More from Lecture 2
Summary:

- Databases is stored as a series of pages, tuples organized within those pages
- OLTP (Row store)
- OLAP (Column store)

N-ary storage model (NSM)

- Stores all attributes for a single tuple contiguously in a page. Works well for OLTP
- But, this can also make us read a lot of unwanted stuff when we have a query.

So use the Decomposition storage model (DSM)

- Stores values of a single attribute for all tuples contiguously in a page.
- Good for OLAP where read-only queries perform large scans over a subset of attributes (not interested in all the attributes)
- Store the info for each column in a DSM Disk page (e.g. one page has all of the hostnames, one has all of the userIDs, etc.)
- Then the query only needs to read a couple disk pages instead of all of them.
- If you want to reconstruct, you can go through each page and get the x-th offset (so each attribute of a tuple will be the same offset on each of the attribute pages).
- Advantages:
    - Reduces the amount of wasted I/O because it only reads the data it needs.
    - Better query processing and data compression
- Disadvantages:
    - Slow for point queries, inserts, updates, and deletes because of tuple splitting/stitching
    - For example, if you want to read just one tuple, you have to read through multiple tables.
    - And for writing, you have to write to a bunch of different pages.

Observation about DSM vs. NSM:

- I/O is the main bottleneck if the DBMS fetches data from disk during query execution. So you want to minimize the amount of data you read/write
- The DBMS can compress pages to increase the utility of the data moved per I/O operation (like DSM does) e.g. compress repeated values
- Tradeoff: speed vs compression ratio
    - Compressing the database reduces DRAM requirements
    - It may decrease CPU costs during query execution

Lecture 3 (1/13) - Memory Management

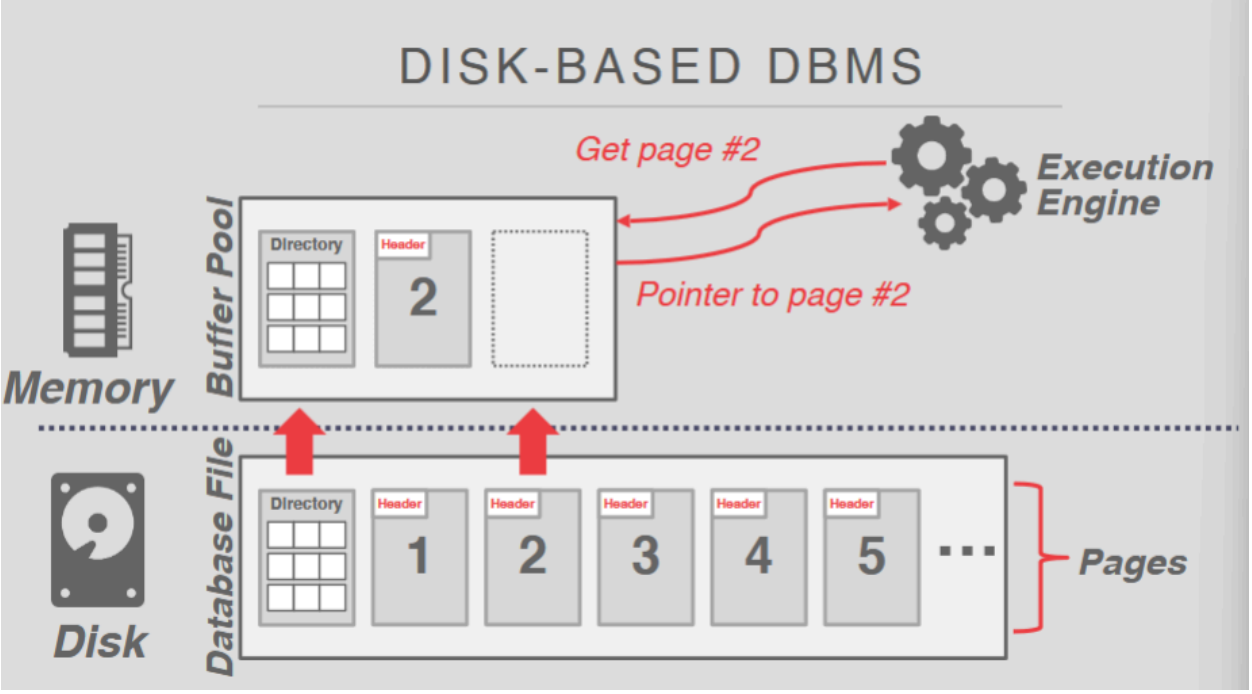
(Last time was problem 1: how the DBMS represents database in files on the disk)
This time is problem 2: how the DBMS manages memory and transfers data to/from disk.

Two storage topics:

- Spatial control:

    - Where to write pages on disk, layout of table; disks read sequentially (so avoid random reads)
    - The goal is to keep pages that are often accessed together physically close on disk.
- Temporal control:
    - When to read from memory, when to write to disk. Best way to bring multiple pages together (e.g. load page with account info and page with payment info at the same time to avoid I/O stalls)
    - Minimize stalls from having to perform disk I/O.

Recall: Query Execution



1. Query comes in, execution engine sends request for page 2.
2. Reads directory into buffer pool to lookup where page 2 is.
3. Read page 2 into buffer pool (can go in whatever free spot in buffer pool).
4. Return pointer to page 2 in the buffer pool.
5. Once we're done, it'll eventually get evicted.

It's only guaranteed that page 2 will still be in buffer pool when the execution engine is using in. It'll be locked (mutex to prevent buffer pool from moving it). But after execution engine is done, unlocked so free to move it.

Why not use the OS?

- The DBMS can use memory mapping (mmap) to store the contents of a file into the address space of a program; the OS is responsible for moving pages of the file in and out, so DBMS doesn't need to do it, doesn't need to deal with the locking.
- However, consider this example:

    - You have on-disk file with 4 pages. You have the virtual memory, so when you request a page, the OS puts it into the physical memory.
    - When physical memory fills up, needs to swap one of them for page 2. At the application layer, it's gonna see a page fault so one of them is gonna get evicted and the application will stop while the page swapping happens.
    - So you can't predict which pages are needed, which leads to arbitrary stalls.
- If you try to do multiple threads to hide page fault stalls, issues with concurrency (multiple writers).

Four main problems of memory mapped I/O (above)

1. Transaction Safety: OS can flush dirty pages at any time. When a transaction makes updates to data, you need to make sure that all of the writes happen or none of them happen. There's no guarantee that the OS will keep the changes.
2. I/O Stalls: OS doesn't know which pages are in memory; stall a thread on page fault.
3. Error Handling: difficult to validate pages. Any access can cause a SIGBUS that the DBMS must handle. Don't know when to perform validation.
4. Performance Issues: TLB (Translate between virtual and physical) shootdowns (invalidating cache entries). This is an expensive operation since all cores need to invalidate the entry in their TLB. OS data structure contention. Also, OS doesn't know about the DBMS's temporal locality (e.g. in B+Tree, need to access root every time, but OS doesn't know that)

So the DBMS should control things itself:

- Flushing dirty pages to disk in the correct order.
- Specialized prefetching.
- Buffer replacement policy.
- Thread/process scheduling.

Buffer Pool Manager

- Buffer pool organization (a buffer pool is basically a cache)
    - Memory region organized as an array of fixed-size pages. An array entry is called a frame (slots for the pages to go in)
    - When the DBMS requests a page, an exact copy is placed into one of the frames.
    - When we update, the writes in a page are buffered in memory (called a write-back cache, not written to disk immediately, only when evicting it from buffer pool; as opposed to a write-through goes to disk immediately).
- Buffer pool metadata: the page table keeps track of pages that are currently in memory (managed inside of the buffer pool, not the same page table as the OS). Maps page id to where it is located, returns a pointer to it.
    - Contains metadata (dirty flag, pin/reference counter)
    - When the execution engine is working on page 3, we lock it
- Page table vs. page directory
    - Directory is mapping from page ids to page locations in database files. Changes must be recorded on disk to allow the DBMS to find them on restart

- Table maps page ids to a copy of the page in buffer pool frames. So this is an in-memory data structure that doesn't need to be stored on disk (so if restarts, doesn't have it anymore)

Replacement Policies

(how to decide what to evict)

- Considerations:
  - Correctness (don't evict stuff that isn't committed, wait until it's persisted)
  - Accuracy
  - Speed
  - Meta-data overhead (how much metadata do we need to manage this?)

Basic policies:

- LRU: Maintains the timestamp for when page was last accessed. E.g. linked list sorted by time stamps.
  - Problems:
    - Cyclical access pattern causes issues
    - Keeping list updated and sorted is slow
    - Space to maintain the data structure (Additional metadata/overhead)
- Clock (approximation of LRU that doesn't need separate timestamp)
  - Each page has a reference bit. When a page is accessed, set to 1.
  - Organize pages in a circular buffer with a clock hand. Whenever, buffer is full, start moving clock hand. If a page's bit is set to 1, set to 0, go to next. If it's 0, evict (we haven't seen it recently).
  - Problems:
    - Susceptible to sequential flooding: a query performs a sequential scan that reads every page. Pollutes the buffer pool with pages that are read once and then never again.
    - Example of sequential flooding: if you have a tuple that gets read frequently, you keep evicting and then re reading the tuple for this query.

Better ones:

- LRU-K: track the history of the K references (K most recent accesses) to each page as timestamps and compute the interval between subsequent accesses. Then use this history to estimate the next time that page is going to be accessed.
  - Idea is to keep the pages with small intervals between their accesses (i.e., accessed more frequently)
- Localization: choosing on a per-query basis. maintains a ring buffer that is private to the query (keep track of the pages that a query has accessed). Minimizes the pollution of the buffer pool from each query
- Priority hints: DBMS knows the context of each page during query execution. Provides hints to the buffer pool about whether or not a page is important. (e.g. if your pages are organized in a tree, you know that you'll have to traverse the root, etc. to get to other pages).

Other Optimizations

- Dirty page eviction
    - Fast path: if hasn't been modified, i.e., not dirty, drop it.
    - Slow path: if it's dirty, then DBMS must write it back to disk to ensure that its changes are persisted.
    - Trade-off between fast evictions vs. writing dirty pages that will not be read again in the future.
        - If we do fast path too much, the buffer pool will fill up with a bunch of dirty pages that you don't evict. And then thrashing will occur with the ones that are not dirty.
        - So we prefer slow path
- Background writing
    - DBMS periodically walk through the page table and write dirty pages to disk.
- OS Page Cache (Avoiding the OS):
- Multiple Buffer pools: improves locality and reduces latch contention. Two approaches to implement
    - Object ID: maintain a mapping from objects to specific buffer pools.
    - Hashing: hash the page id
- Pre-fetching: DBMS knows the access pattern (OS doesn't know this) based on a query plan. Prefetch the next ones from memory so it's ready.
    - Sequential scans
    - Index scans
- Scan sharing (synchronized scan): If two queries are accessing the same data, retrieve the data at the same time.

Conclusion:

- DBMS manages memory better than OS.
- LEverate semantics of the query plan to make better decisions:
    - Evictions
    - Allocations
    - PRe-fetching

Lecture 4 (1/15) - Hash Tables

Private cloned repo, not fork.

- Part of access methods (consists of two data structures, hash tables and trees)
- Regarding how to support the DBMS's execution engine to efficiently read/write data from pages.
- Where data structures are used:
    - Internal metadata
    - Core data storage
    - Temporary data structures

- Table indexes

Design Decisions:

- Data organization: how to lay out data structure and what info to store.
- Concurrency: how to enable multiple threads to access.

Hash Tables

A hash table implements an unordered associative array that maps keys to values.

- Uses a hash function to compute an offset into this array for a given key, then can find value.
- Space complexity: O(n)
- Time complexity: Average O(1), worst O(n) (but there's overhead that means these complexities are not correct in reality, we need to look at the constants).
- E.g., with a hash function, it only does point lookups, not a range. So we often use trees.

Static Hash Table: Allocate a giant array that has one slot for every element. To find an entry, mod by number of elements and that will be its offset.

- Assumes we know the number of elements ahead of time and that it's fixed.
- Assumes that all keys are unique.
- Assumes that all of the keys will map to unique spots. (i.e., perfect hash function).

Design decision for hash tables:

1. Hash function: how to map a large key space into a smaller domain. Tradeoff between speed and collision rate.
2. Hashing scheme: once we have a collision, how do we resolve that? Tradeoff between allocating a large hash table vs. additional instructions to grow it.

Hash Functions

For any input key, return an integer representation of that key. Want fast and low collision.
Performance also depends on alignment sizes e.g. cache sizes, register sizes, page sizes, etc.

Static Hashing Schemes

(ways of approaching collisions)

<u>Approach 1: Linear Probe Hashing</u>

Single giant table of slots. Resolve collisions by linearly searching for the next free slot.

- In order to know whether we're finished or not, we need to store the key in the index.
- Deletes: hash the value, find the slot that it was supposed to go to, if it's not that, scan forward until you get there and delete. Then, there's an empty slot there, which means the next time we look up, we'll think we're done and stop our scanning here. To deal with this (2 options):
    1. Re-hash: update all the slots for all keys. Not efficient, need to redo everything that came after the one that got deleted.

2. Tombstone (this is done in practice): set a marker indicating that the entry has been logically deleted. You can insert here if you want; if you are doing a look-up, you have to keep going. However, may require periodic garbage collection (e.g. if you delete a bunch of stuff without doing any inserting).

- What to do with non-unique keys.
    1. Separate linked lists: map from key to a value list (so you store the values the key can take on in a separate list)
    2. Redundant keys: store duplicate keys entries together in the hash table. Easier to implement, but lookup will need to scan more to find all potential iterations of that key rather than a single one.

Approach 2: Robin Hood Hashing

Variant of linear probe that steals slots from "rich" keys (those who are closer to the slots that they were originally hashed to) to "poor" keys.

- Each key tracks the number of positions they are from its optimal position in the table.
- On insert, a key takes the slot of another key if the first key is farther away from its optimal position than the second key (do a swap)
- Will balance the length of the linear scan you have to do to find each key.

Approach 3: Cuckoo Hashing

Use multiple hash tables with different hash function seeds.

- On insert, check every table and pick anyone that has a free slot. (If there's a hashing collision, check the next hash table).
- If no table has a free slot, evict the element from one of them and then re-hash it to find a new location. (e.g. if A is in table 1 and B is in table 2 and C wants to go in those spots, evict B and put C there. Then rehash B, but since that goes where A is, evict A and replace it with B. Then A goes in its spot in table 2). If you get a cycle, have to change something about the structure (e.g. number of hash tables, hash function, etc.)
- Lookups and deletions are always O(1) because only one location per hash table is checked (stuff will always go into a slot that it should be in in one of the tables).

Dynamic Hashing Schemes

The static hashing schemes require the DBMS to know the number of elements it wants to store. Otherwise, it needs to grow/shrink the size and rebuild the table.

Dynamic hash tables resize themselves on demand.

Chained Hashing

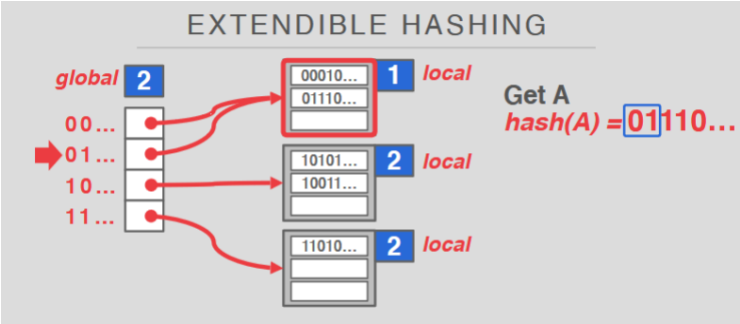
Maintain a linked list of buckets for each slot in the hash table. (Each slot is a pointer to a linked list).

- Resolve collisions by placing all elements with the same hash key into the same bucket.
- You can keep growing the linked lists.

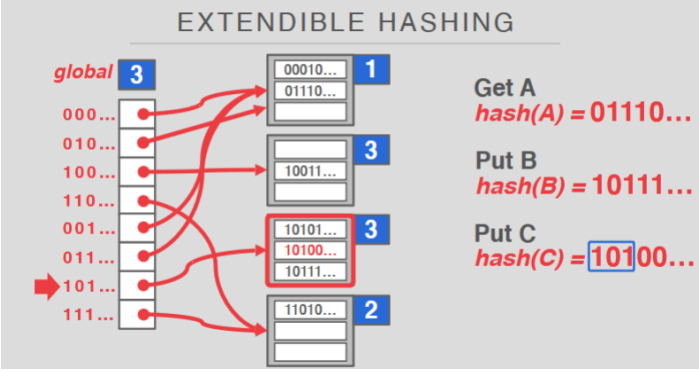
Extendible Hashing

Chained hashing approach where we split buckets instead of letting the linked list grow forever.

- So multiple slot locations can point to the same bucket chain. Idea is to rebalance the size of the linked lists.
- Reshuffle bucket entries on split and increase the number of bits to examine.



- Global counter is how many bits we consider to figure out which bucket it goes to.
- However, if we try to insert but the bucket is already full, increase the global counter (expand the size of the bucket lookup table). Split the bucket that's overflowing.



- Local counter refers to how many bits you look at to get there. For example, in the above, anything that starts with 100 goes to the second bucket (with local counter 3), and anything that starts with 11 goes to the bottom bucket.

Linear Hashing

The hash table maintains a pointer that tracks the next bucket to split.

- If a bucket overflows, split the bucket at the pointer location.
- Make a second hash function (same as the first hash function but maps to the offset). For anything that maps to something before the split pointer, you must use the second hash function.
- When the split pointer reaches the last slot, you can delete the first hash function and move back to the beginning.

Lecture 5 (1/22) - B+Tree Index

Cheat sheet for exam 1/29

Recall the data structures relevant to DBMS:

- Internal meta-data

- Core data storage
- Temporary data structures
- Table indexes (e.g. storing keys and values): a table index is a replica of a subset of a table's attributes that are organized and/or sorted for efficient access using those attributes.
    - Need to ensure that the contents of the table and the index (watch for concurrency issues).
    - DBMS figures out the best indexes to use to execute each query.
    - Trade-off: number of indexes to create per database (storage overhead and maintenance overhead). E.g. if you have more read queries, ok to do more indexes, but if you're doing a lot of writing and updating, too difficult to maintain.

B+Tree Overview

A B+tree is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions (i.e., both point and range queries) in O(log n).

- Generalization of a binary search tree, since each node can have more than two children (M children).
- Optimized for systems that read and write large blocks of data.

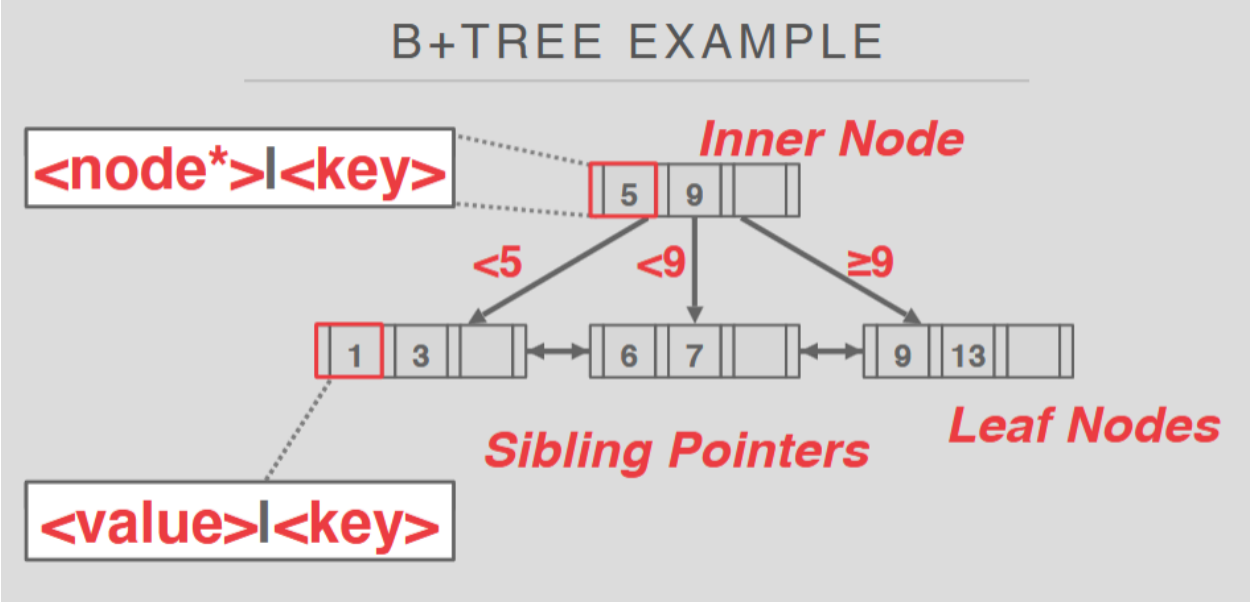
Properties:

A B+Tree is an M-way search tree with the following properties (where M is the number of children):

- Perfectly balanced (i.e., every leaf node is at the same depth).
- Every node other than the root is at least half-full (M/2 - 1 <= #keys <= M-1). Exception for root node because if you have a small tree, root might not be half full. If you have workloads with a lot of insertions and deletions, you're gonna have to reorganize the tree a lot, so it might be better to set the threshold lower than ½ so we don't have to keep reorganizing.
- Every inner node with k keys has k+1 non-null children.

Use in a DBMS

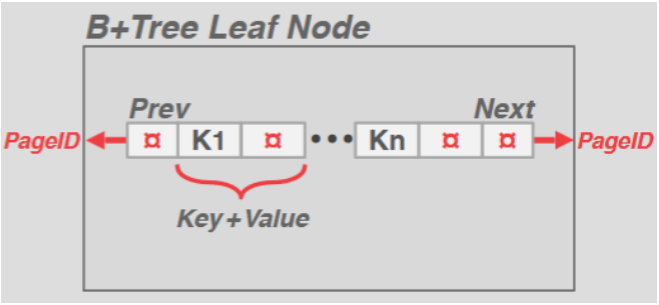
Leaf Nodes:



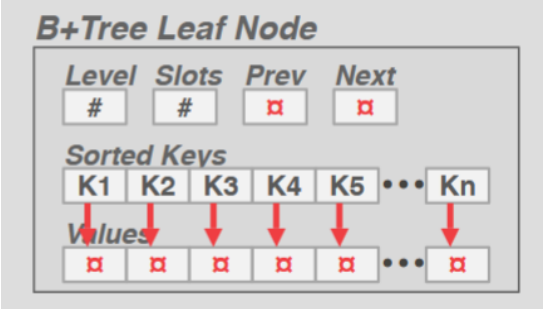
- k=2 keys in the root partition the space into k+1=3.
- Values only occur in the leaf nodes (inner nodes have key / node pointer pairs whereas leaf nodes have key / value pairs).

Every B+Tree node consists of an array of key/value pairs.

- Inner nodes have key/pointer pairs, leaf nodes have key/value pairs
- Keys are derived from the attributes that the index is based on.
- Arrays are usually kept in sorted key order. Sorted is good for finding value in the array (for read-heavy); unsorted is good since you don't need to reorganize each time to maintain sorted (for write-heavy)



- The sibling pointers are the ones on either side; not physical memory pointers, but just pointers to the pageID.



- Level is how deep, slots is how many slots are free
- Prev and next are sibling pointers
- We do separate key/value arrays because we don't always need to see the corresponding values. E.g. searching for the 5th key, the corresponding value would be in the 5th position of the value array.

Leaf Node Values:

- Approach 1: record IDs
    - A pointer to the location of the tuple to which the index entry corresponds.
    - E.g. key 42 maps to record 123 in the database
- Approach 2: tuple data
    - The leaf nodes store the actual contents of the tuple.
    - Secondary indexes must store the record ID as their values.
- Trade-off: if you stored record IDs, it takes longer but less stuff to store. Also, for approach 2, you need to make sure the data in the tuples and in the nodes match.

<u>B-Tree vs. B+Tree:</u>

- B-Tree stores keys and values in all nodes of the trees; more space-efficient, since each key only appears once in the tree. Downside is that traversal is not necessarily in order (need to jump up and down from parent to child).
- B+Tree only stores values in leaf nodes (e.g. in the previous example, 5 and 9 don't have values) and inner nodes only guide the search process; so if you want to do sequential scan, you can just go → through the leaf nodes.

<u>Insertion and Deletion:</u>

Insert:

- Find correct leaf node L.
- Insert data entry into L in sorted order.
- If there's enough space in L, done.
- Otherwise, split L keys into L and a new node L2
    - Redistribute entries evenly, copy middle key
    - Insert index entry pointing to L2 into parent of L
    - So you redistribute the entries and push up the middle key.
- Example: suppose you have 1, 5, 9 where keys per node is 3 and want to insert 2.
    - Then it will partition into <5 and <=5. So 5 in the root node, 1 & 2 in the left child, 5 & 9 in the right child.

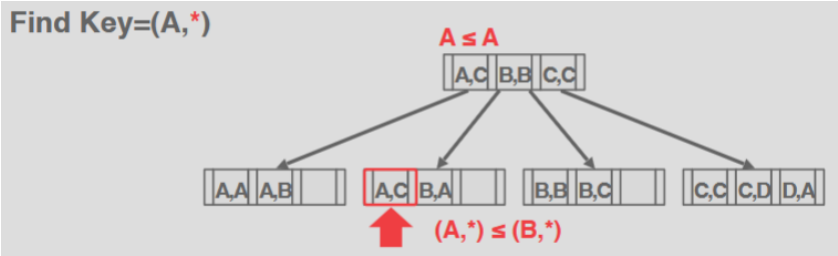
Delete:

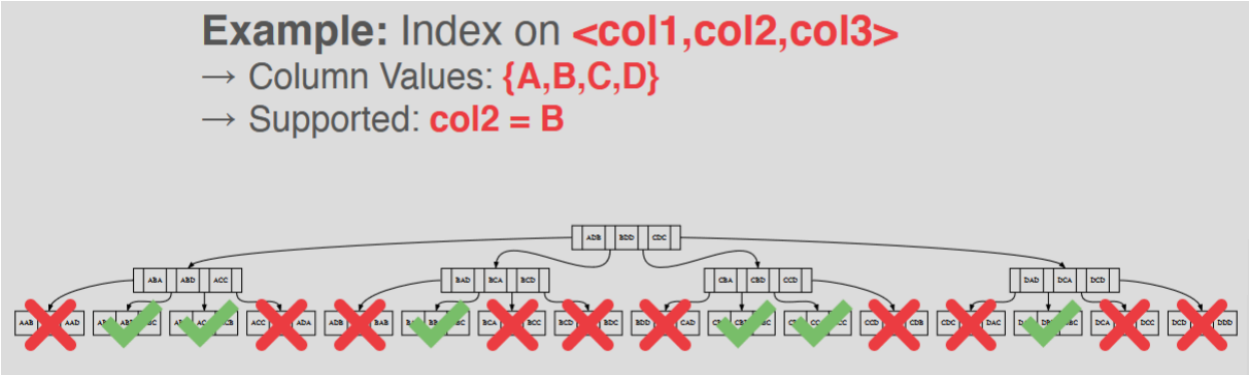
- Find the correct leaf node L.
- Remove the entry.
- If L is at least half-full, done.
- Otherwise, try to redistribute by borrowing from the sibling.
    - If that fails, merge L and the sibling. Then you must delete the entry from the parent of L.

Selection Conditions:

The DBMS can use a B+Tree index if the query provides any of the attributes of the search key.

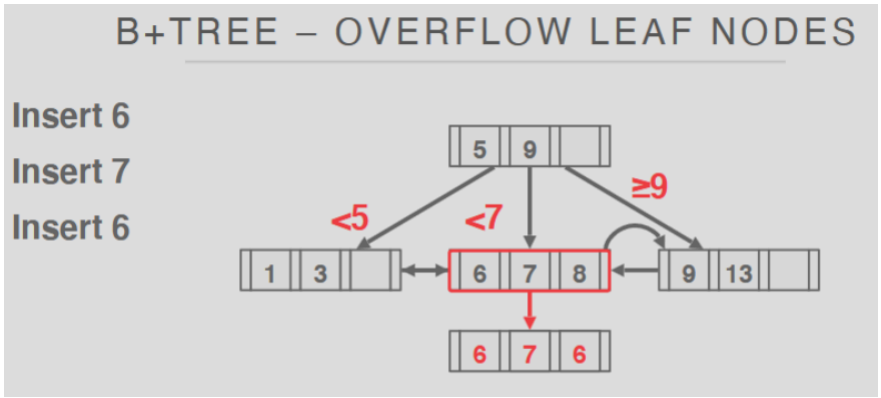
- In hash index, we can only do exact point queries over all keys.
- But B+Trees let us do queries with only some of the attributes.





Duplicate Keys:

- Approach 1: append record ID
  - The record ID is unique, so add it as part of the key to ensure that all keys are unique.
  - The DBMS can still use partial keys to find tuples (like in the previous examples of knowing which ones to skip).
  - Example slide 23
- Approach 2: overflow leaf nodes
  - Allow leaf nodes to spill into overflow nodes that contain the duplicate keys.
  - More complex to maintain and modify.



Clustered Indexes

The table is stored in the sort order specified by the primary key. Slides 25-26

Index Scan Page Sorting

- Retrieving tuples in the order they appear in a non-clustered index is inefficient due to redundant reads.
- To resolve this, we first read them and enumerate them, and then sort them based on their page ID.

Design Choices

Node Size:
The slower the storage device, the larger the optimal node size.

Merge Threshold:
If you have a write-heavy workload, it may be helpful to delay merging. Also it may be good to just let smaller nodes exist and then periodically rebuild the entire tree.

Variable-Length Keys:

- Approach 1: pointers (store the keys as pointers to the tuple's attribute). Problem: need to keep jumping to the attribute when you are traversing.
- Approach 2: variable-length nodes
- Approach 3: padding (wastes space)
- Approach 4: key map/indirection (embed an array of pointers that map to the key and value list within the node). Same idea as slotted pages.

Intra-Node Search:
How to search for the key inside a node.

- Approach 1: linear (scan node from beginning to end). Use SIMD to vectorize comparisons (using a mask)
- Approach 2: binary
- Approach 3: interpolation. Get the approximate location of the desired key based on the known distribution of keys.

Optimizations (Covered in Lecture 6)

Prefix Compression:
Sorted keys in the same leaf node are likely to have the same prefix. So, extract the common prefix and only store the unique suffix. (e.g. prefix rob → bed, bing, ot).

Deduplication:
Non-unique indexes can end up storing multiple copies of the same key in leaf nodes. So, we store the key once in the tree and then maintain a list of tuples with that key (like what we did in hash tables).

Suffix Truncation:

Recall that the inner nodes are only guideposts for traversal. Store the minimum prefix that is needed to correctly route to the index.

Pointer Swizzling:

- Nodes use page ids to reference other nodes in the index (all are logical page ids, not the actual pointer to memory).
- If we do this, you have to go through this whole process of going through the page id to translate to physical pointer to memory, etc.
- So this optimization removes that big process. If a page is pinned in the buffer pool, then we can store raw pointers instead of page id.
- Problem: if a page gets evicted from memory, its address will change. So what we have to do is make it back into its logical id when it's evicted.

Bulk Insert:

The fastest way to build a new B+Tree for an existing table is to first sort the keys and the build the index from the bottom up (as opposed to doing insert a bunch of times).

- Good to leave extra space in leaf nodes for future inserts.
- But if you know that you won't really be inserting in the future, can just pack them all into as few leaf nodes as possible.

Buffer Updates:

Lecture 6 (1/27) - Index Concurrency Control

Focus of exam will be everything before today. Only stuff about today will be high level overview stuff.

A DBMS must allow multiple threads to concurrently access data structures.

A **concurrency control** protocol is the method that the DBMS uses to ensure correct results. Two types of correctness:

- Logical correctness: can a worker (thread or process) who is executing a query see only the data it is supposed to see? (No leakage).
- Physical correctness: is the internal representation of the object sound? I.e., are the data structures doing ok? For example, in B+Tree, are there invalid pointers? (talk about this today)

Latches Overview

Locks vs. Latches:

- Locks:
    - Protect the database's logical contents from other transactions.
    - Held for the duration of the transaction.

    - Need to be able to rollback changes.
- Latches: more of a fine grain thing
    - Protect just the critical sections of the internal data structure from other threads.
    - Held for the duration of the operation.
    - No need to be able to roll back.

Latch Modes:

- Read mode:
    - Multiple threads can read the same object at the same time.
    - A thread can acquire the read latch if another thread has it in read mode.
    - Shared.
- Write mode:
    - Only one thread can access the object.
    - A thread cannot acquire a write latch if anyone else any kind of latch.

Implementations

Approach 1: Blocking OS Mutex

- Std::mutex m.lock() // critical section m.unlock()
- Simple to use but not scalable.

Approach 2: Reader-Writer Latches

- Allows for concurrent readers. Manages read/write queues.
    - Suppose a reader shows up. Give it the read latch.
    - Another reader. Give it read latch.
    - A write shows up. Can't give it the write latch because the read latches are used.
    - Another reader. You can logically give it the read latch, but doing so means that there will be starvation for the writers.
    - So there's a tradeoff that also needs to consider how much reading vs. writing you are doing.
- Can be implemented on top of spinlock.

Compare and Swap

Atomic instruction that compares contents of memory location M to a given value V.

- If values are equal, atomically installs new given value V' in M. (i.e., keep checking if value at M is V.)
- Otherwise fails.

Hash Table Latching

For hash tables, it's easy to support concurrent access due to the limited ways threads access the data structure. (Only way to do it is to use the hash function and then do whatever search operation)

- All threads move in the same conceptual direction (they're all hashing, traversing, etc.) and only access a single page/slot at a time.
- Deadlocks are not possible because they are going in the same direction.
- To resize the table, take a global write latch to protect the entire table. This is another downside of static hashing.

Approach 1: Page Latches
- Each page has its own reader-writer latch that protects its own contents.
- Threads acquire a latch before they can access a page.

Approach 2: Slot Latches
- More fine grained. Each slot/entry in the hash table has its own latch.
- Enables more concurrency; however, need to maintain a lot of metadata, a lot of individual lock implementations.

B+Tree Latching

Need to protect against two types of problems:
1. Threads trying to modify the contents of a node at the same time.
2. One thread traversing the tree while another thread splits/merges nodes.

Example: rebalancing. If you delete something first and then need to rebalance (if the leaf node is less than half full), need to make sure that it's updated for everyone. Avoid this using latch crabbing/coupling.

Latch Crabbing/Coupling
- A protocol that allows multiple threads to access/modify a B+Tree.
    - Get latch for parent.
    - Get latch for the child that you are visiting.
    - Release the latch for the parent if its safe (a safe node is one that will not split or merge when updated. For read, the parent is always safe. If the child node is not full (on insertion) and more than half full (on deletion), then it's safe).
- Find: start at root and traverse down the tree. Acquire read on child, unlatch the parent, and continue until we reach the leaf node.
- Insert/Delete: start at root and go down, obtaining write latches as needed. Once a child is latched, check if it's safe. If it is safe, release all ancestors.
    - Slide with "we know that D will not merge with C" that's because D is more than half full, so no need to move anything if you delete.
    - Slide with "We know that if D needs to split, B has room" because B is not full.

  Note that you need to latch the root node every time. Leads to bottleneck.
    - A better version: assume that there won't be a split or merge. Then optimistically traverse the tree with read latches. Once you reach the leaf node, write latch it.
    - If you determine that the leaf node is not safe, go back and re-do with pessimistic traversal.

Leaf Node Scans

If we need to move from one leaf node to another: have to hold the latch on one sibling until you latch to another sibling (similar to the above process). Can lead to deadlock though.

- The latches themselves don't have deadlock detection.
- Take care of this by, for example, make sure traversal is only in one direction or tell threads to not wait and just abort.

Lecture 7 (2/3) - Sorting and Aggregation

Now looking at operator execution (how to execute queries)

A query plan is a logical representation of your SQL query.

- Operators are arranged in a tree. Data flows from leaf nodes to the root.
- The output of the root node is the result of the query.

Note about storage:

- We cannot assume that query results fit in memory.
- Use buffer pool to implement algorithms that need to spill to disk.
- We prefer algorithms that maximize the amount of sequential I/O.

Why we need to sort:

- Relational model/SQL is unsorted.
- Queries may request that tuples are sorted (SQL: order by)
- Other reasons:
    - Trivial to support duplicate elimination (SQL: distinct)
    - Bulk loading into B+ tree is faster
    - Aggregations (Group by)

If we assume that everything is in memory:

- We can use any standard sorting algorithm. For example, quicksort.
- However, if not everything fits in memory, it's not aware that some subset of the range is in memory.

Top-N Heap Sort

(Get top n elements)

- Order by with a limit; then only need top n elements.
- The ideal scenario is if the top-n elements all fit in memory.
    - Maintain an in-memory sorted priority queue.
    - Scan through the original data and for each element, put it in the sorted heap. If full and you have an element that's larger than an existing thing, you can evict.

External Merge Sort

(general purpose algorithm)

A divide and conquer algorithm that partitions data into separate runs, sort individually, and combine them into sorted runs.

1. Phase 1: sort. Each chunk of data that fits in a chunk of memory; write back the sorted chunks to a file on disk.
2. Phase 2: merge. Combine sorted runs into larger chunks. Get written back to disk.
3. Repeat this process over and over

Specifics:

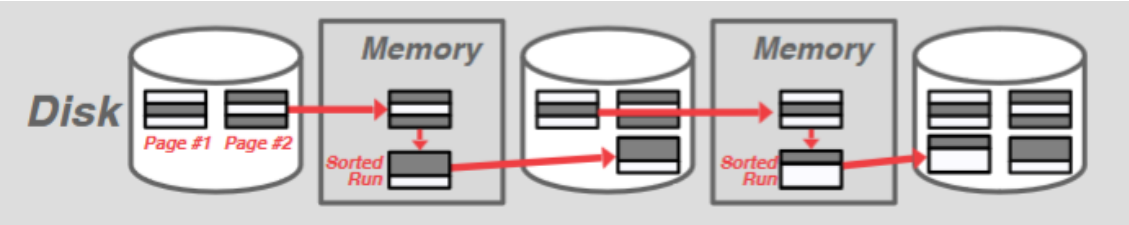
- A run is a list of key/value pairs.
    - The key is what we compare to compute the sort order.
    - The value can be a tuple (early materialization) or a record id (late materialization) that maps the key to location of tuple.
    - Recall: we prefer early for data of fixed length and relatively small. We prefer late if long data (so no need to copy around).

<u>2-Way External Merge Sort</u>

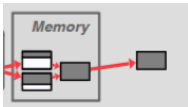
- 2 refers to the number of runs that we are going to merge into a new run for each part (merging phase)
- N is the number of pages; let B be the number of buffer pool frames that hold the data.

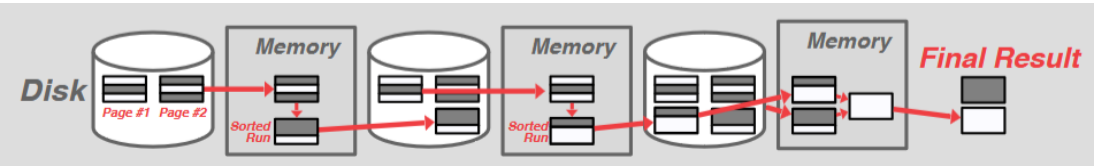
Process:

- Pass 0: read all B pages of the table into memory. Sort pages into runs and write them back to disk.



- Pass 1, 2, 3, ...: recursively merge pairs of runs into runs twice as long. Uses 3 buffer pages (2 for input pages, 1 for output)





Number of passes:

- 1 + ceiling(1+log_2N) (the +1 is for the initial sort?)

Total I/O cost:

- 2N * (# of passes)

In the example: the first two pages merges gets the first sorted pair, puts that in memory, and then gets the next sorted pair (e.g. [3,4] [2,6] → first output buffer [2,3] put that in memory → second output buffer [4,6]).
Each buffer page fits 2 integers. There are three total buffer frames; 2 in, 1 out
For 1→2:

- Read in first page from each of the runs ([2,3] [4,7])
- First output page is [2,3]. Write out into memory.
- Second output page is [4,4]. Write out into memory.
- Read in next page from each of the runs

Notes:

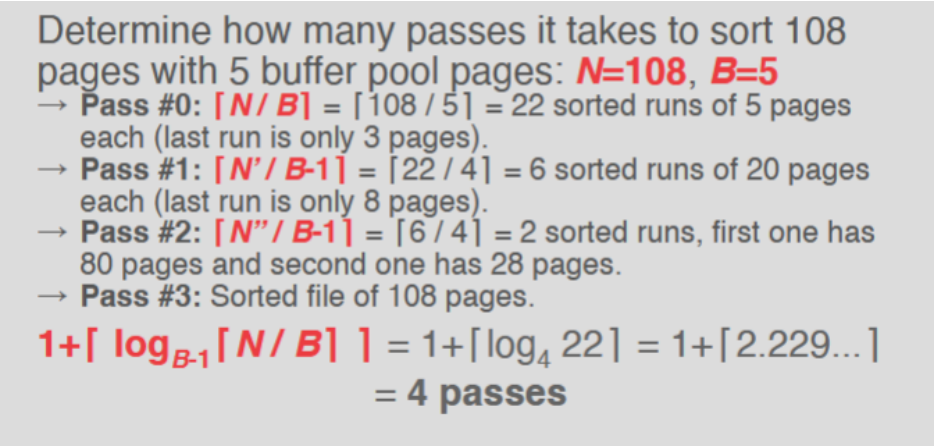
- Only requires 3 buffer pool pages to perform sorting.
- But if you have more than 3 buffer pool pages (more space available), it is not efficient. So we use general external merge sort

<u>General External Merge Sort</u>

- Pass 0: product ceiling(N/B) sorted runs of size B.
- Pass 1, 2, 3, …: Merge B-1 runs
- Number of passes is 1 + ceiling(log_{B-1}ceiling(N/B))
- Total I/O cost: 2N * (# of passes)

Example (exam question): determine how many passes it takes to sort 108 pages with 5 buffer pool pages.
N=108, B=5.

- N' is the number of sorted runs produced by previous pass.



<u>Double Buffering Optimization</u>

- Prefetch the next run in the background and store it in a second buffer while the system is processing the current run.
- Reduces the wait time for I/O requests at each step by continuously using the disk.
- Downside: you will have to either use half of the buffer pool or double ???

Comparison Optimizations:

1. Approach 1: Code specialization: instead of providing a comparison function as a pointer to sorting algorithm, create a hardcoded version of sort that is specific to a key type (e.g., one sort that works for integers, one sort that works for strings, etc. instead of using a switch statement).
2. Approach 2: Suffix truncation: (like with B+ trees). First compare the binary prefix. If they're the same, then do slower version of looking at the whole string.

Using B+Trees for Sorting:

- If table is already in sorted order slide 20
- Cases:
    a. Clustered B+Tree: traverse the leaves. Good because no computational cost and all disk access is sequential.
    b. Unclustered B+Tree (i.e., B+tree is sorted in a different order): you'd have to chase each pointer to the page that contains the data. Almost always a bad idea because you'd end up with one I?O per data record. To resolve this, do one scan to enumerate all pages that you want to access, sort those pages in memory, and then as you scan again, can grab all at once.

Aggregations

(through sorting and through hashing)

Collapse values for a single attribute from multiple tuples into a single scalar value. Need a way to quickly find tuples with the same distinguishing attributes for grouping. Two implementation choices: sorting and hashing.

Sorting Aggregation

- Also good for removing duplicates.
- But if you don't need the data to be ordered (e.g. group by or distinct)? Better to do hashing. Computationally cheaper than sorting.

Hashing Aggregation

- Build an ephemeral hash table as the DBMS scans the table. For each record, check whether there is already an entry in the hash table:
    - Distinct: discard duplication
    - Group by: perform aggregate computation
- Easy if everything fits in memory
- But spilling data into disk, leads to a bunch of random access for the parts of the hash table in memory.

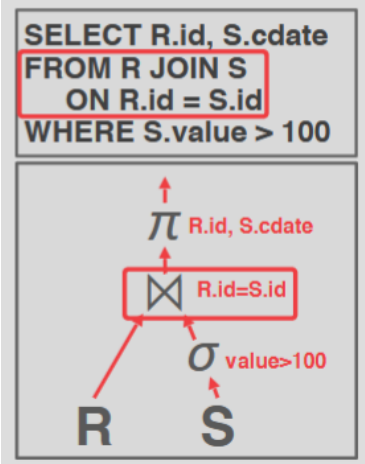
External Hashing Aggregation

1. Partition: divide tuples into buckets based on hash key. Write them out to disk when they get full.
    - Use a hash function h_1 to split tuples into partitions. Fit into buffer? If not keep partitioning.
    - Assume we have B buffers. Use B-1 buffers for partitions and 1 for the input data. If you fill up the B-1, evict and continue partitioning.
2. Rehash: build in-memory hash table for each partition and compute the aggregation.
    - For each partition on disk, read it into memory and build an in-memory hash table based on the second hash function h_2. Then go through each bucket of the hash table to bring together matching tuples.
    - Assumes that each partition fits in memory. If not, partition more until you get something that fits.

Hash Summarization (e.g. storing a running sum)

- During the rehash phase, store pairs of the form GroupKey/RunningVal

Lecture 8 (2/5) - Joins



Why we need to join:

- We normalize tables in a relational database to avoid unnecessary repetition of information. (How things are split up between tables and how the relationships between tables are modeled.)
- Then use the join operator to reconstruct the original tuples.

Join Algorithms:

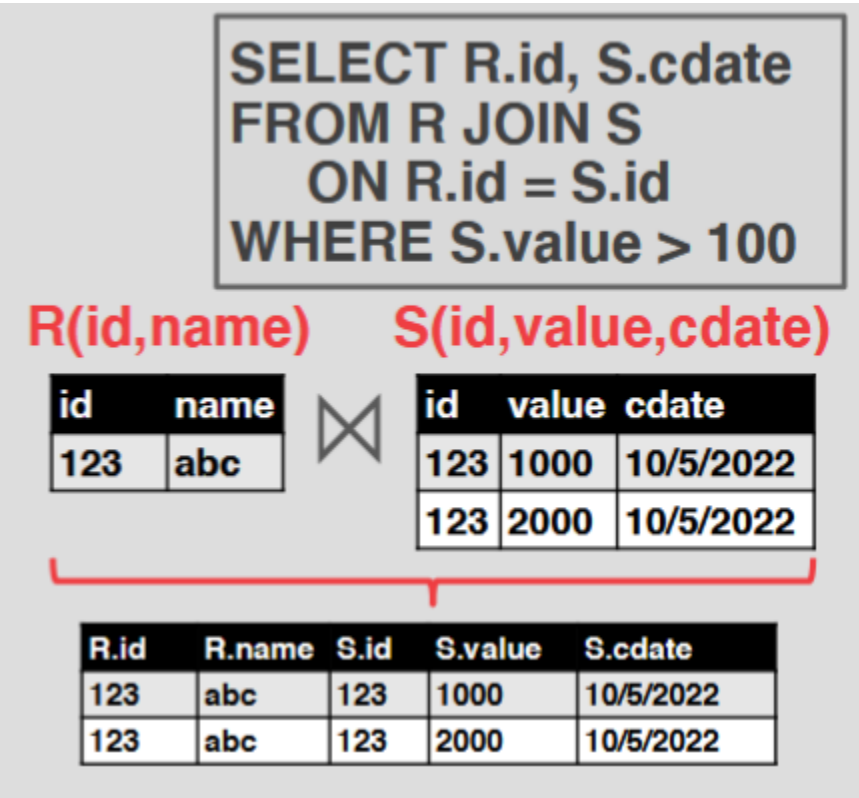
- We will focus on binary joins (two tables). Focus on inner equijoin algorithms (joining on attributes that are equal; other types involve inequality comparison, called theta join; or when something in one table doesn't exist in another, called anti join). This is for matching foreign and primary keys.
- Outer join will include everyone, even if there's no match (e.g. student with no classes).
- In general, we want the smaller table to always be the left table ("outer table") in the query plan. Using the algorithms we discuss, it will be cheaper to do this. Slide 4

Two Decisions:

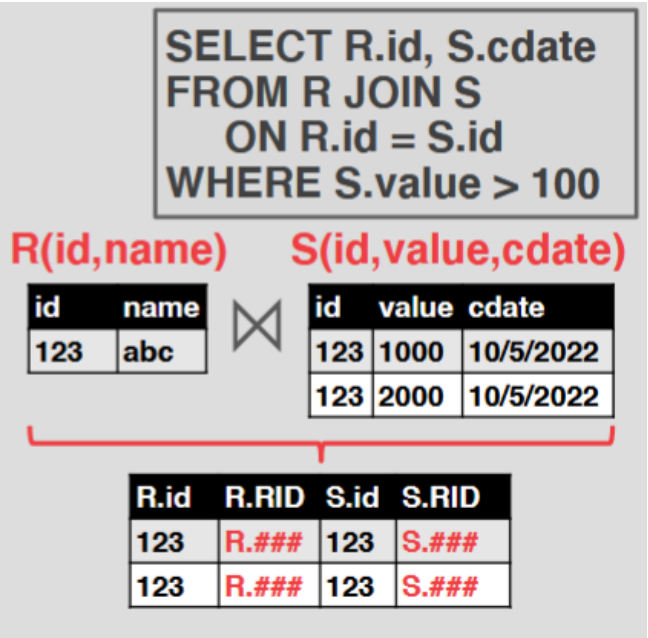
1. Output: what data does the join operator emit to its parent operator in the query plan tree (e.g. early materialization vs late materialization)?
2. Cost analysis: how can we tell if one join algorithm is better than the other?

Operator output: data

- Early materialization:
  - Copy the values for the attributes in outer and inner tuples into a new output table.
  - That way, subsequent operators don't need to go back to the original tables.



- Late materialization:
  - Only copy the join keys with the record ids of the matching tuples.
  - So we need to go back to the base table to get the actual value.
  - Good for column stores because the DBMS doesn't need to copy data that is not needed for the query.



Cost Analysis

Assume table R has M pages and m tuples, and S has N pages and n tuples. The cost metric is the number of IOs the join algorithm is going to perform.

Join vs Cross Product

- Join: filtering to subset where matches exist.
- Cross Product: all the possible pairs. If you try to join using the cross product and then filtering, it will be very inefficient because cross product produces a lot of data.

Nested Loop Join

Simple

Two for loops.

- For each tuple in r:
  For each tuple in s:
  Emit if r and s match.
- Outer table is on the left side in the notation. R ⋈ S.
- The cost is M + (m * N). Note that the simple nested loop doesn't care that there are pages.
- Suppose that M > N, m > n. If you use the smaller table as the outer table, becomes N + (n * M), which is less.

Block

```
foreach block B_R ∈ R:
  foreach block B_S ∈ S:
    foreach tuple r ∈ B_R:
      foreach tuple s ∈ B_s:
        emit, if r and s match
```

- Performs fewer disk accesses because for every block in R, we scan S once.
- So number of IOs is M + (M * N) (assuming the block size is one page)
- To make the smaller outer, we compare size based on the number of pages, not the number of tuples.

What is the block size:

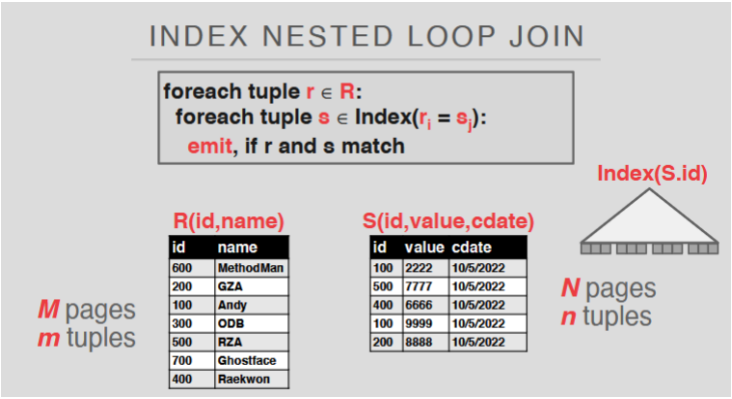
Suppose we have B buffers available.

```
foreach B-2 pages p_R ∈ R:
  foreach page p_S ∈ S:
    foreach tuple r ∈ B-2 pages:
      foreach tuple s ∈ p_s:
        emit, if r and s match
```

- We will have B-2 buffers for scanning the outer table.
- One buffer is for the inner table, and one buffer is for storing output (hence B-2).
- Then it will be M + (ceiling(M/(B-2)) * N)
- Suppose the outer relation completely fits in the memory (B > M+2). Cost will be M+N.

Index

We can avoid doing a sequential scan to check for a match with this method. Uses an index to find inner table matches.



- Assume the cost of each index probe is a constant C per tuple.
- Then the cost is M + (m * C)

Summary

Takeaways:

- Pick the smaller table as outer. (for basic, it's number of tuples; for blocking, it's number of pages)
- Buffer as much of the outer table in memory as possible.
- Loop over the inner table or use an index.

Sort-Merge Join

Phases:

1. Sort: sort both tables on the join key(s). Use external merge sort.
2. Merge: step through the two sorted tables with cursors and emit matching tuples. We may need to backtrack depending on the join type. If there are duplicate keys in the outer relation, it must backtrack.

```
sort R,S on join keys
cursor_R ← R_sorted, cursor_S ← S_sorted
while cursor_R and cursor_S:
 if cursor_R > cursor_S:
  increment cursor_S
 if cursor_R < cursor_S:
  increment cursor_R
 elif cursor_R and cursor_S match:
  emit
  increment cursor_S
```

Cost:

- Sort R: 2M * (1 + ceil(log_{B-1} ceil(M/B))
- Sort S: Same with N
- Merge cost: M+N
- Total is the sum of all of these.

Worst case is when the join attribute of all the tuples in both relations contains the same value: (M*N)+sort

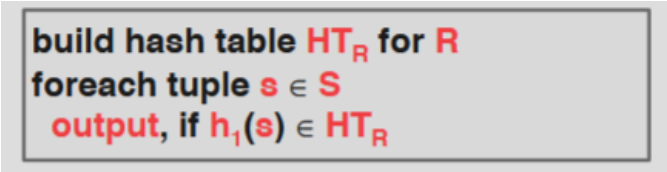
Useful when:

- One or both tables are already sorted on the join key.
- Output has to be sorted on the join key.
- The input relations may be sorted either by an explicit sort operator, or by scanning the relation using an index on the join key.

Hash Join

If a tuple r in R and s in S satisfy the join condition, they have the same value for the join attributes. For a hash function to be well defined, if we map a key to some partition i, the R tuple must be in r_i and the S tuple in s_i. Therefore, R tuples in r_i need only to be compared with S tuples in s_i.

- Phase 1: build
    - Scan the outer relation and populate the hash table using hash function h_1 on the join attributes.
    - Linear probing works the best.
- Phase 2: probe
    - Scan the inner relation and use h_1 on each tuple and jump to a location in the hash table and find a matching tuple.



Contents:

- Key contains the attribute(s) that the query is joining the tables on.
- Values: depends on what the operations above the join in the query plan expect as its input. Also keep in mind early vs. late materialization.

Optimization

Probe Filter
Slide 36

In order for the bloom filter to have covered that, need to have both bits set.
When the bloom filter returns a true, then go look at the actual value.

Notes from 2/10:
A bloom filter is a probabilistic data structure (bitmap) that answers set membership queries.

- False negatives will never occur.
- False positives can sometimes occur.

Slide 38 details the process.

Partitioned Hash Join
What happens if we do not have enough memory to fit the entire hash table?

Two phases:

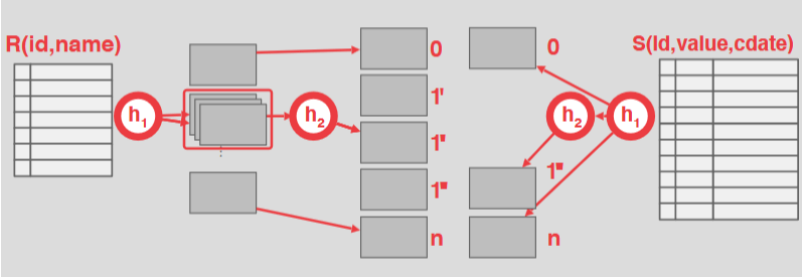
1. Build phase: hash both tables on the join attribute into partitions.
    - Hash R into x buckets and hash S into the same number of buckets. Use the same hash function.

2. Probe phase: compare tuples in corresponding partitions for each table.
    - Then you can perform regular hash join on each pair of matching buckets in the same level between R and S (so only need to compare tuples that are in the same partition).

build hash table $HT_{R,0}$ for $bucket_{R,0}$
foreach tuple s ∈ $bucket_{S,0}$
output, if $h_2(s) \in HT_{R,0}$

What if a partition doesn't fit in memory? Then use recursive partitioning to split the tables into chunks that will fit.

- Will use a new hash function, not equal to the first function, to hash into these smaller partitions.
- Then probe it for each tuple of the other table's bucket at that level.



Cost: 3(M+N), assuming we have enough buffers (no recursive splitting).

- Partitioning phase is read + write both tables, so 2(M+N).
- Probing phase is reading both tables, so M+N.

Note that if the DBMS knows the size of the outer table, you can use a static hash table (which means that there's less computation for building/probing). Otherwise, you must use a dynamically resizable hash table.

JOIN ALGORITHMS: SUMMARY

Algorithm	IO Cost	Example
Simple Nested Loop Join	$M + (m \cdot N)$	1.3 hours
Block Nested Loop Join	$M + (M \cdot N)$	50 seconds
Index Nested Loop Join	$M + (m \cdot C)$	Variable
Sort-Merge Join	$M + N$ + (sort cost)	0.75 seconds
Hash Join	$3 \cdot (M + N)$	0.45 seconds

- Hashing is almost always better than sorting.
- Sorting is good for non-uniform data (e.g. a lot of duplicates) and when the result needs to be sorted.

Lecture 9 (2/10) - Query Execution

Processing Models

A DBMS' processing model defines how the system executes a query plan.

3 Approaches:

1. Iterator Model
    - Each query plan operator implements a Next() function.
    - Every time an operator finishes (e.g. in its tree), it will return either a single tuple or a null marker if there are no more tuples.
    - The operator implements a loop that calls Next() on its children to retrieve their tuples and then process them.
    - Also called the Volcano or Pipeline model.
    - Visualize the query plan as a tree. Leaf nodes are scans over the table.
        - Slide 6.
        - Every time we get a tuple, we'll emit that tuple t to the parent.
        - The the parent will have the for loop for t in child.Next()
        - Continue all the way to the top.
    - Pipeline breakers: sorting, aggregates, building hash tables – anything that requires a full materialization of your result (e.g. need to build the entire hash table).
    - Probing is not a pipeline breaker, so you can emit the result of the join as you go.
    - It's very easy to do output control for this. For example, limit 10 you just call Next() 10 times.
2. Materialization Model
    - Each operator processes its input all at once and then emits its output all at once ("materializes" its output).
    - For limiting, unlike the iterator, the DBMS has to push down a hint to avoid scanning too many.
    - The output can be a materialized row (whole tuples; NSM, e.g. Amazon shopping cart, not too many things in your cart so materializing all of those items isn't a big deal) or a single column (DSM)
    - Slide 9
        - Each child returns the entire output.
        - For example, do the entire join at once.
    - Good for OLTP workloads because you only access a small number of tuples. In iterator, you would need to call Next() a bunch of times and need to keep locking and unlocking. Here, you just do whatever with the resource and then unlock.
    - Not good for OLAP with large intermediate results.
3. Vectorized / Batch Model (kind of the in between of the previous two)
    - Uses Next() function, but emits a batch of tuples instead of a single one.
    - Slide 12
        - Once your output reaches the threshold size, you emit the output.

    - Good for OLAP because it reduces the number of invocations per operator.

Plan Processing

1. Approach 1: Top to Bottom
    - Aaaaa slide 14
2. Approach 2: Bottom to Top
    - Push-based
    - Start with leaf nodes and push data to their parents
    - Allows for tighter control of caches/registers in pipelines
    - Allows you to do query plans for non-tree based ones

Access Methods

An access method is the way that the DBMS accesses the data stored in a table. (So how we look at the tables R and S)

Three basic approaches:

- Sequential Scan: for each page in table, retrieve it from the buffer pool (or read from disk) and iterate over each tuple and check whether we emit it.
    - The DBMS maintains an internal cursor that tracks the last page/slot it examined.
    - Implemented as for loop with a predicate. For page in table pages: for t in page tuples: if predicate: do something.
    - Optimization: zone maps. Precomputed aggregates for the attribute values in a page. DBMS checks the zone map first to decide whether to access the page.
        - So compute aggregates for a particular attribute for a range of pages; typically, the DBMS puts it in the page header, but elsewhere is fine (e.g. snowflake stores those zone maps somewhere else together so you can just read the zone maps without needing to pull pages in to look at page headers). E.g. min max, avg, sum, count.
        - So when you are looking, you look at the attribute and just by looking at that, you determine if you want to look at the page at all.
        - E.g. for query with where val > 600, you know that if the max is 400, you can skip that page.
- Index Scan: DBMS picks an index to find the tuples that the query needs. Pick the index that best reduces the result set.
    - Suppose we have a single table with 100 tuples and two indexes (age and department) and we want students where age < 30, department = CS, and country = US. If you have 99 people under the age of 30 and only 2 people in the CS department, then the index should be department.
- Multi-index Scan: if there are multiple indexes that the DBMS can use for a query (so not something that's super selective), apply each index separately and find the intersection of those results.
    - Compute sets of record IDs using each matching index.

- Then combine based on predicates.

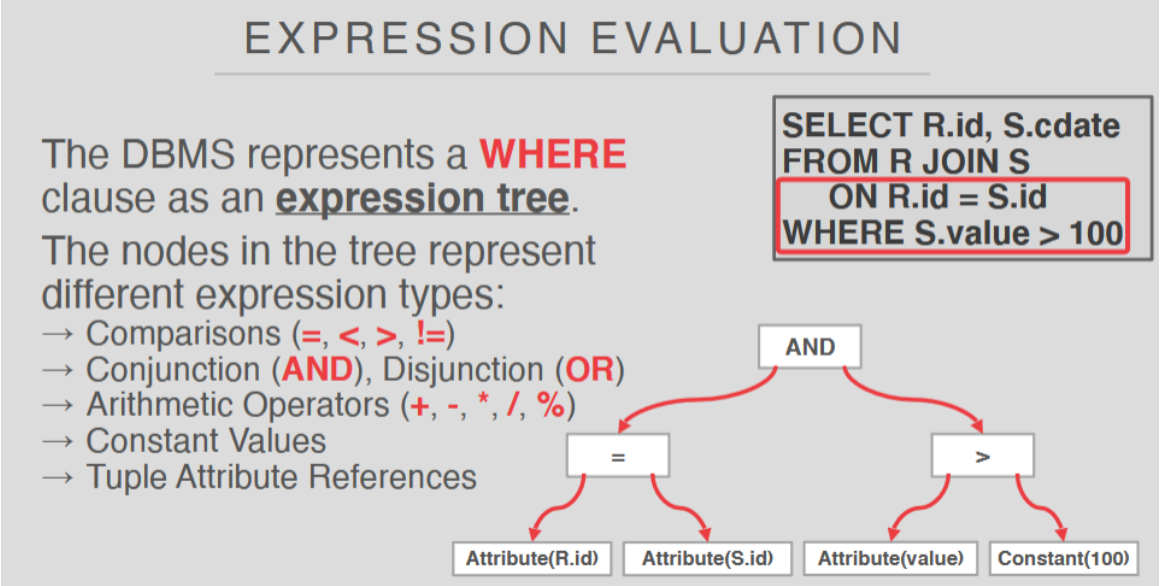
Modification Queries

Operators that modify the database are responsible for modifying the target table and its indexes (e.g. the tuple itself and any corresponding indexes).

- Need to check constraints (e.g. if something you insert has to satisfy a certain constraint)

Expression Evaluation

The DBMS represents a WHERE clause as an expression tree. The nodes in the tree represent different expression types:



Slide 23: $1 is the parameter.

- However, this is slow. The DBMS traverses the tree and for each node that it visits it must figure out what the operator needs to do. It's basically a function call for each node.
- Instead, evaluate the expression directly by generating a function that does it automatically. Replaces the iterator-based approach with a simple function. Apply this function to every extracted value from tuples in the table.

Parallel Execution

The above are all assuming single-threaded.

Why we care about parallelism:

- Higher throughput, lower latency → increased performance with same hardware.
- Increased responsiveness (no waiting for I/Os to finish)
- A potentially lower total cost of ownership (less physical needs)

Parallel vs. Distributed:

Parallel DBMS:

- Resources are physically close to each other e.g. single multi-core machine.
- Communication is assumed to be cheap and reliable.

Distributed DBMS: (not in this class)

- Resources far away.
- Communication cost and problems.

2 Types of Parallelism

1. Inter-query: execute multiple disparate queries simultaneously. Increases throughput and reduces latency.
    - If queries are read only, easy – little coordination between queries needed.
    - Updating requires coordination (next time)
2. Intra-query: execute the operations of a single query in parallel (break up a single query). Decreases latency for long-running queries, especially OLAP.
    - Think of organization of operators in terms of producer/consumer paradigm.
    - There are parallel versions of every operator.
    - You can have either multiple threads access centralized data structures or use partitioning to divide work up.

Three approaches to Intra-Query Parallelism

1. Intra-Operator (Horizontal): decompose operators into independent fragments that perform the same function on different subsets of data. Then use the exchange operator to coalesce/split results from multiple children/parent operators. Example slide 34. Three types of exchange operators:
    - Type 1 - Gather: combine results from multiple workers into a single output stream.
    - Type 2 - Distribute: split a single input stream into multiple output streams.
    - Type 3 - Repartition: shuffle multiple input streams across multiple output streams (for example if you have 3 inputs and 2 outputs).

    Slide 36 example:
    - Use pi (projection) to remove some of the unneeded attributes.
    - Exchange operator splits out a single hash table on the left side (build).
    - On the right side, for B, you are doing a probe on the right side since the hash table is already built, so you can probe all at once).
    - There are ways to make partitions better to avoid one worker finishing way before another one finishes. For example, you can not do partitioning beforehand, but instead grab as you go.
2. Inter-Operator (Vertical / pipeline parallelism): operations are overlapped in order to pipeline data from one stage to the next without materialization. Workers execute operators from different

segments of a query plan at the same time. Good for streaming systems (continuous queries). Slide 38 example.
    - Note that you are bound by your slowest operator. E.g. projection is much faster than join, so it'll be sitting around waiting for the join to finish.
3. Bushy Parallelism: hybrid.

I/O Parallelism

Note that performance is bound by disk no matter how much parallelism you have. Sometimes the performance is worse if a thread is accessing different segments of the disk at the same time. Slide 41 xxx.

- Can use multi-disk parallelism: configure OS/hardware to store the DBMS's files across multiple storage devices. This is transparent to the DBMS. E.g. partitioning the database across disk (striping) but cost is losing data, or mirroring (each contains the same thing, prevents loss from failure, but you can't store as much anymore because you have duplication).

Lecture 10 (2/12 and 2/19) - Query Optimization

Add formulas to notes (including the ones with logs).

The query planner converts a query into a plan that makes it optimized (e.g. which order should we join, which index to select over, which physical operations e.g. hash join, etc.). Need to use heuristics to limit the search space.

Logical vs. Physical Plans

- Logical: maps query to physical algebra expressions (e.g. we are doing a join). The abstract version.
- Physical: defines a specific execution strategy using an access path. Not necessarily a 1:1 mapping from logical to physical – can break stuff up into multiple operators or combine to one (e.g. if you want to sort and join, use merge-sort join).

Architecture Overview: slide 6

- Binder matches the phrases to what they are referencing in the databases. Connects the abstract syntax tree to the data itself.
- Tree rewriter looks at the schema info and sees if you can combine or make anything better.

High Level Categories of Approaches

- Heuristics/Rules:
    - Rewriting the query to remove inefficient things
    - These need to examine catalog, but they do not need to examine data
    - For example, if you know something will always evaluate to true, you can ignore that.
- Cost-Based Search:
    - Use a model to estimate the cost of executing a plan. Estimate number of I/O.
    - Enumerate multiple equivalent plans for a query and pick the one with the lowest cost.

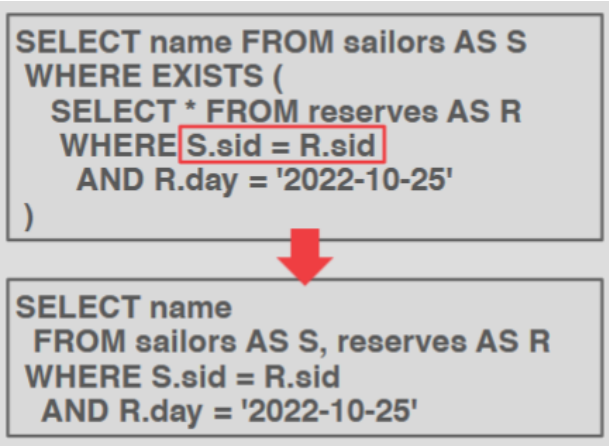
Heuristic/Rule-Based Optimization

This is logical plan optimization:

- Transform a logical plan into an equivalent logical plan using pattern matching rules with the goal of increasing the likelihood of enumerating the optimal plan in the search.
- You cannot compare plans because there is no cost model
- Examples:
    - Splitting conjunctive predicates: ands. Decompose predicates into their simplest forms to make it easier for the optimizer to move them around. (Apply each "and" one after the other).
    - Selection/Predicate Pushdown: move the predicate to the lowest applicable point in the plan. See example on slide 12. E.g. if you only need one table to look at this predicate, you can evaluate it right away. If you filter things out earlier on, less processing work later.
    - Replace Cartesian Products with Inner Joins: join criteria is your predicate.
    - Projection Pushdown: eliminate redundant attributes (columns) before pipeline breakers to reduce materialization cost.

Nested Sub-Queries: The DBMS treats nested sub=queries in the where clause as functions that take parameters and return a single value or set of values. Two approaches:

- Rewrite to de-correlate and/or flatten them (for example, do a join instead of finding each pairwise comparison). A simple example that's not join but could use that:



- Decomposing nested query and store results in a temporary table. Breaks queries into blocks and then concentrates on one block at a time. Stores in a temporary table that is discarded after the query finishes. Slide 18.

Expression Re-Writing: transforms a query's expressions into the minimal set (e.g. instead of looking for people in age ranges 1-15 or 10-20, just do 1-20). Examples:

- Impossible/Unnecessary Predicates: Select * from A where 1 = 0 → always false, so get rid of it.
- Merging Predicates: the age range example.

Query Cost Models (Cost-Based Models)

DBMS uses a cost model to predict the behavior of a query plan given a database state. Three components:

1. Physical Costs: CPU cycles, I/O, cache misses (depends on hardware)
2. Logical Costs: Estimate output size per operator. (The focus is normally here).
3. Algorithmic Costs: Complexity of the operator algorithm implementation.

Statistics: the DBMS stores internal statistics about tables, attributes, and indexes in its internal catalog.

Examples of statistics:

- Selection Cardinality (how much filtering an individual predicate is going to perform). The selectivity (sel) of a predicate P is the fraction of tuples that qualify = #occurrences/size of total tuples in table. “Highly selective” (in the intuitive sense) means a lot of filtering, selectivity is close to 0. Assumptions:
    - Uniform Data (in buckets): for example buckets for age 1-10, 11-20, etc.
    - Independent Predicates: predicates on attributes are independent (not a good assumption in the real world)
    - Inclusion Principle: the domain of join keys overlap such that each key in the inner relation will also exist in the outer table.
- Correlated Attributes. Example of where independence fails: make of car = Honda and model = Accord, but the selectivity (assuming that make and model are independent) is very low because only 1/10 * 1/100. True selectivity (since only Honda makes Accords) is 1/100.

Ways to describe statistics:

- Histograms: to not assume uniformity, make buckets →
    - Equi-width histogram: maintain counts for a group of values instead of each unique key. All buckets have the same width.
    - Equi-depth histogram: vary the width so that the total number of occurrences for each bucket is roughly the same.
- Sketches: probabilistic data structure that gives an approximate count for a given value.
- Sampling: DBMS maintains a small subset of each table that it then uses to evaluate expressions to compute selectivity (e.g. about ⅓ of my sample is in the age range, so it’s likely that about ⅓ of the entire population is in the age range).

Cost-Based Optimization

After rule-based rewriting, the DBMS will enumerate different plans for the query and estimate their costs.

Single-Relation Query Planning

- Pick the best access method (e.g. sequential vs. index vs. binary scan). Simple heuristics are often good enough; easy for OLTP.
- For OLTP: easy because they are sargable (slide 38)

Multiple Relations

- Choice 1: Bottom-up optimization – start with nothing and then build up the plan to get to the outcome that you want.
    - Use static rules to perform initial optimization.
    - Use dp to determine the best join order for tables using a divide-and-conquer search method.

  System R Optimizer (review this) (left deep means you do ((a join b) join c) join d as opposed to (a join b) join (c join d). But this method doesn't know if you need the final result to be sorted. Which is a reason you might want to do top-down optimization.
- Choice 2: Top-down Optimization – start with the outcome that you want, and then work down the tree to find the optimal plan that gets you to that goal.

On the exam, you don't need to know the specifics of the bottom up and top down algorithms, but know the difference between the two.

Lecture 11 (2/26) - Concurrency Control

Two more parts of the architecture:

- Concurrency Control: lost updates
- Durability: recovery

Transactions: the execution of a sequence of one or more operations (e.g. SQL queries) to implement some higher-level functionality.

- No partial transactions are allowed.
- Example: transferring $100 from checking to savings.
    - Check whether the checking account has $100.
    - Deduct $100 from the checking account.
    - Add $100 to the savings account.

Goal is to allow concurrent execution of independent transactions.

- Arbitrary interleaving of operations can lead to
    - Temporary inconsistency (fine)
    - Permanent inconsistency (bad)
- Need formal correctness criteria to let us decide if interleaving is valid.

Formal Definitions:

- Database: a fixed set of data objects (ignore inserts and deletes for now). Denote A, B, C, etc.
- Transaction: a sequence of read and write operations. E.g. R(A), W(B), etc.

Transactions in SQL:

- A new transactions starts with BEGIN and stops with COMMIT or ABORT:
    - If ABORT (e.g. not enough money to transfer), all changes are undone.
    - If COMMIT, DBMS either saves the changes or aborts it.
- Abort can be either self-inflicted (having enough money) or by the DBMS for some other reason. Possible other reasons:

    - No more room on disk.
    - Internal error.
    - Integrity constraints e.g. primary key, foreign key, uniqueness, can't have negative ages.

Overview of ACID correctness criteria:

- Atomicity: Either all actions happen or none happen.
- Consistency: Transactions always bring the database from one consistent state to another. Consistent state means that the integrity constraints are satisfied. If something starts in a consistent state, it must end with integrity constraints still satisfied.
- Isolation: Execution of one transaction is isolated from that of other transactions. Shouldn't be able to see any states from other transactions.
- Durability: Committed transactions are permanent.

Atomicity

Two possible outcomes of a transaction:

- Commit after completing all actions.
- Abort after executing some or all actions.

The DBMS guarantees atomic transactions.

Mechanisms for Ensuring Atomicity

1. Logging:
    - DBMS logs all actions so that it can undo the actions of aborted transactions.
    - Maintain undo records both in memory and on disk.
    - Log is written to disk upon commit?
2. Shadow Paging:
    - DBMS makes copies of pages and transactions make changes visible to those copies.
    - Page changes are made visible to others only when DBMS commits.

Consistency

The "world" represented by the database is logically correct. All queries about the data are given logically correct answers.

Database Consistency

The DB accurately models the real world and follows integrity constraints (e.g. all ages are positive). All future transactions see the effects of transactions committed in the past inside of the database.

Transaction Consistency

Individual transactions that start consistent will also end consistent. Constraint correctness is the application's responsibility.

Isolation

Each transaction submitted by a user executes as if running by itself.

Concurrency Control Protocol: how the DBMS decides the proper interleaving of operations from multiple transactions. Two categories:

1. Pessimistic: do more work up front to prevent conflicts from happening in the first place (assuming conflicts will happen so avoid them by doing a lot of work).
2. Optimistic: assume conflicts are rare and deal with them only after they happen.

The net effect must be equivalent to these two transactions running serially in some order. Good interleaving is when you have changes to A in T1 first and then T2 and the same with B. There will be a matching serial execution. In summary, **a schedule is correct if it is equivalent to some serial execution.**

- Serial Schedule: a schedule that does not interleave the actions of different transactions.
- Equivalent Schedules: for any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.
- Serializable Schedule: a schedule that is equivalent to some serial execution of the transactions.

To formalize the notion of equivalence, define conflicting. Two operations conflict if:

- They are done by different transitions.
- They are on the same object and one of them is a write.

Interleaved Execution Anomalies:

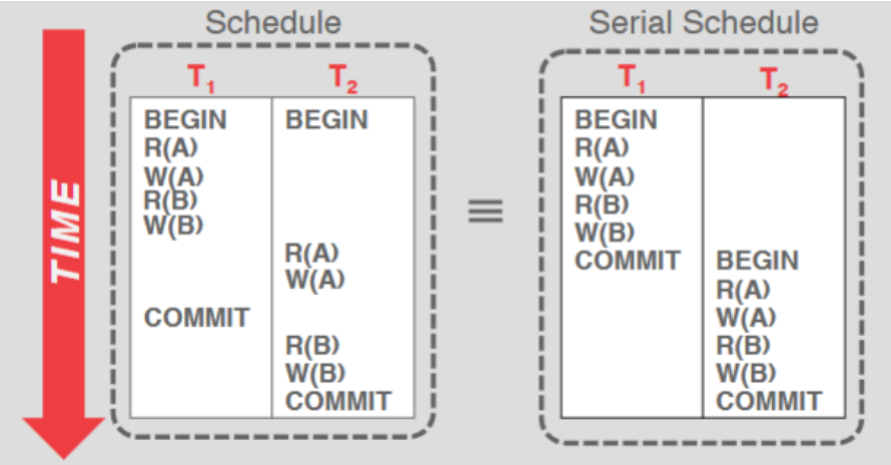
- Read-Write Conflicts: transaction gets different values from subsequent reads.
    - A write has been done in T2 in between the two reads in T1. But that value should never change? (Unrepeatable Read)
- Write-Read Conflicts (Dirty Reads): one transaction reads uncommitted data from another.
    - Adding stuff to something that was not committed. Slide 38. You should return to the state before anything happened, but in this example, T2 already committed.
- Write-Write Conflicts (lost update): one transaction overwrites uncommitted data from another.

Formal Properties of Schedules:
Given these conflicts we can check whether schedules are correct (we can't generate a correct schedule yet).

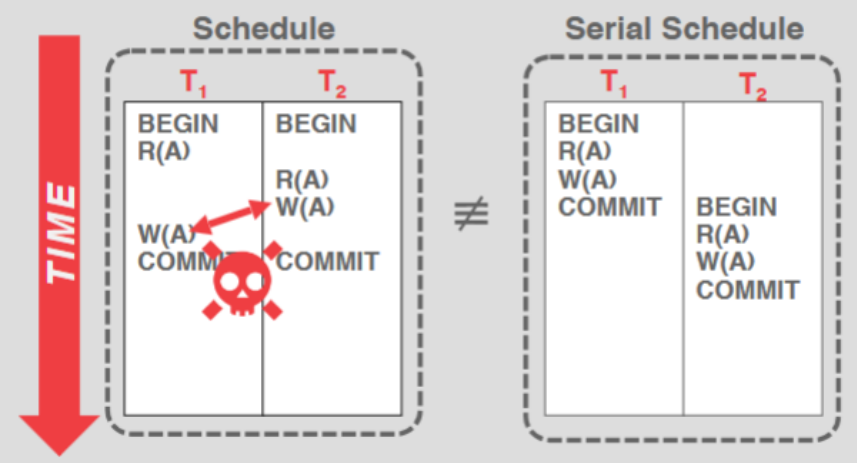
Two different levels of serializability

- Conflict Serializability
    - Two schedules are conflict equivalent iff:
        - They involve the same actions of the same transactions.
        - Every pair of conflicting actions is ordered the same way.
    - Schedule S is conflict serializable if:
        - S is conflict equivalent to some serial schedule

- You can transform S into a serial schedule by swapping consecutive non-conflicting operations of different transactions.



Non-example:



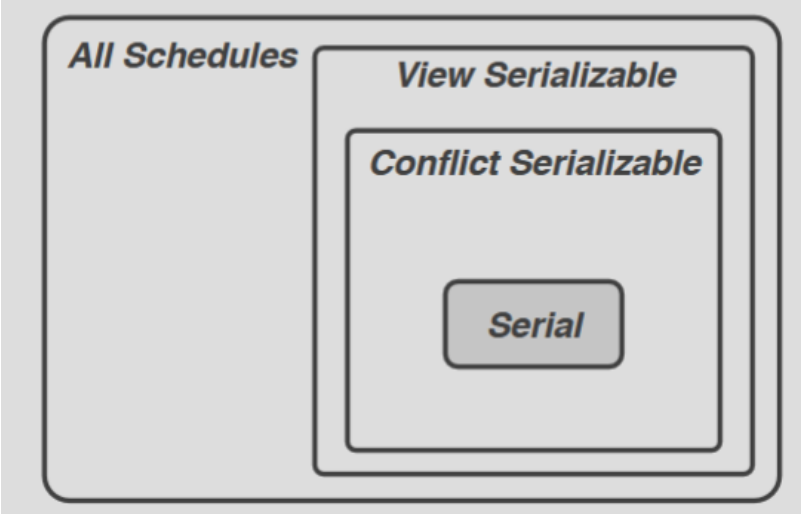
To develop a faster algorithm to test a schedule for conflict serializability, use a dependence graph (aka precedence graph).

- One node per transaction.
- Edge from T_i to T_j if an operation O_i of T_i conflicts with an operation O_j of T_j and O_i appears earlier in the schedule in O_j.
- A schedule is conflict serializable iff its dependency graph is acyclic.

Example 1: slide 15.
W(A) → R(A) since we did W(A) in T1 and reading that is dependent on that write.
Similarly, W(B) → R(B), so cycle.

Example 2: slide 16. Based on this dependency graph, the serialized version should have T3 going after T2 (so the serial order would be T2, T1, T3).

- View Serializability (more of a theoretical thing)
  Example on slide 49 is not conflict serializable but is view serializable. This is because even though you do a bunch of writes, you never check them. This is an example of blind writes.



Durability

All changes of committed transactions should be persistent. If we told the user that we committed, we must have committed. DBMS can use logging or shadow paging.

Lecture 12 (3/3) - Two-Phase Locking

A way to guarantee that all execution schedules are correct (i.e., serializable) without knowing the entire schedule ahead of time.

Lock Types

Slide 6 table.

- For “During”, locks are for entire transactions (for example, to ensure serializability) whereas latches are lower level.

Basic Lock Types (like read and write latches):

- S-Lock: shared locks for reads.
- X-Lock: exclusive locks for writes.

Executing with Locks:
Transactions request locks. The lock manager grants or blocks requests. Transactions release locks.
The lock manager has an internal lock-table separate from the data itself (unlike the latches being stored in the B+Tree for example).

Locking does not guarantee that transactions are serializable.

Two-Phase Locking

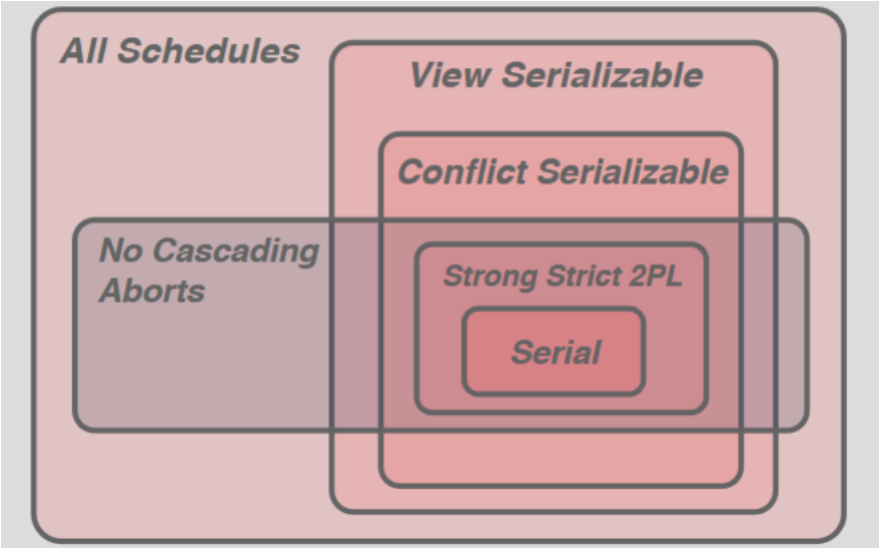
A concurrency control protocol that determines whether a transaction can access an object in the database at runtime. The protocol does not need to know all the queries that a transaction will execute ahead of time.

1. Growing phase: each transaction requests locks from the lock manager. The lock manager grants or denies these requests.
2. Shrinking phase: the transaction is only allowed to release/downgrade (X lock to S lock) locks it previously required and cannot acquire new locks.

2PL guarantees conflict serializability because the graph is acyclic, but there is a problem with cascading aborts. Causes wasted work.

To deal with this, use strong strict 2PL:

- Only allow a transaction to release locks after it has ended (i.e., committed or aborted).
- Allows only conflict serializable schedules, but stronger than regular 2PL.
- A schedule is strict if a value written by a transaction is not read or overwritten by other transactions until that transaction finishes.
- Advantages:
    - Does not incur cascading aborts
    - Aborted transactions can be undone by just restoring original values of modified tuples.
- Disadvantages:
    - Slower, less concurrency



Deadlock Detection and Prevention

A deadlock is a cycle of transactions waiting for locks to be released by each other. 2 Approaches

1. Detection: DBMS creates a waits-for graph to keep track of what locks each transaction is waiting to acquire (quite similar to the dependence graph). Edge from T_i to T_j if T_i is waiting for T_j to release. DBMS must check for cycles and break them.
    - When a deadlock is detected, DBMS picks a "victim" transaction to roll back. Victim will either restart or abort. Picking the victim (a lot of variables):
        - Age (lowest timestamp which would mean a lot of things locked or most recent which means fewer items locked) (tradeoff for this)
        - Progress (least/most queries executed)
        - The number of items already locked
        - The number of transactions that we have to roll back with it
        - Should also consider the number of times a transaction has been restarted to avoid starvation.
    - Tradeoff between frequency of checking for deadlocks and how long transactions wait before deadlocks are broken.
    - Rollback length: after selecting a victim, need to decide how far to roll back the transaction's changes. Two approaches:
        - Slide 32
2. Prevention: when a txn tries to acquire a lock that is held by another txn, the DBMS kills one of them to prevent a deadlock. Doesn't need the graph, but also could lead to killing too many. Assign priorities based on timestamp where older means higher priority.
    - Wait-Die (old waits for young): if requesting txn is higher than holding, then requesting waits for holding. Otherwise, requesting aborts.
    - Wound-Wait (young waits for old): if requesting has higher priority, holding aborts. Otherwise, requesting waits.

    Slide 35 has example. This works because there is only one direction.
    Note: when a txn restarts, keep its original priority to prevent starvation.

Hierarchical Locking

Note that the above assumes that we are only acquiring locks for individual tuples, but in practice, we want to obtain the fewest number of locks possible (for example, if we want locks on all tuples, but there are a million tuples, would need a ton of locks, so instead, use intention).

<u>Lock Granularity</u>
When a … slide 38
Slide 39 for hierarchy

Intention Locks

Allows a higher-level node to be locked in shared or exclusive mode without having to check all descendent nodes. Three types:

1. Intention-Shared (IS): Explicit locking at lower level with S locks.
2. Slide 41

41-43

(Taking a regular shared lock on a parent implicitly means a shared lock on all of its children, whereas with intention, it tells us that some of the children have a shared lock)

The three transactions example
T1 needs SIX because we need to read all but only exclusive lock one. (Choose this one because 1 is much smaller than the total number of tuples. If you need to write a majority, would just do IX.
T2 needs IS because it's reading only a single tuple.
T3 has to wait because it scans all tuples and wants an S.

Lock Escalation
The DBMS can automatically switch to coarser-grained locks when a txn acquires too many low-level locks.

Locking in Practice
Slide 49

Lecture 13

Alternative to 2PL (which is pessimistic). Timestamp Ordering is optimistic. Slide 5-6

Basic Timestamp Ordering (T/O) Protocol

- Txns read and write objects without locks.
- Every object X is tagged with timestamp of the last txn that successfully did read/write: W-TS(X) is write timestamp on X; R-TS
- Check timestamps for every operation: it txn tries to access an object from the future, it aborts and restarts.

Reads
Writes

Example 2: abort T1

-

Optimistic Concurrency Control

NOTE: Exam will not include multi

Phases:

3. Read: Reading objects into private workspace to do reads and writes. Once you're there, no one else outside of the transaction can do anything with it. Slide 18

4. Validate: when a txn commits, check whether it conflicts with other txns; assign TS at commit time to ensure no conflicts. Slide 19 (two approaches):
    a. Backward validation: ensures that any transactions made in the past must be visible to the future transaction, i.e., checking whether the committing txn intersects its read/write sets with those of any txns that have already committed. Slide 20 picture.
    b. Forward validation: ensures that any reads or writes of currently running or things that may commit in the future are visible to ?? i.e., check whether the committing txn intersects its read/write sets with any active txns that have not yet committed. Slide 12 picture. Each txn's Ts is assigned at the beginning of the validation phase. Check the timestamp ordering of the committing txn with all other running txns. If TS(T_i) <TS(T_j), then one of the following conditions must hold:
        i. T_i completes all three phases before T_j begins execution. → This means there is serial ordering.
        ii. T_i completes before T_j starts its Write phase and T_i doesn't write to any object read by T_j, i.e., WriteSet(T_i)\cap ReadSet(T_j) = empty. If not, T_i has to abort.
            1. Second example validation TS of T_2 is lower than previous so it logically thinks it happened first. Slide 25 and 26
        iii. T_i completes its read phase before T_j completes its read phase and T_i does not write to any object that is either read or written by T_j (i.e., Write(T_i) \cap ReadSet(T_j) = empty and WriteSet(T_i)\cap WriteSet(T_j) = empty).
            1. Slice 28 example
5. Write: if validation succeeds, apply private changes to the database. Otherwise abort and restart the txn.
    a. Serial commits: use a global latch to limit a single txn to be in the validation/write phases at a time.
    b. Parallel commits: Use fine-grained write latches. Txns acquire latches in primary key order to avoid deadlocks.

Observations:

- OCC works well when number of conflicts is low.
    - Ideal scenario is all txns are read-only.
    - Txns access disjoint subsets of data.
    - If database is large and the workload is not skewed, then there is a low probability of conflict, so locking is wasteful.
- Performance issues:
    - High overhead for copying data locally.
    - Validation/Write phase bottlenecks.
    - Aborts are more wasteful than in 2PL because they only occur after a txn has already executed. Whereas in 2PL in mid transaction, if you can't acquire lock, abort right then.

Note that this assumes that we are not adding or removing from the database (static).

<u>The Phantom Problem</u>

T1 locked existing tuples and not newly inserted ones.

Approaches: (of course you can lock the entire table, naive approach)

1. Re-Execute Scans: run queries again to commit to see whether they produce a different result to identify missed changes.
    - Keep track of WHERE clause for all queries. Retain the scan set for every range query in a txn.
    - At commit time, re-execute just the scan portion of each query and check whether it generates the same result. Was there a change that we didn't see?
    - Works if your scan range is relatively small; small updates (OLTP systems)
2. Predicate Locking: locking a logical range and determining the overlap of predicates before queries start running. In the example, we want to lock all tuples where status is x. Not really practical
    - Shared lock on the predicate in a where clause of a select query.
    - Exclusive lock on the predicate in a where clause of any update, insert, or delete.
3. Index Locking: use keys in indexes to protect ranges. Special case of predicate locking that is potentially more efficient.
    - E.g. in B+Tree, suppose we are at root and we want to look at ages 20-30. So lock that range to prevent any insertion in that range.

Isolation Levels

Serializability is useful because it ignores concurrency issues, but enforcing it can allow too little concurrency. We may want to use a weaker level of consistency to improve scalability (e.g. if you don't need a precise number, it's ok to have that phantom situation).

Isolation levels control the extent that a txn is exposed to the actions of other concurrent txns.

Isolation levels (high to low): table on slide 42.

- Serializable: no phantoms, all reads repeatable, no dirty reads.
    - Obtain all locks first, index locks, strict 2PL (hold all locks until the end)
- Repeatable reads: phantoms may happen (most databases do this).
    - Same as above but no index locks.
- Read committed: phantoms and unrepeatable reads may happen.
    - Same as above but S locks are released immediately.
- Read uncommitted: all of them may happen.
    - Same as above but allows dirty reads (no S locks).