Query Processing

Chapter 9

*Database System Concepts*, Cap. 13
A. Silberschatz, H. Korth, and S. Sudarshan

*Database Management Systems*, Cap. 13 & 14
R. Ramakrishnan and J. Gehrke
Overview

- Basic Steps of Query Processing
- Measures of Query Cost
- Selection Operation
- External Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions
Basic Steps in Query Processing

- Parsing and translation
- Optimization
- Evaluation
Basic Steps in Query Processing

- Parsing and translation
  - Translate a SQL query into its internal form. This is then translated into relational algebra.
  - Parser checks syntax, verifies relations

- Optimization
  - The query optimizer translates a relational algebra expression into an execution plan.
  - A relational algebra expression may have many equivalent expressions, each of which gives rise to a different evaluation plan.
  - Amongst all equivalent evaluation plans choose the one with lowest cost.

- Evaluation
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.
Query Processing: Optimization

- A relational algebra expression may have many equivalent expressions
  - e.g., $\sigma_{balance<2500}(\Pi_{balance}(account))$ is equivalent to $\Pi_{balance}(\sigma_{balance<2500}(account))$

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

- Annotated expression specifying detailed evaluation strategy is called an evaluation-plan.
  - e.g., can use an index on balance to find accounts with balance < 2500,
  - or can perform complete relation scan and discard accounts with balance $\geq$ 2500

- Query Optimization: Amongst all equivalent evaluation plans choose the one with lowest cost.
  - Cost is estimated using statistical information from the database catalog
    - e.g. number of tuples in each relation, size of tuples, etc.
Query Processing vs. Optimization

- In this chapter (Query Processing) we study
  - How to measure query costs
  - Algorithms for evaluating relational algebra operations
  - How to combine algorithms for individual operations in order to evaluate a complete expression

- In the next chapter (Query Optimization)
  - We study how to optimize queries, that is, how to find an evaluation plan with lowest estimated cost
Formal Relational Query Languages: a review

- Two mathematical Query Languages form the basis for “real” languages (e.g. SQL), and for implementation: relational algebra and relational calculus.

- **Relational Algebra**: More operational, very useful for representing execution plans.

- **Relational Calculus**: Lets users describe what they want, rather than how to compute it. (Non-procedural, declarative.)

- Understanding Algebra & Calculus is key to understanding SQL, query processing!
5 Basic Operations of the Relational Algebra: a review

- **Selection** \( (\sigma) \) Selects a subset of rows from relation (horizontal).
- **Projection** \( (\pi) \) Retains only wanted columns from relation (vertical).
- **Cross-product** \( (\times) \) Allows us to combine two relations.
- **Set-difference** \( (-) \) Tuples in \( r_1 \), but not in \( r_2 \).
- **Union** \( (\cup) \) Tuples in \( r_1 \) or in \( r_2 \).

Since each operation returns a relation, operations can be composed! (Algebra is "closed").
Overview

- Basic Steps of Query Processing
- Measures of Query Cost
- Selection Operation
- External Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions
Measures of Query Cost

- 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
    - 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
Measures of Query Cost

- For simplicity we just use the number of block transfers from disk and the number of seeks as the cost measures
  - \( t_T \) – time to transfer one block
  - \( t_S \) – time for one seek
  - Cost for \( b \) block transfers plus \( S \) seeks
    \[ b \cdot t_T + S \cdot t_S \]
- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account
- We don’t include cost to writing output to disk in our cost formulae
- 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
Overview

- Basic Steps of Query Processing
- Measures of Query Cost
- Selection Operation
- External Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions
Selection Operation \((\sigma)\): using basic scan algorithms

- **File scan** – search algorithms that locate and retrieve records that fulfill a selection condition.

- **Algorithm Al** (*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 is slow, but it is general because it can be applied regardless of
    - selection condition or
    - ordering of records in the file, or
    - availability of indices
Selection Operation ($\sigma$): using basic scan algorithms

- **A2 (binary search)**. Applicable if selection is an equality comparison on the attribute on which file is ordered.
  - Assume that the blocks of a relation are stored contiguously
  - Cost estimate (number of disk blocks to be scanned):
    - Cost of locating the first tuple by a binary search on the blocks
      - $[\log_2(b_r)] \times (t_t + t_s)$
    - If there are multiple records satisfying selection
      - Add transfer cost of the number of blocks containing records that satisfy selection condition
      - Will see how to estimate this cost in the next chapter
Selection Operation (\(\sigma\)):
using indexes

- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index.

- **A3** *(primary index on candidate key, equality)*. Retrieve a single record that satisfies the corresponding equality condition
  - Cost = \((h_i + 1) \times (t_T + t_S)\)

- **A4** *(primary index on nonkey, equality)* Retrieve multiple records.
  - Records will be on consecutive blocks
    - Let \(b\) = number of blocks containing matching records
    - Cost = \(h_i \times (t_T + t_S) + t_S + t_T \times b\)

- **A5** *(secondary index on search-key, equality)*.
  - Retrieve a single record if the search-key is a candidate key
    - Cost = \((h_i + 1) \times (t_T + t_S)\)
  - Retrieve multiple records if search-key is not a candidate key
    - each of \(n\) matching records may be on a different block
    - Cost = \((h_i + n) \times (t_T + t_S)\)
      - Can be very expensive!
Selections Involving Comparisons

- Can implement selections of the form \( \sigma_{A \leq V}(r) \) or \( \sigma_{A \geq V}(r) \) by using
  - a linear file scan or binary search,
  - or by using indices in the following ways:

- A6 (primary index, comparison). (Relation is sorted on A)
  - For \( \sigma_{A \geq V}(r) \) use index to find first tuple \( \geq v \) and scan relation sequentially from there
  - For \( \sigma_{A \leq V}(r) \) just scan relation sequentially till first tuple > v; do not use index

- A7 (secondary index, comparison).
  - For \( \sigma_{A \geq V}(r) \) use index to find first index entry \( \geq v \) and scan index sequentially from there, to find pointers to records.
  - For \( \sigma_{A \leq V}(r) \) just scan leaf pages of index finding pointers to records, till first entry > v
  - In either case, retrieve records that are pointed to
    - requires an I/O for each record
    - Linear file scan may be cheaper
Complex Selections

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

- **A8** (*conjunctive selection using one index*).
  - Select a combination of \( \theta_i \) and algorithms A1 through A7 that results in the least cost for \( \sigma_{\theta_i}(r) \).
  - Test other conditions on tuple after fetching it into memory buffer.

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

- **A10** (*conjunctive selection by intersection of identifiers*).
  - Requires indices with record pointers.
  - Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers.
  - Then fetch records from file
  - If some conditions do not have appropriate indices, apply test in memory.
Complex Selections

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

- **All** *(disjunctive selection by union of identifiers)*.
  - Applicable if *all* conditions have available indices.
  - Otherwise use linear scan.
  - Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
  - Then fetch records from file.

- **Negation**: \( \sigma_{\neg \theta}(r) \)
  - Use linear scan on file
  - If very few records satisfy \( \neg \theta \), and an index is applicable to \( \theta \)
    - Find satisfying records using index and fetch from file
Overview

- Basic Steps of Query Processing
- Measures of Query Cost
- Selection Operation
- External Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions
External 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.
Why sort?

- A classic problem in computer science!
- Data *requested* in sorted order
  - e.g., find students in increasing *gpa* order
- Sorting is first step in *bulk loading* B+ tree index.
- Sorting useful for *eliminating* *duplicate* copies in a collection of records (Why?)
- *Sort-merge join* algorithm involves sorting.

- Problem: sort 1Gb of data with 1Mb of RAM.
  - why not virtual memory?
- Secondary problem:
  - I/O costs dominate, so design algorithms to reduce I/O
2-Way Sort: Requires 3 Buffers

- Pass 1 (SORT stage): read a page, sort it, write it.
  - only one buffer page is used
- Pass 2, 3, …, etc. (MERGE stage):
  - three buffer pages used.
Example: 2-way external sort-merge

- **Costs per pass**: $2 \times N$

- **# of passes**: height of tree

- **total cost**: product of above

**Idea**: *Divide and conquer*: sort subfiles and merge

\[ N = \text{number of pages of the input file} \]
Cost of the 2-way sort-merge algorithm

- Number of pages \( N = 2^k \) (\( k = 0, 1, \ldots, P-1 \)), where \( P \) is \# of passes.
  - Pass 0 produces \( 2^k \) sorted runs of one page each;
  - Pass 1 produces \( 2^{k-1} \) sorted runs of two pages each;
  - Pass 2 produces \( 2^{k-2} \) sorted runs of four page each;
  - ...
  - Pass \( k \) produces \( 2^0 \) sorted runs of \( 2^k \) pages.

- Costs per pass: \( 2 \cdot N \)
  - each page we read + each page we write into file
- \( P = (\log_2 N) + 1 \)
- Total cost (I/Os) : \( 2 \cdot N \cdot P = 2N(\log_2 N + 1) \)
External Sort-Merge Algorithm:

9-way sort

- Two stages:
  - Sorting
  - Merging
Example: 9-way external sort-merge

For example, for sorting 900 megabytes of data using only 100 megabytes of RAM:

1. Read 100 MB of the data in main memory and sort by some conventional method (usually quicksort).
2. Write the sorted data to disk.
3. Repeat steps 1 and 2 until all of the data is sorted in 100 MB chunks, which now need to be merged into one single output file.
4. Read the first 10 MB of each sorted chunk (call them input buffers) in main memory (90 MB total) and allocate the remaining 10 MB for output buffer.
5. Perform a 9-way merging and store the result in the output buffer. If the output buffer is full, write it to the final sorted file. If any of the 9 input buffers gets empty, fill it with the next 10 MB of its associated 100 MB sorted chunk or otherwise mark it as exhausted if there is no more data in the sorted chunk and do not use it for merging.
General External Merge Sort

To sort a file with $N$ pages using $B$ buffer pages:

- Pass 0: use $B$ buffer pages
  - Produce $(N/B)$ sorted runs of $B$ pages each.
- Pass 1, 2, ..., etc.: merge $B-1$ runs.
**External Sort-Merge Algorithm: SORT stage**

- Let $M$ denote memory buffer size (in frame pages, i.e. the number of disk blocks whose contents can be buffered in memory).
- Create sorted runs.
  - $i = 0$;
  - `repeat`
    - (a) Read $M$ blocks of relation into memory;
    - (b) Sort the in-memory blocks;
    - (c) Write sorted data to run file $R_i$;
    - (c) increment $i = i + 1$;
  - `until` the end of the relation
External Sort-Merge Algorithm: MERGE stage

- Merge the runs (N-way merge). We assume (for now) that $N < M$.
  - 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.
  - repeat
    - Select the first record (in sort order) among all buffer pages.
    - Write the record to the output buffer. If the output buffer is full write it to disk.
    - 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.
  - until all input buffer pages are empty
External Sort-Merge Algorithm: **MERGE stage**

- 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.
Cost of the general sort-merge algorithm

- Number of passes
  \[ P = 1 + \log_{B-1} \left( \frac{N}{B} \right) \]
- I/O cost = 2 . \( N \cdot P \)

**EXAMPLE:**
- Buffer: with 5 buffer pages
- File to sort: 108 pages
  - Pass 0: \([108/5] = 22\) sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: \([22/4] = 6\) sorted runs of 20 pages each (last run is only 8 pages)
  - Pass 2: \([6/4] = 2\) sorted runs, 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages

- Overall I/O cost = 2 . \( N \cdot 4 \)
### Number of Passes of External Sort

- Gain of utilizing all available buffers
- Importance of a high fan-in during merging

<table>
<thead>
<tr>
<th>N</th>
<th>B=3</th>
<th>B=5</th>
<th>B=9</th>
<th>B=17</th>
<th>B=129</th>
<th>B=257</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1,000</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10,000</td>
<td>13</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>100,000</td>
<td>17</td>
<td>9</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1,000,000</td>
<td>20</td>
<td>10</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>10,000,000</td>
<td>23</td>
<td>12</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>100,000,000</td>
<td>26</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>30</td>
<td>15</td>
<td>10</td>
<td>8</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>
Using Double Buffering for Sorting

- It allows to overlap CPU and I/O processing, reducing the I/O cost.
- To reduce wait time for I/O request to complete, can \textit{prefetch} into `shadow block`; potentially, more passes; in practice, most files still sorted in 2-3 passes.
Sorting has become a blood sport!
- Parallel sorting is the name of the game ...

Datamation: Sort 1M records of size 100 bytes
- Typical DBMS: 15 minutes
- World record: 3.5 seconds
  - 12-CPU SGI machine, 96 disks, 2GB of RAM

New benchmarks proposed:
- Minute Sort: How many can you sort in 1 minute?
- Dollar Sort: How many can you sort for $1.00?
Using B+ Trees for Sorting

- Scenario: Table to be sorted has B+ tree index on sorting column(s).
- Idea: Can retrieve records in order by traversing leaf pages.
- *Is this a good idea?*
- Cases to consider:
  - B+ tree is clustered  *Good idea!*
  - B+ tree is not clustered  *Could be a very bad idea!*
Clustered B+ Trees for Sorting

- Cost: root to left-most leaf, then retrieve all leaf pages (Alternative 1)
- For Alternative 2, additional cost of retrieving data records: each page fetched just once.

⇒ Always better than external sorting!
Unclustered B+ Trees for Sorting

- Alternative (2) for data entries; each data entry contains rid of a data record. In general, one I/O per data record!
### External Sorting vs. Unclustered Index (worst-case numbers)

<table>
<thead>
<tr>
<th>N</th>
<th>Sorting</th>
<th>p=1</th>
<th>p=10</th>
<th>p=100</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>200</td>
<td>100</td>
<td>1,000</td>
<td>10,000</td>
</tr>
<tr>
<td>1,000</td>
<td>2,000</td>
<td>1,000</td>
<td>10,000</td>
<td>100,000</td>
</tr>
<tr>
<td>10,000</td>
<td>40,000</td>
<td>10,000</td>
<td>100,000</td>
<td>1,000,000</td>
</tr>
<tr>
<td>100,000</td>
<td>600,000</td>
<td>100,000</td>
<td>1,000,000</td>
<td>10,000,000</td>
</tr>
<tr>
<td>1,000,000</td>
<td>8,000,000</td>
<td>1,000,000</td>
<td>10,000,000</td>
<td>100,000,000</td>
</tr>
<tr>
<td>10,000,000</td>
<td>80,000,000</td>
<td>10,000,000</td>
<td>100,000,000</td>
<td>1,000,000,000</td>
</tr>
</tbody>
</table>

Cost of External Sorting ($B=1000$, $b=32$) vs. Unclustered Index

- $N =$ number of data pages
- $p =$ average number of records per page
- $B = 1,000$ and block size $b=32$ pages for sorting
- $p = 100$ is the more realistic value.
Summary

- External sorting is important; DBMS may dedicate part of buffer pool for sorting!

- External merge sort minimizes disk I/O cost:
  - Pass 0: Produces sorted runs of size $B$ (# buffer pages). Later passes: merge runs.
  - # of runs merged at a time depends on $B$, and block size.
  - Larger block size means less I/O cost per page.
  - Larger block size means smaller # runs merged.
  - In practice, # of runs rarely more than 2 or 3.

- Choice of internal sort algorithm may matter.

- The best sorts are wildly fast:
  - Despite 40+ years of research, we’re still improving!

- Clustered B+ tree is good for sorting; unclustered tree is usually very bad.
Overview

- Basic Steps of Query Processing
- Measures of Query Cost
- Selection Operation
- External Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions
Joins are among the most expensive operators in a DBMS, and their implementation has a big impact on performance.

Example: $R \bowtie S$

\[
\begin{align*}
\text{SELECT} & \quad * \\
\text{FROM} & \quad \text{Reserves } R, \text{ Sailors } S \\
\text{WHERE} & \quad R.\text{sid}=S.\text{sid}
\end{align*}
\]

After discussing the join operator, we consider implementation of the binary operators cross-product, intersection, union, and set-difference.

We also discuss the implementation of grouping and aggregate operators, which are extensions of relational algebra.
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 customer: 10,000
  - Number of blocks of customer: 400
  - Number of records of depositor: 5000
  - Number of blocks of depositor: 100
Nested-Loop Join

- Algorithm: we scan the outer relation $R$, and for each tuple $r \in R$, we scan the entire inner relation $S$:

  ```
  foreach tuple $r \in R$ do
    foreach tuple $s \in S$ do
      if pair $(r,s)$ satisfies the join condition $\theta$
      then add $(r,s)$ to the result
  ```

- 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.
Nested-Loop Join: Cost

- In the **worst case**, if there is enough memory only to hold one block of each relation, the estimated cost is $b_R + n_R \cdot b_S$ block transfers, plus $b_R + n_R$ seeks.
- **Best case**: 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 *depositor* as outer relation:
    - $5000 \cdot 400 + 100 = 2,000,100$ block transfers,
    - $5000 + 100 = 5100$ seeks
  - with *customer* as the outer relation
    - $10000 \cdot 100 + 400 = 1,000,400$ block transfers and 10,400 seeks
- If smaller relation (*depositor*) fits entirely in memory, the cost estimate will be 500 block transfers.
- Block nested-loops algorithm (next slide) is preferable.
Block Nested-Loop Join

Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

```plaintext
foreach block \( B_R \) of \( R \) do
  foreach block \( B_S \) of \( S \) do
    foreach tuple \( r \in B_R \) do
      foreach tuple \( s \in B_S \) do
        if pair \( (r,s) \) satisfies the join condition \( \theta \) then
          add \( (r,s) \) to the result
```
**Block Nested-Loop Join: Cost**

- **Worst case** estimate: $b_R + b_R \cdot b_S$ block transfers and $2 \cdot b_R$ seeks
  - Each block in the inner relation $s$ is read once for each block in the outer relation (instead of once for each tuple 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 = $\left\lfloor b_R / (M-2) \right\rfloor \cdot b_S + b_R$ block transfers + $2 \left\lfloor b_R / (M-2) \right\rfloor$ 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 \( r \in R \), use the index to look up tuples in the inner relation \( S \) that satisfy the join condition with tuple \( r \).

- **Worst case**: buffer has space for only one page of \( R \) and one page of the index; for each tuple in \( R \), we perform an index lookup on \( S \).

- **Worst case cost of the join**: \( b_R + n_R \cdot c \)
  - Where \( c \) is the cost of traversing index and fetching all matching \( S \) tuples for one tuple of \( R \)
  - \( c \) 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.
Example of Nested-Loop Join Costs

- Compute \( \text{depositor} \times \text{customer} \), with \( \text{depositor} \) as the outer relation.

- Let \( \text{customer} \) have a primary \( \mathbb{B}^+ \)-tree index on the join attribute \( \text{customer-name} \), which contains 20 entries in each index node.

- Since \( \text{customer} \) has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data

- \( \text{depositor} \) has 5000 tuples

- Cost of block nested loops join
  - \( 400 \times 100 + 100 = 40,100 \) block transfers + \( 2 \times 100 = 200 \) seeks
    - assuming worst case memory
    - may be significantly less with more memory

- Cost of indexed nested loops join
  - \( 100 + 5000 \times 5 = 25,100 \) block transfers and seeks.
  - CPU cost likely to be less than that for block nested loops join
(Sort-)Merge-Join

- **Sort** both relations on their join attribute (if not already sorted on the join attributes).
- **Merge** the sorted relations to join them
  - Join step is similar to the merge stage of the sort-merge algorithm.
  - Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched
  - Detailed algorithm in book
Merge-Join: Cost

- 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:
  \[ b_R + b_S \text{ block transfers} + \left( \frac{b_R}{b_b} \right) + \left( \frac{b_S}{b_b} \right) \text{ seeks} \]
  + 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

\[ n_R = \text{number of tuples of } R \]
\[ b_R = \text{number of blocks containing tuples of } R \]
\[ b_b = \text{number of buffer blocks allocated for each relation} \]
Hash-Join

- Applicable for equi-joins and natural joins.
- A hash function $h$ is used to partition tuples of both relations.
- $h$ maps $\text{JoinAttrs}$ values to $\{0, 1, \ldots, n\}$, where $\text{JoinAttrs}$ 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 $i = h(t_r[\text{JoinAttrs}])$.
  - $s_0, s_1, \ldots, s_n$ denotes partitions of $s$ tuples
    - Each tuple $t_s \in s$ is put in partition $s_i$, where $i = h(t_s[\text{JoinAttrs}])$.

- 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.)

- $r$ 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 $s_i$. 

Não abordado!
Hash-Join Algorithm

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

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.
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 $\lceil b_s/M \rceil \times f$ 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., recursive partitioning not needed for relations of 1GB or less with memory size of 2MB, with block size of 4KB.
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
  \[3(b_r + b_s) + 4 \times n_h \text{ block transfers} +
  2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) \text{ seeks}\]

- If recursive partitioning required:
  - number of passes required for partitioning build relation
    \[s = \lceil \log_{M-1}(b_s) - 1 \rceil\]
  - best to choose the smaller relation as the build relation.
  - Total cost estimate is:
    \[2(b_r + b_s \lceil \log_{M-1}(b_s) - 1 \rceil) + b_r + b_s \text{ block transfers} +
    2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) \lceil \log_{M-1}(b_s) - 1 \rceil \text{ seeks}\]

- 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

- Assume that memory size is 20 blocks
- $b_{depositor} = 100$ and $b_{customer} = 400$.
- *depositor* 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 *customer* 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, *depositor* 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.

- *customer* 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 $M \gg \sqrt{b_s}$
Complex Joins

- Join with a conjunctive condition:

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

  - Either use nested loops/block nested loops, or
  - Compute the result of one of the simpler joins \( r \Join_{\theta_i} s \)
    
    final result comprises those tuples in the intermediate result that satisfy the remaining conditions

    \[ \theta_1 \land \ldots \land \theta_{j-1} \land \theta_{j+1} \land \ldots \land \theta_n \]

- Join with a disjunctive condition

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

  - Either use nested loops/block nested loops, or
  - Compute as the union of the records in individual joins \( r \Join_{\theta_i} s \):

    \[ (r \Join_{\theta_1} s) \cup (r \Join_{\theta_2} s) \cup \ldots \cup (r \Join_{\theta_n} s) \]
Overview

- Basic Steps of Query Processing
- Measures of Query Cost
- Selection Operation
- External Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions
**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.
**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 $\setminus$): can either use variant of merge-join after sorting, or variant of hash-join.
- E.g., Set operations using hashing:
  - Partition both relations using the same hash function
  - Process each partition $i$ as follows.
    - Using a different hashing function, build an in-memory hash index on $r_i$.
    - Process $s_i$ as follows
      - $r \cup s_i$:
        - Add tuples in $s_i$ to the hash index if they are not already in it.
        - At end of $s_i$ add the tuples in the hash index to the result.
      - $r \cap s_i$:
        - Output tuples in $s_i$ to the result if they are already there in the hash index
      - $r \setminus s_i$:
        - for each tuple in $s_i$ if it is there in the hash index, delete it from the index.
        - At end of $s_i$ add remaining tuples in the hash index to the result.
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 $r - \Pi_R(r \bowtimes s)$
  - 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.
- 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 $s_i$ output non-matched $r$ tuples padded with nulls
Overview

- Basic Steps of Query Processing
- Measures of Query Cost
- Selection Operation
- External Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions
Evaluation of Expressions

- So far: we have seen algorithms for individual operations

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

- We study above alternatives in more detail
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 

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

then compute the store its join with customer, and finally compute the projections on customer-name.
Materialization (Cont.)

- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be quite high
  - Our cost formulas for operations ignore cost of writing results to disk, so
    - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk
- Double buffering: use two output buffers for each operation, when one is full write it to disk while the other is getting filled
  - Allows overlap of disk writes with computation and reduces execution time
Pipelining

- **Pipelined evaluation**: evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don’t store result of \( \sigma_{balance<2500}(\text{account}) \)
  - instead, pass tuples directly to the join. Similarly, don’t store result of join, pass tuples directly to projection.
- Much cheaper than materialization: no need to store a temporary relation to disk.
- Pipelining may not always be possible – e.g., sort, hash-join.
- For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation.
- Pipelines can be executed in two ways: **demand driven** and **producer driven**.
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
Implementation of demand-driven pipelining

Each operation is implemented as an iterator implementing the following operations:

- **open()**
  - E.g. file scan: initialize file scan
    - state: pointer to beginning of file
  - E.g. merge join: sort relations;
    - state: pointers to beginning of sorted relations

- **next()**
  - E.g. for file scan: Output next tuple, and advance and store file pointer
  - E.g. for merge join: continue with merge from earlier state till next output tuple is found. Save pointers as iterator state.

- **close()**
Evaluation Algorithms for Pipelining

- Some algorithms are not able to output results even as they get input tuples
  - E.g. merge join, or hash join
  - Intermediate results written to disk and then read back

- Algorithm variants to generate (at least some) results on the fly, as input tuples are read in
  - E.g. hybrid hash join generates output tuples even as probe relation tuples in the in-memory partition (partition 0) are read in
  - **Pipelined join technique**: Hybrid hash join, modified to buffer partition 0 tuples of both relations in-memory, reading them as they become available, and output results of any matches between partition 0 tuples
    - When a new $r_0$ tuple is found, match it with existing $s_0$ tuples, output matches, and save it in $r_0$
    - Symmetrically for $s_0$ tuples
Summary

- Overview
- Measures of Query Cost
- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions