Query Processing: The Basics

Chapter 10

External Sorting

- Sorting is used in implementing many relational operations
- Problem:
  - Relations are typically large, do not fit in main memory
  - So cannot use traditional in-memory sorting algorithms
- Approach used:
  - Combine in-memory sorting with clever techniques aimed at minimizing I/O
  - I/O costs dominate => cost of sorting algorithm is measured in the number of page transfers

External Sorting (cont’d)

- External sorting has two main components:
  - Computation involved in sorting records in buffers in main memory
  - I/O necessary to move records between mass store and main memory

Simple Sort Algorithm

- \( M \) = number of main memory page buffers
- \( F \) = number of pages in file to be sorted
- Typical algorithm has two phases:
  - Partial sort phase: sort \( M \) pages at a time; create \( F/M \) sorted runs on mass store, cost = \( 2F \)

Example: \( M = 2, F = 7 \)

Simple Sort Algorithm

- **Merge Phase**: merge all runs into a single run using \( M-1 \) buffers for input and 1 output buffer

  - Merge step: divide runs into groups of size \( M-1 \) and merge each group into a run; cost = \( 2F \)

  *each step reduces number of runs by a factor of \( M-1 \)*

Merge: An Example
**Simple Sort Algorithm**

- Cost of merge phase:
  - \( \frac{(F/M)/(M-1)^k} \) runs after \( k \) merge steps
  - \( \lceil \log_{M-1}(F/M) \rceil \) merge steps needed to merge an initial set of \( F/M \) sorted runs
  - \( \text{cost} = \lceil 2F \log_{M-1}(F/M) \rceil \approx 2F(\log_{M-1}F - 1) \)
- Total cost = cost of partial sort phase + cost of merge phase \( \approx 2F \log_{M-1}F \)

**Duplicate Elimination**

- A major step in computing projection, union, and difference relational operators
- Algorithm:
  - Sort
  - At the last stage of the merge step eliminate duplicates on the fly
  - No additional cost (with respect to sorting) in terms of I/O

**Duplicate elimination During Merge**

**Sort-Based Projection**

- Algorithm:
  - Sort rows of relation at cost of \( 2F \log_{M-1}F \)
  - Eliminate unwanted columns in partial sort phase (no additional cost)
  - Eliminate duplicates on completion of last merge step (no additional cost)
- Cost: the cost of sorting

**Hash-Based Projection**

- **Phase 1**:
  - Input rows
  - Project out columns
  - Hash remaining columns using a hash function with range \( 1 \ldots M-1 \) creating \( M-1 \) buckets on disk
  - Cost = \( 2F \)
- **Phase 2**:
  - Sort each bucket to eliminate duplicates
    - Cost (assuming a bucket fits in \( M-1 \) buffer pages) = \( 2F \)
- Total cost = \( 4F \)

**Computing Selection \( \sigma_{(attr \ op \ value)} \)**

- No index on \( attr \):
  - If rows are not sorted on \( attr \):
    - Scan all data pages to find rows satisfying selection condition
    - Cost = \( F \)
  - If rows are sorted on \( attr \) and \( op \) is \( =, >, < \) then:
    - Use binary search (at \( \log_2 F \) ) to locate first data page containing row in which \( (attr \ text{ } value) \)
    - Scan further to get all rows satisfying \( (attr \ text{ } value) \)
    - Cost = \( \log_2 F + \) (cost of scan)
Computing Selection $\sigma_{(\text{attr op value})}$

- **Clustered** $B^+$ tree index on $\text{attr}$ (for "$=$" or range search):
  - Locate first index entry corresponding to a row in which $(\text{attr} = \text{value})$. Cost = depth of tree.
  - Rows satisfying condition packed in sequence in successive data pages; scan those pages.
  - Cost: number of pages occupied by qualifying rows.

- **Unclustered** $B^+$ tree index on $\text{attr}$ (for "$=$" or range search):
  - Locate first index entry corresponding to a row in which $(\text{attr} = \text{value})$.
  - Cost = depth of tree.
  - Index entries with pointers to rows satisfying condition are packed in sequence in successive index pages.
  - Cost: number of rows that satisfy selection condition.

Unclustered $B^+$ Tree Index

- **Hash index** on $\text{attr}$ (for equality search):
  - Hash on $\text{value}$. Cost = 1.2
  - 1.2 typical average cost of hashing (> 1 due to possible overflow chains).
  - Finds the (unique) bucket containing all index entries satisfying selection condition.
  - Clustered index - all qualifying rows packed in the bucket (a few pages)
  - Unclustered index - sort row Ids in the index entries to identify data pages with qualifying rows.
  - Each page containing at least one such row must be fetched once.
  - Cost: min(number of qualifying rows in bucket, number of pages in file).

Access Path

- **Access path** is the notion that denotes algorithm + data structure used to locate rows satisfying some condition.
- **Examples**:
  - File scan: can be used for any condition.
  - Hash: equality search; all search key attributes of hash index are specified in condition.
  - $B^+$ tree: equality or range search; a prefix of the search key attributes are specified in condition.
  - $B^+$ tree supports a variety of access paths.
  - Binary search: Relation sorted on a sequence of attributes and some prefix of that sequence is specified in condition.
Access Paths Supported by $B^+$ tree

- **Example**: Given a $B^+$ tree whose search key is the sequence of attributes $a_2$, $a_1$, $a_3$, $a_4$
  - Access path for search $\sigma_{a_2=3 \land a_1 > 5 \land a_3 = 'x'}(R)$: find first entry having $a_2=3$ and scan leaves from there until entry having $a_2 > 3$ or $a_3 \neq 'x'$. Select satisfying entries
  - Access path for search $\sigma_{a_2=3 \land a_3 > 'x'}(R)$: locate first entry having $a_2 = 3$ and scan leaves until entry having $a_2 > 3$. Select satisfying entries
  - Access path for search $\sigma_{a_1 > 5 \land a_3 = 'x'}(R)$: Scan of $R$

Choosing an Access Path

- **Selectivity** of an access path = number of pages retrieved using that path
- If several access paths support a query, DBMS chooses the one with **lowest** selectivity
- Size of domain of attribute is an indicator of the selectivity of search conditions that involve that attribute

Example: $\sigma_{\text{CrsCode}=\text{CS305} \land \text{Grade}=\text{B}}(\text{Transcript})$
- a $B^+$ tree with search key $\text{CrsCode}$ has lower selectivity than a $B^+$ tree with search key $\text{Grade}$

Computing Joins

- The cost of joining two relations makes the choice of a join algorithm crucial
- Simple **block-nested loops** join algorithm for computing $r \bowtie_s A=B$

  ```plaintext
  foreach page $p_r$ in $r$ do
      foreach page $p_s$ in $s$ do
          output $p_r \bowtie_s A=B p_s$
  ```

Block-Nested Loops Join

- If $\beta_r$ and $\beta_s$ are the number of pages in $r$ and $s$, the cost of algorithm is
  \[
  \beta_r + \beta_s + \beta_r \cdot \beta_s + \text{cost of outputting final result}
  \]

  - If $r$ and $s$ have $10^3$ pages each, cost is $10^3 + 10^3 \cdot 10^3$
  - Choose smaller relation for the outer loop:
    - If $\beta_r < \beta_s$ then $\beta_r + \beta_r \cdot \beta_s < \beta_s + \beta_s \cdot \beta_s$

  Cost can be reduced to
  \[
  \beta_r + (\beta_s/(M-2)) \cdot \beta_s + \text{cost of outputting final result}
  \]

  by using $M$ buffer pages instead of 1.

Block-Nested Loop Illustrated
Index-Nested Loop Join $r \bowtie_{A=\beta} s$

- Use an index on $s$ with search key $B$ (instead of scanning $s$) to find rows of $s$ that match $t_r$
  - Cost $= \beta_s + \tau_s \cdot \alpha + \text{cost of outputting final result}$
- Effective if number of rows of $s$ that match tuples in $r$ is small (i.e., $\alpha$ is small) and index is clustered

```
foreach tuple $t_r$ in $r$ do {
  use index to find all tuples $t_s$ in $s$ satisfying $t_r.A = t_s.B$
  output $(t_r, t_s)$
}
```

Sort-Merge Join $r \bowtie_{A=\beta} s$

```
sort $r$ on $A$;
sort $s$ on $B$;
while $\text{!eof}(r)$ and $\text{!eof}(s)$ do {
  Scan $r$ and $s$ concurrently until $t_r.A = t_s.B = c$;
  Output $\sigma_{A=c}(r) \bowtie_{A=c, B=c} (s)$
}
```

Join During Merge Illustrated

Cost of Sort-Merge Join

- Cost of sorting assuming $M$ buffers:
  $$2 \beta_r \log_M \beta_r + 2 \beta_s \log_M \beta_s$$
- Cost of merging:
  - Scanning $\sigma_{A=c}(r)$ and $\sigma_{B=c}(s)$ can be combined with the last step of sorting of $r$ and $s$ --- costs nothing
  - Cost of $\sigma_{A=c}(r) \bowtie_{A=c, B=c} (s)$ depends on whether $\sigma_{A=c}(r)$ can fit in the buffer
    - If yes, this step costs 0
    - In no, each $\sigma_{A=c}(r) \bowtie_{A=c, B=c} (s)$ is computed using block-nested join, so the cost is the cost of the join. (Think why indexed methods or sort-merge are inapplicable to Cartesian product.)
- Cost of outputting the final result depends on the size of the result

Hash-Join $r \bowtie_{A=\beta} s$

- Step 1: Hash $r$ on $A$ and $s$ on $B$ into the same set of buckets
- Step 2: Since matching tuples must be in same bucket, read each bucket in turn and output the result of the join
- Cost: $3 (\beta_r + \beta_s) + \text{cost of outputting final result}$
  - assuming each bucket fits in memory

Hash Join

Cost of Hash Join

- Cost of hashing:
  $$2 \beta_r \log_M \beta_r + 2 \beta_s \log_M \beta_s$$
- Cost of matching:
  - Scanning $\sigma_{A=c}(r)$ and $\sigma_{B=c}(s)$ can be combined with the last step of sorting of $r$ and $s$ --- costs nothing
  - Cost of $\sigma_{A=c}(r) \bowtie_{A=c, B=c} (s)$ depends on whether $\sigma_{A=c}(r)$ can fit in the buffer
    - If yes, this step costs 0
    - In no, each $\sigma_{A=c}(r) \bowtie_{A=c, B=c} (s)$ is computed using block-nested join, so the cost is the cost of the join. (Think why indexed methods or sort-merge are inapplicable to Cartesian product.)
- Cost of outputting the final result depends on the size of the result
Star Joins

- $r \bowtie_{cond_1} r_1 \bowtie_{cond_2} \cdots \bowtie_{cond_n} r_n$
  - Each $cond_i$ involves only the attributes of $r_i$ and $r$

Computing Star Joins

- **Use join index** (Chapter 11)
  - Scan $r$ and the join index \{<r,r_1,\ldots,r_n>\} (which is a set of tuples of rids) in one scan
  - Retrieve matching tuples in $r_1,\ldots,r_n$
  - Output result

Choosing Indices

- DBMSs may allow user to specify
  - Type (hash, B+ tree) and search key of index
  - Whether or not it should be clustered
- Using information about the frequency and type of queries and size of tables, designer can use cost estimates to choose appropriate indices
- Several commercial systems have tools that suggest indices
  - Simplifies job, but index suggestions must be verified

Computing Star Joins

- **Use bitmap indices** (Chapter 11)
  - Use one bitmapped join index, $J_i$, per each partial join $r \bowtie_{cond_i} r_i$
  - Recall: $J_i$ is a set of <v, bitmap>, where v is an rid of a tuple in $r_i$ and bitmap has 1 in k-th position iff k-th tuple of $r_i$ joins with the tuple pointed to by v
  1. Scan $J_i$ and logically OR all bitmaps. We get all rids in $r$ that join with $r_i$
  2. Now logically AND the resulting bitmaps for $J_1, \ldots, J_n$
  3. Result: a subset of $r$, which contains all tuples that can possibly be in the star join
    - **Rationale**: only a few such tuples survive, so can use indexed loops

Choosing Indices – Example

- If a frequently executed query that involves selection or a join and has a large result set, use a clustered B+ tree index
  - **Example**: Retrieve all rows of Transcript for StudId
- If a frequently executed query is an equality search and has a small result set, an unclustered hash index is best
  - Since only one clustered index on a table is possible, choosing unclustered allows a different index to be clustered
  - **Example**: Retrieve all rows of Transcript for (StudId, CrsCode)