Chapter 12: Query Processing

Database System Concepts, 6th Ed.

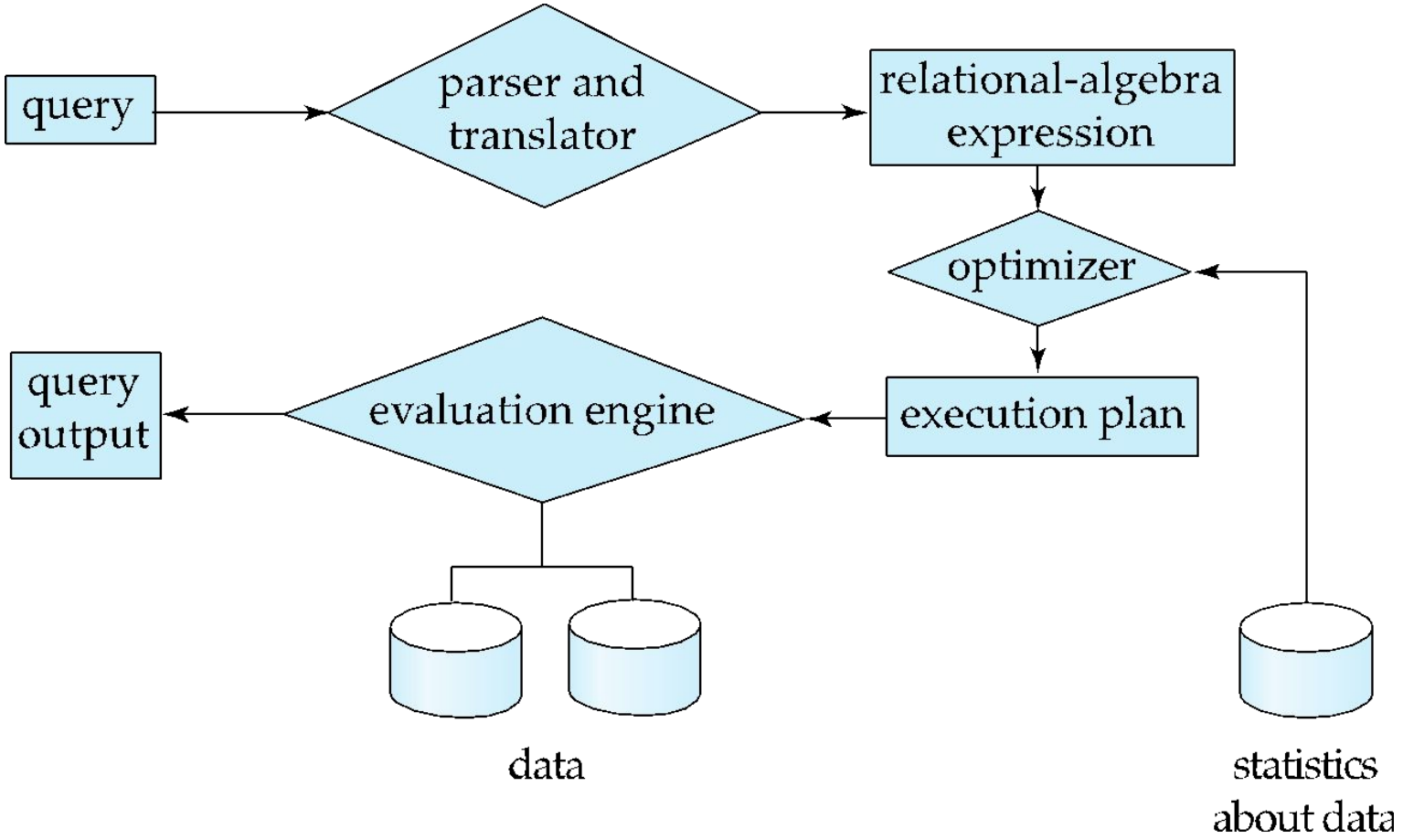
Chapter 12: Query Processing

- Overview
- Measures of Query Cost
- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions

Query Processing

- **Query processing** refers to the range of activities involved in extracting data from a database. The activities include
  - translation of queries written in high-level database languages (SQL) into expressions (Extended Relational Algebra) that can be used at the physical level of the file system,
  - a variety of query-optimizing transformations, and
  - actual evaluation of queries.
- So, the basic steps involved in processing a query are:
  - 1. Parsing and translation
  - 2. Optimization
  - 3. Evaluation
- Parsing and translation
  - System translate the query into its internal form. This is similar to the work performed by the parser of a compiler

Basic Steps in Query Processing



Basic Steps in Query Processing (Cont.)

- Parser checks syntax of the query, verifies relations names
- The system constructs a parse-tree representation of the query, which it then translates into a relational-algebra expression.

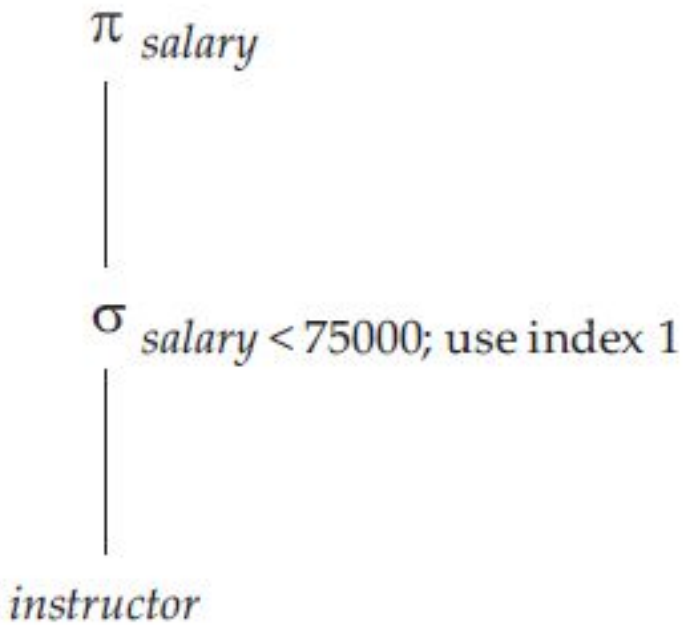
- Evaluation
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.
- Given a query, there are generally a variety of methods for computing the answer.
  - In SQL, a query could be expressed in several different ways.
  - Each SQL query can itself be translated into a relational algebra expression in one of several ways.
  - Furthermore, the relational-algebra representation of a query specifies only partially how to evaluate a query; there are usually several ways to evaluate relational-algebra expressions

Basic Steps in Query Processing

- A relational algebra expression may have many equivalent expressions
  - E.g., $\sigma_{salary<75000}(\Pi_{salary}(instructor))$ is equivalent to $\Pi_{salary}(\sigma_{salary<75000}(instructor))$
- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- It needs not only to provide the relational-algebra expression, but also to annotate it with instructions specifying how to evaluate each operation. 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 ≥ 75000

Basic Steps in Query Processing

- A relational algebra operation annotated with instructions on how to evaluate it is called an **evaluation primitive.**
- A sequence of primitive operations that can be used to evaluate a query is a **query-execution plan** or **query-evaluation plan.**
- The **query-execution engine** takes a query-evaluation plan, executes that plan, and returns the answers to the query.
- The different evaluation plans for a given query can have different costs.



A query-evaluation plan.

Basic Steps: Optimization (Cont.)

- **Query Optimization**: It is the responsibility of the system to construct a query evaluation plan that minimizes the cost of query evaluation; this task is called *query optimization.*
  - 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 individual operations (pipeline) in order to evaluate a complete expression
- In Chapter 16 (Query Optimization) 7th Edition
  - We study how to optimize queries, that is, how to find an evaluation plan with lowest estimated cost

Measures of Query Cost

- There are multiple possible evaluation plans for a query, and it is important to be able to compare the alternatives in terms of their (estimated) cost, and choose the best plan.
- To do so, we must estimate the cost of individual operations and combine them to get the cost of a query evaluation plan.
- Cost is generally measured as total elapsed time for answering query
  - Many factors contribute to time cost
    - *disk accesses,*
    - *CPU* time to execute a query,
    - network communication, or
    - in parallel and distributed database systems, the cost of communication

Measures of Query Cost

- For large databases resident on magnetic disk, the I/O cost to access data from disk usually dominates the other costs; thus, early cost models focused on the I/O cost.
- However, with flash storage becoming larger and less expensive, most organizational data today can be stored on solid-state drives (SSDs) in a cost effective manner.
- In addition, main memory sizes have increased significantly, and the cost of main memory has decreased enough in recent years so that organizational data can be stored cost-effectively in main memory for querying, although it must of course be stored on flash or magnetic storage to ensure persistence.
- With data resident in-memory or on SSDs, I/O cost does not dominate the overall cost, and we must include CPU costs when computing the cost of query evaluation.

Measures of Query Cost (Cont.)

- For simplicity we just use the **number of block transfers** from disk and the **number of random I/O accesses** as the cost measures
  - t_T – average time to transfer one block
  - t_S – average block access time (seek + rotational latency)
  - Cost for b block transfers plus S random I/O accesses

    $$b * t_T + S * t_S$$
- Typical values for high-end disks today (2018) would be t_S = 4 milliseconds and t_T = 0.1 milliseconds, assuming a 4-kilobyte block size and a transfer rate of 40 megabytes per second.
- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account

Selection Operation

- **File scan** :
  - In query processing, the **file scan** is the lowest-level operator to access data.
  - File scans are search algorithms that locate and retrieve records that fulfill a selection condition.
  - In relational systems, a file scan allows an entire relation to be read in those cases where the relation is stored in a single, dedicated file.
- 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
  - In case blocks of the file are not stored contiguously, extra seeks may be required, but we ignore this effect for simplicity.

A1	Linear Search	$t_S + b_r * t_T$	One initial seek plus b_r block transfers, where b_r denotes the number of blocks in the file.

Selection Operation

- If selection is on a key attribute, can stop on finding record
  - cost = $(b_r/2)$ block transfers + 1 seek

A1	Linear Search, Equality on Key	Average case $t_S + (b_r/2) * t_T$	Since at most one record satisfies condition, scan can be terminated as soon as the required record is found. In the worst case, b_r blocks transfers are still required.

  - Although it may be slower than other algorithms for implementing selection, 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 structures are referred to as **access paths**, since they provide a path through which data can be located and accessed.
- **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
  - $Cost = (h_i + 1) * (t_T + t_S)$

A2	Primary B+-tree Index, Equality on Key	$(h_i + 1) * (t_T + t_S)$	(Where h_i denotes the height of the index.) Index lookup traverses the height of the tree plus one I/O to fetch the record; each of these I/O operations requires a seek and a block transfer.

Selections Using Indices

- **A3** (**primary index, equality on nonkey**) Retrieve multiple records.
  - the records must be stored consecutively in the file since the file is sorted on the search key.
    - Let b = number of blocks containing matching records
  - $Cost = h_i * (t_T + t_S) + t_S + t_T * b$

A3	Clustering B^+-tree Index, Equality on Non-key	$h_i * (t_T + t_S) + t_S + b * t_T$	One seek for each level of the tree, one seek for the first block. Here b is the number of blocks containing records with the specified search key, all of which are read. These blocks are leaf blocks assumed to be stored sequentially (since it is a clustering index) and don't require additional seeks.

Selections Using Indices

- **A4** (**secondary index, equality on nonkey**).
  - Retrieve a single record if the search-key is a candidate key
  - Same cost as primary index, equality on 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 and the block fetches are randomly ordered
    - Cost = $(h_i + n) * (t_T + t_S)$
  - Can be very expensive! even worse than that of linear search if a large number of records are retrieved.

A4	Secondary B$^+$-tree Index, Equality on Nonkey	$(h_i + n) * (t_T + t_S)$	(Where n is the number of records fetched.) Here, cost of index traversal is the same as for A3, but each record may be on a different block, requiring a seek per record. Cost is potentially very high if n is large.

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

A5	Clustering B$^+$-tree Index, Comparison	$h_i * (t_T + t_S) + t_S + b * t_T$	Identical to the case of A3, equality on non-key.

Selections Involving Comparisons

- **A6** (**secondary index, comparison**).
  - A secondary ordered index can be used to guide retrieval for comparison conditions involving $<, \leq, \geq,$ *or* $>$
  - For $\sigma_{A \geq V}(r)$ use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records.
  - For $\sigma_{A \leq V}(r)$ just scan leaf pages of index finding pointers to records, till first entry $> v$
  - In either case, retrieve records that are pointed to may require an I/O for each record, since consecutive records may be on different disk blocks
    - If the number of retrieved records is large, linear file scan may be cheaper
    - the secondary index should be used only if very few records are selected.

A6	Secondary B$^+$-tree Index, Comparison	$(h_i + n) * (t_T + t_S)$	Identical to the case of A4, equality on nonkey.

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 A6 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 or record identifiers
  - 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.

Implementation of Complex Selections

- The cost of algorithm **A9** is the sum of the costs of the individual index scans, plus the cost of retrieving the records in the intersection of the retrieved lists of pointers.
- This cost can be reduced by sorting the list of pointers and retrieving records in the sorted order. Thereby,
  - 1. all pointers to records in a block come together, hence all selected records in the block can be retrieved using a single I/O operation, and
  - 2. blocks are read in sorted order, minimizing disk-arm movement.

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 access path (index) is available for *all* conditions
    - Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
    - Then fetch records from file
- Otherwise, if there is even one condition does not have an access path in the disjunction, the most efficient access method is a linear scan, with the disjunctive condition tested on each tuple during the scan.
- **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

Sorting

- Sorting of data plays an important role in database systems for two reasons.
- First, SQL queries can specify that the output be sorted.
- Second, equally important for query processing, several of the relational operations, such as joins, can be implemented efficiently if the input relations are first sorted.
- We may build an index on the relation, and then use the index to read the relation in sorted order. However, such a process orders the relation only *logically*, through an index, rather than *physically*.
- Hence, the reading of tuples in the sorted order may lead to a disk access (disk seek plus block transfer) for each record, which can be very expensive, since the number of records can be much larger than the number of blocks. For this reason, it may be desirable to order the records physically.

External Sort-Merge

- For relations that fit in memory, techniques like quicksort can be used. For relations that don't fit in memory, **external sort-merge** is a good choice.
- **External Sort-Merge Algorithm**
  - 1. In the first stage, a number of sorted **runs** are created
  - Let *M* denote the number of blocks in the main-memory buffer available for sorting, that is, the number of disk blocks whose contents can be buffered in available main memory.

```
i = 0;
repeat
    read M blocks of the relation, or the rest of the relation,
        whichever is smaller;
    sort the in-memory part of the relation;
    write the sorted data to run file R_i;
    i = i + 1;
until the end of the relation
```

External Sort-Merge (Cont.)

2. **Merge the runs (N-way merge)**. We assume (for now) that N (total number of runs) < M.
   - Use N blocks of memory to buffer input runs, and 1 block to buffer output. The merge stage operates as follows:

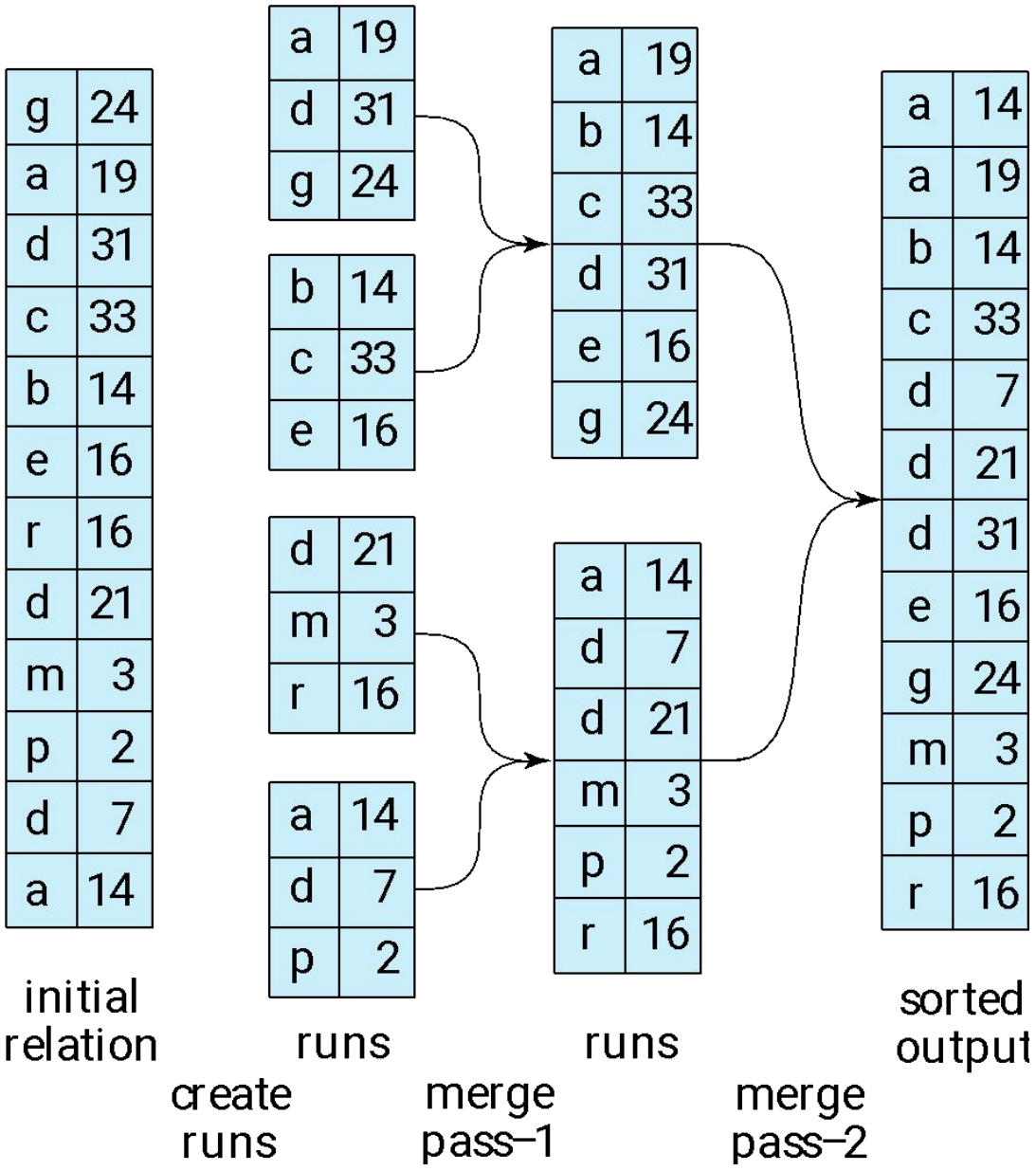
```
read one block of each of the N files Ri into a buffer block in memory;
repeat
    choose the first tuple (in sort order) among all buffer blocks;
    write the tuple to the output, and delete it from the buffer block;
    if the buffer block of any run Ri is empty and not end-of-file(Ri)
        then read the next block of Ri into the buffer block;
until all input buffer blocks are empty
```

   - The output of the merge stage is the sorted relation. The output file is buffered to reduce the number of disk write operations.
   - This merge operation is a generalization of the two-way merge used by the standard in-memory sort– merge algorithm; it merges N runs, so it is called an ***N-way merge.***

External Sort-Merge (Cont.)

- If $N \geq M$, several merge *passes* are required.
  - In each pass, contiguous groups of M - 1 runs are merged.
  - A pass reduces the number of runs by a factor of M -1, and creates runs longer by the same factor.
    - E.g. If M=11, and there are 90 runs, one pass reduces the number of runs to 9, each 10 times the size of the initial runs
  - If this reduced number of runs is still greater than or equal to M, another pass is made, with the runs created by the first pass as input.
  - Repeated passes are performed till all runs have been merged into one.
- For an example relation (next slide), we assume that only one tuple fits in a block ($f_r = 1$), and we assume that memory holds at most three (3) blocks. During the merge stage, two (2) blocks are used for input and one (1) for output.

Example: External Sorting Using Sort-Merge



External Merge Sort (Cont.)

- Cost analysis:
  - 1 block per run leads to too many seeks during merge
    - Instead use b_b buffer blocks per run
      - read/write b_b blocks at a time
    - Can merge $\lfloor M/b_b \rfloor - 1$ runs in one pass
  - Total number of merge passes required: $\lceil \log_{\lfloor M/bb \rfloor - 1}(b_r/M) \rceil$.
  - Block transfers for initial run creation as well as in each pass is $2b_r$
    - for final pass, we don't count write cost
      - we ignore final write cost for all operations since the output of an operation may be sent to the parent operation without being written to disk
    - Thus total number of block transfers for external sorting:

$$b_r \left(2 \lceil \log_{\lfloor M/bb \rfloor - 1} (b_r / M) \rceil + 1 \right)$$

  - Seeks: next slide

External Merge Sort (Cont.)

- Cost of seeks
  - During run generation: one seek to read each run and one seek to write each run
    - $2 \lceil b_r / M \rceil$
  - During the merge phase
    - Need $2 \lceil b_r / b_b \rceil$ seeks for each merge pass
      - except the final one which does not require a write
    - Total number of seeks:
      $$2 \lceil b_r / M \rceil + \lceil b_r / b_b \rceil (2 \lceil \log_{\lfloor M/bb \rfloor - 1}(b_r / M) \rceil - 1)$$
- YouTube Video

https://www.youtube.com/watch?v=AfF6ftMUNwo

Join Operation

- **equi-join** to refer to a join of the form $r \bowtie_{r.A=s.B} s$, where A and B are attributes or sets of attributes of relations r and s, respectively.
- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Examples use the following information: for *student* $\bowtie$ *takes*
- *student (ID, name, dept_name, tot_cred)*
- *takes (ID, course_id, sec_id, semester, year, grade)*
  - Number of records of *student*: $n_s = 5{,}000$ *takes*: $n_t = 10{,}000$
  - Number of blocks of *student*: $b_s = 100$ *takes*: $b_t = 400$

Nested-Loop Join

- To compute the theta join $r \bowtie_\theta s$
- r is called the **outer relation** and s the **inner relation** of the join.
- Algorithm:

  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
- Like linear file-scan, it 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.
- Natural join is simple extension, since it can be expressed as a theta join followed by elimination of repeated attributes by a projection.

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

- In the **best** case, there is enough space for both relations to fit simultaneously in memory, so each block would have to be read only once
  - Reduces cost to $b_r + b_s$ block transfers and 2 seeks
- If one of the relations fits entirely in main memory, it is beneficial to use that relation as the inner relation, since the inner relation would then be read only once.
  - Cost is $b_r + b_s$ block transfers and 2 seeks
  - The same cost as that for the case where both relations fit in memory

Nested-Loop Join (Cont.)

- Now consider the natural join of *student* and *takes*. Assume for now that we have no indices whatsoever on either relation, and that we are not willing to create any index.
- We can use the nested loops to compute the join; assume that *student* is the outer relation and *takes* is the inner relation in the join.
- Assuming worst case memory availability cost estimate is
  - Assume t_S = 4 milliseconds and t_T = 0.1 milliseconds.
  - with *student* as outer relation:
    - 5000 * 400 + 100 = 2,000,100 block transfers, (200,010 ms)
    - 5000 + 100 = 5,100 seeks (20,400 ms), total: 220,410 ms = 220.41 s
  - with *takes* as the outer relation
    - 10000 * 100 + 400 = 1,000,400 block transfers, (100,040 ms)
    - 10000 + 400 = 10,400 seeks (41,600 ms), total: 141,640 ms = 141.62 s
- If both relations (best case) or the smaller relation (*student*) fits entirely in memory, the cost estimate will be 500 (400+100) block transfers + 2 seeks.

Block Nested-Loop Join

- If the buffer is too small to hold either relation entirely in memory, we can still obtain a major saving in block accesses if we process the relations on a per-block basis, rather than on a per-tuple basis.
- Block nested-loop join is a variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

```
for each block B_r of r do begin
  for each block B_s of s do begin
      for each tuple t_r in B_r do begin
          for each tuple t_s in B_s do begin
              Check if (t_r,t_s) satisfy the join condition
              if they do, add t_r • t_s to the result.
          end
      end
  end
end
```

Block Nested-Loop Join (Cont.)

- Worst case estimate: $b_r * b_s + b_r$ block transfers + 2 * b_r seeks
  - Each block in the inner relation *s* is read once for each *block* in the outer relation
  - Each scan of the inner relation requires one seek, and the scan of the outer relation requires one seek per block
- Clearly, it is more efficient to use the smaller relation as the outer relation, in case neither of the relations fits in memory.
- In the best case, where the inner relation fits in memory cost will be: $b_r + b_s$ block transfers + 2 seeks.
- Worst case cost with *student* as outer relation:
  - 100 * 400 + 100 = 40,100 block transfers + 2 * 100 = 200 seeks
  - Significant improvement over 2,000,100 block transfers plus 5100 seeks the for worst case for the basic nested-loop join.
- Best case: cost remains the same
  - 100 + 400 = 500 block transfers and 2 seeks

Comparison of Joins Algorithms

Join Type	Case	Block Transfer	Seek
Nested-Loop	Worst	$n_r * b_s + b_r$	$n_r + b_r$
Nested-Loop	Best	$b_r + b_s$	2
Block Nested-Loop	Worst	$b_r * b_s + b_r$	$2 * b_r$
Block Nested-Loop	Best	$b_r + b_s$	2

Block Nested-Loop Join (Cont.)

- Improvements to nested loop and block nested loop algorithms:
  - In block nested-loop, use M - 2 disk blocks as blocking unit for outer relations, where M = memory size in blocks; use remaining two blocks to buffer inner relation and output
    - Cost = $\lceil b_r / (M-2) \rceil * b_s + b_r$ block transfers + $2 \lceil b_r / (M-2) \rceil$ seeks
  - If the join attributes in a natural join or an equi-join form a key on the inner relation, then for each outer relation tuple the inner loop can terminate as soon as the first match is found.
  - Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer and thus reducing the number of disk accesses needed.
  - If index is available on inner relation, file scans can be replaced with more efficient index lookups (next slide)

Indexed Nested-Loop Join

- Index lookups can replace file scans if
  - join is an equi-join or natural join and
  - an index is available on the inner relation's join attribute
    4. Can construct an index just to compute a join.
- For each tuple t_r in the outer relation r, use the index to look up tuples in s that satisfy the join condition with tuple t_r.
- Worst case: buffer has space for only one block of r and one block of the index, for each tuple in r, perform an index lookup on s.
- Cost of the join: $b_r (t_T + t_S) + n_r * c$
  - Where c is the cost of traversing index and fetching all matching s tuples for one tuple of r
- If indices are available on join attributes of both r and s, use the relation with fewer tuples as the outer relation.

Indexed 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
  - 100*400 + 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 disk accesses, each of which requires a block transfer and a seek.
  - No. of Seeks increased and overall cost increased, since seek time is costlier than block transfer time. However, if we had a selection on the *student* relation that reduces the number of rows significantly and thus would perform better than block-nested loop join.

Merge-Join

- also called the **sort-merge-join algorithm**
- can be used to compute natural joins and equi-joins

1. Sort both relations on their join attribute (if not already sorted on the join attributes).
2. Merge the sorted relations to join them
    1. Join step is similar to the merge stage of the sort-merge algorithm.
    2. Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched
    3. Detailed algorithm in book

Merge-Join (Cont.)

- Can be used only for equi-joins and natural joins
- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory)
- Thus the cost of merge join is:

  $b_r + b_s$ block transfers $+ \lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$ seeks
  - + the cost of sorting if relations are unsorted.
- **hybrid merge-join:** If one relation is sorted, and the other has a secondary B^+-tree index on the join attribute
  - Merge the sorted relation with the leaf entries of the B^+-tree .
  - Sort the result on the addresses of the unsorted relation's tuples
  - Scan the unsorted relation in physical address order and merge with previous result, to replace addresses by the actual tuples
    4. Sequential scan is more efficient than random lookup

Hash-Join

- Applicable for equi-joins and natural joins.
- A hash function h is used to partition tuples of both relations into sets that have the same hash value on the join attributes.
- h maps *JoinAttrs* values to $\{0, 1, ..., n\}$, where *JoinAttrs* denotes the common attributes of r and s used in the natural join.
  - $r_0, r_1, \ldots, r_n$ denote partitions of r tuples, each initially empty
    - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[JoinAttrs])$.
  - $s_0, s_1 \ldots, s_n$ denotes partitions of s tuples, each initially empty
    - Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s[JoinAttrs])$.
- *Note:* The hash function h should have the "goodness" properties of *randomness* and *uniformity*

Hash-Join (Cont.)

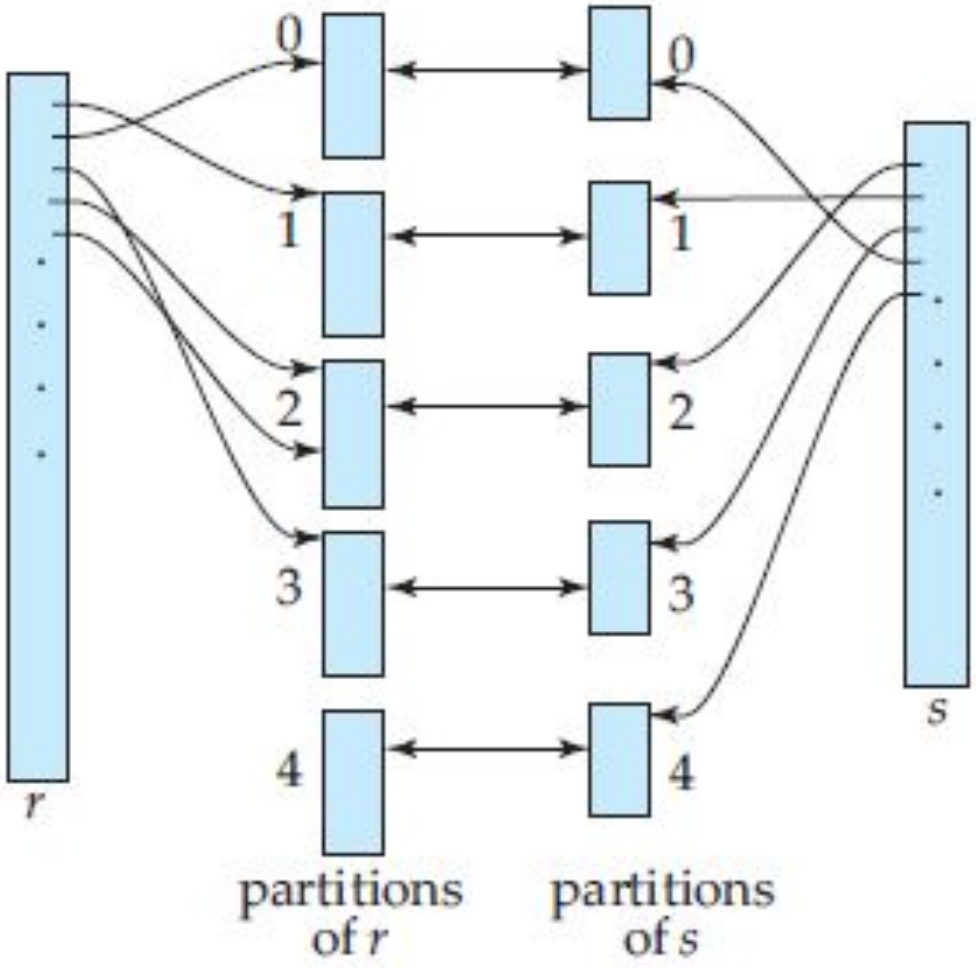


Figure 12.9 Hash partitioning of relations.

Hash-Join (Cont.)

- r tuples in r_i need only to be compared with s tuples in s_i. Need not be compared with s tuples in any other partition, since:
  - an r tuple and an s tuple that satisfy the join condition will have the same value for the join attributes.
  - If that value is hashed to some value i, the r tuple has to be in r_i and the s tuple in s_i.
- We must test corresponding partitions to see whether the values in their join attributes are the same, since it is possible that corresponding partitions have different values that have the same hash value.

Hash-Join Algorithm

The hash-join of r and s is computed as follows.

1. Partition the relation s using hashing function h. When partitioning a relation, one block of memory is reserved as the output buffer for each partition.
2. Partition r similarly.
3. For each i:
   (a) Load s_i into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one h.
   (b) Read the tuples in r_i from the disk one by one. For each tuple t_r locate each matching tuple t_s in s_i using the in-memory hash index. Output the concatenation of their attributes.

Relation s is called the **build input** and r is called the **probe input**.

Hash-Join algorithm (Cont.)

- The value n and the hash function h is chosen such that each s_i should fit in memory.
  - Typically n is chosen as $\lceil b_s/M \rceil * f$ where f is a "**fudge factor**", typically around 1.2
  - The probe relation partitions r_i need not fit in memory
- **Recursive partitioning** required if number of partitions n is greater than number of pages M of memory.
  - instead of partitioning n ways, use $M - 1$ partitions for s
  - Further partition the $M - 1$ partitions using a different hash function
  - A relation does not need recursive partitioning if $M > n_h + 1$, or equivalently $M > (b_s/M)+1$, which simplifies (approximately) to $M > \sqrt{b_s}$.
  - Rarely required: e.g., with block size of 4 KB, recursive partitioning not needed for relations of < 1GB with memory size of 2MB, or relations of < 36 GB with memory of 12 MB

Handling of Overflows

- Partitioning is said to be **skewed** if some partitions have significantly more tuples than some others
- **Hash-table overflow** occurs in partition s_i if s_i does not fit in memory. Reasons could be
  - Many tuples in s with same value for join attributes
  - Bad hash function
- **Overflow resolution** can be done in build phase
  - Partition s_i is further partitioned using different hash function.
  - Partition r_i must be similarly partitioned.
- **Overflow avoidance** performs partitioning carefully to avoid overflows during build phase
  - E.g. partition build relation into many partitions, then combine them
- Both approaches fail with large numbers of duplicates
  - Fallback option: use block nested loops join on overflowed partitions

Cost of Hash-Join

- If recursive partitioning is not required: cost of hash join is

  $3(b_r + b_s) + 4 * n_h$ block transfers +

  $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) + 4 * n_h$ seeks

- If recursive partitioning required:
  - number of passes required for partitioning build relation s to less than M blocks per partition is $\lceil log_{\lfloor M/bb \rfloor - 1}(b_s/M) \rceil$
  - best to choose the smaller relation as the build relation.
  - Total cost estimate is:

    $2(b_r + b_s) \lceil log_{\lfloor M/bb \rfloor - 1}(b_s/M) \rceil + b_r + b_s$ block transfers +

    $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) \lceil log_{\lfloor M/bb \rfloor - 1}(b_s/M) \rceil$ seeks
- If the entire build input can be kept in main memory no partitioning is required
  - Cost estimate goes down to $b_r + b_s$.

Example of Cost of Hash-Join

instructor ⋈ *teaches*

- Assume that memory size is 20 blocks
- $b_{instructor} = 100$ and $b_{teaches} = 400$.
- *instructor* is to be used as build input. Partition it into five partitions, each of size 20 blocks. This partitioning can be done in one pass.
- Similarly, partition *teaches* into five partitions,each of size 80. This is also done in one pass.
- Therefore total cost, ignoring cost of writing partially filled blocks:
  - $3(100 + 400) = 1500$ block transfers +
    $2(\lceil 100/3 \rceil + \lceil 400/3 \rceil) = 336$ seeks

Complex Joins

- Nested-loop and block nested-loop joins can be used regardless of the join conditions.
- The other join techniques are more efficient than the nested-loop join and its variants, but can handle only simple join conditions, such as natural joins or equi-joins.
- We can implement joins with complex join conditions, such as conjunctions and disjunctions, by using the efficient join techniques.

Complex Joins (Cont.)

- Join with a conjunctive condition:

$$r \bowtie_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n} s$$

  - Either use nested loops/block nested loops, or
  - Compute the result of one of the simpler joins $r \bowtie_{\theta i} s$
    4. final result comprises those tuples in the intermediate result that satisfy the remaining conditions

$$\theta_1 \wedge \ldots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \ldots \wedge \theta_n$$

- Join with a disjunctive condition

$$r \bowtie_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n} s$$

  - Either use nested loops/block nested loops, or
  - Compute as the union of the records in individual joins $r \bowtie_{\theta i} s$:

$$(r \bowtie_{\theta 1} s) \cup (r \bowtie_{\theta 2} s) \cup \ldots \cup (r \bowtie_{\theta n} s)$$

Other Operations: Duplicate Elimination

- Other relational operations and extended relational algebra operations include duplicate elimination, projection, set operations, outer join, and aggregation
- **Duplicate elimination** can be implemented via sorting or hashing.
  - Using **sorting** duplicates will come adjacent to each other, and all but one set of duplicates can be deleted.
  - With external sort–merge, duplicates found while a run is being created can be removed before the run is written to disk, thereby reducing the number of block transfers. The remaining duplicates can be eliminated during merging, and the final sorted run has no duplicates.
  - The worst-case cost estimate for duplicate elimination is the same as the worst-case cost estimate for sorting of the relation.
  - Hashing is similar – duplicates will come into the same bucket.

Other Operations: Duplicate Elimination

- For **hashing**, use hash-join algorithm.
- First, the relation is partitioned on the basis of a hash function on the whole tuple.
- Then, each partition is read in, and an in-memory hash index is constructed. While constructing the hash index, a tuple is inserted only if it is not already present. Otherwise, the tuple is discarded.
- After all tuples in the partition have been processed, the tuples in the hash index are written to the result.
- The cost estimate is the same as that for the cost of processing (partitioning and reading each partition) of the build relation in a hash join.

- Because of the relatively high cost of duplicate elimination, SQL requires an explicit request by the user to remove duplicates; otherwise, the duplicates are retained.

Other Operations: Projection

- **Projection**:
  - Projection can be implemented easily by performing projection on each tuple, which gives a relation that could have duplicate records, and then removing duplicate records.
  - Duplicates can be eliminated by sorting or hashing.
  - If the attributes in the projection list include a key of the relation, no duplicates will exist; hence, duplicate elimination is not required.
  - Generalized projection can be implemented in the same way as projection.

Other Operations : Aggregation

- **Aggregation** can be implemented in a manner similar to duplicate elimination. We use either sorting or hashing, just as we did for duplicate elimination, but based on the grouping attributes.
- However, instead of eliminating tuples with the same value for the grouping attribute, we gather them into groups, and apply the aggregation operations on each group to get the result.
- The cost estimate for implementing the aggregation operation is the same as the cost of duplicate elimination.
- Instead of gathering all the tuples in a group and then applying the aggregation operations, we can implement the aggregation operations **sum, min, max, count,** and **avg** on the fly as the groups are being constructed.
  - For the case of **sum, min** and **max**, when two tuples in the same group are found, the system replaces them by a single tuple containing the **sum, min,** or **max**, respectively, of the columns being aggregated.

Other Operations : Aggregation

- For the **count** operation, it maintains a running count for each group for which a tuple has been found.
- Finally, we implement the **avg** operation by computing the sum and the count values on the fly, and finally dividing the sum by the count to get the average.

- If all tuples of the result fit in memory, both the sort-based and the hash-based implementations do not need to write any tuples to disk.
- As the tuples are read in, they can be inserted in a sorted tree structure or in a hash index.
- When we use on-the-fly aggregation techniques, only one tuple needs to be stored for each of the groups.
- Hence, the sorted tree structure or hash index fits in memory, and the aggregation can be processed with just b_r block transfers (and 1 seek)

Other Operations : Set Operations using Hashing

- **Set operations** ($\cup$, $\cap$ and $-$): can either use variant of merge-join after sorting, or variant of hash-join.
- E.g., Set operations using hashing:
  1. Partition both relations using the same hash function. Create the partitions $r_0, r_1, \ldots, r_{nh}$ and $s_0, s_1, \ldots, s_{nh}$ .Depending on the operation, the system then takes these steps on each partition $i = 0, 1, \ldots, n_h$
  2. Process each partition i as follows.
     1. Using a different hashing function, build an in-memory hash index on r_i.
     2. Process s_i as follows
        - $r \cup s$:
          1. Add tuples in s_i to the hash index if they are not already in it.
          2. At end of s_i add the tuples in the hash index to the result.

Other Operations : Set Operations using Hashing

- $r \cap s$:
  1. for each tuple in s_i , probe the hash index and output the tuple to the result only if it is already present in the hash index.
- $r - s$:
  1. for each tuple in s_i, if it is there in the hash index, delete it from the index.
  2. At end of s_i add remaining tuples in the hash index to the result.

Other Operations : Set Operations using sorting

- We can implement the *union*, *intersection*, and *set-difference* operations by first sorting both relations, and then scanning once through each of the sorted relations to produce the result.
- In $r \cup s$, when a concurrent scan of both relations reveals the same tuple in both files, only one of the tuples is retained.
- The result of $r \cap s$ will contain only those tuples that appear in both relations.
- We implement *set difference*, $r - s$, similarly, by retaining tuples in r only if they are absent in s.
- For all these operations, only one scan of the two sorted input relations is required
- Cost is $b_r + b_s$ block transfers + $b_r + b_s$ disk seeks. The number of seeks can be reduced by allocating extra buffer blocks.
- If the relations are not sorted initially, the cost of sorting has to be included.

Other Operations : Outer Join

- **Outer join** can be computed either as
  - 1. A join followed by addition of null-padded non-participating tuples.
  - 2. by modifying the join algorithms.
- 2.1 It is easy to extend the nested-loop join algorithms to compute the left outer join:
  - Tuples in the outer relation that do not match any tuple in the inner relation are written to the output after being padded with null values.
  - However, it is hard to extend the nested-loop join to compute the full outer join.
- 2.2 Natural outer joins and outer joins with an equi-join condition can be computed by extensions of the merge-join and hash-join algorithms.

Other Operations : Outer Join

- Modifying merge join to compute r s
  - In r s, non participating tuples are those in $r - \Pi_R(r \quad s)$
  - Modify merge-join to compute r s
    4. During merging, for every tuple t_r from r that do not match any tuple in s, output t_r padded with nulls.
  - Right outer-join and full outer-join can be computed similarly.

Other Operations : Outer Join

- Modifying hash join to compute r $\overrightarrow{\bowtie}$ s
  - If r is probe relation, output non-matching r tuples padded with nulls
  - If r is build relation, when probing keep track of which r tuples matched s tuples. At end of s_i output non-matched r tuples padded with nulls

Evaluation of Expressions

- So far: we have seen algorithms for individual operations
- Now consider how to evaluate an expression containing multiple operations.
- Alternatives for evaluating an entire expression tree
  - **Materialization**: evaluate an expression by evaluating one operation at a time, in an appropriate order. The result of each evaluation is **materialized** in a temporary relation for subsequent use.
  - A disadvantage to this approach is the need to construct the temporary relations, which (unless they are small) must be written to disk
  - **Pipelining**: An alternative approach is to evaluate several operations simultaneously in a **pipeline**, with the results of one operation passed on to parent operations even as an operation is being executed, without the need to store a temporary relation.

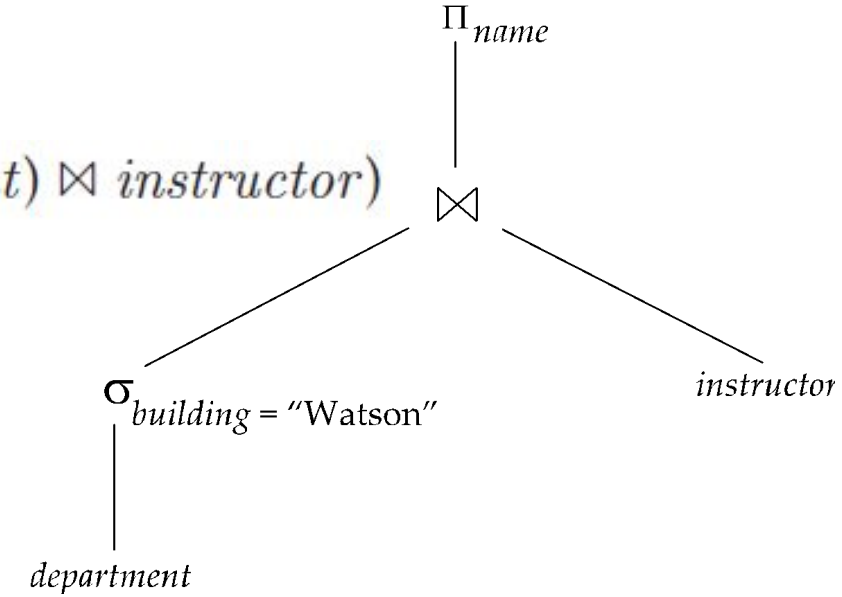
Materialization

- **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

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

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

$$\Pi_{name}(\sigma_{building=\text{“Watson”}}(department) \bowtie instructor)$$



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. Execute more quickly by performing CPU activity in parallel with I/O activity.
  - Allows overlap of disk writes with computation and reduces execution time

Pipelining

- **Pipelined evaluation** : query-evaluation efficiency can be improved by reducing the number of temporary files by combining several relational operations into a *pipeline* of operations, in which the results of one operation are passed along to the next operation in the pipeline. Evaluation as just described is called **pipelined evaluation.**
- Creating a pipeline of operations can provide two benefits:
  - 1. It eliminates the cost of reading and writing temporary relations, reducing the cost of query evaluation.
  - 2. It can start generating query results quickly, if the root operator of a query evaluation plan is combined in a pipeline with its inputs. This can be quite useful if the results are displayed to a user as they are generated, since otherwise there may be a long delay before the user sees any query results.

Pipelining (Cont.)

- A pipeline can be implemented by constructing a single, complex operation that combines the operations that constitute the pipeline.
- 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.)

- **Demand driven** or **lazy** evaluation
  - The system makes repeated requests for tuples from the operation at the top of the pipeline.
  - Each time that an operation receives a request for tuples, it computes the next tuple (or tuples) from children operations as required to be returned, and then returns that tuple.
  - If it has some pipelined inputs, the operation also makes requests for tuples from its pipelined inputs.
  - Using the tuples received from its pipelined inputs, the operation computes tuples for its output, and passes them up to its parent.
  - In between calls, operation has to maintain "**state**" so it knows what to return next

Pipelining (Cont.)

- **Producer-driven** or **eager** pipelining
  - Operations do not wait for requests to produce tuples, but instead generate the tuples **eagerly**.
  - Each operation in a producer-driven pipeline is modeled as a separate process or thread within the system that takes a stream of tuples from its pipelined inputs and generates a stream of tuples for its output.
  - The processes or threads corresponding to different operations execute concurrently. For each pair of adjacent operations in a producer-driven pipeline, the system creates a buffer to hold tuples being passed from one operation to the next.
    - 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

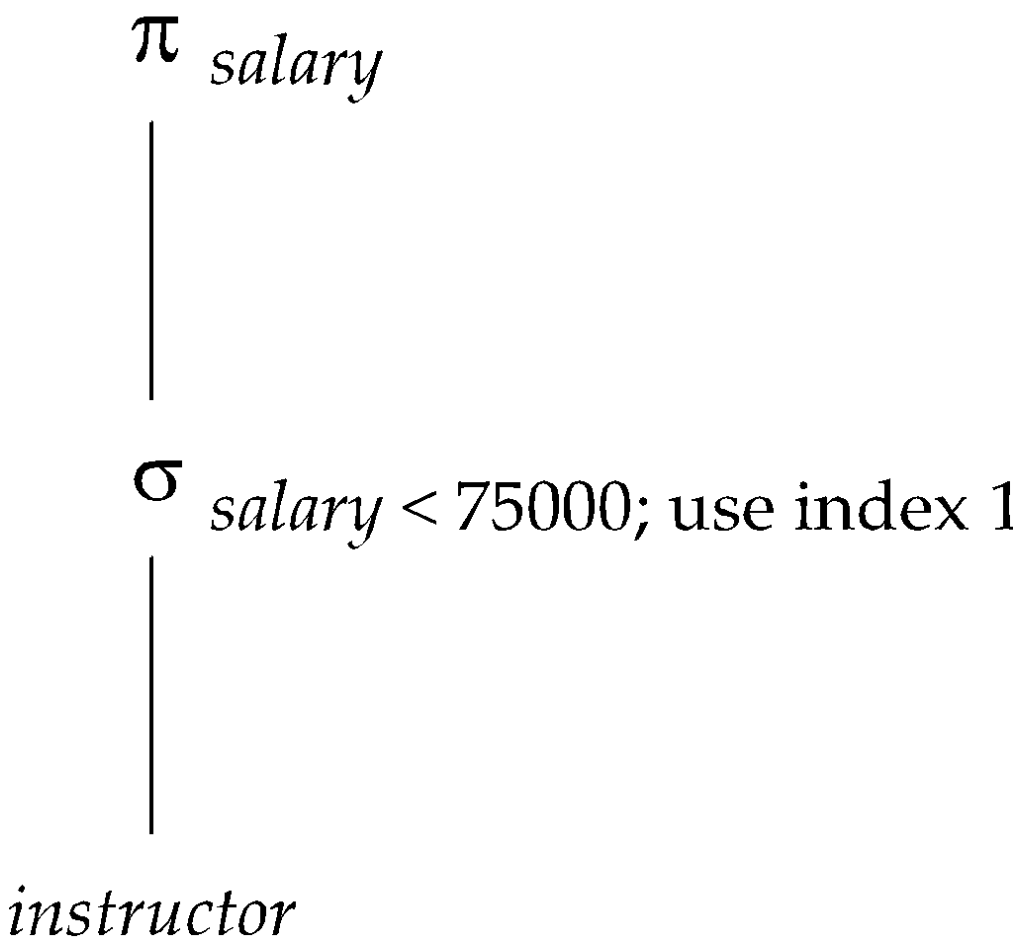
Evaluation Algorithms for Pipelining

- 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

- Summary of the Chapter Query Processing

End of Chapter

Figure 12.02



Selection Operation (Cont.)

- **Old-A2** *(binary search)*. Applicable if selection is an equality comparison on the attribute on which file is ordered.
  - Assume that the blocks of a relation are stored contiguously
  - Cost estimate (number of disk blocks to be scanned):
    - cost of locating the first tuple by a binary search on the blocks
      - $\lceil \log_2(b_r) \rceil * (t_T + t_S)$
    - If there are multiple records satisfying selection
      - *Add transfer cost of the* number of blocks containing records that satisfy selection condition
      - Will see how to estimate this cost in Chapter 13