Pete Wilcox — CruzID: pcwilcox — Student ID: 1593715

- **ACID**
  - **ATOMICITY**: An atomic transaction happens as one unit, either the whole thing commits or none of it does.
  - **CONSISTENCY**: A consistent transaction brings the DB from one valid state to another valid state with respect to any constraints.
  - **ISOLATION**: Concurrent isolated transactions would have the same result if run sequentially.
  - **DURABILITY**: A committed transaction will remain committed even in the event of a hardware failure.
- **RAID Levels**
  - Level 0: No redundancy (just stripin)
  - Level 1: Mirrored (two identical copies)
    - Each disk has an exact mirror image
    - Parallel reads; writes involve two disks
    - Maximum transfer rate = transfer rate of one disk
  - Level 0+1 (Level 10): Striping and Mirroring
    - Parallel reads; writes involve two disks
    - Maximum transfer rate = aggregate bandwidth
  - Level 3: Bit-interleaved parity
    - Striping Unit: one bit (or byte) (one check disk)
    - Each read and write request involves all disks; disk array can process one request at a time
  - Level 4: Block-interleaved parity
    - Striping unit: one disk block (one check disk)
    - Parallel reads possible for small requests, large requests can utilize full bandwidth
    - Writes involve modified block *and* check disk
  - Level 5: Block-interleaved distributed parity
    - Similar to RAID level 4 but parity blocks are distributed over all disks
- **Buffer Management in a DBMS**
  - DBMS maintains buffer pool of frames, each frame holds a page, info is in <frame#, pageid> table
  - Choice of frame replacement dictated by replacement policy such as LRU
  - When a page is requested:
    - If requested page is not in pool:
      - Choose a frame for replacement
      - If that frame is dirty, write it to disk
      - Read requested page into chosen frame
    - Pin the page and return its address
    - When done the requestor must indicate whether the page has been modified (dirty bit) and unpin
    - Page in pool may be requested many times
      - A pin count is used and a page is a candidate for replacement iff `pin_count = 0`
      - Pinning increments pin count and unpinning decrements
    - Concurrency control and recovery may entail additional I/O when a frame is chosen for replacement (write-ahead log protocol)
    - Frame is chosen for replacement using LRU, clock, MRU, etc
    - Sequential flooding: Caused by using LRU when the number of buffer frames is less than the number of pages in the file
- **Files of Records**
  - Page or block is ok when doing I/O but higher levels of DBMS operate on *records* and thus want *files of records*
  - **FILE:** A collection of pages each containing a collection of records. Must support
    - Insert (append)/delete/modify record
    - Read a particular record specified using *record id*
    - Scan all records possibly with some conditions on the records to be retrieved
  - **Unordered ''Heap''Files:**
    - Simplest file structure that contains records in no particular (logical) order
    - As file grows and shrinks, disk pages are allocated and de-allocated
    - To support record-level operations we must:
      - Keep track of the *pages* in a file: `page id (pid)`
      - Keep track of the *free space* on a page
      - Keep track of the *records* on a page: `record id (rid)`
      - Keep track of *fields* within records
    - Operations: create/destroy file, insert/delete record, fetch record with specific `rid`, scan all records
  - Record formats: `Fixed Length`
    - Information about field types is the same for all records in file; it is stored in *system catalogs*
    - Finding the $i^{th}$ field of a record does not require scanning the record
  - Record formats: `Variable length`
    - Several alternative formats (# of fields is fixed)
    - Fields delimited by special symbols (e.g. $ between fields)
    - Fields preceded by lengths
  - Record formats: `Variable length with directory`
    - Use array of offsets at start of record
  - **Heap file implemented as a list**
    - The header page id and heap file name must be stored someplace
    - Each page contains two extra pointers in this case
    - Refinement: use several lists for different degrees of free space
  - Page formats:
    - File -¿ collection of pages
    - Page -¿ collection of tuples/records
    - Query operators deal with tuples
    - Slotted page format:
      - Each page has a collection of *slots*
      - Each slot contains a record
    - RID: <page id, slot number>
  - **Heap file using a page directory**
    - Page entries can include the number of free bytes on each page
    - Directory is a collection of pages; linked list is one possible implementation
  - **System catalogs:**
    - For each relation:
      - name, file, file structure
      - name, type, length (if fixed) for each attribute
      - Index name, target, and kind for each index
      - also integrity constraints, defaults, nullability, etc
    - For each index: structure (e.g. B+ tree) and search key fields
    - For each view: view name and definition (including query)
    - Plus statistics, authorization, buffer pool size, etc
  - **Column Stores:**
    - Store data "vertically"
    - Contrast with a "row-store" that stores all the attributes of a tuple/record contiguously
    - Each column can be stored as a separate file and compressed
    - SAP HANA:
      - Dictionary compression per column
      - Column main: read-optimized store for immutable data. Uses high data compression and heuristic algorithms to order data to maximize secondary compression
      - Column delta: write-optimized store for inserts, updates, deletes. Uses less compression, appends updates to the end, and merges with main periodically.
    - Additional types: prefix coding, run length coding, cluster coding, sparse coding, indirect coding
  - **Indexes:**
    - Speeds up selections on the search key fields for the index
    - Contains a collection of data entries and supports efficient retrieval of all data entires $k^*$ with a given key value $k$
  - **B+ Tree Indexes**
    - Leaf pages contain *data entries* and are chained (prev & next)
    - Non-leaf pages have *index entries*, used to direct searches
    - Insert/delete at $\log_F N$, keep tree *height-balanced* ($F$ = fanout, $N$ = # leaf pages)
    - Minimum 50% occupancy (in all nodes except root). Each node contains $d \leq m \leq 2d$ entries; $d$ = the *order* of the tree.
    - Typical order $d = 100$
    - Percentage of node that is full is more useful, typical fill-factor 67%
    - Average *fanout* for non-leaves $F = 133$
    - Inserting a data entry:
      - Find correct leaf $L$
      - Put data entry onto $L$
      - If $L$ has enough space, done
      - Otherwise, must split $L$. Redistribute entries evenly, copy up the middle key (key must still exist in leaf). Insert index entry pointing to $L_2$ into parent of $L$.
      - This can happen recursively: if parent of $L$ grows, need to push up middle key.
      - Splits "grow" the tree; root split increases height.
    - Deleting a data entry:
      - Start at root, find leaf $L$ where entry belongs
      - Remove the entry
      - If $L$ is at least half full, done
      - Otherwise, if $L$ has only $d-1$ entries, try to redistribute, borrowing from sibling (adjacent node with same parent)
      - If redistribution fails, merge $L$ and sibling
      - If merge occurred, must delete entry from parent (pointing to merged node)
      - Merge can propagate to root, decreasing height of the tree

| $p$ | $q$ | $p$ OR $q$ | $p$ AND $q$ | $p = q$ |
|---|---|---|---|---|
| T | T | T | T | T |
| T | F | T | F | F |
| T | U | T | U | U |
| F | T | T | F | F |
| F | F | F | F | T |
| F | U | U | F | U |
| U | T | T | U | U |
| U | F | U | F | U |
| U | U | U | U | U |

| $p$ | NOT $p$ |
|---|---|
| T | F |
| F | T |
| U | U |

| Isolation Level | dirty reads | non-repeatable reads | phantoms |
|---|---|---|---|
| READ UNCOMMITTED | Y | Y | Y |
| READ COMMITTED | N | Y | Y |
| REPEATABLE READ | N | N | Y |
| SERIALIZABLE | N | N | N |