cassandra 3.x官方文档(7)---内部原理之如何读写数据 - 原谅我一生放荡不羁爱自由 - 博客频道

分类：
NoSql（28）

写在前面
cassandra3.x官方文档的非官方翻译。翻译内容水平全依赖本人英文水平和对cassandra的理解。所以强烈建议阅读英文版cassandra 3.x 官方文档。此文档一半是翻译，一半是个人对cassandra的认知。尽量将我的理解通过引用的方式标注，以示区别。另外文档翻译是项长期并有挑战的工作，如果你愿意加入cassandra git book,可以发信给我。当然你也可以加入我们的QQ群,104822562。一起学习探讨cassandra.

如何写

Cassandra写的时候分好几个阶段写处理数据，从立即写一个write操作开始，到把数据写到磁盘中

- 写log到commit log
- 写数据到memtable
- 从memtable中flush数据
- 将数据存储在SSTables中的磁盘中

写Log及memtable存储
当一个写发生的时候，Cassandra将数据存储在一个内存结构中，叫memtable,并且提供了可配置

的持久化，同时也追加写操作到磁盘上的commit log中。commit log记载了每一个到Cassandra 节点的写入。

这些持久化的写入永久的存活即使某个节点掉电了。memtable是一个数据分区写回的缓存。Cassandra通过key

来查找。memtables顺序写，直到达到配置的限制，然后flushed.

从memtable中Flushing数据
为了flush数据，Cassandra以memtable-sorted的顺序将数据写入到磁盘。同时也会在磁盘上创建一个分区索引，

将数据的token map到磁盘的位置。当memtable 内容超过了配置的阈值或者commitlog的空间超过了

commitlog_total_space_in_mb的值，memtable 会被放入到一个队列中，然后flush到磁盘中。这个队列可以通过

cassandra.yaml文件中memtable_heap_space_in_mb,或者memtable_offheap_space_in_mb来配置。如果待flush的

数据超过了memtable_cleanup_threshold，Cassandra会block住写操作。直到下一次flush成功。你可以手动的flush一张表，

使用nodetool flush 或者nodetool drain(flushes memtables 不需要监听跟其他节点的连接)。为了降低commit log

的恢复时间，建议的最佳实践是在重新启动节点之前，flush memtable.如果一个节点停止了工作，将会从节点停止前开始，将commit log

恢复到memtable中。

当数据从memtable中flush到磁盘的一个SSTable中，对应的commit log数据将会被清除。

将数据存储到磁盘中的SSTables中
Memtables 和 SSTables是根据每张表来维护的。而commit log则是表之间共用的。SSTables是不可改变的，当memtable被flushed后，

是不能够重新写入的。因此，一个分区存储着多个SSTable文件。有几个其他的SSTable 结构存在帮助读操作。

对于每一个SSTable，Cassandra 创建了这些结构:

Data(Data.db)
SSTable的数据

Primary Index(Index.db)
行index的指针，指向文件中的位置

Bloom filter (Filter.db)
一种存储在内存中的结构，在访问磁盘中的SSTable之前,检查行数据是否存在memtable中

Compression Information(CompressionInfo.db)
保存未压缩的数据长度，chunk的起点和其他压缩信息。

Statistics(Statistics.db)
SSTable的内容统计数据元数据。

Digest(Digest.crc32, Digest.adler32, Digest.sha1)
保存adler32 checksum的数据文件

CRC (CRC.db)
保存没有被压缩的文件中的chunks的CRC32

SSTable Index Summary(SUMMARY.db)

存储在内存中的的分区索引的一个样例。

SSTable Table of Contents(TOC.txt)
存储SSTable TOC 中所有的组件的列表。

Secondary Index(SL_.*.db)
内置的secondary index。每个SSTable可能存在多个SIs中。

SSTables是存储在磁盘中的文件。SSTable文件的命名从Cassandra 2.2开始后发生变化为了

缩短文件路径。变化发生在安装的时候,数据文件存储在一个数据目录中。对于每一个keyspace,

一个目录的下面一个数据目录存储着一张表。例如,

/data/data/ks1/cf1-5be396077b811e3a3ab9dc4b9ac088d/la-1-big-Data.db 代表着

一个数据文件.ks1 代表着keyspace 名字为了在streaming或者bulk loading数据的时候区分

keyspace。一个十六进制的字符串，5be396077b811e3a3ab9dc4b9ac088d在这个例子中，被加到

table名字中代表着unique的table IDs.

Cassandra为每张表创建了子目录，允许你可以为每个table创建syslink，map到一个物理驱动或者数据

磁盘中。这样可以将非常活跃的表移动到更快的媒介中，比如SSDs,获得更好的性能，同时也将表拆分到各个

挂载的存储设备中，在存储层获得更好的I/O平衡。

数据是如何维护

Cassandra 写入过程中将数据存入到的文件叫做SSTables.SSTables 是不可更改的。Cassandra在写入或者更新时不是去覆盖已有的行，而是写入一个带有新的时间戳版本的数据到新的SSTables中。Cassandra删除操作不是去移除数据，而是将它标记为墓碑。

随着时间的推移，Cassandra可能会在不同的SSTables中写入一行的多个版本的数据。每个版本都可能有独立的不同的时间戳的列集合。随着SSTables的增加，数据的分布需要收集越来越多的SSTables来返回一个完整的行数据。

为了保证数据库的健康性，Cassandra周期性的合并SSTables,并将老数据废弃掉。这个过程称之为合并压缩。

合并压缩

Cassandra 支持不同类型的压缩策略，这个决定了哪些SSTables被选中做compaction，以及压缩的行在新的SSTables中如何排序。每一种策略都有自己的优势，下面的段落解释了每一种Cassandra’s

compaction 策略。

尽管下面片段的开始都介绍了一个常用的推荐，但是有很多的影响因子使得compaction策略的选择变得很复杂。

SizeTieredCompactionStrategy(STCS)

建议用在写占比高的情况。
当Cassandra 相同大小的SSTables数目达到一个固定的数目(默认是4),STCS 开始压缩。STCS将这些SSTables合并成一个大的SSTable。当这些大的SSTable数量增加，STCS将它们合并成更大的SSTables。在给定的时间范围内，SSTables大小变化如下图所示

STCS 在写占比高的情况下压缩效果比较好，它将读变得慢了，因为根据大小来合并的过程不会将数据按行进行分组，这样使得某个特定行的多个版本更有可能分布在多个SSTables中。而且，STCS不会预期的回收删除的数据，因为触发压缩的是SSTable的大小，SSTables可能增长的足够快去合并和回收老数据。随着最大的SSTables 大小在增加，disk需要空间同时去存储老的SSTables和新的SSTables。在STCS压缩的过程中可能回超过一个节点上典型大小的磁盘大小。

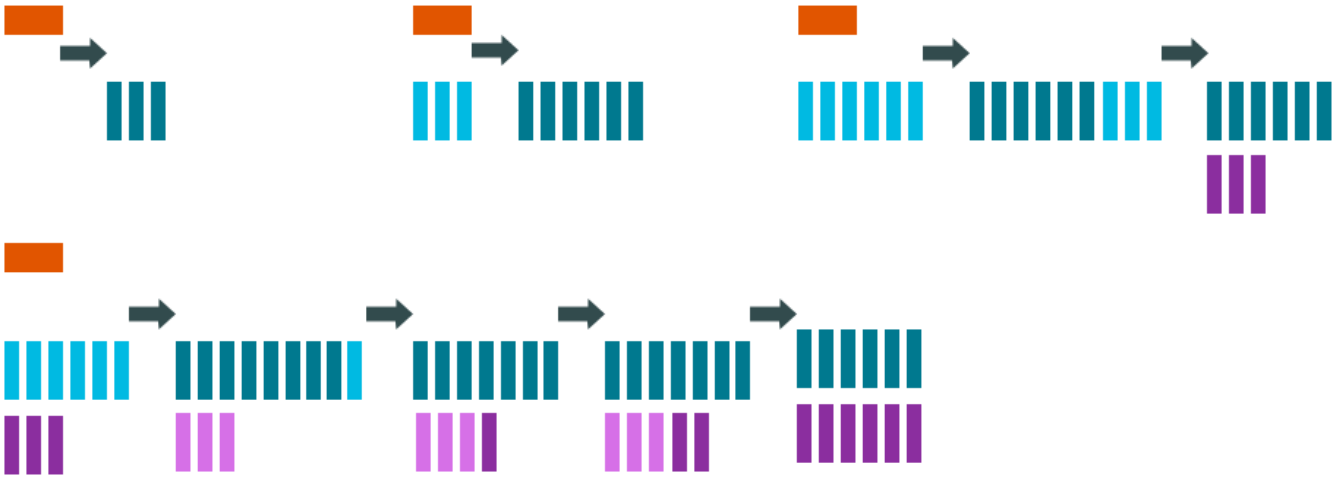
- 优势：写占比高的情况压缩很好
- 劣势：可能将过期的数据保存的很久，随着时间推移，需要的内存大小随之增加。

LeveledCompactionStrategy(LCS)

建议用在读占比高的情况。

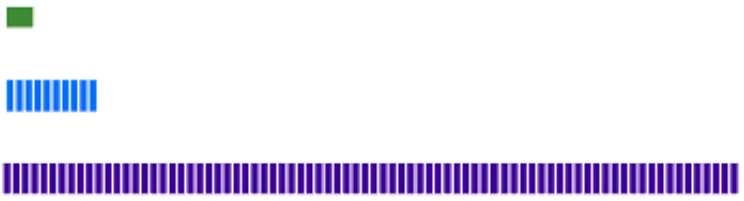
LCS减少了STCS多操作的一些问题。这种策略是通过一系列层级来工作的。首先，memtables中数据被flush到SSTables是第一层(L0)。LCS 压缩将这些第一层的SSTables合并成更大的SSTables L1。

Leveled compaction —— 添加SSTables



高于L1层的SSTables会被合并到一个大小大于等于sstable_size_in_md（默认值:160MB）的SSTables中。如果一个L1层的SSTable存储的一部分数据大于L2，LCS会将L2层的SSTable移动到一个更高的等级。

许多插入操作之后的Leveled compaction



在每个高于L0层的等级中，LCS创建相同大小的SSTables。每一层大小是上一层的10倍，因此L1层的SSTable是L0层的10倍，L2层是L0层100倍。如果L1层的压缩结果超过了10倍，超出的SSTables就会被移到L2层。

LCS压缩过程确保了从L1层开始的SSTables不会有重复的数据。对于许多读，这个过程是的Cassandra能够从一个或者二个SSTables中获取到全部的数据。实际上，90%的都能够满足从一个SSTable中获取。因为LCS不去compact L0 tables。资源敏感型的读涉及到多个L0 SSTables的情况还是会发生。

高于L0层，LCS需要更少的磁盘空间去做压缩——一般是SSTable大小的10倍。过时的数据回收的更频繁，因此删除的数据占用磁盘空间更少的比例。然而，LCS 压缩操作使用了更多的I/O操作，增加了节点的I/O负担。对于写占比高的情况，使用这种策略的获取的报酬不值得付出的I/O操作对性能造成损失的代价。在大多数情况下，配置成LCS的表的测试表明写和压缩I/O饱和了。

注：
当使用LCS策略bootstrapping一个新节点到集群中，Cassandra绕过了compaction操作。初始的数据被直接搬到正确的层级因为这儿没有已有的数据，因此每一层没有分片重复。获取更多的信息，查看

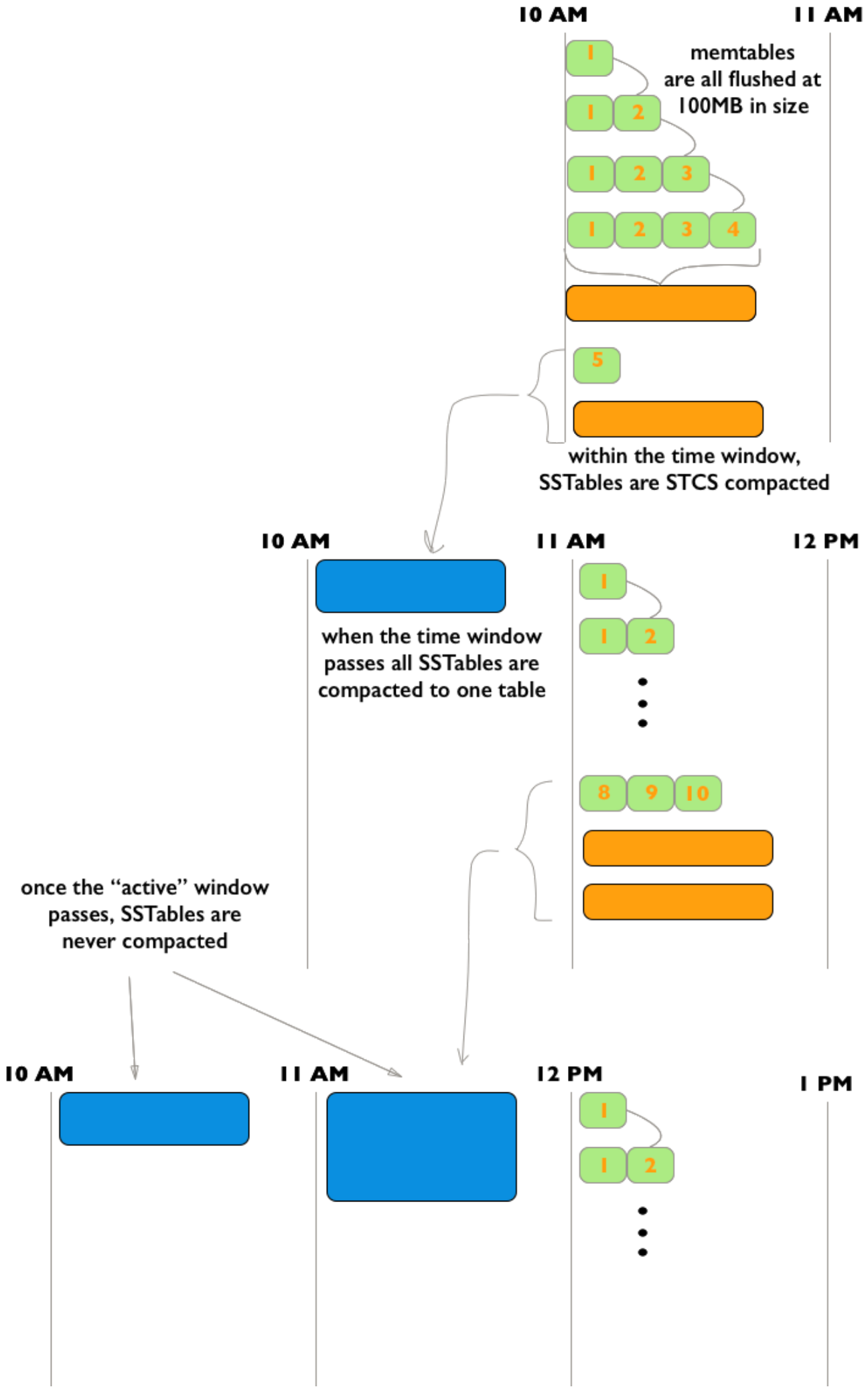
优势： 磁盘空间的要求容易预测。读操作的延迟更容易预测。过时的数据回收的更及时。

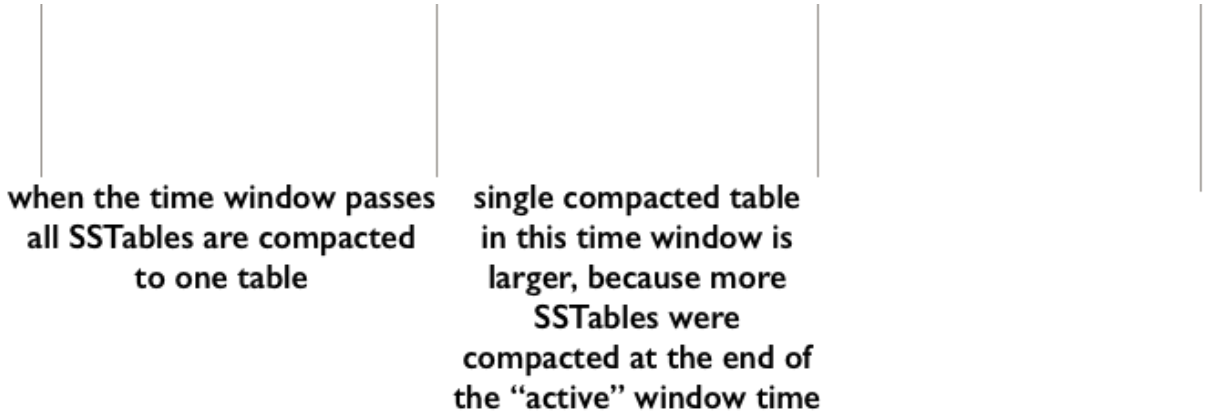
劣势: 更高的I/O使用影响操作延迟。

TimeWindowCompactionStrategy(TWCS)

建议用在时间序列且设置了TTL的情况。

TWCS有点类似于简单设置的DTCS。TWCS通过使用一系列的时间窗口将SSTables进行分组。在compaction阶段，TWCS在最新的时间窗口内使用STCS去压缩SSTables。在一个时间窗口的结束，TWCS将掉落在这个时间窗口的所有的SSTables压缩层一个单独的SSTable，在SSTable maximum timestamp基础上。一旦一个时间窗口的主要压缩完成了，这部分数据就不会再有进一步的压缩了。这个过程结束之后SSTable开始写入下一个时间窗口。


10 AM
11 AM
memtables are all flushed at 100MB in size
within the time window, SSTables are STCS compacted
10 AM
11 AM
12 PM
when the time window passes all SSTables are compacted to one table
once the "active" window passes, SSTables are never compacted
10 AM
11 AM
12 PM
1 PM



如上图所示，从上午10点到上午11点，memtables flush到100MB的SSTables中。使用STCS策略将这些SSTables压缩到一个更大的SSTables中。在上午11点的时候，这些SSTables被合并到一个单独的SSTable,而且不会被TWCS再进行压缩了。在中午12点，上午11点到中午12点创建的新的SSTables被STCS进行压缩，在这个时间窗口结束的时候，TWCS压缩开始。注意在每个TWCS时间窗口包含不同大小的数据。

注：可以在这里看动画解释。

TWCS配置有两个主要的属性设置

- compaction_window_unit：时间单位，用来定义窗口大小(milliseconds,seconds,hours等等)
- compaction_window_size：每个窗口有多少单元(1,2,3等等)

上面配置的一个例子：compaction_window_unit = ‘minutes’,compaction_window_size = 60

优势:用作时间序列数据，为表中所有数据使用默认的TTL。比DTCS配置更简单。

劣势：不适用于时间乱序的数据，因为SSTables不会继续做压缩，存储会没有边界的增长，所以也不适用于没有设置TTL的数据。相比较DTCS，需要更少的调优配置。

DateTieredCompactionStrategy(DTCS)

Cassandra 3.0.8/3.8 中弃用了。

DTCS类似于STCS。但是STCS压缩事基于SSTable 大小，而DTCS是基于SSTable年纪(在一个SSTable中，每一列都标记着一个写入的时间戳。)对于一个SSTable的年纪，DTCS使用SSTable中任意列中的oldest(最小的)时间戳。

配置DTCS时间戳

哪一种压缩策略最好

为了实现最好的压缩策略:

1. Review 你应用的需求
2. 配置表使用最合适的策略
3. 测试压缩策略

下面的问题基于有Cassandra 开发者和使用者的使用经验以及上述描述的策略。

你的表处理的是时间序列的数据吗
如果是的，你最好的选择是TWCS 或者DTCS。想要看更多的细节，参考上面的描述。

如果你的表不是局限于时间序列的数据，选择就变得更加复杂了。下面的问题可能会给你其他的考虑去做选择。

你的表处理读比写多，或者写多于读吗
LCS 是一个好的选择，如果表处理的读是写的两倍或者更多——尤其是随机读。如果读的比重和写的比重类似。LCS对性能的影响可能不值得获取的好处。要意识到LCS可能会被一大批写很快覆盖掉。

表中的数据是否经常改变
LCS的一个优势在于它将相关的数据都保持在小范围的SSTables中，如果你的数据是不可更改的或者不是经常做upserts.STCS可以实现相同类型的分组而不用付出LCS在性能上影响。

是否需要预测读和写等级
LCS 保证SSTables在一个可预测的大小和数量中。例如，如果你的表读/写比例比较小，对于读，期望获得一致的服务层面的一致性，或许值得付出写性能的牺牲去确保读速率和延迟在一个可预测的等级。而且可以通过水平扩展（添加更多的节点）来克服掉写入牺牲。

表会通过一个batch操作插入数据吗
batch 写或batch读，STCS表现的都比LCS好。batch过程造成很少或没有碎片，因此LCS的好处实现不了，batch操作可以通过LCS配置来覆盖。

系统有受限的磁盘空间吗
LCS处理磁盘空间比STCS更高高效：除了数据本身占用的空间，需要大约10%的预留空间。STCS和DTCS需要更多，在某些情况下，差不多需要50%。

系统是否到达I/O限制
LCS相比较DTCS 或者STCS，对I/O更加的敏感。换成LCS，可能需要引入更多的I/O来实现这种策略优势。

数据如何更新？

Cassandra将每个新行都当做一个upsert:如果一个新行和一个已有的行有相同的primary key，Cassandra就会对这已有的行进行更新操作。

写数据时，Cassandra将每个新行都添加到数据库中，而不去检查是否有重复的记录存在。这个方法将有可能导致相同行在数据库中有多个版本存在。想要了解更多的信息，请查看数据如何写入。

间断性的，存储在内存中的行会被冲刷到磁盘中的叫做SSTables结构文件中。在一个确定的间隔中，Cassandra将小的SSTables compacts成大的SSTables。如果在这个过程中，Cassandra遇到2个或更多版本的同一行数据。Cassandra只会将最新版本的数据写到新的SSTable。compaction结束后，Cassandra会删除掉原来的SSTables，以及过时的行。

大部分的Cassandra环境会将每一行的副本存在两个或多个节点上。每个节点都独立的执行compaction。这意味着即使一个过时版本的行数据在某个节点上被删除了，可能仍然存在于其他节点上。

这就是为什么Cassandra会在读操作过程中会执行另外一轮比较。当一个客户端根据一个特定的主键请求数据时，Cassandra会从一个或多个副本中产生多个版本的行数据。带有最新的时间戳的数据是唯一会被返回到客户端的数据(“最后写赢策略”)。

注：数据库操作可能只会更新一行数据的部分字段，因此有些版本的行只包括一些列，而不是全部。在写操作和compaction阶段，Cassandra从各个部分更新收集一个完整的行数据。每一列都使用最新的版本。

笔者注：

> Cassandra更新数据不同于一般数据库，不会更新原来的数据。只会追加新的数据。因此数据库中可能会有多个版本的数据，会在compaction和repair阶段去修复多个版本的行数据。以及在读阶段从多个版本的行数据中获取最新的那行数据。
> 另外cassandra每行每个字段都有一个时间戳，所以行版本都是取每列的最新版本。

数据是如何删除的

Cassandra删除数据的过程是为了提高性能，以及和Cassandra一些内置的属性是实现数据的分布，和故障容忍。

Cassandra将一个删除视为一个插入或者upsert。DELETE命令中添加的数据会被加上一个删除标记，叫做墓碑。墓碑标记走的也是Cassandra写过程，会被写到一个或多个节点的SSTables。墓碑最主要的不同点在于:它有一个内置的过期日期/时间。在它过期的时候(更多细节在下面),墓碑会作为Cassandra’ compaction的一部分过程删除掉。

在分布式系统中做删除操作

在一个多节点的集群中，Cassandra可以在2个或者更多的节点上存储一份数据的多个副本。这样可以防止数据丢失，但是会使得删除过程变得复杂。如果一个节点接收到删除，在它存在本地，节点将这个特定记录标记为墓碑，然后尝试将墓碑传递给其他包含这个记录的节点。但是如果某个存有该数据的节点在这个时间点没有应答，不能够马上收到墓碑，因此它仍会有这些记录的删除前的版本。如果在这个节点恢复过来之前，集群中其他被标记的墓碑数据都已经被删除了，Cassandra会将恢复过来的这个节点上的这个记录作为新的数据，并且将它传送到集群的其他节点。这种类型的删除大但是仍存活下来的记录叫做zombie

为了阻止zombies重新出现，Cassandra给每个墓碑一个grace时间段。grace时间段的目的就是给没有应答的节点时间去回复和正常处理墓碑。如果一个客户端在grace period多墓碑数据写入了一个新的更新。Cassandra会覆盖掉墓碑。如果一个客户端在grace时间段，读墓碑记录，Cassandra会无视墓碑，如果可能的话从其他副本中读取记录。

当一个无应答的节点恢复过来，Cassandra使用hinted handoff来恢复节点down期间错过的数据的更改。Cassandra不会去为墓碑数据恢复更改在它grace period期间。但是如果在grace period结束后，节点还

没有恢复，Cassandra会错过删除。

当墓碑的grace period结束后，Cassandra会在compaction阶段删除墓碑。

一个墓碑的grace period是通过设置`gc_grace_seconds`属性来设置的。默认值是86400秒(10天)。每个表这个属性可以单独设置。

更多关于Cassandra删除

细节：

- 墓碑数据过期的日期/时间是它创建的日期/时间加上表属性`gc_grace_seconds`值。
- Cassandra也支持批量插入和更新。这个过程通过会带来恢复一个记录的插入的风险，当这条记录已经被集群的其他节点移除掉了。Cassandra不会在grace period期间，为一个墓碑数据恢复批量更改。
- 在一个单节点的集群中，你可以设置`gc_grace_seconds`的值为0
- 为了完全阻止zombies记录的再出现，在节点恢复过来后、以及每个表的`gc_grace_seconds`，跑 nodetool repair
- 如果一个表的所有记录在创建的时候设置了TTL，以及所有的都允许过期，且不能手动删除，就没有必要为这张表定期跑nodetool repair
- 如果使用了SizeTieredCompactionStrategy 或者 DateTieredCompactionStrateg，你可以通过手动开启compaction立即删除掉墓碑。
  小心：
  如果你强制compaction,Cassandra可能从所有的数据中创建一个非常大的SSTable。Cassandra在很长的时间段中不会再触发另外一个compaction。在强制compaction期间创建的SSTable数据可能会变得非常过时在一个长的没有compaction阶段。
- Cassandra 允许你为整张表设置一个默认的`_time_to_live`属性。列和行都被标记为一个TTLs;但是如果一条记录超过了表级别的TTL，Cassanda会立即将它删除，而没有标记墓碑或者compaction过程。
- Cassandra支持通过DROP KEYSPACE 和 DROP TABLE立即执行删除。

indexes是如何存储和更新的

Secondary indexes是被用来过滤非primary key列的表查询。例如，一个表存储cyclist names 和 ages。使用cyclist的last name作为主键，可能会有一个age字段的secondary index，使得能够允许根据年龄来查询。查询匹配非主键的列是反范式的，因为这样的查询经常会导致表连续数据切片产生。

如果一个表根据last names时存储数据，表将会分成多个parittions存储在不同的节点上。基于last names的某个特定范围的查询，比如所有的last name 为Matthews名字的cyclists，会导致表的顺序查

询，但是基于age的查询，比如哪些cyclists 28岁，会导致所有节点都会去查一个值。不是primary keys在数据存储时是乱序的。这种根据非主键的查询会导致全partitions的扫描。扫描所有的partitions会导致非常高昂的读延迟，是不被允许的。

Secondary indexes 可以为表的某一个列构建。这些indexes通过一个后台进程都存储在每个节点的本地的一个隐藏表中。如果一个secondary index被用在一个查询中而且没有限制一个特定的partition key,这样的query同样会有高昂的查询延迟，因为所有的节点都得查询。带这些参数的查询只允许查询选项为ALLOW FILTERING。这个选项不适用生产环境。如果一个查询包括一个partiton key的条件和secondary index列条件，这样的查询才会成功，因为会被转换为单个节点的分区。

然而这种技术并不能保证零麻烦的索引，因此需要知道什么时候用/不用index。像上面描述的例子，age列上的index可以使用，但是更好的解决方案是创建一个物化视图或者额外的一张根据age排序的表。

和关系型数据一样，维护index需要处理时间和资源，因此不必要的indexes应该要避免。当一个列被更新的时候，它的index同样需要更新。如果当一个老的列值还存在于memtable中，这个经常发生在重复的更新某些行，Cassandra会将对应的过时的index删除;否则老的index entry仍然会被compaction清除掉。如果读操作在compaction清理之前看到一个过时的index entry，reader线程会设置它无效。

注:

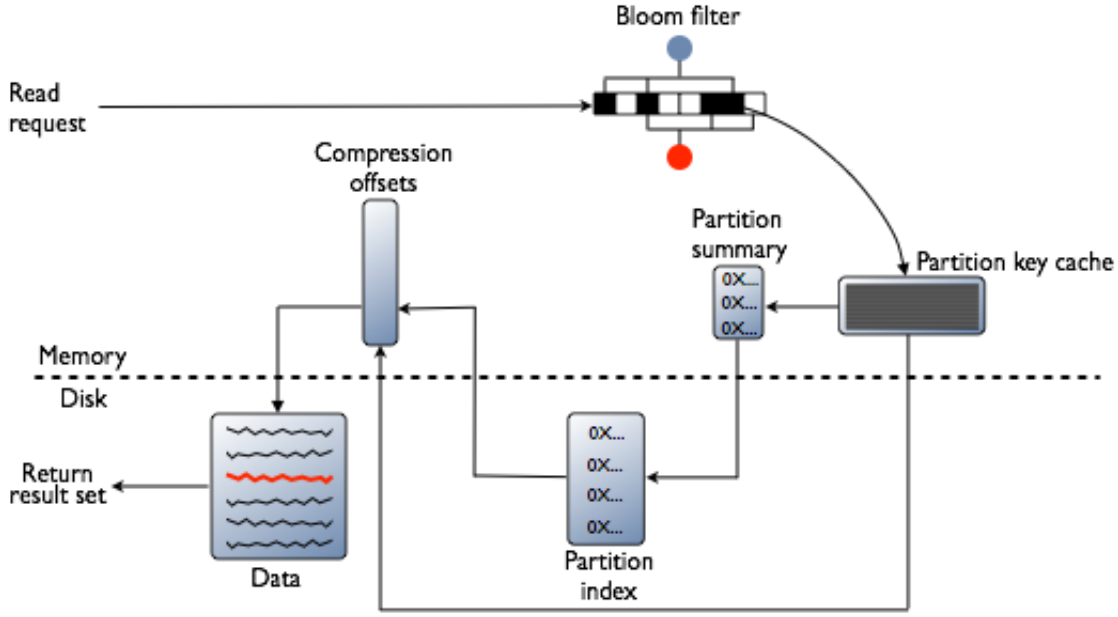
数据是如何读的

为了满足读，Cassandra必须要从存活的memtable和潜在的多个SSTables中联合查询结果。

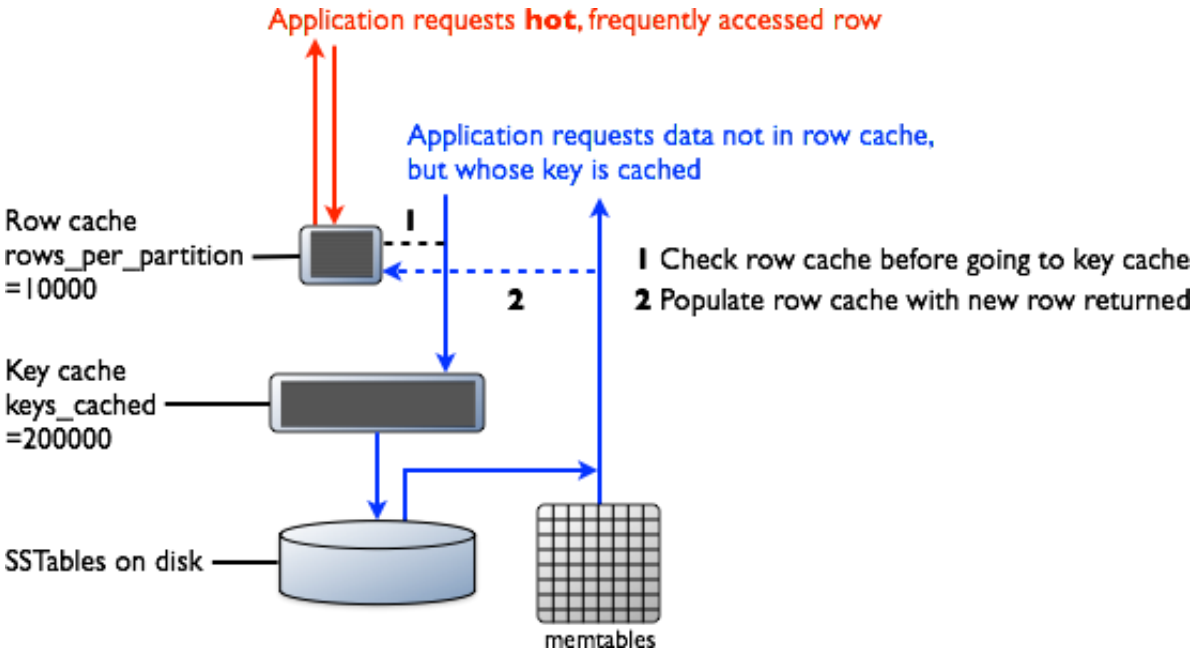
Cassandra在读的过程中为了找到数据存储在何处，需要在好几个阶段处理数据。从memtables开始，到SSTables结束。

- 检查 memtable
- 如果enabled了,检查row cache
- 检查Bloom filter
- 如果enable了,检查partition key 缓存
- 如果在partition key缓存中找到了partition key,直接去compression offset mao中，如果没有，检查 partition summary
- 根据compression offset map找到数据位置
- 从磁盘的SSTable中取出数据

读请求流程图



行缓存和键缓存请求流程图



MemTable

如果memtable有目标分区数据，这个数据会被读出来并且和从SSTables中读出来的数据进行合并。SSTable的数据访问如下面所示的步骤。

Row Cache

通常意义上，对于任何数据库，当读的大部分数据都在内存中读取的时候是非常快的。操作系统的页缓存是最有利提升性能的，经过行缓存对于读占比高的应用有一定性能提升，如读操作占95%。Row Cache对于写占比高的系统是禁用的。如果开启了row cache.就会将一部分存储在磁盘的SSTables数据存储在了内存中。在Cassandra2.2+，它们被存储在堆外内存，使用全新的实现避免造成垃圾回收对JVM造成压力。存在在row cache的子集数据可以在特定的一段时间内配置一定大小的内存。row cache使用LRU(least-recently-userd)进行回收在申请内存，当cache满的时候。

row cache的大小是可以配置的，值是可以存多少行。配置缓存的行数是一个非常有用的功能，使得类似’最后的10条’查询可以很快。如果row cache 开启了。目标的数据就会从row cache中读取，潜在的节省了对磁盘数据的两次检索。存储在row cache中的数据是SSTables中频繁被访问的数据。存储到row cache中后，数据就可以被后续的查询访问。row cache不是写更新。如果写某行了，这行的缓存就会失效，并且不会被继续缓存，直到这行被读到。类似的，如果一个partition更新了，整个partition的cache都会被移除，但目标的数据在row cache中找不到，就会去检查Bloom filter。

Bloom Filter

首先，Cassandra检查Bloom filter去发现哪个SSTables中有可能有请求的分区数据。Bloom filter是存储在堆外内存。每个SSTable都有一个关联的Bloom filter。一个Bloom filter可以建立一个SSTable没有包含的特定的分区数据。同样也可以找到分区数据存在SSTable中的可能性。它可以加速查找partition key的查找过程。然而，因为Bloom filter是一个概率函数，所以可能会得到错误的结果，并不是所有的SSTables都可以被Bloom filter识别出是否有数据。如果Bloom filter不能够查找到SSTable，Cassandra会检查partition key cache.

Bloom filter 大小增长很适宜，每10亿数据1~2GB。在极端情况下，可以一个分区一行。都可以很轻松的将数十亿的entries存储在单个机器上。Bloom filter是可以调节的，如果你愿意用内存来换取性能。

Partition Key Cache

partition key 缓存如果开启了，将partition index存储在堆外内存。key cache使用一小块可配置大小的内存。在读的过程中，每个”hit”保存一个检索。如果在key cache中找到了partition key。就直接到compression offset map中招对应的块。partition key cache热启动后工作的更好，相比较冷启动，有很大的性能提升。如果一个节点上的内存非常受限制，可能的话，需要限制保存在key cache中的partition key数目。如果一个在key cache中没有找到partition key。就会去partition summary中去找。

partition key cache 大小是可以配置的，意义就是存储在key cache中的partition keys数目。

Partition Summary

partition summary 是存储在堆外内存的结构，存储一些partition index的样本。如果一个partition index包含所有的partition keys。鉴于一个partition summary从每X个keys中取样，然后将每X个key map到index 文件中。例如，如果一个partition summary设置了20keys进行取样。它就会存储SSTable file开始的一个key,20th 个key，以此类推。尽管并不知道partition key的具体位置，partition summary可以缩短找到partition 数据位置。当找到了partition key值可能的范围后，就会去找partition index。

通过配置取样频率，你可以用内存来换取性能，当partition summary包含的数据越多，使用的内存越多。可以通过表定义的index interval属性来改变样本频率。固定大小的内存可以通过index summary capacity in mb属性来设置，默认是堆大小的5%。

Partition Index

partition index驻扎在磁盘中，索引所有partition keys和偏移量的映射。如果partition summary 已经查到partition keys的范围，现在的检索就是根据这个范围值来检索目标partition key。需要进行单次检索和顺序读。根据找到的信息。然后去compression offset map中去找磁盘中有这个数据的块。如果partition index必须要被检索，则需要检索两次磁盘去找到目标数据。

Compression offset map

compression offset map存储磁盘数据准确位置的指针。存储在堆外内存，可以被partition key cache或者partition index访问。一旦compression offset map识别出来磁盘中的数据位置，就会从正确的SStable(s)中取出数据。查询就会收到结果集。

注: 在一个分区里，所有的行查询代价并不是一致的。在一个分区的开始（第一行，根据clustering key定义）想对来说代价更小，应为没有必要执行partition-level的index。

compression offset map 每TB增长量为1~3GB。压缩的数据越多，压缩块的数量越多，压缩偏移表就会越大。Compression 默认是开启的，即使压缩过程中会消耗CPU资源。开启compression使得页缓存更加的高效,通常来说是值得的。

注:

> cassandra读取的数据是memtable中的数据和SStables中数据的合并结果。读取SSTables中的数据就是查找到具体的哪些的SSTables以及数据在这些SSTables中的偏移量(SSTables是按主键排序后的数据块)。首先如果row cache enable了话，会检测缓存。缓存命中直接返回数据。没有查找Bloom filter，查找可能的SSTable。然后有一层Partition key cache，找partition key的位置。如果有根据找到的partition去压缩偏移量映射表找具体的数据块。如果缓存没有，则要经过Partition summary,Partition index去找partition key。然后经过压缩偏移量映射表找具体的数据块。

读是如何影响写的

思考集群中写操作会对读操作有什么影响是非常重要的。compaction 策略的类型使得数据的处理是可配置的，同时会影响读的性能。使用SizeTieredCompactionStrategy或者DateTieredCompactionStrategy可能会在行被频繁更新的时候造成数据碎片化。LeveledCompactionStrategy(LCS)在这种情况下是被设计用来阻止数据碎片化的。

顶
0