***T. Rehenel**, BA student*
***M. Koshelieva**, senior lecturer*
*Zhytomyr Polytechnic State University*

# PURPOSE, APPLICATION AND TRADEOFFS OF DUAL-DB APPROACH

The aim of this study was to explore the dual database approach in software architecture. After identifying the problem, the core idea and most common forms of this approach were explored and tied to their application. Finally, general tradeoffs and challenges of this approach were addressed alongside the most common ways to mitigate or overcome them.

As information technology serves more fields and solves an increasingly wide range of problems at global scale, software systems load increases accordingly. Critical systems and often business needs require software to be reliable and resilient.

The most basic way of making the system able to handle more load is vertical scaling – allocating more hardware resources. However, vertical scaling is limited by the hardware itself, and at its extreme can introduce unnecessary high infrastructure costs. Thus, distributed systems and robust software architecture are relevant research topics and corresponding knowledge and skills are highly valuable.

In its essence, all the software can be described in three words: algorithms on data. This perspective implies that there are two key components in every system at the most general level: data and its processing.

For further overview, as most of modern high load systems are operating in the web space – web platforms, “software as a service”, cloud computing and others – it makes sense to split the data component in two, more specific and different in their nature components: data storage and data transfer. This allows us to explore solutions, patterns and tools independently for processing (compute), persisting (storage) and exchanging (transfer) data in high load systems. This abstract is focused on the storage component.

Besides vertical scaling there is horizontal scaling – instantiating multiple nodes and distributing load across them instead of allocating more resources to a single node. This technique can be applied not only for the compute component, but also for the storage. Compute in its general case is stateless, which makes its horizontal scaling easier as all nodes can be considered equal and independent. However, that is not the case for storage, as data is the state itself, and cannot have multiple independent instances representing the same entity. This leads to having different strategies for scaling storage horizontally.

One of the ways to mitigate this complexity is partitioning and sharding, which introduces a way to distinguish state by some criteria (date, id, hash, region), and use it to distribute load (and state itself) across independent storage instances. It works, but introduces complexity of routing, rebalancing and cross-shard queries. Also, relational databases are generally considered harder to shard.

Another two important factors are the kind of stored data and access patterns. These factors are the foundation for horizontal storage scaling, as they allow

introducing different roles and distributing load through distributing responsibilities. Thus, there's no attempt to pack data and queries that differ in their nature in a single place. Instead, the best-suited database engine, environment and tools can be used for each kind of data individually [2, p. 2]. That is the core idea of the dual-database approach: there's no data storage that is optimal for all kinds of data and access patterns.

This core idea is implemented in a variety of forms, one of them being CQRS at the database instances level: one database instance acts as a write replica, receives write queries, prioritises ACID and relational integrity, while other instances act as read replicas serving a projection of the written data shaped for specific query needs. Such separation brings several benefits: independent scaling for read and write needs (valuable when read/write ratio is very uneven, e.g. in social media platforms), or ability to use database engines that are adapted better for read queries, e.g. Elasticsearch for text search or Cassandra for wide-column reads, vector data processing, graph traversal, geospatial queries etc [4, p. 653]. On the other hand, it comes with consistency challenges, as you now need to make sure that writes propagate to read replicas, as well as handling cases when it takes significantly long time. Change Data Capture (CDC), domain events and built-in engine-level replication are common ways of dealing with this problem.

Another form of dual-DB approach is hot/cold storage split. It implies having separate databases for recent, frequently accessed data and older, archival data. Such separation allows using fast and expensive storage only when it's worth it, while the rest of the data is delegated to slower, but cheaper storage.

One of the most effective solutions for high read loads on databases is caching, and that is what primary + cache DB form of dual-DB is about. This form suggests having two databases: primary database, which acts as a source of truth, and cache layer that uses technologies with low read latency (e.g. Redis). The main challenge here is cache invalidation – basically ensuring that the data stored in the cache layer is up-to-date with the source of truth when writes are happening.

Given the fact of different databases being optimized for different kinds of data, the next form of dual-DB approach – polyglot persistence – comes naturally. In this form, different databases are used for different types of data or concerns within the same system. For example, a relational database may be used for structured business data while a graph database is used for relationships at the same time.

There is also a natural architectural solution of having two separate databases, or schemas within the same instance, serving different parts of the application, different tenants or domain contexts. While this solution introduces similar benefits, e.g. independent scaling, it is not generally considered as a dual database approach in its traditional understanding, even though there are multiple databases with some separation logic in fact. The core reason for this is that the dual-DB approach specifically implies that the same data (or overlapping data) is stored in two places for a technical reason – performance, read/write separation, caching, etc. However, a provided setup with multiple databases/schemas can still be a valid architectural choice and correspond to such patterns as shared-key distributed data model or federated data

model. Those are widely applied in complex systems with interconnected business logic areas, or can be a clear implementation of DDD by each database serving its domain or microservice.

In any form, the dual database approach introduces eventual consistency problems, sensitivity to sync pipeline failures and schema evolution problems when drift occurs between different databases. Other issues and their typical solutions were addressed previously, but these are general ones.

Eventual consistency, meaning that read replicas or secondary databases will always be somewhat behind the source of truth, is often the most significant problem [1, p. 31]. It is typically mitigated by the read-your-own-writes technique where the write result is returned to a consumer instead of read-replica's record, causal tokens (e.g. timestamps) that are used by the client in requests to make sure only up-to-date data will be served, or even optimistic UI.

Common ways to deal with sync pipeline failures include lag monitoring, dead letter queues (keeping failed events for inspection or retries), idempotent replication events design or periodic reconciliation jobs – background processes that periodically compare samples of records between two databases and flag or repair divergences.

In conclusion, despite introducing operational costs, complexity and challenges or even design restrictions, the dual-DB approach and the benefits it brings can be a valuable part of a good software architecture. The key to its successful application is identifying access patterns, the read/write ratio, bottlenecks and the types of data in a system, extracting the most impactful responsibilities or performance-critical areas and distributing them across multiple databases, each well-suited for the role it serves. It helps systems operate reliably, allows teams to maintain them easily, and ensures software quality corresponds to business needs and field requirements.

## REFERENCES


1. Lloyd W. Don't settle for eventual consistency / W. Lloyd, M. J. Freedman, M. Kaminsky, D. G. Andersen // Queue. 2014. Vol. 12. P. 30–45.
2. Prasad S. NextGen data persistence pattern in healthcare: Polyglot persistence / S. Prasad, M. S. N. Sha // 2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT). 2013. P. 1–8.
3. Roh S. An efficient microservices architecture for MLOps / S. Roh, K.-M. Jeong, H.-Y. Cho, E.-N. Huh // 2023 Fourteenth International Conference on Ubiquitous and Future Networks (ICUFN). 2023. P. 652–654.