Capítulo

2

What Comes After NoSql? NewSql: A New Era of Challenges in DBMS Scalable Data Processing

José Maria Monteiro, Angelo Brayner, Júlio Alcântara Tavares

Abstract

For decades, relational databases have dominated the scene of large system data storage. As a counterpart, in the last years, new classes of DBMS paradigm has raised, whose one of the main features is to offer complementary characteristics to relational model and to transaction ACID properties. One of these paradigms is called NewSQL, which currently represents the next step after the raise of NoSQL technology. NewSQL main idea is to present similar NoSQL performance characteristics and addressing one of the main weaknesses found in NoSQL era: to preserve ACID characteristics. NewSQL technology can be characterized by the fact of (i) behaving as main memory system and (ii) presenting novel core DBMS mechanisms, such as specific concurrency control protocols. NewSQL also allows the use of SQL language to handle data processing, which it is essential to maintain compatibility with existing applications. In this work, we focus on NewSQL DBMS, due to its plethora of research opportunities and its wide acceptance by the industry, despite representing a recent but promising paradigm.

Resumo

Durante décadas, os bancos de dados relacionais dominaram o cenário de armazenamento de dados em sistemas de informação com grandes volumes de dados. Em contrapartida, nos últimos anos, novos paradigmas surgiram, visando oferecer características complementares ao modelo relacional e às propriedades ACID. Um desses paradigmas é chamado NewSQL, que representa o próximo passo após o uso das tecnologias NoSQL. A principal ideia do paradigma NewSQL é apresentar as características de desempenho e escalabilidade semelhantes às do NoSQL, porém preservando as propriedades ACID. As tecnologias NewSQL podem ser caracterizadas essencialmente por (i) utilizar tecnologias de armazenamento em memória principal e (ii) apresentar mecanismos inovadores

nos principais componentes dos SGBDs, tais como protocolos específicos para controle de concorrência. As tecnologias NewSQL também permitem a utilização da linguagem SQL para manipulação de dados, o que é essencial para manter a compatibilidade com as aplicações existentes. O foco deste trabalho está em descrever tecnologias associadas com o paradigma NewSQL, devido às oportunidades de pesquisa e a sua ampla aceitação por parte da mercado, representando um paradigma recente, mas extremamente promissor.

2.1. Introduction

Over the last decades, relational database systems (RDBMS) brought efficient solutions to provide data management in complex computer systems. Such systems are usually accessed by a huge amount of users, which brings up a complex scenario for data processing and transaction management. Nonetheless, new massive data-centric applications have emerged, such as: social networks, Internet of Things (IOT) among others. In this new context, some aspects of data management in relational databases needs a paradigm shift [Moniruzzaman 2014] [Pavlo and Aslett 2016]. The main reasons for such requirement are the complexity of managing RDBMS core components and the fact that relational databases may not be a feasible choice for very large distributed systems, due to its restriction in scaling horizontally.

Recently, a great number of new data store systems have been designed to provide quite efficient horizontal scalability for simple read/write database operations on distributed servers [Cattell 2011]. These systems are referred to as “NoSQL”. The definition of NoSQL, which stands for “Not Only SQL” or “Not Relational”, is not entirely according to. Nevertheless, NoSQL systems share six key features:

1. The ability to horizontally scale I/O operation (read/write) throughput distributed on several servers;
2. The ability to replicate and to distribute data over a high number of servers (data partition);
3. Support of a simple call level interface or protocol (instead of the SQL language);
4. The use of more permissive concurrency control mechanisms than the ACID transaction model, implemented by most RDBMS;
5. Efficient approaches of distributed indexes and the use of main memory to store data, and
6. The ability to dynamically add new attributes to data records (schemeless).

In order to deliver horizontal scaling, NoSQL implements a "shared nothing" strategy for replicating and partitioning data on many servers. This characteristic supports the execution of a large number of simple read/write operations per second. On the other hand, NoSQL systems generally do not provide ACID transactional properties. Thus, updates are eventually propagated and there are limited guarantees on data consistency.

On the other hand, NewSQL technology emerged as a promising alternative to manage large amounts of data and to support analytics on data in real time, trying to offer the best of both worlds: to preserve ACID properties, one of the main weakness of NoSQL systems, and to reach scalable performance. NewSQL acronym has been used for the first time by Matthew Aslett, in his research about the new challenges faced by conventional DBMS technologies [Gurevich 2015]. As stated before, the key idea behind the concept of NewSQL database is to present scalable NoSQL performance, but avoiding ACID idiosyncrasies presented by conventional database systems.

In this sense, this chapter presents and discusses characteristics and properties of how NewSQL DBMSs handle data in scalable and distributed environments. In other words, as major objectives, this course aims at to:

1. Present the evolution of the main databases and data processing paradigms, over the past decades;
2. Show which databases techniques have been used to improve data management in NewSQL paradigm;
3. Discuss how NewSQL technologies can provide efficient data access and query processing in scalable and distributed DBMSs environments;
4. Discuss and compare NewSQL, NoSQL and Relational DBMSs behavior, regarding the most critical characteristics presented by ACID properties;
5. Show practical examples and benchmarks, using the most important free and commercial (Relational, NoSQL, NewSQL) DBMSs;

2.2. Background

This section provides a brief background, including Big Data challenges, Database Transaction Models, Transaction Isolation Levels, ACID (Atomicity, Consistency, Isolation, and Durability) properties and BASE (Basic Availability, Soft state, and Eventual consistency) and the CAP Theorem (also known as Brewer's theorem).

2.2.1. The Big Data Scenario

In the last decade, the industry, in nearly every sector of the economy, has moved to a data-driven world. There has been an explosion in data volume, which, in turn, induced the need for richer and more flexible data storing and processing systems. The database technology is currently at an unprecedented and exciting inflection point. On one hand, the need for data storing and processing systems has never been higher than the current level, on the other hand the number of new data management products that are available has exploded over the recent years [Floratou et al. 2012].

For the last four decades, data management typically meant relational database management systems (RDBMSs). However, RDBMSs are no longer the only viable alternative for data-driven applications. For instance, consider applications in interactive data-serving environments, where consumer-facing artifacts must be computed on-the-fly from a database. Examples of applications in this category include social networks

and multi-player games. In the past, the standard way to run such applications was to use an RDBMS for the data management component. Nowadays, NoSQL data storage technology is an attractive alternative to RDBMS. NoSQL data store mechanism is often designed to have a simpler data model (in contrast to the relational data model), and is designed to work seamlessly in cluster environments. Many of NoSQL systems have built-in "sharding" or partitioning primitives, which split large data sets across multiple nodes and keep the shards balanced as new records and/or nodes are added to the system [Floratou et al. 2012].

This new application domain is largely characterized by queries that read or update a very small amount of the entire dataset. Besides, in this context read are much more frequent than the write operations. Thus, one can think of this class of applications as the “new OLTP” domain, bearing resemblance to the traditional OLTP world in which the workload largely consists of short “bullet” queries [Floratou et al. 2012].

On the other hand, some big data applications resemble On-line Analytical Processing (OLAP) systems, which are characterized by complex queries on massive amounts of data. The need for these analytical decision support systems has been growing rapidly as well. It is important to emphasize that some data-warehouse-oriented DBMSs came out in the 2000s, for instance, Greenplum, Vertica and Aster Data). Such DBMSs should not be considered NoSQL data stores. OLAP DBMSs are focused on executing complex read-only queries (i.e., aggregations, multiway joins, etc). Big data application are characterized by executing read-write transactions that (i) are short-lived, (ii) touch a small subset of data using index lookups (i.e., no full table scans or large distributed joins), and (iii) are repetitive (i.e., executing the same queries with different inputs) [Pavlo and Aslett 2016]. In this context, Parallel RDBMSs were largely the only solution for such class of applications in the past, but now they face competition from another new class of NoSQL systems - namely, systems based on the MapReduce paradigm. MapReduce-based NoSQL data stores are tailored for large-scale analytics, and are designed specifically to run on clusters of commodity hardware. So, they assume that hardware/software failures are common, and incorporate mechanisms to deal with such failures. These NoSQL data stores typically also scale easily when adding or removing nodes to an operational cluster [Floratou et al. 2012].

2.2.2. Database Transaction Model

A database consists of a collection of objects representing entities of the real world. The set of values of all objects stored in a database at a particular moment in time is called database state. The real world imposes some restrictions on its entities. Additionally, databases must capture such restrictions, called consistency constraints. If the values of objects of a particular database state satisfy all the consistency constraints, the database state is said to be consistent.

A transaction is an abstraction which represents a finite sequence of reads and writes operations on database objects. We use the notation $r_i(x)$ and $w_i(x)$ to represent a read and write operation by a transaction T_i on object x. $OP(T_i)$ denotes the set of all operations executed by T_i . We will assume that the execution of a transaction preserves the database consistency, if this transaction runs entirely and in isolation from other trans-

actions.

Inherently a transaction is characterized by four properties (commonly referred as ACID):

1. Atomicity: all of the operations in the transaction will complete, or none will.

2. Consistency: transactions never observe or result in inconsistent data.

3. Isolation: the transaction will behave as if it is the only operation being performed.

4. Durability: upon completion of the transaction, the operation will not be reversed, but will be persistent

The concurrent execution of a set of transactions is carried out by the interleaving of the database operations of the various transactions. Some interleavings may produce inconsistent database states. Hence, it is necessary to define when an execution of concurrent transactions is correctly interleaved. Henceforward we will call correctly interleaved execution: correct execution. A schedule or history models the execution of concurrent transactions.

Two operations of different transactions conflict (or they are in conflict) if and only if they access the same object of the database and at least one of them is a write operation. The notation $p <_S q$ indicates that operation p was executed before operation q in global schedule S. Let S be a schedule on a set $T = \{T_1, T_2, ..., T_n\}$ of transactions. The serialization graph for S, represented by $SG(S)$, is defined as the directed graph $SG(S) = (N, E)$ in which each node in N corresponds to a transaction in T. The set E contains the edges in the form $T_i \rightarrow T_j$, if and only if $Ti, Tj \in N$ and there are two operations $p \in OP(T_i)$, $q \in OP(T_j)$, where p conflicts with q and $p <_S q$. A schedule S is conflict serializable if and only if the serialization graph for S ($SG(S)$) is acyclic. A schedule S is correct if it is serial or conflict serializable [Bernstein et al. 1986], i.e., a correct schedule preserves the ACID properties.

As mentioned, the classical model for concurrency control in DBMSs adopts *serializability* as the correctness criterion for the execution of concurrent transactions. In existing DBMS, serializability is ensured by the two-phase locking (2PL) protocol [Eswaran et al. 1976]. The 2PL protocol implements a locking mechanism which requires that a transaction obtains a lock on a database object before accessing it. A transaction may only obtain a lock if no other transaction holds an incompatible lock (e.g., a lock for a read operation is incompatible with a write lock) on the same object. When a transaction obtains a lock, it is retained until the transaction ends its execution by a commit or an abort operation in case of the rigorous version of 2PL (for short, 2PL-Rigorous).

In a multiversion concurrency control (MVCC) protocol, a write operation of a transaction t_i on database object x, denoted $w_i(x)$, is treated as the generation of a new version for x, let say x'. Thus, the scheduler does not reject read operations on x, while t_i does not execute a commit operation, which validates the new version x' of x.

Locking Isolation Level	Proscribed Phenomena	Read Locks on Data Items and Phantoms	Write Locks on Data Items and Phantoms
Degree 0	none	none	Short write locks
Degree 1 = Locking READ UNCOMMITTED	P0	none	Long write locks
Degree 2 = Locking READ COMMITTED	P0, P1	Short read locks	Long write locks
Locking REPEATABLE READ	P0, P1, P2	Long data-item read locks, Short phantom read locks	Long write locks
Degree 3 = Locking SERIALIZABLE	P0, P1, P2, P3	Long read locks	Long write locks

Table 2.1. Consistency Levels and Locking ANSI-92 Isolation Levels.

2.2.3. Transaction Isolation Levels

An isolation level is the degree in which the execution of a given transaction is isolated from all other concurrent transactions. Real-world applications usually require a compromise between data consistency and concurrency degree. In general, one should specify the less restrictive isolation level for a transaction, since that isolation level does not violate transaction's requirements in order to obtain a potential higher concurrency degree. Thus, a more restrictive isolation level (e. g., serializable) enables a higher data consistency degree (avoiding undesirable phenomena like dirty reads or lost updates) but a lower concurrency degree. On the other hand, by relaxing the isolation level, one could increase transaction concurrency degree (and transaction throughput) for the price of reducing data consistency degree. It is important to note that isolation levels are implemented by a locking mechanism which delays the execution of database operations to ensure a given isolation level. The ANSI/ISO SQL-92 standard defines four isolation levels: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ and SERIALIZABLE.

The concept of isolation levels was first introduced in [Eswaran et al. 1976] under the name Degrees of Consistency (degrees 0, 1, 2 and 3). The goal of that seminal work was to increase concurrency degree by sacrificing the guarantee of perfect isolation, which in turn enforces data consistency. The idea proposed in [Eswaran et al. 1976] set the stage for the ANSI/ISO SQL-92 definitions for isolation levels [ANSI 1992], where the goal was to develop a standard that was implementation-independent. The ANSI/ISO SQL-92 approach was to avoid certain types of undesirable behavior, called phenomena, by applying the following premises: more restrictive consistency levels disallow more phenomena and serializability does not permit any phenomenon. The isolation levels were named READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE; some of these levels were intended to correspond to the degrees of [Eswaran et al. 1976] (Table 2.1). However, these definitions are based on locking schemes and fail to meet the goals of ANSI-SQL w.r.t. implementation-independence. A subsequent paper [Berenson et al. 1995] showed that the definitions provided in [ANSI 1992] were ambiguous and defined a more precise set of phenomena (P0, P1, P2 and P3), which should be avoided. Table 2.1 summarizes the isolation levels as defined in [Berenson et al. 1995] and relates them to a lock-based implementation. The REPEATABLE READ and SERIALIZABLE isolations levels implement 2PL-Rigorous protocol.

The approach presented in [Adya et al. 2000] provides precise and implementation-

Conflicts Name	Description (T_j conflicts on T_i)	Notation in DSG
Directly write-depends	T_i install x_i, and T_j install x's next version	$T_i \overset{WW}{\rightarrow} T_j$
Directly read-depends	T_i install x_i, and T_j reads x_i or T_j performs a predicated-based read, x_i changes the matches of T_j's read, and x_i is the same or an earlier version of x in T_j's read	$T_i \overset{WR}{\rightarrow} T_j$
Directly anti-depends	T_i reads x_i and T_j install x's next version or T_i performs a predicated-based read and T_j overwrites this read	$T_i \overset{RW}{\rightarrow} T_j$

Table 2.2. Definitions of direct conflicts between transactions.

independent definitions for the ANSI SQL isolation levels [ANSI 1992]. These definitions are based on three kinds of read/write conflicts (called direct conflicts): Directly Write-Depends, Directly Read-Depends and Directly Anti-Depends (Table 2.2). That work defines each isolation level in terms of phenomena that must be avoided at each level. However the phenomena are prefixed by "G" to denote the fact that they are general enough to allow locking and optimistic implementations; these phenomena are named G0, G1, G2 and G3 (by analogy with P0, P1, P2 and P3 [ANSI 1992]). Initially, Adya *et al* have proposed four isolation levels: PL-1, PL-2, PL-2.99 and PL-3 (Table 2.3), where PL-1 is more permissive than Gray's Degree 1, PL-2 is more permissive than Degree 2, PL-3 is more permissive than Degree 3 and PL-2.99 generalize "REPEATABLE READ".

To identify the isolation level for a given schedule S Adya *et al* [Adya et al. 2000] introduces the concept of Direct Serialization Graph (DSG). In a DSG write-dependencies are denoted by $T_i \overset{WW}{\rightarrow} T_j$, direct read-dependencies are denoted by $T_i \overset{WR}{\rightarrow} T_j$, and direct anti-dependencies by $T_i \overset{RW}{\rightarrow} T_j$. Table 2.2 reviews the definitions for direct dependencies. The DSG for a given schedule S, denoted DSG(S), is built as follows. Each node in the graph corresponds to a committed transaction and directed edges correspond to different types of direct conflicts. There is a read/write/anti-dependency edge from transaction T_j to transaction T_i if T_j directly read/write/anti-depends on T_i. Then, a schedule S is PL-1 if DSG(S) no contains a directed cycle consisting entirely of write-dependency edges. A schedule S is PL-2 if DSG(S) no contains a directed cycle consisting by write-dependency and/or read-dependency edges. A schedule S is PL-3 if DSG(S) no contains a directed cycle consisting by write-dependency, read-dependency and/or anti-dependency edges. Finally, a schedule S is PL-2.99 if DSG(S) no contains a directed cycle consisting by write-dependency, read-dependency and item-anti-dependency edges.

Level	Phenomena disallowed	Informal Description (T_i can commit only if:)
PL-1	G0	T_i's writes are completely isolated from the writes of other transactions
PL-2	G1	T_i has only read the updates of transactions that have commited by the time T_i commits (along with PL-1 guarantees)
PL-2.99	G1, G2-item	T_i is completely isolated from others transactions with respect to data items and has PL-2 guarantees for predicated-based reads
PL-3	G1, G2	T_i is completely isolated from other transactions, i.e., all operations of T_i are before or after all operations of any other transaction

Table 2.3. Summary of portable ANSI isolation levels.

2.2.4. The CAP Theorem

In 2000, Eric Brewer introduced the idea that there is a fundamental trade-off between consistency, availability, and partition tolerance [Gilbert and Lynch 2012]. This trade-off, which has become known as the CAP Theorem (or Brewer's Theorem), has been widely discussed ever since. Eric Brewer first presented the CAP Theorem in the context of a web service. A web service is implemented by a set of servers, possibly distributed over a set of geographically distant data centers. Clients make requests to the service, which redirects a request to a particular server. When a server receives a request from the service, it sends a response. Notice that such this generic notion of web service can capture a wide variety of applications, such as search engines, e-commerce, on-line music services, or cloud-based data storage [Gilbert and Lynch 2012].

The CAP theorem states that it is impossible for a distributed computer system (or any shared-data system) to simultaneously provide all three of the following guarantees:

1. Consistency: includes all modifications on data that must be visible to all clients once they are committed. At any given point in time, all clients can read the same data.

2. Availability: means that all operations on data, whether read or write, must end with a response within a specified time;

3. Partition Tolerance: means that even in the case of components' failures, operations on the database must continue;

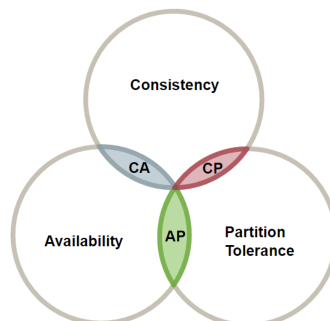

Figure 2.1. CAP Theorem and Related Properties.

Figure 2.1[1] depicts such properties. Theoretically, there are three options for a shared-data distributed system:

1. Forfeit Partition Tolerance: Eric Brewer names 2-Phase-Commit (2PC) as a trait of this option, since 2PC supports temporarily partitions (node crashes, lost messages) by waiting until all messages are received.

2. Forfeit Consistency: In case of partition data can still be used, but since the nodes cannot communicate with each other there is no guarantee that the data is consistent. It implies optimistic locking and inconsistency resolving protocols.

---

[1] Figure available at: http://goo.gl/zmRv0N

3. Forfeit Availability: Data can only be used if its consistency is guaranteed. This implies pessimistic locking, since we need to lock any updated object until the update has been propagated to all nodes. In case of a network partition it might take quite long until the database is in a consistent state again, thus we cannot guarantee high availability anymore.

The option of forfeiting Partition Tolerance is not feasible in cloud and big data environments, since we will always have network partitions. Thus it follows that we need to decide between Availability and Consistency, which can be represented by ACID (Consistency) and BASE (Availability). However, Brewer already recognized that the decision is not binary. The whole spectrum in between is useful; mixing different levels of Availability and Consistency usually yields a better result.

The term BASE [Brewer 2000] stands for "Basically Available", "Soft state" and "Eventual consistency":

1. Basically Available: There will be a response to any request. But, that response could still be 'failure' to obtain the requested data or the data may be in an inconsistent or changing state.
2. Soft state: The state of the system could change over time, so even during times without input there may be changes going on due to "eventual consistency", thus the state of the system is always "soft".
3. Eventual consistency: The system will eventually become consistent once it stops receiving input. The data will propagate to everywhere it should sooner or later, but the system will continue to receive input and is not checking the consistency of every transaction before it moves onto the next one.

So, according to Brewer, the BASE model gives up on ACID property of consistency and isolation in favor of "availability", "graceful degradation" and "performance". Table 2.4 illustrates some differences between ACID and BASE models per Eric Brewer.

Table 2.4. ACID vs BASE

ACID	BASE
Strong consistency for transactions highest priority	Availability and scaling highest priorities
Availability less important	Weak consistency
Pessimistic	Optimistic
Rigorous analysis	Best effort
Complex mechanisms	Simple and fast

2.2.5. Data Consistency in Replicated Database

A replicated database is a special case of distributed database, in which multiple copies of the data items are stored in different servers interconnected by a communication network. The notion of replicated database has been widely used as a strategy to improve data availability and to maximize transaction throughput (number of committed transactions per second). Thus, applications may access data items from any of the replicated servers, i.e., they do not interrupt their executions (waiting for data access), even if some replicated

servers are not available or reachable (through the network). In a replicated database all copies of a given data item should represent the same snapshot of the real world. In other words, all copies of a given data should be in the same database state. That feature is denoted data consistency. The servers, which can update their copies, are called master servers (master sites), to differentiate from those whose copies (replicated data) are read-only. If the number of master servers is equal to the number of replicated servers, one has a multi-master (or read-any/write-any) scheme. The servers allow access (read and write) to the replicated data even when they are disconnected.

In conventional replicated databases, data consistency is guaranteed by the following two correctness criteria:

(C1) Let DB be a database with $n > 1$ replicas DB_{R_i}, $0 < i \leq n$, then the concurrent execution of a set T of transactions over k replicas $(1 < k \leq n)$ of DB should be equivalent to a serial execution of the same set T over the same database DB without replication (one-copy database).
(C2) All replicas (copies) of a given data item x will eventually converge to a unique consistent final state, independently of the operations executed on data copies of x.

The first correctness criterion is called one-copy serializability (1SR) while the second one is also known as eventual consistency. In traditional replicated DBMSs, these criteria are guaranteed by a replica control protocol, which can be classified into three groups: Primary-Copy protocols (also called of pessimistic replication), Quorum-Consensus protocols and Available-Copies protocols (also known as optimistic replication) [Bernstein et al. 1986].

Due to potential gains in increasing data availability and transaction throughput, data replication has also been applied to different computing environments, such as: mobile databases, cloud databases, etc. A cloud replicated database is comprised of several servers and clients interconnected through a network. Of course, cloud replicated databases should ensure data consistency as well. However, the 1SR correctness criterion is too restrictive for cloud replicated databases. For that reason, several approaches have been proposed in which replica control protocols use correctness criteria less restrictive than 1SR.

2.3. NoSQL Databases

NoSQL databases vary widely by data model and have some distinct features on its own. The general taxonomy for NoSQL database, based on the data model, includes four categories:

- Key-value Stores;
- Document Stores;
- Column Family Stores;
- Graph Databases;

Figure 2.2[2] depicts some NoSQL DBMSs and it's related category. It is also important to remark that new technologies and NoSQL DBMSs are rising everyday.

Relational	Key/Value	Column Family	Document	Graph
• Windows Azure SQL Database • SQL Server • Oracle • MySQL • SQL Compact • SQLite • Postgres	• Windows Azure Blob Storage • Windows Azure Table Storage • Windows Azure Cache • Redis • Memcached • Riak	• Cassandra • HBase	• MongoDB • RavenDB • CouchDB	• Neo4J

Figure 2.2. NoSQL DBMSs and it's related category.

2.3.1. Key-value Stores

Key-value data stores are sometimes considered the "simplest" form of database. Those data stores are usually schema-less. In key-value data stores, the data is stored in a form of a pair key-value. So, this data model resembles structure similar to Hash Array (or Hash Table) data structure (also known as a map or dictionary). In this context, keys are used as indexes to fetch the data (value) and this makes those data stores much more efficient for data retrieval than traditional RDBMS. Figure 2.3[3] provides an example illustrating the key-value data model.

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Figure 2.3. Key-Value Storage Example.

The key-value data model is very simple and sometimes it is used as a base while more complex data models are implemented on top of it. For instance, the key-value model can be extended to an ordered model that maintains keys in lexicographic order. This extension is very powerful, since it can efficiently process key ranges [Katsov 2012].

It is important to emphasize that usually the key-value databases prefer availability over consistency (in terms of the CAP theorem). Example of key-value databases include Voldemort, Amazon Dynamo DB, Aerospike, Couchbase, Dynamo, FairCom c-treeACE, FoundationDB, HyperDex, MemcacheDB, MUMPS, Oracle NoSQL Database, OrientDB, Redis, Riak, Berkeley DB, among others.

[2]Figure available at: http://goo.gl/hQLZaf
[3]Figure available at: http://goo.gl/Y5yv8q

2.3.2. Document-based Stores

Currently, document-based data stores are probably most popular among other NoSQL types and their popularity keeps growing. These databases store their data in a form of documents. Usually, document-based databases encapsulates "key-value" concepts, while key is the ID of a document and the value is the document itself, which can be retrieved by its ID. besides, different formats can be used as a metadata for document-oriented databases, such as: XML, JSON and some others.

In contrast with the classical RDBMS, where every row follows the schema, in document-oriented databases each document may have a different structure. Additionally, those databases usually provide indexing structures based on the document contents, which is one of the main enhancements of document-based databases over the more basic key-value data stores. Thus, document-based databases provide querying mechanism to query data by the value (document) contents, besides queries based on the "primary key". However, similarly to key-value data stores, document-based databases are less efficient when application requires multiple key transactions.

Figure 2.4[4] illustrates the differences between the relational data model and the document-based data model.

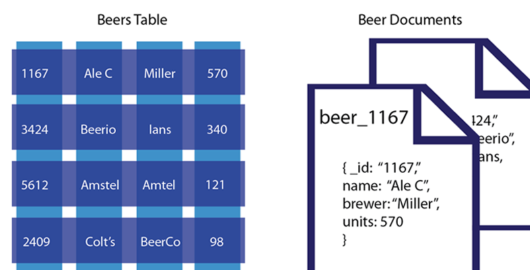

Figure 2.4. Comparing document-oriented and relational data.

Example of document-based databases include Cloudant, Couchbase, CouchDB, DocumentDB, MarkLogic, MongoDB, OrientDB, RethinkDB, SimpleDB, OpenLink Virtuoso, among others.

2.3.3. Column Family Stores

The standard definition of column family is an object which contains columns of related data. The idea of accessing and processing data by column rather than by rows has been used in analytic tools for quite a long time. The term column family means a pair that consists of key and value, where the key is mapped to a value that is a set of columns. In analogy with relational databases, a column family can be considered as a "table", while each key-value pair represent a "row". However, a same "table" (column family) can contain different columns and different number of columns. Besides, there is also a similarity to the key-value model, since the row key functions as a "key", while the set of columns resembles the "value". So, due to the data model specifics column-families usually do not handle complex relational logic. Hence as in case of the key-value databases, if complex relational queries are required, then they had to be implemented in the client side.

[4] Figure available at: http://goo.gl/o5ZssU

Recently, column-family stores gained popularity mainly because of the emergence of the Google Bigtable [Chang et al. 2006]. The Google researchers describe the column-family data model as “sparse, distributed, persistent, multidimensional sorted maps”. In Bigtable the dataset consists of several rows, each can be addressed by a key (primary key). Some of the most popular databases in this category implement the basic Google Bigtable model, e.g. HBase [Apache 2016b]. On the other hand, others column-family databases like Cassandra [Apache 2016a] extend the column-family model by introducing super-columns, where the value can be column-family by itself.

Figure 2.5[5] describes a layout of a single row within column-family and figure 2.6[6] depicts a row within super column-family. Example of column-family databases include InfiniDB, Druid, MonetDB, Apache Arrow, Hypertable, 1010data, Amazon Redshift, Google BigQuery, Endeca, EXASOL, IBM DB2, SAP HANA, Teradata, Vertica, among others.

Row key1	Column Key1	Column Key2	Column Key3	...
	Column Value1	Column Value2	Column Value3	
⋮				

Figure 2.5. Column Family Layout Example.

Row key1	Column Key1	Column Key2	Column Key3	...
	Column Value1	Column Value2	Column Value3	
⋮				

Figure 2.6. Super Column Family Layout Example.

2.3.4. Graph Databases

Graph databases had its origins in the graph theory. Moreover, the graph data model of these NoSQL databases is based on a graph structure. Thus, it means that data is stored in a form of nodes (vertices), while relations between data are presented as edges that interconnect the vertices.

These NoSQL databases do not offer SQL support and the data model is not similar to relational. Nevertheless, many graph databases, including Neo4j [Neo4J 2016], are fully ACID-compliant. It is important to emphasize that graph databases are not an extension of “key-value” concept and they are more efficient for storing interconnected data and handling relational querying. Then, they are naturally more suitable for dependency analysis problem solving and some of the social networking scenarios.

[5] Figure available at: http://goo.gl/szgxnN
[6] Figure available at: http://goo.gl/004qcU

However, while being effective in those areas, graph databases might be less suitable in other Big Data scenarios. For instance, these databases are not efficient on horizontal scaling as key-value or column-families. This “weakness” stems from the fact that if related data is stored on a different servers, than traversing such a graph can be very “expensive” operation in terms of performance. Figure 2.7[7] provides a graphical example of NoSQL Graph Database.

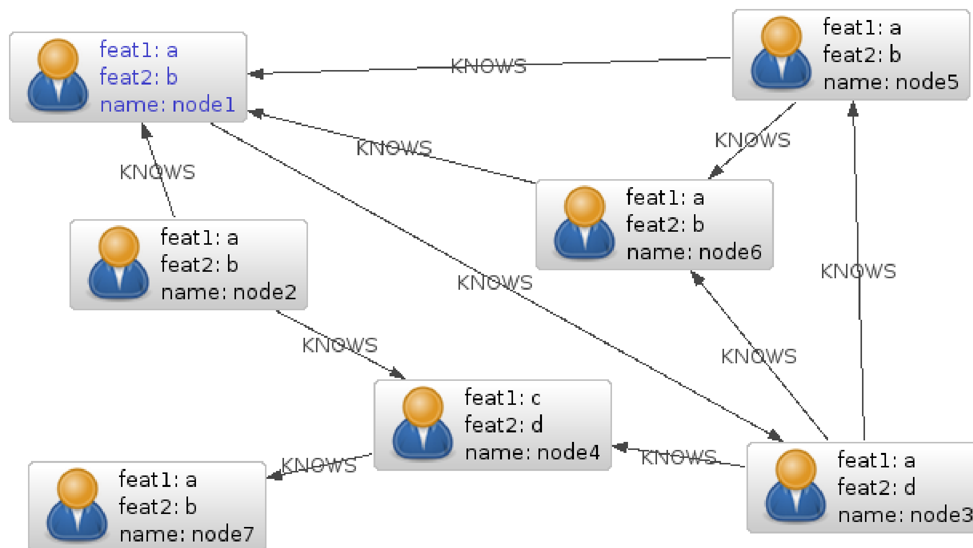

Figure 2.7. Graph Database Relationships Example.

Example of graph databases include Neo4j, AllegroGraph, Oracle Spatial and Graph, Teradata Aster, ArangoDB, Graphbase, HyperGraphDB, VelocityGraph, OrientDB, Blazegraph, GraphDB, MapGraph, VertexDB, among others.

2.4. NewSQL Databases

NewSQL is a class of modern database management systems based on the relational model. Besides NewSQL databases seek to provide the same scalable performance of NoSQL systems for online transaction processing (OLTP) read-write workloads, while still maintaining the ACID properties of a relational database system. However, despite NewSQL databases use SQL as main interface language and provide support for relational concepts such as “tables” and “relations”, their actual internal representation might be absolutely different from that of traditional (relational) database systems [Gurevich 2015]. For example, NuoDB can store its data into key-value store.

The term “NewSQL” was first used by Matt Aslett from “the 451 group”. He wrote: “NewSQL is our shorthand for the various new scalable/high performance SQL database vendors. We have previously referred to these products as “ScalableSQL” to differentiate them from the incumbent relational database products. Since this implies horizontal scalability, which is not necessarily a feature of all the products, we adopted the term “NewSQL” in the new report. And to clarify, like NoSQL, NewSQL is not to be taken too literally: the new thing about the NewSQL vendors is the vendor, not the SQL”.

[7]Figure available at: http://goo.gl/VQAUw4

So, according to Michael Stonebraker, the NewSQL goals are to bring the benefits of the relational databases to distributed store systems or to provide such good performance that horizontal scalability is no longer a need.

Some researchers have argued for a more narrow definition where a NewSQL system's implementation has to use (i) a lock-free concurrency control scheme and (ii) a shared-nothing distributed architecture [Pavlo and Aslett 2016]. All of the DBMSs that were classified as NewSQL by Pavlo and Aslett [Pavlo and Aslett 2016] indeed share these properties and then we agree with this assessment.

Table 2.5 provides a comparison of the main characteristics of Relational (Traditional), NoSQL and NewSQL databases. For instance, it is important to note that only NoSQL databases are considered to be schema-less, meaning that they allow to store unstructured data without prior knowledge of the schema; traditional or NewSQL databases do not allow it.

Table 2.5. The NewSQL Main Features

Feature	Traditional	NoSQL	NewSQL
Relational Model	Yes	No	Yes
SQL Support	Yes	No	Yes
ACID Properties	Yes	No	Yes
Horizontal Scalability	No	Yes	Yes
Performance/Big Volume	No	Yes	Yes
Schema-less	No	Yes	No

In this paper we re exploring the following two NewSQL data stores:

- **VoltDB** - a new RDBMS designed for high performance (per node) as well as scalability. VoltDB is an open-source system (AGPL v3.0 license) written in Java and C++ [VoltDB 2016].
- **NuoDB** - another distributed database which is SQL compliant. It is is defined as "client/cloud relational database". NuoDB is a "closed source" system. It is available in Amazon Web Services (AWS) marketplace as a service, as an appliance and as a stand-alone software [NuoDB 2016b].

2.4.1. Feature Comparison

This section compares VoltDB and NuoDB database by a number of determinative features, such as querying possibilities, concurrency control, replication, scalability, partitioning and consistency.

2.4.1.1. Query Possibilities

Most of the NewSQL databases are providing SQL support as one of their main features. Though, NuoDB is considered to be more SQL-compliant, while VoltDB has various restrictions, for example: it is not possible to use "having" clause, tables can not join

themselves and all joined tables must be partitioned over the same values. In the context of query possibilities, besides the SQL support it is important to analyze the Rest API and MapReduce support [Gurevich 2015].

The REST API is an API (Application Program Interface) that adheres to the principles of REST (Representational State Transfer). REST is an architectural style that uses simple HTTP calls for inter-machine communication. Using REST means API calls will be message-based and reliant on the HTTP standard.

MapReduce is a parallel programming model for large data sets processing introduced by Google, which is typically used to do distributed computing on clusters of computers. In the MapReduce model [Dean and Ghemawat 2008], a user specifies the computation by two functions, denoted Map and Reduce:

- **Map Phase**: In the mapping phase, MapReduce takes the input data and feeds each data element to the mapper.
- **Reduce Phase**: In the reducing phase, the reducer processes all the outputs from the mapper and arrives at a final result. In simple terms, the mapper is meant to filter and transform the input into something that the reducer can aggregate over.

It is important to note that the underlying MapReduce library automatically parallelizes the computation, and handles complex issues like data distribution, load balancing and fault tolerance. The original MapReduce implementation by Google, as well as its open source counterpart, callde Hadoop [Shvachko et al. 2010], is aimed for parallelizing computing in large clusters of commodity machines.

Table 2.6 summarizes querying capabilities of VoltDB and NuoDB.

Table 2.6. Querying Capabilities

NewSQL Database	Rest API	Query Language	Other API	MapReduce Support
VoltDB	Yes	SQL	CLI and API in different languages. JDBC support	No
NuoDB	No	SQL	CLI and drivers for most common data access APIs (JDBC, ODBC, ADO.NET). Also provides a C++ support	No

2.4.1.2. Concurrency Control

Concurrency control is a topic of special interest in NewSQL databases, since they generally need to accommodate a huge amount of users that have access to a data source in parallel. Relational databases management systems (RDBMS) use pessimistic concurrency strategies with exclusive access on a dataset. Pessimistic concurrency control (or pessimistic locking) protocols assume that two or more concurrent users will try to

update the same tuple (object or record) at the same time. To prevent this situation a lock is placed onto the accessed object, so that exclusive access is guaranteed to a single user operation only, other clients accessing the same data will have to wait until the initial one finishes its work. These strategies can be suitable, if costs for locking are low. Nevertheless, in database clusters which are distributed over large geographical distances, pessimistic concurrency control protocols can lead to performance degradation, especially when applications have to support high read request rates.

On the other hand, optimistic concurrency control (or optimistic locking) protocols assume that conflicts are possible, but they are not common. Thus, before commit, every transactions checks whether another transactions have any conflicting modifications on the same data objects. If conflicts are identified, the transaction will be rolled back. This strategy can work well if updates are very rare, and, consequently, chance for conflicting are relatively low. In this situation rolling back will be cheaper that locking data set exclusively as in pessimistic locking protocols [Grolinger et al. 2013].

Several databases, including NuoDB, implement the multi-version concurrency control - MVCC [Bernstein and Goodman 1983]. In this protocol, multiple versions of the same data object are stored, but only the one is marked as current, all the rest are marked as old. Using MVCC, a read operation can see the data as it was when it started reading, while a concurrent process can accomplish write operation on the same data object in the meantime. So, the concurrent access is not managed with locks but by organization of many unmodifiable chronological ordered versions.

It is important to note that MVCC causes higher space requirements as multiple versions are kept in parallel. Thus, usually, to work with MVCC it is necessary to use a garbage collector that deletes no longer needed versions and also some conflict resolving algorithm to deal with inconsistencies. This fact increases the database management system complexity.

Some NewSQL databases also implement innovative approaches to concurrency control. For example, VoltDB assumes that the whole data base can fit into memory, meaning that there is enough memory and transactions are short-lived. Based on these assumptions, transactions are executed sequentially in a lock-free, single-threaded environment. Table 2.7 summarizes the concurrency control mechanisms used in VoltDB and NuoDB databases.

Table 2.7. Concurrency Control

Database	Concurrency Control Mechanism
VoltDB	Single threaded model, no concurrency control
NuoDB	MVCC

2.4.1.3. Replication

Data replication is a important strategy to increases scalability, improves performance though load balancing, besides brings better reliability and fault tolerance. There two types of replication: master-slave and master-master replication. The master-slave replication is a scheme where a single node is defined as master and this is the only node which

accepts and processes write requests. Thus, changes are being propagated from master to the slave nodes.

In the multi-master replication scheme multiple nodes can process write requests, which is then propagated to the remaining nodes. So, basically in master-slave the propagation works in one directions only, from master to slave, which in multi-master replication propagation can occur in various directions.

Figure 2.8[8] and 2.9[9] demonstrates differences between these two types of replication: master-slave and multi-master.

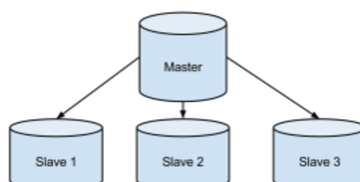

Figure 2.8. Master-Slave Architecture: All writes are performed in the master and are propagated to the slaves node.

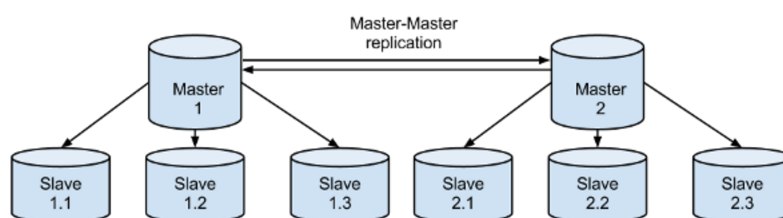

Figure 2.9. Master-Master Architecture: Writes operation cab be performed in master or other nodes.

The replication models in NewSQL databases can be considered as masterless or multi-master, because any node can receive update statements. In NuoDB, the rows (tuples) are internally represented as "in memory" objects which asynchronously communicate to replicate their state changes. VoltDB has a transaction/session manager which is responsible for receiving the updates and forwarding it to all replicas to be executed in parallel. Table 2.8 describes different replication techniques used in NuoDB and VoltDB databases.

Table 2.8. Replication Mechanism

Database	Replication Mechanism
VoltDB	Updates executed on all replicas at the same time
NuoDB	Asynchronous multi-master (distributed object replication)

2.4.1.4. Partitioning

When we are dealing with Big Data, while huge amounts of data and very high read and write request rates exceed the capacity of one single server, databases have to be partitioned across clusters. Traditional relational database management systems (RDBMS)

[8]Figure available at: http://goo.gl/JWwgEB
[9]Figure available at: http://goo.gl/YFpgOP

usually do not support horizontal scalability, because of their normalized data model and ACID properties. Consequently doubling the number of servers in a RDBMS cluster does not double the performance. This problem is one of the reasons why big Web players like Google, Amazon and Facebook started to develop their own data store systems, which are designed to scale horizontally and therefore satisfy the very high requirements on performance the Web applications.

There are many techniques of data partitioning used by various database systems. Most NewSQL databases implement some kind of horizontal partitioning or sharding, which involves storing sets of tuples (rows) into different segments or shards. The two major strategies related to horizontal partitioning are:

- **Range Partitioning**: this strategy distribute datasets to partitions residing on different servers based on ranges of a partition key. Initially, a routing server splits the whole dataset by the range of their keys. Afterwards, every node is accountable for storing and read/write handling of a specific range of keys. The advantage of this approach is that query by range might become very efficient, since the neighbor keys usually reside on the same node (within the same range). Nevertheless, there are some disadvantages of this approach, such hot spots and load-balancing issues. For instance, since the routing server is responsible for load balancing and key load allocation, the processing of many repetitive tasks will be always concentrated in just one server (or a few servers when the outing server is replicated). Thus, the availability of the whole cluster depends on this specific routing server, if it fails the cluster work is impaired.

- **Consistent Hashing**: this strategy provides higher availability and simpler cluster structure comparing to range partitioning. The consistent hashing works as follows: the dataset is represented as a ring, which is divided into a number of ranges equal to a number of available nodes, and every node, is mapped to a corresponding point on the ring. In order to determine the node where the data object should be placed, the system calculates the hash value of the object's key and finds its location on the ring. Next, the ring is walked clockwise until the first node is encountered, and the object gets assigned to that node. Thus, each node is responsible for the ring region between itself and its predecessor.

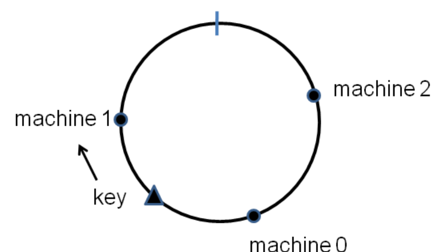

Figure 2.10. Consistent Hashing example, with three nodes.

One of the major advantages of the consistent hashing is that there is no need for a mapping service (routing server) as in range partitioning and the target location of an

object can be calculated very fast. Besides that this approach is also efficient in dynamic resizing: if nodes are added to or removed from the ring, only neighboring regions are to be reassigned to different nodes, and the majority of records remain unaffected. However, the consistent hashing mechanism might have a bad impact on range queries because adjacent keys are distributed across a number of different nodes.

VoltDB uses a consistent hashing approach in which each table is partitioned using a single key and tuples are distributed among nodes. Stored procedures can be executed on a single partition or on all of them, which should be indicated by the user [Grolinger et al. 2013]. Figure 2.10 depicts a consistent hashing example, with three nodes. Each node will be responsible for a data range.

On the other hand, NuoDB uses a completely different approach for data partitioning. NuoDB uses a number of Storage Managers (SM) and Transaction Managers (TM) nodes. The SMs are the nodes responsible for storing the data, while the TMs are the nodes that process the queries. Each SM has a complete copy of the entire data, which means that no partitioning takes place within the SM. However, the underlying key-value store used by the SMs can partition the data by itself, although this is neither controllable nor viewable by the user [NuoDB 2016a].

Table 2.9 describes different partitioning techniques used in NuoDB and VoltDB databases.

Table 2.9. Partitioning Mechanism

Database	Partitioning Mechanism
VoltDB	Consistent hashing. Users define whether stored procedures should run on a single server or on all servers
NuoDB	No partition. The underlying keyvalue store can partition the data, but it is not visible by the user

2.4.1.5. Consistency

Consistency is closely related to the partitioning implementation discussed previously. Usually, there are 2 major types of consistency:

- **Strong Consistency**: ensures that when all the write requests are confirmed, the same data is visible to all the subsequent read operations. In general, strong (immediate) consistency can be obtained by synchronous replication or a complete lack of replication. However, these two options may introduce a large latency and impacts availability, or has a bad effect on scalability.
- **Eventual Consistency**: in this model the changes propagate to the replicas given sufficient time. Then, some nodes may be outdated (inconsistent) for some period of time. Asynchronous replication will lead to eventual consistency, because there is a lag between write confirmation and change propagation.

Table 2.10 summarizes the consistency approach used in VoltDB and NuoDB data stores. VoltDB is strongly consistent and fully transactional. On the other hand, NuoDB uses asynchronous replication and therefore can be defined as eventually consistent data store.

Table 2.10. Consistency Type

Database	Consystence Type
VoltDB	Strong Consistency
NuoDB	Eventual Consistency

2.4.1.6. Implementation and Architecture

Pavlo and Aslett [Pavlo and Aslett 2016] classified the NewSQL data stores in three categories, according to their implementation and architectural aspects:

- Novel systems that are built from the ground-up using a new architecture;
- Middleware that re-implement the same sharding infrastructure that was developed in the 2000s by Google and others, and
- Database-as-a-service offerings from cloud computing providers that are also based on new architectures.

The "new architectures" category contains the NewSQL data stores built from scratch. That is, rather than extending an existing DBMS (e.g., Microsoft's Hekaton for SQL Server), they are designed from a new codebase without any of the architectural baggage of legacy systems. The data stores in this category are based on distributed architectures that operate on shared-nothing resources and contain components to support multi-node concurrency control, fault tolerance through replication, flow control, and distributed query processing. The advantage of using a entiry new data store that is built for distributed execution is that all parts of the system can be optimized for multi-node environments. This includes different aspects like the query optimizer and communication protocol between nodes. The majority of data stores in this category manages their own primary storage, either in-memory or on disk. This is an important aspect because it allows the data store to "send the query to the data" rather than "bring the data to the query", which results in significantly less network traffic since transmitting the queries is typically less network traffic than having to transmit data to the computation. Besides, it enables the data store to employ more sophisticated replication mechanisms. In general, it allows these data stores to achieve better performance than other systems that are layered on top of other existing technologies (e.g., "SQL on Hadoop" systems). Examples of data stores in this category include: VoltDB, NuoDB, Clustrix, CockroachDB, Google Spanner, H-Store, HyPer, MemSQL and SAP HANA.

The "transparent sharding middleware" category includes products that provide the same kind of sharding middleware that eBay, Google, Facebook, and other companies developed in the 2000s. These allow an organization to split a dataset into multiple shards that are stored across a cluster of single-node DBMS instances. In these data stores, the centralized middleware component routes queries, coordinates transactions, as well as manages data placement, replication, and partitioning across the nodes. The key advantage of using a "transparent sharding middleware" is that they are often a drop-in replacement for an application that is already using an existing single-node DBMS. Thus, developers do not need to make any changes on their application in order to use the new

sharded data store. Examples of data stores in this category include: AgilData Scalable Cluster, MariaDB MaxScale, ScaleArc and ScaleBase.

Finally, there are cloud computing providers that offer NewSQL database-as-a-service (DBaaS) products. So, with these services, companies do not have to maintain the data store on either their own private hardware or on a cloud-hosted virtual machine (VM). Instead, the DBaaS provider is responsible for maintaining the physical configuration of the data store, including system tuning, replication and backups. The customer uses a connection URL to the data store, along with a dashboard or API, to control the system. In this model, DBaaS customers pay according to their expected application's resource utilization. Examples of data stores in this category include: Amazon Aurora and ClearDB.

Table 2.11 illustrates the main features of the NewSQL databases according to the three categories proposed by Pavlo and Aslett [Pavlo and Aslett 2016], which considers their implementation and architectural aspects.

2.5. Case Study

This section describes some practical aspects of VoltDB and NuoDB, used by development teams. As stated before, these two database management systems are evidenced as full-fledged NewSQL DBMSs and, besides providing a downloadable version, both are supported by commercial initiatives. With distinct characteristics, they are designed to support data-intensive/data analytics applications, deal with scalability constraints and allow businesses to achieve increasing competitive advantages. Next subsections depicts scripts used to configure and install these databases.

2.5.1. VoltDB NewSQL Database

This section discuss some practical examples related to the VoltDB operating system (OS) environment configuration, DBMS installation and some use examples.

To install VoltDB, Linux CentOS 7 x64 has been chosen due to it's compatibility with VoltDB installation package and related configurations. The following installation example assumes that Linux installation has already been performed. After installation/-configuration (see figures 2.11 and 2.12) and database startup (see figure 2.13), it is possible to perform SQL commands in the VoltDB command line interface (see Figure 2.14).

1. **VoltDB Installation Steps:**
    (a) Download the latest VoltDB version available in tar.gz format, at VoltDB site. It is necessary to register personal information to download the installation package. The default download version comes with VoltDB Server, command-line tools, VoltDB Management Center, Java Client Driver and Java Examples.
    (b) Once downloaded, it is necessary to extract and move the installation package to the definitive installation folder. In a production server, another security requirements may be revised in order to avoid any security leaks.
    (c) It's necessary to export the PATH environment variable do include VoltDB binaries, as stated in figure 2.11.

Table 2.11. Implementation and Architectural Aspects

Product	Category	Released	Main Memory	Summary
VoltDB	New Archit.	2008	Yes	Single-threaded execution engines per partition. Supports streaming operators.
NuoDB	New Archit.	2013	Yes	Split architecture with multiple in-memory executor nodes and a single shared storage node.
Clustrix	New Archit.	2006	No	MySQL-compatible DBMS that supports shared-nothing.
CockroachDB	New Archit.	2014	No	Built on top of distributed key-value store.
Google Spanner	New Archit.	2012	No	WAN-replicated, shared-nothing DBMS that uses special hardware for timestamp generation.
H-Store	New Archit.	2007	Yes	Single-threaded execution engines per partition. Optimized for stored procedures.
HyPer	New Archit.	2010	Yes	HTAP DBMS that uses query compilation and memory efficient indexes.
MemSQL	New Archit.	2012	Yes	Distributed, shared-nothing DBMS using compiled queries. Supports MySQL wire protocol.
SAP HANA	New Archit.	2010	Yes	Hybrid storage (rows + cols). Amalgamation of previous TREX, P*TIME, and MaxDB systems
AgilData	Middleware	2007	No	Shared-nothing database sharding over singlenode MySQL instances.
MariaDB MaxScale	Middleware	2015	No	Query router that supports custom SQL rewriting. Relies on MySQL Cluster for coordination.
ScaleArc	Middleware	2009	No	Rule-based query router for MySQL, SQL Server and Oracle.
Aurora	DBaaS	2014	No	Custom log-structured MySQL engine for RDS.
ClearDB	DBaaS	2010	No	Centralized router that mirrors a single-node MySQL instance in multiple data centers.

(d) To start VoltDB DBMS processes, just run the *"voltdb create"* command in linux shell.

```
# Download Installation Package
[admin@localhost Downloads]$ pwd
/home/admin/Downloads

[admin@localhost Downloads]$ ls -lrth
total 55M
-rw-rw-r--. 1 admin admin 55M Abr  1 12:44 voltdb-ent-6.1.tar.gz

# Unpack Installation Package
[root@localhost ~]# tar -zxvf voltdb-ent-6.1.tar.gz -C $HOME/

# Export PATH Environment Variable with "bin" folder
[root@localhost voltdb-ent-6.1]# PATH=$PATH:/root/voltdb-ent-6.1/
bin/
[root@localhost voltdb-ent-6.1]# export PATH
[root@localhost voltdb-ent-6.1]# echo $PATH
/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/
root/bin:/root/voltdb-ent-6.1/bin
```

Figure 2.11. Linux Shell - VoltDB Installation Package and PATH OS Environment Configuration.

```
# Huge Pages Configuration
[root@localhost voltdb-ent-6.1]# voltdb create
ERROR: The kernel is configured to use transparent huge pages (
THP). This is not supported when running VoltDB. Use the
following commands to disable this feature for the current


# Run the following commands to disable Transparent Huge Pages
[root@localhost voltdb-ent-6.1]# sudo bash -c "echo never >
/sys/kernel/mm/transparent_hugepage/enabled"
[root@localhost voltdb-ent-6.1]# sudo bash -c "echo never >
/sys/kernel/mm/transparent_hugepage/defrag"
```

Figure 2.12. Linux Shell - Configuring OS environment and Disable Transparent Huge Pages.

i. In Linux CentOS 7, the default kernel configuration uses transparent huge pages (THP). This is not supported when running VoltDB. In order to solve this issue, it is necessary to disable this feature running the commands shown in Figure 2.12.

ii. Since the previous configuration has been executed, now it's time to startup VoltDB DBMS processes. To startup the database, just call *voltdb create* command in linux shell. If every configuration above has been performed, a simillar screen to Figure 2.13 will be shown in linux bash screen.

As stated before, the installation process is very simple. Figures 2.11 and 2.12 depicts the OS environment configuration, disabling TPH (Transparent Huge Pages) running *sudo bash -c "echo never > /sys/kernel/mm/transparent_hugepage/enabled"* and *sudo bash -c "echo never > /sys/kernel/mm/transparent_hugepage/defrag"* commands, also exporting PATH variable.

```
# VoltDB Process Initialization
[root@localhost voltdb-ent-6.1]# voltdb create
Initializing VoltDB...

 _    __      ____  ____  ____
| |  / /___  / / /_/ __ \/ __ )
| | / / __ \/ / __/ / / / __  |
| |/ / /_/ / / /_/ /_/ / /_/ /
|___/\____/_/\__/_____/_____/

--------------------------------

Build: 6.1 voltdb-6.1-0-gb9d339f-local Enterprise Edition
Connecting to VoltDB cluster as the leader...
Host id of this node is: 0
Starting VoltDB with trial license. License expires on abr 18,
2016.
Initializing the database and command logs. This may take a
moment...
WARN: This is not a highly available cluster. K-Safety is set
to 0.
Server completed initialization.


# Execute sqlcmd client console
[root@localhost bin]# ./sqlcmd
SQL Command :: localhost:21212
1>
```

Figure 2.13. Linux Shell - Starting VoltDB DBMS Processes.

```
# Execute sqlcmd client console
[root@localhost bin]# ./sqlcmd
SQL Command :: localhost:21212
1>

2> select count(*) as total from employees;
TOTAL
------
  5018

(1 row(s) affected)

3> select town, state from towns ORDER BY town;
TOWN         STATE
------------ ------
Bay View     OH
Billerica    MA
Buffalo      NY

(3 row(s) affected)
```

Figure 2.14. VoltDB Command Line Client and SQL Examples.

Next, Figure 2.13 shows how to initiate a VoltDB instance using the *voltdb create* command. At this point, the DBMS processes are started and it's possible to connect using some graphical or command line compatible database client. Figure 2.13 also shows a successful database startup, when *Server completed initialization* outputs on linux bash screen. Once started, it's possible to execute SQL queries and administrative tasks using VoltDB default command line client. Figure 2.14 shows this practical example. Finally, it's important to remark that VoltDB is SQL compliant. Thus, it allows the use of SQL commands previously used in legacy applications to handle data storage.

2.5.2. NuoDB NewSQL Database

NuoDB is a distributed cloud database management system with a rich SQL implementation and full support for transactions. It provides in memory data access and is compliant with SQL and ACID properties.

As it's main features, it's possible to cite: Scale-Out Performance, Continuous Availability, Multi-Tenancy and No Knobs Administration [NuoDB 2016c]. NuoDB is also available in Amazon Cloud (AWS) as a service, and can be used within this cloud infrastructure, on demand.

It's architecture is based on a distributed, durable, cache. As a NewSQL database, NuoDB redesign the DBMS components to address high scalability requirements and cloud elasticity, also keeping ACID properties (what makes it a NewSql compliant).

In practical terms, in order to handle consistency and availability requirements, NuoDB works with the *domain* concept. A *domain* is a logical grouping that collaborates and work together to support concurrent access. It distributes data and processing responsibilities among each node, which perform a specific role in this distributed database system. Each instance, in a domain, has specific responsibilities as Broker, Agent, Transaction engine and Storage manager. Figure 2.15[10] depicts NuoDB architecture. In this figure, it's possible to see the aforementioned concepts illustrated.

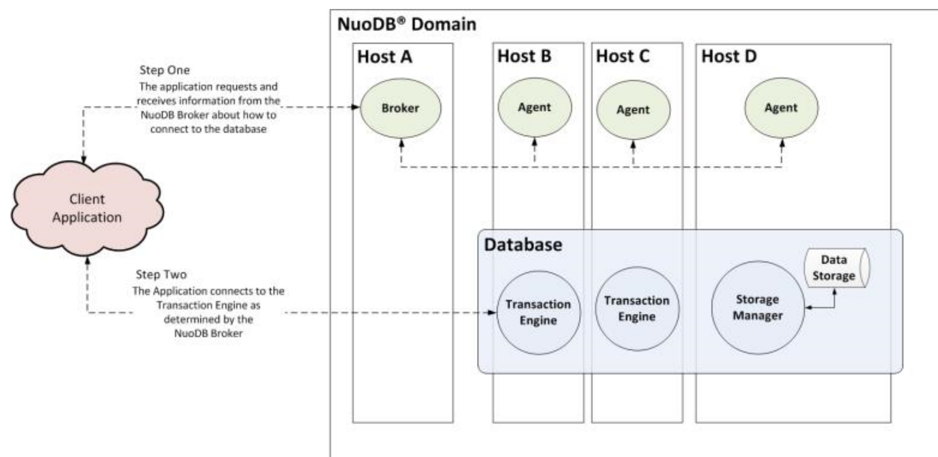

Figure 2.15. NuoDB Architecture.

In order to manage scalability, reliability, and redundancy goals, NuoDB introduces the concept of a database template. A template specifies a Service Level Agreement (SLA) for any databases to which it is applied.

NuoDB's management tier enforces the SLA at all times and automatically manage hosts as necessary to ensure that the template's specification is maintained. Examples of templates are: Single Host, Minimally Redundant, Multi-Host and Geo-Distributed.

[10]Figure Available at: http://goo.gl/aqV7Ea

A template specifies at a logical level how resources in the domain should be used to create and manage an appropriate configuration across the domain's hosts and regions as required to maintain the SLA required for a given database. This feature assists one of it's main characteristics: the *"zero knob configuration"* database.

NuoDB installation process is very simple. Using Linux CentOS 7 x64, the DBMS comes in a RPM package. Thus, the installation step is very simple. Figure 2.16 shows the command necessary to install NuoDB.

```
# NuoDB Install - Using RPM Package Manager
$ sudo rpm --install nuodb-ce.x86_64.rpm

# Execute SQL Commands in NuoDB Console
$ nuosql test@localhost --user dba --password SBBD
SQL>
SQL> USE hockey;
SQL> SHOW tables;
     Tables in schema HOCKEY
        HOCKEY
        PLAYERS
        SCORING
        TEAMS
        VW_PLAYER_STATS is a view

# Perform SQL Commands
SQL> SELECT * FROM teams ORDER BY name;

# Type help to display a list of NuoDB SQL commands:
SQL> HELP
ALTER SEQUENCE       Change definition of a sequence
ALTER TABLE          Change definition of a table
ALTER TRIGGER        Change definition of a trigger
ALTER USER           Change definition of a user
...
Furthermore, you can type HELP followed by a command to get
more information about that command:

# Exit nuosql by typing quit at the SQL> prompt:
SQL> quit
```

Figure 2.16. NuoDB - Install and SQL Client Interface in Linux Shell.

Since NuoDB allows SQL compliance, the use of this query language is allowed as in a relational database. Figure 2.16 also shows SQL commands executed in linux console. Another NuoDB important feature is the use of updatable views, which brings flexibility to the development team. With such characteristics, NuoDB represents a robust NewSQL DBMS and a feasible choice to scale data in cloud and data analytics applications.

2.6. Final Remarks

In the mid- and long-term, NewSQL DBMSs promise to power the next generation of data-intensive applications that rely on fast, smart data and provides fast data ingestion and export with massive scalability and real-time analytics [VoltDB 2016]. From a DBMS perspective, NewSQL databases can provide businesses to gain significant competitive advantage in both back-room and cloud environments, processing a huge amount of data stream in real time.

Regarding NewSQL context and it's main characteristics, some authors advocate that quite often OLTP databases can fit within main memory [Harizopoulos et al. 2008]. Of course, such feature may positively impact DBMS components to achieve high performance. Nonetheless, this is not the case of OLTPs running on very large databases, neither OLAP processing (e.g. very large data warehouses). Furthermore, for main memory databases, at least the log file has to reside in non-volatile memory and the recovery process may take too much time, which is not reasonable for many OLTP applications at all [Fang et al. 2011].

It is important to highlight that ACID and CAP properties have a huge impact on the most important core database management system components, such as the recovery mechanisms and concurrency control protocols. Thus, novel techniques and algorithms designed specifically for NewSQL DBMSs should consider the dependency between the core database components.

In near future, the design of novel techniques and algorithms according to NewSQL characteristics is a critical goal. Many related studies advances towards a NewSQL era, which strengthens the premisse that a new database architecture may be essential to provide even better performance on DBMS data processing [Moniruzzaman 2014] [Doshi et al. 2013] [Grolinger et al. 2013] [Corbett et al. 2013] [Ãetintemel et al. 2014] [Arulraj et al. 2015] [DeBrabant et al. 2013] [Floratou et al. 2012]. Finally, another research direction is to define algorithms to autonomously and dynamically adapt NewSQL characteristics according to workload changes over the time.

Acknowledgments This Research was partially supported by LSBD/UFC, CNPQ - Brazil and Faculdade Metropolitana da Grande Fortaleza - FAMETRO. We acknowledge that this work is a partial result of the Automatic Management of Cloud Databases project supported by CNPq (MCTI/CNPq 14/2014 - Universal) under grant number 446090/2014-0.

References

[Adya et al. 2000] Adya, A., Liskov, B., ONeil, B., and ONeil, P. (2000). *Generalized Isolation Level Definitions*. In proceedings of the IEEE International Conference on Data Engineering, San Diego, CA, March.

[ANSI 1992] ANSI (1992). *ANSI X3.135-1992*. ANSI/ISO.

[Apache 2016a] Apache (2016a). Apache cassandra. http://cassandra.apache.org. Accessed at September 2016.

[Apache 2016b] Apache (2016b). Apache hbase. http://wiki.apache.org/hadoop/Hbase. Accessed at September 2016.

[Arulraj et al. 2015] Arulraj, J., Pavlo, A., and Dulloor, S. R. (2015). Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 707–722, New York, NY, USA. ACM.

[Berenson et al. 1995] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. (1995). A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10, New York, NY, USA. ACM.

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

[Bernstein et al. 1986] Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1986). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Brewer 2000] Brewer, E. A. (2000). Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA. ACM.

[Cattell 2011] Cattell, R. (2011). Scalable sql and nosql data stores. *SIGMOD Rec.*, 39:12–27.

[Chang et al. 2006] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2006). Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA. USENIX Association.

[Corbett et al. 2013] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W. C., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. (2013). Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8.

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

[DeBrabant et al. 2013] DeBrabant, J., Pavlo, A., Tu, S., Stonebraker, M., and Zdonik, S. (2013). Anti-caching: A new approach to database management system architecture. *Proc. VLDB Endow.*, 6(14):1942–1953.

[Doshi et al. 2013] Doshi, K. A., Zhong, T., Lu, Z., Tang, X., Lou, T., and Deng, G. (2013). Blending sql and newsql approaches: Reference architectures for enterprise big data challenges. In *Proceedings of the 2013 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, CYBERC '13, pages 163–170, Washington, DC, USA. IEEE Computer Society.

[Eswaran et al. 1976] Eswaran, K. P., Gray, J., Lorie, R. A., and Traiger, I. L. (1976). The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633.

[Fang et al. 2011] Fang, R., Hsiao, H.-I., He, B., Mohan, C., and Wang, Y. (2011). High Performance Database Logging using Storage Class Memory. In *Proceeding of the IEEE International Conference on Data Engineering*, pages 1221–1231, USA.

[Floratou et al. 2012] Floratou, A., Teletia, N., DeWitt, D. J., Patel, J. M., and Zhang, D. (2012). Can the elephants handle the nosql onslaught? *Proc. VLDB Endow.*, 5(12):1712–1723.

[Gilbert and Lynch 2012] Gilbert, S. and Lynch, N. A. (2012). Perspectives on the cap theorem. *IEEE Computer*, 45(2):30–36.

[Grolinger et al. 2013] Grolinger, K., Higashino, W. A., Tiwari, A., and Capretz, M. A. (2013). Data management in cloud environments: Nosql and newsql data stores. *J. Cloud Comput.*, 2(1):49:1–49:24.

[Gurevich 2015] Gurevich, Y. (2015). Comparative survey of nosql/newsql db systems. Master's thesis, Open University of Israel, Computer Science Division.

[Harizopoulos et al. 2008] Harizopoulos, S., Abadi, D. J., Madden, S., and Stonebraker, M. (2008). OLTP through the Looking Glass, and What We Found There. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 981–992, Vancouver, Canada.

[Katsov 2012] Katsov, I. (2012). Nosql data modeling techniques. https://highlyscalable.wordpress.com/2012/03/01/nosql-datamodeling-techniques/. Accessed at September 2016.

[Moniruzzaman 2014] Moniruzzaman, A. B. M. (2014). Newsql: Towards next-generation scalable RDBMS for online transaction processing (OLTP) for big data management. *CoRR*, abs/1411.7343.

[Neo4J 2016] Neo4J (2016). Neo4j graph nosql database. http://neo4j.com/. Accessed at September 2016.

[NuoDB 2016a] NuoDB (2016a). Nuodb technical whitepaper. http://go.nuodb.com/white-paper.html. Accessed at September 2016.

[NuoDB 2016b] NuoDB (2016b). Nuodb: The sql database for the cloud. http://www.nuodb.com. Accessed at September 2016.

[NuoDB 2016c] NuoDB, I. (2016c). Nuodb at a glance. http://doc.nuodb.com/Latest/Default.htmNuoDB-At-a-Glance.htm. Accessed at September 2016.

[Pavlo and Aslett 2016] Pavlo, A. and Aslett, M. (2016). What's really new with newsql? *SIGMOD Rec.*, 45(2):45–55.

[Shvachko et al. 2010] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA. IEEE Computer Society.

[VoltDB 2016] VoltDB (2016). Voltdb: The world's fastest, in-memory operational database. http://www.voltdb.com. Accessed at September 2016.

[Ãetintemel et al. 2014] Ãetintemel, U., Du, J., Kraska, T., Madden, S., Maier, D., Meehan, J., Pavlo, A., Stonebraker, M., Sutherland, E., Tatbul, N., Tufte, K., Wang, H., and Zdonik, S. B. (2014). S-Store: A Streaming NewSQL System for Big Velocity Applications. *PVLDB*, 7(13):1633–1636.