Cassandra 性能优化

作者：黄金华

2017/04/05

1 Cassandra 简介

1.1 引言

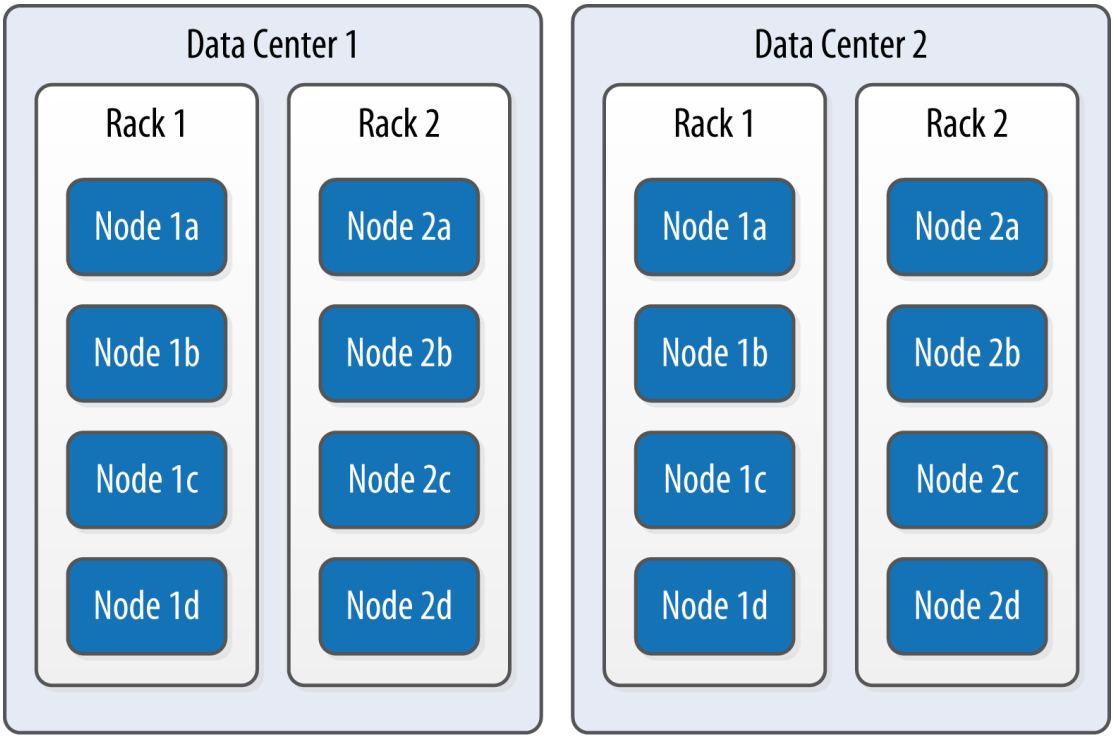
起初 Cassandra 是 Facebook 为解决内部业务大量“写”请求而开发的开源分布式 NoSQL 数据库，它具有以下特点：

- 非集中式：不存在主从角色，所有节点都可以处理读写请求，同时数据数据天然分布式部署；
- 高可靠性：数据按分机架、数据中心等复制多份，提高可靠性；
- 灵活的水平扩展性：根据一致性 Hash 算法，可任意添加或删除机器节点，对整体影响较小；
- 可调节的一致性要求：本质上是 NoSQL 的“最终一致性”，不过可在读写请求中单独设置一致性要求；
- 面向行的 K/V 存储系统：虽然存在表结构，但与关系型数据库中表有本质不同，其内部存储本质上就是 SortedMap<row_key, SortedMap<column_key, column_value>>。其中“行关键字”（row key）本质上也是“Shard key”（分区关键字）

2 Cassandra 内部机制

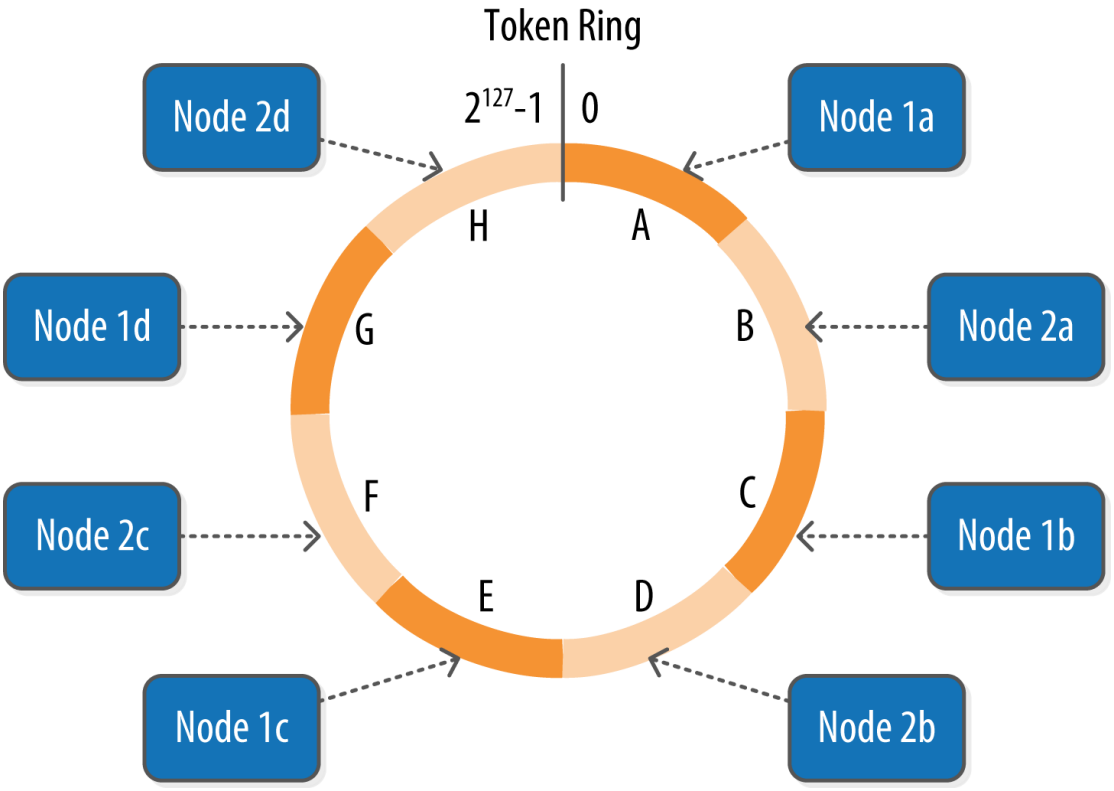
2.1 架构总览

总体上，Cassandra 是一个集群（Cluster），并由若干 Data Center 组成。而每个 Data Center 又由若干 Rack 组成，并且每个 Rack 包含若干 Node。其示例图如下：



图表 1 一个 Cluster 示例

此外，**每个 Data Center 包含一个完整的 Token Ring**。其中每个 Node 包含若干 Token(或者 Virtual nodes)。当存储数据时，当根据 Partition key（或者 Shard Key）（也就是 row key）进行 Hash 运算得到的 Token 值小于或等于该 Node 存储的 Token 值，并大于 Ring 位置上后一个 Node 存储的 Token 值时，则该 Node 节点存储对应的数据。其 Ring 示例图如下：



图表 2 Cassandra 一个 Data Center 中 Ring 示例

同时，对于分布式数据库中的一些关键问题和主要设计点，描述如下：

- **数据如何分布（Partitioner）：**
  Cassandra 本质上是 Share-nothing 架构，其数据根据 Partition key（也就是 row key）以及 Partitioner 生成的 Token 存储到对应的 Node 上。同时，为了更高效处理新的节点加入以及失效节点的删除，其 Partitioner 引入了一致 Hash(Consistent Hash)算法：
  - Murmur3Partitioners（默认）：根据 Murmur3 Hash 算法（该算法具有计算快、碰撞率低的优点）生成 tokens，其优点是数据能够相对比较均衡的分布；
  - RandomPartitioner : 选择的 MD5 Hash 算法与 Murmur3 相比，效率相对较低；
  - Order-Preserving Partitioner: 以 UTF-8 字符串为 Token 并据此排序；
  - ByteOrderedPartitioner: 与 Order-PreservingParitioner 类似，只是生成的 token 不再以字符串为单位排序，而是以原始比特为单位；

- **数据如何复制备份同步：**
  - 基本复制策略
    - SimpleStrategy：首先由 Partitioner 决定第一个副本位置，然后其他副本存储该 Node 对应的 Ring 上后续连续节点上；
    - NetworkTopologyStrategy: 首先由 Partitioner 决定第一个副本位置，然后其他副本尽量将部署到不同机架（或不同 Data Center）；

  - 每个 Data Center 单独指定对应复制策略以及副本个数 “Replication Factor”；
  - **Hinted Handoff**（解决副本节点临时不可用问题，提升可用性）：客户端可以与任何一个 Cassandra Node 通信存储数据，而当该 Node 如果实际并不包含对应的 Token 时则充当“协调者”角色，将读写请求转发给对应的 Node。而如果此时对应的 Node 暂时不可用时，该“协调者”临时将写数据存储起来，待后续得知对应的 Node 可用时则将写请求再次转发给 Node（如果指定的时间间隔内对应的 Node 依旧不可用，则 Hinted 数据可能被丢弃）。

  - **数据同步：**
    - 读同步（Read Repair）：在读取数据时根据一致性要求从多个副本读取数据，如发现副本数据不一致，则让其他节点与最新副本节点同步保持数据一致；
    - 主动同步：由“nodetool repair”命令触发，并根据 anti-entropy 协议
    - **数据同步时如何快递判两部分数据的差异部分：**
      - Anti-entryopy 对数据块生成 **Merkle Tree**(一种 Hash Tree, 叶子节点是数据块的 Hash 值，而父节点是所有子节点的 Hash 值)，通过判断 **Merkle Tree** 节点的 Hash 值则可快速定位存在差异的数据块

- **数据一致性要求：**
  - Cassandra 起初主要是为了提升“写”的能力（任何节点都可以处理“写”请求，同时数据写入内存后返回，后续再将数据连续写入磁盘，尽量避免随机 I/O 从而降低 I/O 性能），因此并不实现“强一致性”要求，而是实现“最终一致性”；

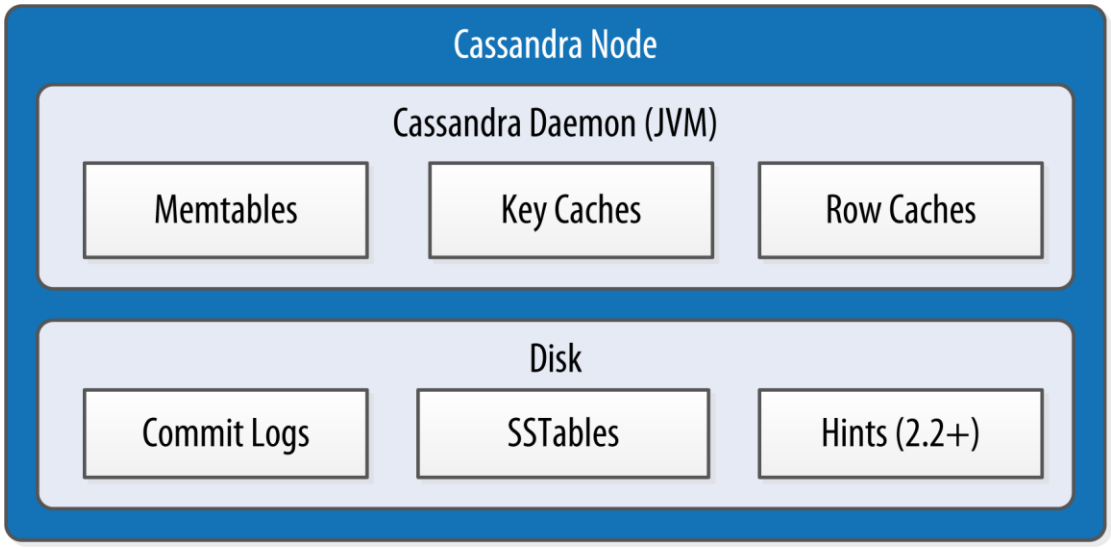
- 不过可以读写请求中单独调节一致性要求：
  - ANY: 任何一个节点响应成功则可（支持 Hinted Handoff）
  - ONE：至少一个节点响应成功则可
  - TWO: 至少两个节点响应成功则可
  - THREE: 至少三个节点响应成功则可
  - ALL : 必须所有复制节点响应成功
  - QUORUM/LOCAL_QUORUM(DC Only)：过半数复制节点响应成功则可
  - SERIAL ：用来读取"linghtweight transaction"写入后的最新数据

- **机器故障监测以及状态如何同步：**
  - 故障检测（Phi Threshold and Accrual Failure Detector）
    - 其基本思想是不再简单根据"时延"判别节点是否是"active"或"dead"，而是根据节点时延的均值（mean）来生成 Phi 值以及对应的 Phi 阈值来判别，同时可以根据网路情况调整阈值；

  - **Gossip** 协议（类似传递"流言蜚语"）：分享网络节点状态信息等，适合对"一致性"要求并不严格的网络环境：
    - Peer-to-Peer 协议
    - 初始阶段从配置文件中"seed"（种子）节点获取集群中所有节点信息；
    - 每秒随机与当前的"active"节点交换节点状态信息等；

- **如何决定集群中节点网络拓扑结构以及之间的"距离"（Snitches ）：**
  - SimpleSnitch ：适合单机模式，不考虑网络拓扑结构，不适合多个数据中心部署的情况；
  - PropertyFileSnitch：考虑机柜因素；
  - GossipingProperyFileSnitch: 考虑机柜因素，并且通过 gossip 协议与节点交换机柜和数据中心位置等信息；
  - RackInferringSnitch: 假定集群中节点 IP 地址安装一致性部署，或者说如果两个节点 IP 地址属于同一个网段，就默认两个节点属于同一个数据中心（或者机柜）；
  - CloudSnitches: 适合云计算中心部署
    - Ec2Snitch/Ec2MultiRegionSnitch: 为 Amazon's Eslastic Compute Cloud 设计；
    - GoogleCloudSnitch: Google 云平台；
    - CloudstackSnitch: 基于 Cloudstack 的云计算平台；
  - DynamicEndpointSnitch: 能够识别网络拓扑结构，并动态监测读写性能，从而为每次读写提供"最近"副本节点；

- **每个节点如何存储数据：**

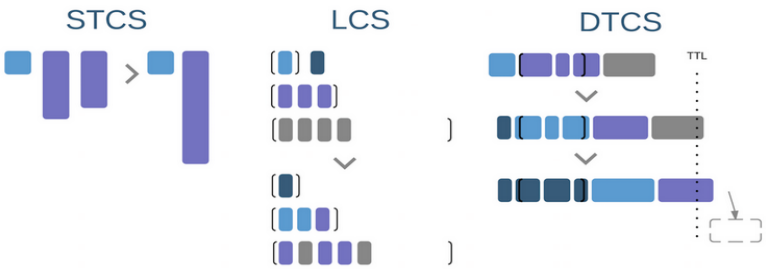


- 所有数据必须先写入“commit logs”，然后再写入 Memtables（本质是常驻内存的 write-back cache）；
- 定期（或者当 memtable 大小超过阈值或者 key 数量超过阈值）将 Memtables 压缩写入 SSTables(只读，不可更改，主要是为了将随机 I/O 转换成顺序 I/O，同时一个 table 可能存在多个 SSTables，因此需定期将多个 SSTables 压缩合并，同时删除数据时只是标记“**tombstones**”，在压缩合并时才将数据真正删除)；同时每个 SSTable 包含：
  - Datafile：安装 key 排序顺序保持的数据文件；
  - Indexfile：保持每个 Key 在 Datafile 中的位置偏移；
  - Filterfile：保存布隆过滤器对应的 key 查找树；

- **数据如何缓存**
  - Key Cache（JVM Heap）：存储 partition keys 到 row index 的映射，方便快速检索 SSTables（每个 SSTable 可能只是存储部分 keys）；
  - Row Cache（Off-heap Memory）：缓存整个行（包括对应所有列的 key/value 对，对“写”性能没有影响，只是影响“读”性能）；
  - Counter Cache：缓存计数器（主要是为了便于计数，减少 I/O 操作）

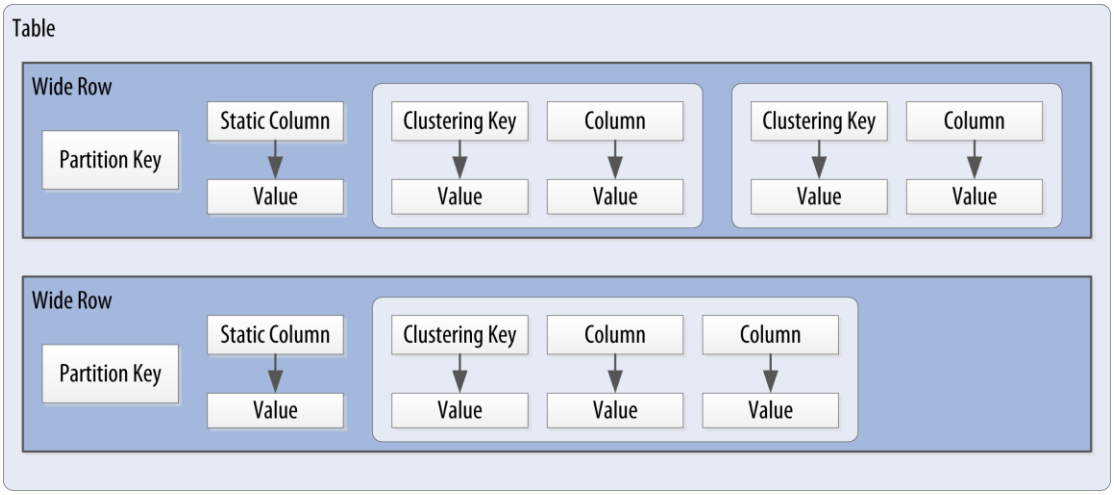
- **如何快速判别某个 key 是否在对应的 SSTable 中：**
  - **布隆过滤器（Bloom Filters）**：实际也是由几个 Hash 函数组成的一个映射，同时存在一定的误报率（没有命中则肯定不存在，而命中则小概率不存在）

- **SSTables 如何压缩**
  - SizeTieredCompactionStrategy (STCS，默认)：适合存在大量写的情况；
  - LeveledCompactionStrategy(LCS)：适合存在大量“读”的情况（每一层是上一层的 10 倍，可以保证约 90%的数据可以从一个 sstable 文件读取）；
  - DateTieredCompactionStategy (DTCS):适合时间序列或基于日期的数据；



图表 3 Compaction 示例图

2.2 数据模型



图表 4 Cassandra 数据模型图

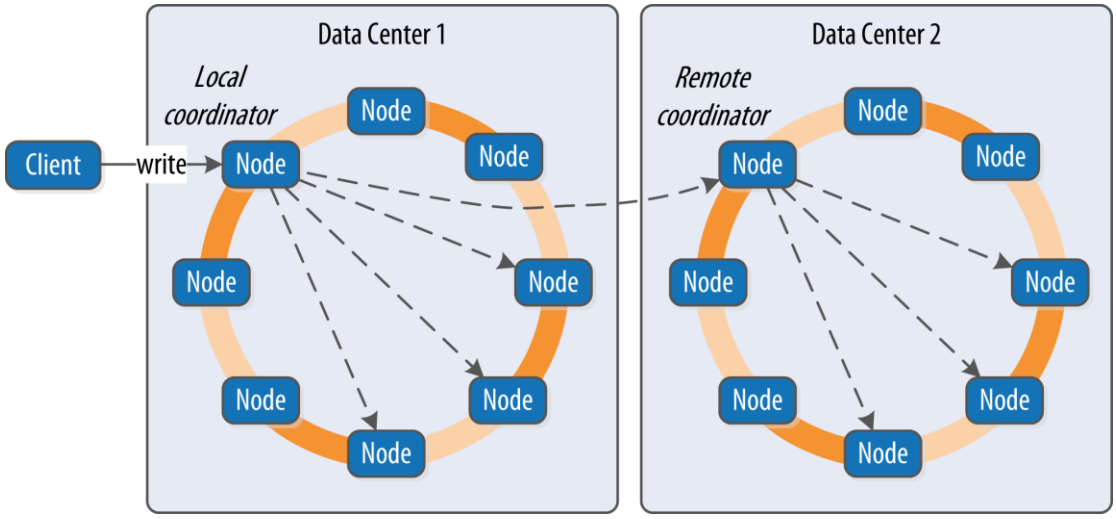
数据模型中关键概念：

- **Column**：列 key/value 对；
- **Row**：由 row key 以及若干列组成组成的 key/value 对；
- **Table**：由若干 row 组成；
- **Keyspace**： 由若干 Table 组成（类似关系数据库中的 Database）；
- **Clusters**：由若干 keyspace 组成；

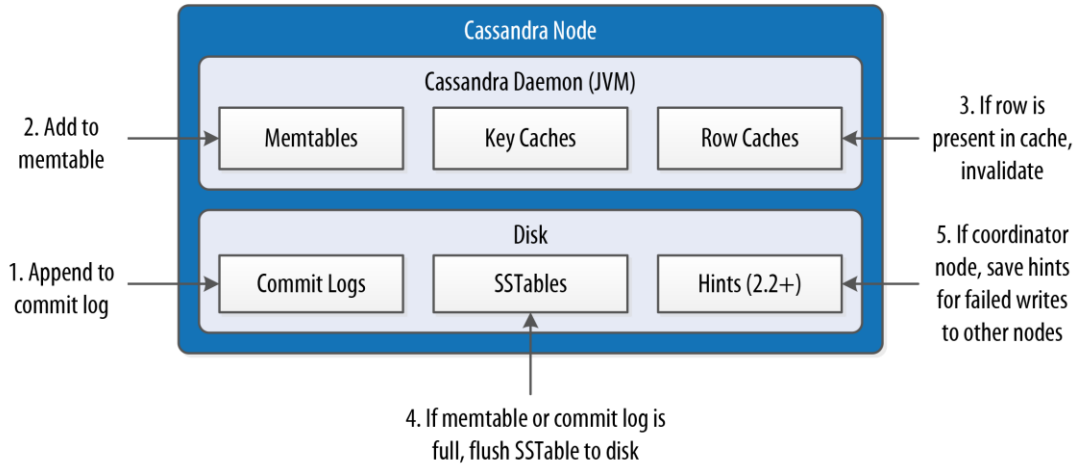
综述，其数据模型本质上是 **SortedMap<row_key, SortedMap<column_key, column_value>>**。

2.3 写流程

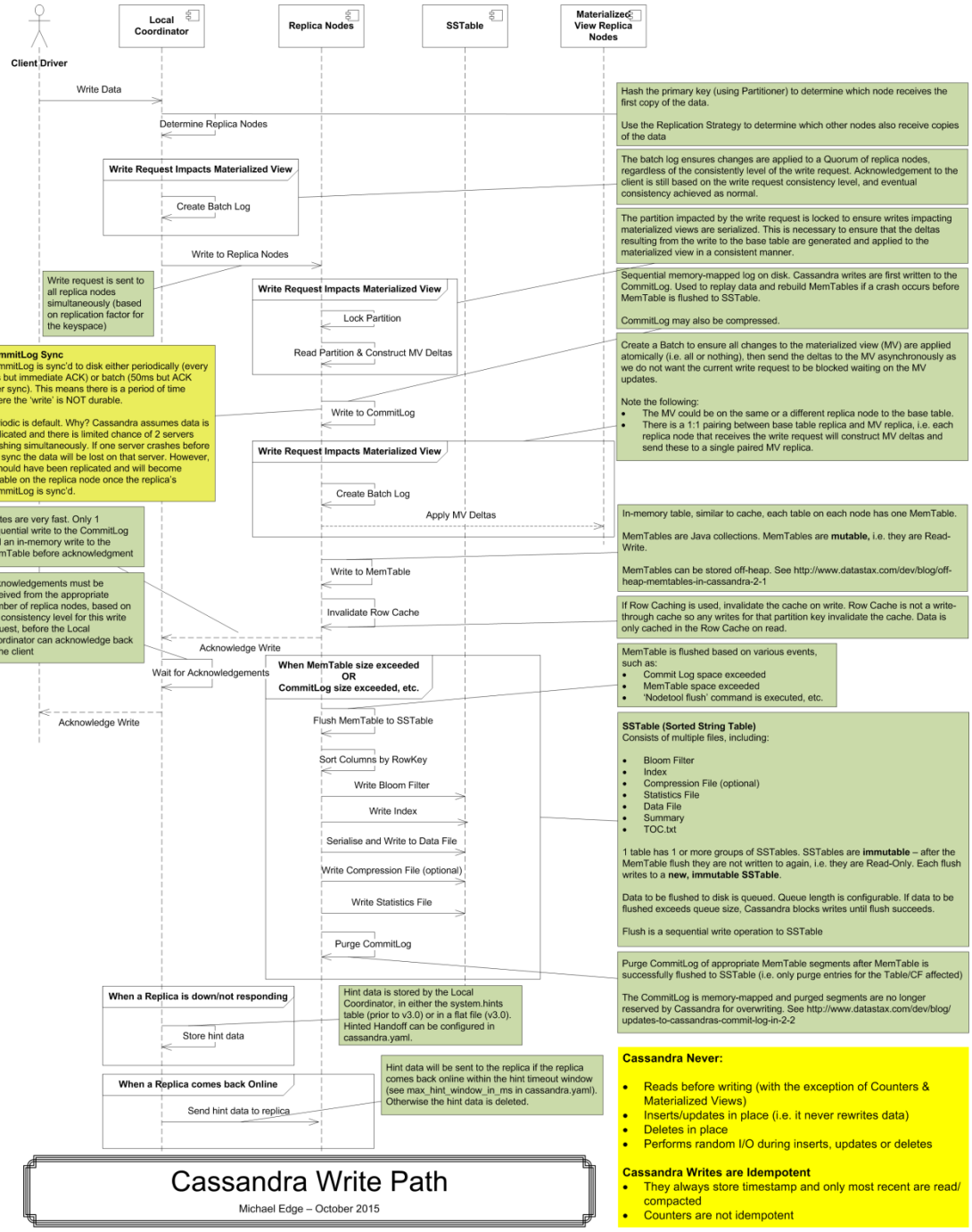
Cassandra 主要写流程如下：



图表 5 Cassandra“写”流程总览

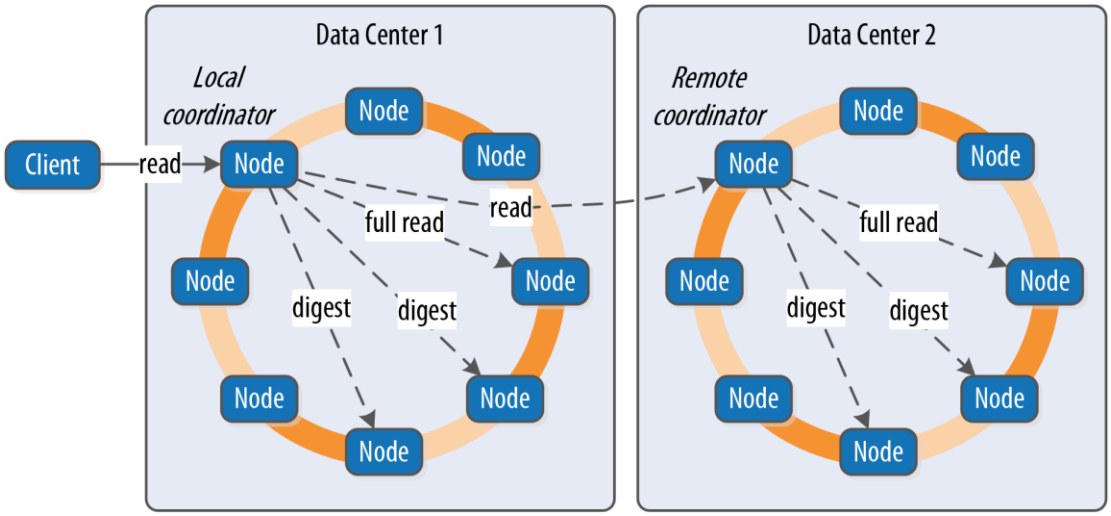


图表 6 Cassandra 单节点内部写流程图

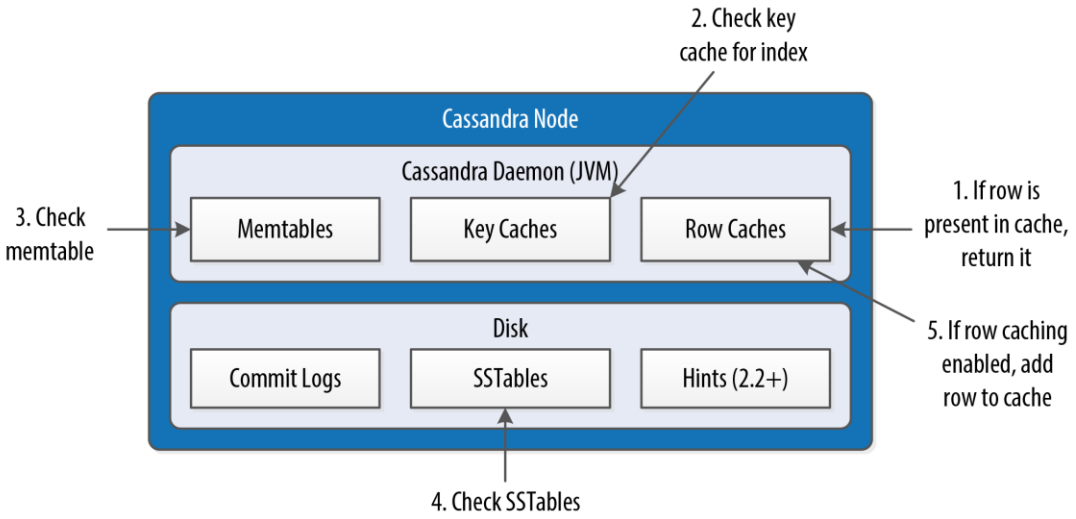


图表 7 Cassandra 写流程图

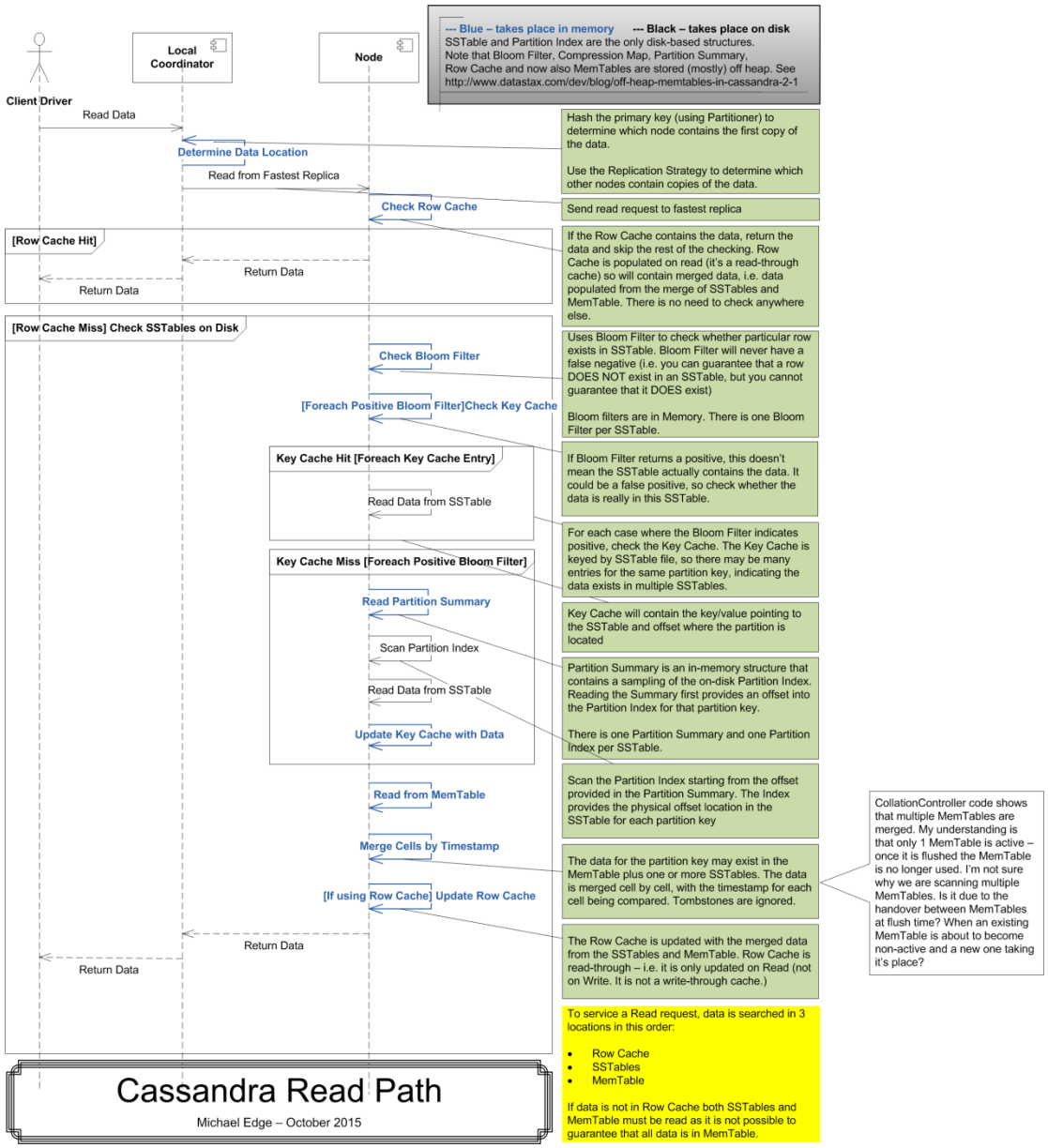
2.4 读流程



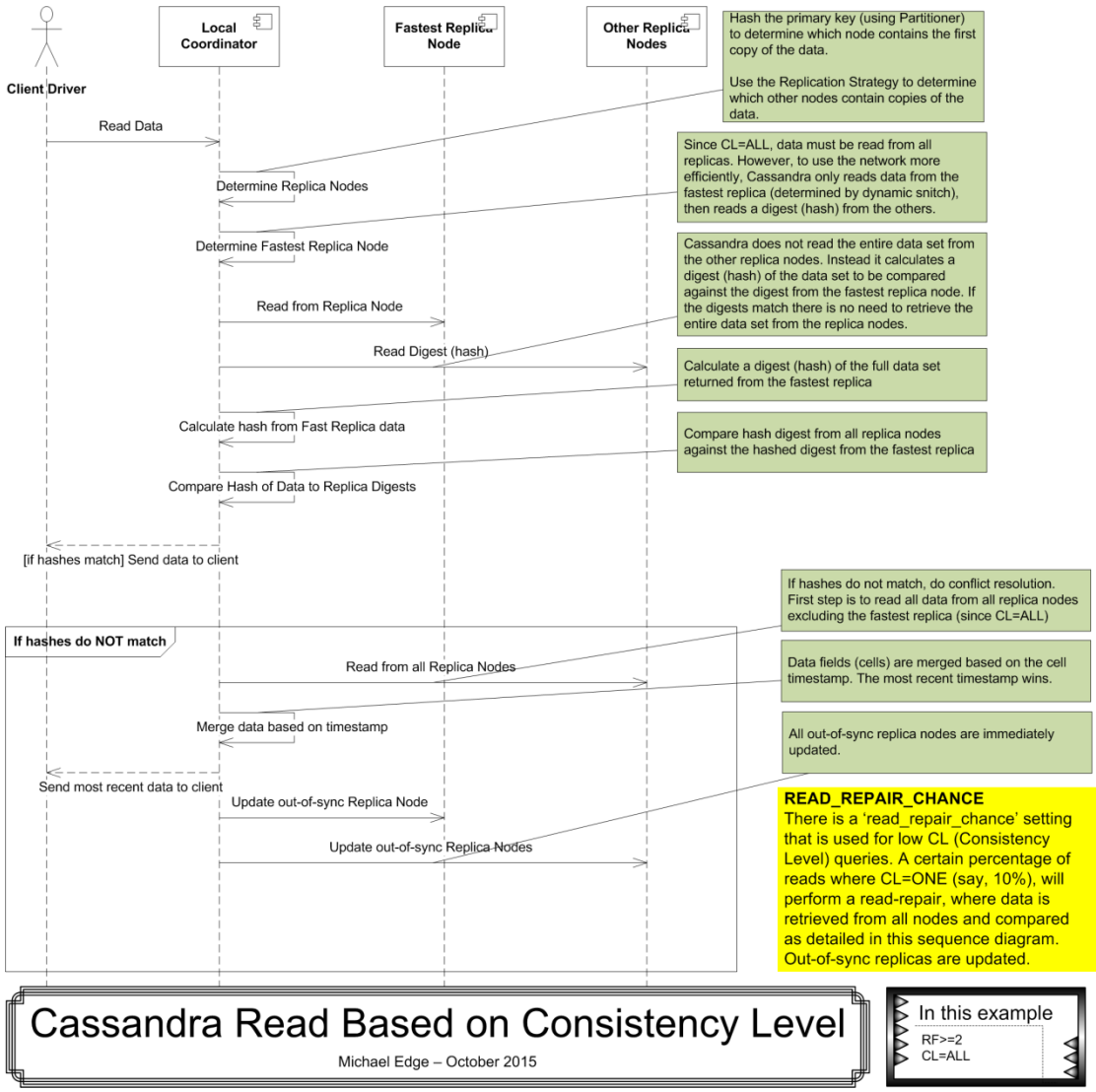
图表 8 Cassandra 读流程总览



图表 9 Cassandra 单节点内部读流程图



图表 10 Cassandra 读流程图（1）



图表 11 Cassandra 读流程图（2）

3 Cassandra 性能优化

3.1 基本方法论

对于性能优化的基本方法论，以及基本工具可参阅：https://zhuanlan.zhihu.com/p/25013051

3.2 基准测试

在性能优化之前需要衡量数据库系统性能，而对数据库而言，其性能的两个主要指标是：时延和吞吐量。为此，常需要借助基准测试来衡量系统性能，并且将时延或吞吐量作为优化目标。

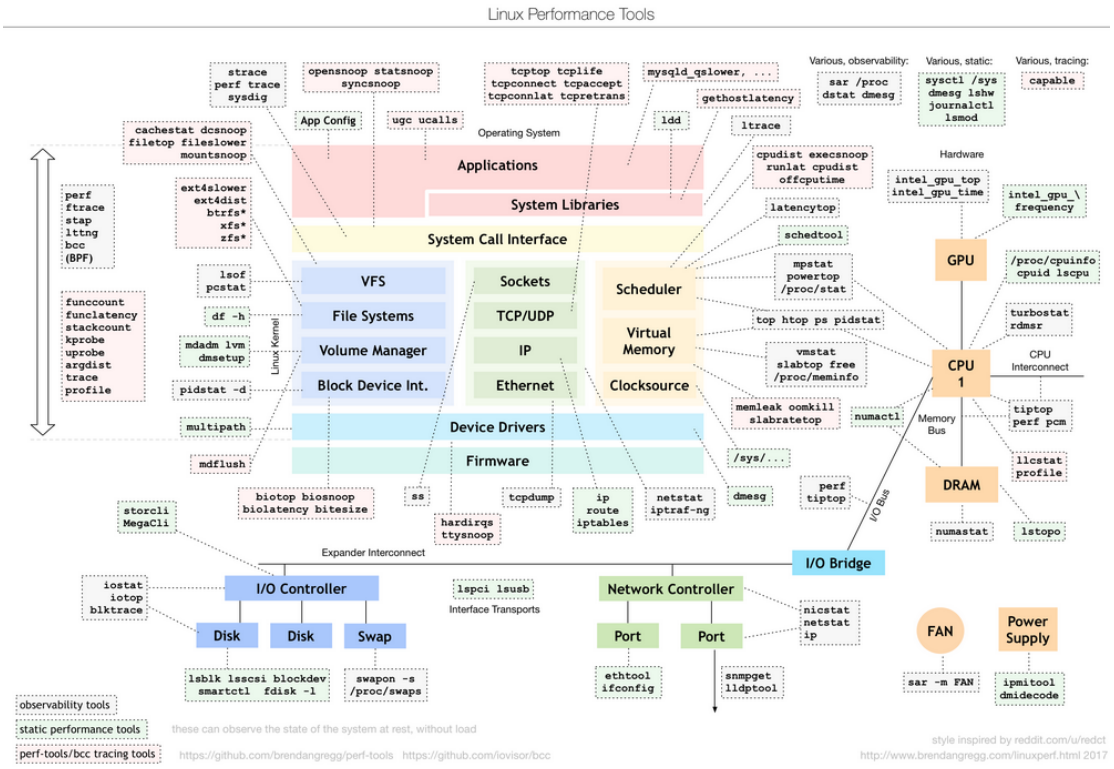
关于基准测试用例配置、部署等，可参考自动基准测试脚本（https://github.com/sjtuhjh/applications/apps/tree/master cassandra）。

3.3 性能监控

3.3.1 基础性能监控

对 CPU/Memory/Disk/Networking/Kernel 等监控，可采用 perf/ftrace/bcc/top/dstat 等多种工具，既可以支持静动跟踪，也支持动态跟踪。

具体工具可参考 http://github.com/sjtuhjh/perftools，在此不再赘述。



图表 12 Linux 性能监测工具总览

3.3.2 Cassandra 内部监控

总体性能分析：

- 通过 Jconsole （连接 Cassandra JMX 端口 7199） 查看各种性能数据
  1) 可在客户端（支持桌面系统）运行 Jconsole 然后连接服务器 Cassandra JMX 端口
  2) 或者使用命令行工具（jmxterm）也单独查看 Metrics 值；

- nodetool
  1) 通过 ”nodetool status/info/cfstats <keyspace>/tablestats <keyspace>/ compactionstats/gcstats/tpstats”等查看 Cassandra 整体性能；
  2) 或者通过“nodetool proxyhistograms / tablehistograms <keyspace>”等查看整体性能；

- 查看分析慢日志（debug.logs）；

单个 Query 性能分析：

- 内部 Trace 工具（通过 cqlsh）：
  1) 通过“cqlsh”设置“TRACING ON”然后再运行具体 QUERY 命令；
  2) 通过“TRACING OFF”关闭对应的跟踪；
  3) 或者通过运行“nodetool settraceprobability <0.0 ~ 1.0>”为集群节点部分或所有 QUERY 设置跟踪；

更多分析工具，可参考 https://github.com/sjtuhjh/perftools/tree/master/apptools/cassandra/

3.4 基础优化

3.4.1 BIOS/CPU/Memory

在对应用优化之前，要确保 BIOS/CPU/Memory 配置恰当，否则后续优化往往事倍功半。为此，需要借助一些测试工具来验证硬件功能是否工作正常（或者最优）。值得注意的是，开始测试的时候常常无法根据数据来确定是否工作正常，因此可将数据与历史优化后的数据或者对应的 X86 测试数据进行对比。常见的测试工具如下：

- **CPU**: SpecInt2006(https://github.com/sjtuhjh/applications/tree/master/apps/cpu/spec_cpu2006)
- **Memory**: lmbench/Stream
- **DISK : I/O** : fio 测试

Location	Disks	RAID Level	Purpose
/ (root)	2	1	Operating system
$PGDATA	6+	10	Database
$PGDATA/pg_xlog	2	1	WAL
Tablespace	1+	None	Temporary files

  - RAID1：将两块硬盘构成 RAID 磁盘阵列，其容量仅相当于一块磁盘，但另一块磁盘总是保持完整的数据备份。一般支持“热交换”。
  - RAID10：首先建立两个独立的 RAID1，然后将两个独立的 RAID1 再组成一个 RAID0

  建议配备 RAND10，同时支持 BBU(Battery Backup Unit)和 WRITE-BACK 功能的存储设备。这样既可以有良好的读写性能，又能防止意外情况下（例如突然断电）存储内容丢失。

- **BIOS 电源管理**：将电源管理设置成“性能”模式以便最大发挥 CPU 性能；

常见的基准测试工具也可参考 https://github.com/sjtuhjh/applications/tree/master/apps。

3.4.2 OS/Kernel

3.4.2.1文件系统

通常建议采用 XFS（其次是 EXT4）文件系统。

- Barriers I/O (Barrier 之前的所有 I/O 请求必须在 Barrier 之前结束，并持久化到非易

失性介质中。而 Barrier 之后的 I/O 必须在 Barrier 之后执行。如果本身存储设备自带电池足以预防在突然掉电时其缓存内容依然可以被正确持久化，则可以关闭 Barrier 一般提高性能)。一般文件系统挂载时提供选项可以关闭 barriers,例如：mount –o nobarrier /dev/sda /u01/data

- 打开 noatime，减少不必要的 I/O 操作，例如：mount –o nobarrier,noatime …
- 调整 vm.overcommit (将其设置为 2，尽量避免 OOM(Out of Memory killer))
- 写缓存优化（需要根据需要调整参数，当脏页刷写太频繁，会导致过多 I/O 请求，而刷写频率过低，又可能导致所占内存过多，而在不得不刷写脏页时导致应用性能波动）：
  - vm.dirty_background_ratio: 当脏页数量达到一定比例时，触发内核线程异步将脏页写入磁盘；
  - vm.dirty_ratio: 当脏页数量达到一定比例时，系统不得不将脏页写入缓存。此过程可能导致应用的 I/O 被阻塞。
- I/O 调度
  - 对于普通磁盘，建议使用 deadline 调度方式，而固态硬盘各种调度方式影响并不大；
- vm.swappiness=0 （尽量避免使用 SWAP）

3.4.2.2NUMA

默认情况下使用”numactl –interleave”，或者可以在 JVM 配置中增加“-XX:+UseNUMA”。如果运行多个 JVM，可考虑是用 numactl 将 JVM 绑定到特定 NUMA 节点。

3.4.2.3资源限制(ulimit)

一般修改/etc/security/limits.conf，包括：

- -f (file size): unlimited
- -t (cpu time): unlimited
- -v (virtual memory): unlimited
- -n (open files) : 大于 20,000
- -m (memory size): unlimited
- -u (processes/threads): 大于 20,000

3.4.2.4存储设备设置优化

3.4.2.4.1 分区对齐

分区对齐的目的是让逻辑块与物理块对齐从而减少 I/O 操作。通常可以借助 parted 命令来自动对齐，例如：

device=/dev/sdb

parted –s –a optimal $device mklable gpt

parted –s –a optimal $device mkpart primary ext4 0% 100%

其中 optimal 表示要尽量分区对齐从而达到最好性能。

3.4.2.4.2 SSD 优化

SSD 支持 TRIM 功能，打开该功能可以防止未来读写性能恶化。通过 Btrfs, EXT4, JFS 和 XFS 文件系统都支持该功能。可通过下列命令进行配置：

- 普通磁盘: 在 mount 的时候添加 discard 选项，例如：
  /dev/sda / ext4 rw,discard 0 1
- LVM (Logical Volumes)： 在/etc/lvm/lvm.conf 中添加 issue_discards=1；

3.4.2.4.3 I/O 调度算法

一般建议将/sys/block/<设备名>/queue/scheduler 设置为 deadline 或 noop。尤其是对普通机械硬盘，不要将其设置为 cfq 算法。

3.4.2.4.4 SWAP

尽量禁止使用 SWAP，从而避免性能波动。可在/etc/sysctl.conf 中添加 vm.swappiness=0 并通过 sysctl –p 使其生效。

3.4.2.5网络内核参数优化

确保网络内核有足够的资源处理连接请求：

- net.core.rmem_max
- net.core.wmem_max
- net.core.somaxconn
- net.ipv4.tcp_rmem
- net.ipv4.tcp_wmem
- net.ipv4.tcp_max_syn_backlog

3.5 JVM 配置优化

- 尽量用新进的 JAVA(例如 JAVA 8);
- CMS 相对比较稳定，但是 G1 会提升吞吐量（大约 10%）;
- 使用 G1GC (例如：JVM_OPTS=”$JVM_OPTS –XX:+UseG1GC”
- 关闭”Biased Locking” (例如： -XX:-UseBiasedLocking)
- 使能 Pre-Touch（例如： -XX:+AlwaysPreTouch）

- 使能 TLAB(Thread Local Allocation Blocks)（例如： -XX:+UseTLAB –XX:+ResizeTLAB）
- Heap Size: -Xmx/-Xms 设置相同值；
- 安装 JNA；

3.6 Cassandra 配置调优

3.6.1 Cache 优化配置

- Key Cache: 为每个 Table 设置对应的“key_cache_size_in_mb”(默认为 5% JVM Heap）
- Row Cache：通过设置“rows_per_partition”，可以提升“读性能”。其设置取决于具体“读”请求情况（一般“读请求”占比越多，提升该值优化效果越明显）
- Counter Cache: 一般采用默认值即可
- save_caches: 通过设置 key_cache_save_period/row_cache_save_period/ counter_cache_save_period 定期将 cache 写入磁盘以便下次启动能快速恢复；
- 也可通过”nodetool invalidatekeycache/…/setcachecapacity 设置相关 cache；

3.6.2 Memtables 相关配置优化

- memtable_heap_space_in_mb、memtable_offheap_space_in_mb：指定 JVM Heap 和 off-heap 内存大小来存储 memtable;
- memtable_allocation_type:
  - heap_buffers: 使用 JAVA NEW I/O(NIO) API 仅在 heap 为 Memtable 分配内存；
  - offheap_buffers: 使用 JAVA NIO 在 on-heap 和 off-heap 为 Memtable 分配内存；
  - offheap_objects: 使用操作系统提供接口分配内存，Cassandra 本身负责内存管理和垃圾回收；（当存在大量“写”时，使用此方式可以减少 JVM Heap 内存存储从而提升性能）
- memtable_flush_writers: 指定将 memtable 写入磁盘的线程数；根据磁盘 I/O 能力适当增大该值；
- memtable_flush_period_in_ms: 指定将 memtable 写入磁盘的间隔；（默认关闭，也就是说只有 Commitlog 或者 memtable 大小达到一定阈值时才触发将 Memtable 写入磁盘）

3.6.3 Commit Logs 相关配置优化

- commitlog_segment_size_in_mb:
- commitlog_total_space_in_mb:
- commitlog_compression: 指定压缩算法
- commitlog_sync:
  - perioc: 定期刷新 commit log （间隔由 commitlog_sync_period_in_ms 指定）
  - batch: 每次确保将 commit log 写入磁盘再返回写入成功（对性能会有一定影

响）

- 将 commit logs 与 data files 配置到不同磁盘；

3.6.4 SSTables 相关配置优化

- file_cache_size_in_mb: 缓存 SSTables，可适当增大；
- buffer_pool_use_heap_if_exhausted: 当为 true 时允许使用 Heap 内存当 file cache(或 buffer pool)已满时
- index_summary_capacity_in_mb: 适当增大空间存储 SSTable index 摘要信息；
- index_summary_resize_interval_in_minutes:
- min_index_interval/max_index_interval: 为每个 Table 指定允许的 index entry 最小和最大值；

3.6.5 Hinted Handoff 相关配置优化

- hinted_handoff_throttle_in_kb :（或者通过 nodetool setintedhandoff）控制用来 hint 的带宽阈值；
- hinted_directory:指定用来存储 hints 目录；
- max_hints_file_size_in_mb: 指定允许用来存储 hints 的最大磁盘空间；
- max_hint_window_in_ms: 用来指定 hint 超时的时间间隔；

3.6.6 Compaction 相关配置优化

- Compaction Strategy :
  - SizeTieredCompactionStrategy（默认） : 适合“写”请求占大多数情况；
  - LeveledCompactionStrategy: 适合“读”请求占大多数情况；
  - DateTieredCompactionStrategy: 适合经常“读”最近才“写”的数据模式；
- 可以通过 node tool set/getcompactionthreshold <table> 为每个 table 指定 compaction 阈值；
- compaction_throughtput_mb_per_sec: 设置 compaction 吞吐量阈值（为 0 是关闭阈值）
- concurrent_compactors: 用于 compaction 的并发线程数；

3.6.7 并发配置

- concurrent_read/concurrent_writes: 指定并发读写线程数，可适当根据 I/O 能力增大该值；
- max_hints_delivery_threads: 允许用来处理 hint 的最大线程数；
- memtable_flush_writes: 刷写 memtalbe 的线程数；
- concurrent_compactors: 允许 compaction 的线程数；
- native_transport_max_threads: 允许处理 CQL 请求的最大线程数；

3.6.8 网络以及 Timeout 配置

- Timeout 设置
  - read_request_timeout_in_ms: 指定“协调者”等待读请求超时时间；
  - range_request_timeout_in_ms: 指定“协调者” 等待范围读请求超时时间；
  - cas_contention_timeout_in_ms: 指定“协调者”多长时间尝试一次“lightweight 事务”
  - streaming_socket_timeout_in_ms: 要确保改值为非零值
  - 。。。
- stream_throughput_outbound_megabits_per_sec/inter_dc_stream_throughput_outbound_megabits_per_sec:
- native_transport_max_frame_size_in_mb：指定 CQL 请求最大报文大小；
- native_transport_max_concurrent_connections/native_transport_max_concurrent_connections_per_ip

3.7 数据模型优化

- 不要按照“关系型表”思维来构造数据模型，而是应该按照 Map 数据结构类型（key/value 对）来构造数据模型；
- 根据检索模型来构造数据模型行列结构；
- 必要时为提升“读”性能，使用嵌套 Key，同时允许数据重复，而不是一定要满足某种范式；
- 最优的数据模型取决于检索模型和应用场景（因此要避免不知道具体应用场景和检索请求的前提下构造数据模型）；
- 将“值”存储到列的名字（key）中（甚至其 value 为空）也是可行的；同时考虑使用“wide row”(行的 key 覆盖的粒度相对较大，从而可以将更多数据保持到列中)；
- 选择恰当的“行关键字”（row key），因为这是 Cassandra 中“shard key”（与其他数据库显著不同的一点）；
- 将主要“读”的数据和主要“写”的数据尽量分离；
- 确保行和列的关键字（key）是唯一的；
- 为行和列选择合适的数据类型（行的数据类型为 Validator，列的数据类型为 Comparator）
- 设计模型时尽量考虑其操作的幂等性；
- 如有必要，根据事务要求设计模型；
- 优先考虑使用“组合列（composite columns）”而不是“超级列（super columns）”因为“超级列”会有性能瓶颈，同时要考虑“组合列”中子列的顺序因为其关系到数据的排序；
- 优先使用嵌套的“组合列”，而不是自定义的；
- 优先使用静态“组合列”，而非动态组合列；

3.8 算法代码优化

与具体 CPU 相关的优化，待补充。

4 附录

[1] cassandra.apache.org
[2] Cassandra – A Decentralized Structured Storage System
[3] https://wiki.apache.org/cassandra