Querying Data on the Web

Alvaro A A Fernandes
School of Computer Science
University of Manchester

COMP62421:: 2015-2016

Part II

The Relational/SQL Setting [2]

Teaching Day 2 Outline

1. Introduction
2. Relational Query Processing [1]
   - Query Processing: An Overview
   - Query Processing: An Example
   - Query Processing: Rewriting, Costing, Evaluating
3. Relational Query Processing [2]
   - Logical Optimization: Algebraic Equivalences
   - Logical Optimization: A Heuristic Strategy
   - Logical Optimization: An Example
4. Relational Query Processing [3]
   - Algorithmic Strategies
   - Query Evaluation Strategies
   - Physical Operators
5. Relational Query Processing [4]
   - Cost-Based Plan Selection
   - Generating and Ranking Plans
   - Selecting Join Orders
   - Choosing Physical Operators
6. Relational Query Processing [5]
   - Parallel Query Processing: Data Partitioning
   - Parallelizing Relational Algebraic Operators
7. Conclusion

Last Week: Topics

- We revised the basic notions of
  - what a **database management system** (DBMS) is
  - the role that is played by the languages a DBMS supports
  - how DBMS-centred applications are designed
- We adopted an approach to describing the internal architecture of DBMSs that allowed us to discuss
  - the strengths and weaknesses of classical DBMSs
  - the recent trends in the way organizations use DBMS
  - how architectures have been diversifying recently
- We revised the various kinds of query languages by looking into
  - the (tuple) relational calculus
  - a relational algebra
  - SQL

Last Week: Learning Objectives

- We aimed to revise and reinforce undergraduate-level understanding of DBMS as software systems, rather than as software development components.
- We aimed to get under way in our postgraduate-level exploration of query languages, in preparation for further delving into query processing techniques this week.

This Week: Topics

- We look in detail into logical optimization, i.e., heuristic-based rewriting of algebraic plans.
- We explore some of the basic strategies for designing physical operators and for combining them into query evaluation plans (QEPs).
- We briefly discuss some of the challenges in cost-based QEP selection, in particular the hard problem of choosing a join order.
- We find out how data partitioning strategies, allied with the algebraic properties of query languages, make it possible to automatically parallelize QEP execution.

This Week: Learning Objectives

- We aim to complete our exploration of classical query processing.
- We aim to be ready to use what we learn on our way to consider querying data on the Web, in the narrower sense, starting with XML/XQuery.

Lecture 6

Relational Query Processing [1]

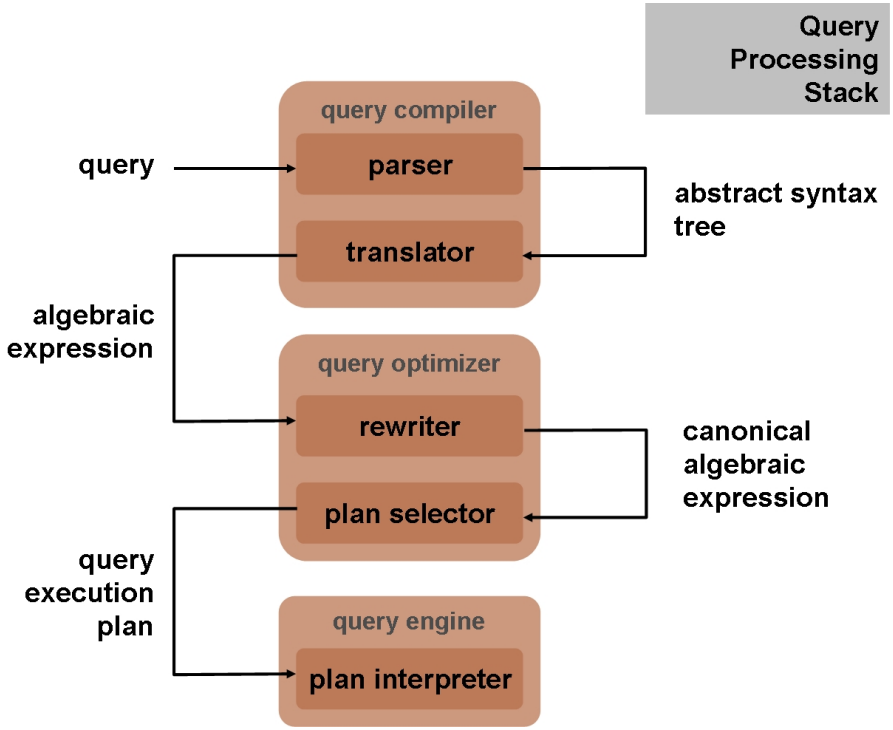
Section 1

Query Processing: An Overview

Overview of Query Processing (1)

The Query Processing Stack

1. **Parse** the declarative query
2. **Translate** to obtain an algebraic expression
3. **Rewrite** into a canonical, heuristically-efficient logical query execution plan (QEP)
4. **Select** the algorithms and access methods to obtain a quasi-optimal, costed concrete QEP
5. **Execute** (the typically interpretable form of) the procedural QEP



Overview of Query Processing (2)

Example Relations

- Let Flights and UsedFor be example relations.
- Flights asserts which flight number departs from where to where and its departure and arrival times.
- UsedFor asserts which plane is used for each flight on which weekday.
- Usable asserts which plane type (e.g., a 767) can be used for which flight.
- Certified asserts which pilot can fly which plane type.
- Their schemas are shown below.
- Underlined sets of fields denote a key.

Example (Schemas)

Flights (fltno: string, from: string, to: string, dep: date, arr: date)
UsedFor
(planeid:string, fltid: string, weekday: string)
Usable (flid:string, pltype: string)
Certified (pilid:string, planetype: string)

Section 2

Query Processing: An Example

Overview of Query Processing (3)

Example SQL Query and Corresponding Translator Output

Example (SQL)

```
SELECT   U.planeid, F.from
FROM     Flights F, UsedFor U
WHERE    F.fltno = U.fltid
```

Example (Algebraic Expression)

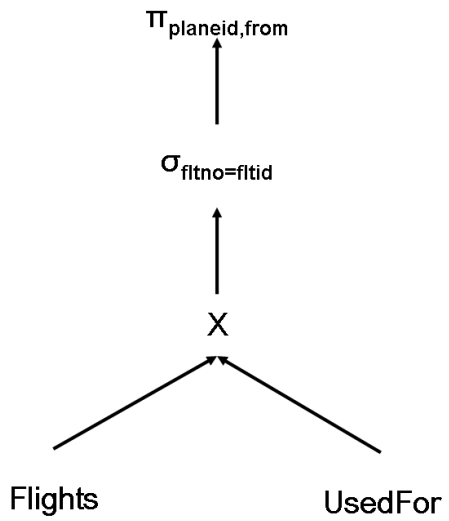
$\pi_{U.planeid,F.from}(\sigma_{F.fltno=U.fltid}(Flights \times UsedFor))$

Overview of Query Processing (4)

Algebraic Expression and Corresponding Algebraic Operator Tree

Example (Algebraic Expression)

$\pi_{U.planeid,F.from}(\sigma_{F.fltno=U.fltid}(Flights \times UsedFor))$

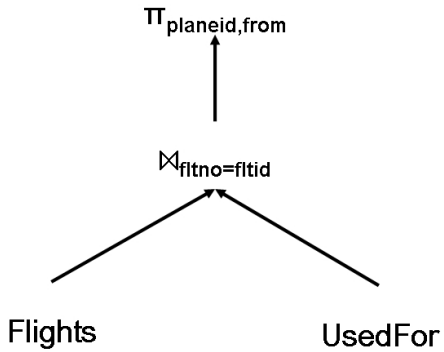


Overview of Query Processing (5)

Example Rewriting Rule and Corresponding Outcome

Example (Join Insertion Rule)

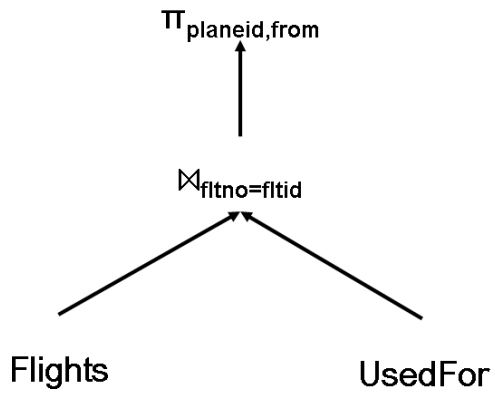
$$\sigma_\theta(R \times S) \Leftrightarrow (R \bowtie_\theta S)$$



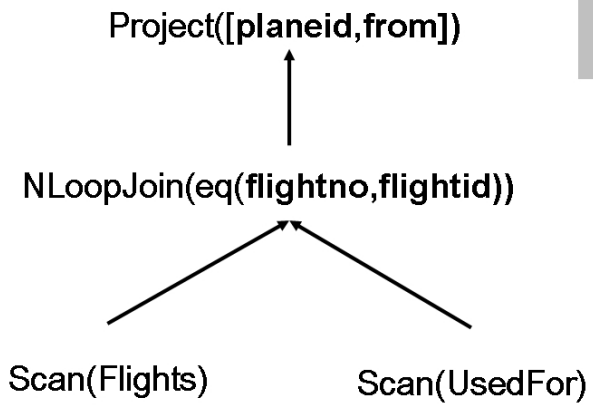
Overview of Query Processing (6)

Rewritten Operator Tree and Corresponding QEP

Rewritten Operator Tree



Query Evaluation Plan



Section 3

Query Processing: Rewriting, Costing, Evaluating

Overview of Query Processing (7)

Translate, then Rewrite

- The outcome of translation is a relational-algebraic expression derived from a direct, clause-by-clause translation.
- A relational-algebraic expression can be represented as an algebraic operator tree.
- By applying rewrite rules, the (usually naive) algebraic operator tree can be rewritten into a heuristically-efficient canonical form, often called the **logical** QEP for the query.

Overview of Query Processing (8)

Compute Costs to Select the Algorithms, then Evaluate

- Typically, every algebraic operator can be implemented by different concrete algorithms.
- Using cost models, the plan selector considers which concrete algorithm to use for each operator in the logical QEP.
- The various concrete algorithms that implement each operator often adopt an iterator pattern.
- The result of this process is a **physical** QEP, i.e., one that expresses a concrete computational process in the actual environment in which the query is to be evaluated.
- The nodes (i.e., the selected concrete algorithms) in a physical QEP are often referred to as **physical operators**.
- The physical QEP can be compiled into an executable or, more commonly, remains an interpretable structure that a **query evaluation engine** knows how to process.

Overview of Query Processing (9)

The Big Questions

- There are three main issues in query optimization:
  1. **For a given algebraic expression, which rewrite rules to use to canonize it?** This determines what heuristic optimization decisions are carried out.
  2. **For a given canonical algebraic expression, which different concrete algorithm assignments to consider?** This determines the search space for finding the desired QEP.
  3. **If the desired plan is the one that results in the shortest response time, what cost models should be used to estimate the response time of a QEP?** This determines the (necessarily sub-optimal) choice of which QEP to use to evaluate the query.
- The above sequence of questions (with the strategy they imply) was first proposed in 1979 in IBM's System R, the first practical relational DBMS, and remains the dominant paradigm.

Lecture 7

Relational Query Processing [2]

Section 1

Logical Optimization: Algebraic Equivalences

Relational-Algebraic Equivalences (1)

Logical Optimization (1)

- Logical optimization involves the transformation of an expression E in a language L into an equivalent expression E' also in L where E' is likely to admit of a more efficient evaluation than E.
- Transformations are often expressed as rewrite rules that express relational-algebraic equivalences of various kinds.
- The different rewrite rules have different purposes, among which:
  - to break down complex predicates in selections and long attribute lists in projections in order to enable more rewrites;
  - to move selections (which are cardinality-reducing) and projections (which are arity-reducing) upstream (i.e., towards the leaves) and thereby reduce the data volumes that downstream operators must contend with;
  - to enable entire subtrees to be implemented by a single efficient algorithm.

Relational-Algebraic Equivalences (2)

Logical Optimization (2)

- Recall that an **associative law** states that two applications of an operation ω can be performed in either order:
  $(x\ \omega\ y)\ \omega\ z \Leftrightarrow x\ \omega\ (y\ \omega\ z)$
- Recall that a **commutative law** states that the result an operation ω is independent of the order of the operands:
  $(x\ \omega\ y) \Leftrightarrow (y\ \omega\ x)$
- For example:
  1. $+$ is both commutative and associative.
  2. $-$ is neither.

Relational-Algebraic Equivalences (3)

Primitive/Derived Transformations

- The definitions of derived operations in terms of primitive ones are relational-algebraic equivalences.
- The most widely used among these (call it **R0** now, we have alluded to it already as a join-insertion rule) rewrites a Cartesian product followed by a selection into a join if the selection condition is a join condition, i.e.:

$$\sigma_\theta(R \times S) \quad \Leftrightarrow \quad (R \bowtie_\theta S)$$

Relational-Algebraic Equivalences (4)

Selection

- **R1**: A conjunctive selection condition can be broken up into a cascade (i.e., a sequence) of individual σ operations, i.e.:

$$\sigma_{\theta_1 \wedge \ldots \wedge \theta_n}(R) \quad \Leftrightarrow \quad \sigma_{\theta_1}(\ldots(\sigma_{\theta_n}(R))\ldots)$$

- **R2**: The σ operation is commutative, i.e.:

$$\sigma_{\theta_1}(\sigma_{\theta_2}(R)) \quad \Leftrightarrow \quad \sigma_{\theta_2}(\sigma_{\theta_1}(R))$$

Relational-Algebraic Equivalences (5)

Projection

- **R3**: In a cascade (i.e., a sequence) of individual π operations all but the last one can be ignored, i.e.:

$$\pi_{L_1}(\ldots(\pi_{L_n}(R))\ldots) \quad \Leftrightarrow \quad \pi_{L_1}(R)$$

- **R4**: The σ and π operations commute if the selection condition θ only involves attributes in the projection list $a_1, \ldots, a_n$, i.e.:

$$\pi_{a_1,\ldots,a_n}(\sigma_\theta(R)) \quad \Leftrightarrow \quad \sigma_\theta(\pi_{a_1,\ldots,a_n}(R))$$

Relational-Algebraic Equivalences (6)

Commutativity of Join and Cartesian Product (1)

- **R5**: Both the $\bowtie$ and the $\times$ operations are commutative, i.e.:

$$\begin{aligned} R \bowtie_\theta S &\Leftrightarrow S \bowtie_\theta R \\ R \times S &\Leftrightarrow S \times R \end{aligned}$$

- **R6.1**: The σ and $\bowtie$ (resp., $\times$) operations commute if the selection condition θ only involves attributes in one of the operands, i.e.:

$$\sigma_\theta(R \bowtie S) \Leftrightarrow \sigma_\theta(R) \bowtie S$$

- **R6.2**: The σ and $\bowtie$ (resp., $\times$) operations commute if the selection condition θ is of the form $\theta_1 \wedge \theta_2$ and θ_1 only involves attributes in one operand and θ_2 only involves attributes in the other, i.e.:

$$\sigma_\theta(R \bowtie S) \Leftrightarrow \sigma_{\theta_1}(R) \bowtie \sigma_{\theta_2}(S)$$

Relational-Algebraic Equivalences (7)

Commutativity of Join and Cartesian Product (2)

- **R7.1**: The π and $\bowtie$ (resp., $\times$) operations commute if the projection list is of the form $L = a_1, \ldots, a_m, b_1, \ldots, b_n$, the a_i are attributes of one operand, the b_j are attributes of the other, and the join condition θ only involves attributes in L, i.e.:

$$\pi_L(R \bowtie_\theta S) \quad \Leftrightarrow \quad (\pi_{a_1,\ldots,a_m}(R)) \bowtie_\theta (\pi_{b_1,\ldots,b_n}(S))$$

- **R7.2**: If the join condition θ includes R-attributes $a_{m+1}, \ldots, a_{m+k}$ and $S-$attributes $b_{n+1}, \ldots, b_{n+l}$ not in L, then they need also to be projected from R and S and a final projection of L is still required, i.e.:

$$\pi_L(R \bowtie_\theta S) \Leftrightarrow \pi_L(\pi_{a_1,\ldots,a_m,a_{m+1},\ldots,a_{m+k}}(R)) \bowtie_\theta (\pi_{b_1,\ldots,b_n,b_{n+1},\ldots,b_{n+l}}(S))$$

- **R7.3**: Since there is no predicate in a Cartesian product **R7.1** always applies with $\bowtie_\theta$ replaced with $\times$.

Relational-Algebraic Equivalences (8)

Commutativity of Set Operations (1)

- **R8**: Both the $\cup$ and the $\cap$ (but **not** the $\setminus$) operations are commutative, i.e.:

$$\begin{aligned} R \cup S &\Leftrightarrow S \cup R \\ R \cap S &\Leftrightarrow S \cap R \end{aligned}$$

- **R9**: The $\bowtie$, $\times$, $\cup$ and $\cap$ operations are individually associative, i.e.:

$$\begin{aligned} (R \bowtie S) \bowtie T &\Leftrightarrow R \bowtie (S \bowtie T) \\ (R \times S) \times T &\Leftrightarrow R \times (S \times T) \\ (R \cup S) \cup T &\Leftrightarrow R \cup (S \cup T) \\ (R \cap S) \cap T &\Leftrightarrow R \cap (S \cap T) \end{aligned}$$

Relational-Algebraic Equivalences (9)

Commutativity of Set Operations (2)

- **R10**: The σ operation commutes with the $\cup$, $\cap$, **and** $\setminus$ operations, i.e.:

$$\begin{aligned} \sigma_\theta(R \cup S) &\Leftrightarrow (\sigma_\theta(R)) \cup (\sigma_\theta(S)) \\ \sigma_\theta(R \cap S) &\Leftrightarrow (\sigma_\theta(R)) \cap (\sigma_\theta(S)) \\ \sigma_\theta(R \setminus S) &\Leftrightarrow (\sigma_\theta(R)) \setminus (\sigma_\theta(S)) \end{aligned}$$

- **R11**: The π operation commutes with the $\cup$ operation, i.e.:

$$\pi_L(R \cup S) \Leftrightarrow (\pi_L(R) \cup \pi_L(S))$$

Relational-Algebraic Equivalences (10)

Other Transformations

- Predicates can also be rewritten (e.g., using De Morgan's laws to push negation into conjuncts and disjuncts).
- Note that using **R1** to rewrite separate predicates into a conjunction thereof could, depending on the compiler, lead to a contradiction, in which case the selection would be satisfied by no tuple in the input, resulting in an empty result.

Section 2

Logical Optimization: A Heuristic Strategy

A Heuristic Algebraic Optimization Strategy (1)

1. Use **R1** to break up complex select conditions into individual ones. This creates opportunities for pushing selections down towards the leaves, thereby reducing the cardinality of intermediate results.
2. Use **R2**, **R4**, **R6** and **R10**, which define commutativity for select and pushing selections down towards the leaves.
3. Use **R5**, **R8** and **R9**, which define commutativity and associativity for binary operations, to rearrange the leaf nodes. The following criteria can be used:
    1. Ensure that the leaves under the most restrictive selections (i.e., those that reduce the size of the result) are positioned to execute first (typically, as the left operand). This can rely on selectivity estimates computed from metadata in the system catalogue, as we shall see.
    2. Avoid causing Cartesian products to be used, i.e., override the above criterion if it would not lead to a join being placed above the operands.

A Heuristic Algebraic Optimization Strategy (2)

4. Use **R0** to combine selections on the result of Cartesian products into a join. This allows efficient join algorithms to be used
5. Use **R3**, **R4**, **R7** and **R11**, which define how project cascades and commutes with other operations, to break up and move projection lists down towards the leaves, thereby reducing the arity of intermediate results. Only those attributes needed downstream in the plan need be kept from any operation.
6. Identify subtrees that express a composition of operations for which there is available a single, specific algorithm that computes the result of the composition.

Section 3

Logical Optimization: An Example

An Example Run (1)

Schemas and Query

Example (Simplified Schema)

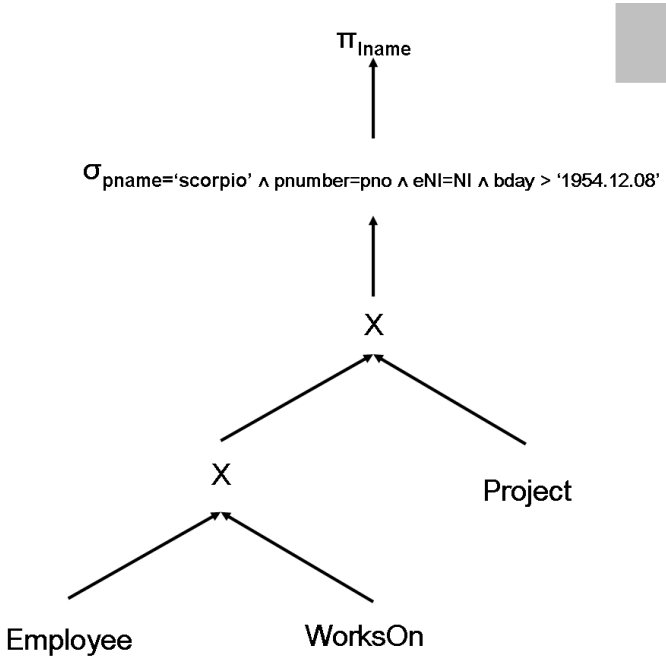
Employee (fname, mname, lname, bday, address, sex, sal, NI, dno)
Project (pname, pnumber, ploc, dnum)
WorksOn (eNI, pno, hours)

Example (SQL Query)

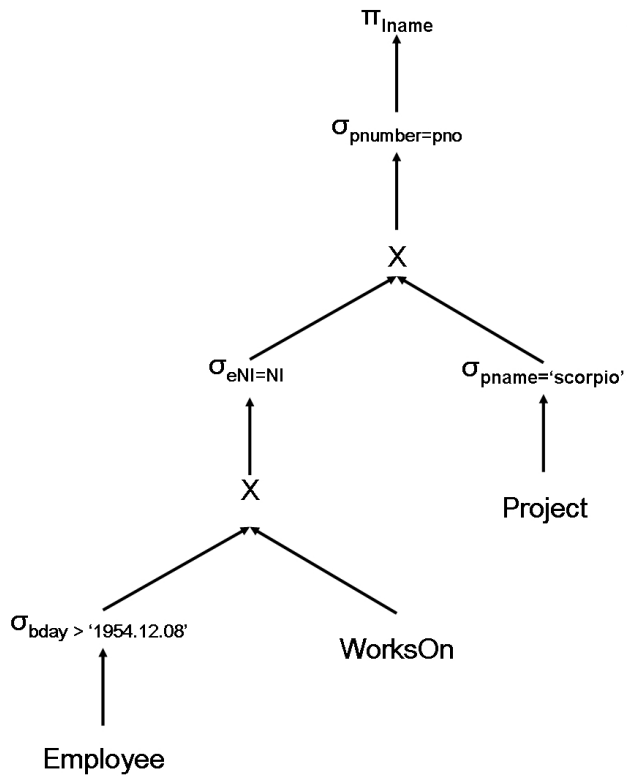
```
SELECT E.lname
FROM   Employee E, WorksOn W, Project P
WHERE  P.pname = 'Scorpio'
  AND  P.pnumber = W.pno AND W.eNI = E.NI
  AND  E.bday > '1954.12.08'
```

An Example Run (2)

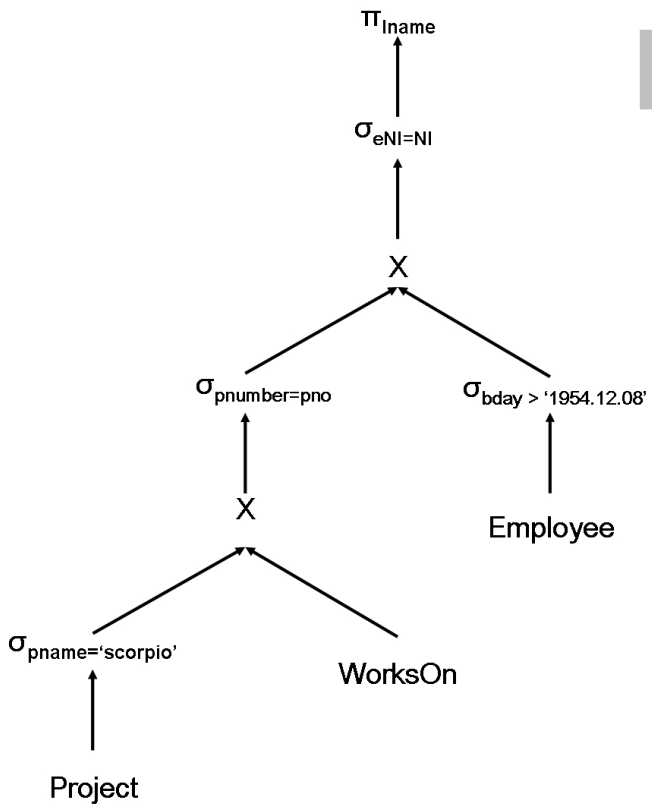
Translated Operator Tree



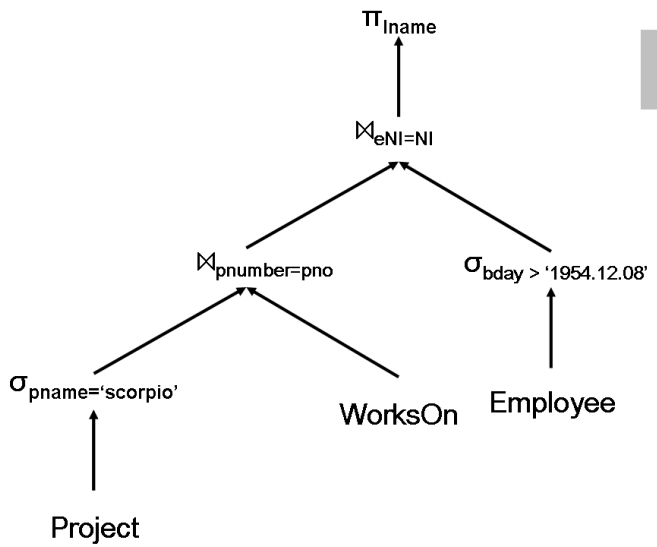
Operator Tree after Steps 1 and 2



An Example Run (3)

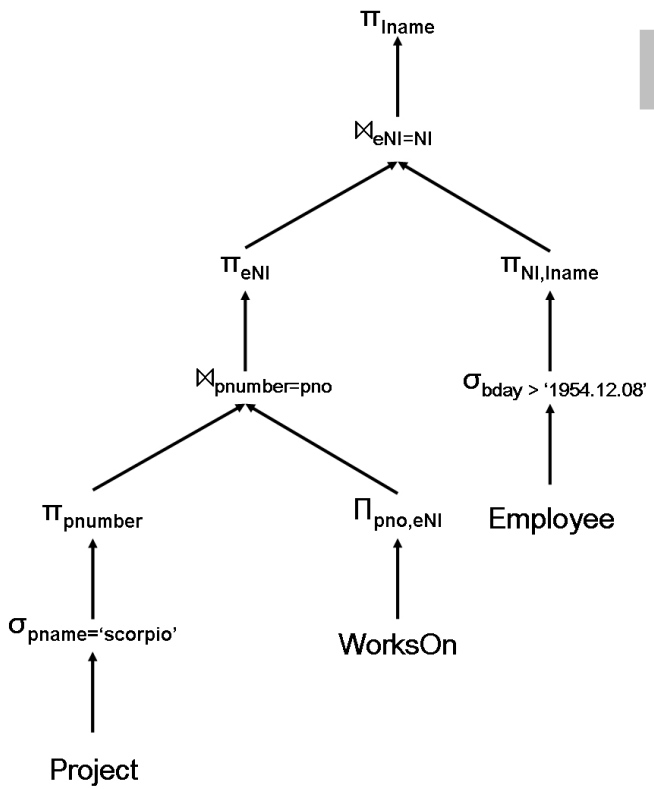


Operator Tree after Step 3

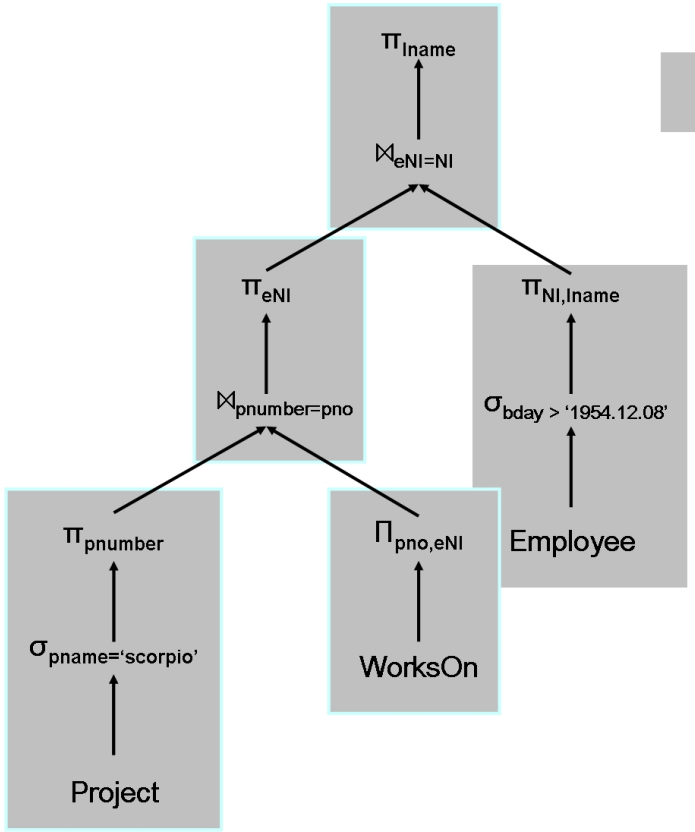


Operator Tree after Step 4

An Example Run (4)



Operator Tree after Step 5



Operator Tree after Step 6

Summary

Equivalence-Based Rewriting of Logical QEPs

- For a given algebraic expression, deciding which rewrite rules to use to canonize it is a major issue in query processing.
- This determines what heuristic optimization decisions are carried out.
- Equivalence-based rewriting of logical QEPs is a classical strategy.
- It is sufficiently well-established and well-understood for there to be a consensus on what works fairly well in the classical cases.
- There is great uncertainty as to what extent rewriting is also useful in several kinds of advanced DBMSs.
- In any case, cost-based optimization is always fundamentally required.

Acknowledgements

The material presented mixes original material by the author and by Norman Paton as well as material adapted from

- [Elmasri and Navathe, 2006]
- [Garcia-Molina et al., 2002]
- [Ramakrishnan and Gehrke, 2003]
- [Silberschatz et al., 2009]

The author gratefully acknowledges the work of the authors cited while assuming complete responsibility any for mistake introduced in the adaptation of the material.

Lecture 8

Relational Query Processing [3]

Section 1

Algorithmic Strategies

Query Evaluation (1)

Some Common Algorithmic Strategies

Algorithms for implementing query-algebraic operators tend to adopt one of the following strategies:

Scanning Sometimes it is faster to scan all the tuples even if there is an index. Sometimes we can scan the data entries in an index instead of the table itself.

Indexing It often pays handsomely to build indexes on attributes so that when predicates are evaluated (e.g., in selections, and, most importantly, in joins) one retrieves from the index only a small set of tuples.

Partitioning Using sorting or hashing, we can partition the input tuples on a given key and thereby decompose an operation into a collection of operations on smaller inputs, i.e., the partitions.

Section 2

Query Evaluation Strategies

Query Evaluation (2)

Operator-to-Operator Protocol (3)

Materialization

- The result produced by each physical operator is stored either in memory or on disk as it is produced.
- This process of temporarily storing complete, but intermediate, results is referred to as **materialization**.

Pipelining

- While individual physical operators may keep state, no complete intermediate results are temporarily stored.
- In pipelining (unlike in materialization), many physical operators are likely to be actively executing soon after the start of the evaluation of the QEP as a whole.

Summary

Query Evaluation Strategies

- Most algorithms that implement relational-algebraic operators can be seen to adopt one or more of just a few algorithmic strategies.
- Likewise, the interaction protocols between algorithms in an executing QEP are, in the classical case, few in kind.

The Iterator Model

The Interface: {Open, Next, Close}

- Iterators are widely used to implement pipelining.
- In the iterator model, each physical operator must implement the following functions:

Open Initialize any local data structure used by the operator to keep state, and call `Open` on all its inputs (i.e., its child(ren) in the QEP).

Next Compute the next tuple of the result (typically by first calling `Next` on its child(ren)), update any local data structures so as to allow continuation, and return that result tuple. Return a not-found flag if there are no more tuples to return.

Close Call `Close` on the child(ren) and clear up any local data structures used by the operator.

The Iterator Model

The Consequences (1)

- Calls cascade, recursively, down the tree from the root to the leaves.
- Since, by definition, the leaves are childless, the returns from such calls bubble up from (i.e., bounce back up as they hit) the leaves to the root.
- The evaluation of a QEP thus goes, roughly, through distinct phases:
  1. There is a wavefront of `Open` calls from the root to the leaves that bubble up in return.
  2. This followed by a period of productive activity in which children operators respond to `Next` calls sent by their parents.
  3. This period gradually winds down as a result of a reverse wavefront (from the leaves to the root) of failed calls to `Next` (because all results that the children could produce have been produced) that return a not-found flag.
  4. There is a final wavefront of `Close` calls from the root to the leaves that bubble up in return, thereby concluding the evaluation.

The Iterator Model

The Consequences (2)

- Under the iterator model, results are pulled, i.e., produced on demand.
- Operators tend to be busy at the level they can afford to.
- This reduces the chances that the query engine will hog resources and draw punitive action by the operating system (OS).
- If each operator were to run as a distinct thread, then the OS scheduler is more likely to keep its cool and not intervene.

Section 3

Physical Operators

Categories of Physical Operator

Physical operators can be classified based on the number of times data is read from disk:

One-pass algorithms read data from disk only once, but usually only work when at least one operand fits in memory.

Two-pass algorithms read data from disk in chunks the size of the available memory, process that data in memory (e.g. by hashing or sorting it), and then write the processed chunk to disk. The processed chunk can then be read in sorted order for subsequent processing.

This course unit assumes one-pass algorithms most of the time.

Physical Operators (1)

Iterator-Based Scan (1)

- Scan reads the tuples from a table one at a time.
- It is parameterized by the relation R that we wish to scan.
- We assume that the notion of reading a block and the auxiliary notions it suggests are supported by the storage manager.
- The Open method essentially obtains a reference to the start of the area of disk containing R.

```
class Scan implements Iterator
  Block b;  // Block being read from
  Tuple t;  // Next tuple to be returned
  Relation R;

  Open()
    b := the first block of R;
    t := the first tuple of b;

  ...
```

Physical Operators (2)

Iterator-Based Scan (2)

- The `Next` method uses the state stored in `t` and `b` to remember where it has reached.
- The `Close` method has nothing to do in this case.

```
class Scan implements Iterator
  ...
  Next()
    IF (t is after the last tuple in b)
      increment b to next block;
      IF (there is no next block)
        RETURN not-found;
      ELSE t := first tuple in b;

    res := t;
    increment t to next tuple of b;
    RETURN res;


  Close()
```

Physical Operators (3)

Iterator-Based Select (1)

- Select has local, tuple-at-a-time semantics: we need not be aware of any other tuple than the one we have read.
- $\sigma_\theta(R)$ can be implemented in a single pass over the collection R.
- As for all other non-leaf physical operators, the results of any other physical operator can constitute the input to select.
- Tuples that satisfy θ are returned, those that do not are passed over.

Physical Operators (4)

Iterator-Based Select (2)

```
class Select implements Iterator
  Iterator i;  // Input collection
  Condition theta;

  Open()
    i.Open();


  Next()
    Tuple t;
    WHILE ((t:=i.Next()) != nil)
      IF (t satisfies theta) RETURN t;
    RETURN not-found;


  Close()
    i.Close();
```

Physical Operators (5)

Project

- Under bag semantics, $\pi_{a_1,\ldots,a_n}(R)$ also has local, tuple-at-a-time semantics.
- It can be implemented in a single pass over the collection R.
- New tuples are generated that retain only the attributes $a_1, \ldots, a_n$ of R.
- Under set semantics, project is a full-relation operator, i.e., it has global, set-at-a-time semantics: we do need to be aware of other tuples than the one we have read, otherwise the projection may generate duplicates, which sets do not have.

Physical Operators (6)

Join Algorithms: Nested-Loop Join (1)

- Joins are among the most expensive operations performed in a relational database.
- Much effort has been directed at finding efficient strategies for evaluating them.
- One (not very efficient) algorithm for $R \bowtie S$ is (tuple-based) **nested-loop join (NLoopJoin)**:

$$
\begin{array}{l}
result := \emptyset \\
\texttt{FOR EACH } r \in R \texttt{ DO} \\
\quad \texttt{FOR EACH } s \in S \texttt{ DO} \\
\quad\quad \texttt{IF } s \text{ matches } r \\
\quad\quad\quad result := result \cup \{concat(r, s)\}
\end{array}
$$

Physical Operators (7)

Join Algorithms: Nested-Loop Join (2)

- Tuple-based NLoopJoin has very high I/O costs (as the inner relation S is scanned in its entirety for each tuple in R).
- Variants include:
  - reading pages rather than tuples (i.e., we have $r \in RPage$ and $s \in SPage$ instead;
  - making use of buffers more wisely, e.g., if we can hold the smaller relation, say R, in memory and have two extra buffer pages, we can use one of those to scan S and the other one to place the output tuples in;
  - if we cannot hold R in its entirety, we can still use blocks into which we fit as much of R as we can;
  - if there is an index on the join attributes for either R or S, we can make it the inner relation and rather than scanning it for every tuple in the outer relation, we only look-up and retrieve the matching tuples from it.

Physical Operators (8)

Join Algorithms: Hash Join (1)

- Joins based on partitioning strategies use sorting or hashing, e.g., **sort-merge join** and **hash join**.
- Hash join is altogether more efficient than nested-loop join.
- Firstly, a hash table is populated, indexed on the join attributes, with one entry for each tuple in one (typically the smaller) of the inputs. Then, the hash table is probed, again on the join attributes, using every tuple in the other input.
- Assuming that the hash table look-up retrieves exact matches (rather than every item with the same hash position, e.g., bucket), every tuple that is retrieved using the probe contributes to the join result.

Physical Operators (9)

Join Algorithms: Hash Join (2)

$$
\begin{array}{l}
\mathit{result} := \emptyset \\
\mathit{hashtable} := \texttt{new } \mathit{HashTable}() \\
\texttt{FOR EACH } r \in R \texttt{ DO} \\
\quad \mathit{hashtable}.\texttt{insert}(r(a_1, \ldots, a_n), r) \\
\texttt{FOR EACH } s \in S \texttt{ DO} \\
\quad \mathit{matches} = \mathit{hashtable}.\texttt{lookup}(s(a_1, \ldots, a_n)) \\
\quad \texttt{WHILE } ((r := \mathit{matches}.\texttt{Next}()) \neq \texttt{nil}) \\
\quad\quad \mathit{result} := \mathit{result} \cup \{\mathit{concat}(r, s)\}
\end{array}
$$

Physical Operators (10)

Two-Pass Algorithms

- If the tables to be operated on are too big to fit in memory, one option is to base algorithms on sorted or hashed partitions.
- The two passes for an operator involve:
  1. Scanning the data from the original collection(s) in order to generate a number of partitions whose size takes into account the amount of memory available.
  2. Storing the partitions on disk either hashed or sorted, so that data items can be accessed in a systematic way.
- Let a **block** (sometimes called a **(disk) page**) be the basic unit of data storage on disk; let a **buffer** (sometimes called a **buffer page**) be the unit of main memory associated with a block on disk; and let a **bucket** be an entry in a hash table.

Physical Operators (11)

Hash-Based Partitioning

If there are M memory buffers available, then the following partitions R into no more than $M-1$ buckets on the attributes $a_1, \ldots, a_n$:

```
FOR i := 1 TO M − 1 DO  initialize buffer B_i
FOR EACH  block b ∈ R DO
    read b into the M^th buffer
    FOR EACH  tuple t ∈ b DO
        IF B_hash(t(a_1,...,a_n)) is full
            write that buffer to disk
            initialize a new buffer
        copy t to B_hash(t(a_1,...,a_n))
FOR i := 1 TO M − 1 DO
    IF B_i is not empty
        write that buffer to disk
```

Physical Operators (12)

Two-Pass Hash-Join (1)

- The first pass of a two-pass hash-join is one partitioning step (as described) for each operand.
- The tuples of each operand are hashed (using the same hash function for both operands) into buckets and written out.
- In the second pass, corresponding partitions are operated upon as if they were entire relations themselves.
- In this second pass, one can, in principle, use any one-pass join algorithm.
- The use of the in-memory hash join algorithm in the second pass of a hash-partitioned two-pass join is known as a GRACE join (for the system in which it was first used).

Physical Operators (13)

Two-Pass Hash-Join (1)

- Using a nested-loop strategy is also possible.
- For $R \bowtie S$, on join attributes $a_1, \ldots, a_n$, where BR_i (resp., BS_i) is the bucket with index i in the hash table for R (resp., S), the second pass is:

$$
\begin{array}{l}
result := \emptyset \\
\texttt{FOR EACH } \text{bucket } BR_i \in R \texttt{ DO} \\
\quad \texttt{FOR EACH } \text{tuple } t \in BR_i \texttt{ DO} \\
\quad\quad \texttt{IF } t \text{ matches some tuple } s \in BS_i \\
\quad\quad\quad result := result \cup \{concat(r, s)\}
\end{array}
$$

- The buckets associated with R can be read in any order; the buckets associated with S are read directly, based on the hash index of the current R bucket.

Physical Operators (14)

Set Operations

- There are two groups:
  1. Intersection and Cartesian product are special cases of join.
  2. Set union and set difference are very similar.
- As with joins, one can use sort-based approaches or hash-based ones.
- In a sort-based approach to union:
  1. Sort both relations (on the combination of all attributes).
  2. Scan the sorted relations and merge them, skipping duplicates.
- The hash-based approach to union is very similar to the approach used for hash join:
  1. Partition R and S using the same hash function h on both.
  2. For each S-partition, build an in-memory hash table (using another hash function h'), then scan the corresponding R-partition and add tuples to the table, skipping duplicates.

Physical Operators (15)

Aggregate Operations

- There are two cases:
  1. Without grouping
  2. With grouping
- Ungrouped aggregates can be computed by scanning the relation while keeping running information (e.g., counts, sums, smallest value, largest value).
- Given an existing index whose search key includes all attributes in the SELECT or WHERE clauses, it is possible to use an index (as opposed to a table) scan.
- Like join and set union/difference, grouped aggregates can be computed using sort-based or hash-based partitioning.

Summary

Evaluating Relational-Algebraic Operators

- It is a fact of central importance in query processing that the collection of concrete algorithms that implement relational-algebraic operators is well-defined and well-studied.
- This a priori knowledge essentially implies that QEPs are compositions of primitives whose functional and non-functional models (e.g., their space and time costs) are well-known.
- This knowledge allows for cost-based optimization, as we will shortly see.

Lecture 9

Relational Query Processing [4]

Section 1

Cost-Based Plan Selection

Cost-Based Plan Selection

- The number of I/Os performed by a plan is influenced by:
  1. The logical operators used to implement the query.
  2. The physical operators used to implement the logical operators.
  3. The sizes of intermediate relations.
  4. The evaluation order of operators.
  5. The way in which information is passed between operators.
- The following slides indicate how these issues can be taken into account by a query optimizer.

Obtaining Values for Cost Model Parameters

- As we have seen, the data dictionary stores statistical information for use by the optimizer.
- The database administrator has general responsibility for configuring parameters and for updating statistics.
- In general:
  - M is normally a configuration parameter.
  - $B(R)$ is easily computed from the information on where/how a relation is stored.
  - $T(R)$ is either stored explicitly or can be estimated (as exemplified above).
  - $V(R, A)$ (like $T(R)$) can be computed in a single scan through a relation.
  - To avoid scanning the complete relation, $V(R, A)$ may be estimated by sampling.

Section 2

Generating and Ranking Plans

Ranking a Logical QEP

- As logical QEPs are not associated with physical operators, there is no direct way to compute the disk I/Os of a logical QEP.
- Thus, as exemplified by the heuristics-based rewrite algorithm studied before, logical optimization proceeds by:
  - Structuring the process by which transformations are applied so that it is directed by chosen heuristics.
  - For example, that a logical QEP with likely smaller intermediate results is to be preferred on efficiency grounds.
  - Appealing to and applying transformations that are consistent with the chosen heuristics.
  - For example, that, by pushing selections and projections to lie as close as possible to the leaves, smaller intermediate results are likely to result.

Annotating a Logical QEP

Size-Estimate Annotations

Example

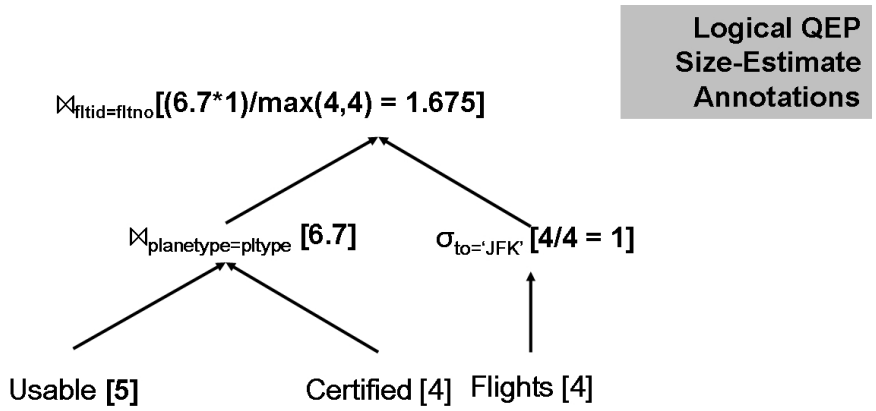
Given the following statistics:

Table	T(R)	V(R,A)		
Usable	5	V(Usable,pltype)	=	3
		V(Usable,flid)	=	4
Certified	4	V(Certified,planetype)	=	2
Flights	4	V(Flights,fltno)	=	4
		V(Flights,to)	=	4

the logical QEP

$(Usable \bowtie_{pltype=planetype} Certified) \bowtie_{flid=fltno} (\sigma_{to=JFK}(Flights))$

can be annotated with size estimates as shown to the right.



Generating Alternative QEPs

- Optimizers differ in where alternative QEPs are generated.
- Two possibilities are:
  1.
     - The logical optimizer generates alternative tree shapes by applying transformation rules, and retains those that are judged to be of reasonable quality (e.g., considering the size of intermediate results).
     - The physical optimizer then assigns physical operators for their logical counterparts and ranks the resulting alternative QEPs.
  2.
     - The logical optimizer works in a purely heuristic manner (e.g., in particular, it is agnostic about deciding on join ordering).
     - The physical optimizer then considers alternative join orderings and assigns physical operators for their logical counterparts.

Section 3

Selecting Join Orders

Selecting a Join Order (1)

An Intractable Problem

- Recall that joins are commutative and associative, therefore in a sequence of joins, varying the order in which they are applied does not affect the result.
- However, different orders have very different associated costs.
- In practice, selecting a join order is crucial for efficient query evaluation.
- There are many different algorithms for choosing a join ordering for queries with many joins.
- For complex queries, it is impractical to enumerate all the possibilities.

Selecting a Join Order (2)

A Greedy Algorithm

- Let $\mathcal{R} = \{R_1, \ldots, R_n\}$ be a collection of relations, and let $minsize(\mathcal{P}, S, \mathcal{R})$ be true iff the relation S in $\mathcal{R}$ is the one that leads to the smallest estimated result size when joined with the logical QEP fragment $\mathcal{P}$.
- Then, the following greedy algorithm seeks to keep intermediate relations as small as possible:

$$\mathcal{P} := \{R_i \bowtie R_j | \forall i, j : R_i \in \mathcal{R}, R_j \in \mathcal{R} : minsize(R_i, R_j, \mathcal{R})\}$$
$$\mathcal{R} := \mathcal{R} \setminus \{R_i, R_j\}$$
$$\texttt{WHILE } (\mathcal{R} \neq \emptyset) \texttt{ DO}$$
$$\quad \mathcal{P} := \{\mathcal{P} \bowtie R_i \mid \forall i : R_i \in \mathcal{R} : minsize(\mathcal{P}, R_i, \mathcal{R})\}$$
$$\quad \mathcal{R} := \mathcal{R} \setminus \{R_i\}$$
$$\texttt{RETURN } \mathcal{P}$$

Selecting a Join Order (3)

Example

- Given the logical QEP $R \bowtie S \bowtie T \bowtie U$ and the following statistics:

Join	$R \bowtie S$	$R \bowtie T$	$R \bowtie U$	$S \bowtie T$	$S \bowtie U$	$T \bowtie U$
Size	5,000	20,000	10,000	2,000	40,000	1,000

- The initial rewritten logical QEP becomes $T \bowtie U$.
- Then, there are two possibilities to consider:

Join	$(T \bowtie U) \bowtie R$	$(T \bowtie U) \bowtie S$
Size	10,000	2,000

- Assuming the estimates above, the rewritten logical QEP becomes $(T \bowtie U) \bowtie S$.
- Then, there is only one possibility left and the final rewritten logical QEP is $((T \bowtie U) \bowtie S) \bowtie R$.
- Note that the search was not exhaustive and hence the result may not be optimal.

Section 4

Choosing Physical Operators

Choosing Physical Operators (1)

Selection

Options for $\sigma_\theta(R)$ include:

- Scanning the tuples in R, which is necessary if there is no index on R, in which case the I/O cost is $B(R)$, or zero if R is already in memory.
- Using an index on R if it exists, and θ includes $A = c$, where A is an indexed attribute and c is a constant.
- For a typical indexed select, the I/O cost is $\frac{T(R)}{V(R,A)}$, assuming that the cost of reading the index is negligible and each tuple identified as a result of an index lookup is stored in a different block.

Choosing Physical Operators (2)

Join

- The join operator of choice depends on:
  - the anticipated sizes of operand collections
  - the amount of available memory
  - the availability of indexes
  - whether or not the data is sorted
- If both collections are sorted on the join attribute, a merge join is typically used.
- If there is an index on a join attribute, then index-based join algorithms significantly reduce the total I/O cost.
- If operands are likely to fit in memory, then one-pass algorithms are to be preferred, but their performance deteriorates rapidly when there is not enough memory.
- For large collections, two-pass algorithms provide more predictable performance.

Choosing Physical Operators (3)

Pipelining and Materialization

Some operators fit in better with pipelining than others.

Select always works well pipelined, requiring one input and one output buffer.

Project may need to eliminate duplicates, and, if so, it needs to cache the whole of its result table, leading to storage overheads.

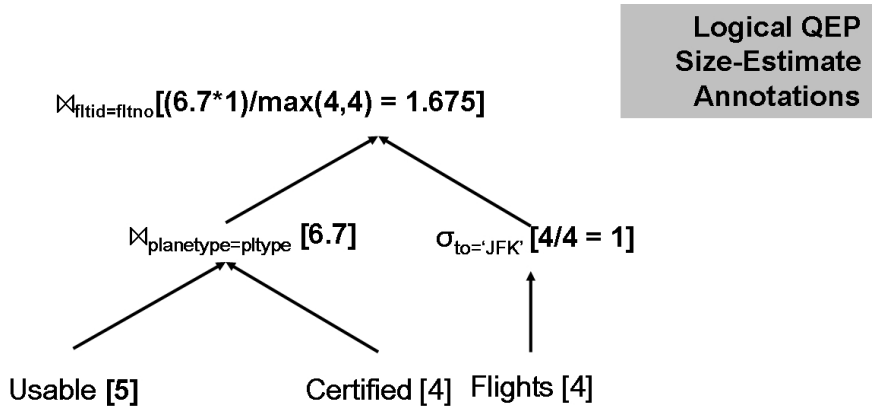
Nested-Loop Join (if it is to be practical) reads and caches one operand, again leading to storage overheads.

Hash Join reads one operand into a hash table, once more leading to storage overheads.

Pipelined versions of the last three above exist that can be used for pipelining.

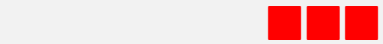
Estimating Memory Use

- It is important that query evaluation avoids causing thrashing.
- It is easier to anticipate the memory needs of materialized than pipelined plans.
- For example, if the QEP to the right is evaluated using materialization, the temporary memory of the lower join can be freed up before the upper join starts to be evaluated.
- Not so with pipelining.
- However, with materialization, the result of the lower join must be stored until the upper join starts to be evaluated.
- Not so with pipelining.



Summary

Query Processing



- Query processing is what distinguishes DBMSs from other systems.
- Very few other classes of complex software artifacts have emerged that offer such quality guarantees to so many and so varied applications.
- The query processing stack, the advanced and fundamentally elegant concepts and ideas that it embodies, is what delivers added-value and empowers users and applications to an unprecedented extent.
- The remainder of this course will look primarily into how the query processing stack has been changing to deliver advanced functionalities that classical DBMSs are not as well-equipped to deliver.

Lecture 10

Relational Query Processing [5]

Section 1

Parallel Query Processing: Data Partitioning

Key Techniques for Parallel Query Processing

1. Partition relation extents across multiple mass-storage units.
2. Pipeline tuples between relational operators.
3. Execute multiple copies of relational operators across multiple processing elements.

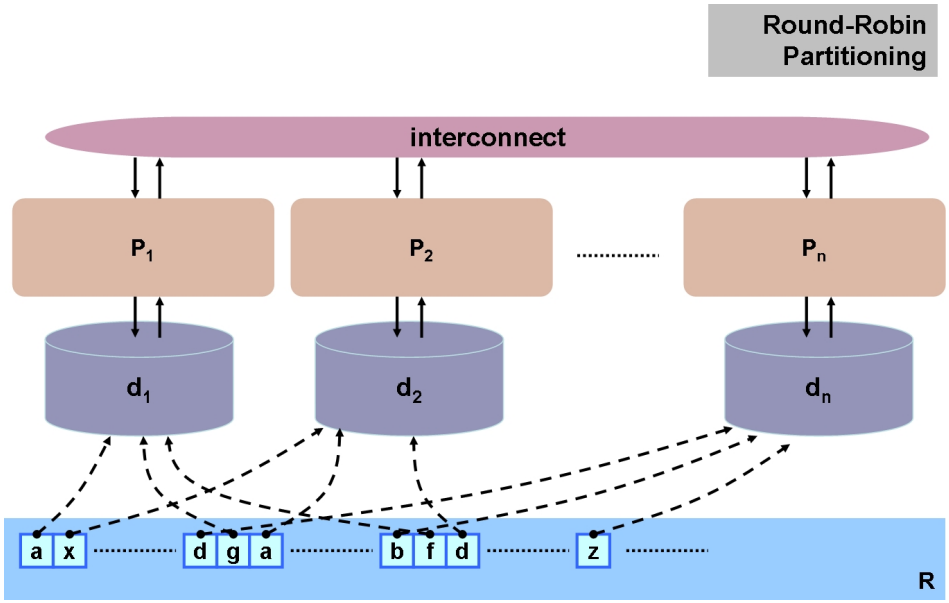
Data Partitioning (1)

- Partitioning a relation extent involves distributing its tuples across several hardware-architectural elements.
- In the cases of mass-storage units, this can provide superior I/O bandwidth superior to RAID-style devices without any specialized hardware.
- The three basic partitioning strategies are **round-robin**, **hash-based**, and **range-based**.

Data Partitioning (2)

Round-Robin Partitioning

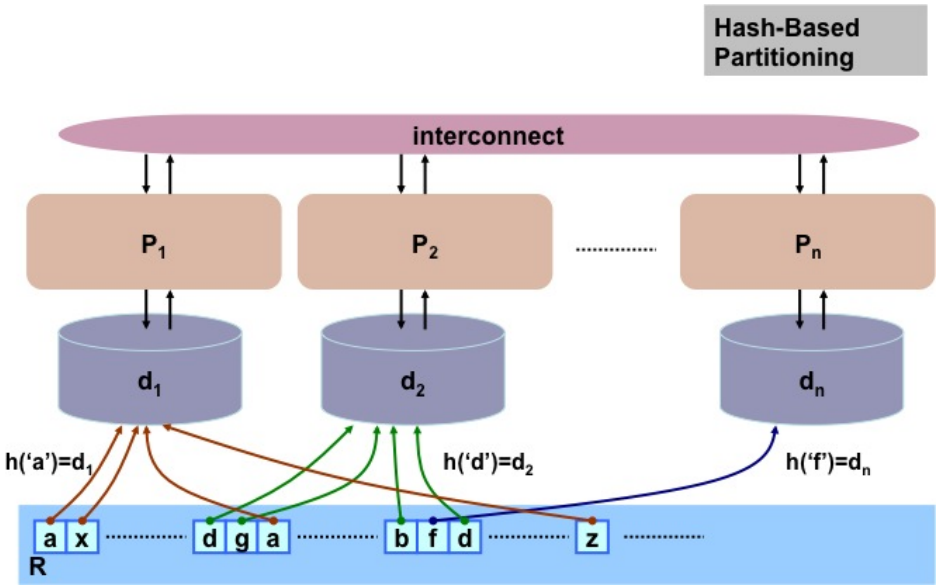
- Given n architectural elements, round-robin partitioning maps the i-th tuple in the data to the element $i \ mod \ n$.
- In the figure, assuming n disks, the i-th tuple in R is loaded onto the $i \ mod \ n$ disk.



Data Partitioning (3)

Hash-Based Partitioning

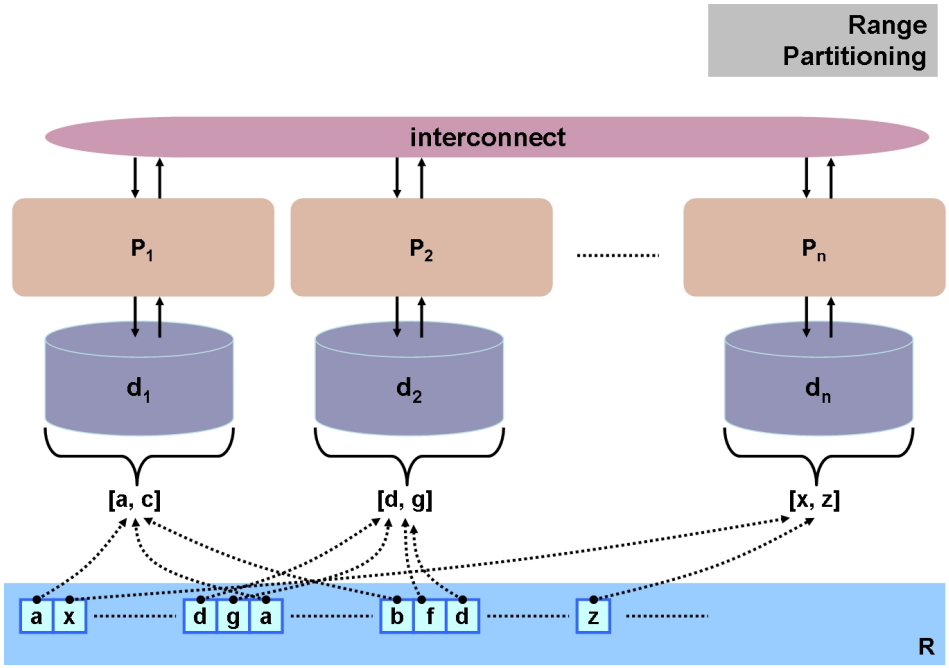
- Given n architectural elements, hash-based partitioning maps the i-th tuple in the data to the element returned by a hash function (whose range has cardinality n) applied to the chosen attribute in the tuple.
- In the figure, for loading a relation onto disks, assuming n disks, the disk each tuple is loaded into is determined by its hashed value.



Data Partitioning (4)

Range Partitioning

- Given a relation, let a set of k disjoint clusters on one of its attributes be defined over it.
- This can be obtained by different methods (e.g., a simple one could split the lexicographically-ordered domain of the attribute into k intervals).
- Given $n = k$ architectural elements and a $(k = n)$-clustered data set, range-based partitioning maps the i-th tuple in the data to the element j if the corresponding value of the clustering attribute in that tuple belongs to the j-th cluster.
- In the figure, for loading a relation onto disks, assuming n disks, each disk stores the tuples in the disk corresponding to its interval.



Data Partitioning (5)

Summary

- Round robin is excellent for sequential access of full relations but poor for associative access, i.e., one requiring all tuples with a particular value.
- Hashing is excellent for associative access (i.e., queries involving exact matches).
- While hashing tends to randomize data, range partitioning clusters it, making it useful in both sequential and associative access.
- However, range partitioning can cause **data skew** (i.e., uneven allocation of tuples across storage elements) and **execution skew** (i.e., uneven load across processing elements), where the other approaches are less susceptible to that.
- Clearly, picking appropriate partitioning criteria in range partitioning is crucial.

Section 2

Parallelizing Relational Algebraic Operators

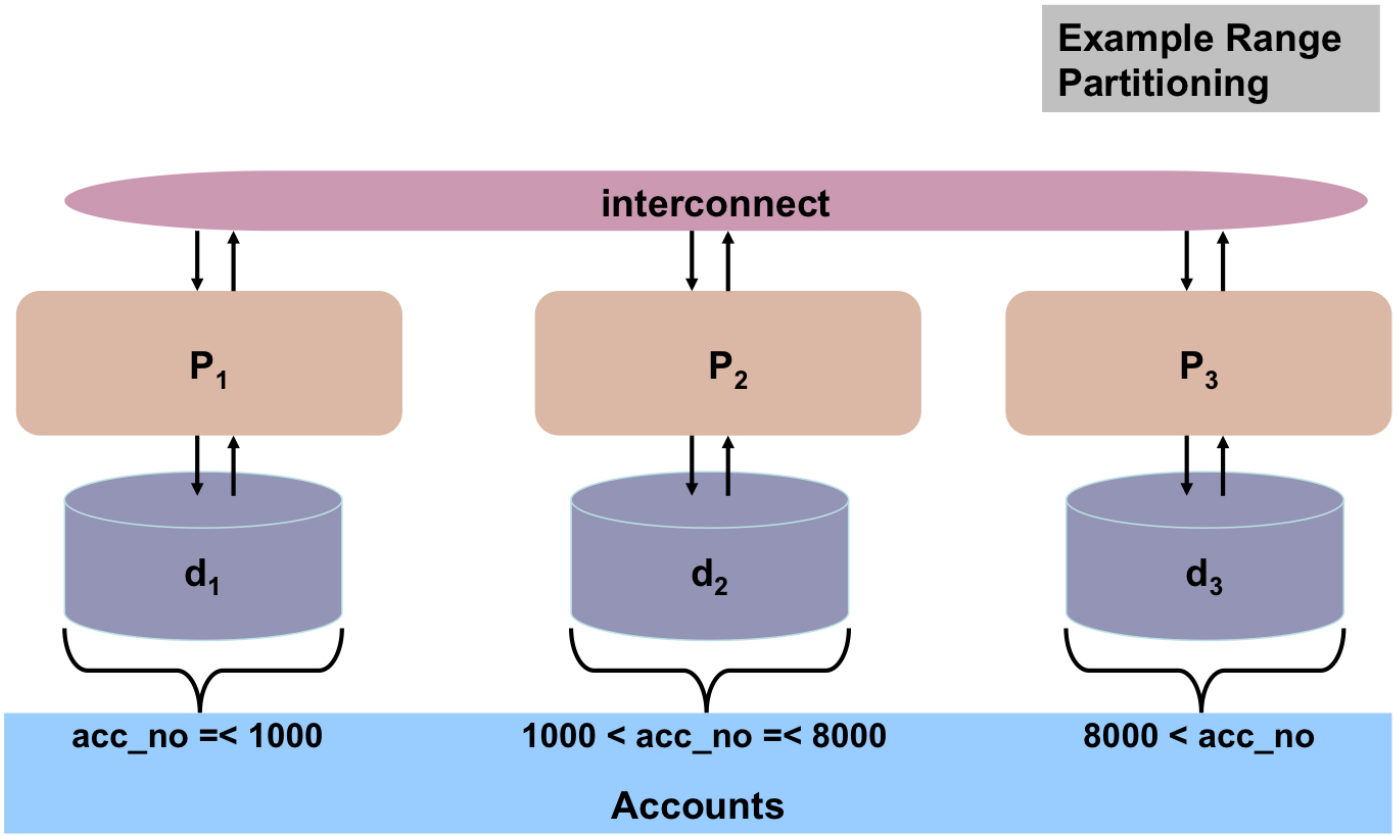
Parallelizing Relational Operators (1)

Assumptions and Requirements

- The goal is to use existing operator implementations without modification.
- Any new mechanisms for parallelization should either act on operators or be operators themselves
- In other words, all the characteristics of elegance and economy of design in the relational model and algebra aim to be preserved here.
- Given:
  - a shared-nothing architecture, and
  - operating system support for at least reliable datagrams and processes/threads,
- Three new mechanisms are added:
  1. operator replication
  2. merge operator
  3. split operator
- The result is a parallel DBMS capable of providing linear speed-up and linear scale-up.

Parallelizing Relational Operators (2)

An Example Partitioned Relation



Parallelizing Relational Operators (3)

Operator Replication

Example

Assume the following queries:

$q_1 =$

```
SELECT   A.name, A.balance
FROM     Accounts A
WHERE    A.balance > 10000
```

$q_2 =$

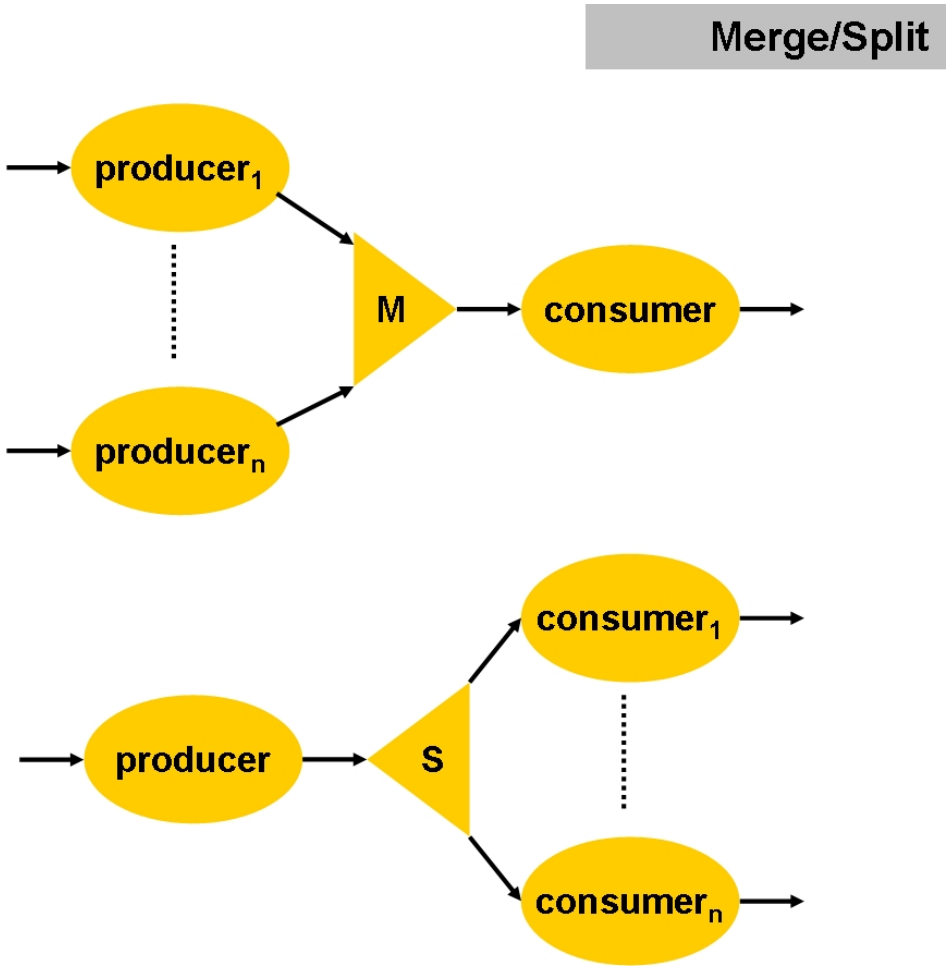
```
SELECT   A.name, A.balance
FROM     Accounts A
WHERE    A.acc_no = 339
```

- q_1 executes with maximum degree of parallelism (3, in this case), i.e., on p_1, p_2, and p_3
- q_2 executes on p_1; the data in p_2 and p_3 is not scanned; p_2 and p_3 are free to run other queries in parallel with q_2.
- Operator replication scales arbitrarily, subject to the overheads of starting an operator on each participating processing element and of their interfering with one another.

Parallelizing Relational Operators (4)

Merge and Split Operators

- Given input streams stemming from *n* producers, a **merge operator** generates from them a single output stream destined for a consumer.
- Given one input stream stemming from a producer, a **split operator** defines a mapping from one or more attribute values to a set of output streams each destined for a corresponding consumer process.
- For example, on a numeric field (like acc_no in Accounts) if there are *n* consumers a splitting function could be *acc_no mod n*.



Parallelizing Relational Operators (5)

Merge: An Example

Example

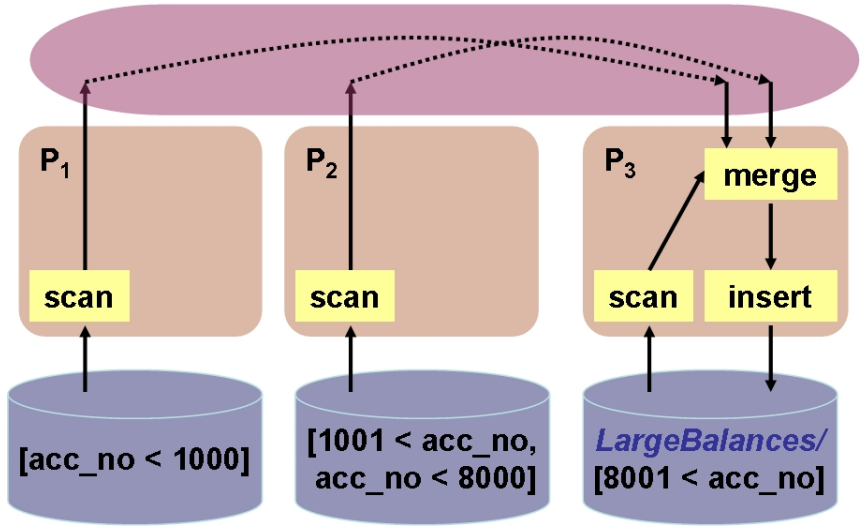
Assume the following SQL statement:

$q_3 =$

```
INSERT
INTO        LargeBalances
  SELECT    A.name, A.balance
  FROM      Accounts A
  WHERE     A.balance > 10000
```

- Scans are replicated and run in parallel.
- Assuming a cost-neutral interconnect, the merge operator could run anywhere.
- Assuming that *LargeBalances* is meant to be co-located with $acc\_no > 8001$, the merge ensures that the result of the parallel scans end up there to be inserted.

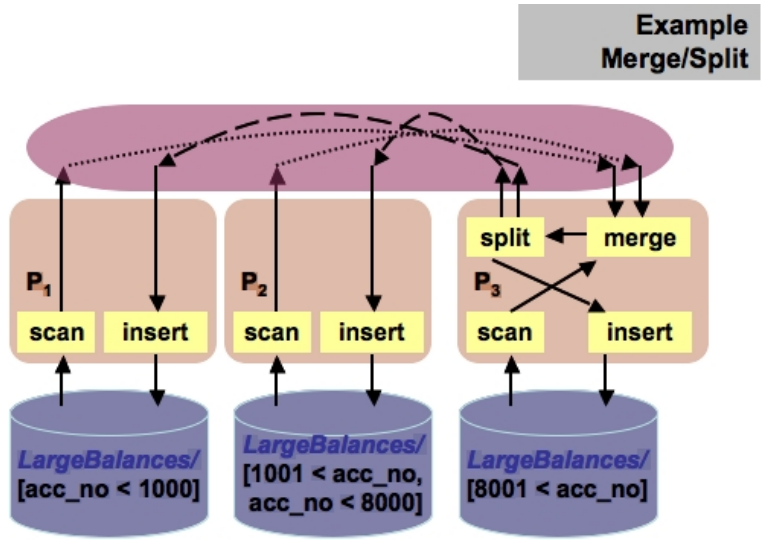
Example Merge



Parallelizing Relational Operators (6)

Merge/Split: An Example

- Assume now that *LargeBalances* is meant to be split across the storage elements according to some splitting function with a range whose cardinality is three.
- Both scans and inserts are replicated and run in parallel.
- The merge coalesces the results of the scans and feeds the split.
- The split redistributes the results of the merge to feed the inserts.
- Again, the merge and split could run anywhere, even separately.
- Note that merge and split take responsibility for flow of control and data.
- For example, they buffer, and then block producers if buffers are full until consumption resumes.



Parallelizing Relational Operators (7)

Merge-Split Parallel Join: An Example

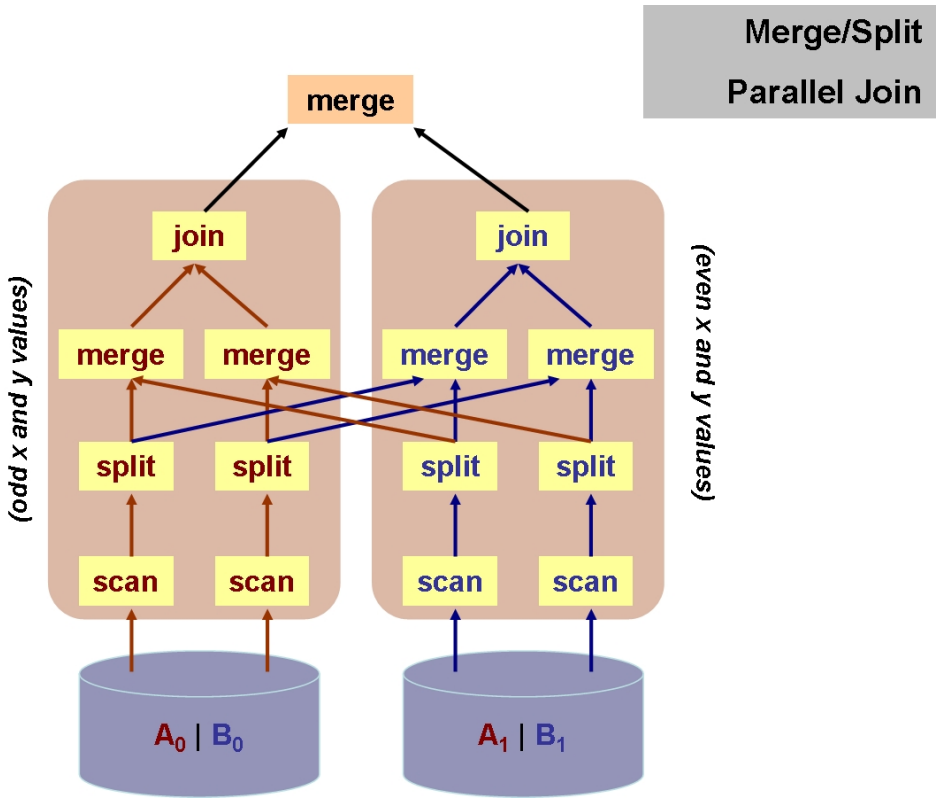
Example

Assume the following query:

$q_4 =$

```
SELECT   *
FROM     A, B
WHERE    A.x = B.y
```

- The figure shows how a join can be parallelized.
- Splits send odd values of the join attribute to the left processing element and even ones to the right one.
- The topmost merge may turn out to be just a union operator.



Parallelizing Relational Operators (8)

In Practice

- There will be lots of processes, lightweight threads are better.
- Merge is handled automatically by communication mechanisms (e.g., sockets).
- Split is tacked onto the producing process themselves (e.g., scan or join)
- It is possible, and elegant, to encapsulate merge and split as two-parts of a single operator, known in the literature as an **exchange** operators.
- Exchange operators further abstract away from low-level mechanisms, which turns out to be useful (as we will see later on).

Summary

Query Processing Techniques in Parallel DBMSs

- The strategies for processing queries in parallel in DBMSs are remarkably simple.
- They require no disruption of the classical operators, just three additional mechanisms: one to replicate operators, one to split data to feed them and one to merge the results of the replicated operators.
- These mechanisms scale well and compose with few constraints.
- They can be used to parallelize query execution, with linear speed-up and scale-up being more achievable than in most other areas.

Summary

Parallel Database Management Systems

- Parallel DBMSs can be seen to have been the first major class of complex software systems to be satisfactorily capable of benefitting from parallelization.
- The role of the relational model and algebra in this development was paramount.
- As high-quality, low-cost commodity hardware became the norm and as software techniques evolved to enable abstraction from low-level communication mechanisms, parallel DBMSs capitalized elegantly on such developments.
- It is clearer than ever that most data management systems will rely heavily on massively-parallel designs, which we will explore later on in the course.

This Week: Summary

- We have looked in detail into logical optimization, i.e., heuristic-based rewriting of algebraic plans.
- We have explored some of the basic strategies for designing physical operators and for combining them into query evaluation plans (QEPs).
- We have briefly discussed some of the challenges in cost-based QEP selection, in particular the hard problem of choosing a join order.
- We have found out how data partitioning strategies, allied with the algebraic properties of query languages, make it possible to automatically parallelize QEP execution.

Next Week: Topics

- We will take a quick, example-driven tour of XQuery, to remind ourselves of the basic characteristics of the language,
- We will introduce and briefly explore an algebraic view of XQuery, and some equivalence laws that arise, on the basis of which one can conceive of a rewriting approach to logical optimization for XQuery.
- We will then study storage, optimization and evaluation of XQuery as implemented in one specific XML native DBMS, viz., BaseX.

Next Week: Learning Objectives

- Having completed our exploration of classical query processing, we are now ready to use what we have learnt and explore querying data on the Web, in the narrower sense, starting with XML/XQuery.
- We will aim to acquire a deeper understanding of some of the challenges arising in querying over large XML collections using XQuery.

Part III

Supplementary Material on Relational Query Processing

Supplementary Material: Outline

8 Plan Cost Estimation in Query Processing
- Metadata in the System Catalogue
- Plan Cost Estimation

9 Parallel Query Processing: Background
- Historical Background
- Parallelism: Goals and Metrics
- Parallel Architectures

Supplement 1

Plan Cost Estimation in Query Processing

Section 1

Metadata in the System Catalogue

The System Catalogue (1)

- A relational DBMS maintains descriptive statistical information about every table and index that it contains.
- This information is itself stored in a collection of special tables called the **catalogue tables**.
- The catalogue tables are also commonly referred to as the **(system) catalogue** and the **data dictionary**.
- It records information about users and about the contents.

The System Catalogue (2)

Metadata on Tables, Indexes and Views

- For each table, the catalogue typically stores:
  - The table name, the file name in which the table is stored and the organization (e.g., heap file) of the file
  - The name and type of each attribute
  - The name of every index on the table
  - The integrity constraints on the table
- For each index, the catalogue typically stores:
  - The name and organization (e.g., B+ tree) of the index
  - The search-key attributes
- For each view, the catalogue typically stores:
  - The name of the view
  - The definition of (i.e., the query used to compute) the view

The System Catalogue (3)

Statistical Information (1)

Cardinality The number of tuples $T(R)$ for each relation R.

Size The number of blocks $B(R)$ for each relation R.

Index Cardinality The number of distinct key values $NKeys(I)$ for each index I.

Index Size The number of blocks $B(I)$ for each index I.

Index Height The number of non-leaf levels $Ih(I)$ for each tree index I.

Index Range The minimum present key value $IL(I)$ and the maximum present key value $IH(I)$ for each index I.

The System Catalogue (4)

Statistical Information (2)

- Statistical information is updated periodically.
- Updating statistical information every time the data is updated is too expensive.
- A great deal of approximation takes place anyway, so slight inconsistency or staleness is not overly damaging, most of the time.
- More detailed information (e.g., histograms of the value distribution for some attributes) are sometimes stored.

Section 2

Plan Cost Estimation

The Need for Cost Estimates

- It should be clear from the previous material that:
  1. There are many different equivalent algebraic representations of a typical query.
  2. There are often several different algorithms for implementing algebraic operators.
- A query optimizer has to be able to select efficient plans from the many options available.
- This is done by estimating the cost of evaluating alternative expressions.

Dominant Costs

Memory Access Dominates, and Secondary Memory Dominates Primary Memory

- It takes typically 10^{-7} to 10^{-8} seconds to access a word in primary memory.
- It takes typically 10^{-2} seconds to read a block from secondary memory into primary memory.
- As access to main memory is so much faster than access to disk, many cost models assume that I/O cost is dominant in query processing.
- In this course, we focus on the I/O cost for different operations.
- In so doing, the general assumption is that the inputs of a physical operator are read off disk, but that the outputs are not written back onto disk.
- Later, we will also consider transfer costs over interconnects.

Cost Model Parameters

Main Memory M denotes the number of main memory buffers available to an operator.

A buffer is the same size as a disk block.

Relation Size $B(R)$ is the number of blocks need to store the number of tuples $T(R)$ (or just T) in relation R.

It is assumed that data is read from disks in blocks, and that the blocks of a relation are clustered together.

Value Distributions $V(R, a)$ denotes the number of distinct values that appear in the a column of R.

$V(R, [a_1, \ldots, a_n])$ is the number of distinct n-tuples for the columns $a_1, \ldots, a_n$.

Example Schemas

Example

Flights (fltno: string, from: string, to: string, dep: date, arr: date)

UsedFor (planeid:string, fltid: string, weekday: string)

Usable (flid:string, pltype: string)

Certified (pilid:string, planetype: string)

Estimating I/O Costs (1)

Scan

- The algorithm for `Scan` given earlier reads a block at a time from disk.
- Thus, for a table R, if the tuples are clustered, the number of disk I/Os is $B(R)$.
- If R is not clustered (e.g., because its tuples have been deliberately clustered with another relation), there could be as many as $T(R)$ I/Os required to scan R.

Estimating I/O Costs (2)

Nested-Loop Join

- Assuming $R \bowtie S$, the amount of I/O required by a nested loop join depends on the sizes of R and S relative to the available memory.
- In the algorithm given earlier:
  - If $B(R) < M$ then I/O cost $= B(R) + B(S)$.
  - In general, the smaller of the operand relations is used in the inner loop (assume R in what follows).
  - If only one buffer is available to each relation, then I/O cost $= B(S) + T(S) \times B(R)$.
  - The outer relation, S, is read only once, but the inner relation, R, is read once for every tuple in S.

Estimating I/O Costs (3)

Hash Join

- The one-pass hash join is very dependent on the hash table fitting in main memory (otherwise the algorithm causes thrashing).
- For $R \bowtie S$, for the algorithm given earlier, if $B(R) < M$ then I/O cost $= B(R) + B(S)$.
- Thus, if there is plenty of memory, the I/O costs of hash join and nested-loop join are the same.
- But note that the number of tuple comparisons in nested loop is $T(R) \times T(S)$, whereas it is generally nearer to $T(S)$ in hash-join.
- For the two pass hash join, the I/O cost is: $3(B(R) + B(S))$.
- This is because each block is read, then the corresponding hashed partition is written, then each block is read one more time during the matching phase of the join.

Estimating Sizes (1)

The Problem of Intermediate Results

- So far, we have assumed that the sizes of the operands are known.
- Data dictionaries generally store size and cardinality details for stored data, so I/O costs can be estimated for operations acting on base relations (i.e., the leaf nodes in a QEP).
- However, the size and cardinality of intermediate relations depends both on the inputs and on the operation that generates them.
- This is so, for every non-leaf node, and recursively for its non-leaf child(ren).
- Although the size and cardinality of intermediate results cannot be known for certain when queries are optimized, they are important for cost estimation, and hence for plan selection.
- Thus, the sizes of intermediate data sets must be estimated, based on the nature of the operators and on known properties of their inputs.

Estimating Sizes (2)

Projection

- The size of the result of a projection can be computed directly from the size of its input.
- Given the function $length(A)$ which computes the (average) number of bytes occupied by the list of attributes $A = a_1, \ldots, a_n$:

$$B(\pi_A(R)) = \frac{B(R) \times length(A)}{length(R)}$$

Example

Given $\pi_{from,to}(Flights)$, if *Flights* occupies 500 blocks, and *from* and *to* each, in average, occupies 25 bytes from a total of 100 bytes in a *Flights* tuple, then:

$$B(\pi_{from,to}(Flights)) = \frac{500 \times 50}{100} = 250 \; blocks$$

Estimating Sizes (3)

Selection (1)

- Unlike projection, any computation of the size of a selection really is an estimate.
- There are several cases depending on the form of predicate θ.
- In the following, A is an attribute in R, and c is a constant.

Estimating Sizes (4)

Selection (2): Typical Cases

- $S = \sigma_{A=c}(R)$: Given statistics on the number of distinct values for A in R:

$$T(S) = \frac{T(R)}{V(R, A)}$$

where the denominator is the **selectivity (factor)** of a σ operation, i.e., the proportion of tuples that it retains, so if $\sigma_{A=c}(R)$ then $sel(\sigma_{A=c}(R)) = \frac{1}{V(R,A)}$.

- $S = \sigma_{A<c}(R)$: One estimate could be that in practice half the tuples satisfy the condition, another that a smaller proportion (say, a third) do:

$$T(S) = \frac{T(R)}{2}$$

or

$$T(S) = \frac{T(R)}{3}$$

Estimating Sizes (5)

Selection (3): Compound Conditions

- Compound conditions must combine the selectivity factors of the component conditions, e.g.:
  - $S = \sigma_{\theta_1 \wedge \theta_2}(R)$: Given the splitting laws, this can be treated as a succession of simple selections.
  - The effect is to obtain an overall selectivity factor by multiplying the selectivity factors of each condition.

$$\sigma_{A=c_1 \wedge B=c_2}(R) \Leftrightarrow \sigma_{A=c_1}(\sigma_{B=c_2}(R)) \rightarrow \frac{1}{V(R,A)} \times \frac{1}{V(R,B)}$$

  - $S = \sigma_{\theta_1 \vee \theta_2}(R)$: One possibility is to assume that no tuple satisfies every condition, which leads to the overall selectivity factor being the sum of the selectivity factors of individual conditions.
  - When multiplied with $T(R)$, this could yield a cardinality estimate greater than $T(R)$, in which case we use $T(R)$ as the estimate for the cardinality of the output.

Estimating Sizes (6)

Selection (4)

Example

Recall the *Usable* table, and let its instance **U1** =

flid	pltype
BA83	A319
BA83	737
BA85	A319
DE87	767
DE89	767

Given $T(Usable) = 5$, $V(Usable, flid) = 4$, $V(Usable, pltype) = 3$, then:

- $T(\sigma_{flid=BA83}(Usable)) = \frac{1}{4} \times 5 = 1.25$
- $T(\sigma_{flid=BA83 \wedge pltype=A319}(Usable)) = \frac{1}{4} \times \frac{1}{3} \times 5 = 0.42$.

Estimating Sizes (7)

Join (1)

- Given a join of R with schema (X, Y) and S with schema (Y, Z) on a single attribute Y, there are various outcomes possible:
  1. The relations have disjoint sets of Y values, in which case $T(R \bowtie_Y S) = 0$.
  2. Y may be the key of S and a foreign key of R, so each tuple of R joins with one tuple of S, in which case $T(R \bowtie_Y S) = T(R)$.
  3. Almost all tuples of R and S have the same Y value, in which case $T(R \bowtie_Y S) \approx T(R) \times T(S)$.
- Thus, the possible range of size estimates for a join is very wide, although the second outcome above is very common in practice.
- Recall, when we speak of a single attribute Y that the actual name in the schema of each relation may differ (i.e., may be different at the syntactic level).

Estimating Sizes (8)

Join (2): Assumptions

- The following assumptions are made regarding the value sets:

  Containment
  - If Y is an attribute appearing in several relations, then each relation takes its values from the front of a list $y_1, y_2, y_3, \ldots$ of values in $dom(Y)$ and has all the values in that prefix.
  - As a consequence, if $V(R, Y) \leq V(S, Y)$ then every Y-value of R will be a Y-value of S.

  Preservation If A is an attribute of R, but not of S, then
  $$V(R \bowtie S, A) = V(S \bowtie R, A) = V(R, A).$$

- Both of these conditions are satisfied when Y is a key of S and a foreign key of R.

Estimating Sizes (9)

Join (3)

- Given the previous assumptions, and further assuming that $V(R, Y) \leq V(S, Y)$:
  1. Every tuple $t \in R$ has a chance equal to $\frac{1}{V(S,Y)}$ of joining with a given tuple of S.
  2. As there are $T(S)$ tuples in S, t can be expected to join with $\frac{T(S)}{V(S,Y)}$ tuples from S.
  3. As there are $T(R)$ tuples in R

$$T(R \bowtie S) = \frac{T(R) \times T(S)}{V(S, Y)}$$

Estimating Sizes (10)

Join (4)

- A symmetrical argument gives

$$T(R \bowtie S) = \frac{T(R) \times T(S)}{V(R, Y)}$$

- In general, the larger divisor is used

$$T(R \bowtie S) = \frac{T(R) \times T(S)}{max(V(R, Y), V(S, Y))}$$

Estimating Sizes (11)

Join (5)

Example

Recall the *Usable* and *Certified* tables, and the instance **U1** of the former. Let the following be an instance of the latter, **C1** =

pilid	planetype
Smith	A319
Jones	737
Atkinson	A319
Smith	737

Given the previous metadata about *Usable* and $T(Certified) = 4$, $V(Certified, planetype) = 2$, then (abbreviating names):

$$T(U \bowtie C) = \frac{T(U) \times T(C)}{max(V(U, pltype), V(C, planetype))} = \frac{5 \times 4}{max(3, 2)} = 6.7$$

Summary

Cost Estimation

In query optimization:

1. Cost estimation is important, as it is necessary to be able to distinguish between plans.
2. I/O cost is often considered to be the dominant cost, which is incurred from accesses to base relations and from disk storage of intermediate results.
3. Identifying the sizes of intermediate results involves estimates based on information stored in the data dictionary.

Supplement 2

Parallel Query Processing: Background

Section 1

Historical Background

Parallel Database Management Systems (1)

Pre-1985±

- (Beware that, this chunking into decades is highly idealized: nothing is ever as simple as that.)
- Successive failed attempts were made.
- The causes were of two main kinds:
  - Too much hope was placed on solving the problem with specialist hardware (e.g., special non-volatile primary memory, such as bubble memory, or specially-fast secondary memory, such as head-per-track disk drives).
  - Key software mechanisms were not yet widely available (e.g., message-passing techniques, client-server protocols, etc.).

Parallel Database Management Systems (2)

1985-1995±

Breakthroughs were finally made in the wake of:

- the acceptance of the relational model,
- the commoditization of hardware components (i.e., mainframes were dislodged by PCs from their previous central role), and
- progress in basic software infrastructure.

Parallel Database Management Systems (3)

1995-2005±

- Shared-nothing parallelization becomes the dominant approach.
- Most challenges have moved from query execution to query optimization.
- (Wait a bit more for what goes on from 2005 onwards.)

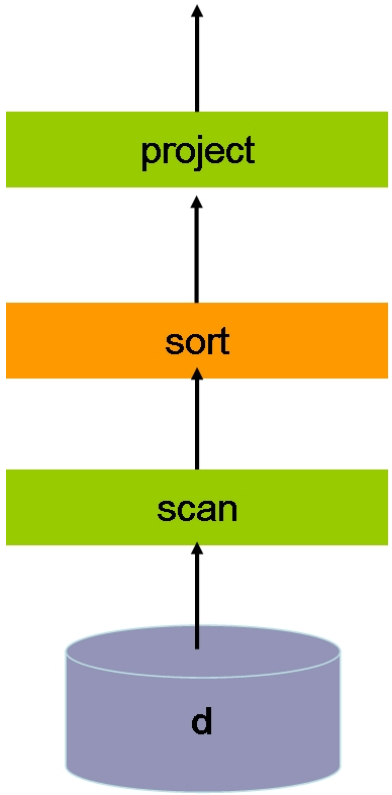
The Impact of the Relational Model

- The domination of the commercial market by relational DBMS allowed a focussing of minds and money.
- Relational queries are ideally suited to parallel execution:
  - Queries are a composition of a small collection of semantically well-understood, type-uniform operators applied to streams of data of a single underlying collection type.
  - The relational algebra is closed: each operator consumes one or more relations as input and produces a new relation as output.
  - Many operations are non-blocking (i.e., can produce a result based only on the last tuple(s) read) and those that are not (e.g., join) are easier to parallelize than most other algorithms.
  - Two forms of parallelism are made possible: **pipelined** and **partitioned**.
  - These are also referred, in the database literature, as **inter-operator** and **intra-operator** parallelism, respectively.

Parallelism in Relational Query Execution (1)

Pipelined Parallelism

- With pipelining, each operator can execute in its own thread of control.
- However, if the operator has blocking semantics (i.e., if it needs to read all the tuples of any input before it can produce an output tuple), then pipelined-parallel execution yields limited benefits.
- In the figure, while the **scan** and **project** operators can take good advantage of pipelined parallelism, the **sort** operator blocks, thereby limiting the overall benefits.

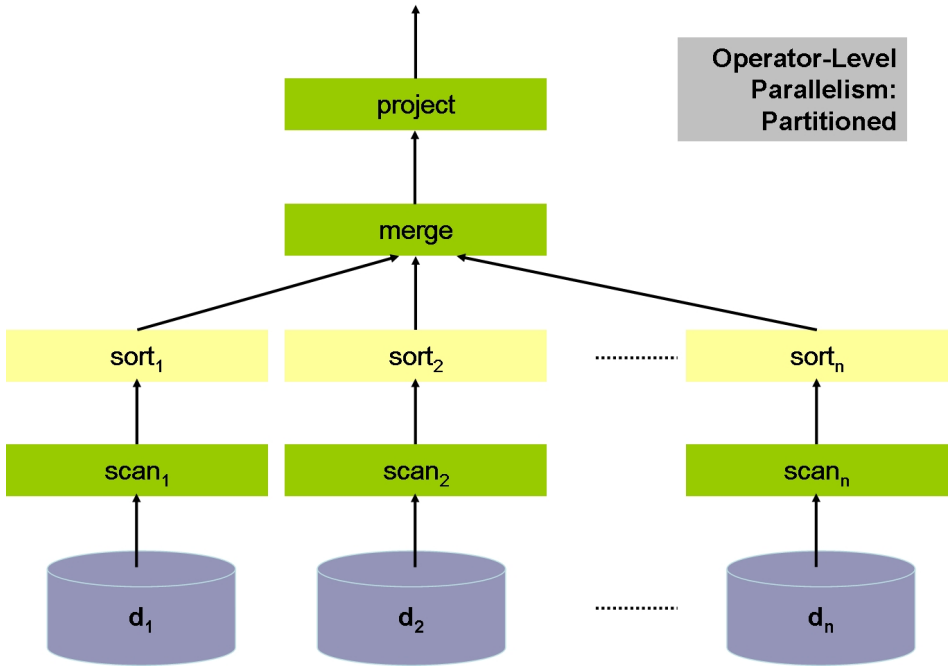


Operator-Level Parallelism: Pipelined

Parallelism in Relational Query Execution (2)

Partitioned Parallelism

- With partitioning, a plan fragment is replicated to execute on separate resources with their separate share of the overall load.
- The benefits can be much more significant.
- In the figure, the **sort** operator still blocks, but, assuming loads are balanced (i.e., that data partitions are assigned intelligently), the negative impact of the blocking behaviour is much reduced.



Commoditization of Hardware

- It has become increasingly difficult to build mainframe computers powerful enough to satisfy CPU and I/O demands of many large applications.
- Clusters based on relatively-small processing units became easier to build, and they:
  - provide more total power at a lower price,
  - have a modular architecture that allows for incremental growth, and
  - employ commodity components.

Enabling Software Technologies

- Tools for client-server computing are now commonplace, e.g. (remote procedure call mechanisms at various grains of functionality).
- Networking software is now commonplace (e.g., over Internet protocols).
- DBMSs themselves have evolved enough that it is commonplace to encapsulate them as components (e.g., using various kinds of connectivity middleware).

Summary

Historical Background

- DBMSs offer the opportunity for both pipelined and partitioned parallelism.
- The abstract nature of relational languages and their minimalism, elegance and constrained expressiveness make database operations easier to parallelize than general computations.
- With the rise of the relational model, the availability of high-performance commodity hardware and network, and the development of powerful, general-purpose software mechanisms for making use of the latter, parallel DBMSs became easier to build.

Section 2

Parallelism: Goals and Metrics

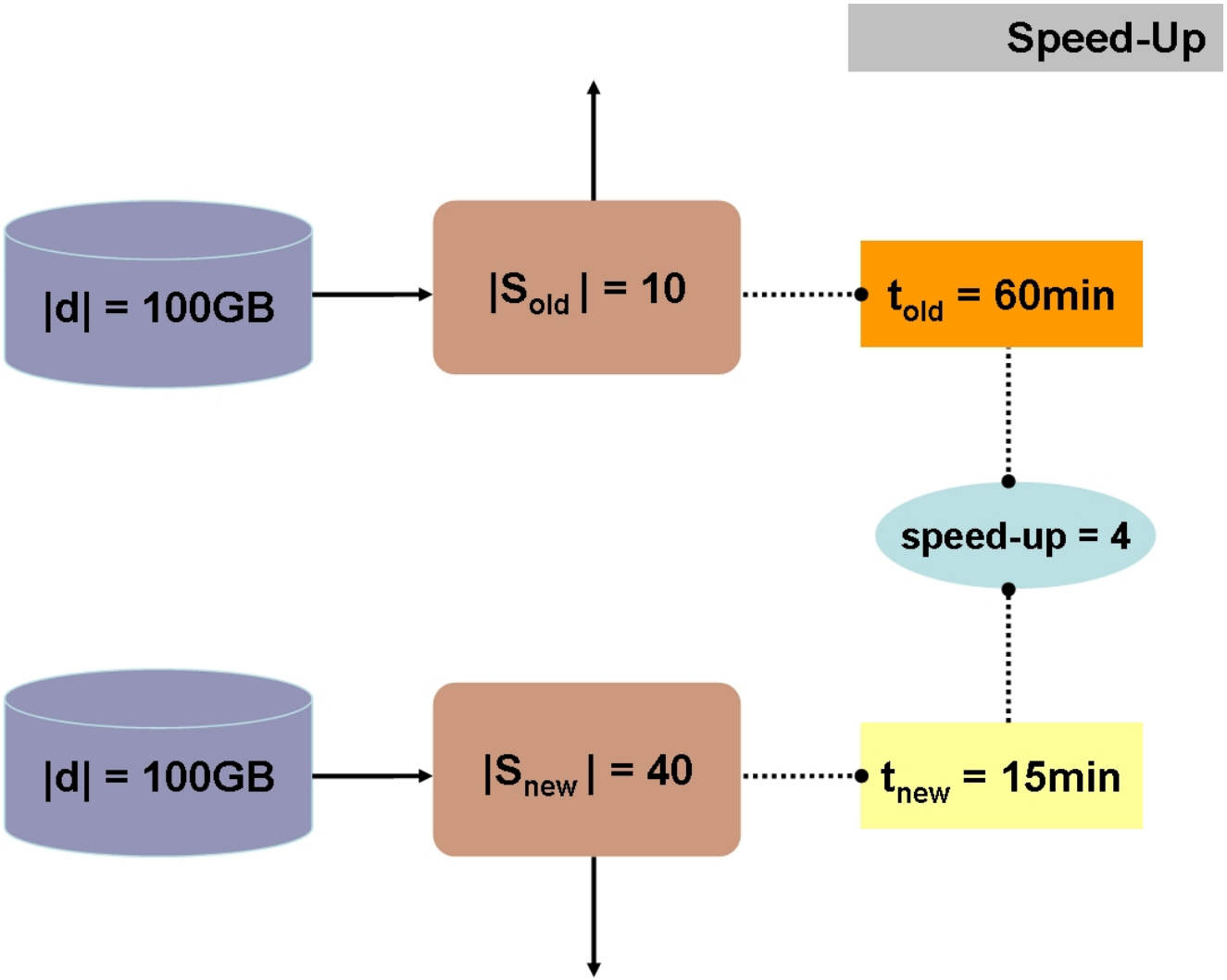
Parallelism Goals and Metrics (1)

Speed-Up (1)

- One goal is **linear speed-up**, e.g., twice as much hardware can perform the same task in half the elapsed time.
- A speed-up design performs a job that took elapsed time t in a system of size s in elapsed time $t \div k$ in a system of size $k \times s$.
- The **speed-up** is the ratio between the elapsed time in the old, smaller system and the elapsed time in the new, larger system.
- Speed-up holds the problem size constant and grows the system.

Parallelism Goals and Metrics (2)

Speed-Up (2)



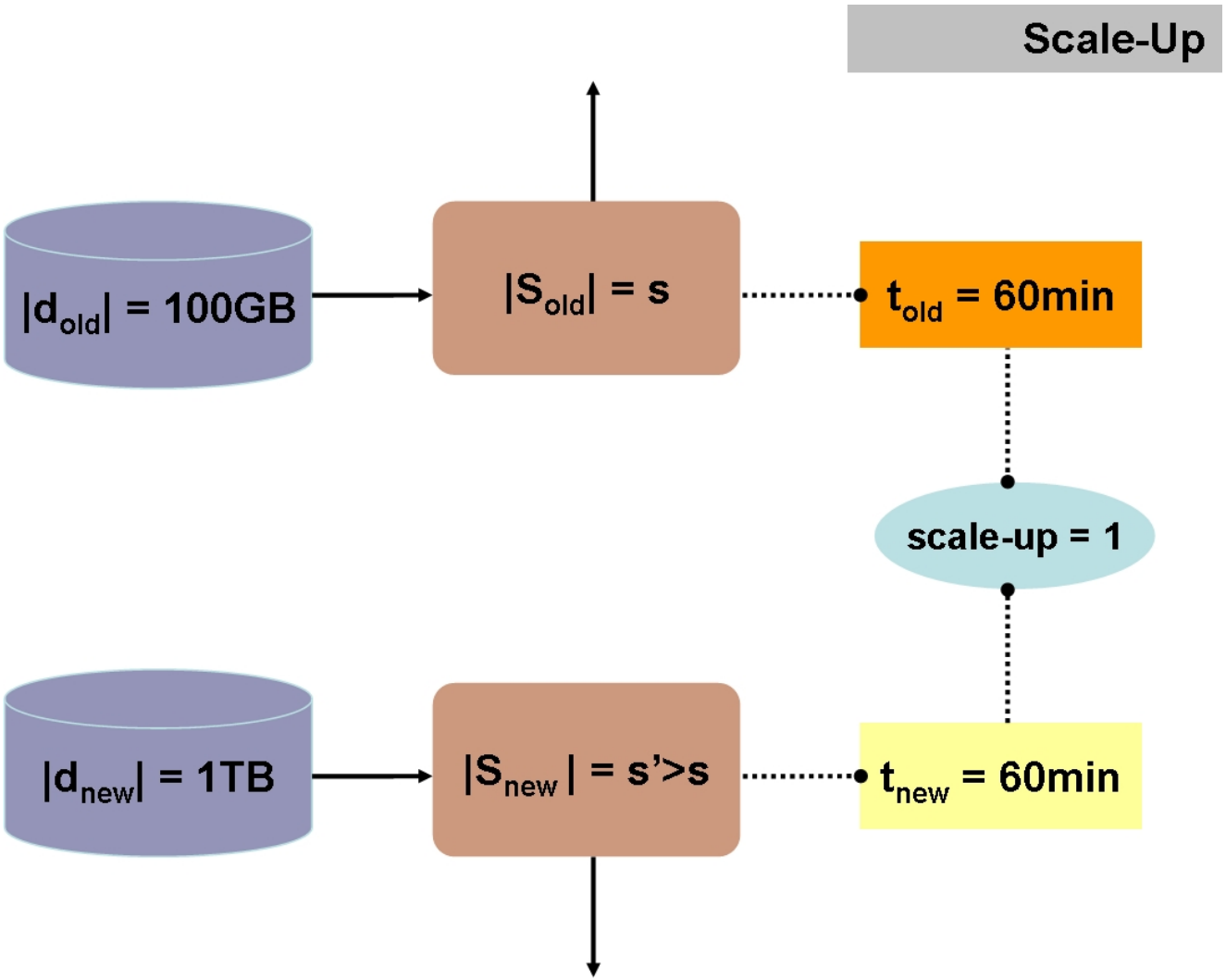
Parallelism Goals and Metrics (3)

Scale-Up (1)

- The **scale-up** is the ratio between the elapsed time in the old, smaller system on the old, smaller problem and the elapsed time in the new, larger system on the new, larger problem.
- Scale-up measures the ability to grow both the problem and the system.
- Another goal, therefore, is **linear scale-up**, e.g., growing the system in response to the growth of the problem achieves a scale-up equal to 1.
- A scale-up design performs a k-times larger job in the same elapsed time as it takes a k-times larger system.
- (We will consider the notion of scale-out later.)

Parallelism Goals and Metrics (4)

Scale-Up (2)



Parallelism Goals and Metrics (5)

Barriers to Linear Speed-Up and Scale-Up

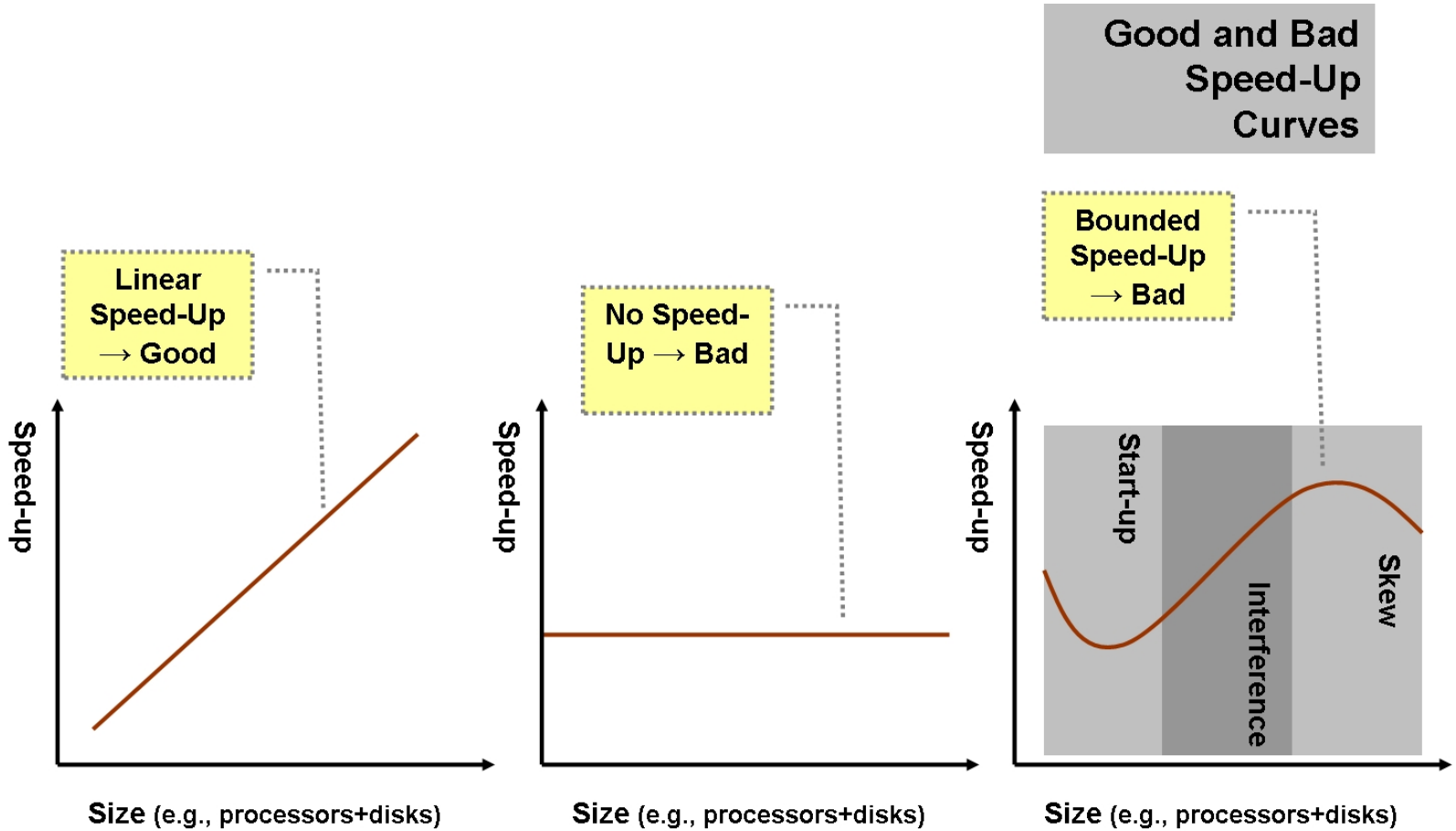
start-up is the time needed to start a parallel operation: too many start-ups can come to dominate the processing time.

interference is the slow down each new parallel operation imposes on all others due to increased competition for shared resources.

skew as the number of parallel operations increase, the average size of each step decreases, but the variance grows significantly: the time taken ends up being the time taken by the slowest step.

Parallelism Goals and Metrics (6)

Good and Bad Speed-Up Curves



Summary

Parallelism Goals and Metrics

- Parallelization aims to achieve linear speed-up and linear scale-up.
- Often overheads caused by start-up costs, interference and skew lead to sub-linear behaviour.

Section 3

Parallel Architectures

Parallel Architectures (1)

The Ideal and the Approximation of the Ideal

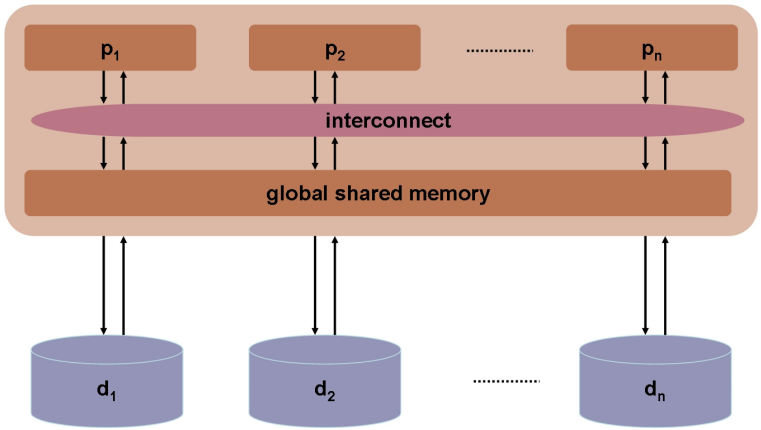
- The ideal environment has infinitely-fast processing with infinite memory and infinite bandwidth.
- The challenge is to approximate this ideal out of a large number of components with finite capabilities.
- In other words, a very fast processing capability out of very many processors of individually-limited capability, and a very large store with very large bandwidth out of very many memory and disk units of individually-limited capability.
- In the DBMS arena, the spectrum of possible designs is describable with three cases:
  - shared-memory designs
  - shared-disk designs
  - shared-nothing designs

Parallel Architectures (2)

Shared-Memory Designs

- All the processors share direct access to a common global memory and to all disks.
- The limitations to scaling are:
  - The bandwidth of the interconnect must equal the sum of the processors and disks, which is hard to achieve in large scales.
  - Severe shared-resource interference (e.g., lock tables, buffer access), which is hard to avoid.
  - Cache hit rates must be high, which is hard to ensure.
- The response is a move towards **affinity scheduling** (i.e., each process has a propensity to use a certain processor).

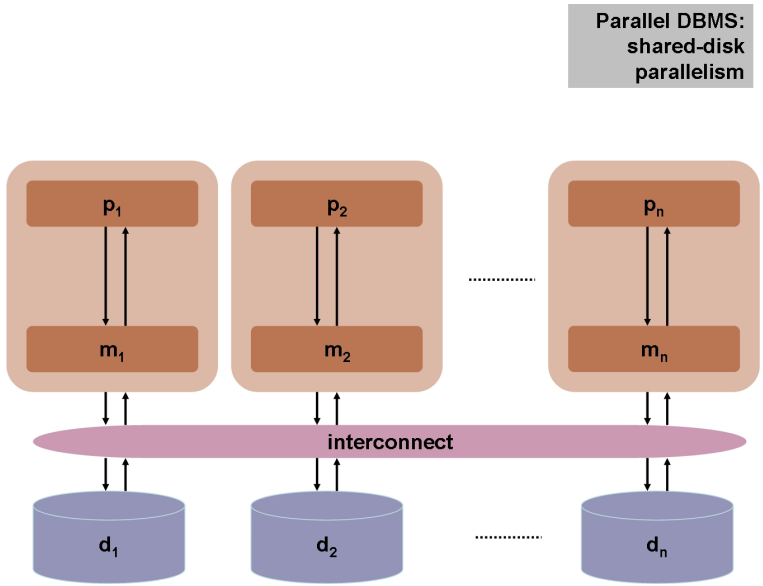
Parallel DBMS: shared-memory parallelism



Parallel Architectures (3)

Shared-Disk Designs

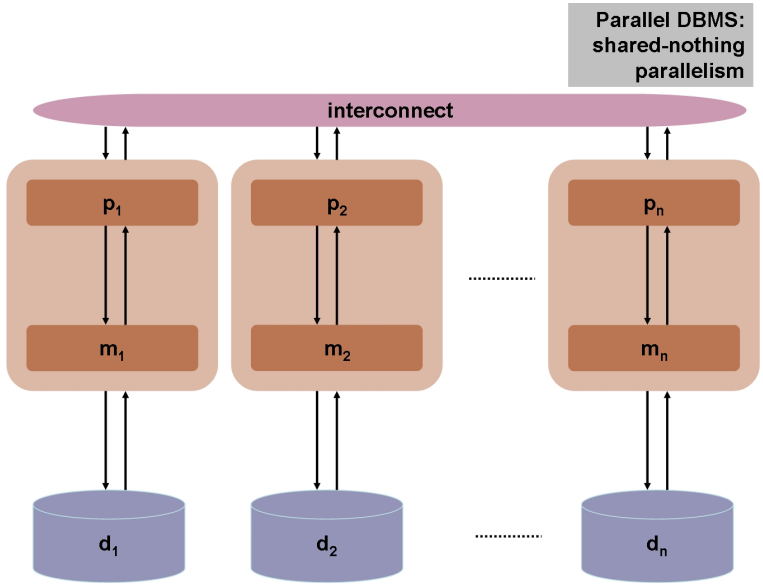
- Each processor has direct access to its private memory but shares direct access to all disks.
- One limitation to scaling in this case is that the software mechanisms required to coordinate low-level access to shared data in the presence of updates are complex, and more so in larger scales.
- Interference may become, here too, a major issue.
- The response is a move towards **data affinity** (i.e., each data item has a propensity to use a certain processor that, then, through low-level message exchange, serves the other processors).



Parallel Architectures (1)

Shared-Nothing Designs

- Each memory and disk unit is owned by some processor that acts as a server for that data.
- This offers the best hope of scaling because it minimizes the data that moves through the interconnect: in principle, only questions and answers do so.



Parallel Architectures (2)

The DBMS Case for Shared-Nothing Designs (1)

- The move in shared-memory designs towards affinity scheduling is a move towards implicit data partitioning.
- In this sense, it is a move towards shared-nothing designs, but incurs the same load balancing problems in the presence of skew without reaping the benefits of a simpler interconnect.
- It is easier to make the interconnect scale to many more units in shared-disk designs, but this only works well for read-only databases or databases with little concurrent sharing (e.g., some kinds of data warehouse).
- Otherwise, data affinity is needed and, again, this is a move towards shared-nothing designs.
- In shared-nothing designs, messages are exchanged at a much higher level (viz., of queries and answers) than in shared-disk designs.

Parallel Architectures (3)

The DBMS Case for Shared-Nothing Designs (2)

- The case for shared-nothing designs became more compelling as the availability of simple, high-performance, low-cost components grew.
- The case was made stronger by the ascendancy of the relational paradigm because the constrained expressiveness of the relational languages makes parallelization simpler than that of general computations.
- In particular, it is possible to take interesting SQL-based workloads written with a single processor in mind and execute them in parallel in shared-nothing architectures with near-linear speed-ups and scale-ups.

Summary

Parallel Architectures

- In databases, particularly those involving concurrent access to shared data in the presence of updates, shared-nothing architectures are often the best design.
- They minimize the burden on the interconnect since only queries and answers are exchanged.

References


Elmasri, R. and Navathe, S. B. (2006).
Fundamentals of Database Systems.
Addison-Wesley, 5^{th} edition.


Garcia-Molina, H., Ullman, J. D., and Widom, J. (2002).
Database Systems: The Complete Book.
Pearson Education Limited, 1^{st} edition.


Ramakrishnan, R. and Gehrke, J. (2003).
Database Management Systems.
McGraw-Hill Education - Europe, 3^{rd} edition.


Silberschatz, A., Korth, H. F., and Sudarshan, S. (2009).
Database System Concepts.
McGraw-Hill Education - Europe, 6^{th} edition.