Dr. Vishwanath Karad
MIT WORLD PEACE UNIVERSITY | PUNE
TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

MIT-WPU
॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

CS312
Database Management Systems

School of Computer Engineering and Technology

CS312 Database Management Systems

Course Objectives:

1) Understand and successfully apply logical database design principles, including E-R diagrams and database normalization.
2) Learn Database Programming languages and apply in DBMS application
3) Understand transaction processing and concurrency control in DBMS
4) Learn database architectures, DBMS advancements and its usage in advance application

Course Outcomes:

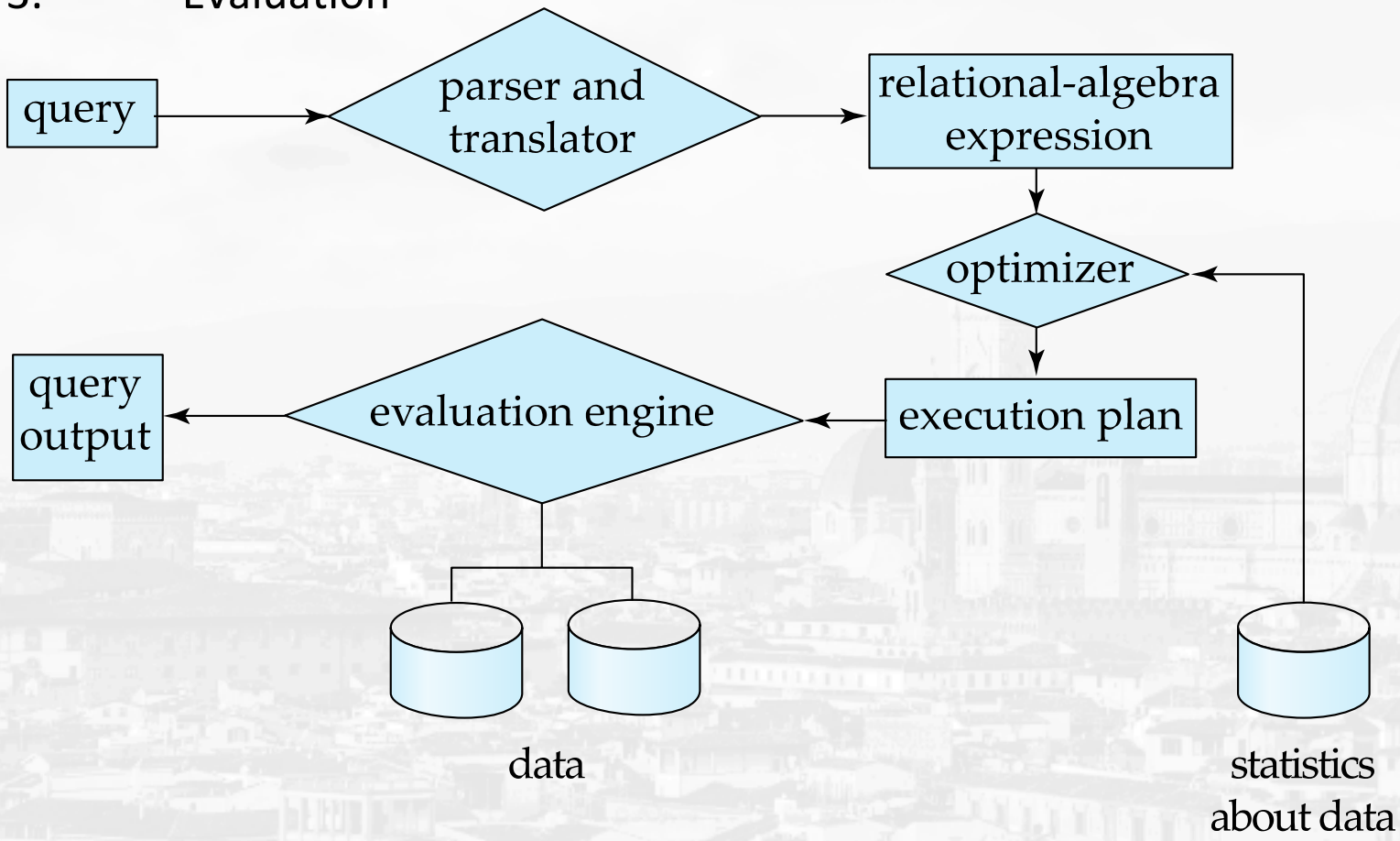
5) Design ER-models to represent simple database application scenarios and Improve the database design by normalization.
6) Design Database Relational Model and apply SQL , PLSQL concepts for database programming
7) Describe Transaction Processing and Concurrency Control techniques for databases
8) Identify appropriate database architecture for the real world database application

Query Processing

Overview
Measures of Query Cost
Selection Operation
Sorting
Join Operation
Evaluation of Expressions

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Basic Steps in Query Processing (Cont.)

Parsing and translation

translate the query into its internal form. This is then translated into relational algebra.

Parser checks syntax, verifies relations

Evaluation

The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

Basic Steps in Query Processing : Optimization

A relational algebra expression may have many equivalent expressions

E.g., $\sigma_{salary<75000}(\prod_{salary}(instructor))$ is equivalent to $\prod_{salary}(\sigma_{salary<75000}(instructor))$

Each relational algebra operation can be evaluated using one of several different algorithms

Correspondingly, a relational-algebra expression can be evaluated in many ways.

Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.

E.g., can use an index on *salary* to find instructors with salary < 75000,

or can perform complete relation scan and discard instructors with salary $\geq$ 75000

Basic Steps: Optimization (Cont.)

Query Optimization: Amongst all equivalent evaluation plans choose the one with lowest cost

Cost is estimated using statistical information from the database catalog

e.g. number of tuples in each relation, size of tuples, etc.

In this chapter we study

How to measure query costs

Algorithms for evaluating relational algebra operations

How to combine algorithms for individual operations in order to evaluate a complete expr

In Chapter 14

We study how to optimize queries, that is, how to find an evaluation plan with lowest esti cost

Measures of Query Cost

Cost is generally measured as total elapsed time for answering query

- Many factors contribute to time cost
  - *disk accesses, CPU*, or even network *communication*

Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account

- Number of seeks * average-seek-cost
- Number of blocks read * average-block-read-cost
- Number of blocks written * average-block-write-cost
  - Cost to write a block is greater than cost to read a block
    - data is read back after being written to ensure that the write was successful

Measures of Query Cost (Cont.)

For simplicity we just use the **number of block transfers** *from disk and the* **number of seeks** as the cost measures

- t_T – time to transfer one block
- t_S – time for one seek
- Cost for b block transfers plus S seeks

$$b * t_T + S * t_S$$

We ignore CPU costs for simplicity

- Real systems do take CPU cost into account

We do not include cost to writing output to disk in our cost formulae

Measures of Query Cost (Cont.)

Several algorithms can reduce disk IO by using extra buffer space

- Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution
  - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available

Required data may be buffer resident already, avoiding disk I/O

- But hard to take into account for cost estimation

Selection Operation

File scan

Algorithm **A1** (linear search). Scan each file block and test all records to see whether they satisfy the selection condition.

- Cost estimate = b_r block transfers + 1 seek
  - b_r denotes number of blocks containing records from relation r
- If selection is on a key attribute, can stop on finding record
  - cost = (b_r/2) block transfers + 1 seek
- Linear search can be applied regardless of
  - selection condition or
  - ordering of records in the file, or
  - availability of indices

Note: binary search generally does not make sense since data is not stored consecutively

- except when there is an index available,
- and binary search requires more seeks than index search

Selections Using Indices

Index scan – search algorithms that use an index

selection condition must be on search-key of index.

A2 (**primary index, equality on key**). Retrieve a single record that satisfies the corresponding equality condition

$Cost = (h_i + 1) * (t_T + t_S)$

A3 (**primary index, equality on nonkey**) Retrieve multiple records.

Records will be on consecutive blocks

Let b = number of blocks containing matching records

$Cost = h_i * (t_T + t_S) + t_S + t_T * b$

Selections Using Indices

A4 (secondary index, equality on nonkey).

Retrieve a single record if the search-key is a candidate key

$Cost = (h_i + 1) * (t_T + t_S)$

Retrieve multiple records if search-key is not a candidate key

each of *n* matching records may be on a different block

Cost = $(h_i + n) * (t_T + t_S)$

Can be very expensive!

Selections Involving Comparisons

Can implement selections of the form $\sigma_{A \le V}(r)$ or $\sigma_{A \ge V}(r)$ by using
a linear file scan,
or by using indices in the following ways:

A5 (**primary index, comparison**). (Relation is sorted on A)

For $\sigma_{A \ge V}(r)$ use index to find first tuple $\ge v$ and scan relation sequentially from there

For $\sigma_{A \le V}(r)$ just scan relation sequentially till first tuple $> v$; do not use index

A6 (**secondary index, comparison**).

For $\sigma_{A \ge V}(r)$ use index to find first index entry $\ge v$ and scan index sequentially from there, to find pointers to records.

For $\sigma_{A \le V}(r)$ just scan leaf pages of index finding pointers to records, till first entry $> v$

In either case, retrieve records that are pointed to

requires an I/O for each record

Linear file scan may be cheaper

Implementation of Complex Selections

Conjunction: $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \theta n}(r)$

A7 (**conjunctive selection using one index**).

Select a combination of θ_i and algorithms A1 through A7 that results in the least cost for $\sigma_{\theta i}(r)$.

Test other conditions on tuple after fetching it into memory buffer.

A8 (**conjunctive selection using composite index**).

Use appropriate composite (multiple-key) index if available.

A9 (**conjunctive selection by intersection of identifiers**).

Requires indices with record pointers.

Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers.

Then fetch records from file

If some conditions do not have appropriate indices, apply test in memory.

Algorithms for Complex Selections

Disjunction: $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \theta n}(r)$.

A10 (**disjunctive selection by union of identifiers**).

Applicable if *all* conditions have available indices.

Otherwise use linear scan.

Use corresponding index for each condition, and take union of all the obtained sets of record pointers.

Then fetch records from file

Negation: $\sigma_{\neg\theta}(r)$

Use linear scan on file

If very few records satisfy $\neg\theta$, and an index is applicable to θ

Find satisfying records using index and fetch from file

Join Operation

Several different algorithms to implement joins

- Nested-loop join
- Block nested-loop join
- Indexed nested-loop join
- Merge-join
- Hash-join

Choice based on cost estimate

Examples use the following information

- Number of records of *student*: 5,000 *takes*: 10,000
- Number of blocks of *student*: 100 *takes*: 400

Nested-Loop Join

To compute the theta join $r \bowtie_{\theta} s$

for each tuple t_r **in** r **do begin**

  for each tuple t_s **in** s **do begin**

    test pair (t_r, t_s) to see if they satisfy the join condition θ

    if they do, add $t_r \bullet t_s$ to the result.

  end

end

r is called the **outer relation** and s the **inner relation** of the join.

Requires no indices and can be used with any kind of join condition.

Expensive since it examines every pair of tuples in the two relations.

Nested-Loop Join (Cont.)

In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is

$n_r * b_s + b_r$ block transfers, plus

$n_r + b_r$ seeks

If the smaller relation fits entirely in memory, use that as the inner relation.

Reduces cost to $b_r + b_s$ block transfers and 2 seeks

Assuming worst case memory availability cost estimate is

with *student* as outer relation:

5000 * 400 + 100 = 2,000,100 block transfers,

5000 + 100 = 5100 seeks

with *takes* as the outer relation

10000 * 100 + 400 = 1,000,400 block transfers and 10,400 seeks

If smaller relation (*student*) fits entirely in memory, the cost estimate will be 500 block transfers.

Block nested-loops algorithm (next slide) is preferable.

Example of Nested-Loop Join Costs

Compute *student* ⋈ *takes,* with *student* as the outer relation.

Let *takes* have a primary B^+-tree index on the attribute *ID,* which contains 20 entries in each index node.

Since *takes* has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data

student has 5000 tuples

Cost of block nested loops join

400*100 + 100 = 40,100 block transfers + 2 * 100 = 200 seeks

assuming worst case memory

may be significantly less with more memory

Cost of indexed nested loops join

100 + 5000 * 5 = 25,100 block transfers and seeks.

CPU cost likely to be less than that for block nested loops join

Evaluation of Expressions

So far: we have seen algorithms for individual operations

Alternatives for evaluating an entire expression tree

- **Materialization**: generate results of an expression whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat.
- **Pipelining**: pass on tuples to parent operations even as an operation is being executed

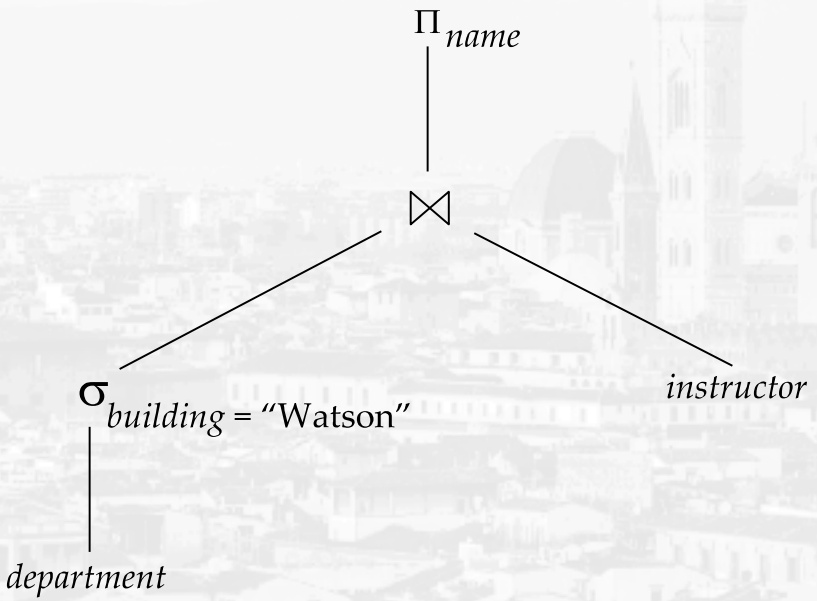
We study above alternatives in more detail

Materialization

Materialized evaluation: evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.

E.g., in figure below, compute and store $\sigma_{building="Watson"}(department)$

then compute the store its join with *instructor,* and finally compute the projection on *name.*



Materialization (Cont.)

Materialized evaluation is always applicable

Cost of writing results to disk and reading them back can be quite high

Our cost formulas for operations ignore cost of writing results to disk, so

Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk

Double buffering: use two output buffers for each operation, when one is full write it to disk while the other is getting filled

Allows overlap of disk writes with computation and reduces execution time

Pipelining

Pipelined evaluation : evaluate several operations simultaneously, passing the results of one operation on to the next.

E.g., in previous expression tree, don't store result of

$$\sigma_{building="\text{Watson}"}(department)$$

instead, pass tuples directly to the join.. Similarly, don't store result of join, pass tuples directly to projection.

Much cheaper than materialization: no need to store a temporary relation to disk.

Pipelining may not always be possible – e.g., sort, hash-join.

For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation.

Pipelines can be executed in two ways: **demand driven** and **producer driven**

Pipelining (Cont.)

In **demand driven** or **lazy** evaluation

- system repeatedly requests next tuple from top level operation
- Each operation requests next tuple from children operations as required, in order to output its next tuple
- In between calls, operation has to maintain "**state**" so it knows what to return next

In **producer-driven** or **eager** pipelining

- Operators produce tuples eagerly and pass them up to their parents
  - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
  - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
- System schedules operations that have space in output buffer and can process more input tuples

Alternative name: **pull** and **push** models of pipelining

Pipelining (Cont.)

Implementation of demand-driven pipelining

Each operation is implemented as an **iterator** implementing the following operations

open()

E.g. file scan: initialize file scan

state: pointer to beginning of file

E.g.merge join: sort relations;

state: pointers to beginning of sorted relations

next()

E.g. for file scan: Output next tuple, and advance and store file pointer

E.g. for merge join: continue with merge from earlier state till next output tuple is found. Save pointers as iterator state.

close()

Evaluation Algorithms for Pipelining

Some algorithms are not able to output results even as they get input tuples

- E.g. merge join, or hash join
- intermediate results written to disk and then read back

Algorithm variants to generate (at least some) results on the fly, as input tuples are read in

- E.g. hybrid hash join generates output tuples even as probe relation tuples in the in-memory partition (partition 0) are read in
- **Double-pipelined join technique**: Hybrid hash join, modified to buffer partition 0 tuples of both relations in-memory, reading them as they become available, and output results of any matches between partition 0 tuples
  - When a new r_0 tuple is found, match it with existing s_0 tuples, output matches, and save it in r_0
  - Symmetrically for s_0 tuples

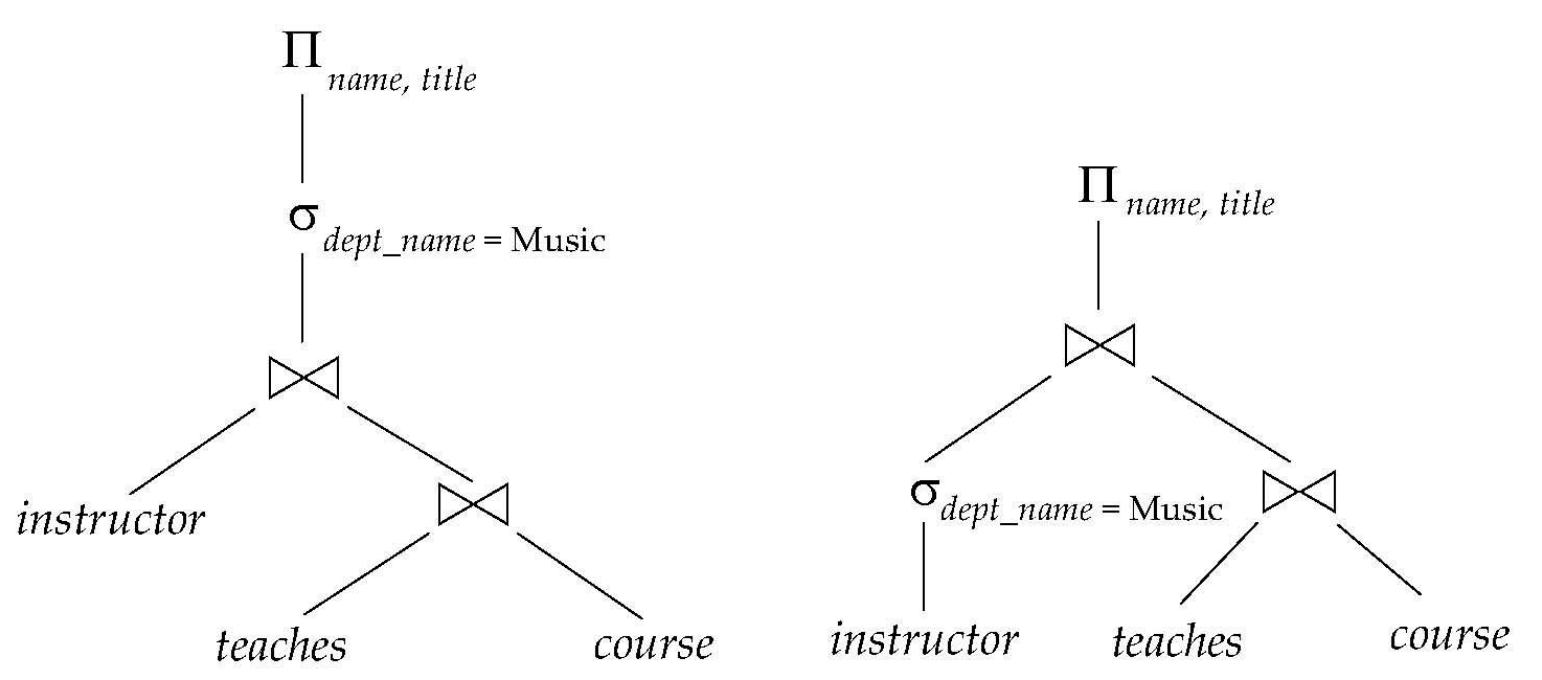
MIT-WPU
॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

Introduction to Query optimization

Introduction

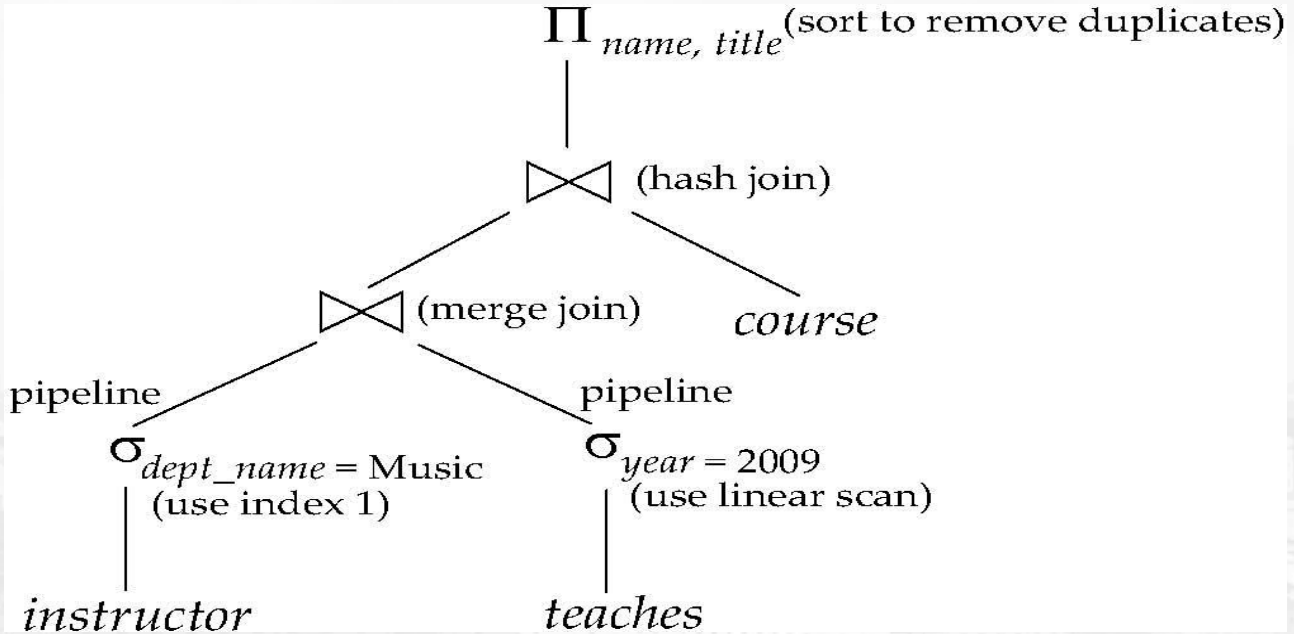
Alternative ways of evaluating a given query

- Equivalent expressions
- Different algorithms for each operation



Introduction (Cont.)

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



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

Introduction (Cont.)

Cost difference between evaluation plans for a query can be enormous

E.g. seconds vs. days in some cases

Steps in **cost-based query optimization**

1. Generate logically equivalent expressions using **equivalence rules**
2. Annotate resultant expressions to get alternative query plans
3. Choose the cheapest plan based on **estimated cost**

Estimation of plan cost based on:

Statistical information about relations. Examples:

number of tuples, number of distinct values for an attribute

Statistics estimation for intermediate results

to compute cost of complex expressions

Cost formulae for algorithms, computed using statistics

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
- we don't care if they generate different results on databases that violate integrity constraints

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.

An **equivalence rule** says that expressions of two forms are equivalent

- Can replace expression of first form by second, or vice versa

Equivalence Rules

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

$$\sigma_{q_1 \wedge q_2}(E) = \sigma_{q_1}(\sigma_{q_2}(E))$$

2. Selection operations are commutative.

$$\sigma_{q_1}(\sigma_{q_2}(E)) = \sigma_{q_2}(\sigma_{q_1}(E))$$

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

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

Equivalence Rules

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

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

6. (a) Natural join operations are associative:

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

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

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

where θ_2 involves attributes from only E_2 and E_3.

Equivalence Rules (Cont.)

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

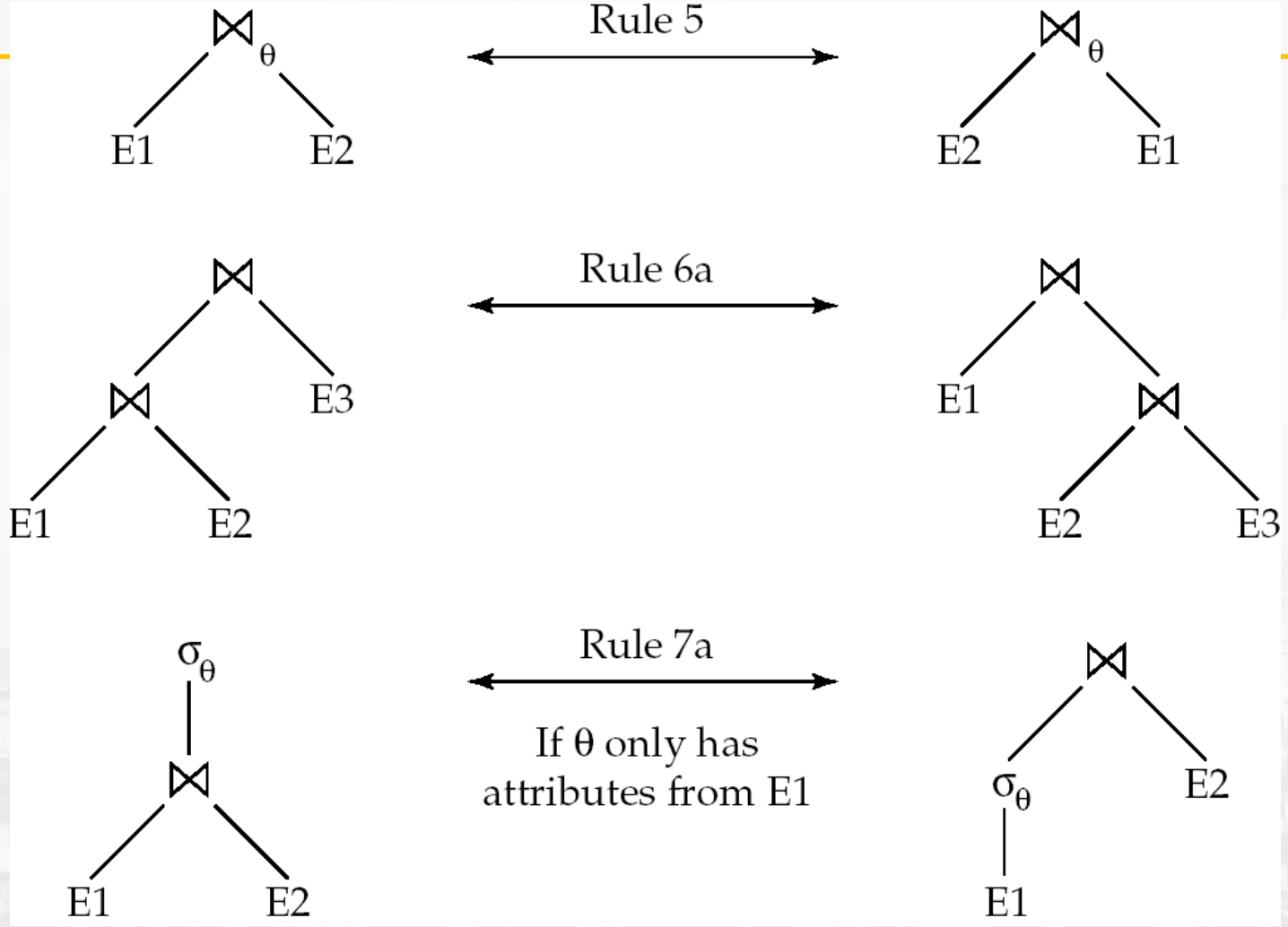
(a) When all the attributes in θ_0 involve only the attributes of one of the expressions (E_1) being joined.

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

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

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

Pictorial Depiction of Equivalence Rules



End