Chapter 15: Query Processing

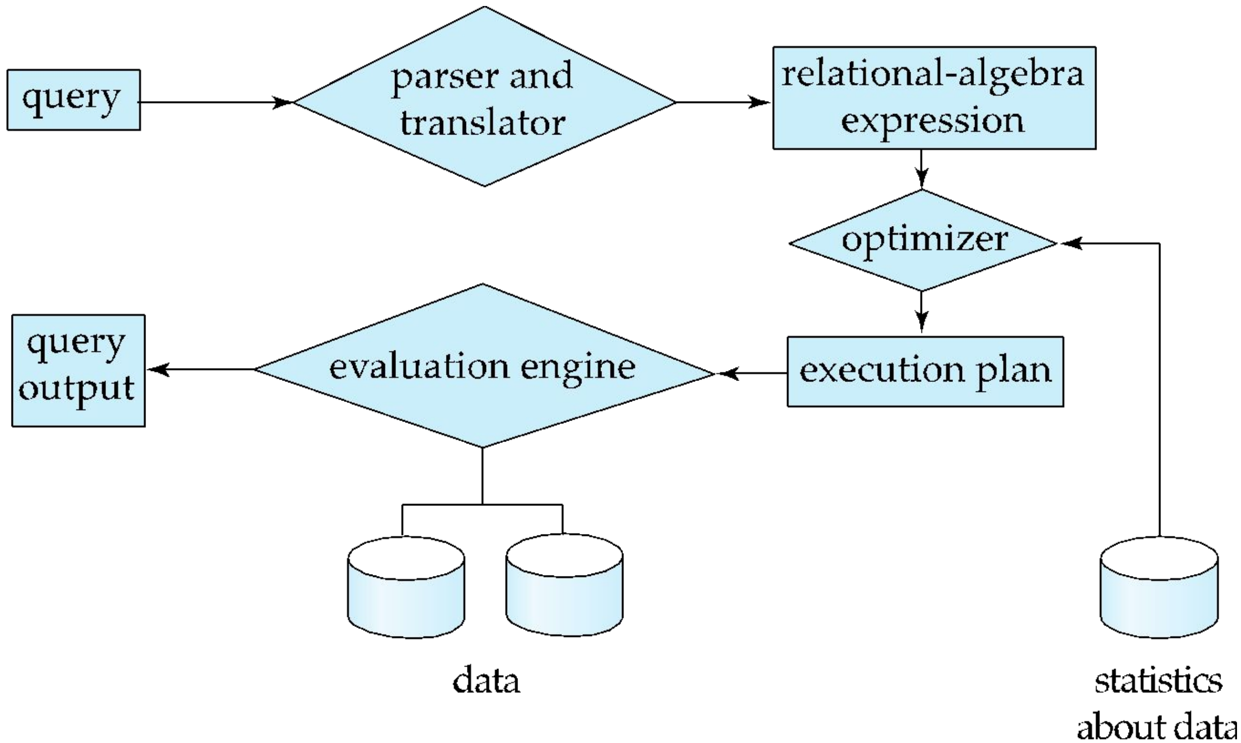
Database System Concepts, 7th Ed.

Chapter 15: Query Processing

- Overview
- Measures of Query Cost
- Sorting
- Selection Operation
- 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: Optimization

- A relational algebra expression may have many equivalent expressions
  - E.g., $\sigma_{salary<75000}(\prod_{salary}(instructor))$
    is equivalent to
    $\prod_{salary}(\sigma_{salary<75000}(instructor))$
- 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

- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- **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.

Measures of Query Cost

- Many factors contribute to time cost
  - *disk access, CPU*, and network *communication*
- 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
  - 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 I/O 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

Sorting

In-Memory vs. External Sorting

- Assume we want to sort 60GB of data on a machine with only 8GB of RAM
  - In-Memory Sort (e.g., Quicksort) ?
    - Yes, but data do not fit in memory
    - What about relying on virtual memory?

- In this case, external sorting is needed
  - In-memory sorting is *orthogonal* to external sorting!

A Simple Two-Way Merge Sort

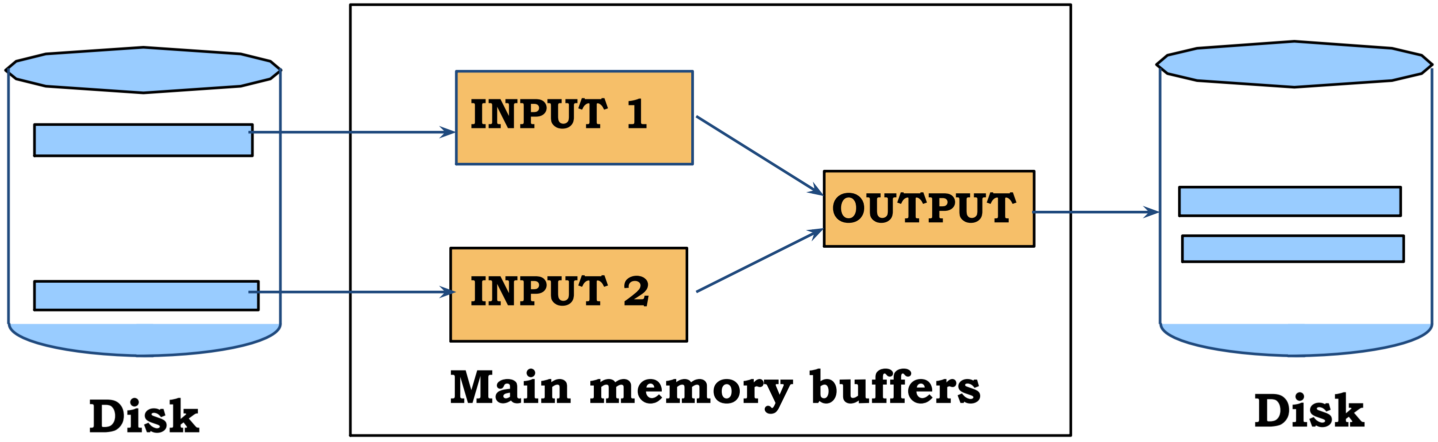
- IDEA: Sort sub-files that can fit in memory and merge

- Let us refer to each sorted sub-file as a *run*

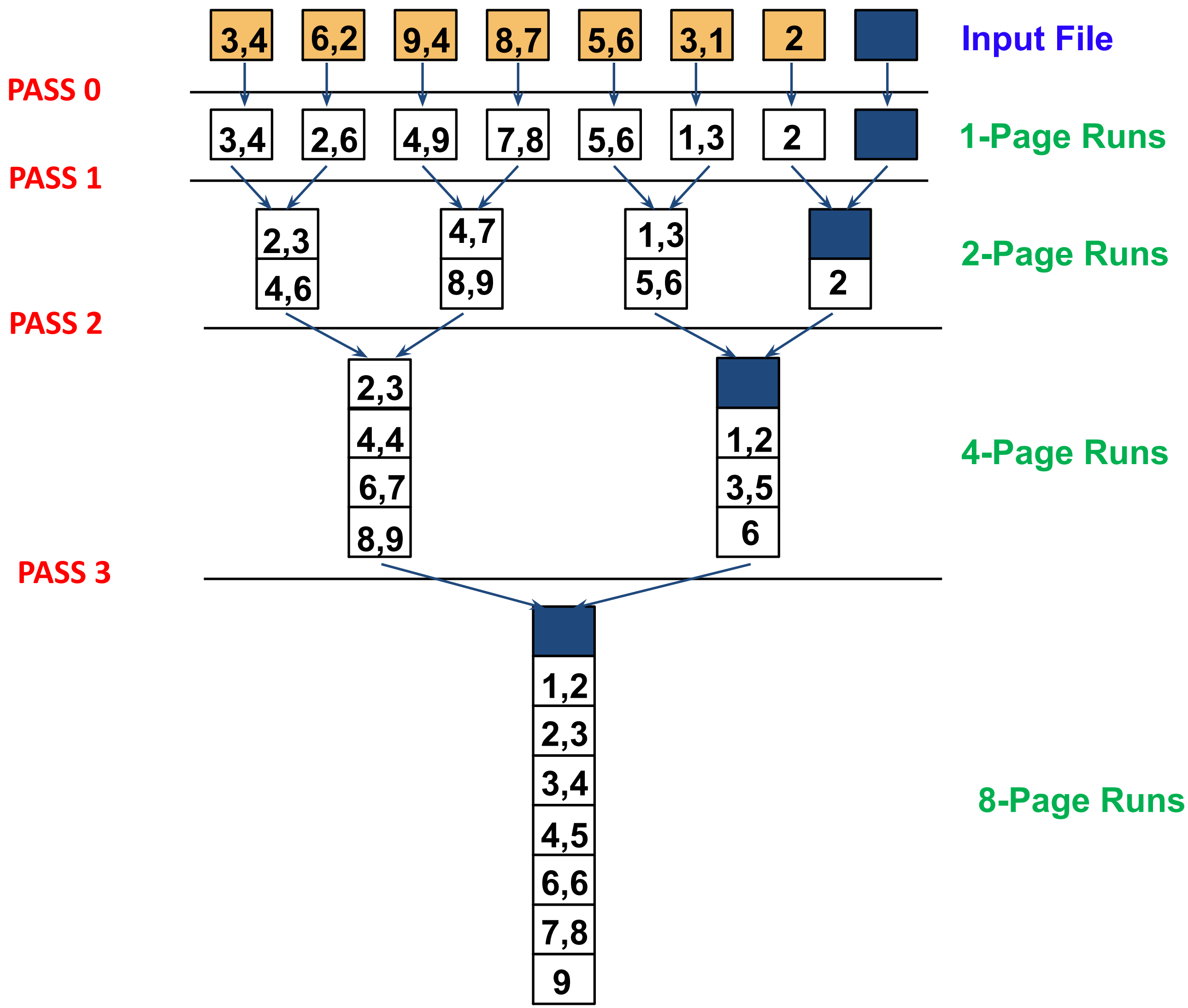
- Algorithm:
  - Pass 0: Read a page into memory, sort it, write it
    - 1-page runs are produced
  - Passes 1, 2, …: Merge *pairs* (hence, 2-way) of runs to produce longer runs until only one run is left

A Simple Two-Way Merge Sort

- Algorithm:
  - Pass 0: Read a page into memory, sort it, write it
    - How many buffer pages are needed? ONE
  - Passes 1, 2, …: Merge *pairs* (hence, 2-way) of runs to produce longer runs until only one run is left
    - How many buffer pages are needed? THREE



2-Way Merge Sort: An Example



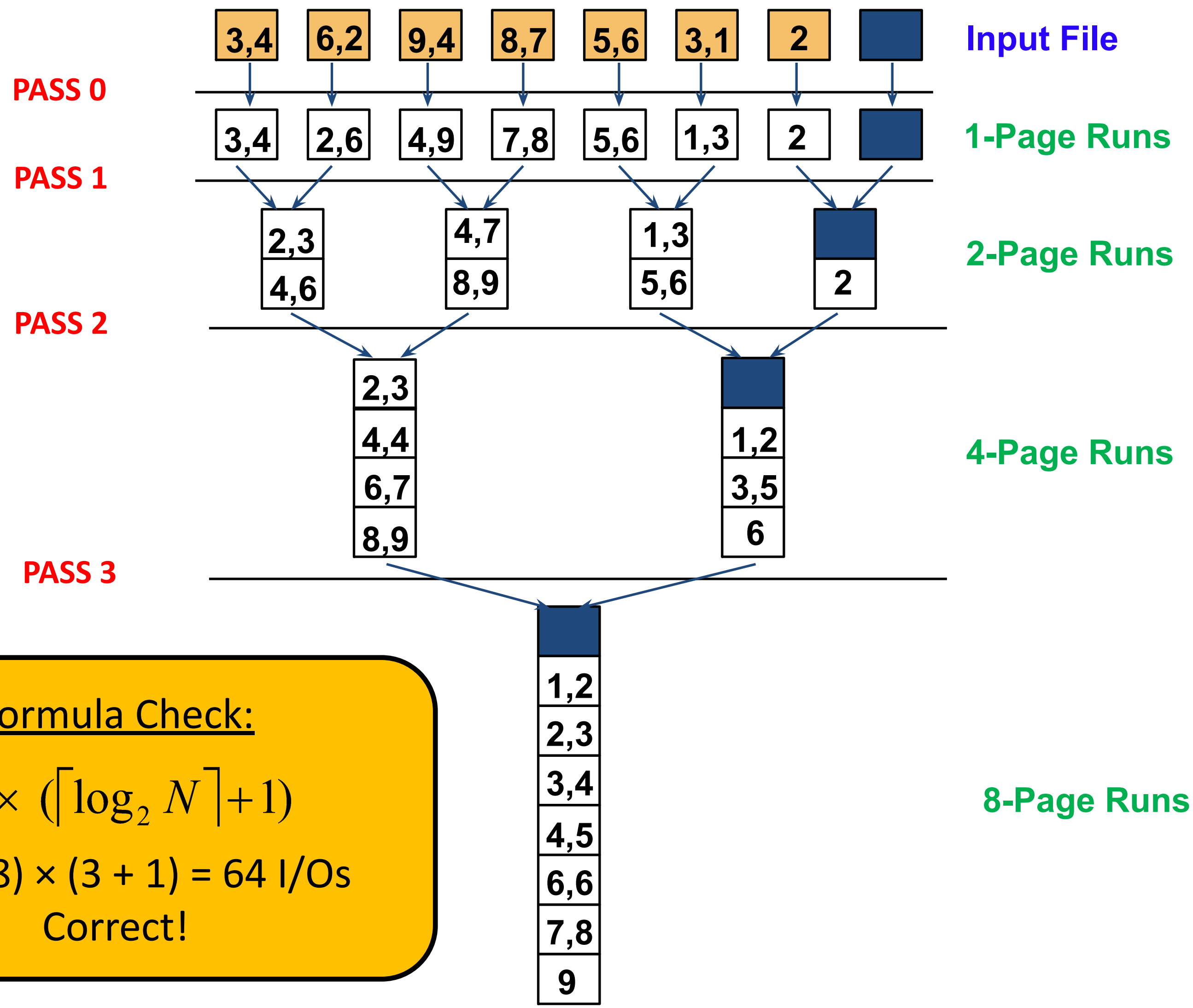
2-Way Merge Sort: I/O Cost Analysis

- If the number of pages in the input file is 2^k
  - How many runs are produced in pass 0 and of what size?
    - 2^k 1-page runs
  - How many runs are produced in pass 1 and of what size?
    - 2^{k-1} 2-page runs
  - How many runs are produced in pass 2 and of what size?
    - 2^{k-2} 4-page runs
  - How many runs are produced in pass k and of what size?
    - 2^{k-k} 2^k-page runs (or 1 run of size 2^k)
  - For *N* number of pages, how many passes are incurred?
    - $\boxed{\lceil \log_2 N \rceil + 1}$
  - How many pages do we read and write in each pass?
    - 2*N*
  - *What is the overall cost?*
    - $\boxed{2N \times (\lceil \log_2 N \rceil + 1)}$

2-Way Merge Sort: An Example

$$\lceil \log_2 8 \rceil + 1$$

= 4 passes



<u>Formula Check:</u>

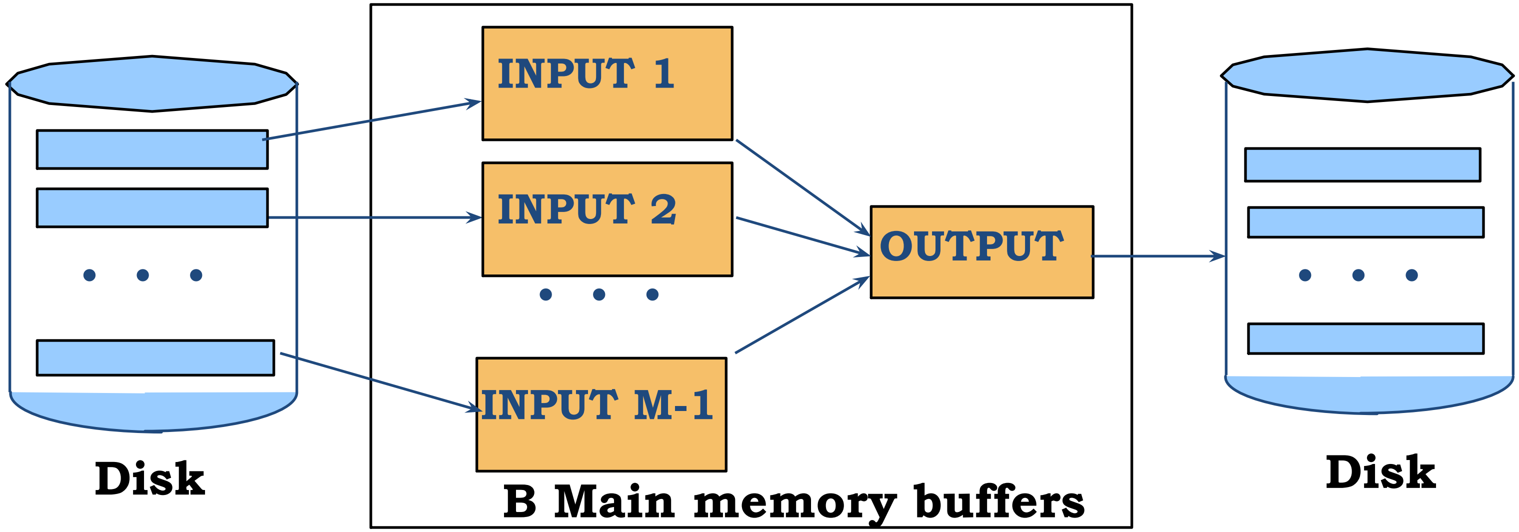
$$2N \times (\lceil \log_2 N \rceil + 1)$$

= (2 × 8) × (3 + 1) = 64 I/Os

Correct!

M-Way Merge Sort

- How can we sort a file with N pages using ***M*** buffer pages?
  - Pass 0: use M buffer pages and sort internally
    - This will produce $\lceil N/M \rceil$ sorted M-page runs
  - Passes 1, 2, ….: use M – 1 buffer pages for input and the remaining page for output; do (M-1)-way merge in each run



B-Way Merge Sort: I/O Cost Analysis

- I/O cost = 2N × Number of passes

- Number of passes = $1 + \lceil \log_{M-1} \lceil N/M \rceil \rceil$

- Assume the previous example (i.e., 8 pages), *but* using 5 buffer pages (instead of 3)
  - I/O cost = 32 (*as opposed to 64*)

- Therefore, increasing the number of buffer pages minimizes the number of passes and accordingly the I/O cost!

Number of Passes of M-Way Sort

N	M=3	M=5	M=9	M=17	M=129	M=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

High Fan-in during merging is crucial!

Relational Operators – Implementation Algorithms

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
  - $Cost = h_i * (t_T + t_S) + t_S + t_T$
- **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 $\sigma_{A \le V}(r)$ or $\sigma_{A \ge V}(r)$ by using
  - a linear file scan,
  - or by using indices in the following ways:
- **A5** (**clustering index, comparison**). (Relation is sorted on A)
  - For $\sigma_{A \ge V}(r)$ use index to find first tuple ≥ *v* and scan relation sequentially from there
  - For $\sigma_{A \le V}(r)$ just scan relation sequentially till first tuple > *v;* do not use index
- **A6** (**clustering index, comparison**).
  - For $\sigma_{A \ge V}(r)$ use index to find first index entry ≥ *v* and scan index sequentially from there, to find pointers to records.
  - For $\sigma_{A \le V}(r)$ just scan leaf pages of index finding pointers to records, till first entry > *v*
  - In either case, retrieve records that are pointed to
  - requires an I/O per record; Linear file scan may be cheaper!

Implementation of Complex Selections

- **Conjunction:** $\sigma_{\theta 1} \wedge_{\theta 2} \wedge \ldots_{\theta n}(r)$
- **A7** (**conjunctive selection using one index**).
  - Select a combination of θ_i and algorithms A1 through A7 that results in the least cost for $\sigma_{\theta i}(r)$.
  - Test other conditions on tuple after fetching it into memory buffer.
- **A8** (**conjunctive selection using composite index**).
  - Use appropriate composite (multiple-key) index if available.
- **A9** (**conjunctive selection by intersection of identifiers**).
  - Requires indices with record pointers.
  - Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers.
  - Then fetch records from file
  - If some conditions do not have appropriate indices, apply test in memory.

Algorithms for Complex Selections

- **Disjunction:** $\sigma_{\theta 1} \lor {}_{\theta 2} \lor \ldots {}_{\theta n} (r)$.
- **A10** (**disjunctive selection by union of identifiers**).
  - Applicable if *all* conditions have available indices.
    - Otherwise use linear scan.
  - Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
  - Then fetch records from file
- **Negation:** $\sigma_{\neg\theta}(r)$
  - Use linear scan on file
  - If very few records satisfy $\neg\theta$, and an index is applicable to θ
    - Find satisfying records using index and fetch from file

Join Operation

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

Nested-Loop Join

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

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

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

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 $t_r \bullet t_s$ to the result.
            end
        end
    end
end

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.*

- If indices are available on join attributes of both *r* and *s,* use the relation with fewer tuples as the outer relation.

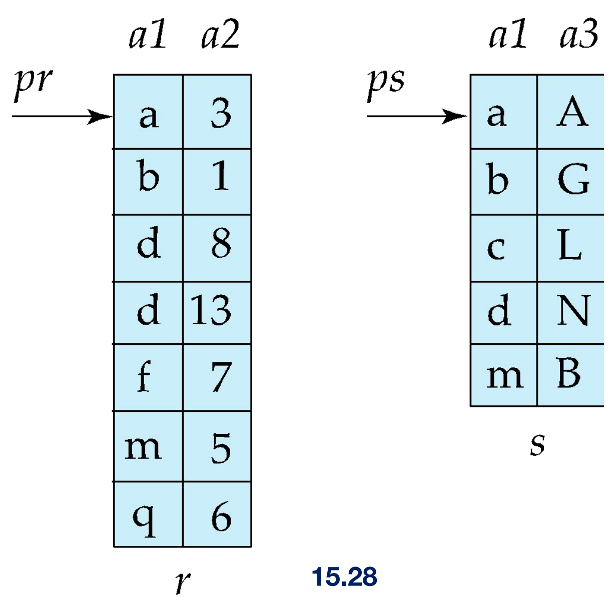
Merge-Join

1. Can be used only for equi-joins and natural joins

2. Used when relations are sorted on the join attributes. If not sorted, first sort both relations on their join attribute.

3. Merge joins are faster and uses less memory than hash joins.

3. 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



Hash Join

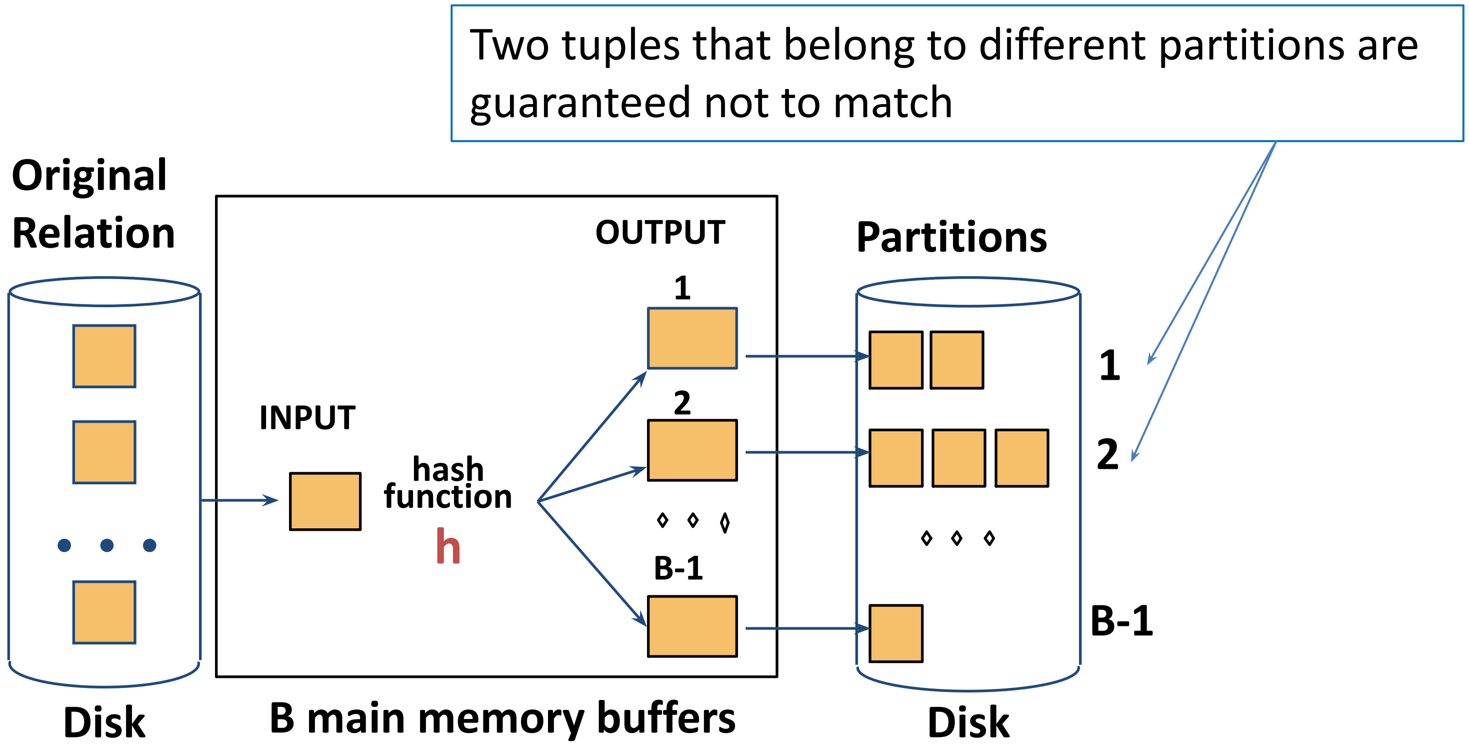
- The join algorithm based on hashing has two phases:
  - Partitioning (also called *Building*) Phase
  - Probing (also called *Matching*) Phase

- Idea: Hash both relations *on the join attribute* into ***k*** partitions, using the ***same*** hash function ***h***

- Premise: R tuples in partition ***i*** can join only with S tuples in the same partition ***i***

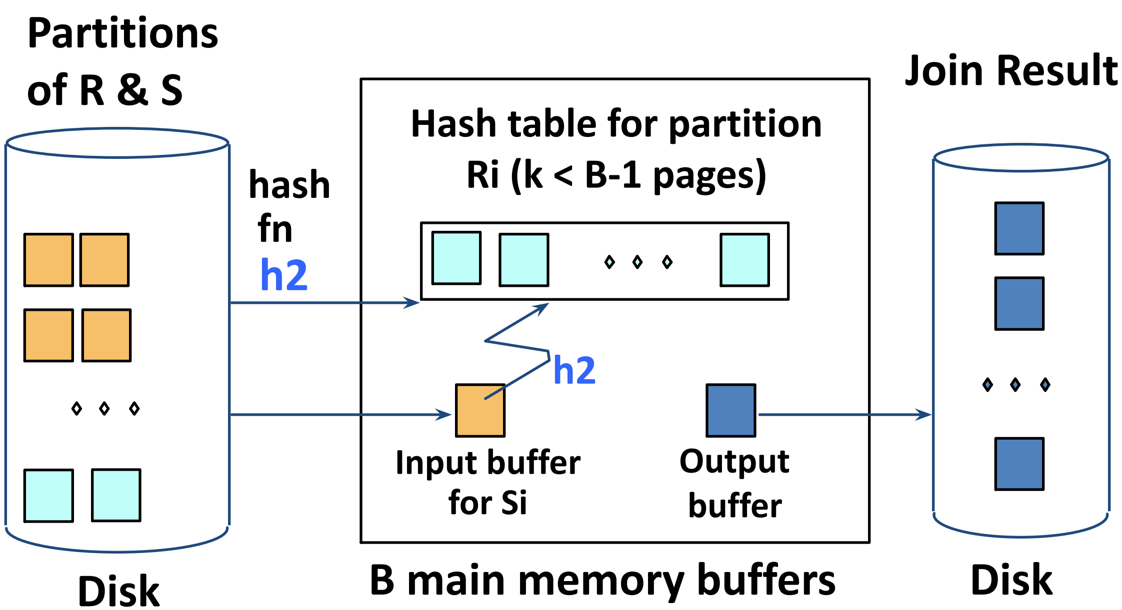
Hash Join: Partitioning Phase

- Partition both relations using hash function ***h***



Hash Join: Probing Phase

- Read in a partition of R, hash it using ***h2* (!= *h*)**
- Scan the corresponding partition of S and search for matches



Complex Joins

- 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 $r \bowtie_{\theta i} s$
    - 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** 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.
- **Projection**:
  - perform projection on each tuple
  - followed by duplicate elimination.

Other Operations : Aggregation

- **Aggregation** 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

Other Operations : Set Operations

- **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
  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

- E.g., Set operations using hashing:
  1. as before partition *r* and *s*,
  2. as before, process each partition *i* as follows
     1. build a hash index on r_i
     2. Process s_i as follows
        - $r \cap s$:
          1. output tuples in s_i to the result if they are already there 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.

Answering Keyword Queries

- Indices mapping keywords to documents
  - For each keyword, store sorted list of document IDs that contain the keyword
    - Commonly referred to as a **inverted index**
    - E.g.,: database: d1, d4, d11, d45, d77, d123
      distributed: d4, d8, d11, d56, d77, d121, d333
  - To answer a query with several keywords, compute intersection of lists corresponding to those keywords
- To support ranking, inverted lists store extra information
  - "**Term frequency**" of the keyword in the document
  - "**Inverse document frequency**" of the keyword
  - **Page rank** of the document/web page

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.

Pipelining

- **Pipelined evaluation**: evaluate several operations simultaneously, passing the results of one operation on to the next.
- Much cheaper than materialization: no need to store a temporary relation to disk.
- Pipelining may not always be possible – e.g., sort, hash-join.
- Pipelines can be executed in two ways:
  - **demand driven** (lazy evaluation or pull model)
  - **producer driven** (eager evaluation or push model).