Distributed Database Systems and Edge/Fog/Cloud Computing

A distributed database system is one in which the data belonging to a single logical database is distributed to two or more physical databases. Beyond that simple definition, there are a confusing number of possibilities for when, how, and why the data is distributed. Some are applicable to edge and/or fog computing, some others are applicable to fog and/or cloud computing, and some are applicable across the entire spectrum of edge, fog and cloud computing.

McObject LLC

33309 1st Way South
Suite A-208
Federal Way, WA 98003

Phone: 425-888-8505

E-mail: info@mcobject.com

www.mcobject.com

Introduction

A distributed database system is one in which the data belonging to a single logical database is distributed to two or more physical databases. Beyond that simple definition, there are a confusing number of possibilities for when, how, and why the data is distributed. Some are applicable to edge and/or fog computing, some others are applicable to fog and/or cloud computing, and some are applicable across the entire spectrum of edge, fog and cloud computing.

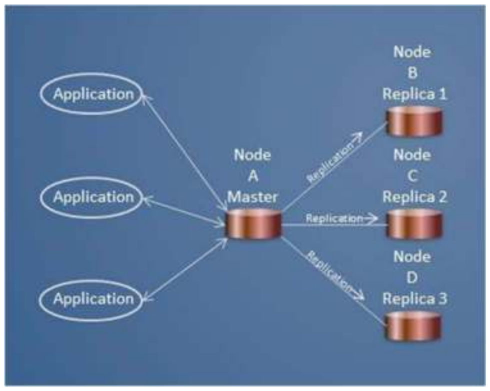
This paper will cover the types of distributed database systems in the context of edge, fog and cloud computing, explain “when, how and why” the data is distributed, and why those details make certain distributed database systems applicable (or not) to specific needs in edge, fog and cloud computing.

Definition

Wikipedia authors have taken a collective stab at defining a distributed database: “A distributed database is a database in which storage devices are not all attached to a common processor. It may be stored in multiple computers, located in the same physical location; or may be dispersed over a network of interconnected computers. Unlike parallel systems, in which the processors are tightly coupled and constitute a single database system, a distributed database system consists of loosely coupled sites that share no physical components.” That definition, itself, derives partly from the Institute for Telecommunications Sciences associated with the U.S. Department of Commerce.

The definition above is actually pretty narrow. There are at least three other use cases that are applicable under the general heading of “distributed database”: High Availability, Cluster Database, and Blockchain. A distributed database à la Wikipedia, High Availability and Cluster are relevant to data management in the Internet of Things.

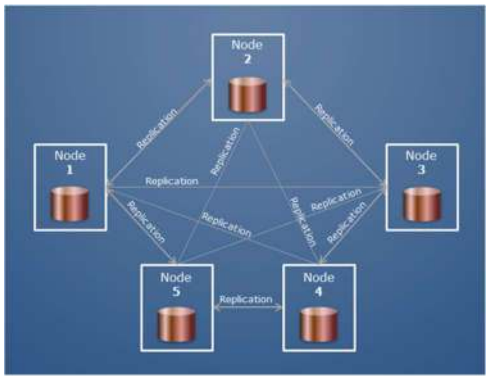
High Availability



For a database system to achieve High Availability (HA), it needs to maintain, in real time, an identical copy of the physical database in a separate hardware instance. In other words, it must keep the copy consistent with the master. In this scenario, there are (at least) two copies of the database that we call the master and the slave(s) (sometimes called replicas). Actions applied to the master database (i.e. insert, update, delete operations,) must be replicated on the slave and the slave must be ready to change its role to master at any time. This is called failover. The master and replica are normally attached to different physical systems, though in telecommunications a common

HA set up consists of multiple boards within a chassis: a master controller board, a standby controller board, and some number of line cards that each serve some protocol (BGP, OSPF, etc.). Here, the master database is maintained by processes on the master controller board. The database system replicates changes to a slave database on the standby controller board, which has identical processes waiting to take over processing in the event that the master controller board fails. In the Internet of Things, High Availability is desirable for mission critical industrial systems, to maintain availability of gateways, and in the cloud to ensure that real-time analytics can continue to execute in the face of hardware failures.

Cluster Database



An eXtremeDB Cluster Database is one in which multiple physical copies of the entire database are kept synchronized. It is different from High Availability in that any physical instance of the database can be modified, and will replicate its modifications to the other database instances within the cluster. This is where the similarities among database cluster implementations end. Broadly speaking, there are two implementation models: ACID and Eventual Consistency. In ACID (Atomic, Consistent, Isolated and Durable) implementations, modifications are replicated synchronously in a two-phase commit protocol to assure that once they are committed, the changes are immediately reflected in every physical instance of the database. In other words, all the database instances are consistent, all the time. With Eventual Consistency, changes are replicated asynchronously, possibly long after the originating node committed the change to the database. This implies some sort of reconciliation process to resolve conflicting changes originated by two or more nodes. With Eventual Consistency, applications must be written to contend with the possibility of having stale data in the physical instance of the database to which they're attached.

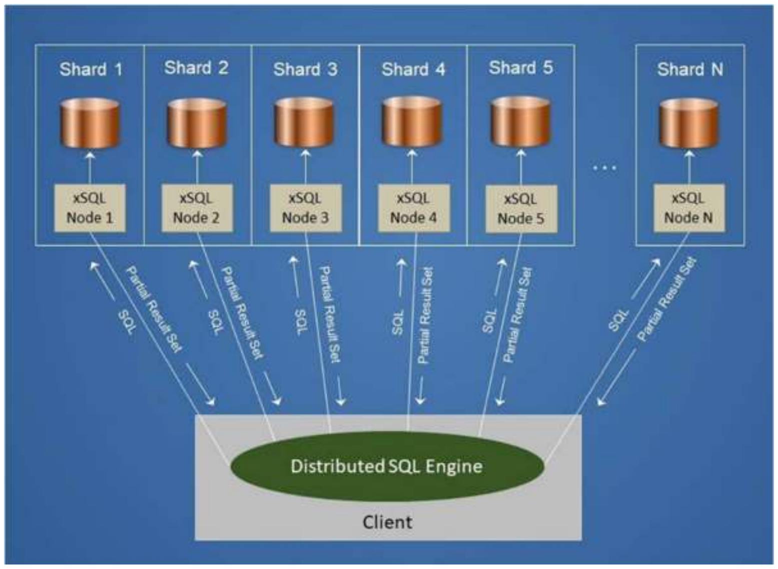
For example, consider a worldwide online bookseller. There may be one copy of a certain book in stock; buyers in New York and Sydney will both see that the book is available, and both can put the book in their shopping cart and check out. The system will have to sort out who really gets the book and whose order is backordered. In this instance users have come to accept this. However, this model would never work for a cellular telephone network needing to verify that a subscriber has a certain service, or has sufficient funds. This type of system requires a consistent database view. Because of the nature of the synchronous replication required for the ACID implementation, horizontal scalability is limited, but the implementation is straightforward (no conflict resolution needed). Scalability of Eventual Consistency implementations is quite high, but so is the complexity. Cluster implementations abound in the Internet of Things. For example, IoT gateways can be clustered for improved scalability and reliability. (See cluster image above.) The number of nodes in each gateway cluster is modest, so both the ACID and Eventual Consistency model are suitable. The cluster can handle more traffic from edge devices than a single gateway would be able to, and reliability/availability is improved (the inherent limits to scalability with the ACID Consistency model does not come in to play in small clusters).

Blockchain

The term "Distributed Database" is often associated with blockchain technologies (BitCoin being the most well-known). It is used synonymously with "Distributed Ledger", which is more apt (in this author's opinion). One problem with using the term distributed database in the context of blockchain technologies is that 'distributed database' implies a distributed database management system, but there is rarely a database management system involved in blockchain. In this case and others, it is important to draw a distinction between a database and a database management system. A blockchain is, in fact, a distributed database. But, as previously mentioned, there is rarely a database management system involved in creating/maintaining the blockchain distributed ledger.

Partitioned Database

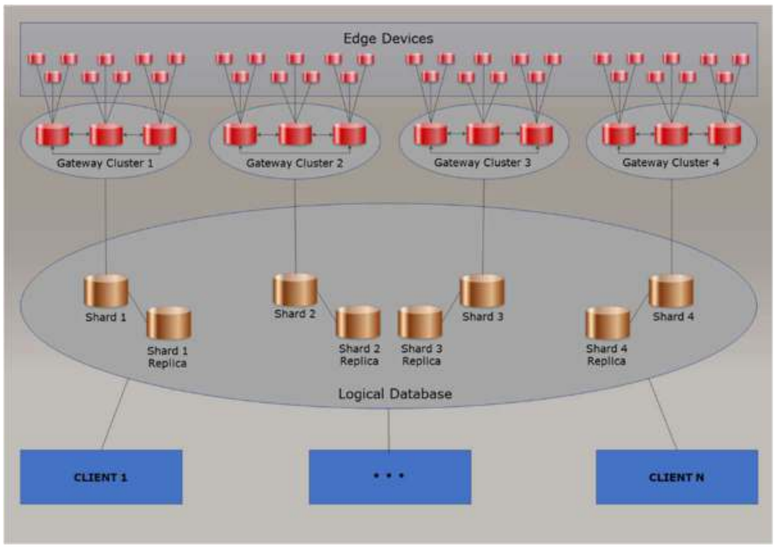
The distributed (partitioned) database topology implied by the Wikipedia definition "...stored in multiple computers, located in the same physical location…" is what is colloquially called sharding a database. The key difference between sharding versus HA and Cluster distributed databases is that each physical database instance (shard) houses just a fraction of all the data. All the shards together represent a single logical database, which is manifested in many physical shards. Unlike Wikipedia, this paper will not differentiate between 'distributed' and 'parallel' database systems. Logically, the purpose is the same: scalability. Whether the shards are distributed across servers, or partitioned on a single server to leverage multiple CPUs or CPU cores is immaterial. Further, in all cases, processing is parallel. How the shards are physically distributed is an unimportant artifact.



For example, in the STAC-M3 published benchmarks conducted by McObject since 2012, we've utilized single servers with 24 cores, creating 72 shards, and we've utilized 4 to 6 servers each with 16 to 22 cores, creating 64 to 128 shards. In all cases, the goal is to saturate the I/O channels to get data into the CPU cores for processing. While STAC-M3 is a capital markets (tick database) benchmark, the principles apply equally to the Internet of Things' Big Data analytics. IoT data is overwhelmingly time series data, (e.g. sensor measurements,) just like a tick database is time series data.

Sharding a database implies support for distributed query processing. Each shard is managed by its own instance of a database server. Since each shard/server represents some fraction of the whole logical database, the potential exists that a query result returned by any shard is just a partial result set and needs to be merged with the partial result sets of every other shard/server and only then be presented to the client application as a complete result set. If the data is distributed among the shards in the most optimal way, then all of the data for a given query can be found on a single shard, and the query can be distributed to the specific server instance that manages that shard. Often, both approaches must be supported.

For example, consider a large smart-building IoT deployment that spans multiple campuses, each with multiple buildings. Property management might choose to distribute (shard) information about each campus across multiple physical databases. If they want to calculate some metric for a specific building (e.g. power consumption in 15-minute windows,) they only need query the shard that contains the data for that building. But, if they want to calculate the same metric for multiple buildings and/or across campuses, then they need to distribute that query to many shards/servers, and this is where the parallelism comes in to play. Each server instance works on its portion of the problem in parallel with every other server instance.



Database sharding also supports vertical scalability (i.e. being able to store 10s or 100s of terabytes, petabytes and beyond). To create a single 100 terabyte logical database, a developer can create 50 instances of 2 terabyte physical databases. Distributed database systems often support "elastic" scalability, allowing for the addition of shards, which could also mean adding servers to the distributed system. This will make the system scalable in both the vertical and horizontal dimensions.

Vertical and horizontal scalability is essential for large IoT systems that generate very high volumes of data. Vertical scalability is necessary to handle the ever-growing volume of data, and horizontal scalability is necessary to maintain the ability for timely processing/analytics of the data as it grows from 1TB to 10TB to 100TB to petabytes and beyond.

Conclusion

In summary, the term distributed database encompasses three different database system arrangements for three distinct purposes. High Availability database systems distribute a master database to one or more replicas for the express purpose of preserving the availability of the system in the face of failure. Cluster database systems distribute a database for massive/global scalability (Eventual Consistency) or for cooperative computing among a relatively small number of nodes (ACID). Finally, sharding partitions a logical database into multiple shards to facilitate parallel processing and horizontal scalability. All capabilities are integral to the deployment of scalable and reliable IoT systems.