



Lecture 28 of 42

Query Processing: Fast Joins Discussion: MP6 and PS7

Monday, 30 October 2006

William H. Hsu
Department of Computing and Information Sciences, KSU

KSOL course page: <http://snipurl.com/va60>

Course web site: <http://www.kddresearch.org/Courses/Fall-2006/CIS560>

Instructor home page: <http://www.cis.ksu.edu/~bhsu>

Reading for Next Class:

Second half of Chapter 13, Silberschatz *et al.*, 5th edition



Chapter 12: Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B⁺-Tree Index Files
- B-Tree Index Files
- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Index Definition in SQL
- Multiple-Key Access





Multiple-Key Access

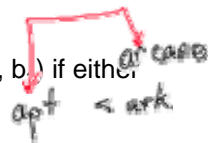
- Use multiple indices for certain types of queries.
- Example:

```
select account_number
from account
where branch_name = "Perryridge" and balance = 1000
```
- Possible strategies for processing query using indices on single attributes:
 1. Use index on *branch_name* to find accounts with balances of \$1000; test *branch_name* = "Perryridge".
 2. Use index on *balance* to find accounts with balances of \$1000; test *branch_name* = "Perryridge".
 3. Use *branch_name* index to find pointers to all records pertaining to the Perryridge branch. Similarly use index on *balance*. Take intersection of both sets of pointers obtained.



Indices on Multiple Keys

- **Composite search keys** are search keys containing more than one attribute
 - * E.g. (*branch_name*, *balance*)
- Lexicographic ordering: $(a_1, a_2) < (b_1, b_2)$ if either:
 - * $a_1 < b_1$, or
 - * $a_1 = b_1$ and $a_2 < b_2$





Indices on Multiple Attributes

Suppose we have an index on combined search-key
(*branch_name*, *balance*).

- With the **where** clause
where *branch_name* = "Perryridge" and *balance* = 1000
the index on (*branch_name*, *balance*) can be used to fetch only records that satisfy both conditions.
 - * Using separate indices is less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions.
- Can also efficiently handle
where *branch_name* = "Perryridge" and *balance* < 1000
- But cannot efficiently handle
where *branch_name* < "Perryridge" and *balance* = 1000
 - * May fetch many records that satisfy the first but not the second



Non-Unique Search Keys

- Alternatives:
 - * Buckets on separate block (bad idea)
 - * List of tuple pointers with each key
 - ⇒ Extra code to handle long lists
 - ⇒ Deletion of a tuple can be expensive
 - ⇒ Low space overhead, no extra cost for queries
 - * Make search key unique by adding a record-identifier
 - ⇒ Extra storage overhead for keys
 - ⇒ Simpler code for insertion/deletion
 - ⇒ Widely used



Other Issues

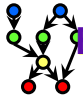
- Covering indices
 - * Add extra attributes to index so (some) queries can avoid fetching the actual records
 - ⇒ Particularly useful for secondary indices
 - ◆ Why?
 - * Can store extra attributes only at leaf
- Record relocation and secondary indices
 - * If a record moves, all secondary indices that store record pointers have to be updated
 - * Node splits in B⁺-tree file organizations become very expensive
 - * Solution: use primary-index search key instead of pointer in secondary index
 - ⇒ Extra traversal of primary index to locate record
 - ◆ Higher cost for queries, but node splits are cheap
 - ⇒ Add record-id if primary-index search key is non-unique



Static Hashing

- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function**.
- Hash function h is a function from the set of all search-key values K to the set of all bucket addresses B .
- Hash function is used to locate records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.





Example of Hash File Organization (Cont.)

Hash file organization of *account* file, using *branch_name* as key
(See figure in next slide.)

- There are 10 buckets,
- The binary representation of the *i*th character is assumed to be the integer *i*.
- The hash function returns the sum of the binary representations of the characters modulo 10
 - * E.g. $h(\text{Perryridge}) = 5$ $h(\text{Round Hill}) = 3$ $h(\text{Brighton}) = 3$



Example of Hash File Organization

Hash file organization of *account* file, using *branch_name* as key
(see previous slide for details).

bucket 0			bucket 5		
			A-102	Perryridge	400
			A-201	Perryridge	900
			A-218	Perryridge	700
bucket 1			bucket 6		
bucket 2			bucket 7		
			A-215	Mianus	700
bucket 3			bucket 8		
A-217	Brighton	750	A-101	Downtown	500
A-305	Round Hill	350	A-110	Downtown	600
bucket 4			bucket 9		
A-222	Redwood	700			





Hash Functions

- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
- Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.
- Typical hash functions perform computation on the internal binary representation of the search-key.
 - * For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned. .



Handling of Bucket Overflows

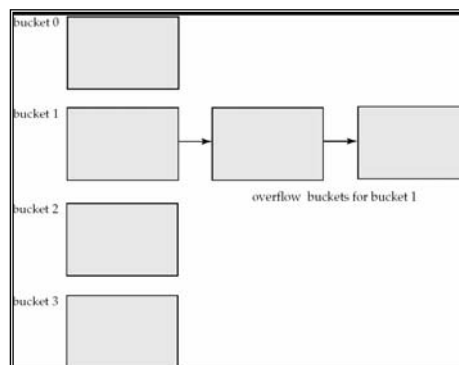
- Bucket overflow can occur because of
 - * Insufficient buckets
 - * Skew in distribution of records. This can occur due to two reasons:
 - ⇒ multiple records have same search-key value
 - ⇒ chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using *overflow buckets*.





Handling of Bucket Overflows (Cont.)

- Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called closed hashing.
 - * An alternative, called open hashing, which does not use overflow buckets, is not suitable for database applications.

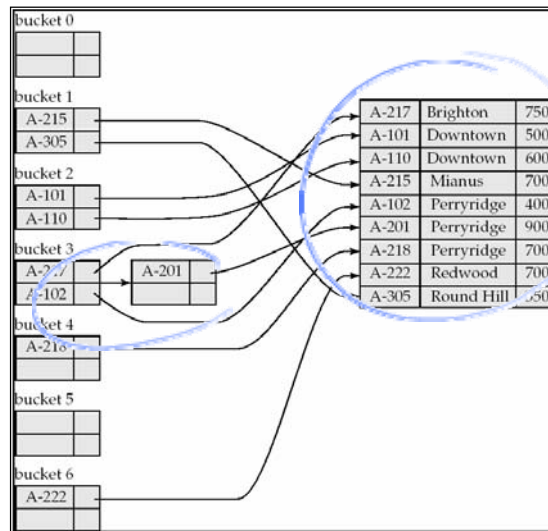


Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation.
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- Strictly speaking, hash indices are always secondary indices
 - * if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
 - * However, we use the term hash index to refer to both secondary index structures and hash organized files.



Example of Hash Index



Deficiencies of Static Hashing

- In static hashing, function h maps search-key values to a fixed set of B of bucket addresses.
 - * Databases grow with time. If initial number of buckets is too small, performance will degrade due to too much overflows.
 - * If file size at some point in the future is anticipated and number of buckets allocated accordingly, significant amount of space will be wasted initially.
 - * If database shrinks, again space will be wasted.
 - * One option is periodic re-organization of the file with a new hash function, but it is very expensive.
- These problems can be avoided by using techniques that allow the number of buckets to be modified dynamically.

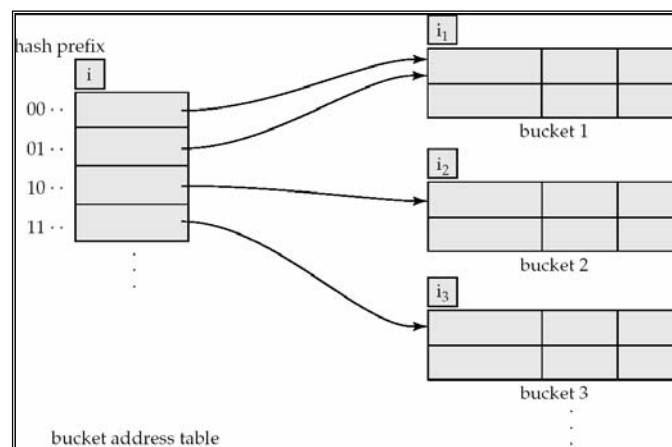


Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- **Extendable hashing** – one form of dynamic hashing
 - * Hash function generates values over a large range — typically b -bit integers, with $b = 32$.
 - * At any time use only a prefix of the hash function to index into a table of bucket addresses.
 - * Let the length of the prefix be i bits, $0 \leq i \leq 32$.
 - * Bucket address table size = 2^i . Initially $i = 0$
 - * Value of i grows and shrinks as the size of the database grows and shrinks.
 - * Multiple entries in the bucket address table may point to a bucket.
 - * Thus, actual number of buckets is $< 2^i$
 - ⇒ The number of buckets also changes dynamically due to coalescing and splitting of buckets.



General Extendable Hash Structure



In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$ (see next slide for details)





Use of Extendable Hash Structure

- Each bucket j stores a value i_j ; all the entries that point to the same bucket have the same values on the first i_j bits.
- To locate the bucket containing search-key K_j :
 1. Compute $h(K_j) = X$
 2. Use the first i high order bits of X as a displacement into bucket address table, and follow the pointer to appropriate bucket
- To insert a record with search-key value K_j
 - * follow same procedure as look-up and locate the bucket, say j .
 - * If there is room in the bucket j insert record in the bucket.
 - * Else the bucket must be split and insertion re-attempted (next slide.)
 - ⇒ Overflow buckets used instead in some cases (will see shortly)



Updates in Extendable Hash Structure

To split a bucket j when inserting record with search-key value K_j :

- If $i > i_j$ (more than one pointer to bucket j)
 - * allocate a new bucket z , and set i_j and i_z to the old $i_j + 1$.
 - * make the second half of the bucket address table entries pointing to j to point to z
 - * remove and reinsert each record in bucket j .
 - * recompute new bucket for K_j and insert record in the bucket (further splitting is required if the bucket is still full)
- If $i = i_j$ (only one pointer to bucket j)
 - * increment i and double the size of the bucket address table.
 - * replace each entry in the table by two entries that point to the same bucket.
 - * recompute new bucket address table entry for K_j
Now $i > i_j$ so use the first case above.





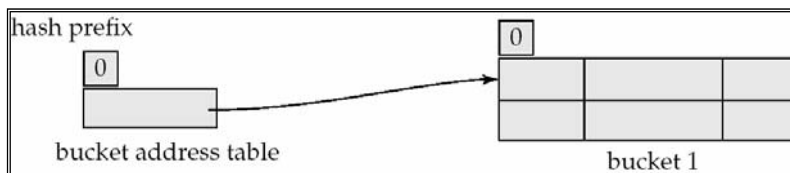
Updates in Extendable Hash Structure (Cont.)

- When inserting a value, if the bucket is full after several splits (that is, i reaches some limit b) create an overflow bucket instead of splitting bucket entry table further.
- To delete a key value,
 - * locate it in its bucket and remove it.
 - * The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
 - * Coalescing of buckets can be done (can coalesce only with a "buddy" bucket having same value of i_j and same $i_j - 1$ prefix, if it is present)
 - * Decreasing bucket address table size is also possible
 - ⇒ Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table



Use of Extendable Hash Structure: Example

<i>branch_name</i>	$h(\text{branch_name})$
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001



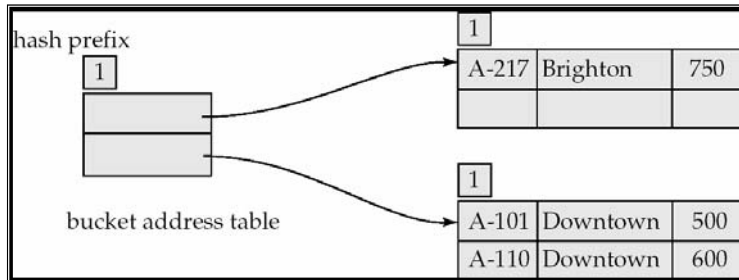
Initial Hash structure, bucket size = 2





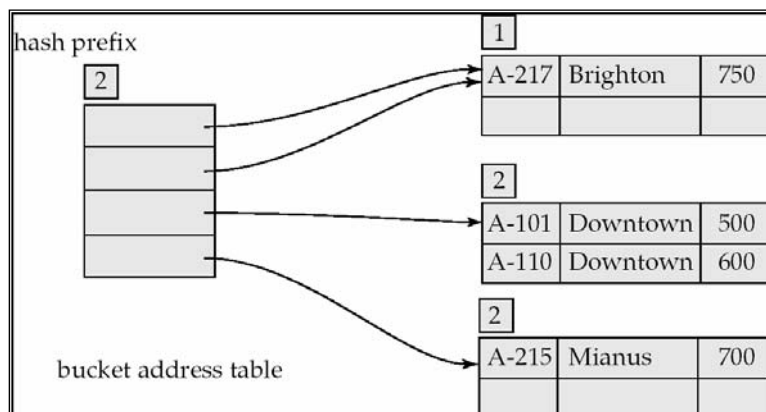
Example (Cont.)

- Hash structure after insertion of one Brighton and two Downtown records



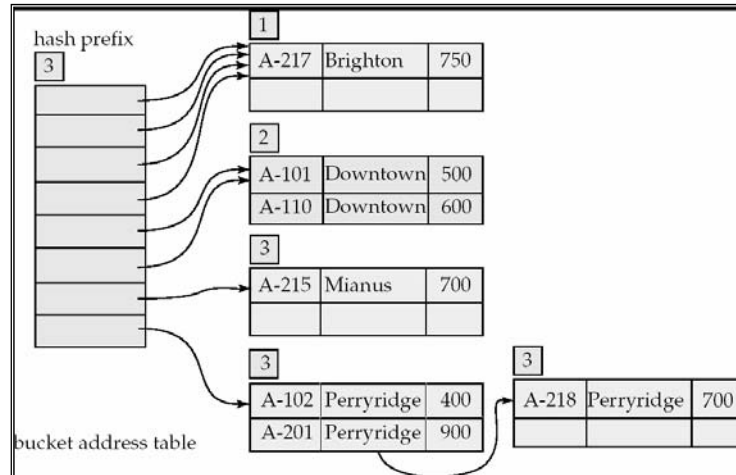
Example (Cont.)

Hash structure after insertion of Mianus record





Example (Cont.)

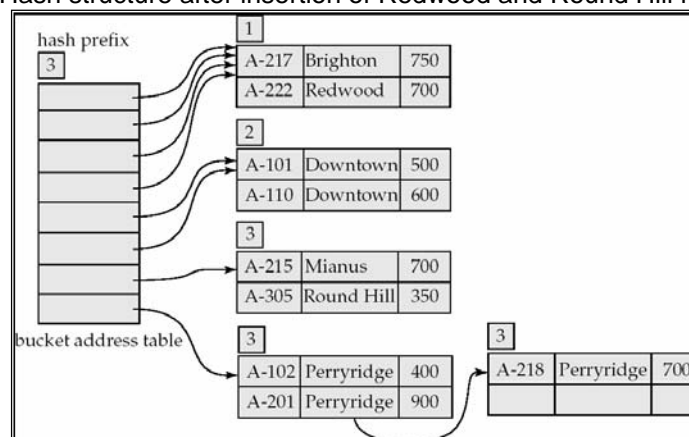


Hash structure after insertion of three Perryridge records



Example (Cont.)

- Hash structure after insertion of Redwood and Round Hill records





Extendable Hashing vs. Other Schemes

- Benefits of extendable hashing:
 - * Hash performance does not degrade with growth of file
 - * Minimal space overhead
- Disadvantages of extendable hashing
 - * Extra level of indirection to find desired record
 - * Bucket address table may itself become very big (larger than memory)
 - ⇒ Need a tree structure to locate desired record in the structure!
 - * Changing size of bucket address table is an expensive operation
- Linear hashing is an alternative mechanism which avoids these disadvantages at the possible cost of more bucket overflows



Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
 - * Hashing is generally better at retrieving records having a specified value of the key.
 - * If range queries are common, ordered indices are to be preferred





Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered sequentially from, say, 0
 - * Given a number n it must be easy to retrieve record n
 - ⇒ Particularly easy if records are of fixed size
- Applicable on attributes that take on a relatively small number of distinct values
 - * E.g. gender, country, state, ...
 - * E.g. income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
- A bitmap is simply an array of bits



Bitmap Indices (Cont.)

- In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute
 - * Bitmap has as many bits as records
 - * In a bitmap for value v , the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

record number	name	gender	address	income_level	Bitmaps for gender		Bitmaps for income_level					
					m	f	L1	L2	L3	L4	L5	
0	John	m	Perryridge	L1	1	0	1	0	1	0	0	0
1	Diana	f	Brooklyn	L2	0	1	0	1	0	0	0	0
2	Mary	f	Jonestown	L1	0	1	0	0	0	0	0	1
3	Peter	m	Brooklyn	L4	1	0	0	0	0	1	0	0
4	Kathy	f	Perryridge	L3	0	1	0	0	0	0	0	0





Bitmap Indices (Cont.)

- Bitmap indices are useful for queries on multiple attributes
 - * not particularly useful for single attribute queries
- Queries are answered using bitmap operations
 - * Intersection (and)
 - * Union (or)
 - * Complementation (not)
- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
 - * E.g. $100110 \text{ AND } 110011 = 100010$
 $100110 \text{ OR } 110011 = 110111$
 $\text{NOT } 100110 = 011001$
 - * Males with income level L1: $10010 \text{ AND } 10100 = 10000$
 - ⇒ Can then retrieve required tuples.
 - ⇒ Counting number of matching tuples is even faster



Bitmap Indices (Cont.)

- Bitmap indices generally very small compared with relation size
 - * E.g. if record is 100 bytes, space for a single bitmap is 1/800 of space used by relation.
 - ⇒ If number of distinct attribute values is 8, bitmap is only 1% of relation size
- Deletion needs to be handled properly
 - * Existence bitmap to note if there is a valid record at a record location
 - * Needed for complementation
 - ⇒ $\text{not}(A=v)$: $(\text{NOT bitmap-}A-v) \text{ AND ExistenceBitmap}$
- Should keep bitmaps for all values, even null value
 - * To correctly handle SQL null semantics for $\text{NOT}(A=v)$:
 - ⇒ intersect above result with $(\text{NOT bitmap-}A\text{-Null})$





Efficient Implementation of Bitmap Operations

- Bitmaps are packed into words; a single word and (a basic CPU instruction) computes and of 32 or 64 bits at once
 - * E.g. 1-million-bit maps can be anded with just 31,250 instruction
- Counting number of 1s can be done fast by a trick:
 - * Use each byte to index into a precomputed array of 256 elements each storing the count of 1s in the binary representation
 - ⇒ Can use pairs of bytes to speed up further at a higher memory cost
 - * Add up the retrieved counts
- Bitmaps can be used instead of Tuple-ID lists at leaf levels of B⁺-trees, for values that have a large number of matching records
 - * Worthwhile if > 1/64 of the records have that value, assuming a tuple-id is 64 bits
 - * Above technique merges benefits of bitmap and B⁺-tree indices



Index Definition in SQL

- Create an index

```
create index <index-name> on <relation-name>
      (<attribute-list>)
```

E.g.: **create index** *b-index* on *branch(branch_name)*
- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key is a candidate key.
 - * Not really required if SQL **unique** integrity constraint is supported
- To drop an index

```
drop index <index-name>
```



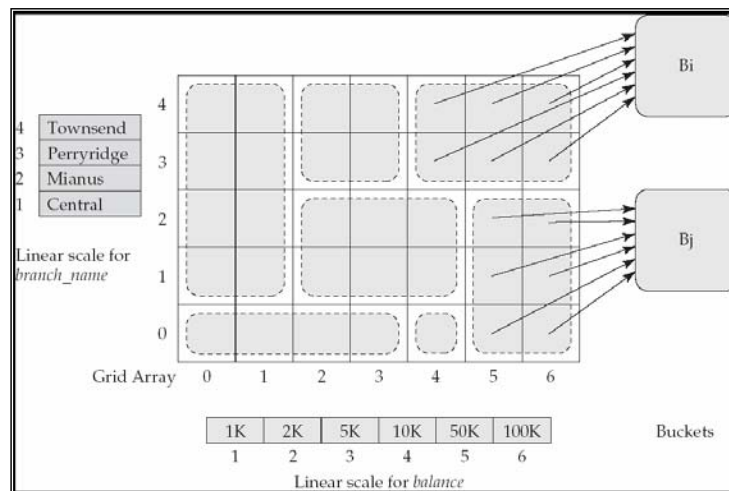


Grid Files

- Structure used to speed the processing of general multiple search-key queries involving one or more comparison operators.
- The grid file has a single grid array and one linear scale for each search-key attribute. The grid array has number of dimensions equal to number of search-key attributes.
- Multiple cells of grid array can point to same bucket
- To find the bucket for a search-key value, locate the row and column of its cell using the linear scales and follow pointer



Example Grid File for *account*





Queries on a Grid File

- A grid file on two attributes A and B can handle queries of all following forms with reasonable efficiency
 - * $(a_1 \leq A \leq a_2)$
 - * $(b_1 \leq B \leq b_2)$
 - * $(a_1 \leq A \leq a_2 \wedge b_1 \leq B \leq b_2)$.
- E.g., to answer $(a_1 \leq A \leq a_2 \wedge b_1 \leq B \leq b_2)$, use linear scales to find corresponding candidate grid array cells, and look up all the buckets pointed to from those cells.



Grid Files (Cont.)

- During insertion, if a bucket becomes full, new bucket can be created if more than one cell points to it.
 - * Idea similar to extendable hashing, but on multiple dimensions
 - * If only one cell points to it, either an overflow bucket must be created or the grid size must be increased
- Linear scales must be chosen to uniformly distribute records across cells.
 - * Otherwise there will be too many overflow buckets.
- Periodic re-organization to increase grid size will help.
 - * But reorganization can be very expensive.
- Space overhead of grid array can be high.
- R-trees (Chapter 23) are an alternative





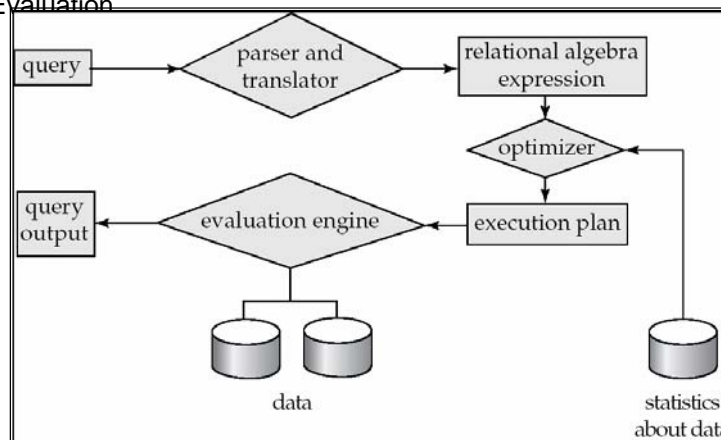
Chapter 13: Query Processing

- Overview
- Measures of Query Cost
- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions



Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation





Basic Steps in Query Processing (Cont.)

- Parsing and translation
 - * translate the query into its internal form. This is then translated into relational algebra.
 - * Parser checks syntax, verifies relations
- Evaluation
 - * The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.



Basic Steps in Query Processing : Optimization

- A relational algebra expression may have many equivalent expressions
 - * E.g., $\sigma_{balance < 2500}(\Pi_{balance}(account))$ is equivalent to $\Pi_{balance}(\sigma_{balance < 2500}(account))$
- Each relational algebra operation can be evaluated using one of several different algorithms
 - * Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.
 - * E.g., can use an index on *balance* to find accounts with balance < 2500,
 - * or can perform complete relation scan and discard accounts with balance ≥ 2500





Basic Steps: Optimization (Cont.)

- **Query Optimization:** Amongst all equivalent evaluation plans choose the one with lowest cost.
 - * Cost is estimated using statistical information from the database catalog
 - ⇒ e.g. number of tuples in each relation, size of tuples, etc.
- In this chapter we study
 - * How to measure query costs
 - * Algorithms for evaluating relational algebra operations
 - * How to combine algorithms for individual operations in order to evaluate a complete expression
- In Chapter 14
 - * We study how to optimize queries, that is, how to find an evaluation plan with lowest estimated cost



Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query
 - * Many factors contribute to time cost
 - ⇒ *disk accesses, CPU, or even network communication*
- Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account
 - * Number of seeks * average-*seek-cost*
 - * Number of blocks read * average-*block-read-cost*
 - * Number of blocks written * average-*block-write-cost*
 - ⇒ Cost to write a block is greater than cost to read a block
 - ◆ data is read back after being written to ensure that the write was successful





Measures of Query Cost (Cont.)

- For simplicity we just use the *number of block transfers from disk and the number of seeks* as the cost measures
 - * t_T – time to transfer one block
 - * t_S – time for one seek
 - * Cost for b block transfers plus S seeks

$$b * t_T + S * t_S$$
- We ignore CPU costs for simplicity
 - * Real systems do take CPU cost into account
- We do not include cost to writing output to disk in our cost formulae
- Several algorithms can reduce disk IO by using extra buffer space
 - * Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution
 - ⇒ We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available
- Required data may be buffer resident already, avoiding disk I/O
 - * But hard to take into account for cost estimation



Selection Operation

- **File scan** – search algorithms that locate and retrieve records that fulfill a selection condition.
- Algorithm **A1** (*linear search*). Scan each file block and test all records to see whether they satisfy the selection condition.
 - * Cost estimate = b_r block transfers + 1 seek
 - ⇒ b_r denotes number of blocks containing records from relation r
 - * If selection is on a key attribute, can stop on finding record
 - ⇒ cost = $(b_r/2)$ block transfers + 1 seek
 - * Linear search can be applied regardless of
 - ⇒ selection condition or
 - ⇒ ordering of records in the file, or
 - ⇒ availability of indices





Selection Operation (Cont.)

- **A2 (binary search).** Applicable if selection is an equality comparison on the attribute on which file is ordered.
 - * Assume that the blocks of a relation are stored contiguously
 - * Cost estimate (number of disk blocks to be scanned):
 - ⇒ cost of locating the first tuple by a binary search on the blocks
 - ◆ $\lceil \log_2(b_r) \rceil * (t_T + t_S)$
 - ⇒ If there are multiple records satisfying selection
 - ◆ Add transfer cost of the number of blocks containing records that satisfy selection condition
 - ◆ Will see how to estimate this cost in Chapter 14



Selections Using Indices

- **Index scan** – search algorithms that use an index
 - * selection condition must be on search-key of index.
- **A3 (primary index on candidate key, equality).** Retrieve a single record that satisfies the corresponding equality condition
 - * $Cost = (h_i + 1) * (t_T + t_S)$
- **A4 (primary index on nonkey, equality)** Retrieve multiple records.
 - * Records will be on consecutive blocks
 - ⇒ Let b = number of blocks containing matching records
 - * $Cost = h_i * (t_T + t_S) + t_S + t_T * b$
- **A5 (equality on search-key of secondary index).**
 - * Retrieve a single record if the search-key is a candidate key
 - ⇒ $Cost = (h_i + 1) * (t_T + t_S)$
 - * Retrieve multiple records if search-key is not a candidate key
 - ⇒ each of n matching records may be on a different block
 - ⇒ $Cost = (h_i + n) * (t_T + t_S)$
 - ◆ Can be very expensive!





Selections Involving Comparisons

- Can implement selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by using
 - * a linear file scan or binary search,
 - * or by using indices in the following ways:
- **A6 (primary index, comparison).** (Relation is sorted on A)
 - ⇒ For $\sigma_{A \geq V}(r)$ use index to find first tuple $\geq v$ and scan relation sequentially from there
 - ⇒ For $\sigma_{A \leq V}(r)$ just scan relation sequentially till first tuple $> v$; do not use index
- **A7 (secondary index, comparison).**
 - ⇒ For $\sigma_{A \geq V}(r)$ use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records.
 - ⇒ For $\sigma_{A \leq V}(r)$ just scan leaf pages of index finding pointers to records, till first entry $> v$
 - ⇒ In either case, retrieve records that are pointed to
 - ◆ requires an I/O for each record
 - ◆ Linear file scan may be cheaper

