Acknowledgements and copyrights: these slides are a result of combination of notes and slides with contributions from: Michael Kiffer, Arthur Bernstein, Philip Lewis, Anestis Toptsis, Addison Wesley, 4411 textbook. They serve for teaching purposes only and only for the students that are registered in CSE4411 and should not be published as a book or in any form of commercial product, unless written permission is obtained from each of the above listed names and/or organizations.
RAID Levels (con’t)

• **Level 3**: Data is striped over \( n \) disks and an \((n+1)\)th disk is used to store the exclusive or (XOR) of the corresponding bytes on the other \( n \) disks
  
  – The \((n+1)\)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
    
    \[
    \begin{array}{ccccccc}
    1 & 0 & 1 & 0 & 1 & 1 & (\text{parity disk})
    \end{array}
    \]
  
  – If any disk fails, its information can be reconstructed as the XOR of the information on all the other disks
Level 3 (con’t)

• Whenever a write is made to any disk, a write must be made to the parity disk
  \[
  \text{New}_{-}\text{Parity}_{-}\text{Bit} = \text{Old}_{-}\text{Parity}_{-}\text{Bit} \oplus (\text{Old}_{-}\text{Data}_{-}\text{Bit} \oplus \text{New}_{-}\text{Data}_{-}\text{Bit})
  \]
• 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**: 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
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)
  - 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
Data is stored in files

- Data is stored in files.
- Files usually have certain structure in order to facilitate efficient space use and fast retrievals and updates.
- Main file structures are:
  - Unsorted file (heap)
  - Sorted file
  - Indexed file (trees, hash tables).
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

<table>
<thead>
<tr>
<th>ID</th>
<th>Course</th>
<th>Semester</th>
<th>Grade</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>666666</td>
<td>MGT123</td>
<td>F1994</td>
<td>4.0</td>
<td>0</td>
</tr>
<tr>
<td>123456</td>
<td>CS305</td>
<td>S1996</td>
<td>4.0</td>
<td>0</td>
</tr>
<tr>
<td>987654</td>
<td>CS305</td>
<td>F1995</td>
<td>2.0</td>
<td>0</td>
</tr>
<tr>
<td>717171</td>
<td>CS315</td>
<td>S1997</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>666666</td>
<td>EE101</td>
<td>S1998</td>
<td>3.0</td>
<td>1</td>
</tr>
<tr>
<td>765432</td>
<td>MAT123</td>
<td>S1996</td>
<td>2.0</td>
<td>1</td>
</tr>
<tr>
<td>515151</td>
<td>EE101</td>
<td>F1995</td>
<td>3.0</td>
<td>1</td>
</tr>
<tr>
<td>234567</td>
<td>CS305</td>
<td>S1999</td>
<td>4.0</td>
<td>2</td>
</tr>
<tr>
<td>878787</td>
<td>MGT123</td>
<td>S1996</td>
<td>3.0</td>
<td>2</td>
</tr>
</tbody>
</table>
Heap File - Performance

• Assume file contains $F$ pages

• *Inserting a row:*
  – Scan the file until either find the row (do not insert), or do not find it (and then insert at the end)
  – Avg. $F/2$ page transfers if row already exists
  – $F+1$ page transfers if row does not already exist

• *Deleting a row:*
  – Scan the file until either find the row (and then delete it), or not find it (nothing to be deleted).
  – Avg. $F/2+1$ page transfers if row exists
  – $F$ page transfers if row does not exist
Heap File - Performance

• Query
  – scan the file
  – Organization efficient if query returns all rows and order of access is not important. E.g.,
    SELECT * FROM Transcript
  – Organization inefficient if a few rows are requested
  • Average $F/2$ pages read to get a single row. E.g.,
    SELECT T.Grade
    FROM Transcript T
    WHERE T.StudId=12345 AND T.CrsCode =‘CS305’
    AND T.Semester = ‘S2000’
Heap File - Performance

– Organization inefficient when a subset of rows is requested: $F$ pages must be read

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

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

• Rows are sorted based on some attribute(s)
  – Typically use binary search to scan the file.
  – 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

<table>
<thead>
<tr>
<th>Roll No.</th>
<th>Course Code</th>
<th>Section</th>
<th>Year</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>111111</td>
<td>MGT123</td>
<td>F1994</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>111111</td>
<td>CS305</td>
<td>S1996</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>123456</td>
<td>CS305</td>
<td>F1995</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>123456</td>
<td>CS315</td>
<td>S1997</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>123456</td>
<td>EE101</td>
<td>S1998</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>232323</td>
<td>MAT123</td>
<td>S1996</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>234567</td>
<td>EE101</td>
<td>F1995</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>234567</td>
<td>CS305</td>
<td>S1999</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>313131</td>
<td>MGT123</td>
<td>S1996</td>
<td>3.0</td>
<td></td>
</tr>
</tbody>
</table>
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 (how much space is determined by a *fillfactor*).

• **Partial Solution 2**: Use *overflow pages* (*chains*).
  – Disadvantages:
    • Successive pages no longer stored contiguously
    • Overflow chain not sorted, hence cost no longer $\log_2 F$
### Overflow

<table>
<thead>
<tr>
<th>Page 0</th>
<th>Page 1</th>
<th>Page 2</th>
<th>Page 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Page 0" /></td>
<td><img src="image2.png" alt="Page 1" /></td>
<td><img src="image3.png" alt="Page 2" /></td>
<td><img src="image4.png" alt="Page 3" /></td>
</tr>
</tbody>
</table>
Index

• Mechanism for efficiently locating row(s) without having to scan entire table
• Based on a *search key*
• 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
  – For example, for a table STUDENT, the primary key may be *stNumber*, and the search key may be *stGPA*. 
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 – *non-integrated* 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)
Integrated Storage Structure

Contains table
and (main) index
Index File With Separate Storage Structure

In this case, 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)
Index Structure

Search key value

Location mechanism facilitates finding index entry for $S$

Location Mechanism (e.g. how to navigate tree, or what hash function to apply)

Index entries

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

Actual data
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 actual data rows are ordered in the same way
  - An integrated storage structure is always also 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, and it is usually called the main index.*
  - CREATE TABLE generally creates an integrated, clustered (main) index on primary key
Clustered and integrated Index

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

Each entry contains both search key and data.
Clustered (and non-integrated) Index
Unclustered Index

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

Index entry is last in index table. The corresponding data entry is first in data table. ==> unclustered.
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 – Cost of Range Search

• Assume, data file has 10,000 pages, 100 rows in search range

• Page transfers for table rows (assume 20 rows/page):
  – Heap: 10,000 page transfers (entire file must be scanned)
  – File sorted on search key: \( \log_2 10000 + (5 \text{ or } 6) \approx 19 \) (log cost to locate first entry in index, and then 5 or 6 I/Os to transfer the 100 table rows that follow the first row).
  – Unclustered index: \( \leq 100 \) (100 transfers in the worst case, assuming that each row incurs a separate page transfer).
  – Clustered index: 5 or 6 (assuming that index file is in memory)
Sparse vs. Dense Index

- **Dense index**: has index entry for each data record
  - Unclustered index *must* be dense
  - Clustered index need not be dense
  - Sparse index must be clustered.

- **Sparse index**: has index entry for each page of data file
Sparse Vs. Dense Index

Sparse, clustered index sorted on Id

Data file sorted on Id

Dense, unclustered index sorted on Name

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>00940632</td>
<td>Jacob Taylor</td>
<td>MGT</td>
</tr>
<tr>
<td>10120230</td>
<td>John Smyth</td>
<td>CS</td>
</tr>
<tr>
<td>12123234</td>
<td>David Jones</td>
<td>EE</td>
</tr>
<tr>
<td>13114115</td>
<td>Anita Cohen</td>
<td>CS</td>
</tr>
<tr>
<td>23456789</td>
<td>Mary Brown</td>
<td>ECO</td>
</tr>
<tr>
<td>33344455</td>
<td>Ying Chen</td>
<td>CHE</td>
</tr>
<tr>
<td>44455566</td>
<td>Sanjay Sen</td>
<td>ENG</td>
</tr>
<tr>
<td>55566677</td>
<td>Mary Doe</td>
<td>CS</td>
</tr>
<tr>
<td>66677788</td>
<td>Mary Brown</td>
<td>PHY</td>
</tr>
<tr>
<td>90012045</td>
<td>Ann White</td>
<td>MAT</td>
</tr>
</tbody>
</table>

| Mary Brown |
| Mary Brown |
| Ying Chen  |
| Anita Cohen|
| Mary Doe   |
| David Jones|
| Sanjay Sen |
| John Smyth |
| Jacob Taylor|
Sparse Index

Search key should be candidate key (and therefore, unique) of data file (else additional measures required)

Searching for ‘20’ causes a problem if search key is not unique.
Tree indices

• Tree indices can be inspired from binary search …
• Binary search: equivalent to a BST (binary search tree) search.
• BSTs have nodes, with each node holding one key (or one record).
• How about if have a ‘MST’ (Multiway Search Tree) instead of a BST?
• Such trees actually have been thought of before and they are called m-way trees.
M-way trees

• Idea: pack records into blocks, say 7 records per block.

• Possible gain:
  – Instead of one I/O per record, we do one I/O per block (which contains 7 records).
  – … transparencies …