CMPT 454: Database Systems II

A Course Overview

Jeffrey Leung
Simon Fraser University

Spring 2021

Contents

1 Introduction

- Database (DB): Set of files designed to contains data which can be modified and retrieved, and is optimized for certain operations
- Database Management System (DBMS): Program which manipulates a database

2 Data Storage

- DBMS manages disk space and buffer management instead of delegating to the OS
  - Reason: Needs to support multiple OS platforms and files which span multiple disks
- **Main memory:** RAM data storage; location where current working data is stored
  - Data is automatically removed upon power loss
  - The whole DB is not stored in memory due to high storage cost and volatility
- **Disk:** Hard drive data storage; location where the main database is stored
  - Faster than tapes (which are sequential) due to being random access
- Tapes: Infrequent method for archiving older database versions
- **System catalog:** Database relation which contains information about relations, indices, and views
  - Relation/file information: Name, structure type, attributes and types, indices, integrity constraints
  - Index information: Name, structure type, search key
  - View information: Name, definition

2.1 Physical Disk Components

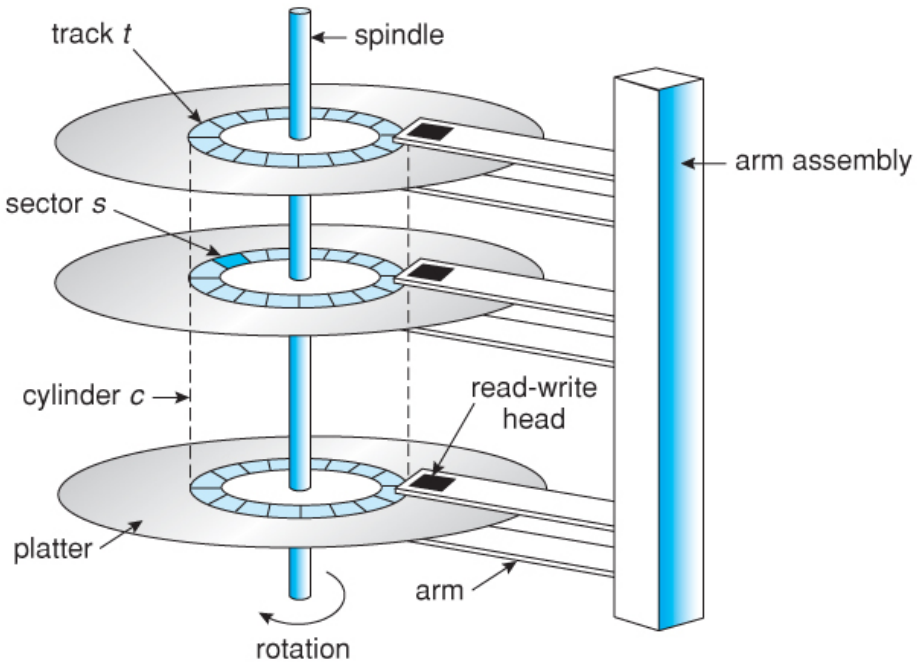


Figure 1: Arm Assembly Diagram

- Arm: Component which moves along platters to read data
  - Head: Component on an arm which reads data
- Platter: Single double-sided physical plate in a hard drive which rotates

  - **Track:** Circular section of data on a platter
    - **Cylinder:** Set of circular tracks vertically adjacent across all platters
  - **Sector:** Pie-shaped section of a platter, from the outer edge to the middle
    - **Track sector:** Section of a track in a specific sector
- Read/write movements:
  - Seek time: High-cost time required to position an arm on a disk
  - Rotational delay: High-cost time required to rotate a platter to a position
  - Transfer time: Low-cost time to read data from the disk with the head
- I/O operations are high-cost because they need mechanical components
  - Read: Transfer of data from disk to main memory
  - Write: Transfer of data from main memory to disk

2.2 Data Pages and Frames

- **Page:** Fixed-size block of data
  - Files are stored as a collection of pages
  - On physical disk:
    - Consists of a contiguous set of sectors on a single track
    - Smallest possible unit of data retrieval
    - Location impacts performance
    - Page ID is uniquely identified by (b, t, c, d): Block b of track t of cylinder c of disk d
  - **Next page concept:** Sequential ordering of pages on the disk to minimize access time
    - Process:
      - Next page on the same track
      - Next page on the same cylinder
      - Next page on the adjacent cylinder
    - Minimizes seek and rotational delay
  - **Sequential scan:** Reading of multiple disk pages sequentially
    - Allows pre-fetching of several pages at a time
  - Fragmentation: Deleted data leaves unallocated spaces of varying sizes on a disk
    - Means that files are not guaranteed to be stored sequentially on disk
- **Frame:** Fixed-size space for data in physical memory
  - For any read/write operation, the target data must first be in memory (i.e. the page containing the data must be allocated in a frame)
  - **Dirty frame:** Frame which has unsaved modifications
    - A bit is used to mark dirty status upon modification
  - **Pinned frame:** Frame which is currently in use and cannot be replaced
    - **Pin count:** Number of currently active users of a page

2.3 Accessing Data Pages

- **Page fault:** Operation when a page is required but not currently in memory
  - If a page fault occurs:
    - A non-pinned frame is selected
    - If the frame is dirty, it is saved to the disk
    - The new frame replaces the old frame
    - The new frame is pinned
    - The lookup table is updated
    - The address of the new frame is returned
- Access pattern: Pattern by which data pages are requested for access
  - **Sequential flooding:** Data page access pattern where each page request is for a new page
- Replacement policy: Strategy which determines which frames are kept in memory or removed
  - Different policies are optimized for different access patterns
  - Examples: First-in-first-out (FIFO), least-recently-used (LRU), most-recently-used (MRU)

2.4 Storing Records on a Page: Fixed-Length

- Introduction to records and fields:
  - **Field:** Section of a record; represents a single primitive datatype
    - Abbreviation: Fx where x is the number of the given field in order
    - Abbreviation of the length of a field: Lx where x is the number of the given field in order
  - **Record:** Section of a file; represents a unique data entity
    - Base address (B): Beginning location of a record on its page
    - Length of a record equals the sum of the length of all its fields
      - Calculation: $L = \sum Li$
      - Location of the i^{th} record: $B = (i-1) \times L$
    - **Slot:** Location of a record on a page
    - **Record ID (RID):** Location of a record
      - Identified by (page ID, slot #)
- **Fixed-length record format:** Data page format where all records have the same types and numbers of fields
  - Length of records are identical
  - Diagram: See figure 2

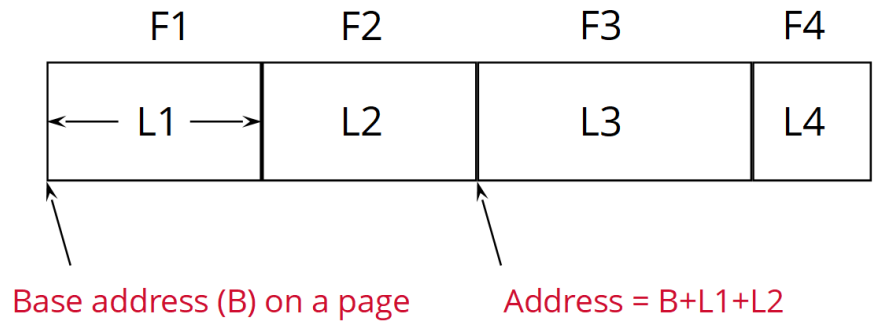


Figure 2: Fixed-length record format

- **Packed page:** Fixed-length record format where free space is always organized contiguously
  - Diagram: See figure 3
  - Contains a value with the number of currently allocated records
  - Record addition: Allocate a slot from the start of free space
  - Record deletion: De-allocate the slot, move the following data up to be contiguous
  - Incurs no operation overhead as the data is in memory
- **Unpacked page:** Fixed-length record format where free space is not contiguous
  - Diagram: See figure 4

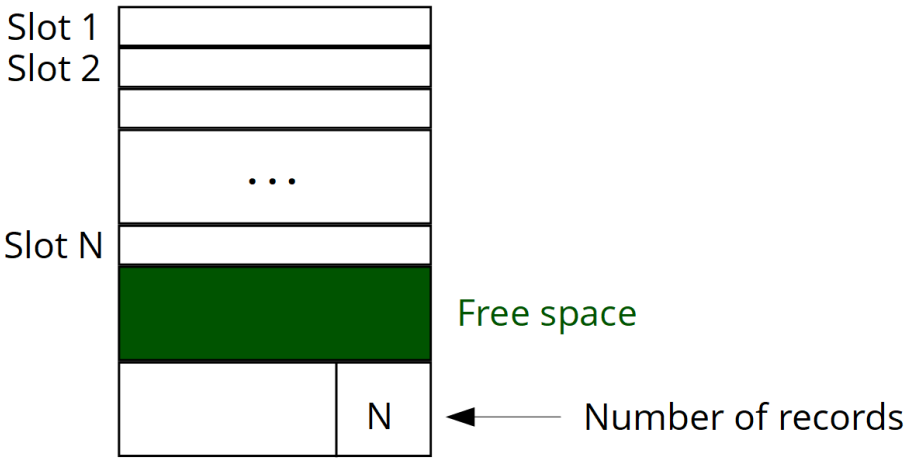


Figure 3: Packed page with fixed-length ecords

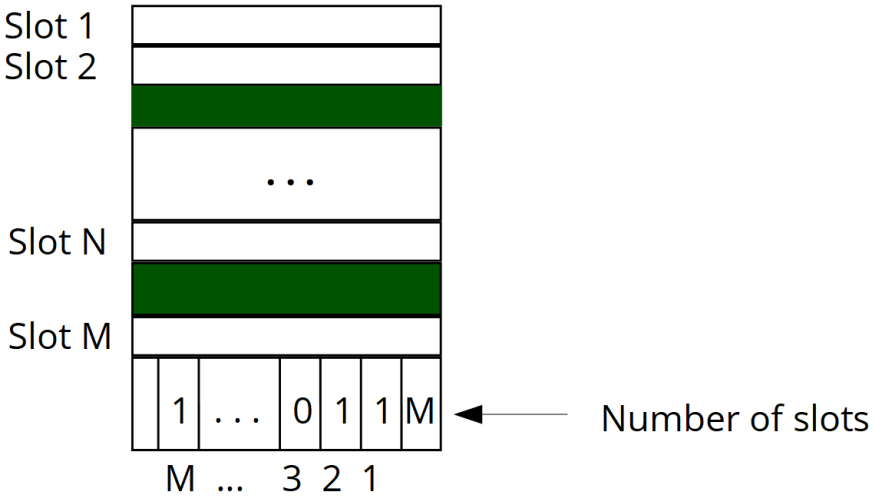


Figure 4: Unpacked page with fixed-length records

- Contains a value with the number of currently unallocated records
- Free space is tracked by a bitmap at the end of the page, which represents whether a slot is allocated or free
- Record addition: Check the bitmap to find a free slot to allocate
- Record deletion: De-allocate the slot in the bitmap
- Incurs no operation overhead as the data is in memory

2.5 Storing Records on a Page: Variable-Length

- **Variable-length record format:** Record format where each record on a page may have differing fields

- Records contain a value representing the number of fields
- **Fixed-fields variable-length record format:** Data page format where, for all records, the number of fields cannot dynamically change (but the records may have differing fields)
  - Diagram: See figure 5

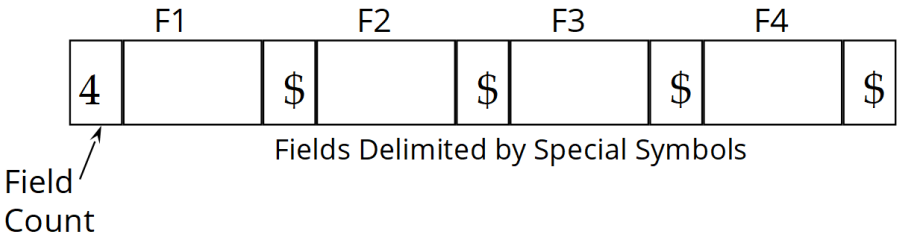


Figure 5: Variable-length records with fixed fields

  - Fields are delimited by unique symbols
  - Access method: Sequential scan from beginning to end
- **Variable-fields variable-length record format:** Data page format where, for all records, the number of fields cannot dynamically change (but the records may have differing fields)
  - Diagram: See figure 6

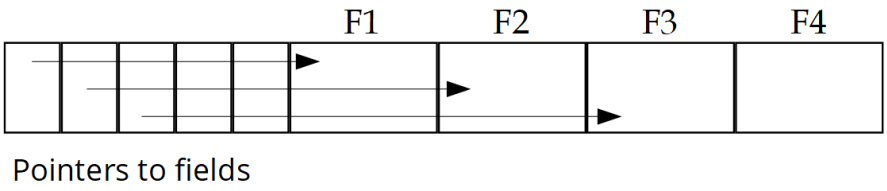


Figure 6: Variable-length records with variable fields

  - Record contains a set of pointers to each field
    - Incurs a space overhead
  - Access method: Directly from the set of pointers
  - Allows easy nullable storage

- Storage of variable-length records on a page:
  - Diagram: See figure 7
  - Page contains a:
    - Value with the number of currently allocated slots
    - Pointer to first available free space
    - Slot directory (at the end of the page) with the starting location of each records
  - Records are allocated by filling the next free space, and linking to it with any free slot in the directory

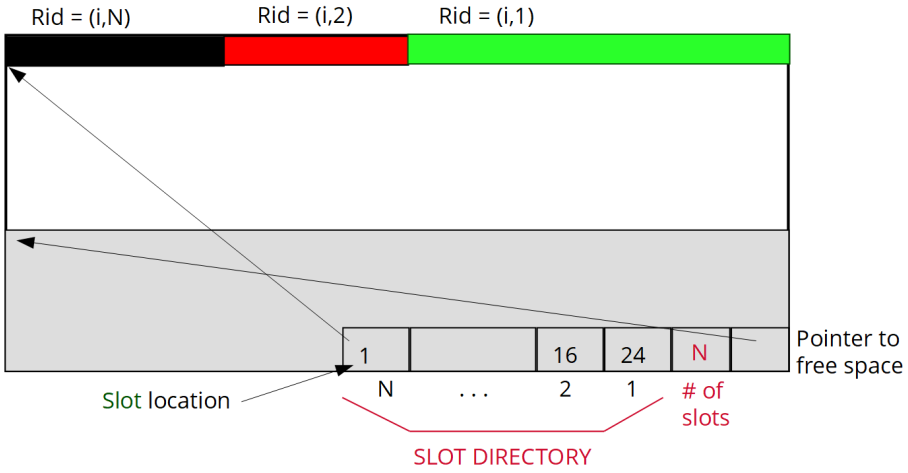


Figure 7: Page of variable-length records

3 Heap File Structure

- **Logical record:** Abstract concept of data which can be inserted/deleted/modified/searched
  - Value-based searches include:
    - Equality search: Search for records equal to a given value
    - Range search: Search for records greater/less than a given value
- **Heap files:** File organization structure where records are unordered
  - Disk pages are allocated/deallocated dynamically as required
  - Efficient for inserting, searching by RID, and file scanning
  - Inefficient for searching values of the fields
- **Linked list heap file implementation:** Heap file implementation containing one linked list with full pages, and one linked list with unallocated pages
  - Allocation process: Search through each non-full page to find one with enough space
  - Diagram: See figure 8

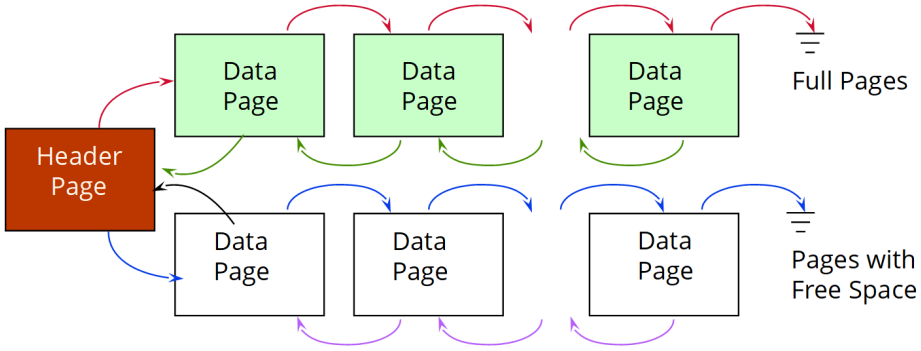


Figure 8: Heap File: Linked List Implementation

- **Page directory heap file implementation:** Heap file implementation consisting of a linked list where each node contains a set of pointers to pages, and the amount of free space on the pages
  - Allocation process: Search through nodes to find a page with enough space (reading only one page)
  - Diagram: See figure 9

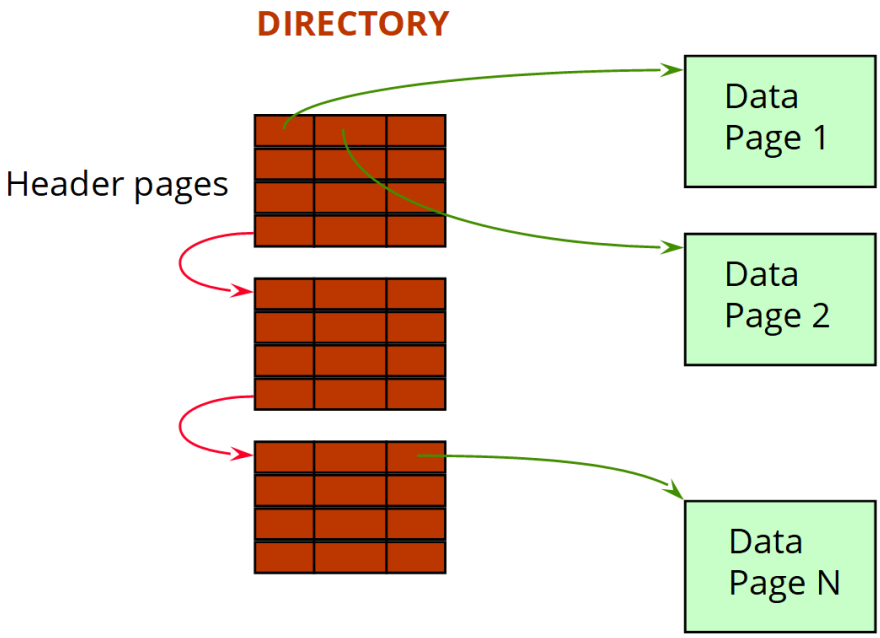


Figure 9: Heap File: Page Directory Implementation

4 Indexed File Structure

4.1 Introduction

- **Indexed file structure:** File organization structure where records are organized and searchable by certain configured search keys
  - Designed to be efficient for value-based searches
- Structure of an index:
  - **Search key:** Field(s) on which a file/record is organized for searchability
    - Not guaranteed to be unique
    - Creating an index associates each record with a corresponding search key
    - **Composite search key:** Search key composed of multiple fields
      - Sorting by a composite key sorts by its fields in order
  - **Data entry:** Single search key and IDs of all records which contain/match it
    - Notation: Given search key value k, the index emits a set of data entries $k*$
    - Always sorted by the search key k
    - Possible formats:
      - $\langle k, \text{data record} \rangle$
        - DB cannot have any other indices
      - $\langle k, \text{RID} \rangle$
      - $\langle k, \text{list of RIDs} \rangle$

4.2 Types of Indices

- Methods of indexing:
  - **Sorted file structure:** Indexed file structure where records are sorted by search key (generally using a tree)
    - Efficient for equality/range searches
  - **Hashed file structure:** Indexed file structure where records are grouped in buckets by a hashed search key
    - Efficient for equality searches
    - Types: Static hashing, extendible hashing, linear hashing
- **Unique index:** Index format where the search key contains a candidate key
  - **(Candidate) Key:** Set of fields which uniquely defines a record
  - I.e. Searching the index using a candidate key returns either 0 or 1 result
- **Clustered index:** Index for which the data records are sorted by k (in addition to the data entries sorted by k)
  - Maximum of 1 clustered index per file
  - Records are sequential on the page, though the pages may have pointers to the next page rather than be sequential

- Diagram: See figure 10

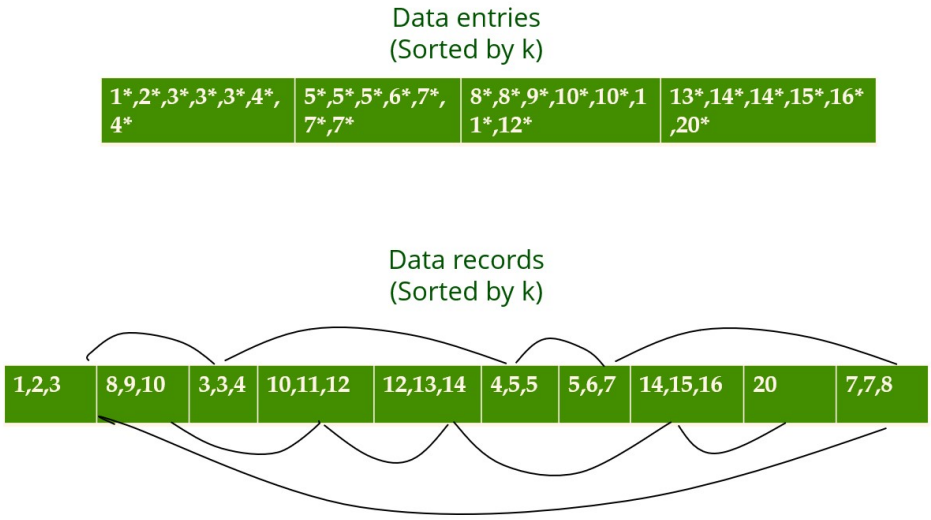


Figure 10: A clustered index and its records

- *Unclustered index:* Index for which the data records are not sorted by k
  - Diagram: See figure 11

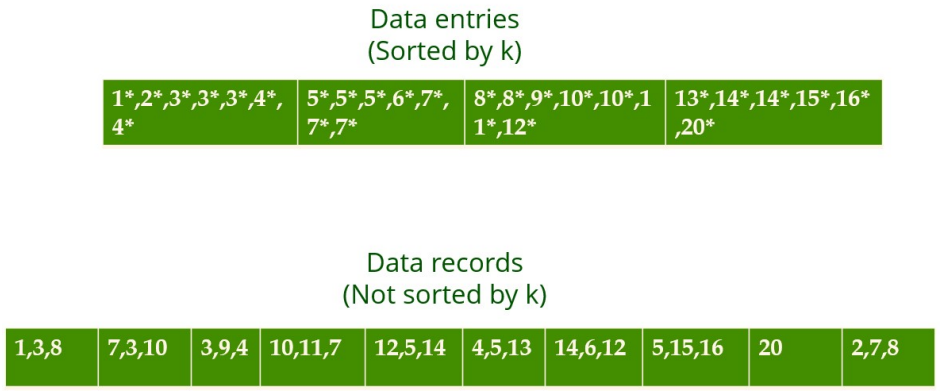


Figure 11: An un clustered index and its records

- Data entry format $\langle k, \text{data record} \rangle$ is always clustered by the single index
- Search process for a given data range in a clustered index:
  - Use the data entry to find the page with the first value to read
  - Read through all data records in order until reaching a value greater than the end

- **Dense index:** Index where, for every search key k, there are one or more data entries $k*$
  - Data entry format $\langle k, \text{data record} \rangle$ is always dense

– **Sparse index:** Index where there is at least one search key k which does not map to any data entries

    * In a sparse index, data entries are always clustered for efficient scanning

4.3 Tree-Structured Indexes

- **Index search tree:** Search structure where index pages branch out to primary leaf pages which contain data entries
  - Root and index pages consist of splitting keys and ordered pointers to trees containing values greater/less than the splitting keys
  - Common components:
    - **Index pages:** Index search tree node which contains pivots for the search key values and pointers to index/leaf pages
      - Does not directly contain data entries
      - **Splitting key:** Index page value which acts as a pivot value for partitioned subtrees
      - Root: Top node of an index search tree which contains a single pivot value and pointers to index pages
    - **Primary leaf page:** Index search tree leaf node which contains a set of data entries with pointers to data records
      - Will not be deallocated even if empty

4.3.1 ISAM Trees

- **Indexed sequential access method (ISAM):** Index search tree where index pages are static and leaf pages contain optional overflow pages
  - **Overflow page:** Linked list starting from a primary leaf page, which contains additional search key values
    - Allocated/deallocated as required; all other nodes in the tree are not affected during insert/delete operations
    - Can degrade performance
- Diagram: See figure 12

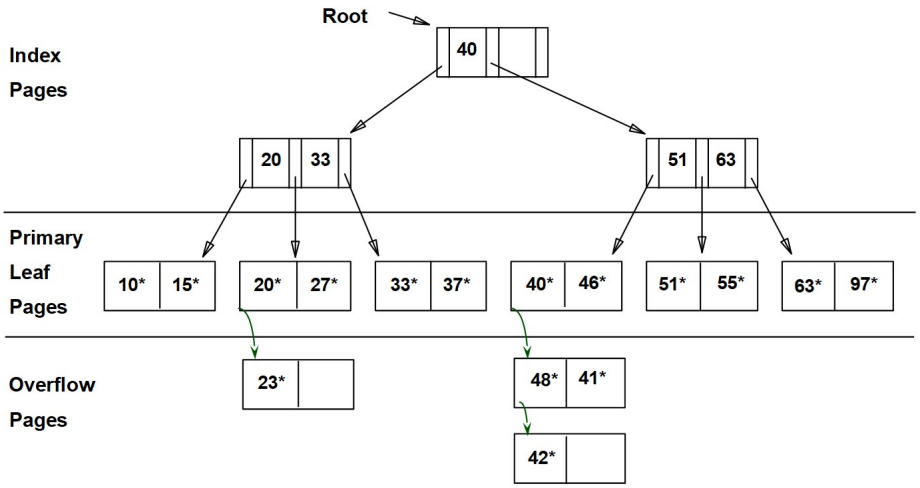


Figure 12: An ISAM Tree

- Search process:
  - Follow the search from the root through the index pages to the matching primary leaf page
  - Search the leaf page
  - Search any linked overflow pages until a result is found
- Rebalanced during system maintenance

4.3.2 B+ Trees

- **B+ tree:** Index search tree which dynamically rebalances nodes and children during operation
- Diagram: See figure 13

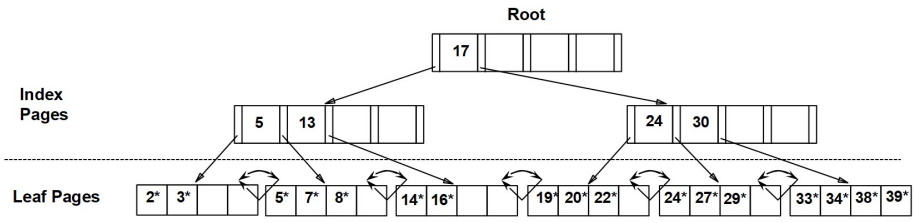


Figure 13: A B+ Tree

- **Order:** Minimum and maximum splitting key capacity of a B+ tree (which can differ for index and leaf nodes)
  - Denoted by d
  - Each B+ tree node must have between d and $2d$ search keys (inclusive), except for the root (minimum 1)
  - *Fanout:* Branching factor of a B+ tree
    - Denoted by F
    - Fanout of root: 2 to $2d + 1$
    - Fanout of index nodes: $d + 1$ to $2d + 1$
    - Fanout of leaf nodes: d to $2d$
- Ensures insert/delete/search operations are $\log_F N$ where $N =$ number of leaf nodes
- Operations:
  - **Copying up:** B+ tree operation where a search key in a leaf is copied to the parent index node
    - Used when splitting a leaf in two
  - **Pushing up:** B+ tree operation where a splitting key in an index node is moved to the parent index node, and removed from the original
    - Used when splitting an index node in two
  - Insertion into a full leaf node:
    - Split leaf L into two nodes, L and $L2$
    - Redistribute entries evenly
    - Copy up the middle key to the parent to create a new partition
    - Link $L2$ to the parent
    - If the parent index node is not being split, the process is repeated recursively but with pushing-up instead of copying-up
  - Deletion:
    - Search for and remove the entry

* If the leaf node which contained the entry has only $d-1$ entries remaining:
    - If the sibling node has more than $d-1$ entries, then re-balance entries with the sibling so both have at least $d-1$ entries
    - Else, merge the node and its sibling (with d entries) to create a node of $2d-1$ entries
        - Delete the key pointing to the deleted leaf node
        - Propagate merge upwards recursively

– **Bulk loading:** Initialization of a B+ tree with a number of records, which avoids constant rebalancing through standard insertion

* Process:
    - Sort data entries by search key
    - Allocate data entries on the disk
    - Create parent nodes for data entry leaf pages from left to right
    - If the parent of the current index node has more slots available, split index nodes into siblings
    - If the parent of the current index node has no slots available, split the parent node

4.4 Hash-Based Indices

- **Hash-based index:** Search structure where primary pages, each identified by a hashed search key, point to data entry pages
  - Optimized for equality search; does not support range search
  - Vulnerable to skewed hash distributions
- Common components:
  - Hash function: Function which hashes a search key to a value
    - Denoted by $h(k)$ for a given data entry with search key k
    - Bucket to store the entry is chosen from N buckets by calculating $h(k) \bmod N$
  - **Primary (bucket) page:** Hash-based index page identified by a hashed search key and containing/linking to a bucket with data entries
    - Always allocated sequentially for speed of access
  - **Directory:** Collection of primary pages
    - Not in all types of hash indices
  - **Overflow page:** Hash index page containing data entries, linked to from a primary page
    - Not in all types of hash indices
    - Allocated and de-allocated as required

4.4.1 Static Hashing

- **Static hashing:** Hash-based index method where the primary pages are never modified or deallocated
  - Uses overflow pages to hold additional data entries
    - May create overflow chains when there are too many records, or hashes skew towards several values
- Diagram: See figure 14

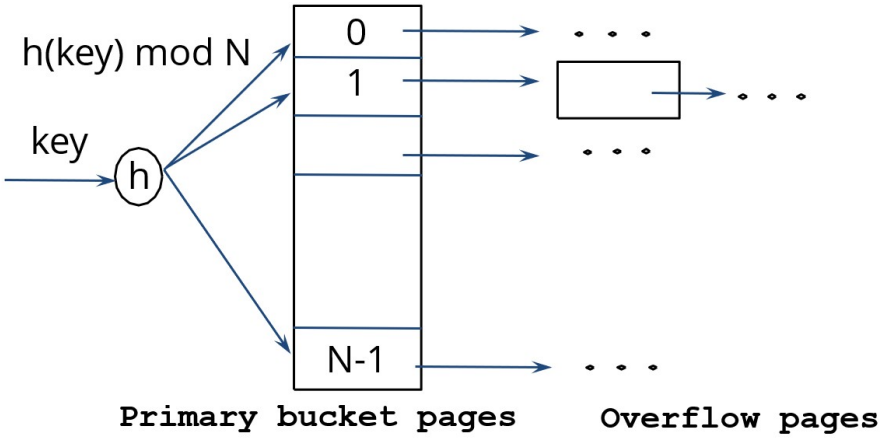


Figure 14: A Static Hash Index

4.4.2 Extendible Hashing

- **Extendible hashing:** Hash-based index method which splits overflowing buckets and dynamically increases the directory
  - Does not use overflow pages
  - When a bucket overflows, its contents are split into a new bucket which is added to the directory
  - Requires accessing a directory page to look up an entry page
  - Bucket access is approximately 1.2 page accesses amortized (due to overflows)
- **Global depth:** Value on an extendible hashing directory which tracks the number of bits in the current directory size
- **Local depth:** Value on an extendible hashing bucket which (approximately) tracks the number of buckets matching the same hash
  - If local depth is less than global depth, then the bucket can be split without increasing directory size
  - If local depth equals the global depth, then splitting the bucket will require doubling the directory
- Diagram: See figure 15

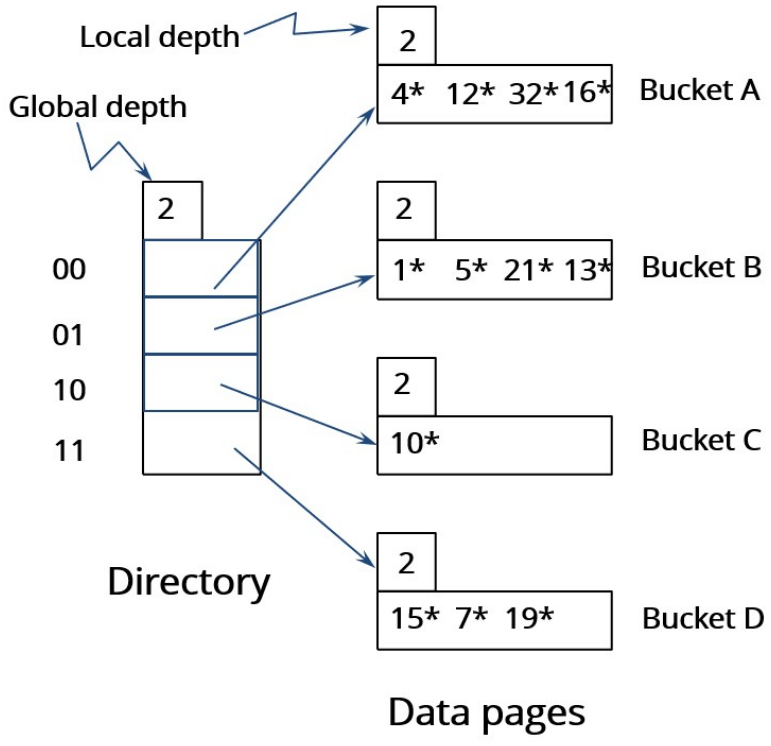


Figure 15: An Extendible Hash Index

- Operations:
  - Insertion:
    - When a bucket overflows, split its contents into a new bucket of the same hash
    - Increment the local depth of both buckets
    - If the directory has no space for a new bucket of the same hash (i.e. the global depth is less than the new local depth), the directory is increased to accommodate hashing the new bucket:
      - Double the directory (adding a new leftmost bit to the directory index)
      - Increment the global depth
      - Point the newly created directory indices to the existing buckets with the same hash values
    - Link the new duplicate hash value in the directory to the new bucket
    - Diagram: See figure 16

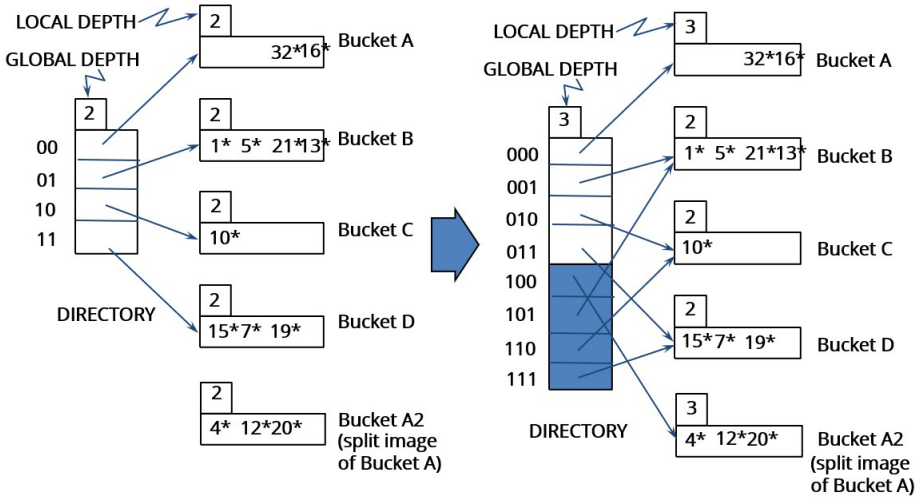


Figure 16: Splitting an Extendible Hash Bucket

  - Deletion:
    - Remove the data entry from the corresponding bucket
    - If the bucket has no more entries, deallocate it and point the key in the directory to the matching bucket
    - If half or more of the buckets of all hashes are empty/deallocated, the directory can be halved

4.4.3 Linear Hashing

- **Linear hashing:** Hash-based index method which uses overflow pages, splits buckets using a round-robin strategy, and dynamically doubles the buckets
  - Has no directory; does not require directory page lookup before accessing an entry page
  - Splits a bucket into two (chosen by round-robin strategy) whenever any overflow page is created
    - Every bucket is kept split evenly, rather than focusing on splitting heavy-use buckets
  - Bucket access is approximately 1.2 page accesses amortized (due to overflows)
- Components:
  - **Overflow page (linear hashing):** Data bucket which holds additional data entries matching the index, which have not yet been split into new buckets
  - Uses two hash algorithms at any given time; when the buckets are doubled, the old one is discarded and a new hash algorithm is added
  - **Level:** Index of the hash algorithms currently being used, and the number of times the buckets have been doubled
    - Begins at 0; increments without limit
    - Hash algorithm of a certain level x is denoted by h_x
  - **Next:** Beginning index of buckets in a linear hashing index which have not yet been split
    - Begins at 0; increments when a bucket is split; reset to 0 after splitting all buckets once
    - All buckets before next have been split in the current round-robin implementation
      - All buckets after or including next have not yet been split in the current round-robin implementation
- Diagram: See figure 17
- Operations:
  - Search:
    - Use h_{level} to find the index of the bucket where the value belongs
      - If the index is less than next, the search key belongs in the index found by $h_{\mathrm{level}+1}$ (as the bucket has been split)
      - If the index is greater than or equal to next, the search key belongs in this (as the bucket has not yet been split)
    - Search the selected bucket and any linked overflow pages
  - Insertion:
    - Apply the search process above to find the bucket to insert the data entry
    - Add the value to the bucket, creating an overflow page if necessary
    - If an overflow page is created:
      - Split the bucket at index next
      - Index the new bucket at the bottom of the others
      - Use $h_{\mathrm{level}+1}$ to redistribute the values between the old and new buckets
      - Increment next

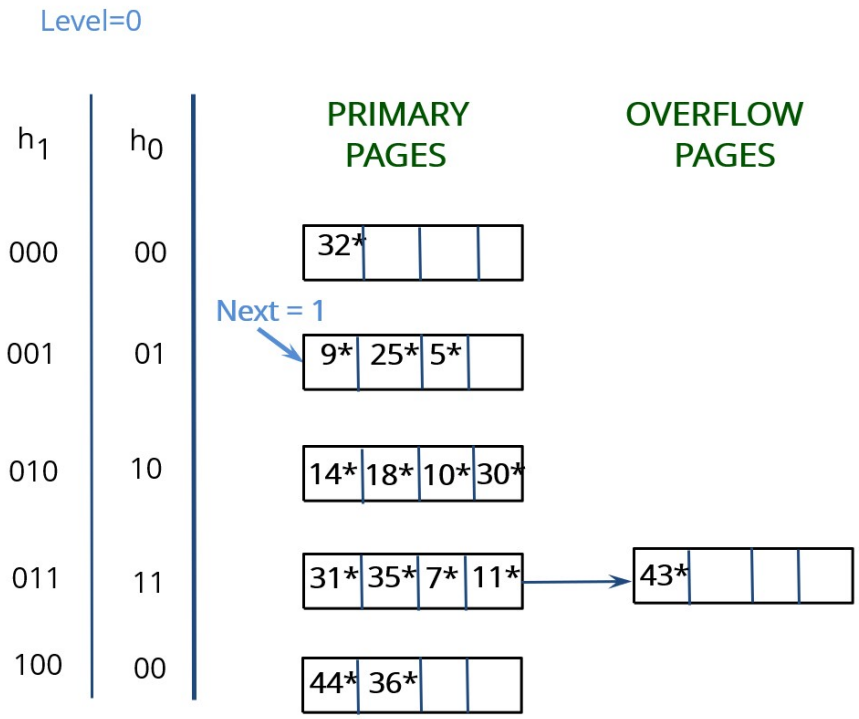


Figure 17: A Linear Hash Index

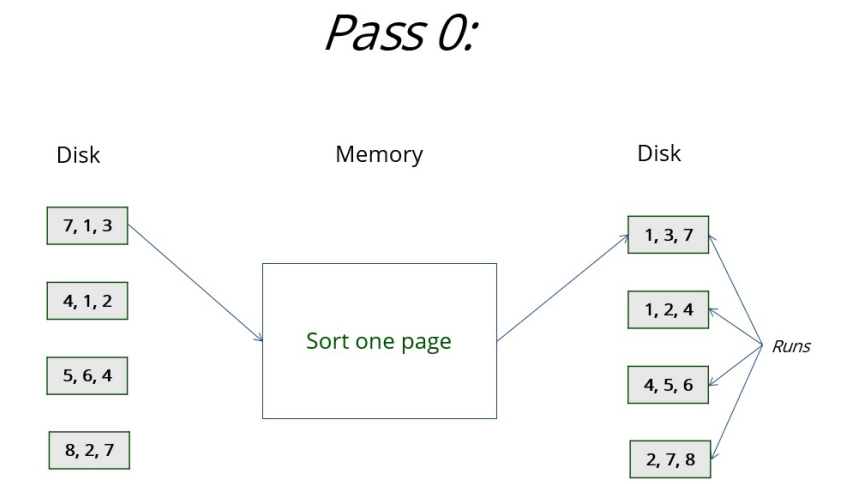
* If next has looped through all buckets once (not including the newly split buckets):
  - Discard hash algorithm h_{level}
  - Increment level
  - Create a new hash algorithm $h_{\mathrm{level}+1}$
  - Reset next to 0

5 External Sorting

- Sorting is essential for:
  - Outputting data in sorted order
  - Removing duplicates
  - Preparing data for B+ tree insertion
- B+ tree: If the data to be retrieved is clustered, then it is efficient to retrieve as sorted
- Evaluation strategy is chosen before query execution, for optimized performance
- **External sort:** Sorting algorithm which operates on primitive data pages but is not executed by the database
  - **Run:** Sorted subfile used during a larger sort strategy
  - **Input buffer:** Memory slot which holds data to be processed
  - **Output buffer:** Memory slot which holds data to be written to the disk

5.1 Merge Sort

- Merge sort: External sorting algorithm
  - Process:
    - On the first pass:
      - Read pages (up to the number of buffers)
      - Sort each page individually
      - Write pages back to disk
    - On subsequent passes:
      - Reserve a single buffer for output
      - Merge two 'runs' into a single run, in the output buffer
        - Given B available buffers, $B-1$ runs can be merged in a single pass
      - Write the output buffer back to disk
      - Repeat
    - Diagram: See figure 18
- Page I/O (given n data pages to sort):
  - n for the first pass
  - $2n$ for each subsequent pass (as each processed value is read once and written once)
- Buffer management:
  - Given n available buffers:
    - One buffer is used as an sorted output buffer (written to disk whenever full and emptied)
    - $n-1$ buffers are used to track one run each, from which data is fed into the output buffer
  - Merge as many buffers as possible during each pass, to minimize time



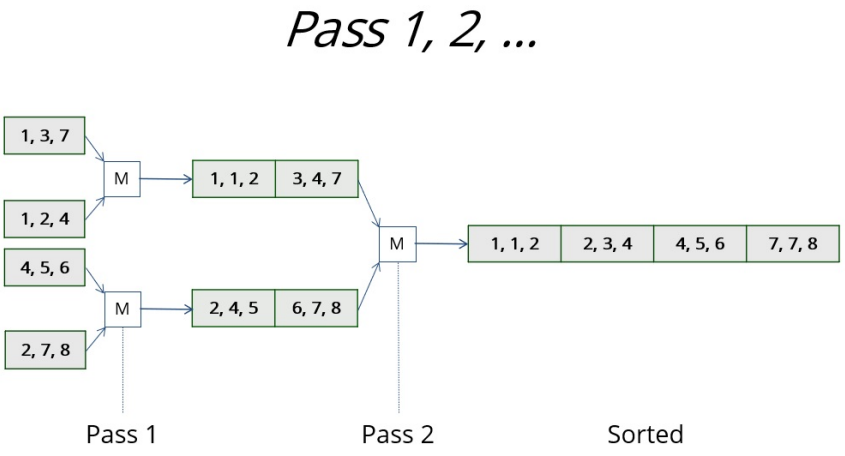


Figure 18: Diagram of the First and Subsequent Passes of a Merge Sort

5.2 Replacement Sort

- **Replacement sort:** External sorting algorithm where a sorted run is continuously built from a current working set of available data
  - Consists of one input buffer, one 'current set' of sortable data, and one output buffer
  - Process:
    * Whenever the input buffer is empty, fill it with a new page from the disk
    * Whenever the current set has an empty slot, fill it with data from the input buffer
    * From the current set, take the value which meets the following conditions and add it to the output buffer:
      - The minimum possible value in the set, and
      - Equal or greater than the maximum value in the output buffer
    * Whenever the output buffer is full, write it to disk and empty it
    * Repeat until all data has been read from the disk once (completing one round)
    * Repeat rounds until the current set is empty at the end of the round
- Diagram: See figure 19

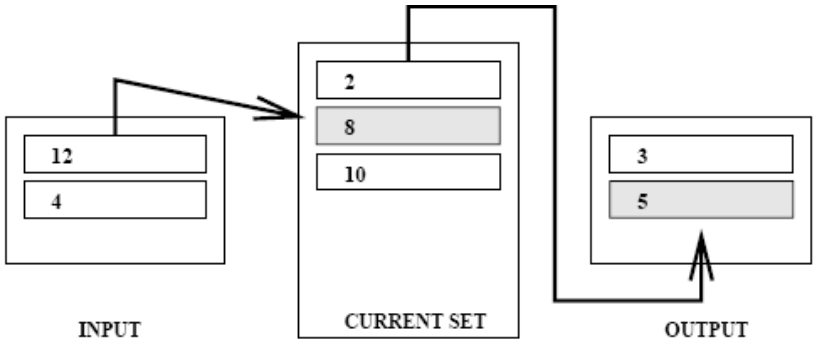


Figure 19: Diagram of the Replacement Sort Data Structure

6 Query Optimization

- Notation: Given a schema R:
  - The size in pages is denoted as $|R|$
  - The number of records per page is denoted as p_R
  - The size of the buffer is B
- Clustered indices are only applicable for terms which form a prefix of the search key
  - E.g. The index $< A, B, C >$ can be used for the search key $A = 1, B = 3$

6.1 Optimizing Equality Joins

- **Join:** Database operation where a single set of rows are returned, created from two relations
- E.g. $\text{SELECT * FROM Sailors S, Reserves R WHERE R.id} = \text{S.id}$
- **Simple Nested Loops (SNL) join:** Equality join implementation where nested loops compare equality values using 1-page buffers
  - Consists of multiple nested loops, each of which loops through a table
    - **Outer table:** Table in an SNL join which is scanned outside the other nested loop(s)
    - **Inner table:** table in an SNL join which is scanned inside the other nested loop(s)
  - Runtime:

$$|Outer| + |Outer| \cdot |Inner|$$

    - $|Outer|$ is the cost of loading each value from the outer table into memory
    - $|Outer| * |Inner|$ is the cost of loading each value from the inner table into memory, and comparing with an outer table value
    - To optimize, smaller tables should be implemented as the outer
- **Block Nested Loops (BNL) join:** Equality join implementation where nested loops compare equality values using multi-page buffers
  - **Block:** Fixed-size set of pages representing the size of a buffer
  - Runtime:

$$\begin{aligned} &|Outer| + |Outer\ Blocks| \cdot |Inner| \\ &= |Outer| + \left\lceil \frac{|Outer|}{B-2} \right\rceil \cdot |Inner| \end{aligned}$$

    - Two buffer pages are used; one as the output buffer and one as the input buffer for the inner table
    - To optimize, smaller tables should be implemented as the outer
- **Index Nested Loops (INL) join:** Equality join implementation where nested loops compare equality values, and inner loops use an index to search matching values from the outer loop
  - Runtime:

$$|Outer| + |Outer| \cdot p_{outer} \cdot \text{probing costs}$$

    - $|Outer| \cdot p_{outer}$ represents the number of records in table R
    - **Probing cost:** I/O cost to search for an entry and record using an index
      - Probing cost of data entries in a hash index: 1.2
      - Probing cost of data entries in a B+ tree: 3-4
      - Probing cost of data records when the search key is a candidate key: 1
      - Probing cost of data records when the search key is not a candidate key, in a clustered index: 1
      - Probing cost of data records when the search key is not a candidate key, in an unclustered index: $\frac{\text{records in inner}}{\text{records in outer}}$

- Equality with multiple attributes: Index must apply to one or more of the attributes
- Inequality: Index must be clustered B+ tree

- **Sort-merge-join (SMJ):** Equality join implementation where tables are sorted externally, written back to the disk, merged by equality, then sent to output
  - Runtime:
    - Sorting:

$$2 \cdot |T_1| \cdot (\left\lceil \log_{B-1}(\frac{|T_1|}{B}) \right\rceil + 1)$$
$$+2 \cdot |T_2| \cdot (\left\lceil \log_{B-1}(\frac{|T_2|}{B}) \right\rceil + 1)$$

      where T_1, T_2 are tables and B is the buffer size

    - Merging:

$$|T_1| + |T_2|$$

    - Optimization 1: Find the tables which have the fewest maximum count of any repeating value, then use those those tables as the outer loop (to fit as much outer loop data as possible into the input buffer)
    - Optimization 2: Join the runs of the sorted tables after only the first pass of the external sort
      - When joining, each run is allocated a single buffer
      - Only possible if $B \geq \sqrt{\max\{|T_1|, |T_2|\}}$
      - Runtime:

$$\begin{aligned}&(\text{Cost of pass } 0) + (\text{Merging all runs})\\&= 2(|R| + |S|) + (|R| + |S|)\\&= 3(|R| + |S|)\end{aligned}$$

  - Equality with multiple attributes: Sort based on all joined attributes
  - Inequality: Not applicable

- **In-memory probing by hash table:** CPU runtime optimization which hashes database table pages into buckets
  - Reduces excess page retrieval and optimizes access speed to specific pages

- **Hash-join (HJ):** Equality join implementation where tables are partitioned into hash buckets, then the values in each hash bucket is joined with the matching values in the hash bucket from the other table
  - Hash bucket notation:
    - T_i represents the contents of hash bucket i created by partitioning T
    - Assuming partitions are roughly equally sized: $|T_i| = \frac{|T|}{B-1}$
  - Tables can be hash-joined in one scan if and only if $B - 2 \geq \min\{|R_i|, |S_i|\}$ for all R_i, S_i given tables R, S
    - Implies $B \geq \sqrt{\min\{|R_i|, |S_i|\}}$

* If tables cannot be hash-joined in one scan, recursively apply hash-join to the bucket(s) which are too large to fit in memory

- Runtime (assuming R_i or S_i can fit in memory):

$$\begin{aligned}&\text{Cost of partitioning both relations - read and write} + \text{Cost of joining relations}\\&= 2(|R| + |S|) + (|R| + |S|)\\&= 3(|R| + |S|)\end{aligned}$$

- Parallelizable
- Sensitive to data skew
- Equality with multiple attributes: Hash based on all joined attributes
- Inequality: Not applicable

6.2 Optimizing Selections

- **Selection:** Database operation where a subset of rows are returned from a relation
- Notation:
  - **Reduction factor (RF):** Percent of data that a table is reduced by through a selection operation
- Size of result:

$$|T| \cdot \mathrm{RF}$$

- Runtime:
  - No index or not clustered on the attributes: $|T|$
  - Clustered on the attributes: $|T| \cdot \mathrm{RF}$
- Page I/O:
  - No index or not clustered on the attributes: $|T| \cdot \mathrm{RF} \cdot p_T$
    - Optimization: Collect and sort record IDs before retrieving them
  - Clustered on the attributes: $|T| \cdot \mathrm{RF}$
    - Clustered records are more likely to be on the same page
- Converted to conjunctive normal form (CNF) before processing
- Optimizing with multiple indices:
  - Approach 1: Retrieve a set of records using the most selective/restrictive index (minimizing returned values), then apply remaining terms
  - Approach 2: Retrieve sets of record IDs using all indices, intersect them, sort them, retrieve the records, then apply remaining terms

6.3 Optimizing Projections

- **Projection:** Database operation where a subset of columns are returned from a relation
  - Often includes a $\mathrm{DISTINCT}$ operation
- **Standard projection:** Projection implementation using a modified version of the external sort Pass 0
  - When writing during external sort Pass 0, drop unnecessary fields
  - During external sort merging of runs, drop duplicate rows
  - Page I/O:

$$\begin{aligned}&(\text{Reading all records}) + (\text{Writing partial columns})\\&+ (\text{Reading the same data})\\&+ (\text{Merging non-duplicate rows})\\=&|T| + |T| \cdot (\%\text{ of data projected}) * 2\\&+ |T| \cdot (\%\text{ of data projected}) \cdot (\%\text{ of distinct data merged})\end{aligned}$$

- **Hash projection:** Projection implementation where hashing is used to eliminate duplicates
  - Process: Use a hash function on all wanted attributes to partition the table into $B-1$ partitions (while discarding unprojected fields), use a new hash function on all fields of each partition to discard duplicates
- Index terms:
  - **Index-only projection:** Projection operation where all projected attributes are in the index's search key
    - Requires sorting afterwards if the projected attributes are not a prefix of the search key
    - Requires duplicate removal
  - **Index-only, no sort needed projection:** Projection operation where all projected attributes constitute a prefix of the index's search key
    - Requires duplicate removal

6.4 Optimizing Unions

- **Union:** Database operation where two relations with matching columns are combined
  - Includes removal of duplicates
- **Hash-based union:** Union implementation which utilizes two hash functions
  - Process:
    * Hash both relations into partitions
    * Re-hash a partition from the first relation, into new partitions
    * Re-hash the matching partition from the second relation, into the same re-partitions (avoiding adding duplicates)

6.5 Optimizing Differences

- **Difference:** Database operation between two relations where any row from the first relation which exists in the second relation is removed
- **Hash-based difference:** Difference implementation which utilizes two hash functions
  - Process:
    * Hash both relations into partitions
    * Re-hash a partition from the first relation, into new partitions
    * For each value in the second relation's matching partition:
      · Re-hash it to find the corresponding partition
      · Remove it from the partition if it exists

6.6 Optimizing Aggregation

- **Aggregation:** Database operation where numerical values in rows are combined
- If the index is clustered based on the attribute to be grouped, then no sorting will be required
- **Index-only aggregation:** Aggregation case where the search key contains all attributes to be searched and grouped, therefore no record access is required (only entries)

6.7 Combined Operations

- If grouping by a specific attribute:
  - Ensure the relation is clustered/sorted on the attribute (if not, sort it externally)
  - Use the relation with the attribute as the outer loop to preserve order

7 Fundamentals of Transactions

- Operations:
  - **Commit operation:** Database transaction operation which declares the transaction to be applied, even upon system crash
  - **End operation:** Database transaction operation which declares the transaction to be complete
- Notation; given transaction 1, page A, and LSN X:
  - Read operation: $R1(A)$
  - Write operation: $W1(A)$ or $T1UPDATEA$
    - Modifies RAM; writing to disk is determined separately by replacement policy
  - Abort operation: $T1ABORT$
  - Undo operation: $T1CLRLSNX, undonextLSN = X$
  - Commit operation: $C1$ or $T1COMMIT$
  - End operation: $T1END$
  - Begin/end checkpoints: $Begin_checkpoint$ and $End_checkpoint$
- **Transaction (Xact):** Single data access footprint of an SQL statement
  - Consists of a sequence of read, write, commit, and abort actions
  - Notation: Tx where x is a number signifying a specific transcation
    - Sequence of concurrent transactions: $T1T2\ldots$
  - **Transaction ID (XID):** Unique identifier for a transaction
  - **Atomic:** Transaction where either all read/write actions are executed or none are
- **Schedule:** Execution of multiple transactions where the order of reads/writes in each individual transaction are preserved
  - **Complete schedule:** Schedule where every transaction ends with either a commit or abort action
  - ACID (atomicity, consistency, isolation, durability) properties of schedules:
    - **Atomic schedule:** Schedule where all transactions are atomic
    - **Consistent schedule:** Schedule where, if the initial DB is in a consistent state, the resulting DB must also be in a consistent state
    - **Isolated schedule:** Schedule where an aborted transaction does not cause other transactions to be inconsistent
    - **Durable schedule:** Schedule where committed transactions must persist their write operations, regardless of system crash
- **Blind write:** Write operation on a given data item, which has no previous read operation on the data item in the transaction

8 Concurrency Control

- Concurrency control:
  - If atomicity is maintained and there is no system crash, then concurrency control guarantees consistency and isolation

8.1 Consistency

- **Serial schedule:** Consistent schedule where transactions are executed sequentially
  - Different transaction orders may create different DB states, all of which are acceptable
  - Example: Given transactions:

$$T1 : R1(A)\ W1(A)$$
$$T2 : R2(A)\ W2(A)$$

The two possible serial schedules are:

$$T1\ T2 = R1(A)\ W1(A)\ R2(A)\ W2(A)$$
$$T2\ T1 = R2(A)\ W2(A)\ R1(A)\ W1(A)$$

- **(View) Serializable schedule:** Consistent schedule where, for every data item, all transaction read and final write actions have the same effect as a serial schedule of the same transactions
  - Example: Given transactions:

$$T1 : R1(A)\ W1(A)\ R1(B)\ W1(B)$$
$$T2 : R2(B)\ W2(B)$$

A serializable schedule is:

$$R1(A)\ W1(A)\ R2(B)\ W2(B)\ R1(B)\ W1(B)$$

A non-serializable schedule is:

$$R1(A)\ W1(A)\ R1(B)\ R2(B)\ W1(B)\ W2(B)$$

  - Explanation: For each table in each transaction, the final write operation is dependent on all previous reads in the same transaction.
    Therefore, if for every data item in every transaction in a given schedule, the read operations and final write operation have the same effect as in a serial schedule, then the schedules are equivalent.
  - Manual method to determine if a schedule is serializable, given a set of transactions:
    - Write out all possible serial schedules
    - For each serial schedule possibility:
      - For each table:

        - In the test schedule and the current serial schedule, isolate the read operations and final write operation on the current table
        - Compare the isolated operations
        - If any operation does not have the same effect, then move to the next serial schedule possibility
      - If all tables' read and final write operations have the same effect as the current serial schedule, then the schedule is serializable
    * If no serial schedule possibility matches, then the schedule is not serializable

– **Conflict-based testing:** Method to determine if a schedule is serializable through analyzing its conflict actions

  * Actions from an aborted transaction are ignored when calculating conflicts

  * **Conflict actions:** Two subsequent read/write actions (at least one write) from different transactions (in a schedule) which operate on the same data item

    - Every conflict action **induces** the order of transactions to match the order in the two operations
    - Types of conflict actions:
      - If the operations are write then read:
        Given operations $W1\ R2$, **transaction** $T2$ **reads the effect of transaction** $T1$
      - If the operations are read then write:
        Given operations $R1\ W2$, **transaction** $T1$ **does not read the effect of transaction** $T2$
      - If the operations are both write:
        Given operations $W1\ W2$, **transaction** $T2$ **overwrites transaction** $T1$

  * If and only if all conflict actions in a schedule induce the same priority of transactions, then the schedule is serializable

  * Precedence graph is created by analyzing each conflict action and adding directional edges to transactions in their induced orders

    - An acyclic precedence graph can be navigated in any topological order to create a serial schedule

  * **Conflict-serializable schedule:** Consistent serializable schedule which has an acyclic precedence graph

    - Subset of serializable schedules

8.2 Isolation

- **Isolation:** Property of a transaction where, if it is aborted, other transactions should not be affected
  - **Dependent:** Transaction which reads from or writes to a data item, with the result dependent on the write action of another transaction
    - Conflict actions which create a dependency: WW or WR
    - E.g. If the order of actions is $W1\ R2$, then $T2$ depends on $T1$
- **Cascading abort:** Abort action which causes a different transaction to also abort
  - Invalidates the isolation property
  - E.g. Given concurrent transactions $T1$ and $T2$:
    - $W1\ R2\ ABORT(1)$ means that the value before $W1$ must be restored, which invalidates $R2$. Therefore, $T2$ must be aborted in addition to $T1$.
    - $W1\ W2\ ABORT(1)$ means that the value before $W1$ must be restored, which loses the write by $W2$. Therefore, $T2$ must be aborted in addition to $T1$.
- **Recoverable schedule:** Isolated schedule where a transaction can only commit after the transactions it depends on have committed
  - Cascading aborts are possible, but are safe to do
  - **Avoid Cascading Aborts (ACA) schedule:** Isolated schedule which avoids cascading aborts by waiting to access a resource with an uncommitted write, until the action is committed
    - Subset of recoverable schedules
    - All dependencies are replaced with waits
    - The only conflict action allowed is RW
      - If a schedule has WW or WR conflict actions, then it violates ACA

8.3 Enforcing Consistency and Isolation

- Resource locks:
  - **Share lock (S-lock):** Lock on a resource during a read operation
    - Notation: $S(T)$
  - **Exclusive lock (X-lock):** Lock on a resource during a write operation
    - Notation: $X(T)$
    - **Upgrade:** Action where a lock is changed from share to exclusive, due to a transaction requesting an exclusive lock while holding a share lock on the same resource
  - Unlock notation: $U(T)$
  - Locking by itself does not guarantee serializability
  - **Lock table:** Hash table of lock entries
    - **Lock entry:** Data structure representing a single resource which maintains a list of transactions holding locks and a wait queue
    - **On lock release (OID/XID):** Operation where the next available locks in the wait queue are processed
- **2PL:** Resource access lock protocol which ensures conflict serializability (consistency)
  - Rules: For each transaction:
    - Before reading, a share lock must be activated
    - Before writing, an exclusive lock must be activated
    - All unlocks must happen after all lock operations
      - **Lock point:** Time in a schedule at which all required locks have been acquired by a transaction
  - Does not enforce isolation
- **Strict 2PL:** Resource access lock protocol which ensures conflict serializability (consistency) and ACA (isolation)
  - Rules: For each transaction:
    - Before reading, a share lock must be activated
    - Before writing, an exclusive lock must be activated
    - Unlocks only happen when committing
  - E.g. Schedule which violates string 2PL: $R2(A)\ W1(A)\ W2(A)$

8.4 Deadlocks

- **Deadlock:** Situation where two transactions hold locks while waiting for the other transaction to release their lock
  - Can occur in 2PL or strict 2PL

8.4.1 Detection Strategies

- **Waits-for-graph:** Deadlock detection strategy where a directed graph is maintained between transactions waiting for another transaction to release a lock, and if a cycle (deadlock) is detected, a transaction in the cycle is aborted
  - Aborted transaction is chosen by criteria such as fewest locks, least work done, farthest from completion, etc.
- **Timeout:** Deadlock detection strategy where a transaction which exceeds a certain duration is aborted

8.4.2 Prevention Strategies

- Each transaction is assigned a unique timestamp; waits can only occur in one temporal direction
  - **Wait-die:** Deadlock prevention strategy where older transactions may wait for newer transactions
  - **Wound-wait:** Deadlock prevention strategy where newer transactions may wait for older transactions
  - If a transaction violates this, the newer transaction is aborted and restarted after some time with the original timestamp

8.5 B+ Tree Concurrency

- B+ tree indices also require locking
- Locking a node also locks the subtree
- **Crabbing locking:** B+ tree locking strategy which releases ancestor locks as child locks are obtained, 'crabbing' down the tree
  - Search: Release a lock on a node when a child node of it is locked
  - Insert/delete: Release a lock on any ancestor nodes when a child node of it is locked, and the child is not full
    - Reason: Any split in the index will propagate to the non-full child, then stop

9 Logging and Recovery

- **Logging and recovery:** Strategy to ensure database atomicity and durability
  - If consistency and isolation are maintained (e.g. through strict 2PL), then logging and recovery guarantees Atomicity and Durability
  - **Recovery manager:** Component which manages logging and recovery
  - Example logs:

```
10 T1 UPDATE P1
20 T2 UPDATE P2
30 T1 UPDATE P3 (STEAL)
40 T1 COMMIT
50 T2 UPDATE P3
60 T1 END
70 T2 COMMIT
80 T2 END
```

- Atomicity:
  - Aborted transactions should have their effects nullified, and should not affect other transactions
- Durability:
  - Committed transactions (whether written to disk or not) should persist
  - Uncommitted transactions should be rolled back
  - Assumes that the buffer-pool is updated on writes, and write to disk only occurs occasionally
    - **Buffer-pool:** Main memory holding the frames of data currently used by the database

9.1 Solutions for Atomicity and Durability

- **Force:** Solution for durability where every write operation is immediately written to disk
  - Response time is slow
- **Steal:** Solution for atomicity where dirty buffer frames in use by in-progress, uncommitted transactions are written to disk to free up the frame
  - Example logs of a steal:

```
10 T1 UPDATE P1 (STEAL)
```

  - Before writing a stolen page to disk, the old disk page is logged
  - If the transaction which was stolen from is aborted, then the disk write must be rolled back to the saved logged value
  - Increases throughput
  - Flushes the log tail (see subsection 9.2)
- Optimal solution: **Steal/No Force policy**
  - Before writing a stolen page to disk, the old disk page is logged
  - Before committing a transaction, the value of the dirty page is logged
  - Upon a system crash and restart:

    - Crashed transactions must UNDO all changes by restoring from the saved logged pages (Steal strategy)
    - Committed transactions which had dirty pages must REDO all dirty pages and the commit by restoring from the saved logged pages (No Force strategy)

- **Write-Ahead Logging (WAL):** Logging strategy for recovery where all changes are logged, and the log buffer is flushed to the log just before stealing a dirty page or committing a page

9.2 Transaction Log Information

- **Logging:** Strategy where each write operation also creates REDO and UNDO information in a log file
  - Must use diff information
  - Must be safe from crashes
- **Log tail:** Memory buffer used to hold data before writing to a log
  - Flushed upon:
    - A steal
    - A commit action
    - $\mathrm{end\_checkpoint}$ (see below)
- **Log record:** Line of data in a log representing a single change
  - Format: $\langle \mathrm{XID}, \mathrm{pageID}, \mathrm{offset}, \mathrm{length}, \mathrm{old\ data}, \mathrm{new\ data} \rangle$
  - Significantly smaller than entire pages; represents a change made to a page
  - **Log Sequence Number (LSN):** Value which uniquely identifies a log record
    - **flushedLSN:** Value identifying the last-flushed log record from the log tail
    - **pageLSN:** LSN value on a disk page representing the most recent update
- **Transaction Table (TT):** Table containing a record for each active transaction
  - Records are:
    - Added upon first execution of a transaction's operation
    - Updated upon a transaction's subsequent operations (including commits)
    - Removed upon execution of the END operation
  - Format: $\langle \mathrm{XID}, \mathrm{status}, \mathrm{lastLSN} \rangle$
    - Status: Transaction status (uncommitted/committed/aborted)
    - **lastLSN:** LSN value representing the most recent log record when the transaction executed a write action
  - Example of a TT: See figure 20

Figure 20: Example of a Transaction Table

XID	Status	lastLSN
T1	U	100
T2	C	80

- **Dirty Page Table (DPT):** Table containing a record for each dirty page
  - Records are:
    - Added upon the modification of a non-dirty page
    - Removed upon STEALing the page or upon writing the page to disk
  - Format: $\langle \mathrm{pageID}, \mathrm{recLSN} \rangle$
    - **recLSN:** LSN value representing the first log record which caused the page to become dirty

Figure 21: Example of a Dirty Page Table

Page	recLSN
P1	50
P3	70

  - Example of a DPT: See figure 21

- **Checkpoint (chkpt):** Regular interval at which the Transaction Table and Dirty Page Table are written to disk for crash prevention
  - Limits the amount of log required to analyze upon recovery
  - **begin_checkpoint:** Log record signifying the time at which the TT and DPT were saved
  - **end_checkpoint:** Log record signifying at which time the checkpoint was finished being written to disk
    - Flushes the log tail
  - Checkpoint LSNs are logged - for example:

```
10 T1 UPDATE P1
20 begin_checkpoint
30 T1 UPDATE P2
40 end_checkpoint
```

9.3 Aborting

- **Abort:** Action where an in-progress, uncommitted transaction is reverted
- Procedure to abort a single transaction T:
  - Write a $\mathrm{T\ ABORT}$ record to log
  - From the TT, retrieve T's $\mathrm{lastLSN}$
  - For each update starting at $\mathrm{lastLSN}$:
    * Undo the update to disk
    * Log the UNDO action as $T\_CLRLSN\_undonextLSN = \_$
      · **CLR:** Log record value in an ABORT/UNDO action which states that a specified LSN has been undone
    * Take the $\mathrm{prevLSN}$ from the undone action to set the $\mathrm{undonextLSN}$ value
      · **undonextLSN:** Log record value which indicates the next LSN to be undone
    * Update the $\mathrm{pageLSN}$ in the Transaction Table
    * Queue another action to undo $\mathrm{undonextLSN}$
  - If there are no more actions to undo:
    * Release all locks held
    * Write a $\mathrm{T\ END}$ record to log
- Example logs from an abort:

```
10 T1 UPDATE P1
20 T1 UPDATE P2
30 T1 ABORT
40 T1 CLR LSN 20, undonextLSN=10
50 T1 CLR LSN 10, undonextLSN=nil
60 T1 END
```

9.4 Recovering from Crashes

- **Crash:** Loss of power which causes everything in memory (log tail, TT, DPT, and dirty pages) to be lost
  - Flushed log, master record, and written data pages survive a crash
  - **Repeated crash:** Subsequent crash which occurs during recovery of an initial crash
- **Recovery:** Crash restart process where the TT and DPT are restored, all active transactions are redone, and all uncommitted transactions are undone
- Procedures in order: ANALYZE, REDO, UNDO
  - Only UNDO will add new log records
- **ANALYZE:** Recovery procedure which restores the TT and DPT from the most recent $\mathrm{begin\_checkpoint}$ (logged on the master record) and reconstructs the remaining unsaved changes using the logged actions
  - Difference from the TT and DPT lost at crash: Steals are not reflected in the reconstructed DPT
- **REDO:** Recovery procedure which begins from the smallest recLSN in the DPT and re-dos all logged actions
  - Applies actions to disk
  - Applies to all actions, even for aborted transactions
  - No logs are added/modified, only data actions are executed
  - No redo is executed (i.e. an update has already been written to disk) if any of the following are true:
    - The page is not in the DPT
    - The recLSN in the DPT is greater than the current LSN
    - The $\mathrm{pageLSN}$ of the page is greater or equal to the current LSN
  - When redoing a CLR record:
    - Reapply the UNDO from the record to the data
    - Update $\mathrm{pageLSN}$ to the CLR record to avoid repeated redos

- **UNDO:** Recovery procedure for a single uncommitted transaction which begins from the $\mathrm{lastLSN}$ in the TT, and un-dos all actions in the transaction
  - **Loser:** Transaction which was not yet committed when a crash happened, and requires rolling back
  - Transactions are undone starting from the greatest $\mathrm{lastLSN}$
  - Procedure to UNDO transactions:
    * Create a list of LSNs ToUndo
    * For each transaction which was uncommitted when the crash happened, add its $\mathrm{lastLSN}$ to ToUndo
    * Until ToUndo is empty, take the largest LSN:
      - If the action is an Abort, add its $\mathrm{prevLSN}$ to ToUndo
      - If the action is an Update:
        - Undo the update on disk
        - Write CLR as the new log record
        - If the undone record's $\mathrm{prevLSN}$ is not null, add it to ToUndo
      - If the action is a CLR record:
        - If $\mathrm{undoNextLSN}$ is null, then write the END record for this transaction
        - If $\mathrm{undoNextLSN}$ is not null, then add the $\mathrm{undoNextLSN}$ record to ToUndo
  - Example logs from UNDO:

```
110 CLR T1 UNDO LSN = 100, undoNextLSN =10
120 CLR T2 UNDO LSN = 90, undoNextLSN = 50
130 CLR T2 UNDO LSN = 50, undoNestLSN = 20
```

- Optimization:
  - To reduce work required by the ANALYSIS phase, save DPT and TT checkpoints more frequently
  - To reduce work required by the REDO phase, flush pages to disk often and asynchronously
  - To reduce work required by the UNDO phase, avoid long-running transactions (e.g. break up into shorter transactions)
- **Media recovery:** Recovery of data from a corrupted database object
  - Process:
    * Occasionally copy a data item
    * Upon corruption, restore the copy and use checkpoints and logs to update
      - Redo actions of committed transactions
      - Undo actions of uncommitted transactions
  - Regularly copying a data item is expensive