CSE 532 – Theory of Database Systems

Lecture 17 (Chapter 9)
Physical Data Organization and Indexing

Ilchul Yoon
Assistant Professor
State University of New York, Korea

Disks

- Capable of storing large quantities of data cheaply
- Non-volatile
- Extremely slow compared with cpu speed
- Performance of DBMS largely a function of the number of disk I/O operations that must be performed
Physical Disk Structure

- Data files decomposed into pages
  - Fixed size piece of contiguous information in the file
  - Unit of exchange between disk and main memory
- Disk divided into page size blocks of storage
  - Page can be stored in any block
- Application’s request for read item satisfied by:
  - Read page containing item to buffer in DBMS
  - Transfer item from buffer to application
- Application’s request to change item satisfied by
  - Read page containing item to buffer in DBMS (if it is not already there)
  - Update item in DBMS (main memory) buffer
  - (Eventually) copy buffer page to page on disk
### I/O Time to Access a Page

- **Seek latency** – time to position heads over cylinder containing page
  - 1st HDD: 600ms, mid-70s: 25ms, in 80s: 20ms,
  - ~2013: 4~10ms, SSD: 0.08~0.16ms
- **Rotational latency** – additional time for platters to rotate so that start of block containing page is under head
  - For 7200rpm HDD, ~4ms
- **Transfer time** – time for platter to rotate over block containing page (depends on size of block)
  - As of 2011, max. transfer time was 25 ~ 100 MBs

- **Latency** = seek latency + rotational latency
- **Goal:** minimize average latency, reduce number of page transfers

### Reducing Latency

- Store pages containing related information close together on disk
  - **Justification:** If application accesses x, it will next access data related to x with high probability

- Page size tradeoff:
  - Large page size – data related to x stored in same page; hence additional page transfer can be avoided
  - Small page size – reduce transfer time, reduce buffer size in main memory
  - Typical page size – 4096 bytes
Reducing Number of Page Transfers

- Keep cache of recently accessed pages in main memory
  - *Rationale*: request for page can be satisfied from cache instead of disk
  - Purge pages when cache is full
    - For example, use LRU algorithm
    - Record clean/dirty state of page (clean pages don’t have to be written)

Accessing Data Through Cache
RAID Systems

- RAID (Redundant Array of Independent Disks) is an array of disks configured to behave like a single disk with:
  - Higher throughput
    - Multiple requests to different disks can be handled independently
    - If a single request accesses data that is stored separately on different disks, that data can be transferred in parallel
  - Increased reliability
    - Data is stored redundantly
    - If one disk should fail, the system can still operate

Striping

- Data that is to be stored on multiple disks is said to be *striped*
  - Data is divided into *chunks*
    - Chunks might be bytes, disk blocks etc.
  - If a file is to be stored on three disks
    - First chunk is stored on first disk
    - Second chunk is stored on second disk
    - Third chunk is stored on third disk
    - Fourth chunk is stored on first disk
    - And so on
Levels of RAID System

- **Level 0**: Striping but no redundancy
  - A striped array of $n$ disks
  - The failure of a single disk ruins everything

- **Level 1**: Mirrored Disks (no striping)
  - An array of $n$ mirrored disks - all data stored on two disks
  - Increases reliability?
    - If one disk fails, the system can continue
  - Increases speed of reads?
    - Both disks can be read concurrently
  - Decreases speed of writes?
    - Each write must be made to two disks
  - Requires twice the number of disks

RAID Levels (con’t)

- **Level 3**: bit-interleaved parity
  - Data is striped over $n$ disks and an $(n+1)\text{th}$ disk is used to stores the exclusive or (XOR) of the corresponding bytes on the other $n$ disks
  - The $(n+1)\text{th}$ disk is called the parity disk
  - Chunks are bytes
Level 3 (con’t)

- Redundancy increases reliability
  - Setting a bit on the parity disk to be the XOR of the bits on the other disks makes the corresponding bit on each disk the XOR of the bits on all the other disks, including the parity disk
    
    \[1 0 1 0 1 \ 1\] (parity disk)
  - If any disk fails, its information can be reconstructed as the XOR of the information on all the other disks
  - Whenever a write is made to any disk, a write must be made to the parity disk
    
    \[\text{New Parity Bit} = \text{Old Parity Bit} \ \text{XOR} (\text{Old Data Bit} \ \text{XOR} \ \text{New Data Bit})\]
  - Thus, each write requires 4 disk accesses
  - The parity disk can be a bottleneck since all writes involve a read and a write to the parity disk

raid levels (con’t)

- Level 5: Block-interleaved distributed parity
  - Data is striped and parity information is stored as in level 3, but
  - The chunks are disk blocks
  - The parity information is itself striped and is stored in turn on each disk
    - Eliminates the bottleneck of the parity disk
  - Level most often recommended for transaction processing applications
RAID Levels (con’t)

- **Level 10 or 1+0**: A combination of levels 0 and 1
  - not an official level. But, some researchers define this strategy as *Level 1 (disk mirroring with block striping)*
  - A striped array of n disks (as in level 0)
  - Each of these disks is mirrored (as in level 1)
    - Achieves best performance of all levels
    - Requires twice as many disks

Levels of RAID System

- **RAID 2**: memory-style error-correcting codes
  - ECC using Hamming code
  - Not used in practice

- **RAID 4**: block-interleaved parity
  - Block-level striping
  - Read request may need less disk accesses
  - Support parallel disk accesses

- **RAID 6**: P+Q redundancy
Hamming Code

- Parity bits for \(2^k\)-th position in a data stream
  - Each parity bit takes care of specific digit in binary number representation of the bit positions

<table>
<thead>
<tr>
<th>Bit position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoded data</td>
<td>p1</td>
<td>p2</td>
<td>d1</td>
<td>d2</td>
<td>d3</td>
<td>d4</td>
<td>d5</td>
<td>d6</td>
<td>d7</td>
<td>d8</td>
<td>d9</td>
<td>d10</td>
<td>d11</td>
<td>d12</td>
<td>d13</td>
<td>d14</td>
<td>d15</td>
<td>d16</td>
<td>d17</td>
<td>d18</td>
</tr>
<tr>
<td>Parity bit</td>
<td>p1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>p2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>p3</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>p4</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>p5</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>p16</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

E.g., \(p_k\) is for the parity for the LSB. Computed from the bit positions:
- \(k=1: 0001, 0011, 0101, 0111, 1001, 1011, 1101, 1111, ...\)
- \(k=2: 0010, 0011, 0110, 0111, 1010, 1011, 1110, 1111, ...\)
- \(k=4: 0100, 0101, 0110, 0111, 1100, 1101, 1110, 1111, ...\)

- Example
  - Original Data: 1 1 0 0 1 1 0
  - Received Data: 1 1 0 0 1 0 0 (w/ error)
  - Position : p p d p d d d

From received data, compute
\(p_1\) must be 1 (okay)
\(p_2\) must be 0 (incorrect)
\(p_3\) must be 1 (incorrect)
So, error loc = 1102 = 6 (Read erroneous from \(d_3\))

Levels of RAID System

- RAID 6: P+Q redundancy
  - Double parity calculation based on error correcting codes
  - Can tolerate two disk failures
Controller Cache

- To further increase the efficiency of RAID systems, a controller cache can be used in memory
  - When reading from the disk, a larger number of disk blocks than have been requested can be read into memory

- In write back cache, the RAID system reports that the write is complete as soon as the data is in the cache (before it is on the disk)
  - Requires some redundancy of information in cache

- If all the blocks in a stripe are to be updated, the new value of the parity block can be computed in the cache and all the writes done in parallel

Access Path

- Refers to the algorithm + data structure (e.g., an index) used for retrieving and storing data in a table

- The choice of an access path to use in the execution of an SQL statement has no effect on the semantics of the statement

- However, this choice can have a major effect on the execution time of the statement
Heap Files

- Rows appended to end of file as they are inserted
  - Hence the file is unordered
- Deleted rows create gaps in file
  - File must be periodically compacted to recover space

### Transcript Stored as a Heap File

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MGT123</td>
<td>S1994</td>
<td>4.0</td>
<td>MGT123</td>
<td>S1994</td>
<td>4.0</td>
</tr>
<tr>
<td>666666666</td>
<td>CS305</td>
<td>S1996</td>
<td>4.0</td>
<td>666666666</td>
<td>CS305</td>
<td>S1996</td>
</tr>
<tr>
<td>123454321</td>
<td>CS305</td>
<td>F1995</td>
<td>2.0</td>
<td>123454321</td>
<td>CS305</td>
<td>F1995</td>
</tr>
<tr>
<td>987654321</td>
<td>CS305</td>
<td>E101</td>
<td>3.0</td>
<td>987654321</td>
<td>CS305</td>
<td>E101</td>
</tr>
<tr>
<td>111111111</td>
<td>MGT123</td>
<td>F1997</td>
<td>3.0</td>
<td>111111111</td>
<td>MGT123</td>
<td>F1997</td>
</tr>
<tr>
<td>234567890</td>
<td>CS305</td>
<td>S1997</td>
<td>4.0</td>
<td>234567890</td>
<td>CS305</td>
<td>S1997</td>
</tr>
<tr>
<td>123454321</td>
<td>CS305</td>
<td>S1991</td>
<td>3.0</td>
<td>123454321</td>
<td>CS305</td>
<td>S1991</td>
</tr>
<tr>
<td>123454321</td>
<td>MGT123</td>
<td>S1996</td>
<td>3.0</td>
<td>123454321</td>
<td>MGT123</td>
<td>S1996</td>
</tr>
<tr>
<td>234567890</td>
<td>EE101</td>
<td>F1995</td>
<td>3.0</td>
<td>234567890</td>
<td>EE101</td>
<td>F1995</td>
</tr>
<tr>
<td>234567890</td>
<td>EE101</td>
<td>F1995</td>
<td>3.0</td>
<td>234567890</td>
<td>EE101</td>
<td>F1995</td>
</tr>
<tr>
<td>234567890</td>
<td>EE101</td>
<td>F1995</td>
<td>3.0</td>
<td>234567890</td>
<td>EE101</td>
<td>F1995</td>
</tr>
<tr>
<td>234567890</td>
<td>EE101</td>
<td>F1995</td>
<td>3.0</td>
<td>234567890</td>
<td>EE101</td>
<td>F1995</td>
</tr>
<tr>
<td>234567890</td>
<td>EE101</td>
<td>F1995</td>
<td>3.0</td>
<td>234567890</td>
<td>EE101</td>
<td>F1995</td>
</tr>
<tr>
<td>234567890</td>
<td>EE101</td>
<td>F1995</td>
<td>3.0</td>
<td>234567890</td>
<td>EE101</td>
<td>F1995</td>
</tr>
<tr>
<td>234567890</td>
<td>EE101</td>
<td>F1995</td>
<td>3.0</td>
<td>234567890</td>
<td>EE101</td>
<td>F1995</td>
</tr>
<tr>
<td>234567890</td>
<td>EE101</td>
<td>F1995</td>
<td>3.0</td>
<td>234567890</td>
<td>EE101</td>
<td>F1995</td>
</tr>
</tbody>
</table>

**FIGURE 9.3** Transcript table stored as a heap file. At most four rows per page. **FIGURE 9.4** Transcript table of Figure 9.3 after deletion of rows.
Heap File - Performance

- Assume file contains $F$ pages
- Inserting a row:
  - Access path is scan
  - Avg. $F/2$ page transfers if row already exists (duplicate key)
  - $F+1$ page transfers if row does not already exist
    - $F$ page read and 1 page write
- Deleting a row:
  - Access path is scan
  - Avg. $F/2+1$ page transfers if row exists
  - $F$ page transfers if row does not exist

Heap File - Performance

- Query
  - Access path is scan
  - Organization efficient if query returns all rows and order of access is not important
    SELECT * FROM Transcript
  - Organization inefficient if a few rows are requested
    - Average $F/2$ pages read to get a single row

SELECT T.Grade
FROM Transcript T
WHERE T.StudId=12345 AND T.CrsCode = 'CS305'
AND T.Semester = 'S2000'
Heap File - Performance

- Inefficient when a subset of rows is requested:
  - $F$ pages must be read

```
SELECT T.Course, T.Grade
FROM Transcript T
WHERE T.StudId = 123456  -- equality search

SELECT T.StudId, T.CrsCode
FROM Transcript T
WHERE T.Grade BETWEEN 2.0 AND 4.0  -- range search
```

Sorted File

- Rows are sorted based on some attribute(s)
  - Access path is binary search
  - Equality or range query based on that attribute has cost $\log_2 F$
    to retrieve page containing first row
  - Successive rows are in same (or successive) page(s) and cache
    hits are likely
  - By storing all pages on the same track, seek time can be
    minimized

- Example – Transcript sorted on StudId :

```
SELECT T.Course, T.Grade
FROM Transcript T
WHERE T.StudId = 123456

SELECT T.Course, T.Grade
FROM Transcript T
WHERE T.StudId BETWEEN 111111 AND 199999
```
Transcript Stored as a Sorted File

Maintaining Sorted Order

- **Problem:**
  - After the correct position for an insert has been determined, inserting the row requires (on average) \( F/2 \) reads and \( F/2 \) writes (because shifting is necessary to make space)

- **Partial Solution 1:**
  - Leave empty space in each page: *fillfactor*

- **Partial Solution 2:**
  - Use overflow pages (chains).
  - **Disadvantages:**
    - Successive pages no longer stored contiguously
    - Overflow chain not sorted, hence cost no longer \( \log_2 F \)
Additional Discussion #1 – Relying on OS VM?

- Why does DBMS care for the page management?
  - Indeed, it behaves like virtual-memory manager in OS

  - However...
    - Database size could be larger than hardware address space.
    - DBMS understands query execution plan and access patterns. So, better (and sophisticated) cache management (e.g., pre-fetching) is possible
    - DBMS may pin a page in buffer pool, and force it to be written in disk. (recovery)

  - In addition,
    - It looks like some DBMS (e.g., Oracle) can directly access disk bypassing the use of the OS file system buffers.
Index

- Mechanism for efficiently locating row(s) without having to scan entire table

- Based on a search key: rows having a particular value for the search key attributes can be quickly located

- Don’t confuse candidate key with search key:
  - Candidate key: set of attributes; guarantees uniqueness
  - Search key: sequence of attributes; does not guarantee uniqueness – just used for search

Index Structure

- Contains:
  - Index entries
    - Can contain the data tuple itself (index and table are integrated in this case); or
    - Search key value and a pointer to a row having that value; table stored separately in this case – unintegrated index
  - Location mechanism
    - Algorithm + data structure for locating an index entry with a given search key value
  - Index entries are stored in accordance with the search key value
    - Entries with the same search key value are stored together (hash, B-tree)
    - Entries may be sorted on search key value (B-tree)
Index Structure

Location mechanism facilitates finding index entry for S

Once index entry is found, the row can be directly accessed

Storage Structure

- Structure of file containing a table
  - Heap file (no index, not integrated)
  - Sorted file (no index, not integrated)
  - Integrated file containing index and rows (index entries contain rows in this case)
    - ISAM
    - B+ tree
    - Hash
Integrated Storage Structure

- Contains table and (main) index

Index File With Separate Storage Structure

- The storage structure might be a heap or sorted file, but often is an integrated file with another index (on a different search key – typically the primary key)
Indices: The Down Side

- Additional I/O to access index pages (except if index is small enough to fit in main memory)
- Index must be updated when table is modified.

- SQL-92 does not provide for creation or deletion of indices
  - Index on primary key generally created automatically
  - Vendor specific statements:
    - CREATE INDEX ind ON Transcript (CrsCode)
    - DROP INDEX ind

Clustered Index

- Clustered index: index entries and rows are ordered in the same way
  - An integrated storage structure is always clustered (since rows and index entries are the same)
  - The particular index structure (e.g., hash, tree) dictates how the rows are organized in the storage structure
    - There can be at most one clustered index on a table
  - CREATE TABLE generally creates an integrated, clustered (main) index on primary key
Clustered Main Index

**Storage structure contains table and (main) index; rows are contained in index entries**

![Data File](image)

Clustered Secondary Index

![Data File](image)

**FIGURE 9.8** A clustered index that references a separate data file.
Unclustered Index

- **Unclustered (secondary) index**: index entries and rows are not ordered in the same way

- An *secondary index* might be clustered or unclustered with respect to the storage structure it references
  - It is generally unclustered (since the organization of rows in the storage structure depends on main index)
  - There can be many secondary indices on a table
  - Index created by `CREATE INDEX` is generally an unclustered, secondary index

Unclustered Secondary Index

![Diagram of an unclustered index over a data file.](image)

**FIGURE 9.9** An unclustered index over a data file.
Clustered Index

- Good for range searches when a range of search key values is requested
  - Use location mechanism to locate index entry at start of range
    - This locates first row.
  - Subsequent rows are stored in successive locations if index is clustered (not so if unclustered)
  - Minimizes page transfers and maximizes likelihood of cache hits

Example – Range Search Cost Comparison

- Data file has 10,000 pages, 100 rows in search query range
- Page transfers for table rows (assume 20 rows/page):
  - Heap: 10,000 (entire file must be scanned)
  - File sorted on search key: \( \log_2 10000 + (5 \text{ or } 6) \approx 19 \)
  - Unclustered index: \( \leq 100 \)
  - Clustered index: 5 or 6
- Page transfers for index entries (assume 200 entries/page)
  - Heap and sorted: 0
  - Unclustered secondary index: 1 or 2 (all index entries for the rows in the range must be read)
  - Clustered secondary index: 1 (only first entry must be read)