A Survey of Distributed Database Management Systems

Big data has been described as having some or all of the following characteristics: high velocity, heterogeneous structure, extremely large volume, and “complexity in data distribution and synchronization.” As big data became more prominent in the professional world, DBMS’s with a capacity to handle such extreme data became necessary. Several different types of DBMS’s were developed, one of the most prominent being the distributed DBMS, which allows for the replication of data and concurrency of operations across multiple nodes. Distributed systems in general face numerous obstacles during development, including heterogeneity, openness, security, scalability, fault handling, concurrency, and transparency. Following is a list of numerous popular distributed database management systems with their implementation and attempts to tackle those challenges.

The Apache Cassandra Database Management System was developed as an efficient system for handling big data by Google, Amazon, and Facebook, which ultimately open sourced the DBMS to the Apache Foundation in 2009. The result was a highly available, highly reliable, and highly scalable NoSQL alternative to existing RDBMS technologies. Cassandra has been utilized by numerous well-known corporations for which data integrity, security, and availability are essential, including eBay, Netflix, Hewlett Packard, and many others.

The Cassandra DBMS was developed with a peer-to-peer distributed architecture, as opposed to the regularly used master-slave system or “manual and difficult-to-maintain sharded design.” Nodes communicate through a gossip protocol and thus, the absence of a master node allows Cassandra to operate without such a single point of failure that commonly cripples master-slave systems. Cassandra uses the concept of a “ring” to denote a group of nodes across which data is automatically and transparently distributed in either a randomized (default) or ordered manner. After selecting the desired number of copies, Cassandra
transparently handles data replication, which takes place across multiple nodes in the same “ring.” This can be further customized with a variety of options, including the ability to further safeguard data on a physical location basis by choosing to store copies in separate physical racks, separate data centers, and even separate cloud platforms.

To preserve data integrity, the data is preserved in an atomic, isolated, and durable manner. In terms of read-write capabilities, Cassandra is described as “location independent,” allowing for data from any database node to be accessed from any other node in the database. To ensure data integrity, when updated, data is first recorded to a commit log and logged temporarily in a volatile memtable, which is later flushed into the more permanent disk-based sorted strings table. In case of node failure during a write operation, data is written to an alternate node, and when the failed node returns to operation, relevant data is synched with the rest of the system. To request data, the user accesses and requests relevant data from a node, deemed the user’s “coordinator node,” in the system. Concurrently, the data is requested from one or more, in case of failure, nodes with the relevant data. In terms of data consistency, the relevant nodes’ response can be customized from an immediate response to a more gradual approach and can be configured on a “per operation basis,” allowing for greater control. Read-write operations scale linearly as the number of nodes increase (“Introduction to Apache Cassandra”).

The Apache HBase DBMS is a Java-implemented, HDFS-based, open-source, distributed, column-oriented derivation of Google’s BigTable system. The Java-based HBase is a top-layer component of both Apache Hadoop and Apache Zookeeper (Rabl). The Hadoop Distributed File System was developed for efficient storage of extremely large data. It uses a master-slave architecture, composed of one master NameNode (a second NameNode can be configured for failure recovery purposes), which manages the system’s namespace and performs access control, and numerous subservient DataNodes, which manage data stored on the physical machines. Using the file system namespace, HDFS stores data in file structures, which are stored as data blocks on sets of DataNodes. Furthermore, the NameNode handles file and directory access and naming, as well as block dispersal to DataNodes. DataNodes handle data
access and perform block and data manipulation and replication, when prompted to by the NameNode. MapReduce, the primary way HDFS data is analyzed, is able to operate on a large amount of data by performing operations on smaller block segments of the overall data. ("Comparing the Hadoop Distributed File System (HDFS) with the Cassandra File System (CFS).") MapReduce allows HDFS to perform complex operation on large amounts of data, but does not allow for efficient access of individual records (Karnataki).

The HBase DBMS is a NoSQL distributed database designed for use with high volumes of data built on top of HDFS (“Chapter 9. Architecture.”). Similarly to the HDFS, HBase uses a master-slave architecture with a single Master node and numerous Region Servers, each of which are composed of multiple Regions. Regions store data in the form of tables. The region system allows for the reallocation of data for expanding tables, by allowing a table to be stored in numerous regions, increasing its potential size. The Master node handles administrative tasks associated with maintenance of the Cluster and the load across the system, as well as Region allocation to the Region Servers. The Region Servers maintain their assigned Regions, including automatic splitting in response to Region growth, and handle client requests. Since the data in HBase is stored in Tables, unlike the HDFS, the HBase is better suited for accessing individual records than performing operations on large numbers of records (Karnataki). To prevent unintended results, HBase implements concurrency control on write operations by locking accessed structures (i.e. rows, columns) (Chanan).

The Voldemort system is a distributed, highly scalable NoSQL DBMS developed by LinkedIn. Foregoing commonly maintained ACID integrity guidelines, Voldemort was designed as a “distributed, fault-tolerant, persistent hash table.” Voldemort’s data model allows for the automatic and transparent replication of information across nodes and a high degree of horizontal scalability. Voldemort can be easily integrated with other DBMS’s to add persistence through uncomplicated API calls (Rabl).

Voldemort’s architecture is made up of numerous nodes with data partitioned in a way that each node maintains a unique subset of the data, enabling the cluster to expand without reconfiguration of every node. Rather than tables, data is maintained in “stores,” which allow
for more freedom in data type choice (i.e. lists and mappings could be stored as well). Queries are performed with hashtable key-value semantics and thus data is retrieved through the use of primary keys. While not inherently included one-to-many relationships can be established through the use of the list structure type. To improve ease of load estimates and storage transparency, only simple queries are allowed. To reduce load times, rather than preserving data consistency at write time, inconsistencies are allowed until they are resolved at read time through the use of versioning. This inconsistency leads to an increase probability of failed operations. To maintain the state of included data, Voldemort supplies an API for persistence purposes, allowing data to be stored in separate storage entities (“Design”).

The Redis DBMS is a key-value data store designed for use with “in-memory” data and is programmed in ANSI C and can be used with most POSIX systems (Campanelli). Though the data is expected to exist only in short-term memory, data can be “(semi-) persisted” through the use of periodic snapshots of the data or operational logs. Redis supports keys with the string, hash, list, set, and sorted set types, which can be modified through numerous atomic operations. Data can be asynchronously replicated through the use of an extendible master-slave architecture. Through the use of a non-blocking structure, master and slave nodes do not interfere with each other through the replication and synchronization processes, ultimately allowing for a highly scalable architecture (Rabl).

Redis uses an “extendible” master-slave architecture, which allows for numerous master nodes each connected to numerous slave nodes. Slave nodes can also be interconnected. As previously mentioned, replication is non-blocking on both the master and slave level, allowing either to continue working during the synchronization process. The Master node can continue to issue orders to Slave nodes and Slave nodes can respond to user read-only queries with older data copies. As all Slave nodes can respond to read-only queries, replication can be used for both scalability and redundancy purposes. (“Replication”) Redis can be reached through API, applicable with most programming languages. Though there are no inherent concurrency control mechanisms, a Redis implementation can be programmed to use a series of locks when modifying a piece of data (Wagstaff).
The VoltDB DBMS is a RDBMS designed for use with in-memory data and is a derivation of the “research prototype H-Store.” To preserve data integrity, the DBMS complies with the ACID guidelines. VoltDB partitions data across multi-node clusters by assigning disjoint sets of data to each node. VoltDB supports SQL queries and tactically executes them only at the node with the relevant partitioned data, reducing load time. The in-memory nature of the data allows for further temporal efficiency as this eliminates the overhead from I/O and network access (Rabl).

Data is handled in-memory to reduce access times associated with disk seeks. Using a special “high performance stored procedure interface,” VoltDB ensures at most one round trip between the server and the client for each transaction, reducing communication latencies. VoltDB is composed of multiple partitions, “in-memory execution engines,” made up of data and processing structures. VoltDB implements and distributes these partitions to the many CPU cores, each of which is single threaded and contains a process queue to be executed sequentially, of the cluster. Sequential execution helps ensure transactional consistency. VoltDB clusters can be easily scaled, without any changes to the cluster layout, both by increasing the number of nodes and their individual capacities. VoltDB allows for small commonly used tables to be replicated across multiple nodes, allowing for quicker access times. VoltDB provides fault tolerance through three distinct methods: K-safety, Network Fault Detection, and Live Node Rejoin. K-safety allows for the user to specify a number, K, of copies for all partitions. VoltDB, then, transparently replicates the data, which thereafter syncs with each change, and in the event of node failure, the system continues using other copies of that node’s partition. Due to K-safety, in the event of a network disconnection, both parties may be able to continue functioning, leading to possible synchronization errors. Network fault detection enables the DBMS to recognize the hardware failure and chooses a party to continue operation and discretely shuts down the other. Failed nodes can be “rejoined,” syncing relevant partition data from sibling nodes, with the rest of the system without significantly impacting operations. In lieu of possible hardware failures, the DBMS can be configured to periodically record snapshots of the database at selected intervals and to record all inter-snapshot transactions through a process called “command logging,” enabling the system to be
brought back to work in a timely manner. For extremely critical databases, VoltDB offers a 
Database Replication service, through which every change made to the primary DB is made to a 
replica one, so that in the event of extreme hardware failure, the replica DB can be brought 
online. VoltDB can be accessed by numerous programming language interfaces as well as a 
JDBC API ("VoltDB Technical Overview").

The ACID-compliant, open source MySQL DBMS is the most frequently utilized RDBMS in 
the world. It was developed as a quicker and more flexible alternative to the precursor mSQL. 
It was developed with a similar API to mSQL and was ultimately named after co-founder Monty 
Widenius’s daughter, My. It can be implemented as either part of a client-server system or as a 
smaller embedded library linkable to an external application.

MySQL is built upon an architecture with three distinct node types: Storage Nodes, 
Management Server Nodes, and MySQL Server Nodes. Storage nodes maintain the DB data, 
which is replicated across multiple storage nodes. The Management Server Node(s) handle the 
base configuration of the system and are only used at startup, meaning after their initial 
actions, the system can run without them. The MySQL Server nodes act as the gateways 
between the user and the storage node clusters, sending requests for information. All MySQL 
Server nodes are connected to all Storage nodes, allowing for any change to DB data to be 
immediately visible to all MySQL Server nodes. The combined properties of each node type 
contribute to the High Availability of MySQL, allowing for continued function in the event of any 
node type meaning there is no single point of failure. The cluster is designed using a shared-
nothing architecture, meaning disk and memory storage are not shared between nodes. As all 
applications are connected to the MySQL Cluster through the MySQL Server and do not directly 
interact with the nodes, the system maintains data, network, distribution, replication, and 
partition transparency. Each transaction triggers a synchronization of all applicable nodes, 
maintaining data integrity. MySQL detects failures by either registering a communication loss 
between normally connected nodes or through a heartbeat system. In the event of the first 
case, all nodes are informed and carry on with the missing node classified as failed and when 
the node recovers, it is joined as a new node. The heartbeat system works as the nodes are
connected in a “logical circle,” and each node periodically pings the next node in the chain. In the event of three consecutive missed messages from the prior node, the previous node is classified as dead. In the event of node failure, multiple groups of nodes could become isolated and if they each maintain access to the entirety of the DB, both groups could carry on operations leading to possibly conflicting data. To solve this problem MySQL implements a network partitioning paradigm that determines if a set of nodes contains at least half of the original total number of nodes. If so, the set of nodes is allowed to continue operation. If multiple nodes fail concurrently, to allow for their safe restart, the Master Storage Node uses the number of nodes that have deemed each fail to determine fail order. In the event that a Master Storage Node fails, a replacement is immediately instantiated. The failure order is used to determine the order in which the nodes should be restarted. In the event that the whole system is disrupted, MySQL can keep track of the system state through logging and both local and global checkpoints, which can then be used to recover from hardware failure. In the event of a targeted storage node failure during a transaction, MySQL uses a round-robin approach to transparently select a separate node with the relevant data (Ronstrom).

Below is a table describing the challenges each DBMS chose to address during their development.

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Heterogeneity</th>
<th>Openness</th>
<th>Security (Integrity)</th>
<th>Scalability</th>
<th>Fault Handling</th>
<th>Concurrency</th>
<th>Transparency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cassandra</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>HBase</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Voldemort</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Redis</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>VoltDB</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MySQL</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

All of the above DBMS’s tackle the challenges of distributed structures (i.e. heterogeneity, openness, security, scalability, fault handling, concurrency, and transparency) in their own unique way. Some forego certain aspects in an attempt to strengthen separate
areas. The DBMS’s all take into consideration heterogeneity, openness, scalability, and fault handling, but several forego data integrity, concurrency of operations, and transparency.
References


