Learn Cassandra

teddyma

Published with GitBook

Table of Contents

Learn Cassandra

This book step-by-step guides **developers** to understand what Cassandra is, how Cassandra works and how to use the features and capabilities of Apache Cassandra 2.0.

This book focus more on the develop's view, meaning you will not find too much about installation or administration of Cassandra, which could be another separate big topic for DBAs.

License

About Cassandra

In this chapter, we will try to summarize what Cassandra can do for you.

Apache Cassandra is a massively scalable open source NoSQL database.

Cassandra is the right choice for managing large amounts of structured, semi-structured, and unstructured data across multiple data centers and the cloud, when you need scalability and high availability without compromising performance.

Cassandra delivers continuous availability, linear scalability, and operational simplicity across many commodity servers with no single point of failure.

Cassandra's data model offers the convenience of column indexes with the performance of log-structured updates, strong support for denormalization and materialized views, and powerful built-in caching.

The History of Cassandra

[1]Apache Cassandra was developed at Facebook to power their Inbox Search feature by Avinash Lakshman (one of the authors of Amazon's Dynamo) and Prashant Malik. It was released as an open source project on Google code in July 2008. In March 2009, it became an Apache Incubator project. On February 17, 2010 it graduated to a top-level project.

Releases after graduation include:

- 0.6, released Apr 12 2010, added support for integrated caching, and Apache Hadoop MapReduce
- 0.7, released Jan 08 2011, added secondary indexes and online schema changes
- 0.8, released Jun 2 2011, added the Cassandra Query Language (CQL), self-tuning memtables, and support for zero-downtime upgrades
- 1.0, released Oct 17 2011, added integrated compression, leveled compaction, and improved read performance
- 1.1, released Apr 23 2012, added self-tuning caches, row-level isolation, and support for mixed ssd/spinning disk deployments
- 1.2, released Jan 2 2013, added clustering across virtual nodes, inter-node communication, atomic batches, and request tracing
- 2.0, released Sep 4 2013, added lightweight transactions (based on the Paxos consensus protocol), triggers, improved compactions, CQL paging support, prepared statement support, SELECT column alias support

References

1. http://en.wikipedia.org/wiki/Apache_Cassandra

The CAP Theorem

[1]The CAP theorem, also known as Brewer's theorem, states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:

- Consistency (all nodes see the same data at the same time)
- Availability (a guarantee that every request receives a response about whether it was successful or failed)
- Partition tolerance (the system continues to operate despite arbitrary message loss or failure of part of the system)

According to the theorem, a distributed system cannot satisfy all three of these guarantees at the same time.

Cassandra and CAP

Cassandra is typically classified as an AP system, meaning that availability and partition tolerance are generally considered to be more important than consistency in Cassandra. But Cassandra can be tuned with replication factor and consistency level to also meet C.

References

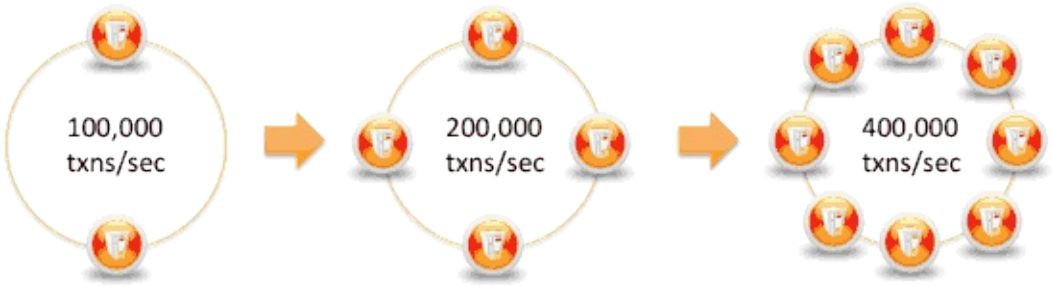
1. http://en.wikipedia.org/wiki/CAP_theorem

What Can Cassandra Do?

The architecture of Cassandra is “masterless”, meaning all nodes are the same. Cassandra provides automatic data distribution across all nodes that participate in a “ring” or database cluster. There is nothing programmatic that a developer or administrator needs to do or code to distribute data across a cluster because data is transparently partitioned across all nodes in a cluster.

Cassandra also provides built-in and customizable replication, which stores redundant copies of data across nodes that participate in a Cassandra ring. This means that if any node in a cluster goes down, one or more copies of that node’s data is available on other machines in the cluster. Replication can be configured to work across one data center, many data centers, and multiple cloud availability zones.

[1]Cassandra supplies linear scalability, meaning that capacity may be easily added simply by adding new nodes online. For example, if 2 nodes can handle 100,000 transactions per second, 4 nodes will support 200,000 transactions/sec and 8 nodes will tackle 400,000 transactions/sec:



References

1. http://www.datastax.com/documentation/cassandra/2.0/cassandra/gettingStartedCassandraIntro.html

Data Model & CQL

In this chapter, we will introduce the data model, CQL, the indexing of Cassandra, and how CQL maps to Cassandra's internal data structure.

Internal Data Structure

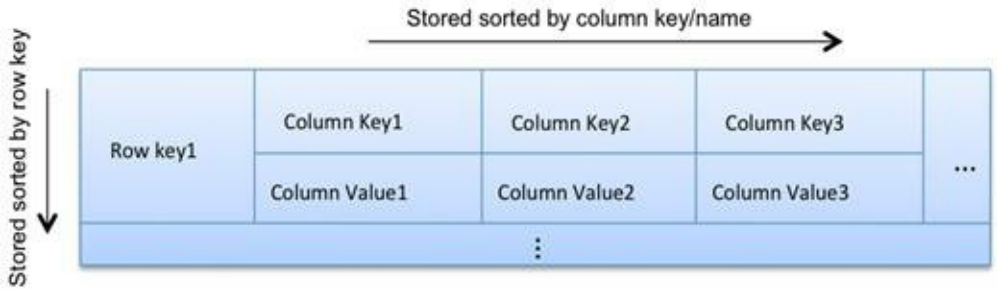
[1]The Cassandra data model is a schema-optional, column-oriented data model. This means that, unlike a relational database, you do not need to model all of the columns required by your application up front, as each row is not required to have the same set of columns.

[2]The Cassandra data model consists of keyspaces (analogous to databases), column families (analogous to tables in the relational model), keys and columns.

Relational Model	Cassandra Model
Database	Keyspace
Table	Column Family (CF)
Primary key	Row key
Column name	Column name/key
Column value	Column value

For each column family, don't think of a relational table. Instead, think of a nested sorted map data structure. A nested sorted map is a more accurate analogy than a relational table, and will help you make the right decisions about your Cassandra data model.

```
Map<RowKey, SortedMap<ColumnKey, ColumnValue>>
```



A map gives efficient key lookup, and the sorted nature gives efficient scans. In Cassandra, we can use row keys and column keys to do efficient lookups and range scans. The number of column keys is unbounded. In other words, you can have wide rows.

A key can itself hold a value as part of the key name. In other words, you can have a valueless column.

References

1. http://www.datastax.com/docs/0.8/ddl/index
2. http://www.bodhtree.com/blog/2013/12/06/my-experience-with-cassandra-concepts/

CQL

[1]CQL consists of statements. Like SQL, statements change data, look up data, store data, or change the way data is stored. Statements end in a semicolon (;).

For example, the following is valid CQL syntax:

```
SELECT * FROM MyTable;

UPDATE MyTable
  SET SomeColumn = 'Some Value'
  WHERE columnName = 'Something Else';
```

This is a sequence of two CQL statements. This example shows one statement per line, although a statement can usefully be split across lines as well.

Uppercase and Lowercase

Keyspace, column, and table names created using CQL are case-insensitive unless enclosed in double quotation marks. If you enter names for these objects using any uppercase letters, Cassandra stores the names in lowercase. You can force the case by using double quotation marks.

For example:

```
CREATE TABLE test (
  Foo int PRIMARY KEY,
  "Bar" int
);
```

CQL keywords are case-insensitive. For example, the keywords SELECT and select are equivalent.

CQL Data Types

[2]CQL defines built-in data types for columns:

CQL Type	Constants	Description
ascii	strings	US-ASCII character string
bigint	integers	64-bit signed long
blob	blobs	Arbitrary bytes (no validation), expressed as hexadecimal
boolean	booleans	true or false
counter	integers	Distributed counter value (64-bit long)
decimal	integers, floats	Variable-precision decimal
double	integers, floats	64-bit IEEE-754 floating point
float	integers, floats	32-bit IEEE-754 floating point
inet	strings	IP address string in IPv4 or IPv6 format*
int	integers	32-bit signed integer
list<T>	n/a	A collection of one or more ordered elements, T could be any non-collection CQL

		data type, such as int, text, etc
map<K,V>	n/a	A key-value dictionary, K and V could be any non-collection CQL data type, such as int, text, etc
set<T>	n/a	A collection of one or more elements, T could be any non-collection CQL data type, such as int, text, etc
text	strings	UTF-8 encoded string
timestamp	integers, strings	Date plus time, encoded as 8 bytes since epoch
uuid	uuids	A UUID in standard UUID format
timeuuid	uuids	Type 1 UUID only
varchar	strings	UTF-8 encoded string
varint	integers	Arbitrary-precision integer

CQL Command Reference

For a complete CQL command reference, see CQL commands.

References

1. http://www.datastax.com/documentation/cql/3.1/cql/cql_reference/cql_lexicon_c.html
2. http://www.datastax.com/documentation/cql/3.1/cql/cql_reference/cql_data_types_c.html

CQL & Data Structure

The recommended API for creating schemas in Cassandra since 0.7 is via CQL. But Cassandra encourages developer to share schema information to achieve more transparency. Why? Because although CQL looks very much like SQL, they don't work in a similar way internally. Remember, for each column family, don’t think of a relational table. Instead, think of a nested sorted map data structure.

A Simple Example

For example, creating ColumnFamily(Table) with CQL:

```
CREATE TABLE example (
   field1 int PRIMARY KEY,
   field2 int,
   field3 int);
```

And insert a few example rows:

```
INSERT INTO example (field1, field2, field3) VALUES (1,2,3);
INSERT INTO example (field1, field2, field3) VALUES (4,5,6);
INSERT INTO example (field1, field2, field3) VALUES (7,8,9);
```

How is the data stored?

```
RowKey: 1
=> (column=, value=, timestamp=1374546754299000)
=> (column=field2, value=00000002, timestamp=1374546754299000)
=> (column=field3, value=00000003, timestamp=1374546754299000)
-------------------
RowKey: 4
=> (column=, value=, timestamp=1374546757815000)
=> (column=field2, value=00000005, timestamp=1374546757815000)
=> (column=field3, value=00000006, timestamp=1374546757815000)
-------------------
RowKey: 7
=> (column=, value=, timestamp=1374546761055000)
=> (column=field2, value=00000008, timestamp=1374546761055000)
=> (column=field3, value=00000009, timestamp=1374546761055000)
```

For each item above, there are 3 important things to look at: the row key (RowKey: <?>), the column name (column=<?>) and the column value (value=<?>). From these examples, we can make a couple of initial observations about the mapping from CQL statements to their internal representations.

- The value of the CQL primary key is used internally as the row key (which in the new CQL paradigm is being called a “partition key”)
- The names of the non-primary key CQL fields are used internally as columns names, and the values of the non-primary key CQL fields are then internally stored as the corresponding column values
- The columns of each *RowKey* is sorted by column names

You may have also noticed that these rows all contain columns with no column name and no column value. This is not a bug! It’s actually a way of handling the fact that it should be possible to declare the existence of field1=<some number> without necessarily specifying values for field2 or field3.

A More Complex Example

A more complicated example:

```
CREATE TABLE example (
    partitionKey1 text,
    partitionKey2 text,
    clusterKey1 text,
    clusterKey2 text,
    normalField1 text,
    normalField2 text,
    PRIMARY KEY (
        (partitionKey1, partitionKey2),
        clusterKey1, clusterKey2
        )
    );
```

Here we've named the fields to indicate where they'll end up in the internal representation. And we've also pulled out all the stops. Our primary key is not only compound, but it also uses compound partition keys, (this is represented by the double nesting of the parentheses of the PRIMARY KEY) and compound cluster keys (more on partition vs. cluster keys in a bit).

And insert a row of data:

```
INSERT INTO example (
    partitionKey1,
    partitionKey2,
    clusterKey1,
    clusterKey2,
    normalField1,
    normalField2
    ) VALUES (
    'partitionVal1',
    'partitionVal2',
    'clusterVal1',
    'clusterVal2',
    'normalVal1',
    'normalVal2');
```

How is the data stored?

```
RowKey: partitionVal1:partitionVal2
=> (column=clusterVal1:clusterVal2:, value=, timestamp=1374630892473000)
=> (column=clusterVal1:clusterVal2:normalfield1, value=6e6f726d616c56616c31, timestamp=1374630892473000)
=> (column=clusterVal1:clusterVal2:normalfield2, value=6e6f726d616c56616c32, timestamp=1374630892473000)
```

- Looking at *partitionVal1* and *partitionVal2* we see that these are concatenated together and used internally as the *RowKey* (a.k.a. the partition key)
- *clusterVal1* and *clusterVal2* are concatenated together and used in the names of each of the non-primary key columns
- The actual values of *normalfield1* and *normalfield2* are encoded as the values of columns *clusterVal1:clusterVal2:normalfield1* and *clusterVal1:clusterVal2:normalfield2* respectively, that *is6e6f726d616c56616c31* becomes *normalVal1* and *6e6f726d616c56616c32* becomes *normalVal2*
- The columns of each *RowKey* is sorted by column names, and since the values of *clusterKey1* and *clusterKey2* are part of the column names, they are actually sorted by values of *clusterKey1* and *clusterKey2* first, and then the original column names specified in CQL

Map, List & Set

An example of how sets, lists, and maps work under the hood:

```
CREATE TABLE example (
   key1 text PRIMARY KEY,
   map1 map<text,text>,
   list1 list<text>,
   set1 set<text>
   );
```

Insert:

```
INSERT INTO example (
   key1,
   map1,
   list1,
   set1
   ) VALUES (
   'john',
   {'patricia':'555-4326','doug':'555-1579'},
   ['doug','scott'],
   {'patricia','scott'}
   )
```

How is data stored?

```
RowKey: john
=> (column=, value=, timestamp=1374683971220000)
=> (column=map1:doug, value='555-1579', timestamp=1374683971220000)
=> (column=map1:patricia, value='555-4326', timestamp=1374683971220000)
=> (column=list1:26017c10f48711e2801fdf9895e5d0f8, value='doug', timestamp=1374683971220000)
=> (column=list1:26017c12f48711e2801fdf9895e5d0f8, value='scott', timestamp=1374683971220000)
=> (column=set1:'patricia', value=, timestamp=1374683971220000)
=> (column=set1:'scott', value=, timestamp=1374683971220000)
```

Internal column names for map, list and set items have different patterns:

- For map, each item in the map becomes a column, and the column name is the combination of the map column name and the key of the item, the value is the value of the item
- For list, each item in the list becomes a column, and the column name is the combination of the list column name and the UUID of the item order in the list, the value is the value of the item
- For set, each item in the set becomes a column, and the column name is the combination of the set column name and the item value, the value is always empty

References

1. http://www.opensourceconnections.com/2013/07/24/understanding-how-cql3-maps-to-cassandras-internal-data-structure/
2. http://www.opensourceconnections.com/2013/07/24/understanding-how-cql3-maps-to-cassandras-internal-data-structure-sets-lists-and-maps/

Where Is Data Stored

For each column family, there are 3 layer of data stores: memtable, commit log and SSTable.

[1]For efficiency, Cassandra does not repeat the names of the columns in memory or in the SSTable. For example, data is inserted using these CQL statements:

```
INSERT INTO k1.t1 (c1) VALUES (v1);
INSERT INTO k2.t1 (c1, c2) VALUES (v1, v2);
INSERT INTO k1.t1 (c1, c3, c2) VALUES (v4, v3, v2);
```

In the memtable, Cassandra stores this data:

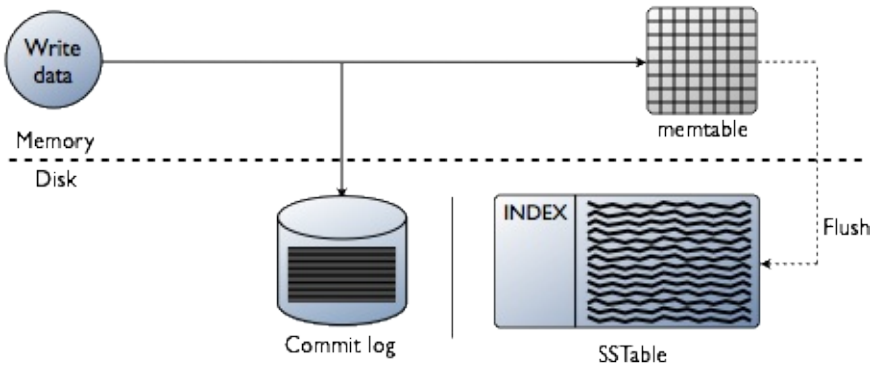
```
k1 c1:v4 c2:v2 c3:v3
k2 c1:v1 c2:v2
```

In the commit log on disk, Cassandra stores this data:

```
k1, c1:v1
k2, c1:v1 C2:v2
k1, c1:v4 c3:v3 c2:v2
```

In the SSTable on disk, Cassandra stores this data after flushing the memtable:

```
k1 c1:v4 c2:v2 c3:v3
k2 c1:v1 c2:v2
```



The Write Path

When a write occurs, Cassandra stores the data in a structure in memory, the memtable, and also appends writes to the commit log on disk.

The memtable is a write-back cache of data partitions that Cassandra looks up by key. The more a table is used, the larger its memtable needs to be. Cassandra can dynamically allocate the right amount of memory for the memtable or you can manage the amount of memory being utilized yourself. The memtable, unlike a write-through cache, stores writes until reaching a limit, and then is flushed.

When memtable contents exceed a configurable threshold, the memtable data, which includes indexes, is put in a queue to be flushed to disk. To flush the data, Cassandra sorts memtables by partition key and then writes the data to disk sequentially. The process is extremely fast because it involves only a commitlog append and the sequential write.

Data in the commit log is purged after its corresponding data in the memtable is flushed to the SSTable. The commit log is

for recovering the data in memtable in the event of a hardware failure.

SSTables are immutable, not written to again after the memtable is flushed. Consequently, a partition is typically stored across multiple SSTable files So, if a row is not in memtable, a read of the row needs look-up in all the SSTable files. This is why read in Cassandra is much slower than write.

Compaction

To improve read performance as well as to utilize disk space, Cassandra periodically does compaction to create & use new consolidated SSTable files instead of multiple old SSTables.

[2]You can configure two types of compaction to run periodically: SizeTieredCompactionStrategy and LeveledCompactionStrategy.

- SizeTieredCompactionStrategy is designed for write-intensive workloads
- LeveledCompactionStrategy for read-intensive workloads

For more information about compaction strategies, see When to Use Leveled Compaction and Leveled Compaction in Apache Cassandra.

References

1. http://www.datastax.com/documentation/cassandra/2.0/cassandra/dml/dml_stores_data_c.html
2. http://www.datastax.com/documentation/cassandra/2.0/cassandra/dml/dml_write_path_c.html

Indexing

[1]An index (former name: secondary index) provides means to access data in Cassandra using non-primary key fields other than the partition key. The benefit is fast, efficient lookup of data matching a given condition. Actually, if there is no index on a normal column, it is even not allowed to conditionally query by the column.

An index indexes column values in a separate, hidden column family (table) from the one that contains the values being indexed. The data of an index is local only, which means it will not be replicated to other nodes. This also means, for data query by indexed column, the requests has to be forwarded to all the nodes, waiting for all the resonses, and then the results are merged and returned. So if you have many nodes, the query response slows down as more machines are added to the cluster.

Caution:

By the current version (2.0.7) of Apache Cassandra, you could only query by an indexed column with equality comparison condition. Range select or order-by by an indexed column is not supported. The reason is the keys stored in the hidden column family are unsorted.

When to Use An Index?

[2]Cassandra's built-in indexes are best on a table having many rows that contain the indexed value. The more unique values that exist in a particular column, the more overhead you will have for querying and maintaining the index. For example, suppose you had a playlists table with a billion songs and wanted to look up songs by the artist. Many songs will share the same column value for artist. The artist column is a good candidate for an index.

When Not to Use An Index?

[2]Do not use an index in these situations:

- On high-cardinality columns because you then query a huge volume of records for a small number of results
- In tables that use a counter column
- On a frequently updated or deleted column
- To look for a row in a large partition unless narrowly queried

References

1. http://www.datastax.com/documentation/cql/3.1/cql/ddl/ddl_primary_index_c.html
2. http://www.datastax.com/documentation/cql/3.1/cql/ddl/ddl_when_use_index_c.html

Data Replication

In this chapter, we will introduce how data is distributed & replicated in Cassandra nodes.

[1]Cassandra is designed as a peer-to-peer system that makes copies of the data and distributes the copies among a group of nodes.

Data is organized by column family (table). A row of data is identified by the partition key of the row. Cassandra uses the consistent hashing mechanism to distribute data across a cluster. Each cluster has a partitioner configured for hashing each partition key. The hash value of each partition key determines which node the data of the row should be stored on.

Copies of rows are called replicas. When data is first written, it is also referred to as the first replica of the data.

References

1. http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architectureDataDistributeAbout_c.html

Partitioners

[1]A partitioner determines how data is distributed across the nodes in the cluster. Basically, a partitioner is a hash function for computing the token (it's hash) of a partition key. Each row of data is uniquely identified by a partition key and distributed across the cluster by the value of the token.

The partitioner configuration is a global configuration for the entire cluster. The Murmur3Partitioner is the default partitioning strategy for new Cassandra clusters and the right choice for new clusters in almost all cases.

Cassandra offers the following partitioners:

- Murmur3Partitioner (default): uniformly distributes data across the cluster based on MurmurHash hash values.
- RandomPartitioner: uniformly distributes data across the cluster based on MD5 hash values.
- ByteOrderedPartitioner: keeps an ordered distribution of data lexically by key bytes

Both the Murmur3Partitioner and RandomPartitioner uses tokens to help assign equal portions of data to each node and evenly distribute data from all the column families (tables) throughout the ring.

[2]ByteOrderedPartitioner is for ordered partitioning. It orders rows lexically by key bytes. Only when using ordered partitioner, it allows ordered row scan by partition key. This means you can scan rows as though you were moving a cursor through kind of a traditional primary index in RDBMS. For example, if your application has user names as the partition key, you can scan rows for users whose names fall between Jake and Joe. This type of query is not possible using randomly partitioned partition keys because the keys are stored in the order of their hash values (not sequentially).

Although having the capability to do range scans on rows sounds like a desirable feature of ordered partitioners, **using an ordered partitioner is not recommended for the following reasons:**

- Difficult load balancing
- Sequential writes can cause hot spots
- Uneven load balancing for multiple tables

References

1. http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architecturePartitionerAbout_c.html#concept_ds_dwv_npf_fk
2. http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architecturePartitionerBOP_c.html

Replication Strategies

[1]Cassandra stores replicas on multiple nodes to ensure reliability and fault tolerance. A replication strategy determines the nodes where replicas are placed.

The total number of replicas across the cluster is referred to as the replication factor. A replication factor of 1 means that there is only one copy of each row on one node. A replication factor of 2 means two copies of each row, where each copy is on a different node. All replicas are equally important; there is no primary or master replica. As a general rule, the replication factor should not exceed the number of nodes in the cluster. However, you can increase the replication factor and then add the desired number of nodes later. When replication factor exceeds the number of nodes, writes are rejected, but reads are served as long as the desired consistency level can be met.

Two replication strategies are available:

- SimpleStrategy: Use for a single data center only. If you ever intend more than one data center, use the NetworkTopologyStrategy
- NetworkTopologyStrategy: Highly recommended for most deployments because it is much easier to expand to multiple data centers when required by future expansion, it specifies how many replicas you want in each data center

There are the two primary considerations when deciding how many replicas to configure in each data center:

- Being able to satisfy reads locally without incurring cross data-center latency
- Failure scenarios

The two most common ways to configure multiple data center clusters are:

- Two replicas in each data center: This configuration tolerates the failure of a single node per replication group and still allows local reads at a consistency level of ONE
- Three replicas in each data center: This configuration tolerates either the failure of a one node per replication group at a strong consistency level of LOCAL_QUORUM or multiple node failures per data center using consistency level ONE

References

1. http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architectureDataDistributeReplication_c.html

Turnable Consistency

[1]Consistency refers to how up-to-date and synchronized a row of Cassandra data is on all of its replicas. Cassandra extends the concept of eventual consistency by offering tunable consistency for any given read or write operation, the client application decides how consistent the requested data should be.

Write Consistency Levels

The consistency level specifies the number of replicas on which the write must succeed before returning an acknowledgment to the client application.

Level	Description	Usage
ANY	A write must be written to at least one node. If all replica nodes for the given row key are down, the write can still succeed after a hinted handoff has been written. If all replica nodes are down at write time, an ANY write is not readable until the replica nodes for that row have recovered.	Provides low latency and a guarantee that a write never fails. Delivers the lowest consistency and highest availability compared to other levels.
ALL	A write must be written to the commit log and memory table on all replica nodes in the cluster for that row.	Provides the highest consistency and the lowest availability of any other level.
EACH_QUORUM	A write must be written to the commit log and memory table on a quorum of replica nodes in all data centers.	Used in multiple data center clusters to strictly maintain consistency at the same level in each data center. For example, choose this level if you want a read to fail when a data center is down and the QUORUM cannot be reached on that data center.
LOCAL_ONE	A write must be sent to, and successfully acknowledged by, at least one replica node in the local datacenter.	In a multiple data center clusters, a consistency level of ONE is often desirable, but cross-DC traffic is not. LOCAL_ONE accomplishes this. For security and quality reasons, you can use this consistency level in an offline datacenter to prevent automatic connection to online nodes in other data centers if an offline node goes down.
LOCAL_QUORUM	A write must be written to the commit log and memory table on a quorum of replica nodes in the same data center as the coordinator node. Avoids latency of inter-data center communication.	Used in multiple data center clusters with a rack-aware replica placement strategy (NetworkTopologyStrategy) and a properly configured snitch. Fails when using SimpleStrategy. Use to maintain consistency at locally (within the single data center).
LOCAL_SERIAL	A write must be written conditionally to the commit log and memory table on a quorum of replica nodes in the same data center.	Used to achieve linearizable consistency for lightweight transactions by preventing unconditional updates.
ONE	A write must be written to the commit log and memory table of at least one replica node.	Satisfies the needs of most users because consistency requirements are not stringent. The replica node closest to the coordinator node that received the request serves the request

Caution:

Even at consistency level ONE or LOCAL_QUORUM, the write is still sent to all replicas for the written key, even replicas in other data centers. The consistency level just determines how many replicas are required to respond that they received the write.

Read Consistency Levels

Level	Description	Usage
ALL	Returns the record with the most recent timestamp after all replicas have responded. The read operation will fail if a replica does not respond.	Provides the highest consistency of all levels and the lowest availability of all levels.
EACH_QUORUM	Returns the record with the most recent timestamp once a quorum of replicas in each data center of the cluster has responded.	Same as LOCAL_QUORUM
LOCAL_SERIAL	Same as SERIAL, but confined to the data center.	Same as SERIAL
LOCAL_QUORUM	Returns the record with the most recent timestamp once a quorum of replicas in the current data center as the coordinator node has reported. Avoids latency of inter-data center communication.	Used in multiple data center clusters with a rack-aware replica placement strategy (NetworkTopologyStrategy) and a properly configured snitch. Fails when using SimpleStrategy.
LOCAL_ONE	Returns a response from the closest replica, as determined by the snitch, but only if the replica is in the local data center.	Same usage as described in the table about write consistency levels.
ONE	Returns a response from the closest replica, as determined by the snitch. By default, a read repair runs in the background to make the other replicas consistent.	Provides the highest availability of all the levels if you can tolerate a comparatively high probability of stale data being read. The replicas contacted for reads may not always have the most recent write.
QUORUM	Returns the record with the most recent timestamp after a quorum of replicas has responded regardless of data center.	Ensures strong consistency if you can tolerate some level of failure.
SERIAL	Allows reading the current (and possibly uncommitted) state of data without proposing a new addition or update. If a SERIAL read finds an uncommitted transaction in progress, it will commit the transaction as part of the read.	Lightweight transactions.
TWO	Returns the most recent data from two of the closest replicas.	Similar to ONE.
THREE	Returns the most recent data from three of the closest replicas.	Similar to TWO.

About QUORUM Level

The QUORUM level writes to the number of nodes that make up a quorum. A quorum is calculated, and then rounded down to a whole number, as follows:

(sum_of_replication_factors / 2) + 1

The sum of all the replication_factor settings for each data center is the sum_of_replication_factors.

For example, in a single data center cluster using a replication factor of 3, a quorum is 2 nodes—the cluster can tolerate 1 replica nodes down. Using a replication factor of 6, a quorum is 4—the cluster can tolerate 2 replica nodes down. In a two data center cluster where each data center has a replication factor of 3, a quorum is 4 nodes—the cluster can tolerate 2 replica nodes down. In a five data center cluster where each data center has a replication factor of 3, a quorum is 8 nodes.

If consistency is top priority, you can ensure that a read always reflects the most recent write by using the following formula:

(nodes_written + nodes_read) > replication_factor

For example, if your application is using the QUORUM consistency level for both write and read operations and you are using a replication factor of 3, then this ensures that 2 nodes are always written and 2 nodes are always read. The combination of nodes written and read (4) being greater than the replication factor (3) ensures strong read consistency.

References

1. http://www.datastax.com/documentation/cassandra/2.0/cassandra/dml/dml_config_consistency_c.html

Concurrency Control

Cassandra does not use RDBMS ACID transactions with rollback or locking mechanisms, but instead offers atomic, isolated, and durable transactions with eventual/tunable consistency that lets the user decide how strong or eventual they want each transaction's consistency to be.

Lightweight Transactions

[1]While durable transactions with eventual/tunable consistency is quite satisfactory for many use cases, situations do arise where more is needed. Lightweight transactions, also known as compare and set, that use linearizable consistency can probably fulfill those needs.

For example, if a user wants to ensure an insert they are about to make into a new accounts table is unique for a new customer, they would use the IF NOT EXISTS clause:

```
INSERT INTO customer_account (customerID, customer_email)
   VALUES ('LauraS', 'lauras@gmail.com')
   IF NOT EXISTS;
```

Modifications via UPDATE can also make use of the IF clause by comparing one or more columns to various values:

```
UPDATE customer_account
   SET customer_email='laurass@gmail.com'
   IF customer_email='lauras@gmail.com';
```

Behind the scenes, Cassandra is making four round trips between a node proposing a lightweight transaction and any needed replicas in the cluster to ensure proper execution so performance is affected. Consequently, reserve lightweight transactions for those situations where they are absolutely necessary; Cassandra's normal eventual consistency can be used for everything else.

A SERIAL consistency level allows reading the current (and possibly uncommitted) state of data without proposing a new addition or update. If a SERIAL read finds an uncommitted transaction in progress, it will commit it as part of the read.

Atomicity

[2]In Cassandra, a write is atomic at the row-level, meaning inserting or updating columns in a row is treated as one write operation. Cassandra does not support transactions in the sense of bundling multiple row updates into one all-or-nothing operation. Nor does it roll back when a write succeeds on one replica, but fails on other replicas. It is possible in Cassandra to have a write operation report a failure to the client, but still actually persist the write to a replica.

For example, if using a write consistency level of QUORUM with a replication factor of 3, Cassandra will replicate the write to all nodes in the cluster and wait for acknowledgement from two nodes. If the write fails on one of the nodes but succeeds on the other, Cassandra reports a failure to replicate the write on that node. However, the replicated write that succeeds on the other node is not automatically rolled back.

Cassandra uses timestamps to determine the most recent update to a column. The timestamp is provided by the client application. The latest timestamp always wins when requesting data, so if multiple client sessions update the same columns in a row concurrently, the most recent update is the one that will eventually persist.

Durability

[3]Writes in Cassandra are durable. All writes to a replica node are recorded both in memory and in a commit log on disk before they are acknowledged as a success. If a crash or server failure occurs before the memory tables are flushed to

disk, the commit log is replayed on restart to recover any lost writes. In addition to the local durability (data immediately written to disk), the replication of data on other nodes strengthens durability.

References

1. http://www.datastax.com/documentation/cassandra/2.0/cassandra/dml/dml_ltwt_transaction_c.html
2. http://www.datastax.com/documentation/cassandra/2.0/cassandra/dml/dml_atomicity_c.html
3. http://www.datastax.com/documentation/cassandra/2.0/cassandra/dml/dml_durability_c.html

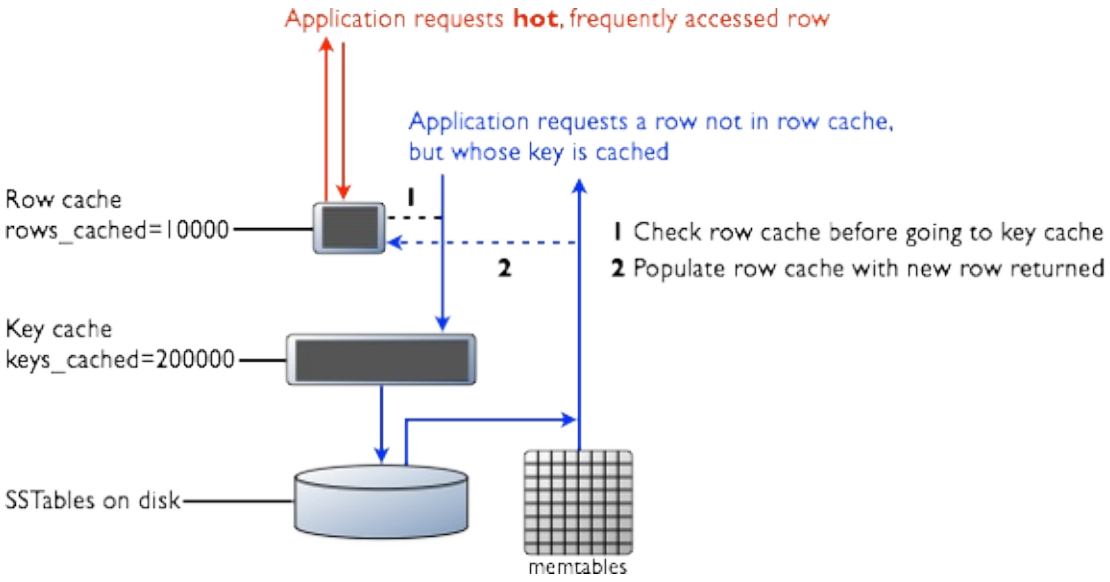
Data Caching

[1]Cassandra includes integrated caching and distributes cache data around the cluster for you. The integrated cache solves the cold start problem by virtue of saving your cache to disk periodically and being able to read contents back in when it restarts. So you never have to start with a cold cache.

There are two layers of cache:

- Partition key cache
- Row cache

How Does Caching Work?



[2]One read operation hits the row cache, returning the requested row without a disk seek. The other read operation requests a row that is not present in the row cache but is present in the partition key cache. After accessing the row in the SSTable, the system returns the data and populates the row cache with this read operation.

Tips for Efficient Cache Use

[3]Some tips for efficient cache use are:

- Store lower-demand data or data with extremely long rows in a table with minimal or no caching.
- Deploy a large number of Cassandra nodes under a relatively light load per node.
- Logically separate heavily-read data into discrete tables.

References

1. http://www.datastax.com/documentation/cassandra/2.0/cassandra/operations/ops_configuring_caches_c.html
2. http://www.datastax.com/documentation/cassandra/2.0/cassandra/operations/ops_how_cache_works_c.html
3. http://www.datastax.com/documentation/cassandra/2.0/cassandra/operations/ops_cache_tips_c.html

Client Requests

[1]Client read or write requests can go to any node in the cluster because all nodes in Cassandra are peers.

In this chapter, we will describe how a client request connect to a node and how nodes communicate with each other.

Reference

1. http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architectureClientRequestsAbout_c.html

Which Node to Connect

Which node to connect depends on the configuration of the client side driver.

For example, if you are using the Datastax drivers, there are two main configuration at client side which controls which node to talk to:

- Contact points
- Load balancing policies

Contact Points

Contact points setting is a list of one or many node address. When creating a Cluster instance at client side, the driver tries to connect to the nodes specified in the "contact points" in order. If it fails to connect to the first node, try the next ... As long as one node is successfully connected, it does not try to connect the other nodes anymore.

Why it is not necessary to connect to all the contact points is because, each node contains the metadata of all the other nodes, meaning as long as one is connected, the driver could get infomation of all the nodes in the cluster. The driver will then use the metadata of the entire cluster got from the connected node to create the connection pool. This also means, it is not necessary to set address of all your nodes to the contact points setting. The best practice is to set the nodes which responds the fastest to the client as contact points.

Load Balancing Policies

By default, a client side Cluster instance manages connections to all the nodes in the cluster and ramdonly connect to one node for any client request, which might not be performant enough, especially when you have multiple data centers.

For example, if you have a Cluster consists of nodes in two data centers, one at China and one at US. If your client is at China, you don't want to connect to a US node, because it might be too slow.

"Load balancing policies" setting on a Cluster instance determines the strategy of assigning connection of nodes to client requests.

One of the most commonly used build-in load balancing policy is the DCAwareLoadBalancePolicy, which could make the Cluster only assign connections of the nodes in the specified data center to any client requests.

You are also allowed to implement your own custom load balanacing policies.

Coodinator

[1]When a client connects to a node and issues a read or write request, that node serves as the coordinator for that particular client operation.

The job of the coordinator is to act as a proxy between the client application and the nodes (or replicas) that own the data being requested. The coordinator determines which nodes in the ring should get the request based on the cluster configured partitioner and replication strategy.

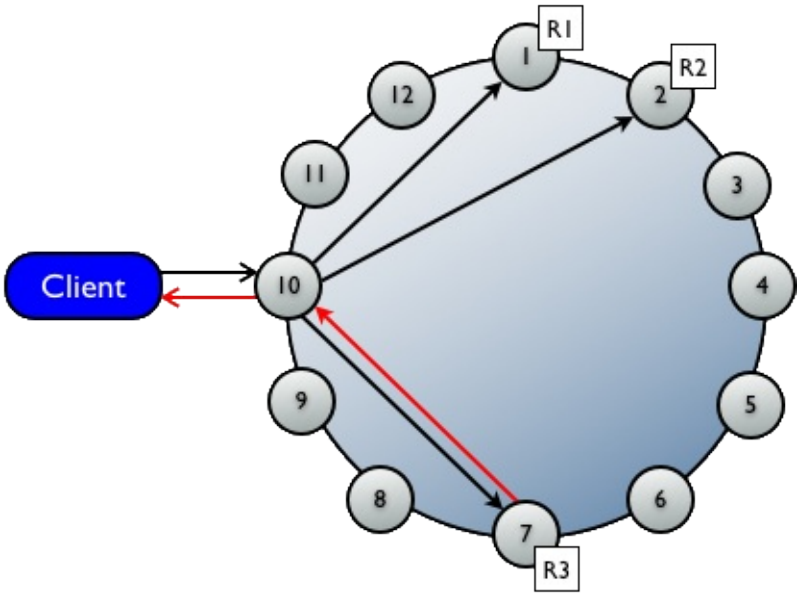
Reference

1. http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architectureClientRequestsAbout_c.html

Write Requests

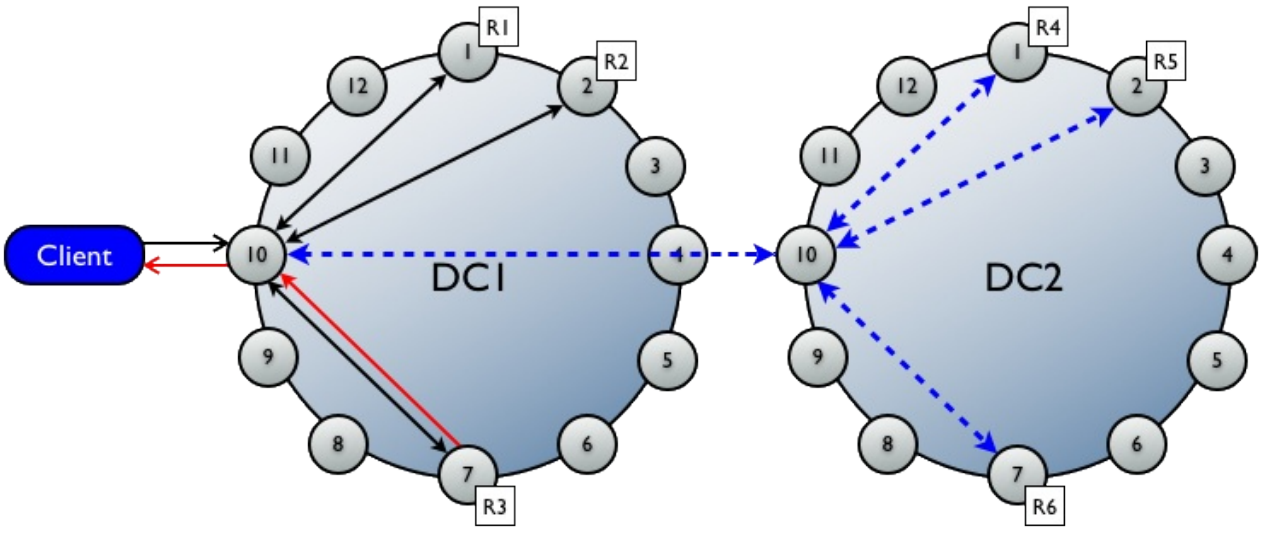
[1]For nodes in the same local data center of the coordinator, the coordinator sends a write request to all replicas that own the row being written. As long as all replica nodes are up and available, they will get the write regardless of the consistency level specified by the client. The write consistency level determines how many replica nodes must respond with a success acknowledgment in order for the write to be considered successful.

For example, in a single data center 10 node cluster with a replication factor of 3, an incoming write will go to all 3 nodes that own the requested row. If the write consistency level specified by the client is ONE, the first node to complete the write responds back to the coordinator, which then proxies the success message back to the client. When a node writes and responds, that means it has written to the commit log and puts the mutation into a memtable.



[2]In multiple data center deployments, Cassandra optimizes write performance by choosing one coordinator node in each remote data center to handle the requests to replicas within that data center. The coordinator node contacted by the client application only needs to forward the write request to one node in each remote data center.

If using a consistency level of ONE or LOCAL_QUORUM, only the nodes in the same data center as the coordinator node must respond to the client request in order for the request to succeed. This way, geographical latency does not impact client request response times.



Reference

1. http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architectureClientRequestsWrite.html

2. http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architectureClientRequestsMultiDCWrites_c.html

Read Requests

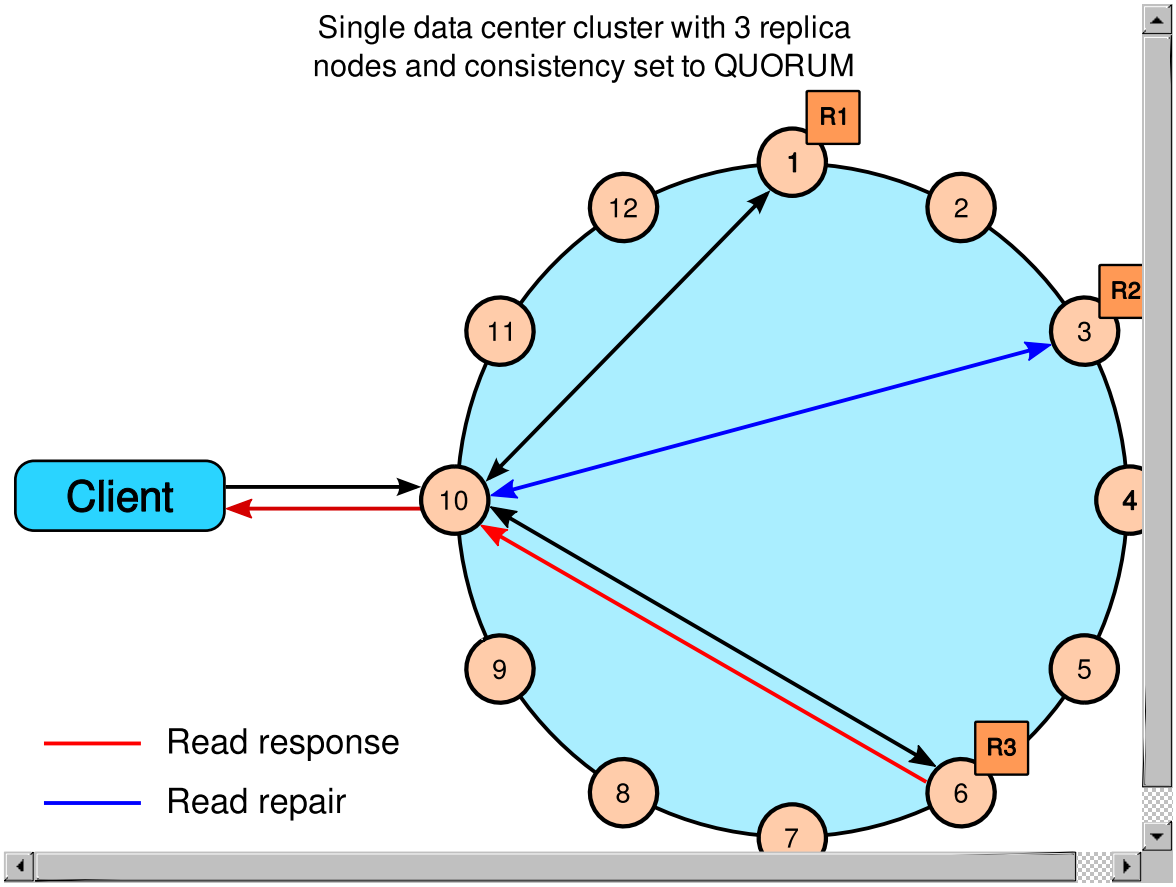
The number of replicas contacted by a client read request is determined by the consistency level specified by the client. The coordinator sends these requests to the replicas that are currently responding the fastest. The nodes contacted respond with the requested data; if multiple nodes are contacted, the rows from each replica are compared in memory to see if they are consistent. If they are not, then the replica that has the most recent data (based on the timestamp) is used by the coordinator to forward the result back to the client.

To ensure that all replicas have the most recent version of frequently-read data, the coordinator also contacts and compares the data from all the remaining replicas that own the row in the background. If the replicas are inconsistent, the coordinator issues writes to the out-of-date replicas to update the row to the most recent values.

Examples

A single data center cluster with a consistency level of QUORUM

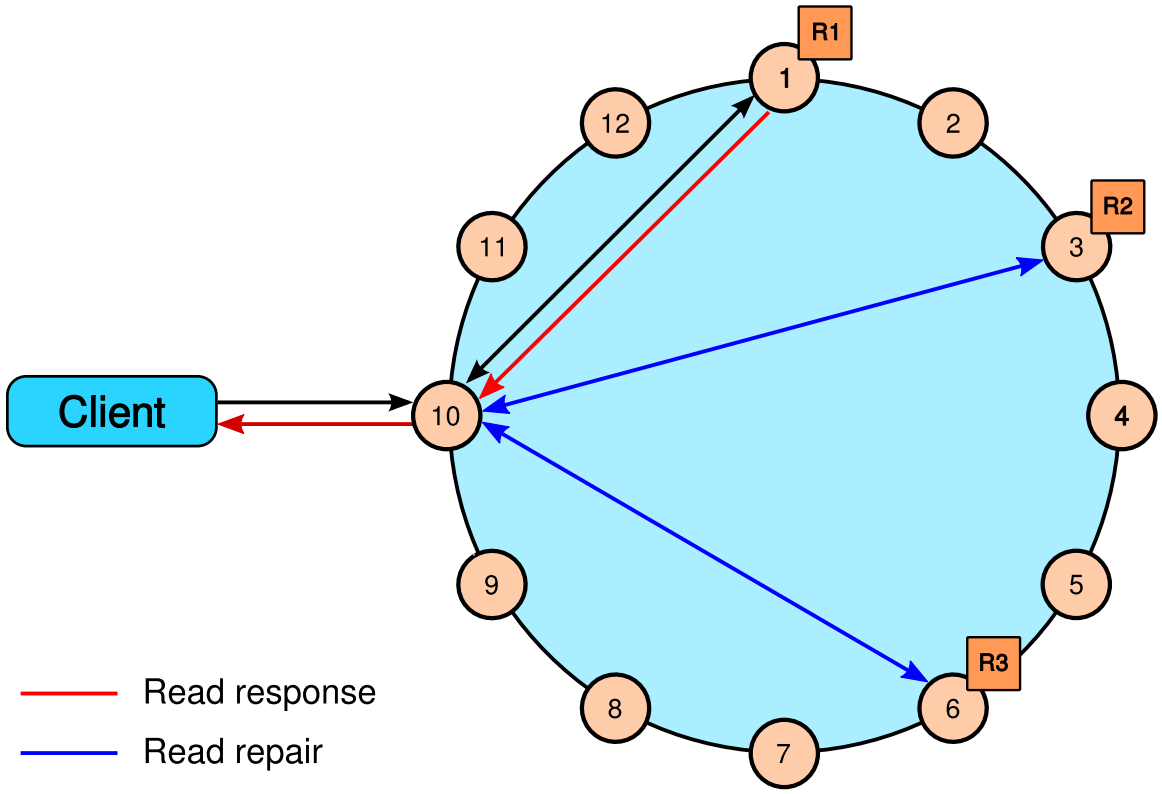
In a single data center cluster with a replication factor of 3, and a read consistency level of QUORUM, 2 of the 3 replicas for the given row must respond to fulfill the read request. If the contacted replicas have different versions of the row, the replica with the most recent version will return the requested data. In the background, the third replica is checked for consistency with the first two, and if needed, a read repair is initiated for the out-of-date replicas.



A single data center cluster with a consistency level of ONE

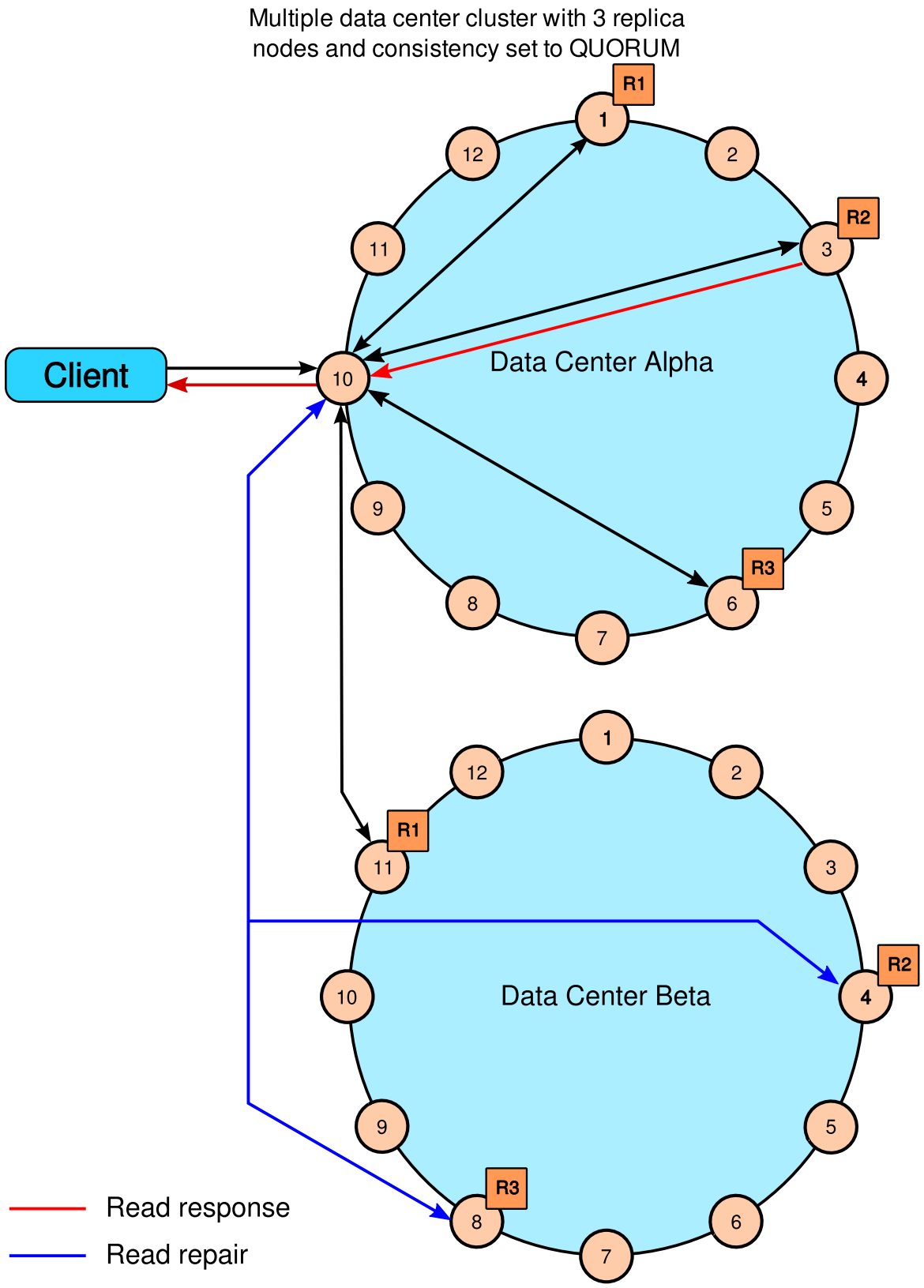
In a single data center cluster with a replication factor of 3, and a read consistency level of ONE, the closest replica for the given row is contacted to fulfill the read request. In the background a read repair is potentially initiated, based on the read_repair_chance setting of the table, for the other replicas.

Single data center cluster with 3 replica nodes and consistency set to ONE



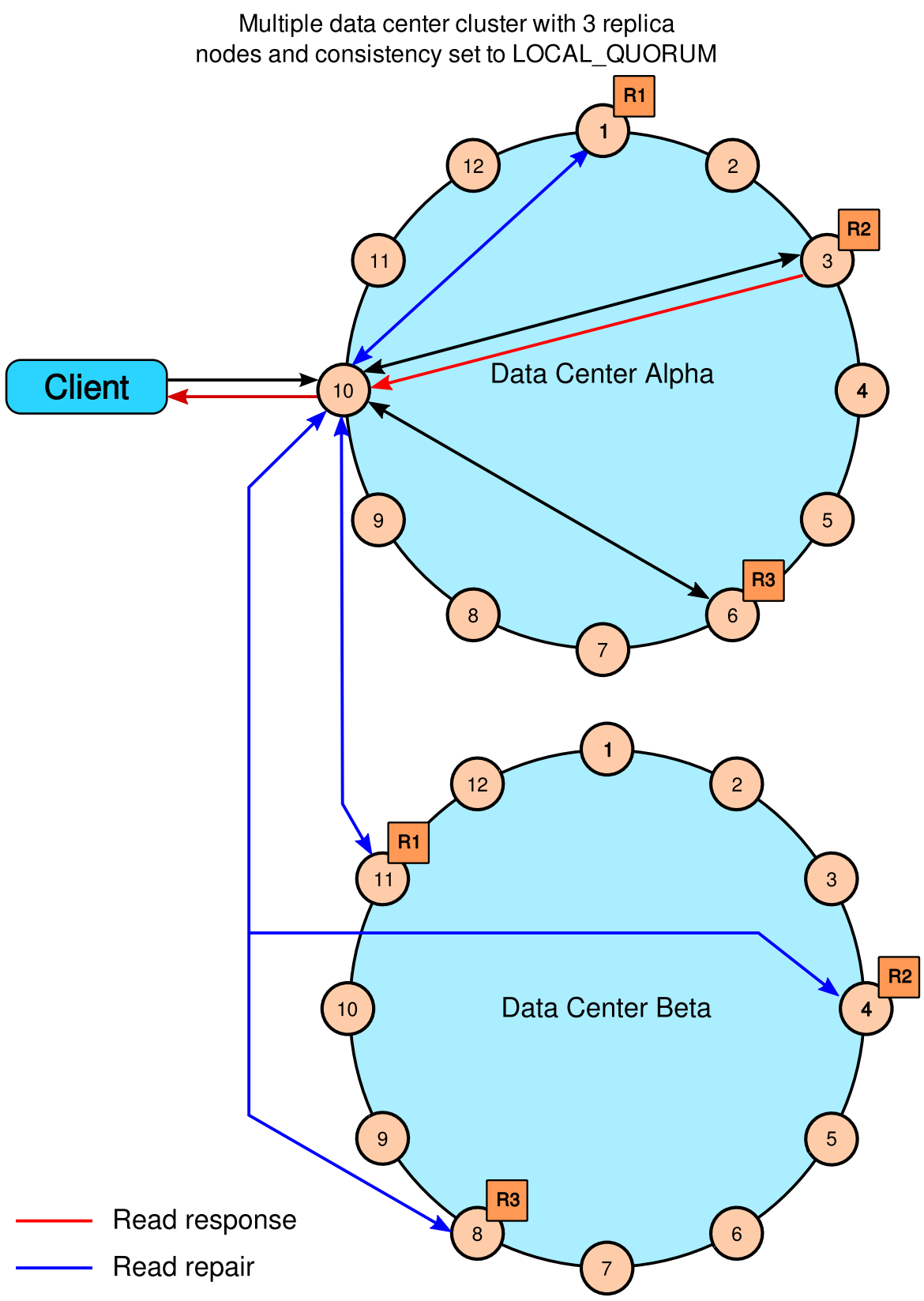
A two data center cluster with a consistency level of QUORUM

In a two data center cluster with a replication factor of 3, and a read consistency of QUORUM, 4 replicas for the given row must resond to fulfill the read request. The 4 replicas can be from any data center. In the background, the remaining replicas are checked for consistency with the first four, and if needed, a read repair is initiated for the out-of-date replicas.



A two data center cluster with a consistency level of LOCAL_QUORUM

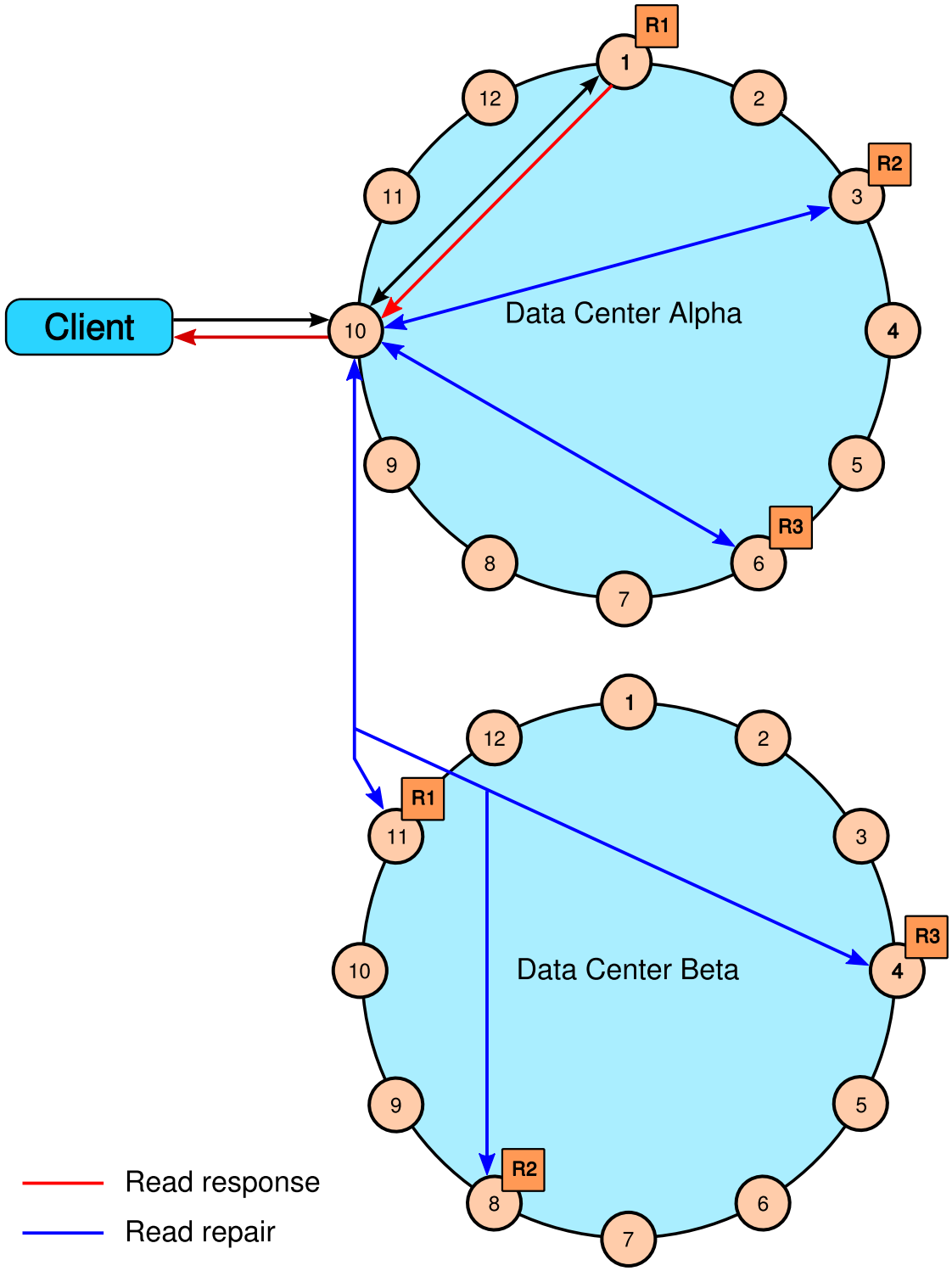
In a multiple data center cluster with a replication factor of 3, and a read consistency of LOCAL_QUORUM, 2 replicas in the same data center as the coordinator node for the given row must respond to fulfill the read request. In the background, the remaining replicas are checked for consistency with the first 2, and if needed, a read repair is initiated for the out-of-date replicas.



A two data center cluster with a consistency level of ONE

In a multiple data center cluster with a replication factor of 3, and a read consistency of ONE, the closest replica for the given row, regardless of data center, is contacted to fulfill the read request. In the background a read repair is potentially initiated, based on the read_repair_chance setting of the table, for the other replicas.

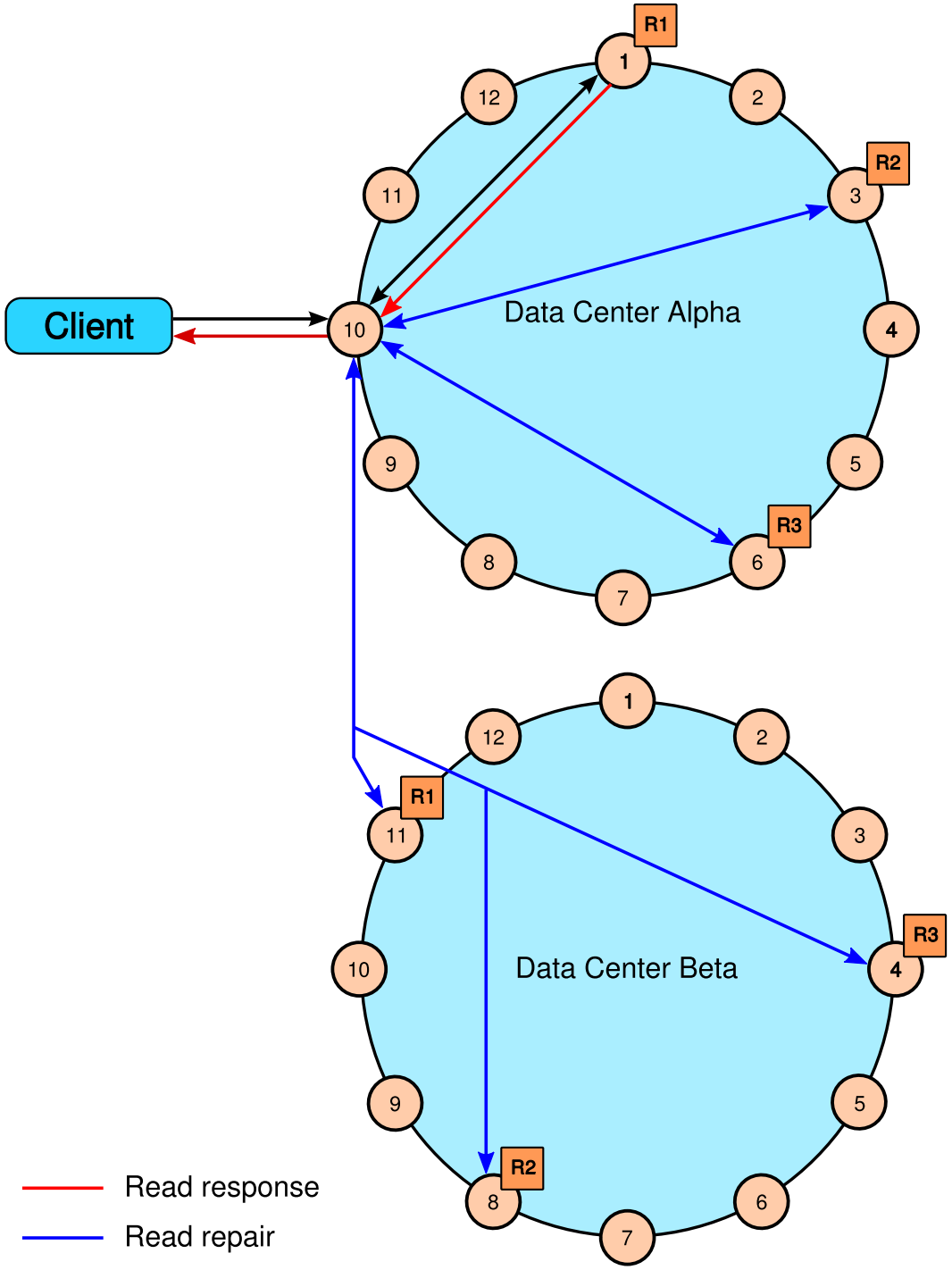
Multiple data center cluster with 3 replica nodes and consistency set to ONE



A two data center cluster with a consistency level of LOCAL_ONE

In a multiple data center cluster with a replication factor of 3, and a read consistency of LOCAL_ONE, the closest replica for the given row in the same data center as the coodindator node is contacted to fulfill the read request. In the background a read repair is potentially initiated, based on the read_repair_chance setting of the table, for the other replicas.

Multiple data center cluster with 3 replica nodes and consistency set to LOCAL_ONE



Rapid read protection using speculative_retry

Rapid read protection allows Cassandra to still deliver read requests when the originally selected replica nodes are either down or taking too long to respond. If the table has been configured with the speculative_retry property, the coordinator node for the read request will retry the request with another replica node if the original replica node exceeds a configurable timeout value to complete the read request.

Recovering from replica node failure with rapid read protection