读过本文才算真正了解 Cassandra 数据库

infoq.cn/article/j0mfq1cntskbk5rbdpvl

Cassandra 数据库，值得介绍的技术细节其实挺多的。因为它很多实现思路和关系型数据库或者其他的 NoSQL 数据库，是有一些不同的。这种不同是在数据库设计实现思路上也是根源上的。所以衍生开来的诸多特点，在介绍起来就不太容易和其他数据库去类比。那么 Cassandra 有这么大量的内容，本文只能选讲其中的一部分，这部分内容是如何挑选的呢？

在《Cassandra The Definitive Guide》这本书里，有一段概括性的描述，即用 50 个 word 描述 Cassandra。它归纳了 Cassandra 的几大特性，依次为：开源、分布式、去中心化、可扩展性、高可用、容错性、可配置的一致性、行存储。

我把这几大特性分为四类：

- 第一类开源，这个不需要讨论。其余的三类 7 个特性，就是选讲的核心提纲。
- 第二类是高可用、容错性、可配置的一致性，这是围绕着多节点冗余数据的特性，换句话说，如果 Cassandra 的数据，每一行数据只有一份而没有副本，那么第二类特点就是不存在的。
- 第三类是分布式、去中心化、可扩展性，这三个特点围绕的是数据库的可拆分性，且各节点可以独立运行的能力。若只装一个单机的 Cassandra，那这一类特点就不存在。
- 第四类是行存储，是描述数据库底层存放数据的最基本的存储结构特征，也是我切入的第一个特征。

行存储结构

任何数据库设计和优化始终围绕一个核心事情——查询优化。查询永远是使用数据的核心需求。为什么要 INSERT？为了以后这个数据要查询。为什么要 DELETE？因为不再查询，并且让其他的数据更快的查询。为什么要 UPDATE，因为要实时的查询使用。无论是数据库的存储结构，像 ORACLE 的段、区、块的设计，还是辅助的存储结构，像索引这种，归根结底，为了更快速的查询出需要数据。Cassandra 也不例外，了解它的存储结构，就更加能够理解它是如何在这个存储体系下提高查询性能的，即便它是一个号称更擅长于 INSERT 的数据库。

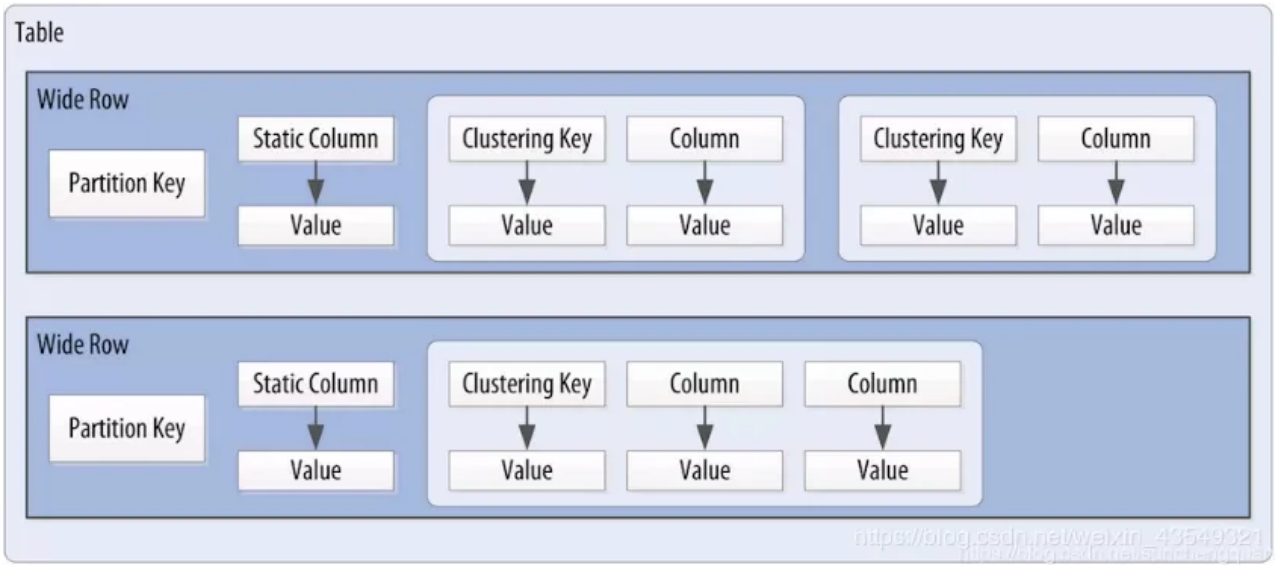
在早期的 Cassandra 中，数据库表一直被称之为 ColumnFamily（列族），我也有很长的时间将其理解为列的集合的意思。所以，我有一段时间认为 Cassandra 是一个列存储的数据库。那为什么 Cassandra 的数据模型，可以认为是行存储（ROW-ORIENTED）的，但又会在早期表被称之 ColumnFamily 呢？因为从根本上来讲，Cassandra 不能算一个严格的行存储，当然它更不是列存储，它的数据是存储在一个稀疏矩阵中的。可能这个解释略微抽象。那么我先来说下它为什么不是行存储。

任何传统的行存储数据库，一旦 DDL 定义了数据表有多少个列。那么这一行数据一定存储了所有的列值。即使出现了这一列没有值的情况，那么也一定存储了一个 NULL 值，或者是由应用程序存储一个空格或 0 来表示没有值。这一列对应的存储空间一定是存在的，当然数据库中的 varchar 或者压缩算法会使得这个存储空间尽可能小。

但是，Cassandra 允许对于任何给定的行，你可以只包含其中几列，而并非一行数据要有所有的列，当然 KEY 列是要有的。这种在列值存储上的动态性，是传统的行存储数据库根本不具备的。我猜这也可能早期为何有 ColumnFamily 概念的根源。

前文提到了，我有一段时间认为 Cassandra 是一个列存储的数据库。但是，我从来都认为它是一个不彻底的列存储数据库，而是一个受限的列存储数据库。不彻底在哪里？大部分的列存储数据库，都是为了 OLAP 而生的，它的优势在于，在某一列上做聚合的性能无语伦比。

比如我一个表有 100 列，我要对某一列求个 SUM。列存储数据库可以完美绕过多余的 99 列，只把需要的这一列一个不差的拿出来做 SUM。但是，用过 Cassandra 数据库的人都知道，在任何一列上做全列级的聚合，那简直是灾难性的。就凭 Cassandra 会将不同的 KEY 部署在不同的数据库节点/分区 PARTITION（注意这里和传统数据库分区不同），任何列级的操作，都会需要在多个数据库上打转。更何况，CQL 语句到来的时候，还要搞清楚，这个聚合列在这行数据上有没有。所以，Cassandra 是不具备列存储数据库的特质的。



为什么，最后 Cassandra 还是被描述为是一个 ROW-ORIENTED 呢？

首先，它的存储是紧紧围绕着 Key 的。Key，它是一行数据的唯一标识符号。一行数据围绕着 Partition Key 存在一起,并且围绕着 Clusterting Key 局部有序。可见它 ROW-ORIENTED 的特点还是很鲜明的。什么样的存储结构，就决定一个数据库擅长做什么。按主键排序的行存储 DB2 数据库最擅长的是什么？是在 OLTP 系统里，通过主键做单条记录的快速查询（Select by Key），这也正是 Cassandra 最为常见的 CQL 形态。什么样的存储结构，也决定了什么样的操作会有限制。

在理解了 Cassandra 数据库的 ROW-ORIENTED 的稀疏矩阵存储之后，再来看看 CQL 语句的语法限制，那么这些限制就很容易理解。例如：Select 语句，Where 条件里，一定要送 Partition Key（没有次索引的情况）。如果不送，则语法上必须要添加 ALLOW FILTERING。

为什么是这样，刚才提到了，Partition Key 决定了数据存在哪里，它像是一个指针，直接指向了这一行数据的物理位置。ALLOW FILTERING 表示什么，表示的是 Cassandra 数据库获得这条记录是通过筛选得来的，而不是通过直接定位得来的。

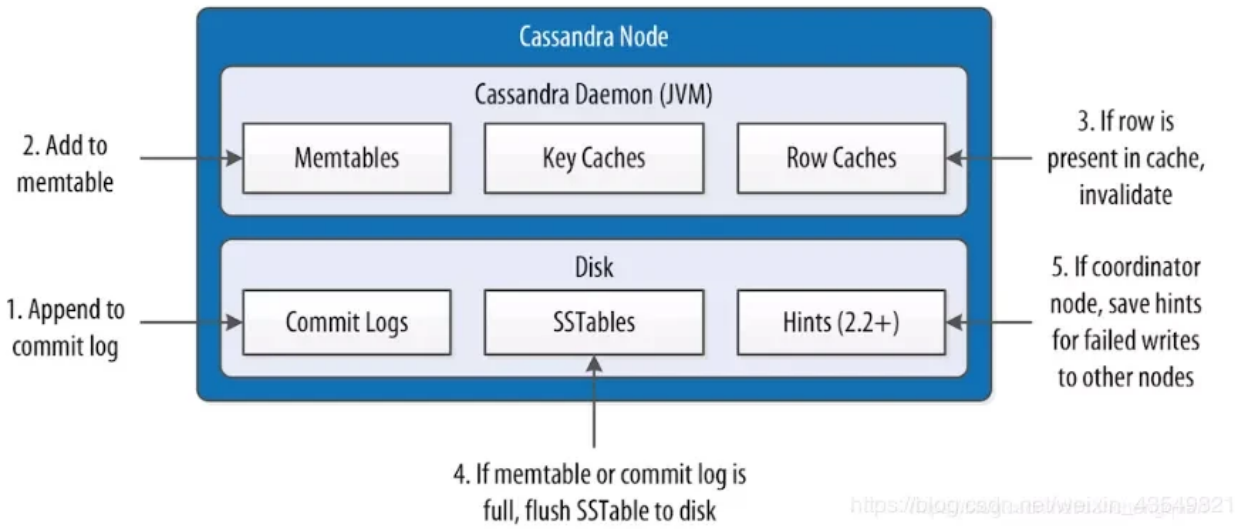
类比一下传统数据库，Where 条件送 Partition Key 就好比通过 HASH 索引定位记录，ALLOW FILTERING 就如同先做一次 TABLE SCAN，读出大量记录再从记录里过滤出符合 WHERE 条件的。再看看关于 Clusterting Key 的，CQL 语法要求，范围查找、Order by 一类的语法都需要使用 Clusterting Key，这就十分好理解。在定位的 Partition Key 确定了位置之后，同一 Partition Key 的数据，都是 Clusterting Key 有序存放的，那么通过在这

个有序的 Key 列上，无论范围也好、排序也好，都不会需要数据库引擎真正去排序，这就好像在传统数据库里，ORDER BY 的列，和某一个索引一致的情况下，执行计划里不会真的排序是一个道理。

搞清楚了 Cassandra 的存储结构之后，我们来看 Cassandra 在某一个节点上怎么做增删改查。无论 Cassandra 的多节点特点多么鲜明，在单一节点上面，数据的读写，永远才是数据库性能的根基。节点再多，如果单节点上读写性能不行，那数据库终究是快不起来的。所以这里我们来看一下，Cassandra 是怎么样读写数据的。

先翻译《Cassandra The Definitive Guide》一段话。“在 Cassandra 中，写入数据非常快，因为它的 memtables 和 SSTables 设计，使它插入时，不需要执行磁盘读取或搜索，这些减慢数据库速度的操作。Cassandra 中的所有写入都是追加形态的。”

我们看一下 Cassandra 的写入步骤，来解读它的写入优势。



第一步，写 Commit Logs。这个步骤完全不是什么新发明。我觉得它和传统数据库的 REDO Log 几乎是一样的。无论是什么数据库，这个 Log 的写入，都是追加形态的。但是，注意看这个图，Commit Logs 直接写在硬盘上，我认为这个描述并不准确。无论时传统数据库的 REDO LOG 还是 Cassandra 的 Commit Logs，它都是先到内存，再 FLUSH 到磁盘上的。而 FLUSH 的策略是由一些参数决定的，比如 commitlog_sync。这和传统数据库非常相似，这里不展开来讨论，只需要认识到一点，FLUSH 的动作频率越高，系统奔溃时丢失的数据越少，同时损失部分数据插入性能。就像 Mysql 数据库的参数 Innodb_flush_log_at_trx_commit=1 时，Mysql 是最安全，但是也是最慢的。

```
# the other option is "periodic" where writes may be acked immediately
# and the CommitLog is simply synced every commitlog_sync_period_in_ms
# milliseconds.
commitlog_sync: periodic
commitlog_sync_period_in_ms: 10000
```

第二步，Add to memtable，这是关键的一步，Cassandra 的这一步是完完全全的内存动作。而若是传统的数据库，则大约需要做这么几个动作：

- 逐层搜索索引，若这个索引块不在 DATA BUFFER 里，触发磁盘 IO。
- 通过索引定位数据块，若数据块不在 DATA BUFFER 里，触发磁盘 IO。
- 修改索引块，修改数据块，如果修改并发量大时，可能产生锁等。

当然，若是像 Oracle 数据库那样的堆表设计，纯粹的 INSERT 动作在 b 的 IO 触发可能性要少一点，但是在 UPDATE（Cassandra 中也是 Insert）场景下，这些开销都是不可少的。Cassandra 为什么可以在 Memtable 上纯粹的做追加写入，这个 Cassandra 记录的 Timestamp 概念是分不开的，即无论你写入多少次，数据库只会以最新 Timestamp 的记录为准。这样就不需要去对记录资源上锁。这样的设计，不要说没有锁冲突了，就连去把需要上锁的记录找出来的开销都省了，快就快在这个地方。

但是，这个快是有代价的，那就是数据的一致性。比如一个简单的需求，在数据写入之前，需要看看这条数据是不是存在，如果存在了就不能插入（CQL 的 IF NOT EXIST 语法），或者 UPDATE 需要看数据条件（WHERE IF Column = '*'） 。一旦这种带条件 CQL 使用，那可以推断，上面的这些优势，也就不存在了。

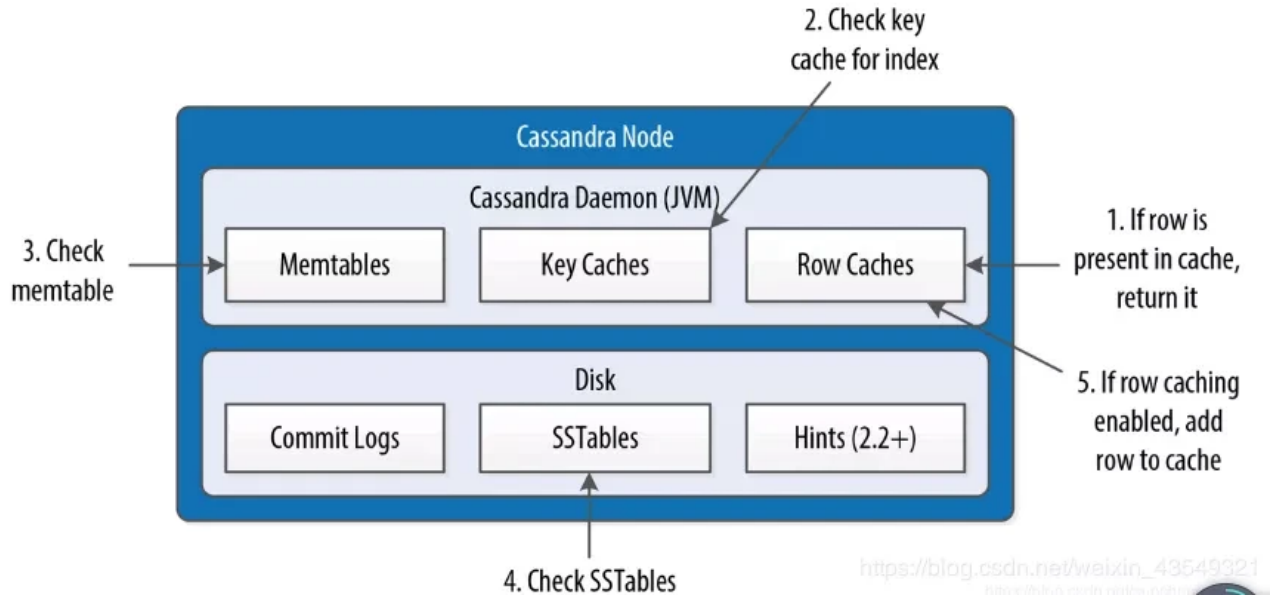
看第三步，如果这行数据在 Row Caches 里，使它失效。注意这个地方，Row Caches 里的记录是不改的。那么 Row Caches 的使用场景，只有特别热点的数据读取的时候使用，它并不适合高并发热点数据修改的场景。

常规交易，做完这三步就返回成功了，不需要等待 Memtable 的内容落盘。换句话说，直接影响交易性能的步骤，结束了。这和传统数据库也没有太大的差别。那么接下来的步骤，就不直接影响数据库的写入能力。

第四步，数据的落盘，这个动作通常是异步的，在后面会详细展开将 SSTable 的存储。第五步，这个就是多节点特性了，是一个节点异常的处理过程。

总结一下，传统的数据库的写入（包括 INSERT、UPDATE、Delete），通常是一个读后写的过程。而 Cassandra 的写入，是没有先读这个动作的，这也是它快的根本原因。一旦使用了 IF NOT EXIST 之类的语法，那么它的写入性能也就会要受损。

接下来看一下 Cassandra 的读取，它的读取是多节点、多副本的读取。此处，我们先关注一个节点上的情况。



第一步，如果这一行数据在 Row Caches 中，直接返回数据，这个好理解。

第二步，检查检查 KeyCaches 里的索引，这里可以理解为，这是一个主键索引，它存储的是未来在 Memtables 或者 SSTables 用来定位的信息（书上原文是 offset location）。需要注意的是，这里面的值，在第三步、和第四步的时候，都可能用得上，而不是仅仅用于第三步。看到这个地方，可以发现，其实这个图有个问题，就是没有指出 KeyCaches 的维护。Cache 是可以在建表时配置的一个参数。可以推测，假如我们建表的时候，keys 的 Cache 采用了 ALL 的设置，那么应该是在有新的 KEY 值写入 Memtables 时，维护到了 Key Caches 中。

第三步，这一步需要关注是，对于一个指定的表（或列族），是只会使用唯一的一个 Memtable 的，那么这个搜索就是线性的。Memtable 中的内容，是还没有 FLUSH 到 SSTables 里的数据，在查询是，它里面的内容和 SSTables 中的内容，都是要同步读取的，但对单节点而言，它的内容通常更新。

与写入场景大有不同的地方是，读数据的关键步骤，是第 4 步，读 SSTables。这里在后面的内容展开，看一下第五步，如果 Row Caches 还可用，把这条记录加入的 Row Caches。Row Caches 放的是一整行的数据，如前面提到了，适合于存放热点读取的数据。

所有的数据库，通常都是有四大常规操作，谓之“增删改查”。介绍了写入、查询之后，这里简单的介绍一下 Cassandra 的删和改。一句话简述之，Cassandra 的删除都是修改，Cassandra 的修改都是写入，所以 Cassandra 只有写入和查询。Cassandra 一直写入数据，岂不是会存储爆炸不成。在这里，我们介绍三个新概念（相对传统数据库）- Tombstones，Timestamps，Compaction。

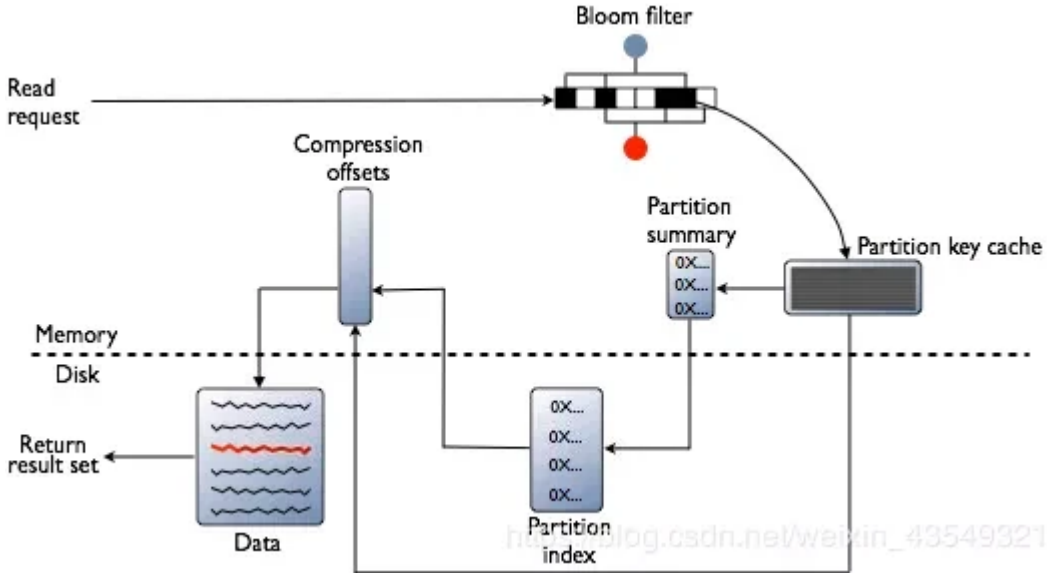
Cassandra 的删除都是修改，这个好理解，在很多业务数据库表里面，经常会为了保留痕迹，而做一个逻辑删除动作。也就是修改某个标识，表示这条记录以及删除或作废，而并没有在数据库里真正的删除。Cassandra 在收到 Delete 命令时，并不会立刻去删除这行记录。而是会给这行记录一个 Tombstones，表示它被删除了。

Cassandra 的修改都是写入。前面提到 Cassandra 速度快，快在不需要定位数据。任何 Update 命令，在传统数据库上，都需要把这条记录读到内存里，并上锁。而在 Cassandra 上，Update 命令会变成一条 INSERT 语句，那岂不是在系统里重 KEY 了吗？这里便要依靠记录上的 Timestamps。

Cassandra 的每次查询，都会把所有重的 KEY 读出来，但是永远会以最新的 Timestamps 为准。这就解决了把所有的修改，都变成写入的问题。但是，这么干有两大显而易见问题。第一，数据会无限的膨胀，吃掉磁盘。第二，数据膨胀会带来查询需要读出的重复数据增加，无限的膨胀则会无限的增加，读取性能就会受损。

所以这里，就要介绍压缩（Compaction）的概念。这里要特别的注意，这不是我们通常说的数据库压缩技术，那个通常用的 Compress。只是由于多个官方文档都把 Compaction 翻译成了压缩，我个人觉得它更应该翻译成数据的整理。Compaction 是在数据库后台异步做的，接着前面的内容，它的内容至少有比如把墓碑数据真实移除，把时间戳比较老的数据移除，重新整理 SSTable 的存储文件等。这样来解决前面那两个问题。这个动作在某种意义上来讲甚至有一点像 DB2 数据库的 REORG 动作。不同的数据库表，可以在 Keyspace 级别选择不一样的 CompactionStrategy。它常翻译为压缩算法，我觉得翻译成整理策略更加合适。我觉压缩算法，应该和 Compress 的那个概念一致。毕竟，这个 Compaction 没有给数据文件里连续的值，用个 RLE 算法，或者建个字典什么的对吧。介绍完了这些之后，让我们来直面数据库最大的瓶颈。

只要一个数据库不是内存数据库，那它永远都要面对它最大的性能瓶颈，磁盘 IO。我们前面提到的诸多概念，比如 Cache、列存储、索引等等，他们优化性能的本质都指向一处，减少磁盘 IO。前面讲读写部分时，都跳过了第 4 步。而对于 SSTable 的读取，其实才是影响性能的关键步骤。



我们来看一下，SSTable 到底是什么，它的读取是什么样子的。我们根据 SSTable 的访问顺序来看，在 3.0 版本中，SSTable 包含以下这么几个文件：

Filter.db 这是 SSTable 的 Bloom 过滤器，简单的讲，它告诉你，你要的 Key，在我这里有没有。Bloom 过滤器的工作方式是将数据集中的值映射到位数组，并使用散列函数将较大的数据集压缩为摘要字符串。根据定义，摘要使用的内存量比原始数据少得多。它速度快，可能误报，但不会漏。简言之，有可能告诉你有，但是没有。但绝不会告诉没有，却有。注意！这里划一个重点，Cassandra 会维护一个 Bloom filter 的副本在内存里面。所以，这一步不一定会有实际 IO。在书上也提到，如果加大内存，是可以减少 Bloom 过滤器误报的情况。

Summary.db，这里是索引的抽样，用来加速读取的。

Index.db，提供 Data.db 里的行列偏移量。

CompressionInfo.db 提供有关 Data.db 文件压缩的元数据。这里值得关注的是，它用了 Compression 这个词，我猜测，如果 Data.db 里面采用了压缩算法，比如说字典压缩之类的，那么这个文件里面应该就会存储字典数据，或者类似的 Compress 相关的元素据。这也就是为什么这个文件，在访问流程中是不可绕过的。因为一旦 Data.db 的数据进行了压缩，那就必须依靠相关的元数据来解压缩数据。从图上可以看出，这个元数据在内存中，相对性能会比较快。

Data.db 是存储实际数据的文件，是 Cassandra 备份机制保留的唯一文件。它是唯一的真实数据，其他的都是辅助数据。比如索引可以重建，字典可以重建等等。

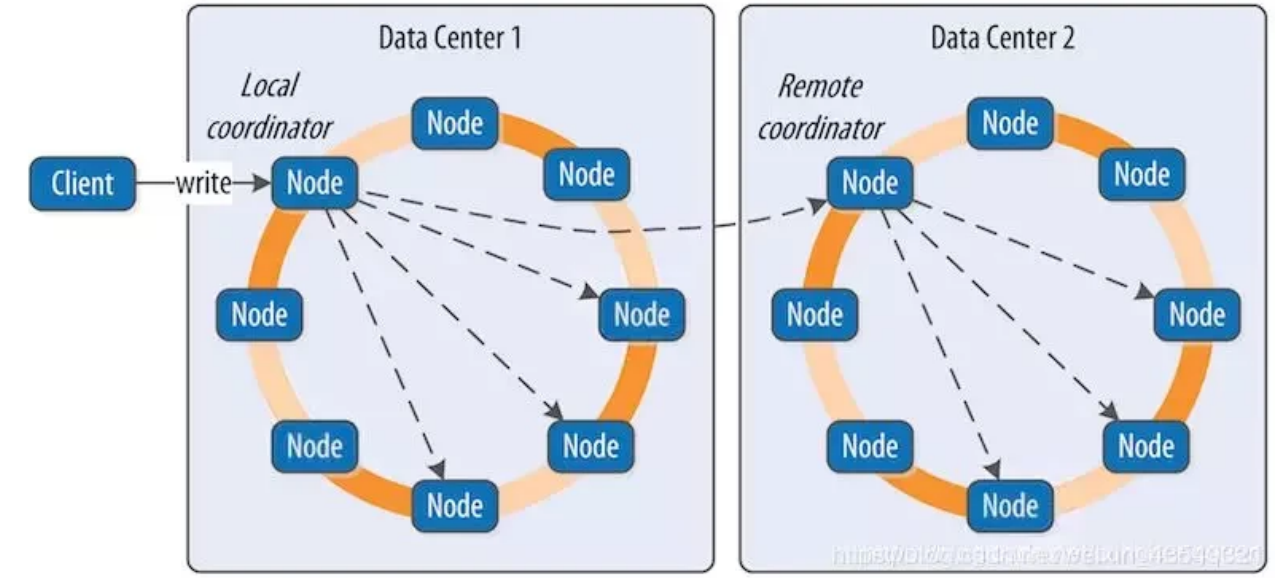
Digest.adler32 是 Data.db 校验用的。

Statistics.db 存储 nodetool tablehistograms 命令使用的有关 SSTable 的统计信息。

TOC.txt 列出此 SSTable 的文件组件。

其中 1-5 是跟 SSTable 访问数据性能相关的文件。如果 Cache 是 ALL 的情况下，Cassandra 在通常都可以在内存访问之后，直接定位到 SSTable 的具体文件和数据所在偏移量中去。相对于传统数据库，B 树索引层层向下，遇到没有的索引块就要 IO。这个性能应该还是非常可观的。

讲到这里，不知道你有没有感受到，Cassandra 的一个重要精华所在，那就是没有锁，或者叫没有资源上的冲突和争抢。通过 Timestamps 概念，解决数据可相同 Key 数据不要上锁的问题。尽管我们前面的内容，全部都还只是在围绕单节点数据库介绍。但是 Timestamps 的使用，是为 Cassandra 分布式、去中心、可扩展、高可用、容错性、可配置一致性提供了更多灵活方便的地方。



分布式、去中心、可扩展性

前面我们把这六条分成了两类，分布式、去中心、可扩展，这三个围绕的是 KEY 的独立性。尤其是 Partition Key，它是具有极强的独立性的。由于它的极度独立，理论上任何不同 Partition Key 的数据，就都可以放在不同机器上，去独立的提供服务，也就成就它的分布式、去中心和可扩展。对照这几条特性看一下。

分布式，百度词条上解释为，建立在网络上的软件系统。有四大特性：

- 分布性。分布式系统由多台计算机组成，它们在地域上是分散的，可以散布在一个单位、一个城市、一个国家，甚至全球范围内。整个系统的功能是分散在各个节点上实现的，因而分布式系统具有数据处理的分布性。一个逻辑上的数据库表，他是分散存储来多个 Node 中的。不同的 Key 值的记录会由 Cassandra 的不能节点提供分散的的服务。
- 自治性。分布式系统中的各个节点都包含自己的处理机和内存，各自具有独立的处理数据的功能。通常，彼此在地位上是平等的，无主次之分，既能自治地进行工作，又能利用共享的通信线路来传送信息，协调任务处理。Cassandra 只有在 Partition Key 划分数据所属 Node 的存储位置时，有主次副本之分。比如说，我的 Node1 要存放的 Key 值是多少到多少，其他的勉强称之为副本。其实 Cassandra 存的是多个地位平等主本，且都具备独立处理数据等能力，它们协同处理任务，并非传统意义上的主备数据概念。

- 并行性。一个大的任务可以划分为若干个子任务，分别在不同的主机上执行。每个 Node 自然是自己提供涉及的 Key 的服务，相互之间独立、并行。对于不同的 CQL 而言，可能会由不同 Node 来完成查询。也可以是一个 CQL 里面涉及的多个 Node，它们也基本上是并行来完成这个 CQL 的。
- 全局性。分布式系统中必须存在一个单一的、全局的进程通信机制，使得任何一个进程都能与其他进程通信，并且不区分本地通信与远程通信。同时，还应当有全局的保护机制。系统中所有机器上有统一的系统调用集合，它们必须适应分布式的环境。在所有 CPU 上运行同样的内核，使协调工作更加容易。Cassandra 是完全符合这个定义的，Coordinator 节点并不是固定的。每个节点都可以接受任何的 CQL，并且来充当协调者的角色。重要的是，对于一个应用程序或者客户端而且，可以不关心 Cassandra 后来是怎么样存储和查询数据的。它从外面看到的，始终只有一张完整的逻辑数据表。

有了分布式的基础，Cassandra 可以运行在多个 Node 下，并且多个 Node 可以部署在真实的不同的数据中心机房里，不同机架上，也就能做到去中心 Decentralized。有了这个基础，就可以配合 Cassandra 的多中心的复制策略 NetworkTopologyStrategy，在每一个数据中心定义数据复制了。

可扩展这个词其实，并不是特别准确，它的重点其实是可水平扩展。简而言之，就是在图中环上加 Node，就可以提高 Cassandra 的处理能力。这其实和它的分布式特点是密不可分的。Cassandra 的拆分粒度最细，理论上几乎可以到一个 Partition KEY。或者说，每一个 Partition KEY，都可以被看作可以拆分的，独立处理的最小的单位。增加数据的同时只要增加 Node 就可以了，这就使得它的水平扩展性是很好的。

做一个偏激的假设，如果 Cassandra 只有一份数据存储，就凭 Key 独立的特点，把不同的 Key 分到不同的机器上提供服务，也可以算得上是分布式、去中心和可扩展的。但是它这个特点是不完美，不彻底的。因为机器分得越多，任何一台机器故障，它提供的服务就是不完整的。

高可用、容错性、可配置一致性

接下来，我们继续看另外三个特点，高可用、容错性、可配置的一致性，这些特点围绕的核心就数据冗余。

任何的高可用背后，一定是有数据冗余的。传统数据库通常偏爱的是主备模式，就是当提供服务的数据库节点 DOWN 掉之后，备节点开始提供服务。这时候往往故障检测、主备切换，应用切换的时间就会成为关注的焦点，做得好一点的数据库可以在 1 分钟或者几十秒

内完成切换。不过在如今 7*24*365 的环境下，1 分钟的故障恢复时间通常并不能让用户十分满意，当切换时间压缩到一定程度，还会出现一个矛盾点，就是数据库异常时间监测阈值。如果设得太长，主备切换就慢，设太短了，一个网络抖动，就可能触发不必要的主备切换误判。

Cassandra 的数据复制（replicas）并不像传统的备份数据，它更像是多份主数据，这些数据都是时时刻刻对外提供服务的，换句话说，有一个数据库节点 DOWN 掉，完全不需要主备切换时间。在资源充足的情况下，甚至是几乎无感的（比如 7 个 replicas 坏了 1 个）。

在 Cassandra 里面，数据复制（replicas）多少份，怎么存储，这个策略是可以根据不同的 Keyspace 来设置的，相当于提供一个灵活的选择，可以根据数据库表的实际使用场景和形态，来决定数据复制的策略。

数据冗余可以说是分布式系统中的常规操作，像参数数据之类的，经常会采用数据冗余的方法来处理。然而，有冗余的地方就有同步。数据一致性问题，永远和数据冗余相伴而生。好在 Cassandra 有 Timestamps 来解决一致性问题，容错性只是一致性的一个衍生产品，简单的说，只是 Cassandra 发现了一个老 Timestamps 的错误数据，后台修复一下而已。

而可配置一致性，就是 Cassandra 的一个特别重要的特性了。因为它的影响但不仅仅是对于高可用，它还直接影响数据库性能。就传统数据库而言，开不开备库，对 OLTP 交易性能也是有直接影响的（包括 Redis 也是）。从理论上来说，Cassandra 要等更多的 Node 写入数据，那响应时间就会越慢。这个响应时间取决与最慢的那个 Node。若要交易响应更快，就需要通过异步的方式。所以 Cassandra 通常都不会等所有的 Node 都响应，等多少 Node，等哪些 Node，就是可配置一致性。

在数据写入读取方面 Cassandra 的一直性级别有：

ANY（仅写入）,ONE,TWO,THREE ,QUORUM,ALL

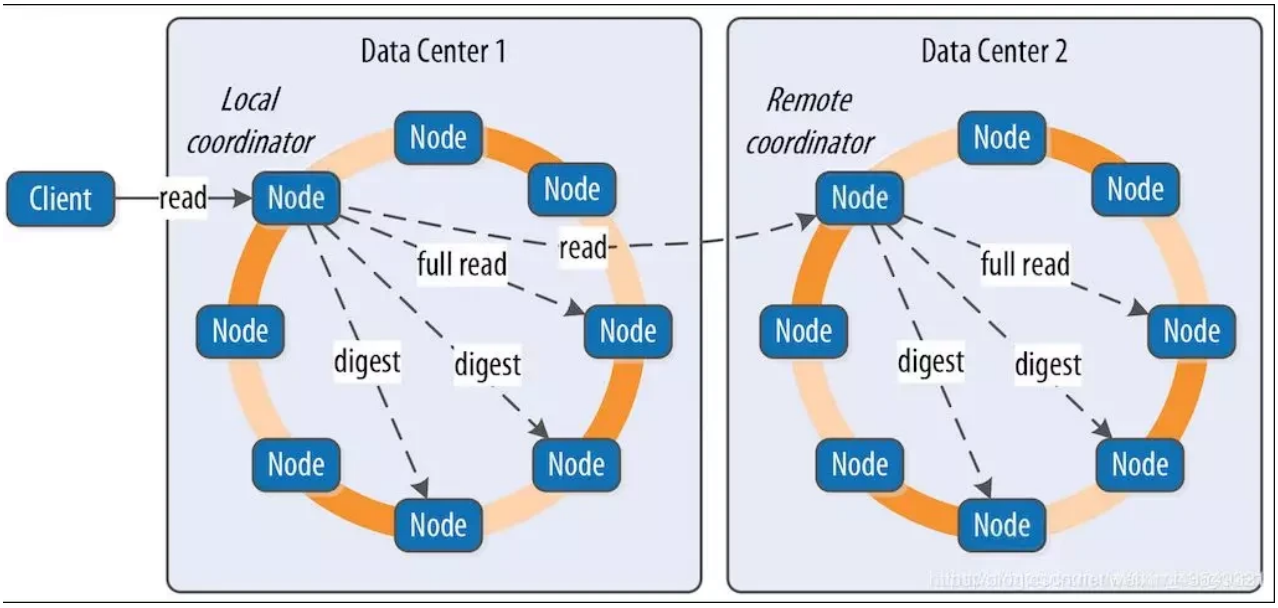
LOCAL_ONE, LOCAL_QUORUM,EACH_QUORUM

以上这些级别很好理解，不需要逐个解释。关于高可用和强一致性，永远都是鱼和熊掌。假如我们的系统使用了最快的方式写入，比如写 ANY，读 ONE。那么读到的数据并不是最实时的准确数据的可能性就会大幅增加。如上面的图，有 6 个节点在写入数据，任意一个写成功，程序就成功返回。那么假定其余 5 个节点还没有完成写入。那这时候，有一个读 ONE 的程序，恰好读到了这 5 节点中的一个，并成功返回，这就产生了数据的不一致。要做到数据的强一致，读写策略就必须配合设置，满足这样的条件。

W+R>RF W—写一致性级别 R—读一致性级别 RF—副本数

Cassandra 的这个设计非常的巧妙，它提供极好的调优灵活性。数据库调优的本质无非是个损有余而补不足的过程，这个有余并非指损性能好的地方去补性能不好的地方。

数据库或数据，有些地方有些功能，我们不用或者少用，性能不需要那么好，称之为有余；有些地方有些功能我们常用，主要用，性能要越快越好，我们称之为不足。比如很多系统的某个数据库表，它的访问形态是有局限性的。有可能一张表，100 次插入，只有 1 次读取，像流水数据。有可能一张表，1 次插入，100 次读取，像参数数据。这里面就有了极大的灵活性，我们可以损失冷门操作的性能，来保障我们的主要操作。例如，以读取为主的表，我们可以设置写入的一致性为 ALL ,读取的一致性为 ONE。从而获得一个非常高效的系统性能。



需要注意的是，数据的复制因子，是定义在 Keyspace，也就是在存储方面决定。而读取的一致性，是由客户端决定的。同样的数据，也可以根据不同使用场景来使用不同的一致性级别。比如说，对数据实时性要求高时，可以设置成读 QUORUM 或者 ALL，实时性要求

低时，选择读 ONE。

总结

至此，我已经完整的讲解了 Cassandra 的分布式、去中心化、可扩展性、高可用、容错性、可配置的一致性、行存储的特性。

回顾一下，我们先讲了 Cassandra 单节点上的行存储结构，然后围绕 Cassandra 数据 Key 的独立性介绍了分布式、去中心化、可扩展性。继而讨论了关于 Cassandra 多副本数据带来的高可用、容错性、和可配置一致性。

当然 Cassandra 数据库还有很多值得探讨和介绍的内容和概念，如 Secondary Index、Tokens、Hinted 等等。此外在 Cassandra 数据库的使用过程中，也还有监控、备份恢复、性能调优、安全等等内容值得关注学习，这里就不一一介绍了，未来有机会，再做续集吧。

作者介绍：

宇文湛泉，现任金融行业核心业务系统 DBA，主要涉及 Oracle、DB2、Cassandra 等数据库开发工作。