CHAPTER 12

Strategies for Query Processing

Slide Credits: Elmasri and Navathe

Introduction

- DBMS techniques to process a query
  - Scanner identifies query tokens
  - Parser checks the query syntax
  - Validation checks all attribute and relation names
  - Query tree (or query graph) created
  - Execution strategy or query plan devised
- Query optimization
  - Planning a good execution strategy

Query Processing

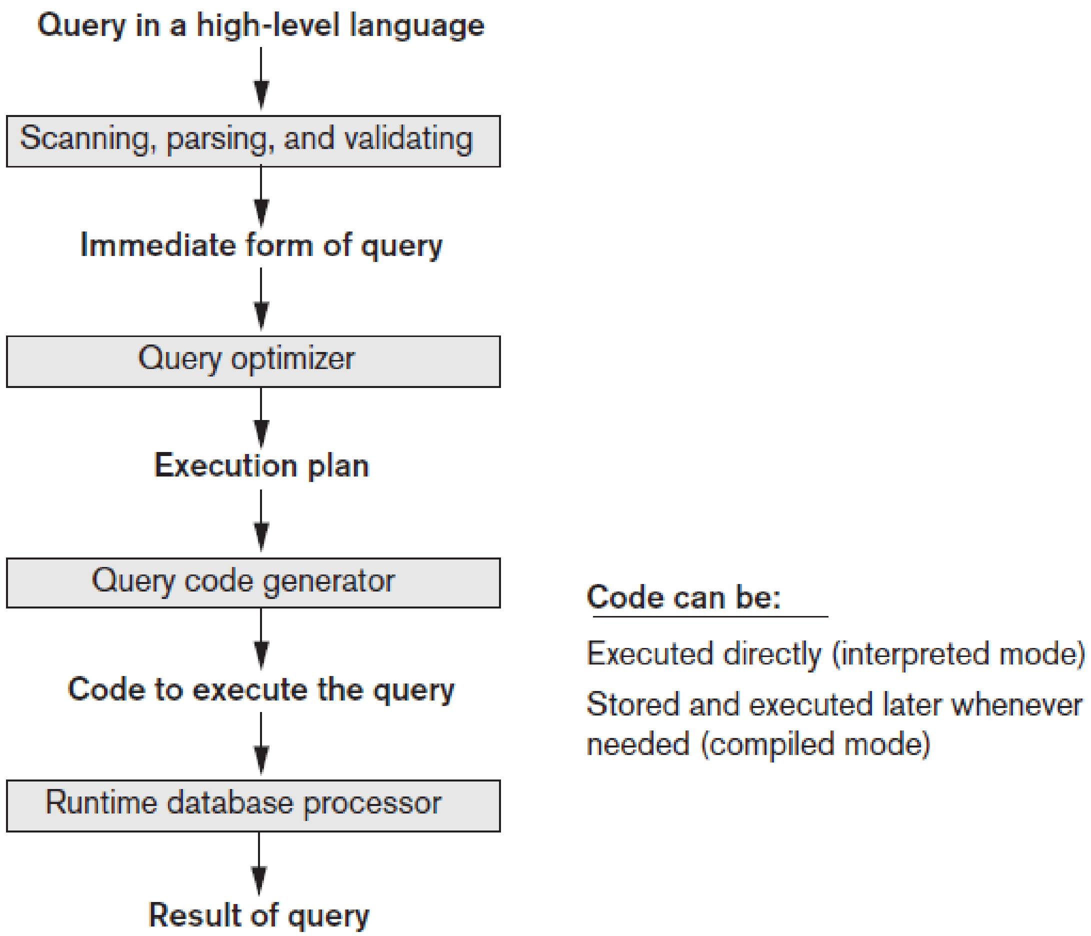


Figure 12.1 Typical steps when processing a high-level query

12.1 Translating SQL Queries into Relational Algebra and Other Operators

- SQL
  - Query language used in most RDBMSs
- Query decomposed into query blocks
  - Basic units that can be translated into the algebraic operators
  - Contains single SELECT-FROM-WHERE expression
    - May contain GROUP BY and HAVING clauses

Translating SQL Queries (cont'd.)

- Example:

```
SELECT Lname, Fname
FROM EMPLOYEE
WHERE Salary > ( SELECT MAX (Salary)
                 FROM   EMPLOYEE
                 WHERE  Dno=5 );
```

  - Inner block

```
( SELECT MAX (Salary)
  FROM   EMPLOYEE
  WHERE  Dno=5 )
```

  - Outer block

```
SELECT  Lname, Fname
FROM    EMPLOYEE
WHERE Salary > c
```

Translating SQL Queries (cont'd.)

- Example (cont'd.)
  - Inner block translated into:

$$\Im_{\text{MAX Salary}}(\sigma_{Dno=5}(\text{EMPLOYEE}))$$

  - Outer block translated into:

$$\pi_{\text{Lname,Fname}}(\sigma_{Salary>c}(\text{EMPLOYEE}))$$

- Query optimizer chooses execution plan for each query block

Additional Operators Semi-Join and Anti-Join

- Semi-join
  - Generally used for unnesting EXISTS, IN, and ANY subqueries
  - Syntax: T1.X S = T2.Y
    - T1 is the left table and T2 is the right table of the semi-join
  - A row of T1 is returned as soon as T1.X finds a match with any value of T2.Y without searching for further matches

Additional Operators Semi-Join and Anti-Join (cont'd.)

- Anti-join
  - Used for unnesting NOT EXISTS, NOT IN, and ALL subqueries
  - Syntax: T1.x *A* = T2.y
    - T1 is the left table and T2 is the right table of the anti-join
  - A row of T1 is rejected as soon as T1.x finds a match with any value of T2.y
  - A row of T1 is returned only if T1.x does not match with any value of T2.y

12.2 Algorithms for External Sorting

- Sorting is an often-used algorithm in query processing
- External sorting
  - Algorithms suitable for large files that do not fit entirely in main memory
  - Sort-merge strategy based on sorting smaller subfiles (runs) and merging the sorted runs
  - Requires buffer space in main memory
    - DBMS cache

Algorithms for External Sorting (cont'd.)

- Degree of merging
  - Number of sorted subfiles that can be merged in each merge step
- Performance of the sort-merge algorithm
  - Number of disk block reads and writes before sorting is completed

12.3 Algorithms for SELECT Operation

- SELECT operation
  - Search operation to locate records in a disk file that satisfy a certain condition
  - File scan or index scan (if search involves an index)
- Search methods for simple selection
  - S1: Linear search (brute force algorithm)
  - S2: Binary search
  - S3a: Using a primary index
  - S3b: Using a hash key

Algorithms for SELECT Operation (cont'd.)

- Search methods for simple selection (cont'd.)
  - S4: Using a primary index to retrieve multiple records
  - S5: Using a clustering index to retrieve multiple records
  - S6: Using a secondary (B+ -tree) index on an equality comparison
  - S7a: Using a bitmap index
  - S7b: Using a functional index

Algorithms for SELECT Operation (cont'd.)

- Search methods for conjunctive (logical AND) selection
  - Using an individual index
  - Using a composite index
  - Intersection of record pointers
- Disjunctive (logical OR) selection
  - Harder to process and optimize

Algorithms for SELECT Operation (cont'd.)

- Selectivity
  - Ratio of the number of records (tuples) that satisfy the condition to the total number of records (tuples) in the file
  - Number between zero (no records satisfy condition) and one (all records satisfy condition)
- Query optimizer receives input from system catalog to estimate selectivity

12.4 Implementing the JOIN Operation

- JOIN operation
  - One of the most time consuming in query processing
  - EQUIJOIN (NATURAL JOIN)
  - Two-way or multiway joins
- Methods for implementing joins
  - J1: Nested-loop join (nested-block join)
  - J2: Index-based nested-loop join
  - J3: Sort-merge join
  - J4: Partition-hash join

Implementing the JOIN Operation (cont'd.)

- Available buffer space has important effect on some JOIN algorithms
- Nested-loop approach
  - Read as many blocks as possible at a time into memory from the file whose records are used for the outer loop
  - Advantageous to use the file with fewer blocks as the outer-loop file

Implementing the JOIN Operation (cont'd.)

- Join selection factor
  - Fraction of records in one file that will be joined with records in another file
  - Depends on the particular equijoin condition with another file
  - Affects join performance
- Partition-hash join
  - Each file is partitioned into *M* partitions using the same partitioning hash function on the join attributes
  - Each pair of corresponding partitions is joined

Implementing the JOIN Operation (cont’d.)

- Hybrid hash-join
  - Variation of partition hash-join
  - Joining phase for one of the partitions is included in the partition
  - Goal: join as many records during the partitioning phase to save cost of storing records on disk and then rereading during the joining phase

12.5 Algorithms for PROJECT and Set Operations

- PROJECT operation
  - After projecting *R* on only the columns in the list of attributes, any duplicates are removed by treating the result strictly as a set of tuples
- Default for SQL queries
  - No elimination of duplicates from the query result
    - Duplicates eliminated only if the keyword DISTINCT is included

Slide Credits: Elmasri and Navathe

Algorithms for PROJECT and Set Operations (cont'd.)

- Set operations
  - UNION
  - INTERSECTION
  - SET DIFFERENCE
  - CARTESIAN PRODUCT
- Set operations sometimes expensive to implement
  - Sort-merge technique
  - Hashing

Algorithms for PROJECT and Set Operations (cont'd.)

- Use of anti-join for SET DIFFERENCE
  - EXCEPT or MINUS in SQL
  - Example: Find which departments have no employees

```
Select Dnumber from DEPARTMENT MINUS Select Dno from EMPLOYEE;
```

```
SELECT  DISTINCT DEPARTMENT.Dnumber
FROM    DEPARTMENT, EMPLOYEE
WHERE DEPARTMENT.Dnumber A = EMPLOYEE.Dno
```

12.6 Implementing Aggregate Operations and Different Types of JOINs

- Aggregate operators
  - MIN, MAX, COUNT, AVERAGE, SUM
  - Can be computed by a table scan or using an appropriate index
- Example:

```
SELECT    MAX(Salary)
FROM      EMPLOYEE;
```

  - If an (ascending) B+ -tree index on Salary exists:
    - Optimizer can use the Salary index to search for the largest Salary value
    - Follow the rightmost pointer in each index node from the root to the rightmost leaf

Implementing Aggregate Operations and Different Types of JOINs (cont'd.)

- AVERAGE or SUM
  - Index can be used if it is a dense index
  - Computation applied to the values in the index
- COUNT
  - Number of values can be computed from the index

Implementing Aggregate Operations and Different Types of JOINs (cont'd.)

- Standard JOIN (called INNER JOIN in SQL)
- Variations of joins
  - Outer join
    - Left, right, and full
    - Example:

```
SELECT  E.Lname, E.Fname, D.Dname
FROM    (EMPLOYEE E LEFT OUTER JOIN DEPARTMENT D ON E.Dno = D.Dnumber);
```

  - Semi-Join
  - Anti-Join
  - Non-Equi-Join

12.7 Combining Operations Using Pipelining

- SQL query translated into relational algebra expression
  - Sequence of relational operations
- Materialized evaluation
  - Creating, storing, and passing temporary results
- General query goal: minimize the number of temporary files
- Pipelining or stream-based processing
  - Combines several operations into one
  - Avoids writing temporary files

Combining Operations Using Pipelining (cont'd.)

- Pipelined evaluation benefits
  - Avoiding cost and time delay associated with writing intermediate results to disk
  - Being able to start generating results as quickly as possible
- Iterator
  - Operation implemented in such a way that it outputs one tuple at a time
  - Many iterators may be active at one time

Combining Operations Using Pipelining (cont'd.)

- Iterator interface methods
  - Open()
  - Get_Next()
  - Close()
- Some physical operators may not lend themselves to the iterator interface concept
  - Pipelining not supported
- Iterator concept can also be applied to access methods

12.8 Parallel Algorithms for Query Processing

- Parallel database architecture approaches
  - Shared-memory architecture
    - Multiple processors can access common main memory region
  - Shared-disk architecture
    - Every processor has its own memory
    - Machines have access to all disks
  - Shared-nothing architecture
    - Each processor has own memory and disk storage
    - Most commonly used in parallel database systems

Parallel Algorithms for Query Processing (cont'd.)

- Linear speed-up
  - Linear reduction in time taken for operations
- Linear scale-up
  - Constant sustained performance by increasing the number of processors and disks

Parallel Algorithms for Query Processing (cont’d.)

- Operator-level parallelism
  - Horizontal partitioning
    - Round-robin partitioning
    - Range partitioning
    - Hash partitioning
- Sorting
  - If data has been range-partitioned on an attribute:
    - Each partition can be sorted separately in parallel
    - Results concatenated
  - Reduces sorting time

Parallel Algorithms for Query Processing (cont'd.)

- Selection
  - If condition is an equality condition on an attribute used for range partitioning:
    - Perform selection only on partition to which the value belongs
- Projection without duplicate elimination
  - Perform operation in parallel as data is read
- Duplicate elimination
  - Sort tuples and discard duplicates

Parallel Algorithms for Query Processing (cont'd.)

- Parallel joins divide the join into *n* smaller joins
  - Perform smaller joins in parallel on *n* processors
  - Take a union of the result
- Parallel join techniques
  - Equality-based partitioned join
  - Inequality join with partitioning and replication
  - Parallel partitioned hash join

Parallel Algorithms for Query Processing (cont'd.)

- Aggregation
  - Achieved by partitioning on the grouping attribute and then computing the aggregate function locally at each processor
- Set operations
  - If argument relations are partitioned using the same hash function, they can be done in parallel on each processor

Parallel Algorithms for Query Processing (cont'd.)

- Intraquery parallelism
  - Approaches
    - Use parallel algorithm for each operation, with appropriate partitioning of the data input to that operation
    - Execute independent operations in parallel
- Interquery parallelism
  - Execution of multiple queries in parallel
  - Goal: scale up
  - Difficult to achieve on shared-disk or shared-nothing architectures

12.9 Summary

- SQL queries translated into relational algebra
- External sorting
- Selection algorithms
- Join operations
- Combining operations to create pipelined execution
- Parallel database system architectures

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** (**primary index, equality on key**). Retrieve a single record that satisfies the corresponding equality condition
  - $Cost = (h_i + 1) * (t_T + t_S)$
- **A3** (**primary index, equality on nonkey**) Retrieve multiple records.
  - Records will be on consecutive blocks
    - Let b = number of blocks containing matching records
  - $Cost = h_i * (t_T + t_S) + t_S + t_T * b$

t_T – time to transfer one block
t_S – time for one seek

Selections Using Indices

- A4 (secondary index, equality on nonkey).
  - 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!

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.

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)…..*

Slide Credits: Elmasri and Navathe

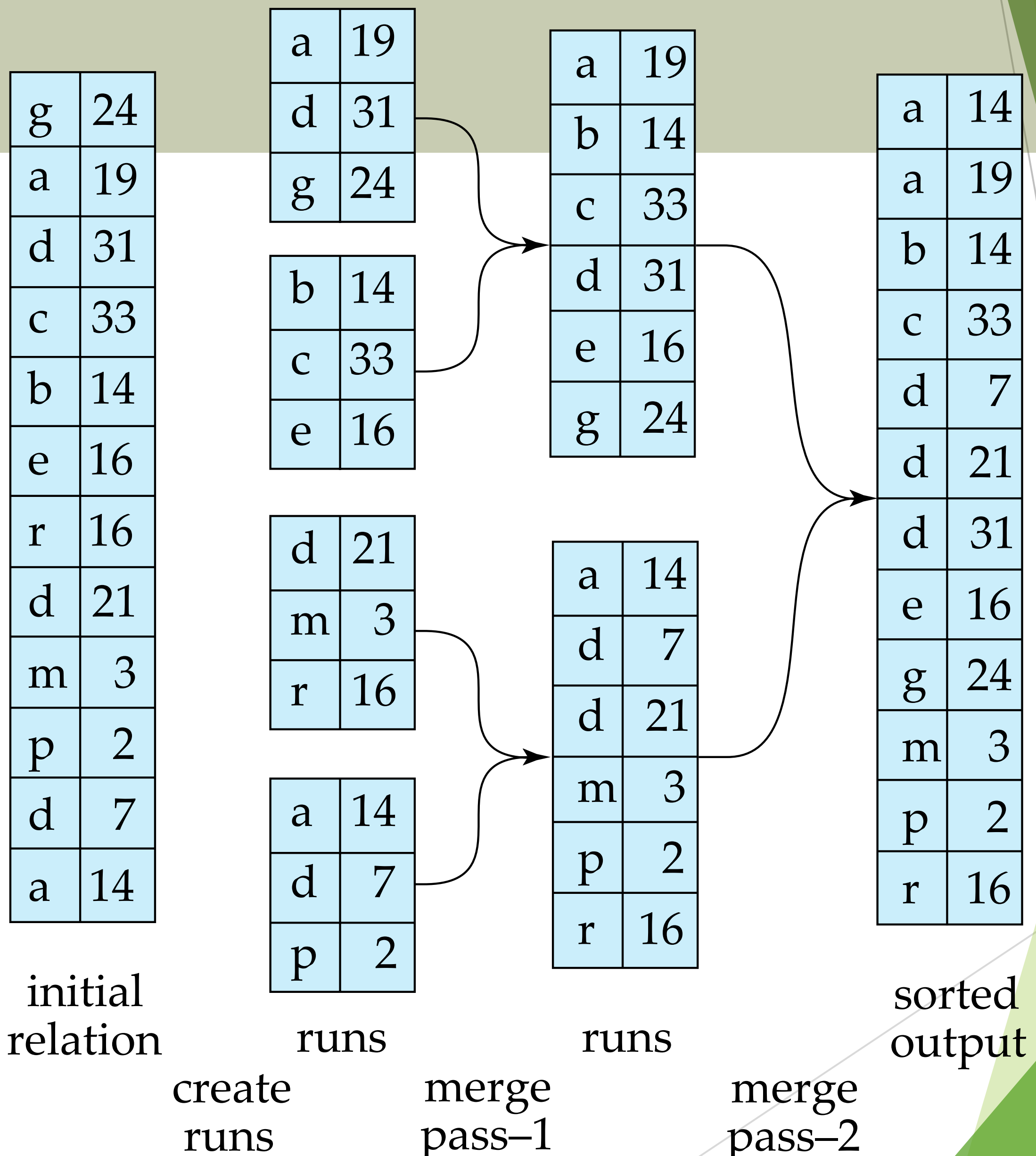
External Sort-Merge (Cont.)

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

Example: External Sorting Using Sort-Merge



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 $t_r \bullet t_s$ 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.

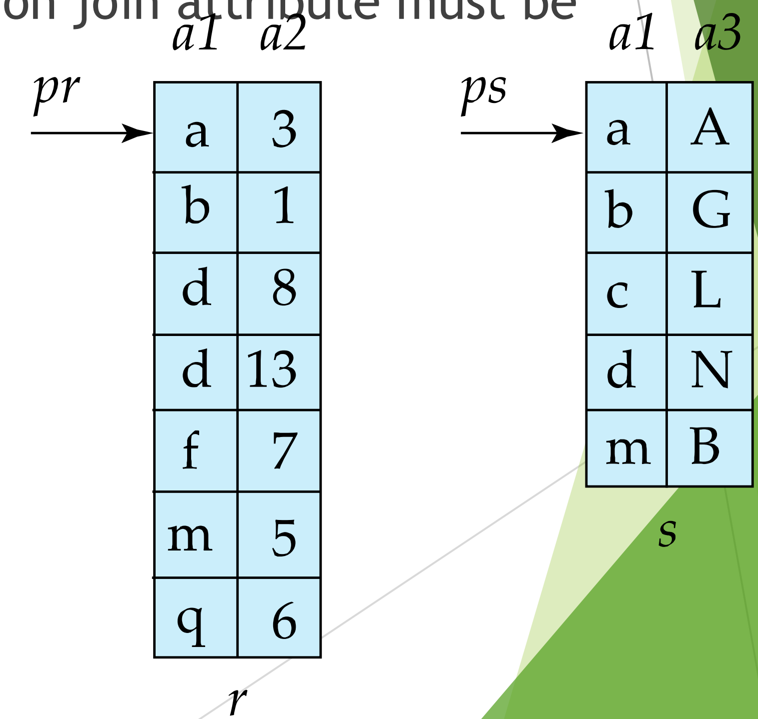
for each block B_r **of** r **do begin**
    for each block B_s **of** s **do begin**
        for each tuple t_r **in** B_r **do begin**
            for each tuple t_s **in** B_s **do begin**
                Check if (t_r, t_s) satisfy the join condition
                if they do, add $t_r \cdot t_s$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

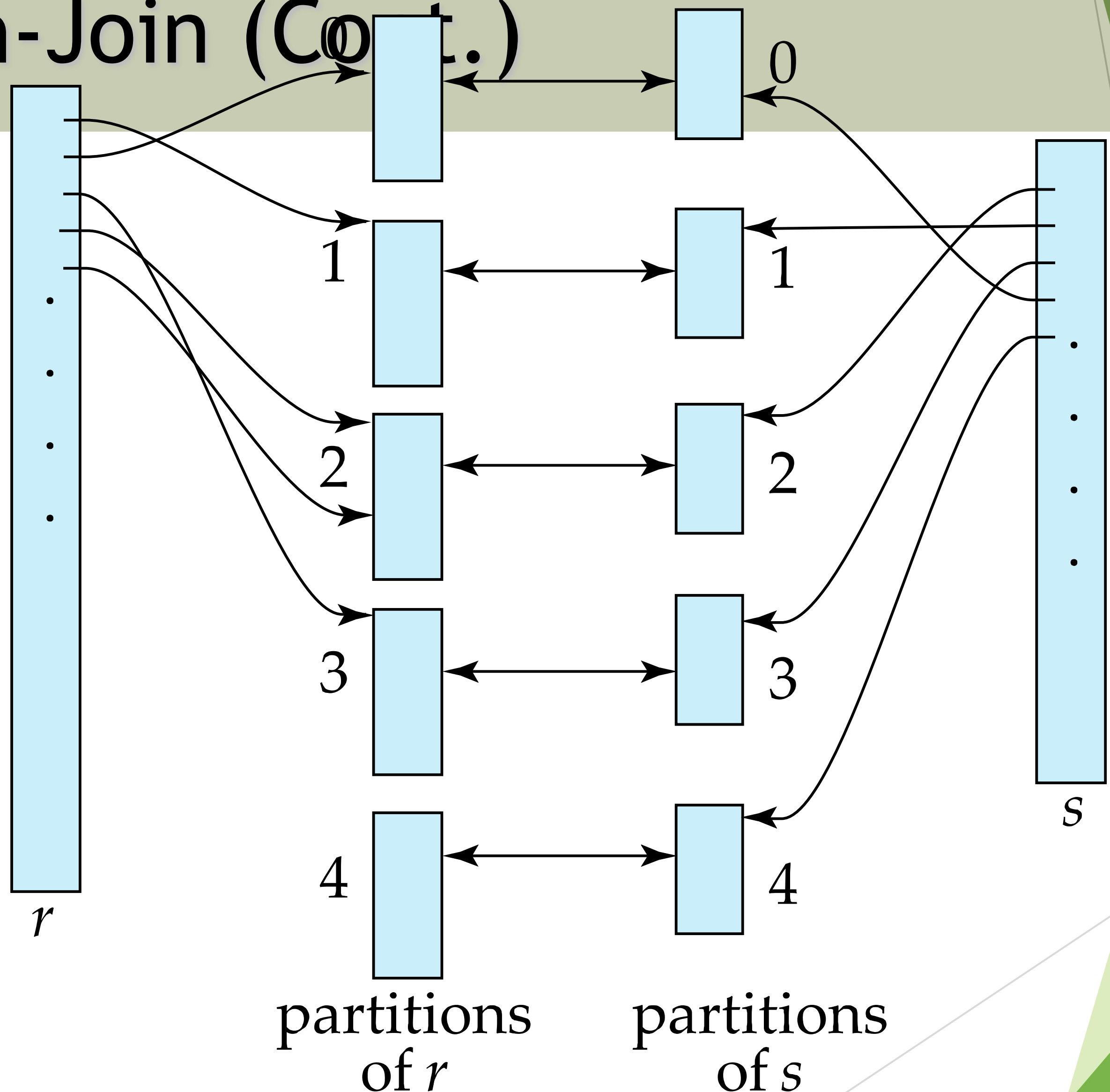


Hash-Join

- 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, ..., *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 tuple $t_r \in r$ is put in partition r_i where $i = h(t_r [JoinAttrs])$.
  - $r_0,, r_1. \ldots, r_n$ denotes partitions of *s* tuples
    - Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s [JoinAttrs])$.

Slide Credits: Elmasri and Navathe

Hash-Join (Cont.)



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.

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**.

Slide Credits: Elmasri and Navathe

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

Hybrid Hash-Join

- Useful when memory sized are relatively large, and the build input is bigger than memory.
- **Main feature of hybrid hash join:**

  Keep the first partition of the build relation in memory.

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:* 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 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.

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 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:

      $b_r (2 \lceil \log_{\lfloor M/bb \rfloor - 1} (b_r / M) \rceil + 1) \lceil$
  - Seeks: next slide

Slide Credits: Elmasri and Navathe

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

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

- 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
- 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 (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 = $\lceil b_r \ / \ (M\text{-}2) \rceil * b_s + b_r$ block transfers + $2 \lceil b_r \ / \ (M\text{-}2) \rceil$ seeks
  - 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
    - 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 or 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.

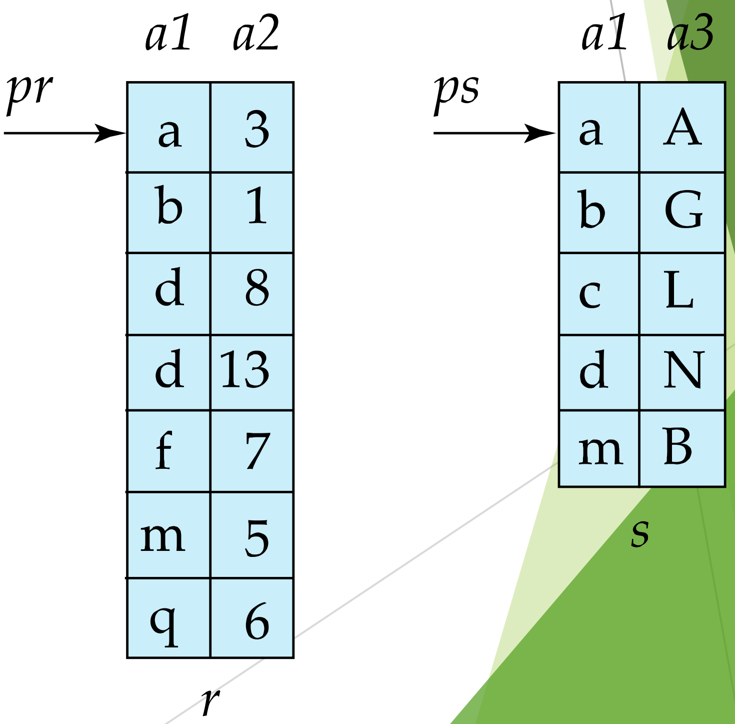
Slide Credits: Elmasri and Navathe

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
- *student* has 5000 tuples
- 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



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

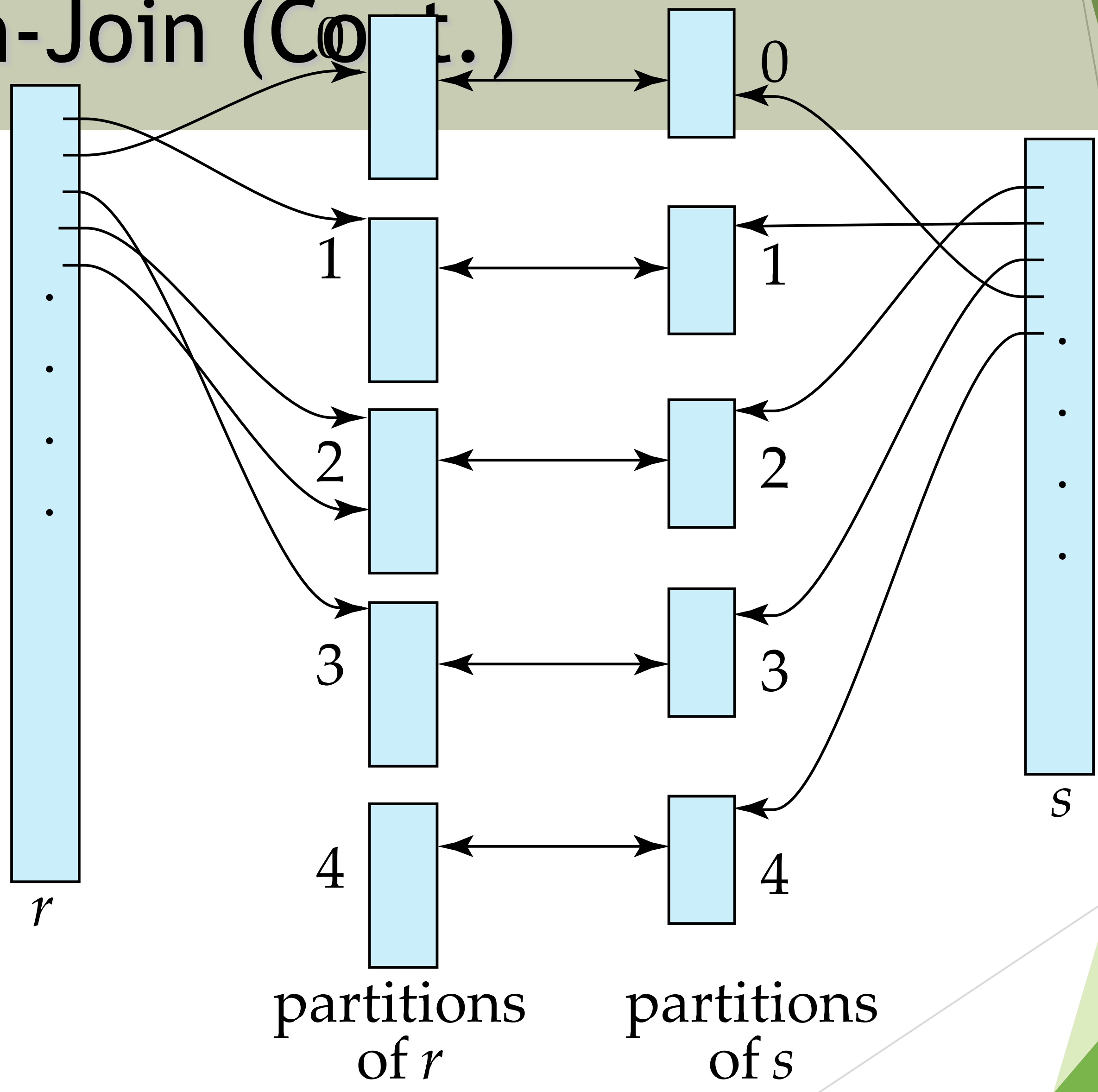
  - + the cost of sorting if relations are unsorted.
- **hybrid merge-join**: If one relation is sorted, and the other has a secondary B$^+$-tree index on the join attribute
  - Merge the sorted relation with the leaf entries of the B$^+$-tree .
  - Sort the result on the addresses of the unsorted relation's tuples
  - Scan the unsorted relation in physical address order and merge with previous result, to replace addresses by the actual tuples
    - Sequential scan 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
- 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, . . ., r_n denote partitions of r tuples
    - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r [JoinAttrs])$.
  - r_0,, r_1. . ., r_n denotes partitions of s tuples
    - Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s [JoinAttrs])$.

- *Note*: In book, 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 (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.

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 s_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 < 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

  $$3(b_r + b_s) + 4 * n_h \text{ block transfers } +$$
  $$2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) \text{ seeks}$$

- 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:

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

- 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 + $2(\lceil 100/3 \rceil + \lceil 400/3 \rceil)$ = 336 seeks

Hybrid Hash-Join

- Useful when memory sized are relatively large, and the build input is bigger than memory.
- **Main feature of hybrid hash join:**

  Keep the first partition of the build relation in memory.
- E.g. With memory size of 25 blocks, *instructor* can be partitioned into five partitions, each of size 20 blocks.
  - Division of memory:
    - The first partition occupies 20 blocks of memory
    - 1 block is used for input, and 1 block each for buffering the other 4 partitions.
- *teaches* is similarly partitioned into five partitions each of size 80
  - the first is used right away for probing, instead of being written out
- Cost of 3(80 + 320) + 20 +80 = 1300 block transfers for hybrid hash join, instead of 1500 with plain hash-join.
- Hybrid hash-join most useful if $M >> \sqrt{b_s}$

Slide Credits: Elmasri and Navathe

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:* 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 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.

Slide Credits: Elmasri and Navathe