2 involves attributes from only E_2 and E_3

Equivalence Rules (Cont.)

7. (a) Selection distributes over theta join in the following manner:

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

where θ_1 involves attributes from only E_1

(b) Complex selection distributes over theta join in the following manner:

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

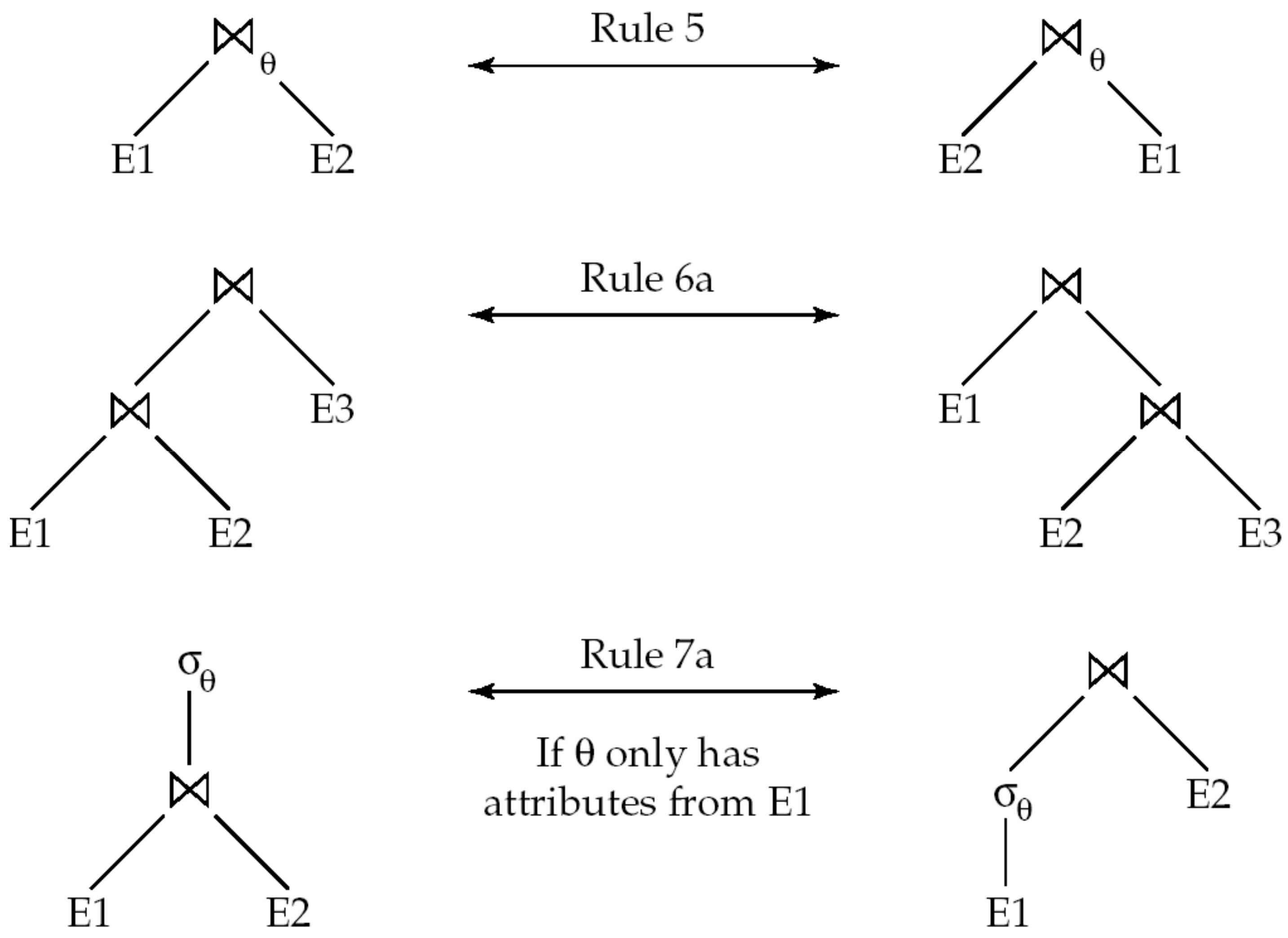
where θ_1 involves attributes from only E_1
and θ_2 involves attributes from only E_2

More equivalences at Ch. 16.2 of the book *

---

* Silberschatz, Korth, and Sudarshan, ***Database System Concepts***, **7° ed.**

Pictorial Depiction of Equivalence Rules



Exercise

- Disprove the equivalence

$$(R ⟕ S) ⟕ T = R ⟕ (S ⟕ T)$$

Exercise

- Disprove the equivalence

$$(R ⟕ S) ⟕ T = R ⟕ (S ⟕ T)$$

Definition (left outer join): the result of a left outer join T = R ⟕ S is a super-set of the result of the join T' = R ⋈ S in that all tuples in T' appear in T. In addition, T preserve those tuples that are lost in the join, by creating tuples in T that are filled with *null* values

Exercise

- Disprove the equivalence

$$(R ⟕ S) ⟕ T = R ⟕ (S ⟕ T)$$

Definition (left outer join): the result of a left outer join T = R ⟕ S is a super-set of the result of the join T' = R $\bowtie$ S in that all tuples in T' appear in T. In addition, T preserve those tuples that are lost in the join, by creating tuples in T that are filled with *null* values

STUD

stud_id	name	surname
1	gino	bianchi
2	filippo	neri
3	mario	rossi

TAKES

stud_id	course	grade
1	Math	30
2	DB	22
2	Logic	30

stud_id	name	surname	course	grade
1	gino	bianchi	Math	30
2	filippo	neri	DB	22
2	filippo	neri	Logic	30

Exercise

- Disprove the equivalence

(R ⟕ S) ⟕ T = R ⟕ (S ⟕ T)

Definition (left outer join): the result of a left outer join T = R ⟕ S is a super-set of the result of the join T' = R ⋈ S in that all tuples in T' appear in T. In addition, T preserve those tuples that are lost in the join, by creating tuples in T that are filled with *null* values

STUD

stud_id	name	surname
1	gino	bianchi
2	filippo	neri
3	mario	rossi

TAKES

stud_id	course	grade
1	Math	30
2	DB	22
2	Logic	30

STUD ⟕ TAKES

stud_id	name	surname	course	grade
1	gino	bianchi	Math	30
2	filippo	neri	DB	22
2	filippo	neri	Logic	30
3	mario	rossi	***null***	***null***

Exercise

- Disprove the equivalence

$$(R ⟕ S) ⟕ T \;=\; R ⟕ (S ⟕ T)$$

Definition (left outer join): the result of a left outer join T = R ⟕ S is a super-set of the result of the join T' = R ⋈ S in that all tuples in T' appear in T. In addition, T preserve those tuples that are lost in the join, by creating tuples in T that are filled with *null* values

STUD

stud_id	name	surname
1	gino	bianchi
2	filippo	neri
3	mario	rossi

TAKES

stud_id	course	grade
1	Math	30
2	DB	22
2	Logic	30

STUD ⟕ TAKES

stud_id	name	surname	course	grade
1	gino	bianchi	Math	30
2	filippo	neri	DB	22
2	filippo	neri	Logic	30
3	mario	rossi	***null***	***null***

TAKES ⟕ STUD ???

Exercise

- Disprove the equivalence

$$(R ⟕ S) ⟕ T = R ⟕ (S ⟕ T)$$

Definition (left outer join): the result of a left outer join T = R ⟕ S is a super-set of the result of the join T' = R ⋈ S in that all tuples in T' appear in T. In addition, T preserve those tuples that are lost in the join, by creating tuples in T that are filled with *null* values

STUD

stud_id	name	surname
1	gino	bianchi
2	filippo	neri
3	mario	rossi

TAKES

stud_id	course	grade
1	Math	30
2	DB	22
2	Logic	30

STUD ⟕ *TAKES*

stud_id	name	surname	course	grade
1	gino	bianchi	Math	30
2	filippo	neri	DB	22
2	filippo	neri	Logic	30
3	mario	rossi	***null***	***null***

TAKES ⟕ *STUD* ???

equivalent to *TAKES* ⋈ *STUD*

Solution

- Disprove the equivalence $(R ⟕ S) ⟕ T = R ⟕ (S ⟕ T)$

Solution

- Disprove the equivalence (R ⟕ S) ⟕ T = R ⟕ (S ⟕ T)

R

A	A_R
1	1

S

A	A_S
2	1

T

A	A_T
1	1

Solution

- Disprove the equivalence (R ⟕ S) ⟕ T = R ⟕ (S ⟕ T)

R

A	A_R
1	1

S

A	A_S
2	1

T

A	A_T
1	1

R ⟕ S

A	A_R	A_S
1	1	null

Solution

- Disprove the equivalence (R ⟕ S) ⟕ T = R ⟕ (S ⟕ T)

R

A	A_R
1	1

S

A	A_S
2	1

T

A	A_T
1	1

R ⟕ S

A	A_R	A_S
1	1	null

(R ⟕ S) ⟕ T

A	A_R	A_S	A_T
1	1	null	1

Solution

- Disprove the equivalence $(R ⟕ S) ⟕ T = R ⟕ (S ⟕ T)$

R

A	A_R
1	1

S

A	A_S
2	1

T

A	A_T
1	1

$R ⟕ S$

A	A_R	A_S
1	1	null

$S ⟕ T$

A	A_S	A_T
2	1	null

$(R ⟕ S) ⟕ T$

A	A_R	A_S	A_T
1	1	null	1

Solution

- Disprove the equivalence $(R ⟕ S) ⟕ T = R ⟕ (S ⟕ T)$

R

A	A_R
1	1

S

A	A_S
2	1

T

A	A_T
1	1

$R ⟕ S$

A	A_R	A_S
1	1	null

$S ⟕ T$

A	A_S	A_T
2	1	null

$(R ⟕ S) ⟕ T$

A	A_R	A_S	A_T
1	1	null	1

$R ⟕ (S ⟕ T)$

A	A_R	A_S	A_T
1	1	null	null

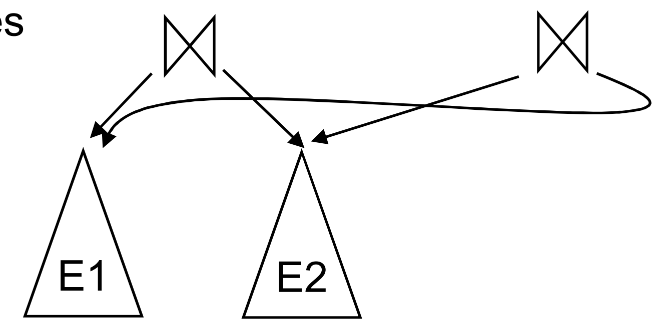
Equivalence derivability and minimality

- Some equivalence can be derived from others
  - example: 2 can be obtained from 1 (exploiting commutativity of conjunction)
    7b can be obtained from 1 and 7a
- Optimizers use **minimal** sets of equivalence rules

Enumeration of Equivalent Expressions

- Query optimizers use equivalence rules to **systematically** generate expressions equivalent to the given one
- Can generate all equivalent expressions as follows:
  - Repeat (starting from the set containing only the given expression)
    - apply all applicable equivalence rules on every sub-expression of every equivalent expression found so far
    - add newly generated expressions to the set of equivalent expressions

    Until no new equivalent expressions are generated
- The above approach is very expensive in space and time
  - Space: efficient expression-representation techniques
    - 1 copy is stored for shared sub-expressions
  - Time: partial generation
    - Dynamic programming
    - Greedy techniques (select best choices at each step)
    - Heuristics, e.g., single-relation operations (selections, projections) are pushed inside (performed earlier)



Estimating Statistics of Expression Results

- The Catalog
- Size estimation
  - Selection
  - Join
  - Other operations (projection, aggregation, set operations, outer join)
- Estimation of number of distinct values

These slides are a modified version of the slides provided with the book:

Database System Concepts, 6th Ed.

(however, chapter numeration refers to 7th Ed.)

The original version of the slides is available at: https://www.db-book.com/

Cost Estimation

- Cost of each operator computed as described in Chapter 15 *
  - Need statistics of input relations
    - E.g. number of tuples, number of blocks
- Statistics are collected in the **Catalog**
- Inputs can be results of sub-expressions
  - Need to estimate statistics of expression results
  - Estimation of size of intermediate results
    - # of tuple in input to successive operations
  - Estimation of number of distinct values in intermediate results
    - selectivity rate of successive selection operations
- Statistics are not totally accurate
  - Information in the catalog might be not always up-to-date (delay)
  - A precise estimate for intermediate results might be impossible to compute

---

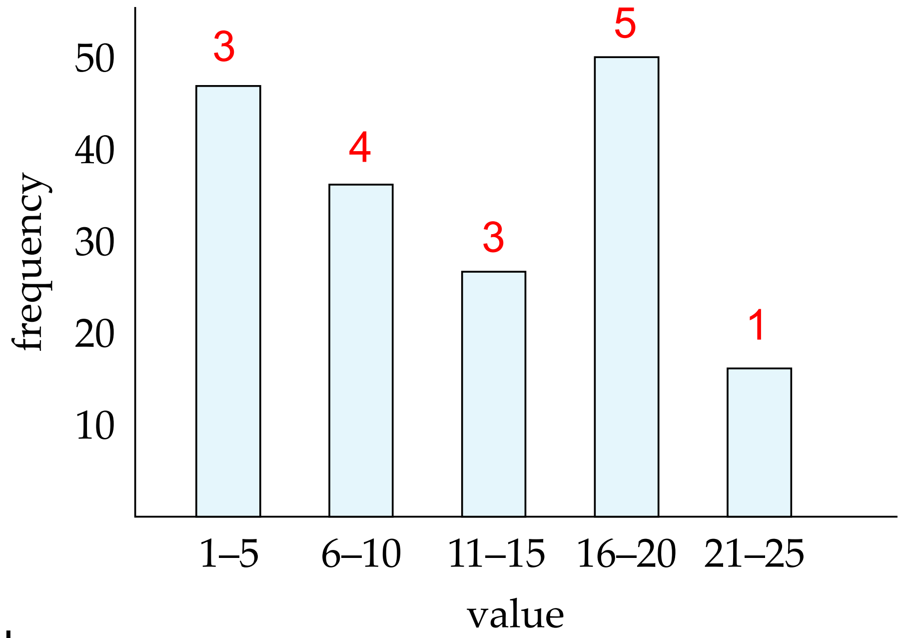
* **Silberschatz, Korth, and Sudarshan, *Database System Concepts*, 7° ed.**

Statistical Information for Cost Estimation

- Statistics information is maintained in the **Catalog**
- The catalog is itself stored in the database as relation(s)
- It contains:
  - 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* (in bytes)
  - 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 set of attributes *A*
    - $V(A, r)$ = the size of $\prod_A(r)$ – if A is a key, then $V(A,r) = n_r$
  - *min(A,r)*: smallest value appearing in relation *r* for set of attribute *A*;
  - *max(A,r)*: largest value appearing in relation *r* for set of attribute *A*;
  - statistics about indices (height of B^+-trees, number of blocks for leaves, …)
- We assume tuples of *r* are stored together physically in a file; then: $b_r = \lceil n_r / f_r \rceil$
- Information not always up-to-date
  - Catalog is not updated to every DB change (done during periods of light system load)

Histograms

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



- For each range
  - Number of records (tuples) with value in the range
  - Also, number of distinct values in the range (red numbers in the picture)
- Without histogram information, uniform distribution is assumed
- Little space occupation
  - Histograms for many attributes on many relations can be stored

Selection Size Estimation

- # of records that will satisfy the selection predicate (aka selection condition)
- $\sigma_{A=v}(r)$ *(we are assuming that v actually is present in A)*

Selection Size Estimation

- # of records that will satisfy the selection predicate (aka selection condition)
- $\sigma_{A=v}(r)$ *(we are assuming that v actually is present in A)*
  - $n_r / V(A,r)$ (no histogram, uniform distribution)
  - 1 if A is key

Selection Size Estimation

- \# of records that will satisfy the selection predicate (aka selection condition)
- $\sigma_{A=v}(r)$ *(we are assuming that v actually is present in A)*
  - $n_r / V(A,r)$ (no histogram, uniform distribution)
  - 1 if A is key
- $\sigma_{A \leq v}(r)$ *(case $\sigma_{A \geq V}(r)$ is symmetric)*
  - 0 if $v < min(A,r)$
  - n_r if $v >= max(A,r)$

Selection Size Estimation

- # of records that will satisfy the selection predicate (aka selection condition)
- $\sigma_{A=v}(r)$ *(we are assuming that v actually is present in A)*
  - $n_r / V(A,r)$ (no histogram, uniform distribution)
  - 1 if A is key
- $\sigma_{A \le v}(r)$ *(case $\sigma_{A \ge v}(r)$ is symmetric)*
  - 0 if $v < min(A,r)$
  - n_r if $v >= max(A,r)$
  - $n_r * \dfrac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$ otherwise (no histogram, uniform distribution)
  - In absence of statistical information or when *v* is unknown at time of cost estimation (e.g., *v* is computed at run-time by the application using the DB), the we assume
    - $n_r / 2$

Selection Size Estimation

- # of records that will satisfy the selection predicate (aka selection condition)
- $\sigma_{A=v}(r)$ *(we are assuming that v actually is present in A)*
  - $n_r / V(A,r)$ (no histogram, uniform distribution)
  - 1 if A is key
- $\sigma_{A \leq v}(r)$ *(case $\sigma_{A \geq V}(r)$ is symmetric)*
  - 0 if $v < min(A,r)$
  - n_r if $v >= max(A,r)$
  - $n_r * \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$ otherwise (no histogram, uniform distribution)
  - In absence of statistical information or when *v* is unknown at time of cost estimation (e.g., *v* is computed at run-time by the application using the DB), the we assume
    - $n_r / 2$
- If histograms are available, we can do more precise estimates
  - use values for restricted ranges instead of n_r , $V(A,r)$, $min(A, r)$, $max(A,r)$

Complex Selection Size Estimation

- Conjunction $E = \sigma_{\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_n} (r)$
  - we compute s_i = size selection for θ_i $(i = 1,\ldots, n)$
  - **selectivity rate** (**SR**) of $\sigma_{\theta_i}(r)$: $SR(\sigma_{\theta_i}(r)) = s_i / n_r$ $(i = 1,\ldots, n)$
  - $SR(E) = \Pi_i (SR(\sigma_{\theta_i}(r))) = s_1 / n_r * \ldots * s_n / n_r$ Π_i is multiplication with $i = 1,\ldots,n$
  - # of record for $E = n_r * SR(E) = n_r * \dfrac{s_1 * s_2 * \ldots * s_n}{(n_r)^n}$

Complex Selection Size Estimation

- Conjunction $E = \sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n} (r)$
  - we compute s_i = size selection for θ_i $(i = 1, \dots, n)$
  - **selectivity rate** (**SR**) of $\sigma_{\theta_i}(r)$: $SR(\sigma_{\theta_i}(r)) = s_i / n_r$ $(i = 1, \dots, n)$
  - $SR(E) = \Pi_i (SR(\sigma_{\theta_i}(r))) = s_1 / n_r * _{\dots} * s_n / n_r$ Π_i is multiplication with $i = 1, \dots, n$
  - # of record for $E = n_r * SR(E) = n_r * \dfrac{s_1 * s_2 * \dots * s_n}{(n_r)^n}$
- Disjunction $E = \sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n} (r) = \sigma_{\neg(\neg\theta_1 \wedge \neg\theta_2 \wedge \dots \wedge \neg\theta_n)} (r)$
  - $SR(E) = 1 - SR(\sigma_{\neg\theta_1 \wedge \neg\theta_2 \wedge \dots \wedge \neg\theta_n} (r))$
  - $SR(\sigma_{\neg\theta_1 \wedge \neg\theta_2 \wedge \dots \wedge \neg\theta_n} (r)) = (1 - s_1 / n_r) * _{\dots} * (1 - s_n / n_r)$
  - # of record for $E = n_r * SR(E) = n_r * \left[1 - (1 - \dfrac{s_1}{n_r}) * (1 - \dfrac{s_2}{n_r}) * \dots * (1 - \dfrac{s_n}{n_r}) \right]$

Complex Selection Size Estimation

- Conjunction $E = \sigma_{\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_n}(r)$
  - we compute s_i = size selection for θ_i $(i = 1, \ldots, n)$
  - **selectivity rate** (**SR**) of $\sigma_{\theta_i}(r)$: $SR(\sigma_{\theta_i}(r)) = s_i / n_r$ $(i = 1, \ldots, n)$
  - $SR(E) = \Pi_i (SR(\sigma_{\theta_i}(r))) = s_1 / n_r * \ldots * s_n / n_r$ Π_i is multiplication with $i = 1, \ldots, n$
  - # of record for $E = n_r * SR(E) = n_r * \frac{s_1 * s_2 * \ldots * s_n}{(n_r)^n}$
- Disjunction $E = \sigma_{\theta_1 \vee \theta_2 \vee \ldots \vee \theta_n}(r) = \sigma_{\neg(\neg\theta_1 \wedge \neg\theta_2 \wedge \ldots \wedge \neg\theta_n)}(r)$
  - $SR(E) = 1 - SR(\sigma_{\neg\theta_1 \wedge \neg\theta_2 \wedge \ldots \wedge \neg\theta_n}(r))$
  - $SR(\sigma_{\neg\theta_1 \wedge \neg\theta_2 \wedge \ldots \wedge \neg\theta_n}(r)) = (1 - s_1 / n_r) * \ldots * (1 - s_n / n_r)$
  - # of record for $E = n_r * SR(E) = n_r * \left[1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \ldots * (1 - \frac{s_n}{n_r}) \right]$
- Negation $E = \sigma_{\neg\theta}(r)$
  - # of record for $E = n_r$ - # of record for $\sigma_\theta(r)$

Join Size Estimation

- # of records that will be included in the result

Join Size Estimation

- # of records that will be included in the result
- *(cartesian product) r* x *s:* *# of records =* $n_r * n_s$

Join Size Estimation

- \# of records that will be included in the result
  - *(cartesian product) r* x *s:* *# of records =* $n_r * n_s$
  - *(natural join on attribute A) r* ⋈ *s:*

Join Size Estimation

- # of records that will be included in the result
- *(cartesian product) r* x *s:* # of records = $n_r * n_s$
- *(natural join on attribute A) r ⋈ s:*
  - for each tuple t_r of *r* there are in average n_s / *V(A,s)* many tuples of *s* selected

Join Size Estimation

- # of records that will be included in the result
- *(cartesian product) r* x *s:* *# of records =* $n_r * n_s$
- *(natural join on attribute A) r* ⋈ *s:*
  - for each tuple t_r of *r* there are in average $n_s / V(A,s)$ many tuples of *s* selected
  - thus, ***# of records =*** $n_r * n_s / V(A,s)$

Join Size Estimation

- \# of records that will be included in the result
- *(cartesian product) r* x *s:* *# of records =* $n_r * n_s$
- *(natural join on attribute A) r* ⋈ *s:*
  - for each tuple t_r of *r* there are in average $n_s / V(A,s)$ many tuples of *s* selected
  - thus, ***# of records =*** $n_r * n_s / V(A,s)$
  - by switching the role of *r* and *s* we get ***# of records =*** $n_r * n_s / V(A,r)$

Join Size Estimation

- # of records that will be included in the result
- *(cartesian product) r* x *s:* $\#\ of\ records = n_r * n_s$
- *(natural join on attribute A) r ⋈ s:*
  - for each tuple t_r of *r* there are in average $n_s / V(A,s)$ many tuples of *s* selected
  - thus, $\#\ of\ records = n_r * n_s / V(A,s)$
  - by switching the role of *r* and *s* we get $\#\ of\ records = n_r * n_s / V(A,r)$
  - lowest is more accurate estimation $\#\ of\ records = n_r * n_s / max\{ V(A,r), V(A,s) \}$

Join Size Estimation

- # of records that will be included in the result
- *(cartesian product) r* x *s:* # of records = $n_r * n_s$
- *(natural join on attribute A) r* ⋈ *s:*
  - for each tuple t_r of *r* there are in average $n_s / V(A,s)$ many tuples of *s* selected
  - thus, **# of records = $n_r * n_s / V(A,s)$**
  - by switching the role of *r* and *s* we get **# of records = $n_r * n_s / V(A,r)$**
  - lowest is more accurate estimation # of records = $n_r * n_s / max\{ V(A,r), V(A,s) \}$
  - histograms can be used for more accurate estimations
    - histograms must be on join attributes, for both relations, and with same ranges
    - use values for each range of the histogram, instead of n_r , n_s , $V(A,r)$, $V(A,s)$, and then sum estimations obtained for all ranges

Join Size Estimation

- # of records that will be included in the result
- *(cartesian product) r* x *s:* *# of records =* $n_r * n_s$
- *(natural join on attribute A) r* ⋈ *s:*
  - for each tuple t_r of *r* there are in average $n_s / V(A,s)$ many tuples of *s* selected
  - thus, **# of records =** $\boldsymbol{n_r * n_s / V(A,s)}$
  - by switching the role of *r* and *s* we get **# of records =** $\boldsymbol{n_r * n_s / V(A,r)}$
  - lowest is more accurate estimation *# of records =* $n_r * n_s / \max\{ V(A,r), V(A,s) \}$
  - histograms can be used for more accurate estimations
    - histograms must be on join attributes, for both relations, and with same ranges
    - use values for each range of the histogram, instead of n_r , n_s , $V(A,r)$, $V(A,s)$, and then sum estimations obtained for all ranges
  - if *A* is key for *r*, then *# of records <=* n_s (and vice versa)
    - in addition, if *A* is not null in *s*, then *# of records =* n_s (and vice versa)

Join Size Estimation

- # of records that will be included in the result
- *(cartesian product) r* x *s:* $\quad$ *# of records =* $n_r * n_s$
- *(natural join on attribute A) r* ⋈ *s:*
  - for each tuple t_r of *r* there are in average $n_s / V(A,s)$ many tuples of *s* selected
  - thus, $\quad$ ***# of records =*** $\boldsymbol{n_r * n_s / V(A,s)}$
  - by switching the role of *r* and *s* we get $\quad$ ***# of records =*** $\boldsymbol{n_r * n_s / V(A,r)}$
  - lowest is more accurate estimation $\quad$ *# of records =* $n_r * n_s / max\{ V(A,r), V(A,s) \}$
  - histograms can be used for more accurate estimations
    - histograms must be on join attributes, for both relations, and with same ranges
    - use values for each range of the histogram, instead of n_r , n_s , $V(A,r)$, $V(A,s)$, and then sum estimations obtained for all ranges
  - if *A* is key for *r*, then $\quad$ *# of records <=* n_s $\quad$ (and vice versa)
    - in addition, if *A* is not null in *s*, then $\quad$ *# of records =* n_s $\quad$ (and vice versa)
- *(theta join)* $r \bowtie_\theta s$

Join Size Estimation

- # of records that will be included in the result
- *(cartesian product)* r x s: — *# of records =* $n_r * n_s$
- *(natural join on attribute A)* $r \bowtie s$:
  - for each tuple t_r of r there are in average $n_s / V(A,s)$ many tuples of s selected
  - thus, — ***# of records =*** $n_r * n_s / V(A,s)$
  - by switching the role of r and s we get — ***# of records =*** $n_r * n_s / V(A,r)$
  - lowest is more accurate estimation — *# of records =* $n_r * n_s / max\{ V(A,r), V(A,s) \}$
  - histograms can be used for more accurate estimations
    - histograms must be on join attributes, for both relations, and with same ranges
    - use values for each range of the histogram, instead of n_r , n_s , $V(A,r)$, $V(A,s)$, and then sum estimations obtained for all ranges
  - if A is key for r, then — *# of records <=* n_s — (and vice versa)
    - in addition, if A is not null in s, then — *# of records =* n_s — (and vice versa)
- *(theta join)* $r \bowtie_\theta s$
  - $r \bowtie_\theta s = \sigma_\theta (r \times s)$ — use formulas for cartesian product and selection

Size Estimation for Other Operations

- projection (no duplications): *# of records = V(A,r)*
- aggregation ${}_G\gamma_F\ (r)$ *# of records = V(G,r)*
- set operations
  - between selections on same relation: use formulas for selection
    - es.: $\sigma_{\theta_1}(r) \cup \sigma_{\theta_2}(r) = \sigma_{\theta_1 \vee \theta_2}(r)$
  - $r \cup s$: *# of records =* $n_r + n_s$
  - $r \cap s$: *# of records =* $\min\{ n_r, n_s \}$
  - $r - s$: *# of records =* n_r
- outer join
  - left outer join: *# of records = # of records for inner join +* n_r
  - right outer join: *# of records = # of records for inner join +* n_s
  - full outer join: *# of records = # of records for inner join +* $n_r + n_s$

Estimation for Number of Distinct Values

- # distinct values in the result for expression *E* and attribute (or set of attributes) *A*: ***V(A,E)***
- Selection $E = \sigma_{\theta}(r)$
  - *V(A, E)* is a specific value for some conditions
    - e.g., if condition θ is ***A=3*** , then *V(A, E) = 1*
    - e.g., if condition θ is ***3 < A <= 6***, then *V(A, E) = 3* (assuming domain of A is the integers)
  - condition *A* < *v* (or *A* > *v*, *A* >= *v*, …) *V(A,E) = V(A,r) * selectivity rate of the selection*
  - otherwise $V(A,E) = min \{ n_E , V(A,r) \}$
- Join $E = r \bowtie s$
  - *A* only contains attributes from *r* $V(A,E) = min \{ n_E , V(A,r) \}$
  - *A* only contains attributes from *s* $V(A,E) = min \{ n_E , V(A,s) \}$
  - *A* contains attributes *A1* from *r* and attributes *A2* from *s*

$$V(A,E) = min \{ n_E , V(A1, r) * V(A2 – A1, s) , V(A2, s) * V(A1 – A2, r) \}$$

Choice of Evaluation Plans

- Dynamic Programming for Choosing Evaluation Plans

These slides are a modified version of the slides provided with the book:

Database System Concepts, 6th Ed.

(however, chapter numeration refers to 7th Ed.)

The original version of the slides is available at: https://www.db-book.com/

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

- A big part of a cost-based optimizer (based on equivalence rules) is choosing the “best” order for join operations
- Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$.
- 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 17.6 billion!
- No need to generate all the join orders. Exploiting some monotonicity (optimal substructure property), the least-cost join order for any subset of $\{r_1, r_2, \ldots, r_n\}$ is computed only once.

Cost-Based Optimization: An example

- Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4 \bowtie r_5$
- Number of possible different join orderings: $\frac{(2(n-1))!}{(n-1)!} = \frac{8!}{4!} = 1680$
- The least-cost join order for any subset of $\{ r_1, r_2, r_3, r_4, r_5 \}$ is computed only once
- Assume we want to compute $N_{123/45}$: number of possible different join orderings where r_1, r_2, r_3 sare grouped together, e.g.,

$(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$ $(r_2 \bowtie r_3 \bowtie r_1) \bowtie (r_5 \bowtie r_4)$ $r_4 \bowtie (r_5 \bowtie (r_1 \bowtie (r_2 \bowtie r_3)))$ …

- The naïve approach
  - $N_{123/45} = N_{123} * N_{45}$
  - $N_{123} = \frac{4!}{2!} = 12$ (N_{123} : # ways of arranging r_1, r_2, and r_3)
  - $N_{45} = N_{123} = 12$ (N_{45} : # ways of arranging r_4 and r_5 wrt. block of r_1, r_2, and r_3)
  - $N_{123/45} = 12 * 12 = 144$
- Exploiting optimal substructure property:
  - compute only once best ordering for $r_1 \bowtie r_2 \bowtie r_3$: 12 possibilities (N_{123})
  - compute best ordering for $R_{123} \bowtie r_4 \bowtie r_5$: 12 possibilities (N_{45})
  - Therefore, $N_{123/45} = 12 + 12 = 24$

Dynamic Programming in Optimization

- To find best join tree (equivalently, best join order) for a set of *n* relations:
  - Consider all possible plans of the form:

    $$S' \bowtie (S \setminus S')$$

    for every non-empty subset *S'* of *S*
  - Recursively compute (and store) costs of best join orders for subsets *S'* and *S* \ *S'*. Choose the cheapest of the $2^n - 2$ alternatives
  - Base case for recursion: find best algorithm for scanning relation
  - When a plan for a subset is computed, store it and reuse it when it is required again, instead of re-computing it
    - Dynamic programming

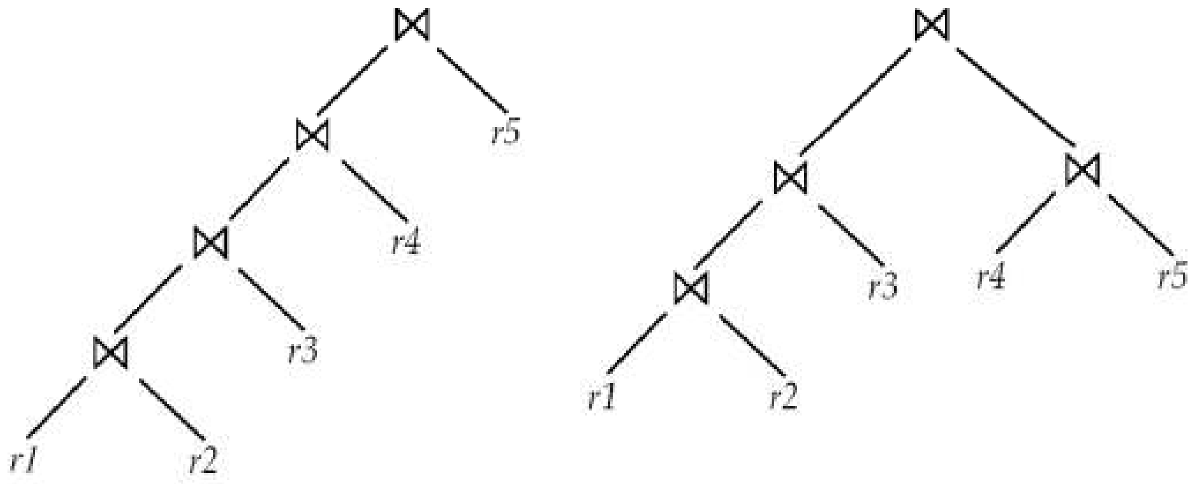
Join Order Optimization Algorithm

```
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 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 + cost of A
        if cost < bestplan[S].cost
            bestplan[S].cost = cost
            bestplan[S].plan = "execute P1.plan; execute P2.plan;
                                join results of P1 and P2 using A"
    return bestplan[S]
```

* This is the algorithm shown in the 6th edition of the textbook.
It is slightly different from the algorithm we presented during our class, especially the way the base case is handled.

Cost of Optimization

- With dynamic programming time complexity of optimization is $O(3^n)$.
  - With n = 10, this number is 59000 instead of 17.6 billion!
- Space complexity is $O(2^n)$
- Better time performance when considering only left-deep join tree $O(n\ 2^n)$
  Space complexity remains at $O(2^n)$ (heuristic approach)



(a) Left-deep join tree

(b) Non-left-deep join tree

- Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10)

Cost Based Optimization with Equivalence Rules

- **Physical equivalence rules** equates logical operations (e.g., join) to physical ones (i.e., implementations – e.g., nested-loop join, merge join)
  - Relational algebra expression are converted into QEP with implementation details
- Efficient optimizer based on equivalence rules depends on
  - A space efficient representation of expressions which avoids making multiple copies of sub-expressions
  - Efficient techniques for detecting duplicate derivations of expressions
  - Dynamic programming or memoization techniques, which store the "best" plan for a sub-expression the first time it is computed, and reuses in on repeated optimization calls on same sub-expression
  - Cost-based pruning techniques that avoid generating all plans (greedy, heuristics)

Heuristic Optimization

- Cost-based optimization is expensive, even with dynamic programming
- Systems may use *heuristics* to reduce the number of possibilities choices that must be considered
- 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
  - Only consider left-deep join orders (particularly suited for pipelining as only one input has to be pipelined, the other is a relation)

Structure of Query Optimizers

- Some systems use only heuristics, others combine heuristics with partial cost-based optimization.
- Many optimizers considers only left-deep join orders.
  - Plus heuristics to push selections and projections down the query tree
  - Reduces optimization complexity and generates plans amenable to pipelined evaluation.
- Heuristic optimization used in some versions of Oracle:
  - Repeatedly pick "best" relation to join next
    - it obtains and compares *n* plans (each starting with one relation) In each plan, pick the best next relation for the join

End of Chapter

These slides are a modified version of the slides provided with the book:

Database System Concepts, 6th Ed.

(however, chapter numeration refers to 7th Ed.)

The original version of the slides is available at: https://www.db-book.com/