

# Query Processing: The Basics

## Chapter 13

1

---

---

---

---

---

---

---

---

## Topics

- How does DBMS compute the result of a SQL queries?
- The most often executed operations:
  - Sort
  - Projection, Union, Intersection , Set Difference
  - Selection
  - Join

2

---

---

---

---

---

---

---

---

## 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

3

---

---

---

---

---

---

---

---

## 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

4

---

---

---

---

---

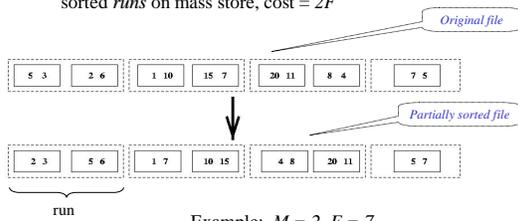
---

---

---

## 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 =  $2F$



5

---

---

---

---

---

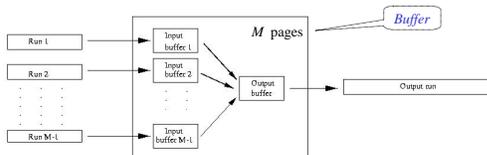
---

---

---

## 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  $M-1$*




---

---

---

---

---

---

---

---

## Simple Sort Algorithm

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

7

---

---

---

---

---

---

---

---

## 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

8

---

---

---

---

---

---

---

---

## Sort-Based Projection

- Algorithm:
  - Sort rows of relation at cost of  $2F \text{Log}_{M-1}F$
  - 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

9

---

---

---

---

---

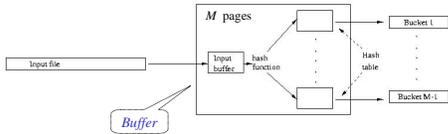
---

---

---

## Hash-Based Projection

- **Phase 1:**
  - Input rows
  - Project out columns
  - Hash remaining columns using a hash function with range  $1 \dots M-1$  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 $\sigma_{(attr \text{ op } value)}$

- **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 ( $attr = value$ )
    - Scan further to get all rows satisfying ( $attr \text{ op } value$ )
    - Cost =  $\log_2 F + (\text{cost of scan})$

11

---

---

---

---

---

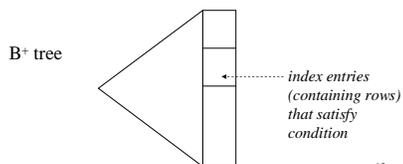
---

---

---

## Computing Selection $\sigma_{(attr \text{ op } value)}$

- **Clustered B<sup>+</sup> tree index on  $attr$**  (for “=” or range search):
    - Locate first index entry corresponding to a row in which ( $attr = value$ ). 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



12

---

---

---

---

---

---

---

---

## Computing Selection $\sigma_{(attr \text{ op } value)}$

- **Unclustered B<sup>+</sup> tree index on attr** (for “=” or range search):
  - Locate first index entry corresponding to a row in which (*attr* = *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 Ids 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

13

---

---

---

---

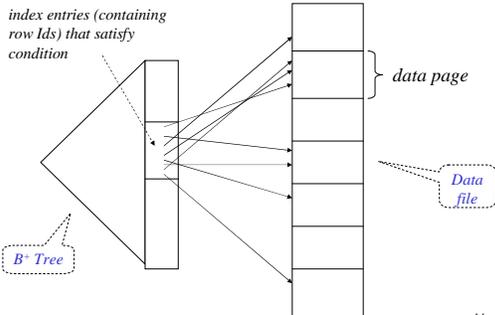
---

---

---

---

## Unclustered B<sup>+</sup> Tree Index



14

---

---

---

---

---

---

---

---

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

- **Hash index on attr** (for “=” search only):
  - Hash on *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 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: number of rows in bucket

15

---

---

---

---

---

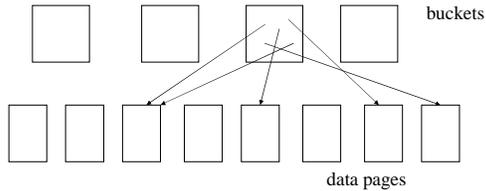
---

---

---

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

- Unclustered hash index on *attr* (for equality search)



16

---

---

---

---

---

---

---

---

## 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<sup>+</sup> tree:* equality or range search; a *prefix* of the search key attributes are specified in condition
    - B<sup>+</sup> 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

17

---

---

---

---

---

---

---

---

## Access Paths Supported by B<sup>+</sup> tree

- *Example:* Given a B<sup>+</sup> tree whose search key is the sequence of attributes *a2, a1, a3, a4*
  - Access path for search  $\sigma_{a1>5 \wedge a2=3 \wedge a3='x'}(R)$ : find first entry having  $a2=3 \wedge a1>5 \wedge a3='x'$  and scan leaves from there until entry having  $a2>3$  or  $a3 \neq 'x'$ . Select satisfying entries
  - Access path for search  $\sigma_{a2=3 \wedge a3>'x'}(R)$ : locate first entry having  $a2=3$  and scan leaves until entry having  $a2>3$ . Select satisfying entries
  - Access path for search  $\sigma_{a1>5 \wedge a3='x'}(R)$ : Scan of *R*

18

---

---

---

---

---

---

---

---

## 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:  $\sigma_{CrsCode='CS305' \wedge Grade='B'}$ 
  - a B<sup>+</sup> tree with search key *CrsCode* has lower selectivity than a B<sup>+</sup> tree with search key *Grade*

19

---

---

---

---

---

---

---

---

## 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 pr in r do
  foreach tuple tr in pr do
    foreach page ps in s do
      foreach tuple ts in ps do
        if tr.A = ts.B then output (tr, ts)
```

20

---

---

---

---

---

---

---

---

## Block-Nested Loops Join

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

$$\beta_r + \beta_r * \beta_s + \text{cost of outputting final result}$$

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

21

---

---

---

---

---

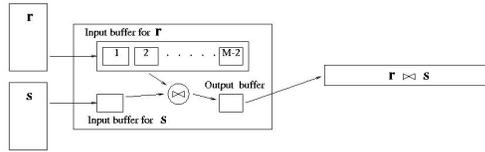
---

---

---

## Block-Nested Loops Join

- Cost can be reduced to  $\beta_r + (\beta_r / (M-2)) * \beta_s + \text{cost of outputting final result}$  by using M buffer pages instead of 1. Number of scans of relation s



22

---

---

---

---

---

---

---

---

---

---

## Index-Nested Loop Join $r \bowtie_{A=B} s$

- Use an index on s with search key B (instead of scanning s) to find rows of s that match  $t_r$ 
  - Cost =  $\beta_r + \tau_r * \omega + \text{cost of outputting final result}$

Number of rows in r

avg cost of retrieving all rows in s that match  $t_r$

- 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 ( $t_r, t_s$ )
}
```

23

---

---

---

---

---

---

---

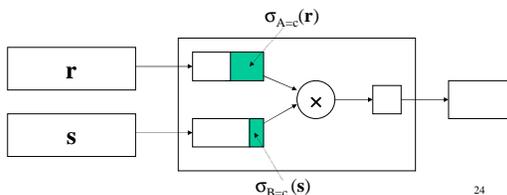
---

---

---

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

```
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  $\sigma_{A=c}(r) \times \sigma_{B=c}(s)$  }
}
```



24

---

---

---

---

---

---

---

---

---

---

## Sort-Merge Join

- Cost of *sorting* assuming  $M$  buffers:  
 $2 \beta_r \log_{M-1} \beta_r + 2 \beta_s \log_{M-1} \beta_s + \text{cost of final result}$
- Cost of *merging* depends on whether  $\sigma_{A=c}(r)$  and  $\sigma_{B=c}(s)$  can be fit in buffers
  - If yes, merge step takes  $\beta_r + \beta_s$
  - In no, then each  $\sigma_{A=c}(r) \times \sigma_{B=c}(s)$  has to be computed using *block-nested join*.  
 (Think why indexed methods or sort-merge are inapplicable.)

25

---

---

---

---

---

---

---

---

## Hash-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 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

26

---

---

---

---

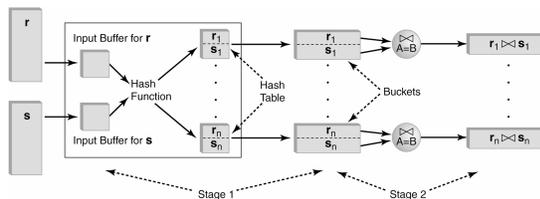
---

---

---

---

## Hash Join



27

---

---

---

---

---

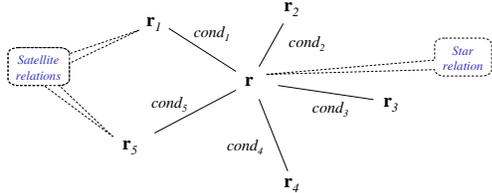
---

---

---

## Star Joins

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



28

---

---

---

---

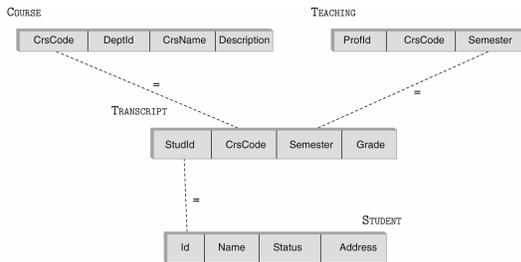
---

---

---

---

## Star Join



29

---

---

---

---

---

---

---

---

## Computing Star Joins

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

30

---

---

---

---

---

---

---

---

## Computing Star Joins

- Use *bitmap indices* (Chapter 11)
  - Use one bitmapped join index,  $J_i$ , per each partial join
    - $\mathbf{r} \bowtie_{cond} \mathbf{r}_i$
  - *Recall*:  $J_i$  is a set of  $\langle v, bitmap \rangle$ , where  $v$  is an rid of a tuple in  $\mathbf{r}_i$  and *bitmap* has 1 in  $k$ -th position iff  $k$ -th tuple of  $\mathbf{r}$  joins with the tuple pointed to by  $v$

1. Scan  $J_i$  and logically OR all bitmaps. We get all rids in  $\mathbf{r}$  that join with  $\mathbf{r}_i$
2. Now logically AND the resulting bitmaps for  $J_1, \dots, J_n$ .
3. Result: a subset of  $\mathbf{r}$ , which contains all tuples that can possibly be in the star join
  - *Rationale*: only a few such tuples survive, so can use indexed loops

31

---

---

---

---

---

---

---

---

## 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*

32

---

---

---

---

---

---

---

---

## 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*)

33

---

---

---

---

---

---

---

---