Paulo Ricardo Almeida Moita

Degree in Computer Science and Engineering

Modular and Adaptive Key-Value Storage Systems

Dissertation plan submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Informatics Engineering

Adviser: João Carlos Antunes Leitão, Assistant Professor,
NOVA University of Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2019

The work reported in this document was partially supported by
Fundação para a Ciência e Tecnologia through the Project NG-STORAGE (PTDC/CCI-INF/32038/2017)

Abstract

Large-scale internet applications deal with an increasing number of users and consequently of data, requiring capable support infrastructures to follow the inherent evolution. Distributed storage systems have a fundamental role in this process as an essential layer to provide a good user experience, which in the case of not being acceptable, could represent a loss of operator revenues.

For this, these systems have, besides other properties, to provide high availability, fault tolerance and simultaneously keep the user data consistent, taking in consideration that is not possible the coexistence of all these functionalities. This way, each application have to opt for some guarantees usually provided by tightly coupled components, not allowing to change its specification in real-time.

In this document, we propose a novel design of the popular NoSQL Cassandra storage system which allows a flexible modification or swap of its components using a modular methodology. Following this approach, we propose the incorporation of two replication protocols with different consistency guarantees and support to geo-replicated scenarios.

Keywords: Distributed storage systems, user experience, consistency, replication, modular, geo-replication, Cassandra

Resumo

As aplicações de internet de larga-escala lidam com um número cada vez maior de utilizadores e consequentemente de dados, necessitando de infraestruturas de suporte capazes de acompanhar a sua inerente evolução. Os sistemas de armazenamento distribuídos têm um papel fundamental neste processo dado que são uma camada chave responsável por fornecer uma boa experiência de utilizador, que no caso de não ser aceitável pode representar perdas de receitas para o seu operador.

Para tal estes sistemas devem, entre outras propriedades, garantir alta disponibilidade, tolerância a falhas e ao mesmo tempo manter os dados dos utilizadores consistentes, tendo em conta que não é possível a coexistência de todas estas funcionalidades. Desta forma, cada aplicação tem de optar por determinadas garantias fornecidas por componentes que se encontram fortemente acoplados, não sendo possível a alterar a sua especificação em tempo real.

Neste documento, propomos um desenho inovador do popular sistema de armazenamento não-relacional Cassandra que permite flexibilidade na alteração ou substituição dos seus componentes através de uma metodologia modular. No seguimento dessa abordagem propomos ainda incorporar dois protocolos de replicação com diferentes garantias de consistência e suporte a cenários geo-replicados.

Palavras-chave: Sistemas de armazenamento distribuído, experiência de utilizador, consistência, replicação, modular, geo-replicação, Cassandra

Contents

List of Figures

List of Tables

Acronyms

CRDT	Conflict-free replicated datatype.
DDBMS	Distributed database management system.
DVV	Dotted version vector.

Chapter 1

Introduction

1.1 Context

The continuous growth in the number of users of large-scale Internet applications requires these applications to use support infrastructures with strict performance requirements to keep low latencies and ensure a good user experience, thus avoiding their departure. Composed by thousands of servers, these systems need to provide high throughput, scalability, availability and at the same time resilience to failures.

Distributed datastore management plays an important role in this since it is a key applications' layer responsible to overcome those challenges, while efficiently dealing with larger and richer data sources. An enhanced user experience is tightly bounded by low communication overhead, which can be accomplished adjusting the key structures of the underlying datastore such as Partitioning, the Replication Protocol and its enforced Consistency Model, the Membership management scheme, among others.

Partitioning is usually employed to provide higher performance in aspects regarding availability, manageability or load balancing. Systems relying on vertical partitioning usually split data by columns, distinguishing static from dynamic data to boost query performance. Horizontal partitioning is formed placing different rows into another table on the same node or across multiple nodes in a technique named Sharding. In scenarios with an increasing traffic load where elastic cluster scaling is a requirement, horizontal partitioning allows the incremental addition of hardware to provide higher throughput and scalability to match user demands.

Replication is a fundamental way to keep system availability towards failures. Its implicit redundancy makes data accessible from other replicas and turns the loss of data very improbable, thus improving reliability. Large-scale solutions often geo-replicate data spreading replicas across distant physical locations, keeping the data closer to the

users to improve latency. Simultaneously, a higher complexity development is required to assure some kind of consistency between replicas and restrict the visibility of operation effects to the users.

A Consistency Model defines an admissible order of events in a spectrum of strong to weak guarantees and consequently what users can observe depending on the defined order. Strong consistency models trade availability for consistency, and usually present higher response times. In large-scale platforms, weaker consistency models are usually employed to provide high availability and low latency for response times.

The set of nodes in these systems is not static and must be coordinated to sustain node additions and removals, either implicitly by failures or explicitly by an administrator. A Cluster Membership protocol with a global known membership is unreasonable due to its expensive up-to-date requirements in cases where the set of processes is extensive. Protocols using epidemic (gossip) broadcast are more feasible since they deal with partial views allowing a better node and network resource-saving.

This thesis aims at providing assistance in dealing with these tensions and inherent tradeoffs in the design choices of distributed systems, with special emphasis on distributed datastores. In particular, we will use the popular NoSQL Cassandra datastore system as the use case.

1.2 Problem Statement

The main issue in such distributed databases is the inter-dependency of components derived from monolithic implementations, hindering the independent evolution of the underlying modules or exploring different implementation alternatives, which leads to different limitations:

- Individual module scaling, swap or validation becomes impractical;
- The whole datastore redeployment is an obstacle for continuous deployment;
- Scalable development is sacrificed restricting different teams to focus on a specific and independent functional area;
- Unbound components are difficult to understand, promoting a lower quality of code and solution.

As an example of such limitations, in the specific case of Cassandra we can primarily identify that the messaging model is tightly coupled with many of the architecture components:

- Membership (Gossip and Heartbeat messages);
- Conflict Resolution (Read-repair and Hinted handoff messages)
- Read and Write Operations (Mutations)

- Lightweight Transactions (Paxos messages)

This work will propose a modular version of the NoSQL Cassandra key-value store to tackle these challenges, presenting a re-usable component solution with plug-and-play capabilities. This approach should make possible, among other aspects, to switch between a set of multiple replication algorithms with different guarantees, potentially at run-time. The final solution will be experimentally benchmarked against the baseline Cassandra system using FMKe[15].

1.3 Document Organization

Chapter 2 discusses related work including more detailed coverage of current key concepts used in distributed database solutions.
Chapter 3 reports the proposed work and work plan.

Chapter 2

State-of-the-Art

This chapter will approach some of the main topics in distributed database systems related to the work to be conducted during the elaboration of the thesis, being organized as follow:

Section 2.1 presents data partitioning criteria and schemes;

Section 2.2 addresses different techniques used in replication protocols and identify their trade-offs;

Section 2.3 discusses multiple consistency models, with a greater focus in weak consistency models which provides greater availability and latency conditions;

Section 2.4 describes cluster membership protocols;

Section 2.5 provides an overview of related systems;

Section 2.6 presents and discusses multiple Cassandra key concepts.

2.1 Data Partitioning

Partitioning consists of splitting the original set of data in multiple parts according to some criteria. In the context of DDBMS, multiple partitions can be distributed across a cluster to reduce node contention, spreading the load, improving scalability, availability or even manageability. There is no default strategy that is better than every other since such techniques highly depend on the workload of applications.

2.1.1 Partitioning Criteria

Assigning data to a partition requires a computation based on a partition key to determining the target partition. This key is constituted by a single or multiple sets of attributes used to segregate the data across the cluster, based in one of the following principles:

- **Range**: The partitions are distinguished by key ranges and the data is assigned verifying which partition holds the interval where the partition key fits in. An example could be a cluster partitioned by the range of student numbers.

- **List**: A partition has a list of values assigned. A practical example is a set of partitions representing continents and their countries as admissible values.

- **Hash**: An hash function is applied to determine the target partition. An exact-match is served by exactly one node if the function is applied on the partitioning attribute, otherwise, the query must be processed by all nodes.

- **Composite**: A combination of the above methods where one is applied to partitioning the data and another to subdivide each partition. Consistent Hashing is a good example since it can be seen as an implementation of a hash function method to reduce the keyspace to a size that can be listed.

- **Round-Robin**: A naive distribution of data where the **i**th tuple of data is inserted in the (**i mod n**) partition, with **n** being the number of nodes in the cluster.

The fundamental factor is to choose one criterion that favors a uniform distribution of data, allowing a better resource distribution and avoiding hotspots.

2.1.2 Partitioning Schemes

A logical table or index can be decomposed in three ways:

- Horizontal division, often called Sharding, separates distinct data rows across multiple partitions placed in different nodes, using some criteria like a key range. This should be considered when a table grows indefinitely leading to a slower query response due to the quantity of data processed. Because only a node or a reduced set of nodes will be responsible for each value range, the set of processed data is significantly reduced, boosting insertions, deletions or searches. However, the chosen criteria must be accurate because multiple partition lookups can cause poor performance compared with the original single data partition.

- Vertical separation consists in distributing data columns into different tables, keeping a common attribute to allow rebuilding the original table if necessary. This technique is particularly useful if the partitioning is done by data usage principles, with much-used data columns separated from columns not used at all. Performance improvements on search and update queries are expected if a small set of partitions are used, due to the reduced amount of data processed. Insertions and deletions will require various operations in all partitions being more costly, respectively, by the splitting process and seeking on every table where the data is.

- Hybrid partitioning is a mixed approach that uses horizontal and vertical mechanisms together. Its usage is a particular case where there is a need for optimal access to vertical and horizontal fragments usually accessed.

2.2 Replication

Replication protocols are designed to provide fault tolerance by removing single points of failure, thus maintaining system availability. Considering calamity situations where an entire datacenter is affected, the given redundancy can prevent losses of essential data if the replicas are distributed in different geographic locations. This kind of replica arrangement, also known as Geo-replication, can decrease latency if replicas are placed closer to the users, reducing the communication overhead between them. The growth in the number of users and the respective scalability concerns are addressed by replication which can be used to balance the user requests among the replicas.

2.2.1 Replication Types

In many database systems, nodes can be distinguished by the type of client operations they can serve. Processes allowed to execute any operation are called Masters (primaries) and the ones processing just read requests are known as Slaves (secondaries). A Single-Master setup, also known as Primary-Backup, is composed by one Master node allowed to modify the state and Slaves, while Multi-Master structures are composed by multiple Master replicas able to process any kind of request.

The concrete execution of operations is another key aspect present in replication. Active replication strategy, initially proposed by Lamport under the name **state machine replication** [10] was designed considering the execution of each client request by all replicas, in a deterministic way. In contrast, passive replication approaches hold one primary site processing user demands and secondary replicas copying the resulting state of the primary executions.

2.2.2 Replication Schemes

Under full replication systems, each replica is equal and holds a full copy of the database, boosting read query response times since user requests can be served by any node. Higher levels of availability can be achieved allowing system operation even with just one active node. However, this technique is wasteful considering disk space for the complete copies and makes hard to achieve concurrency if strong consistency is used since all replicas have to be kept consistent which involves frequent and costly synchronization.

Partial replication solutions consume less disk fractions given that the replicas only contain a subset of the whole data maintained by the system. Dealing with distinct fractions of data allows the execution of parallel operations by different nodes improving load balancing but also increases scalability since the writes only need to be applied

to some nodes. This strategy combined with a precise replica geographic distribution provides lower response times enhancing user experience.

2.2.3 Update Propagation Strategies

When updates are processed on a replica it is necessary to propagate the operation to other replicas in the cluster. There are two alternatives:

- Eager Update approaches synchronize all the replicas within the data item context before the response is sent to the user. Consequently, all the copies have the same value and it is not possible to get stale data, being the main reason for its use to enforce strong consistency models. This synchronous behavior has scalability and fault tolerance issues, since it increases communication overhead with the client which has to wait for the successful execution in all replicas, and in cases where one replica gets unavailable the operation cannot succeed.

- Lazy Update technique uses asynchronous propagation on replicas, returning a response to the user as soon as the operation succeeds in one or a small configurable number of replicas. This method allows faster response times which is suitable for large-scale applications with scalability requirements. Since the replication process is done concurrently with the response, replicas will experience momentary divergence in their states allowing users to read stale data and making it difficult to hold state invariants. For this reason, this kind of concept is mostly used in weaker consistency models.

2.3 Consistency

Consistency is a term that is often misunderstood between two important concepts, the CAP theorem[4] in distributed systems and the ACID transaction properties in databases. In both cases the "C" acronym stands for Consistency, however, in the first it is related with what users can observe and on the second means a transaction has to preserve integrity constraints. In this document, we are interested in the former.

2.3.1 CAP Theorem and its implications

CAP is used to explain trade-offs and asserts that a distributed data store can simultaneously provide at most two of the following properties:

- Consistency - Any user should have a consistent view of the data. This property is related with strong consistency models;

- Availability - Every request has a response (even if it doesn't contain the most recent data);

- Partition Tolerance - System keeps operating correctly in the presence of network partitions.

However, in distributed environments, partition tolerance cannot be sacrificed since we can't assume a network that never drops or delay messages and whose nodes never crash. For that reason, many academic research[3][8] revisited the theorem affirming partition tolerance as a requirement. Therefore, based on the application requirements programmers have to opt for Consistency (CP) or Availability (AP). Large-scale web services focused on user experience frequently choose availability to provide low communication overhead, whereas traditional applications using databases with ACID guarantees favor consistency providing to the user a consistent view of the system.

Consistency models define a set of safety conditions for a replication algorithm, restricting the states that can be observed by a client given the system operations and his own local history. The implementation overhead of each model decreases as the models become "weaker"since these models enforce fewer restrictions on the values that can be observed by the users.

2.3.2 Strong Consistency

Systems relying on strong consistency guarantees, ensure that users will always observe operation effects in the same order (corresponding to a consistent state). To execute an operation at least a quorum of replicas has to be contacted, thus avoiding the possibility of returning old values. This is feasible in applications with consistent, up-to-date data requirements, and easier to reason about for application programmers but has a negative impact on the communication overhead and throughput.

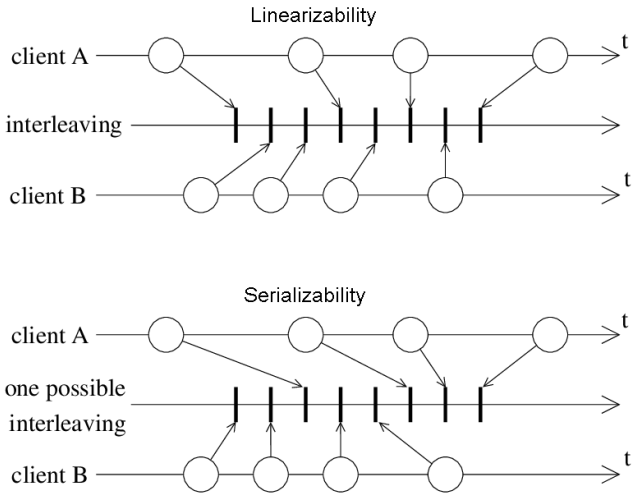

Figure 2.1: Interleaving of operations

Serializability

Following this model users should see the operations issued to the system in the same order, without concerning about the real-time order in which they were requested. To ensure a consistent view all replicas should appear to execute operations simultaneously, avoiding different reads to return different values.

Linearizability

A set of operations is linearizable if its execution corresponds to a sequential execution that respects the real-time issue order. In Figure 2.1[11], sequential consistency is representing one of many admissible orders that should be respected by the two clients, while in linearizability we only have one possible interleaving correspondent to the request order.

2.3.3 Weak Consistency

Under weak consistency principles, replicas can see operations in a different order which enables users to read stale values. Operations are executed in a set of replicas typically smaller than a majority with replication performed in the background, allowing responses to the client to be sent before the replication takes place. Geo-replicated systems and applications with strict performance requirements usually choose this kind of consistency due to the smaller communication overhead introduced, and because this approach allows to have a lower latency.

Eventual Consistency

Eventual consistency strategies only provide a liveness property which is the replica convergence when no write operations happen after some (unknown) amount of time. During this process replicas will have divergent states and users can read stale data. There are no underlying safety properties associated with this consistency model. To reach a consistent state, it may be necessary to employ some kind of conflict resolution protocol.

Causal Consistency

Causality exists conceptually when an operation is caused or influenced by another, which means users cannot see the effects of a second operation without being ensured that they will also observe the effects of the first in advance. For instance, a practical example consists in questions and answers on an online board where answers are causally dependent on the corresponding questions, which means users should be able to see every question before its correspondent answers. To provide this kind of guarantees, the implementation might require storing additional metadata to capture dependencies between related operations.

Causal+ Consistency

Causal+ is the strongest weak consistency model available in light of the CAP theorem. It fundamentally is achieved by enforcing causality between related operations and employing conflict resolution protocols to solve inconsistencies raised by concurrent writes.

2.3.4 Conflict Resolution Protocols

As a consequence of using replication protocols under weak consistency models, conflicts can occur when multiple replicas concurrently change the same (or related) data items. These conflicts are usually detected using timestamps or vector clocks and can be solved using reconciliation techniques, where all replicas update their local visions about the conflicting items and produce a final consistent state. Based on the application semantics, multiple practices can be employed to reach convergence:

- Client-based policies expose the divergences to clients and let them decide what is the correct and adequate value that will be written back to the system
- Last-writer wins will choose a deterministic criteria like time to choose the converged item state, dropping all the other concurrent values. In large-scale distributed systems vector clocks are the common approach to define a relative order among operations since local clock synchronization is difficult to achieve.
- Merge procedures will collect all the divergent states and show them as a combined single value. This is commonly used in e-commerce applications for the shopping cart and collaborative editing using CRDTs[14] (Conflict-Free Replicated Data Types).

2.4 Membership

Node coordination has an extreme relevance for the cluster efficiency. Centralized techniques often establish a bottleneck due to the dependency on a single node to forward all the requests but also present a single point of failure which can compromise availability. To use decentralized solutions nodes need to be aware of the underlying network and processes. Global knowledge memberships can be expensive if the system cardinality is considerable and inconstant leading to significant overhead for storing nodes information and ensure they have the most updated view of the system, whereas partial views of the system might offer a resilient solution.

2.4.1 Overlay Networks

Overlay Networks are abstractions of physical networks where nodes establish neighbor relationships materialized by virtual links. In these networks, nodes should know a low

number of processes which help in reducing the membership management overhead. Overlay classification depends on the degree of centralization and network topology, which is commonly chosen depending on the amount of churn[1].

Degree of centralization

Centralized techniques often use a central node to share information about the cluster members with newer nodes. They are easier to build than decentralized solutions but compose a single point of failure with the drawbacks mentioned above.

Decentralized solutions don't rely exclusively on dedicated nodes, trying to make every node equal like in peer-to-peer systems. This allows increased scalability since every node can handle requests or route them to others, but have higher network overhead since flooding is required to spread the queries over the cluster. Some systems use super-peers as nodes with higher responsibilities usually chosen for their heterogeneous advantage capabilities, providing a balance between a centralized solution and the autonomy of distributed environments.

Network Topology

Unstructured networks are formed by arbitrarily chosen links between nodes. Since every node has just a partial view of the network they have to spread information in an epidemic way to reach everyone in the cluster, reducing network efficiency. However, their big advantage is the ability to handle the entropy caused by nodes entering or leaving the system, since there are no centralized node indexes.

Structured overlays overcome the communication overhead of unstructured overlays since nodes established neighbor relations in a strategic way which makes them responsible for a specific part of the data, usually forming a distributed hash table (DHT). A common procedure consists in splitting the data identifiers range in uniform portions to distribute the data evenly by all nodes and applying hash functions to determine which node has a specific content. This kind of behavior combined with decentralized solutions allow every node to perform one-hop routing, thus avoiding flooding practices. However, under high churn circumstances performance is affected because it is necessary to re-distribute the identifier range for the actual cluster. A better approach in this scenario is the consistent hashing which on average only needs to remap K/N identifiers, with K as the number of keys and N the number of nodes.

2.4.2 Peer-to-Peer Systems

Peer-to-Peer networks are an example of decentralized systems where each peer implements client and server features and cooperate while sharing their resources. Peers are autonomous in an organizational point of view making necessary the use of a rendezvous

[1] Churn measures the number of nodes entering or leaving a cluster per unit of time

server as a meeting point. The addition of nodes makes the system scale since more bandwidth, computation and storage capacity is available. Also, this kind of systems is harder to attack compared with client-server architectures, due to the peers' heterogeneity and the fact there is no single point of failure.

2.5 Storage Systems Overview

Dynamo[6] is a highly available noSQL key-value storage system which provides eventual consistency to replicated data using a quorum-like protocol with configurable read and write parameters. Conflict resolution is made under read operations time using vector clocks. Its membership is totally decentralized adopting a gossip-based protocol to deal with node liveness information and consistent hashing to provide incremental scalability. The successor, DynamoDB, provides strong and weak read consistency levels.

Riak[2] is a scalable key-value store inspired by Dynamo. In general, it follows the same architecture but uses an exchangeable storage component and a richer query model that includes full-text search, secondary indexing and map reduce operations support. Conflict resolution uses a fine-grained causality tracking mechanism known as DVV and there is a support to concurrent operations merge through CRDTs[14].

Redis[13] is a key-value database which can operate as a cache, performing in-memory operations, or as a store with persistence achieved over snapshots or an append-only file. It supports partitioning through sharding with node selection managed by a proxy (Twemproxy) or directly from the client. Replication follows the master-slave model and by default is done asynchronously with eventual consistency guarantees. It has a synchronous replication mode, however, does not provide strong guarantees since acknowledge writes could be lost during a failover. Conflict resolution is done using epochs to control the versions of events where the greatest number succeeds.

ChainReaction[1] is a geo-replicated key-value store that offers full replication and causal+ consistency guarantees through a new variant of chain replication. This variant innovates leveraging the multiple replicas in the chain to perform read requests when the dependencies are stable while keeping metadata overhead low, instead of using exclusively the tail node to perform read requests as in the original algorithm.

C^3[7] is a partial replication algorithm which balances the execution of operations between local and remote datacenters to provide lower data visibility times. Metadata usage is reduced since it is necessary just to allow concurrency in the execution of remote operations. It supports client data migration between datacenters to provide lower response times.

Blotter[12] is a transactional geo-replicated storage system which combines Paxos consensus protocol to perform state machine replication between multiple DC and a concurrency protocol with Non-Monotonic Snapshot Isolation semantics to provide low-latency transactions on local DC. The underlying concurrency protocol relaxes the Snapshot Isolation (SI) guarantees allowing transactions to see snapshots which reflect writes made between their start and commit time, avoiding the tracking of the start times.

2.6 Cassandra: A Case Study

Cassandra[5, 9] is a NoSQL distributed wide-column storage system influenced by Dynamo which enforces high availability and incremental scalability. It was designed to support the Facebook inbox search feature that requires high write throughput to serve billions of users. At this scale, failures are a rule and not an exception, requiring the system to be resilient to high churn conditions while maintaining low communication overhead. Its architecture natively supports multiple data-center configurations and offers tunable consistency to control the latency and data freshness. We now present some of the techniques used in its key components.

Data Model and Partitioning

Cassandra organizes data in tables using a horizontal partitioning scheme and employs a distributed hash table to allow efficient data retrieval. Each row has a primary and partition keys, respectively identifying univocally the row and the attributes used to determine where it will be stored. In addition, clustering columns are used to sort the partitions order as shown in Figure 2.2.

```
CREATE TABLE user_videos (
   userid uuid,
   added_date timestamp,
   videoid uuid,
   name text,
   preview_image_location text,
   PRIMARY KEY (userid, added_date, videoid)
);
```

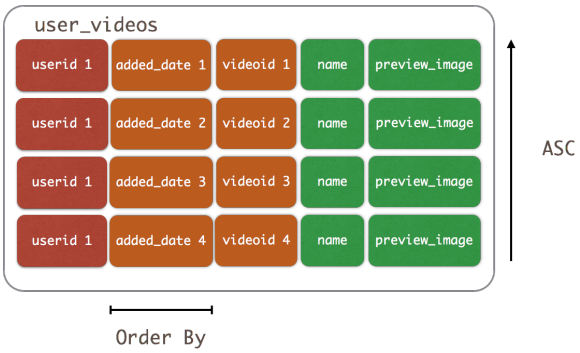

Figure 2.2: *user_videos* table sorted by *added_date* and *videoid*

Partitioners are functions used to determine how data is distributed in the cluster, typically, applying a hash to the partition key producing a token used to assign the row to a node. There are three strategies available:

- Murmur3Partitioner is the default which uses a technique called MurmurHash to produce a 64-bit token value;

- RandomPartitioner uses a cryptographic MD5 hash generating a 128-bit token;
- ByteOrderedPartitioner is included for backward compatibility and uses key bytes as criteria. Is not recommended since it can cause hotspots in the cluster

Using the first two will lead to an evenly distributed cluster with a ring structure where each node is responsible for a single contiguous token range. However, this strategy will require a token calculation and assignment for each node, which is expensive in a cluster composed by hundreds of nodes and worsened by high churn conditions.

The introduction of virtual nodes allows a fine-grained distribution of data to solve this problem. Each node contains a configurable number of virtual nodes responsible for different token ranges, or in other words, each node has multiple non-contiguous ranges of tokens. They allow automatic token calculation and assignment, cluster rebalancing, and a faster recovery and bootstrapping processes when replication is active since multiple nodes can transfer their copies.

Replication

Before presenting the different replication strategies provided by Cassandra it is important to understand the notion of a Snitch. A snitch is responsible for informing the system about the network topology and calculate which datacenter and rack nodes belong to it, allowing an efficient routing and data distribution. Although many snitches exist, all of them use a dynamic layer to monitor the read latency and avoid routing requests to overloaded nodes.

In Cassandra all the nodes can respond to queries, having no master or slave nomenclatures. Replication is defined per keyspace which is a namespace typically used to separate data partitions with different replication requirements. This way tables on different keyspaces can be under different replication strategies and replicated by a different number of nodes and datacenters. The total number of row copies is also defined for each keyspace and is known as the replication factor. For instance, a keyspace defined with a replication factor of 2 means every row on that keyspace is replicated on 2 nodes. Two replication strategies are available:

- SimpleStrategy doesn't have any locality information about the nodes and is intended to be used on single datacenter and rack clusters. The original row is placed on the node determined by the partitioner and the additional copies are placed on the next nodes clockwise in the ring
- NetworkTopologyStrategy should be used in multiple racks and datacenter configurations. This strategy uses the snitch information about the network topology and tries to place multiple copies in distinct racks because nodes in the same rack may fail simultaneously

Consistency

Cassandra offers eventual consistency for a highly available system and has a tunable configuration to provide stronger guarantees if needed. Consistency level could be defined globally by cluster or per-operation an its defined by the number of replicas (N) that have to successfully acknowledge a read (R) or write (W) operation. For instance, if the condition 2.1 is valid then the system is operating under eventual consistency, otherwise if condition 2.2 is true, it is operating under strong consistency guarantees.

$$R + W <= N \tag{2.1}$$

$$R + W > N \tag{2.2}$$

Multiple consistency levels are available for single and multiple datacenter configurations, varying from ALL where all replicas responsible for a respective partition in any datacenter have to acknowledge the operation, to ANY where one replica has to confirm the operation even if it is not responsible for the token.

Concurrent operations in items with strict read and write sequential order requirements can be done relying on lightweight transactions. This kind of transactions offer linearizable consistency guarantees and is implemented using Paxos consensus protocol, emulating isolation at a similar level provided by the serializability in relational databases.

When multiple replicas perform concurrent writes under weak consistency guarantees they could end up having different local values leaving the system in an inconsistent state. This divergence could be resolved in one of three different mechanisms:

1. Read repair is performed when inconsistencies are detected in a read operation path. The node which receives the request for a data item asks the replicas for a digest, comparing each one for inconsistencies, and writes the latest version in outdated replicas. This process is done synchronously so the client has to wait for its completion.

2. Hinted handoff is a mechanism used to allow write operations to succeed when one or more replicas responsible for a partition are down. The coordinator stores a hint which is information about the failed replicas and the respective data item, to allow a later synchronization when replicas recover.

3. Anti-entropy repair is an operation triggered manually that builds Merkle trees on every replica and compares them to find differences. Merkle trees are a faster way to discover inconsistencies without requiring the coordinator to download each replica data set, performing comparisons in multiple branches instead checking every data item.

Membership

Cassandra has a structured network where nodes are responsible for portions of data, forming a ring. They all have the same roles as in peer-to-peer systems and use the Scuttlebutt gossip protocol to spread state information about themselves and the nodes they know. Nodes joining a cluster use a seed node which is a regular node just with the privilege of helping new nodes on the bootstrapping process, sharing information about the cluster. This seed node is not a single point of failure since it is recommended to have more than one seed node in each datacenter.

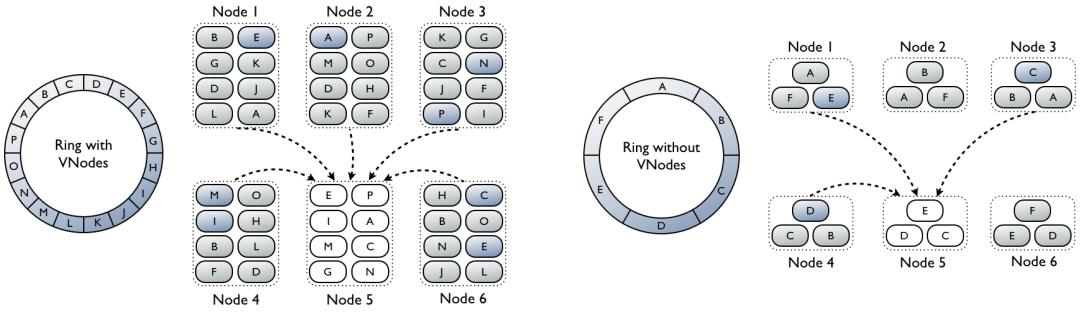

Figure 2.3: Bootstrapping process on a ring with and without VNodes

Virtual nodes were introduced to deal with high churn conditions. Each physical node contains virtual nodes responsible for non-contiguous portions of data. As illustrated in Figure 2.3, VNodes have a finer granularity on data distribution which allows more nodes to participate in bootstrapping or recovery procedures, increasing performance. In the case nodes leave the cluster permanently the other nodes will handle the load of its partitions. Consistent hashing is another technique employed by Cassandra to deal with the membership entropy. In regular conditions, almost all keys will need to be remapped since the mapping between tokens and nodes use a modular operation. With this mechanism only K keys of N nodes are reassigned, thus minimizing cluster reorganization.

Chapter 3

Proposed Work

3.1 Proposed Solution

To address the problems described in Chapter 1.2 derived from monolithic approaches employed on systems design, we will present a modular solution of the Cassandra where independent components can be exchanged without compromising other modules. This contribution goes beyond simply organizing the implementation of Cassandra, as it will allow future works to reuse or introduce elements, and validate the solutions against a highly available and scalable baseline solution.

Overcoming this first challenge will allow, for instance, to swap the standard replication component by new components exploring different strategies which have to take into consideration the low-latency requirements of the large-scale internet applications.

To demonstrate this we would to implement modules to emulate the behaviour of the following solutions:

- Blotter[12], which provides strong consistency semantics using a new concept of isolation that offers low-latency transactions.

- C^3[7], which enforces causal+ consistency guarantees with minimal metadata overhead.

To allow run-time switching between both strategies we have to ensure that all replicas are in a quiescent state before changing the replication protocol. For that reason, it may be necessary to stop request processing for a brief period to replace the replica strategy.

A possible extension to this work could comprise a dynamic switch between the two replication protocols adaptive to the network and cluster load conditions.

3.2 Validation

The evaluation of this solution will be performed using the benchmark tool FMKe and should allow, in a first phase, to compare the throughput and latency obtained between the baseline Cassandra solution and the modular version. Since the loosely coupled approaches often require more data exchanged between component interfaces, it is expected that the modular version offers a lower performance.

In a second phase, which corresponds to the conclusion of the implementation of the replication protocols, the performance of both solutions will be compared in terms of throughput and latency to validate its effectiveness. The implementation effort will also be measured, using time and number of code lines as main metrics.

3.3 Work Plan

Task	Start	End	Weeks
A - Modularization	1 March	10 May	10
Design			2
Implementation			4
Validation			4
B - Replication Components	10 May	2 August	12
Implementation			8
Validation			4
C - Final Evaluation	2 August	30 August	4
D - Writing	30 August	23 September	3.5

Table 3.1: Work Schedule

Task A - The modularization process corresponds to the component separation in independent modules and its evaluation against the baseline Cassandra implementation.

Task B - This phase comprehends the implementation and incorporation of the two replication strategies in Cassandra. It also comprises the building of the run-time switch mechanism and validation of its efficiency.

Task C - The final evaluation includes exhaustive final tests and their correlation with the strategies adopted

Task D - In this final stage we elaborate the final document

Month/Week	March				April				May				June				July				August				September			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Modularization																												
Design																												
Implementation																												
Validation																												
Replication comp.																												
Implementation																												
Validation																												
Final Evaluation																												
Writting																												

Bibliography

[1] S. Almeida, J. a. Leitão, and L. Rodrigues. "ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication." In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: ACM, 2013, pp. 85–98. ISBN: 978-1-4503-1994-2. DOI: 10.1145/2465351.2465361. URL: http://doi.acm.org/10.1145/2465351.2465361.

[2] Basho. *A Technical Overview of Riak KV Enterprise*. 2015. URL: http://basho.com/wp-content/uploads/2015/04/RiakKV-Enterprise-Technical-Overview-6page.pdf.

[3] E. Brewer. "CAP twelve years later: How the "rules" have changed." In: *Computer* 45.2 (2012), pp. 23–29. ISSN: 0018-9162. DOI: 10.1109/MC.2012.37.

[4] E. Brewer. "Towards robust distributed systems." In: Jan. 2000, p. 7. DOI: 10.1145/343477.343502.

[5] Datastax. *Apache Cassandra*. 2019. URL: https://docs.datastax.com/en/cassandra/3.0/.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's Highly Available Key-value Store." In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 205–220. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294281. URL: http://doi.acm.org/10.1145/1294261.1294281.

[7] P. Fouto, J. Leitao, and N. Preguica. "Practical and Fast Causal Consistent Partial Geo-Replication." In: Nov. 2018, pp. 1–10. DOI: 10.1109/NCA.2018.8548067.

[8] C. Hale. *You Can't Sacrifice Partition Tolerance*. 2010. URL: https://codahale.com/you-cant-sacrifice-partition-tolerance/.

[9] A. Lakshman and P. Malik. "Cassandra — A Decentralized Structured Storage System." In: *Operating Systems Review* 44 (Apr. 2010), pp. 35–40. DOI: 10.1145/1773912.1773922.

[10] L. Lamport. "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems." In: *ACM Trans. Program. Lang. Syst.* 6 (Apr. 1984), pp. 254–280. DOI: 10.1145/2993.2994.

[11] J. Lehtinen and J. Tuominen. "A design for a distributed file system featuring peer-to-peer caching." In: (Jan. 2005).

[12] H. Moniz, J. a. Leitão, R. J. Dias, J. Gehrke, N. Preguiça, and R. Rodrigues. "Blotter: Low Latency Transactions for Geo-Replicated Storage." In: *Proceedings of the 26th International Conference on World Wide Web*. WWW '17. Perth, Australia: International World Wide Web Conferences Steering Committee, 2017, pp. 263–272. ISBN: 978-1-4503-4913-0. DOI: 10.1145/3038912.3052603. URL: https://doi.org/10.1145/3038912.3052603.

[13] Redislabs. *Redis Documentation*. 2019. URL: https://redis.io/documentation.

[14] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. "Conflict-Free Replicated Data Types." In: vol. 6976. July 2011, pp. 386–400. DOI: 10.1007/978-3-642-24550-3_29.

[15] G. Tomás, P. Zeller, V. Balegas, D. Akkoorath, A. Bieniusa, J. a. Leitão, and N. Preguiça. "FMKe: A Real-World Benchmark for Key-Value Data Stores." In: *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '17. Belgrade, Serbia: ACM, 2017, 7:1–7:4. ISBN: 978-1-4503-4933-8. DOI: 10.1145/3064889.3064897. URL: http://doi.acm.org/10.1145/3064889.3064897.