Distributed Database Systems: The Case for NewSQL

Patrick Valduriez, Ricardo Jiménez-Peris, M. Tamer Özsu

Distributed Database Systems: The Case for NewSQL

Patrick Valduriez[1], Ricardo Jimenez-Peris[2] and M. Tamer Özsu[3]

[1] Inria, University of Montpellier, CNRS, LIRMM, France
[2] LeanXcale, Madrid, Spain
[3] University of Waterloo, Canada

Abstract. Until a decade ago, the database world was all SQL, distributed, sometimes replicated, and fully consistent. Then, web and cloud applications emerged that need to deal with complex big data, and NoSQL came in to address their requirements, trading consistency for scalability and availability. NewSQL has been the latest technology in the big data management landscape, combining the scalability and availability of NoSQL with the consistency and usability of SQL. By blending capabilities only available in different kinds of database systems such as fast data ingestion and SQL queries and by providing online analytics over operational data, NewSQL opens up new opportunities in many application domains where real-time decisions are critical. NewSQL may also simplify data management, by removing the traditional separation between NoSQL and SQL (ingest data fast, query it with SQL), as well as between operational database and data warehouse / data lake (no more ETLs!). However, a hard problem is scaling out transactions in mixed operational and analytical (HTAP) workloads over big data, possibly coming from different data stores (HDFS, SQL, NoSQL). Today, only a few NewSQL systems have solved this problem. In this paper, we make the case for NewSQL, introducing their basic principles from distributed database systems and illustrating with Spanner and LeanXcale, two of the most advanced systems in terms of scalable transaction management.

Keywords: Distributed database, database system, DBMS, SQL, NoSQL, NewSQL, polystore, scalable transaction management.

1 Introduction

The first edition of the book Principles of Distributed Database Systems [10] appeared in 1991 when the technology was new and there were not too many products. In the Preface to the first edition, we had quoted Michael Stonebraker who claimed in 1988 that in the following 10 years, centralized DBMSs would be an "antique curiosity" and most organizations would move towards distributed DBMSs. That prediction has certainly proved to be correct, and most systems in use today are either distributed or parallel. The fourth edition of this classic textbook [11] includes the latest developments, in particular, big data, NoSQL and NewSQL.

NewSQL is the latest technology in the big data management landscape, enjoying a rapid growth in the database system and business intelligence (BI) markets. NewSQL combines the scalability and availability of NoSQL with the consistency and usability

of SQL (SQL here refers to SQL database systems, which is the common term used for relational database systems). By providing online analytics over operational data, NewSQL opens up new opportunities in many application domains where real-time decisions are critical. Important use cases are IT performance monitoring, proximity marketing, e-advertising, risk monitoring, real-time pricing, real-time fraud detection, internet-of-things (IoT), etc.

NewSQL may also simplify data management, by removing the traditional separation between NoSQL and SQL (ingest data fast, query it with SQL), as well as between the operational database and data warehouse or data lake (no more complex data extraction tools such as ETLs or ELTs!). However, a hard problem is scaling transactions in mixed operational and analytical (HTAP) workloads over big data, possibly coming from different data stores (HDFS files, NoSQL databases, SQL databases). Currently, only a few NewSQL database systems have solved this problem, e.g., the LeanXcale NewSQL database system that includes a highly scalable transaction manager [9] and a polystore [6].

NewSQL database systems have different flavors and architectures. However, we can identify the following common features: support of the relational data model and standard SQL; ACID transactions; scalability using data partitioning in shared-nothing clusters; and high availability using data replication. We gave a first in-depth presentation of NewSQL in a tutorial at the IEEE Big Data 2019 conference [15], where we provide a taxonomy of NewSQL database systems based on major dimensions including targeted workloads, capabilities and implementation techniques.

In this paper, we make the case for NewSQL. First, we briefly recall the principles of distributed database systems (Section 2), which are at the core of NewSQL systems. Then, we briefly introduce SQL systems (Section 3) and NoSQL systems (Section 4). In Section 5, we present in more details our case for NewSQL. Finally, Section 6 concludes and discusses research directions in NewSQL.

2 Principles of Distributed Database Systems

The current computing environment is largely distributed, with computers connected to Internet to form a worldwide distributed system. With cluster and cloud computing, organizations can have geographically distributed and interconnected data centers, each with hundreds or thousands of computers connected with high-speed networks. Within this environment, the amount of data that is captured has increased dramatically, creating the big data boom. Thus, there is a need to provide high-level data management capabilities, as with database systems, on these widely distributed data in order to ease application development, maintenance and evolution. This is the scope of distributed database systems [11], which have moved from a small part of the worldwide computing environment a few decades ago to mainstream today.

A distributed database is a collection of multiple, logically interrelated databases located at the nodes of a distributed system. A distributed database system is then defined as the system that manages the distributed database and makes the distribution transparent to the users, providing major capabilities such as data integration, query processing, transaction management, replication, reliability, etc.

There are two types of distributed database systems: geo-distributed and single location (or single site). In the former, the sites are interconnected by wide area networks that are characterized by long message latencies and higher error rates. The latter consist of systems where the nodes are located in close proximity allowing much faster exchanges leading to much shorter message latencies and very low error rates. Single location distributed database systems are typically characterized by computer clusters in one data center and are commonly known as parallel database systems [4] emphasizing the use of parallelism to increase performance as well as availability. In the context of the cloud, it is now quite common to find distributed database systems made of multiple single site clusters interconnected by wide area networks, leading to hybrid, multisite systems.

In a distributed environment, it is possible to accommodate increasing database sizes and bigger workloads. System expansion can usually be handled by adding processing and storage power to the network. Obviously, it may not be possible to obtain a linear increase in "power" since this also depends on the overhead of distribution. However, significant improvements are still possible. That is why distributed database systems have gained much interest in scale-out architectures in the context of cluster and cloud computing. Scale-out (also called horizontal scaling) should be contrasted with the most commonly used scale-up (also called vertical scaling), which refers to adding more power to a server, e.g., more processors and memory, and thus gets limited by the maximum size of the server. Horizontal scaling refers to adding more servers, called "scale-out servers" in a loosely coupled fashion, to scale almost infinitely. By making it easy to add new component database servers, a distributed database system can provide horizontal scaling, in addition to vertical scaling.

In the context of the cloud, there is also the need to support elasticity, which is different from scalability, in order to adapt to workload changes and be able to process higher (or lower) transaction rates. Elasticity requires dynamically provisioning (or deprovisioning) of servers to increase (or decrease) the global capacity and live data migration 1, e.g., moving or replicating a data partition from an overloaded server to another, while the system is running transactions.

In observing the changes that have taken place since the first edition of the book [14], which focused on distribution in relational databases, what has struck us is that the fundamental principles of distributed data management still hold, and distributed data management can still be characterized along three dimensions: distribution, heterogeneity and autonomy of the data sources (see Fig. 1). This characterization has been useful for all subsequent editions, to include the new developments in many topics, which are reflected in Fig. 1, which identifies four main categories: the early client-server database systems, with limited distribution; the recent distributed and parallel systems, including NoSQL and NewSQL with high distribution; the P2P systems with high distribution and autonomy; and the multidatabase systems and polystores with high distribution, autonomy and heterogeneity.

What has changed much since the first edition of the book and made the problems much harder, is the scale of the dimensions: very large-scale distribution in the context of cluster, P2P, and cloud; very high heterogeneity in the context of web and big data; and high autonomy in the context of web and P2P. It is also interesting to note that the fundamental principles of data partitioning, data integration, transaction management, replication and SQL query processing have stood the test of time. In particular, new

techniques and algorithms (NoSQL, NewSQL) can be presented as extensions of earlier material, using relational concepts.

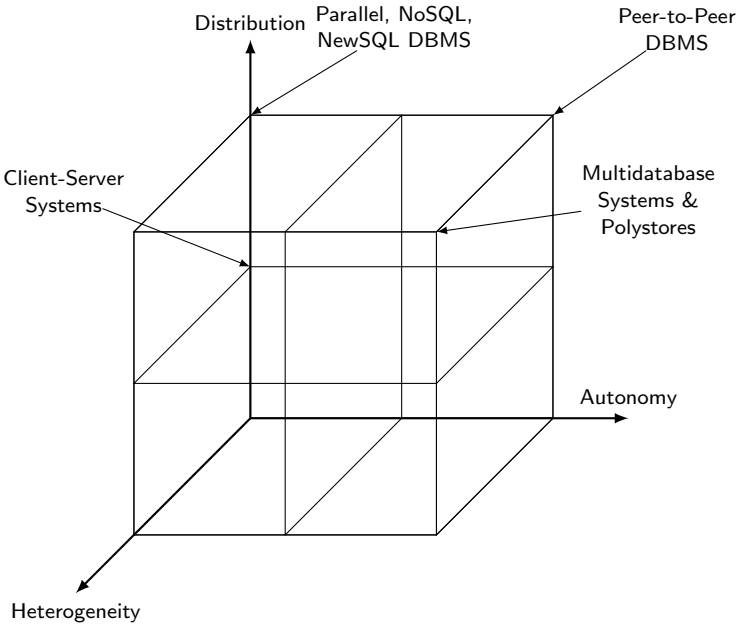

Fig. 1. Distributed database system dimensions (modified after [11]).

3 SQL Database Systems

SQL (relational) database systems are at the heart of any information system. The fundamental principle behind relational data management is data independence, which enables applications and users to deal with the data at a high conceptual level while ignoring implementation details. The major innovation of SQL database systems has been to allow data manipulation through queries expressed in a high-level (declarative) language such as SQL. Queries can then be automatically translated into optimized query plans that take advantage of underlying access methods and indices. Many other advanced capabilities have been made possible by data independence: data modeling, schema management, consistency through integrity rules and triggers, ACID transaction support, etc.

The fact that SQL has been an industry standard language has had a major impact on the data analysis tool industry, making it easy for tool vendors to provide support for reporting, BI and data visualization on top of all major SQL database systems. This trend has also fostered fierce competition among tool vendors, which has led to high-quality tools for specialized applications.

The data independence principle has also enabled SQL database systems to continuously integrate new advanced capabilities such as object and document support and to adapt to all kinds of hardware/software platforms from very small devices to very large computers (NUMA multiprocessor, cluster, etc.) in distributed environments. In particular, distribution transparency supports the data independence principle.

SQL database systems come in two main variants, operational and analytical, which have been around for more than three decades. The two variants are complementary and data from operational databases have to be uploaded and pre-processed first so it can be queried.

Operational database systems have been designed for OLTP workloads. They excel at updating data in real-time and keeping it consistent in the advent of failures and concurrent accesses by means of ACID transactions. But they have a hard time in answering large analytical queries and do not scale well. We can distinguish two kinds of systems with respect to scalability: systems that scale up (typically in a NUMA mainframe) but do not scale out and systems that scale out, but only to a few nodes using a shared-disk architecture. Fig. 2 illustrates the SQL shared-disk architecture, with multiple DB servers accessing the shared disk units. Within a DB server, all SQL functionality is implemented in a tightly coupled fashion. The reason for such tight coupling is to maximize performance, which, in the early days of databases, was the only practical solution. Shared-disk requires disks to be globally accessible by the nodes, which requires a storage area network (SAN) that uses a block-based protocol. Since different servers can access the same block in conflicting update modes, global cache consistency is needed. This is typically achieved using a distributed lock manager, which is complex to implement and introduces significant contention. In the case of OLTP workloads, shared-disk has remained the preferred option as it makes load balancing easy and efficient, but scalability is heavily limited by the distributed locking that causes severe contention.

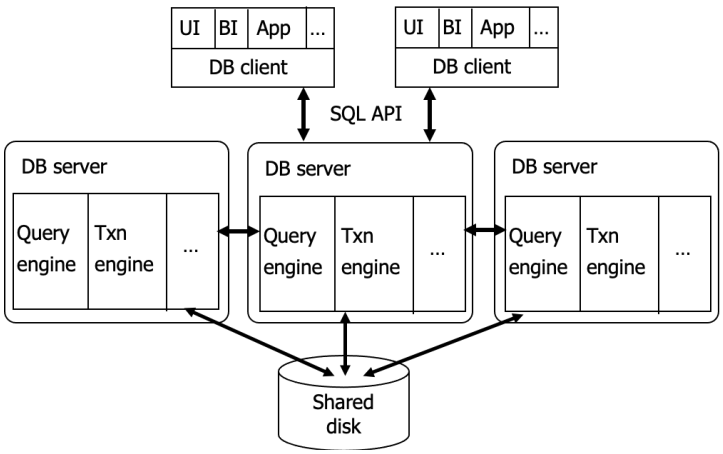

Fig. 2. SQL shared-disk architecture.

Analytical database systems have been designed for data warehouses and OLAP workloads. They exploit parallel processing, in particular, intraquery parallelism, to reduce the response times of large analytical queries in read-intensive applications. Unlike operational database systems that use shared-disk, they use a shared-nothing architecture. The term "shared-nothing" was proposed by Michael Stonebraker [12] to better contrast with two other popular architectures, shared-memory and shared-disk. In the 1980's, shared-nothing was just emerging (with pioneers like Teradata) and shared-memory was the dominant architecture.

A shared-nothing architecture is simply a distributed architecture, i.e., the server nodes do not share either memory or disk and have their own copy of the operating system. Today, shared-nothing clusters are cost-effective as servers can be off-the-shelf components (multicore processor, memory and disk) connected by a low-latency network such as Infiniband or Myrinet. Shared-nothing provides excellent scalability, through scale-out, unlike shared-memory and shared-disk. However, it requires careful partitioning of the data on multiple disk nodes. Furthermore, the addition of new nodes in the system presumably requires reorganizing and repartitioning the database to deal with the load balancing issues. Finally, fault-tolerance is more difficult than with shared-disk as a failed node will make its data on disk unavailable, thus requiring data replication. However, it is for its scalability advantage that shared-nothing has been first adopted for OLAP workloads, as it is easier to parallelize read-only queries. For the same reason, it has been adopted for big data systems, which are typically read-intensive.

4 NoSQL Database Systems

SQL databases, whether operational or analytical, are inefficient for complex data. The rigid schema is the problem. For instance, creating a schema for rows that have optional or structured fields (like a list of items), with a fixed set of columns is difficult, and querying the data even more. Maintainability becomes very hard. Graph data is also difficult to manage. Although it can be modeled in SQL tables, graph-traversal queries may be very time consuming, with repeated and sparse accesses to the database (to mimic graph traversals on tables), which can be very inefficient, if the graph is large.

About a decade ago, SQL database systems were criticized for their "One Size Fits All" approach. Although they have been able to integrate support for all kinds of data (e.g., multimedia objects, documents) and code (e.g., user-defined functions, user-defined oeprators), this approach has resulted in a loss of performance, simplicity, and flexibility for applications with specific, tight performance requirements. Therefore, it has been argued that more specialized database systems are needed.

NoSQL database systems came to address the requirements of web and cloud applications, NoSQL meaning "Not Only SQL" to contrast with the "One Size Fits All" approach of SQL database systems. Another reason that has been used to motivate NoSQL is that supporting strong database consistency as operational database systems do, i.e., through ACID transactions, hurts scalability. Therefore, most NoSQL database systems have relaxed strong database consistency in favor of scalability. An argument

to support this approach has been the famous CAP theorem from distributed systems theory [5], which states that a distributed system with replication can only provide two out of the following three properties: (C) consistency, (A) availability, and (P) partition tolerance. However, the argument is simply wrong as the CAP theorem has nothing to do with database scalability: it is related to replica consistency (i.e., strong mutual consistency) in the presence of network partitioning. Thus, a weaker form of consistency that is typically provided is eventual consistency, which allows replica values to diverge for some time until the update activity ceases so the values eventually become identical (and mutually consistent).

As an alternative to SQL, different NoSQL database systems support different data models and query languages/APIs other than standard SQL. They typically emphasize one or more of the following features: scalability, fault-tolerance, availability, sometimes at the expense of atomicity, strong consistency, flexible schemas, and practical APIs for programming complex data-intensive applications. This is obtained by relaxing features found in SQL database systems such as full SQL support, ACID transactions, in particular, atomicity and consistency, which become weaker (e.g., atomicity only for some operations, eventual consistency) and strong schema support.

There are four main categories of NoSQL database systems based on the underlying data model, i.e., key-value, wide column, document, and graph. Within each category, we can find different variations of the data model, as there is just no standard (unlike the relational data model), and different query languages or APIs. For document database systems, JSON is becoming the de facto standard, as opposed to older, more complex XML. There are also multi-model systems, to combine multiple data models, typically document and graph, in one system.

To provide scalability, NoSQL database systems typically use a scale-out approach in a shared-nothing cluster. They also rely on replication and failover to provide high availability. NoSQL database systems are very diverse as there is just no standard. However, we can illustrate the architecture of key-value database systems that is more uniform and quite popular (see Fig. 3) and the basis for many variations. A major difference with the SQL architecture, which is client-server, is distribution by design to take full advantage of a shared-nothing cluster. In particular, it supports the popular master–worker model for executing parallel tasks, with master nodes coordinating the activities of the worker nodes, by sending them tasks and data and receiving back notifications of tasks done.

This key-value (KV) database system architecture has two loosely-coupled layers: KV engine and data store. The KV engine is accessed by KV clients, with a simple KV API that is specific to each system, but with typical operations such as put, get and scan (enables to perform a simple range query). Such API eases integration with code libraries and applications. The functionality provided by a KV engine corresponds to the low-level part of an SQL system: i.e., operation processing using indexes, replication, fault-tolerance and high availability. But there is no such thing as query engine or transaction engine. The data store is in many cases a distributed, fault-tolerant file system, such as HDFS, and in other cases a custom storage engine running on top of the local file system providing caching facilities and basic operations for reading and updating data.

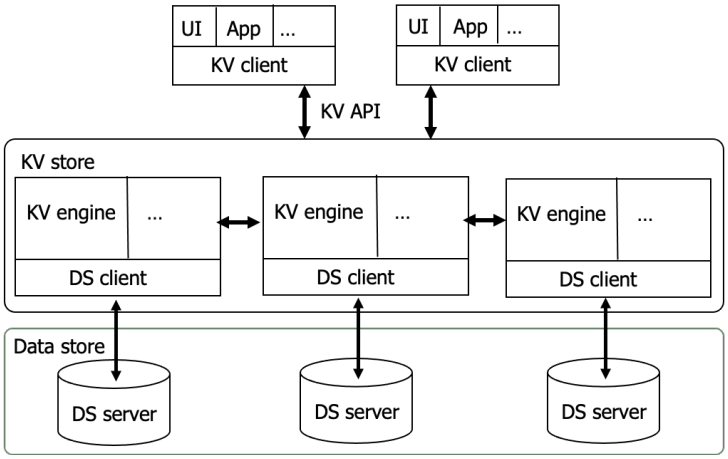

Fig. 3. NoSQL key-value architecture.

Key-value database systems, e.g., DynamoDB and Cassandra, are schemaless so they are totally flexible with the stored data. They are in most cases scalable to large number of nodes. And they are very efficient at ingesting data. Wide column database systems, e.g., BigTable and HBase, enable vertical partitioning of the tables, while providing a key-value interface and sharing many features with key-value database systems. They provide some more functionality such as range scans and the possibility of defining a schema. Document database systems, e.g., MongoDB and CouchBase, use a flexible data representation, such as JSON or XML, and provide an API to access these data or sometimes a query language. Graph database systems, e.g., Neo4J and OrientDB, are specialized in storing and querying graphs efficiently. They are very efficient when the graph database can be centralized at one machine, and replicated at others. But if the database is big and needs to be distributed and partitioned, efficiency is lost.

5 The Case for NewSQL

NewSQL is a new class of database system that seeks to combine the advantages of NoSQL (scalability, availability) and SQL (ACID consistency and usability) systems. The main objective is to address the requirements of enterprise information systems, which have been supported by SQL database systems. But there is also the need to be able to scale out in a shared-nothing cluster in order to support big data, as characterized by the famous 3 big V's (Volume, Velocity, Variety). Examples of popular NewSQL database systems are LeanXcale, Spanner, CockroachDB, EsgynDB, SAP HANA, MemSQL, NuoDB, and Splice Machine. In the rest of this section, we present NewSQL

in more detail, and illustrate with two of the most advanced systems in terms of scalable transaction management: Spanner and LeanXcale.

5.1 NewSQL database systems

NewSQL database systems come in different flavors, each targeted at different workloads and usages. Our taxonomy in [15] is useful to understand NewSQL database systems based on major dimensions including targeted workloads (OLTP, OLAP, HTAP), features and implementation techniques. The main new features are: scalable ACID transactions, in-memory support, polystore, and HTAP support. Each of these features can be mapped into one of the 3 V's of big data. NewSQL database systems typically support more than one feature, but are often characterized by the one feature in which they excel, e.g., scalable ACID transactions. Note that some systems with some of the features have existed long before the term NewSQL was coined, e.g., main memory database systems.

Scalable ACID transactions are provided for big OLTP workloads using a distributed transaction manager that can scale out to large numbers of nodes. ACID properties of transactions are fully provided, with isolation levels such as serializability or snapshot isolation. The few systems that have solved this hard problem are LeanXcale [9] and Spanner [3], through horizontal scalability in a shared-nothing cluster typically on top of a key-value data store.

Scaling transaction execution to achieve high transaction throughput in a distributed or parallel system has been a topic of interest for a long time. In recent years solutions have started to emerge; we discuss two approaches in the next sections as embodied in the Spanner and LeanXcale database systems. Both of them implement the ACID properties in a scalable and composable manner. In both approaches, a new technique is proposed to serialize transactions that can support very high throughput rates (millions or even a billion transactions-per-second). Both approaches have a way to timestamp the commit of transactions and use this commit timestamp to serialize transactions. Spanner uses real time to timestamp transactions, while LeanXcale uses logical time. Real time has the advantage that it does not require any communication, but requires high accuracy and a highly reliable realtime infrastructure. The idea is to use real time as timestamp and wait for the accuracy to elapse to make the result visible to transactions. LeanXcale adopts an approach in which transactions are timestamped with a logical time and committed transactions are made visible progressively as gaps in the serialization order are filled by newly committed transactions. Logical time avoids having to rely on creating a real-time infrastructure.

Main memory database systems, such as MemDB, SAP HANA and VoltDB, keep all data in memory, which enables processing queries at the speed of memory without being limited by the I/O bandwidth and latency. They come in two flavors, analytical and operational. The main memory analytical database systems are very fast. But the main limitation is that the entire database, the auxiliary data, and all intermediate results must fit in memory.

Polystores, also called multistore systems [2], provide integrated access to a number of different data stores, such as HDFS files, SQL and NoSQL databases, through one or more query languages. They are typically restricted to read-only queries, as supporting distributed transactions across heterogeneous data stores is a hard problem. They are reminiscent of multidatabase systems (see Database Integration Chapter in [11]), using a query engine with the capability of querying different data stores in realtime, without storing the data themselves. However, they can support data sources with much more heterogeneity, e.g., key-value, document and graph data. Polystores come in different flavors, depending on the level of coupling with the underlying data stores: loosely-coupled (as web data integration systems), tightly-coupled (typically in the context of data lakes), and hybrid.

HTAP (Hybrid Transactional Analytical Processing) systems are able to perform OLAP and OLTP on the same data. HTAP allows performing real-time analysis on operational data, thus avoiding the traditional separation between operational database and data warehouse and the complexity of dealing with ETLs. HTAP systems exploit multiversioning to avoid conflicts between long analytical queries and updates. However, multiversioning by itself is not enough, in particular, to make both OLAP and OLTP efficient, which is very hard.

NewSQL database systems can have one or more of these features and/or technologies, e.g., scalable operational, HTAP and main memory, or for the most advanced systems, HTAP with scalable transactions and polystore capabilities. The architecture of the most advanced NewSQL database systems is interesting, typically using a key-value store to provide a first level of scalability (see Fig. 4). This approach can provide the ability of the NewSQL database system to support a key-value API, in addition to plain SQL, in order to allow for fast and simple programmatic access to rows. Although appealing, this approach makes it complex for application developers since the key-value store typically does not know that the stored data is relational, how it is organized and how metadata is kept. Atop a key-value store layer, the SQL engine can also be made scalable. Thus, we can have a fully distributed architecture, as shown below, where the three layers (key-value store, transaction engine and SQL query engine) can scale out independently in a shared-nothing cluster.

Such a distributed architecture provides many opportunities to perform optimal resource allocation to the workload (OLAP vs OLTP). To scale out query processing, traditional parallel techniques from SQL analytical database systems are reused. However, within such an architecture, the SQL query engine should exploit the key-value data store, by pushing down within execution plans the operations that can be more efficiently performed close to the data, e.g., filter and aggregate. Scaling out transaction management is much harder and very few NewSQL database systems have been able to achieve it.

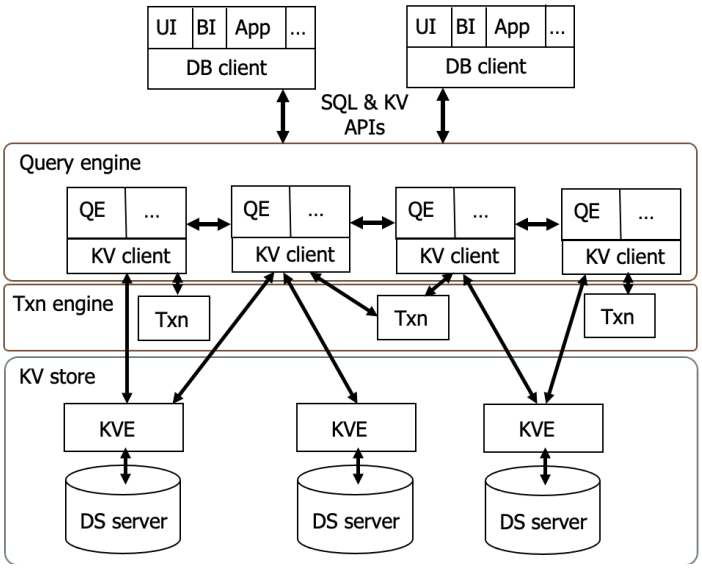

Fig. 4. NewSQL distributed architecture.

5.2 Spanner

Spanner [3] is an advanced NewSQL database system with full SQL and scalable ACID transactions. It was initially developed by Google to support the e-advertising AdWords application, which is update intensive with millions of suppliers monitoring their maximum cost-per-click bids.

Spanner uses traditional locking and 2PC and provides serializability as isolation level. Since locking results in high contention between large queries and update transactions, Spanner also implements multiversioning, which avoids the conflicts between reads and writes and thus the contention stemming from these conflicts. However, multiversioning does not provide scalability. In order to avoid the bottleneck of centralized certification, updated data items are assigned timestamps (using real time) upon commit. For this purpose, Spanner implements an internal service called TrueTime that provides the current time and its current accuracy. In order to make the TrueTime service reliable and accurate, it uses both atomic clocks and GPS since they have different failures modes that can compensate each other. For instance, atomic clocks have a continuous drift, while GPS loses accuracy in some meteorological conditions, when the antenna gets broken, etc. The current time obtained through TrueTime is used to timestamp transactions that are going to be committed. The reported accuracy is used to compensate during timestamp assignment: after obtaining the local time, there is a wait time for the length of the inaccuracy, typically around 10 milliseconds. To deal with deadlocks, Spanner adopts deadlock avoidance using a wound-and-wait approach

thereby eliminating the bottleneck of deadlock detection. Storage management in Spanner is made scalable by leveraging Bigtable, a wide column data store.

Multiversioning is implemented as follows. Private versions of the data items are kept at each site until commitment. Upon commit, the 2PC protocol is started during which buffered writes are propagated to each participant. Each participant sets locks on the updated data items. Once all locks have been acquired, it assigns a commit timestamp larger than any previously assigned timestamp. The coordinator also acquires the write locks. Upon acquiring all write locks and receiving the prepared message from all participants, the coordinator chooses a timestamp larger than the current time plus the inaccuracy, and bigger than any other timestamps assigned locally. The coordinator waits for the assigned timestamp to pass (waiting for the inaccuracy) and then communicates the commit decision to the client. Using multiversioning, Spanner also implements read-only transactions that read over the snapshot at the current time.

5.3 LeanXcale

LeanXcale is an advanced NewSQL HTAP database system with full SQL and polystore support with scalable ACID transactions. It incorporates three main components: storage engine, query engine and transactional engine – all three distributed and highly scalable (i.e., to hundreds of nodes).

LeanXcale provides full SQL functionality over relational tables with JSON columns. Clients can access LeanXcale with any analytics tool using a JDBC or ODBC driver. An important capability of LeanXcale is polystore access using the scripting mechanism of the CloudMdsQL query language [6, 7]. The data stores that can be accessed range from distributed raw data files (e.g., HDFS through parallel SQL queries) to NoSQL database systems (e.g., MongoDB, where queries can be expressed as JavaScript programs).

The storage engine is a proprietary relational key-value store, KiVi, which allows for efficient horizontal partitioning of tables and indexes, based on the primary key or index key. Each table is stored as a KiVi table, where the key corresponds to the primary key of the LeanXcale table and all the columns are stored as they are in KiVi columns. Indexes are also stored as KiVi tables, where the index keys are mapped to the corresponding primary keys. This model enables high scalability of the storage layer by partitioning tables and indexes across KiVi data nodes. KiVi provides the typical put and get operations of key-value stores as well as all single table operations such as predicate-based selection, aggregation, grouping, and sorting, i.e., any algebraic operator but join. Multitable operations, i.e., joins, are performed by the query engine and any algebraic operator above the join in the query plan. Thus, all algebraic operators below a join are pushed down to the KiVi storage engine where they are executed entirely locally.

The query engine processes OLAP workloads over operational data, so that analytical queries are answered over real-time (fresh) data. The parallel implementation of the query engine follows the single-program multiple data (SPMD) approach, which combines intra-query and intra-operator parallelism. With SPMD, multiple symmetric

workers (threads) on different query instances execute the same query/operator, but each of them deals with different portions of the data.

The query engine optimizes queries using two-step optimization. As queries are received, query plans are broadcast and processed by all workers. For parallel execution, an optimization step is added, which transforms a generated sequential query plan into a parallel one. This transformation involves replacing table scans with parallel table scans, and adding shuffle operators to make sure that, in stateful operators (such as group by, or join), related rows (i.e., rows to be joined or grouped together) are handled by the same worker. Parallel table scans combined with pushed down filtering, aggregation and sorting divide the rows from the base tables among all workers, i.e., each worker will retrieve a disjoint subset of the rows during table scan. This is done by dividing the rows and scheduling the obtained subsets to the different query engine instances. Each worker then processes the rows obtained from subsets scheduled to its query engine instance, exchanging rows with other workers as determined by the shuffle operators added to the query plan. To process joins, the query engine supports two strategies for data exchange (shuffle and broadcast) and various join methods (hash, nested loop, etc.), performed locally at each worker after the data exchange takes place.

The query engine is designed to integrate with arbitrary data stores, where data resides in its natural format and can be retrieved (in parallel) by running specific scripts or declarative queries. This makes it a powerful polystore that can process data from its original format, taking full advantage of both expressive scripting and massive parallelism. Moreover, joins across any native datasets, such as HDFS or MongoDB, including LeanXcale tables, can be applied, exploiting efficient parallel join algorithms [8]. To enable ad-hoc querying of an arbitrary data set, the query engine processes queries in the CloudMdsQL query language, where scripts are wrapped as native subqueries.

LeanXcale scales out transactional management [9] by decomposing the ACID properties and scaling each of them independently but in a composable manner. First, it uses logical time to timestamp transactions and to set visibility over committed data. Second, it provides snapshot isolation. Third, all the functions that are intensive in resource usage such as concurrency control, logging, storage and query processing, are fully distributed and parallel, without any coordination. The transactional engine provides strong consistency with snapshot isolation. Thus, reads are not blocked by writes, using multiversion concurrency control. The distributed algorithm for providing transactional consistency is able to commit transactions fully in parallel without any coordination by making a smart separation of concerns, i.e., the visibility of the committed data is separated from the commit processing. In this way, commit processing can adopt a fully parallel approach without compromising consistency that is regulated by the visibility of the committed updates. Thus, commits happen in parallel, and whenever there is a longer prefix of committed transactions without gaps the current snapshot is advanced to that point. For logical time, LeanXcale uses two services: the commit sequencer and the snapshot server. The commit sequencer distributes commit timestamps and the snapshot server regulates the visibility of committed data by advancing the snapshot visible to transactions.

In addition to scalability, in the context of the cloud, LeanXcale supports non-intrusive elasticity and can move data partitions without affecting transaction processing.

This is based on an algorithm that guarantees snapshot isolation across data partitions being moved without actually affecting the processing over the data partitions.

6 Conclusion

In this paper, we made the case for NewSQL, which is enjoying a fast growth in the database system and BI markets in the context of big data. NewSQL combines the scalability and availability of NoSQL with the consistency and usability of SQL. NewSQL systems come in different flavors, each targeted at different workloads and usages. However, they all rely on the principles of distributed database systems, in particular, data partitioning, query processing, replication and transaction management, which have stood the test of time.

We characterized NewSQL systems by their main new features, which are: scalable ACID transactions, main memory support, polystore, and HTAP support. NewSQL database systems typically support more than one feature, but are often characterized by the one feature in which they excel, e.g., scalable ACID transactions.

By blending capabilities only available in different kinds of database systems such as fast data ingestion and SQL queries and by providing online analytics over operational data, NewSQL opens up new opportunities in many application domains where real-time decision is critical. Important use cases are IT performance monitoring, proximity marketing, e-advertising (e.g., Google AdWords), risk monitoring, real-time pricing, real-time fraud detection, IoT, etc. Before NewSQL, these applications could be developed, but using multiple systems and typically at a very high price. NewSQL may also simplify data management, by removing the traditional separation between NoSQL and SQL (ingest data fast, query it with SQL), as well as between operational database and data warehouse / data lake (no more ETLs!).

A hard problem is scaling out transactions in mixed operational and analytical (HTAP) workloads over big data, possibly coming from different data stores (HDFS, SQL, NoSQL). Today, only a few NewSQL systems have solved this problem, e.g., Spanner and LeanXcale, which we described in more details. Both of them implement the ACID properties in a scalable and composable manner, with a new technique to serialize transactions that can support very high throughput rates (millions or even a billion transactions-per-second). Both approaches have a way to timestamp the commit of transactions and use this commit timestamp to serialize transactions. Spanner uses real time to timestamp transactions, while LeanXcale uses logical time, which avoids having to rely on a complex real-time infrastructure. Furthermore, in addition to scalable transactions, LeanXcale provides polytore support, to access multiple data stores such as HDFS and NoSQL, and non-intrusive elasticity.

NewSQL is relatively new, and there is much room for research. Examples of research directions include: JSON and SQL integration, to seamlessly access both relational and JSON data, and thus combine the advantages of SQL and NoSQL with one database system; streaming SQL to combine data streams and NewSQL data; integration with popular big data frameworks such as Spark to perform analytics and machine learning; and defining specific NewSQL HTAP benchmarks.

References

1. Agrawal, D., Das, S., El Abbadi, A.: Data Management in the Cloud: Challenges and Opportunities. Synthesis Lectures on Data Management. Morgan & Claypool Publishers (2012).
2. Bondiombouy, C. and Valduriez, P.: Query processing in multistore systems: an overview. International Journal of Cloud Computing, 5(4):309–346 (2016).
3. Corbett, J.C., et al.: Spanner: Google's Globally-Distributed Database. Proc. USENIX Symposium on Operating Systems Design and Implementation, pp 251–264 (2012).
4. DeWitt, D. and Gray, J.: Parallel Database Systems: the future of high-performance database systems. Communications of the ACM, 35(6):85–98 (1992).
5. Gilbert, S., Lynch, N.A.: Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. SIGACT News, 33(2): 51-59 (2002).
6. Kolev, B., Valduriez, P., Bondiombouy, C., Jimenez-Peris, R., Pau, R., Pereira, J.: CloudMdsQL: Querying Heterogeneous Cloud Data Stores with a Common Language. Distributed and Parallel Databases, 34(4): 463–503 (2016).
7. Kolev, B., Valduriez, P., Bondiombouy, C., Jimenez-Peris, R., Pau, R., Pereira, J.: The CloudMdsQL Multistore System. Proc. ACM SIGMOD Conference, pp. 2113–2116 (2016).
8. Kolev, B., Levchenko, O., Pacitti, E., Valduriez, P., Vilaça, R., Gonçalves, R., Jimenez-Peris, R., Kranas, P.: Parallel Polyglot Query Processing on Heterogeneous Cloud Data Stores with LeanXcale. Proc. IEEE BigData Conference, pp. 1757–1766 (2018).
9. Jimenez-Peris, R. and Patiño-Martinez, M.: System and Method for Highly Scalable Decentralized and Low Contention Transactional Processing. European Patent #EP2780832, US Patent #US9,760,597 (2011).
10. Özsu, M. T. and Valduriez, P.: Principles of Distributed Database Systems, 1st Edition, Prentice-Hall (1991).
11. Özsu, M. T. and Valduriez, P.: Principles of Distributed Database Systems, 4th Edition, Springer (2020).
12. Stonebraker, M.: The Case for Shared Nothing, IEEE Database Engineering Bulletin 9(1): 4-9 (1986).
13. Stonebraker, M., Weisberg, A.: The VoltDB Main Memory DBMS. IEEE Data Eng. Bull. 36(2): 21-27 (2013).
14. Valduriez, P.: Principles of Distributed Data Management in 2020? Proc. International Conference on Databases and Expert Systems Applications, pp.1–11 (2011).
15. Valduriez, P. and Jimenez-Peris, R.: NewSQL: principles, systems and current trends. Tutorial, IEEE Big Data Conference, https://www.leanxcale.com/scientific-articles (2019).