A survey on auto-scaling cloud-native databases

Venkata Saikiranpatnaik Balivada Vismay Srivastava

July 9, 2023

Abstract

Today, we have numerous *cloud-native databases* that have many unique features and differ in both high-level architecture as well as their design tradeoffs. While the market provides an amazing array of choices for a cloud-native database, new users often do not have the required knowledge or expertise to decide on the ”right” database. Since no database is ”one size fits all”, we believe there is value in understanding the design considerations of each of these databases and their approach to solving major challenges that arise from designing a *distributed transactional database*. Moreover, a comprehensive study also helps us identify the similarities and differences between these commercial databases. For this reason, we survey *seven* cloud-native auto-scaling databases. Finally, our aim not only extends to finding out how the databases tackle the common challenges faced by cloud databases but also to identify some *limitations* these databases face from either the architectural or the implementation standpoint and consequently suggest ways to overcome the same.

1 Introduction

In recent years, there is a steep increase in the popularity of cloud-based storage. The corporate world is increasingly migrating their data from on-premises to the cloud. Moreover, emerging startups are leveraging cloud services to get up and running. There are numerous reasons for this trend. First, Cloud DBs are **easier to set up** since they handle the maintenance of systems, allowing you to invest your time, money, and resources into fulfilling your core business strategies. Second, Cloud DBs are often **scalable**, catering to contemporary big data requirements. Third, users can request and release resources **elastically** in real-time to accommodate changes in their needs, thus creating an illusion of infinite resources. Fourth, the cloud offers **high availability**, crucial given the scale of a typical cloud system. Fifth, the pay-as-you-go model and the economies of scale help make cloud DBs **cost-effective**. To elaborate, users no longer need to purchase expensive hardware in advance, eliminating the need to guess future hardware requirements. Also, the economy of scale amortizes various costs necessary to maintain such systems across users. Finally, cloud DBs provide multiple other features and fully managed services that ensure salient properties such as **durability, integrity, and security**.

Utilizing the full power and flexibility of cloud DBs requires a deep understanding of the current cloud DBs. Henceforth, in this report, we,

- Discuss various challenges that need to be overcome to design a cloud-native distributed transactional database in section 2
- Discuss an overview of various commercial database systems in section 3
- Compare to provide similarities and differences between surveyed DBs in section 4
- Mention limitations of current systems and suggest ways to tackle the said problems in section 5

2 Challenges

In this section, we discuss major challenges that arise when scaling database systems.

2.1 Scaling and Fault-Tolerance

In the late 1990s, during the age of the internet, the amount of data stored on corporate systems have shot up significantly. The storage needs have only increased since then and companies such as Google began using 100s and 1000s of server machines to store their data. This is achieved by *partitioning* the data across these machines. To minimize costs, companies such as Google used *commodity hardware* to run their servers. This leads to a scenario where server failure is the norm and not the exception. In response, Google introduced the file system GFS[10] and the compute engine MapReduce[8]. In summary, GFS handles node failures through *replication*. Whenever a node fails, GFS

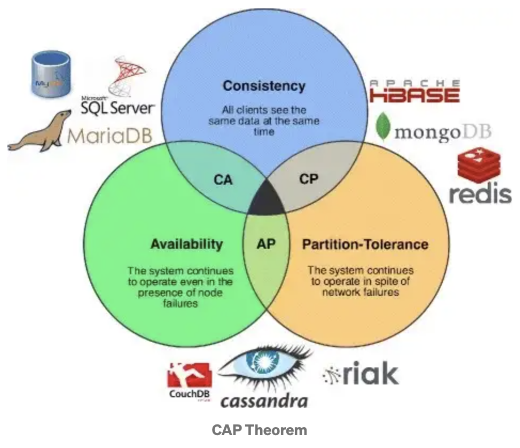

Consistency
All clients see the same data at the same time
CA
CP
AP
Availability
The system continues to operate even in the presence of node failures
Partition-Tolerance
The system continues to operate in spite of network failures
SQL Server
MariaDB
HBASE
mongoDB
redis
CouchDB
cassandra
riak
CAP Theorem

Figure 1: The CAP Theorem

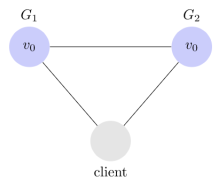


Figure 2: Replicated state machine

recovers failed nodes by through the use of ”backup” nodes whose state is the same as the ”primary” node.

2.2 The Consistency Problem

Although GFS is wildly successful for storing append-only data such as logs, users quickly realized the need for scaling structured data storage systems. While Google conceived Big-Table[6], a Key-Value store in response, many users were flustered with the weak consistency models of these systems. The weak consistency of these models is not a mere coincidence. In fact, Brewer’s conjecture on the CAP Theorem[11] suggests that scaling systems with strong consistency is a difficult problem to solve, see figure 1. To elaborate, say an object v_0 is replicated across server machines 1 and 2, see figure 2. While the system can recover if machine 1 fails, it may have missed some updates from the machine and thus lose strong consistency. Alternatively, machine 2 may stall until machine 1 is back up to respond to clients. This approach leads to extremely high latencies, thus leading to poor availability and frustrated users. In summary, the CAP theorem states that database systems can never be fault-tolerant, always available, and strictly consistent, all at the same time.

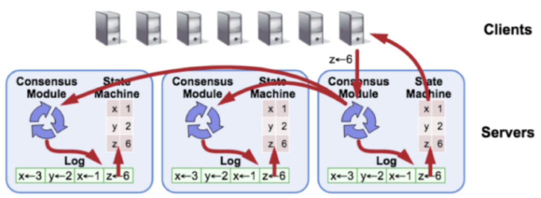


Figure 3: Raft Consensus Protocol

2.3 Consensus Protocols

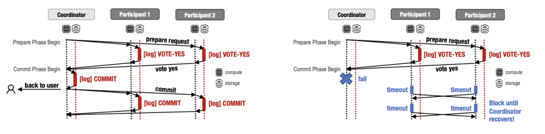
Does that mean it is impossible to have database systems with transactional consistency? The reality is far from it. Distributed systems researchers have designed state-replicated machines using what are called distributed consensus protocols. These protocols ensure strong consistency between replicas while also providing high availability as long as the node failures do not cross a certain threshold. One such popular consensus protocol is Raft[15], see figure 3. Raft is used in various distributed databases to provide strong consistency while also enabling storage systems to scale. Raft becomes unavailable either when there are too many node failures or when the nodes fail to elect a leader, which are both rare occurrences in practice.

2.3.1 Raft Performance

As implied earlier, distributed transactional systems often use consensus protocols such as raft to keep their replicas consistent. However, the protocol does not come cheap. Raft involves significant communication between the leader and follower nodes (similar to primary and backup nodes of GFS) that the latency of a confirmed write is quite high and utilizes significant network bandwidth. Hence, executing operations sequentially is too slow for a database system. Therefore, commercial systems employ various optimizations to run the consensus protocol asynchronously. Such approaches are possible because a database transaction can always rollback in the face of undesirable scenarios. That said, the commit operation is now even more expensive, whose overhead the commercial systems try to minimize.

2.4 Distributed Conflict Resolution

In the previous subsection, we briefly discussed how consensus protocols may be used to provide strong consistency despite providing high availability in practice. In this section, we briefly discuss what happens on node failures. At a very high-level, in raft, a group of nodes elect a node to be their leader



(a) 2PC with no failure. (b) 2PC with coordinator failure (cooperative termination protocol).

Figure 4: 2-phase-commit protocol

each round. The leader is responsible for sending updates to its followers. When a leader fails, the nodes re-elect a new leader and the state replication process continues. This transfer of ”power” poses a subtle problem for transactional systems. Say a transaction T_0 holds the lock for object v_0, and another transaction T_1 is waiting for the lock on v_0 to be released. If the server with the leader fails, then the lock needs to be with transaction T_0 and not T_1. Henceforth, there is a need for distributed conflict resolution, often done using replicated locks or distributed lock services such as Chubby. Alternatively, systems can use global timestamps to resolve conflicts. This approach, however, limits parallelism and increases the transaction abort rate. Moreover, often, distributed DBs use 2-phase-locking for their concurrency control because using OCC approaches in distributed settings is prone to high abort rates. Finally, apart from providing strong consistency such as serializability, commercial databases also provide transactions isolated via snapshots. In many scenarios, users are content with only reading a previous snapshot of data and would rather avoid waiting for or blocking other transactions. Typically, this is achieved using a mechanism popularly known as Multi-version Concurrency Control (MVCC[4]).

2.4.1 Commit Protocols

Most databases use the 2-phase-commit protocol (see 4) or its variations to commit distributed transactions. However, the 2-phase-commit protocol by itself has two glaring limitations[12]. First, 2PC suffers from high latency. The protocol involves two network round-trips as well as synchronous writes to the log system. Second, coordinator failure is too expensive. Future transactions having conflicts with the committing transaction have to block for the coordinator to be back up. Therefore, systems try to mitigate these issues through various optimizations such as Parallel Commits[18] in CockroachDB.

2.5 Elasticity

Databases serve varying workloads due to several reasons. First, is the seasonal variations in the workloads. Instances such as Black Friday sales can heavily increase network traffic on e-commerce websites such as Amazon. Second, the workload may also have unpredictable spikes such as a sudden increase in demand for toilet paper during the pandemic, see figure 5. This variation is common and motivates one of the major advantages of a cloud system – *elasticity*. Databases often provide live reconfiguration mechanism to scale up or down the number of nodes in the system in order for users to adapt to ever-changing workloads. This reconfiguration triggers a live migration system that moves data shards across machines. It is important to note here that the data migration should happen concurrently with transactions to avoid blocking transactions. However, manual reconfiguration is too slow since humans are not quick to react. Therefore, databases provide mechanisms to scale automatically. One such abstraction is Google’s Auto-scaling[1] architecture, see figure 6. In a typical auto-scaling system, the user provides some policies such as load limits on server machines. The *Monitoring* module in turn detects policy violations so that the *Scaler* module can adjust the number of nodes.

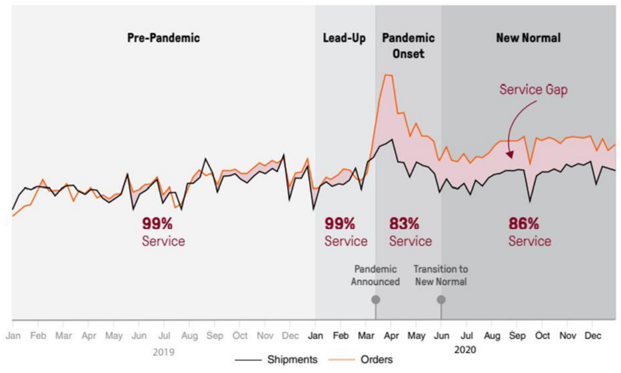


Figure 5: Graph demonstrating low-availability of toilet paper during the pandemic

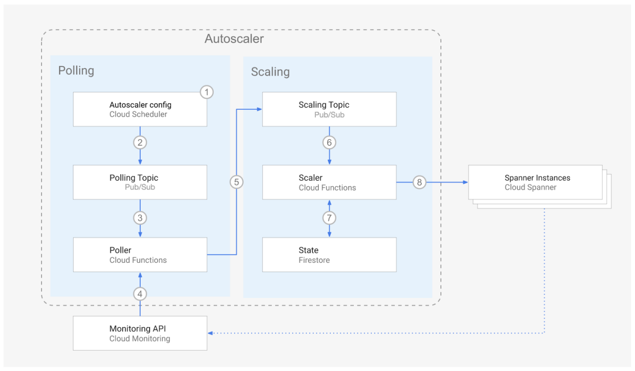
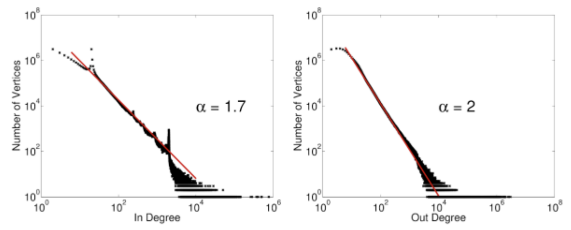


Figure 6: Google Autoscaler architecture



(a) Twitter In-Degree (b) Twitter Out-Degree

Figure 1: The in and out degree distributions of the Twitter follower network plotted in log-log scale.

Figure 7: Twitter Follower Network

2.6 Load Balancing

The data migration system introduced in the previous sub-section has other uses. In practice, workloads often skew towards accessing some data more than others. For example, see figure 7. In the figure, observe that fewer and fewer twitter users have an exponentially higher number of followers. This phenomenon occurs naturally in real-world graphs and is termed the *Power Law*. Such skew is bad news for distributed databases since some server nodes are heavily overloaded in the absence of requisite load balancing creating *Hot Spots*. Hot Spots can easily bring a system down and hence care must be taken to allocate proportionately more resources to highly contested data. One popular approach splits hotly contested data shards and merges cold data partitions. Other systems such as CockroachDB use a two-level partitioning scheme to keep the size of hot data shards small.

2.7 Abstractions

Single-node databases by themselves are notorious for being extremely complex pieces of software. Solving the above-mentioned challenges on top of that quickly becomes intractable without the proper use of abstractions. In this sub-section, we describe popular components of a typical cloud DB architecture.

Almost every DB has a control plane and a data plane. The control plane is responsible for meta-level operations such as placement of replicas, parsing and optimization of user queries, handling database configuration, handling schema changes, load balancing, coordination between server nodes, etc. The data plane stores the actual data and/or executes user queries. At the lowest level is the storage layer which stores logs and/or data pages. The storage layer is often implemented as a Key-Value store in distributed settings. Not only can we scale Key-Value stores easily but we can also translate SQL queries to transactions on the key-value store with reasonable effort.

2.7.1 Storage Disaggregation

Another more recent but useful abstraction is the separation of the storage layer from the compute layer. With storage disaggregation architectures, systems like aurora are able to drastically simplify the compute layer. However, storage disaggregation-based architectures suffer from having the network as the bottleneck. This arises when data and logs move between the computational layer and the storage layer. This creates a lot of network overhead in these systems making I/O slower[19].

2.7.2 SQL Compatibility

Structured Query Language[2], abbreviated as SQL, is an age-old query language system for databases, known to almost every programmer on earth. Leveraging such a popular query interface is crucial to the success of emerging cloud DB systems. For this reason, many cloud vendors provide a SQL-like interface, often going the extra mile by making their systems either MySQL and/or PostgreSQL compatible for easy migration.

2.8 HTAP

There is a recent emergence of Hybrid Transactional Analytical Processing systems (HTAP for short). This trend is motivated by the need for post-processing for business intelligence. Business intelligence runs numerous ad-hoc analytical queries to gain insight into past transactions. These insights are then used to make business decisions for the company. Before HTAP, analytical databases were constructed from production OLTP databases by running complex ETL pipelines to convert row-store transactional data to column-store analytical data format. Data is often moved only at the end of the day and hence users were not able to run ad-hoc analytical queries on fresh data. HTAP attempts to solve this problem.

3 DBs Surveyed

We have surveyed seven cloud-native databases for our project. These include AWS Aurora, Azure Hyperscale/Socrates, TiDB, FoundationDB, Google Spanner, CockroachDB, and YugabyteDB. We will now study the architecture in detail of the above-mentioned databases to understand how each of them is able to solve different problems faced commonly in such systems.

Feature	**Spanner**	**CRDB**	**Yugabyte**	**Aurora**	**Socrates**	**FDB**	**TiDB**
Architecture	Shared nothing	Shared nothing	Shared nothing	Disagg. storage	Shared storage	Disagg. Storage	Shared nothing
Consistency	Externally consistent	Serializable	Serializable	Serializable	Serializable	Serializable	Serializable
Snapshot Isolation	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Replication Protocol	Paxos	Raft	Raft	BinLog replication, Logical replication	Using page servers	Allows Downtime	Raft-based
Concurrency Protocol	2PL with Chubby	2PL with replicated locks	Explicit locking with both optimistic & pessimistic locks.	MVCC	MVCC with 2PL	OCC	2PL
Storage Layer	Colossus	RocksDB	DocDB	AWS S3	Azure Storage (XSTORE)	Storage Servers	RocksDB
Source License	N/A	BSL	Apache 2.0 and Polyform Free Trial 1.0.0	N/A	N/A	Apache 2.0	Apache 2.0
SQL Compatibility	Yes	Yes	Yes	Yes	Yes	No	Yes
DB Compatibility	PostgreSQL	PostgreSQL*	PostgreSQL	PostgreSQL & MySQL	PostgreSQL	N/A	MySQL and PostgreSQL*

Table 1: Feature comparison of popular cloud DBs

3.1 AWS Aurora

AWS Aurora is a distributed cloud-native database and is completely managed by Amazon RDS. It is fully compatible with MySQL and PostgreSQL. It is designed for handling OLTP workloads. Aurora coupled along with other AWS services like RDS, VPC, Cloud Watch, Events and other services provides metrics, events, alarms, easy start/stop, cloning and more such operations.

Aurora[19] has three Availability Zones(AZs) having two nodes each. It uses a quorum model with 6 votes. With this model, inspite of one AZ(or two nodes) failure, write availability can be maintained. Similarly, inspite of one AZ and one node failure, read availability can be maintained. Due to this, Aurora is able to provide high availability. Aurora also maintains six or more replicas of the data, which helps maintains fault tolerance and regular backup. Aurora replicates the data enabling fast local reads with low latency in each region, and provides disaster recovery from region–wide outages[16].

In Aurora, no pages are written to storage. Only the redo-logs are written to the storage therefore reducing the network bottleneck as discussed in section 2. This redo log applicator is decoupled from the database and works continuously in the storage. During recovery when the databases starts-up, it can perform volume recovery with the storage service, leading to a quick crash recovery($<$ 10 seconds). Also, instance lifetime does not correlate well with the storage, as the instance can be scaled up or down, or start or shut down as per need, hence Aurora has decoupled compute from storage. This reason also contributes to the fast recovery process as mentioned before.

Another benefit of AWS Aurora is the scalability and cost-effectiveness. More workers/nodes can added/removed as per increase/decrease in demand. This requires no setup and hardly takes few minutes to go live. Also, the pay-as-you-go model helps you to save money and pay only for the resources used. High performance of Aurora is another added feature that makes Aurora so popular. (five times the throughput of standard MySQL and three times the throughput of standard PostgreSQL).

3.2 Azure Hyperscale/Socrates

Socrates[3] is a newly implemented SQL Server that is available in Microsoft Azure SQL as Hyperscale. It is an improvement over Azure SQL.

The Socrates design adopts the separation of compute and log from storage which helps separate the durability(from logs) and high availability(from storage). Also, a design consideration is local fast storage and cheap durable storage. The pages which are generally requested by the compute nodes are stored in the third tier in fast storage, however, the fourth tier is composed of cheaper storage options. Socrates also provides pushdown features which allows compute functions to be passed over to the storage tier.

The socrates is a 4 tier architecture which comprises of Compute tier, XLOG service, storage tier, and XStore. The compute tiers consists of the primary and secondary nodes. These primary nodes handles all read/write transactions, while the secondary nodes acts as a failover nodes. The XLOG Service implements the separation of log from storage. This helps socrates achieve low latencies during the commit and high scalability in storage. Moving on, we have the third tier, which is the storage tier and it is implemented using page servers. Finally, the fourth tier is the XStore, also known as Azure Storage Service. This service is completely managed by Azure. This is the cheaper storage option as compared to the third tier.

3.3 Google Spanner

Google Spanner[7] is one of the early databases conceived in response to internal complaints about the weak consistency model provided by then scalable storage systems such as BigTable. Spanner is unique in that it provides externally consistent transactions. External consistency is similar to linearizability but for transactions. To sum up, two externally consistent transactions execute one after another according to the wall-clock time. On the other hand, serializability allows reordering transactions. So, Spanner provides a stricter consistency model. This is achieved using globally assigned timestamps in combination with the in-house true-time API provided by the cluster management software. The true-time API allows servers to determine the approximate time at which a message is sent within a bounded uncertainty (difficult problem in a globally distributed setting).

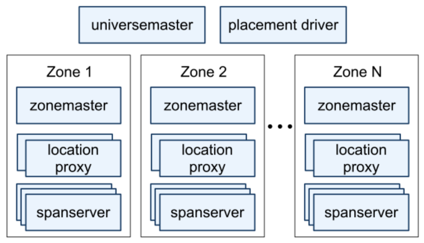


Fig. 1. Spanner server organization.

Figure 8: Spanner Architecture

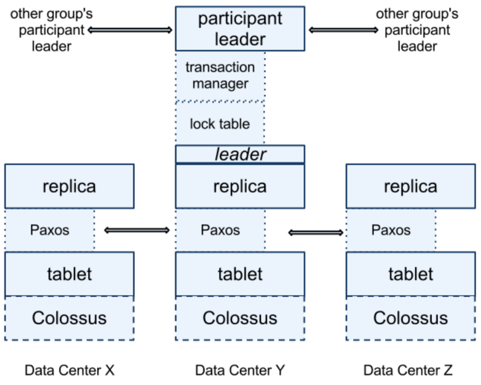
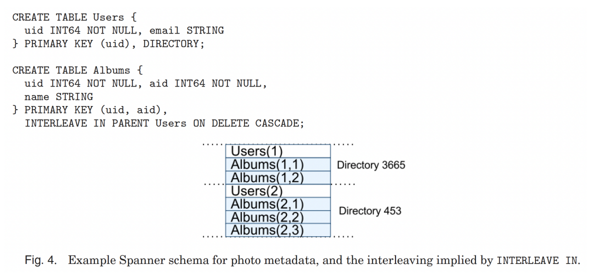


Fig. 2. Spanserver software stack.

Figure 9: Paxos Protocol

A high-level architecture of Spanner can be seen at 8. The data is partitioned and stored within span-servers. There are 100s to 1000s of span-servers in each zone and the zone master manages the span server. The location proxies store information about replica placement to route requests and data. The placement driver is responsible for the placement of individual replicas. Replica placement is a combination of user-provided policies and automatic placement for better fault-tolerance and higher availability. The universe master monitors various system metrics such as health for displaying status to users.

The data in Spanner is range partitioned across span-servers for more efficient range-queries. The data shards are also replicated using the Paxos9 state-machine protocol for fault-tolerance as discussed in 2. Spanner uses Colossus as a storage layer and hence inherits some design choices such as MVCC for efficient snapshot reads and chubby lock manager for distributed conflict resolution. Finally, Spanner also allows user-specified data locality through the *INTERLEAVE IN* operator, see 10.



```
CREATE TABLE Users {
  uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
  uid INT64 NOT NULL, aid INT64 NOT NULL,
  name STRING
} PRIMARY KEY (uid, aid),
  INTERLEAVE IN PARENT Users ON DELETE CASCADE;
```

Fig. 4. Example Spanner schema for photo metadata, and the interleaving implied by `INTERLEAVE IN`.

Figure 10: Data locality using the interleave operator

3.4 YugabyteDB

YugabyteDB[5] is a hybrid and multi-cloud, geo-distributed (distributed in different regions to offer low latency), cloud-native relation database that offers compatibility with PostgreSQL.

YugabyteDB consists of two logical layers, Yugabyte Query Layer(YQL) and DocDB distributed document store. Applications interact directly with YQL layer using the client drivers. YQL comprises of the YSQL API, YCQL API and the query engine. YSQL is a distributed SQL API that is built by re-using the PostgreSQL language layer code. YCQL is a semi-relational language that is based on Cassandra Query Language. DocDB document store is the place where the sharding, replication, load balancing, and transaction management take place. Also, it holds a custom RocksDB storage engine.

YugabyteDB is consistent and partition tolerant, yet it provides high availability. It achieves this by having an active replica that is ready to take over as a new leader in a matter of seconds after the failure of the current leader and serve requests. This replication takes place is DocDB using the Raft consensus protocol. Yugabyte supports two ways of sharding data - hash and range sharding. Data sharding helps in scalability and geo-distribution by horizontally partitioning data.[14]

3.5 CockroachDB

CockroachDB[17], inspired by Google Spanner, created a distributed transactional system that runs on commodity hardware. One of the main observation of CockroachDB is that Google Spanner is slow for providing strong consistency guarantees. The database system introduced new mechanisms to exploit parallelism even further whilst providing serializable isolation guarantees. Like Spanner, CockroachDB provides fault-tolerance and high availability through replication, employs a shared-nothing architecture, provides geo-replication, and support snapshot reads through MVCC. Unlike Spanner, however, CockroachDB uses raft for replication and replicated locks, write intents and transaction records for conflict resolution.

CockroachDB uses transaction pipelining for fast writes and parallel commits for a more efficient commit protocol. Transaction pipelining allows concurrent writes respecting program dependencies. On the other hand, parallel commits makes transaction status durable through replication but avoids two rounds of consensus in the fast path using a separate STAGING phase.

3.6 FoundationDB

FoundationDB[20] is one of the early systems to provide transactional key-value stores. It is designed to be extremely modular and attempts to minimize hard state in many of its modules as a design philosophy. FoundationDB also provides a simulation framework which makes debugging and thus adding new features more developer-friendly. FoundationDB boasts an unique serialization mechanism that avoids locking entirely during conflict resolution. The modular structure also provides stronger fault-tolerance.

At a high-level, the FoundationDB architecture can be divided into two major parts, a Control Plane and a Data Plane, see 11. The Control Plane hosts a Coordinator that stores metadata about configuration and deployment. While the cluster manager monitors the health of the servers via heartbeats. The data plane on the other hand can be divided into a transaction layer, a log system and a storage layer. The log system and the storage layer are exactly what they sound like, one storing logs and other data. Transaction is interesting because it has a sequencer that assigns sequence timestamps to transactions that decides the commit order of transactions. The conflict resolution happens in the resolvers where FoundationDB allows false positive conflicts to greatly simplify concurrency control. The fault-tolerance on the other hand, is trivial for most modules since they are stateless. For stateful modules, FoundationDB introduces a minor downtime recover the state. This downtime doesn't affect snapshot reads.

3.7 TiDB

TiDB[13] is the first major database system that supported both transactional and analytical workloads. TiDB's motivation comes from the fact that traditional approaches to analytical processing are 'slow'. Traditionally, users inject transactional data into analytical engines as part of the 'end-of-the-processing' ETL pipelines. This approach severely prohibits business intelligence to gain fresh real-time insights as described in 2. So, TiDB provides both transactional

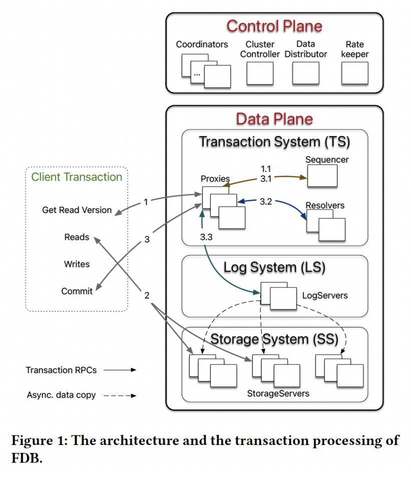


Figure 11: Foundation DB architecture

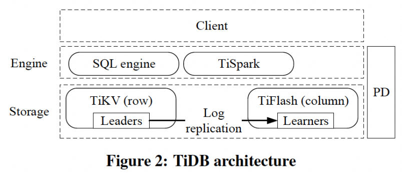


Figure 12: TiDB Architecture

and analytical modules in one database system where analytical queries fetch 'fresh' data.

See 12 for a high-level overview of TiDB architecture. TiDB's transactional key-value store, TiKV, is heavily inspired from Google Spanner. TiKV provides highly available, consistent, transactional key-value store whose data is range partitioned. For the purposes of load balancing, TiKV splits hot partitions and merges cold partitions.

One major contribution of TiDB is its replication protocol. TiKDBextends the vanilla raft protocol to include replica types called learners. Learners are separate from followers. They are part of TiFlash, their analytical engine and do not participate in the leader election process. Instead, they convert the row-format data from the leader into a column-based format providing better efficiency for analytical queries.

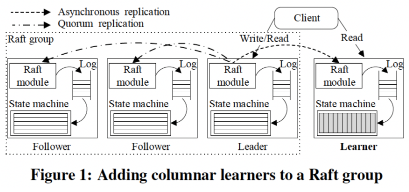


Figure 13: TiDB Learner Architecture

4 DB Comparison

We layout the major feature parity in table 1. In summary, more recent databases have started adopting *Storage Disaggregation* architectures to cleanly separate the storage layer from the compute layer. All the studied databases are capable of at least the *Serializable* isolation property.

CockroachDB and YugaByte are geo-distributed. Aurora may also belong in this category, because it has data replicas available in different geographic locations. Due to this, they are able to provide low latency access to the data. Moreover, we understand DBs that are generally part of the Cloud platform generally have their own storage system, like S3 for Aurora, XStore for Hyperscale. However, this isn't the case for the others as they prefer to build over an existing storage system like RocksDB. Also, the databases like Aurora and Hyperscale also have migration services which make it easier to migrate data easily.

For higher availability and faster recovery, data replication is something common witnessed across almost the cloud databases with Raft consensus protocol being really popular. These databases store multiple data copies(some in different geo-locations as well).

All these databases except FoundationDB provide SQL compatibility which allows the users to migrate easily without changing much of their architecture and the practices followed. Aurora and Socrates separates compute/log from storage in order to reduce network overhead. Aurora sends only REDO logs to the storage whereas Socrates uses another layer, XLOG to store the logs. The YugabyteDB synchronous replication architecture is inspired by Google Spanner.

5 Limitations and Open Issues

In this section, we lay out the various limitations we identified across the databases we surveyed.

5.1 A modular HTAP system

Currently, TiDB includes both OLTP and OLAP modules inside a single system. While this approach simplifies deployment, users often want to run their own database for OLTP given the latency-sensitive and production-sensitive nature of transactional workloads. However, the data freshness provided by TiDB is very valuable in the market. Now the question is, can we design a system that provides plug-ins for popular OLTP and OLAP databases? Such a system enables modular plugins for desirable OLTP and OLAP systems, preferably adding a minimal run-time overhead to both these systems. An ideal solution to the problem will involve providing a clean minimal abstraction to both OLTP and OLAP systems for communication. Moreover, such a system should design an intermediate data representation to convert data from numerous OLTP systems and load it to various other OLAP systems. We must avoid any format conversions on the OLTP side to minimize overhead.

5.2 OCC in Distributed DBs

Optimistic Concurrency Control is rarely used in distributed database systems. One reason for that is the high conflict rate of its constituent transactions. However, OLTP workloads often require predictable latencies and high abort rates drastically degrades the 99 percentile latency. Here, abort rates are highly sensitive to query scheduling. We pose the problem of whether using learning methods can improve query scheduling and thus reduce the query conflict rate enough to make them viable alternatives to 2PL. For example, the query to update the phone number of 'Marjorie' intuitively conflicts with a query to count the number of women in the class. Can learning methods detect such conflicts so that the SQL/Transactional layer can serialize the queries by itself?

5.3 Auto-scaling Abstraction

Currently, most auto scaling mechanisms are reactive in the sense that auto scalers reconfigure only after the load becomes 'too heavy'. As a consequence, not only is the load high, but the system is working towards re-partitioning the data thus imposing heavy resource load during the migration phase. Instead, it would be amazing if we could predict increases and decreases in query rate and what better tool to use than predictive learning algorithms. Although there is no 'one size fits all' solution for all users, the auto scaler can expose multiple learning modules and provide a relatively simple abstraction to configure and compose these modules for prediction.

6 Conclusion

In conclusion, a thorough study of existing commercial cloud database systems enabled us to

- Present various challenges in building such systems
- Discuss and compare various cloud database systems
- Identify limitations and pose potential research problems in the area

References

[1] Autoscaling cloud spanner. `https://cloud.google.com/architecture/autoscaling-cloud-spanner`.

[2] Sql. `https://en.wikipedia.org/wiki/SQL`.

[3] P. Antonopoulos, A. Budovski, C. Diaconu, A. Hernandez Saenz, J. Hu, H. Kodavalla, D. Kossmann, S. Lingam, U. F. Minhas, N. Prakash, V. Purohit, H. Qu, C. S. Ravella, K. Reisteter, S. Shrotri, D. Tang, and V. Wakade. Socrates: The new sql server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1743–1756, New York, NY, USA, 2019. Association for Computing Machinery.

[4] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, dec 1983.

[5] A. Budholia. Newsql monitoring system. *SJSU Scholar Works*, 2021.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 205–218, 2006.

[7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), aug 2013.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, jan 2008.

[9] B. Foster. Raft protocol: What is the raft consensus algorithm? `https://www.yugabyte.com/tech/raft-consensus-algorithm/`, May 2022.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, oct 2003.

[11] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, jun 2002.

[12] Z. Guo, X. Zeng, K. Wu, W.-C. Hwang, Z. Ren, X. Yu, M. Balakrishnan, and P. A. Bernstein. Cornus: Atomic commit for a cloud dbms with storage disaggregation. *Proc. VLDB Endow.*, 16(2):379–392, nov 2022.

[13] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang, W. Wei, C. Liu, J. Zhang, J. Li, X. Wu, L. Song, R. Sun, S. Yu, L. Zhao, N. Cameron, L. Pei, and X. Tang. Tidb: A raft-based htap database. *Proc. VLDB Endow.*, 13(12):3072–3084, aug 2020.

[14] S. Mukherjee. The battle between nosql databases and rdbms. *Available at SSRN 3393986*, 2019.

[15] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA, 2014. USENIX Association.

[16] J. Petrovska and J. Ajdari. Amazon's role in the field of cloud relational and nosql databases: A comparison between amazon aurora and dynamodb. *ISCBE 2019*, page 214, 2019.

[17] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1493–1509, New York, NY, USA, 2020. Association for Computing Machinery.

[18] N. VanBenschoten. Parallel commits: An atomic commit protocol for globally distributed transactions. `https://www.cockroachlabs.com/blog/parallel-commits/`, November 2019.

[19] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1041–1052, New York, NY, USA, 2017. Association for Computing Machinery.

[20] J. Zhou, M. Xu, A. Shraer, B. Namasivayam, A. Miller, E. Tschannen, S. Atherton, A. J. Beamon, R. Sears, J. Leach, D. Rosenthal, X. Dong, W. Wilson, B. Collins, D. Scherer, A. Grieser, Y. Liu, A. Moore, B. Muppana, X. Su, and V. Yadav. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2653–2666, New York, NY, USA, 2021. Association for Computing Machinery.