A Survey on Logging and Resilience System in Cloud-native Database

Huiyu Bao
UW-Madison
huiyu.bao@wisc.edu

Wendi Li
UW-Madison
wendi.li@wisc.edu

Zhikang Hao
UW-Madison
hao45@wisc.edu

ABSTRACT

Logging is a crucial part of the database system. When failure happens, logging can fast recover the database system to normal status and make the database system more resilient. Almost all of the database has a logging system. However, the logging technology used in the conventional database may not that suitable for a cloud-native database system. So many logging management systems working for cloud-native databases are proposed. In this paper, we will review various approaches of logging designed specifically for this new sort of database. We will give them a thorough analysis and do evaluations to compare them, to figure out which method has the best performance in different aspects.

1 INTRODUCTION

With the increasing demand for elasticity and use-as-needed concepts, along with the development of the cloud computing infrastructure, cloud-native databases are becoming more and more popular these days. By using the cloud service vendors, users do not have to deploy and tune the fundamental facilities by themselves. Typically, a cloud-native database is built, deployed on the cloud platforms. The vendors will handle all of the hardware and infrastructure, and provide maintenance. It will allow individual users or companies to store data, manipulate data, and do analysis work totally online. There are many existing cloud-native databases provided by big companies, such as the Google Cloud, Microsoft Azure, Amazon Web Service, Tencent Cloud, and Alibaba Cloud, etc. Those can be called database-as-a-service (DBaaS). Users can simply choose the CPU, memory, disks, and other components as they wish. Then, the vendors will do the rest and provide service to users. By using cloud-native databases, some challenges that cannot be directly addressed by traditional databases can be solved by using cloud applications. Nowadays, most start-up companies choose to move their data and workloads on the cloud so that they only need to care about the workloads themselves and entrust their vendors to handle other low-level tasks. Such a pay-as-you-go paradigm can avoid paying much attention to under-utilized machines. Besides that, the cloud service is expected to be highly elastic. It means that when the workload grows or shrinks, the service can adapt the scale well so that the users can take advantage of the pay-as-you-go paradigm.

The advantage of cloud service also comes from the nature that the marketing activities are often unstable, which is mainly caused by some widespread activities. For example, on Double Eleven Day or Black Friday, people tend to buy things more frequently, which results in a sudden expansion of the number of transactions. In traditional databases, users need to add extra resources or use load balancing to deal with peak time. However, if it is on a cloud service, users just need to expand their deployments by sending requests to the vendors and the vendors will do it for them. Moreover, with the development of the hardware, it would be very troublesome for users to upgrade their infrastructure periodically, depreciation of old components may cause financial loss. But with the cloud service, vendors can quickly evolve the cloud-native database systems to make sure they can maintain competitiveness.

However, there are also many new challenges that show up when deploying and running cloud-native databases. As all components are tightly coupled under the monolithic machine architecture, so it is hard to make sure the elasticity and the expansibility under the old architecture. It would be a bin-packing problem if DBaaS wants to assign database instances to machines. It is also difficult for arranging different resources on a single machine while keeping a high utilization rate. Another challenge is that it is hard to satisfy the frequently changed resource requirements by flexibly adjusting individual resources. It is common that users will have different resource requirements at different time periods (such as the higher requirements for sellers on bargain days). How to meet the users' demands is a critical part similar to the load balancing process. Moreover, there is a concept of fate sharing, which means that in a tightly coupled system with many resources, a failure of one resource will result in the failure of others. Recovering all the failed resources causes a longer system recovery time.

To deal with the challenges above, DBaaS should adopt a new architecture. One available solution is the separation of computing and storage [7]. By splitting the computational part and the storage part, resource utilization can be improved. But, there are still some challenges that such an architecture cannot address, how some recent cloud-native databases solve these challenges would be a focus in our paper.

In this paper, we first compare the state-of-the-art cloud-native databases with traditional databases, analyzing their advantages of them, as well as how they deal with the new challenges that show up in the cloud service. Moreover, we carry out experiments on these cloud-native databases with general metrics to compare unique advantages or challenges they have.

2 BACKGROUND AND MOTIVATION

2.1 background

In a database, logging is a fundamental component. Whenever we do some manipulation on the data, we need to record some logs to make sure we can track these changes later. Then we could know anything done to a database. All of the operations will be recorded in the memory or disk. Then when we roll back a transaction, the log system will help us undo corresponding operations. Also when failure happens, the log system will help us recover the database to a consistent status.

The database logging often contains much information, such as the beginning and the end time of transactions, each data change done by transactions, and the commit or abort of transactions. Lots of databases will also use checkpoints to help the recovery process. Database writes all of the data pages safely to the disk at each checkpoint. During recovery, the database restores all of the data to a prior status. When the database is restarted, it will start a recovery process. The database will use log records to check that what operations need to be redone and what operations need to be undone. The database also needs to check what transactions need to abort and roll back their operations. Because of the existence of checkpoint, which means the checkpoint has already guaranteed that all of the changes before that have persisted to the disk, we need to do nothing for the records before the checkpoint. The restarting process will be executed more quickly.

Logging is very important for the database system. It provides fault tolerance in case of some machines are accidentally down. We can consider that all of the log records make up all of the data in the database. In traditional databases, there are lots of popular approaches to implement logging systems.

Some early database systems, such as system R [9], use shadow paging strategy to avoid in-place updates. When a transaction writes data, the system will make a copy for the origin data page, which is the shadow page, and write on that shadow page. After the transaction commits, the system will update the origin data page.

A write-ahead-logging (WAL) protocol was defined in ARIES [12], which is widely used as a centralized logging system. ARIES uses no-force, steal buffer management. Dirty data pages can be flushed into the disks before a transaction commits, and dirty data pages can also stay in the memory after a transaction commits. These flexible managements make the performance of the database improve, but also let the recovery process more difficult. As a result, ARIES heavily depends on its logging system. Before any writes happen, ARIES needs to record the log of this change and persist it on the disk. When failure happens, ARIES will use the log system to undo and redo the log, and recover the database to consistency and correct status.

2.2 motivation

In cloud-native databases, there are many differences. If we still use a centralized logging system, it will bring in heavy write operations and heavy network traffic. To address this problem, various methods are proposed in this new background. Amazon proposed Aurora [16] to offload redo processing to storage. Hoang Tam Vo et al. proposed a log-structured database in the cloud called LogBase [17], to remove the write bottleneck and support fast recovery. LogBase [17] is a cloud database system that applies LFS system, where the log is the database, so persisting the transactions is writing the logs. To keep resilience, logs are persisted on multiple scalable servers. Several optimizations are described to mitigate the performance of innately slow reading compared to table-based databases.

The databases with decoupled compute and storage build multi-tenant database services on top of a shared storage pool. CockroachDB [14] (CRDB) uses the Raft consensus algorithm for consistent replication. CRDB first uses range-partitioning on the keys to divide the data into contiguous ordered chunks, which are stored across the cluster. The chunks are called "Ranges". Raft maintains a consistent, ordered log of updates across a Range's replicas, and each replica individually applies commands to the storage engine as Raft declares them to be committed to the Range's log. CRDB uses Range-level leases, and to ensure that only one replica holds a lease at a time, replicas commit a special lease acquisition log entry to attempt to acquire a lease. Socrates [2] transforms the on-premise SQL Server into a DBaaS and further separates logging and storage from database kernel to dedicated services. In PolarDB Serverless [7], the disaggregated architecture allows nodes of each type to failover independently. It includes one primary and multiple read replicas. PolarDB Serverless adopts an ARIES-style recovery algorithm, which makes failed read replicas can be easily replaced with a new one using pages in the shared memory.

We are interested in these methods of implementing the logging system in a cloud-native database and what their advantages and disadvantages are, and also compared to the conventional database, what significant problems they solve.

3 RELATED WORK

Distributed logging protocol and server design [8] are introduced by the paper written by Daniels, Spector, and Thompson. And several database designs involve reducing the overhead of logging. Eleda [10] is a framework that is capable of scalable logging over multi-cores. With disaggregated storage, we can treat the multi-core system as multi-node clusters composed of a single core machine using the internet as data migrating path.

A study on the cost analysis on cloud database [15] has involved multiple databases like Redshift and Vertica. This paper draws a conclusion that different system architectures used in proprietary cloud offerings highlight interesting trade-offs when they are compared to non-proprietary systems. The aggressive intraquery parallelism of Redshift can bring significant performance advance for a single user but is degraded quickly when concurrency occurs. Athena, one of the popular serverless systems, provides on-demand querying capabilities and optimizes cloud DBMS to farm out different workloads.

4 CASE STUDIES

In this section, we will review and summary the design of some cloud-native databases. We first give a scratch of one database, and we then detailed illustrate the logging system of them. We also pay more attention to what is the difference between their logging system design and traditional system design, and why the new design will help improve the performance of the database. The brand new design is also associated with the specific features of this cloud-native database.

4.1 Microsoft Hyder

Hyder [4] is a database designed by Microsoft in 2011. Hyder uses a data-sharing architecture, and a cluster of servers running Hyder have shared access to the bottom shared log, which is stored on flash chips, via the network. The structure is shown in figure 1.

The log storage layer is the bottom layer of the Hyder structure. Compared to traditional design, which partitions the database and logs on many machines, Hyder only uses a shared log design.

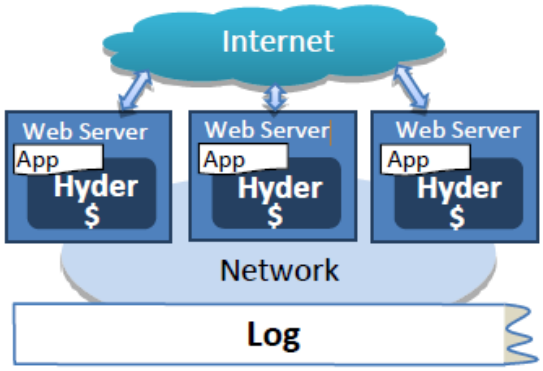

Figure 1: The architecture of Hyder

Traditional design has many disadvantages on the cloud-native background. Distributed transactions are difficult to avoid, and due to the expensive cost of a two-phase commit, also the need of transmitting and shift data and logs on the network, the performance of this design will be hurt. While in Hyder's design, all of the above servers can read and write in the entire database, Each transaction containing write operations will directly write to the bottom shared log, which can avoid the expensive two-phase commit. Also, the system can avoid the related locking behavior of a two-phase commit. With this design, Hyder can easily scale-out. There is no transmission between the above server, so we can extend the number of servers. All of the transmissions on the network happen between the above servers and the bottom shared log.

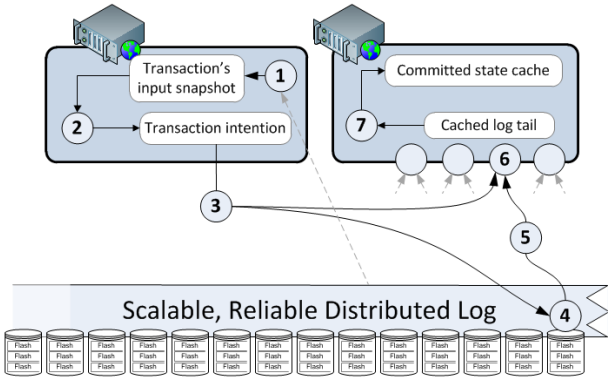

Figure 2: The transaction process of Hyder

Hyder's log system helps the database do transaction management. The general process is in figure 2. Hyder uses an optimistic concurrency control, which means it writes intent first and then validates the transaction. Each of the servers has the access to all of the databases, so a transaction only needs to be executed by one server. Firstly, the server will get a snapshot of the database. After the transaction is executed, the server collects all updates and builds an intention log. The intention log will broadcast to all other servers and also append to the bottom shared log. At the shared log layer, the logging system will generate the offset of one intention log, and also broadcast it to all other servers. If other servers find there is some intention log lost, they will directly request it from the bottom shared log. Each other server will execute the intention log and use a melding process to decide whether this transaction can actually commit. A conflict zone is the intentions between the last committed transaction in the log that contributed to the database state that the transaction reads and the intention log. The melding process will check each transaction committed in the conflict zone, and judge if the operation violates the isolation level of the database set. Because all of the servers run the same meld process and read the same log from the shared log, they will make the same decision on whether the transaction can commit. Using this method, there is no need for Hyder servers to communicate on the commit of transactions. The two-phase commit can be totally avoided.

The logging system is an important component for database fault tolerance. If a packet loses on the network, each server can simply reach the actual log from the bottom shared log. Also because the bottom flash storage is reliable and persistent, we can fast recover the database to a normal status. The servers don't store data pages, and the shared log is keeping reliable. The recovery process will be completed at a fast speed. Related recovery methods and failure handlers can also be found in [3].

4.2 Amazon Aurora

Aurora [16] is a database service designed for OLTP workloads by Amazon. The structure of Aurora is shown in figure 3.

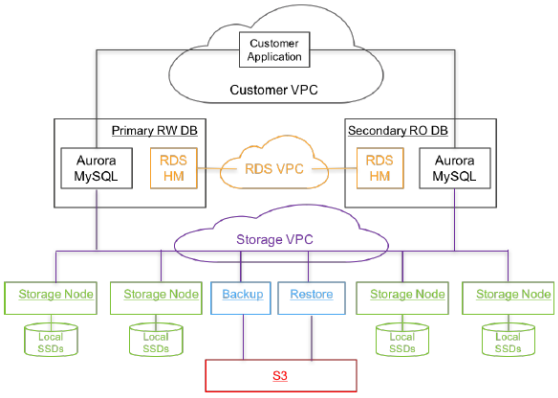

Figure 3: The architecture of Amazon Aurora

The database engine is developed from MySQL/InnoDB, and the biggest difference is how the engine operates reads, and writes operations to the disk. In InnoDB, writes operations will change the data in buffer page, and also record redo log in the WAL buffer. The InnoDB requires to redo log records should be persisted to the disk. While in Aurora, the redo log record must be executed atomically in each "mini transaction". These records are shared by the Protection Groups (PG) each log belongs to. We can treat each final log of a mini transaction as a consistency point.

Aurora heavily depends on the design of shifting redo log between Availability Zones. The key idea is shown in figure 4. In

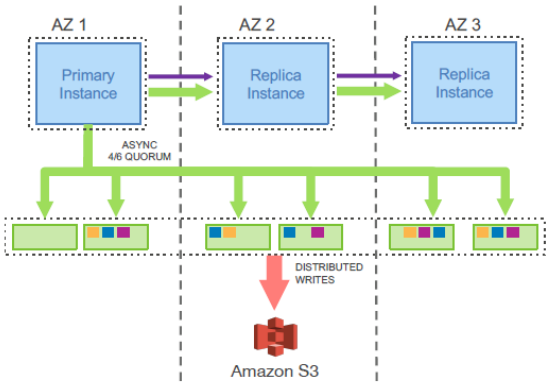

Figure 4: The redo log shifting of Amazon Aurora

traditional design, we need to exchange lots of information between database servers, such as log, binlog, data, double-write, and FRM files, which will bring a heavy burden on the network traffic, and severely affect the performance of cloud-native databases. The availability zones will also spend lots of time sending various information. After one availability zone receives information, it will write it to its own storage. This kind of design will several times amplify the write operations. To avoid the problems of heavy network traffic and heavy writing operations, Aurora chooses to only shift redo log. When a transaction wants to change one data page, it will generate a redo log. Instead of applying this redo log in the memory and writing back to the disk, Aurora simply pushes the redo log to the storage layer, and the log applicator will apply this redo log in the storage in the background or only on demand. This approach can strongly reduce the needed network I/O. Compared to Mirrored MySQL (traditional design), Aurora can reduce network I/O to one-seventh of original results.

The logging system also helps Aurora minimize the latency for write operations. The storage nodes retrieve the log records from the logging system, then persist record and acknowledge. Then the foreground process is done. The storage nodes will organize records and gossip with peers to fill up lost records. The storage nodes will periodically store log and data pages to the storage layer, Amazon S3. Many parts of the process are executed just in the background and obviously, reduce writing latency.

In the recovery process, Aurora also relies on its logging system. General ideas are similar to the traditional database, which rolls forward the log records and generate the prior state. Aurora still pushes the log applicator to the storage layer and goes through the redo log records. This characteristic helps Aurora obtain a fast recovery process. Aurora has no need to rebuild the runtime state during recovery. It contacts each protection group, which is sufficient to guarantee the discovery of any data that already reached a written quorum. In addition, Aurora needs to perform undo process to roll back the operations of an ongoing transaction when failure happens.

4.3 LogBase

LogBase [17] is a scalable log-structured database system. It is specifically designed to be deployed on cloud clusters to make full use of the scalability of the cloud environment. The key idea for LogBase is still removing the write bottleneck and speeding up the recovery process. There are also some differences between LogBase and the standard log-structured database system (LFS). LogBase provides an abstraction on top of the segmented log, which is fine-grained access to data records instead of accessing data page in general LFS. LogBase also implements in-memory indexes for achieving log records quickly.

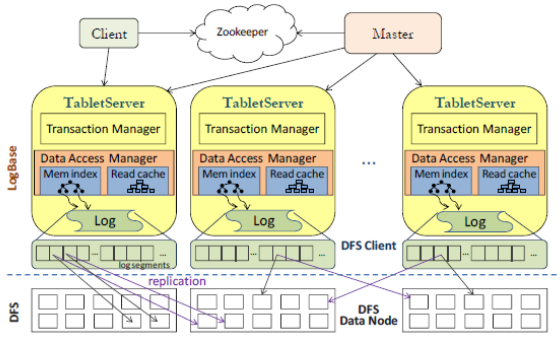

Figure 5: The architecture of LogBase

We can see the architecture of LogBase in 5. There are three layers in LogBase, which are transaction manager, data access manager, and log repository. In the structure, each tablet server will handle horizontal partitions of a table. The tablet server records data into its single log instance. Below that, there is a distributed file system (DFS) to store all of the log records and be shared by all of the tablet servers. Log Repository is at the bottom layer of the structure.

Due to the design of using DFS to store log records, we can use the log as the unique data repository, which will advance those write-heavy transactions. In LogBase, it is guaranteed that the system has a similar ability to recover data compared with the WAL protocol. In the specific implementation, LogBase uses Hadoop Distributed File System (HDFS) to store log records. Replicas of a data block in HDFS will be synchronously maintained and copied to other machines before returns.

A log record consists of two parts, the LogKey and Data. LogKey contains much information, such as log sequence number (LSN), table name, and table information. The Data also consists of two parts, the RowKey and Value. RowKey points out the position of change, and Value is the content of change. Log records are forced to be persisted before write operations return. This is similar to traditional database design.

4.4 Socrates

Socrates [2] is a DBaaS architecture that is implemented in Microsoft SQL Server, it is also available in Azure as SQL DB Hyperscale. The architecture of Socrates is shown in Figure 6. The design principles and goals include: 1) separation of computing and

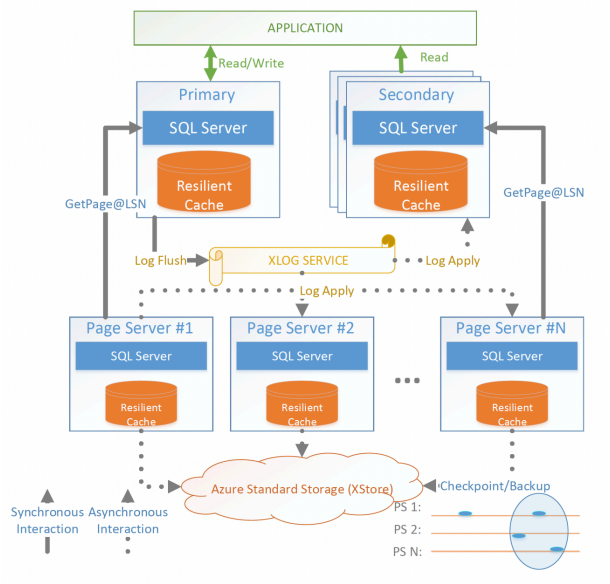

Figure 6: The architecture of Socrates.

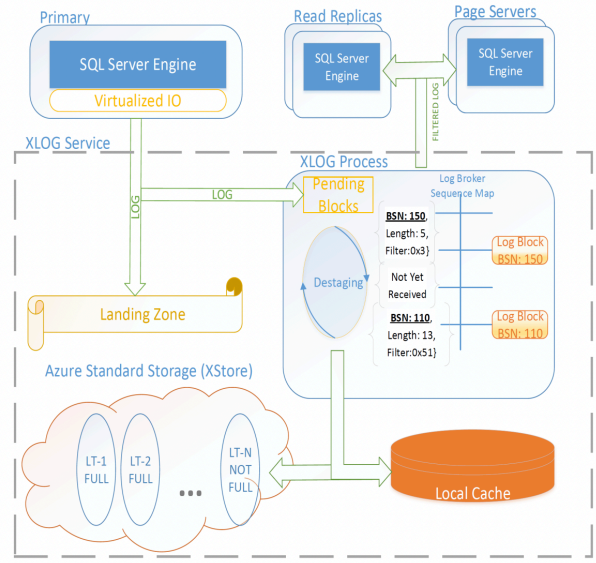

Figure 7: The XLOG service in Socrates.

storage, 2) tiered and scale-out storage, 3) bounded operations, 4) separation of the log from computing and storage, 5) opportunities to move the functionality into storage tier, 6) reuse of existing components.

There are tiers in the Socrates architecture, and the applications interact with the compute nodes. The implementation of compute nodes include query optimization, concurrency control, and security. There is one primary node, and when it fails, one of the secondaries becomes the new primary.

Socrates separates log from computing and storage, as it is shown as XLOG service in the architecture. The separation idea is also shown in some previous literature [5]. The authors think that this separation makes Socrates different from other cloud database systems. The XLOG service achieves low commit latency and good scalability at the storage tier. As only primary writes to the log, the primary will process all updates. The single writer approach asserts low latency and high throughput when writing to the log. The other secondaries consume the log in an asynchronous way to keep their copies of data up to date.

The XLOG service in Socrates is shown in Figure 7. The primary compute node writes log blocks directly to a landing zone which is a fast and durable storage service that provides strong guarantees on data integrity, resilience, and consistency. When it comes to writing, as the landing zone is supposed to be fast but small, the primary writes log blocks synchronously and directly to the landing zone for the lowest possible commit latency. The format of the log is a backward-compatible extension of the traditional SQL Server log format used in Microsoft's SQL services. A key property of this log extension is that it allows concurrent log readers to read consistent information in the presence of log writers without any synchronization. Writing log blacks to the XLOG process will disseminate the log blocks to Page Servers and secondaries. The writing process to the landing zone and the XLOG process are parallel. To avoid data loss, XLOG only disseminates hardened log blocks, which are blocks that have already been made durable in the landing zone. So the process can be concluded as: the primary writes all log blocks first into a pending area of the XLOG process, then primary will inform XLOG of all hardened log blocks. XLOG will move them from the pending area for dissemination once they are hardened.

4.5 CockroachDB

CRDB [14] is a scalable SQL DBMS that is built to support global OLTP workloads while maintaining high availability and strong consistency. CRDB uses a shared-nothing architecture, in which all nodes are used for both data storage and computation. CRDB guarantees fault tolerance and high availability through replication of data, automatic recovery mechanisms, and strategic data placement. The replication process is carried out using the Raft consensus algorithm [13]. Replicas of a Range form a Raft group. Raft maintains a consistently ordered log of updates across a Range's replicas, and each replica individually applies commands to the storage engine as Raft declares them to be committed to the Range's log. CRDB uses Range-level leases, there is a single replica in the Raft group that plays the role of the leaseholder. It is the only replica allowed to serve authoritative up-to-date reads or propose writes to the Raft group leader. There are mechanisms to ensure only one replica holds a lease at a time, as well as prevent two replicas from acquiring leases overlapping in time.

As for replica placement, CRDB supports both manual and automatic ways to control replica placement. For the manual part, users can set the configuration of each individual node in CRDB with a set of attributes. For the automatic way, CRDB will spread replicas across failure domains to tolerate varying severities of failure nodes. It also uses various heuristics to balance load and disk utilization.

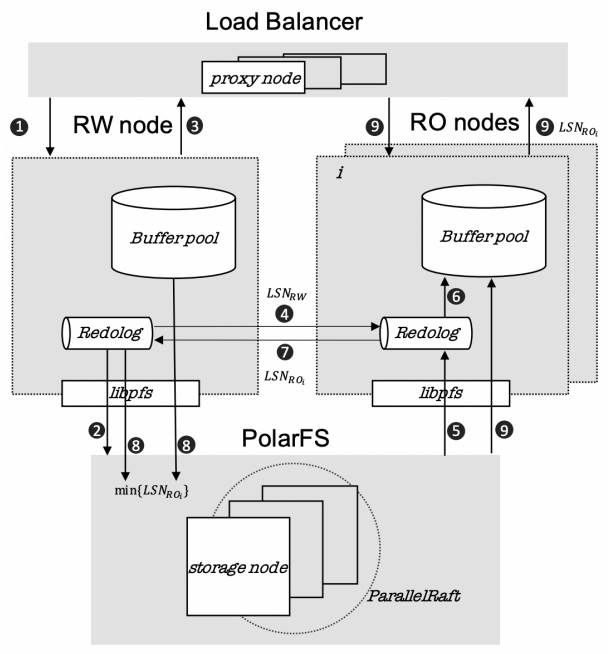

Figure 8: The architecture of PolarDB Serverless.

4.6 PolarDB Serverless

PolarDB Serverless follows the disaggregation design paradigm, which means the CPU resource on compute nodes is decoupled from remote memory poll and storage pool. The advantage of following this paradigm is that each resource pool can grow or shrink independently. Therefore, on-demand provisioning at multiple dimensions can be provided and reliability can also be kept. THe PolarDB Serverless is an extension of the original PolarDB [11], with disaggregation architecture applied to the design. The architecture of PolarDB is shown in Figure 8, it is based on shared storage architecture. It includes one primary (RW node) and multiple read replicas (RO nodes) in the compute node layer. The storage poll and code base is on PolarFS [6]. PolarDB Serverless shows that this new version provides new opportunities for the design of new cloud-native and serverless databases. The efficiency and some performance drawbacks can also be avoided under this architecture. The idea of introducing PolarDB Serverless is that serverless databases are highly elastic derivatives of cloud-native databases. The main design goal of serverless databases is on-demand resource provisioning, which enables agile resource adjustment according to actual workloads and provides a pay-as-you-go payment model.

Traditional monolithic databases periodically flush dirty pages to durable storage, which causes a large amount of network communication between RW, memory nodes, and the PolarFS component. As a result, it affects the performance of high-priority operations in the critical path. In fact, such a problem is common in many cloud-native databases. Previous cloud-native databases use different solutions, such as Aurora [16] propose the concept that "log is the database", so it treats redo log records as a series of incremental page modifications and generates the latest version by applying redo logs continuously to the pages on the storage nodes. Socrates [2] separates log from storage, etc. Here in Polar DB Serverless, adopts a similar approach to Socrates. It extends PolarFS so that logs and pages are separately stored in two types of chunks, which are called log chunks and page chunks.

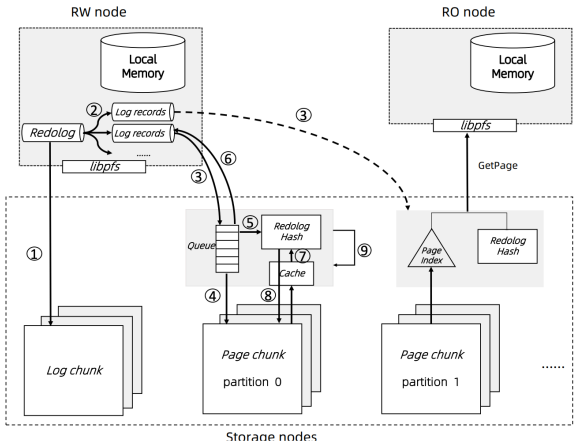

Figure 9: The page materialization offloading process of PolarDB Serverless.

In PolarDB Serverless, logs are sent only to the leader node of the page chunk, which will materialize pages and propagate updates to other replicas through the ParallelRaft consensus algorithm. In Figure 9, it is shown that a storage node in PolarFS can host multiple log chunks and page chunks. RW will flush changes of redo log files into log chunks before a transaction commits. After the page chunk's leader node receives log records from RW, these records are synchronized to replicas to ensure durability.

4.7 LegoBase

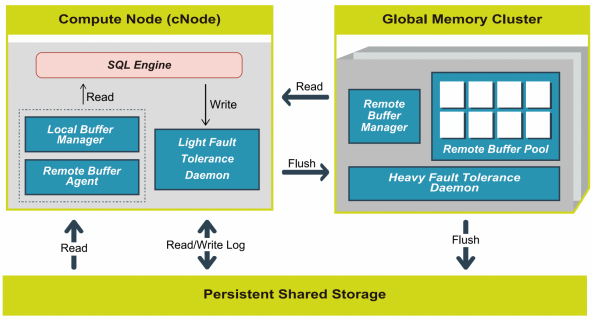

Figure 10: The architecture of LegoBase.

LegoBase [18] tries to split conventional ARIES fault tolerance protocol to independently handle the local and remote memory failures for fast recovery of compute instances. According to the authors' view, current cloud-native databases still embrace the monolithic server architecture, which makes CPU and memory tightly coupled. Such a situation makes it hard to fulfill the ever-growing

and highly elastic resource demands from web-scale applications, as they find that the CPU utilization of database instances cloud is low most of the time, but occasionally may reach high levels when handling bursty traffic such as a promotion event. Therefore, in LegoBase, they advocate the co-design of database kernel and memory disaggregation to exploit the full potential of flexible memory and CPU resource allocation in the context of the cloud-native database.

The overview architecture of LegoBase is shown in Figure 10. There are 3 key components in the architecture: 1) Persistent Shared Storage (pStorage) what employs replication protocols for storing write-ahead logs, checkpoints, and database tables, 2) Compute Node (cNode) which performs SQL queries by consuming data from pStorage, 3) Global Memory Cluster (gmCluster), which allocates infinite remote memory to hold the bulk of cNode's working set that cannot fit into its local memory. Although memory is split across local and memory buffers in the context of memory disaggregation, conventional designs still treat the two buffers as an entirety. So the whole memory space can be recovered by replaying the write-ahead logs when a local compute node crashes. In LegoBase, logs are replicated to a collection of replicas, and the consensus protocol ParallelRaft is used to guarantee the data consistency among replicas.

5 EVALUATION AND DISCUSSION

In Section 4 we have studied cases for multiple database logging systems. Now in this part, we will quantitatively present the performance improved by the design of the resilience system listed in Section 4. Since each resilience and logging system is designed for a specific database, it is inevitable to analyze the logging with the database architecture. And with the variety of corresponding databases, direct evaluation and comparison among different logging system performances are extremely hard. As a result, we will separately present the evaluation and experiment using the data from corresponding papers. In Section 5.2 we will propose some factors to classify or compare the logging system designs among listed databases, and think further from logging itself.

5.1 evaluation

5.1.1 LogBase. The experiments of LogBase focus on writing performance and reading trade-offs. A variety of workloads are tested on the database. Note the LogBase is built on the top of HDFS, the baseline for comparison is HBase, a NoSQL database native to HDFS. The experiment is conducted on 24 quad-core machines, each with 8 GB main memory,500 GB of disk (HDD), and 1 Gb network Ethernet connection. In the following micro-benchmark, Less than 1 million tuples are written to LogBase and HBase.

Figure 11 indicates the writing performance of LogBase is about 5X than HBase with writing 250K tuples, while about 2.5X with 1 million tuples. The reason for the difference is ascribed to HBase must persist changed records into a memtable along with the logs. In other words, LogBase only writes once while HBase must write twice.

The reason for decreased improvement with increased writing size is not discussed in the paper but implicitly pointed out to

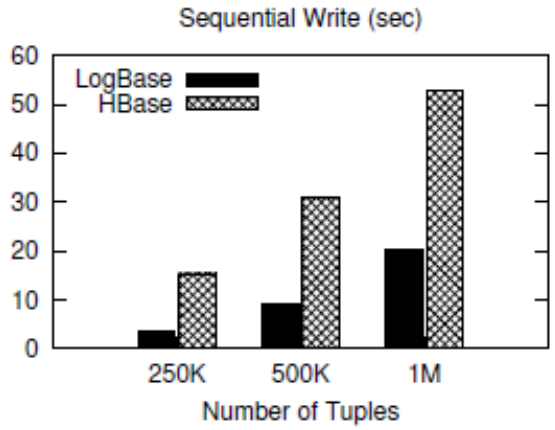

Figure 11: Micro Write Benchmark

mainly be the compacting process. Compaction is regularly performed to optimize reading, especially ranged scanning, of LogBase. Such operation is necessary for log-structured storage to optimize reading performance.

The random reading performance is shown in figure 12. With caching, LogBase and HBase have similar performance. By contrast, the dense in-memory index provides a significant boost of out-cache random access. However, the log-structured storage is not suitable for range scans. Figure 13 indicates the linearly correlated access latency and scan range without compaction. Even with compaction, LogBase will not outperform HBase for the ranged scan.

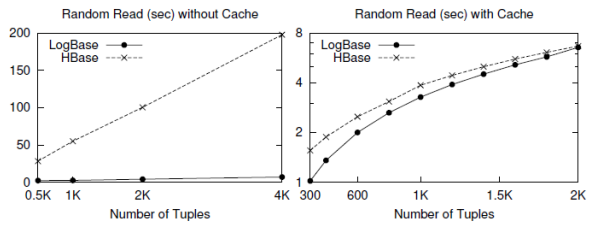

Figure 12: Random Read Benchmark

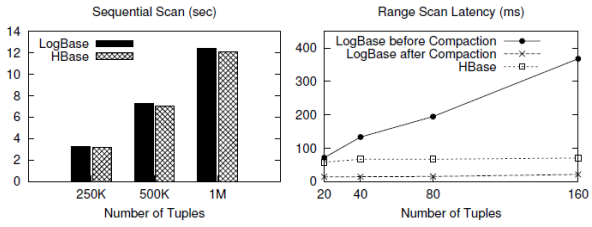

Figure 13: Ranged Scanning Benchmark

The empirical data on standard test YCSB as shown in figure 14 insertion for LogBase is 2X faster than HBase, and throughput outperforms HBase. The larger throughput gap of 75% update is ascribed to out-cache read performance.

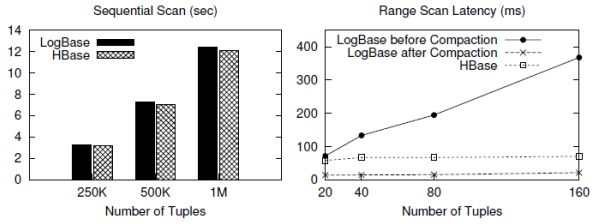

Figure 14: LogBase YCSB

5.1.2 Hyder. Hyder is an in-memory database whose design is based on the new hardware around 2011, which are flash memory, multi-core processors, and a High-Speedy Network of up to 40Gb/s. The experiments are simulated instead of empirical data on standard benchmarks, using the parameters of the hardware, such as latency, to generate theoretical results. To simulate the experiment, the parameters of hardware is set to follow:

- data size - 100 bytes
- network - 40Gb/s, 80% utilized, latency 400ns
- read latency 2us from the cache
- write latency 10us
- Each transaction performs 8 reads and 2 writes.
- Melding the logs has 10us latency.

The key under this logging design is that flash memory enables random updating in a specific location, and reading is never divided into sequential and random as stated by the author. As a result, logs can be persisted to multiple locations on the flash in parallel, although the theoretical analysis indicates logging is still the bottleneck of this implementation to be 100K TPS limited by the flash memory bandwidth.

Note the Hot-X-y indicates X% of transactions access y% of data, indicating the skewness level of transactions. Note that DB size

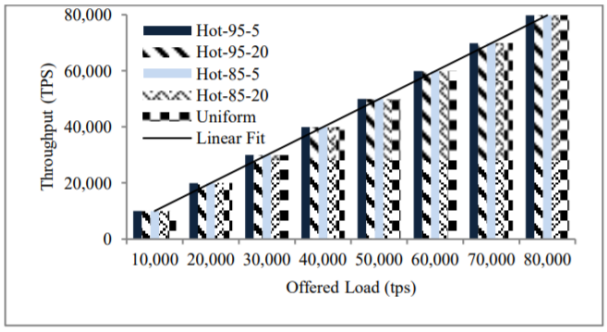

Figure 15: Throughput for offered load

means the size of the database (number of records); the smaller the DB is, the more likely transactions have conflicts and cause an abort. R:W ration means read/write ratio of each transaction, TXN-20 is a setting where more than 50% of transactions abort.

Figure 15 shows throughput for all workloads in the experiments scales linearly. The underlying conclusions are: first, the above workload does not touch bottleneck correlated to one component;

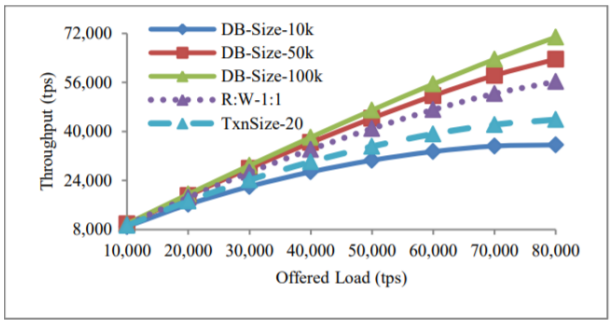

Figure 16: Throughput for different workload

second, the cost of scaling out itself is little. In figure 16 the DB skewness is set to 99% transaction accessing 1% of data. Even with a 50% writing ratio, Hyder still performs well on throughput, showing logging has little damage on throughput. The log-structured database works well on such a setting that can scale out easily to hardware limits with little performance damage.

5.1.3 Aurora. Aurora is a cloud-native database sharing the storage across all computation nodes. The experiment of Aurora is comparing its performance with MySQL deployed on the cloud. The setups are EC2 instances with 32 vCPUs and 244GB of memory, Intel Xeon E5-2670 v2, r3.8xlarge is set to 170GB. The first experiment summarized by figure 17 shows Aurora can scale-out linearly, achieving 5X throughput. Although not explicitly stated, the storage (Disk) IO is no longer the bottleneck.

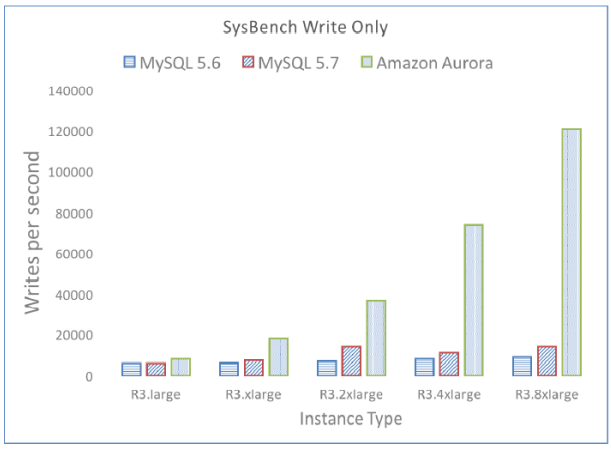

Figure 17: Write Performance of Aurora under SysBench 1GB data on 5 EC2 instances

Since redo Logs are sent to the storage as well as other replicas, the latency of synchronizing including network IO can become the bottleneck, thus synchronizing latency is meaningful to analyze the logging performance in shared storage architecture. In figure 18, the read lag, the time span needed for replicas to be updated with the log, is less than 5.5 milliseconds when more than 10K writing occurs per second. Compared to MySQL whose read lag can come up to 300 seconds, 5 milliseconds is trivial.

Writes/sec	Amazon Aurora	MySQL
1,000	2.62	< 1000
2,000	3.42	1000
5,000	3.94	60,000
10,000	5.38	300,000

Figure 18: Reading Lag of Secondary Replicas

5.1.4 Socrates. Socrates is a cloud database developed by Microsoft, which extends the idea of shared storage similar to Aurora. However, the difference in storing and applying logs provides Socrates with outstanding logging performance. The figure 19 states the latency of commitment is less than 0.5 milliseconds, where commitment is right after persisting logs. In other words, the latency of the log is less than 0.5 milliseconds.

	Today	Socrates
Max DB Size	4TB	100TB
Availability	99.99	99.999
Upsize/downsize	O(data)	O(1)
Storage impact	4x copies (+backup)	2x copies (+backup)
CPU impact	4x single images	25% reduction
Recovery	O(1)	O(1)
Commit Latency	3 ms	< 0.5ms
Log Throughput	50MB/s	100+ MB/s

Figure 19: Socrates Info

The experiment of Socrates mainly focuses on comparing the throughput and log throughput with HADR on Azure. Figure 20 shows the throughput of default mix of CDB workload with 1TB data size. Socrates is a 5% fall over HADR due to the higher network IO cost. However, in figure 21, when dealing with heavy write CDB workload, Socrates's log throughput is about 50% more than HADR's. The results show that Socrates has better logging performance than HADR, ascribing to better log management.

	CPU %	Write TPS	Read TPS	Total TPS
HADR	99.1	347	1055	1402
Socrates	96.4	330	1005	1335

Figure 20: CDB Throughput: HADR vs. Socrates (1TB)

	SF	Log MB/s	CPU %
HADR	30000	56.9	46.2
Socrates	30000	89.8	73.2

Figure 21: CDB Log Throughput: HADR vs. Socrates

5.1.5 CockRoachDB. CockRoackDB, although developed around 2020, uses a shared-nothing structure. However, its focus is on high availability and consistency across geographical locations. Stated on the website [1], read and write latency of CockRoachDB are 4.3ms and 0.7ms for SysBench Benchmark in single availability zone shown in fugure 22. However, if the servers are located in different availability zone, the latency will increase significantly as shown in figure 23.

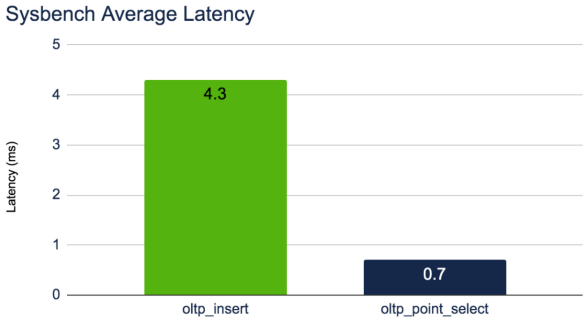

Figure 22: Latency Sysbench single Availability Zone

	Warehouses		
	1,000	**10,000**	**100,000**
CockroachDB			
Max tpmC	12,474	124,036	1,245,462
Efficiency	97.0%	96.5%	98.8%
NewOrder p90 latency	39.8 ms	436.2 ms	486.5 ms
Machine type (AWS)	c5d.4xlarge	c5d.4xlarge	c5d.9xlarge
Node count	3	15	81
Amazon Aurora [55]			
Max tpmC	12,582	9,406	-
Efficiency	97.8%	7.3%	-
Latency, machine type, and node count not reported			

Figure 23: TPC-C Across Geo-Locations

5.1.6 PolarDB. PolarDB is also a cloud-native database developed by Alibaba. The design inherits Aurora but is closer to Socrates. The logs are persisted in the storage but are only sent to leader nodes who will push changes to replicas. However, the paper does not explicitly state the performance of logging or distributing logs across nodes but claims the latency of applying logs asynchronously is higher than Socrates' while not significant.

Besides regular benchmarks, the experiment listed in the PolarDB paper involves fast recovery, as shown in figure 24. The results show fully recovering from an unpredicted crash takes 43 seconds with remote memory, while 125 seconds without.

5.1.7 LegoBase. LegoBase is an innovative database employing a shared memory matrix in its architecture. With shared memory, logs are persisted to shared storage, while local dirty pages are flushed to shared memory. In the LegoBase paper, the experiments are focused on latency and throughput within LegoBase and Infiniswap in different local data ratios, as well as recovery time between

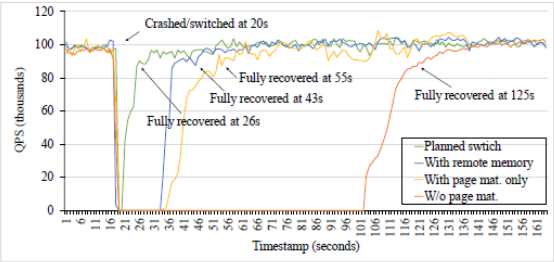

Figure 24: Recovery Time for RW nodes

LegoBase and MySQL. The setup of the experiments are three machines with two Xeon CPU E5-2682 v4 processors, 512GB DDR4 DRAM, and 25Gbps network; both the gmCluster and Infiniswap's remote paging are composed of 200GB memory space.

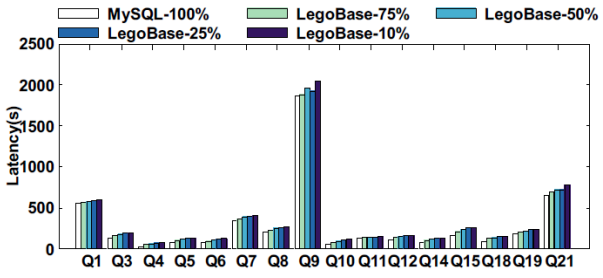

Figure 25: Latency of Benchmarking on TPC-H

Figure 25, provides the comparison of TPC-C latency among local data ratio (percentage of data in local memory). Compared to MySQL, which operates fully on local memory, LegoBase with more than 50% local data has comparable throughput and latency. And with only 10% performance drag when only 10% data is stored in local memory.

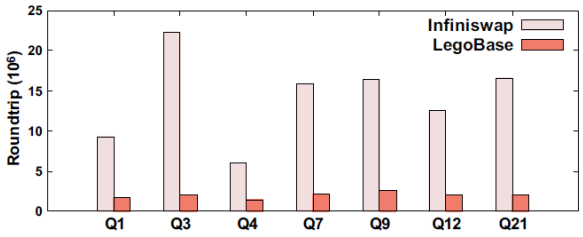

Figure 26: Network Access during TPC-H Benchmark

Figure 26, shows the number of remote memory access for TPC-H query on LegoBase and Infiniswap. LegoBase has about 5X less network traffic between local and remote memory.

Recovery of local memory crashes is fast because of shared-matrix contains most in-memory content, so recovering what is left in the crashed memory is enough. The time for recovering from a local memory crash usually takes 2-3 seconds if less than 50% of data is in crashed local memory. As shown in figure 27, the

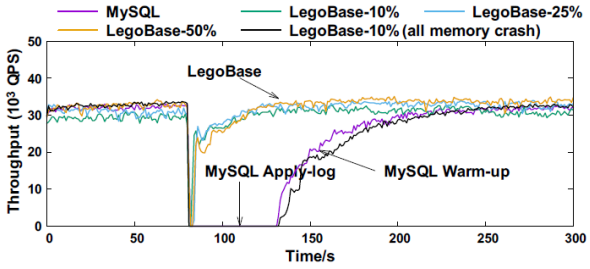

Figure 27: Time for recovering from failure and warming up

throughput of LegoBase 3 seconds after memory crash restores about 80%. This feature can be ascribed to frequent checkpointing from local memory to shared memory.

With the shared memory and disk, high-speed network, the workload of logging is alleviated by easily scalable storage and low latency network.

5.2 discussion

With the experiments above, it is not hard to see the similarity and differences between different designs. More or less, the database listed in this paper employs the idea of "log is the database", where logs are no longer merely a backup for resilience. Now we propose some perspectives to think about the log system design.

5.2.1 Architecture. The architecture refers to the way of implementing the database. Here we classify architectures into three classes:

- Shared-nothing: distinct server connected through the network
- Shared-Storage: segregate storage to another tier with computation power
- Shared-Storage & Memory: adding a shared memory matrix with computation power

In a shared-nothing architecture, globally centralized logging implementing ARIES usually becomes the bottleneck due to disk IO constraints. By contrast, shared-Storage enables scalable storage thus storage IO of the local disk is less a concern, but such design requires high bandwidth low latency networks between storage and computation tier. With the disaggregated-memory architecture, fast recovery is possible. In this design, logging is not the main focus of the study because it is not the bottleneck of normal operations.

5.2.2 Workload.

- Write: LogBase
- Read: Aurora, Socrates, PolarDB, LegoBase
- Hybrid: Hyder

The workload is also a crucial perspective to discuss the logging, which records the changes in volatile memory. In general, a write-intensive workload creates more logs and logging is more likely to become a bottleneck hampering the performance. For example, logging performance is critical for LogBase, where the majority of operations are writing operations. Thus, the design overhauls the conventional database to optimize writing before thinking over any optimization to alleviate reading.

However, given that the majority of workloads are reading, scanning, and selecting based on the common benchmark used in the listed papers, the throughput or latency of logging in read-intensive workloads like TPC-C is less a concern than the performance of other components. Moreover, given the logging serves as a method for consistency, one should consider the time used for synchronizing multiple servers without using logs along with persisting logs to disks. Therefore the significance of logging latency is further reduced. As a result, recent papers listed in this paper do not describe logging as a distinct component of database design.

5.2.3 Hardware. The major hardware that influences the design of logging listed in this paper are:

- Network: connections between machines or between computation and storage
- Disk: the main component of the storage tier

The evolution of logging design is closely correlated with the hardware. The high bandwidth low latency network enables transferring a large amount of data among nodes. Flash memory enables in place parallel writing. Both alleviate the performance drag of logging and allow researchers to focus on another perspective of database design.

6 CONCLUSION

In this paper, we reviewed and evaluated multiple database logging designs with three architectures. Our evaluation indicates logging is an inevitable cost for all database implementations. The goal of database logging design is mitigating the performance drag caused by logging using new hardware, reconstructing file systems, introducing new architectures and workload analysis. It is extremely difficult to state which solution to boosting logging performance is better because the setup of DBMS are keep changing and the requirements from users vary. It is entirely possible a breakthrough in hardware subverts the current design or moves back to the older design. As a result, database logging design problems should be regularly visited as new technology emerges.

7 CONTRIBUTION

We evenly do the task on this project. We three all participated in the initial brainstorming, finding related papers, making presentations, and paper writing. Also we three joined all of the meetings when we discussed the direction of the project and scratch of the proposal, presentation, and final report. More detailed in this report, Huiyu wrote section 2, Section 3, and the case study of {Microsoft Hyder, Amazon Aurora, LogBase}, and Wendi wrote Section 1 and the case study of {Socrates, CockroachDB, PolarDB Serverless, LegoBase}, and Zhikang wrote Section 5 and Section 6.

REFERENCES

[1] [n.d.]. CockroachDB performance. https://www.cockroachlabs.com/docs/stable/performance.html

[2] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, et al. 2019. Socrates: The new sql server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data*. 1743–1756.

[3] Mahesh Balakrishnan, Philip A. Bernstein, Dahlia Malkhi, Vijayan Prabhakaran, and Colin W. Reid. 2010. Brief Announcement: Flash-Log - A High Throughput Log. In *Distributed Computing, 24th International Symposium, DISC 2010, Cambridge, MA, USA, September 13-15, 2010. Proceedings (Lecture Notes in Computer Science)*, Nancy A. Lynch and Alexander A. Shvartsman (Eds.), Vol. 6343. Springer, 401–403. https://doi.org/10.1007/978-3-642-15763-9_39

[4] Philip A Bernstein, Colin W Reid, and Sudipto Das. 2011. Hyder-A Transactional Record Manager for Shared Flash.. In *CIDR*, Vol. 11. 9–20.

[5] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. 2008. Building a database on S3. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 251–264.

[6] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1849–1862.

[7] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, et al. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *Proceedings of the 2021 International Conference on Management of Data*. 2477–2489.

[8] Dean S. Daniels, Alfred Z. Spector, and Dean S. Thompson. 1987. Distributed Logging for Transaction Processing. *SIGMOD Rec.* 16, 3 (Dec. 1987), 82–96. https://doi.org/10.1145/38714.38728

[9] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. 1981. The Recovery Manager of the System R Database Manager. *ACM Comput. Surv.* 13, 2 (jun 1981), 223–242. https://doi.org/10.1145/356842.356847

[10] Hyungsoo Jung, Hyuck Han, and Sooyong Kang. 2017. Scalable Database Logging for Multicores. *Proc. VLDB Endow.* 11, 2 (Oct. 2017), 135–148. https://doi.org/10.14778/3149193.3149195

[11] Feifei Li. 2019. Cloud-native database systems at Alibaba: Opportunities and challenges. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2263–2272.

[12] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.* 17, 1 (March 1992), 94–162. https://doi.org/10.1145/128765.128770

[13] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 305–319.

[14] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1493–1509.

[15] Junjay Tan, Thanaa Ghanem, Matthew Perron, Xiangyao Yu, Michael Stonebraker, David DeWitt, Marco Serafini, Ashraf Aboulnaga, and Tim Kraska. 2019. Choosing a Cloud DBMS: Architectures and Tradeoffs. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2170–2182. https://doi.org/10.14778/3352063.3352133

[16] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1041–1052. https://doi.org/10.1145/3035918.3056101

[17] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. 2012. LogBase: A Scalable Log-Structured Database System in the Cloud. *Proc. VLDB Endow.* 5, 10 (June 2012), 1004–1015. https://doi.org/10.14778/2336664.2336673

[18] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Xinjun Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, et al. 2021. Towards cost-effective and elastic cloud database deployment via memory disaggregation. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1900–1912.