CLOUD SERVICES / DATABASES / SOFTWARE DEVELOPMENT

Database Scalability and the Giant Flea: A Lesson in Complexity

As databases scale, data distribution and consistency become harder to manage.

Feb 11th, 2025 11:00am by Sunny Bains

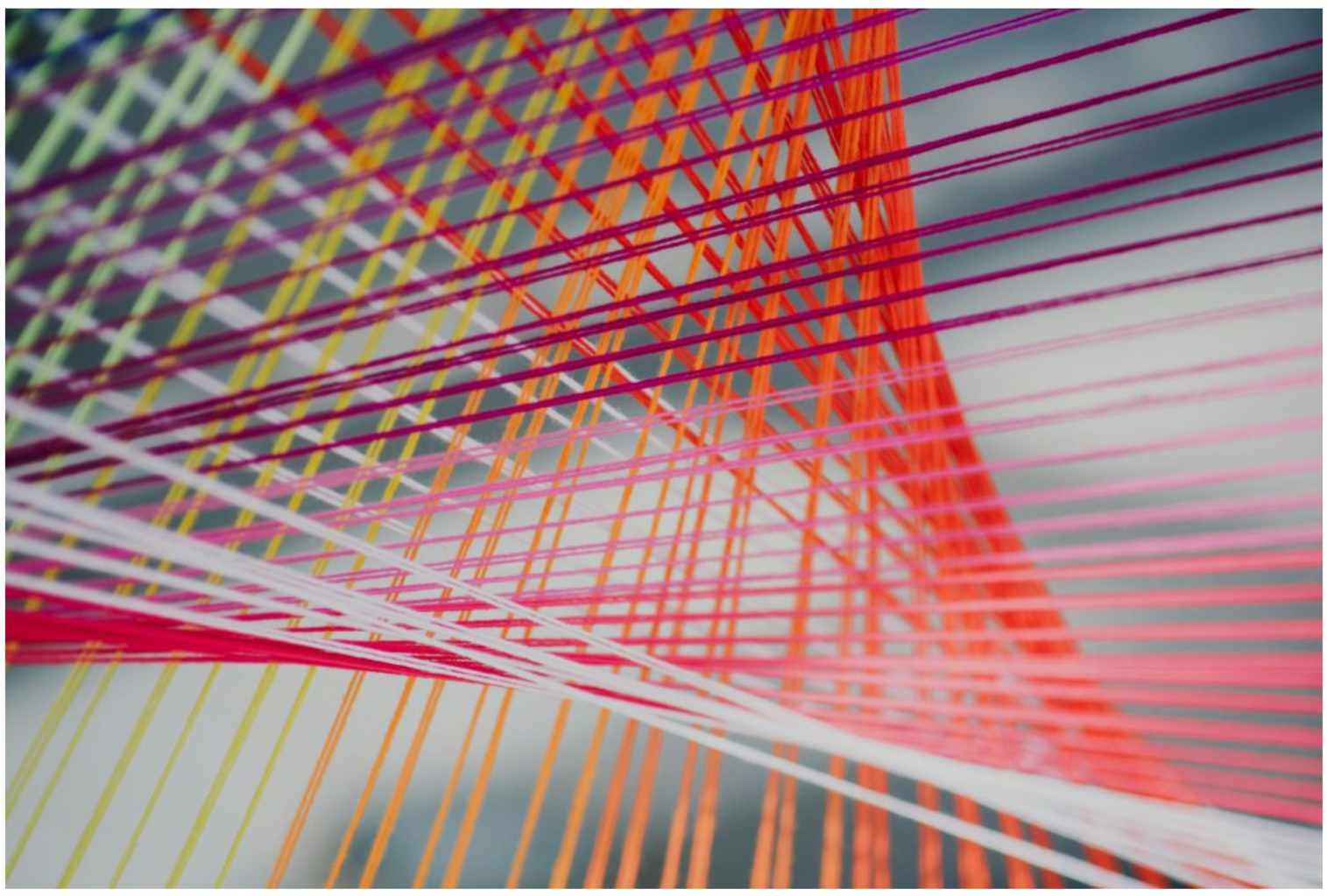

Photo by CHUTTERSNAP on Unsplash.

Consider the flea. It's a tiny creature, barely visible to the naked eye. Yet it's powerful, capable of leaping over 100 times its body length. Now imagine scaling the flea to the size of a human. Suppose you could make it physically identical to its smaller form, just larger in every dimension.

This scaled-up super flea could jump a 30-story building. But it can't. The reason concerns a principle of biomechanics called the square-cube law. As the flea "scales up," its volume (mass) increases faster than its surface area. The larger it grows, the less capable it is of supporting itself. Eventually, the flea collapses under its weight.

The lesson? You can't just expand a small one to get a giant flea. You need to design a different flea.

Something similar is true about scalability and database architecture. A system optimized to deliver X performance at a small scale probably won't deliver 10,000X performance at a 10,000X scale. Its performance is likely to deteriorate as it grows. As the number of nodes and links increases, so does the complexity of managing the database. This is primarily due to the increased need to manage data distribution and inter-node communication and maintain consistency across all nodes. This can lead to configuration, monitoring, and troubleshooting challenges, especially in large clusters with many nodes. The more nodes you add, the harder it becomes to track where data resides and how to access it across the cluster efficiently.

The Nature of Operational Complexity

Complexity has two main sources: technological and operational. As the name suggests, technological complexity refers to the purely technical aspects of creating a system that can operate at scale. These might include automating scale-out, managing containers, distributing storage, provisioning complex networks, and abstracting manual operations. Most people think of these challenges when they think of scaling software.

SHARE THIS STORY

TRENDING STORIES

1. Vector Search Is Reaching Its Limit. Here's What Comes Next
2. Database Categories Are Dead: Here's What's Next
3. When Your Relational Database Isn't the Right Tool Anymore
4. Kubernetes Finally Solves Its Biggest Problem: Managing Databases
5. Why Startups Are Betting Everything on Apache DataFusion

Operational complexity is the flip side of technological complexity. It refers to the difficulty of managing and using a system at scale. It's the time and labor tax a technology system imposes on its operators. It's all the work that isn't automated but must be done for the system to function. Because it is wrapped up in human factors that are hard to quantify, operational complexity is less appreciated and understood than technological complexity. Still, I would argue it's just as significant.

In the cloud, we've figured out how to build databases that can grow as large as anyone wants. By abstracting the mind-bending complexity of hyperscale infrastructure, the cloud has made it possible to build technologically simple systems while delivering "speeds and feeds" that would have been mind-boggling just a short time ago.

TRENDING STORIES

1. Vector Search Is Reaching Its Limit. Here's What Comes Next
2. Database Categories Are Dead: Here's What's Next
3. When Your Relational Database Isn't the Right Tool Anymore
4. Kubernetes Finally Solves Its Biggest Problem: Managing Databases
5. Why Startups Are Betting Everything on Apache DataFusion

That sounds great. But there's a downside. We've created databases that can *technically* run at any size but are not practical to *operate* at scale. As an industry, we're still struggling with the human problems of running and managing hyperscale systems.

Operational Complexity — Sharding

A classic example of this phenomenon is sharding, which is the practice of subdividing a data store into more manageable fragments. The most common form of sharding, "naive sharding," assumes a uniform distribution of data (an important assumption, but we'll get to that). The performance is fantastic. Latency is low. Throughput is high. Everything is good. From a technological point of view, sharding seems like an excellent way to maintain performance as a database grows.

From an operational standpoint, it looks different. Sharding a database creates a new problem: locating data across shards. In naive sharding, the database doesn't know where a given piece of data lives. The "sharding key" information lives at the application layer. This means keeping track of data is now the developer's problem.

When a database is relatively small, say 10 GB and the number of shards is relatively few, this is a manageable task. But now, the database has grown to 100 GB or 200 GB. Each shard is now many times the size of the entire database when the initial sharding occurred. Data is no longer equally distributed between the shards. As a consequence, massive hot spots start to appear. Performance plummets. It's time to reshard the database.

Now, operational complexity kicks in. The data must be manually redistributed and rebalanced, creating new shard keys. Sharding keys already hard-coded into existing applications must be updated because sharding in this design is explicit. The application must be aware of the data's location. Each line of code that uses a sharding key needs to be reviewed. Data that lived on the same shard before now must be manually assigned to separate shards, and the code needs to be refactored accordingly.

This task is the stuff of nightmares. No grep find and replace can handle it automatically. With their limitations, humans must grapple with the operational complexity sharding has created.

Operational Complexity — Scaling Out

Operational complexity increases as an organization adds servers or nodes to the system. Nodes are generally added for redundancy and load sharing, so changes on one node must be reflected on the others, or the data must be spread out to avoid hotspots. Moving data is the same as resharding. You want the movement of data in the cluster to be invisible to the user. It's the job of the distributed SQL front-end nodes to fetch the data from whichever node it's located on.

In most databases, including MySQL, a change committed to one node might, in the worst case, take seconds or even minutes to register on the others due to replication lag. As an example, one culprit of replication lag is DDL. A DDL operation on the primary node can block the propagation of events to the

follower nodes. As a result, until the change has propagated across all nodes in the database, a user might see different results depending on which node their query is directed to.

It's a trade-off known as "eventual consistency." It's the rule with MySQL and Amazon Aurora, MySQL Postgres, and any databases built on the MySQL model. (MySQL Group Replication does, technically, offer a strong consistency mode, but it's rarely used as the documentation indicates that it's to be used at one's own risk.)

Why does this matter? It probably doesn't if the change in question is a "like" on a social post. But if someone is withdrawing money from their bank or trying to sign in after changing their password, it matters very much whether every node has the same information at the same time.

Here again, the burden falls on the developer. Since the underlying system doesn't guarantee strong consistency, the developer needs to devise a workaround in code. These limitations are not apparent in a single-node database. They result from architectural decisions that make perfect sense on a smaller scale. However, this weaker consistency mode makes the developer's job harder and more complex as the system grows.

Tradeoffs and Solutions

In software engineering, as in life, there's no free lunch. Skimp on automation and abstraction (i.e. technological complexity), and you get more operational complexity. Simplify operations, and you need more complex technology to compensate. NoSQL database MongoDB was partly inspired by frustration with schema-based DDL operations. However, stripping out schemas, as MongoDB and other NoSQL solutions do, creates scalability problems. Among other issues, it makes analytics much more challenging to run in a large organization.

As I've written elsewhere, this is why NoSQL implementations tend to fail at scale. It's not because they lack the sheer technological capacity but because they have trouble addressing the operational needs of a large enterprise.

Complexity is unavoidable. It's just a question of what form it takes, in what proportions — and who "pays" for it. We can't eliminate technological and operational complexity, but we can try to find a middle ground between the extremes.

The most recent attempts to do this are distributed SQL solutions like CockroachDB and TiDB, based on the technology behind Google's Spanner project. Distributed SQL systems rely on a traditional relational schema but automate horizontal scale-out, DDL replication, and other functions that have historically limited the ability of structured databases to grow.

Distributed SQL doesn't aim to hit *quite* the raw storage and retrieval speed levels a NoSQL database can provide or to replicate the sheer simplicity of a single-node MySQL database. The goal is to find a spot between performance (i.e. speed, latency), reliability, versatility, and ease of management. In practice, this means increasing the technological complexity under the hood to reduce the operational complexity tax on the end user.

Right now, most large data systems are like our hypothetical giant flea, at a stage somewhere short of keeling over. They're still recognizably themselves. But they *function* worse than they did when they were smaller. And the larger they get, the worse they do.

Most database development is about redesigning the flea. That could take many forms. Distributed SQL is one approach, and others are possible. However they approach the problem, they must find a solution to operational complexity. What we have now is scale on paper. What businesses need is scale in practice.

TNS

Sunny Bains is a software architect at PingCAP, the company behind TiDB. He has worked on storage engines for more than 22 years. His first acquaintance with database kernel work was in 2001, when he was tasked with writing a...

Read more from Sunny Bains →

The Art and Science of Sizing Search Nodes
12 August 2025

Boost Connected Car Developments with MongoDB Atlas and AWS
11 August 2025

You Don't Always Need Frontier Models to Power Your RAG Architecture
11 August 2025

Real-time CDC for inventory tracking with PostgreSQL
12 August 2025

Exporting data from Redpanda to S3 in batched JSON
7 August 2025

Redpanda 25.2: Advancing Iceberg integrations for the real-time lakehouse
5 August 2025

Tabnine Adds Support for NVIDIA Nemotron Models, Bringing Powerful New Reasoning Capabilities to Enterprise AI
11 August 2025

Accelerating Enterprise AI: Tabnine Integrates NVIDIA Universal LLM NIM Microservices for Secure, Scalable Deployments
11 June 2025

What It Really Takes to Be Air-Gapped: Inside the Architecture of Secure AI Development
11 June 2025