DBMS Internals
How does it all work?
May 3rd, 2004

What Should a DBMS Do?
- Store large amounts of data
- Process queries efficiently
- Allow multiple users to access the database concurrently and safely.
- Provide durability of the data.
- How will we do all this??

Main Points to Take Away
- IO model of computation
  - We only count accesses to disk.
- Indexing:
  - Basic techniques: B+-tree, hash indexes
  - Secondary indexes.
- Efficient operator implementations: join
- Optimization: from what to how.

Agenda
- Comments on phase 2 of the project
- HW 3 is out.
- Today: DBMS internals part 1 --
  - Indexing
  - Query execution
- Next week: query optimization.

Generic Architecture
- User/Application
- Query compiler/optimizer
- Execution engine
- Index/record manager
- Buffer manager
- Storage manager
- Transaction manager:
  - Concurrency control
  - Logging/recovery

The Memory Hierarchy
- Main Memory
  - Cache:
    - Access time ~10 nano's
  - Disk:
    - 5-10 MB/s transfer rate
    - 3 igs of storage
    - Average time to access a block: 10-15 m secs.
  - Tape:
    - 1.5 MB/s transfer rate
    - 280 GB typical capacity
    - Only sequential access
    - Not for operational data
Main Memory

- Fastest, most expensive
- Today: 512MB-2GB are common on PCs
- Many databases could fit in memory
  - New industry trend: Main Memory Database
    - E.g. Times Ten
- Main issue is volatility

Secondary Storage

- Disks
- Slower, cheaper than main memory
- Persistent!!!
- Used with main memory buffer

Buffer Management in a DBMS

Data must be in RAM for DBMS to operate on it!
- Table of <frame#,pageid> pairs is maintained.
- LRU is not always good.

Buffer Manager

Manages buffer pool: the pool provides space for a limited number of pages from disk.

Needs to decide on page replacement policy.

Enables the higher levels of the DBMS to assume that the needed data is in main memory.

Why not use the Operating System for the task??

- DBMS may be able to anticipate access patterns
- Hence, may also be able to perform prefetching
- DBMS needs the ability to force pages to disk.

Tertiary Storage

- Tapes or optical disks
- Extremely slow: used for long termarchiving only

The Mechanics of Disk

- Mechanical characteristics:
  - Rotation speed (5400 RPM)
  - Number of platters (0-30)
  - Number of tracks (<10000)
  - Number of bytes/track (10^9)
Disk Access Characteristics

- Disk latency = time between when command is issued and when data is in memory
- Is not following Moore's Law!
- Disk latency = seek time + rotational latency
  - Seek time = time for the head to reach cylinder
    - 10ms - 40ms
  - Rotational latency = time for the sector to rotate
    - Rotation time = 10ms
  - Average latency = 10ms
- Transfer time = typically 10MB/s
- Disks read/write one block at a time (typically 4KB)

The I/O Model of Computation

- In main memory algorithms we care about CPU time
- In databases time is dominated by I/O cost
- Assumption: cost is given only by I/O
- Consequence: need to redesign certain algorithms
- We'll illustrate here with sorting

Sorting

- Illustrates the difference in algorithm design when your data is not in main memory:
  - Problem: sort 1GB of data with 1MB of RAM
  - Arises in many places in database systems:
    - Data requested in sorted order (ORDER BY)
    - Needed for grouping operations
    - First step in sort-merge join algorithm
    - Duplicate removal
    - Bulk loading of B+-tree indexes.

2-Way Merge-sort:

- Requires 3 Buffers
  - Pass 1: Read a page, sort it, write it.
    - only one buffer page is used
  - Pass 2, 3, ..., etc.:
    - three buffer pages used.

Two-Way External Merge Sort

- Each pass we read + write each page in file.
- N pages in file => the number of passes
- So total cost is:
  \[ T = \lceil \log_2 N \rceil + 1 \]
- In passes are started with larger size
- Sort 1GB with 1MB main memory in 10 passes
- \[ 2N \left\lceil \log_2 N \right\rceil + 1 \]

Can We Do Better?

- We have more main memory
- Should use it to improve performance
Cost Model for Our Analysis

- B: Block size
- M: Size of main memory
- N: Number of records in the file
- R: Size of one record

External Merge-Sort

- Phase one: load M bytes in memory, sort
  - Result: runs of length M/R records

Phase Two

- Merge M/B - 1 runs into a new run
- Result: runs have now M/R(M/B - 1) records

Phase Three

- Merge M/B - 1 runs into a new run
- Result: runs have now M/R(M/B - 1)^2 records

Cost of External Merge Sort

- Number of passes: \(1 + \lceil \log_{M/B} \left( \frac{N}{M} \right) \rceil\)
- Think differently
  - Given B = 4KB, M = 64MB, R = 0.1KB
    - Pass 1: runs of length M/R = 64000
      - Have now sorted runs of 64000 records
    - Pass 2: runs increase by a factor of M/B - 1 = 16000
      - Have now sorted runs of 1024000000 = 10^{10} records
    - Pass 3: runs increase by a factor of M/B - 1 = 16000
      - Have now sorted runs of 10^{12} records
  - Nobody has so much data!
  - Can sort everything in 2 or 3 passes!

Number of Passes of External Sort

<table>
<thead>
<tr>
<th>N</th>
<th>B=3</th>
<th>B=5</th>
<th>B=9</th>
<th>B=17</th>
<th>B=129</th>
<th>B=257</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1,000</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10,000</td>
<td>13</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>100,000</td>
<td>17</td>
<td>9</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1,000,000</td>
<td>20</td>
<td>10</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>10,000,000</td>
<td>23</td>
<td>12</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>100,000,000</td>
<td>26</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>30</td>
<td>15</td>
<td>10</td>
<td>8</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

B: number of frames in the buffer pool; N: number of pages in relation.
**Data Storage and Indexing**

**Representing Data Elements**

- **Relational database elements:**
  ```sql
  CREATE TABLE Product (
    pid INT PRIMARY KEY,
    name CHAR(20),
    description VARCHAR(200),
    maker CHAR(10) REFERENCES Company(name)
  )
  ```
  - A tuple is represented as a record.

**Record Formats: Fixed Length**

- Information about field types same for all records in a file; stored in system catalogs.
- Finding i'th field requires scan of record.
- Note the importance of schema information!

**Record Header**

- The header is needed because:
  - The schema may change for a while; new and old may coexist.
  - Records from different relations may coexist.

**Variable Length Records**

- Place the fixed fields first: F1, F2.
- Then the variable length fields: F3, F4.
- Null values take 2 bytes only.
- Sometimes they take 0 bytes (when at the end).

**Records With Repeating Fields**

- Needed e.g. in Object Relational systems or fancy representations of any n:m relationships.
Storing Records in Blocks

- Blocks have fixed size (typically 4k)

Storage and Indexing

- How do we store efficiently large amounts of data?
- The appropriate storage depends on what kind of accesses we expect to have to the data.
- We consider:
  - primary storage of the data
  - additional indexes (very very important).

Cost Model for Our Analysis

- As a good approximation, we ignore CPU costs:
  - B: The number of data pages
  - R: Number of records per page
  - D: (Average) time to read or write disk page
  - M: Assuring num ber of page 10's ignore gains of pre-fetching blocks of pages; thus, even 10 cost is only approximated.
  - A very-case analysis; based on several simplistic assumptions.

File Organizations and Assumptions

- Heap Files:
  - Equality selection on key; exactly one match.
  - Insert always at end of file.
- Sorted Files:
  - Files compacted after deletions.
  - Selections on sort field(s).
- Hashed Files:
  - No overflow buckets; 80% page occupancy.
  - Single record insert and delete.

Cost of Operations

<table>
<thead>
<tr>
<th></th>
<th>Heap File</th>
<th>Sorted File</th>
<th>Hashed File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan all recs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Equality Search</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Range Search</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insert</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Indexes

- An index on a file speeds up selections on the search key fields for the index.
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - Search key is not the same as a key in this set of fields that uniquely identify a record in a relation.
- An index contains a collection of data entries, and supports efficient retrieval of all data entries with a given key value k.
### Index Classification
- Primary/Secondary
- Clustered/Unclustered
- Dense/Sparse
- B+ tree/Hash table/...

### Primary Index
- File is sorted on the index attribute
- Dense index: sequence of (key, pointer) pairs

### Primary Index
- Sparse index

### Primary Index with Duplicate Keys
- Dense index:

### Primary Index with Duplicate Keys
- Sparse index: pointer to lowest search key in each block:
- Search for 20:

### Primary Index with Duplicate Keys
- Better: pointer to lowest new search key in each block:
- Search for 20:
- Search for 15? 35?
Secondary Indexes

- To index other attributes than primary key
- Always dense (why?)

Clustered/Unclustered

- Primary indexes = usually clustered
- Secondary indexes = usually unclustered

Clustered vs. Unclustered Index

Secondary Indexes

- Applications:
  - index other attributes than primary key
  - index unsorted files (heap files)
  - index clustered data

Applications of Secondary Indexes

- Clustered data
  Company(name, city), Product(pid, maker)

Composite Search Keys

Examples of composite key indexes using lexicographic order.
B+ Trees

- Search trees
- Idea in B Trees:
  - make 1 node = 1 block
- Idea in B+ Trees:
  - make leaves into a linked list (range queries are easier)

B+ Trees Basics

- Parameter d = the degree
- Each node has \( d \) and \( \leq 2d \) keys (except root)
- Each leaf has \( d \) and \( \leq 2d \) keys:

B+ Tree Example

- \( d = 2 \)
- Find the key 40

B+ Tree Design

- How large \( d \)?
- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 bytes
- \( 2d \times 4 + (2d + 1) \times 8 \leq 4096 \)
- \( d = 170 \)

Searching a B+ Tree

- Exact key values:
  - Start at the root
  - Proceed down, to the leaf
- Range queries:
  - As above
  - Then sequential traversal

B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: \( 133^4 = 312,900,700 \) records
  - Height 3: \( 133^3 = 2,352,637 \) records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 M byte
  - Level 3 = 17,689 pages = 133 M Bytes
Hash Tables

- Secondary storage hash tables are much like main memory ones
- Recall basics:
  - There are n buckets
  - A hash function f(k) maps a key k to \([0, 1, \ldots, n-1]\)
  - Store in bucket f(k) a pointer to record with key k
- Secondary storage: bucket = block, use overflow blocks when needed

Hash Table Example

- Assume 1 bucket (block) stores 2 keys + pointers
  - \(h(a) = 0\)
  - \(h(b) = h(f) = 1\)
  - \(h(g) = 2\)
  - \(h(a) = h(c) = 3\)

Searching in a Hash Table

- Search for a:
  - Compute \(h(a) = 3\)
  - Read bucket 3
  - 1 disk access

Insertion in Hash Table

- Place in right bucket, if space
  - E.g. \(h(d) = 2\)

Insertion in Hash Table

- Create overflow block, if no space
  - E.g. \(h(k) = 1\)

Hash Table Performance

- Excellent, if no overflow blocks
- Degrades considerably when number of keys exceeds the number of buckets (i.e. many overflow blocks).
- Typically, we assume that a hash lookup takes 1.2 I/Os.
Where are we?

- File organizations: sorted, hashed, heaps.
- Indexes: hash index, B+-tree
- Indexes can be clustered or not.
- Data can be stored in the index or not.

Hence, when we access a relation, we can either scan or go through an index:
- Called an access path.

Current Issues in Indexing

- Multi-dimensional indexing:
  - how do we index regions in space?
  - Document collections?
  - Multi-dimensional sales data
  - How do we support nearest neighbor queries?

- Indexing is still a hot and unsolved problem!

Multi-dimensional Indexes

- Applications: geographical databases, data cubes.
- Types of queries:
  - partial match (give only a subset of the dimensions)
  - range queries
  - nearest neighbor
  - Where am I? (DB or not DB?)
- Conventional indexes don’t work well here.

Indexing Techniques

- Hash like structures:
  - Grid files
  - Partitioned indexing functions
- Tree like structures:
  - Multiple key indexes
  - kdtrees
  - Quad trees
  - R-trees

Grid Files

- Each region in the grid corresponds to a bucket.
- We color all even if we only have partial matches.
- Some buckets may be empty.
- Reorganization requires moving grid lines.
- Number of buckets grow exponentially with the dimensions.

Partitioned Hash Functions

- A hash function produces k bits identifying the bucket.
- The bits are partitioned among the different attributes.
- Example:
  - Age produces the first 3 bits of the bucket number.
  - Salary produces the last 3 bits.
- Supports partial matches, but is useless for range queries.
**Tree Based Indexing Techniques**

- Salary, 150
- Age, 60
- Age, 47
- 70, 110
- 85, 140
- Salary, 300

**Multiple Key Indexes**

- Each level has an index for one of the attributes.
- Works well for partial matches if the match includes the first attributes.

**Multiple Key Indexes**

Index on first attribute
Index on second attribute

**KD Trees**

Adaptation to secondary storage:
- Allow multiway branches at the nodes, or
- Group interior nodes into blocks.

**Quad Trees**

Each interior node corresponds to a square region (or k-dimes)
- When there are too many points in the region to fit into a block, split it in 4.
- A access algorithm similar to those of KD-trees.

**R-Trees**

- Interior nodes contain sets of regions.
- Regions can overlap and not cover all parent’s region.
- Typical query:
  - Where am I?
- Can be used to store regions as well as data points.
- Inserting a new region may involve extending one of the existing regions (in k-dims).
- Splitting leaves is also tricky.

**Query Execution**

- User/Application
- Query update
- Query compiler
- Query execution plan
- Execution engine
- Read/write pages
- Storage manager
- Buffer manager
- Index, record manager
- Index, record manager
Query Execution Plans

```
SELECT S.name 
FROM Purchase P, Person Q 
WHERE P.buyer=Q.name AND 
Q.city='seattle' AND 
Q.phone > '5430000'
```

Query Plan:
- logical tree 
- in plan entation 
- choice at every node 
- scheduling of operations.

Some operations are from relational algebra, and others (e.g., scan, group) are not.

The Leaves of the Plan: Scans
- Table scan: iterate through the records of the relation.
- Index scan: go to the index, from there get the records in the file (when would this be better?)
- Sorted scan: produce the relation in order. In plan entation depends on relation size.

How do we combine Operations?
- The iterator model. Each operation is in plan ented by 3 functions:
  - Open: sets up the data structures and performs initializations
  - GetNext: returns the next tuple of the result.
  - Close: ends the operations. Cleans up the data structures.
- Enables pipelining!
- Contrast with data-driven materialize model. Sometimes it's the same (e.g., sorted scan).

Implementing Relational Operations
- We will consider how to in plan ent:
  - Selection (\( \sigma \)) Selects a subset of rows from relation.
  - Projection (\( \pi \)) Deletes unwanted columns from relation.
  - Join (\( \Join \)) Allows us to combine two relations.
  - Set-difference Tuples in reln. 1, but not in reln. 2.
  - Union Tuples in reln. 1 and in reln. 2.
  - Aggregation (\( \sum, \min, \text{etc.} \)) and GROUP BY

Schema for Examples

Purchase (buyer: string, seller: string, product: integer),

Person (name: string, city: string, phone: integer)

- Purchase:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages (i.e., 100,000 tuples, 4M B for the entire relation).
- Person:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages (i.e, 40,000 tuples, 2M B for the entire relation).

Simple Selections

```
SELECT * 
FROM Person R 
WHERE R.phone < '543%'
```

- Offset from \( R \) attr op value (\( R \))
- With no index, unsorted: Must essentially scan the whole relation; cost is \( M \) (pages in \( R \)).
- With an index on selection attribute: Use index to find qualifying data entries, then retrieve corresponding data records. Hash index useful only for equality selections.
- Result size estimation:
  - Size of \( R \) \(*\) reduction factor:
    - \( M \) one on this later.
Using an Index for Selections

Cost depends on qualifying tuples, and clustering.
- Cost of finding qualifying data entries typically small plus cost of retrieving records.
- In example, assuming uniform distribution of phones, about 54% of tuples qualify (500 pages, 50000 tuples). With a clustered index, cost is little more than 500 I/Os; if unclustered, up to 50000 I/Os!
- Important refinement for unclustered indexes:
  1. Find sort the ids' of the qualifying data entries.
  2. Fetch ids in order. This ensures that each data page is looked at just once (though # of such pages likely to be higher than with clustering).

Two Approaches to General Selections

First approach: Find the most selective access path, retrieve tuples using it, and apply any remaining terms that don't match the index:
- Most selective access path: An index or file scan that we estimate will require the fewest page I/Os.
- Consider city = "seattle" AND phone < 543%
  - A hash index on city can be used; then, phone < 543% must be checked for each retrieved tuple.
  - Similarly, a b-tree index on phone could be used; city = "seattle" must then be checked.

Intersection of Rids

Second approach
- Get sets of rids of data records using each matching index.
- Then intersect these sets of rids.
- Retrieve the records and apply any remaining terms.

Implementing Projection

Two parts:
- Remove unwanted attributes,
- Remove duplicates from the result.

Refinements to duplicate removal:
- If an index on a relation contains all wanted attributes, then we can do an index-only scan.
- If the index contains a subset of the wanted attributes, you can remove duplicates locally.

Equality Joins with One Join Column

\[ \text{SELECT DISTINCT} \]
\[ \text{R.name, R.phone} \]
\[ \text{FROM Person R, Purchase S WHERE R.name=S.buyer} \]

Discussion

- How would you implement join?
Simple Nested Loops Join

For each tuple \( r \) in \( R \) do
  for each tuple \( s \) in \( S \) do
    if \( r \) == \( s \) then add \( <r, s> \) to result

- Cost: \( M + (p_R \cdot M) \cdot N = 1000 + 100 \cdot 100 \cdot 1000 \) I/Os; 140 hours!

Page-oriented Nested Loops join: For each page of \( R \), get each page of \( S \), and write out matching pairs of tuples \( <r, s> \), where \( r \) is in \( R \)-page and \( S \) is in \( S \)-page.
- Cost: \( M + M \cdot N = 1000 + 100 \cdot 500 \) (1.4 hours)

Examples of Index Nested Loops

- Hash-index on name of Person (as inner):
  - Scan Purchase: 1000 page I/Os, 100 * 1000 tuples.
  - For each Person tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get the exactly one) m matching Person tuples. Total: 220,000 I/Os. (36 minutes)

- Hash-index on buyer of Purchase (as inner):
  - Scan Person: 500 page I/Os, 80 * 500 tuples.
  - For each Person tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Purchase tuples. Assuming uniform distribution, 2.5 purchases per buyer \( (100,000 / 40,000) \). Cost of retrieving them is 1 or 2.5 I/Os depending on clustering.

Index Nested Loops Join

For each tuple \( r \) in \( R \) do
  for each tuple \( s \) in \( S \) where \( r \) == \( s \) do
    add \( <r, s> \) to result

- If there is an index on the join column of one relation (say \( S \)), can be used in the inner.
  - Cost: \( M + (M \cdot p_R) \cdot \text{cost of finding matching } S \text{ tuples} \)
- For each \( R \) tuple, cost of probing \( S \) index is about 1.2 for hash index, 2-4 for \( B+ \) tree. Cost of then finding \( S \) tuples depends on clustering.
  - Clustered index: 1 I/O typical, unclustered up to 1 I/O per matching \( S \) tuple.

Sort-Merge Join (\( R \bowtie S \))

- Sort \( R \) and \( S \) on the join column \( n \), then scan them to do a "merge" on the join column.
  - Advance scan of \( R \) until current \( R \)-tuple >= current \( S \)-tuple, then advance scan of \( S \) until current \( S \)-tuple >= current \( R \)-tuple; do this until current \( R \)-tuple = current \( S \)-tuple.
  - At this point, all \( R \) tuples with same value and all \( S \) tuples with same value match; output \( <r, s> \) for all pairs of such tuples.
  - Then resume scanning \( R \) and \( S \).

Cost of Sort-Merge Join

- \( R \) is scanned once; each \( S \) group is scanned once per matching \( R \) tuple.
- Cost: \( M \cdot \log M + N \cdot \log N + M \cdot N \)
  - The cost of scanning, \( M \cdot N \), could be \( M \cdot N \) (unlikely!)
- With 35, 100 or 300 buffer pages, both Person and Purchase can be sorted in 2 passes; total: \( 7500 \) (75 seconds).

Block Nested Loops Join

- Use one page as an input buffer for scanning the inner \( S \), one page as the output buffer, and use all remaining pages to hold "block" of outer \( R \).
- For each matching tuple \( r \) in \( R \)-block, \( s \) in \( S \)-page, add \( <r, s> \) to result. Then read next \( R \)-block, scan \( S \), etc.
**Hash-Join**

- Partition both relations using hash fn h: R tuples in partition i will only match S tuples in partition i.

  - Read in a partition of R, hash it using h2 (<> h!), scan matching partition of S, search for matches.

  - Has h-table for partition Ri (k < B-1 pages)

**Cost of Hash-Join**

- In partitioning phase, read+write both relations; 2 (M + N).
- In matching phase, read both relations; M + N I/Os.
- In our running example, this is a total of 4500 I/Os. (45 seconds!)
- Scat-Merge Join vs. Hash Join:
  - Given a minimum amount of memory both have a cost of 3 (M + N) I/Os. Hash Join superior on this count if relation sizes differ greatly. A lao, Hash Join shown to be highly parallelizable.
  - Scat-Merge less sensitive to data skew; result is sorted.

**Double Pipelined Join (Tukwila)**

- Partially pipelined: no output until inner read
- Asymmetric (inner vs. outer) - optimization requires source behavior knowledge

- Outputs data immediately
- Symmetric - requires less source knowledge to optimize

**Discussion**

- How would you build a query optimizer?