Tampere University

Nayeong Song

REDESIGN TACTILON AGNET DATABASE IN DISTRIBUTED ENVIRONMENT

Master's thesis
Faculty of Information Technology and Communication Sciences (ITC)
Examiners: Maarit Harsu
Luis Zuniga-Mayorga
Apr 2022

ABSTRACT

Nayeong Song: Redesign Tactilon Agnet database in distributed environment
Master's thesis
Tampere University
Master's Degree Programme in Computing Sciences
Apr 2022

As the amount of data that businesses manage is surging, many companies face the problem of data management. Understanding the big data has become an important challenge for businesses. Big data means not only data itself, but also making data usable and manageable after all. Management of the data typically refers to not only storing data itself, but also making sure that data is not lost and highly available.

In this thesis, we identify the problems of the existing database being used in the application named Tactilon Agnet and giving new solutions on it. Tactilon Agnet is an end-to-end encrypted messaging protocols communications solution developed by Airbus Defence and Space. The focus of this thesis is to redesign the existing structure to more highly available and redundant.

We implemented InnoDB Cluster as a solution and the results show that data loss can be prevented in case of database server failure. However, there are still improvements left behind to remove any single point of failure and make our database highly available. As a testing purpose, all servers are configured in local computer in our development, but the actual implementation can be configured on different locations.

Keywords: Relational Database, MySQL, Distributed Database, InnoDB Cluster

PREFACE

This thesis was written in Helsinki, Finland between December 2021 and April 2022. Thesis was commissioned by the Airbus Defence and Space Oy, which is located in Helsinki. Thesis was designed to analyze the existing database and give a new solution for the database being used at Airbus. Luis Zuniga-Mayorga, who is a software Architect at Airbus helped me set up a database and give a lot of technical support during the thesis process, and university lecturer Maarit Harsu gave great advise on the overall structure and contents of the thesis. This thesis would not have been possible without the support from them, and I would like to show my gratitude to my supervisors.

Helsinki, 6.4.2022

LIST OF FIGURES

LIST OF TABLES

LIST OF PROGRAMS

LIST OF ABBREVIATIONS AND SYMBOLS

DB	Database
DBMS	Database management system
GCS	Group communication system
CSV	Comma-separated values
RTO	Recovery time objective
RPO	Recovery point objective
HA	High availability
IPV	IP virtual server
VRRP	Virtual redundancy routing protocol
IAM	Identity and access management
AZ	Availability zone

CONTENTS

1 INTRODUCTION

Many of today's businesses rely on database systems. As the size of the data grows, the importance of database design and optimization plays an important role. Managing a database includes not only storing data but also making sure that data is safe and not lost. Database outage is not uncommon, and many companies suffered database outages that caused significant damage to their business. Outages are not always planned, and although companies cannot prevent them fully, it is important to have a sustainable database design to prevent them as much as we can. To achieve this, it is important to make the database fault-tolerant regardless of disasters, server disconnection, etc. The distributed database ensures that data is stored in multiple places and increases the availability of the data, so distributed computing becomes a more popular option in many companies these days. However, implementing a distributed database design requires additional work and introduces complexity, so many large enterprises today use centralized databases so that data can be available in one place which facilitates better decision making.

In this thesis, I am trying to implement a new distributed database solution for the existing centralized database being used at Airbus Defence and Space. The existing database structure has only one backend server and one centralized database server where all data resides in a single site. A centralized database is easy to maintain in the sense that there is a single point to access the data, but this is not fault-tolerant and increases the risk of data loss in case of disaster. Disaster includes any downtime or significant data loss regardless of the cause. The cost of disaster can significantly vary depending on the scale of it, and in some cases, it may be enough to cause a company to go out of business. Organizations should prepare an appropriate disaster recovery plan to minimize downtime. However, is 100% uptime possible? Unfortunately achieving 100% uptime is practically not possible, and the costs for running a database typically increase as an enterprise tries to lower the downtime (Figure 2.1). [22]

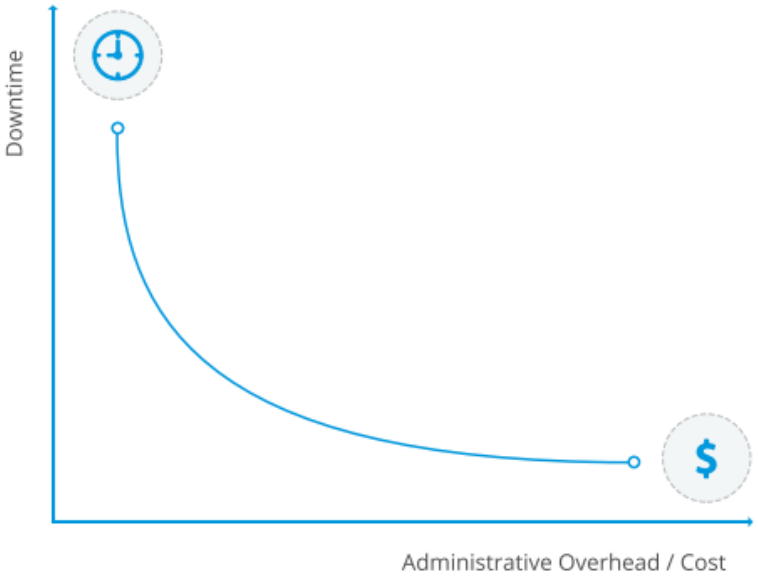

Figure 1.1 *Downtime and costs relationship [22]*

Outages are inevitable, and it is important to prevent and prepare for the recover when it is needed. With regards to the database, different mechanisms can be implemented as a part of disaster recovery plan in order to prepare for an outage. In this thesis, I propose possible solution(s) to achieve high availability on existing centralized database, and compare them in respect to practicality and reliability.

2 SCOPE

To understand what is the scope of this thesis as well as what is the optimal solution of it, we need to understand the structure of the existing database and what are the potential problems of it. In this section, we first introduce the application, the structure of existing database being used. We identify what are the potential problems of the existing structure and suggest new database structure.

2.1 Tactilon agnet

Today's organizations use a lot of group communication, although the security of applications is often questionable. Our application is Tactilon Agnet, which is a service providing communications solution especially to security-critical organizations. The Agnet dispatch and smart phone application user interfaces were developed and maintained by Airbus Defence and Space. Agnet enables organizations to use their smartphones inside private digital network, and organization members can benefit from secure communications using this.

Figure 2.1 *Tactilon Agnet [9]*

Not only simple message exchanges, but also various forms of data such as voice, video or location can be exchanged with end-to-end encryption. Downloading an

application can be simply done in user's smartphone from app store, and both government and private mobile networks can be operated in this application. [9] Agnet is mission-critical, meaning that failure in the system results in serious impacts on organization's operations and may cause catastrophes. Our database contains various kinds of sensitive data such as user's information or critical messages. Thus, it is important to ensure that all data has backups in multiple places and recover at any time if failure occurs in database.

2.2 Existing database studies and goal

The goal of the thesis is to give a solution to improve existing database structure. Our database management system is MySQL, and it is being updated and retrieved by the backend server. Backend server is connected to our database, which is stored in a single place. Our data can only be accessed from one location, and this structure is called centralized database (Figure 2.2). Backend acts as a client, which is an active program that initiates a request for the services of the server while database server waits for requests coming from the client. There may be one or more clients and servers, but our existing system has one server and client. Since there is a single point to access the data (server), there is no duplication of the data and maintaining is cheaper than distributed database management systems. However, in case of database failure, data loss may happen if there is no fault-tolerant setup.

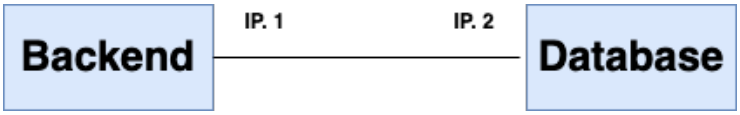

Figure 2.2 *Existing database structure*

To eliminate any single point of failure and prevent data loss, it is important to ensure that backup of the data exist across different physical different locations. Here, location can be either physical location (data center) or network, and achieving this is one of the key principles of high availability of the database. In this thesis, we will try to allocate the replicas of the data across multiple places and this database management system is called distributed database. However, having replicas introduce complexities to our existing database system despite of advantages. For example, whenever our data is updated by the requests from the application, which one of replicas should be updated should be defined. Although we decide which database to update, how should we make sure that other replicas are in same state? All these

questions shall be handled and resolved by giving a possible solution and we identify the defects of our solution as well.

As an initial setup, test dummy data is setup in MySQL as our database management system is MySQL. Dummy data is stored in one server, which means that database is centralized in one place. Database access grant was given by my supervisor, and data was accessed from local computer using VPN. For convenience, logical backups of the original data was made using mysqldump so that we do not need to connect to VPN server for fetching data every time, but the copy of the database will be deleted after this work for security. Once we setup database with dummy data in local computer, we test whether server failure or data recovery is handled in various situations. As a future improvement, connection with our application can be tested.

3 DATABASES

A database is a group of structured information and is typically stored on computer systems. Database Management System (DBMS) is a system which is in charge of controlling a database, and it consists of data and its associated applications. The most common type of database today is a structured database, which includes rows and columns in a set of relations (tables) for efficient processing and querying data. However, there are different types of databases, and which database to use particularly depends on how the data is being used.

3.1 Relational database

In set theory, which is the field of mathematics, a relation means a table with rows (tuples) and columns (attributes). In the relational data model, *table* is used as synonym of *relation*. As of relation, there must be only one value at the intersection of row and column. Besides, each row should be unique (there are no duplicate rows) and each column's name is unique as well. However, in two or more relations within the same relational database, columns may have the same name which is desirable in some situations.

A term *primary key* means a combination of columns with a value that is a unique identifier for each row. A primary key is important in the sense that every piece in the database should be able to be retrieved. Along with uniqueness, the primary cannot be *null*.

There are two types of tables that work with relational databases: base table, virtual table. *Base table* means the table which exists in the database whereas *virtual table* only exists in main memory and is created during relational operations. Using a virtual table has several benefits. It can enhance the query performance since virtual tables are not saved on a disk but only in the main memory. [12]

3.2 Centralized database

A centralized database is stored in one location and data can be modified and accessed only from a single location. In the centralized database, data, process, and interface components are central. There are several benefits to using centralized database over other kinds of database systems. First, it is more cost-effective as labor and power costs are minimized. Besides, since all data physically reside in a single location, it is easier to maintain integrity and keep the data updated at all times.

Despite their benefits, centralized databases do come with certain limitations. When all requests try to access the same data entity at the same time, the bottleneck may easily occur, and often distributed system is considered to overcome this problem. Additionally, this type of databases are not tolerant to database failure since there is a single point of failure. Data loss is not easy to recover and in most cases, it would have to be done manually. [20]

3.3 Distributed database system

A distributed database is a database where storage devices are not bind to an identical processing unit such as CPU (Central Processing Unit). Data may be stored in the same physical location or spread over a set of computer nodes (loosely coupled system). However, the main objective of the distributed database is that it should look like a centralized database to the end-user. [20] In short, a distributed database is a single logical database that is split into partitions or fragments. Using a distributed database, data can be accessed either locally or globally. Local applications require access to local data in a single site whereas global applications access to data from multiple remotes sites in a distributed system. [14]

Distributed database management system (DBMS) is composed of at least one global application and several local applications. Data in DBMS may be split into several partitions (fragments), and fragments may be replicated in a distributed system. Replicas of fragments are allocated in different sites, in which each site has its way of handling local applications. [21]

Then, why do we use distributed database? Firstly, errors can be kept local rather than the entire relationship being affected. Besides, since queries and updates are largely localized, network bottlenecks can be prevented, unlike centralized databases. Besides, access can be restricted only to each user's portion of the data.

However, although the advantages are remarkable, additional communication introduced between applications and data servers may introduce disadvantages. When the data is fragmented and replicated, replicates are required to have the same value (one copy equivalence). Also in terms of performance, performance may not be significantly improved or be worse if the interaction between sites is heavy. [20]

There are three possible architectures of the distributed database as figure below: shared-nothing, networked architecture with one centralized database, and truly distributed database.

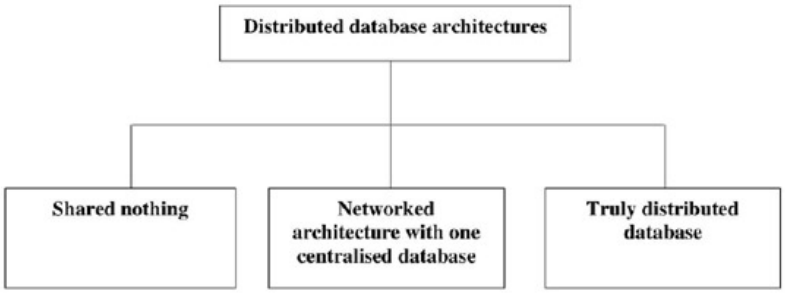

Figure 3.1 *Distributed Database Architectures [13]*

In a shared-nothing architecture, each computer has its database and multiple computers (nodes) are connected to form a distributed database. In this architecture, although computers may share the same network, no computer shares database with other computers. In the networked architecture with one centralized database, all the computers share a single database, so one common database is shared by all the computers in a distributed database system. Finally, each computer has its database in truly distributed database architecture, but all databases are shared, unlike shared-nothing architecture.

4 TYPES OF DISTRIBUTED DATABASE SYSTEMS

4.1 Fragmentation

Fragmentation distributes not only the physical location of the table but also the table itself. Using fragmentation, a table is split into a collection of subsets of the original table, and these members are called fragments of the table. Here, a subset can be a subset of tuples (rows) or attributes (columns). Partitioning tuples is called horizontal fragmentation, and partitioning attributes is called vertical fragmentation. Fragmentation aims to improve reliability, performance, communication costs, and balanced storage capacity.

In horizontal fragmentation, a relation is partitioned into a set of sub-relations, each of which is a subset of tuples (rows) of the original relation. Each fragment has unique rows but all have the same attributes (columns). Horizontal fragmentation is classified into primary and derived horizontal fragmentation.

Figure 4.1 *Horizontal fragmentation of the relation [2]*

In vertical fragmentation, relation is partitioned into separate sets of attributes (columns) except the primary key. Every set must include at least one common column such as primary key so that the original relation is reconstructed using the common column.

Figure 4.2 *Vertical fragmentation of the relation [2]*

Data fragmentation enables to tailor the database to the specific needs of local applications, and overall performance can be improved from the increased locality. However, fragmentation may raise the additional complexity in operations. Complexity depends on the design of the distributed database. In case of disjoint fragmentation (zero redundancy), DDBMS can be more easily handled compared to sophisticated fragmentation in which some subsets of the relations are shared. [8]

When fragmentation is introduced in a database, correctness rules of fragmentation should be followed. Correctness rules ensure no loss and no redundancy of the data. For this, three rules should be followed when designing fragmentation of the database: Completeness, Reconstruction, Disjointness. These rules also make sure that the database undertakes the semantic change during fragmentation.

If the relation instance R is fragmented into $F_R = \{\mathrm{R}_1, \mathrm{R}_2, ..., \mathrm{R}_\mathrm{n}\}$, completeness states that each item in R should be able to be found in one of the fragments $\mathrm{R_i}$. This ensures loseless property of the data. [14] Reconstruction states that there exists a relational operator $\bigtriangledown$ which satisfies

$$R = \bigtriangledown R_i, \forall R_i \in F_R \tag{4.1}$$

This property (reconstruction) guarantees that fragments can rebuild the original database. Finally, disjointness defines that if a relation R is horizontally fragmented, data item $\mathrm{d_j}$ in $\mathrm{R_j}$ should not exist in any other fragment $\mathrm{R_k}(\mathrm{k} \neq \mathrm{j})$ In case of vertical fragmentation, disjointness is defined only on the non-primary key attributes (columns) of the relation since primary key typically appears in all its fragments for reconstruction. [21]

4.2 Replication and allocation

Replication stores copies of a relation at different servers. A distributed database is typically replicated, but the degree of replication can range from no replication to full replication. If the data is fully replicated, the whole database is available at every site and this can remarkably improve the availability of the data. This can improve global query retrieval performance since such queries can be obtained from any site locally. However, this approach can drastically slow down the update performance because updates should be performed on every copy in different servers (sites). On the other hand, a non-replicated database allocates each fragment to different sites, and each server contains only one copy of each fragment. Here, all fragments must be disjoint except primary keys in vertical fragments. However, there is still a wide spectrum of partial replication of the data where part of the data may be replicated while others may not. [7]

Using replication itself has several benefits. Replication makes a copy of the data to be available on a site even when some sites fail to connect, so reliability and availability of the data can be significantly improved. Because of this feature, each transaction of the database may be performed without interaction with other networks, so the transaction becomes user-desirable. Performance can be improved by locating data into physically close locations as well. Geographically dispersed data results in the response time reduction. [20]

Once a database is fragmented properly, each fragment must be allocated to a server (site) in a distributed system. During allocation, data (each fragment) may be replicated or remain as non-replicated. The purpose of the allocation is to find the optimal distribution of the fragments to available network sites. That is, although processing and storage constraints, we want to minimize the total costs of the distributed database model. [1]

5 TRANSPARENCY MANAGEMENT OF DISTRIBUTED DATA

Transparency means disengaging the high-level semantics of the system from low-level implementation. Although distributed database may introduce additional complexities in implementation, implementation details should be hidden from users. Typically, a relation is divided into partitions and each partition is stored at different sites, and each partition may also be replicated in a distributed database. However, fully transparent should be able to pose queries regardless of fragmentation or replication. For full transparency, various types of transparency should be dealt with: data independence, replication transparency, and fragmentation transparency.

Among the transparency types, data independence is the most fundamental property of transparency management in a distributed database system. Generally, data is defined at two levels: one level at the logical structure and another level at the physical structure. The logical structure defines the schema of the data and its independence makes sure that changes in the logical structure (schema) of the database are hidden from the end-users. In contrast, physical data independence refers to the immunity against hiding details of the storage structure.

Secondly, replication transparency refers to the immunity to the existence of copies of data when the data is replicated across different sites. Generally, specifying certain actions should not be based on the copies of the data from the user's point of view, but the responsibility of specifying an action can be delegated to the user as well. Doing so makes the transaction management simpler, but the flexibility of the database is lost. Besides, how many copies exist is decided by the application, not by the system itself, and any changes on this affect the data independence. Thus, when designing a distributed database, replication transparency should be a standard.

The final form of transparency in a distributed database system is fragmentation transparency. When a relation is fragmented, queries defined on the entire relation scope should be executed on sub-relations. Thus, translation from global query

to several fragment queries is required to achieve fragmentation transparency and although queries are specified on relations rather than fragments, processing strategy should be found in the opposite. Besides, most importantly, users should be able to access data regardless of fragmentation.

Ideally, to provide efficient and easy distributed database management service, full transparency is preferred. Despite this, transparency is a compromise between costs and ease of use, as it is difficult to achieve high levels of transparency. [14]

6 DISTRIBUTED CONCURRENCY CONTROL AND RECOVERY SOLUTIONS

Unlike centralized database systems, distributed database systems often include concurrency control, query processing, data replication, fragmentation, etc. These are typically not introduced in a centralized database, and dealing with them may introduce additional complexities and problems. For example, data replication helps achieve better performance in a distributed database, but it poses a major challenge due to data inconsistency. When there are data copies in multiple networks/sites, site or network failure may arise. If a transaction spans multiple sites, some sites may not be able to roll back changes while others are successful. Keeping the data updated when multiple transactions are concurrently executed is the most important thing to keep in mind in a distributed database.

6.1 Database failure

Failures are defined by two concepts: RTO (Recovery Time Objective) and RPO (Recovery Point Objective). A recovery time objective (RTO) refers to the time it takes the application to recover from a single failure, whereas a recovery point objective (RPO) refers to how much data can be lost as a result of a failure (how quickly databases can be recovered). RTO and RPO values differ depending on the type of failure, and the types of failure can be further subdivided into high availability, disaster recovery, and human error. Specifically, high availability refers to the failure of a single server or network partition, while disaster recovery refers to the failure of the entire system or region. Besides, data loss can be caused by human error when somebody drops a database or introduces a bug into the application.

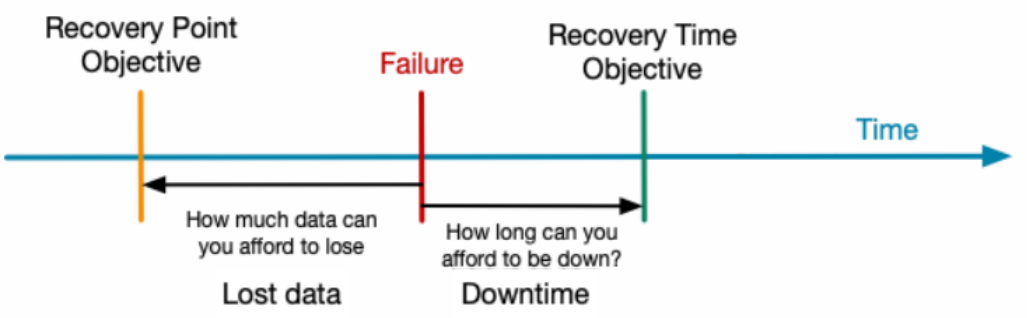

Figure 6.1 *Failure concepts [18]*

To overcome these problems, recovering techniques must be taken action. Typically, recovering algorithm consists of two parts. One is the action taken during normal operations to ensure the system can recover from a failure such as backup or log file, and the other action is to take after a failure to restore the database to the consistent state. [13]

6.2 Distinguished copy technique

To deal with data inconsistency in distributed database with multiple sites, distinguished copy concept is introduced. Distinguished copy (main copy) is designated when relation (data item) exists at three sites, and the other copies depend on it. Any locking or unlocking from transactions is applied to the distinguished copy and the other tables are depend on main copy. *Coordinator site* (main site) is the site that contains distinguished copy (main copy) of the data.

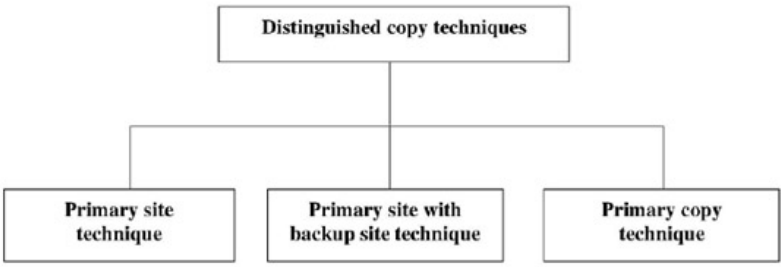

Figure 6.2 *Distinguished copy techniques [13]*

In primary site technique, one primary site is assigned to coordinator site for all data items and any locking or unlocking happens at this site. In other words, primary site is the only site that takes care of transaction management. This method is similar to centralized DBMS in a way that all locking/unlocking happens at the same site.

This technique is easy to understand, but overload can easily happen if there are too much transactions coming into primary site. Besides, if the primary site fails, all transactions being applied to the primary site at the moment should be aborted. Afterwards, the primary site is changed to the new one, and the transactions that were aborted need to be redone.

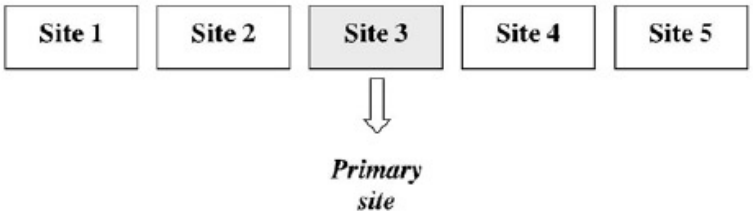

Figure 6.3 *Primary site technique [13]*

To deal with the problems occurring in primary site technique, backup site can be additionally assigned along with primary site. This technique is called primary site with backup site technique. Unlike primary site technique, locking/unlocking happens in both primary and backup site in this technique. When a primary site fails, the backup site becomes the primary site and a new backup site is selected. Unlike the primary site technique, however, we do not need to abort all transactions during failure, rather we can put them on hold and resume once a new primary site is chosen.

However, there still needs to be logic on how to choose a new backup site in case of primary site failures. The status of the coordinator site should be recognized based on unsuccessful attempts. Failed site needs to send a message to all other sites in distributed systems proposing to become a new coordinator site. These other sites can then decide to approve or reject it. This mechanism is called *election*. Not only current primary site, but few more other sites can also declare their election, and the site which gets highest votes become the new coordinator site. This technique ensures that whole system does not fail, but there is still a small delay for changes to happen. Compared to primary site technique, locking and unlocking happens in two places, so performance inevitably becomes lower. Primary site technique and primate site with backup site technique can be a great compromise, but these techniques are more or less merely an extension of the centralized database system and can still cause overloads in primary/backup sites. Operations still needs to be distributed more evenly to overcome overloads problem. In primary copy technique, data is fragmented and distributed over the sites. Instead of primary site responsible for locks/unlocks, primary copy of the data which needs to be handled is identified first and locked. This technique reduces the load on single site, but identifying

primary copy or backup is challenging and maintaining distributed data may require extra work.

6.3 Voting method

Unlike primary copy and primary copy with backup methods, the voting method does not designate one of the sites as a distinguished copy. Whenever a data item is to be updated (i.e. transaction requests to update a data item), all sites with a copy of the data receive a locking request. In other words, every copy (site) of the data item is associated with a lock, and the copy (site) has the right to decide whether to grant or deny a lock. Here, the decision to say yes or no is called a vote. There is a minimum threshold that requires a transaction to proceed, and the transaction is made based on the threshold and the number of votes. If the sum of all *Yes* votes is equal to or greater than the threshold, the transaction obtains a lock on the data and informs all the other sites about it. [11] Otherwise, the transaction abandons its locking request and informs sites about it. The voting method is more distributed compared to distinguished copy and primary site techniques. However, locking and unlocking requests/responses require the messages to travel between sites frequently, which makes the system slower. [13]

This majority voting principle is based on a quorum-based technique in distributed computing. Let us assume that the total number of votes in our system is V and every site is assigned a vote V_i. Additionally, we define abort quorums and commit quorums as V_a and V_c, respectively. To implement the commit protocol under this condition, the following rules must be followed:

1. $V_a + V_c > V$, where $0 \leq V_a, V_c \leq V$
2. Commit quorum V_c must be obtained before a transaction commits (6.1)
3. Abort quorum V_a must be obtained before a transaction aborts

To ensure consistency, the first rule states that a transaction cannot be committed and aborted at the same time. The second and the third rule makes sure that the transaction needs to obtain votes to make a transaction. These rules are important especially when there is network partitioning and sites are not able to communicate with each other properly. In the case of a fully replicated database, the initial voting algorithm states that equal votes should be assigned to each site, and votes from the majority of the sites should be earned for a transaction to execute. [14]

7 DATABASE CLUSTER

Our database is built on MySQL, so the existing methods in MySQL can be used for optimizing the existing database. MySQL Cluster is the open-source technology providing shared-nothing, distributed, and partitioning for the MySQL database management system. MySQL Cluster originates from Network Database (NDB or NDB Cluster). A network database is the extension of the hierarchical database, which is built in a parent-child manner. In a parent-child relationship, a child can be related to only one parent (owner) while a parent can be related to more than one child (member). However, this structure has problems in representing many-to-many relationships since the whole structure needs to be changed when adding a new relationship. By allowing each child to have multiple parents rather than one parent, the network database enables having more complex relationships. Typically, network databases are represented as graphs whereas a hierarchical model organizes data in a tree structure and relational model stores data in tables. For this reason, MySQL cluster is often called as NDBCluster (NDB) as well.[15]

The goal of the MySQL Cluster is to achieve the high availability and performance. Typically, MySQL clusters are built from three different types of node: management node, data node, and SQL node. Here, node does not necessarily mean a physical machine, but rather a process that is part of a cluster. Thus, running multiple nodes in one computer is possible. Management node should be started before starting any other nodes, and this node is responsible for controlling the other nodes within the NDB Cluster. In addition to reading the cluster configuration file, this node deploys the NDB cluster configuration to other nodes when the other nodes start. On top of the management node, there should be as many data nodes as the multiplication of the number of fragments and the number of replicas of the data. The NDB Cluster can support one to four replicas, and all node groups must have the same number of replicas. Theoretically, three or four replicas improves availability of the system and provides more redundancy, but two replicas are commonly considered sufficient. It is not possible to change the number of clusters after the cluster has been started. Finally, SQL nodes are responsible for accessing the cluster data and designating application that accesses Cluster data.

Cluster configuration defines configuring each node and setting up individual communication between nodes. Each node in the cluster fetches data from configuration data in management server, so the location of the management server should be determined. Whenever an event occurs in data nodes, nodes shifts information to the management server and writes the information to the cluster log. Configuration options include the configuration of management node, data node, SQL node, node interconnect, etc.

8 MYSQL INNODB CLUSTER

The key component of a robust, always-on infrastructure is high availability, as well as enterprise-grade databases. In recent years, MySQL has steadily added high availability features. Among these features are detailed management, storage engine configuration, reporting, automatic failover, etc. Storage engine is a mechanism for storing data in multiple ways: CSV, archive, memory, blackhole, InnoDB, etc. The method for storing data varies depending on the type of storage engine. It is important to choose the right storage engine, and InnoDB is a general-purpose storage engine that provides both high performance and high reliability. MySQL 5.7 uses InnoDB as a default storage engine, which means creating tables generates InnoDB tables unless another storage engine is configured separately. Another thing to ensure highly availability is to eliminate a single point of failure. For this, building a distributed environment is necessary.

8.1 Initial design

Before we build a distributed database, it is important to ensure the correctness rules of the database. Following one of the properties of correctness rules, reconstruction property, original database should be able to be reconstructed using fragment replicas if the data was fragmented over multiple places. Assume that we have three replicas of data located in each server at different places and one client backend server (Figure 8.1). In this case, the client accesses either one or multiple servers at the same time. Should the client connect to one server, the user must know which server (database) holds which data elements. Data elements can be either partitioned (non-replicated) or replicated and assigned to different sites. In the case of replicated data, data can be either fully replicated or partially replicated where each partition of the data is stored at multiple sites. Assuming that we have fully replicated databases in three different servers, recovery techniques on how to decide primary/secondary nodes should be decided. When the primary database becomes unavailable, other replicas should be automatically redirected to other available databases. But how can this be achieved?

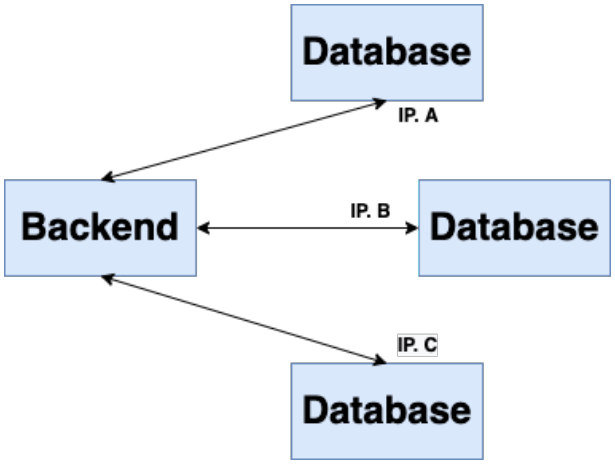

Figure 8.1 *Possible new database structure*

In the case above (Figure 8.1), the client is directly connected to the servers and the client needs to know which server to connect although all databases hold the same data elements. Using a primary site recovery option, the backend must know which database is primary and secondary in the event of failure of the primary server. Additionally, it should be defined if one database gets updated at a time, or if all databases get updated at a time. However, the use of middleware can significantly accelerate the development of distributed systems and simplify their interconnection (see Figure 8.2). The middleware is capable of routing or redirecting backend requests, and it can also determine which database to connect instead of the backend.

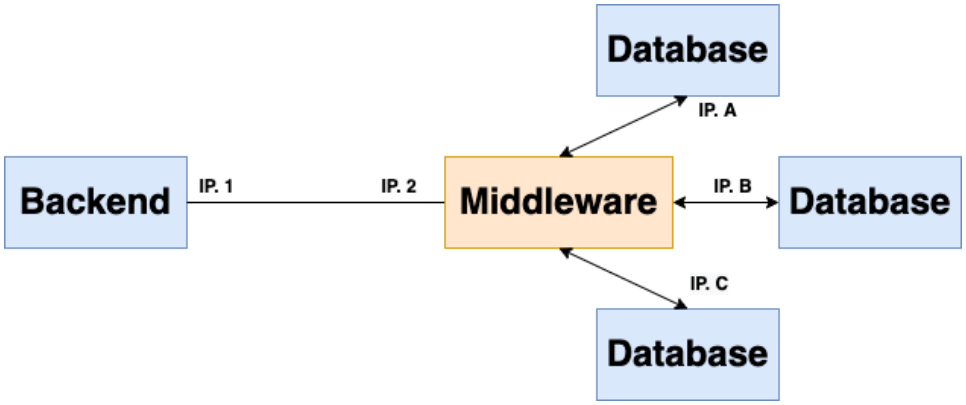

Figure 8.2 *Possible new database structure*

There are various MySQL middleware options which can be used: MySQL Proxy, MySQL Router, etc.

8.2 InnoDB cluster structure

Our initial design contains one backend server and middleware that connects backend server and multiple databases with replicas of the original data at each site. Middleware should be able to not only route requests from backend but also identify the status of the database servers real-time. With the help of InnoDB Cluster, these problems can be handled with implementation free. The core component of the InnoDB Cluster is InnoDB storage engine. InnoDB Cluster includes Group Replication, Routers to route connections, and MySQL Shell to simplify setup and configuration (Figure 8.3). To achieve high availability, at least three servers are required which includes one primary server with read and write permission and two other secondary servers with read-only permission by default. However, multi-primary mode can be configured in InnoDB Cluster as well, and in this case, there won't be any primary instances. On top of MySQL servers, MySQL Router lies between the application and servers and the connection to server should be made through this. Besides, MySQL Shell is needed to perform various administration operations and query data.

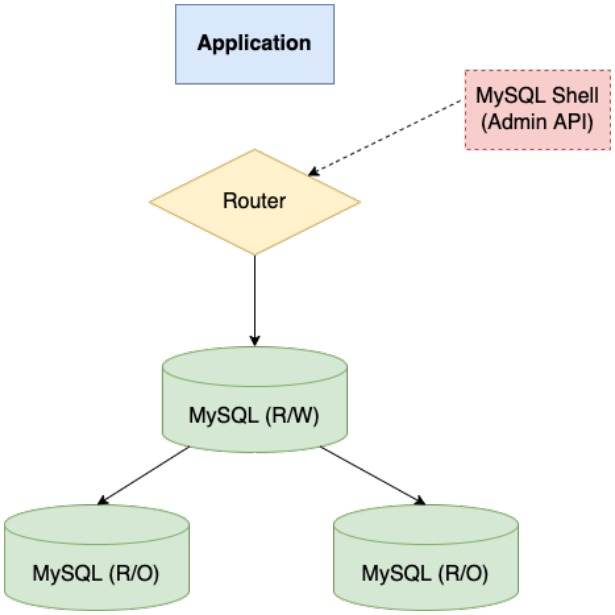

Figure 8.3 *InnoDB Cluster Structure*

8.3 Group replication

Creating a fault-tolerant system usually involves making redundant components so that when one of the components is removed, the system continues to operate. This, however, raises the complexity of such systems to a new level. Maintaining and administering a replicated database takes place on several servers instead of just one.

In addition, as the number of servers grows and interacts, network partitioning and splitting problems may arise. Therefore, coordinating several servers in a consistent and simple way is the ultimate challenge when using a replicated database. With MySQL Group replication, replication is distributed and coordinated between servers.

A group replication system ensures continuous database availability, and it equips fault-tolerant systems. Servers located in the same group coordinate automatically. Group replication automatically selects a single primary server to accept updates by default, but multi-primary mode can be set as well with additional configuration. Whenever one of the databases in a group becomes unavailable, the clients connected to that server are either redirected or failovered to a different server in the group.

Here, redirect or failover can be done using middleware such as connector, load balancer, router, or some other middleware services. Group replication does not have a built-in method for this, but MySQL InnoDB cluster uses MySQL Router as a middleware. Deployment of the group replication can be done either in multiple server instances or locally. Deploying in multiple servers is the most common way to achieve the high availability of the database, but local deployment can be performed for testing purposes.

In order for a transaction to commit, it must be agreed on by the majority of the group. Essentially, all servers take the same decision in relation to whether to commit or abort a transaction, but the decision is taken within the same server. In case of a network partition, where servers cannot reach an agreement, the system cannot move forward until the matter is resolved. In this case, manual action is required to elect a new primary database.

All of this is facilitated by the Group Communication System (GCS) protocols. Group Communication System provides a failure detection mechanism, a group membership service, and secure, in-order delivery of messages. In order to ensure that data is consistently replicated throughout the group of servers, all these properties must be in place. This technology is based on an implementation of the Paxos algorithm, which acts as the engine for group communication. [19]

8.4 MySQL router

MySQL Router routes application requests to backend MySQL servers through lightweight middleware. The routing details should be hidden from the application side, meaning that routing should be transparent. When a database failover occurs

in an application, it is typically handled by specifying which database is primary, but using MySQL router, this can be handled without implementation. Behind the scenes, the router stores a series of InnoDB Cluster servers and their status. A configuration file originally contains this list (or cache) of servers, and subsequent communication between the router and the cluster ensures that it is updated as the topology changes. As soon as a server goes offline, the router marks it as offline, and it skips it. Also, when a new server is added to the cluster, the cache of the router updates to include it as well. To keep the cache up-to-date, the router queries the cluster metadata from the performance schema database while maintaining a connection with one of the servers. Whenever a change in cluster state is detected - for example, if one of the MySQL servers has been shut down unexpectedly - these tables (or views) are updated in real-time. [19]

When using the router to connect your applications, the connection should be always made to the MySQL port instead of directly to the cluster servers' ports. Connecting to router will automatically results in connecting to the master instance since MySQL router handles it without implementation. Additionally, MySQL router should be able to reconnect to the next master server when a master server goes offline, which means that retry mechanism should be featured in connections. When connection attempt fails, MySQL router should redirect the connections and does not read packets or perform any analysis on the unavailable server. In this case, MySQL router returns the connection error to the application. Thus, applications should be written in a manner that identifies connection errors and, if necessary, retries the connection. In most cases, developers build this functionality directly into their applications, so you may only have to change the connection from your router to that of the application. [6]

8.5 MySQL shell

The MySQL Shell is one of the components of InnoDB Cluster, and is an open-source interactive interface which supports multi-language with three modes: Javascript, Python, or SQL (Figure 8.4). Shell has fully integrated built-in help system, which means that the user can access within the shell to have the commands and other features of it without being needed to get out from it. Both interactive and various administrative batch operations can be performed using MySQL shell.

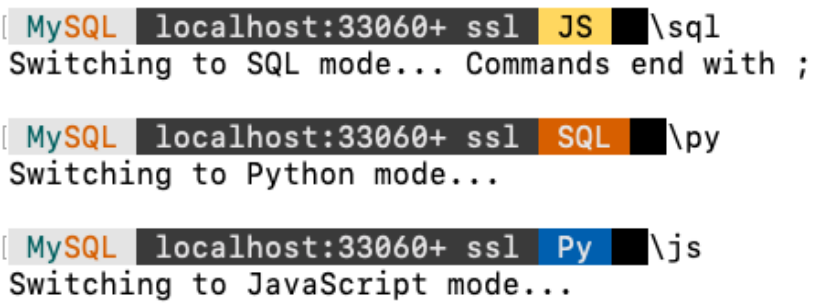

Figure 8.4 *MySQL Shell multi-language support*

MySQL Shell has three built-in APIs (X DevAPI, ShellAPI, AdminAPI) and all APIs are available in both JavaScript and Python. Among them, the AdminAPI for setting up InnoDB Cluster and InnoDB Clusterset and it can be accessed via the global variable *dba* and its methods. The class *dba* provides various methods for deploying, configuring, and administering InnoDB Cluster. In our development, we will use MYSQL Shell for checking the status of the cluster/instances, and updating data. [6]

8.6 Configuring in Docker Compose

Setting up InnoDB Cluster can be done by separately downloading MySQL router, MySQL shell and deploying, but using Docker Compose enables us to set up all required services in one place and execute them at once with one command *docker-compose up*. We will use Docker compose version 3.8, which is the latest version at the moment of writing this paper. Consider the case where we have three MySQL instances, MySQL shell, and MySQL router in our distributed architecture. We can define five different docker-compose services named server-1, server-2, server-3, router and shell. However, for defining them, we need to first set the default values for environment variables related to Compose file under the name *.env* (Program 8.1). There are various cluster options available, but as a starting point, we left cluster options as empty, which means that our InnoDB Cluster will be running as single primary mode.

Program 8.1 *Environemnt variables definition*

```
MYSQL_VERSION=8.0
MYSQL_ROOT_USER=agnet_root
MYSQL_ROOT_PASSWORD=mysql
MYSQL_HOST=server-1
MYSQL_PORT=%
```

```
MYSQL_INNODB_NUM_MEMBERS=3
MYSQL_CLUSTER_NAME=agnet
MYSQL_CLUSTER_OPTIONS={}
```

We will use the latest stable version of MySQL which is 8.0, and define MySQL username and password since MySQL stores accounts in *user* table of the MySQL system database. MYSQL_HOST for default host name used for command *mysql* and MYSQL_INNODB_NUM_MEMBERS ensures that three MySQL servers are up to create a MySQL Router. These environment variables will be used when specifying each server in Compose file. For example, each MySQL Server can be defined as (see Program 8.2):

Program 8.2 *MySQL Server definition in Docker Compose*

```
server-1:
    image: "mysql/mysql-server:${MYSQL_VERSION}"
    ports:
      - "30001:3006"
      - "40001:40060"
    command:
      - "--server_id=1"
    environment:
      - MYSQL_PASSWORD
      - MYSQL_ID
    volumes:
      - "./my.cnf:/etc/my.cnf"
      - "server1volume:/var/lib/mysql"
    restart: always
```

A Docker volume is a file system mounted on a Docker container to store data generated by running containers. Container volumes are stored on the host, regardless of their lifecycle. This enables users to back up and share data between containers. In our case, we define volumes for three servers, and define each volume mounting from local directory (left) to the directory in container. The path */etc/my.cnf* is a default location of the MySQL server configuration file where global options can be set. Local configuration file at *./my.cnf* overwrites the global configurations in container at */etc/my.cnf*. Besides, MySQL database data files are deployed at /var/lib/mysql in container, and this makes it possible to keep track of data. Program 8.2 shows the service definition for service 1, but same settings are configured for the second and third server except for using different ports and volumes.

Along with three MySQL Servers, MySQL Shell and Router should be defined as one of the services in docker compose as well. Predefined environment variables are passed to containers. (see Programs 8.3 and 8.4). Here, since MySQL Shell and Router requires the same environment variables and both depends on three MySQL Servers, we define them as *sql-environment* and *sql-dependents* respectively to prevent declaring same variables twice. Since both Shell and Router depends on MySQL Servers, they cannot start being run before three servers are started. Besides, router defines that port 6446 is used for primary access, and 6447 is used for secondary servers access.

Program 8.3 *MySQL Shell definition in Docker Compose*

```
shell:
    image: "mysql/mysql-server:${MYSQL_VERSION}"
    entrypoint: "/bin/shell-entryscript"
    environment: &sql-environment
      - MYSQL_USER
      - MYSQL_ROOT_PASSWORD
      - MYSQL_ROOT_HOST
      - MYSQL_PORT
      - MYSQL_CLUSTER_NAME
      - MYSQL_CLUSTER_OPTIONS
      - MYSQL_INNODB_NUM_MEMBERS
    volumes:
      - "./shell-entryscript.sh:/bin/shell-entryscript"
    depends_on: &sql-dependents
      - server-1
      - server-2
      - server-3
```

Program 8.4 *MySQL Router definition in Docker Compose*

```
router:
    image: "mysql/mysql-router:${MYSQL_VERSION}"
    ports:
      - "6446:6446"
      - "6447:6447"
    environment: *sql-environment
    volumes:
      - "./data/router:/var/lib/mysqlrouter"
    depends_on: *sql-dependents
    restart: always
```

Once we define all the required services (MySQL Server Instances, Shell, and Router) for MySQL InnoDB Cluster, we need to define shell commands to add instances and create a cluster. MySQL Shell commands can be written with JavaScript or Python or MySQL commands, but JavaScript was chosen in this work. As explained earlier, AdminAPI is accessed by the global variable *dba*, and we can create a cluster with predefined name and options and add server instances using this class *dba*. Firstly, we try making an InnoDB Cluster without additional options, which means that InnoDB Cluster will have a single primary instance.

8.7 Test of site failure/recovery

Once we configure and run InnoDB Cluster successfully, site failure can be simulated. Since we simply configured three MySQL servers in different ports in local computer, we cannot achieve complete disaster recovery since all servers reside in same site, but site (server) failure can still be demonstrated as a testing purpose. We use MySQL Shell to check the status of the cluster and instances running. Figure 8.5 indicates that three servers are in ONLINE status and there is only one primary node (server-1) which has read and write permission and two secondary nodes which has only read permission.

```
 MySQL  localhost:33060+ ssl  JS  cluster.status();
{
    "clusterName": "agnet",
    "defaultReplicaSet": {
        "name": "default",
        "primary": "server-3:3306",
        "ssl": "REQUIRED",
        "status": "OK",
        "statusText": "Cluster is ONLINE and can tolerate up to ONE failure.",
        "topology": {
            "server-1:3306": {
                "address": "server-1:3306",
                "mode": "R/O",
                "readReplicas": {},
                "role": "HA",
                "status": "ONLINE"
            },
            "server-2:3306": {
                "address": "server-2:3306",
                "mode": "R/O",
                "readReplicas": {},
                "role": "HA",
                "status": "ONLINE"
            },
            "server-3:3306": {
                "address": "server-3:3306",
                "mode": "R/W",
                "readReplicas": {},
                "role": "HA",
                "status": "ONLINE"
            }
        }
    },
    "groupInformationSourceMember": "mysql://root@server-3:3306"
}
```

Figure 8.5 *InnoDB Cluster status*

Next, we will force killing the primary node and see whether our InnoDB Cluster can tolerate the primary site failure. After stopping a primary server (server-1) container, Compose gives warning messages regarding the server disconnection, and the status of the cluster says that server-1 is offline (missing) and the server-3 has been automatically selected to primary node (Figure 8.6). But why is only one failure allowed in InnoDB Cluster? We configured our InnoDB Cluster to be a single primary mode by default, and according to the voting method, the primary partition should earn more than half of the votes to achieve quorum. We have a cluster of three members and let's imagine that one of the servers goes down, which results in the network split into two partitions: one partition with online and one with offline. In this case, we can still reach quorum since one partition still has quorum with 2/3 (66%).

```
MySQL  localhost:33060+ ssl  JS  cluster.status();
{
    "clusterName": "agnet",
    "defaultReplicaSet": {
        "name": "default",
        "primary": "server-3:3306",
        "ssl": "REQUIRED",
        "status": "OK_NO_TOLERANCE",
        "statusText": "Cluster is NOT tolerant to any failures. 1 member is not active",
        "topology": {
            "server-1:3306": {
                "address": "server-1:3306",
                "mode": "R/O",
                "readReplicas": {},
                "role": "HA",
                "status": "(MISSING)"
            },
            "server-2:3306": {
                "address": "server-2:3306",
                "mode": "R/O",
                "readReplicas": {},
                "role": "HA",
                "status": "ONLINE"
            },
            "server-3:3306": {
                "address": "server-3:3306",
                "mode": "R/W",
                "readReplicas": {},
                "role": "HA",
                "status": "ONLINE"
            }
        }
    },
    "groupInformationSourceMember": "mysql://root@server-3:3306"
}
```

Figure 8.6 *InnoDB Cluster status*

However, if more serious network failure happens which results in splitting the cluster into three partitions, the cluster is no longer online since none of the partitions can reach quorum with 1/3 (33%). In this case, a manual trigger is required to activate a cluster back online.

Next, we test updating tables in primary node which have read/write options, and observe whether updates are in other nodes (Figure 8.7). Firstly, we investigate the most basic case where all instances are online and update the data in the primary node. For this, we connect to the MySQL Router's port and try inserting one row in one of the tables in the database. The goal of this is to observe the consistency of the data in all instances in the InnoDB Cluster. After updating data in primary, we indeed observed that data is updated in the other two secondaries as well.

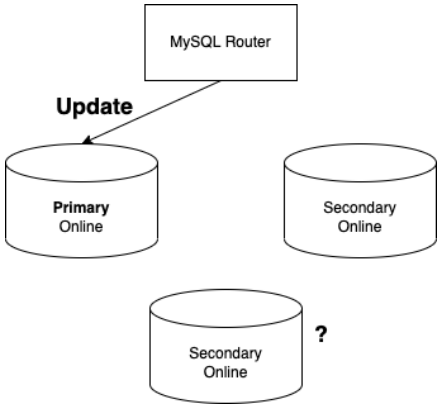

Figure 8.7 *Update table in InnoDB Cluster - case 1*

What if one of the secondaries were offline? Next, we test updating data in primary and see whether the instance that were offline updates data after it became back to online (Figure 8.8).

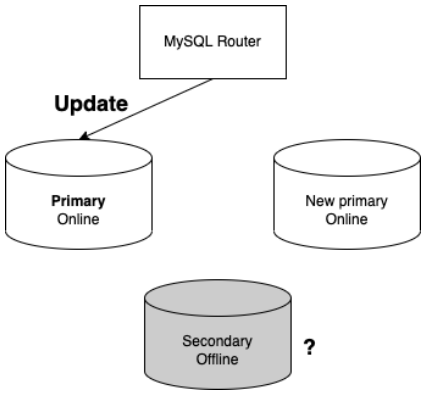

Figure 8.8 *Update table in InnoDB Cluster - case 2*

Finally, we kill primary node as well, and this results in selecting the remaining node as primary and there will be no secondary nodes left. What if we update the remaining primary node in this case (Figure 8.9)?

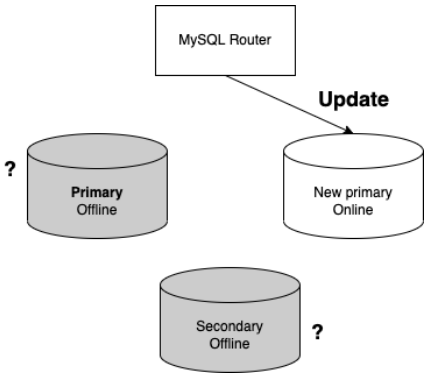

Figure 8.9 *Update table in InnoDB Cluster - case 3*

After the experiment, we observed that in both cases (Figure 8.8 and Figure 8.9), data was consistently updated to the nodes that rejoined the cluster after being disconnected. Similarly, when server leaves the group either by being disconnected or voluntarily, other nodes will automatically be informed. We can achieve RPO as zero using InnoDB Cluster, but this is not always the case especially when servers are located in different sites or if there are multiple servers in multiple sites.

8.8 Monitor status with prometheus/grafana

Once we set up our cluster, its metrics and status can be monitored in different ways. One of the methods to monitor is using the global variable *dba* to check the status of the our cluster in MySQL shell (Figure 8.5), but this method is manual and time-consuming. Thus, we need to figure out more convenient and automatic ways to monitor our cluster and possibly set the alerting rules. To configure monitoring system on top of existing cluster, we first need to set up MySQL exporter to collect MySQL metrics such as queries per second (QPS) and InnoDB buffer pool size. To show extracted data from MySQL exporter, we will use prometheus for relaying collected metrics and grafana for visualizing on dashboards. Prometheus is an open-source system for monitoring metrics and alerting in certain situations. MySQL exporter metrics are Prometheus-style, so Prometheus can be directly plugged-in to relay the metrics exported by MySQL exporter.

Similarly as configuring MySQL InnoDB Cluster, we will use Docker for configuring MySQL Exporter, Prometheus instead of downloading them locally. For MySQL Exporter, we use the Docker image named $prom/mysqld-exporter$ [17] and for Prometheus, we use $prom/prometheus$ [17].

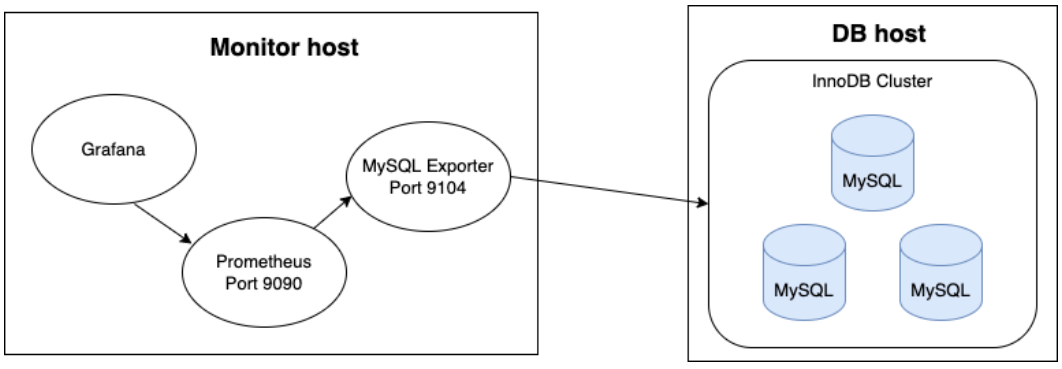

Figure 8.10 *MySQL metrics visualization structure*

MySQL exporter is set to run on port 9104 and fetch the data from MySQL master instance and Prometheus running on port 9090 is configured to monitor data being

fetched into MySQL Exporter (Figure 8.10). We first need to configure which target we want to collect the metrics from inside Prometheus configuration file, and Prometheus will initiate the connection to all configured targets and scrap the metrics at specific intervals. Once we configured the exporter and Prometheus, directing to the port 9090 shows the various time series data which can be visualized in graph. Metrics such as fetched/failed/successful messages per second, critical time in seconds can be selected in real-time. (Table 8.1 and Figure 8.11). However, Prometheus is mainly used for event monitoring and has very little to do with the visualization part, so we can use Grafana for better optimization of data.

Name	Count
mysql_global_status_commands_total	167
mysql_exporter_collector_duration_seconds	7
mysql_global_status_handlers_total	18
mysql_global_status_connection_errors_total	6

Table 8.1 *Memory usage and time profiling for list comprehension and generator*

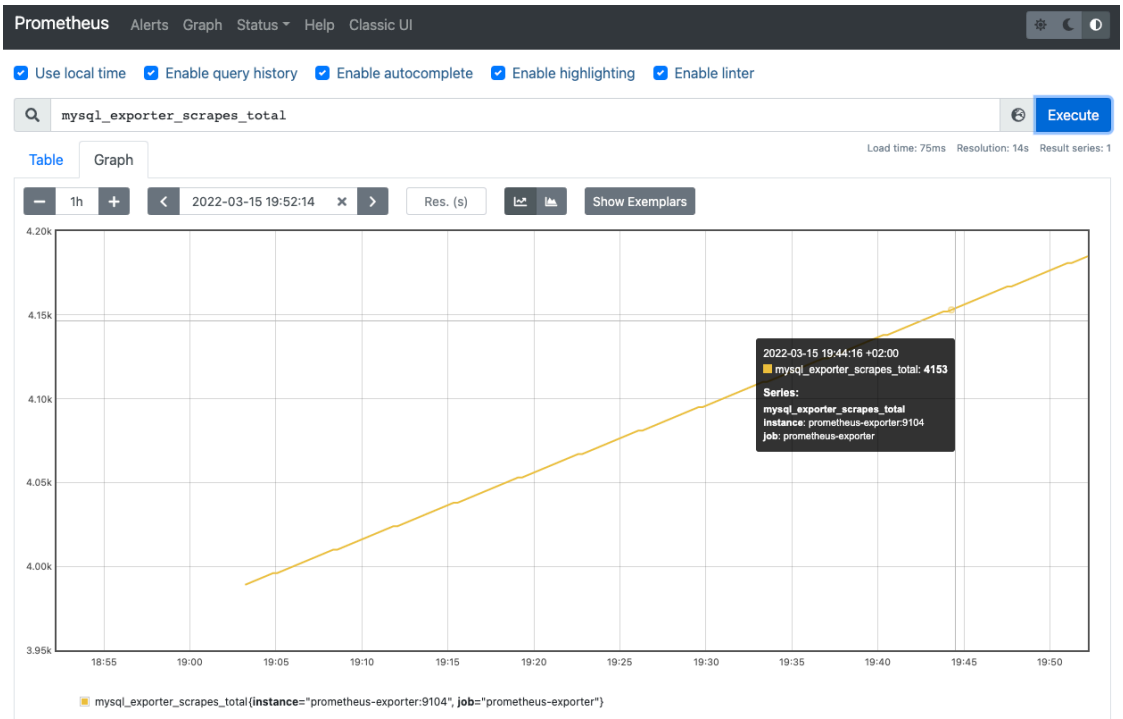

Figure 8.11 *Prometheus metrics*

Prometheus offers simple interface to query and read data, but it is mainly for collecting and storing it, not visualizing it. Therefore, we will deploy another application called Grafana for nicer visualization. Grafana can query metrics from

all Prometheus servers and use specific language called PromQL. We simply use Grafana web application and create a basic authentication for for remote writing for Prometheus. After we generate an authentication, username, password and url for remote writing should be specified in Prometheus configuration file.

After logging in to Grafana, we can visualize the information in a user-friendly dashboard. Figure 8.12 is one of the most basic dashboards, and it shows MySQL Overview of our database. Parameters such as uptime, InnoDB Buffet Pool, or Current QPS is shown in the dashboard. Besides, we import InnoDB metrics dashboard in Grafana [10] and Figure 8.13 shows more detailed information of the MySQL operations, I/O and query performance.

Figure 8.12 *MySQL status dashboard at Grafana*

Figure 8.13 *MySQL InnoDB metrics at Grafana*

9 IMPROVEMENT IDEAS

In the previous section, we implemented an InnoDB Cluster with three different servers running at different ports in local computers (Figure 8.3). Here, MySQL Router acts as a load balancer and routes transactions from the application server to InnoDB Cluster. However, is this structure highly available? Highly available system should never rely on single components in the system, which means that single point of failure would cause entire system to fail. Unfortunately, our structure in Figure 8.3 still has a single point of failure on MySQL Router side. Failure of the MySQL Router means that there will be no load balancer, and this eventually may results in failure of the whole infrastructure.

9.1 Router HA

To tackle single point of failure problem of the MySQL Router, we need to first have more than one highly available application servers. Assuming that we have four application servers running, multiple MySQL Routers can be installed at each application server, which routes the traffic to InnoDB Cluster (Figure 9.1). This way, we can keep the consistency of the servers: if the application server goes down, the MySQL Router server in the same server will be unavailable as well. Router is a lightweight process which can be installed on the application server, so setup can be done easily. Besides, router gets cluster's metadata from its configuration and does not require much resources or maintenance.

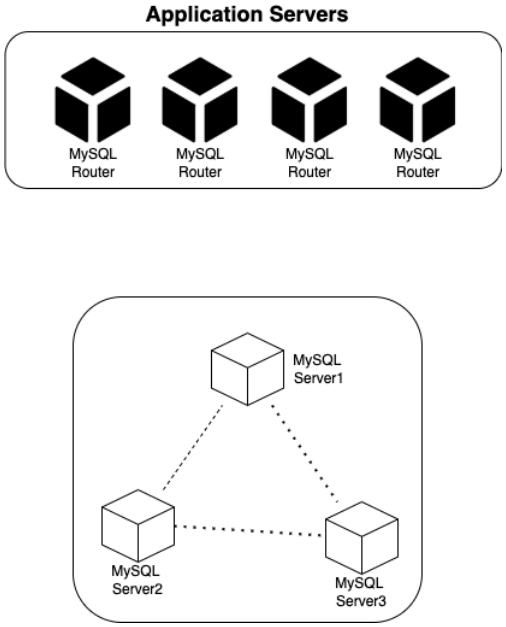

Figure 9.1 *InnoDB Cluster with Router HA*

However, some organizations/firms may not want to install MySQL Router in their application server, so MySQL Router may locate outside of the application in this case. However, we need to have more than one MySQL Router servers to avoid single point of failure. In this case, organizations can have HA for the MySQL Router and the choice of HA depends on the needs of the infrastructure.

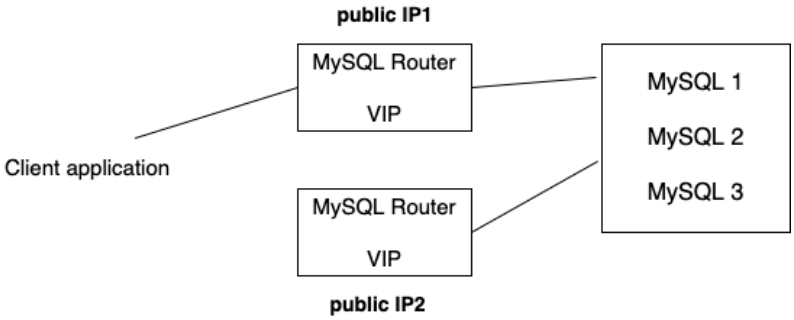

Figure 9.2 *Router HA structure*

Figure 9.2 has two routers connected to the application and MySQL Servers. Only one router can be connected to the application at a time, so master should be defined. However, although there is a backup router, if there is no logic to automatically select the backup router is not defined, reconnecting to backup router will be a manual process. Then how do we configure so that backup router is automatically connected to application when master router becomes available? With the help of framework named Keepalived, automatic failover of the MySQL Router can be configured.

Load-balancing frameworks rely on Linux Virtual Server (IPVS), a widely used kernel module that provides Layer 4 load balancing. Maintaining and managing load-balanced server pools dynamically and adaptively based on their health is the main purpose of Keepalived. Master router sends its advertisements using VRRP at regular intervals, and backup notices when the master stops sending advertisements. When failure is detected, Keepalived tries to reconnect to the new master. Similarly, when router is back to online, router is automatically added back to Keepalived load balancer.

For configuring Keepalived, first we need to have two MySQL routers. File named *keepalived.conf* consists of two parts: VRRP script in which VRRP instances are monitoring and VRRP instance. The most basic configuration enables Keepalived to share IP address between servers (master and backups). With Keepalived, MySQL Router can be equipped with automatic failover functionality, which means that backup router will be automatically elected as a primary when primary router goes offline.

9.2 InnoDB ClusterSet

Another thing to consider in our cluster is to prevent on region failure. Previously, we implemented three servers in different ports at local computer in InnoDB Cluster as a testing purpose(Figure 8.3). When InnoDB Cluster is set within a single region, RPO is zero and RTO is seconds depending on how fast is the recovery action. However, this method does not prevent the downtime in case of region failure. Thus, to tackle this, our existing InnoDB Cluster model can be calibrated in two ways: InnoDB Cluster across three regions, InnoDB Clusterset.

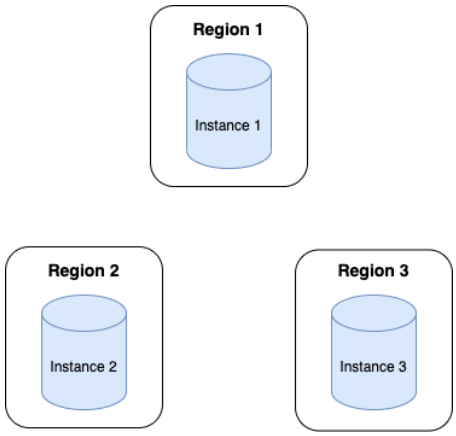

Figure 9.3 *InnoDB Cluster in three different regions*

In the first case (Figure 9.3), if we spread two or three members in each region, we

will have three or six members in different regions in total. In this environment, we can achieve no data loss when a region fails, and failover time is in seconds depending on how fast is the manual failover action. This method yet requires stable network between three data centers. Packet loss will cost and network partition will have an impact if the network goes down for minutes. In addition, this method requires at least three data centers for automatic failover. Why do we need at least three data centers? Let's assume that two members are located in one region and the other member is in other region (Figure 9.4). InnoDB Cluster is based on paxos algorithm, which means majority is acknowledged for the transaction to occur. In case of Figure 9.4, if the data center with two members goes down, the third member will wait for the response since it does not have a majority (33%). Thus, it should be guaranteed that the remaining instance gains majority before it gets elected as a new master.

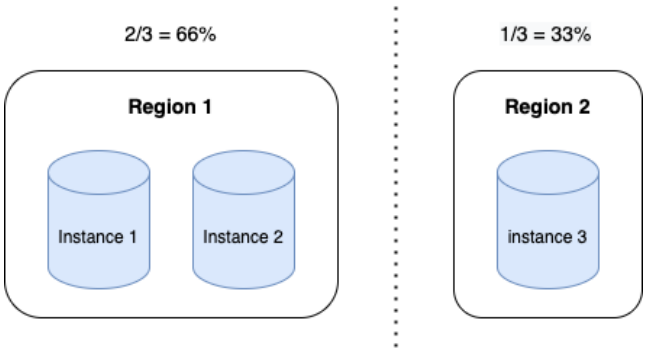

Figure 9.4 *InnoDB Cluster in two different regions - Case 2*

Unlike InnoDB Cluster in a single region, RPO cannot be zero in the above case (Figure 9.4) since consensus can be achieved in a single region. Transaction will happen only if Region 1 with two members is acknowledged, so RPO cannot be zero in this case.

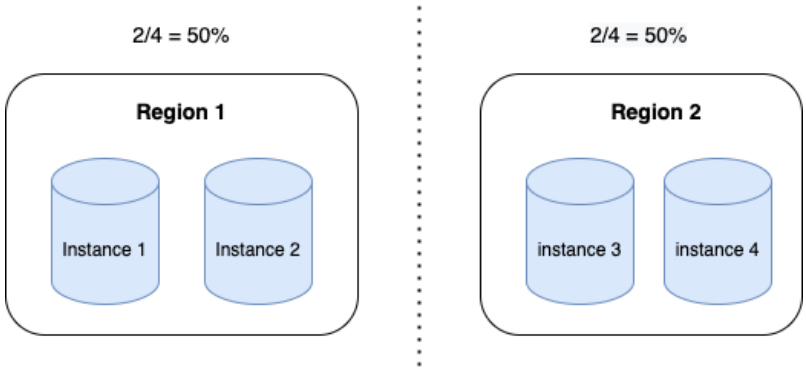

Figure 9.5 *InnoDB Cluster in two different regions - Case 3*

Another solution is to have two members in each data center and two in other center (Figure 9.4). In this case, we can achieve RPO 0, but RTO will still not be 0. If one region fails, we will lose two instance members out of four, which is 50%. In order to have an automatic network partitioning handling, we need to have more than majority (50%) so the secondary data center will wait until it reaches majority. Thus, manual action is required in this sense. That is why we need at least three data centers in order to have automatic failover.

Next solution is MySQL InnoDB ClusterSet. MySQL InnoDB Clusterset is a solution that links primary InnoDB Cluster to replicas of it in other regions. [16] Using this, automatic failover is guaranteed within a region and asynchronous replication between clusters with a manual failover. With this case, we cannot achieve RPO as not zero and RTO depends on how fast the manual failover is done. The fact that RPO is not zero means that some data loss may happen, so MySQL ClusterSet is ideal if data loss is acceptable in the unlikelihood events of disaster.

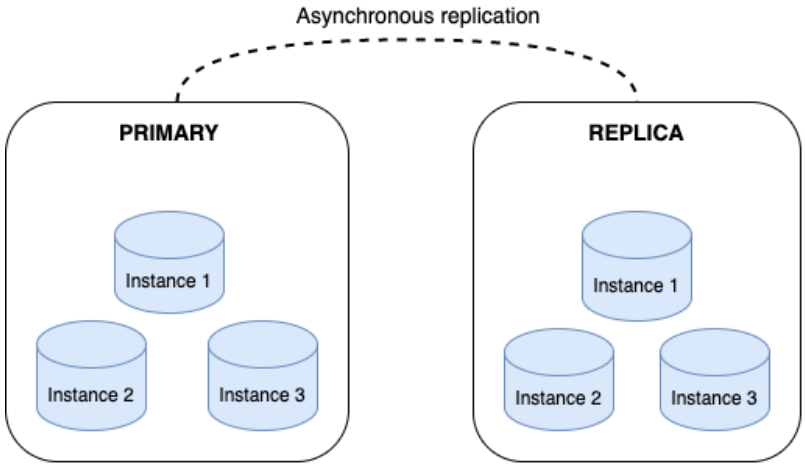

Figure 9.6 *InnoDB ClusterSet*

10 COMMERCIAL SOLUTION - AMAZON AURORA

Although a highly available database can be custom-made, commercial solutions can help us handle unexpected threats and secure our database in more structured way. For example, one of the commercial solutions, Amazon Aurora DB cluster has similar structure as our InnoDB Cluster (Figure 8.3). It consists of one or more database instances and a cluster volumes that manages the data of the instances. Multiple copies of the data are span across different availability zones supported by AWS region and single primary (master) has read and write access. Whenever primary is updated, Aurora synchronously replicates the data across availability zones.

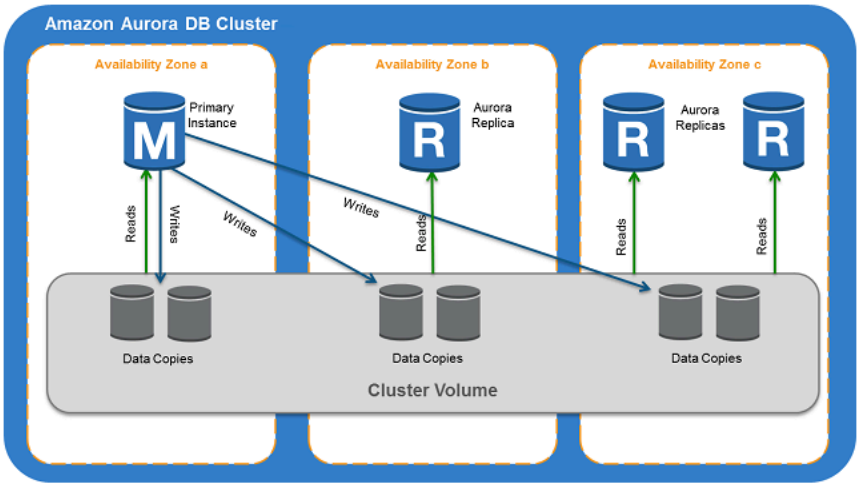

Figure 10.1 *Amazon Aurora DB Cluster [3]*

It is not necessary to use already-made commercial solutions, but commercial solutions handle many things that couldn't be considered in our solution such as security, recovery solutions. Cloud security is the highest priority when our actual application is deployed unlike local hosting as a testing purpose, and Aurora takes care of security threats such as data encryption, authentication. In our solution, we simply set root's username and password in configuration file, but using Aurora enables to use Identity and Access Management (IAM) database authentication. Using IAM

allows us to connect database instance without password, but rather with authentication token. Each token is generated by Amazon RDS and valid for 15 minutes. This feature is externally managed and the user does not need to store credentials in database themselves. [5]

As for high availability, an Aurora cluster with single-master replication mode can create up to 15 read-only replicas. These read-only replicas are also called as reader instances, and Aurora features automatic failover so that one of the reader instances takes into place when primary instance goes down. In Aurora, we can set the priorities in reader instances as well. Some readers can have a same priority, but in this case, a reader node that has the largest in size promotes as a primary instance. For example, if the entire AZ fails because of outages, failover mechanism depends on whether multi-AZ configuration is on or not. If other instances exist in other AZs, one of those readers become a primary node, otherwise we must manually create one or more new DB instances in another AZ. However, using Aurora Serverless enables to create a new primary instance in another AZ. [4]

11 CONCLUSIONS

As the amount of data surges, the importance of data management becomes key role in today's businesses. Management of the data not only includes the data itself but also to make sure that data is not lost and highly available in multiple places. Our application is mission-critical, which means that it is especially important to keep the application works continuously without being disconnected. However, it is impossible to predict all the errors and prevent it beforehand, so it is crucial to have backups in multiple places, possibly in different networks or regions. Our application has centralized database in one region, and this does not prevent from data loss during outages. Thus, we seek out new ways to redesign our existing centralized database in this thesis. Centralized database has its own benefits such as the ease of maintenance or low costs, but it does not cover disaster recovery, so we moved onto distributed environment where we locate replicas of the data in different networks/regions. Traditionally, MySQL worked well with asynchronous replication. However, asynchronous replication does not execute the replicated transactions to the slaves in parallel, which means that the contents of the slaves may not be synchronized as master. Therefore, we found a new solution where we can synchronously update the data and automatically choose the available primary instance: InnoDB Cluster.

We used Docker for configuring three MySQL Instances, MySQL Shell and Router. All these components were bind with Docker Compose so that all components can be run at once with one command. After we configure InnoDB Cluster, we test the consistency of the data when a server is forcefully killed. We first test killing a primary and observe whether a new instance is elected as a primary. The result indicated that when a primary was killed, another instance was automatically elected as a primary. After we started a old primary instance again, data was consistently updated. We also tested killing slave instances and observed the data, and data was automatically updated when they were back online. On top of InnoDB Cluster, we also added monitoring logic using Prometheus and Grafana. Prometheus relays changes from MySQL exporter and Grafana queries data from Prometheus, and finally Grafana displays MySQL metrics in dashboard. Although InnoDB Cluster significantly improves the existing system in the aspect of high availability, there are

still many things left to study and improve in order to build a fully highly available system where there is no single point of failure and manual failover. For example, our cluster has one MySQL Router connected to an application, but if this fails, the whole infrastructure will fail, so our cluster still has a single point of failure in this sense. To improve this, we need to have at least two application servers and MySQL Router installed in each application server. With Keepalived, we can have automatic failover functionality on Routers. In addition, our servers were built on localhost as a testing purpose, but in real application, it is optimal to locate each server in different regions. This way, we can have disaster recovery solution, but the downside is that it requires stable WAN to enable this structure. Thus, another solution comes into place: InnoDB ClusterSet. InnoDB ClusterSet is a group of InnoDB Clusters configured across multiple regions. Using this, automatic failover is guaranteed within a region and asynchronous replication is applied between regions. However, the downside of it is that manual failover is needed across the regions, so some data loss may happen in this case.

We suggest potential improvements on top of our InnoDB Cluster, but some challenges are still left behind such as security or server availability. That is why many businesses use commercial solution which takes care of everything that we did not potentially consider in this thesis. For example, Amazon Aurora is compatible with MySQL and takes care of highly available features. Although some or all of database instances in the cluster becomes unavailable, Aurora makes sure that no data loss happens across multiple availability regions. However, using commercial solution still needs a lot of customization depending on the needs of the businesses, and some firms prefer to use their own solution over commercial solutions. Thus, it is still meaningful to research on what are the problems of the existing solution and how to optimize it.

REFERENCES

[1] P. Apers. "Data Allocation in Distributed Database Systems". In: *ACM Trans. Database Syst.* 13 (Sept. 1988), pp. 263–304. DOI: 10.1145/44498.45063.

[2] A. A.-T. Asma H. Al-Sanhani Ali Al Dahoud. "A comparative Analysis of Data Fragmenation in Distributed Database". In: (2017). Normal order. DOI: 10.1109/ICITECH.2017.8079934.

[3] A. AWS. *Amazon Aurora DB Clusters*. URL: https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Aurora.Overview.html.

[4] A. AWS. *High availability for Amazon Aurora*. URL: https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Concepts.AuroraHighAvailability.html.

[5] A. AWS. *IAM database authentication for MariaDB, MySQL, AND PostgreSQL*. URL: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/UsingWithRDS.IAMDBAuth.html.

[6] C. Bell. *Introducing InnoDB Cluster Learning the MySQL High Availability Stack*. eng. 1st ed. 2018. Berkeley, CA: Apress, 2018. ISBN: 1-4842-3885-0.

[7] BrainKart. *Data Fragmentation, Replication, and Allocation Techniques for Distributed Database Design*. [Online; accessed 24-12-2021]. URL: https://www.brainkart.com/article/Data-Fragmentation,-Replication,-and-Allocation-Techniques-for-Distributed-Database-Design_11593/.

[8] C. T. C. MEGHINI. *The complexity of operations on a fragmented relation*. 3rd ed. NEW YORK:ACM. ISBN: 0362-5915.

[9] A. S. L. Communications. "Tactilon Agnet". In: (). URL: https://www.securelandcommunications.com/tactilon-agnet.

[10] *Grafana MySQL InnoDB Metrics*. URL: https://grafana.com/grafana/dashboards/7365.

[11] B. Hardekopf, K. Kwiat, and S. Upadhyaya. "A decentralized voting algorithm for increasing dependability in distributed systems". In: (July 2001).

[12] J. L. Harrington. *Relational database design and implementation clearly explained*. 3rd. Amsterdam ; Boston : Morgan Kaufmann/Elsevier, 2009. ISBN: 1-282-25841-9.

[13] A. Kahate. *Introduction to database management systems*. eng. 1st edition. Always Learning. New Delhi, India: Pearson, 2004. ISBN: 81-317-7077-X.

[14] P. V. M. Tamer Özsu. *Principles of Distributed Database Systems*. 4th. NEW YORK:ACM, 2020. ISBN: 3030262537.

[15] MariaDB. *Understanding the Hierarchical Database Model*. URL: https://mariadb.com/kb/en/understanding-the-hierarchical-database-model/+license/.

[16] MySQL. *MySQL InnoDB ClusterSet*. URL: https://dev.mysql.com/doc/mysql-shell/8.0/en/innodb-clusterset.html.

[17] *MySQL Server Exporter Docker image*. URL: https://hub.docker.com/r/prom/mysqld-exporter.

[18] Oracle. *Disaster Recovery Solutions: MySQL InnoDB ClusterSet*. URL: https://fosdem.org/2022/schedule/event/mysql_clusterset/attachments/slides/4959/export/events/attachments/mysql_clusterset/slides/4959/MySQL_InnoDB_ClusterSet.pdf.

[19] Oracle. *MySQL 8.0 Reference manual*. URL: https://dev.mysql.com/doc/refman/8.0/en/.

[20] M. Parul Tomar. "An overview of distributed databases". In: (2017). Normal order. ISSN: 0974-2239. URL: https://www.ripublication.com/irph/ijict_spl/ijictv4n2spl_15.pdf.

[21] C. Ray. *Distributed Database Systems*. Pearson Education India, 2009. ISBN: 8131727181.

[22] severalnines. *Disaster Recovery Planning for MySQL MariaDB*. URL: https://severalnines.com/resources/whitepapers/disaster-recovery-planning-mysql-mariadb.