Consistent Range-Queries in Distributed Key-Value Stores

Providing Consistent Range-Query Operations for the CATS NoSQL Database

SEYED HAMIDREZA AFZALI

Degree project in
Software Engineering of Distributed Systems
Second cycle
Stockholm, Sweden 2012
Consistent Range-Queries in Distributed Key-Value Stores

Providing Consistent Range-Query Operations for the CATS NoSQL Database

SEYED HAMIDREZA AFZALI

Master of Science Thesis in
Software Engineering of Distributed Systems

At the Swedish Institute of Computer Science (SICS)

Supervisor: Dr. Jim Dowling
Examiner: Prof. Seif Haridi

School of Information and Communication Technology,
Royal Institute of Technology (KTH)

Stockholm, Sweden 2012

TRITA-ICT-EX-2012:104
Abstract

Big Data is data that is too large for storage in traditional relational databases. Recently, NoSQL databases have emerged as a suitable platform for the storage of Big Data. Most of them, such as Dynamo, HBase, and Cassandra, sacrifice consistency for scalability. They provide eventual data consistency guarantees, which can make the application logic complicated for developers. In this master thesis project we use CATS; a scalable and partition tolerant Key-Value store offering strong data consistency guarantees. It means that the value read is, in some sense, the latest value written. We have designed and evaluated a lightweight range-query mechanism for CATS, that provides strong consistency for all returned data items. Our solution reuses the mechanism already available in CATS for data consistency. Using this solution CATS can guarantee strong data consistency for both lookup queries and range-queries. This enables us to build new classes of applications using CATS. Our range-query solution has been used to build a high level data model, which supports secondary indexes, on top of CATS.
In the memory of my beloved grandmother
# Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>ix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgements</td>
<td>xi</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Thesis Outline .......................... 3

## 2 Related Work and Background

2.1 Linearizability .......................... 5
2.2 CAP Theorem .............................. 5
2.3 NoSQL Databases .......................... 6
   2.3.1 Range-Queries in NoSQL Databases .... 7
2.4 Kompics Framework ....................... 9
2.5 ABD Algorithm for Atomic Registers ..... 10
2.6 Chord .................................. 11
2.7 CATS NoSQL Database .................... 12
   2.7.1 Consistent Quorums ................. 13
   2.7.2 System Architecture ................. 14
2.8 Yahoo! Cloud Serving Benchmark (YCSB) 14

## 3 Model

3.1 Specification ........................... 18
3.2 Range-Query Operation ................... 19
3.3 Read-Range Operation .................... 23
3.4 The Collector ............................ 29
3.5 Policies ................................ 30
3.6 Tolerating Node Failures ............... 34

## 4 Implementation

4.1 Consistent Range-Query (CRQ) Component 35
4.2 CATS Interface Client (CIC) ............. 37
4.3 Basic Client ............................ 39
4.4 Experiments ............................. 39
   4.4.1 Scenario Scripts ................... 40
Acknowledgements

This thesis project would have not been possible without the supervision of Dr. Jim Dowling. I would like say a special thanks to him for his continuous support, for the valuable discussions we had and also for guiding me to make use of tools that were extremely beneficial during this project.

Furthermore, I have received valuable feedback and guidance from Cosmin Arad and Tallat Mahmood Shafaat. I would like to thank them for their support during my thesis work. I would also like to thank Dr. Sarunas Girdzijauskas and Muhammad Ehsan ul Haque for sharing their experience with me.

I am grateful to my professors at the Royal Institute of Technology (KTH), especially Prof. Seif Haridi who examined my master thesis.

I also acknowledge the Swedish Institute of Computer Science (SICS) for funding this research project.

Finally, I would like to thank my family for their support during my life and studies. I would not be where I am today without their care, support and love.
Chapter 1

Introduction

A very large amount of data is being generated every day. Texts and digital media published on social networks are just a fraction of this data, which is growing very fast. NoSQL databases, e.g. Dynamo [1], Voldemort [2], Riak [3], BigTable [4], HBase [5], Cassandra [6], have been introduced in response to this growth. NoSQL databases are a class of non-relational storage systems that do not provide all of the ACID\(^1\) properties of databases. They are designed to store very large amount of data and to be highly scalable by distributing data on many machines connected through a computer network. However, they have different architectures and target different types of applications. Some key-value stores, e.g. BigTable, HBase and Cassandra, support range-queries, while some others, e.g. Dynamo and Voldemort, do not provide range-queries internally. Pirzadeh et al. proposed different solutions to implement range-queries on top of lookup queries in Voldemort [7].

According to CAP Theorem [8], it is impossible to provide all of the Consistency, Availability and Network Partition Tolerance guarantees in a distributed system. It is critical to provide Network Partition Tolerance because network partitioning is inevitable. Consequently, a distributed system may provide either Availability and Network Partition Tolerance (AP), or Consistency and Network Partition Tolerance (CP). Most NoSQL databases, e.g. Dynamo, Voldemort and Cassandra, are optimized for Availability and Network Partition Tolerance (AP). They sacrifice strong consistency for high availability, thus providing a weaker form of data consistency known as *eventual consistency*. For example, in Dynamo, "a put() call may return to its caller before the update has been applied at all the replicas, which can result in scenarios where a subsequent get() operation may return an object that does not have the latest updates" [1]. While eventual consistency may be acceptable in some applications such as social networking web applications, it makes the application logic complicated for the applications in which inconsistent data is not acceptable. Developers need to deal with data timestamps in order to avoid data inconsistency and outdated data.

CATS [9] is a scalable and partition tolerant key-value store developed at the

\(^1\) Atomicity, Consistency, Isolation, Durability
Swedish Institute of Computer Science (SICS). It provides strong data consistency guarantees for lookup queries with respect to linearizability. Linearizability is a correctness condition for concurrent objects, that, provides the illusion that each read and write operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response. It is a non-blocking property, which, means a pending invocation on an operation in not required to wait for another invocation to complete [10]. CATS provides Get and Put operations to read and write key-value pairs. It implements a fail-silent algorithm for atomic register, or shared memory, with multiple readers and multiple writers. An atomic register is a shared register providing linearizable read and write operations. The algorithm for reads and writes in the shared atomic register is known as Read-Impose Write-Consult-Majority\(^2\) [11] (a.k.a. ABD\(^3\) [12]). It is used in both Get and Put operations. This algorithm provides strong data consistency with respect to linearizability. Each atomic register (i.e., a key-value pair) is replicated in \(r\) replica machines. It is assumed that the majority of replicas are always alive in every replication group.

Most key-value stores that do not support a range-query API (e.g. Voldemort), employ randomized hash-partitioning to distribute data items on different machines. Although randomized hash-partitioning has its own benefits such as providing automatic load-balancing, it makes range-queries impractical. Using randomized hash-partitioning, sequential keys are stored on random machines. It may be needed to contact all machines in order to query a small range. However, it is not impossible to do range-queries on these key-value stores. Some solutions have been proposed to implement range-queries in such systems, that, of course come with some additional overhead [7]. Normally, key-value stores that offer range-query (e.g. HBase) use range-partitioning, or order-preserving hashing. Order-preserving hashing results in sequential hash codes for sequential keys. Therefore, sequential ranges of data are stored on the same or neighbor machines. Employing order-preserving hashing makes it possible to do range-query by contacting a number of neighbor machines. Cassandra supports both hash partitioners. A user may select to use either randomized hashing to achieve automatic load-balancing, or order-preserving hashing to enable range-query support. When using order-preserving hashing, load-balancing should be handled manually.

The goal of this master thesis project is to provide CATS with a lightweight range-query mechanism that offers strong data consistency guarantees for all returned key-value pairs with respect to linearizability.

In order to avoid the extra overheads of contacting large numbers of servers for range-queries, we use order-preserving hash partitioning in CATS. The range-query API takes a range, which, may be inclusive or exclusive on either endpoints, and a positive integer number, that, limits the number of items to be returned. Our solution supports both range-queries where no consistency guarantees are required

\(^2\)Refer to Appendix A, Algorithm A.1 and Algorithm A.2
\(^3\)Proposed by Attiya, Bar-Noy and Dolev
on the underlying data, as it is immutable, and consistent range-queries for mutable data.

There are two main problems that are targeted in the range-query mechanism we propose, which, are:

- Finding first $l$ keys with non-null values in the range according to the data stored on the majority of replicas in the replication group
- Returning key-value pairs for those keys, providing strong consistency for values, with respect to linearizability

In the second problem, our solution exploits the ABD component already available in CATS to provide strong data consistency. We employ a mechanism to determine the consistency of the data (i.e., the value in key-value pairs) in the range. The goal is to limit the use of ABD algorithm to cases where there are doubts about the data consistency. Therefore, we avoid running ABD algorithm for all key-value pairs that are going to be returned. This leads to a lightweight mechanism for consistent range-query.

Using our proposed solution, CATS supports range-queries, providing strong data consistency for both lookup queries and range-queries.

1.1 Thesis Outline

In the next chapter we will go through related work and background information. In chapter 3, we propose a lightweight model for consistent range-query in CATS. Implementation details are discussed in chapter 4. We present evaluation methods and evaluation results for our range-query solution in chapter 5. The last chapter is devoted to conclusion and future work.
Chapter 2

Related Work and Background

2.1 Linearizability

Linearizability is a strong form of data consistency for distributed shared-data. It is a correctness condition for concurrent objects, which, provides the illusion that each read and write operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response. It is a non-blocking property which means a pending invocation on an operation in not required to wait for another invocation to complete [10]. According to [8], linearizability refers to the property of a single indication/response operation sequence. A system is linearizable if every object in the system is linearizable.

2.2 CAP Theorem

There are three desirable properties, which we would like to provide in a distributed application, e.g. a distributed web service:

- Consistency
- Availability
- Partition-tolerance

According to CAP theorem, it is impossible to provide guarantees for all of these properties [8].

Consistency refers to providing strong data consistency. Eventual data consistency may be acceptable in some applications such as social networking web applications, but there are applications in which inconsistent data is not acceptable, e.g. electronic health record applications. Many applications expect strong data consistency.

Availability refers to the highly availability of the service. It means every request should succeed and receive a response. Of course, availability depends on the
underlying network services. Therefore, the goal of a highly available distributed system is to be as available as the network services it uses [5].

Partition-tolerance refers to fault tolerance in the presence of network partitions. It is critical to provide network partition tolerance because network partitioning is inevitable. Consequently, a distributed system may provide either Availability and Network Partition Tolerance (AP), or Consistency and Network Partition Tolerance (CP).

2.3 NoSQL Databases

Big Data is data that is too large for storage in traditional relational databases. Recently, NoSQL databases, e.g., Dynamo [1], Voldemort [2], Riak [3], BigTable [4], HBase [5], Cassandra [6], have emerged as a suitable platform for the storage of Big Data. They try to be highly scalable by distributing data on several machines connected through a computer network. However, they do not provide all of the ACID (i.e., Atomicity, Consistency, Isolation, Durability) properties of databases. There are many NoSQL databases with different architectures targeting different types of applications. Some of them, like Voldemort, provide a simple key-value data model. Some other NoSQL databases, like BigTable, support a richer data model. Here is how Oracle describes NoSQL databases [13]:

"NoSQL databases are often key-value stores, where the values are often schemaless blobs of unstructured or semi-structured data. This design allows NoSQL databases to partition their data across commodity hardware via hashing on the key. As a result, NoSQL databases may only have partial or no sql support. For example, group-by and join operations are typically performed at the application layer, resulting in many more messages and round trips compared to performing a join on a consolidated relational database. To reduce these round trips, the data may be denormalized such that a parent-child relationship can be stored within the same record."

We briefly mention the properties and the data model of three NoSQL Databases: HBase, Cassandra and Voldemort.

HBase

HBase [5] is an open source version of Google’s BigTable [4] written in Java. It is a part of Apache Hadoop [14] project and runs on top of the Hadoop Distributed File System (HDFS) [15]. Like BigTable, data is maintained in lexicographic order by row key. HBase uses the concept of column-family, introduced in BigTable, for the data model. Each row of data can have several columns and columns are grouped into column-families [4]. Column-families are sets of columns in a row that are typically accessed together. All values of each column-family are stored sequentially.
HBase offers fast sequential scans on rows and columns within a column family in a row. Write operations are fast and read operations are slower compared to writes. HBase provides strong data consistency and supports range-query operations.

**Project Voldemort**

Project Voldemort [2] is an open source implementation of Amazon’s Dynamo [1]. It is a highly available distributed database, providing a simple key-value data model. All read and write operations are allowed and handled, even during network partitions. Voldemort uses a Gossip based membership algorithm to maintain information about other nodes, which, is not a hundred percent reliable. Data update conflicts are detected using a vector clock. They are resolved using conflict resolution mechanisms. The client is also involved in conflict resolution. Voldemort uses randomized hash-partitioner to provided automatic load-balancing. It offers eventual data consistency model. Voldemort provides the option of using different storage engines, e.g. MySQL. There is no support for range-queries in Voldemort. Pirzadeh et al. have proposed different solutions to implement range-query operation on top of lookup queries in Voldemort [7].

**Apache Cassandra**

Apache Cassandra [6] is a NoSQL database initially developed by Facebook. Currently it is a project of the Apache Software Foundation [16]. Similar to HBase, Cassandra uses the rich data model of BigTable [4], and provides faster write operations compared to read operations. It is designed based on the architecture of the Amazon’s Dynamo [1]. Similar to Voldemort, Cassandra uses a Gossip based membership algorithm to maintain information about other nodes, which, is not a hundred percent reliable. It provides eventual data consistency, but consistency level is selectable by the client. Cassandra provides both randomized hash partitioner and ordered-preserving hash partitioner. User may select to use either randomized hashing to achieve automatic load-balancing, or order-preserving hashing to enable range-query support. If the user selects to use order-preserving hashing, Cassandra supports range-query operations. In that case, manual load-balancing is necessary if data is skewed.

### 2.3.1 Range-Queries in NoSQL Databases

Range-query (a.k.a. scan) is an operation to query a database about a bounded sequence of keys. There are different APIs for range-query operations. A basic API for range-query, takes `start_key` and `limit` parameters. The operation returns first `limit` items starting from `start_key`. However, range-query APIs typically takes a range parameter, consisting of a start key, an end key and information about the clusivity of both endpoints. For example, `(start_key, end_key]` represents a range of keys starting from `start_key` (exclusive) and ending at `end_key` (inclusive).
Some of the NoSQL databases, e.g., HBase, perform complete range scan on a given range of keys, i.e., retrieving all available keys, and corresponding values, in the range. This API may be problematic when the user has no idea about the amount of data stored in the range. More controllable range-query APIs, e.g., the Cassandra Thrift API 1.0, take a parameter as an upper bound for the number of items to be returned. They take a range and a positive integer number, limit. Performing a range-query with parameters \([start\_key, end\_key]\) and limit, results in retrieving first limit items between two endpoints of the range, starting from the start_key.

Range-query is not supported by some of NoSQL databases, e.g., Dynamo and Voldemort. Most of NoSQL databases that do not support range-queries, employ randomized hash-partitioning to distribute data items on different machines. Randomized hash-partitioning provides an automatic load-balancing for data distribution on machines, but it makes the range-query impractical. Using randomized hash-partitioning, sequential keys are stored on random machines. It may be needed to contact all machines in order to query a small range. However, it is not impossible to implement range-query operation in such NoSQL databases. Pirzadeh et al. have proposed three solutions to implement range-query operation on top of lookup queries in Voldemort, which, of course come with some additional overhead:

**Indexing Technique:** In their first solution for range-query, they propose building a range index on top of Voldemort. They implement a BLink Tree similar to B-Tree. BLink Tree is a distributed form of B-Tree.

**No-ix Technique:** Their second solution is similar to a shared-nothing parallel RDBMS. It requires to contact all the data partitions, i.e., Voldemort nodes. They propose a two-step procedure for the range-query operation:

1. Performing range-query on all Voldemort nodes, in parallel
2. Aggregate and merge partial results and return the final result.

**Hybrid Technique:** They also propose a combination of Indexing and No-ix techniques introduced earlier. The Hybrid technique benefits from both solutions.

Typically, NoSQL databases that support range-queries (e.g., HBase) use range-partitioning, or order-preserving hashing. Order-preserving hashing results in sequential hash codes for sequential keys. Therefore, sequential ranges of data is stored on same or neighbor machines. Employing order-preserving hashing makes it possible to do range-queries by contacting a number of neighboring machines. Cassandra provides both hash partitioner. The user may select to use either randomized hashing to achieve automatic load-balancing, or order-preserving hashing to enable range-query support. When using order-preserving hashing, load-balancing should be handled manually.
2.4 Kompics Framework

Kompics \cite{20,21,22} is an event-driven component model for building distributed systems. Protocols are implemented as Kompics components. Kompics components execute concurrently. A main component encapsulates several loosely-coupled components, which, communicate asynchronously by message-passing. Messages are transmitted through bidirectional typed ports connected by channels. Components may use services provided by other components, or may provide services to other components. For example, a component may use the Timer component to schedule timeouts, or may use the Network component to communicate with other peers over the network. Kompics contains a simulator component, which, can run the same code in simulation mode according to a simulation scenario. It is useful for debugging and repeatable large-scale evaluations. The simulator component also provides the Network and Timer abstractions and a generic discrete-event simulator.

Concepts in Kompics

**Event:** Events are passive and immutable typed objects with a number of attributes. Components use events as messages for asynchronous communications. Events are represented using \(\text{Event}\) graphical notation.

**Port:** Ports are typed and bidirectional component interfaces through which components send and receive messages (i.e., events). Components that implement a protocol provide a port that accepts events representing indication and response messages. Directions of a port are labeled as positive (+) or negative (-), which, are in and out directions. Ports act as filters for events that are sent or received. They allow a predefined set of event types to pass on each direction. Ports are represented using \(\text{Port}\) graphical notation.

**Channel:** Channels connect component ports of the same type. They forward events in both directions in FIFO order. For example, a component that provides Timer service and a component that uses the Timer service, both have ports of the type Timer which are connected with a channel. Channels are represented using \(\text{channel}\) graphical notation.

**Handler:** An event handler is a procedure inside a component. It is executed when the component receives a particular type of event, which, is a message from another component. It is the place where the logic of how to deal with an event is implemented. Handlers may mutate the local state of the component. They may also trigger new events. Handlers are represented using \(\text{handler}\) graphical notation.

**Subscription:** Subscriptions bind event handlers to component ports. This is how handlers are told to handle events received on a particular port of a component. The event type that the handler accepts must be one of the allowed event types on
the port. Subscriptions are represented using graphical notation.

**Component:** Protocols are implemented as Kompics components. Components are concurrent event-driven state machines. They communicate asynchronously by message-passing, i.e., sending and receiving events. Components provide services to other components through Port interfaces. They also contain local state variables, event handlers and subscriptions. Components bind each event handler to a port through subscription. Events received on a port of a component are accepted by a handler. Two components having the same port type may be connected through a channel. Components are represented using graphical notation.

### 2.5 ABD Algorithm for Atomic Registers

ABD [12] algorithm proposed by Attiya et al., is an algorithm to implement atomic registers with multiple readers and multiple writers. An array of atomic registers can be used as a shared memory. The algorithm is also referred to as Read-Impose Write-Consult-Majority \((N, N)\) [11]. ABD implements atomic registers shared between processes communicating with message passing. It is more convenient for the programmers to use shared memory rather than message passing. Programmers can use this abstraction to develop algorithms for shared memory, while ABD uses message passing to emulate a shared memory. ABD provides strong data consistency with respect to linearizability [10]. It means that the value read is, in some sense, the latest value written. Once ABD returns a value, it will not return an older value. Linearizability provides the illusion that each operation invoked by concurrent processes takes effect instantaneously at some point between its invocation and its response.

ABD has a simple interface with two operations: Read and Write. Registers are initialized with a special value \((\perp)\). A process may invoke a Read operation to read the value of a register, by triggering a \(< Read >\) event. similarly, a Write operation is invoked by triggering a \(< Write \mid v >\) event with a value \(v\). In every write operation, ABD provides a timestamp, which, is unique on the corresponding register. The register stores the value together with this timestamp. When a process invokes an operation, ABD communicates with other processes using message-passing, and sends a response event to that process. An operation invoked by a process, completes when the process receives a response. A correct process is a process that lives long enough to receive responses to all of its invoked operations. An operation invoked by a correct process always completes. All correct processes access a register sequentially. When a process invokes an operation on a register, it does not invoke more operations on the same register, before the operation completes. However, different processes may access the same register concurrently, or a single process may access different registers at the same time.

We are interested in the fail-silent version of the algorithm. The algorithm is
provided in Appendix \[A\] according to [11]. A fail-silent distributed algorithm does not make any assumption on failure detection. ABD works in the presence of process failures. However, it is assumed that the majority is always alive.

2.6 Chord

Chord [23] is a highly scalable distributed lookup protocol. It maps a given key onto a node in a peer-to-peer system. Chord can be used to implement a distributed key-value store. A key-value pair can be stored in the node that Chord maps the key onto it. Chord efficiently locates the node that stores a particular data item. It can answer lookup queries as nodes join and leave the system.

Chord assigns keys to nodes using consistent hashing [24]. It orders identifiers in an identifier circle (i.e. ring) modulo $2^m$. Key $k$ is assigned to its successor node. Successor node is the first node which has an identifier equal to $k$ or follows $k$ on the identifier circle. If the identifier circle is ordered clockwise, successor is the first node clockwise from key $k$. Also, predecessor is the previous node on the identifier circle. Figure 2.1 illustrates a Chord ring with $m = 3$ and three nodes 0, 2, and 4. The successor of identifiers 1, 4 and 5 are respectively nodes 2, 4 and 0. Therefore, key 1 would be located at node 2, key 4 at node 4, and key 5 at node 0.

As the nodes join and leave the system, Chord adapts with minimal reassignments. When a node $n$ joins the ring, some keys which were assigned to its successor, would be reassigned to $n$. Similarly, when a node leaves the ring, its assigned keys would be reassigned to its successor. For example, if a new node with identifier 6 joins the ring in Figure 2.1, the key '5' will be reassigned to it.

Chord maintains pointers to successor nodes. To make routing possible, it suffices that each node have a pointer to its successor. Lookup queries can be forwarded

\[1\] Algorithms [A.1] and [A.2]
node by node to reach the successor node of the given key. This is a correct solution but it is not efficient. A lookup may visit all nodes before it reaches the responsible one. To make the routing process faster, Chord maintains additional routing information. In a system with \( N \) nodes, each node stores a routing table of the size \( O(\log N) \) called finger table. The first entry in the table at node \( n \) is its immediate successor on the ring. The \( i \)th entry in the table points to the first node that succeeds \( n \) by at least \( 2^{i-1} \) on the identifier circle, i.e., \( \text{successor}(n + 2^{i-1}) \), where \( 1 \leq i < m \). Chord maintains the ring and finger tables as nodes join and leave the system.

Each Chord node maintains a successor-list, which contains its first \( r \) successors on the Chord ring. If the successor of a node fails, the node selects the first live node in the list as its new successor. Successor-list is also used to provide mechanisms for data replication. Chord can inform higher level software about first \( k \) successors of a given key. An application can then replicate the data at \( k \) successors of the corresponding key. Chord nodes also inform higher layer software about changes in their successor-list. Thus, the higher level software replicates data on new nodes.

### 2.7 CATS NoSQL Database

CATS [9] is a scalable, self-organizing and partition tolerant key-value store that guarantees strong data consistency with respect to linearizability [10]. The API is simple and provides two operations:

- \( \text{Put}(\text{Token key}, \text{String value}) \)
- \( \text{key} \leftarrow \text{Get}(\text{Token key}) \)

A Client may use \( \text{Put}/\text{Get} \) operations to write/read key-value data items. The key is of the type \( \text{Token} \), which can be string, long or byte arrays. The value is of the type string. To store complex data objects in CATS, the value should be serialized into string by the client.

CATS servers form a ring overlay based on the principle of consistent hashing [24] and use successor-list replication technique similar to Chord [23]. Each server is mainly responsible for the range of keys between its id and its predecessor’s id, i.e., \( (\text{predecessor}_\text{id}, \text{node}_\text{id}) \). As illustrated in Figure 2.2, every data item is also replicated by a number of successor servers (replication group).

CATS uses ABD [12], which is a quorum-based algorithm [25], to read/write key-value pairs from/to servers in a replication group. In Figure 2.2, the replication degree is three and a majority quorum can be formed by any set of two nodes from the nodes in the replication group, i.e., \{b, c, d\}. Different nodes should have the same view about the members of a replication group. Otherwise, it may lead to non-intersecting sets for quorums, which can happen in the presence of false failure suspicions. Figure 2.3 shows different replication groups for keys in the range \((a,b]\) from the perspective of two different nodes. An operation may consider the set...
2.7. CATS NOsql DATABASE

Figure 2.2. Replication group for keys in range \((a, b]\), using consistent hashing with successor-list replication and a replication degree of three.

\{b, c\} for majority quorum while another operation considers the set \{d, e\}. This can violate linearizability and lead to inconsistent results.

Majority quorums for read and write operations can fail to achieve linearizability in a key-value store that provides data replication and uses consistent hashing for automatic reconfiguration. CATS introduces consistent quorums \[26]\ to overcome this problem and provide linearizability in such systems.

Figure 2.3. Inconsistent replication groups from the perspective of two different nodes for the keys in the range \((a, b]\), using consistent hashing with successor-list replication and a replication degree of three.

2.7.1 Consistent Quorums

Quorum-based protocols work based on collaboration of a majority of participants. In such protocols, a coordinator sends requests to a group of participants. Each participant processes the received message and may send a response to the coordinator. The coordinator waits for a majority of responses to complete an operation \[25]\ . The operation will never complete if the coordinator does not receive enough responses from the participants. The operation times out after a predefined period of time and the coordinator can retry with new requests containing new identifiers. Quorums of different operations should intersect to satisfy the safety property of the protocol. \[9]\ 

In dynamic systems, where nodes may be suspected to be failed, it may happen that a node has an inconsistent view of the group membership. For example, it may happen in CATS that a node considers itself to be responsible for a key range
while it is not. This may lead to non-intersecting sets for quorums. To avoid this problem CATS uses the idea of maintaining a membership view of the replication group at the nodes that consider themselves to be a replica for a specific range of keys. In [9], authors define consistent quorum as the following:

**Definition.** "For a given replication group $G$, a consistent quorum is a regular quorum of nodes in $G$ which are in the same view at the time when the quorum is assembled."

Nodes include their current view in all responses and quorums are formed considering messages with the same view. In case of replication group membership changes, CATS uses a group reconfiguration protocol, which is based on the Paxos consensus algorithm [27], to consistently reconfigure the group membership views at all group members.

### 2.7.2 System Architecture

CATS was implemented in Java using the Kompics framework [20]. One of the fundamental tasks in CATS is to build a ring overlay based on the principle of consistent hashing [24]. CATS uses a periodic stabilization protocol [23] to maintain ring overlay pointers in the presence of node dynamism. Furthermore, it uses a ring unification protocol [28] to make the ring overlay partition-tolerant. To maintain the ring, CATS also relies on an unreliable failure detector [29], which may suspect correct nodes to have crashed. These protocols do not guarantee lookup consistency [30], which may lead to non-overlapping quorums. [9]

CATS uses Cyclon gossip-based membership protocol [31], which is a peer sampling service, to build a full membership view of the whole system. It enables CATS to efficiently look up the responsible replicas for a given key [32]. CATS also uses epidemic dissemination to propagate membership changes to the whole system. It relies on the Cyclon protocol to quickly propagate churn events [33].

CATS monitors the ring overlay and reconfigures the replication group memberships according to the changes in the ring. It uses a group reconfiguration protocol based on the Paxos consensus [27] algorithm.

### 2.8 Yahoo! Cloud Serving Benchmark (YCSB)

Yahoo! Cloud Serving Benchmark (YCSB) [34] is an open source benchmarking framework for evaluating the new generation of databases targeting big data, i.e., NoSQL databases. Recently, a large number of NoSQL databases (e.g., Cassandra, HBase and Voldemort) have emerged with different architectures targeting different types of applications. It is difficult for developers to choose the appropriate one for their application. YCSB facilitates to understand the trade-offs between these systems and workloads in which they perform better. The YCSB Client is a Java program made up of four modules:
• Workload Executor
• Database Interface Layers
• Client Threads
• Statistics

The workload executor module generates data and operations for the workload. YCSB provides a package of workloads. Each workload defines a set of operations, data size and request distributions. Supported operations are Insert, Update, Read and Scan, i.e., range-query. Users can change a set of parameters to perform different workloads. It is also possible to develop new workload packages by defining new parameters and writing Java code. To make random decisions (e.g., which operation to perform, which record to choose, how many records to scan), YCSB supports different types of random distributions, such as Uniform and Zipfian. For example, when choosing records for read operations, Uniform distribution will pick records uniformly at random but Zipfian distribution will give more chance to some extremely popular records while most records are unpopular.

The database interface layer is an interface between YCSB and a particular database. It translates operations generated by the workload executor, into the format required by the API of the targeting database. YCSB provides database interface layer for some databases, such as HBase, Cassandra, Voldemort and MongoDB. Users can develop new database interfaces in Java, by extending a Java abstract class. A database interface layer has been developed for CATS. We modified this database interface layer to add support for range-query (i.e. scan) operations.

The workload executor runs multiple client threads. Client threads use database interface layer to execute operations. They also measure the latency and throughput of the operations they execute. The load on the database is controlled by some parameters and the number of client threads.

The statistics module aggregates measurements and prepares a report about the latencies and the throughput. The report presents the average, 95th and 99th percentile latencies. The statistics module also provides histogram or time series of the latencies.
Chapter 3

Model

Range-query is an operation that queries about a range of keys. It returns a sequential set of items in a given range. User provides an upper bound for the number of items to be returned. This is how our model defines data consistency in range-query: consistent range-query is a range-query operation in which, strong data consistency is guaranteed for each and every data item returned, with respect to linearizability. User must provide a range (i.e., two endpoints and their clusivity conditions) in every range-query request. The API takes an optional limit for the number of items to be returned. A default value and an upper bound value for the limit is defined in the system. The same API can be used to execute both normal range-queries and consistent range-queries. Here are two interfaces for the range-query request:

\[
\text{Range-Query (} \text{requestId, consistent, start, startInclusive, end, endInclusive, limit, collectorType)}
\]

\[
\text{Range-Query (} \text{requestId, consistent, range, limit, collectorType)}
\]

\text{requestId} is the operation id provided by the user. The range-query response will be marked with the same id. The user may choose the consistency of the range-query operation by the boolean parameter \text{consistent}. The parameters \text{start} and \text{end} determine two endpoint keys of the range. Keys in CATS are of the type \text{Token}. We use a version of CATS with String Tokens. The parameters \text{startInclusive} and \text{endInclusive} determine whether endpoints are inclusive or exclusive. The \text{limit} is an upper bound for the number of the items to be returned. The \text{Range} class provides information about a range, including two endpoints and their clusivity conditions. \text{collectorType} determines whether to collect the range-query result on a CATS server or on the client. We will see more on this later in this chapter.

For example, \text{Range-Query (1, True, a, True, b, false, 1000, SERVER)}, execute a consistent range-query operation with a request id of "1". It queries about first 1000 items in the range [a, b). The result will be collected on a CATS server before delivering to the client. In this document, we will use the following notations for range-query request and response.
A range-query operation requesting first 1000 items in the range \([a, b)\) and a range-query response, returning 900 items for the same range:

\[
\text{range-query} \ [a, b) \ #1000 \\
\text{range-query.resp} \ [a, b) \ #900
\]

**An Overview**

The user sends a range-query request to a CATS node (i.e. CATS server) asking for \(l\) items. The CATS node forwards the request to a node that is responsible for the requested range. The responsible node executes a consistent range-query for the sub-range it is responsible for. To find first \(l\) keys in the range, the responsible node queries all replicas for first \(l\) keys they store in the range. Number of the replicas depends on the CATS settings. If replicas have some ABD operations in progress, they may respond with different sets of keys. The responsible node waits for the majority of replicas to respond. It aggregates responses and prepares a list of keys according to the view of the majority. Our solution guarantees strong data consistency for each item in this list.

After finding first \(l\) keys, our solution reuses the ABD component that is already available in CATS, to provide strong data consistency. We employ a mechanism to determine data consistency in a range. The goal is to limit the use of ABD algorithm to cases where there are doubts about the data consistency. After obtaining consistent values for all keys in the list, the responsible node responds with the list of key-value pairs. If the responsible node could not find enough items in its sub-range, it sends a new request to the next CATS node. The request queries the remaining sub-range for the remaining number of items. A node that executes a range-query operation, sends the response to a range-query collector. The collector aggregates responses for sub-ranges and provides the final result of the range-query.

### 3.1 Specification

*Null* is an acceptable value for CATS *Put* and *Get* operations. However, our range-query model does not return keys with null values. There are two main problems that we target in our model:

- Finding first \(l\) keys with non-null values in the range according to the data stored on the majority of replicas
- Returning key-value pairs for those keys, providing strong data consistency for values with respect to linearizability.

Module 3.1 shows the interface and properties of the consistent range-query operations in CATS. Algorithms for the consistent range-query operation are available in the Appendix B algorithms B.1 to B.5.
Module 3.1 Interface and properties of CATS consistent range-query

Module:

Name: Consistent-Range-Query, instance crq.

Events:

Request: <net, RANGEQUERY | src, id, range, limit, consistent, collector_type>: Starts a range-query operation by consulting other CATS servers.

Indication: <net, RANGEQUERY.RESP | client, id, range, limit, items, result>: Delivers the range-query response items, and result status to the client.

Indication: <net, READRANGE.RESP | collector, id, seq, range, limit, items, result, is_last, more>: Deliver a chunk of the range-query response.

Properties:

CRQ1: Validity: The operation delivers a sorted list of size 0 to limit, containing consecutive data items from the beginning of the requested range. If consistent is True, it provides strong data consistency with respect to linearizability, per item.

CRQ2: Ignore Nulls: The operation only returns data items with non-null values.

CRQ3: Termination: If a correct process invokes a range-query operation on a cluster of correct CATS servers, the operation eventually completes.

CRQ4: Best Effort: CATS tries to deliver requested number of items (limit). CATS might deliver less items to satisfy the Termination property.

3.2 Range-Query Operation

The user may send range-query requests to any of the CATS nodes. The node that receives the request may decide to forward it to another node based on some policies. We will discuss policies later in this chapter. For now, we consider a basic policy: the range-query request is forwarded to the node that is the first replica for the beginning part of the range. To make it more clear, consider a group of seven CATS nodes, with ids a, b, c, d, e, f, and g (Figure 3.1). A client sends a range-query request for the range (c,g] to the node a. Node a forwards the request to the node d, which is the first replica for the beginning part of the range, i.e., the keys greater than c. Therefore, node a forwards the request to the node d.

Finally, the main replica (node d, in our example) receives and handles the range-query request. The requested range may be larger than the range stored in the node handling the request. Therefore, the node may need to contact neighbor servers for the sub-range that it is not responsible for. Read-range request is an internal request to query a sub-range from a node. It carries information about the range, limit and id of the range-query request. It also contains a read-range sequence number and the address of the range-query collector. The collector is responsible
Figure 3.1. Forwarding the range-query request to a responsible node

for collecting and aggregating read-range results for different sub-ranges. The node that handles the range-query request sends a read-range message with sequence number 0 to itself, querying the whole range. The node that receives a read-range request, runs a consistent range-query for the part of the range that is in the scope of its responsibility. It consults the other replicas and sends the response to the collector in a read-range.resp message. The response contains information about the range-query id, the sequence number of read-range and the operation result. The operation result can be one of the following:

- **SUCCESS**: When the read-range succeeds
- **INTERRUPTED**: When the read-range succeeds partially
- **TIMEOUT**: When the node cannot complete the operation on time

We will discuss read-range operation results in the next section. If the read-range operation results in SUCCESS, but the node could not provide enough items while not covering the whole requested range, the node will send a new read-range request with an incremented sequence number to the next CATS node. The request queries the remaining sub-range for the remaining number of items. The node decides not to send a new read-range request in the following situations:

- The read-range operation results in SUCCESS and provides enough items or covers the whole range
- The read-range operation results in INTERRUPTED
- The read-range operation results in TIMEOUT
If the CATS node does not send a new read-range request, it informs the collector by tagging the response as the last response. For every range-query id, the collector looks forward to receiving read-range responses with that id. When the collector receives a read-range response tagged as the last response, it waits for responses with lower sequence numbers (Figure 3.2, numbers show sequence of events).

![Figure 3.2. read-range requests and corresponding responses for sub-ranges](image)

After receiving the last read-range response and all read-range responses with lower sequence numbers, the collector aggregates results and delivers the range-query response to the user. Range-query operation results are similar to read-range operation results. The result can be one of the following:

- **SUCCESS**: When the range-query succeeds
- **INTERRUPTED**: When the range-query succeeds partially
- **TIMEOUT**: When CATS cannot complete the operation on time

The result of the range-query operation will be **INTERRUPTED**, if the last read-range operation results in **INTERRUPTED**.

Algorithms of the range-query operation are available in the Appendix B.

**Range-Query Timeout**

Timeouts may happen because of failures or because it takes more than a specified time to finish the range-query operation. A default value for operation timeout is provided in the system settings. If the range-query operation does not complete
on time, range-query returns a `range-query.resp` message with TIMEOUT as the operation result. As an option, the client can select timeout value for each range-query operation. The `range-query` message takes three optional parameters for operation timeout:

- **Base-rto**: Base value for range-query operation timeout in milliseconds
- **Retries**: Number of retries after timeout
- **Scale-ratio**: Scale ratio for timeout value

`Scale-ratio` must be a real number larger than 1.0. If the user provides timeout parameters and the range-query operation does not complete on time, the handling node scales up the timeout value and retries. If all retries fail, the user receives a `range-query.resp` message with `TIMEOUT` result. If a range-query operation uses default timeout value, there will be no retry on the timeout.

For simplicity, we did not consider retries on range-query timeouts in the consistent range-query algorithm presented in the Appendix [3].

### An Example Range-Query Operation

We go through an example to see how the whole process works. Consider a group of seven CATS nodes with ids `a, b, c, d, e, f, and g` (Figure 3.3). Each node stores 100 items in the range for which it is the first replica. For example, node `b` is the first replica for the range `(a,b]`. It stores 100 items in this range. A user (i.e., client) sends the `range-query (c, g] #200` request to the node `a`. Node `a` forwards the request to the node that is the first replica for the beginning part of the range. If the range is start-inclusive, the node will be the first replica of the start key. If the range is start-exclusive, the node will be the first replica of the keys that are greater than the start key. In this example, the range is start-exclusive. Node `d` is the first replica for the keys greater than `c`.

Node `a` forwards the range-query request to node `d`. Node `d` handles the forwarded range-query request. It sends a `read-range (c, g] #200` request to itself. Node `d` tries to collect up to 200 items in its range, i.e., in the range `(c,d]`. We know that every node stores 100 items in its range. Node `d` executes the read-range operation, consults the other replicas and sends `read-range.resp (c,d] #100` to the collector. The range-query request asked for 200 items, but node `d` could provide 100 items. It sends a new read-range request to the next CATS node, querying the remaining sub-range with a new limit. If the limit of the current read-range is `l` and the node collects `n` items in the read-range process, the new limit will be `l - n`. Node `d` sends `read-range (d, g] #100` to node `e`. Node `e` executes read-range operation, consults the other replicas and sends `read-range.resp (d, e] #100` to the collector. Although node `e` does not cover the whole requested range, but it provides enough items. So, it tags the read-range response as the last response. It does not send a new read-range request to the next node. If everything
3.3. READ-RANGE OPERATION

In this section we explain read-range operation in more details. This is where we provide solutions for two problems we mentioned earlier, i.e., to find first \( l \) keys with non-null values according to the view of the majority, and to return consistent values for those keys. A node that receives a read-range message, handles the request if it is responsible for a sub-range in the beginning of the requested range. A responsible node that handles a read-range request with limit \( l \), must collect up to \( l \) sequential items in the sub-range of the request that is in the scope of its responsibility. It sends two types of requests concurrently (Figure 3.4, \( rr \) stands for read-range):

- read-range-localitems
- read-range-stat

The responsible node sends a read-range-localitems request and handles the request itself. The goal is to read first \( l \) items in the sub-range, from its local data store. It only considers keys with non-null values. Each item includes a key-value pair and a timestamp. The node delivers the list to itself in a read-range-localitems.resp message.

The node also contacts all the other replicas to obtain information about first \( l \) items they store in the sub-range. The node contacts all the replicas by sending
read-range-stat requests, but it waits only for the majority to respond. A node that receives a read-range-stat request, prepares a list of first \( l \) items it stores in the requested range. The list contains keys and corresponding timestamps, without any value. The node responds to the responsible replica with a read-range-stat.resp message containing this list. The responsible node waits for a read-range-localitems.resp message from itself and some read-range-stat.resp messages from the majority - 1 of the other replicas.

![Diagram of read-range operation in detail](image)

**Figure 3.4.** read-range operation in detail

In the example illustrated in Figure 3.4, the replication factor is 3. Node \( d \) sends a read-range \((d,g]\ #100\) message (event 1) to node \( e \) to query first 100 items in the range \((d,g]\). Node \( e \) handles the sub-range that is inside its own range, i.e., \((d,e]\). It sends two type of messages concurrently: A read-range-localitems \((d,e]\ #100\) message (event 2) to itself, and a read-range-stat \((d,e]\ #100\) message (events 3 and 4) to every other replica of this range (i.e., nodes \( f \) and \( g \)). Right after that, it delivers the read-range-localitems.resp \((d,e]\ #100\) message (event 5) to itself, containing first 100 items stored locally for the range \((d,e]\).

It waits for the majority - 1 nodes to respond to read-range-stat messages. With the replication factor of 3, it waits for one response and proceeds to the next step as soon as it receives a read-range-stat.resp message from node \( f \) or node \( g \). In this example, it receives the read-range-stat.resp message (event 6) sent by node \( g \) and proceeds to the next step. Later, it ignores the read-range-stat.resp message (event 7) received from node \( f \). In the next step, node \( e \) prepares the final read-range result and sends read-range.resp \((d,g]\ #100\) message (event 8) to the collector.

Algorithms of the read-range operation are available in the Appendix B.
Intermediate Results Aggregation

To prepare the list of first \( l \) items in its range, the responsible node compares the local list with the lists provided by the other replicas. The final list will be based on the data stored on the majority of the replicas. If replicas are executing ABD operations that put new keys or keys with null values, the responsible node may receive different sets of keys. In such a case, key sets depend on the data stored on the nodes that were in the majority. However, our solution provides strong consistency for the value of every key it returns.

To make it more clear, we discuss two scenarios illustrated in Figures 3.5 and 3.6. In these figures node \( e \) is a responsible node that handles a read-range message. Suppose that two keys are currently stored in the range \( (d,e] \). A client contacts node \( g \) to put a new key that is in this range. This is concurrent with a read-range operation. At this point, node \( g \) have stored this new key-value pair. In scenario 1 (Figure 3.5), the responsible node delivers a list of two keys to itself, in response to \textit{read-range-localitems}. It also receives a list from node \( f \), in response to \textit{read-range-stat}. The list contains same items with same time stamps. It ignores the response sent by node \( g \). Nodes \( e \) and \( f \) (the majority) are not aware of the new key that is being inserted. The final list returned in a \textit{read-range.resp} message contains those two keys.

![Figure 3.5](image)

Figure 3.5. read-ranges returning different set of keys in the presence of concurrent put operations (Scenario 1)

Now consider scenario 2, illustrated in Figure 3.6. The responsible node delivers a list of two keys to itself in response to the \textit{read-range-localitems} message. But it receives a list from node \( g \), in response to the \textit{read-range-stat}. The list contains
three keys, including the key that is currently being inserted. Node e merges lists and prepares a final list that contains all three keys. It does not have the new key in its local data store and obtains the latest value for the new key using the ABD component of CATS.

![Diagram](image)

**Figure 3.6.** read-ranges returning different set of keys in the presence of concurrent put operations (Scenario 2).

Our solution reuses the ABD component already available in CATS, to provide consistent values. As illustrated in the Figure 3.6, it does not execute ABD algorithm for all key-value pairs. To provide a lightweight mechanism for range-query, it tries to limit the use of the ABD algorithm. Timestamps are used to determine the consistency of key-value pairs, by comparing the list provided locally with the lists obtained from the other replicas. ABD algorithm is executed for two series of keys:

- New keys that are not yet replicated in the responsible node
- Keys for which the responsible node does not have the latest value

The responsible node, executes ABD for keys mentioned above. Upon receiving ABD response for all of them, it updates the list with obtained values and sends a `read-range.resp` message to the collector. The message contains range-query operation id, the result of the read-range operation, the range of the responsible node, possibly a list of key-value pairs, the read-range sequence number and a flag that
determines if it is the last read-range response message. The read-range operation result can be one of the following:

- **SUCCESS**: When the read-range succeeds
- **INTERRUPTED**: When the read-range succeeds partially
- **TIMEOUT**: When the node cannot complete the operation on time

**SUCCESS** means that the operation has completed successfully. If the number of returned items is less than the requested limit, it means that more item were not available in the range. **INTERRUPTED** may happen in a very dynamic systems. It may happen when a request puts a new value for a key, and concurrently, another request puts a null value for the same key. **INTERRUPTED** means that the read-range operation succeeded to return a sequence of key-value pairs from the beginning of the range, but the node failed to return all available keys in its range. User cannot be sure if the range-query operation has returned all available items or there are more items at the end of the range. In such cases, the node does not send a new read-range request to the next node. It tags the response as the last response. The collector checks the operation result in read-range responses. If the result of the last response is **INTERRUPTED**, the collector set the result of the range-query to **INTERRUPTED** in the `range-query.resp` message. The client that receives the response will know that more items may be available in the range. It must start another range-query with a range starting from the last key (exclusive) in the result (Figure 3.7). If the read-range operation does not complete in a predefined time, the collector receives a `read-range.resp` response message with a **TIMEOUT** result.

![Diagram](image-url)

**Figure 3.7.** A range-query operation resulting in **INTERRUPTED**. The client starts a new range-query operation for the remaining sub-range.
Figure 3.8 illustrates an example scenario for the read-range operation in Figure 3.7 that returned a read-range.resp with an INTERRUPTED result (event 6). Node e in Figure 3.7 handles a read-range (d, g] #2 request (event 5). As seen in Figure 3.8, node e stores four keys in the range (d,e]. It must run read-range operation to return consistent values for first two keys. Assume that the system is highly dynamic and clients are updating values of same keys concurrently. Node e reads two keys locally with read-range-localitems (d,e] #2 request and response messages (events 1 and 4). It also contacts nodes f and g (events 2 and 3) and obtains same set of keys from node f (event 5). The set of keys and the corresponding timestamps are depicted in the figure.

Node e has a version of the key 'd200' with the timestamp '5'. It knows that node f has a newer version of the key 'd200' with the timestamp '6'. Therefore, it starts an ABD operation (event 6) for the key 'd200'. Meanwhile, a client sends a Put(d200, null) request and the value of the key 'd200' updates to null with a new timestamp. Node e receives ABD.resp(d200, null) in response to its request (event 7). As we mentioned earlier, range-query does not return keys with null values. Therefore, node e ignores the key 'd200' that now has a null value. The read-range request queried for 2 items in the range. Although node e stores more than two key-value pairs, it returns only one. Node e does not try to provide more items. It responds with one item and sets the result of the read-range operation to INTERRUPTED (event 8).
3.4 The Collector

Collector has two responsibilities:

- Collect read-range results. Prepare and send range-query result to the client
- Manage range-query operation timeouts

We propose two types of collectors: **Server Collector** and **Client Collector**. In Server Collector mode, the CATS server (i.e. node) that handles the range-query request, sends the first read-range request message and mentions itself as the collector. This server is mentioned as the collector also in next read-range requests. The Server Collector collects all read-range.resp messages. It merges all results, prepares the range-query response message and sends it to the client. In this model, the part of the data that is not in the range of the collector, must be transmitted over the network twice. For example, in Figure 3.9, node e sends the read-range result, read-range.resp (d,e] #100, to node d (event 6) over the network. Later when the collector sends the range-query result to the client, the part of the data that is provided by node e is transmitted one more time. This overhead on the network has two cost: bandwidth usage and latency.

![Figure 3.9. Server Collector](image)

In Client Collector mode, the client that sends the range-query request is responsible for collecting read-range responses. The Client Collector is aware of the range-queries initiated by the client. The server (i.e. node) that handles the range-query request, mentions the address of the client as the address of the collector. Nodes send read-range responses directly to the client. The client merges all the responses, prepares the range-query response message and delivers it locally. In this model, the data is transmitted only once. For example, in Figure 3.10, nodes d and
send read-range results directly to the client. The collector in the client delivers the merged result locally.

We will evaluate and compare Server and Client Collectors in the evaluation chapter. As we will see, there is a big difference in the latency when the size of the range-query result is very large.

![Diagram of Client Collector](image)

**Figure 3.10.** Client Collector

The collector is also responsible for managing range-query timeouts. In Server Collector mode, a CATS node that handles a range-query request, sets a timer for that operation. In Client Collector mode, the collector module of the client is responsible for setting a timer for every range-query operation that the client initiates. Therefore, all range-query requests must pass through this module before being transmitted.

We have provided an interface client for CATS, which manages the responsibilities on the client side. Module 3.2 shows the interface and properties of the CATS Interface Client (CIC). Algorithms of the CATS Interface Client (CIC) are presented in the Appendix C, algorithms C.1 and C.2.

### 3.5 Policies

We have defined different policies for the range-query, to make it more flexible for different types of applications. We discuss different policies and their advantages and disadvantages.
Module 3.2 Interface and properties of CATS Interface Client

Module:

Name: CATS-Interface-Client, instance cic.

Events:

Request: \( <\text{cic}, \text{CIRANGEQUERY} | \text{req_id}, \text{range}, \text{limit}, \text{consistent}, \text{collector_type}> \):
Invokes a range-query operation for \( \text{limit} \) items in \( \text{range} \) on a CATS server. If \( \text{consistent} \) is True, asks for strong consistency.

Indication: \( <\text{cic}, \text{CIRANGEQUERY.RESP} | \text{req_id}, \text{range}, \text{limit}, \text{items}, \text{result}> \): Delivers range-query response \( \text{items} \), and success status \( \text{result} \), provided by CATS.

Properties:

CIC2: No Creation: The operation only delivers items returned by the CATS servers.

CIC1: Termination: Eventually, the operation delivers a result and completes.

Policies for Range-Query Forwarding

The user may send a range-query request to any of the CATS nodes. The CATS node forwards the request to a responsible replica. A responsible replica is one of the replicas of the beginning part of the range. We have defined the following policies for forwarding range-query requests:

- Always First: Forwards requests to the first responsible replica
- Random: Forwards requests to a random responsible replica. This policy provides an automatic load-balancing by distributing requests across multiple responsible replicas.
- Check & Forward: Checks which replicas are alive and ready, using a Ping-Pong mechanism. Sends a Ready message to all replicas and forward the request to the first replica that responds. This policy helps to tolerate node failures. It is assumed that the majority is always alive. If a replica crashes, other replicas will respond. There will be no timeout because of forwarding a request to a crashed replica. If a node is overloaded, it responds with an intentional delay. If other replicas are not overloaded they respond earlier, and one of them will be selected. However, if all nodes are overloaded, the intentional latency is added to the range-query time. There is always at least a round trip delay before forwarding a request. Range-queries are usually time consuming operations, so this very short extra latency may be ignored. Nevertheless, this policy introduces extra overhead to the system. It sends a series of messages for every range-query request forwarding. To avoid this
extra overhead, the user can explicitly ask for this policy only after receiving a range-query `TIMEOUT`.

**Figure 3.11.** Check & Forward policy for range-query request forwarding. Replication factor is 3. Node d is overloaded. Node e has crashed.

Figure 3.11 illustrates an example range-query request forwarding scenario that uses Check & Forward policy. In this scenario, the replication factor is 3, node d is overloaded and node e has crashed. Node a receives a range-query request for the range `(c,g]`. It checks if the replicas of the beginning part of the range (i.e., nodes d, e and f) are alive and ready. It sends a "Ready?" message to all the three replicas (event 2). Node d responds with a "Ready!" message (event 5) with an intentional delay, because it is overloaded. Node e has crashed. So, it does not respond. Node f immediately responds with a "Ready!" message (event 3). Node a receives node f’s response first, and forwards the request to this node.

**Enhanced Local Read**

This policy is beneficial when using a Server Collector. It enhances local reads and reduces data transmission over network. We know that each replica stores some consecutive ranges, including its own range and replicated ranges. It is possible to forward a range-query request to a node that has several sub-ranges of the requested range. The node sends read-range requests for those sub-ranges, to itself. In some applications that query short ranges, the node can read the whole range locally. In applications that query large ranges, this policy can still save some network traffic and decrease the latency. This policy can be mixed with any of the range-query forwarding policies.
Figure 3.12 illustrate an example scenario. Node a is using Check & Forward policy together with Enhanced Local Read policy. Nodes e and f stores both (c,d] and (d,e] sub-ranges. Node a checks the status of nodes e and f, and forwards the request to node e. Node e sends two read-range messages to itself to read sub-ranges (c,d] and (d,e] locally. Later, it sends a read-range message to node f to query about the range (e,f].

Figure 3.12. Enhanced Local Read policy for range-query request forwarding. The replication factor is 3. Nodes e and f stores both (c,d] and (d,e] sub-ranges.

Policies for Read-Range Target Selection

A node that handles a read-range request, sends a new read-range request if it cannot provide enough items locally and the requested range in larger than its local range. It should send the new read-range request to one of the replicas of the beginning part of the remaining sub-range. We suggest the following three policies that are similar to range-query request forwarding policies:

- Always First
- Random
- Check & Send
3.6 Tolerating Node Failures

If a CATS node crashes, members of its replication groups will be informed after a short time. They will form a new replication groups using consensus. This is provided by CATS. Before the failure of a crashed node is detected, some range-query requests may be forwarded to the crashed node. Also a node may send a read-range request to its next node that is not alive anymore. In such cases, the user will receive a response with \textit{TIMEOUT} result. The user must retry and retries may result in range-query \textit{TIMEOUT} for few seconds.

We have proposed a simple mechanisms to avoid timeouts in the case of node failures. Using \textit{Check \& Forward} and \textit{Check \& Send} policies, CATS can execute range-queries in the presence of node failures. This policy solves the problem in most cases. One exception is when a node crashes after sending a \textit{"Ready!"} message in response to a \textit{"Ready?"} message, that is a very rare situation. Another exception is when a replication group changes and there are some read-range operations in progress. The change happens because of the failure of a node. A mechanism inherited from CATS guarantees strong consistency in the presence of changes in group memberships. Nodes include their group membership information when they send data in response to read-range messages. Nodes ignore read-range response messages that contain old group membership information.

\textit{Check \& Forward} and \textit{Check \& Send} policies introduces extra overhead to the system. To avoid this extra overhead, user can explicitly ask for this policy only after receiving a range-query \textit{TIMEOUT}. On the other hand, node failures does not happen very often in a data center environment. Therefore, there is a trade-off: we can either avoid most timeouts caused by node failures and accept the overhead, or, we can accept receiving timeouts in the case of node failures and instead have a lighter system.
Chapter 4

Implementation

4.1 Consistent Range-Query (CRQ) Component

CATS is implemented in Java using the Kompics framework. Kompics can be used to implement a complex distributed system in several connected components. Components communicate with each other through message passing. We have implemented a Consistent Range-Query (CRQ) component for the CATS NoSQL database in Java using Kompics. In order to make this component plugable to CATS, we have implemented new Kompics Events (i.e., message) and Port types. We have also modified some of the existing Port types and components in CATS. The CRQ component relies on some services provided by the other CATS components. It provides consistent range-query service through its ports. Figure 4.1 shows the components of a CATS node. The figure illustrates ports and channels that are related to the CRQ component.

The current version of CATS uses Grizzly network framework for Java. Grizzly facilitates building scalable and robust servers using Java New I/O (NIO). The CRQ component uses the network service through the Network component of CATS. Some of the requests and responses in CRQ are sent and received using this port. Figure 4.2 depicts messages that the CRQ component of CATS sends and receives over the network. These messages were discussed in the previous chapter. We briefly go through them:

- **RangeQuery.Req**: range-query request sent by a client
- **RangeQuery.Resp**: range-query response sent by a Server Collector
- **ReadRange.Req**: read-range request, sent to query a sub-range
- **ReadRange.Resp**: read-range response, sent to the Collector
- **ReadRangeStat.Req**: Sent by a responsible node to obtain keys and timestamps from the other replicas
Figure 4.1. A CATS node with a Consistent Range-Query (CRQ) component.

Figure 4.2. Network port and Events related to the range-query component

- **ReadRangeStat.Resp**: Sent by replicas to the responsible node, containing information about the keys and corresponding timestamps in the range
- **Ready.Req**: Used in some policies to check if the replicas are alive and ready
- **Ready.Resp**: Used to inform that the node is ready and alive

The CRQ component relies on the CATS lookup service to find the responsible nodes and successors of a key. It reads data using the CATS component in two ways:

1. To query a sub-range locally it uses *LocalRQ* port with messages *ReadRangeLocalItems.Req* and *ReadRangeLocalItem.Resp* (Figure 4.3).
2. To obtain information about a sub-range from the other replicas, it sends ReadRangeStat.Req messages over the network (Figure 4.2). The other replicas will reply with ReadRangeStat.Resp messages.

As we discussed in the previous chapter, the CRQ component may execute ABD operations to obtain the latest values for some of the keys. For each ABD operation, it sends an ABDGet.Req message to the ABD component through its LocalABD port. The ABD component executes an ABD Get operation and sends the result back in an ABDGet.Resp message using the same port (Figure 4.4). The CRQ component also provides range-query service to the CATS web application.

4.2 CATS Interface Client (CIC)

CATS Interface Client (CIC) is a client for CATS that plays the role of an interface between the CATS NoSQL database and a client. It is a Kompics component that provides simple access to CATS. It requires a Timer component and the Network component of CATS. It also provides the functionality of the Client Collector. So, the end client does not need to deal with the aggregation of range-query responses (read – range.resp messages) directly. In the current version, the address of a CATS node can be provided during the initialization, or with every operation. In some of the experiments for the evaluation, we used the CATS Interface Client with YCSB to access the CATS servers. Figure 4.5 depicts the CATS Interface Client component.

CATS Interface Client (CIC) provides simple access to CATS through CatsInterface port. The following Events are allowed on this port (Figure 4.6):

CIPut: To put a value to CATS. CIPut request takes three parameters and CIPut response contains three state variables.
CIPut.Req\(\text{long}\ \text{requestId}, \ String\ \text{key}, \ String\ \text{value}\)
CIPut.Resp\(\text{long}\ \text{requestId}, \ String\ \text{key}, \ \text{OpResult}\ \text{result}\)

**CIGet**: To get a value from CATS. CIGet request takes two parameters and CIGet response contains four state variables.

\[
\text{CIGet.Req}(\text{long}\ \text{requestId}, \ String\ \text{key})
\]
\[
\text{CIGet.Resp}(\text{long}\ \text{requestId}, \ String\ \text{key}, \ String\ \text{value}, \ \text{OpResult}\ \text{result})
\]

**CIRangeQuery**: To query a range. CIRangeQuery request takes four parameters and CIRangeQuery response provides the given parameters together with the range-query result and a sorted list of key-value pairs.

\[
\text{CIRangeQuery.Req}(\text{longId}, \ boolean\ \text{consistent}, \ \text{Range}\ \text{range}, \ int\ \text{limit})
\]
\[
\text{CIRangeQuery.Resp}(\text{longId}, \ boolean\ \text{consistent}, \ \text{Range}\ \text{range}, \ int\ \text{limit}, \ \text{LinkedList<KeyValue}>\ \text{items}, \ \text{OpResult}\ \text{result})
\]

**CISelectCollector**: It is possible to change the type of Collector using this message. It takes one parameter of the type Collector, which, can be Collector.CLIENT or Collector.SERVER.

\[
\text{CISelectCollector}(\text{Collector}\ \text{collector})
\]

Algorithms [C.1] and [C.2] in the Appendix [C] are related to this component.
4.3 Basic Client

We have developed a Basic Client for CATS in Kompics. It uses a CATS Interface Client component (Figure 4.7). Basic Client takes a set of commands as initial parameters or through system input. It accepts the following commands:

- **c**
  - Select Collector. *c* can be *Client* or *Server*

- **P a=A**
  - Put the key "a" with the value 'A'

- **D a**
  - Delete the key 'a' (i.e., put the key 'a' with the value *null*)

- **G a**
  - Get the value of the key 'a'

- **RQ[a,b]#l**
  - Executes a range-query for first *l* items in the range [a,b]

- **CRQ[a,b]#l**
  - Executes a consistent range-query for first *l* items in the range [a,b]

- **S t**
  - Sleeps for *t* seconds

- **X**
  - Shuts down

A series of commands separated by colons can be provided as initial parameters. For example, with the following set of commands, the client puts two key-value pairs, sleeps for 1000 ms and then executes a range-query:

```
Pn100.fname=John : Pn100.lname=Smith : S1000 : CRQ[n100,n101)#1000
```

We have used a Basic Client in one of the evaluation experiments. It is discussed in the next chapter.

4.4 Experiments

We have modified the CATS experiment Simulator. Node join and failure scenarios were already supported by CATS simulator. Our implementation supports basic
clients in scenarios. A client may join and execute some commands at a certain
time. Scenarios can be provided in a scenario script file or as input parameters.

4.4.1 Scenario Scripts

It is possible to define a scenario for an experiment in a scenario script file. A
sample script is shown in Figure 4.8.

Experiments may be executed in three different modes:

- **sim**: Simulation mode
- **exe**: Execution mode. Nodes are accessible through their web interface
- **test**: Test mode. Range-query results are examined against expected results. Expected results are provided by the Scenario Evaluator.

Scenarios are defined as a series of processes. Each process is assigned to a
variable (e.g., P1) and passed to the experiment simulator. A process starts with
@ sign followed by a number showing the start time of the process. Then the type
of the process is specified. A process may also take some parameters. Process type
can be one of the followings:

- **B**: Bootstrap process, bootstraps initial nodes.
- **J**: Join process, adds a new node after bootstrap at a certain time
- **F**: Fail process, fails a node at a certain time
- **C**: Client process, starts a client at a specified time
- **T**: Terminates the experiment at a specified time
4.5. Test

We used Test Clients, experiments and scenario evaluator class to test various scenarios with JUnit. We also used Cobertura for code coverage test. Cobertura is a tool for Java code coverage analysis.

4.5.1 Scenario Evaluator

We have implemented a scenario evaluator class. It interprets a given scenario and determines the expected result for a range-query operation requested by a client at a certain time. It considers previous put operations and node joins and failures. An expected range-query result is the result that CATS returns, if it works correctly. When not using a policy that prevents timeouts caused by node failures, the scenario evaluator can determine if CATS responds with a \textit{TIMEOUT} result. Scenario evaluator considers put operations to happen immediately. Therefore, a small safety border must be considered for the time between puts and range-queries. Scenario evaluator is used for running experiments in test mode and in JUnit tests.

4.5.2 Test Client

We have implemented a Test Client by extending the Basic Client. It takes commands as an initialization parameter and uses the scenario evaluator to determine the expected result for each range-query operation. We have used the Test Client to run JUnit tests. It is also used for running experiment scenarios in the test mode.

4.6 YCSB

CATS has a DB client for YCSB. We have modified this DB client to support range-queries. We used the CATS Interface Client component, which we have developed for simple access to CATS. Figure 4.9 illustrates a simplified model of the CATS client for YCSB.
We have developed a new YCSB workload class for our experiments. The new workload class takes the following extra parameters, which provides more control over the experiments:

- **collector**: To select range-query collector: server/client (default: server)
- **keylb**: The lower bound of the keys (default: 0)
- **keyub**: The upper bound of the keys
- **minscanlength**: The minimum number of records to be scanned (default: 1)

We used shell scripts to automatically run YCSB with different parameters in different rounds.
Chapter 5

Evaluation

We have evaluated our consistent range-query solution for the CATS NoSQL database from different perspectives. In this chapter we present the results of several experiments. In these experiments, we used a cluster of seven machines at the Swedish Institute of Computer Science (SICS) connected via a Gigabit Ethernet LAN. All machines had the following hardware settings:

- CPU: 2 × Six-Core AMD Opteron™ 2435 (2.6GHz) Processors
- RAM: 32 GB
- Network Interface: Gigabit Ethernet

In the first five experiments, we have evaluated the performance of our range-query solution using the Yahoo! Cloud Serving Benchmark (YCSB) framework. We set up a cluster of seven CATS nodes with a replication factor of 5, each running on one machine.

Figure 5.1 illustrates CATS nodes used in the first experiments, with ids 0999, 1999, 2999, 3999, 4999, 5999 and 6999. We used the YCSB benchmark to load CATS nodes with key-value pairs using keys of the type String from 0000 to 6999. We loaded each CATS node with 1000 key-value pairs. In Experiment 5, we used a cluster of seven CATS nodes with ids 09999 to 69999 and loaded each of them with 10000 key-value pairs.

We used different value (i.e. data) sizes in different experiments. In CATS, values are of the type String. By default, Java stores each character of a string in two bytes. Thus, size of a value stored in CATS is 2 bytes per character. We have also considered $10^3$ bytes to be equal to 1 kilobytes (kB). Sizes of keys and values in the following experiments are based on these considerations.

In Experiment 6, we used the CATS simulator with experiment scripts running in the test mode. Test mode exploits scenario evaluator class to determine the expected result for a given range-query operation, at a given time, in a given scenario.

In Experiment 7, we applied three instances of the Basic Client and a cluster of seven CATS nodes running on SICS machines.
5.1 Experiment 1

In this experiment, we have investigated the average latency, when range-queries cross range boundaries of CATS nodes. We compared client and Server Collectors and studied how the Enhanced Local Read policy affects the latency of a range-query operation. We used the following settings in this experiment:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key-value pairs</td>
<td>7000</td>
</tr>
<tr>
<td>Key size</td>
<td>8 Bytes</td>
</tr>
<tr>
<td>Value size</td>
<td>100 kB</td>
</tr>
<tr>
<td>Threads</td>
<td>1</td>
</tr>
<tr>
<td>Operations</td>
<td>10</td>
</tr>
<tr>
<td>Range-query portion</td>
<td>1.0</td>
</tr>
<tr>
<td>Update portion</td>
<td>0</td>
</tr>
<tr>
<td>Limit</td>
<td>1000</td>
</tr>
<tr>
<td>Start key</td>
<td>0000 to 3000</td>
</tr>
<tr>
<td>Collector</td>
<td>Client / Server</td>
</tr>
</tbody>
</table>

We loaded CATS with 7000 key-value pairs. Each key-value pair had a key size of 8 Bytes and a value size of 100 kB. We executed experiments in three steps. We used client and Server Collectors in steps one and two respectively. In step three, we used Server Collector together with Enhanced Local Read policy. In each step, we executed experiments in several rounds. In the first round we queried 1000 items starting from the key "0000". In the next rounds, we increased the value of the start key and queried 1000 items. We changed the start key from "0000" to "3000". There was no concurrent update while running range-query operations.

Figure 5.2 is plotted based on the results of all three steps of this experiment. It illustrates the average latencies of range-query operations (i.e., range-query time) per start key in each step. A periodic pattern is observable in the graph.

As seen in Figure 5.2, the Client Collector performs much better than a normal Server Collector. It is also seen that the Enhanced Local Read policy can decrease
5.1. EXPERIMENT 1

Figure 5.2. Experiment 1. Average latencies of range-queries (i.e., range-query time) per start key with different collectors and policies. Server collector 2 applies the Enhanced Local Read policy.

the range-query time when using the Server Collector. If a requested range is in the range of two servers (e.g., range [0500,1500) is in the range of servers "0999" and "1999"), it may affect the latency. The effect depends on the type of the collector and the chosen policy.

Consider a Client Collector is used to query the range [0500,1500), which is in the range of two servers "0999" and "1999". As shown in Figure 5.3, each server sends the items in its sub-range directly to the client. One server responds with [500,1000)#500 and another server responds with [1000, 1500)#500. In such cases, that the range-query crosses the boundaries of servers, using a Client Collector has a desirable effect on the range-query time.

Figure 5.3. Using Client Collector for range-query [0500,0500)#1000. One server responds with [0500,1000)#500 and another server responds with [1000, 1500)#500.
As illustrated in Figure 5.4, when using a Server Collector, data is transferred to a collector server, before being transferred to the client. According to this experiment, using Server Collector can increase range-query time up to 100\%, compared to a Client Collector (Figure 5.2).

![Figure 5.4](image)

**Figure 5.4.** Using Server Collector for range-query [0500,0500)\#,1000. Server "0999" receives 500 items in the range [1000,1500) from server "1999", aggregates and sends 1000 items of the whole range to the client.

Using *Enhanced Local Read* policy gives us a constant latency, if the requested range is not larger than the range of \( REPLICATION\_FACTOR/2 + 1 \) neighbor nodes.

![Figure 5.5](image)

**Figure 5.5.** Using Server Collector with *Enhanced Local Read* policy for range-query [0500,0500)\#,1000. Server "1999" replicates sub-ranges [0500,1000) and [1000,1500). It sends all 1000 items to the client.

According to the results of this experiment, Client Collector has the best performance. It has lower latency and consumes less network bandwidth. We suggest to use Client Collector.

### 5.2 Experiment 2

In this experiment we investigated the effect of concurrency in range-query operations. We used the following settings in this experiment:
Initially, we inserted 7000 key-value pairs with a value size of 2 kB into CATS. We used Client Collector on YCSB client for CATS and increased number of YCSB threads from 50 to 400 in rounds. In each round, YCSB threads execute 2000 range-query operations. More threads means more concurrent operations. Each operation queries 1000 items in a range with a random start. For examples, \([0500,0500)\) #1000 refers to the first 1000 items, starting from the key '0500'.

Figure 5.6 shows the average, 95th percentile and max latencies of range-query operations per number of threads. An almost linear increase is seen in the average and the 95th percentile latencies.

![Figure 5.6](image-url)

Figure 5.6. Experiment 2. Average, 95th percentile and max latencies of range-query operations per number of threads.

5.3 Experiment 3

In this experiment, we studied how range-query limit (i.e., scan length) affects range-query time (i.e., latency). We used the following settings in this experiment:
Initially, we loaded the CATS with 7000 key-value pairs with a value size of 2 kB. We increased range-query limit from 1 to 7000 in rounds. Limit of 7000 means reading all key-value pairs stored in the CATS. In each round we used 20 YCSB threads to execute 10,000 operations. 10% of operations were range-query and 90% of operations were update operations. Start keys of the range-query operations were chosen randomly by YCSB.

Figure 5.7 illustrates the range-query latencies per query limit. An almost linear increase is observed in the latency as the limit increases.

Figure 5.7. Experiment 3. Average, 95th percentile and max latencies of range-query operations per query limit. Start key of the range is random.

### 5.4 Experiment 4

This experiment is similar to the previous one. We studied how range-query limit affects range-query time. It is different from the previous experiment in that we uses a fixed start key (i.e., the key "0000") for range-queries. All range-query operations query certain number of items starting from the beginning of the data space. We used the following settings in this experiment:
Initially, we loaded the CATS with 7000 key-value pairs with a value size of 2 kB. We increased range-query limit from 1 to 7000 in rounds. Limit of 7000 means reading all key-value pairs stored in the CATS. In each round we used 20 YCSB threads to execute 10,000 operations. 10% of the operations were range-queries and 90% of operations were update operations. All range-query operations started from the key '0000'.

Figure 5.8 depicts the result, which is similar to the result of the previous experiment. However, the average latency and the 95th percentile are slightly more than the previous experiment.

In the previous experiment, we used random start keys for range-queries. This increases the chance of the request to be handled by more servers. As we saw in experiment 1, when using Client Collector, a range-query request completes faster if it is in the range of two servers. Same thing happens for larger ranges. For example, if a range-query operation requesting 3000 items is handled by three servers in this experiments. But in the previous experiment, it is likely that the range-query is handled by four servers. Therefore, most of the time, range-query latencies are
lower in the previous experiment. According to this graph, crossing boundaries of several servers does not have an undesirable effect on the range-query time.

5.5 Experiment 5

In this experiment, we loaded CATS servers with a total of 70,000 key-value pairs, which, means 10,000 items per server. We investigated how number of items affects the performance, for the same size of data. In Experiment 3, we loaded CATS servers with 7000 items of the size 2 kB. In this experiment, we loaded CATS servers with 70,000 items of the size 200 B. The following settings are used in this experiment:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key-value pairs</td>
<td>70,000</td>
</tr>
<tr>
<td>Key size</td>
<td>10 Bytes</td>
</tr>
<tr>
<td>Value size</td>
<td>200 Bytes</td>
</tr>
<tr>
<td>Threads</td>
<td>20</td>
</tr>
<tr>
<td>Operations</td>
<td>10,000</td>
</tr>
<tr>
<td>Range-query portion</td>
<td>0.1</td>
</tr>
<tr>
<td>Update portion</td>
<td>0.9</td>
</tr>
<tr>
<td>Limit</td>
<td>1 to 70,000</td>
</tr>
<tr>
<td>Start key</td>
<td>Random</td>
</tr>
<tr>
<td>Collector</td>
<td>Client</td>
</tr>
</tbody>
</table>

As shown in Figure 5.9, there is a dramatic increase in the latency when the number of items becomes large. For this particular settings, the latency increases linearly for range-query operations with limits less than 50,000, regardless of the size of the data. Figures 5.7 shows that our range-query solution performs better when the same size of data is stored in less number of key-value pairs.

![Figure 5.9.](image-url) Experiment 5. Average, 95th percentile and max latencies of range-query operations per range-query limit. Value size is 200 bytes.
5.6 Experiment 6

In this experiment we investigated the effect of node failures. We executed CATS simulator with experiment scripts in the test mode, with a replication factor of 3. In each round, we failed one of the replicas in a replication group. Right after the failure, we started a client to execute a range-query on the range of the replication group. After some time, that a new replication group has been formed for the range, the failed node joins again. Right after that we start a new client to execute the range-query. After some time, that replication groups are reformed because of the joined node, we execute the range-query one more time. All tests passed for all range-queries in all rounds.

Figure 5.10 shows the script used in one of the rounds. In this scenario, we failed one of the nodes in the replication group \{0999,1999,2999\} in each round.

![Experiment 6 Experiment script in test mode for testing range-queries in the presence of node joins and failures.]

5.7 Experiment 7

This experiment investigates the likeliness of facing INTERRUPTED in a range-query operation result. We used a cluster of seven CATS servers with a key space of [0000, 6999] (Figure 5.1) and a replication factor of 5. For this experiment we used three clients that were modified versions of the Basic Client. Figure 5.11 shows clients and server nodes that were directly involved in this experiment.

Client 1 continuously sends range-query \{0995,1005\} request to the first replica of the range \{0999,1999\}. It sends a new range-query request after receiving a response for the previous request. Client 2 continuously Puts null and non-null values for the key "0998". Every 10 milliseconds, it periodically sends PUT(0998,null) and PUT(0998,value) requests to the second replica of the range \{0999,1999\}. Client
3 operates similar to Client 2. But, it sends PUT requests to the third replica of the range \([0999,1999)\).

Clients 2 and 3 continuously sends put requests during the experiment time. Client 1 sends 100,000 range-query \([0995,1005)\) requests. The value of the key '0998' is super dynamic; more than a realistic scenario. However, we faced only 86 INTERRUPTED in 100,000 range-query operations. Even in this unrealistic scenario, the chance of facing INTERRUPTED is less than 1 to 1000. We conclude than it is very unlikely to receive a range-query response with INTERRUPTED result.
Chapter 6

Conclusion

CATS key-value store supports Put and Get operations for reading and writing single key-value pairs. It provides strong data consistency with respect to linearizability. However, CATS does not support range-queries, which is useful in many applications. We have designed and evaluated a lightweight consistent range-query mechanism for CATS. Our solution guarantees strong data consistency for all returned key-value pairs, with respect to linearizability. This enables us to build new classes of applications using CATS.

We have evaluated our range-query solution from different perspectives. Our solution scales well for large size of data. There is no decrease in the performance if the requested items are distributed on several servers. With increasing the range-query limit up to some point, the latency increases linearly regardless of the total size of the data. In general, our solution performs better when the same size of data is stored in less number of key-value pairs.

6.1 Future Work

Our range-query solution can be used to build a high level data model, which supports secondary indexes, on top of CATS. A project is in progress at the Swedish Institute of Computer Science (SICS) that targets this data model.

To make range-queries possible, we used order preserving hashing for keys. This might lead to load imbalance if the data is skewed. A load-balancing mechanism is therefore necessary for a version of CATS that supports range-queries. We leave it to future work to provide a load-balancing mechanism for CATS.

Currently, our range-query solution does not perform well for very large limits (e.g., querying tens of thousands of items in one request). It can be optimized for range-queries that return very large number of items.

There might be limitations on the size of messages that the network component of CATS can transfer. To overcome the limitations, large messages must be chunked into smaller ones. Our solution already provides a collector module for aggregating messages on the client side. We suggest using this capability to chunk large messages.
into several smaller messages and aggregate them in the Client Collector.

Our solution provides strong consistency per data item. The solution might be extended to use snapshots in order to provide stronger guarantees for consistent range-queries, e.g., strong consistency per set of items in a range. It might be extended to support multi-item transactions.

This work can also be extended to support range-queries with considerations about the value conditions of the key-value pairs in a given range.

Finally, we suggest implementing a distributed database cursor for our range-query solution.
Bibliography


List of Figures

2.1 A Chord ring and its identifier circle . ........................................ 11
2.2 Replication group ................................................................. 13
2.3 Inconsistent replication groups from the perspective of different nodes . 13

3.1 Forwarding the range-query request to a responsible node .................. 20
3.2 read-range requests and corresponding responses for sub-ranges .......... 21
3.3 range-query and read-range requests and corresponding responses ......... 23
3.4 read-range operation in detail .................................................. 24
3.5 read-range in the presence of concurrent put operations (Scenario 1) .... 25
3.6 read-range in the presence of concurrent put operations (Scenario 2) .... 26
3.7 A range-query operation resulting in INTERRUPTED ...................... 27
3.8 A read-range operation resulting in INTERRUPTED ...................... 28
3.9 Server Collector ................................................................. 29
3.10 Client Collector ............................................................... 30
3.11 Check & Forward policy for range-query request forwarding .......... 32
3.12 Enhanced Local Read policy for range-query request forwarding ....... 33

4.1 A CATS node with a Consistent Range-Query (CRQ) component ......... 36
4.2 Network port and Events related to the range-query component ......... 36
4.3 The LocalRQ port and its corresponding Events ............................ 37
4.4 The LocalABD port and its corresponding Events ............................ 37
4.5 CATS Interface Client (CIC) .................................................. 38
4.6 CATSInterface Port and the corresponding Events .......................... 38
4.7 Basic Client with a CATS Interface Client component .......... 39
4.8 Sample script for an experiment scenario .................................... 40
4.9 Simplified model of the CATS client for YCSB ............................. 42
5.1 A cluster of seven CATS nodes, each running on a single machine . . . 44
5.2 Exp. 1: Average latencies of range-queries per start key . . . . . . . . 45
5.3 Exp. 1: range-query (0500,0500)#1000 using Client Collector . . . . 45
5.4 Exp. 1: range-query (0500,0500)#1000 using Server Collector . . . . 46
5.5 Exp. 1: range-query using Server Collector and Enhanced Local Read . 46
5.6 Exp. 2: range-query latency per number of threads . . . . . . . . . . . 47
5.7 Exp. 3: range-query latency per query limit with random start . . . . 48
5.8 Exp. 4: range-query latency per query limit with constant start . . . 49
5.9 Exp. 5: range-query latency per query limit for large limits . . . . . 50
5.10 Exp. 6: Experiment script for testing range-queries when nodes join/fail 51
5.11 Exp. 7: INTERRUPTED test . . . . . . . . . . . . . . . . . . . . . . . 52
Appendix A

Algorithms of ABD
Algorithm A.1 Read-Impose Write-Consult-Majority [11] (part 1, read and consult)

**Implements:**

\((N, N)\)-AtomicRegister, instance nnar.

**Uses:**

BestEffortBroadcast, instance beb;
PerfectPointToPointLink, instance pl;

upon event < nnar, Init > do

\((ts, wr, val) := (0, 0, \bot)\);

acks := 0;

writeval := \bot;

rid := 0;

readlist := [\bot]^N;

readval := \bot;

reading := \text{FALSE};

upon event < nnar, Read > do

rid := rid + 1;

acks := 0;

readlist := [\bot]^N;

reading := \text{TRUE};

trigger < beb, Broadcast | [READ, rid] >;

upon event < beb, Deliver | p, [READ, r] > do

trigger < pl, Send | p, [VALUE, r, rs, wr, val] >;

upon event < pl, Deliver | q, [VALUE, r, ts', wr', v'] > such that \(r = rid\) do

readlist[q] := (ts', wr', v');

if \(#(\text{readlist}) > N/2\) then

\((\text{maxts}, rr, \text{readval}) := \text{highest}(\text{readlist});\)

readlist := [\bot]^N;

if reading = \text{TRUE} then

trigger < beb, Broadcast | [WRITE, rid, maxts, rr, \text{readval}] >;

else

trigger < beb, Broadcast | [WRITE, rid, maxts + 1, rank(self), writeval] >;
Algorithm A.2 Read-Impose Write-Consult-Majority (part 2, write and write-back)

upon event \(<\text{nnar}, \text{Write} | r >\) do
   \(\text{rid} := \text{rid} + 1;\)
   \(\text{writeval} := v;\)
   \(\text{acks} := 0;\)
   \(\text{readlist} := [\bot]^{N};\)
   \(\text{trigger} < \text{beb}, \text{Broadcast} | \text{READ}, \text{rid} >;\)

upon event \(<\text{beb}, \text{Deliver} | p, \text{[WRITE, r, ts', wr', v']} >\) do
   if \((ts', wr')\) is larger than \((ts, wr)\) then
      \((ts, wr, \text{val}) := (ts', wr', v');\)
      \(\text{trigger} < \text{pl}, \text{Send} | p, \text{[ACK, r]} >;\)

upon event \(<\text{pl}, \text{Deliver} | q, \text{[ACK, r]} \text{ such that } r = \text{rid} >\) do
   \(\text{acks} := \text{acks} + 1;\)
   if \(\text{acks} > N/2\) then
      \(\text{acks} := 0;\)
      if \(\text{reading} = \text{TRUE}\) then
         \(\text{reading} := \text{FALSE};\)
         \(\text{trigger} < \text{nnar}, \text{ReadReturn} | \text{readval} >;\)
      else
         \(\text{trigger} < \text{nnar}, \text{WriteReturn} >;\)
Appendix B

Algorithms of Consistent Range-Query
Algorithm B.1 Consistent Range-Query (part 1, events)

**Implements:**
Consistent-Range-Query, instance `crq`.

**Uses:**
NetworkLink, instance `net`.
Timer, instance `timer`.
CATS-ABD, instance `abd`.

```plaintext
upon event <Init> do
    g_active_rangequeries = ⊥
    g_active_readranges = ⊥

upon event <net, RANGEQUERY | src, id, range, limit, consistent, collector_type> do
    if (the request have to be forwarded according to the policy) then
        forward the request according to the policy
        return
    if (collector_type = SERVER) then
        Initialize(id)
        trigger <timer, SET | RANGEQUERY_TIMEOUT, id>
        collector = self
        g_active_rangequeries.add(id)
        g_client[id] = src
        g_total_range[id] = range
        g_total_limit[id] = limit
    else
        collector = src
        trigger <net, READRANGE | self, id, 0, range, limit, consistent, collector>

upon event <net, READRANGE | src, id, seq, range, limit, consistent, collector> do
    Initialize(id, seq)
    trigger <timer, SET | READRANGE_TIMEOUT | id, seq>
    g_active_readranges.add(<id, seq>)
    g_collector[id][seq] = collector
    g_range[id][seq] = range
    g_limit[id][seq] = limit
    g_consistent[id][seq] = consistent
    g_subrange[id][seq] = the sub-range handled by the node according to the policy
    if (consistent = True) then
        for (replica in Other_Replicas(g_subrange[id][seq])) do
            trigger <net, RR_STAT | replica, id, seq, g_subrange[id][seq], limit, id>
        else
            g_stat_completed[id][seq] = True
    trigger <net, RR_LOCALITEMS | self, id, seq, g_subrange[id][seq], limit>

upon event <net, RR_STAT | src, id, seq, range, limit> do
    if (Is_Responsible(range) = False) then
        return
    items = Local_Read(range, limit, False)
    more = More_Items_Available(range, limit)
    group = Replication_Group(range)
    trigger <net, RR_STAT.RESP | src, id, range, items, group, more>
```
Algorithm B.2 Consistent Range-Query (part 2, events)

upon event \(<net, RR\_LOCALITEMS | src, id, seq, range, limit> do
  if \((Is\_Responsible(range) = False)\) then
    return
  g\_localitems[id][seq] = Local\_Read(range, limit, True)
  g\_local\_group[id][seq] = Replication\_Group(range)
  g\_local\_items\_completed[id][seq] = True
  for (item in g\_local\_items[id][seq]) do
    g\_tokens[id][seq].add(item\_token)
  if \((More\_Items\_Available(range, limit))\) then
    g\_more[id][seq] = True
  if \((g\_stat\_completed[id][seq] = True)\) then
    Check\_Items(id, seq)

upon event \(<net, RR\_STAT\_RESP | src, id, seq, range, items, group, more> do
  if \((g\_stat\_completed[id][seq] = False)\) then
    for (item in items) do
      items\_stat.add(<item\_token, item\_timestamp>)
      g\_tokens[id][seq].add(item\_token)
    g\_stat[id][seq][group].add(<items\_stat>)
    if \((more = True)\) then
      g\_more[id][seq] = True
    if \((g\_stat[id][seq][group].size = REPLICATION\_FACTOR/2)\) then
      g\_stat\_completed = True
      if \((g\_local\_items\_completed[id][seq] = True)\) then
        Check\_Items(id, seq)

case \(<abd, GET\_RESP | id, seq, token, value, result> do
  if \((not g\_active\_readranges\_contain(<id, seq>))\) then
    return
  if \((result = SUCCESS)\) then
    if \((value \neq NULL)\) then
      g\_waiting\_tokens[id][seq].remove(token)
      g\_ready\_tokens[id][seq].add(token)
      g\_items[id][seq].add(<token, value>)
    else
      g\_waiting\_tokens[id][seq].remove(token)
  if \((Is\_Ready\_To\_Respond(id, seq) = True)\) then
    Respond\_And\_Continue\_RR(id, seq)
  else
    Abd\_Get\_If\_Needed(id, seq, token)

upon event \(<timer, READRANGE\_TIMEOUT | id, seq> do
  result = TIMEOUT
  is\_last = True
  more = False
  items = NULL
  range = g\_subrange[id][seq]
  limit = g\_limit[id][seq]
  collector = g\_collector[rq][id]
  g\_active\_readranges\_remove(<id, seq>)

trigger \(<net, READRANGE\_RESP | collector, id, seq, range, limit, items, result, is\_last, more>
Algorithm B.3 Consistent Range-Query (part 3, events and functions)

upon event <timer, RANGEQUERY_TIMEOUT | id> do
    result = TIMEOUT
    items = NULL
    range = g_total_range[id]
    limit = g_total_limit[id]
    g_active_rangequeries.remove(id)
    trigger <net, RANGEQUERY.RESP | g_client[id], id, range, limit, items, result>

upon event <net, READRANGE.RESP | src, id, seq, range, limit, items, result, is_last, more> do
    if (not g_active_rangequeries.contains(id)) then
        return
    if (result = TIMEOUT) then
        trigger <timer, CANCEL | RANGEQUERY_TIMEOUT, id>
        g_active_rangequeries.remove(id)
        items = NULL
        range = g_total_range[id]
        limit = g_total_limit[id]
        trigger <net, RANGEQUERY.RESP | g_client[id], id, range, limit, items, result>
        return
    g_received_chunks[id] += 1
    g_items[id].add_all(items)
    if (is_last = True) then
        g_last_seq[id] = seq
        g_last_chunk_result[id] = result
    if (Have_All_Chunks(id) = True) then
        trigger <timer, CANCEL | RANGEQUERY_TIMEOUT | id>
        g_active_rangequeries.remove(id)
        result = g_last_chunk_result[id]
        range = g_total_range[id]
        limit = g_total_limit[id]
        trigger <net, RANGEQUERY.RESP | g_client[id], id, range, limit, items, result>

function Initialize (id)
    g_received_chunks[id] = 0
    g_last_seq[id] = ⊥
    g_last_chunk_result[id] = ⊥
    g_items[id] = ⊥

function Initialize (id, seq)
    g_localitems_completed[id][seq] = False
    g_stat_completed[id][seq] = False
    g_more[id][seq] = False
    g_localitems[id][seq] = ⊥
    g_tokens[id][seq] = ⊥
    g_waiting_tokens[id][seq] = ⊥
    g_ready_tokens[id][seq] = ⊥
Algorithm B.4 Consistent Range-Query (part 4, functions)

function Check_Items (id, seq)
    if (g_consistent[id][seq] = True) then
        group = g_local_group[id][seq]
        for (token in g_tokens[id][seq]) do
            if (g_localitems[id][seq][token] ≠ NULL) then // has it locally
                if (Same_Timestamps(id, seq, group, token) = True) then
                    Local_Get(id, seq, token)
                else
                    Abd_Get_If_Needed(id, seq, token)
            else
                Abd_Get_If_Needed(id, seq, token)
        if (Is_Ready_To_Respond(id, seq) = True) then
            Respond_And_Continue_RR(id, seq)
        else
            for (token in g_tokens[id][seq]) do
                g_items[id][seq].add(<token, g_localitems[id][seq][token].value>)
            Respond_And_Continue_RR(id, seq)

function Same_Timestamps (id, seq, group, token)
    for (int i = 0; i < g_stat[id][seq][group].size; i++) do
        if (g_stat[id][seq][group].get(i)[token] ≠ NULL) then
            if (g_stat[id][seq][group].get(i)[token] ≠ g_localitems[id][seq][token]) then
                return False
            else
                return False
        return True

function Have_Enough_Items(id, seq)
    if (g_items[id][seq].size = g_limit[id][seq] and g_waiting_tokens[id][seq].first > g_ready_tokens[id][seq].last) then
        return True
    return False

function Replication_Group(range)
    group = members of the replication group for range
    return group

function Other_Replicas(range)
    replicas = Replication_Group(range)
    replicas.remove(self)
    return replicas

function Is_Responsible(range)
    if (node is replicating the range) then
        return True
    return False

function Local_Read(range, limit, return_value)
    if (return_value = True) then
        items = <token, value, timestamp> of first limit items with non-null value in range
    else
        items = <token, NULL, timestamp> of first limit items with non-null value in range
    return items
Algorithm B.5 Consistent Range-Query (part 5, functions)

function More_Items_Available(range, limit)
    if (number of items with non-null values in range > limit) then
        return True
    return False

function Local_Get (id, seq, token)
    g_ready_tokens[id][seq].add(token)
    g_items[id][seq].add(<token, g_localitem[id][seq][token].value>)

function Abd_Get_If_Needed (id, seq, token)
    if (Have_Enough_Items(id, seq)) then
        Respond_And_Continue_RR(id, seq)
    return
    if (g_waiting_tokens[id][seq][token] = NULL) then
        g_waiting_tokens[id][seq].add(token)
    trigger <abd, GET | id, seq, token>

function Is_Ready_To_Respond (id, seq)
    if (g_waiting_tokens[id][seq].is_empty() or Have_Enough_Items(id, seq)) then
        return True
    return False

function Respond_And_Continue_RR (id, seq)
    trigger <timer, CANCEL | READRANGE_TIMEOUT, id, seq>
    size = g_items[id][seq].size
    limit = g_limit[id][seq]
    range = g_range[id][seq]
    subrange = g_subrange[id][seq]
    if (size < limit and g_more[id][seq] = True) then
        is_last = True
        result = INTERRUPTED
    else if (size < limit and subrange ≠ range) then
        // collect more items
        is_last = False
        result = SUCCESS
        remaining_range = range - subrange
        remaining_limit = limit - size
        next = Next_Node()
        trigger <net, READRANGE | next, id, seq + 1, remaining_range, remaining_limit,
                g_consistent[id][seq], g_collector[id][seq]>
    else
        is_last = True
        result = SUCCESS
        g_active_readranges.remove(<id, seq>)
    trigger <net, READRANGE.RESP | g_collector[id][seq], id, seq, g_subrange[id][seq],
                g_limit[id][seq], g_items[id][seq], result, is_last, g_more[id][seq]>

function Have_All_Chunks (id)
    if (g_last_seq[id] = g_received_chunks[id] - 1) then
        return True
    return False

function Next_Node ()
    return one of the successors based on the policy
Appendix C

Algorithms of CATS Interface Client (CIC)
Algorithm C.1 CATS Interface Client (part 1, range-query events)

Implements:
CATS-Interface-Client, instance cic.

Uses:
NetworkLink, instance net.
Timer, instance timer.

upon event <Init> do
  g_active_rangequeries = ⊥
  g_op_count = 0

upon event <cic, CIRANGEQUERY | req_id, range, limit, consistent, collector_type> do
  id = Get Unique Id(req_id)
  if (collector_type = CLIENT) then
    trigger <timer, SET | RANGEQUERY_TIMEOUT, id>
    g_total_range[id] = range
    g_total_limit[id] = limit
    g_received_chunks[id] = 0
    g_last_seq[id] = ⊥
    g_last_chunk_result[id] = ⊥
    g_items[id] = ⊥
    g_active_rangequeries.add(id)
  server = Get Server()
  trigger <net, RANGEQUERY | server, id, range, limit, consistent, collector_type>

upon event <net, READRANGE.RESP | src, id, seq, range, limit, items, result, is_last, more> do
  //when collector_type = CLINET
  if (not g_active_rangequeries.contain(id)) then
    return
  if (result = TIMEOUT) then
    trigger <timer, CANCEL | RANGEQUERY_TIMEOUT, id>
    g_active_rangequeries.remove(id)
    items = NULL
    range = g_total_range[id]
    limit = g_total_limit[id]
    trigger <cic, CIRANGEQUERY.RESP | id.req_id, range, limit, items, result>
    return
  g_received_chunks[id] += 1
  g_items[id].add_all(items)
  if (is_last = True) then
    g_last_seq[id] = seq
    g_last_chunk_result[id] = result
  if (Have All Chunks(id) = True) then
    trigger <timer, CANCEL | RANGEQUERY_TIMEOUT, id>
    g_active_rangequeries.remove(id)
    items = Sort(g_items[id])
    result = g_last_chunk_result[id]
    range = g_total_range[id]
    limit = g_total_limit[id]
    trigger <cic, CIRANGEQUERY.RESP | id.req_id, range, limit, items, result>

upon event <net, RANGEQUERY.RESP | src, id, range, limit, items, result> do
  //when collector_type = SERVER
  trigger <cic, CIRANGEQUERY.RESP | id.req_id, range, limit, items, result>
Algorithm C.2 CATS Interface Client (part 2, range-query events and functions)

upon event \(<\text{timer, RANGEQUERY\_TIMEOUT}|\text{id}>\) do
    result = TIMEOUT
    items = NULL
    range = g\_total\_range[\text{id}]
    limit = g\_total\_limit[\text{id}]
    g\_active\_rangequeries.remove(\text{id})
    trigger \(<\text{req, CIRANGEQUERY\_RESP}|\text{id}\_req\_id, \text{range}, \text{limit}, \text{items}, \text{result}>\)

function Have\_All\_Chunks (\text{id})
    if (g\_last\_seq[\text{id}] = g\_received\_chunks[\text{id}] - 1) then
        return True
    return False

function Get\_Unique\_Id(\text{req\_id})
    g\_op\_count+=1
    id = \text{<req\_id, self, g\_op\_count>}
    return id

function Get\_Server()
    server = address of a CATS node based on the policy (random, ...)
    return server