



Chapter 15: Query Processing

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use



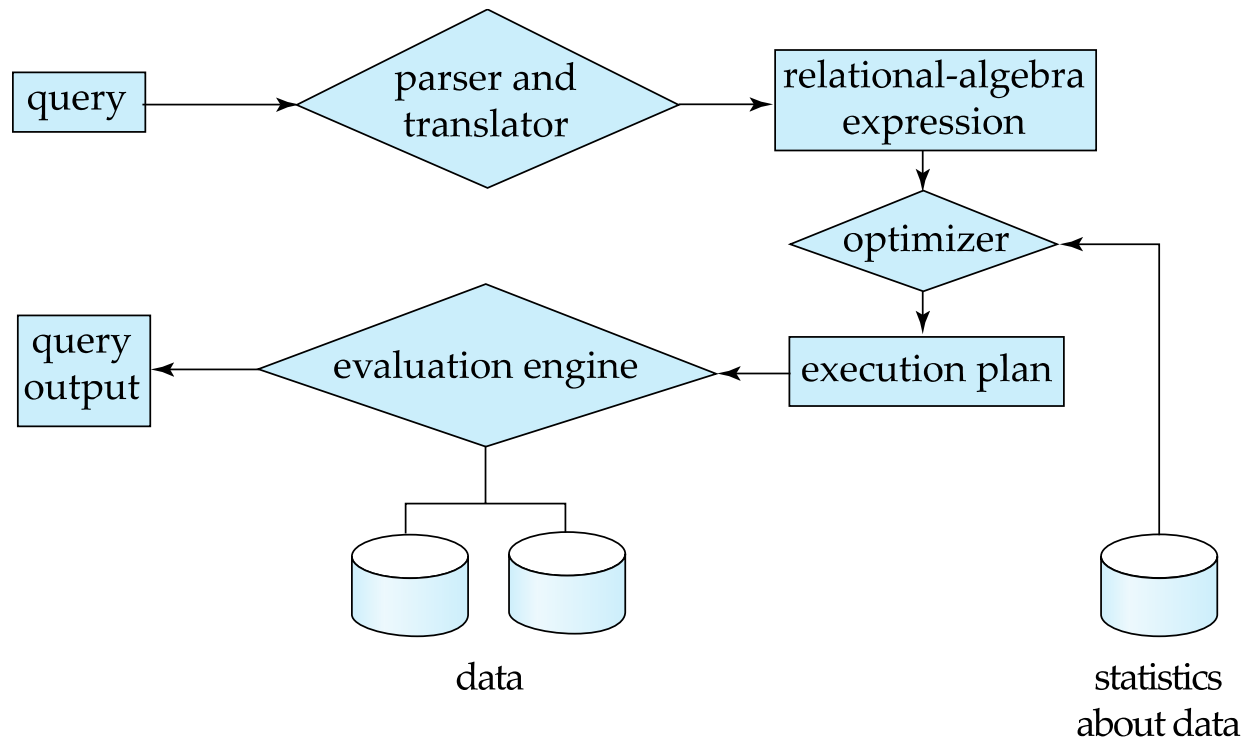
Chapter 15: Query Processing

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



Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation





Basic Steps in Query Processing (Cont.)

- Parsing and translation
 - translate the query into its internal form. This is then translated into relational algebra.
 - Parser checks syntax, verifies relations
- Evaluation
 - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.



Basic Steps in Query Processing: Optimization

- A relational algebra expression may have many equivalent expressions
 - E.g., $\sigma_{salary < 75000}(\Pi_{salary}(instructor))$ 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.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**. E.g.,:
 - Use an index on *salary* to find instructors with salary < 75000,
 - Or perform complete relation scan and discard instructors with salary ≥ 75000



Basic Steps: Optimization (Cont.)

- **Query Optimization:** Amongst all equivalent evaluation plans choose the one with lowest cost.
 - Cost is estimated using statistical information from the database catalog
 - e.g.. number of tuples in each relation, size of tuples, etc.
- In this chapter we study
 - How to measure query costs
 - Algorithms for evaluating relational algebra operations
 - How to combine algorithms for individual operations in order to evaluate a complete expression
- In Chapter 16
 - We study how to optimize queries, that is, how to find an evaluation plan with lowest estimated cost



Measures of Query Cost

- Many factors contribute to time cost
 - *disk access, CPU, and network communication*
- Cost can be measured based on
 - **response time**, i.e. total elapsed time for answering query, or
 - total **resource consumption**
- We use total resource consumption as cost metric
 - Response time harder to estimate, and minimizing resource consumption is a good idea in a shared database
- We ignore CPU costs for simplicity
 - Real systems do take CPU cost into account
 - Network costs must be considered for parallel systems
- We describe how estimate the cost of each operation
 - We do not include **cost to writing output to disk**



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: $t_S = 4$ msec and $t_T = 0.1$ msec
 - SSD: $t_S = 20$ -90 microsec and $t_T = 2$ -10 microsec for 4KB



Measures of Query Cost (Cont.)

- Required data may be buffer resident already, avoiding disk I/O
 - But hard to take into account for cost estimation
- Several algorithms can reduce disk IO by using extra buffer space
 - Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution
- **Worst case** estimates assume that no data is initially in buffer and only the minimum amount of memory needed for the operation is available
 - But more optimistic estimates are used in practice



Selection Operation

- **File scan**
- Algorithm **A1 (linear search)**. Scan each file block and test all records to see whether they satisfy the selection condition.
 - Cost estimate = b_r block transfers + 1 seek
 - b_r denotes number of blocks containing records from relation r
 - If selection is on a key attribute, can stop on finding record
 - cost = $(b_r/2)$ block transfers + 1 seek
 - Linear search can be applied regardless of
 - selection condition or
 - ordering of records in the file, or
 - availability of indices
- Note: binary search generally does not make sense since data is not stored consecutively
 - except when there is an index available,
 - and binary search requires more seeks than index search



Selections Using Indices

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



Selections Using Indices

- **A4 (secondary index, equality on 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 (clustering index, comparison).** (Relation is sorted on A)
 - For $\sigma_{A \geq V}(r)$ use index to find first tuple $\geq v$ and scan relation sequentially from there
 - For $\sigma_{A \leq V}(r)$ just scan relation sequentially till first tuple $> v$; do not use index
- **A6 (secondary 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 (合取):** $\sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(r)$
- **A7 (conjunctive selection using one index).**
 - Select a combination of θ_i and algorithms A1 through A7 that results in the least cost for $\sigma_{\theta_i}(r)$.
 - Test other conditions on tuple after fetching it into memory buffer.
- **A8 (conjunctive selection using composite index).**
 - Use appropriate composite (multiple-key) index if available.
- **A9 (conjunctive selection by intersection of identifiers).**
 - Requires indices with record pointers.
 - Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers.
 - Then fetch records from file
 - If some conditions do not have appropriate indices, apply test in memory.



Algorithms for Complex Selections

- **Disjunction (析取):** $\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r)$.
- **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
 - If very few records satisfy $\neg\theta$, and an index is applicable to θ
 - Find satisfying records using index and fetch from file

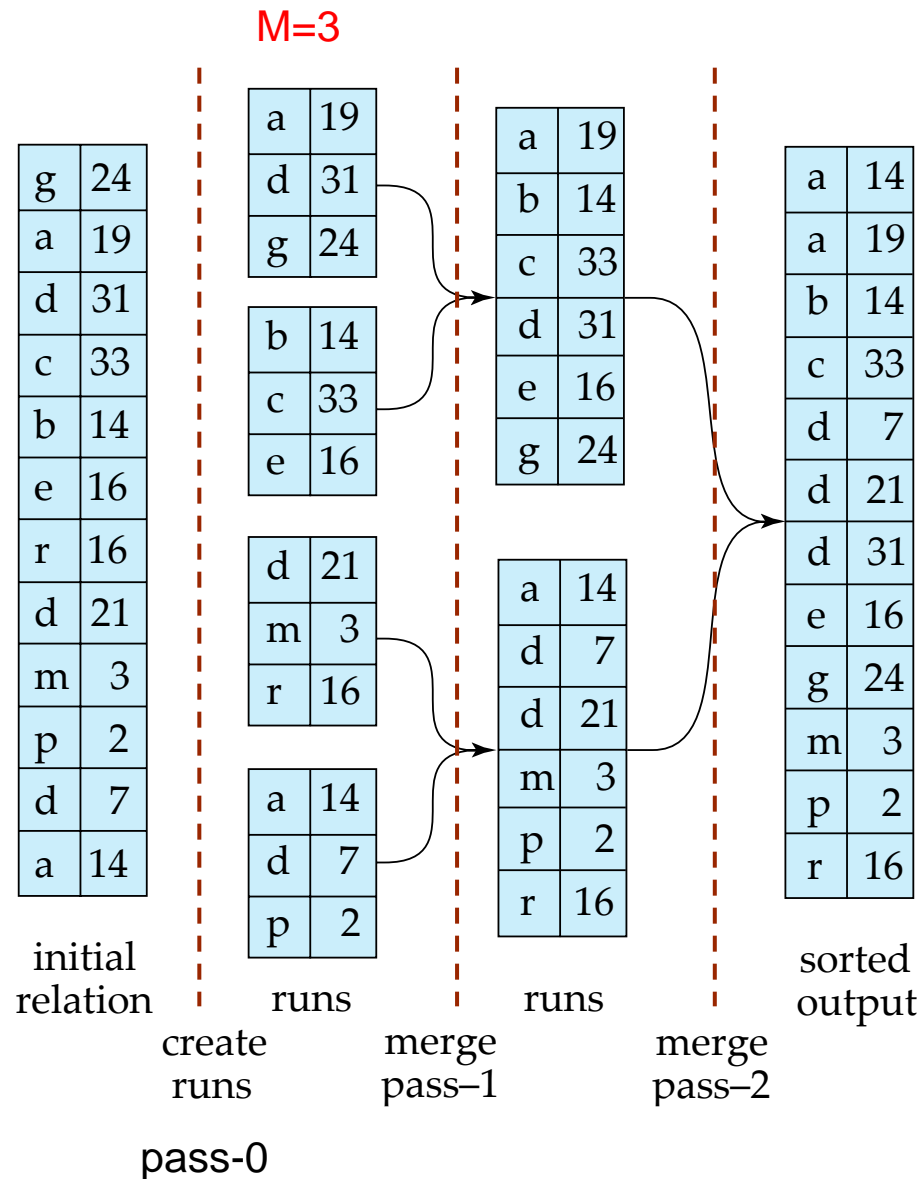


Sorting

- 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.
- Remember (Internal) merge sort?

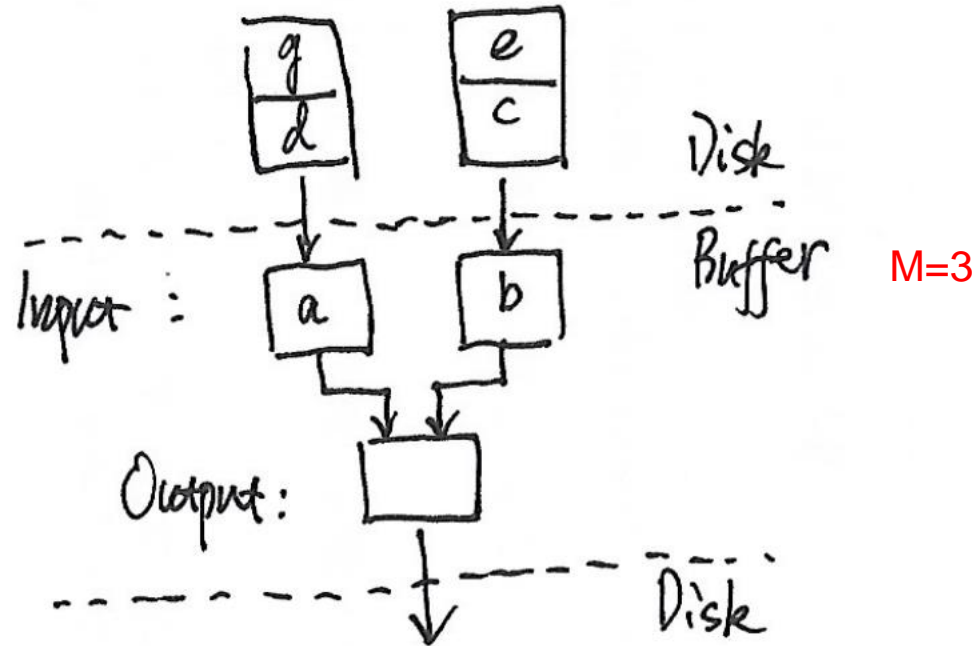


Example: External Sorting Using Sort-Merge





Example: External Sorting Using Sort-Merge



pass-1



2-Way External Merge Sort

Let's assume:

- Buffer size $M=3$
 - Input buffer size = 2 (can merge two blocks each time)
 - Output buffer size = 1
- Number of records $B_r = 12$, and one record per block

Sorting begins:

Pass-0: generate $\lceil B_r / M \rceil$ sorted runs

Pass-1: merge two runs each time, generate $\lceil B_r / M \rceil / 2$ sorted runs

Pass-2: $\lceil B_r / M \rceil / 2^2$ sorted runs ...

Pass-n: $\lceil B_r / M \rceil / 2^n$ sorted runs ...

Until: only 1 sorted run

Total No. of passes: $\lceil \log_2(B_r / M) \rceil + 1$

Total block transfers: $2 * B_r * (\text{No. of passes})$



External Sort-Merge

Let M denote memory size (in pages).

1. Pass-0: **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

$$N = \lceil B_r / M \rceil$$

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:

Total number of passes required: only 2



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.
- Total number of passes required:

$$\lceil \log_{M-1}(B_r / M) \rceil + 1$$

*B_r / M is the
number of runs
in Pass 0*



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: $\lceil \log_{\lfloor M/b_b \rfloor - 1} (B_r/M) \rceil$.
 - 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:
$$B_r (2 \lceil \log_{\lfloor M/b_b \rfloor - 1} (B_r/M) \rceil + 1)$$
 - 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:
$$2 \lceil B_r / M \rceil + \lceil B_r / b_b \rceil (2 \lceil \log_{\lfloor M/bb \rfloor} (B_r / M) \rceil - 1)$$



Join Operation

- Several different algorithms to implement joins
 - Nested-loop join
 - Block nested-loop join
 - Indexed nested-loop join
 - Merge-join
 - Hash-join
- Choice based on cost estimate
- Examples use the following information
 - Number of records of *student*: 5,000 *takes*: 10,000
 - Number of blocks of *student*: 100 *takes*: 400



Nested-Loop Join

- To compute the theta join $r \bowtie_{\theta} s$
 for each tuple t_r **in** r **do begin**
 for each tuple t_s **in** s **do begin**
 test pair (t_r, t_s) to see if they satisfy the join condition θ
 if they do, add $t_r \cdot t_s$ to the result.
 end
 end
- r is called the **outer relation** and s the **inner relation** of the join.
- Requires no indices and can be used with any kind of join condition.
- Expensive since it examines every pair of tuples in the two relations.



Nested-Loop Join (Cont.)

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

$n_r * b_s + b_r$ block transfers, plus $n_r + b_r$ seeks (假定内循环表每个盘块相邻)

- 外循环表每个盘块即使相邻, 最坏情况下也需要 b_r 个seek, 为什么?
- If the smaller relation fits entirely in memory, use that as the inner relation.
 - Reduces cost to $b_r + b_s$ block transfers and 2 seeks (假定内外循环的表每个盘块都相邻)
- Assuming worst case memory availability cost estimate is
 - with *student* as outer relation:
 - $5000 * 400 + 100 = 2,000,100$ block transfers,
 - $5000 + 100 = 5100$ seeks
 - with *takes* as the outer relation
 - $10000 * 100 + 400 = 1,000,400$ block transfers and 10,400 seeks
- If smaller relation (*student*) fits entirely in memory, the cost estimate will be 500 block transfers.
- Block nested-loops algorithm (next slide) is preferable.



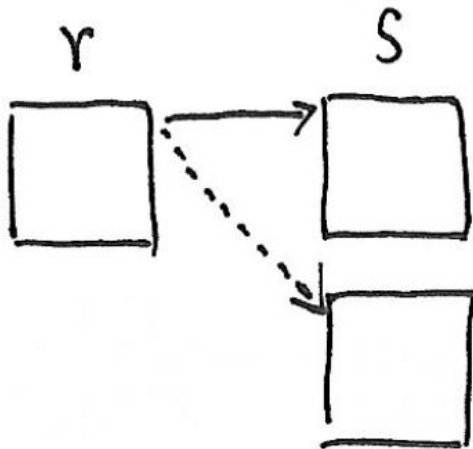
Block Nested-Loop Join

- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

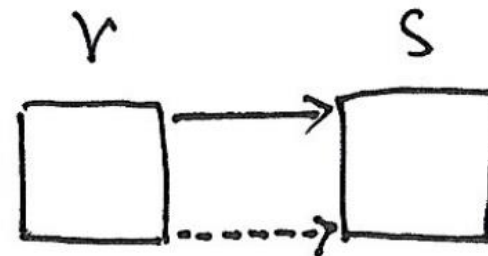
```
for each block  $B_r$  of  $r$  do begin  
    for each block  $B_s$  of  $s$  do begin  
        for each tuple  $t_r$  in  $B_r$  do begin  
            for each tuple  $t_s$  in  $B_s$  do begin  
                Check if  $(t_r, t_s)$  satisfy the join condition  
                if they do,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



Nested-Loop Join



Block Nested-Loop Join





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
- Best case: $b_r + b_s$ block transfers + 2 seeks.
- **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 equi-join attribute forms a key on 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
 - 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 page of r , and, for each tuple in r , we 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
 - 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.



Example of Nested-Loop Join Costs

- Compute $student \bowtie takes$, 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 (400 blocks)
- $student$ has 5000 tuples (100 blocks)
- Cost of block nested loops join
 - $400 * 100 + 100 = 40,100$ block transfers + $2 * 100 = 200$ seeks
 - assuming worst case memory
 - may be significantly less with more memory
- Cost of indexed nested loops join
 - $100 + 5000 * 5 = 25,100$ block transfers and **seeks**.
 - CPU cost likely to be less than that for block nested loops join



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
 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

	<i>a1</i>	<i>a2</i>
\xrightarrow{pr}	a	3
	b	1
	d	8
	d	13
	f	7
	m	5
	q	6
	<i>r</i>	

	<i>a1</i>	<i>a3</i>
\xrightarrow{ps}	a	A
	b	G
	c	L
	d	N
	m	B
	<i>s</i>	



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.
 - Assume b_b buffer blocks are allocated to each relation
- **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
 - Sequential scan more efficient than random lookup



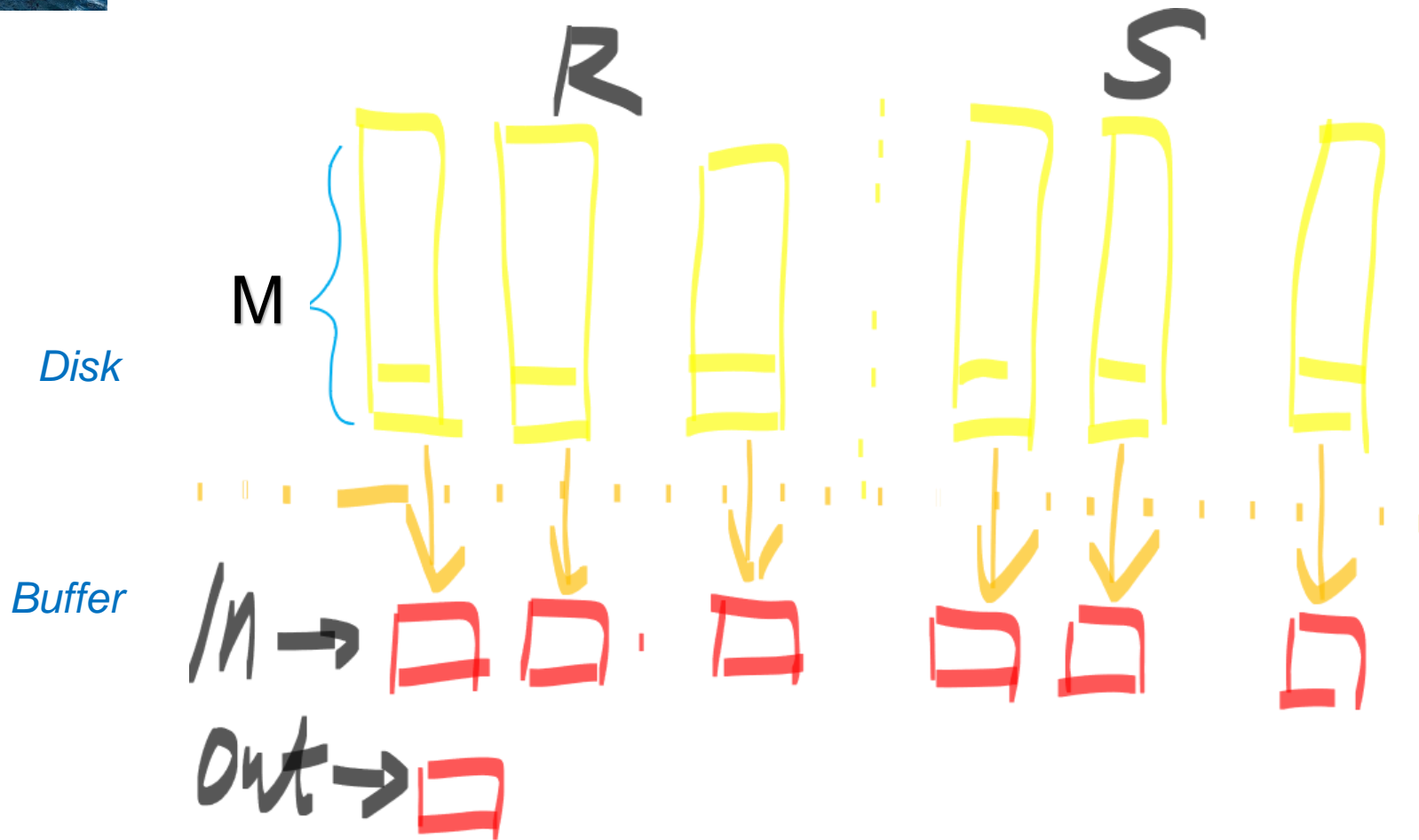
Refinement of Sort-Merge Join

- We can combine the merging phases in the *sorting* of R and S with the merging required for the join.
 - Let B_R and B_S be the # of pages of the **two** relations respectively
 - Let # of buffer pages = M, and $M > \sqrt{B_R + B_S}$
 - Pass 0: use the sorting refinement (p15.21) that produces runs of length M
 - Then, #runs of R is B_R/M , and #runs of S is B_S/M .
 - Pass 1: Allocate 1 page per run of **each relation**, and 'merge' while checking the join condition

因为满足 $B_R/M + B_S/M < M$, 所以两个表的所有有序段(runs)都可以同时合并



How to merge the two relations in Pass-1?





Refinement of Sort-Merge Join

- ❑ Cost:
 - ❑ Pass 0: read+write each relation
 - ❑ Pass 1: read each relation in (only) merging pass (omit writing of result tuples).
- ❑ In example $instructor \bowtie teaches$

Let $M=25$, cost goes to 1500 I/Os.

 - ❑ Pass 0: $(400+100)*2$
 - ❑ Pass 1: $400+100$

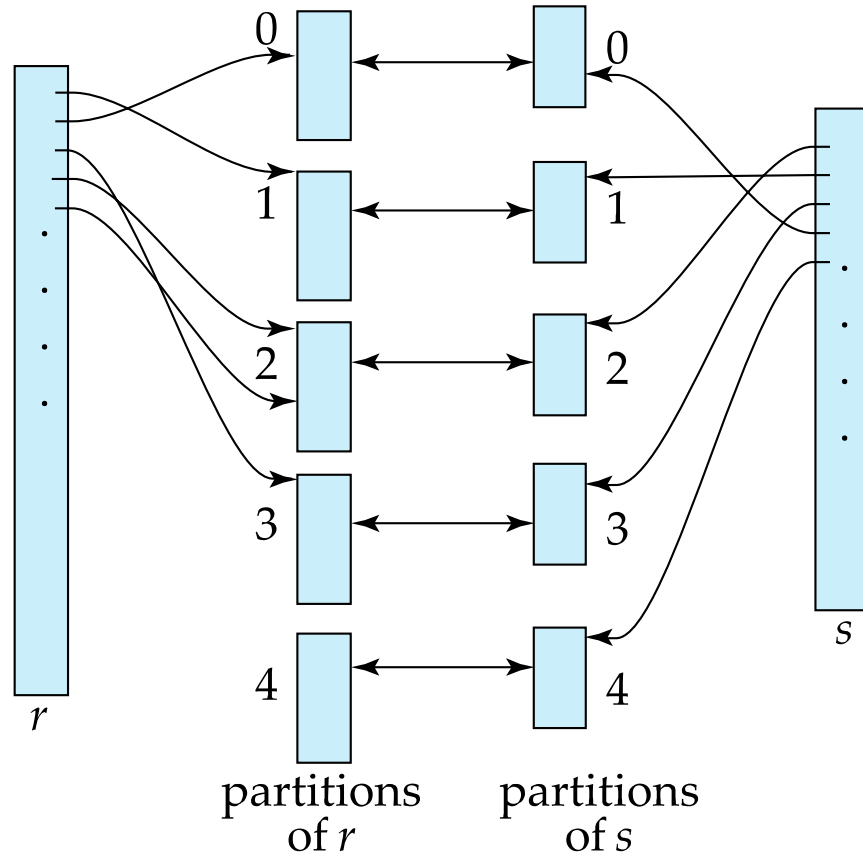


Hash-Join

- Applicable for equi-joins and natural joins.
- A hash function h is used to partition tuples of both relations
- h maps *JoinAttrs* values to $\{0, 1, \dots, n\}$, where *JoinAttrs* denotes the common attributes of r and s used in the natural join.
 - r_0, r_1, \dots, r_n denote partitions of r tuples
 - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[\text{JoinAttrs}])$.
 - s_0, s_1, \dots, s_n denotes partitions of s tuples
 - Each tuple $t_s \in s$ is put in partition s_i , where $i = h(t_s[\text{JoinAttrs}])$.
- *Note:* In book, Figure 12.10 r_i is denoted as H_{ri} , s_i is denoted as H_{si} and n is denoted as n_h .



Hash-Join (Cont.)





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**” (修正因子), 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
 - Use same partitioning method on r
 - Rarely required: e.g., with block size of 4 KB, recursive partitioning not needed for relations of $< 1\text{GB}$ with memory size of 2MB, or relations of $< 36\text{ GB}$ 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
$$3(b_r + b_s) + 4*n \text{ block transfers,}$$
and $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) + 2*n \text{ seeks}$
 - The overhead $4*n$ is usually quite small compared to $b_r + b_s$ and **can be ignored**. b_b is the size of input buffer.
 - See p716 of the textbook for an explanation
- If recursive partitioning required:
 - number of passes required for partitioning build relation s to less than M blocks per partition is $\lceil \log_{\lfloor M/b_b \rfloor - 1}(b_s/M) \rceil$
 - best to choose the smaller relation as the build relation.
 - Total cost estimate is:
$$2(b_r + b_s) \lceil \log_{\lfloor M/b_b \rfloor - 1}(b_s/M) \rceil + b_r + b_s \text{ block transfers} +$$
$$2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) \lceil \log_{\lfloor M/b_b \rfloor - 1}(b_s/M) \rceil \text{ seeks}$$
- 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



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

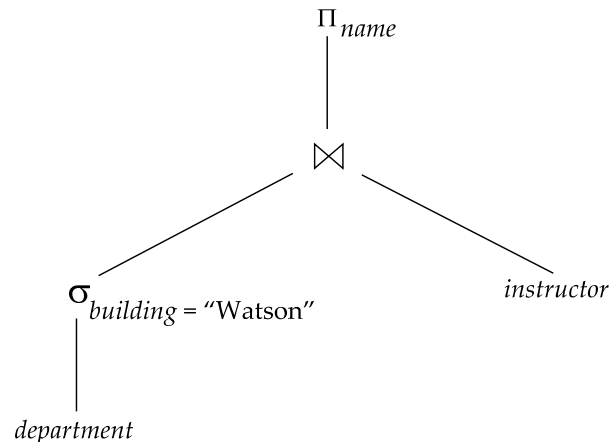


Materialization

- **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 quite high
 - Our cost formulas for operations ignore cost of writing results to disk, so
 - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk
- **Double buffering**: use two output buffers for each operation, when one is full write it to disk while the other is getting filled
 - Allows overlap of disk writes with computation and reduces execution time



Pipelining

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

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

- instead, pass tuples directly to the join.. Similarly, don't store result of join, pass tuples directly to projection.
- Much cheaper than materialization: no need to store a temporary relation to disk.
- Pipelining may not always be possible – e.g., sort, hash-join.
- For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation.
- Pipelines can be executed in two ways: **demand driven** and **producer driven**



Pipelining (Cont.)

- In **demand driven** or **lazy** evaluation
 - system repeatedly requests next tuple from top level operation
 - Each operation requests next tuple from children operations as required, in order to output its next tuple
 - In between calls, operation has to maintain “**state**” so it knows what to return next
- In **producer-driven** or **eager** pipelining
 - Operators produce tuples eagerly and pass them up to their parents
 - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
 - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
 - System schedules operations that have space in output buffer and can process more input tuples
- Alternative name: **pull** and **push** models of pipelining



End of Chapter 15



Assignments

- 15.1, 15.3
- Submission deadline: Dec 8, 2024



- 原来这么多数据库的部分底层模块是用c++实现的
 - https://blog.csdn.net/m0_75032658/article/details/137469550
- 怎么实现一个简单的数据库系统？ - 知乎
<https://www.zhihu.com/question/26802517>
- 做数据库内核开发的是不是很少？ - 知乎
<https://www.zhihu.com/question/445283801>