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 $\{0, 1, \ldots, n\}$, 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 $i = h(t_s[JoinAttrs])$.
- Note: In book, Figure 12.10 $r_i$ is denoted as $H_{r_i}$, $s_i$ is denoted as $H_{s_i}$ and $n$ is denoted as $n_h$. 
Hash-Join (Cont.)

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 $\lceil \frac{b_s}{M} \rceil \times f$ 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
  \[ 3(b_r + b_s) + 4 \cdot n_b \text{ block transfers} + \]
  \[ 2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) \text{ seeks} \]

- If recursive partitioning required:
  - number of passes required for partitioning build relation \( s \) to less than \( M \) blocks per partition is \( \lceil \log_{M/b_b} \rceil \)
  - best to choose the smaller relation as the build relation.
  - Total cost estimate is:
    \[ 2(b_r + b_s)\lceil \log_{M/b_b} \rceil + b_r + b_s \text{ block transfers} + \]
    \[ 2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) \lceil \log_{M/b_b} \rceil \text{ seeks} \]

- If the entire build input can be kept in main memory no partitioning is required
  - Cost estimate goes down to \( b_r + b_s \).

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 \( M >> \sqrt{b_s} \)
### Complex Joins

- **Join with a conjunctive condition:**
  \[ r \bowtie_{\theta_1 \land \theta_2 \land \ldots \land \theta_n} s \]
  - 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
    \[ \theta_1 \land \ldots \land \theta_{i-1} \land \theta_{i+1} \land \ldots \land \theta_n \]

- **Join with a disjunctive condition**
  \[ r \bowtie_{\theta_1 \lor \theta_2 \lor \ldots \lor \theta_n} s \]
  - 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 \(\setminus\)): 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: \( d_1, d_4, d_{11}, d_{45}, d_{77}, d_{123} \)
      distributed: \( d_4, d_8, d_{11}, d_{56}, d_{77}, d_{121}, d_{333} \)
    - 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
Other Operations: Outer Join

- **Outer join** can be computed either as
  - A join followed by addition of null-padded non-participating tuples.
  - by modifying the join algorithms.
- Modifying merge join to compute $r \bowtie s$
  - In $r \bowtie s$, non-participating tuples are those in $r - \Pi_F(r \bowtie s)$
  - Modify merge-join to compute $r \bowtie s$:
    - During merging, for every tuple $t_r$ from $r$ that do not match any tuple in $s$, output $t_r$ padded with nulls.
    - Right outer-join and full outer-join can be computed similarly.

Cache Conscious Algorithms

- Goal: minimize cache misses, make best use of data fetched into the cache as part of a cache line
- For sorting:
  - Use runs that are as large as L3 cache (a few megabytes) to avoid cache misses during sorting of a run
  - Then merge runs as usual in merge-sort
- For hash-join
  - First create partitions such that build+probe partitions fit in memory
  - Then subpartition further s.t. build subpartition+index fits in L3 cache
    - Speeds up probe phase significantly by avoiding cache misses
- Lay out attributes of tuples to maximize cache usage
  - Attributes that are often accessed together should be stored adjacent to each other
- Use multiple threads for parallel query processing
  - Cache misses leads to stall of one thread, but others can proceed