1 预研

1.1 使用场景

- **DAU 50W** 时有 **2W QPS**，写入操作远大于读写；
- 目标 **1000W DAU**，预计 **50 - 100W QPS**；

1.2 横向比较

数据库	模式	特点	驱动
MongoDB	Document-based	文档的形式存储数据，不支持事务和表连接；动态查询；支持分片；二进制数据处理效率高；	主流语言
HBase	Column-based	分布集群，硬件要求低；易扩展，支持自动分片；支持PB数据集，百万级QPS；写入快；	Java & Thrift
Cassandra	Column-based	节点无特征，同级平等；易部署，易扩展，线上动态扩展；写入快；	主流语言

1.3 技术选型

- 写入远大于读写，适合选定 **Column-based** 类 **NoSQL**；
- **HBase** 需要 **Hadoop** 和 **ZooKeeper** 支持，部署相对复杂，但技术成熟，可支持 **PB** 级数据和百万 **QPS**；
- **Cassandra** 部署简单，可线上扩展，横向扩展无限制，更容易使用；
- **HBase** 原生驱动仅支持 **Java**，**PHP** 需要通过 **Thrift**；
- **Cassandra** 已实现主流语言支持，包括 **PHP**；
- **HBase** 和 **Cassandra** 均有大量知名项目应用实例，用户多社区成熟；

考虑到 **Cassandra** 的部署和扩展都比 **HBase** 成本低，优先考察 **Cassandra**；

2 Cassandra 优点

- 保证 **AP**，通过调节 **replication factor** 和 **consistency level** 来满足 **C**；
- 去中心架构，节点平等，内建可定制 **replication** 机制；
- 集群间自动化数据分发，添加和删除节点数据自动整理；
- 节点故障自适应处理，排除故障后自动恢复，基本杜绝单点故障；
- **Replication** 可跨区域跨机房，强容灾，易构建全球数据中心；
- 具有线性扩展性，可简单的再现添加新节点扩展集群；

3 Cassandra 存储模式

3.1 Memtable

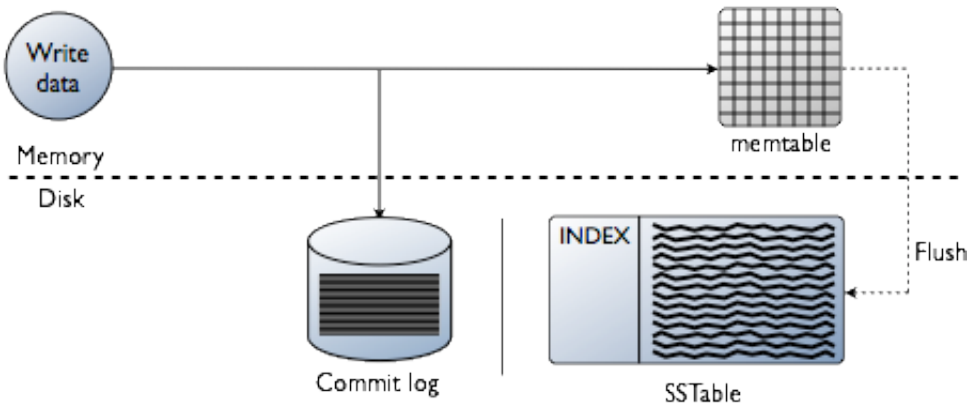
写入的数据会保存在 **Memtable** 中。当 **Memtable** 保存的数据超过配置的阈值，**Cassandra** 会启动数据迁移流程。**Memtable** 中的数据会按 **partition key** 进行排序，保存到 **SSTable**，然后清空 **Memtable** 和 **CommitLog**。

3.2 CommitLog

数据在被写入 **Memtable** 的同时，会同时追加到磁盘上的 **CommitLog** 末尾。发生硬件故障等异常情况时，可以使用 **CommitLog** 来重建 **Memtable** 数据。

3.3 SSTable

SSTables 是不可变的，**Memtable** 被刷新和保存为一个 **SSTable** 文件之后，**SSTable** 不会被再次写入，一个分区一般会被保存为多个 **SSTable** 文件。当需要读取的一行数据不在当前 **Memtable** 里时，需要遍历所有的 **SSTable**，所以 **Cassandra** 中读比写慢得多。**Cassandra** 会周期性地合并多个 **SSTable** 文件中相同 **partition key** 的数据，以提升读的性能和释放磁盘空间。



3.4 从 SSTable 读取数据

3.4.1 Bloom Filter

Bloom Filter 用于判断 **SSTable** 是否包含当前查询所请求数据。如包含则读取 **SSTable**，不包含则跳过并查询下一个 **SSTable**。

3.4.2 Partition Key Cache

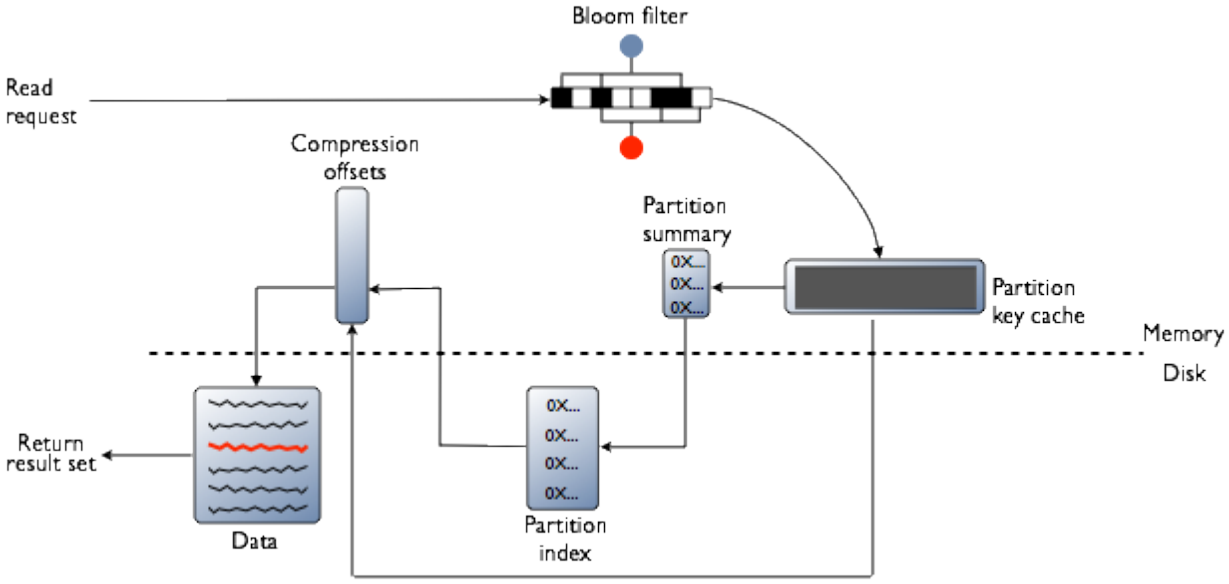
Partition Key Cache 用于缓存查询过的数据的索引项。如果查询的数据位于 **Partition Key Cache** 中，则可通过索引项在 **Compression Offset Map** 中查询数据在 **SSTable** 中的地址，并从该地址返回所需数据。

3.4.3 Partition Summary

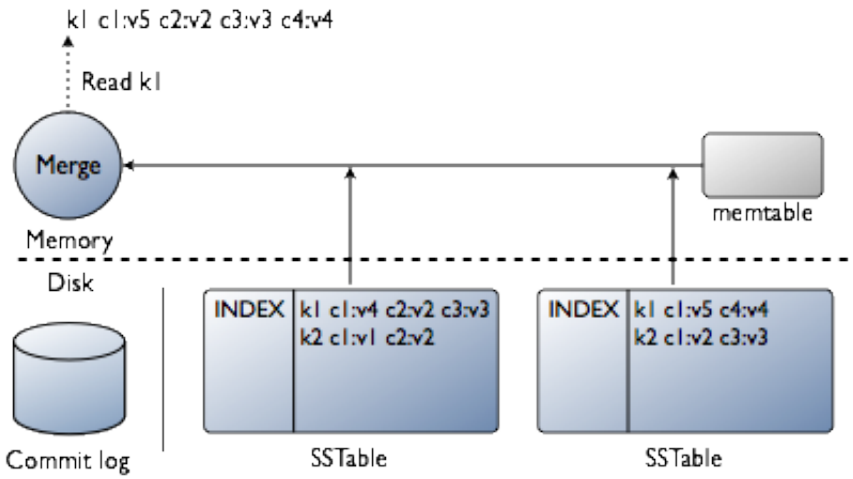
Partition Summary 用于存储所有数据的索引项。如果查询的数据不在 **Partition Key Cache** 中，则会在 **Partition Summary** 找到索引项的大致位置，并从该位置开始搜索 **Partition Index**，直至找到数据的索引项。然后通过索引项从 **Compression Offset Map** 获取到数据的地址，并从该地址返回数据。

3.4.4 Compression Offset Map

Compression Offset Map 用于保存数据索引项和数据地址的对应关系。



一个分区一般会被保存为多个 **SSTable** 文件，所以在从 **SSTable** 读取数据时，会遍历所有存在查询的分区数据的 **SSTable** 文件，并将结果合并后返回。



4 Cassandra 数据结构

4.1 Column-based

传统的关系型数据库中，数据按行被顺序存储，而查询语句是按列的值进行搜索的。所以搜索时需要不断进行偏移计算和跳转，大大降低了关系型数据库的性能。

而在 **Column-based** 数据库中，数据是按列存储的，同一列（仅限索引列）的数据将被连续的存放在一起，遍历效率大大提高。由于仅索引列的数据被按列连续存放，所以 **Cassandra** 的查询语句中，仅限使用索引列作为查询条件。

列的主要属性是 **name**、 **value** 和 **timestamp**。其中 **timestamp** 用于数据的版本控制，**Cassandra** 用 **timestamp** 确定来自多个节点的数据中那些样本是有效的最新数据。 列可以只有列名没有列值，每一行的列的数量最多允许多达 20 亿。

4.2 数据模型

基于列族（**Column Family**）的四维或五维模型。

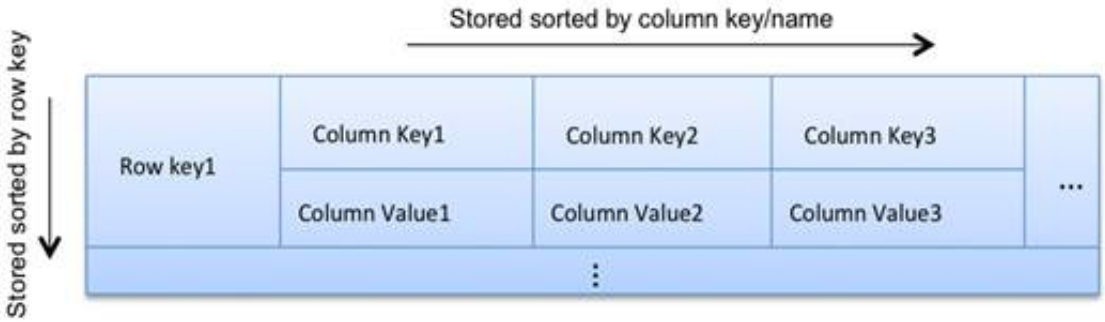
- **keyspaces** （类似关系型数据库里的 **database**）；
- **column families**（类似关系型数据库里的 **table**）；
- 主键（**keys**）；
- 列（**columns**）；

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

4.3 数据结构

每一个 **column family** 相当于一个多层嵌套的排序散列表（**Nested sorted map**）。

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



4.4 CQL

Cassandra 使用CQL进行数据查询，语法类似 **SQL**。

4.4.1 CQL特点

- **keyspace**，**column** 和 **table** 的名称忽略大小写，除非用双引号括起来；

- 数据结构丰富；
- 语法类似 **SQL**；

4.4.2 CQL 数据类型

CQL类型	常量类型	说明
ascii	strings	US-ASCII 字符串
bigint	integers	64 位有符号 long
blob	blobs	任意的 16 进制格式的 bytes
boolean	booleans	true 或 false
counter	integers	分布式 counter 值 (64 位 long)
decimal	integers, floats	可变精度浮点数
double	integers, floats	64 位 IEEE-754 浮点数
float	integers, floats	32 位 IEEE-754 浮点数
inet	strings	IPv4 或 IPv6 格式的IP地址
int	integers	32 位有符号整数
list	n/a	有序集合, T 可以是任意非集合 CQL 数据类型，例如：int, text 等
map	n/a	哈希表, K 和 V 可以是任意非集合 CQL 数据类型，例如：int, text 等
set	n/a	无序集合, T 可以是任意非集合 CQL 数据类型，例如：int, text 等
text	strings	UTF-8 编码字符串
timestamp	integers, strings	日期+时间
uuid	uuids	标准UUID格式
timeuuid	uuids	Type 1 UUID
varchar	strings	UTF-8 编码字符串
varint	integers	任意精度整数

4.4.3 CQL语法

详见 CQL for Apache Cassandra

5 Cassandra 主键

5.1 主键构成

在创建 **column family** 时，可用 `PRIMARY KEY` 指定主键，主键对应的数据要求唯一。

```
CREATE TABLE users (
    uid int,
    age int,
    name varchar,
    data text,
    PRIMARY KEY((uid, age), name)
);
```

5.1.1 Partition Key

`PRIMARY KEY` 指定的第一个域，称为 **Partition Key**，**Partition Key** 将决定数据被存储在集群中的哪些节点上。**Partition Key** 可以由一个或多个列构成，上例中 **Partition Key** 是由 **uid** 和 **age** 一起组成。

5.1.2 Clustering Key

`PRIMARY KEY` 指定了多个域时，**Partition Key** 以外的域称为 **Clustering Key**。如果 `PRIMARY KEY` 仅指定了 **Partition Key** ，则 **column family** 没有 **Clustering Key**。**Clustering Key** 将决定数据在 **Partition** 内部的排序。

5.2 条件查询和排序

- 仅支持对主键字段进行条件查询，查询条件中必须使用 `=` 或 `IN` 明确指定 **Partition Key** 所有列的值；

```
cqlsh:test> SELECT * FROM users WHERE age = 16;
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute this que
ry as it might involve data filtering and thus may have unpredictable performance. If you wan
t to execute this query despite the performance unpredictability, use ALLOW FILTERING"
cqlsh:test> SELECT * FROM users WHERE uid > 1;
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute this que
ry as it might involve data filtering and thus may have unpredictable performance. If you wan
t to execute this query despite the performance unpredictability, use ALLOW FILTERING"
cqlsh:test> SELECT * FROM users WHERE uid = 1;
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute this que
ry as it might involve data filtering and thus may have unpredictable performance. If you wan
t to execute this query despite the performance unpredictability, use ALLOW FILTERING"
cqlsh:test> SELECT * FROM users WHERE uid = 1 AND age >= 16;
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute this que
ry as it might involve data filtering and thus may have unpredictable performance. If you wan
t to execute this query despite the performance unpredictability, use ALLOW FILTERING"
cqlsh:test> SELECT * FROM users WHERE uid = 1 AND age = 16;

 uid | age | name | data
-----+-----+------+------
   1 |  16 |   t1 |

(1 rows)
cqlsh:test> SELECT * FROM users WHERE uid = 1 AND age IN (16,17);

 uid | age | name | data
-----+-----+------+------
   1 |  16 |   t1 |
   1 |  17 |   t2 |

(2 rows)
```

- 条件查询时，仅支持使用 **Clustering Key** 中的列进行范围查询和排序；
- 使用 `IN` 指定 **Partition Key** 时 **Clustering Key** 排序将不可用；
- 使用 **Clustering Key** 中的列进行范围查询和排序时，必须按照 **Clustering Key** 的顺序查询和排序；

```
cqlsh:test> SELECT * FROM users WHERE uid = 1 ORDER BY name;
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute this que
ry as it might involve data filtering and thus may have unpredictable performance. If you wan
t to execute this query despite the performance unpredictability, use ALLOW FILTERING"
cqlsh:test> SELECT * FROM users WHERE uid = 1 AND age IN (16,17) ORDER BY age;
InvalidRequest: Error from server: code=2200 [Invalid query] message="Order by is currently o
nly supported on the clustered columns of the PRIMARY KEY, got age"
cqlsh:test> SELECT * FROM users WHERE uid = 1 AND age IN (16,17) ORDER BY name;
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot page queries wit
h both ORDER BY and a IN restriction on the partition key; you must either remove the ORDER B
Y or the IN and sort client side, or disable paging for this query"
cqlsh:test> SELECT * FROM users WHERE uid = 1 AND age = 16 AND name = 't1';

 uid | age | name | data
-----+-----+------+------
   1 |  16 |   t1 |

(1 rows)
cqlsh:test> SELECT * FROM users WHERE uid = 1 AND age = 16 ORDER BY name;

 uid | age | name | data
-----+-----+------+------
   1 |  16 |   t1 |

(1 rows)
```

- 在语句末尾使用 `ALLOW FILTERING` 可以跳过查询条件限制，但是可能会导致大范围扫描，影响系统性能；

```
cqlsh:test> SELECT * FROM users WHERE uid > 1 ALLOW FILTERING;

 uid | age | name | data
-----+-----+------+------
   2 |  18 |   t2 |

(1 rows)
cqlsh:test> SELECT * FROM users WHERE name = 't1' ALLOW FILTERING;

 uid | age | name | data
-----+-----+------+------
   1 |  16 |   t1 |

(1 rows)
```

5.3 二级索引

> **Cassandra** 支持对非主键子段建立索引，称为 **Secondary Index**

5.3.1 索引特点

- 索引数据保存在本地，不会被复制到其他节点；按索引字段进行查询，请求将被转发到所有节点，并返回合并的查询结果。节点越多，索引查询会越慢；
- 索引无序存储，所以索引字段的条件查询只支持相等比较，不支持对于索引字段的范围查询和字段排序。

5.3.2 索引适用性

- 被索引的列的数据重复性越高，索引效率越高；数据唯一性越强，索引效率越低；

  > 如图书表，由于一个出版社会出版很多本书，出版社字段就适合建立索引； 如时间戳，数据唯一性很高，重复率较低，就不适合建立索引；

- 索引会影响数据写入效率，频繁更新和删除数据的表不适合建立索引；

6 Cassandra 复制和集群

6.1 复制和分发

6.1.1 Partitioner

分区器是用来计算 **partition key** 哈希值的哈希函数，决定了数据如何在集群中被分发。分区器在集群中是全局唯一的。默认分区器是 **Murmur3Partitioner**，已经可以满足绝大部分需求，一般无需更改。

Cassandra 提供了三个分区器：

1. **Murmur3Partitioner**

   > 基于 **MurmurHash** 哈希算法

2. **RandomPartitioner**

   > 基于 **MD5** 哈希算法

3. **ByteOrderedPartitioner**

   > 根据 **partition key** 的 **bytes** 进行有序分区

ByteOrderedPartitioner 对 **partition key** 进行条件范围查询，而其他两种分区器是不允许的。但并不推荐使用 **ByteOrderedPartitioner**，主要是因为：

- 数据分布不平均，负载均衡更困难；
- 容易造成热点；

6.1.2 Replication factor

Cassandra 会将数据存储在多个节点，以确保可靠性和容错性。同一份数据在各个节点上的副本称为 **replica**，每个 **replica** 是平等的，没有主次之分。数据会有多少个 **replica** 由 **replication factor** 控制，**replication factor** 在创建 **keyspace** 时设置。

例如，**replication factor** 设置为 **3**，分区器将根据 **partition key** 的值计算出 **replica** 应该被存放的 **3** 个节点，写入数据时，数据将被同时存储在这三个节点上。**replication factor** 不应超过集群的节点数量，如果设置了大于节点数量的值，写入操作将会被拒绝，但仍可正常读取。

Cassandra 内置了两种 **replication** 策略：

1. **SimpleStrategy**

   > 单数据中心模式，指定每行数据存储在当前数据中心的节点数；

   ```
   CREATE KEYSPACE test WITH replication = {'class': 'SimpleStrategy',
   'replication_factor' : 3};
   ```

2. **NetworkTopologyStrategy**

   > 多数据中心模式，分别指定每行数据存储在集群中的每个数据中心的节点数；

   ```
   CREATE KEYSPACE test WITH replication = {'class':
   'NetworkTopologyStrategy', 'DC1' : 1, 'DC2' : 3};
   ```

如果将来可能使用多个数据中心的集群，则应使用 **NetworkTopologyStrategy**，方便以后的扩展。在设置 **replication** 策略时，要优先考虑读取速度和系统可用性，避免跨数据中心读取和单点故障。

6.2 客户端连接

Cassandra 的每个节点都是对等的，客户端可以连接到集群中的任意节点执行查询请求。

6.2.1 Contact Points

Contact points 是一个集群节点 **IP** 或域名的列表。当数据驱动创建 **Cluster** 实例时，会按照顺序尝试连接**contact points** 中的节点。当成功连接到任意一个节点，则停止尝试连接后面的节点。

集群中任意一个节点都持有整个集群节点的元数据，只要成功连接到任意节点，就可以获取到整个集群的节点信息。然后，数据驱动会使用获取到的集群节点信息，连接到集群的所有节点，并构建连接池。

contact points 不需要配置集群中的所有节点，一般会配置客户端连接最快的节点地址。

```php
<?php
$cluster = Cassandra::cluster()
                ->withContactPoints('10.0.1.24', 'example.com', 'localhost')
                ->build();
$session = $cluster->connect();
```

6.2.2 Load Balancing Policies

默认情况下，客户端的 **Cluster** 实例管理了集群中所有节点的连接，并随机连接一个节点发送客户端查询请求。但在某些情况下，特别是多数据中心时，默认行为并不一定具有最有的性能。例如，对于一个有中国数据中心和美国数据中心的集群，为了优化查询速度，中国的客户端不应连接美国的节点。

目前 PHP 数据驱动支持令牌感知、数据中心感知和延迟感知等负载均衡策略。

1. **token aware** 令牌感知策略，使用与 **Cassandra** 相同的散列算法将数据查询直接路由到 **replica** 节点，避免使用 **coordinator** 进行转发。

   ```php
   <?php
   $cluster = Cassandra::cluster()
                   ->withTokenAwareRouting(true)
                   ->build();
   $session = $cluster->connect();
   ```

2. **datacenter aware** 数据中心感知策略，是一般情况下最常用的策略，仅连接到指定数据中心的节点，查询请求也只会发送到指定数据中心。对于多数据中心的查询和一致性策略，将通过本地数据中心转发。可以通过参数指定连接的数据中心名称，可使用的远程数据中心节点数量，以及本地数据中心节点不可用时能否在远程数据中心执行本地一致性的语句。

   ```php
   <?php
   $cluster = Cassandra::cluster()
                   ->withDatacenterAwareRoundRobinLoadBalancingPolicy("dc1", 2, true)
                   ->build();
   $session = $cluster->connect();
   ```

3. **latency aware** 延迟感知策略，会将请求路由到响应速度最快的节点。

```php
<?php
$cluster = Cassandra::cluster()
                ->withLatencyAwareRouting(true)
                ->build();
$session = $cluster->connect();
```

6.2.3 Coodinator

客户端连接的节点被称为 **Coodinator**。**Coodinator** 作为客户端与集群所有数据节点之间的代理，通过集群配置的分区器和复制策略，决定将客户端查询请求发送到哪些数据节点。**Coodinator** 所在的数据中心被称为本地数据中心。

6.3 可调节一致性

6.3.1 Consistency level

Cassandra 通过 **consistency level** 参数实现可调节的一致性。**consistency level** 指定了读写操作返回成功时，需要的成功完成操作的节点数量，可精确到每一个查询。

1. 在连接时设定连接的全局 **consistency level**;

```php
<?php
$cluster = Cassandra::cluster()
                ->withDefaultConsistency(Cassandra::CONSISTENCY_LOCAL_QUORUM)
                ->build();
$session = $cluster->connect();
```

2. 在查询时设置查询的 **consistency level**;

```php
<?php
$session->execute(
    'SELECT * FROM users',
    array('consistency' => Cassandra::CONSISTENCY_LOCAL_QUORUM)
);

$statement = $session->prepare('SELECT * FROM users');
$session->execute($statement, array(
    'consistency' => Cassandra::CONSISTENCY_LOCAL_QUORUM
));

$batch = new Cassandra\BatchStatement();
$batch->add("UPDATE users SET name = 'may' WHERE uid = 1");
$batch->add("UPDATE users SET name = 'tom' WHERE uid = 2");
$session->execute($batch, array(
    'consistency' => Cassandra::CONSISTENCY_LOCAL_QUORUM
));
```

consistency level 只是设定操作返回成功的条件，并不限制数据的复制和分发。不论 **consistency level** 如何设置，读写操作都会发送给分区器计算出的每个数据存储的节点。

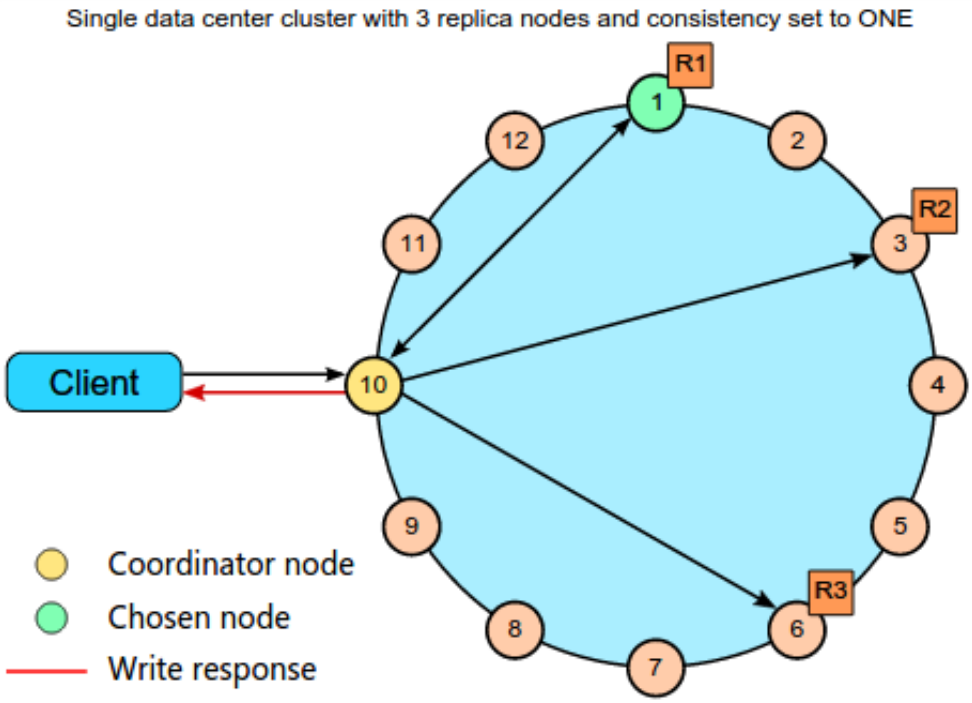
执行读取请求时，如果多个节点返回了数据，**coordinator** 会比较每一列的 **timestamp**，返回合并后的最新的数据。为了确保所有的 **replica** 节点的数据一致性，每次读操作之后，**coordinator** 会在后台同步所有其他 **replica** 节点上的该行数据，确保每个 **replica** 节点上拥有该行数据的最新版本。

6.3.2 Write Consistency Levels

级别	描述	用法
ALL	写操作必须将指定行的数据写到所有 replica 节点的 commit log 和 memtable。	相对于其他级别提供最高的一致性和最低的可用性。
EACH_QUORUM	写操作必须将指定行的数据写到每个数据中心的 quorum 数量的 replica 节点的 commit log 和 memtable。	用于多数据中心集群，严格的保证每个数据中心具有相同级别的一致性。例如，当一个数据中心挂掉但操作成功的 replica 节点数量仍会大于 QUORUM 时，应使用 EACH_QUORUM。
QUORUM	写操作必须将指定行的数据写到所有数据中心的 quorum 数量的 replica 节点的 commit log 和 memtable。	用于单个或多个数据中心集群，保证整个集群的强一致性，但有可能写入失败。

LOCAL_QUORUM	写操作必须将指定行的数据写到 coordinator 所在数据中心内 quorum 数量的 replica 节点的 commit log 和 memtable。避免跨数据中心的通信。	用于使用 NetworkTopologyStrategy 策略的多数据中心集群，保证本地数据中心的数据一致性。 也可与 SimpleStrategy 策略一起使用。
ONE	写操作必须将指定行的数据写到至少一个 replica 节点的 commit log 和 memtable。	不严格要求一致性，满足大多数用户的需求。
TWO	写操作必须将指定行的数据写到至少两个 replica 节点的 commit log 和 memtable。	和 ONE 类似。
THREE	写操作必须将指定行的数据写到至少三个 replica 节点的 commit log 和 memtable。	和 TWO 类似。
LOCAL_ONE	任何一个本地数据中心内的 replica 节点写操作成功。	对于多数据中心的情况，往往期望至少一个 replica 节点写成功，但是，又不希望有任何跨数据中心的通信。LOCAL_ONE 正好能满足这样的需求。在关闭的脱机数据中心和节点使用 LOCAL_ONE 策略，可以阻止自动连接到其他数据中心中的联机节点，确保联机数据的安全。
ANY	任意一个节点写操作已经成功。如果所有的 replica 节点都挂了，写操作还是可以在记录一个 hinted handoff 事件之后，返回成功。如果所有的 replica 节点都挂了，写入的数据，在挂掉的 replica 节点恢复之前无法读取。	最小的延时等待，并且确保写请求不会失败。提供最低的一致性和最高的可用性。

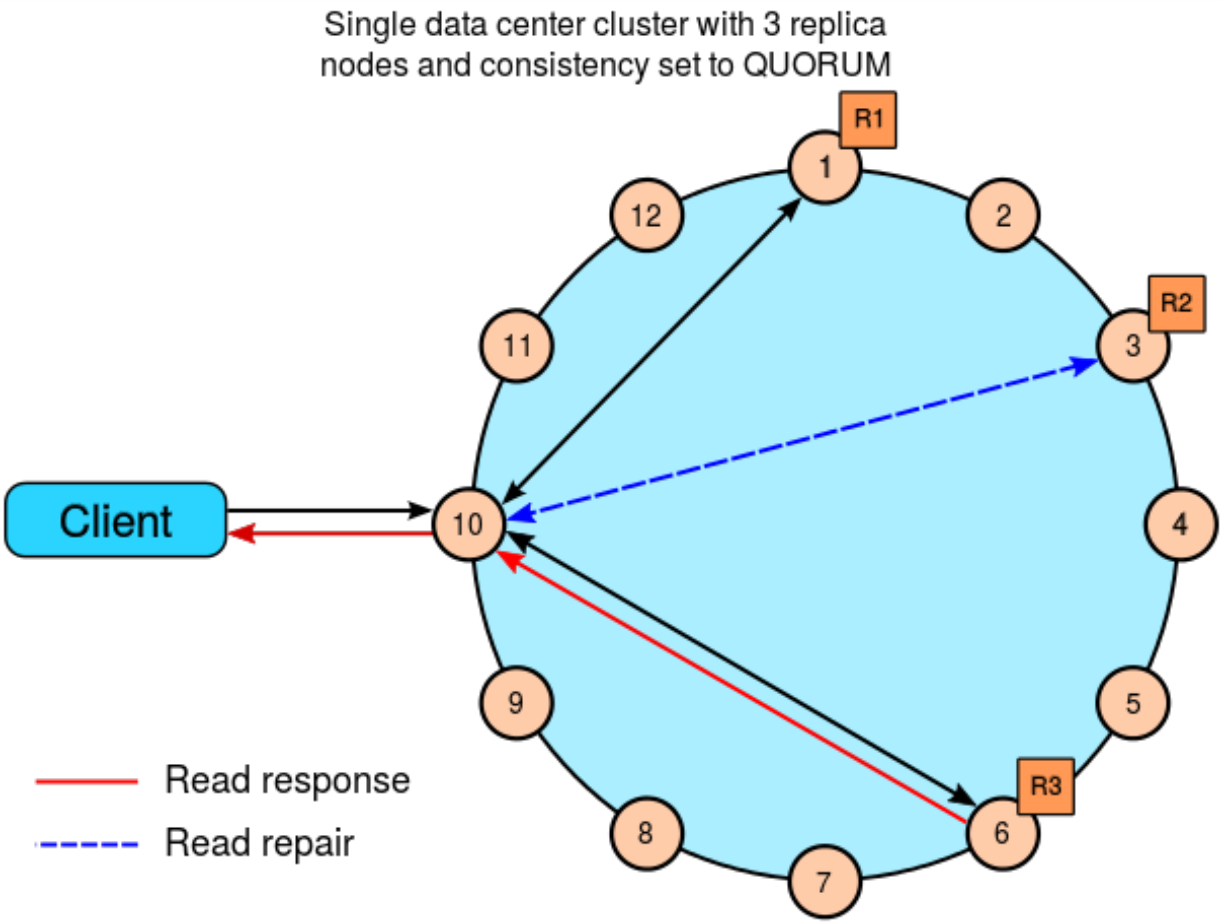
6.3.2.1 Write ONE

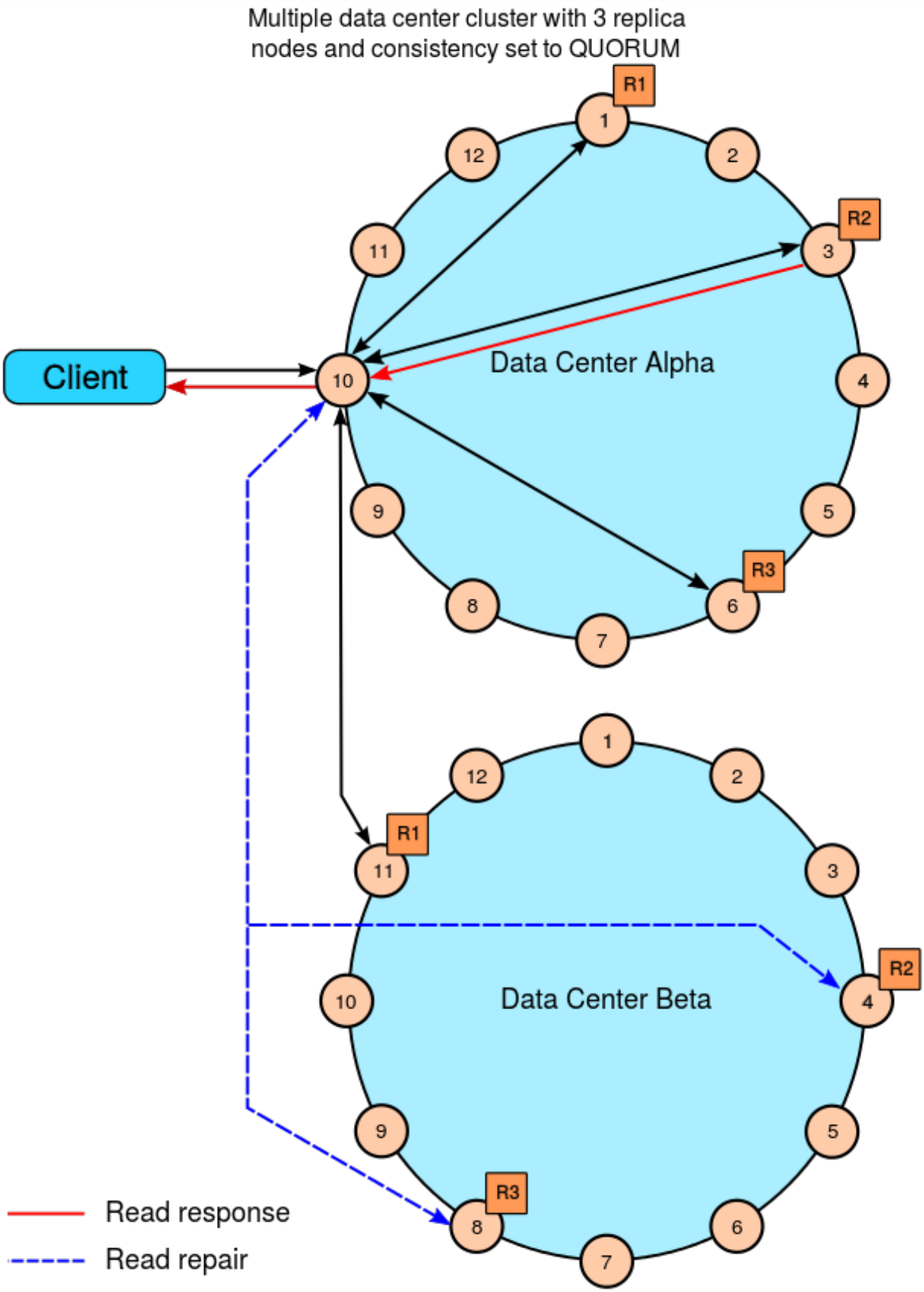


6.3.3 Read Consistency Levels

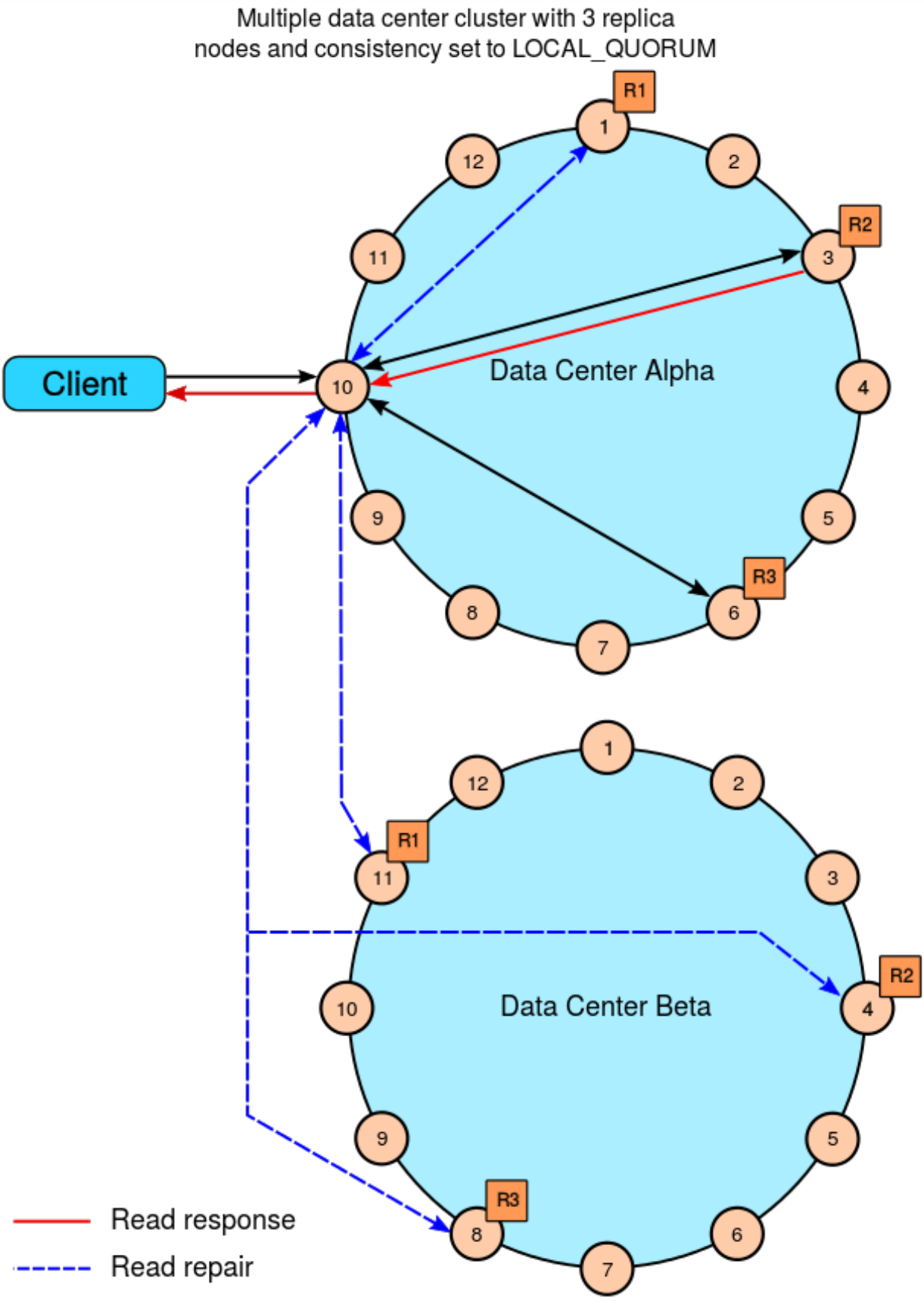
级别	描述	用法
ALL	在所有的 replica 节点响应后返回数据。如果某个 replica 节点没有响应，读操作会失败。	提供所有级别中最高的一致性和最低的可用性。
QUORUM	在所有数据中心中 quorum 数量的节点响应后返回数据。	用于单个或多个数据中心集群，保证整个集群的强一致性，但有可能写入失败。
LOCAL_QUORUM	在coordinator 所在数据中心中 quorum 数量的 replica 节点响应后返回数据。避免跨数据中心的通信。	用于使用 NetworkTopologyStrategy 策略的多数据中心集群。使用 SimpleStrategy 策略时会失败。
ONE	返回由 snitch 决定的最近的 replica 节点返回的结果。默认情况下，后台会触发 read repair 确保其他 replica 的数据一致。	提供最高级别的可用性，但读取到的数据有较高的可能性不是最新写入的数据。
TWO	返回最近的两个 replica 节点的最新数据。	和 ONE 类似。
THREE	返回最近的三个 replica 节点的最新数据。	和 TWO 类似。
LOCAL_ONE	返回本地数据中心内最近的 replica 节点的数据。	同写操作 Consistency level 中该级别的用法。
SERIAL	允许读取当前（可能未提交）的数据状态，而不执行新的添加或更新。 如果读取时发现正在执行未提交的事务，则会将提交事务作为读取操作的一部分。操作成功判断策略与 QUORUM 类似。	在用户调用轻量级事务写入列后，要读取列的最新值，请使用 SERIAL。Cassandra 将会检查轻量级事务的更新，返回最新的数据。
LOCAL_SERIAL	和 SERIAL 类似，但是仅限本地数据中心。操作成功判断策略与 LOCAL_QUORUM 类似。	用于实现轻量级事务的线性化一致性。

6.3.3.1 Read QUORUM

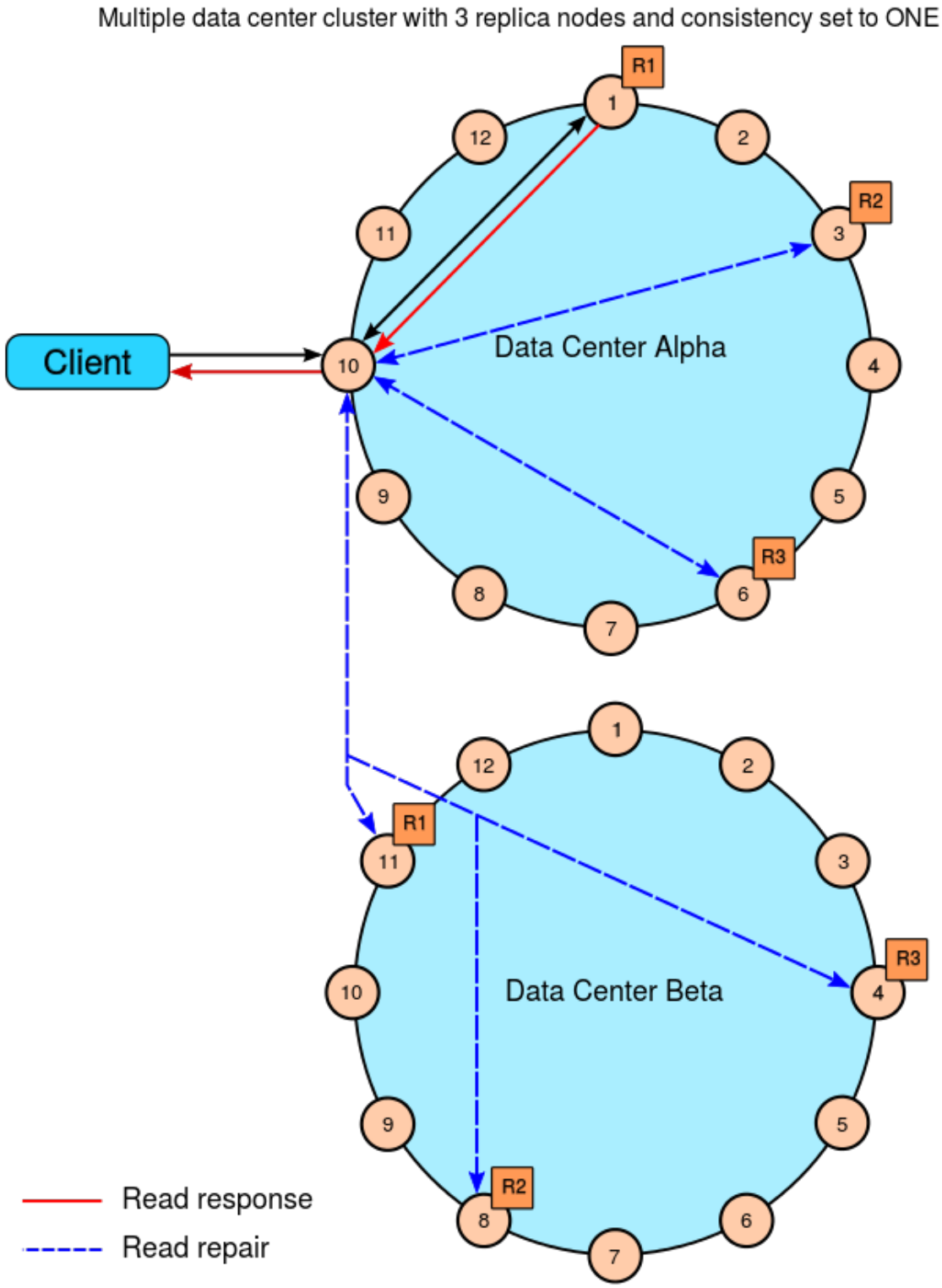

Single data center cluster with 3 replica nodes and consistency set to QUORUM
R1
R2
R3
1
2
3
4
5
6
7
8
9
10
11
12
Client
Read response
Read repair



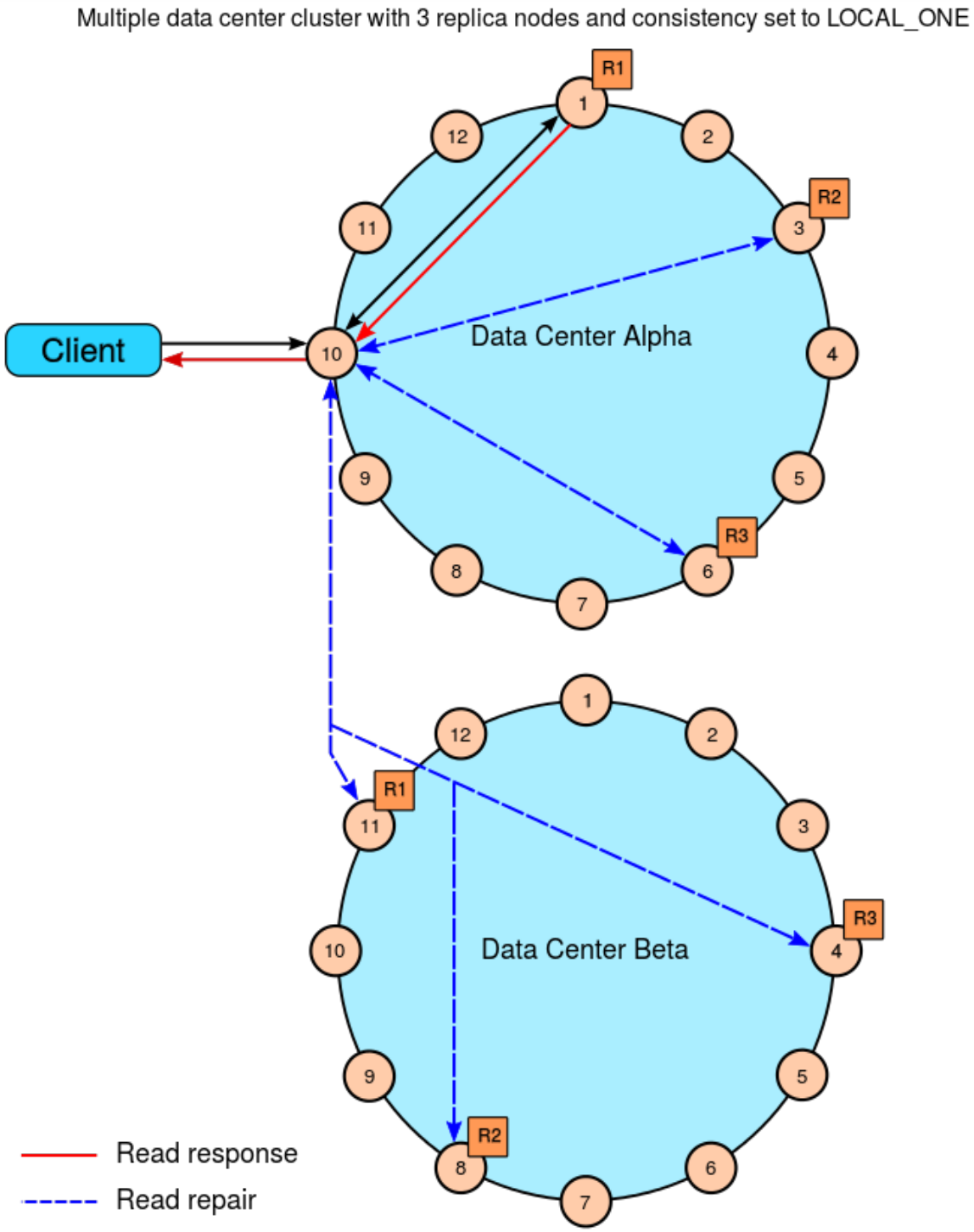
6.3.3.2 Read LOCAL_QUORUM



6.3.3.3 Read ONE



6.3.3.4 Read LOCAL_ONE



6.3.4 QUORUM

QUORUM 级别的节点数量通过下面的公式进行计算：

> quorum = floor((sum_of_replication_factors / 2) + 1)

QUORUM 中 **sum_of_replication_factors** 是数据存储在所有数据中心的节点总数；

> sum_of_replication_factors = datacenter1*RF* + *datacenter2*RF + . . . + datacentern_RF

LOCAL_QUORUM 中 **sum_of_replication_factors** 是数据存储在本地数据中心的节点总数；

根据一致性保证公式 **(nodes_written + nodes_read) > replication_factor** 可知，在读写操作均使用 **QUORUM** 策略时，能够保证强一致性。

7 Cassandra 常用查询

7.1 CREATE KEYSPACE

> 创建 **KEYSPACE**。

7.1.1 语法

```
CREATE KEYSPACE [ IF NOT EXISTS ] keyspace_name WITH options
```

7.1.2 重要选项

- **replication**

  > 复制策略选项

  - **SimpleStrategy**

    > 单数据中心时，指定每行数据存储的节点数

  - **NetworkTopologyStrategy**

    > 多数据中心时，分别指定每行数据存储在集群中的每个数据中心的节点数

7.1.3 实例

```
CREATE KEYSPACE test WITH replication = {'class': 'SimpleStrategy',
'replication_factor' : 3};
CREATE KEYSPACE test WITH replication = {'class':
'NetworkTopologyStrategy', 'DC1' : 2, 'DC2' : 3};
```

7.2 USE

> 切换为当前 **KEYSPACE**。

7.2.1 语法

```
USE keyspace_name
```

7.3 ALTER KEYSPACE

> 调整 **KEYSPACE** 属性。

7.3.1 语法

```
ALTER KEYSPACE keyspace_name WITH options
```

7.3.2 实例

```
ALTER KEYSPACE test WITH replication = {'class': 'SimpleStrategy',
'replication_factor' : 4};
```

7.4 DROP KEYSPACE

> 删除 **KEYSPACE**。

7.4.1 语法

```
DROP KEYSPACE [ IF EXISTS ] keyspace_name
```

7.5 CREATE TABLE

> 创建 **column family**。

7.5.1 语法

```
CREATE TABLE [ IF NOT EXISTS ] table_name
    '('
        column_definition
        ( ',' column_definition )*
        [ ',' PRIMARY KEY '(' primary_key ')' ]
    ')' [ WITH table_options ]
```

7.5.2 重要选项

- **PRIMARY KEY**

  > 设置表的 **Partition Key** 和 **Clustering Key**，参见之前的表索引章节。

- **read_repair_chance**

  > 为了读取修复而查询额外节点（比一致性级别要求更多的节点）的概率，默认 **0.1**。

- **CLUSTERING ORDER**

  > **clustering key** 默认按 **ASC** 排序，当有多个 **clustering key** 时，查询语句的 **ORDER BY** 子句也只能按顺序取部分 **clustering key** 或者全部 **clustering key** 进行的 **ASC** 或 **DESC** 排序，而不能对多个 **clustering key** 分别采取不同的排序。
  >
  > **CREATE TABLE** 时可以指定 **clustering key** 的排序，这样就可以在查询时使用我们需要的查询顺序。

7.5.3 实例

```
CREATE TABLE users (
    uid int PRIMARY KEY,
    name varchar,
    age int,
    sign text
) WITH comment='用户信息表' AND read_repair_chance = 1.0;

CREATE TABLE users (
    uid int,
    name varchar,
    age int,
    sign text
    PRIMARY KEY (uid, age)
) WITH CLUSTERING ORDER BY (age DESC);
```

7.6 ALTER TABLE

> 调整 **column family** 属性。

7.6.1 语法

```
ALTER TABLE table_name
    ADD column_name cql_type ( ',' column_name cql_type )*

    | DROP column_name ( column_name )*
    | WITH options
```

7.6.2 实例

```
ALTER TABLE users ADD content varchar;

ALTER TABLE users WITH comment = '用户信息表' AND read_repair_chance =
0.2;
```

7.7 DROP TABLE

> 删除 **column family**。

7.7.1 语法

```
DROP TABLE [ IF EXISTS ] table_name
```

7.8 TRUNCATE

> 清空 **column family**。

7.8.1 语法

```
TRUNCATE [ TABLE ] table_name
```

7.9 SELECT

> 查询 **column family** 数据。

7.9.1 语法

```
SELECT [ JSON | DISTINCT ] ( select_clause | '*' )
                       FROM table_name
                       [ WHERE where_clause ]
                       [ GROUP BY group_by_clause ]
                       [ ORDER BY ordering_clause ]
                       [ PER PARTITION LIMIT (integer | bind_marker) ]
                       [ LIMIT (integer | bind_marker) ]
                       [ ALLOW FILTERING ]
```

7.9.2 重要选项

- **JSON**

  > SELECT 时使用 JSON 选项可将查询结果输出为 JSON 格式。

- **ALLOW FILTERING**

  > 允许对查询结果进行过滤。默认查询不支持对未指定确定的 Partition Key 的值的结果集进行排序和进一步过滤，开启此选项可以支持这类查询。但可能导致大范围的数据扫描，影响系统性能。

7.9.3 实例

```
SELECT name, age FROM users WHERE uid IN (199, 200, 207);
SELECT JSON uid, name, age FROM users WHERE uid = 199;
SELECT uid, name, sign FROM users WHERE uid = 199 AND age > 16 AND age
<= 20;
SELECT COUNT(*) AS user_count FROM users;
SELECT * FROM users WHERE name='may' ALLOW FILTERING;
```

7.10 INSERT

> 向 **column family** 新增数据。

7.10.1 语法

```
INSERT INTO table_name ( names VALUES tuple_literal | JSON string [ DEFAULT ( NULL | UNSET ) ] )
        [ IF NOT EXISTS ]
        [ USING update_parameter ( AND update_parameter )* ]
```

7.10.2 重要选项

- **JSON**

  > **INSERT** 时使用 **JSON** 选项可将 **JSON** 格式字符串作为数据源添加到 **column family**。

7.10.3 实例

```
INSERT INTO users (uid, name, age, sign) VALUES (1001, 'May', 20, "I am May.");
INSERT INTO users JSON '{"uid": 1001, "name": "May", "age": 20, "sign": "I am May."}';
```

7.11 UPDATE

> 更新 **column family** 中的数据。
>
> **UPDATE** 语句更新表中符合条件行的一个或多个列。**where_clause** 用于筛选更新的行，必须包括组成 **PRIMARY KEY** 的所有列。 **UPDATE** 语句只能更新非 **PRIMARY KEY** 列的值。
>
> 除非使用 IF，否则 UPDATE 并不会检查旧的行是否存在。如果旧的行不存在，则会创建新行，而且更新还是创建是无法获知的。虽然使用 IF 能避免创建新行，但会额外的消耗大量性能，因此应谨慎使用。

7.11.1 语法

```
UPDATE table_name
    [ USING update_parameter ( AND update_parameter )* ]
    SET assignment ( ',' assignment )*
    WHERE where_clause
    [ IF ( EXISTS | condition ( AND condition )*) ]
```

7.11.2 重要选项

- **TIMESTAMP**

  > 设置操作的时间戳，不指定则使用语句开始执行的时间。时间戳用于数据的版本管理。

- **TTL**

  > 设置插入值的生存时间，插入值将在指定的时间后从数据库删除。默认 **TTL** 为 **0**，表示数据不会过期。

7.11.3 实例

```
UPDATE users USING TTL 400 SET sign = 'HaHaHa!', content = 'HeHeHe!'
WHERE uid = 1001 AND age = 20;
```

7.12 DELETE

> 删除 **column family** 中的数据。
>
> **DELETE** 可以删除行的某些列也可以删除整行。 如果在 **DELETE** 关键字之后指定了列名，那么仅删除符合 **WHERE** 子句条件的行的指定列，未指定列则将删除整行。

7.12.1 语法

```
DELETE [ simple_selection ( ',' simple_selection ) ]
    FROM table_name
    [ USING update_parameter ( AND update_parameter )* ]
    WHERE where_clause
    [ IF ( EXISTS | condition ( AND condition )*) ]
```

7.12.2 实例

```
DELETE FROM users USING TIMESTAMP 1240003134 WHERE uid = 1001;
DELETE sign FROM users WHERE uid IN (1001, 1002);
```

7.13 BATCH

> 将多个 **INSERT**, **UPDATE**, **DELETE** 组合起来一次性执行。
>
> **BATCH** 中的语句是一个整体，可以理解为一个简单的事务，要么全部成功，要么全部失败。

7.13.1 语法

```
BEGIN [ UNLOGGED | COUNTER ] BATCH
    [ USING update_parameter ( AND update_parameter )* ]
    modification_statement ( ';' modification_statement )*
    APPLY BATCH
```

7.13.2 重要选项

- **UNLOGGED**

  > 默认情况下，**BATCH** 中的语句是一起成功或者一起失败的，但 **BATCH** 中的语句跨多个分区时，原子性不能够完全保证。此时可使用 **UNLOGGED** 选项，但组合的语句中可能只有部分执行成功。

7.13.3 实例

```
BEGIN BATCH
   INSERT INTO users (uid, age, name, sign) VALUES (1001, 20, 'May', 'HeHe!');
   UPDATE users SET sign = 'HaHa!' WHERE uid = 1001;
   INSERT INTO users (uid, age, name) VALUES (1002, 19, 'Tom');
   DELETE name FROM users WHERE uid = 1003;
APPLY BATCH;
```

8 Cassandra 安装和配置

8.1 系统需求

1. Java >= 1.8 (OpenJDK or Oracle JVMS)
2. Python 2.7 (for cqlsh)

8.2 安装 Cassandra

1. 安装 **JDK**；

   ```
   yum install java-1.8.0-openjdk
   ```

2. 添加仓库；

```
cat > /etc/yum.repos.d/cassandra.repo <<EOF
[cassandra]
name=Apache Cassandra
baseurl=https://www.apache.org/dist/cassandra/redhat/311x/
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://www.apache.org/dist/cassandra/KEYS
EOF
```

3. 安装 **Cassandra**。

```
yum install cassandra
```

4. 启动 **Cassandra**。

```
service cassandra start
```

5. 开机自启动。

```
chkconfig cassandra on
```

8.3 安装 Cassandra for PHP

8.3.1 安装 PHP 扩展

1. 安装前置依赖 **cassandra cpp driver**。

```
wget http://downloads.datastax.com/cpp-driver/centos/7/cassandra/v2.7.0/cassandra-cpp-driver-2.7.0-1.el7.centos.x86_64.rpm
wget http://downloads.datastax.com/cpp-driver/centos/7/cassandra/v2.7.0/cassandra-cpp-driver-devel-2.7.0-1.el7.centos.x86_64.rpm
rpm -ivh cassandra-cpp-driver-2.7.0-1.el7.centos.x86_64.rpm
rpm -ivh cassandra-cpp-driver-devel-2.7.0-1.el7.centos.x86_64.rpm
```

2. 安装 **cassandra php driver**。

```
pecl install cassandra
```

8.3.2 代码实例

```php
<?php
const KEY_SPACE = 'test';

$cluster = Cassandra::cluster()->build();  // 默认连接本地服务器
$session = $cluster->connect(KEY_SPACE);   // 创建会话，并指向 keyspace: system
$statement = new Cassandra\SimpleStatement('SELECT uid, name FROM users');  // 创建查询
$future = $session->executeAsync($statement);  // 异步执行查询
$result = $future->get();  // 等待查询数据返回

// 遍历查询结果
foreach ($result as $row) {
    printf("User %s's name is %s;\n", $row['uid'], $row['uname']);
}
```

8.3.3 更多例子

[代码示例 @ github.com/datastax/php-driver]

8.4 配置 Cassandra

8.4.1 配置集群

1. 在所有的节点上安装 **Cassandra**，并确保防火墙允许以下端口的访问。

端口	访问范围	cassandra.yaml 中的选项
7000	集群内节点通讯（限制外部访问）	storage_port
7001	集群内节点 SSL 通讯（限制外部访问）	ssl_storage_port
7199	集群内 JMX 监控（限制外部访问）	-
9042	客户端 CQL 本地传输	native_transport_port
9160	客户端 Thrift 连接	rpc_port

2. 划分数据中心、机架，确定种子节点，一般按照物理位置划分。

> 例如，一共有 **6** 个节点，分别位于 **2** 个机房，可以划分为 **2** 个数据中心 **DC1**、**DC2**，划分为 **2** 个机架 **RAC1**、**RAC2**，集群命名为 **MyCluster**。

```
DC1(RAC1):
    172.20.2.23 (seed)
    172.20.2.110
    172.20.2.120

DC2(RAC2):
    172.20.3.13 (seed)
    172.20.3.23
    172.20.3.24
```

> **Cassandra** 使用对等通信协议 **gossip** 在节点间周期性的交换节点的状态信息。**gossip** 进程每秒都会与群集中最多三个其他节点交换状态消息，交换的信息中包括本节点和本节点已知的其他节点的信息，所以每个节点都能够快速的获知整个集群的所有节点的状态。**gossip** 交换具有版本的数据，以保持每个节点都能够获得最新的状态信息。
>
> **seed** 又称种子节点，用于在节点首次加入集群中时，引导节点启动 **gossip** 进程，除此之外，种子节点并没有其他特殊用途。在节点首次加入集群后，会获取到并存储集群中其他节点的信息，所以再次启动节点时并不需要连接 **seed**。种子节点在集群建立后可以安全移除，不会造成单点故障。
>
> 为了快速可靠的建立 **gossip** 通讯，集群中所有的节点，应该配置相同的种子节点。

3. 停止所有节点的 **Cassandra** 服务并清理数据。

> **Cassandra** 启动后会存储集群所有节点的信息，所以重新配置集群前需要清理数据。

```
service cassandra stop
rm -rf /var/lib/cassandra/data/system/*
```

4. 修改所有节点的 **Cassandra** 配置文件。
    - 修改 **/etc/cassandra/conf/cassandra.yaml** 中的相关选项：
        - **cluster_name**

          > 集群名称，按上例设定应设置为：

          ```
          cluster_name: 'MyCluster'
          ```

        - **seeds**

          > 种子节点地址，按上例设定应设置为：

          ```
          seed_provider:
              - class_name:
          org.apache.cassandra.locator.SimpleSeedProvider
                parameters:
                    - seeds: "172.20.2.23,172.20.3.13"
          ```

- **listen_address** or **listen_interface**

  > 节点间通讯监听的地址或接口，默认为 **localhost** 和 **eth0**。
  >
  > **listen_address** 和 **listen_interface** 只需且仅能设置其一。
  >
  > 按上例设定，以节点 **172.20.2.23** 为例，应设置为：

```
listen_address: 172.20.2.23
# listen_interface: eth0
```

- **rpc_address** or **rpc_interface**

  > **Thrift RPC** 和 **CQL** 本地传输服务通讯监听的地址或接口，默认为 **localhost** 和 **eth0**。
  >
  > **rpc_address** 和 **rpc_interface** 只需且仅能设置其一。
  >
  > 按上例设定，以节点 **172.20.2.23** 为例，应设置为：

```
rpc_address: 172.20.2.23
# rpc_interface: eth0
```

- **endpoint_snitch**

  > **Cassandra** 使用 **snitches** 来定位节点和路由请求，默认为 **SimpleSnitch**。
  >
  > **SimpleSnitch** 仅支持单数据中心，不能识别数据中心和机架信息。
  >
  > 在生产环境中，应使用 **GossipingPropertyFileSnitch**。
  > GossipingPropertyFileSnitch 将本地节点的数据中心和机架信息定义在 **cassandra-rackdc.properties** 文件中，并通过 **gossip** 传播到其他节点。

```
endpoint_snitch: GossipingPropertyFileSnitch
```

- **auto_bootstrap**

  > 节点启动时自动从集群中迁移正确的数据到本节点。
  >
  > 设置中没有 **auto_bootstrap** 选项 ， **auto_bootstrap** 默认为 **true**。
  >
  > 初始化新集群和添加新的数据中心时，应手动添加选项禁用 **auto_bootstrap**。

```
auto_bootstrap: false
```

- **data_file_directories**

  > 数据文件存储目录，默认为 **/var/lib/cassandra/data**。
  >
  > 如修改为 **/data/cassandra/data**，则应设置为：

```
data_file_directories:
    - /data/cassandra/data
```

- **commitlog_directory**

  > **commitlog** 存储目录，默认为 **/var/lib/cassandra/commitlog**。
  >
  > 如修改为 **/data/cassandra/commitlog**，则应设置为：

  ```
  commitlog_directory: /data/cassandra/commitlog
  ```

- **saved_caches_directory**

  > 缓存数据存储目录，默认为 **/var/lib/cassandra/saved_caches**。
  >
  > 如修改为 **/data/cassandra/saved_caches**，则应设置为：

  ```
  saved_caches_directory: /data/cassandra/saved_caches
  ```

- **hints_directory**

  > hints 存储目录，默认为 **/var/lib/cassandra/hints**。
  >
  > 如修改为 /data/cassandra/hints，则应设置为：

  ```
  hints_directory: /data/cassandra/hints
  ```

- 对于多数据中心的集群，还需要修改 **/etc/cassandra/conf/cassandra-rackdc.properties** 。

  > 按上例设定，节点 **172.20.2.23**、**172.20.2.110**、**172.20.2.120** 应设置为：

  ```
  dc=DC1
  rack=RAC1
  ```

  > 节点 **172.20.3.13**、**172.20.3.23**、**172.20.3.24** 应设置为：

  ```
  dc=DC2
  rack=RAC2
  ```

5. 启动所有节点的 Cassandra 服务，先启动种子节点。

   ```
   service cassandra start
   ```

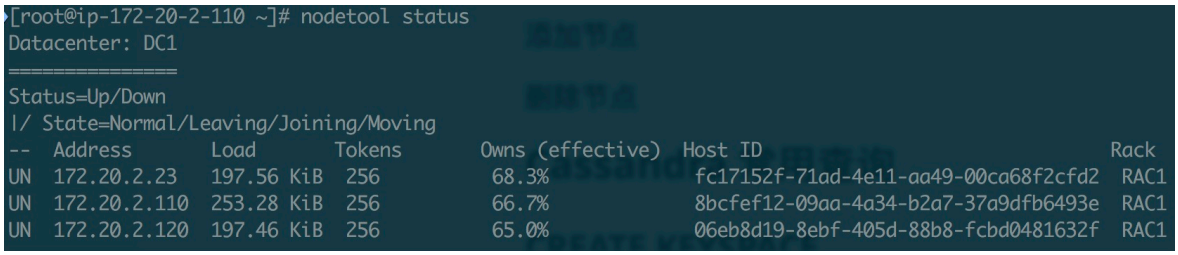
6. 删除 **/etc/cassandra/conf/cassandra.yaml** 中的 **auto_bootstrap** 选项配置。

   > 删除配置文件中的 auto_bootstrap 选项，或者显式的将 auto_bootstrap 设置为 true。
   >
   > 确保节点重新启动时能够正确的从其他节点获取到所有数据。

   ```
   auto_bootstrap: true
   ```

7. 使用 nodetool 确认所有节点状态。

```
nodetool status
```



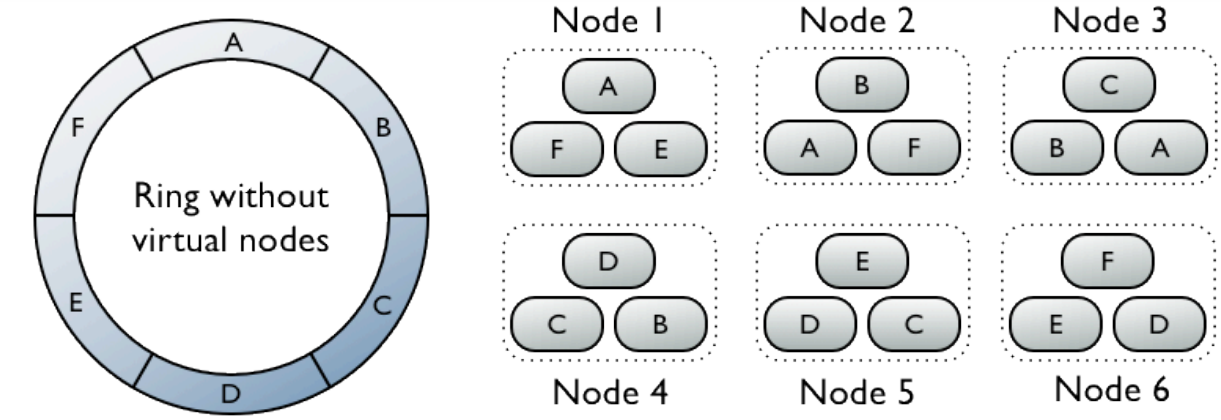
8.4.2 调整节点

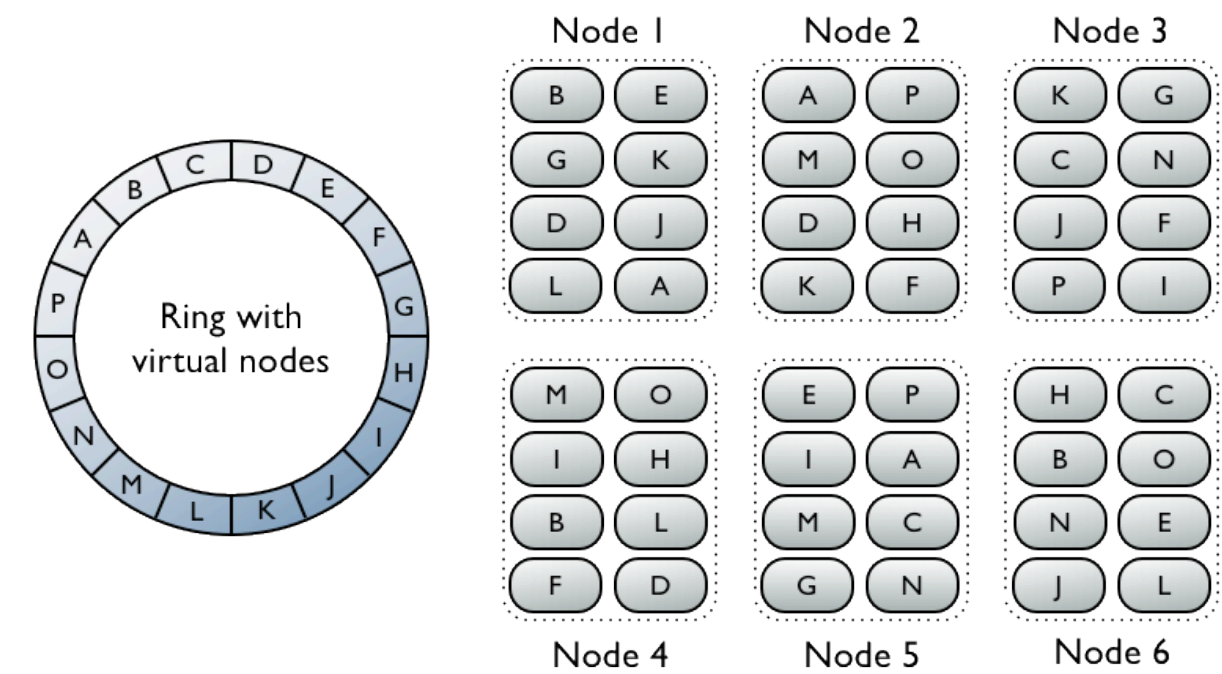
8.4.2.1 虚拟节点

Cassandra 使用了虚拟节点技术，将整个 Ring 分成 256 个虚拟节点（基于 num_tokens 设置），随机的不连续的分布到各个物理节点上。由于虚拟节点的数量是不变的，所以增删节点时不用重新计算和分配令牌。

由于虚拟节点的数量很多，数据能够更平均的分布到各个节点上。增加节点时，新增节点从现有节点上拷贝多个虚拟节点的数据；删除节点时，则将当前节点上的虚拟节点数据拷贝到多个其他的节点上。

平衡数据的工作，将由 Cassandra 在增删节点时自动完成，无需人工干预。所以在 Cassandra 中增删节点是十分容易的。





8.4.2.2 增加节点

1. 和配置集群的操作类似，对新增节点执行配置集群的步骤 3、4、5。

   > 新增节点的 **auto_bootstrap** 应设置为 **true**，确保新节点能够正确的自动从其他节点平衡数据。

2. 使用 nodetool 确认新增节点状态。
3. 在所有节点都正确启动并运行之后，使用 `nodetool cleanup` 命令清理原有节点的数据。

   > `nodetool cleanup` 命令能够清除节点上不属于本节点的数据，经常用于集群新增节点后。
   >
   > 不清理数据并不影响集群的正常使用，所以可以安排在方便的时候进行。
   >
   > 清理时应每个节点依次执行，上一个节点清理完毕后，再在下一个节点执行清理命令。

8.4.2.3 删除节点

1. 使用 nodetool 确认待删除节点状态。

```
nodetool status
```

```
[root@ip-172-20-2-110 ~]# nodetool status
Datacenter: DC1
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address       Load       Tokens       Owns (effective)  Host ID                               Rack
UN  172.20.2.23   294.78 KiB  256          68.3%             fc17152f-71ad-4e11-aa49-00ca68f2cfd2  RAC1
UN  172.20.2.110  350.74 KiB  256          66.7%             8bcfef12-09aa-4a34-b2a7-37a9dfb6493e  RAC1
DN  172.20.2.120  280.03 KiB  256          65.0%             06eb8d19-8ebf-405d-88b8-fcbd0481632f  RAC1
```

2. 如果节点处于启动状态（Up），使用 nodetool 将节点推出集群。

```
# run this on the delete node
nodetool decommission
```

> `nodetool decommission` 会完成三件事：
>
> - 将当前节点负责的数据范围分配给集群的其他节点；
> - 将当前节点的数据分发到对应的节点；
> - 通知其他节点当前节点被移除，并终止当前节点数据流，推出集群；
>
> 可以使用 `nodetool netstats` 监控删除节点的进程。

3. 如果节点处于关闭状态（Down），使用 nodetool 将节点删除。

```
# run this on the delete node
nodetool removenode
```

> **!!! 删除节点有可能导致数据丢失，请尽量恢复节点后使用 `nodetool decommission` !!!**
>
> 节点因硬件故障或网络中断等原因，被集群标记为关闭状态后，本应写入节点的数据将会作为 hints 保存到其他节点上。但当节点关闭超过设置的 max_hint_window_in_ms 时间（默认为 3 小时），hints 将不会再被保存。节点恢复后需要进行数据修复以确保数据一致性，节点关闭过久会导致数据丢失。

4. 如果删除节点后节点仍未停止向其他节点传输数据，使用 nodetool 将节点强制移除。

```
# run this on the delete node
nodetool assassinate
```

> `nodetool assassinate` 会强制删除节点而不进行任何数据平衡操作，是 `nodetool removenode` 删除节点失败后的最后手段。

8.4.3 nodetool

nodetool 是 **Cassandra** 的维护和管理工具，集群日常运维操作都可以用它来完成。

```
nodetool [options] command [args]
```

nodetool 命令的选项如下：

缩写	全写	说明
-h	--host	连接节点的 IP 或主机名（默认 localhost）
-p	--port	连接节点的端口（默认 7199）
-pwf	--password-file	密码文件路径
-pw	--password	密码
-u	--username	用户名

nodetool 的常用命令如下：

- **nodetool assassinate**

  强制删除节点而不进行任何数据平衡操作，是 `nodetool removenode` 删除节点失败后的最后手段。

- **nodetool bootstrap**

  监控和管理节点引导过程。

- **nodetool cleanup**

  清理不属于节点的 **keyspaces** 和 **partition keys**。

- **nodetool compact**

  强制压缩一个或多个表。

- **nodetool decommission**

  停用节点并将节点的数据平衡到集群的其他节点上。

- **nodetool describecluster**

  获取集群基本信息，包括集群名称、**Snitch**、分区器和 **Schema** 版本。

- **nodetool describering**

  获取 **keyspace** 的分区范围。

- **nodetool drain**

  将节点的所有 **Memtables** 刷新到磁盘上的 **SSTables**，并停止监听客户端和其他节点的连接。运行 `nodetool drain` 后需要重启 **Cassandra**。常用于升级 **Cassandra** 版本前整理数据，仅想将 **Memtables** 刷新到 **SSTables** 请用 `nodetool flush`。

- **nodetool flush**

  将节点的所有 **Memtables** 刷新到磁盘上的 **SSTables**。

- **nodetool getendpoints**

  获取持有指定 **partition key** 的节点 **IP** 或主机名。

- **nodetool getsstables**

  获取持有指定 **partition key** 的 **SSTables** 文件路径。

- **nodetool gettimeout**

获取指定类型的超时设置的值（ms）。

- **nodetool gossipinfo**

  获取集群的 **gossip** 信息。

- **nodetool info**

  获取节点信息，包括负载、运行时间、吞吐量等。

- **nodetool netstats**

  获取节点的网络信息。

- **nodetool proxyhistograms**

  获取 **coordinator** 记录的完整请求延迟，包括节点间通信读写延迟。使用此命令查看请求是否遇到慢节点。

- **nodetool rebuild**

  从其他节点重建数据，命令在集群中的多个节点上运行。 使用此命令将新的数据中心添加到现有集群。

  如果重建在未完成时终端，可以重新执行命令，重建将会从之前的中断点开始进行。

- **nodetool rebuild_index**

  完整的重建指定表的指定索引。

- **nodetool refresh**

  加载新的 **SSTables** 到系统中，而不需要重启。

- **nodetool removenode**

  从集群中删除指定节点或查看节点删除状态。

- **nodetool repair**

  修复表数据。

- **nodetool ring**

  获取 **ring** 上的节点状态和信息。

- **nodetool settimeout**

  设置制定的超时的数值（ms），设置为 0 则禁用超时。

- **nodetool status**

  获取集群状态信息，包括节点状态、负载和 ID。

- **nodetool stopdaemon**

  停止 **Cassandra** 守护进程。

- **nodetool tablestats**

  获取表的统计信息。

- **nodetool verify**

  检查表的数据校验和。

- **nodetool version**

  获取节点运行的 **Cassandra** 版本号。

9 Cassandra 性能测试

9.1 硬件配置

三个 Amazon EC2 实例：

- **CPU**: Intel(R) Xeon(R) CPU E5-2666 v3 @ 2.90GHz * 2
- **MEM**: 4G

9.2 集群结构

```
[root@ip-172-20-2-110 ~]# nodetool status
Datacenter: DC1
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address       Load       Tokens       Owns (effective)  Host ID                               Rack
UN  172.20.2.23   72 MiB     256          33.5%             fc17152f-71ad-4e11-aa49-00ca68f2cfd2  RAC1
UN  172.20.2.110  72.09 MiB  256          32.1%             8bcfef12-09aa-4a34-b2a7-37a9dfb6493e  RAC1
UN  172.20.2.120  71.58 MiB  256          34.4%             06eb8d19-8ebf-405d-88b8-fcbd0481632f  RAC1
```

9.2.1 写入测试

```
cassandra-stress write n=1000000 -rate threads=64 -node 172.20.2.110
```

```
Results:
Op rate                   :   15,295 op/s  [WRITE: 15,295 op/s]
Partition rate            :   15,295 pk/s  [WRITE: 15,295 pk/s]
Row rate                  :   15,295 row/s [WRITE: 15,295 row/s]
Latency mean              :    4.2 ms [WRITE: 4.2 ms]
Latency median            :    2.9 ms [WRITE: 2.9 ms]
Latency 95th percentile   :   11.5 ms [WRITE: 11.5 ms]
Latency 99th percentile   :   23.6 ms [WRITE: 23.6 ms]
Latency 99.9th percentile :   63.6 ms [WRITE: 63.6 ms]
Latency max               :  288.6 ms [WRITE: 288.6 ms]
Total partitions          :  1,000,000 [WRITE: 1,000,000]
Total errors              :          0 [WRITE: 0]
Total GC count            : 0
Total GC memory           : 0.000 KiB
Total GC time             :    0.0 seconds
Avg GC time               :    NaN ms
StdDev GC time            :    0.0 ms
Total operation time      : 00:01:05
```

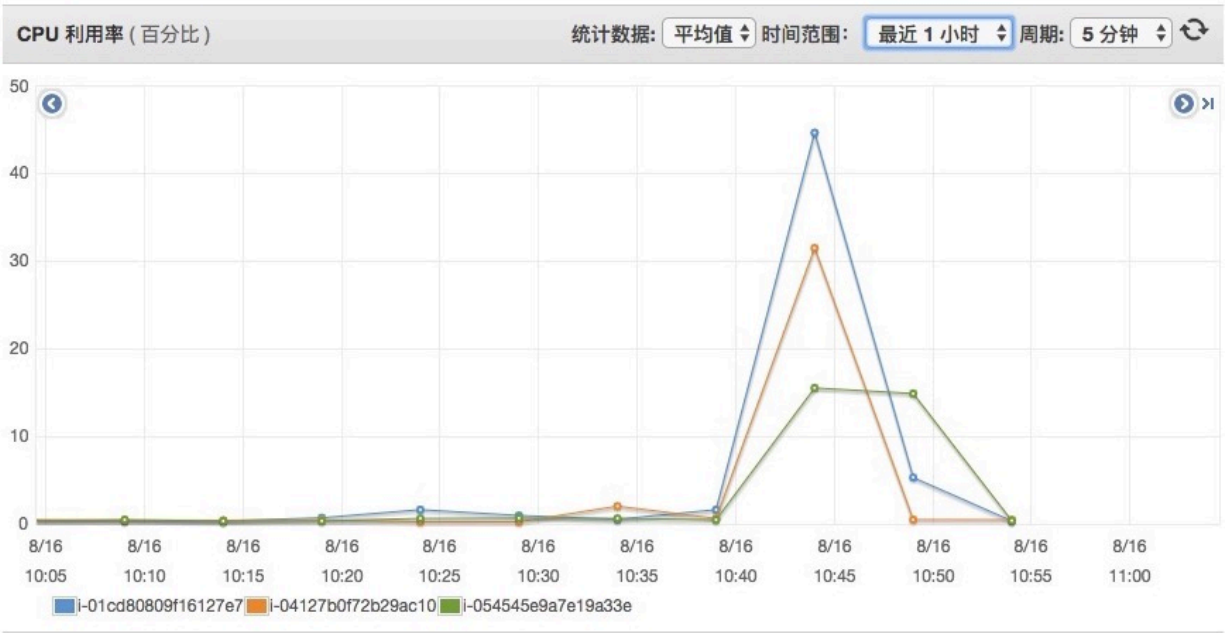
9.2.2 读取测试

```
cassandra-stress read n=1000000 -rate threads=64 -node 172.20.2.110
```

```
Results:
Op rate                   :   16,167 op/s  [READ: 16,167 op/s]
Partition rate            :   16,167 pk/s  [READ: 16,167 pk/s]
Row rate                  :   16,167 row/s [READ: 16,167 row/s]
Latency mean              :    3.9 ms [READ: 3.9 ms]
Latency median            :    3.0 ms [READ: 3.0 ms]
Latency 95th percentile   :   10.6 ms [READ: 10.6 ms]
Latency 99th percentile   :   20.4 ms [READ: 20.4 ms]
Latency 99.9th percentile :   40.3 ms [READ: 40.3 ms]
Latency max               :  152.0 ms [READ: 152.0 ms]
Total partitions          :  1,000,000 [READ: 1,000,000]
Total errors              :          0 [READ: 0]
Total GC count            : 0
Total GC memory           : 0.000 KiB
Total GC time             :    0.0 seconds
Avg GC time               :    NaN ms
StdDev GC time            :    0.0 ms
Total operation time      : 00:01:01
```

9.2.3 负载情况

> 第一个峰值来自 100W 次写入测试，第二个峰值来自 100W 次读取测试。



9.2.4 测试结论

- **Cassandra** 默认使用一般内存，本例是 2G，简单的读写测试中，本地读写均达到了 **1.5W+** 的 **QPS**；
- 内存加到 **8G**，**Cassandra** 使用 4G，本地读提高到 **1.7W**，本地写提高到 **2W**；

- 写入的负载大幅高于读取，排除测试程序的干扰，集群写入均值 **CPU 25%** 左右，读取峰值 **CPU 5%** 左右；
- 考虑到机器配置，**Cassandra** 读写性能表现十分出色；
- **cassandra-stress** 测试使用的 **KEYSPACE** 的 **replication_factor** 为 **1**，不符合实际使用情况，更精确的测试需要自行编码；