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

Diagram:
- **Input:** query
- **Output:** query output
- **Processes:**
  - Parser and translator
  - Relational algebra expression
  - Optimizer
  - Evaluation engine
  - Execution plan
- **Data Sources:**
  - Data
  - Statistics about data
Parsing, Translation and Evaluation

- Parsing and translation:
  - Checks syntax, verifies relations, attributes, etc.
  - Translates the query into its internal, relational algebraic form.

- Optimization:
  - Evaluates and compares different ways to implement the query.
  - Constructs a query-evaluation plan.

- Evaluation:
  - Executes the (chosen) query-evaluation plan.
  - Returns results.

- Our focus will be on optimization…
Observation #1 - A given relational algebra expression has many equivalent expressions:

\[ \sigma_{\text{balance}<2500}(\Pi_{\text{balance}}(\text{account})) \]

\[ \Pi_{\text{balance}}(\sigma_{\text{balance}<2500}(\text{account})) \]

Observation #2 - Each relational algebraic operation can be evaluated using different algorithms:

- Use an index on \textit{balance} to find accounts with balance < 2500, or
- Perform a relation scan and discard accounts with balance \( \geq 2500 \)
Evaluation plan – Annotated relational algebraic expression specifying an evaluation strategy.
  ➢ a.k.a. query execution plan or query plan

Query optimization – The process of choosing an evaluation plan that has lowest estimated “cost.”

A cost estimate for a plan is calculated using statistical information:
  ➢ Number of tuples in each relation
  ➢ Size of tuples
  ➢ Distribution of attribute values, etc.
Optimization (Cont.)

- Query Processing Slides (low-level):
  - How to measure query cost.
  - Algorithms for individual relational algebra operations.
  - How to combine algorithms for individual operations in order to evaluate a complete expression.

- Query Optimization Slides (high-level):
  - How to optimize queries i.e., how to find an evaluation plan with lowest estimated cost.
Measures of Query Cost

- Generally, cost can be defined as the total time for query execution.

- Many factors contribute to cost:
  - disk accesses
  - CPU time
  - network communication
  - sequential vs. random I/O
  - buffer size
  - writing vs. reading

- In database systems the main cost is typically disk access time.
  - relatively easy to estimate
For simplicity our cost measure will be a function of:
  - the number of block transfers to/from disk
  - block access time (seek time + rotational latency) $t_S$
  - block transfer time $t_T$
  - buffer size

Real DBMSs take other factors into account.
The amount of available buffer space depends on several factors:
  - other concurrent processes, memory size, etc.

Available buffer space is therefore hard to determine in the abstract.

The authors use worst case estimates, assuming only the minimum amount of memory is available for query execution (extremely pessimistic).

Other times they use average case estimates, or a somewhat peculiar mix of average and worst case.

The cost to write the final output to disk is **not** included in cost estimates.
Statistical Information for Cost Estimation

- $n_r$: number of tuples in a relation $r$.
- $b_r$: number of blocks containing tuples of $r$.
- $s_r$: size of a tuple of $r$.
- $f_r$: blocking factor of $r$, i.e., the number of tuples that fit into one block.

If tuples of $r$ are stored together physically in a file, then:

$$b_r = \left\lfloor \frac{n_r}{f_r} \right\rfloor$$
More accurately:

\[ b_r \geq \left\lfloor \frac{n_r}{f_r} \right\rfloor \]

- \( t_S \): average block access time (seek plus rotational latency)
- \( t_T \): average transfer time
Search algorithms that perform selections without using an index are referred to as *file scans*.

Algorithm A1 - linear search

- Scan each block and test all records to see whether they satisfy the condition.

Cost estimate:

- Best, worst, and average case $t_S + b_r * t_T$
- If the selection is an *equality comparison* on a *cand. key*, average case drops to $t_S + (b_r/2) * t_T$
  - What are best and worst case scenarios?
- Linear search can be applied regardless of selection condition, ordering of records in the file, or availability of indices

Note that linear search is sometimes referred to as a *table scan* or a *file scan*, although the latter term includes other algorithms.
Note that the books numbering of the algorithms depends on the edition!

- In particular, binary search is not included in the most recent edition.

A2 - binary search

- Applicable primarily if selection is an equality comparison, and
- Tuples are sorted based on the search key.

Cost:

- Assume that the blocks of a relation are stored contiguously
- Worst case is \( \lceil \log_2(b_r) \rceil \times (t_S + t_T) \)
- This is the cost of locating a tuple using a binary search
- Plus the number of blocks containing records that satisfy selection condition.
  - Will see how to estimate this cost in the next chapter
Search algorithms that use an index to perform a selection are referred to as **index scans**.

To use an index, the selection predicate must involve (at least part of) the index search-key.

**A3** - primary index on candidate key, equality
- At most one record is retrieved.
- \( \text{Cost} = (HT_i + 1) \times (t_s + t_T) \), where \( HT_i \) represents the “height” of index \( i \)

**A4** - primary index on non-key, equality
- Zero or more records are retrieved.
- Retrieved records will be on consecutive blocks.
- \( \text{Cost} = HT_i \times (t_s + t_T) + t_s + b \times t_T \), where \( b \) is the number of blocks containing retrieved records (to be estimated in the next chapter).
A5 - equality on search-key of secondary index

- Retrieve a single record if the search-key is a candidate key.
- Retrieve multiple records if search-key is not a candidate key.

Cost:
- If the search-key is a candidate key (single record retrieval):
  - Cost = \((HT_i + 1) \times (t_S + t_T)\),
- If the search-key is not a candidate key (multiple record retrieval):
  - Cost = \((HT_i + n) \times (t_S + t_T)\), where \(n = \#\) of records retrieved
    - Worst case - assumes each record is on a different block
    - In the next chapter we will develop and estimate for \(n\).
    - The book does not include block reads for the bucket.
Selections Involving Comparisons

What about selections that aren’t simple tests for equality?

- The techniques vary, depending on the complexity of the predicate.
- Different vendors will use different “tricks.”
- Note that linear search is always a viable option.

For example: $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$

- File Scan - linear or binary search (if sorted).
- Index Scan - using indices as specified in the following.

A6 - primary index, comparison using $\geq$ or $\leq$

- $\sigma_{A \geq V}(r)$ use index to find first tuple $\geq v$ and scan sequentially from there.
- $\sigma_{A \leq V}(r)$ don’t use the index; scan relation sequentially until first tuple $> v$.
- Exercise – give a cost estimate for both of these options.
Selections Involving Comparisons

A7 - secondary index, comparison

- $\sigma_{A \geq V}(r)$ use the index to find the first index entry $\geq v$; scan the leaves of the index sequentially from there to find record pointers.
- $\sigma_{A < V}(r)$ scan the leaf pages of the index using record pointers, until reaching first entry $> v$.
- Note that both cases requires an I/O for each record, in the worst case.
- Exercise – give a cost estimate for both cases.
Implementation of Complex Selections

- **Conjunction:** $\sigma_{\theta_1 \land \theta_2 \land \cdots \land \theta_n}(r)$

- **A8** - conjunctive selection using one index
  - Select one of the conditions $\theta_i$ and algorithms A1 through A7 that results in the least cost for $\sigma_{\theta_i}(r)$.
  - Test other conditions on tuples after retrieving them into the buffer.

- **A9** - conjunctive selection using multiple-key index
  - Use appropriate composite (multiple-key) index if available.

- **A10** - conjunctive selection by intersection of identifiers
  - Use the index corresponding to each condition (if they exist) to obtain sets of record pointers.
  - Take the intersection of the resulting sets.
  - Retrieve records from data file.
  - Pointers could be sorted prior to retrieval (why would you do this??).
  - If some conditions do not have appropriate indices, apply test in memory.
Algorithms for Complex Selections

- **Disjunction:** $\sigma_{\theta_1 \lor \theta_2 \lor \ldots \lor \theta_n}(r)$.

- **A11 - disjunctive selection by union of identifiers**
  - Use the index corresponding to each condition to obtain sets of record pointers.
  - Take the union of all the obtained sets.
  - Retrieve records from data file.
  - Could sort pointers prior to retrieval.
  - Applicable only if all conditions have available indices; otherwise use linear scan.

- **Negation:** $\sigma_{\neg \theta}(r)$
  - Use linear search.
  - If an index is applicable to $\theta$, find satisfying records using leaf-level of index and fetch from the data file.
Option #1 - Use an existing applicable ordered index (e.g., B+ tree) to retrieve the tuples in sorted order.

Option #2 - Build an index on the relation, and then use the index to retrieve the tuples in sorted order.

Option #3 - For relations that fit in memory, techniques like quick-sort can be used.

Option #4 - For relations that don’t fit in memory, use external sort-merge.

Note that vendors typically have proprietary sorting algorithms.
Let $M$ denote memory size (in blocks), and let $r$ be the relation to be sorted.

1. **Create sorted runs:**

   $i = 0$;
   while (the end of $r$ has not been reached) {
     Read $M$ blocks of $r$ into memory
     Sort the in-memory blocks
     Write sorted data to disk; label it run $R_i$;
     $i = i + 1$;
   }
   $N = i$;

   The end result is $N$ runs, numbered 0 through $N-1$, where each run, except perhaps the last, contains $M$ sorted blocks.
2. **Merge the runs (N-way merge):**

   // We assume (for now) that N < M.
   // Use N blocks, $b_i$, 0<=i<=N-1, of memory to buffer input runs.
   // Use 1 block, $b_M$, of memory to buffer output.

   Read the first block of each run into its buffer block;

   repeat
     Let $t$ be the the smallest record in sorted order from among all blocks, $b_i$, where 0<=i<=N-1, and suppose that $t$ is in block $b_k$;
     Write $t$ to $b_M$;
     if ($b_M$ is full) then
       write $b_M$ to disk;
     Delete $t$ from $b_k$;
     if ($b_k$ is empty) then
       read the next block (if any) of the run into $b_k$;
   until all sorted runs are empty;
External Sort-Merge (N<M)

Original Table
- on disk

Sorted Runs
- on disk
- each has M blocks

Buffer
- Runs are merged block at a time

Final Sorted Table
- on disk

Sort M blocks in memory

\[ N = \left\lfloor \frac{b \cdot r}{M} \right\rfloor \]

Output block
External Sort-Merge ($N \geq M$)

- If $N \geq M$, several merge passes are required.
  - In each pass, contiguous groups of $M - 1$ runs are merged.
  - Repeated passes are performed till all runs have been merged into one.

- Note:
  - 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.
Example: External Sorting
Using Sort-Merge

M=3, and each record takes up 1 buffer block* (very artificial example)
Cost analysis:

- If $N < M$ then total cost is $3b_r$ (excluding final output)
- If $N \geq M$ then the number of merge passes required is $\lceil \log_{M-1}(b_r/M) \rceil$
- Block transfers for initial run creation as well as in each pass is $2b_r$
- For final pass, we ignore final write cost for all operations.

Thus total number of block accesses for external sorting:

$$b_r \left( 2 \lceil \log_{M-1}(b_r/M) \rceil + 1 \right)$$
Join Operation

- Algorithms for implementing joins:
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
  - Various versions of the above

- Choice of algorithm is based on cost estimate.

- Examples use the following information:
  - customer - 10,000 rows, 400 blocks
  - depositor - 5000 rows, 100 blocks
Nested-Loop Join

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

  // r is the outer relation, s is the inner relation
  for (each tuple $t_r$ in $r$)
    for (each tuple $t_s$ in $s$)
      if ($t_r$ and $t_s$ satisfy condition $\theta$)
        add $t_r \cdot t_s$ to the result;

- Does not use or require indices.

- Can be used with any kind of join condition.
Nested-Loop Join (Cont.)

- **Worst case** – if there are only 3 blocks available, 2 for input, 1 for output – the number of disk accesses is:
  - \( n_r \cdot b_s + b_r \)

- **Examples:**
  - \( 5000 \cdot 400 + 100 = 2,000,100 \) with *depositor* as outer relation.
  - \( 10000 \cdot 100 + 400 = 1,000,400 \) with *customer* as the outer relation.

- **Best case** - if both relations fit entirely in memory.
  - \( b_r + b_s \)
  - If only the smaller of the two relations fits entirely in memory then use that as the inner relation; the bound will still hold.

- **Example:**
  - \( 400 + 100 = 500 \) with *depositor* or *customer* as the outer relation.
Enhanced version of the nested-loop join:

```
for (each block \( B_r \) of \( r \))
  for (each block \( B_s \) of \( s \))
    for (each tuple \( t_r \) in \( B_r \))
      for (each tuple \( t_s \) in \( B_s \))
        if (\( t_r \) and \( t_s \) satisfy the join condition)
          add \( t_r \cdot t_s \) to the result;
```

- Does not use or require indices.
- Can be used with any kind of join condition.
Block Nested-Loop Join (Cont.)

- Worst case – if there is only 3 blocks available, 2 for input, and 1 for output – the number of block accesses is: \( b_r \ast b_s + b_r \)

- Examples:
  - \( 100 \ast 400 + 100 = 40,100 \) (depositor as the outer relation)
  - \( 400 \ast 100 + 400 = 40,400 \) (customer as the outer relation)

- Best case: \( b_r + b_s \)

- Example:
  - \( 400 + 100 = 500 \) (same as with nested-loop join)

- Would a nested-loop join ever be preferable?
Cute little optimizations that only a programmer could love…

For a nested-loop join: Scan the inner relation forward and backward alternately, to make use of the blocks remaining in buffer (with LRU replacement).

For a block nested-loop join:
- Use $M - 2$ disk blocks as the blocking unit for the outer relation, where $M =$ memory size in blocks
- Use one buffer block to buffer the inner relation
- Use one buffer block to buffer the output
  - Worst case: $\left\lceil \frac{br}{(M-2)} \right\rceil \ast bs \ast br$
  - Best case is still the same: $bs + br$

For either: if the join is an equijoin, and the join attribute is a candidate key on the inner relation, then stop inner loop on first match.
Index scans are an option if:

- the join is an equijoin or natural join, and
- an index is available on the inner relation’s join attribute

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 \( t_r \).

Example:

- Suppose \( \text{customer} \) has an index on \( \text{customer-name} \).
- \( \text{depositor} \bowtie \text{customer} \)
Worst case - the buffer has space for only one block of \( r \) and one block of the index for \( s \).

- \( b_r + n_r \times c \), where \( c \) is the cost to search the index and retrieve all matching tuples for each tuple or \( r \)

Worst case will assume index blocks are not in buffer, best case will assume that some are (perhaps close to \( br+bs \) if index is pinned in buffer?)
Example of Index
Nested-Loop Join Costs

- Compute `depositor \times customer`, with `depositor` as the outer relation.

- Suppose `customer` has a primary B^+\-tree index on `customer-name`, which contains 20 entries in each index node.

- Since `customer` has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data.

- Recall that `depositor` has 5000 tuples and 100 blocks.

- Cost of indexed nested loops join:
  - 100 + 5000 * 5 = 25,100 disk accesses.

Worst case
Other Options:

- If a supporting index does not exist than it can be constructed “on-the-fly.”
- If indices are available on the join attributes of both \( r \) and \( s \), then use the relation with fewer tuples as the outer relation.
- Or perhaps use the relation which has a primary index on it as the inner relation…
Applicable for equijoins and natural joins.

1. Sort both relations on their join attribute (if not already sorted).
2. Merge the sorted relations to join them:
   - Similar to a “classic” merge.
   - Main difference — every pair of tuples with same value on join attribute must be matched.
   - Detailed algorithm is in the book.

<table>
<thead>
<tr>
<th>ps</th>
<th>pr</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a1)</td>
<td>(a2)</td>
</tr>
<tr>
<td>a</td>
<td>4</td>
</tr>
<tr>
<td>b</td>
<td>4</td>
</tr>
<tr>
<td>a</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>7</td>
</tr>
<tr>
<td>b</td>
<td>7</td>
</tr>
<tr>
<td>a</td>
<td>7</td>
</tr>
<tr>
<td>d</td>
<td>9</td>
</tr>
<tr>
<td>b</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(a2)</th>
<th>(a4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>f</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>e</td>
</tr>
<tr>
<td>5</td>
<td>e</td>
</tr>
<tr>
<td>7</td>
<td>d</td>
</tr>
<tr>
<td>7</td>
<td>f</td>
</tr>
<tr>
<td>8</td>
<td>a</td>
</tr>
<tr>
<td>13</td>
<td>c</td>
</tr>
</tbody>
</table>
Can be used only for equi-joins and natural joins

Best case:
- Suppose all tuples for any given value of the join attribute fit in memory.
- Each block needs to be read only once.
- Thus, the number of block accesses for merge-join is:
  \[ b_r + b_s + \text{the cost of sorting if relations are unsorted}. \]

Worst Case:
- Suppose every tuple from both relations has the same value on the join attribute.
- Thus, all tuples for a given value of the join attribute don’t fit in memory.
- The algorithm is ambiguous in how it deals with this case.
- Block-nested loop is probably the best alternative at this point, in which case the number of block accesses is \( b_r + b_r * b_s \)
Hybrid Merge-Join

- Left as an exercise…

- Applicable if:
  - Join is an equi-join or a natural join
  - One relation is sorted
  - The other has a secondary B+-tree index on the join attribute

- Algorithm Outline:
  - 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.
    - Not really a “scan” of the whole relation.
Applicable for equijoins and natural joins.

For the moment, ignore buffering issues…

Let $h$ be a hash function mapping $JoinAttrs$ to $\{0, 1, ..., n-1\}$.

$h$ is used to partition tuples of both relations:

- The tuples from $r$ are partitioned into $r_0, r_1, \ldots, r_{n-1}$
  - Each tuple $t_r \in r$ is put in partition $r_i$ where $i = h(t_r[JoinAttrs])$.
- The tuples from $s$ are partitioned into $s_0, s_1, \ldots, s_{n-1}$
  - Each tuple $t_s \in s$ is put in partition $s_i$ where $i = h(t_s[JoinAttrs])$.

*Note that the book uses slightly different notation.*
Tuples in $r_i$ need only to be compared with tuples in $s_i$: 

- An $r$ tuple and an $s$ tuple having the same value on the join attribute will have the same hash value.
- If that value is hashed to some value $i$, the $r$ tuple has to be in $r_i$ and the $s$ tuple in $s_i$. 
The hash join of $r$ and $s$ is computed as follows (Preliminary Version):

1. Partition the relation $s$ into $n \geq 1$ partitions using hashing function $h$.
   
   // Assume that $n \leq M-1$, for the moment.
   // When partitioning a relation, one block of memory is used as output for
   // each partition, and one block is used for input.

2. Partition $r$ similarly.

3. For each $i$ where $0 \leq i \leq n-1$:
   
   (a) Load $s_i$ into memory
   
   // Assume for the moment that it fits, with 2 additional blocks to spare.
   (b) Read the tuples in $r_i$ from the disk one by one (block at a time).
      - For each tuple $t_i$, locate each matching tuple $t_s$ in $s_i$
      - Output the concatenation of their attributes.
Hash Join – Partition Phase

Table $s$ (on disk) → Buffer ($M$ blocks) → Partitions of $s$ (on disk)

- Input blocks one at a time
- Tuples are hashed to buckets
- Each consists of "slightly less" than $M$ blocks

$n = \left\lfloor \frac{b_f}{M} \right\rfloor \times f$

*Kind’a like sorting mail or clothes…
<table>
<thead>
<tr>
<th>Partitions of $r$ (on disk)</th>
<th>Partitions of $s$ (on disk)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_0$</td>
<td>$s_0$</td>
</tr>
<tr>
<td>$r_1$</td>
<td>$s_1$</td>
</tr>
<tr>
<td>$r_2$</td>
<td>$s_2$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$r_i$</td>
<td>$s_i$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$r_{n-1}$</td>
<td>$s_{n-1}$</td>
</tr>
</tbody>
</table>
Hash Join – Build Phase

Partitions of \( r \) (on disk)  Partitions of \( s \) (on disk)  Buffer (\( M \) blocks)

\[
\begin{align*}
\text{r}_0 & \quad \text{s}_0 \\
\text{r}_1 & \quad \text{s}_1 \\
\text{r}_2 & \quad \text{s}_2 \\
\vdots & \quad \vdots \\
\text{r}_i & \quad \text{s}_i \\
\vdots & \quad \vdots \\
\text{r}_{n-1} & \quad \text{s}_{n-1}
\end{align*}
\]

\( s_i \) will fit because it consists of "slightly less" than \( M \) blocks

Input \( s_i \) all at once
Hash Join – Build Phase

Partitions of $r$ (on disk) Partitions of $s$ (on disk) Buffer ($M$ blocks)

- $r_0$
- $r_1$
- $r_2$
- $\vdots$
- $r_i$
- $\vdots$
- $r_{n-1}$

- $s_0$
- $s_1$
- $s_2$
- $\vdots$
- $s_i$
- $\vdots$
- $s_{n-1}$

Input $r_i$ block at a time

$s_i$ will fit because it consists of "slightly less" than $M$ blocks

$s_i$ will fit because it consists of "slightly less" than $M$ blocks
Hash Join – Build Phase

Partitions of \( r \) (on disk)  Partitions of \( s \) (on disk)  Buffer (\( M \) blocks)

\[ r_0, s_0 \]
\[ r_1, s_1 \]
\[ r_2, s_2 \]
\[ \vdots \]
\[ r_i, s_i \]
\[ \vdots \]
\[ r_{n-1}, s_{n-1} \]

\[ s_i \text{ will fit because it consists of "slightly less" than } M \text{ blocks} \]

\[ \text{Input } s_i \text{ all at once} \]

\[ \text{Probe } r_i \text{ with tuples from } s_i \text{ to find matching tuples} \]

\[ \text{Input } r_i \text{ block at a time} \]

\[ M-2 \]
\[ M-1 \]
Hash Join – Build Phase

Partitions of $r$ (on disk)  Partitions of $s$ (on disk)  Buffer ($M$ blocks)

- $r_0$
- $r_1$
- $r_2$
- $\ldots$
- $r_i$
- $r_{n-1}$

- $s_0$
- $s_1$
- $s_2$
- $\ldots$
- $s_i$
- $s_{n-1}$

Input $s_i$ all at once

$s_i$ will fit because it consists of "slightly less" than $M$ blocks

Probe $r_i$ with tuples from $s_i$ to find matching tuples

Input $r_i$ block at a time

$s_i$ will fit because it consists of "slightly less" than $M$ blocks

Probe $r_i$ with tuples from $s_i$ to find matching tuples

$s_i$ will fit because it consists of "slightly less" than $M$ blocks

Probe $r_i$ with tuples from $s_i$ to find matching tuples

$s_i$ will fit because it consists of "slightly less" than $M$ blocks

Probe $r_i$ with tuples from $s_i$ to find matching tuples
Hash Join – Build Phase

Partitions of $r$ (on disk) | Partitions of $s$ (on disk) | Buffer ($M$ blocks) | Final Result (join)
---|---|---|---
$r_0$ | $s_0$ | | |
$r_1$ | $s_1$ | | |
$r_2$ | $s_2$ | | |
| | ... | | |
$r_i$ | $s_i$ | | |
| | ... | | |
$r_{n-1}$ | $s_{n-1}$ | | |

 probes

Input $s_i$ block at a time

Input $r_i$ all at once

Input $r_i$ block at a time

Probe $r_i$ with tuples from $s_i$ to find matching tuples

$s_i$ will fit because it consists of "slightly less" than $M$ blocks
The hash join of r and s is computed as follows:

1. Partition the relation s into $n \geq 1$ partitions using hashing function $h$.
   
   // **Assume** that $n \leq M-1$, for the moment.
   // When partitioning a relation, one block of memory is used as output for
   // each partition, and one block is used for input.

2. Partition r similarly.

3. For each $i$ where $0 \leq i \leq n-1$:
   
   (a) Load $s_i$ into memory
       // Assume for the moment that it fits, with 2+ additional blocks to spare.
   (b) **build an in-memory hash index on the join attribute.**
       // **This hash index uses a different hash function than the earlier one** $h$.
   (c) Read the tuples in $r_i$ from the disk one by one (block at a time).
      - For each tuple $t_i$ 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** relation and $r$ is called the **probe** relation.
The number of partitions $n$ and the hash function $h$ is chosen such that each $s_i$ should fit in memory.

- The larger the value of $n$ the smaller the size of each $s_i$
- The smaller the value of $n$ the larger the size of each $s_i$

Typically $n$ is chosen as $\left\lceil \frac{b_s}{M} \right\rceil * f$ where $f$ is a “fudge factor”, typically around 1.2

**Average** size of a partition $s_i$ will be just less than $M$ blocks using the above formula for $n$.

- Fudge factor is for the in-memory index, plus the input block, and final output block.
- Assumes the hash function uniformly distributes.

Note that the probe relation partitions $r_i$ need not fit in memory.
If the build relation $s$ is very large, then the value of $n$ given by the above formula may be greater than $M-2$.

In such a case, the relation $s$ can be recursively partitioned:

- Instead of partitioning $n$ ways, use $M-1$ partitions for $s$.
- Further partition the $M-1$ partitions using a different hash function.
- The same partitioning method must be used on $r$. 

Hash-Join algorithm (Cont.)

- Recursive partitioning is rarely required.

- Recursive partitioning is required if $M \leq \lceil b_s/M \rceil \ast f$
  
  - In other words, if the number of required partitions is at least as big as the number of memory blocks.

- This simplifies roughly to $M \leq \sqrt{b_s}$

- For example, with 12MB of memory and a 4k block size, recursive partitioning is not necessary for tables up to 36GB in size.
Cost of Hash-Join

- If recursive partitioning is not required: cost of hash join is
  \[ 3(b_r + b_s) + 4 \times n \]

- If recursive partitioning is required, the number of passes required for partitioning \( s \) is:
  \[ \left\lceil \log_{\frac{M}{b_s}}(b_s) - 1 \right\rceil \]

- The number of partitions of \( r \) is the same as for \( s \).

- The number of passes for recursive partitioning of \( r \) is the same as for \( s \).

- Worst case: (ignoring partially filled blocks):
  \[ 2(b_r + b_s) \times \left\lceil \log_{\frac{M}{b_s}}(b_s) - 1 \right\rceil + b_r + b_s \]
Example of Cost of Hash-Join

Example:

\[ \text{customer} \Join \text{depositor} \]

Assume that memory size is 21 blocks

\[ b_{\text{depositor}} = 100 \text{ and } b_{\text{customer}} = 400. \]

\[ \text{depositor} \] is used as build input:
- Partitioned into 5 \( (b_{\text{depositor}}/M) \) partitions, each containing 20 blocks.
- This partitioning can be done in one pass.

\[ \text{customer} \] is used as the probe input:
- Partitioned into 5 partitions, each containing 80 blocks.
- This is also done in one pass.

Therefore total cost: \( 3(100 + 400) = 1500 \) block transfers
- Ignores cost of writing partially filled blocks
Cost of Hash-Join, Cont.

- It is best to choose the smaller relation as the build relation.
  - Because of the inner term in the expression.
  - A smaller number of partitions will result (no recursive partitioning).

- If the smaller relation can fit in main memory, it can be used as the build relation and \( n \) can be set to 1 and the algorithm does not partition the relations into temporary files, but may still build an in-memory index.

- Cost estimate goes down to \( b_r + b_s \).
Even if $s$ is recursively partitioned hash-table overflow can occur, i.e., some partition $s_i$ may not fit in memory.

- Many tuples in $s$ with same value for join attributes.
- Bad hash function.

Overflows can be handed in a variety of ways:

- Resolution (during the build phase):
  - Partition $s_i$ is further partitioned using different hash function.
  - Partition $r_i$ must be similarly partitioned.

- Avoidance (during partition phase):
  - Partition build relation into many partitions, then combine them

Most such approaches fail with large numbers of duplicates:

Another option is to use block nested-loop join on overflowed partitions.
Join involving three relations: \( \text{loan} \bowtie \text{borrower} \bowtie \text{customer} \)

**Strategy 1:** Compute \( \text{borrower} \bowtie \text{customer} \); use result to compute \( \text{loan} \bowtie (\text{borrower} \bowtie \text{customer}) \)

**Strategy 2:** Computer \( \text{loan} \bowtie \text{borrower} \) first, and then join the result with \( \text{customer} \).

**Strategy 3:** Perform the two joins at once:
- Build an index on \( \text{loan} \) for \( \text{loan-number} \), and on \( \text{customer} \) for \( \text{customer-name} \) (if they don’t already exist).
- For each tuple in \( \text{borrower} \), look up the matching tuples in \( \text{customer} \) and in \( \text{loan} \).
- Each tuple of \( \text{borrower} \) is examined exactly once.
Other Operations

- **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 \( r \bowtie s \):
  - Modify merge-join to compute \( r \bowtie s \). During merging, for every tuple \( t \) from \( r \) that does not match any tuple in \( s \), output \( t \) padded with nulls.
  - Right outer-join and full outer-join can be computed similarly.

- Modifying hash join to compute \( r \bowtie s \):
  - If \( r \) is probe relation, output non-matching \( r \) tuples padded with nulls.
  - If \( r \) is build relation, when probing keep track of which \( r \) tuples matched \( s \) tuples. At end of \( s \), output non-matched \( r \) tuples padded with nulls.
Other Operations, Cont.

- **Duplicate elimination:**
  - Sorting - duplicates will come adjacent to each other, and all but one can be deleted.
    - *Optimization:* duplicates can be deleted during run generation as well as at intermediate merge steps in external sort-merge.
  - Hashing - similarly, duplicates will end up in the same bucket.

- **Projection:**
  - Perform projection on each tuple followed by duplicate elimination.
  - Use the leaf-level of a B+ tree to extract required attributes.

- **Aggregation:** (implemented similar to duplicate elimination)
  - Sorting or hashing can be used to bring tuples in the same group together, and then 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.
Set operations (∪, ∩ and −): use a variant of merge-join after sorting, or a variant of hash-join.

Hashing:
1. Partition both relations using the same hash function, thereby creating $r_0, ..., r_{n-1}$ and $s_0, ..., s_{n-1}$
2. For each partition $i$ build an in-memory hash index on $r_i$ (using a different hash function) after it is brought into memory.
3. $r ∪ s$: Add tuples in $s_i$ to the hash index if they are not already in it. Tuples in the hash index comprise the result.
   - $r ∩ s$: Output tuples in $s_i$ to the result if they are already there in the hash index.
   - $r − s$: For each tuple in $s_i$, if it appears in the hash index, delete it. At end of $s_i$ add remaining tuples in the hash index to the result.
So far we have seen algorithms for individual operations.

When evaluating an entire expression, algorithms for different operations must be coordinated.

Passing results from one operator to another can be done in different ways:

- **Materialization**: Evaluate a relational algebraic expression from the bottom-up, explicitly generating and storing the results of each operation.

- **Pipelining**: Evaluate operations in a multi-threaded manner, i.e., pass tuples resulting from one operation to the next (parent), as input, while the first operation is still being executed.
Example:

In a *materialized* evaluation, the expression

\[ \sigma_{balance < 2500}(account) \]

would be computed and stored explicitly. The join with *customer* would then be computed and store explicitly. Finally the projection onto *customer-name* would be computed.
Advantage of materialized evaluation - always possible (assuming there is enough space).

Disadvantage of materialize evaluation - cost of writing/reading results to/from disk can be quite high:
In a *pipelined* evaluation, operations are executed in a multi-threaded manner (simultaneously), passing the results of one operation to the next as they are produced.

Pipelining may not always be possible or easy:

- For example, if the result of a hash-join is passed to a sort.

Pipelines can be *demand driven* or *producer driven*. 
End of Chapter...

(extra material follows)
Hybrid Hash–Join

- Left as an exercise…

- Useful when memory sized are relatively large, and the build input is bigger than memory.

- Hybrid hash join keeps the first partition of the build relation in memory.

- Can be generalized beyond what is described here.
  - Keep the first two partitions in memory, if possible.
With memory size of 25 blocks, *depositor* can be partitioned into five partitions, each consisting of 20 blocks.

Division of memory (during partitioning):
- 1 block is used for input, and 1 block each for buffering 4 of the partitions.
- The 5th partition is maintained in the remaining 20 blocks of the buffer.

*Customer* is similarly partitioned into five partitions each of size 80; the first is used right away for probing, instead of being written out and read back.

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 $M > \sqrt{b_s}$
Join with a conjunctive condition:

\[ r \Join_{\theta_1 \land \theta_2 \land \ldots \land \theta_n} s \]

- Use either nested loop or block nested loop join.
- Compute the result of one of the simpler joins \( r \Join_{\theta_i} s \)
  
  - 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:

\[ r \Join_{\theta_1 \lor \theta_2 \lor \ldots \lor \theta_n} s \]

- Use either nested loop or block nested loop join.
- Compute as the union of the records in individual joins \( r \Join_{\theta_i} s \):
  \[ (r \Join_{\theta_1} s) \cup (r \Join_{\theta_2} s) \cup \ldots \cup (r \Join_{\theta_n} s) \]