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 algoriths 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
- **Hash-Based Indexes:**
  - Good for equality selections
  - Index is a collection of *buckets*. Each bucket = *primary page* plus zero or more *overflow pages* (called *static* hashing). Buckets contain data entries.
  - *Hashing function h*: $h(r)$ = bucket in which (data entry for) record r belongs. h looks at the *search key* fields of r.
- Alternatives for Data Entry k^* in index:
  - In a data entry k^* we can store: an actual data record, or `<k, RID>`, or `<k, list of RIDs>`
  - Choice of alternative for entries is orthogonal to the indexing technique
- Alternative 1: data records live in index
  - Index structure is actually a file organization for the data records
  - At most one index on a given collection of data can use this Alternative
  - If data records are very large, # of leaf pages containing data entries is high.
- Alternatives 2 and 3: Key/RID or Key/RIDlist:
  - Data entries are typically much smaller than data records
  - Alternative 3 is more compact but leads to variable-sized data entries, even if the search keys are of fixed length
- Index classification:
  - *Primary vs Secondary:* if search key contains the primary key, index is called the primary index
  - *Clustered vs Unclustered:* If order of data records is the same as (or close to) the order of stored data records then index is called a clustered index.
- A back of the envelope cost model:
  - B: the number of data pages
  - R: number of records per page
  - D: average time to read or write a disk page
  - F: average fanout for a non-leaf page
- Indexes with composite search keys:
  - Composite search keys: search on a combination of fields
  - Equality query: every field value is equal to a constant value
  - Range query: some field value doesn't have equality test
  - Data entries in index sorted by search key to support range queries
- **ISAM:** Index-Sequential Access Method
  - Index file has first key on each page, can binary search index then scan the page.
  - *Static* structure, inserts and deletes only affect leaf or overflow pages.
  - If index is very large, recursively create a second layer (and so on).
  - *File Creation:* Leaf pages first allocated sequentially, sorted by search key; then index pages allocated, and then overflow pages.
  - *Index entries:* `<key value, page id>`; they 'direct' searches for *data entries* which are in leaf pages
  - *Search:* Start at root; use key comparisons to go to leaf. I/O cost $\propto \log_F N$ where F = # entries/index pg, N = # leaf pgs
  - *Insert:* Find leaf where data entry belongs and put it there, using overflow page if necessary.
  - *Delete:* Finda nd remove from leaf; if empty overflow page, deallocate
- **B-Tree Prefix Key Compression:** Increase fan-out by reducing the size of search keys on interior nodes. key values only direct traffic so we only need the minimum length for that
- **Bulk Loading of a B+ Tree**
  - Creating a new B+ tree by inserting one at a time is very slow, bulk loading is better
  - *Initialization:* Sort all data entries, insert pointer to first (leaf) page in a new (root) page
  - Index entries for leaf pages always entered into right-most index page just above leaf level. When this fills up it splits.
- **Log-Structured Merge Tree:** Sequential trees of exponentially larger size. Inserts go to smallest smallest tree, deletes insert tombstone records, spill to next-deeper level on overflow
- **R-Tree:** Tree of rectangles, search for intersections between them
- **Static Hash-based Indexes:**
  - # primary pages is fixed, allocated sequentially, never de-allocated; overflow pages if needed
  - $h(k) \mod M$ = bucket(page) to which data entry with key k belongs (M = # buckets)
  - Buckets contain data entries
  - Hash function works on *search key field* of record r. Must distribute over range $0 \ldots M-1$
  - $h(key) = (a * key + b)$ usually works well; a and b are constants to tune h
  - *Long overflow chains* can develop and degrade performance.
- **Extendible Hashing:**
  - Situation: bucket (primary page) becomes full. Solved by doubling number of buckets instead of using an overflow page
  - Use directory wich pointers to buckets. Double the number of buckets by doubling the directory and splitting buckets as needed
  - Only one bucket at a time splits. No overflow pages
  - *Global Depth* is the last d bits after hashing and indexes into the directory to determine which bucket is used
  - *Local Depth* is used for each bucket. If the $LD == GD$ and the bucket splits, the directory must double.
- **Bitmap Indexes:**
  - Index which allows for fast equality checks. Order the records in some $O(1)$ way and maintain one or more bit vectors storing their values for particular fields.
  - One bitmap for each distinct domain value, and one bitmap for `NULL` if the column can be null.
  - Can use `XOR` operation to reduce number of maps needed by one.
- **External Sorting:**
  - Goal: Need to sort more data than will fit in memory, efficiently.
  - **2-Way Sort:** Requires 3 buffers
    - Pass 0: read a page, sort it, write it out (only one buffer page used)
    - Pass 1, 2, 3, ...: Read and merge pairs of runs. (Three buffer pages are used)
  - Sorting $N = 2^k Pages of Data$:
    - Pass 0: read, sort, write → 2^k 1-page runs
    - Pass 1: Read + merge 1-page pairs, write → 2^{k-1} 2-page runs
    - Pass 2: Read + merge 2-page pairs, write → 2^{k-2} 4-page runs
    - Pass $k-1$: Read + merge 2^{k-2}-page pairs, write → 22^{k-1}-page runs
    - Pass k: Read + merge 2^{k-1}-page pairs, write → 12^k-page result
  - 2-Way External merge sort: N pages in file $\Longrightarrow$ $\lceil \log_2 N \rceil + 1$ passes, total I/O cost = $2N(\lceil \log_2 N \rceil + 1)$
  - *General* external merge sort:
    - Sorting a file with N pages using B buffer pages.
    - Pass 0: use B buffer pages. Produce $\lceil N/B \rceil$ sorted runs of B pages each.
    - Pass 2, ... etc: merge $b-1$ runs
    - Number of passes: $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$
    - Cost = $2N * (\#\text{ of passes })$
  - *Double Buffering*

- To reduce wait time for I/O request to complete, can *prefetch* into `shadow block`
- Potentially more passes; in practice, most files still sorted in 2-3 passes.

– *B+ Tree as "Sorted Access Path"*
- Scenario: table to be retrieved in some order has a B+ tree index on the ordering columns
- Idea: retrieve records in order by traversing the B+ tree's leaf pages
- Very good idea if the tree is clustered, otherwise probably a bad idea

- **`Query Processing`**

– *Access Paths:*
- An access path is a method of retrieving tuples: file scan or index that matches a selection in the query
- A tree index *matches* (a conjunction of) terms that involve only attributes in a *prefix* of the search key.
- A hash index *matches* (a conjunction of) terms that has a term *attribute* = *value* for every attribute in the search key of the index.

– Selection conditions often first converted to be in CNF (`AND`ing of `OR`s)

– One approach to selections:
- Find the *most selective access path*, retrieve tuples using it, then apply any remaining terms which don't match the index
- Most selective access path: an index or file scan that we estimate will require the fewest page I/Os
- Terms that match this index reduce the number of tuples retrieved; other terms used to filter the retrieved tuples on the fly, but don't prevent retrieval of the tuples/pages

– Using an index for selections: cost depends on # qualifying tuples and clustering
- Cost of finding qualifying data entries (typically small) plus cost of retrieving the actual records themselves (can be large without clustering)

– *Duplicate Elimination*
- Relational algebra projection removes duplicates: SQL systems don't remove duplicates unless the keyword `DISTINCT` is specified
- Sorting approach: sort on `<sid, bid>` and remove duplicates. Can optimize by dropping unwanted columns while sorting.
- Hashing approach: hash on `<sid, bid>` to create *partitions*. Load partitions into memory one at a time, build an in-memory hash structure, and eliminate duplicates within it.

– Notation:
- Pages in a heap relation R: Pages_R
- Tuples per page for a relation R: TPP_R
- Number of tuples in R: Card_R
- $\text{Card}_R = \text{Pages}_R * \text{TPP}_R$

– *Simple Nested Loops Join*: foreach tuple in R, for each tuple in S, if $r_i == s_j$ then add `<r,s>` to result.
- For each tuple in the *outer* relation R we scan the entire *inner* relation S. Cost: $\text{Pages}_R + \text{Card}_R * \text{Pages}_S$
- Page-oriented nested loops join: for each *page* of R get each *page* of S and write out matching pairs of tuples

– *Index Nested Loops*
- If there is an index on the join column of one relation, can make it the inner and exploit the index. Cost: $\text{Pages}_R + \text{Card}_R$ * cost of finding matching S 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 (assuming alt. 2 or 3) depends on clustering. Clustered typically 1 I/O, unclustered up to 1 I/O per matching S tuple.

– *Block Nested Loops Join*
- Use one page as an input buffer for scanning the inner S, one pas as the output buffer, and use *all* remaining pages to hold "block" of pages of outer R.
- For each block of R, hash each data entry to a hash table. Then compare all entries in S.

– *Join: Sort-Merge* $(R \bowtie_{i=j} S)$
- Sort R and S on the join column, then scan them to do a 'merge' (on join column) and then output result tupls.
- R is scanned once; each S group is scanned once per matching R tuple.

– *Statistics and Catalogs*
- Catalogs typically contain at least: *# tuples* and *# pages* in each relation; *# distinct key values* and *# pages* for each index; *index height*, *low/high key values* for each tree index
- Catalogs updated periodically: updating each time data changes is too expensive and approximation is fine

– *Cost estimation:* For each plan must estimate cost
- Estimate *Cost* of each operation in plan-tree: depends on input cardinalities
- Estimate *Size* of result for each operation in the tree; for seelctions and joins assume independence of predicates

– *Size estimation and reduction factors*
- Maximum # of tuples in result is the product of cardinalities in the `FROM` clause
- Reduction Factor *RF* associated with each *term* reflects the impact of *term* in reducing result size

– *Grace Hash-Join*
- Like a two-phase index nested loop join
- *Build phase:* Partition both relations using hash function h: R tuples in partition i will only match S tuples in partition i.
- *Match phase:* Read in a partition of R, hash it using $h_2(<> h)$. Scan matching partition of S searching for its R matches.
- # partitions $k \leq B - 1$ and Pages_R/k (size of largest partition to be held in memory) $< B - 1$.
- Assuming uniformy-sized partitions and maximizing k: $k = B - 1$ and $\text{Pages}_R/k < B - 1$ so $(B - 1)^2 > \text{Pages}_R \implies B > \sqrt{\text{Pages}_R}$
- Can hash-join *recursively* to reduce the amount of memory needed
- In build phase: Read + write both relations: $2 * (\text{Pages}_R + \text{Pages}_S)$
- In match phase: Read both relations; $\text{Pages}_R + \text{Pages}_S$ I/Os

– Sort-Merge Join vs Hash Join:
- Given a a reasonable amount of memory ($B > \sqrt{\text{Pages}_R}$), both have cost $3 * (\text{Pages}_R + \text{Pages}_s)$ I/Os
- Hash join is superior is relation sizes differ greatly because it needs less memory, and is also highly parallelizable
- Sort-Merge is less sensitive to data skew and its result is sorted

– *More general join conditions:*

- Equalities over several attributes: for INL, build index on composite key. For sort-merge and hash join, sort/hash-partition on the combination of the join columns
- Inequality conditions: for INL, need (clustered!) B+ tree index; hash-join not usable; merge-join possible; block NL the best

Var	Desc
B	the number of data pages
R	number of records per page
D	average time to read or write a disk page
F	average fanout for a non-leaf page

()	Scan	Equality	Range	Insert	Delete
Heap	BD	$0.5BD$	BD	$2D$	Search + D
Sorted	BD	$D \log_2 B$	$D(\log_2 B + \#$ matching pages)	Search + BD	Search + BD
Clustered	$1.5BD$	$D \log_F 1.5B$	$D(\log_F 1.5B + \#$ matching pages)	Search + D	Search + D
Unclust. Tree	$BD(R + 0.15)$	$D(1 + \log_F 0.15B)$	$D(\log_F 0.15B + \#$ matching pages)	Search + $2D$	Search + $2D$
Unclust. Hash	$BD(R + 0.125)$	$2D$	BD	Search + $2D$	Search + $2D$