Database Management Systems
Winter 2004

CMPUT 391: Query Processing: The Basics

Dr. Osmar R. Zaïane

University of Alberta

Chapter 13 of Textbook

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 \( \implies \) 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 = \( 2F \)
    (read the file and write the file \( \implies \) \( 2F \) I/Os)

Example: \( M = 2, F = 7 \)
**K-Way Merge 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 $M-1$

**Diagram:**

- Run 1
- Run 2
- Run M-1
- Input buffer 1
- Input buffer 2
- Output buffer
- M pages
- Buffer
- Output run

**K-Way Merge Algorithm**

- Cost of merge phase:
  - $(F/M)/(M-1)^k$ 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
    - Cost = # merge steps * $2F$ (R & W every page)
    - $= 2F \lceil \log_{M-1}(F/M) \rceil$
  - Total cost = cost of partial sort phase + cost of merge phase $2F + 2F \lceil \log_{M-1}(F/M) \rceil = 2F(1 + \lceil \log_{M-1}(F/M) \rceil)$

- **Simplification in text book (not to use in 391 tests)**
  - cost = $2F \lceil \log_{M-1}(F/M) \rceil = 2F(\log_{M-1}F - 1) = 2F \log_{M-1}F$

**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

**Sort-Based Projection**

- Algorithm:
  - Sort rows of relation at cost of $2F(1 + \lceil \log_{M-1}(F/M) \rceil)$
    - 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
Quick review ...

- At this point many decisions (regarding query processing/optimizations) will be depend on what type of index(es), if any, are available.
- Hence, a quick review is in order … (for details refer to Chapter 11 in the textbook).

B+ Tree

- Supports equality and range searches, multiple attribute keys and partial key searches
- Either a secondary index (in a separate file) or the basis for an integrated storage structure
  - Responds to dynamic changes in the table

B+ Tree Structure

- Leaf level is a (sorted) linked list of index entries
- Sibling pointers support range searches in spite of allocation and deallocation of leaf pages (but leaf pages might not be physically contiguos on disk)

Example B+ Tree
**Hash Index**

- Index entries partitioned into buckets in accordance with a hash function, \( h(v) \), where \( v \) ranges over search key values
  - Each bucket is identified by an address, \( a \)
  - Bucket at address \( a \) contains all index entries with search key \( v \) such that \( h(v) = a \)
- Each bucket is stored in a page (with possible overflow chain)
- If index entries contain rows, set of buckets forms an integrated storage structure; else set of buckets forms an (unclustered) secondary index

**Equality Search with Hash Index**

Given \( v \):
1. Compute \( h(v) \)
2. Fetch bucket at \( h(v) \)
3. Search bucket

There are techniques to avoid/minimize overflow (see textbook)

**Hash-Based Projection**

- **Phase 1:**
  - Input rows
  - Project out columns
  - Hash remaining columns using a hash function with range \( 1...M \cdot I \)
  - **Cost = 2F**
- **Phase 2:**
  - Sort each bucket to eliminate duplicates
  - **Cost (assuming a bucket fits in \( M \cdot I \) 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 \( (\log_2 F) \) to locate first data page containing row in which \( (attr = value) \)
    - Scan further to get all rows satisfying \( (attr \ op \ value) \)
    - **Cost = \( \log_2 F + (\text{cost of scan}) \)**
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
  - Scan entries and sort record lds to identify table data pages with qualifying rows
  - Any page that has at least one such row must be fetched once.

  **Cost: number of rows that satisfy selection condition**

---

**Unclustered B⁺ Tree Index**

- Index entries (containing row lds) that satisfy condition

- Data page

- $B^+$ Tree

---

**Computing Selection $\sigma_{(\text{attr} = \text{value})}$**

- **Hash index on $\text{attr}$** (for "$=$" search only):
  - Hash on $\text{value}$. **Cost $\approx 1.2$**
    - $1.2$ – 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)
      - **Cost: number of pages occupies by the bucket**
    - **Unclustered** index – sort row lds in the index entries to identify data pages with qualifying rows
      - Each page containing at least one such row must be fetched once
      - **Cost: number of rows in bucket**
Computing Selection $\sigma_{\text{attr} = \text{value}}$

- Unclustered hash index on attr (for equality search)

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 $\sigma_{a_1 > 5 \land a_2 = 3 \land a_3 = 'x'}(R)$: find first entry having $a_2 = 3 \land a_1 > 5 \land a_3 = 'x'$ 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**: $\sigma_{\text{CrsCode} = 'CS305' \land \text{Grade} = 'B'}$ (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
- *Block-nested loops* join algorithm for computing \( r \bowtie_s \)

```
foreach page \( p_r \) in \( r \) do
  foreach tuple \( t_r \) in \( p_r \) do
    foreach page \( p_s \) in \( s \) do
      foreach tuple \( t_s \) in \( p_s \) do
        if \( t_r.A = t_s.B \) then output \((t_r, t_s)\)
```

Block-Nested Loops Join

- If \( \beta_r \) and \( \beta_s \) are the number of pages in \( r \) and \( s \), the cost of algorithm is

\[
\text{cost} = \beta_r + \left( \frac{\beta_r}{M-2} \right) \times \beta_s + \text{cost of outputting final result}
\]

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

Index-Nested Loop Join \( r \bowtie_s \)

- Cost can be reduced to

\[
\beta_r + \left( \frac{\beta_r}{M-2} \right) \times \beta_s + \text{cost of outputting final result}
\]

by using \( M \) buffer pages instead of 1.

- Use an index on \( s \) with search key \( B \) (instead of scanning \( s \)) to find rows of \( s \) that match \( t_r \)
  - Cost = \( \beta_r + \tau_r \times \omega + \text{cost of outputting final result} \)
  - Effective if number of rows of \( s \) that match tuples in \( r \) is small (i.e., \( \omega \ll 1 \)) 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 \( r \bowtie_{A=B} s \)

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

- Each \( \text{cond}_i \) involves only the attributes of \( r_i \) and \( r \)

Computing Star Joins

- Use \textit{bitmap indices} (Chapter 11)
  - Use one bitmapped join index, \( J_i \), per each partial join

  \[ \begin{align*}
  r \quad & \circ \times \text{cond}_1 \quad r_1 \quad r_2 \quad r_3 \quad r_n \\
  \text{Scan } r \quad & \text{and the join index } \{<r_i, \ldots, r_n>\} \quad \text{(which is a set of tuples of rids)} \quad \text{in one scan} \\
  \text{Retrieve matching tuples in } r_1, \ldots, r_n \quad \text{Output result}
  \end{align*} \]

- \textit{Recall:} \( J_i \) is a set of \( <x, \text{bitmap}> \) where \( x \) is an id of a tuple in \( r_i \) and \( \text{bitmap} \) has 1 in \( k \)-th position if \( k \)-th tuple of \( r \) joins with the tuple pointed to by \( x \).

- Scan \( J_i \) and logically OR all bitmaps. We get all rids in \( r_i \) that join with \( r \).

- Now logically AND the resulting bitmaps for \( J_1, \ldots, J_n \).

- Result: a subset of \( r \), which contains all tuples that can possibly be in the star join.

- \textit{Remark:} only a few such tuples survive, so can use indexed loops.
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

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)