L11: Query Processing

- Implementing Relational Operators
  - External sorting
  - Join
  - Selection

Note: Slides whose titles are put in () are for your reference only. Details will be covered in COMP9315.
Query Processing & Optimization

- Any high-level query (SQL) on a database must be processed, optimized and executed by the DBMS.
- The high-level query is scanned, and parsed to check for syntactic correctness.
- An internal representation of a query is created, which is either a query tree or a query graph.
- The DBMS then devises an execution strategy for retrieving the result of the query. (An execution strategy is a plan for executing the query by accessing the data, and storing the intermediate results.)
- The process of choosing one out of the many execution strategies is known as query optimization.
Query Processor & Query Optimizer

- A query processor is a module in the DBMS that performs the tasks to process, to optimize, and to generate execution strategy for a high-level query.
- Queries expressed in SQL can have multiple equivalent relational algebra query expressions. The query optimizer must select the optimal one.
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?
2-Way Sort: Requires 3 Buffers

- Pass 1: Read a page, sort it, write it.
  - only one buffer page is used
- Pass 2, 3, ..., etc.:
  - three buffer pages used.
Two-Way External Merge Sort

- Each pass we read + write each page in file.
- N pages in the file => the number of passes
  \[= \lceil \log_2 N \rceil + 1\]
- So total cost is:
  \[2N \left( \lceil \log_2 N \rceil + 1 \right)\]

**Idea:** Divide and conquer: sort subfiles (aka runs) and merge
General External Merge Sort

More than 3 buffer pages. How can we utilize them?

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

- Pass 0: use $B$ buffer pages. Produce $\left\lceil \frac{N}{B} \right\rceil$ sorted runs of $B$ pages each.
- Pass 1, ..., etc.: merge $B-1$ runs.

More than 3 buffer pages. How can we utilize them?
Cost of External Merge Sort

- **Number of passes:** \(1 + \lceil \log_{B-1} \left( \frac{N}{B} \right) \rceil\)
- **Cost:** \(2N \times \text{(number of passes)}\)
- **E.g., with 5 buffer pages, to sort 108 page file:**
  - Pass 0: \(\lceil \frac{108}{5} \rceil = 22\) sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: \(\lceil \frac{22}{4} \rceil = 6\) sorted runs of 20 pages each (last run is only 8 pages)
  - Pass 2: 2 sorted runs, 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages
### Number of Passes of External Sort

<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 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+ Tree Used for Sorting

- Cost: root to the left-most leaf, then retrieve all leaf pages (Alternative 1)
- If Alternative 2 is used? Additional cost of retrieving data records: each page fetched just once.

Always better than external sorting!
Unclustered B+ Tree Used for Sorting

- Alternative (2) for data entries; each data entry contains \( rid \) of a data record. In general, one I/O per data record!

```
<table>
<thead>
<tr>
<th>Index (Directs search)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Entries</td>
</tr>
<tr>
<td>&quot;Sequence set&quot;</td>
</tr>
<tr>
<td>Data Records</td>
</tr>
</tbody>
</table>
```
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 $B$.
  - In practice, # of runs rarely more than 2 or 3.
Relational Operations

- We will consider how to implement:
  - Selection: Selects a subset of rows from relation.
  - Projection: Deletes unwanted columns from relation.
  - Join: Allows us to combine two relations.

- Since each op returns a relation, ops can be composed! After we cover the operations, we will discuss how to optimize queries formed by composing them.
Schema for Examples

- **Sailors**
  - (\textit{sid}: integer, \textit{sname}: string, \textit{rating}: integer, \textit{age}: real)
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.

- **Reserves**
  - (\textit{sid}: integer, \textit{bid}: integer, \textit{day}: dates, \textit{rname}: string)
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.

- Similar to old schema; \textit{rname} added for variations.
Equality Joins With One Join Column

SELECT * 
FROM Reserves R1, Sailors S1 
WHERE R1.sid=S1.sid

- In algebra: $R \times S$. Common! Must be carefully optimized. $R \times S$ is large; so, $R \times S$ followed by a selection is inefficient.
  - In our examples, $R$ is Reserves and $S$ is Sailors.
- We will consider more complex join conditions later.
- Cost metric: # of I/Os. We will ignore output costs.
Simple Nested Loops Join

for each tuple \( r \) in \( R \) do
  for each tuple \( s \) in \( S \) do
    if \( r_i = s_j \) then add \( <r, s> \) to result

- For each tuple in the outer relation \( R \), we scan the entire inner relation \( S \).
  - Cost: \( M + p_{R} \times M \times N = 1000 + 100 \times 1000 \times 500 \) I/Os.

- Page-oriented Nested Loops join: For each page of \( R \), get each page of \( S \), and write out matching pairs of tuples \( <r, s> \), where \( r \) is in \( R \)-page and \( S \) is in \( S \)-page.
  - Cost: \( M + M \times N = 1000 + 1000 \times 500 \) 40 mins
  - If smaller relation (\( S \)) is outer, cost = \( 500 + 500 \times 1000 \) 70 hours, assuming 200 I/O per sec
Block Nested Loops Join

- Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold ``block'' of outer R.
- For each matching tuple r in R-block, s in S-page, add <r, s> to result. Then read next R-block, scan S, etc.

![Diagram of Block Nested Loops Join](image)
Examples of Block Nested Loops

- **Cost:** Scan of outer + \#outer blocks * scan of inner
  - \#outer blocks = \[\text{no of pages of outer} / \text{blocksize}\]

- **With Reserves (R) as outer, and 100 pages of R:**
  - B \(\geq\) 100+2 = 102
  - Cost of scanning R is 1000 I/Os; a total of 10 blocks.
  - Per block of R, we scan Sailors (S); 10*500 I/Os.
  - Total = 1000 + 10*500 = 6000
  - If space for just 90 pages of R, we would scan S 12 times, and cost will be 1000 + \(\text{ceil}(1000/(90))\)*500 = 7000

- **With 100-page block of Sailors as outer:**
  - B \(\geq\) 102
  - Cost of scanning S is 500 I/Os; a total of 5 blocks.
  - Per block of S, we scan Reserves; 5*1000 I/Os.
  - Total = 500 + 5*1000 = 5500

- **With sequential reads considered, analysis changes:** may be best to divide buffers evenly between R and S.
Index Nested Loops Join

for each tuple r in R do
    for each tuple s in S where r_i == s_j do
        add <r, s> to result

- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
  - Cost: \( M + (M \times pR) \times \text{cost of finding matching S tuples} \)

- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples (assuming Alt. (2) or (3) for data entries) depends on clustering.
  - Clustered index: 1 I/O (typical), unclustered: upto 1 I/O per matching S tuple.
Examples of Index Nested Loops

- **B⁺-tree index (Alt. 2) on sid of Sailors (as inner):**
  - Scan Reserves: 1000 page I/Os, 100*1000 tuples.
  - For each Reserves tuple: 3 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Sailors tuple. Total: 401,000 I/Os.

- **B⁺-tree index (Alt. 2) on sid of Reserves (as inner):**
  - Scan Sailors: 500 page I/Os, 80*500 tuples.
  - For each Sailors tuple: 3 I/Os to find index page with data entries, plus cost of retrieving matching Reserves tuples. Assuming uniform distribution, 2.5 reservations per sailor (100,000 / 40,000). Cost of retrieving them is 1 or 2.5 I/Os depending on whether the index is clustered.
    - Best case: clustered, total cost = 160,500 I/Os
Sort-Merge Join

- Sort R and S on the join column, then scan them to do a "merge" (on join col.), and output result tuples.
  - Advance scan of R until current R-tuple $\geq$ current S tuple, then advance scan of S until current S-tuple $\geq$ current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value in Ri (current R group) and all S tuples with same value in Sj (current S group) match; output $<r, s>$ for all pairs of such tuples.
  - Then resume scanning R and S.

- R is scanned once; each S group is scanned once per matching R tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)
Example of Sort-Merge Join

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
<th>rname</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>103</td>
<td>12/4/96</td>
<td>guppy</td>
</tr>
<tr>
<td>28</td>
<td>103</td>
<td>11/3/96</td>
<td>yuppy</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/10/96</td>
<td>dustin</td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>10/12/96</td>
<td>lubber</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/11/96</td>
<td>lubber</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
<td>dustin</td>
</tr>
</tbody>
</table>

- Cost: $O(M \log_B M) + O(N \log_B N) + (M+N)$
- The cost of scanning, $M+N$, could be $M*N$ (very unlikely!)
- With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.

(BNL cost: 2500 to 15000 I/Os)
(Hash-Join)

- Partition both relations using hash fn $h$: $R$ tuples in partition $i$ will only match $S$ tuples in partition $i$.

- Read in a partition of $R$, hash it using $h_2$ ($\neq h_i$). Scan matching partition of $S$, search for matches.
(Observations on Hash-Join)

- #partitions $k < B-1$ (why?), and $B-2 >$ size of largest partition to be held in memory. Assuming uniformly sized partitions, and maximizing $k$, we get:
  - $k = B-1$, and $M/(B-1) < B-2$, i.e., $B$ must be $> \sqrt{M}$

- If we build an in-memory hash table to speed up the matching of tuples, a little more memory is needed.

- If the hash function does not partition uniformly, one or more $R$ partitions may not fit in memory. Can apply hash-join technique recursively to do the join of this $R$-partition with corresponding $S$-partition.
Cost of Hash-Join

- In partitioning phase, read+write both relns; $2(M+N)$. In matching phase, read both relns; $M+N$ I/Os. So total cost is $3(M+N)$
  - Requires $B > \sqrt{\min(M, N)}$
- In our running example, this is a total of 4500 I/Os.
- Sort-Merge Join vs. Hash Join:
  - Given a minimum amount of memory (what is this, for each?) both have a cost of $3(M+N)$ I/Os. Hash Join superior on this count if relation sizes differ greatly. Also, Hash Join shown to be highly parallelizable.
  - Sort-Merge less sensitive to data skew; result is sorted.
General Join Conditions

- Equalities over several attributes (e.g., \( R.\text{sid}=S.\text{sid} \) AND \( R.\text{rname}=S.\text{sname} \)):
  - For Index NL, build index on \(<\text{sid, sname}>\) (if \( S \) is inner); or use existing indexes on sid or sname.
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.

- Inequality conditions (e.g., \( R.\text{rname} < S.\text{sname} \)):
  - For Index NL, need (clustered!) B+ tree index.
    - Range probes on inner; # matches likely to be much higher than for equality joins.
  - Hash Join, Sort Merge Join not applicable.
  - Block NL quite likely to be the best join method here.
Simple Selections

- Of the form $\sigma_{R.\text{attr}} \text{ op } \text{value}(R)$
- Size of result approximated as size of $R \times$ reduction factor (a.k.a., selectivity); we will consider how to estimate reduction factors later.
- With no index, unsorted: Must essentially scan the whole relation; cost is $M$ (#pages in $R$).
- With an index on selection attribute: (1) Use index to find qualifying data entries, (2) then retrieve corresponding data records (#qualifying tuples, clustered/non-clustered, sparse/dense matters).
  - Hash index useful only for equality selections.

```sql
SELECT * FROM Reserves R WHERE R.rname < 'C%'
```
Using an Index for Selections

- Cost depends on #qualifying tuples, and clustering.
  - Cost of finding qualifying data entries (typically small) plus cost of retrieving records (could be large w/o clustering).
  - In example, assuming uniform distribution of names, about 10% of tuples qualify (100 pages, 10000 tuples). With a clustered index, cost is little more than 100 I/Os; if unclustered, upto 10000 I/Os!

- Important refinement for unclustered indexes:
  1. Find qualifying data entries.
  2. Sort the rid’s of the data records to be retrieved.
  3. Fetch rids in order. This ensures that each data page is looked at just once (though # of such pages likely to be higher than with clustering).
General Selection Conditions

\[(\text{day}<8/9/94 \:\text{AND} \:\text{rname}='\text{Paul}') \:\text{OR} \:\text{bid}=5 \:\text{OR} \:\text{sid}=3]\]

- Such selection conditions are first converted to conjunctive normal form (CNF): \((\text{day}<8/9/94 \:\text{OR} \:\text{bid}=5 \:\text{OR} \:\text{sid}=3) \:\text{AND} \:(\text{rname}='\text{Paul}') \:\text{OR} \:\text{bid}=5 \:\text{OR} \:\text{sid}=3)\)
- We only discuss the case with no ORs (a conjunction of terms of the form attr op value).
- An index *matches* (a conjunction of) terms that involve only attributes in a prefix of the search key.
  - Index on \(<a, b, c>\) matches \(a=5\ \text{AND} \:b=3\), but not \(b=3\).
Two Approaches to General Selections

- First approach: Find the most selective access path, retrieve tuples using it, and apply any remaining terms that don’t match the index:
  - Most selective access path: An index or file scan that we estimate will require the fewest page I/Os.
  - Terms that match this index reduce the number of tuples retrieved; other terms are used to discard some retrieved tuples, but do not affect number of tuples/pages fetched.
  - Consider day<8/9/94 AND bid=5 AND sid=3. A B+ tree index on day can be used; then, bid=5 and sid=3 must be checked for each retrieved tuple. Similarly, a hash index on <bid, sid> could be used; day<8/9/94 must then be checked.
Intersection of Rids

- Second approach (if we have 2 or more matching indexes that use Alternatives (2) or (3) for data entries):
  - Get sets of rids of data records using each matching index.
  - Then intersect these sets of rids.
  - Retrieve the records and apply any remaining terms.
  - Consider day<8/9/94 AND bid=5 AND sid=3. If we have a B+ tree index on day and an index on sid, both using Alternative (2), we can retrieve rids of records satisfying day<8/9/94 using the first index, rids of recs satisfying sid=3 using the second index, intersect, retrieve records and check bid=5.
(The Projection Operation)

- An approach based on sorting:
  - Modify Pass 0 of external sort to eliminate unwanted fields. Thus, runs of about 2B pages are produced, but tuples in runs are smaller than input tuples. (Size ratio depends on # and size of fields that are dropped.)
  - Modify merging passes to eliminate duplicates. Thus, number of result tuples smaller than input. (Difference depends on # of duplicates.)
  - Cost: In Pass 0, read original relation (size M), write out same number of smaller tuples. In merging passes, fewer tuples written out in each pass. Using Reserves example, 1000 input pages reduced to 250 in Pass 0 if size ratio is 0.25
(Projection Based on Hashing)

- **Partitioning phase:** Read R using one input buffer. For each tuple, discard unwanted fields, apply hash function $h_1$ to choose one of $B-1$ output buffers.
  - Result is $B-1$ partitions (of tuples with no unwanted fields). 2 tuples from different partitions guaranteed to be distinct.

- **Duplicate elimination phase:** For each partition, read it and build an in-memory hash table, using hash fn $h_2 (\not= h_1)$ on all fields, while discarding duplicates.
  - If partition does not fit in memory, can apply hash-based projection algorithm recursively to this partition.

- **Cost:** For partitioning, read R, write out each tuple, but with fewer fields. This is read in next phase.
(Discussion of Projection)

- Sort-based approach is the standard; better handling of skew and result is sorted.

- If an index on the relation contains all wanted attributes in its search key, can do index-only scan.
  - Apply projection techniques to data entries (much smaller!)

- If an ordered (i.e., tree) index contains all wanted attributes as prefix of search key, can do even better:
  - Retrieve data entries in order (index-only scan), discard unwanted fields, compare adjacent tuples to check for duplicates.
Impact of Buffering

- If several operations are executing concurrently, estimating the number of available buffer pages is guesswork.

- Repeated access patterns interact with buffer replacement policy.
  - e.g., Inner relation is scanned repeatedly in Simple Nested Loop Join. With enough buffer pages to hold inner, replacement policy does not matter. Otherwise, MRU is best, LRU is worst (sequential flooding).
  - Does replacement policy matter for Block Nested Loops?
  - What about Index Nested Loops? Sort-Merge Join?
Summary

- A virtue of relational DBMSs: queries are composed of a few basic operators; the implementation of these operators can be carefully tuned (and it is important to do this!).
- Many alternative implementation techniques for each operator; no universally superior technique for most operators.
- Must consider available alternatives for each operation in a query and choose best one based on system statistics, etc. This is part of the broader task of optimizing a query composed of several ops.

A list of cost formula will be available in the tutorial slides.