# Advanced Database Management Systems

Lecture 18

Query Processing: Select and Join

Chapter 15

# Algorithms for SELECT

- Consider only single table queries
- Three categories:
  - simple SELECT: one condition, no AND or OR
  - conjunctive select: multiple conditions, connected by AND
  - disjunctive select: multiple conditions, connected by OR

# Simple SELECT

- **Linear search** (brute force):
  - algorithm:
    - Retrieve every record in the file
    - test whether its attribute values satisfy the selection condition
  - works when:
    - always works
    - best on small files
    - only choice when no indexes or ordering
  - cost
    - average case: b/2, where b = # blocks in file
    - worst case: b

# Simple SELECT

- **Binary search:**
  - algorithm:
    - use binary search to find record(s)
  - works when:
    - selection condition is an equality test on an ordering attribute
  - cost:
    - $\log_2 b + \left\lceil \frac{s}{bfr} \right\rceil - 1$ , where b = # blocks in file, s = # selected records

# Simple SELECT

- **Primary index to retrieve a single record:**
  - algorithm:
    - look up record using primary index
  - works when:
    - selection condition is equality test on key attribute with primary index
  - cost:
    - x + 1, where x = # index levels

# Simple SELECT

- **Primary index to retrieve multiple records:**
  - algorithm:
    - use the index to find first record satisfying the corresponding equality condition
    - retrieve all matching subsequent records in the (ordered) file
  - works when:
    - selection condition is >, ≥, <, or ≤ on key field with primary index
  - cost:
    - $x + \left\lceil \frac{s}{bfr} \right\rceil$ , where x = # index levels, s = # selected records

# Simple SELECT

- **Clustering index to retrieve multiple records:**
  - algorithm:
    - use clustering index to retrieve all the records satisfying the selection condition
  - works when:
    - selection condition is equality comparison on a non-key attribute with clustering index
  - cost:
    - $x + \left\lceil \frac{s}{bfr} \right\rceil$, where x = # index levels, s = # selected records

# Simple SELECT

- **Secondary ($B^+$-tree) index:**
  - algorithm:
    - lookup first record through $B^+$-tree
    - scan leaves of $B^+$-tree for additional records
  - works when:
    - selection is an equality test or range query on attribute that has a secondary index
    - selection is a range query on attribute that has a secondary index
  - cost:
    - equality test: $x + s$ (worst case), $x + 1$ (key)
    - range query: $x + b_0/2 + r/2$

# Conjunctive SELECT

- **Conjunctive selection:**
  - algorithm:
    - use one of the simple SELECT algorithms to find records matching one condition
    - check those records for remaining conditions
  - works when:
    - conjunctive selection in which one of the simple SELECT algorithms can be applied to one condition
  - cost:
    - same as simple SELECT cost
  - example:
    - select * from EMPLOYEE where DNO=6 and salary>70000, and an index exists on DNO

# Conjunctive SELECT

- **Conjunctive selection using a composite index**
  - algorithm:
    - use composite index directly
  - works when:
    - selection condition is equality tests on two or more attributes for which a composite index exists
  - cost:
    - x + 1
  - example:
    - select * from EMPLOYEE where LNAME='Jones' and FNAME='Sam', and an index exists on <LNAME, FNAME>

# Conjunctive SELECT

- **Conjunctive selection by intersection of record pointers:**
  - algorithm:
    - use indexes to find record pointers for each equality condition
    - compute intersection of record pointer sets
    - retrieve records using records pointers
  - works when:
    - secondary indexes are available on all (or some of) the fields involved in equality comparison conditions
    - indexes include record pointers (rather than block pointers)
  - cost:
    - sum of index operations plus s, where s = #records selected
  - example:
    - select * from WORKS_ON where PNO=10 and LNAME='Wolfe', and indexes exist for both PNO and LNAME

# Disjunctive SELECT

- Disjunctive selects are much harder to optimize
  - no single condition can be used to ‘pre-filter’ the results
  - result is union of each condition
  - best you can do is to try to optimize each individual query, then compute the union
  - example:
    - select * from EMPLOYEE where DNO=3 or SALARY>80000 or SEX=‘F’

# JOIN Algorithms

- We'll consider joins such as $R \bowtie_{A=B} S$
- Extends to joins like $R \bowtie_{A=B \text{ and } C=D} S$
  by considering <A,C> and <B,D> as single attributes

# Algorithms for JOIN: $R \bowtie_{A=B} S$

- Nested loop (brute force)
  - algorithm:

```
for r in R
    for s in S
        if (r[A] = s[B])
            write < r, s >
```

  - Cost:
    - both tables and result can fit in memory:
      $b_R + b_S$, where = $b_R$ = #blocks in R and $b_S$ =#blocks in S
    - both tables and result cannot fit in memory:
      $b_R * b_S$, where = $b_R$ = #blocks in R and $b_S$ =#blocks in S

# Algorithms for JOIN: $R \bowtie_{A=B} S$

- Single loop
  - algorithm: use an access structure on S:

```
for r in R
    for s in σ r[A] = s[B] S
        write < r, s >
```

  - Cost:
    - $b_R$ * (x+1) , where = $b_R$ # blocks in R and x = levels in index on S

# Algorithms for JOIN: R $\bowtie_{A=B}$ S

- Sort-Merge
  - works when R is ordered on A and S is ordered on B
  - algorithm:
    - Scan both files in order of the join attributes,
      matching the records that have the same values for A and B
  - Cost:
    - $b_R + b_S$, where = $b_R$ = #blocks in R and $b_S$ = #blocks in S

# Algorithms for JOIN: R $\bowtie_{A=B}$ S

- Hash-Join
  - algorithm:
    - assume R is smaller table (without loss of generality)
    - Scan through smaller file, R, and hash records on A
    - Scan through other file, S, and hash records on B (using same hash function)
    - As S's records are hashed:
      if matching record from R is in the bucket, build and store the result tuple, otherwise

  - Cost:
    - $b_R + b_S$, where = $b_R$ = #blocks in R and $b_S$ = #blocks in S
    - works when R can fit in memory

# Algorithms for JOIN: R $\bowtie_{A=B}$ S

- **Hash-join:**
  - The records of files R and S are both hashed to the *same hash file*, using the *same hashing function* on the join attributes A of R and B of S as hash keys.
  - A single pass through the file with fewer records (say, R) hashes its records to the hash file buckets.
  - A single pass through the other file (S) then hashes each of its records to the appropriate bucket, where the record is combined with all matching records from R.

# JOIN Cost Functions

| | |
|---|---|
| nested-loop | $b_R + (b_R * b_S) + ((js * \|R\| * \|S\|) / bfr_{RS})$ |
| single-loop | using secondary index<br>$b_R + (\|R\| * (x_B + s_B)) + ((js * \|R\| * \|S\|) / bfr_{RS})$<br>using primary index<br>$b_R + (\|R\| * (x_B + 1)) + ((js * \|R\| * \|S\|) / bfr_{RS})$ |
| sort-merge | $b_R + b_S + ((js * \|R\| * \|S\|) / bfr_{RS})$ |

js: join selectivity = ratio of join size vs. cross-product size

$(js * |R| * |S|) / bfr_{RS}$ : cost to write result back to disk