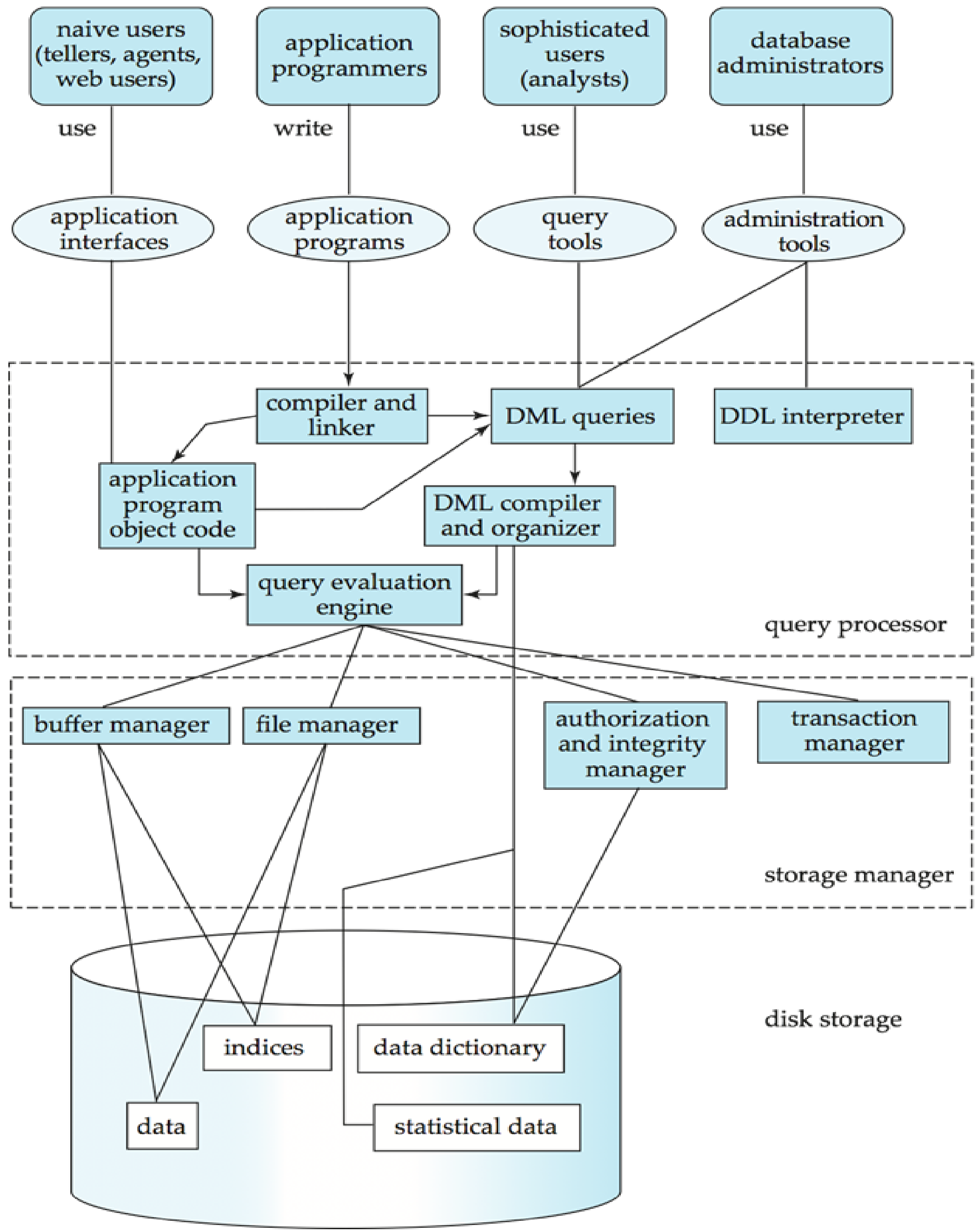
Query Processing

Chapter 12

What we want to cover today

- RDBMS architecture
- Overview of query processing
- Join algorithms

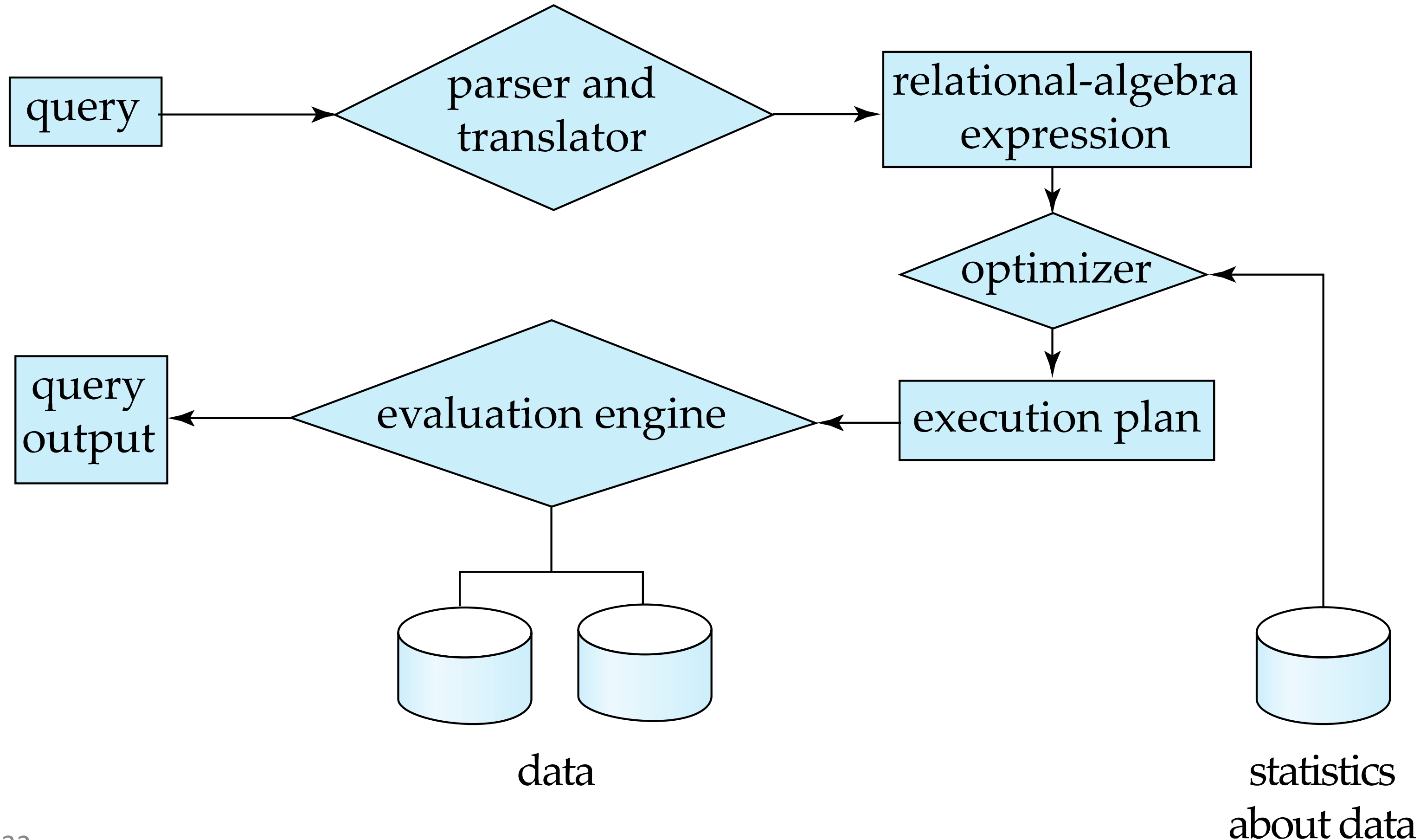
RDBMS ARCHITECTURE


naive users (tellers, agents, web users)
application programmers
sophisticated users (analysts)
database administrators
use
write
use
use
application interfaces
application programs
query tools
administration tools
compiler and linker
DML queries
DDL interpreter
application program object code
DML compiler and organizer
query evaluation engine
query processor
buffer manager
file manager
authorization and integrity manager
transaction manager
storage manager
disk storage
indices
data dictionary
data
statistical data

Chapter 12 – Query Processing

OVERVIEW

Basic Steps in Query Processing



Cost Measures

- Query cost is generally measured as total elapsed time for answering query
  - Many factors contribute to time cost
    - disk accesses, CPU, or even network communication
- Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account
  - Number of seeks * average-seek-cost
  - Number of blocks read * average-block-read-cost
  - Number of blocks written * average-block-write-cost
    - **NOTE:** Cost to write a block is greater than cost to read a block
      - data is read back after being written to ensure that the write was successful

Cost Measures (Cont.)

- 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

    $$b * t_T + S * t_S$$
- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account
- We do not include cost to writing output to disk in our cost formulae

Cost Measures (Cont.)

- 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
    - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available
- Required data may be buffer resident already, avoiding disk I/O
  - But hard to take into account for cost estimation

Evaluation of Expressions

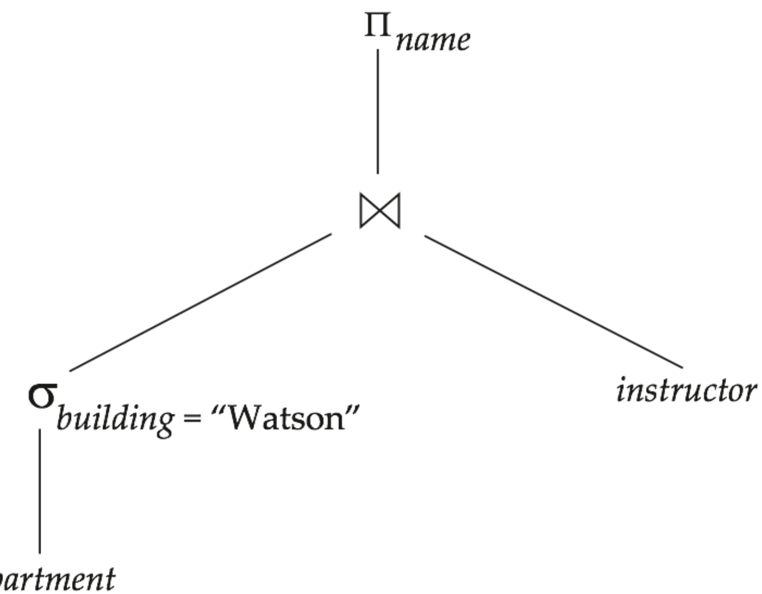
- **Materialization**: generate results of an expression whose inputs are relations or are already computed, **materialize** (store) it on disk.
- **Pipelining**: pass on tuples to parent operations even as an operation is being executed

Materialization

- **Materialized evaluation**: evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.
- E.g., in figure below, compute and store

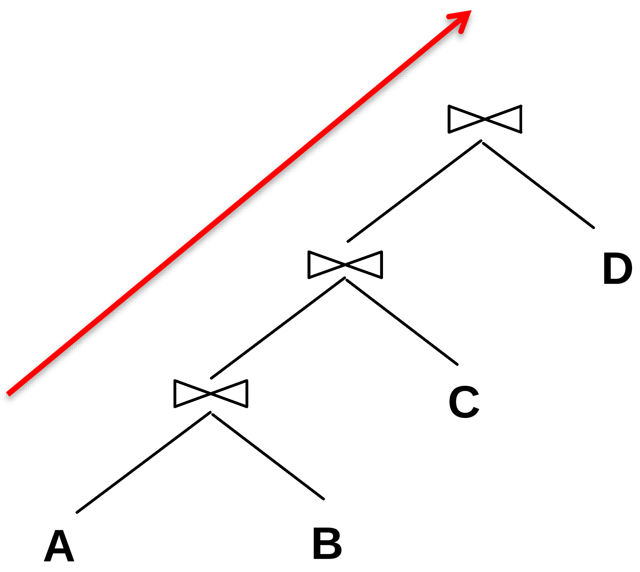
$$\sigma_{building="\text{Watson}"}(department)$$

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



Pipelining

- Result of one operator **pipelined** to another without creating temporary table
- Pipelines can be executed in two ways: **demand driven** and **producer driven**



Pipelined Evaluation

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

Other Common Techniques

- Indexing: Can use WHERE conditions to retrieve small set of tuples (selections, joins)
- Iteration: Sometimes, faster to scan all tuples even if there is an index. (And sometimes, we can scan the data entries in an index instead of the table itself.)
- Partitioning: By using sorting or hashing, we can partition the input tuples and replace an expensive operation by similar operations on smaller inputs.

Iterator Interface

- Relational operators at nodes in plan tree support a uniform iterator interface
  - Open: initializes state by allocating input and output buffers, passes arguments to operator.
  - Get_next: calls operator specific code to process input tuples and generate output tuples.
  - Close: deallocates state info when all output produced.
- Hides whether operator pipelines or materializes input tuples
- Also used to encapsulate access methods like B+tree and hash indexes.

Statistics and Catalogs

- Need information about the relations and indexes involved. ***Catalogs*** typically contain at least:
  - # tuples (NTuples) and # pages (NPages) for each relation.
  - # distinct key values (NKeys) and NPages for each index.
  - Index height, low/high key values (Low/High) for each tree index.
- Catalogs updated periodically.
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.
- More detailed information (e.g., histograms of the values in some field) are sometimes stored.

Chapter 12 – Query Processing

JOIN ALGORITHMS

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.

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

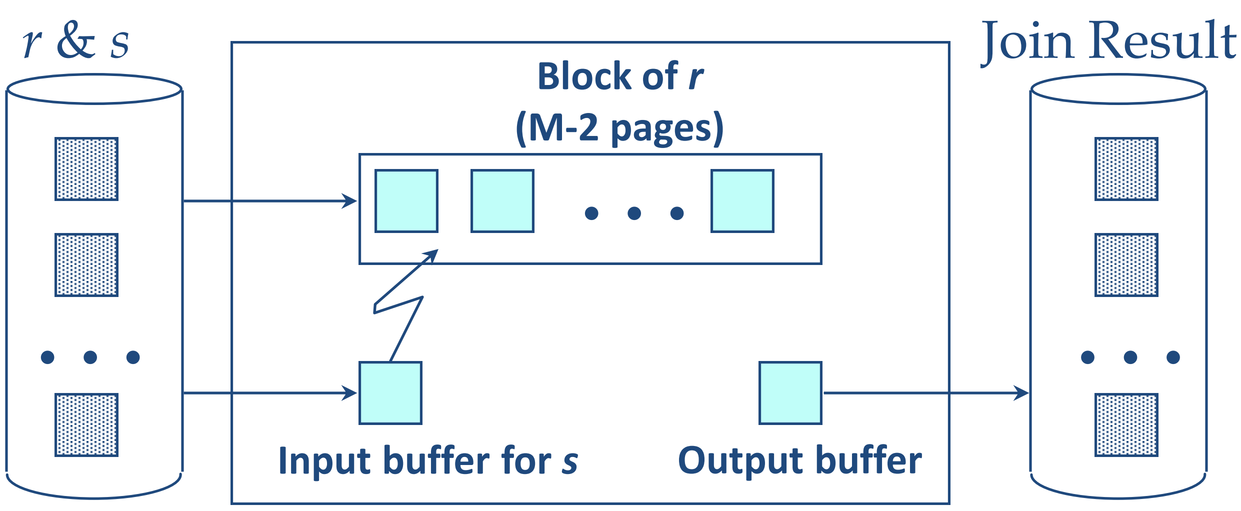
  $n_r * b_s + b_r$ block transfers, plus

  $n_r + b_r$ seeks

  where n_r is number of records in *r*, b_r and b_s are number of blocks in *r* and *s,* respectively
- 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

Block Nested Loops

- If *M* pages of memory available
  - Use *M* — 2 pages as blocking unit for outer relation; use remaining two pagers 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



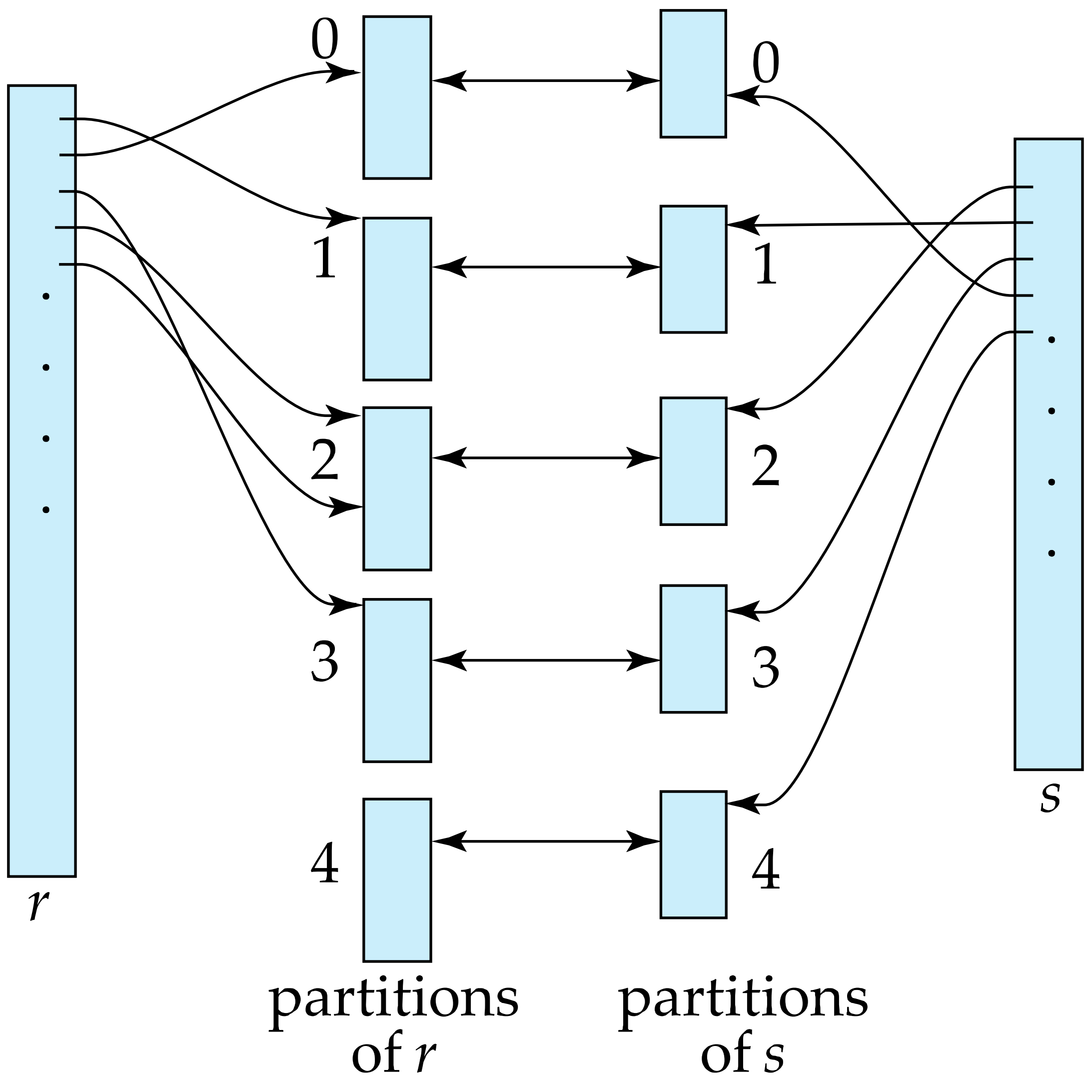
BNL (Cont.)

- Worst case (*M = 3* pages) 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
  - With *student* as outer relation cost
    $(100 * 400 + 100)$ = 40,100 transfers and 200 seeks
- If we have *M = 12* pages of memory available
  - With *student* as outer relation
    ((100 / 10) * 400) + 100 = 4100 transfers and 2 * (100 / 10) = 20 seeks
- Best case ($M = b_r$ pages): $b_r + b_s$ block transfers + 2 seeks.

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 $i = h(t_r[JoinAttrs])$.
  - $s_0, s_1 \ldots, s_n$ denotes partitions of *s* tuples
    - Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s[JoinAttrs])$.

Hash-Join (Cont.)



Hash-Join Algorithm

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

1. Partition the relation *r* using hashing function *h*. When partitioning a relation, one block of memory is reserved as the output buffer for each partition.
2. Partition *s* similarly.
3. For each *i:*
   (a) Load r_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 s_i from the disk one by one. For each tuple t_s locate each matching tuple t_r in r_i using the in-memory hash index. Output the concatenation of their attributes.

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

Hash-Join Algorithm (Cont.)

- The value *n* and the hash function *h* are chosen such that each r_i should fit in memory.
  - Typically n is chosen as $\lceil b_r/M \rceil * f$ 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 *r*
  - Further partition the $M - 1$ partitions using a different hash function
  - Use same partitioning method on *s*
  - Rarely required: e.g., with block size of 4 KB, recursive partitioning not needed for relations of < 1GB with memory size of 2MB, or relations of < 36 GB with memory of 12 MB

Hash Join - Overflows

- Partitioning is said to be **skewed** if some partitions have significantly more tuples than some others
- **Hash-table overflow** occurs in partition r_i if r_i does not fit in memory. Reasons could be
  - Many tuples in *r* with same value for join attributes
  - Bad hash function
- **Overflow resolution** can be done in build phase
  - Partition r_i is further partitioned using different hash function.
  - Partition s_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