Query Processing: The Basics

Chapter 13

1

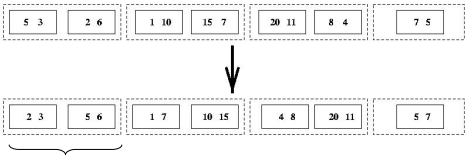
External Sorting

- Sorting is used in implementing many relational operations
- Sorting has two components (since records to be sorted generally do not fit in memory)
  - Computation involved in sorting records in buffers in main memory
  - I/O necessary to move records between mass store and main memory
- I/O costs dominate => cost of sorting algorithm measured in the number of page transfers

2

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*

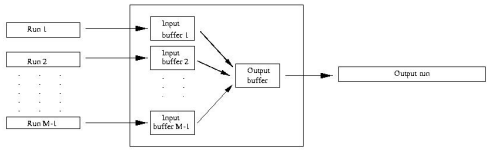


Example: *M = 2, F = 7*

3

Simple Sort Algorithm

- *M-1* way merge: merge a group of *M-1* runs using one buffer for output
- Merge step: divide runs into groups of size *M-1* and apply *M-1* way merge to each group, cost = 2*F*



4

Simple Sort Algortihm

- Merge Phase:
  - $(F/M)/(M-1)^k$ runs after *k* merge steps
  - $log_{M-1}(F/M)$ merge steps needed to merge an initial set of *F/M* sorted runs
  - cost = $2F\ log_{M-1}(F/M) \approx 2F(log_{M-1}F - 1)$
- Total cost = cost of partial sort phase + cost of merge phase $\approx 2F\ log_{M-1}F$

5

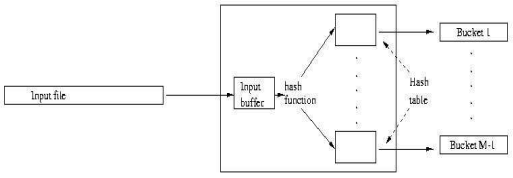
Duplicate Elimination

- Needed in computing projection and union (same technique for computing difference)
- Sort-based projection:
  - Sort rows of relation at cost of $2F\ log_{M-1}F$
  - Eliminate unwanted columns in partial sort phase (no cost)
  - Eliminate duplicates on completion of last merge step (no cost)

6

Duplicate Elimination

- Hash-based projection
  - Phase 1: input rows, project, hash remaining columns using an *M-1* output hash function, and create *M-1* buckets on disk. Cost = *2F*
  - Phase 2: input buckets one at a time, sort, eliminate duplicates, write result. Cost = *2F* (assuming each bucket fits in memory)
  - Total cost = *4F*



Computing Selection

condition: *(attr op value)*

- No index on attr:
  - Rows unsorted: Scan all data pages to find rows satisfying the condition.
    - Cost = *F* + (cost of ouput of selected rows)
  - Rows sorted on attr:
    - Use binary search to locate first data page containing row in which *(attr = value)* (if it exists)
    - Scan further to get all rows satisfying *(attr op value)*
    - Output scanned rows
    - Cost = $log_2 F + 2$(cost of scan)

8

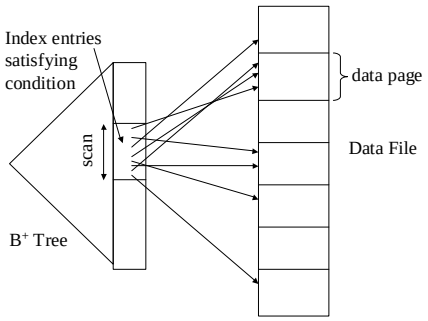
Computing Selection

condition: *(attr op value)*

- B^+ tree index on attr (for equality or range search):
  - Locate first leaf page that can contain a row in which *(attr = value)*.
  - Clustered index - rows satisfying condition packed in sequence in successive data pages; scan those pages and ouput selected rows.
    - Cost = depth of tree + 2(cost of scan)
  - Unclustered index - index entries with pointers to rows satisfying condition packed in sequence in successive index pages; scan entries and sort pointers to identify data pages with qualifying rows, each page (with at least one such row) fetched once.
    - Cost = depth of tree + cost of index scan + cost of fetching and outputting selected rows

9

Unclustered B^+ Tree Index



10

Computing Selection

condition: *(attr = value)*

- Hash index on attr (for equality search only):
  - Hash on value, search the (unique) bucket containing all index entries satisfying condition
    - Cost ≈ 1.2 (to account for possible overflow chain)
    - Clustered index - all qualifying rows packed in bucket.
    - Unclustered index - sort pointers in index entries to identify data pages with qualifying rows, each page (containing at least one such row) fetched once.

11

Access Path

- Algorithm + data structure used to select rows satisfying some condition.
  - File scan: used for any condition
  - Hash: used for equality search; *all* search key attributes of hash index are specified in condition
  - B^+ tree: used for 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 sequence is specified in condition

12

Access Paths Supported by B+ tree

- **Example**: Given a B+ tree whose search key is the sequence of attributes a2, a1, a3, a4
- Access path for search $\sigma_{a1>5 \wedge a2=3.0 \wedge a3='x'}$ *(R):*
  - Search for value *a2=3.0* ∧ *a1=5* ∧ *a3='x';* scan index entries from there until entry having *a2>3.0 ;* select satisfying entries; fetch rows (if index unclustered)
- Access path for search $\sigma_{a2=3.0 \wedge a3>'x'}$ *(R):*
  - Search for value *a2=3.0;* scan index entries from there until entry having *a2>3.0 ;* select satisfying entries; fetch rows (if index unclustered)
- No access path for search $\sigma_{a1>5 \wedge a3='x'}$ *(R)*

13

Choosing an Access Path

- **Selectivity** of an access path is its cost.
- If several access paths support a query, DBMS should choose the one with smallest selectivity
- The size of the domain of an attribute affects the selectivity of an access path.
  - **Example**: $\sigma_{CrsCode='CS305' \wedge Grade='B'}$ - a B+ tree with search key CrsCode is more selective than a B+ tree with search key Grade
- By creating appropriate indices (in the schema) the application programmer makes it possible for the DBMS to choose an efficient access path.

14

Computing Joins

- The cost of joining two relations makes the choice of a join algorithm crucial
- Block-nested loops join algorithm for computing r $\bowtie_{A=B}$ s

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

15

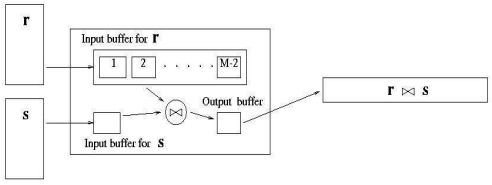
Block-Nested Loops Join

- If β_r and β_s are the number of pages in r and s, the cost of algorithm is $\beta_r + \beta_r * \beta_s$
  - If r and s have 10^3 pages each, cost is $10^3 + 10^3 * 10^3$
  - Choose smaller relation for outer loop

16

Block-Nested Loops Join

- Cost can be reduced to $\beta_r + (\beta_r/M\text{-}2) * \beta_s$ by using M buffers



17

Index-Nested Loop Join

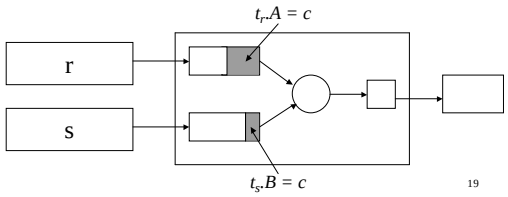
- Use an index on s with search key *B* (instead of scanning s) to find rows of s that match t_r
  - Cost = $\beta_r + \tau_r *$ (av'g cost of retrieving *all* rows in s that match a row in r)
    where τ_r is the number of rows in r
  - Effective if number of rows of s that match t_r is small and index is clustered; r should be the smaller relation

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) }

18

Sort-Merge Join

sort r on *A*;
sort s on *B;*
while !*eof*(r) and !*eof*(s) **do** {
  scan r and s concurrently until $t_r.A=t_s.B$;
  if $(t_r.A=t_s.B=c)$ {output $\sigma_{A=c} \times \sigma_{B=c}$ } }



19

Sort-Merge Join

- Cost of sorts assuming M buffers =
  $2\ \beta_r \log_{M-1} \beta_r + 2\ \beta_s \log_{M-1} \beta_s$
- Cost of merge depends on whether matching rows can be fit in buffers
  - If yes, merge step takes $\beta_r + \beta_s$

20

Hash-Join

- 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 (assuming each bucket fits in memory) $= 3\ (\beta_r + \beta_s)$

21

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

22

Choosing Indices - Example

- If a frequently executed query has a large result set, use a clustered B^+ tree index
  - Ex. 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
  - Ex. Retrieve all rows of Transcript for (StudId, CrsCode)

23