This Lecture

- How the files are stored on rotational media
- Strategies for reading/writing/storing database records
- B-Tree indexes

Hard Drive Basics

- Thin metal platters coated with a magnetic material, rotating at high speed. Data bits are stored as differently oriented magnetic domains.
- Servo motor assembly. The read/write heads are on the end of an array of arms which can swing to different distances from the disk center, thus accessing different “tracks”.
- Read/write heads, one per platter side. These float extremely close to the platters and detect or modify magnetic domains as the spinning disk brings them under the heads.

Data Organization

- Data is stored in tracks, which are further broken into sectors.
- Sectors typically hold between 512 – 4096 bytes.
- At the hardware level, an entire sector is read or written at one time. However, from the operating system perspective, an entire block is read/written. A block is a higher level abstraction which may encompass multiple sectors.
- For the rest of this discussion, we’ll assume a block == a sector.

Access Time

- A block can be read or written extremely fast... once found.
- The cost to read a block is mostly due to the access time, the sum of:
  - Seek time – time to move read/write heads to the right track
  - Rotational latency – time to rotate until block is under read/write head
- Some numbers:
  - Typical average seek time for a drive: 9ms
  - Rotational latency for a high-end server drive (15,000 RPM): 2ms
- Notes:
  - Caching, buffering, and other clever optimizations make these numbers hard to measure exactly
  - Newer storage technologies (SSD, Optane, etc.) have completely different performance characteristics (for now, HDDs are still dominant in DB storage)
**Key Insight**
- A program can do almost anything with a block of data *in memory* on a modern CPU in much less time than 10 ms
  - That is, compared to a read, time to process the data is negligible
  - Disk I/O is thus the primary bottleneck for database query latency

**FILE ORGANIZATION**

**Organizing Records**
- While a key facet of the modern database is data abstraction, the data ultimately has to be stored somewhere/somehow
  - Intersection of hardware, software, and algorithms
  - HDD technology has shaped the technology for 40+ years
  - Many approaches to this in various databases
  - We’ll explore a popular approach organized around primary keys
    - This scheme will lead us to the idea of a hierarchical index and B-Trees
    - Note that PostgreSQL (and probably others) use different schemes

**Ordered Storage by Primary Key**
- Store multiple records in each block
  - Within blocks, order by primary key
  - Maintain a list of blocks as well, ordered by primary key of first record in block
- Note we can now do binary search \(O(\log_2 n)\) lookups:
  - Find a block holding a particular key
  - Find a particular record within a block
- Issues:
  - Insertion/deletion – what to do when out of room / how to fill gaps?
  - Scaling: is it fast enough?
  - What if we want to search by something other than primary key?

**Issue 1: Insertion/Deletion**
- When inserting a record, if no room:
  - Must move records aside to make room
  - This is expensive if we keep everything closed up tight
- Solution: keep some “spare” room around:
  - If no room, split block into two blocks – now each is half full and insert is cheap
  - Good performance, worst-case 2x storage requirement
- Similarly, when deleting, blocks may become largely empty:
  - Requires lots of disk access for relatively few records stored
  - Solution: merge adjacent blocks when less than half full

**Issue 2: Scaling/Performance**
- Suppose we have a table with:
  - 100M records
  - 100 records per block (max)
- Assume a disk with 10ms access time
- What is cost to find a record given a primary key value?
  - \(1M = 2^{20}\) blocks, so binary search→ must read 20 blocks in worst case
  - \(10\text{ms/block} \times 20\text{ blocks} = 200\text{ms}\)
- OK, that sounds fast, but consider a modern transaction processing system (e.g., stock trading) handling thousands of queries per second!
Second Level Index
- Obviously this is not fast enough. What to do?
- Solution: create an index, a kind of table storing primary keys together with pointers to the blocks containing them.
  - Each record represents the first key in a block
  - We can now stuff ~100 or more index records into a block
- How does this help?
  - Now search 1/100th number of blocks: only search 10,000 = $2^{14}$ blocks
  - Cost is ~140ms to search index + 10ms to lookup referenced block

Additional Levels
- 150ms still not good enough:
  - Make another index indexing the second level index
  - Another 100x reduction $\Rightarrow$ 70ms + 10ms + 10ms cost
  - Repeat again until all keys (at top level) fit into one block. Cost ~30ms
  - Cost of searching with hierarchical index no longer $O(\log_2 n)$, more like $O(\log_{100} n)$
- Generalizing this approach leads to the B-Tree data structure
  - By default, all indices in PostgreSQL use B-Trees
  - Not limited to primary keys!

B-Tree Data Structure
- A balanced search tree with a high number of keys in each node
- Basic structure allows efficient searching much like binary search tree:

B-Tree Rules (Knuth)
- For an order-$m$ B-Tree:
  - Every node has at most $m$ children
  - Every non-leaf node (except root) has at least $\lceil m/2 \rceil$ children
  - The root has at least 2 children unless it is itself a leaf
  - A non-leaf node with $k$ children has $k - 1$ keys
  - All leaves appear on the same level
- For $n$ keys:
  - Best case height is: $\lceil \log_m(n + 1) \rceil$
  - Worst case: $\lceil \log_{m/2} \left( \frac{n+1}{2} \right) \rceil$

Insertion
- Values are always inserted at a leaf node:
  - If leaf has no room, split the leaf and promote the median value to the parent node (becomes the separator between the two new children)
  - If parent leaf in turn has no room, recursively split/promote
  - If splitting/promotion reaches root and root has no room, split root and add a level to the tree
Deletion

- If value is in leaf, remove, then rebalance if underflow (too few keys in node)
- If internal node, then find nearest leaf descendant to replace it with – rebalance leaf if this results in underflow
- Rebalancing:
  - If right sibling exists and has more than minimum # of elements, do a left rotation to borrow one element
  - Otherwise, if left sibling, etc.
  - Else both the node and its sibling are small enough to merge
    - Have to pull separating element from parent into merged node
    - This can cause underflow in parent – deal with recursively

Kinds of Indexes

- Earlier section described data itself being stored in a sorted order (by primary key)
- Some databases (e.g., MS SQL Server) do this (“clustered index”)
- Indices other than primary key are called “Secondary indexes”
  - Tiny extra overhead because secondary index must store some kind of row pointer(s) to correct block on disk
  - Extra lookup of actual data block since not stored in index
- PostgreSQL only has secondary indexes
  - Tradeoff: more flexibility
- Other considerations: indexes on non-unique columns
- We’ll talk more about indexing in practice when we talk about query optimization

Database Access Performance

Actual indexed query performance (e.g., flowers) tends to be better than our 30ms example:

- Indexes tend to be on small keys, blocks have gotten large (4 – 16 Kb) – so can fit way more than 100 keys into an index block (e.g. 4Kb -> 1024 integer keys)
- Caching of important disk blocks in memory speeds up repeated accesses – whole indices or even tables can fit in memory!
  - First read may take 10-20ms
  - Subsequent reads may be sub-millisecond
- Clever disk tricks:
  - Striping
  - Storing indices on different drive than data
  - Sharding