Database Internals Masterclass

"A line-by-line revision guide from raw transcript to expert architecture."

PART 1: THE THREE FACTS

❝ TRANSCRIPT

"hey everyone and welcome back this is my second shot at recording for this series... apologies for the technical difficulties last time my iPad recording was getting turned off more frequently than my ex-girlfriend whenever I tried to make sexual advances... we established basically three facts about the data. Number one is the following that we want the data to [be persistent]. Number two is that we want to be doing this on a hard drive... RAM does not store data between computers shutting off and on... The third thing is that the hard drive is in a database or at least we're running database software on some computer."

EXPERT REVISION: THE DURABILITY PROBLEM

Fact Check: Persistence is the 'P' in most architectural discussions (and the 'D' in ACID). The core challenge is that we are storing software-logical structures on physical, mechanical hardware.

Example: Think of **RAM** like a whiteboard (fast to write on, wiped at night) and a **Hard Drive** like a stone tablet (hard to carve, but permanent). A Database is the specialized tool used to carve and read those tablets efficiently.

PART 2: PHYSICAL HARDWARE

❝ TRANSCRIPT

"we've got basically some rectangle with a metallic disc and then some moving arm... the closer that you put data together on disk the faster it is to access... for example if I have one piece of data here and the other here that's kind of bad... instead it would be better if we had our first piece of data here again and the second one right next to it."

EXPERT REVISION: MECHANICAL LATENCY

The "Moving Arm" Problem: This refers to **Seek Time**. Physically moving the arm takes ~5-10ms. In computer time, this is an eternity.

The Proximity Example: If "Jordan's" data is on Sector 1 and "Donald's" data is on Sector 1000, the disk arm must physically move across the platter. If they are in Sector 1 and 2, the data is read in a single "glance" as the disc spins.

PART 3: COMPLEXITY ANALYSIS

❝ TRANSCRIPT

"let's imagine a website right where basically all it does is it stores a bunch of names... Jordan size 18... Donald... size 7... Shaq size 24. in order for me to find rows where name is Jordan I have to check each row individually... what that means... is that this is going to be an O(n) time complexity."

EXPERT REVISION: THE LINEAR SCAN

Linear Search Performance: When data is unindexed, the database must perform a "Full Table Scan." Every single page on the disk must be loaded into memory and checked. Work done $\propto$ Total Rows (N).

TABULAR WALKTHROUGH: SEARCHING FOR "SHAQ" (TRANSCRIPT EXAMPLE)

Step	Disk Block	Data Found	Operation	Status
1	0x0001	[Jordan, 18]	Check "Jordan" == "Shaq"	❌ Skip
2	0x0002	[Donald, 7]	Check "Donald" == "Shaq"	❌ Skip
3	0x0003	[Shaq, 24]	Check "Shaq" == "Shaq"	✅ Match

Conclusion: In the worst case (e.g. searching for a name not in the list), we check every block in the table.

PART 4: THE APPEND-ONLY SHIFT

❝ TRANSCRIPT

"instead of actually doing any physical edits... add a new row to the table where I would now just say Donald with the new shoe size... realized now it's a six... search from the bottom of the table up... we can basically keep a pointer to that address and disk and that makes right times o of one."

EXPERT REVISION: LOG-STRUCTURED WRITES

Append-Only Log: Instead of searching ($O(N)$) to find "Donald" and overwrite his size, we treat the database like a notebook where you only add lines at the bottom. This makes writes **$O(1)$ Constant Time**.

TABULAR WALKTHROUGH: APPENDING DONALD (7 → 6)

Offset	Timestamp	Record	Logical Status
0x001	12:00:01	[Jordan, 18]	Active
~~0x002~~	~~12:00:05~~	~~[Donald, 7]~~	~~Obsolete (Shadowed)~~
0x003	12:00:10	[Shaq, 24]	Active
0x004	**12:05:00**	**[Donald, 6]**	**Current (Latest)**

ℹ The Read Strategy:
To find Donald's current size, the DB scans **backwards** from 0x004. As soon as it hits "Donald" at 0x004, it returns "6" and stops. The old "7" at 0x002 is ignored. This makes writing instant, but search still takes linear work ($O(N)$).

PART 5: SCALING TO FACEBOOK

❝ TRANSCRIPT

"o of n reads are basically slower than Joe Biden trying to read from a teleprompter and he is a boomer... think about Facebook for example... if I have to go through every single Post in the database in order to go ahead and load our news feed that is going to be impractical... this is where the need for indexes come in... show rows with names between A and B... posts from you know one hour ago until now."

EXPERT REVISION: INDEXING & RANGE QUERIES

The Scalability Ceiling: $O(N)$ is only fast when N is small. At Facebook scale ($N = 10^{12}$ posts), linear scanning is physically impossible.

Range Query Example:
"Give me names between A and B (e.g., Abraham Lincoln and Alex Adams)."
Without an index, you scan all 1 billion rows. With a sorted index, you jump to 'A', read until 'B', and stop.

SYSTEM DESIGN QUOTE

"We care a lot more about read speeds than write speeds... indexing can help improve read speeds even if it makes writing more expensive... it's still a big win in the average case for user experience."

Transcript Executive Summary

Core Mechanics

- **Durability:** Persistence requires writing to mechanical disks, not just volatile RAM.
- **Disk Locality:** Placing data "next to each other" minimizes Arm Seek Time, which is the primary driver of performance.
- **The Array Model:** Conceptualize the disk as one giant byte array where indices are offset pointers.

The Trade-off Cycle

- **Naive State:** Both reads and writes are $O(N)$ because finding data takes linear effort.
- **The Optimization:** Using **Append-Only Logs** converts writes to $O(1)$ by removing the "find" step.
- **The Price:** Append-only logs grow N faster and make $O(N)$ reads even slower over time.
- **The Solution:** Database **Indexes** (Keys and Ranges) provide the direct jump needed for modern scale.

RAW VIDEO TRANSCRIPT

Full Verbatim Script

hey everyone and welcome back this is my hey everyone and welcome back this is my second shot at recording for this series second shot at recording for this series uh apologies for the technical uh apologies for the technical difficulties last time my iPad recording difficulties last time my iPad recording was getting turned off more frequently was getting turned off more frequently than my ex-girlfriend whenever I tried to make sexual advances uh but the great to make sexual advances uh but the great thing about this series is that I can thing about this series is that I can really crank these out really crank these out um I'm not going to finish that joke off um I'm not going to finish that joke off uh because i have pretty much already uh because i have pretty much already made them all before so as you can see made them all before so as you can see I'm making my second one of the day I'm making my second one of the day which is great so um you know if you which is great so um you know if you recall from last time we started to get recall from last time we started to get into the contextualization of the into the contextualization of the database problem how we want to actually database problem how we want to actually organize our data that we can read fast organize our data that we can read fast write fast keep it persistent and so write fast keep it persistent and so today what we're going to do is take a today what we're going to do is take a look at the actual internals of a look at the actual internals of a database and how we want to put our data database and how we want to put our data on the disk in order to maximize on the disk in order to maximize performance all right welcome back performance all right welcome back everybody I have now gotten the iPad and everybody I have now gotten the iPad and I'm ready to do some work you may see I'm ready to do some work you may see this is a different day I'm just coming this is a different day I'm just coming home from the office right now and I home from the office right now and I have to

Next Up: **Hash Indexes Deep Dive**