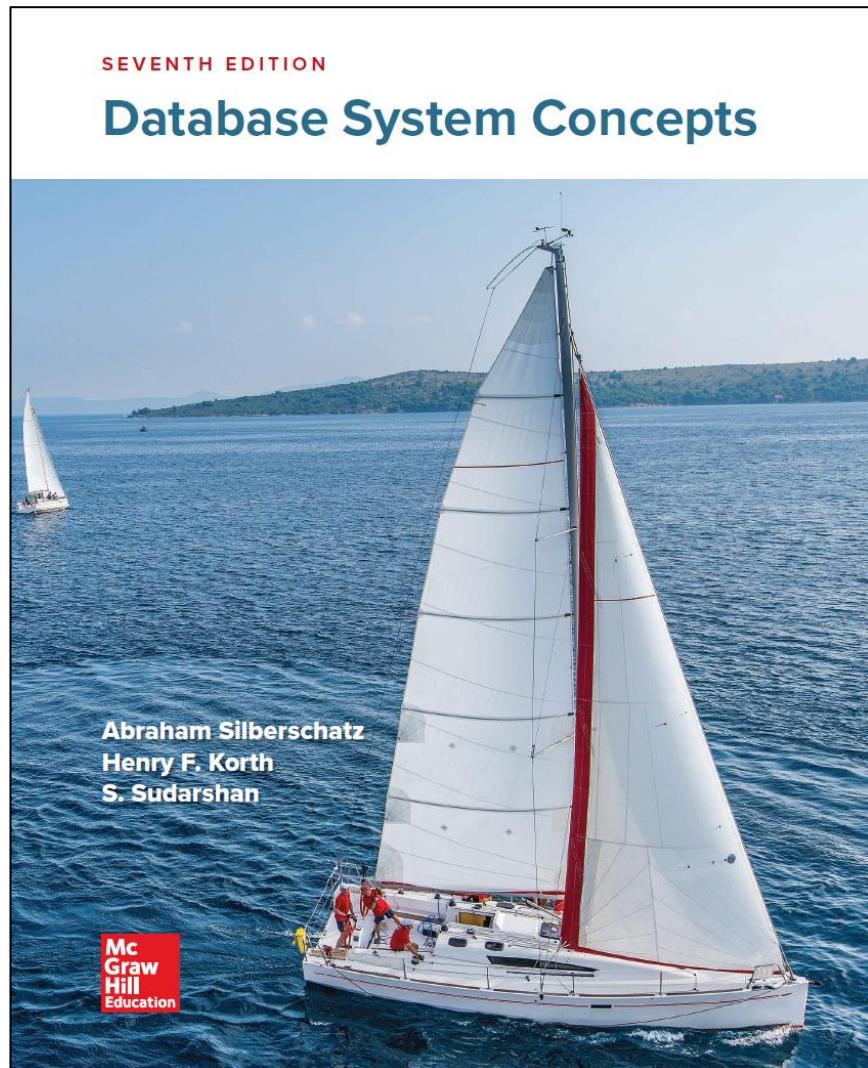


# Data Administration in Information Systems

---

Query processing

# Query processing



### x Contents

#### PART FIVE ■ STORAGE MANAGEMENT AND INDEXING

##### Chapter 12 Physical Storage Systems

12.1 Overview of Physical Storage Media	559	12.6 Disk-Block Access	577
12.2 Storage Interfaces	562	12.7 Summary	580
12.3 Magnetic Disks	563	Exercises	582
12.4 Flash Memory	567	Further Reading	584
12.5 RAID	570		

##### Chapter 13 Data Storage Structures

13.1 Database Storage Architecture	587	13.7 Storage Organization in Main-Memory Databases	615
13.2 File Organization	588	13.8 Summary	617
13.3 Organization of Records in Files	595	Exercises	619
13.4 Data-Dictionary Storage	602	Further Reading	621
13.5 Database Buffer	604		
13.6 Column-Oriented Storage	611		

##### Chapter 14 Indexing

14.1 Basic Concepts	623	14.8 Write-Optimized Index Structures	665
14.2 Ordered Indices	625	14.9 Bitmap Indices	670
14.3 B <sup>+</sup> -Tree Index Files	634	14.10 Indexing of Spatial and Temporal Data	672
14.4 B <sup>+</sup> -Tree Extensions	650	14.11 Summary	677
14.5 Hash Indices	658	Exercises	679
14.6 Multiple-Key Access	661	Further Reading	683
14.7 Creation of Indices	664		

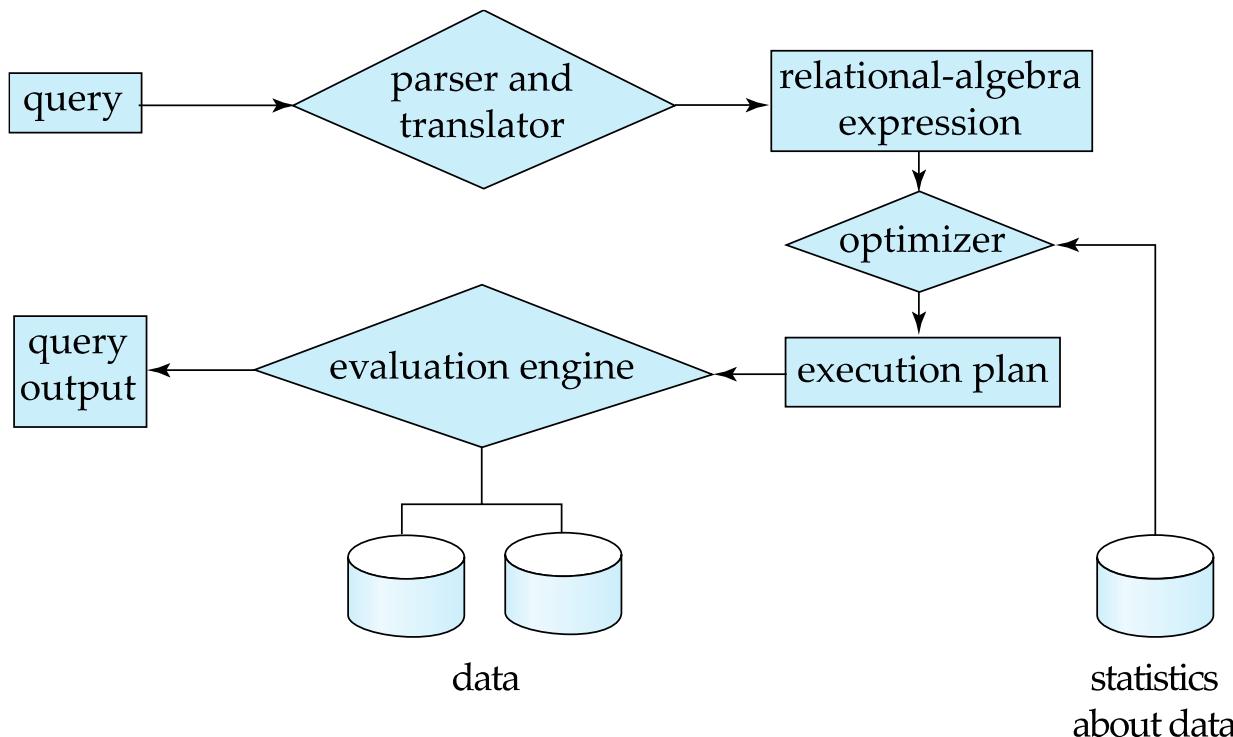
#### PART SIX ■ QUERY PROCESSING AND OPTIMIZATION

##### Chapter 15 Query Processing

15.1 Overview	689	15.7 Evaluation of Expressions	724
15.2 Measures of Query Cost	692	15.8 Query Processing in Memory	731
15.3 Selection Operation	695	15.9 Summary	734
15.4 Sorting	701	Exercises	736
15.5 Join Operation	704	Further Reading	740
15.6 Other Operations	719		

# Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



# Basic Steps in Query Processing (Cont.)

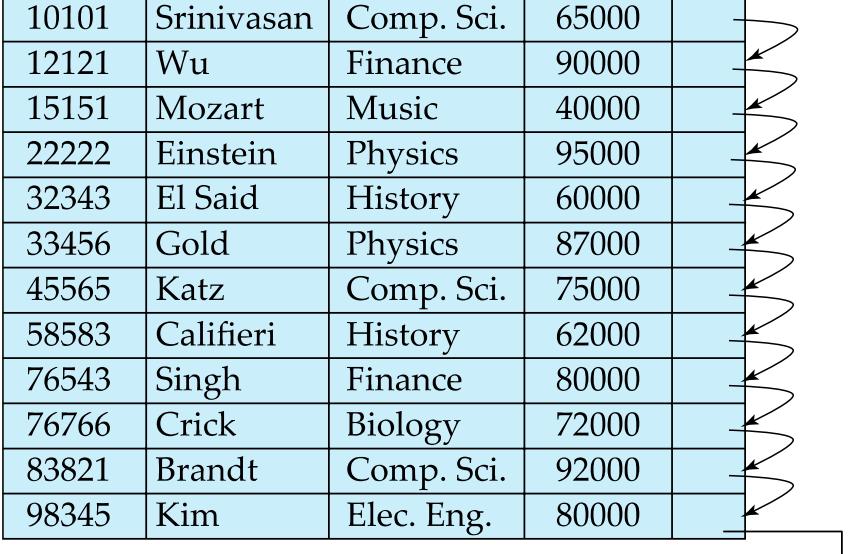
- Parsing and translation
  - Translate the query into its internal form. This is then translated into relational algebra.
  - Parser checks syntax, verifies relations.
- Optimization
  - Construct an execution plan that minimizes the cost of query evaluation.
- Evaluation
  - The evaluation engine takes an execution plan, executes that plan, and returns the answers to the query.

# Basic Steps in Query Processing (Cont.)

- Parsing and translation
  - Translate the query into its internal form. This is then translated into relational algebra.

```
select salary  
from instructor  
where salary < 75000
```

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	



The diagram illustrates the filtering process. It shows a query: **select salary from instructor where salary < 75000**. Arrows point from the **salary** column in the table to the **salary** column in the query, specifically targeting the condition **salary < 75000**. The table contains 12 rows of data, but only the last 11 rows (from 32343 to 98345) have salaries less than 75000, which are highlighted in blue. The first row (10101) is highlighted in light blue and has an arrow pointing to it from the query's **salary** column, indicating it is the result of the query.

$$\prod_{\text{salary}} (\sigma_{\text{salary} < 75000}(\text{instructor}))$$

# Basic Steps in Query Processing (Cont.)

- A relational algebra expression may have many equivalent expressions
  - e.g.,  $\sigma_{\text{salary} < 75000}(\Pi_{\text{salary}}(\text{instructor}))$  is equivalent to  $\Pi_{\text{salary}}(\sigma_{\text{salary} < 75000}(\text{instructor}))$
- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- The expression specifying a detailed evaluation strategy is called an **execution plan**, e.g.:
  - Use an index on *salary* to find instructors with  $\text{salary} \geq 75000$ ,
  - Or perform complete relation scan and discard instructors with  $\text{salary} < 75000$

# Basic Steps in Query Processing (Cont.)

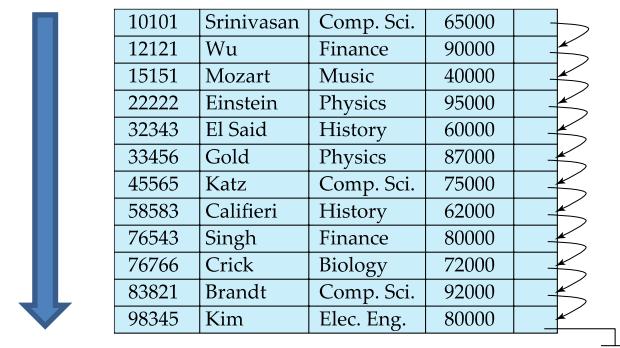
- **Query Optimization:** when multiple possible execution plans are available, choose the one with lowest cost.
  - Cost is estimated using statistical information from the database catalog
    - e.g.. number of tuples in each relation, size of tuples, etc.
- Today we study
  - The cost of individual operations/algorithms
  - How to combine individual operations to evaluate more complex expressions
- Next lecture
  - How to optimize the entire execution plan, i.e. how to find an evaluation plan with lowest estimated cost

# Measures of Query Cost

- Disk cost can be estimated as:
  - Number of seeks \* average-seek-cost
  - Number of blocks read \* average-block-read-cost
  - Number of blocks written \* average-block-write-cost
- For simplicity we just use the **number of block transfers** from disk and the **number of seeks** as the cost measures
  - $t_T$  : time to transfer one block
    - Assuming for simplicity that write cost is same as read cost
  - $t_S$  : time for one seek
  - Cost for  $b$  block transfers plus  $S$  seeks
$$b * t_T + S * t_S$$
- $t_S$  and  $t_T$  depend on where data is stored; with 4 KB blocks:
  - High end magnetic disk:  $t_S = 4$  ms and  $t_T = 0.1$  ms
  - SSD:  $t_S = 20\text{-}90$   $\mu$ s and  $t_T = 2\text{-}10$   $\mu$ s for 4KB

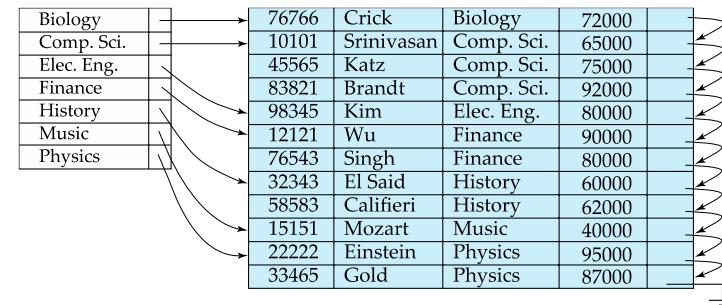
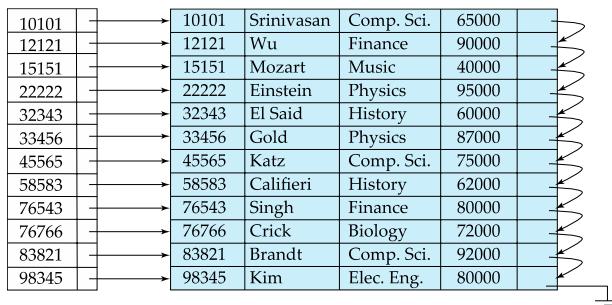
# Selection Operation

- **File scan**
- Algorithm **A1 (linear search)**. Scan each file block and test all records to see whether they satisfy the selection condition.
  - Cost estimate =  $b_r$  block transfers + 1 seek
    - $b_r$  denotes number of blocks containing records from relation  $r$
  - If selection is on a key attribute, can stop on finding record
    - cost =  $(b_r/2)$  block transfers + 1 seek
  - Linear search can be applied regardless of
    - selection condition or
    - ordering of records in the file, or
    - availability of indices



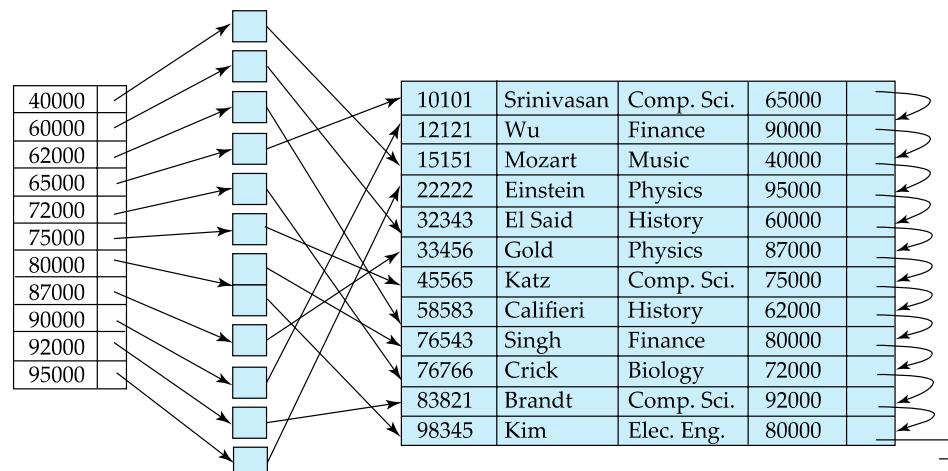
# Selections Using Indices

- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index.
- **A2 (clustered index, equality on key)**. Retrieve a single record that satisfies the corresponding equality condition
  - $Cost = (h_i + 1) * (t_T + t_S)$
- **A3 (clustered index, equality on non-key)** Retrieve multiple records.
  - Records will be on consecutive blocks
    - Let  $b$  = number of blocks containing matching records
  - $Cost = h_i * (t_T + t_S) + t_S + t_T * b$



# Selections Using Indices

- **A4 (non-clustered index, equality on key/non-key).**
  - Retrieve a single record if the search-key is a candidate key
    - $Cost = (h_i + 1) * (t_T + t_S)$
  - Retrieve multiple records if search-key is not a candidate key
    - each of  $n$  matching records may be on a different block
    - $Cost = (h_i + n) * (t_T + t_S)$ 
      - Can be very expensive!



# Selections Involving Comparisons

- Can implement selections of the form  $\sigma_{A \leq V}(r)$  or  $\sigma_{A \geq V}(r)$  by using
  - a linear file scan,
  - or by using indices, in the following ways:
- **A5 (clustered index, comparison).** (Relation is sorted on A)
  - For  $\sigma_{A \geq V}(r)$  use index to find first tuple  $\geq V$  and then scan sequentially
  - For  $\sigma_{A \leq V}(r)$  just scan sequentially till first tuple  $> V$ ; do not use index
- **A6 (non-clustered index, comparison).**
  - 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
    - requires an I/O per record; linear file scan may be cheaper!

# Implementation of Complex Selections

- **Conjunction:**  $\sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(r)$
- **A7 (conjunctive selection using one index).**
  - Select a combination of  $\theta_i$  and algorithms A1 through A7 that results in the least cost for  $\sigma_{\theta_i}(r)$
  - Test other conditions on tuple after fetching it into memory
- **A8 (conjunctive selection using composite index).**
  - Use appropriate composite (multiple-key) index if available.
- **A9 (conjunctive selection by intersection of identifiers).**
  - Requires indices with record pointers.
  - Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers.
  - Then fetch records from file.
  - If some conditions do not have appropriate indices, apply test in memory.

# Algorithms for Complex Selections

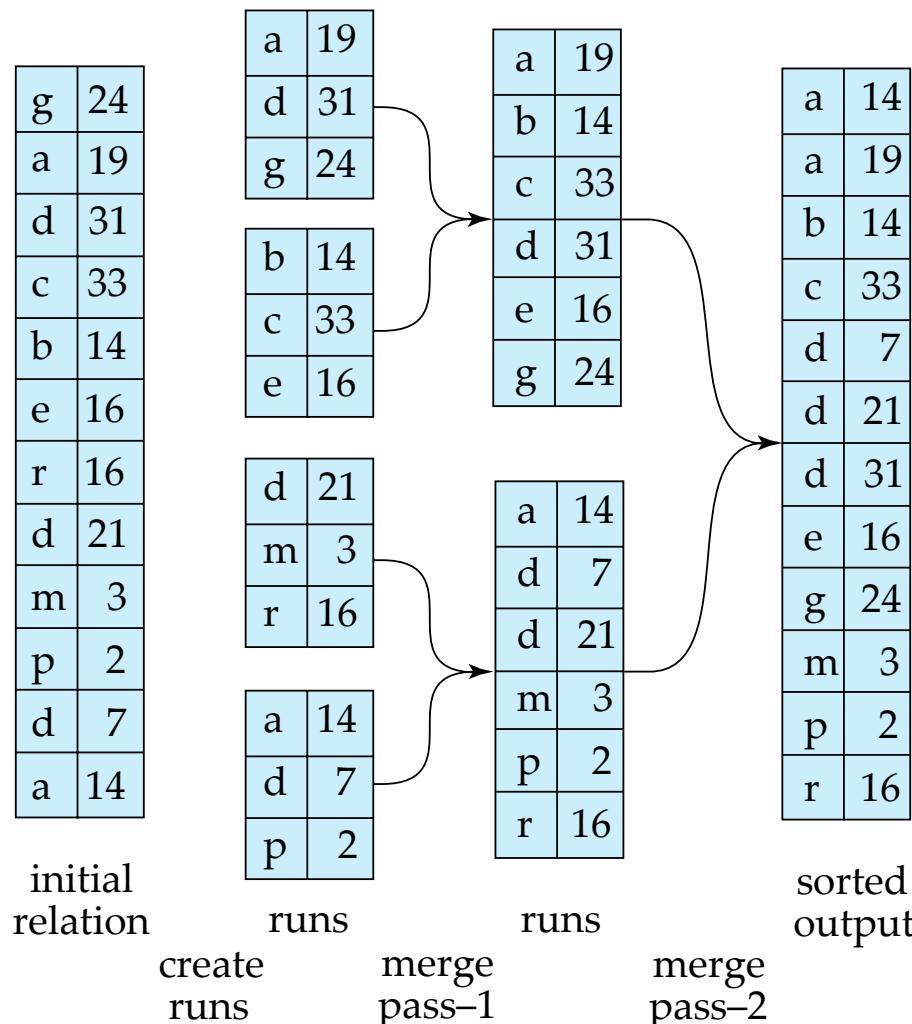
- **Disjunction:**  $\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r)$
- **A10 (disjunctive selection by union of identifiers).**
  - Applicable if *all* conditions have available indices.
    - Otherwise use linear scan.
  - Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
  - Then fetch records from file.
- **Negation:**  $\sigma_{\neg\theta}(r)$ 
  - Use linear scan on file
  - Or transform  $\neg\theta$  into expression without negation  $\theta'$ , and check if an index is applicable to  $\theta'$ 
    - Find satisfying records using index and fetch from file

# Sorting

---

- We may build an index on the relation, and then use the index to read the relation in sorted order.
  - May lead to one disk block access for each tuple.
- 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.

# Example: External Sorting Using Sort-Merge



# External Sort-Merge

Let  $M$  denote memory size (in pages).

1. **Create sorted runs.** Let  $i$  be 0 initially.

Repeatedly do the following till the end of the relation:

- (a) Read  $M$  blocks of relation into memory
- (b) Sort the in-memory blocks
- (c) Write sorted data to run  $R_i$ ; increment  $i$ .

Let the final value of  $i$  be  $N$

2. *Merge the runs (next slide).....*

# External Sort-Merge (Cont.)

## 2. Merge the runs (N-way merge).

We assume (for now) that  $N < M$ .

1. Use  $N$  blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page
2. **repeat**
  1. Select the first record (in sort order) among all buffer pages
  2. Write the record to the output buffer. If the output buffer is full write it to disk.
  3. Delete the record from its input buffer page.  
**If** the buffer page becomes empty **then**  
read the next block (if any) of the run into the buffer.
3. **until** all input buffer pages are empty.

# 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
  - Repeated passes are performed till all runs have been merged into one.

# External Sort-Merge (Cont.)

- Cost analysis:
  - The number of blocks in relation  $r$  is:  $b_r$
  - To create the initial runs, read and write every block:  $2b_r$  block transfers
  - The number of initial runs is:  $\lceil b_r/M \rceil$
  - Each merge pass decreases the number of runs by a factor of  $M-1$
  - The total number of merge passes is:  $\lceil \log_{M-1}(b_r/M) \rceil$
  - Each merge pass reads and writes every block:  $2b_r$  block transfers
  - For the final pass we discount the write cost:  $-b_r$ 
    - we ignore the final write cost since the output may be sent to the parent operation without being written to disk
  - The total number of block transfers is:
$$2b_r + 2b_r \lceil \log_{M-1}(b_r/M) \rceil - b_r = \\ b_r(2\lceil \log_{M-1}(b_r/M) \rceil + 1)$$
  - Seek: next slide

# External Sort-Merge (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  $2b_r$  seeks for each merge pass
    - Except the final one which does not require a write
  - The total number of seeks is:

$$\begin{aligned} 2\lceil b_r/M \rceil + 2b_r\lceil \log_{M-1}(b_r/M) \rceil - b_r &= \\ 2\lceil b_r/M \rceil + b_r(2\lceil \log_{M-1}(b_r/M) \rceil - 1) & \end{aligned}$$

# Join Operation

- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Choice based on cost estimate

# Nested-Loop Join

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

```
for each tuple  $t_r$  in  $r$ 
  for each tuple  $t_s$  in  $s$ 
    test pair  $(t_r, t_s)$  to see if they satisfy the join condition  $\theta$ 
    if they do, add  $t_r \bullet t_s$  to the result
  end
end
```

- $r$  is called the **outer relation** and  $s$  the **inner relation** of the join.
- Requires no indices and can be used with any kind of join condition.
- Expensive since it examines every pair of tuples in the two relations.

# Nested-Loop Join (Cont.)

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is:
  - Block transfers:  $b_r + n_r * b_s$
  - Seek:  $b_r + n_r$
- If the smaller relation fits entirely in memory, use that as the inner relation. Reduces cost to:
  - Block transfers:  $b_r + b_s$
  - Seek: 2
- Block nested-loops algorithm (next slide) is preferable.

# Block Nested-Loop Join

- 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  $r$ 
  for each block  $B_s$  of  $s$ 
    for each tuple  $t_r$  in  $B_r$ 
      for each tuple  $t_s$  in  $B_s$ 
        check if  $(t_r, t_s)$  satisfy the join condition
        if they do, add  $t_r \bullet t_s$  to the result
      end
    end
  end
end
```

# Block Nested-Loop Join (Cont.)

- Worst case estimate:
  - Block transfers:  $b_r + b_r * b_s$
  - Seek:  $b_r + b_r = 2 * b_r$
- Each block in the inner relation  $s$  is read once for each block in the outer relation
- Best case, if the inner relation fits in memory:
  - Block transfers:  $b_r + b_s$
  - Seek: 2

# Block Nested-Loop Join (Cont.)

- Improvements to nested loop and block nested loop algorithms:
  - In block nested-loop, use  $M-2$  disk blocks for outer relation, and use remaining two blocks to buffer inner relation and output:
    - Block transfers:  $b_r + \lceil b_r/(M-2) \rceil * b_s$
    - Seek:  $2 * \lceil b_r/(M-2) \rceil$
  - If equi-join attribute forms a key or inner relation, stop inner loop on first match
  - Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer (with LRU replacement)
  - Use index on inner relation if available (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
    - might also construct an index just to compute the join
- E.g. to compute the natural join:  $r \bowtie s$

```
for each tuple  $t_r$  in  $r$   
  use index on  $s$  to find matching tuple  $t_s$   
  add  $t_r \bullet t_s$  to the result  
end
```

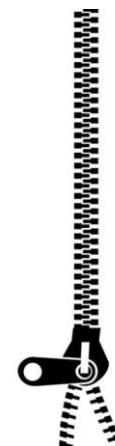
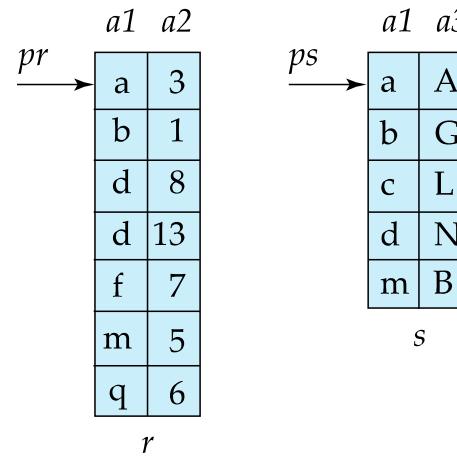
```
for each block  $B_r$  of  $r$   
  for each tuple  $t_r$  in  $B_r$   
    use index on  $s$  to find matching tuple  $t_s$   
    add  $t_r \bullet t_s$  to the result  
  end  
end
```

# Indexed Nested-Loop Join (Cont.)

- 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 page of  $r$ , and, for each tuple in  $r$ , we perform an index lookup on  $s$ .
- Cost of the join:  $b_r (t_r + t_s) + n_r * c$ 
  - where  $c$  is the cost of traversing index and fetching all matching  $s$  tuples for one tuple of  $r$
  - $c$  can be estimated as cost of a single selection on  $s$  using the join condition.
- If indices are available on join attributes of both  $r$  and  $s$ , use the relation with fewer tuples as the outer relation.

# Merge-Join

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



# 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:
  - Block transfers:  $b_r + b_s$
  - Seek:  $\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil$ , if we can read  $b_b$  blocks at once into memory
  - Plus the cost of sorting if relations are unsorted!

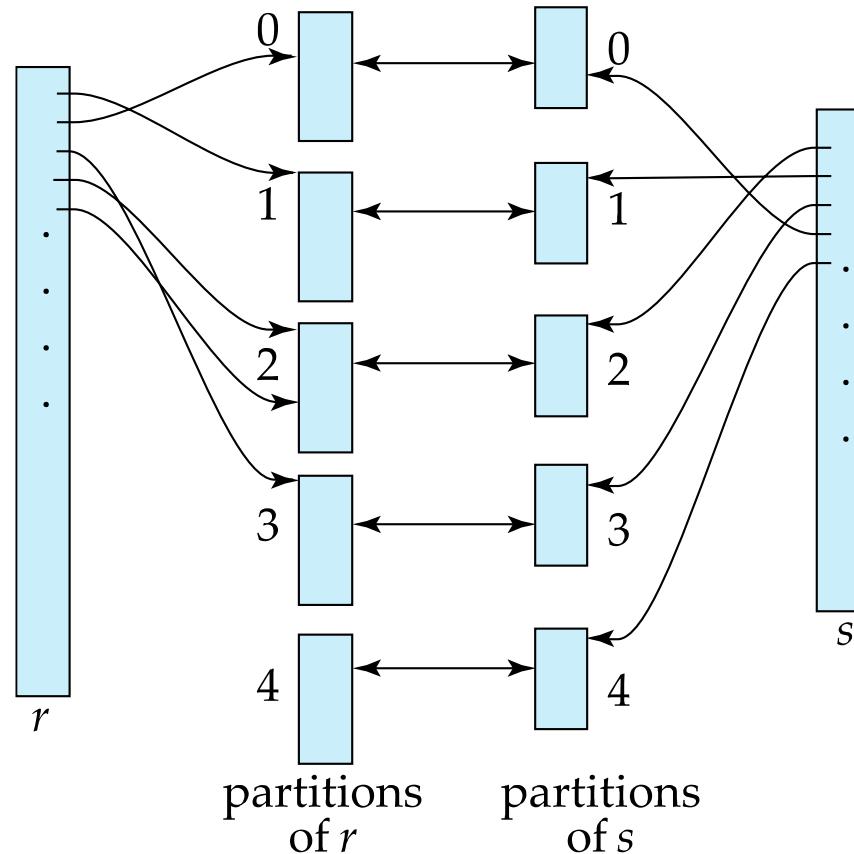
# Merge-Join (Cont.)

- **Hybrid merge-join:** If one relation is sorted, and the other has a secondary B<sup>+</sup>-tree index on the join attribute
  - Merge the sorted relation with the leaf entries of the B<sup>+</sup>-tree
  - Result contains tuples from the sorted relation and addresses for tuples of the unsorted relation
  - 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, replacing addresses by the actual tuples

# Hash-Join

- Applicable for equi-joins and natural joins
- $r$  and  $s$  have common attributes to be used in the natural join
- A hash function  $h$  is used to partition tuples of both relations
- $h$  maps attribute values to buckets or partitions  $\{0, 1, \dots, n\}$ 
  - $r_0, r_1, \dots, r_n$  denote partitions of relation  $r$ 
    - each tuple  $t_r \in r$  is put in partition  $r_i$  where  $i = h(t_r)$
  - $s_0, s_1, \dots, s_n$  denote partitions of relation  $s$ 
    - each tuple  $t_s \in s$  is put in partition  $s_i$  where  $i = h(t_s)$

# Hash-Join (Cont.)



## Hash-Join (Cont.)

- Tuples in  $r_i$  need only to be compared with tuples in  $s_i$
- No need to compare tuples in  $r_i$  with tuples in  $s_j$  ( $i \neq j$ ) 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  $s_i$

## Hash-Join (Cont.)

- Partitioning the two relations  $r$  and  $s$  requires reading and writing every block:  $2*(b_r + b_s)$
- Comparing the tuples in the partitions requires reading them once more:  $b_r + b_s$
- As a result of the partitioning, there can be some partially filled blocks
  - Each partition could have an extra block, and there  $n_h$  partitions
  - These extra blocks must be written (when partitioning) and read (when comparing)
  - There are two relations being partitioned
- Therefore, the cost of the hash-join is:
  - Block transfers:  $3*(b_r + b_s) + 4*n_h$
  - Seek:  $2*(b_r + b_s) + 2*n_h$

# Hash-Join (Cont.)

- If the number of partitions  $n_h$  is larger than memory  $M$  then we need to use **recursive partitioning**
  - Instead of partitioning  $n_h$  ways, use  $M-1$  partitions
  - Further partition the  $M-1$  partitions using a different hash function
  - The number of passes is  $\lceil \log_{M-1}(b_r/M) \rceil$
- The cost with recursive partitioning would be:
  - Block transfers:  $2(b_r + b_s) \lceil \log_{M-1}(b_r/M) \rceil + (b_r + b_s) + \dots$
  - Seek:  $2(b_r + b_s) \lceil \log_{M-1}(b_r/M) \rceil + \dots$

# Complex Joins

- Join with a conjunctive condition:

$$r \bowtie_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n} s$$

- Either use nested loops/block nested loops, or
- Compute one of the simpler joins  $r \bowtie_{\theta_i} s$ 
  - then check which tuples satisfy the remaining conditions  
 $\theta_1 \wedge \dots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \dots \wedge \theta_n$

- Join with a disjunctive condition

$$r \bowtie_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n} s$$

- Either use nested loops/block nested loops, or
- Compute each join separately
  - then union of records in individual joins  $r \bowtie_{\theta_i} s$ :  
 $(r \bowtie_{\theta_1} s) \cup (r \bowtie_{\theta_2} s) \cup \dots \cup (r \bowtie_{\theta_n} s)$

# Other Operations

- **Duplicate elimination (DISTINCT)** can be implemented via hashing or sorting.
  - On sorting duplicates will come adjacent to each other, and all but one set of duplicates can be deleted.
  - *Optimization*: duplicates can be deleted during run generation as well as at intermediate merge steps in external sort-merge.
  - Hashing is similar – duplicates will come into the same bucket.

# Other Operations (Cont.)

- **Aggregation (GROUP BY)** can be implemented in a manner similar to duplicate elimination.
  - **Sorting** or **hashing** can be used to bring tuples in the same group together, and then the aggregate functions can be applied on each group.
  - Optimization: **partial aggregation**
    - combine tuples in the same group during run generation and intermediate merges, by computing partial aggregate values
    - For **count**, **min**, **max**, **sum**: keep aggregate values on tuples found so far in the group.
      - When combining partial aggregate for count, add up the partial aggregates
    - For **avg**, keep sum and count, and divide sum by count at the end

# Evaluation of Expressions

---

- So far: we have seen algorithms for individual operations
- Alternatives for evaluating an entire expression tree
  - **Materialization**: generate results of an expression whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat.
  - **Pipelining**: pass on tuples to parent operations even as an operation is being executed
- We study above alternatives in more detail

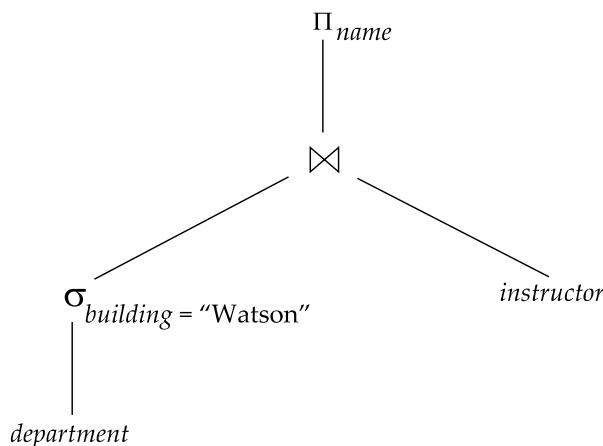
# Materialization

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

- e.g., in figure below, compute and store

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

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



# Materialization (Cont.)

- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be 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

# Pipelining

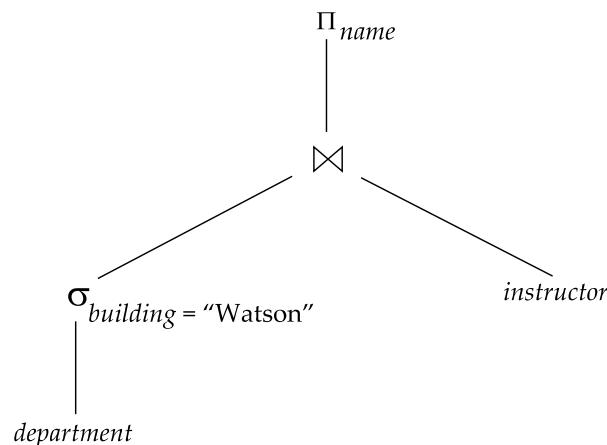
- **Pipelined evaluation:** evaluate several operations simultaneously, passing the results of one operation to the next.
  - e.g., in previous expression tree, don't store result of

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

- 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, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation.
- Pipelines can be executed in two ways: **demand driven** and **producer driven**

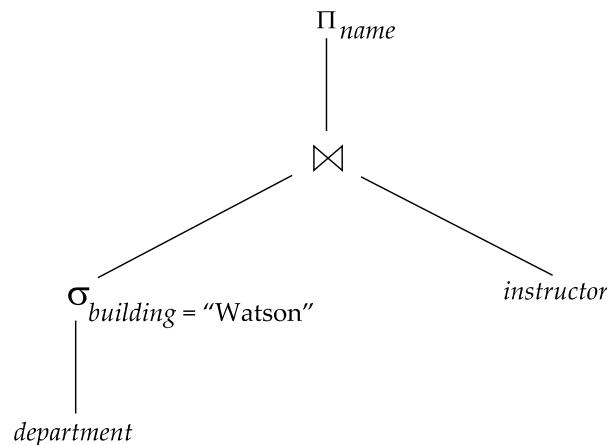
# Pipelining (Cont.)

- In **demand-driven** evaluation
  - System repeatedly requests next tuple from top level operation
  - Each operation requests next tuple from children operations as required
  - In between calls, operation has to maintain "**state**" so it knows what to return next



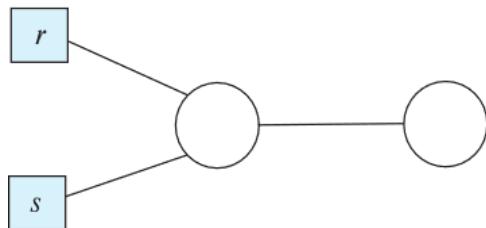
# Pipelining (Cont.)

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

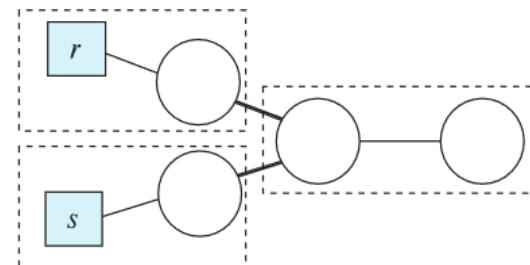


# Blocking Operations

- **Blocking operations:** cannot generate any output until all input is consumed
  - e.g., sorting, aggregation, ...
- But can often consume inputs from a pipeline, or produce outputs to a pipeline
- Key idea: blocking operations often have two suboperations
  - e.g., for sorting: run generation and merge
- Treat them as separate operations



(a) Logical Query

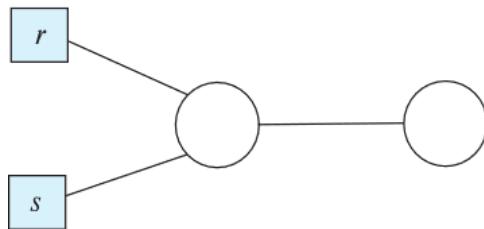


(b) Pipelined Plan

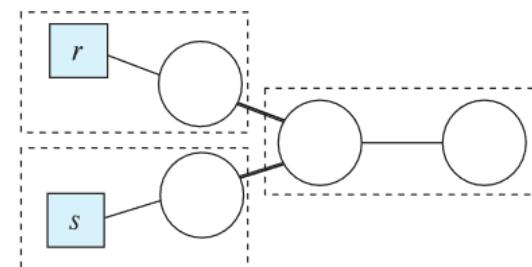
# Blocking Operations (Cont.)

- **Pipeline stages:**

- All operations in a stage run concurrently
- A stage can start only after preceding stages have completed execution



(a) Logical Query



(b) Pipelined Plan

# Pipelining for Continuous-Stream Data

- **Data streams**
  - Data entering database in a continuous manner
  - E.g., sensor networks, user clicks, ...
- **Continuous queries**
  - Results get updated as streaming data enters the database
  - Aggregation on windows is often used
    - e.g., **tumbling windows** divide time into units, e.g., hours, minutes
- Need to use pipelined processing algorithms
  - **Punctuations** used to infer when all data for a window has been received