Paulo Ricardo Almeida Moita

Bachelor Degree in Computer Science

Modular and Adaptive Key-Value Storage Systems

Dissertation 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

April, 2020

Modular and Adaptive Key-Value Storage Systems

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To my friends and family which helped with strong and helpful words, keeping me focused during the course.

Acknowledgements

Over the last years, I've met incredible professors and assistants who took part in my learning process. The list is extensive, so I would like to start acknowledging the FCT Nova institution and especially the Computer Science department for all their teaching and contributions to the foundations of my knowledge.
Then, I would like to express gratitude, in particular, to João Leitão and Pedro Fouto for being my mentors during the execution of this project and to the other Distributed Systems Group members for the friendly work environment.
Last but not least, my group of friends "SAL" and my family, which I'm very thankful for their support and understanding.

This work was partially financed by the Fundação para a Ciência e Tecnologia (Portuguese Foundation for Science and Technology) through the project NG-STORAGE (New Generation of data STORage And manaGement systEms) with grant ID #PTDC/CCI-INF/32038/2017 and the NOVALINCS Laboratory for Computer Science and Informatics research center (UID/CEC/04516/2013), which I would like to thank for their support.

Experiments presented in this project were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

Abstract

Large-scale Internet applications deal with an increasing number of users and consequently of data, requiring capable support infrastructures to follow their 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 application's operator revenue.

For this, these systems have, among other properties, to provide high availability, fault tolerance, and simultaneously keep the user and system data consistent, taking in consideration that is not possible the coexistence of all these properties (as famously captured by the CAP theorem). As such, each application has to opt for some guarantees usually provided by tightly coupled components derived from monolithic architectures of data storage systems, which do not allow to change its specification in real-time.

In this document, we propose a novel design for noSQL distributed datastores, based on the popular NoSQL Cassandra storage system, which allows modification (or swap) of its components using a modular methodology. Following this approach, we illustrate its flexibility by developing two replication protocols with different consistency guarantees and support to geo-replicated scenarios under its modular design.

Keywords: Distributed storage systems, user experience, consistency, availability, fault tolerance, monolithic, modular, geo-replication, Apache Cassandra, CAP theorem

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 perda de receitas para o operador das aplicações.

Para tal estes sistemas devem, entre outras propriedades, garantir alta disponibilidade, tolerância a falhas, e ao mesmo tempo manter os dados dos utilizadores e sistemas consistentes, tendo em conta que não é possível a coexistência de todas estas propriedades (tal como capturado pelo notável teorema de CAP). Como tal, cada aplicação tem de optar por determinadas garantias fornecidas por componentes que se encontram fortemente acoplados, derivados de arquitecturas monolíticas de sistemas de armazenamento, não sendo possível a alterar a sua especificação em tempo real.

Neste documento, propomos um desenho inovador para sistemas de armazenamentos distribuídos não relacionais, baseado no popular Apache Cassandra, que permite maior flexibilidade na alteração (ou substituição) dos seus componentes através de uma metodologia modular. No seguimento dessa abordagem, ilustramos a sua flexibilidade através do desenvolvimento de dois protocolos de replicação com diferentes garantias de consistência e suporte a cenários geo-replicados de acordo com o seu desenho modular.

Palavras-chave: Sistemas de armazenamento distribuído, experiência de utilizador, consistência, disponibilidade, tolerância a falhas, monolítica, modular, geo-replicação, Apache Cassandra, teorema de CAP

Contents

List of Figures

List of Tables

Acronyms

ACID	Atomicity, Consistency, Isolation, Durability.
ANTLR	ANother Tool for Language Recognition.
API	Application Programming Interface.
CAP	Consistency, Availability, Partition Tolerance.
CRDT	Conflict-free Replicated Datatype.
DB	Database.
DC	Datacenter.
DDBMS	Distributed Database Management System.
DHT	Distributed Hash Table.
DM	Data Manager.
DVV	Dotted Version Vector.
JMX	Java Management Extensions.
NMSI	Non-Monotonic Snapshot Isolation.
NoSQL	Non-relational database (or Not only SQL database).
RPC	Remote Procedure Call.

SI	Snapshot Isolation.
TM	Transaction Manager.
YCSB	Yahoo Cloud Service Benchmark.

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 ensure low latency and good user experience, since user abandon rate is directly related with page loading times[3]. Composed by hundreds (or potentially 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 the storage layer is a key applications' layer responsible to deal with most of these challenges, while efficiently dealing with larger and richer data sources. Good performance, and hence good user experience, is tightly bounded by low communication overhead, which can be accomplished adjusting key aspects of the underlying datastore design such as Partitioning Schema, 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 which is commonly referred as Sharding. In scenarios with an increasing traffic load where elastic cluster scaling is a requirement, horizontal partitioning allows the incremental addition of hardware to accommodate increasing amounts of operations (workload) due to the growing number of users, while retaining high throughput and low latency.

Replication is a fundamental way to ensure system availability despite the failure of individual components. 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 more complex development is required to assure some kind of data consistency between replicas and restrict the visibility of operation effects to the users (as to avoid anomalies in the data observed when interacting with the system).

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. This was captured by the CAP theorem that informally states that in a system where network partitions can happen, strong consistency and availability cannot be provided simultaneously.

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. However different system designs might use different alternatives, more suitable for the intended operation of the storage system as a whole.

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 data storage. In particular, we will rely on the popular NoSQL Cassandra datastore system as the basis for our work, as a starting point towards the proposal of a novel design of these systems.

1.2 Problem Statement

The main issue in current 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:

- The scaling, swap, or validation of components becomes impractical (due to their interdependence);

- The whole datastore redeployment is an obstacle for continuous deployment in the sense that new strategies have to be compiled together with the datastore;

- Scalable development is sacrificed restricting different teams[16] 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 a coupled model in many of the architecture components:

- Gossip: Gossip and Membership are tightly coupled, and we cannot use gossip to send other information than the Membership;
- Consistency: Only eventual consistency is available and resorts to the number of nodes acknowledging operations, so algorithms providing stronger guarantees could not be easily integrated without a large number of changes across many of the other components of Cassandra;
- Statements: Uses ANTLR parser generator where statements are previously compiled with the datastore, not allowing new operations to be defined at runtime.

This work will propose a modular version of a NoSQL distributed datastore, using as case study the Cassandra key-value datastore. Our novel modular design will tackle these challenges, presenting a re-usable component solution with plug-and-play capabilities. This approach makes possible, among other aspects, to switch between a set of multiple replication algorithms with different guarantees, potentially at runtime. The final solution was experimentally benchmarked against the baseline Cassandra system using YCSB[10].

1.3 Contributions

The result of this work provide the following main contributions:

- A novel design of a modular approach that makes a distributed datastore easier to modify being flexible enough to rely on different implementations of modules;
- A particular implementation of the Modular design applied to Apache Cassandra and its main components;
- An accessible way for future works to test the performance of distinct implementations of the same or multiple modules;
- Two modular implementations of existing replication schemes (C3 and Blotter);
- An experimental evaluation of the proposed solution and its implementation.

1.4 Document Organization

Chapter 2 discusses related work including more detailed coverage of current key concepts used in distributed database solutions and presents different approaches adopted by other relevant systems.

Chapter 3 considers different methodologies in the design of a Modular architecture and a rationale that leads to the proposed solution.

Chapter 4 specifies implementation details of the Modular version based on Apache Cassandra and the two validation modules.

Chapter 5 presents an experimental evaluation of the final solution. First we describe the setup and parameters used on the test bench, then we perform a comparative study between the baseline and modular versions performance. Also, we extend our experimental work to include the two replication modules with different consistency guarantees implemented.

Chapter 6 presents the conclusion with a summary and reflection about the produced work and recommendations about future works.

Chapter 2

State-of-the-Art

This chapter covers 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.

Section 2.7 introduces the MySQL architecture and particularly its modular and pluggable storage engines.

2.1 Data Partitioning

Partitioning consists of splitting the original set of data maintained by the storage system 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[30]. 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 Strategy

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[31]:

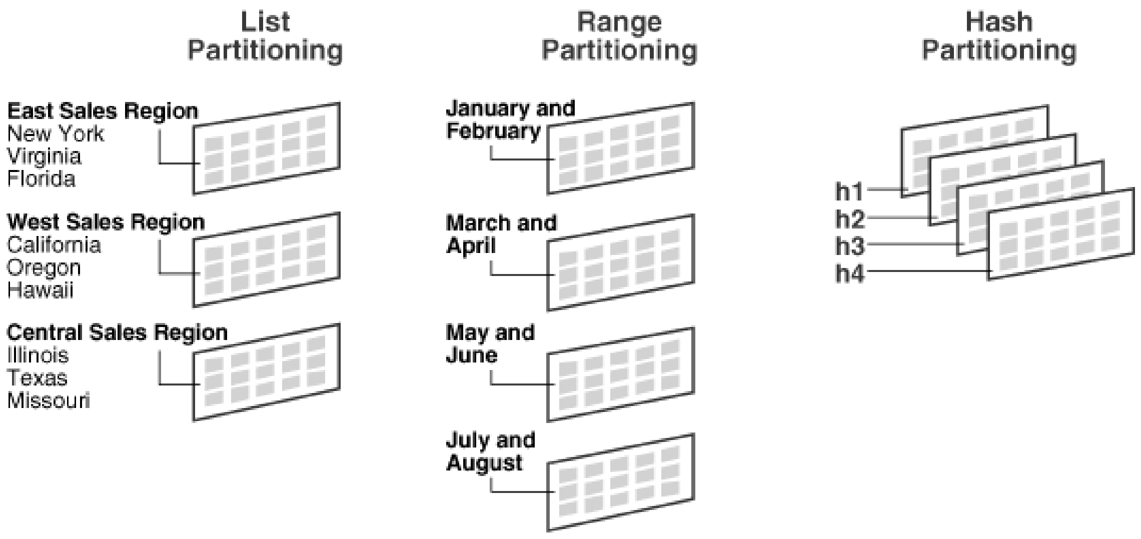


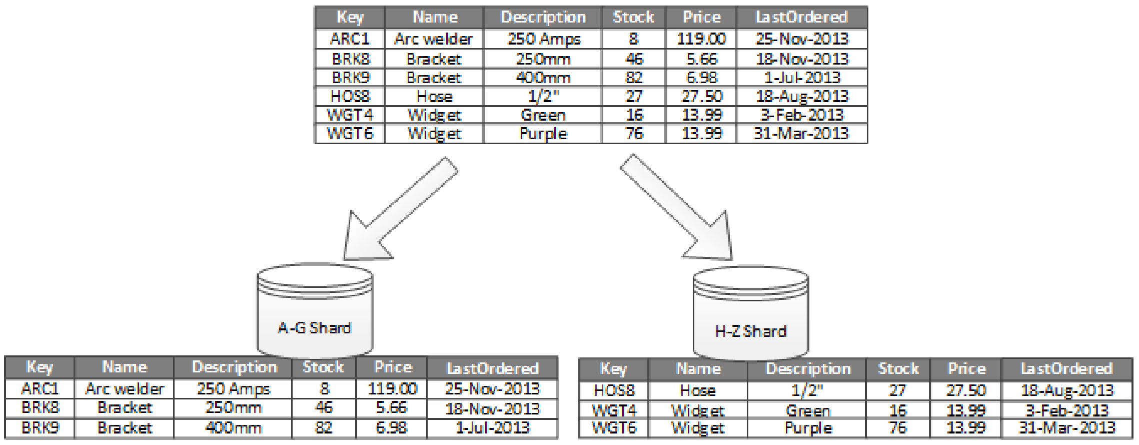
Figure 2.1: Range, List and Hash partitioning schemes

- **Range**: The partitions are distinguished by key ranges and the data is assigned verifying which partition holds the interval where the partition key falls in. Figure 2.1 shows an example of this, partitioning data in ranges composed by different month intervals;
- **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. This is typically employed for discrete 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 total 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

Partition schemes refer to how each table data item is logically divided inside partitions[25]. It is possible to decompose a logical table or index in three ways:



Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013

Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013

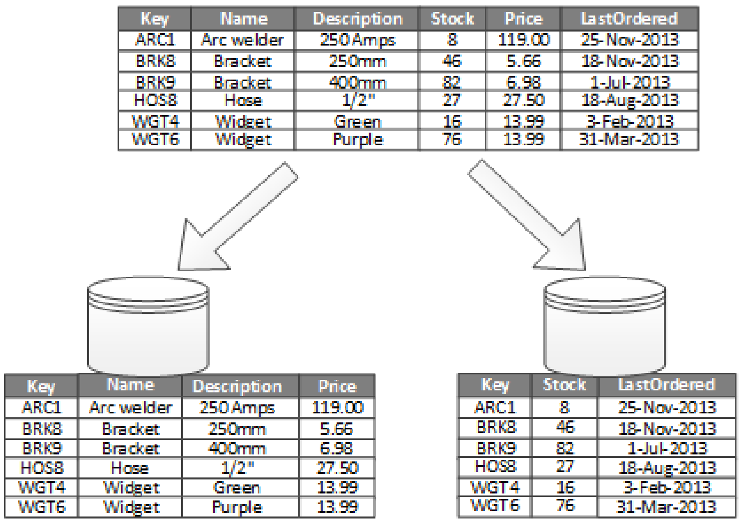
Key	Name	Description	Stock	Price	LastOrdered
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013

Figure 2.2: Horizontal partitioning by key ranges (A-G and H-Z shards)

- **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 the performance of insertions, deletions or searches. However, the chosen criteria must be accurate because multiple partition lookups typically leads to 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 more frequently accessed data columns separated from columns rarely 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 usually limited to particular cases where there is a need for optimal access to vertical and horizontal fragments usually accessed, since such as approach has a higher complexity.



Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013

Key	Name	Description	Price
ARC1	Arc welder	250 Amps	119.00
BRK8	Bracket	250mm	5.66
BRK9	Bracket	400mm	6.98
HOS8	Hose	1/2"	27.50
WGT4	Widget	Green	13.99
WGT6	Widget	Purple	13.99

Key	Stock	LastOrdered
ARC1	8	25-Nov-2013
BRK8	46	18-Nov-2013
BRK9	82	1-Jul-2013
HOS8	27	18-Aug-2013
WGT4	16	3-Feb-2013
WGT6	76	31-Mar-2013

Figure 2.3: Vertical partitioning by pattern of use (assuming key, name, description and price as more frequently used fields)

2.2 Replication

Replication protocols are designed to provide fault tolerance by removing single points of failure, thus maintaining system availability[29]. 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[28]. 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 that is the only one allowed to modify the state and Slaves, while Multi-Master structures are composed by multiple Master replicas able to process any kind of request (i.e., read and write operations).

The concrete execution of operations is another key aspect present in replication. Active replication strategy[20], initially proposed by Lamport under the name **state machine replication** [23] was designed considering the execution of each client request by all replicas, in the exact same order (operations that modify the state must be deterministic). In contrast, passive replication approaches hold one primary site processing user demands and secondary replicas copying the resulting state from the execution of operations by the primary (hence, operations can be non-deterministic).

2.2.2 Replication Schemes

In full replication systems[33], 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 of the full datasets in each replica 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 space 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 by 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[32]:

- 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 mechanisms use asynchronous propagation across replicas, returning a response to the user as soon as the operation succeeds in one or a small and configurable subset 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, which enables users to read stale data and making it difficult to hold state invariants (that might be essential for application correctness). For this reason, this kind of approach 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[8] in distributed systems and the ACID[19] transaction properties in databases. In both cases the "C" in the acronyms stand 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 a famous theorem that captures the inherent 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 researches consider that the most appropriate interpretation of the theorem should consider partition tolerance as a fundamental requirement[7][17]. 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[1] 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 provides a model that is easier to reason about for application programmers but has a negative impact on the performance, namely on communication overhead and throughput.

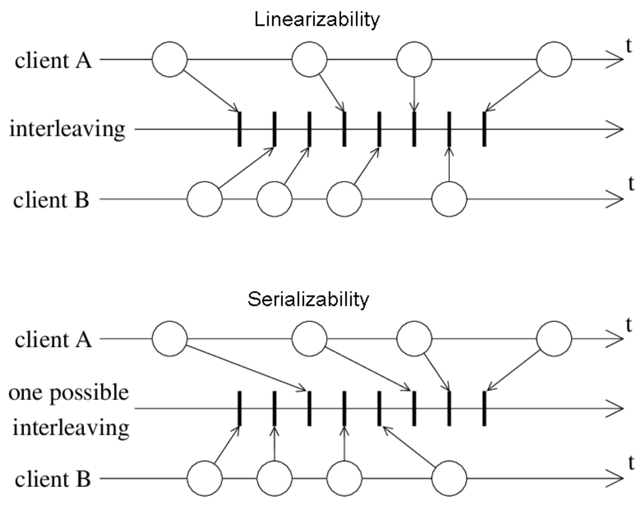


Figure 2.4: Interleaving of operations

Serializability

Following this model users should see the operations issued to the system in the same order, without special restrictions imposed on this order that derive from the real-time order in which they were issued by clients. 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 order in which operations are issued by clients. In Figure 2.4[24], 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 models, replicas can see operations in a different order among them 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 favor this kind of consistency due to the smaller communication overhead introduced, since this approach allows to have a lower latency, while ensures availability under network partitions.

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 or different 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. This implicitly captures the happens-before relationship defined by Lamport[22]. 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 replication protocol might be required to store additional metadata to capture dependencies between related operations.

Causal+ Consistency

Causal+ is the strongest weak consistency model available in light of the CAP theorem if each client only interacts with a single replica. It fundamentally is achieved by enforcing causality between related operations and employing conflict resolution protocols to solve inconsistencies raised by concurrent writes, ensuring the eventual convergence property that defines eventual consistency.

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, overwriting all previous divergent values;

- Last-writer wins employs a deterministic criteria such as the identifiers of the replicas which received client operations that left the system in a divergent state 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 use an application-dependent merge function to compute a combined single value. This is commonly used in e-commerce applications for the shopping cart and collaborative editing using CRDTs[37] (Conflict-Free Replicated Data Types).

2.4 Membership

Node coordination is highly relevant to ensure cluster efficiency. Centralized techniques often establish a bottleneck due to the dependency on a single (physical or logical) node to forward all the requests but also present a single point of failure which can compromise availability[18]. 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 do not 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.

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

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. 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 server as a meeting point. The addition of nodes makes the system scale since more bandwidth, computation, and storage capacity becomes 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[12] is a highly available non-relational 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[5] 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[34]

(Dotted Version Vectors) and there is a support to concurrent operations merge through CRDTs[37].

Redis[35] 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[2] 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[15] is an algorithm focused on partial and geo-replicated scenarios that provides causal+ consistency guarantees, implemented on top of the popular Cassandra storage system. Its design balance the execution of operations between local and remote datacenters in order to provide lower data visibility times, and decouples the metadata information necessary to tracking causality from update operations between datacenters. This is done using a causality layer to manage dependencies between operations in different datacenters instead of propagating them within operations. Cutting the amount of information disseminated, overhead is reduced and throughput increased significantly comparing to other systems offering causal consistency like Saturn[6].

Blotter[26] is a transactional geo-replicated storage system which combines Paxos consensus protocol to perform state machine replication between multiple DC and a new concurrency protocol providing Non-Monotonic Snapshot Isolation, to provide low-latency transactions on local DC. The underlying protocol, relaxes the Snapshot Isolation (SI) semantics allowing transactions to see snapshots which reflect updates made between their start and commit time. This property, also called forward freshness[4], cut some of the latency penalty present in ACID transactions while still provide strong consistency guarantees.

2.6 Cassandra: A Case Study

Cassandra[11, 21] is a NoSQL distributed wide-column storage system influenced by Dynamo which enforces high availability and 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 2.5.

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

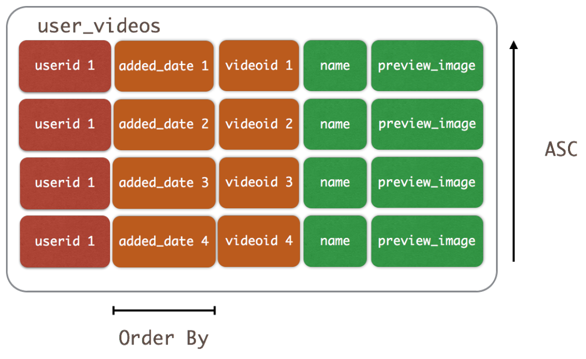


Figure 2.5: *user_videos* table sorted by *added_date* and *videoid*

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 emulate 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, we can have a system under weak consistency semantics if the condition 2.1 is valid, since it could be possible that a replica selected for a read operation does not have the last value written because it is being eventually updated, or emulating strong semantics if condition 2.2 is true, since at least one replica of a quorum of nodes participate in both operations and has access to the last value written. However it is important to mention Cassandra operations follow the BASE paradigm, so the latter condition does not guarantee single-key linerizable reads like in ACID transactions and thus not providing truly strong consistency[9].

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

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

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.

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 those 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.

Concurrent operations on items that require 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. To use them we must write the keyword IF on statements which automatically triggers a 4-round communication to provide strong consistency guarantees, however with a high latency penalty.

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[36] 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.

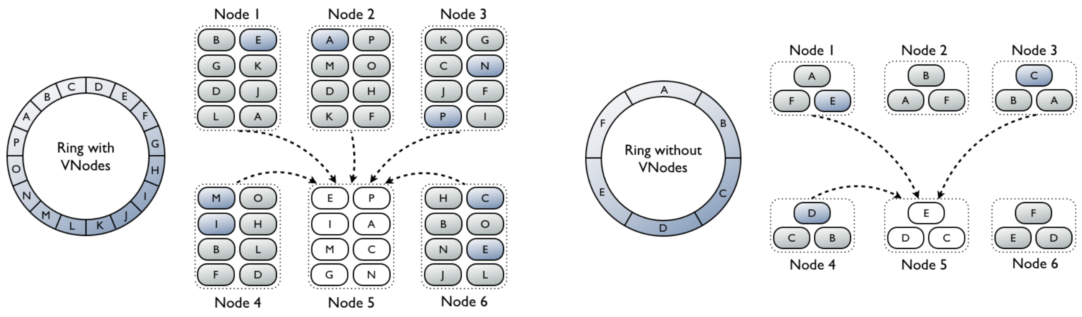


Figure 2.6: Bootstrapping process on a ring with and without VNodes

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 2.6, 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 a smaller number of keys need to be reassigned (i.e., only the keys in the reassigned VNodes have to be moved), thus minimizing cluster reorganization.

2.7 MySQL Modular and Pluggable Storage Engines

MySQL[27] is a relational datastore whose architecture, represented in Figure 2.7, comprehend a storage engine component built in a modular fashion, allowing custom implementations to be integrated as part of the system. As default it provides, along others, four principal kinds of engines:

- **InnoDB:** The default, which supports ACID transactions, featuring commit, rollback and crash-recovery capabilities. Data arrangement is optimized for queries based on primary keys. It replaces the MyISAM since version 5.5, which does not support transactions and foreign keys;

- **MyISAM:** Optimized for heavy read environments, it stores data using two files with entry points to records which are offsets from the beginning of the files. This index structure is particularly useful for fixed data formats since the rows have constant length;
- **Memory:** Stores data in volatile memory and thus should be used as a cache or temporary storage;
- **CSV:** This engine uses text files to store data using comma separated values

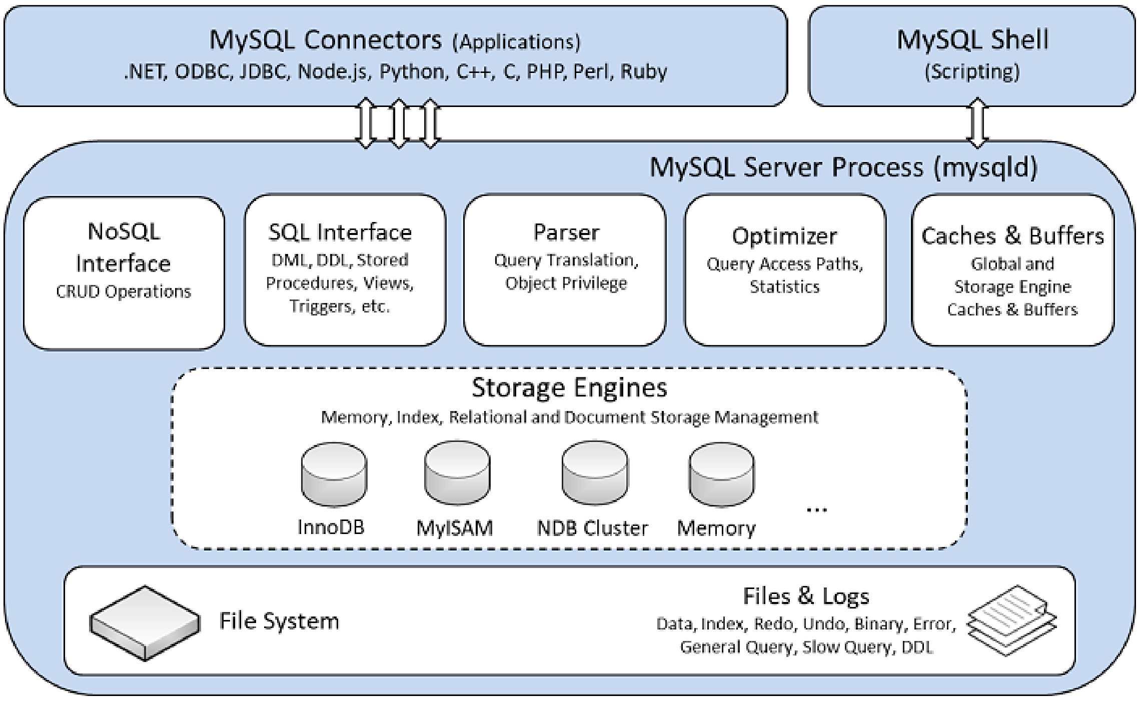

MySQL Connectors (Applications)
.NET, ODBC, JDBC, Node.js, Python, C++, C, PHP, Perl, Ruby
MySQL Shell
(Scripting)
MySQL Server Process (mysqld)
NoSQL Interface
CRUD Operations
SQL Interface
DML, DDL, Stored Procedures, Views, Triggers, etc.
Parser
Query Translation, Object Privilege
Optimizer
Query Access Paths, Statistics
Caches & Buffers
Global and Storage Engine Caches & Buffers
Storage Engines
Memory, Index, Relational and Document Storage Management
InnoDB
MyISAM
NDB Cluster
Memory
...
File System
Files & Logs
Data, Index, Redo, Undo, Binary, Error, General Query, Slow Query, DDL

Figure 2.7: MySQL Architecture

MySQL Server communicates with the storage engines through handlers, which are entry points for the concrete data storage and retrieval tasks. Developers can create custom storage engines implementing the handler interface, written in C language, where they specify the pointers to the methods that implement the engine operations. This interface, represented in listing 2.1, can be implemented incrementally, where we can start with a simple read-only engine and then add support for write operations.

```
typedef struct
  {
    const char *name;
    SHOW_COMP_OPTION state;
    const char *comment;
  enum db_type db_type;
  bool (*init)();
  uint slot;
  uint savepoint_offset;
  int  (*close_connection)(THD *thd);
  int  (*savepoint_set)(THD *thd, void *sv);
  int  (*savepoint_rollback)(THD *thd, void *sv);
  int  (*savepoint_release)(THD *thd, void *sv);
  int  (*commit)(THD *thd, bool all);
  int  (*rollback)(THD *thd, bool all);
  int  (*prepare)(THD *thd, bool all);
  int  (*recover)(XID *xid_list, uint len);
  int  (*commit_by_xid)(XID *xid);
  int  (*rollback_by_xid)(XID *xid);
  void *(*create_cursor_read_view)();
  void (*set_cursor_read_view)(void *);
  void (*close_cursor_read_view)(void *);
  handler *(*create)(TABLE *table);
  void (*drop_database)(char* path);
  int (*panic)(enum ha_panic_function flag);
  int (*release_temporary_latches)(THD *thd);
  int (*update_statistics)();
  int (*start_consistent_snapshot)(THD *thd);
  bool (*flush_logs)();
  bool (*show_status)(THD *thd, stat_prt_fn *print, enum ha_stat_type stat);
  int (*repl_report_binlog)(THD *thd, char *lfile_name, my_off_t end_offset);
  uint32 flags;
} handlerton;
```

Listing 2.1: Handlerton Interface

Engines can be loaded or unloaded, respectively, using INSTALL or UNINSTALL plugin statements while the server is running. These engines are defined by table, allowing the system to have multiple implementations running simultaneously on distinct tables. Plugins can be removed if none of its associated tables are opened, however table access becomes inaccessible. The application programmer must reason if the data is valuable, dumping table content, or simply deleting it before removing its correlated engine. Although MySQL architecture for storage engines represent a clear effort to use a modular approach, other datastore components follow rigid implementations which do not allow to swap its properties in an easy and pluggable way. Beside this limitation, it does not provide a module manager where, for instance, we can define start and end module routines.

Chapter 3

Modular Architecture

Decoupling modules implementation from the datastore allows a better separation of concerns and independent evolution whilst enable the efficient exploration of different implementation alternatives. Our solution should minimize the overhead introduced by the loosely coupled components derived from modularization of the architecture of a distributed datastore while taking into consideration the previous requirements of the baseline solution, such as scalability and availability. In this chapter we consider and discuss different approaches that could be a solution for a modular datastore architecture and, finally, we propose a solution that suits those demands.

3.1 Design Considerations

A module, or component, is a set of functions that contribute to the same logical purpose, such as, for example, a gossip module with functions responsible for exchanging data using an epidemic approach, or a storage engine module where functions related with data stored in disk, compaction or even cache can be provided.

The implementation of a module can be provided in advance, being compiled together with the datastore, or in runtime, where it is compiled separately and provided as a plug-in. The main difference is that the first requires the datastore execution to stop and recompile together with the new module implementation, whether the plug-ins can be attached without stopping the operation of the datastore system. Considering that we want to make possible an independent evolution and runtime swap of modules, and we don't want the datastore to stop, the plug-in technique suits better our objectives.

Communication between the datastore and a module, or even between modules could be done using remote procedure calls (RPC) or direct method calls. RPCs often introduce network overhead which could have a negative impact at the performance of the datastore and thus increase user perceived latency. In contrast, direct method calls require that modules that have dependencies in common, to expose their interfaces among them which can introduce code redundancy if each module has to contain the contract for the modules it depends. Taking in consideration we want to achieve a modular solution with the minimum overhead possible we opt for the latter, using a module manager where all interfaces are exposed and acts as a bridge for interaction between components.

We consider module validation as an orthogonal exercise to the contributions of this work, since every implementation is different and could require different validation techniques such as static or dynamic analysis.

3.2 Proposed Solution

Our modular approach allows extending an existing datastore in the sense that modules can be reused or replaced by ones with different implementations, even at runtime. Therefore we opt to use a pluggable approach with a module manager to load, unload and provide inter-module communication using direct calls. To force a contract between datastore implementation and the components to be loaded we have specified interfaces that should be implemented by different kinds of modules, taking in consideration that they should as generic as possible to allow different implementations.

Modules should be exported as Java Archive files (.jar) and its filename specified in a configuration file as described in listing 3.1, whose properties are read by a configuration loader on datastore initialization. Since the implementation is not known at runtime, the component manager needs to determine which classes or methods are accessible and provide a way of accessing them. For this requirement we use the Java Reflection API which allows us to get a candidate instance of a class, which is made definitive if it could be casted to the Module interface it should represent.

```
#In path please use / as last character
PATH: src/java/org/apache/cassandra/moita/modules/
#In modules use the jar extension (.jar)
DATAPLACEMENT: Placement.jar
FAULTDETECTOR: Detector.jar
GOSSIP: Gossip.jar
LOCATOR: Snitch.jar
MEMBERSHIP: Membership.jar
CONSISTENCY: Blotter.jar
```

Listing 3.1: Module Manager configuration file

To avoid a possible overhead introduced by a repeated use of the reflection techniques, which has to lookup inside each module every time we need to get an object instance, we use reflection only once to get a class which contains wrapper[1] functions. This class is stored in the manager and constitute an entry point common to all the components containing references to object instances, handlers and serializers to be registered on the datastore logic and also start and shutdown routines.

Any interaction between the datastore logic and the modules is then performed by getting the module representation on the manager and a method call to get the desired object instance.

3.2.1 Modules Definition

Since component interfaces should be as generic as possible, we carefully define each one considering operations that are common to many implementations. We now discuss and present each module and its expected behavior, covering the basic cases.

3.2.1.1 Failure Detector

The failure detector is responsible for identifying node failures in a cluster. Multiple algorithms can be implemented but all of them classify nodes using one of the following variants:

- **Boolean Nature** - A node can be alive or dead, respectively, corresponding to a True or False state;
- **Continuous Scale** - There is a suspicious or trust level about the state of each node.

Conventional detectors often classify cluster members using truth values making difficult to reason about how likely a node is about to fail, which could be a requirement in some applications. These requirements could be adaptations to network performance, workload or other conditions to meet the application demands. Using a numeric scale we can accomplish a fine-tune based on thresholds allowing different triggers to produce different behaviors and thus increasing flexibility. We now introduce a snippet with the generic API proposed:

```
public interface IFailureDetector{
    //Check if a node is alive based on a threshold (suspicious) level
    boolean isAlive(InetAddress ep)
    //Inform this module about incoming activity from an endpoint
    void report(InetAddress ep)
    //Remove an endpoint from the detection list
    void remove(InetAddress ep)
    //Inform listeners about an endpoint status
    void forceConviction(InetAddress ep)
    //Register a module interested in Fault Detection events
```

[1]A wrapper function is a subroutine whose main purpose is to call a second subroutine or a system call with little or no additional computation

```
    void registerFDListener(IFDListener listener)
    //Unregister a module from Fault Detection events
    void unregisterFDListener(IFDListener listener)
    //Define a threshold (suspicious) level to classify an endpoint status
    void setPhiConvictThreshold(double phi)
    //Get the current threshold level
    double getPhiConvictThreshold()
}
```

3.2.1.2 Gossiper

The Gossiper module is in charge of spreading information in an epidemic way, avoiding a centralized and dedicated service that constitutes a single point of failure. Depending on the implemented protocol, gossip members could share information with a partial or full set of nodes, which will present different computational demands and also distinct convergence speed. In our particular case, the employed algorithm shares at each second membership information about the cluster with a partial view containing application and heartbeat states.

Following this discussion we suggest the following generic API:

```
public interface IGossiper{
    //Allows to define which kind of messages are recognized by the protocol
    enum MessageKind{...}
    //Disseminate a message of a defined type. Variable-length arguments allow
    //flexibility between different messages with different number of arguments
    void sendMsg(MessageKind kind, Object... args);  //disseminate a message
    void handleMsg(Object msg);  //handle a received message
}
```

3.2.1.3 Membership

The Membership component maintains information about the cluster nodes. Initially, it was implemented together with the Gossiper module, however considering gossiper could be used to propagate information other than the membership we decide to create a separate module. Depending on the chosen peer information the amount of data exchanged can constitute a network overhead, so we opt to simple but generic details like status, heuristics, and statistics that can be used for reconciliation mechanisms, which can be summarized in two different categories:

- **Application State** - Information about network location (DC/Rack), status, heuristics about input/output pressure, disk space, schema versions;
- **Heartbeat State** - Process launch information and a monotonically incremented counter useful to agreement purposes.

In line with this variants we built a general API:

```java
public interface IMembershipManagement{
    //Get heartbeat information associated with an endpoint, which can include,
    //for example, a timestamp or a counter of heartbeats received
    IHeartBeatState getHeartBeatState()
    //Set heartbeat information associated to an endpoint
    void setHeartBeatState(IHeartBeatState newHbState)  //set HeartBeatState
    //Get the application state of an endpoint, which can include, for example,
    //network location, IO metrics, current partition schema version
    Set<Entry<ApplicationState,IVersionedValue>> states()
    //Set the application state of an endpoint
    void addApplicationStates(Set<Entry<ApplicationState,IVersionedValue>> val)
}
```

3.2.1.4 Locator

The Locator or Snitch is a component responsible to identify the physical network location of nodes concerning its datacenter and rack and inform core logic about the respective cluster topology. There are multiple ways to obtain this knowledge, which can be summarized in three variants:

- Using a discovery service similar to a rendezvous protocol;
- From a file with pre-defined information;
- Inferring the network topology using IP address octets.

A centralized discovery service can increase network usage since there are more packets exchanged with remote servers, which can be a penalty to the client's perceived latency but also a single point of failure. Decentralized approaches with a baseline topology are enough to build and provide reliable topology information without incurring in-network overheads. As a result, we opt for a base configuration file that allows the datastore to start with an initial topology while still support runtime node addition or removal. Snitch information is also used to perform optimizations like efficient route requests considering node placement and load but also to distribute replicas in distinct network groups to avoid correlated failures.

Following this concept we introduce the API:

```java
public interface IEndpointSnitch{
    //Get the network rack associated with an endpoint
    String getRack(InetAddress ep)
    //Get the network datacenter of an endpoint
    String getDatacenter(InetAddress ep)
    //Sort nodes considering network proximity to an endpoint
    //This usually corresponds to a sort by latency
    List<InetAddress> sortByProximity(InetAddress origin,
List<InetAddress> targets)
    //Compare two endpoints considering network proximity to an endpoint
    int compareEndpoints (InetAddress target, InetAddress ep1, InetAddress ep2)
    //This method allows to check if it's worth to merge two sequential queries
    //in one single range query, to enhance datastore performance
    //For example it could be useful to merge queries which use local
    //and remote DCs to use just the local one instead
    boolean isWorthMergingForRangeQuery(List<InetAddress> merged,
List<InetAddress> list1, List<InetAddress> list2)
}
```

3.2.1.5 Data Placement

This module is responsible for implementing logic that defines where data replicas are placed. Being aware of single or multiple datacenter deployments where racks or other physical groups of nodes can exist, implementations could explore this diversity to place replicas in distinct places and avoid localized points of failure. For instance, strategies that place data replicas in multiple racks are less prone to unavailability situations comparing to a within-rack system redundancies placements since nodes with higher rack affinity have larger failure co-relation.[13]

In tandem with these challenges we provide a generic interface:

```
public interface IAbstractReplicationStrategy{
    //Allows to create a new replication strategy which define where data
    //replicas will be placed. TokenMetadata contains token to endpoint mapping
    //and snitch contains information about physical network location of nodes
    IAbstractReplicationStategy create(String keyspace, TokenMetadata meta,
IEndpointSnitch snitch, Map<String,String> stratOpt)
    //Get the datacenters using the actual strategy
    Set<String> getDatacenters()
    //Query the number of data replicas defined for a datacenter
    int getReplicationFactor(String... dc)  //get DC replication factor
    //Check if two replication strategies have the same properties
    //For example, we could compare the class used to instantiate strategies
    //and their properties
    boolean hasSameSettings(IAbstractReplicationStrategy other)
    //Calculate the endpoints responsible for a token given the token to
    //endpoints map
    List<InetAddress> calcNaturalEps(Token tk, TokenMetadata meta)
}
```

3.2.1.6 Consistency

Depending on application requirements, infrastructure changes or even to provide better user experience, it is useful to provide a way to change the consistency guarantees related to what users can observe. Weak consistency levels often resort to one or a small subset of nodes to acknowledge operations, providing low latency but allowing the possibility of stale data reads, while on the other hand algorithms with strong semantics tend to use the full set of nodes responsible for the data, trading latency for a consistent view of the datastore. Depending on the application requirements or even for scaling reasons, it could be necessary to change the employed protocol to match the desired properties. Regardless of being single or multiple layer protocols, the consistency module provides a minimalist interface as follow:

```
public interface IConsistency{
    //Perform a read operation. readObj could be a token representing the row
    //key, meta can be additional information needed by the implementation,
    //such as nodes to write, consistency level to achieve
    void read(T readObj, Object... meta)
    //Write the writeObj, which can be a representation of the object to
    //write in multiple nodes that can be passed in meta argument
    void write(T writeObj, Object... meta)
    //This operation can be used either to implement ACID transactions or
    //simple batch operations under weak consistency levels, the word commit
    //is just to provide a meaning for multiple operations
    void commit(T commitObj, Object... meta)
}
```

With all the considered modules described and its interfaces specified, our modular architecture is exemplified in the Figure 3.1, where we have all the pluggable modules attached to the key-value storage system through a module manager built in its core.

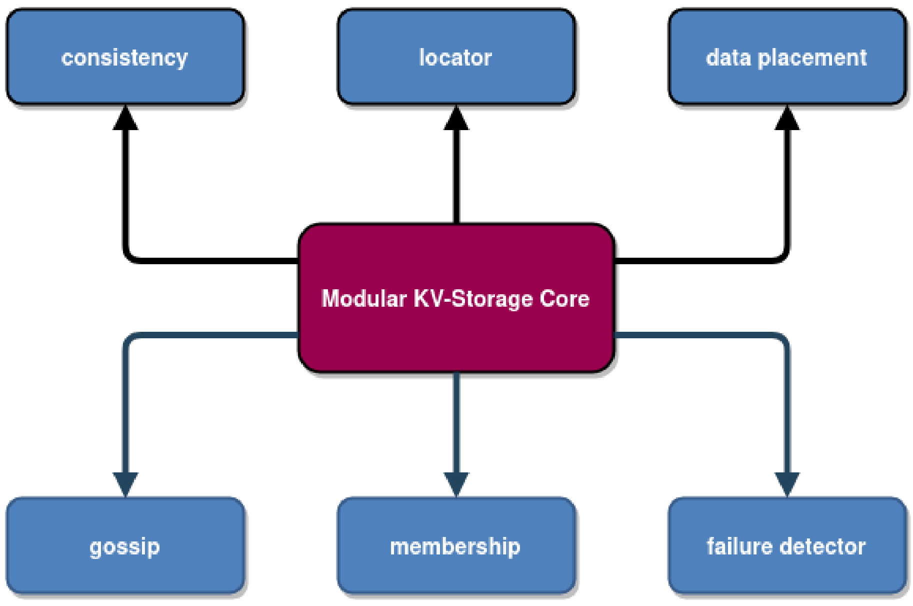


Figure 3.1: Modular Key-Value Storage System with pluggable modules

3.3 Validation

Module implementations and its validation is an orthogonal exercise to the main contributions of this work, however to install a module either at runtime or before starting the system, we must check if it is compatible with our modular solution. For this task we propose the use of a module manager responsible for maintaining and checking module compatibility. Supported by a configuration loader which load the path of each module container, the manager is in charge of checking if the initial class of each module is assignable to the proper interface to determine if it can be successfully installed in the system. This can be done during the boot procedure, or in runtime using local or remote proceduring calls (RPCs).

Evaluation of this solution will be performed, in a first step, comparing our modular solution against the baseline implementation using datastore benchmark tools like Yahoo Cloud Service Benchmark (YCSB)[10], which provides metrics such as latency and throughput. It is expected that the modular version offers lower performance comparing to the baseline since loosely coupled approaches often require more data exchanged between components.

In a second phase, corresponding to the implementation of the replication protocols for the consistency modules, since these provide different consistency guarantees they could not be compared with the baseline and modular approaches running eventual consistency, so we intend to validate that the performance of the datastore decrease as stronger consistency semantics are applied.

Implementation effort will also be measured using time and number of code lines as main metrics to demonstrate how much difficult is to build a module to our modular datastore.

Chapter 4

A Modular Solution based on Apache Cassandra Architecture

Apache Cassandra uses a combination of techniques to provide high availability and fault tolerance properties. Apart from its effort to provide a customizable configuration of different attributes, using either a local configuration file or runtime method invocations to a Java Management Extension (JMX) agent, its implementation is restricted to the default mechanisms provided and discussed in Section 2.6. As previously mentioned in Section 1.2, these limitations are a result of monolithic implementations which make impossible the evolution of individual components and exploring different implementation alternatives, which are the primary challenges to tackle in this work.
We start this chapter describing the way Cassandra baseline is organized and how the modular approach was derived considering its architecture. Next, we explain how the modules are managed and interact. Moreover, we present the communication pipeline and how to introduce new operations. To conclude, we discuss the implementation details of two different datastore solutions offering distinct consistency properties, and its modification in compliance with our design.

4.1 Cassandra Organization and Modular Transformation

Cassandra architecture[14] is structured by packages containing related functionalities, which can be arranged in three big groups:

- The primary stack, which is our main target since its where database engine and core functionality is implemented;

- Support packages, containing core helpers essential to database operation, optional packages for non-essential features and clientside packages integrated for convenience;

- Subordinates which are helpers used by top-level packages.

The primary stack can be perceived as being organized in layers constituting a high-level division of concerns. Top layers expose interfaces to external clients, middle layers form distribution and storage engines, and lower layers are closer to the OS and hardware abstractions.

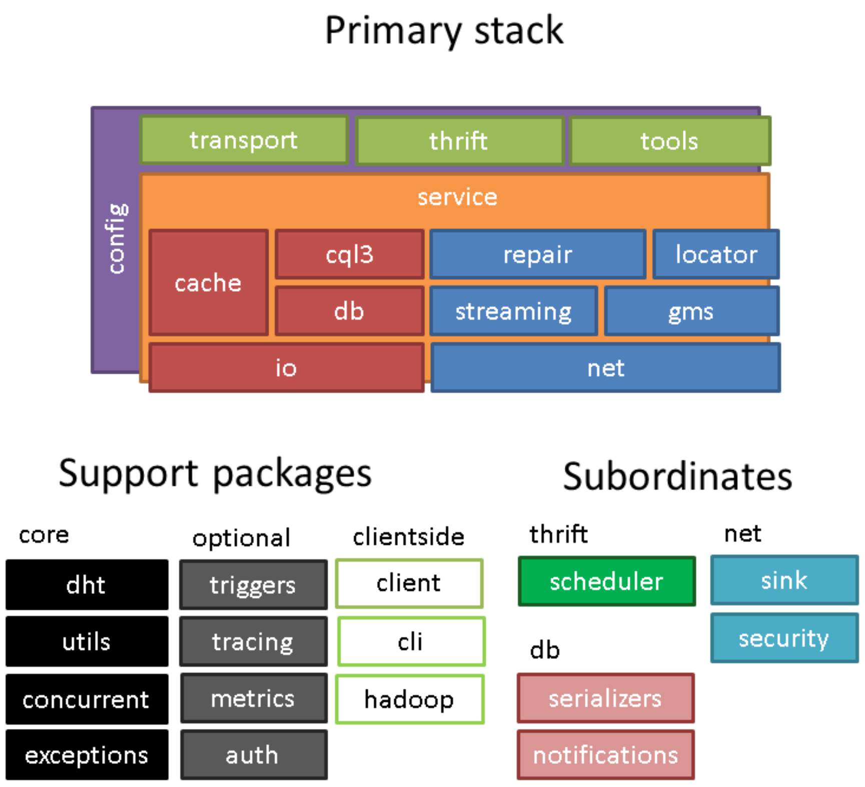


Figure 4.1: Package Diagram of Cassandra Architecture

The different colors in the primary stack on Figure 4.1 represent distinct subjects:

- Red packages establish functions to handle data stored in nodes, from on-disk SSTable format, cache service, the specific Cassandra Query Language (CQL), data division levels (table, row, column) and other storage engine tasks;

- Packages in blue make direct use of the network to provide its functionalities, such as messaging service, gossip implementation, snitches and other procedures, distributing data and work among nodes;

- Green packages present interfaces to expose Cassandra to the external world;

- Orange and purple refer packages whose classes are omnipresent, so they don't fit in a single category.

The choice of the Modules for our framework was made taking into consideration functionalities with a direct impact on requirements such as latency, throughput, and consistency but also the code length effort to build a solution in feasible time. On the next subsections, we describe some of the Cassandra functionalities considered to this work and a description that helps reasoning about why its separation of concerns is not enough to produce new implementations with different properties. Along with this rational, we present our modular transformation which separate functionalities with different purposes.

4.1.1 GMS Package

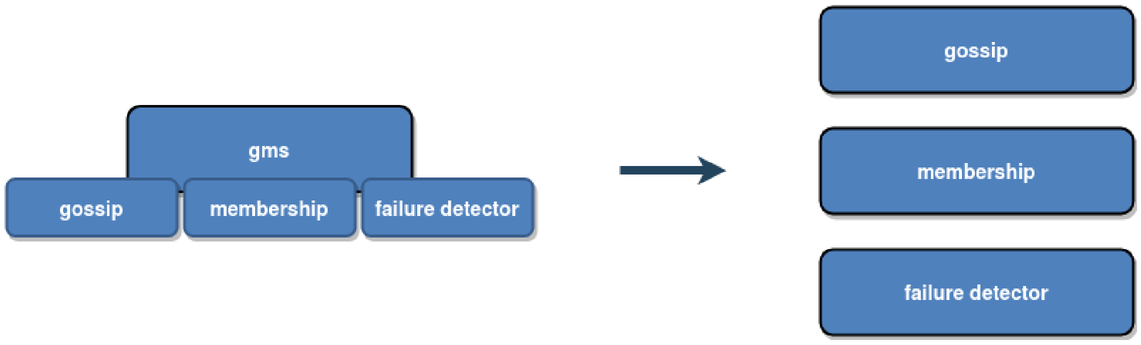


Figure 4.2: GMS package separated in three individual modules

Gossip and Membership package contains classes where the gossip algorithm implemented is entwined to the membership information, restricting the use of the epidemic broadcast as a service to disseminate other information. Clear evidence of this resides in the fact that gossiper class updates directly the heartbeat of the local node whether it should be just an interface to send and receive information, performing that action as a consequence. Furthermore, the implemented failure detector is also present in this package because its algorithm is updated by the gossip protocol employed. Therefore, we split these three functionalities building the respective interfaces and ship as individual modules as shown in Figure 4.2.

4.1.2 Locator Package

Locator package contains an abstract class as a base for all replication strategies provided and an interface to be implemented by all snitches available. Even though these functionalities are not tightly linked, its dissociation is required to implement and export each

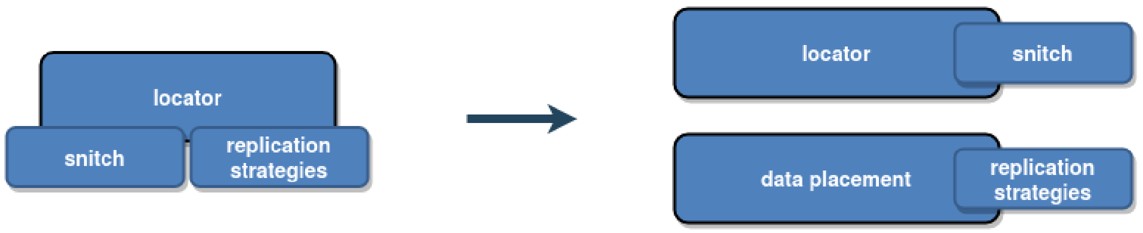


Figure 4.3: Locator package and its modular separation

one individually. We supply an interface to the further replication strategies and build two singular modules as shown in figure 4.3.

4.1.3 DB Package

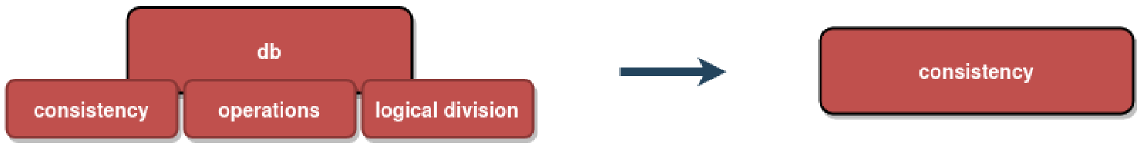


Figure 4.4: DB package and its break-up, generating the consistency module

DB package is the most challenging part analyzed since it contains the storage engine implementation, which is composed by almost a quarter of the entire Cassandra codebase. For that reason and due to its extension, we cannot consider it as a module for our work. Analyzing its structure we opted to consider another functionality which is its consistency feature since its one of the most important aspects of DBMS, having a direct impact on user-perceived latency. Cassandra provides eventual consistency where its implementation just allows us to configure the number of nodes that should confirm operation before the operation is considered terminated and a reply is sent to the client. This consistency level does not provide any kind of guarantees since replica convergence only occurs if the system reaches a quiescent state. For that reason, we detached the consistency feature from this package building a new module, as present in Figure 4.4.

4.2 Module Management and Interaction

Components are maintained using a module manager, which provides realtime load and unload features and a communication pipeline between modules. In the datastore boot procedure, the manager resorts to a configuration loader that reads a file containing the path for modules and its container filename, as described in Listing 3.1. With this information, each module file is loaded and verified as compatible with our system.

```
    //Used to declare everything that should be loaded on the module boot
    //For instance, we can instantiate endpoints for external communication
    void initializer();
    //Specification of stop routines before module shutdown
    void shutdown();
    //Obtain a map of operations and respective handlers to be recognized by
    //the messaging service
    Map<Verb,IVerbHandler> getVerbHandler();
    //Obtain a mapping between operations of the modules and respective stages
    //Stages are used to define in which queues the operations will be placed
    //Different queues are subject to distinct concurrency control algorithms
    Map<Verb, Stage> getVerbStage();
    //Collect a map with operations and respective serialization classes
    Map<Verb,IVersionedSerializer> getVerbSerializer();
    //Gather a map with operations and respective deserialization classes
    Map<Verb,IVersionedSerializer> getCallbackDeserializer();
    //Obtain a map with the operations that could be discarded if messages are
    //lost
    EnumSet<Verb> getDroppableVerbs();
    //Obtain class instances through reflection
    Object getInstance(String name, Object... args) throws ConfigException;
```

Listing 4.1: Initial Interface common to all Modules

Using Java Reflection, which can examine the intern properties of the container, we look for the initial class to check if our interface, shown in Listing 4.1, is assignable from it. In case of success, the class is instantiated and stored in a map for constant and direct access. Subsequently, messaging service handlers and serializers for communication are registered and an initializer method gets invoked. This latter procedure loads anything to be initialized before the module is ready for use, such as an interface to external communication service, for example.
The implementation effort of this interface is minimal since the programmer just needs to configure the initialize and shutdown methods and define which operations should be supported by the messaging service and how they should be serialized and deserialized, apart from the module core functionality.

Module interaction is done by performing a call to the manager, as illustrated in Figure 4.5, which returns a module instance stored in its map. In the event of an exception being thrown on this call, the source module should handle it by itself since the manager could not be prepared to handle different kinds of exceptions due to the free implementation that is allowed to individual modules, which would require the manager to be prepared to handle all kinds of exceptions.

Runtime component load, unload, and reload[1] features are accessible through remote procedure calls to a JMX agent that delivers these operations to the manager. Keeping

[1]Reload performs a consecutive unload and load operation for the same kind of module

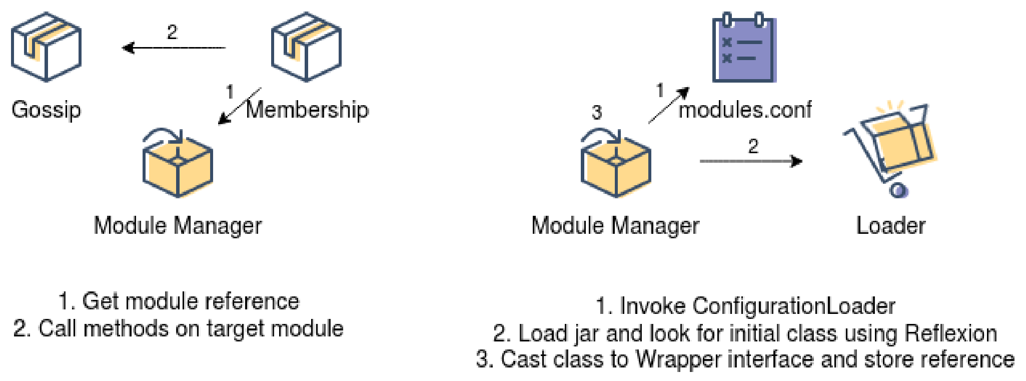

Gossip
Membership
Module Manager
2
1
1. Get module reference
2. Call methods on target module
modules.conf
Module Manager
Loader
1
2
3
1. Invoke ConfigurationLoader
2. Load jar and look for initial class using Reflexion
3. Cast class to Wrapper interface and store reference

Figure 4.5: Module interaction and loading procedures

the datastore consistent while changing its parts is not a trivial challenge, so we use a procedure similar to a two-phase locking as follows:

1. In the first phase the coordinator receives the operation and issue a message to all the nodes in the cluster to lock;
2. The second phase confirms or cancels the operation, respectively, if all the nodes respond positively or a timeout is triggered. Locks are released afterward.

The success of this operation only guarantees that nodes agreed on its execution, but don't provide any guarantees regarding pending and running operations. For that reason, module implementations should block incoming operations and wait for the event pools to be empty before removing the component, which can be implemented, for example, using Netty client-server framework.

4.3 Communication Pipeline and Operations

Cassandra uses the Netty framework to provide non-blocking I/O operations, which follow distinct paths considering the message source:

- Internode communication messages are directly delivered to the messaging service since it has a map with all the message verbs available and knows how to serialize and deserialize them. Our modules, when interested to use this service, should previously register verbs on the map to allow messages to be recognized;
- Client messages are queries syntactically validated and parsed to statements. The parser is built using a grammar, which compiles regular expressions to valid statements, compiled with the core implementation.

The latter path is incompatible with our modular approach since we cannot introduce specific module operations in runtime. This will require to stop the datastore and compile the new operations in the grammar to be recognized as valid statements. Two solutions are possible to solve this constraint:

- Remove the grammar and change the entire CQL to allow runtime insertion of expressions;
- Resort to an additional messaging service for modules that require new client issued operations.

We opted for the last using a Netty server to handle and receive new operations and resort to an individual client for each module.

4.4 Implementation of Existing Solutions in our Framework

Datastores should deal with the increasing amount of data generated by users while being able to adjust to application requirements of any kind. In this section, we start describing the implementation of two algorithms that provide distinct consistency guarantees. Using the modular design proposed by our architecture, both algorithms were implemented in separated consistency modules, following the respective interfaces defined in the previous chapter.

4.4.1 C3

The C3 algorithm provides causal consistency guarantees with low metadata overhead, decoupling a causality layer to manage dependencies between local and remote operations from the operation layer where operations are effectively executed. Its causality layer is implemented separately as a service and accessible through network requests, so we don't need to include it inside our modular design. In respect to supported operations, they're the same as the default, however, its execution follows different procedures, so we must support them as new operations.

This way, to generate a modular implementation we separate the algorithm in two parts:

- Network package, which sends and receives messages to/from causality layer and clients;
- Operational package, with the core handling functions.

Considering the global interface defined for the modules to implement the wrapper, presented in Listing 4.1, we opt to use the initialize and shutdown functions to, respectively, start and close network sockets and their connections. Furthermore, the same functions are used to change the status of native CQL transport service, stopping concurrency in operations from both sources, which can break datastore consistency. Additionally, core functionalities such as the hinted handoff feature, that turns possible that a late write could be delivered to inaccessible nodes, were also disabled since only supports the execution of default operations and it will require changes to the causality layer.

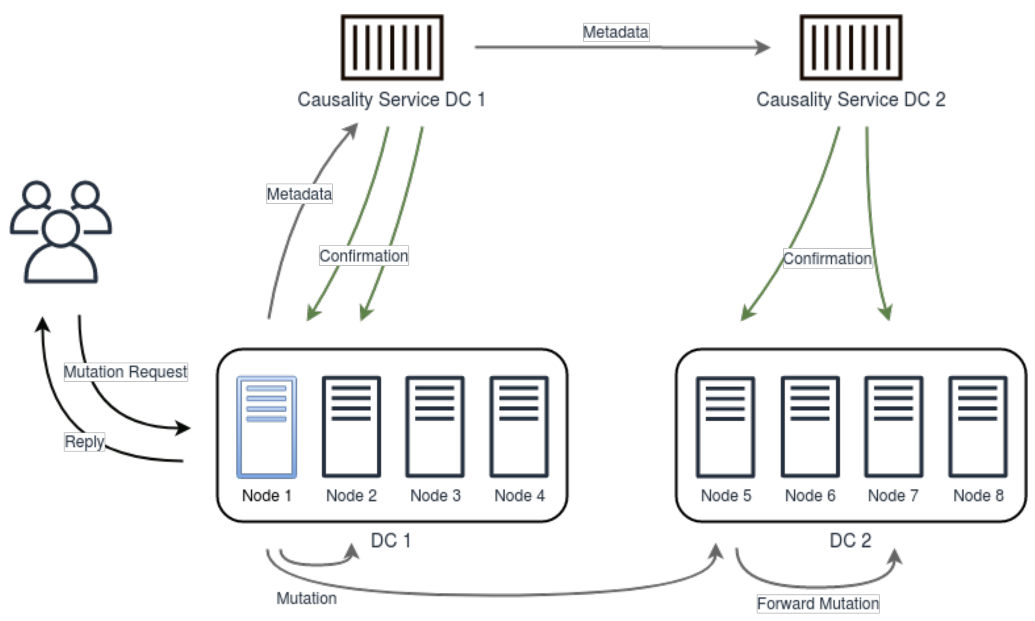


Figure 4.6: Write operation flow, missing acknowledgements

Operation pipeline change accordingly to the requested operation:

- Read operations don't need to go through the causality layer, so they are handled like regular Cassandra reads, implemented gathering a local datacenter quorum of nodes before returning the answer. We follow the same approach as the original C3 implementation;

- Write operations have metadata sent to the causality later, which confirms the operation on the responsible nodes when dependencies are met. The previous implementation intercepts the Cassandra write pipeline sending metadata to the causality layer and uses a modified mutation message handler to confirm the operation. Our solution abandons this approach resorting to new message types and dedicated handlers.

4.4.2 Blotter

Blotter produces low-latency transactions relaxing some of the snapshot isolation properties to provide forward freshness and a tailored configuration of Paxos to perform state machine replication in multiple DC setups. Its implementation resorts to a concurrency control module to provide transaction atomicity and Non-Monotonic Snapshot Isolation properties in a single DC, which consists of three parts:

- Client - Provides the Blotter interface, relaying in the transaction manager to read and commit operations; Writes are cached and only sent to TM in commit time, together with gathered metadata from reads;
- Transaction Manager (TM) - Handle read and commit operations; Reads are simply routed to data manager; Commits are resolved using a two-phase commit protocol to preserve atomicity, where the TM acts as coordinator. A first phase checks for NMSI compliance on the DMs, and a second phase confirms or aborts the operation;
- Data Manager (DM) - Is the core of the concurrency control, checking for conflicts, handling snapshot metadata information and performing read, write and abort operations.

Paxos state-machine replication replicates the commit operation in a multi-DC setup. Its configuration assumes one fixed DC as leader deciding the order of commit operations without compromise liveness.

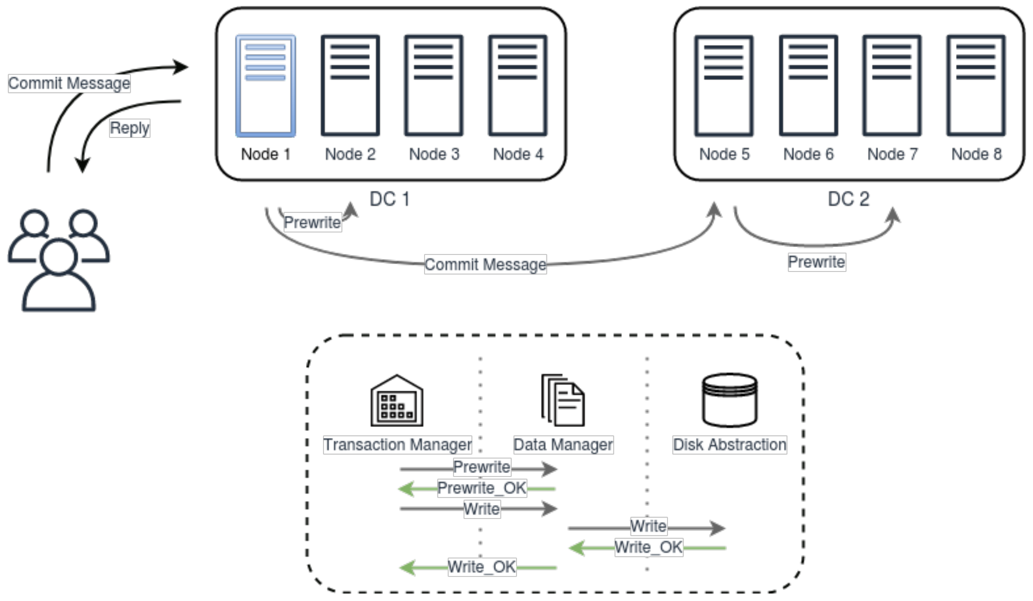


Figure 4.7: Commit operation flow

Figure 4.7 presents a brief overview of modular design considering a commit operation flow. Node 1 TM replicates the commit operation on Node 5, which is a randomly selected coordinator, using Paxos. Node 1 and Node 5 TMs are the coordinators and should execute the intra-DC protocol, respectively, contacting nodes 2 and 7. The dashed rectangle corresponds to the two-phase commit protocol where Node 1 TM is issuing a prewrite to Node 2 DM, which, after validating NMSI requirements, responds and receives the write operation to be written to stable storage. Analogously the same procedure is executed between Nodes 5 and 7.

Our modular implementation, as in the C3 use case, starts breaking the existent approach in two parts: The Network package, to establish connections between clients and the module, and the Operational package with TM and DM core functionalities. Initialize and shutdown procedures trigger the status of network connections, TM and DM data structures, hinted handoff, and native CQL transport.

Core functionalities were the big challenge in this module design for multiple reasons:

- Intra-DC concurrency control logic uses mutation messages to piggyback prewrite, write and abort operations, increasing overhead in regular mutations since they've to perform an additional comparison to match the operations;
- Snapshot versions of objects are stored on-disk, increasing I/O usage comparing to a volatile approach;
- Many Cassandra core functions used by Blotter were implemented in older versions, not compatible with the version used as baseline. This turns in as an additional effort since we need to perform a deep study to adapt the algorithm to the current version.

In this sense, implementation refactoring was not possible, which demands additional work to produce a new solution, preserving the protocol logic.

Our approach tackles these challenges through a new consistency module, which uses a dedicated DataManager messages and handlers compatible with the current version and a volatile data structure local to each node to manage existent snapshot versions for transactions. Furthermore, we have created dedicated clients to issue Blotter operations, which we explain in detail in the following chapter.

CHAPTER 5

Experimental Work

In this chapter, we start describing the hardware and network structures behind the adopted testbed used to experimentally validate this work. Then we present the developed tools and experimental parameters applied to conduct our tests. Furthermore, we discuss the performance outcomes of the validation modules and the development effort inherent to our modular solution.

5.1 Methodology

5.1.1 Setup

To run our experiments, we used the Grid5000 platform, which is a flexible, distributed, and large-scale computational platform for research purposes. It is composed of 8 sites with 31 clusters connected by a 10 Gb dedicated network and a high level of heterogeneity in terms of available hardware.

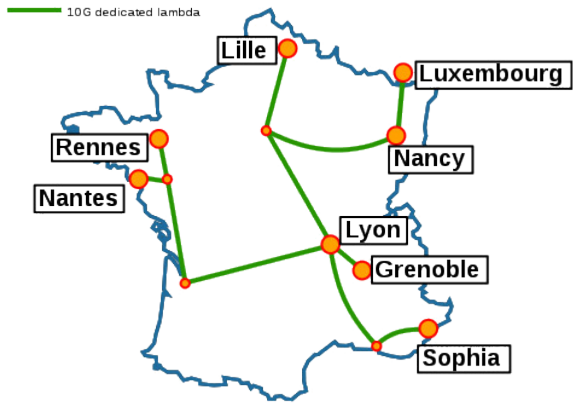


Figure 5.1: Grid5000 Renater-5 10Gb Lambda infrastructure

For our experiments, we choose two clusters from two different sites to simulate a geo-replicated scenario, and, since we're operating datastores, opted for nodes considering high but similar computational resources and enhanced storage devices. About the latter, datastores should have the highest throughput and lowest latency possible to provide a good contribute for the user experience, so we opt for machines with solid-state drives instead optical disk drives after found the difference between them have an impact of 3 orders of magnitude on the throughput.

Figure 5.1 shows a diagram of the Grid5000 infrastructure from where we select the Lille and Nancy sites and, respectively, their Chiclet and Grimoire clusters. Measured latency between both is 10 ms, and all the nodes in each cluster have the same characteristics:

	Lille (Chiclet)	Nancy (Grimoire)
CPU	2x AMD EPYC 7301 (16 cores/CPU)	2x Intel Xeon E5-2630 V3 (8 cores/CPU)
Memory	128 GB RAM	128 GB RAM
Storage	480 GB SSD	200 GB SSD
Network	2x 25 Gbps	4x 10 Gbps

Table 5.1: Cluster nodes specifications

Our datastore infrastructure is composed of 16 machines where each datacenter runs 4 clients and 4 datastore instances. Data partitioning is made through sharding using a replication factor of 3, which means each data replica distributed in 6 different nodes, with a quorum in each DC. Replica placement is performed considering rack awareness, using a Network Topology Strategy which distributes the same item in different network racks (2 in Chiclet and 4 on Grimoire), avoiding localized failures. Network configuration is defined in a Property File Snitch which nodes resort in the bootstrapping phase and, after that procedure, maintained using a gossip protocol to exchange membership information.

5.1.2 Software and Experimental Parameters

To issue operations to the datastore, we use the Yahoo Cloud Service Benchmark (YCSB) project to compare the performance of different systems using metrics such as throughput and latency. This tool is composed of an extensible workload generator (Client) and core workloads, which are a set of configurable scenarios to be executed by the generator.

The default YCSB client implementation uses the original Datastax driver to emit operations to the baseline version of Cassandra. To allow new operations for each Consistency module, we created two different clients, transforming the original YCSB client to use a dedicated channel to communicate with the modules entry-point.
C3 YCSB client uses the same workload generator as the default client since the supported operations in the datastore side (select, insert, update and delete) are the same. However,

Blotter's YCSB client, due to its transactional support, requires a new workload generator and support to the commit operation, along with client-side metadata structures.

To compare the baseline version with our modular solution, we use different configurations, trying to approximate the guarantees provided by both. Since Cassandra only provides eventual consistency, we adjust the number of nodes acknowledging operations to verify the cost introduced by algorithms with more robust semantics, like C3 and Blotter.

We now describe distinct consistency properties available in the default Cassandra setup used by our experiments:

- LOCAL_QUORUM - Response is sent to the client after a quorum of local nodes acknowledges the operation, which corresponds to 2 replica nodes in the same datacenter as the coordinator;
- EACH_QUORUM - Operations should be confirmed by a quorum of nodes in each datacenter, which is proportional to 2 replica nodes in every datacenter;
- ALL - All replica nodes should confirm the operation, equivalent to 6 nodes in our setup.

To perform a fair comparison between Cassandra and our modular datastore, we run the same configuration properties on internal components and an equal consistency definition on both, providing eventual consistency with a local quorum of nodes acknowledging operations. We also used the same client properties and pattern issuing operations to the closest datacenter to the client. With identical configuration, we intend to have an accurate measure of the overhead introduced by component separation on the modular setup.

Regarding the modular approach including two different consistency modules, C3 and Blotter, we want to demonstrate that the performance of the datastore is negatively affected when we increase the consistency guarantees, since we're adding more restrictions on what users can see about the data.

For the C3 module, which offers causal consistency guarantees, we expect to demonstrate a negative impact on user-perceived latency and throughput of the datastore comparing to the eventual semantics provided by default. Since the baseline version only offers eventual consistency and we truly cannot emulate those guarantees, we adjust its consistency algorithm parameters to collect a quorum of replica responses, respectively, in each datacenter on writes and in the local datacenter on reads, which is a similar configuration used by the module apart from the causality layer.

The blotter module provides strong consistency to the datastore and should further exacerbate the penalty performance observed by the usage of the C3 module. Its write operations flow independently in each datacenter after the commit operation gets replicated by Paxos, gathering a response from all replica nodes. Read operations aggregate responses from all the replicas in the local datacenter. In tandem with its strong semantics, we opt to use the ALL property to perform the comparison since it's the highest consistency property available for the baseline eventual consistency.

In respect to the generated workload, we attempt to simulate a real case scenario varying the number of clients and the kind of operations issued to the datastore. Our clients send 8.000.000 operations to a 100.000 item dataset divided by the number of client instances running. Each instance can be multithreaded, which means we can emulate more clients raising the number of threads running simultaneously, and thus increasing the datastore workload. We range that number from 4 to 128, emulating a setup with a maximum of 1024 clients at a time.

The correlation between the kind of operations was also subject to modifications, starting from a high read load proportion (95% reads versus 5% writes) and ending in a high write load setup (95% writes versus 5% reads). In the Blotter client, the write ratio gets replaced by commit operations, where we can still vary the number of writes included. For our tests, we opt to provide 3 writes inside each transaction.

Gathered results should allow measuring the latency and throughput performance penalty introduced by modularization, even if residual. Furthermore, we evaluate the consistency modules output as an example of how consistency guarantees directly impact the user-perceived latency, and the development effort resulting from this work.

5.2 Results and Analysis

In the previous section, we've introduced the setup and relevant configurations used in our experimental work. We now present and analyze the outcomes obtained from our experiments, noting that Cassandra's baseline is used as a reference to measure the cost introduced by the modularization process, being unreliable in a comparison against the modular version with consistency modules attached since different guarantees are provided. In that sense, the latter objective is to show that stronger consistency algorithms often introduce more overhead.

For the latency plots, we use the 99th percentile distribution to obtain accurate results without discard outliers[1].

5.2.1 Modular approach versus Cassandra Baseline

The first comparison represents the Throughput and Latency metrics for the modular approach in the context of this work versus the Cassandra Baseline, as shown in the two plots of Figures 5.2 and 5.3. Each data point represents different numbers of clients issuing operations simultaneously to the datastores, starting with 32 (4*8) and ending

[1]Outliers are data points that differ significantly from other observations, usually hidden by central tendencies like mean and median

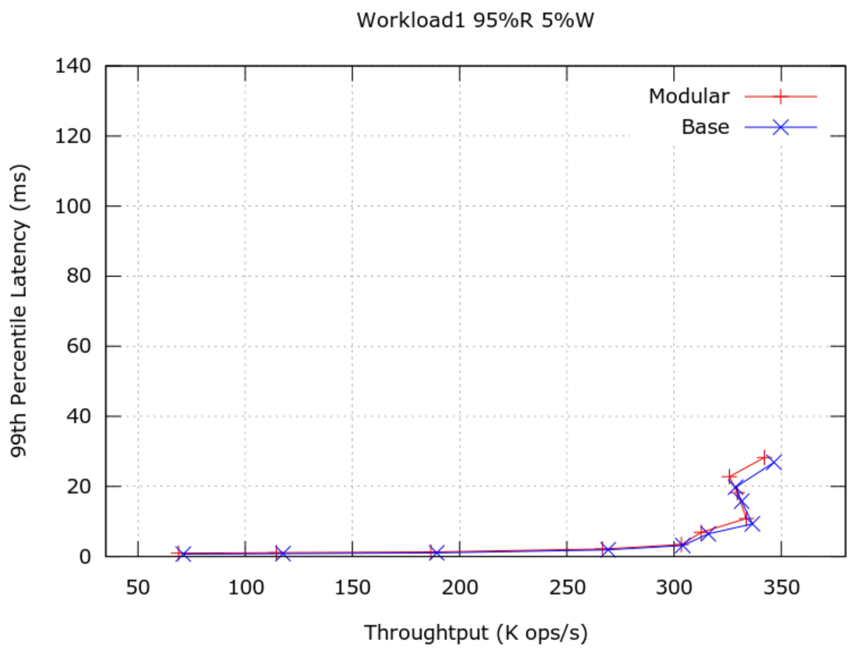


Figure 5.2: Performance of Modular approach versus Cassandra Baseline - 95% Reads

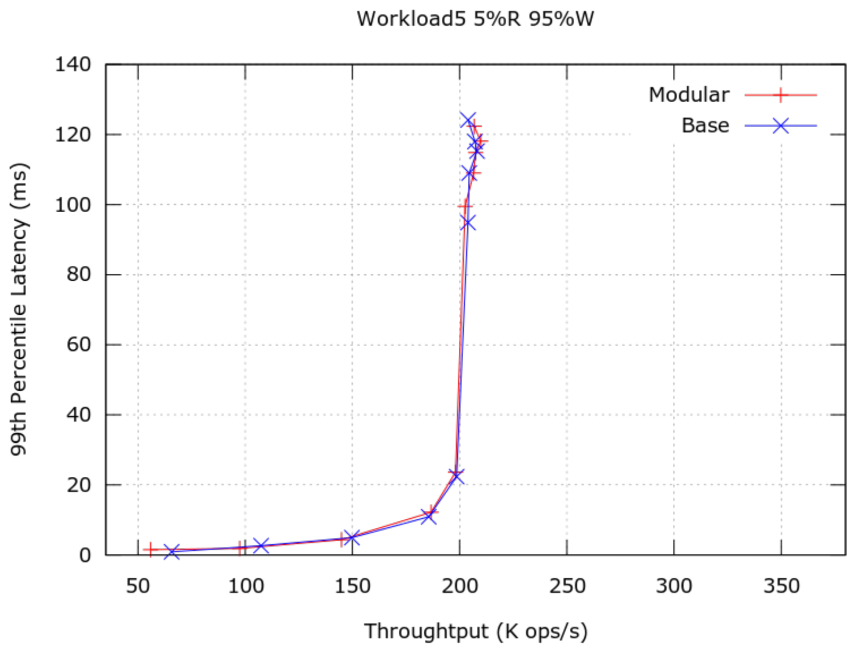


Figure 5.3: Performance of Modular approach versus Cassandra Baseline - 95% Writes

in 1024 (128*8). The better results are located in the bottom and rightmost part of the graphics since they represent the scenarios where best throughput and lowest latency are achieved. For both plots, we opt to show the two workloads with more activity respectively, in terms of reading and write operations. The other workloads can be found in Annex I.

In the first plot with 95% of read operations we reach the best throughput for all the cases, with almost 350000 operations per second for the last point and a low latency value of about 25 ms. Since this workload is not saturating the datastore, we present on the right the most effective plot in terms of load, which represents 95% of write operations requested. Looking at this plot we can see that after the fifth point, which corresponds to 336 clients, the client-perceived latency abruptly increases due to the high demand for write operations produced to the datastores, while throughput remains essentially unchanged.

These results are expected since disk write operations have lower performance compared to reads, which in turn increases latency. In terms of differences between both approaches, we can observe a slightly overhead observed in the Modular version which is translated in less 1,368% of throughput and more 10% of latency overall. Again these results are in line with our expectations since loosely coupling techniques often require indirections, which particularly in our work corresponds to the usage of the Module Manager. Therefore, we can conclude that the modular design proposed has little impact on the performance of the datastore.

5.2.2 Modular approach with C3

Figures 5.4 and 5.5 presents the same metrics measured in Figures 5.2 and 5.3, but in this case, we compare the Modular version using two different consistency properties. The first plot depicts the datastore running eventual consistency, where we adjust the number of nodes acknowledging operations attempting to match the same fraction of nodes used by the C3 consistency module, represented in the second plot. Note that due to the different dimensions of results, we opt to use separate graphics for each setup.

In the top plot, performance is superior since in this setup operations are directly executed and do not need additional coordination on dependencies. On the bottom plot causal consistency is enforced, with write operations forwarded to the causality layer where dependencies are matched, before its execution.

These outcomes are predictable considering causal consistency provides stronger guarantees and thus lower performance. However, the system designer should analyze when the causality violations are admissible before opt for an eventual setup.

5.2.3 Modular approach with Blotter

Similarly to the previous comparisons, we use the same metrics but this time to compare the Modular approach operating eventual consistency, depicted in the Figure 5.6, and strong consistency with transactions provided by the Blotter module, shown in the Figure 5.7.

In the first plot, we show the performance of eventual consistency with operations contacting all the nodes responsible for the respective keys, in a similar way to Blotter. However, we don't resort to any mechanism to provide ACID properties, so our datastore

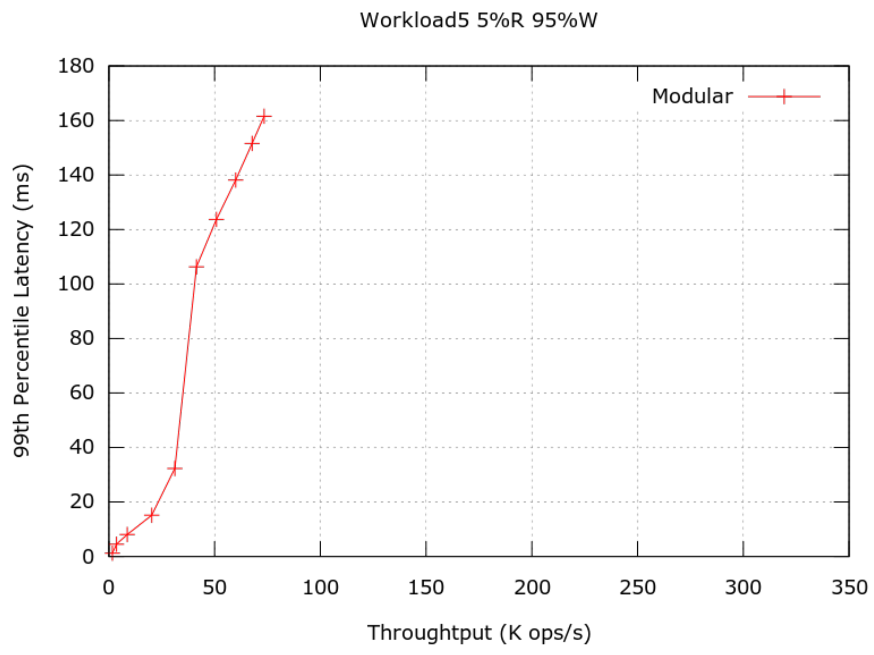


Figure 5.4: Performance of Modular approach running eventual consistency guarantees and contacting a quorum of responsible nodes in each DC - 95% Writes

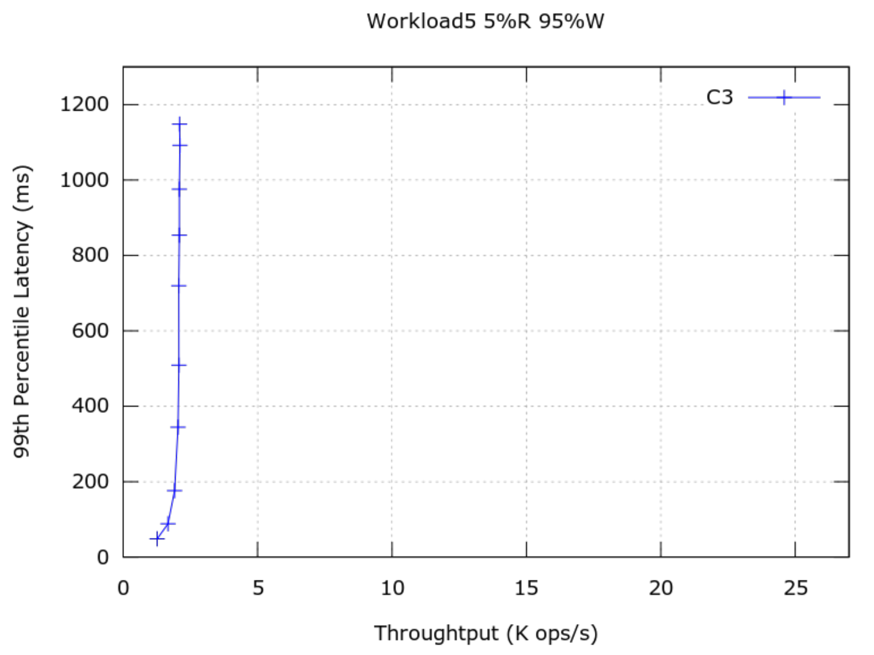


Figure 5.5: Performance of Modular approach running causal consistency guarantees - 95% Writes

still outputs a considerable throughput of about 60000 operations per second even in a high write workload. The second plot, which represents Blotter, has the lowest throughput of all consistency definitions used since it enforces strong consistency semantics. Due

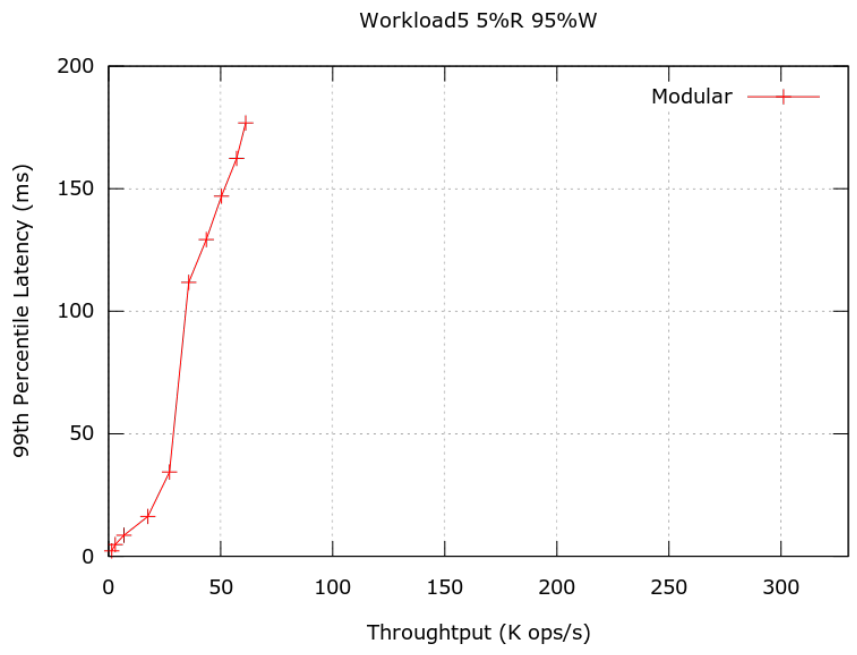


Figure 5.6: Performance of Modular approach running eventual consistency guarantees and contacting all responsible nodes in each DC - 95% Writes

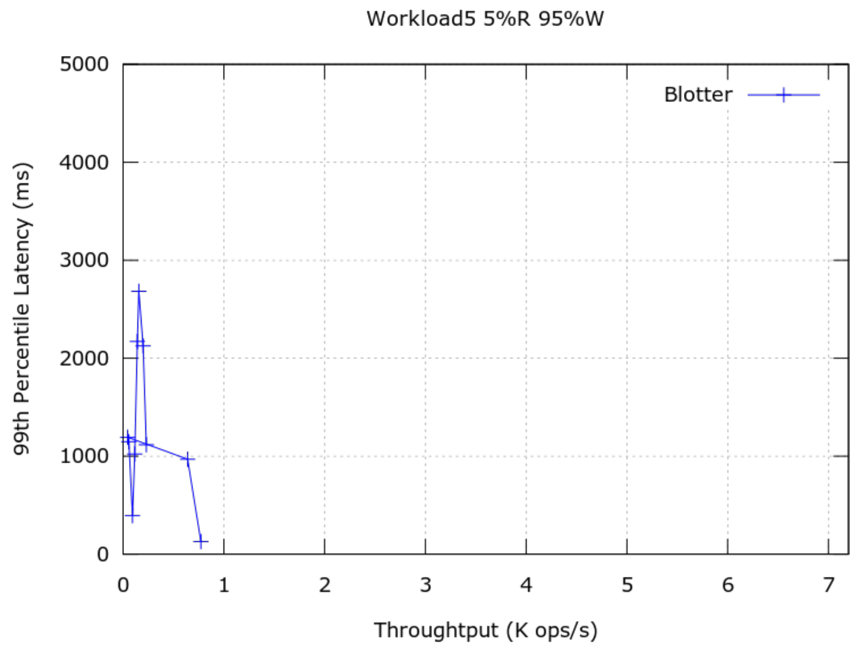


Figure 5.7: Performance of Modular approach running strong consistency guarantees - 95% Writes

to potential deadlocks and write-write conflicts detected during its test, we can see the points don't follow a continuous pattern like in the previous graphics.

The results are in line with our assumptions and prove that throughput and the client-perceived latency are negatively affected by the increase in consistency guarantees. Furthermore, we were able to represent these systems with different semantics as modules for our modular framework.

5.2.4 Development effort

This work comprehends three development phases which we can resume as follow:

- **Modularization core** - The code which allows to load configurations and manage modules, including load, reload and unload procedures, and their synchronization using the 2-phase locking mechanism;
- **Component adaptation to modular architecture** - Includes refactoring the existent classes to match the built interfaces for each module, initialize and shutdown tasks and the concrete functional implementation;
- **Client implementation** - Some modules like C3 and Blotter require new operations, issued with a dedicated client since the original Messaging Service could not handle them.

Component	Time (Months)	Code (Lines)
Manager	3	1307
Blotter	7	3607
C3		1232
Data Placement		808
Fault Detector		446
Gossip		1134
Locator		1895
Membership		2179
Blotter Client	2	821
Blotter YCSB Client		1223
C3 Client		472
C3 YCSB Client		967
		16091

Table 5.2: Development effort in terms of time and code lines

Table 5.2 represents the development effort in terms of time and code lines, whose colors separate the three distinct phases.

Regarding the first phase, the most difficult part was defining boundaries for each module in terms of interface specification, in a clear attempt to have a generic interfaces which suits different implementations.

The second phase requires a deep understanding of Cassandra's baseline architecture and the inherent component implementation, to adapt the existent implementation to a modular style and select which internal structures or features should be present in the initialization and shutdown procedures. Blotter took most of the time since its mechanisms to enforce strong consistency are difficult to interpret.

The last stage comprises four dedicated clients to issue operations to the datastore, which involves mastering the Netty client-server framework and, in the YCSB clients, understand its internal components.

Last but not least, as revealed in the previous table, the development effort to build a module for our framework is acceptable since it could be written in hundreds or a single digit thousand lines of code.

Chapter 6
Conclusion and Future Work

The modular architecture produced in this work, through the separation of concerns, allows increasing scalability not only in terms of development but also in the test and application of new approaches that could translate in superior performance. Furthermore, datastore flexibility confers adaptability characteristics to changes resulting from constant technological evolution, providing a way to meet the ever-increasing user demands.

Notwithstanding the above benefits, our approach also reflects a relevant drawback: Module interfaces must be generic enough to satisfy all, or at least, a large set of patterns. In that sense, its future modification could be necessary to remain in agreement with the latest standards.

As future work, a validation tool would be valuable to perform static or dynamic analysis of modules to anticipate problems in its effective execution, only allowing valid modules to be attached to the datastore.

Additionally, a query cache mechanism similar to the existent in the original Cassandra driver would be useful to avoid repeatedly parse of expressions. That will allow static queries, in terms of table and fields, to progress at a lower cost and reveal an increase in datastore throughput.

Bibliography

[1] D. J. Abadi. *Introduction to Consistency Levels*. 2019. URL: https://fauna.com/blog/demystifying-database-systems-introduction-to-consistency-levels.

[2] 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.

[3] D. An. *New industry benchmarks for mobile page speed*. 2018. URL: https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/.

[4] M. S. Ardekani, P. Sutra, and M. Shapiro. "Non-Monotonic Snapshot Isolation: Scalable and Strong Consistency for Geo-Replicated Transactional Systems." In: *Proceedings of the 2013 IEEE 32nd International Symposium on Reliable Distributed Systems*. SRDS '13. USA: IEEE Computer Society, 2013, 163–172. ISBN: 9780769551159. DOI: 10.1109/SRDS.2013.25. URL: https://doi.org/10.1109/SRDS.2013.25.

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

[6] M. Bravo, L. Rodrigues, and P. Roy. "Saturn: a Distributed Metadata Service for Causal Consistency." In: Apr. 2017, pp. 111–126. DOI: 10.1145/3064176.3064210.

[7] 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.

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

[9] S. Choudhury. *Apache Cassandra: The Truth Behind Tunable Consistency, Lightweight Transactions Secondary Indexes*. 2018. URL: https://blog.yugabyte.com/apache-cassandra-lightweight-transactions-secondary-indexes-tunable-consistency/.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. "Benchmarking Cloud Serving Systems with YCSB." In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, 143–154. ISBN: 9781450300360. DOI: 10.1145/1807128.1807152. URL: https://doi.org/10.1145/1807128.1807152.

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

[12] 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.

[13] D. Ford, F. Labelle, F. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. "Availability in Globally Distributed Storage Systems." In: *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*. 2010.

[14] A. S. Foundation. *TopLevelPackages - Cassandra2 - Apache Software Foundation*. 2019. URL: https://cwiki.apache.org/confluence/display/CASSANDRA2/TopLevelPackages.

[15] 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.

[16] M. Fowler. *Microservices*. 2016. URL: https://martinfowler.com/articles/microservices.html.

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

[18] P. Hooda. *Centralized, Decentralized and Distributed Systems*. 2019. URL: https://www.geeksforgeeks.org/comparison-centralized-decentralized-and-distributed-systems/.

[19] IBM. *ACID properties of transactions*. 2018. URL: https://www.ibm.com/support/knowledgecenter/en/SSGMCP_4.2.0/com.ibm.cics.ts.productoverview.doc/concepts/acid.html.

[20] Jaksa. *Active and Passive Replication in Distributed Systems*. 2009. URL: https://jaksa.wordpress.com/2009/05/01/active-and-passive-replication-in-distributed-systems/.

[21] 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.

[22] L. Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System.” In: *Commun. ACM* 21.7 (July 1978), 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563.

[23] 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.

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

[25] Microsoft. *Data partitioning guidance - Best practices for cloud applications*. 2018. URL: https://docs.microsoft.com/en-us/azure/architecture/best-practices/data-partitioning.

[26] 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.

[27] MySQL. *Overview of MySQL Storage Engine Architecture*. 2017. URL: https://dev.mysql.com/doc/refman/8.0/en/pluggable-storage-overview.html.

[28] Oracle. *Master Replication Concepts and Architecture*. 2012. URL: https://docs.oracle.com/cd/B10501_01/server.920/a96567/repmaster.htm.

[29] Oracle. *Overview of Replication*. 2015. URL: https://docs.oracle.com/database/121/REPLN/repoverview.htm#REPLN126.

[30] Oracle. *Partitioning for Manageability*. 2015. URL: https://docs.oracle.com/database/121/VLDBG/GUID-8A2DF8C1-8065-4C3D-8A23-B3E7A4D7D1C6.htm#VLDBG1057.

[31] Oracle. *Recommendations for Choosing a Partitioning Strategy*. 2015. URL: https://docs.oracle.com/database/121/VLDBG/GUID-9F7809B6-E68A-4425-AC99-BE00C7805408.htm#VLDBG00406.

[32] H. Pandey. *Synchronous vs Asynchronous Replication*. 2016. URL: https://cloudbasic.net/white-papers/synchronous-vs-asynchronous-replication/.

[33] H. Pandey. *Data Replication in DBMS*. 2019. URL: https://www.geeksforgeeks.org/data-replication-in-dbms/.

[34] N. Preguiça, C. Baquero, P. Almeida, V. Fonte, and R. Gonçalves. “Dotted Version Vectors: Logical Clocks for Optimistic Replication.” In: (Nov. 2010).

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

[36] R. van Renesse, D. Dumitriu, V. Gough, and C. Thomas. “Efficient Reconciliation and Flow Control for Anti-Entropy Protocols.” In: *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*. LADIS ’08. Yorktown Heights, New York, USA: Association for Computing Machinery, 2008. ISBN: 9781605582962. DOI: 10.1145/1529974.1529983. URL: https://doi.org/10.1145/1529974.1529983.

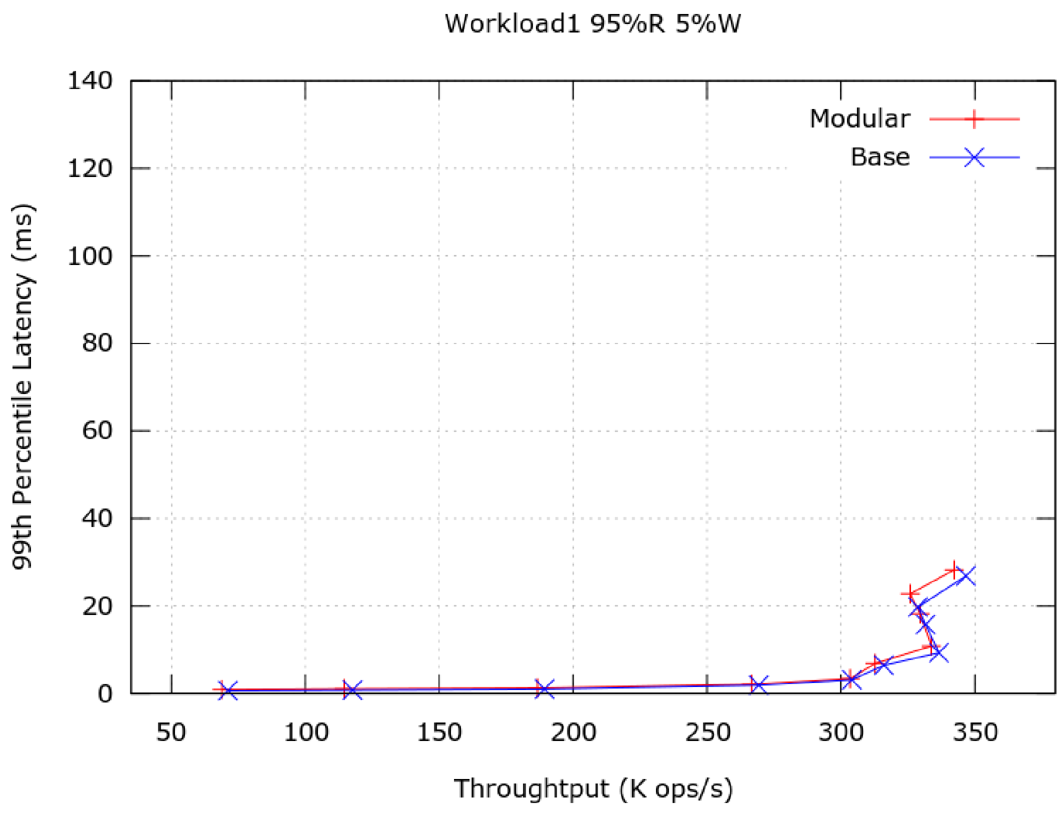
[37] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. “Conflict-Free Replicated Data Types.” In: vol. 6976. July 2011, pp. 386–400. DOI: 10.1007/978-3-642-24550-3_29.

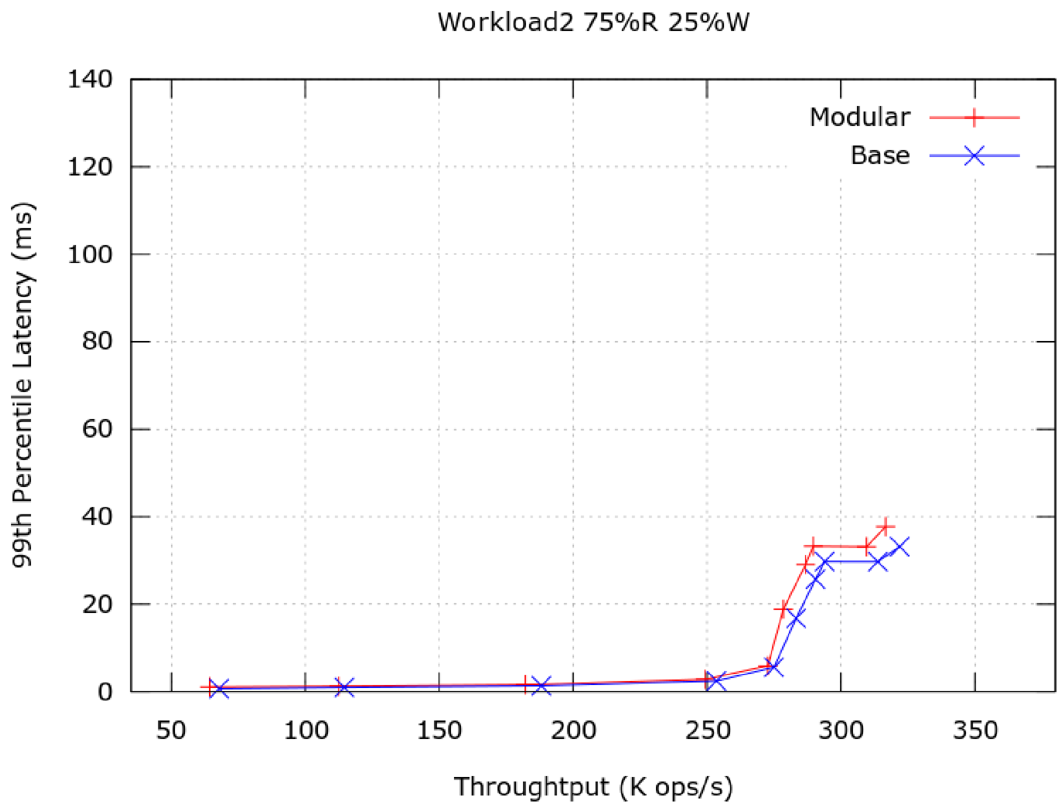
ANNEX I

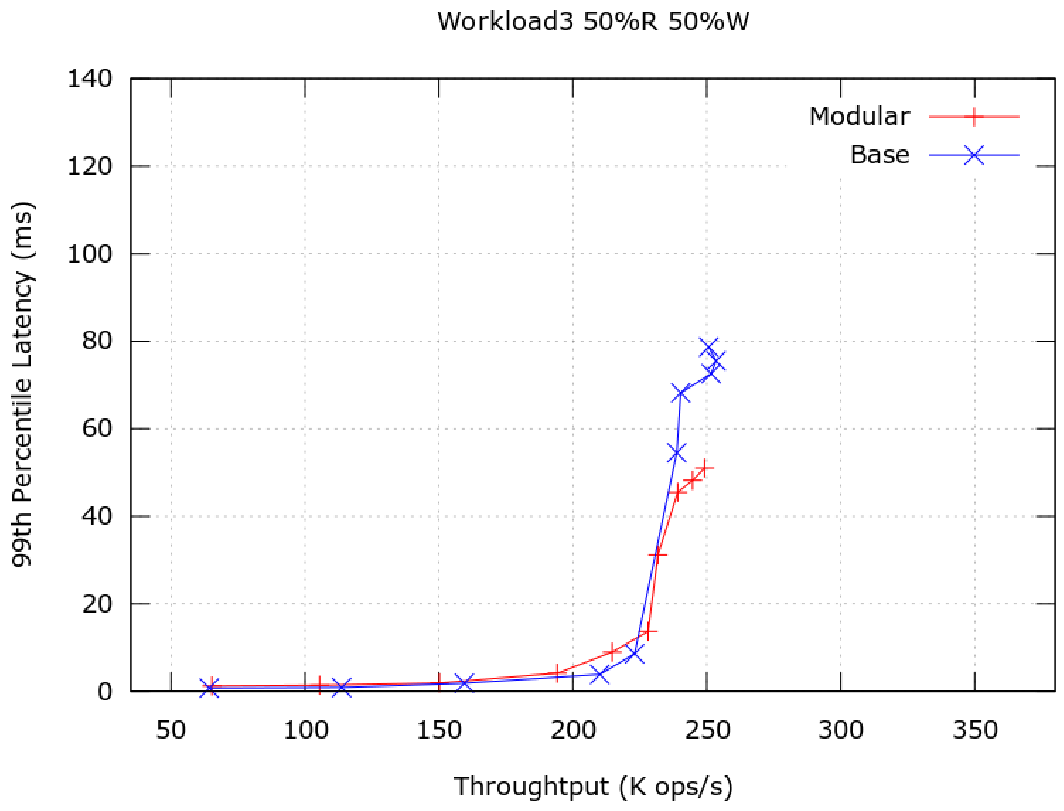
Annex 1 Workloads plot

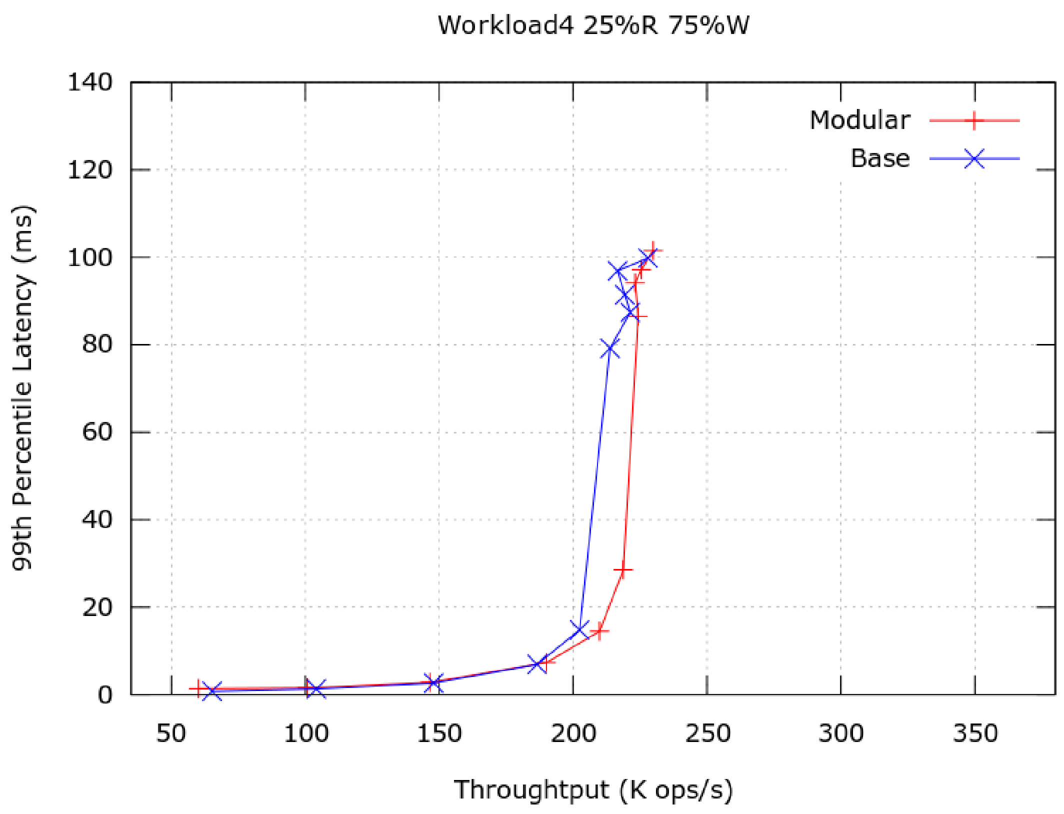
In the following sections we illustrate all the workloads used in our experiments where we vary the read and write operations ratio, along with different numbers of threads and consistency levels, as specified in the Section 5.1.2.

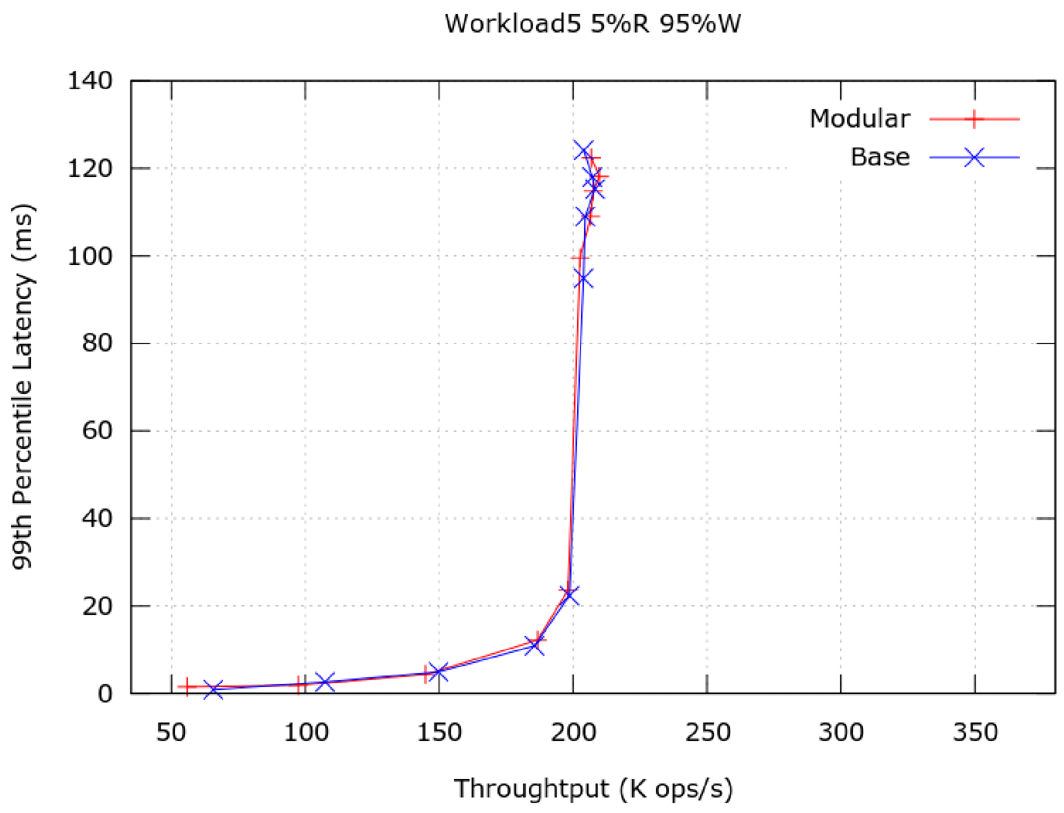
I.1 Modular approach versus Cassandra Baseline



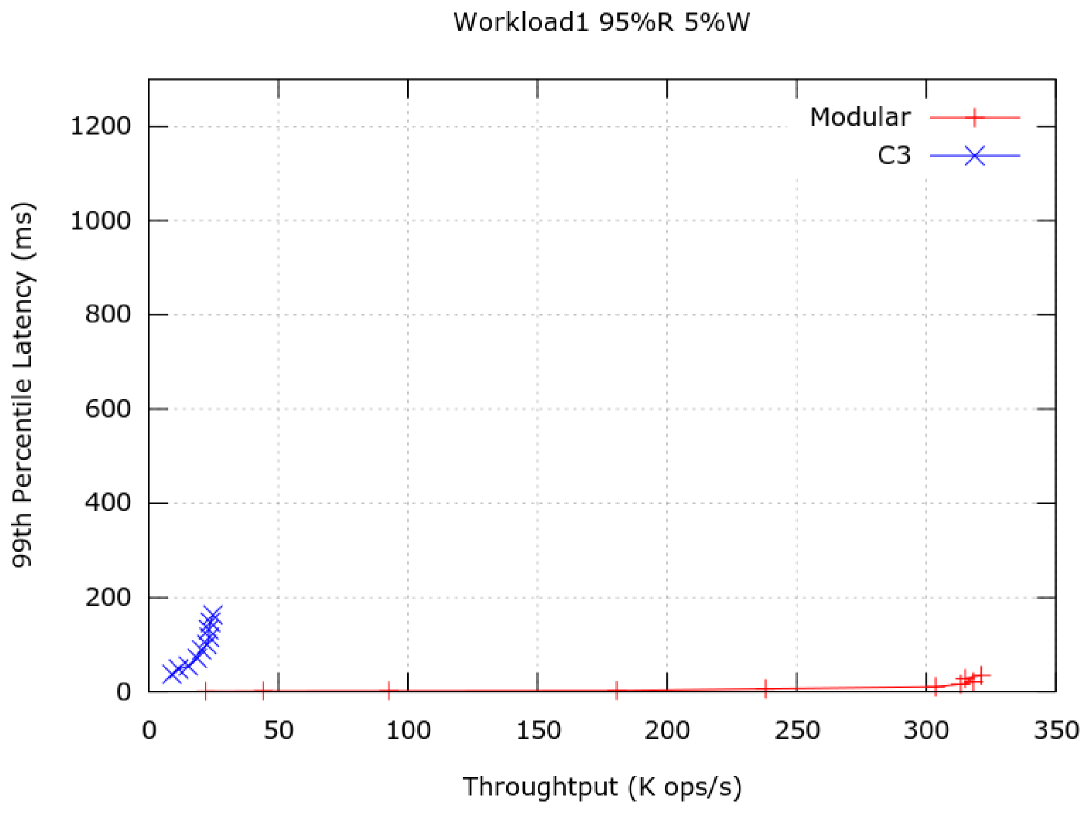

Workload2 75%R 25%W
Modular
Base
99th Percentile Latency (ms)
Throughtput (K ops/s)
0
20
40
60
80
100
120
140
50
100
150
200
250
300
350

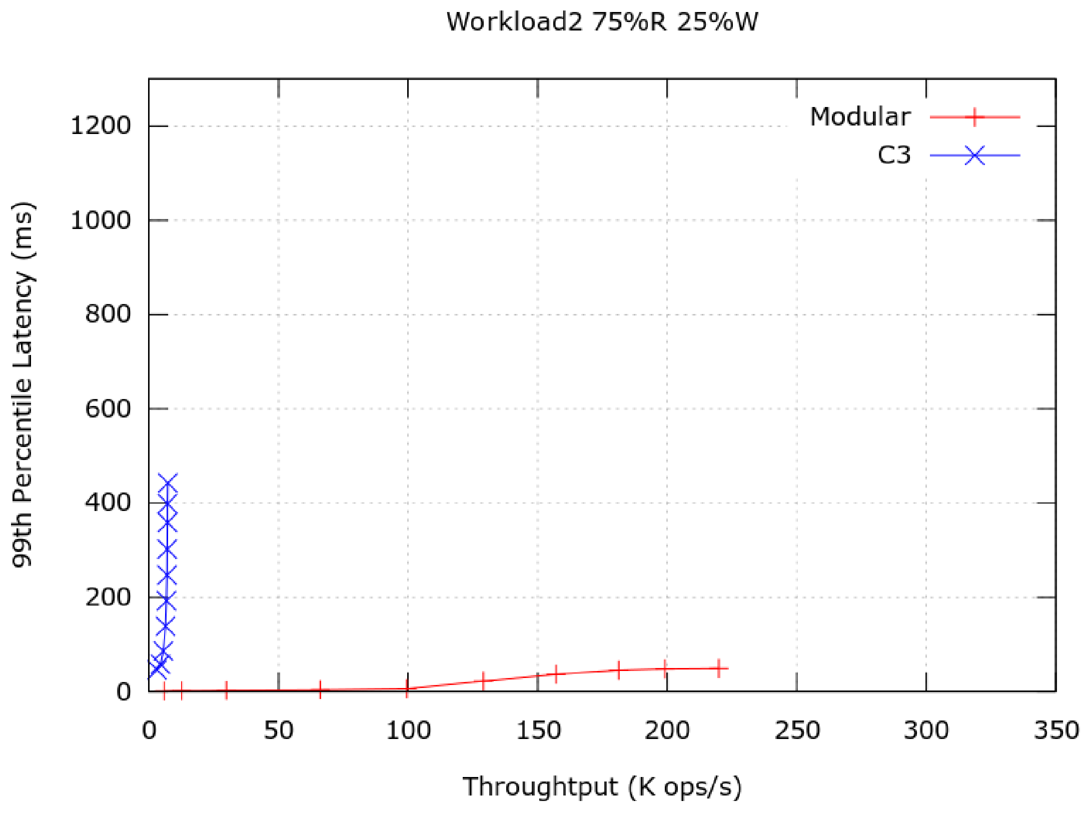

Workload3 50%R 50%W
Modular
Base
99th Percentile Latency (ms)
Throughtput (K ops/s)
0
20
40
60
80
100
120
140
50
100
150
200
250
300
350

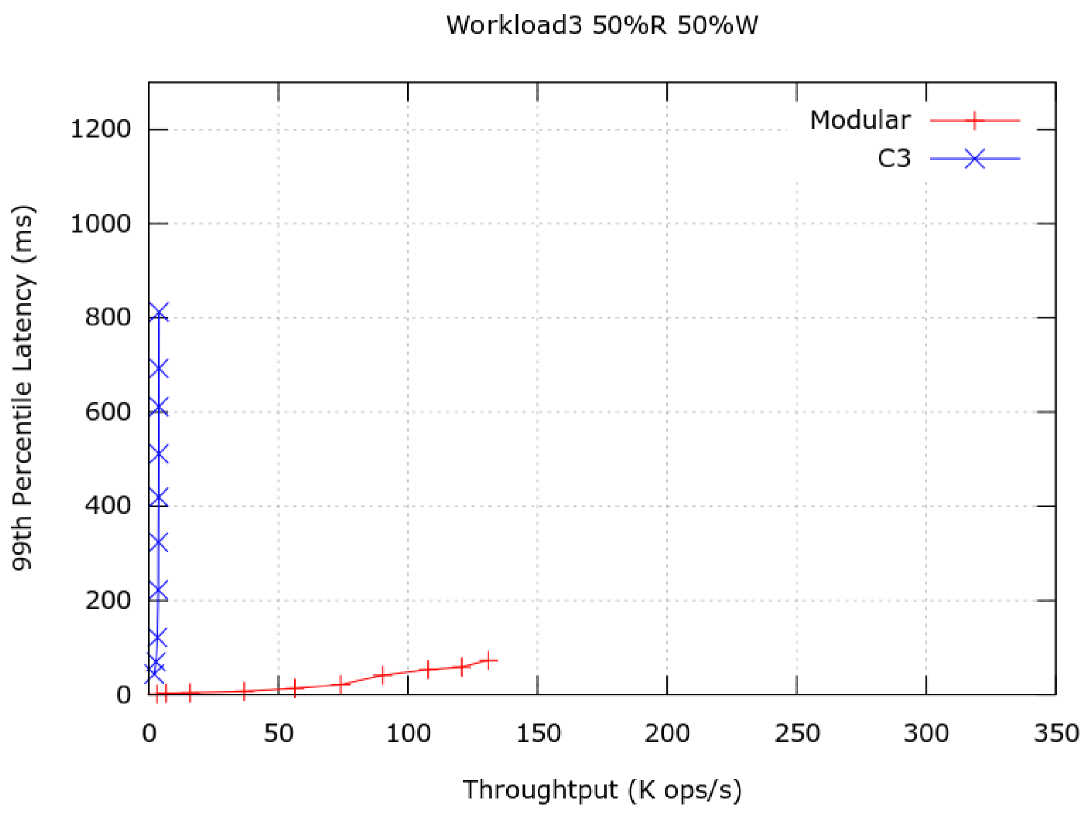

Workload4 25%R 75%W
Modular
Base
99th Percentile Latency (ms)
Throughtput (K ops/s)
0
20
40
60
80
100
120
140
50
100
150
200
250
300
350

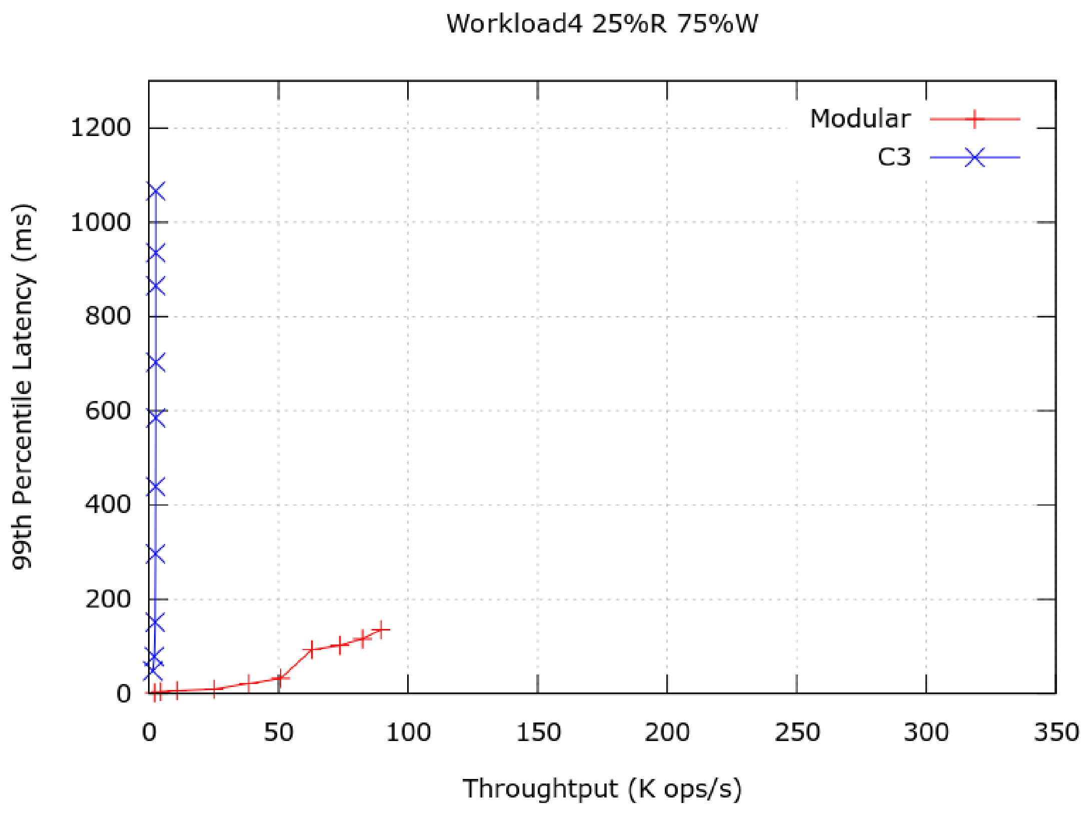

Workload5 5%R 95%W
Modular
Base
99th Percentile Latency (ms)
Throughtput (K ops/s)
0
20
40
60
80
100
120
140
50
100
150
200
250
300
350

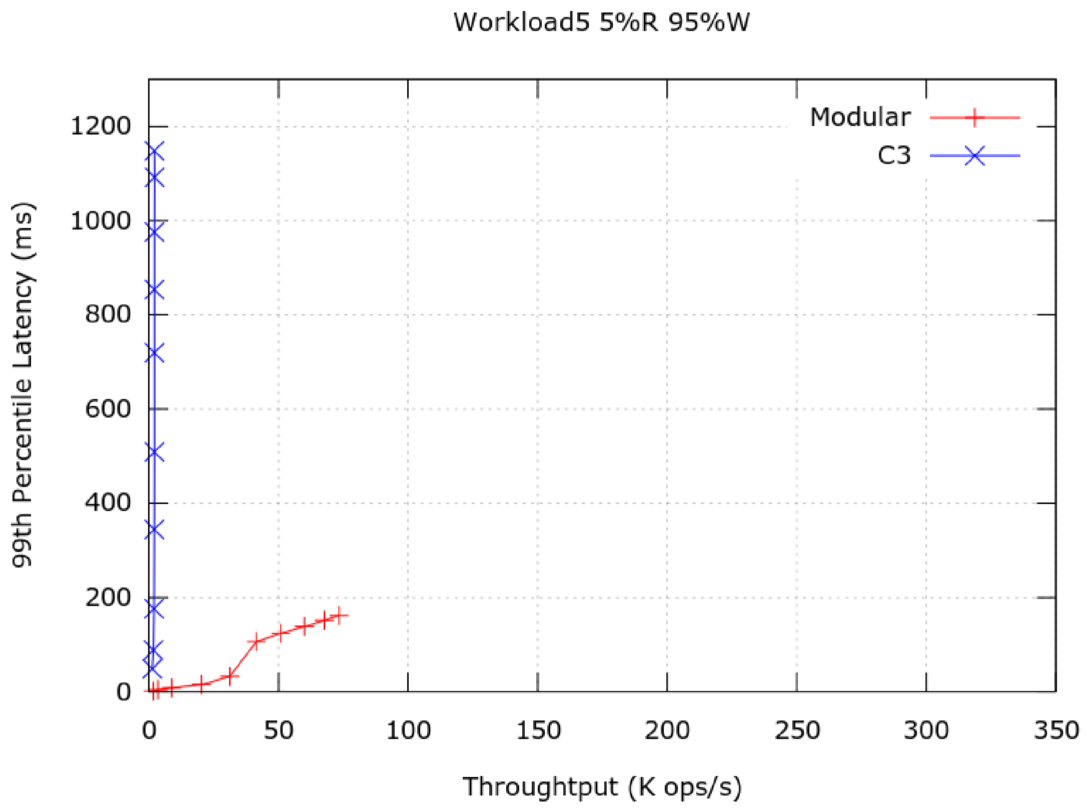
I.2 Modular with C3



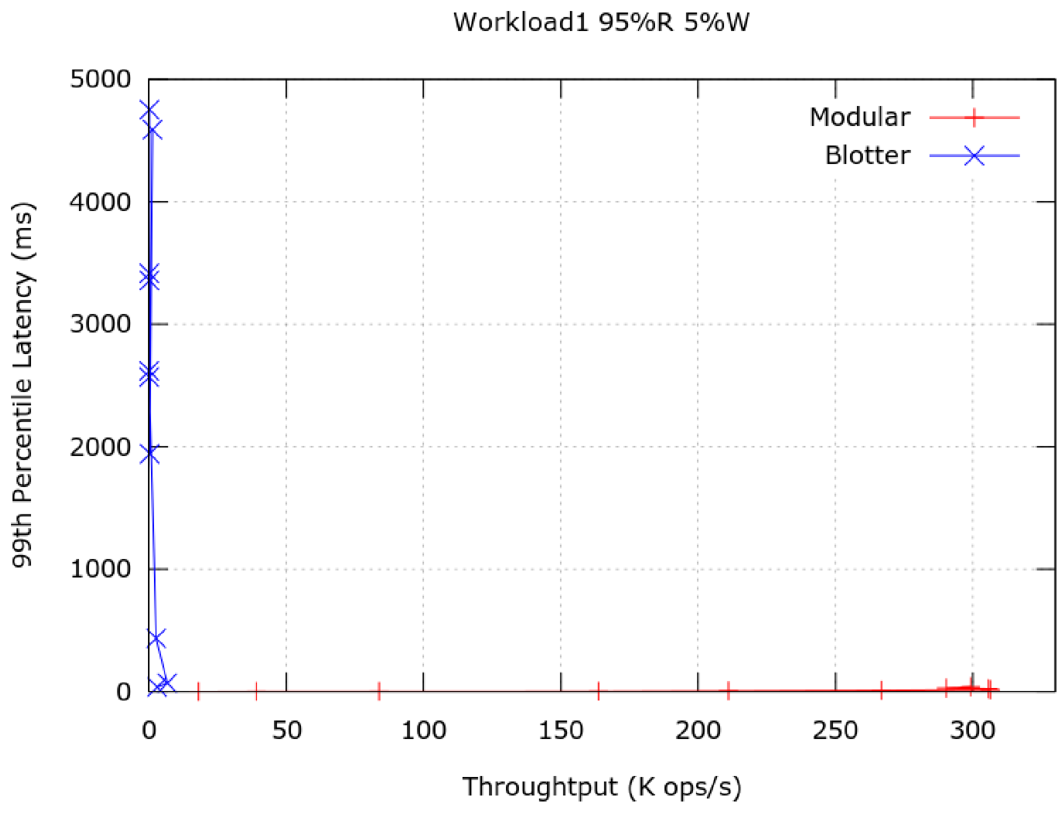


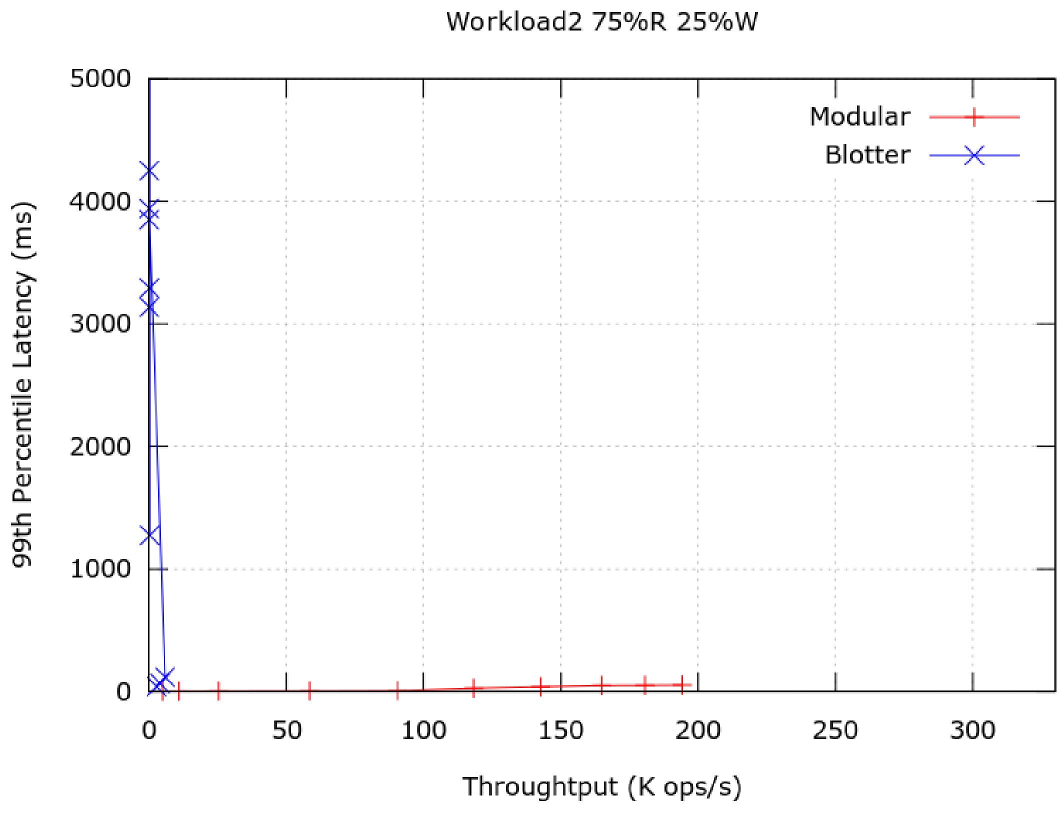

Workload3 50%R 50%W
99th Percentile Latency (ms)
Throughtput (K ops/s)
Modular
C3
0
200
400
600
800
1000
1200
0
50
100
150
200
250
300
350

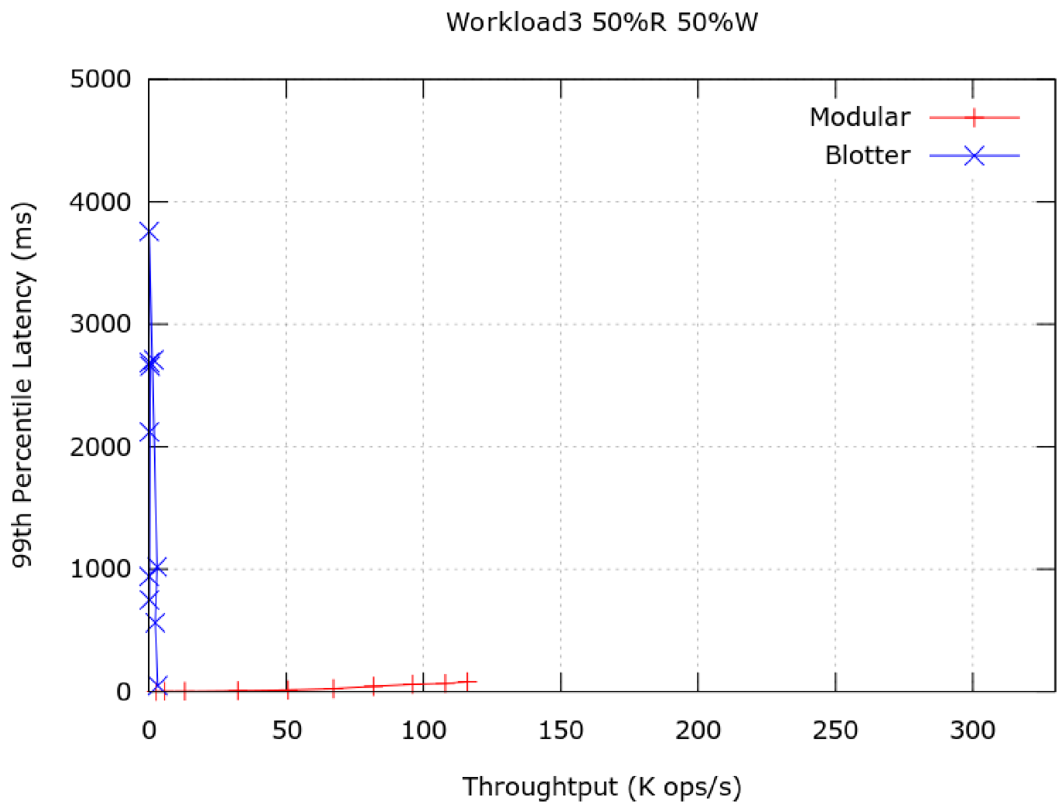

Workload4 25%R 75%W
99th Percentile Latency (ms)
Throughtput (K ops/s)
Modular
C3
0
200
400
600
800
1000
1200
0
50
100
150
200
250
300
350

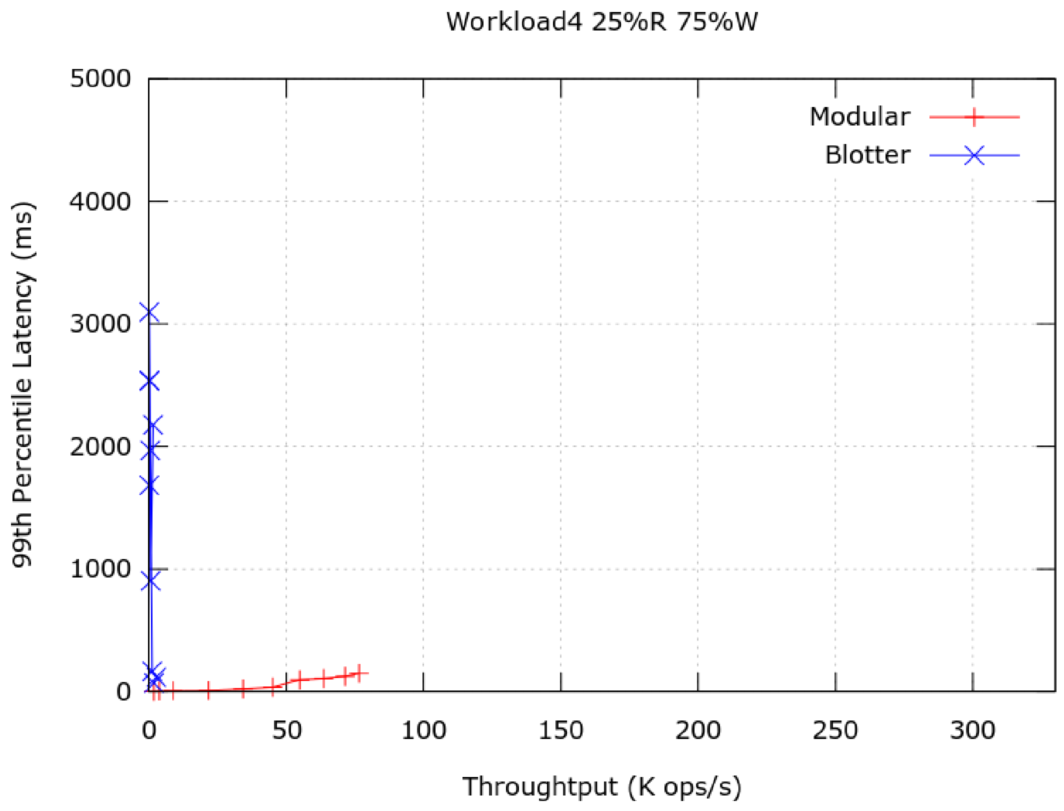

Workload5 5%R 95%W
Modular
C3
99th Percentile Latency (ms)
Throughtput (K ops/s)
0
200
400
600
800
1000
1200
0
50
100
150
200
250
300
350

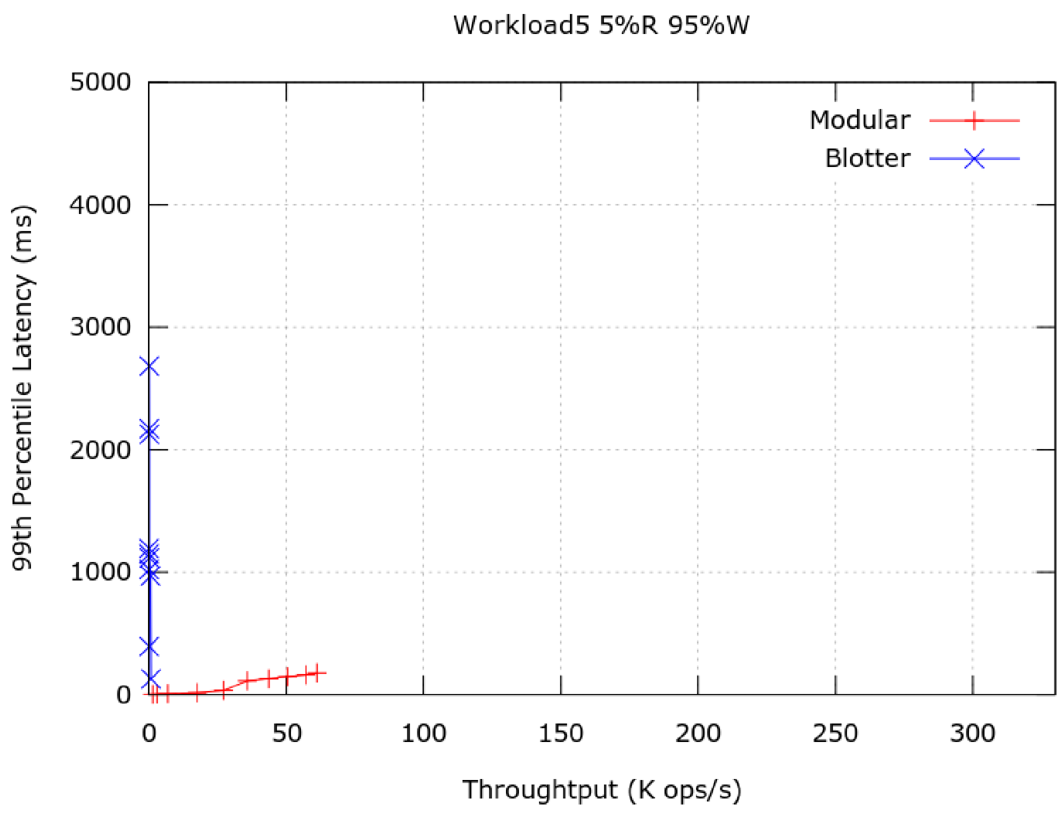
I.3 Modular with Blotter






Workload3 50%R 50%W
Modular
Blotter
99th Percentile Latency (ms)
Throughput (K ops/s)
0
1000
2000
3000
4000
5000
0
50
100
150
200
250
300


Workload4 25%R 75%W
Modular
Blotter
99th Percentile Latency (ms)
Throughput (K ops/s)
0
1000
2000
3000
4000
5000
0
50
100
150
200
250
300


Workload5 5%R 95%W
Modular
Blotter
99th Percentile Latency (ms)
Throughtput (K ops/s)
0
1000
2000
3000
4000
5000
0
50
100
150
200
250
300