BigTable 论文笔记

1 Data Model（数据模型）

BigTable 集群是一个运行 BigTable 软件的进程集合。每个集群都为一组 table 提供服务。BigTable 中的每个 table 都是一个稀疏的分布式持久化的多维有序 map。数据以三个维度进行组织：行，列，时间戳。

1.1 Rows

BigTable 按照行关键字的字典序组织数据。BigTable 的行关键字为任意字符串。通过单个行关键字对数据的读写操作是可串行化的 (不论当前行中的数据有多少不同的列正在被读写)，这一设计使得客户端在面临针对同一行数据的并发更新时，能够更容易理解系统行为。换句话说，BigTable 以“行”作为事务一致性的单位，目前不支持跨行事务。

拥有连续关键字的的行被组织成 tablets，构成了数据分布和负载均衡的单位。这种数据组织方式使得读取小范围内的行更加高效，并且通常只需要和少数的机器进行通信。客户端可以通过选择特定的行关键字来利用这一特性，从而在数据访问中获得更好的局部性 (locality)。

1.2 Columns

列关键字被组织成“列族 (*column families*)”，构成了访问控制的单位。所有通过列族排序的数据通常是同类型的（我们把同一列族的数据一起进行压缩）。一个列族必须在该族中所有数据能够被存储在任意列关键字下之前就被显式创建。一个列族被创建后，改族的任意列关键字都可以被使用：数据可以存储在这样的列关键字之下，而不会影响到 table 的模式。我们的意图是，一个 table 中不同列族的数量要尽可能少（最多只能几百个），而且在数据操作期间几乎不去修改列族；这一限制很大程度上避免了元数据 (metadata) 变得过大。相反，一个 table 拥有的列关键字个数是不受约束的。

对整个列族的删除操作可以通过更改 table 的模式来完成，此时存储在那个列族中所有列关键字下的数据都会被删除。由于 BigTable 不支持跨行事务，如果存储在某个特定列关键字下的数据驻留在多个行中，那么它将不能被原子地删除。

列关键字的命名格式为：列族名: 限定符 (*family:qualifier*)。列族名必须是可打印的，但是限定符可以是任意字符串。

1.3 Timestamps

一个 table 中不同的 cells 可以包含同一数据的多个版本，版本信息通过时间戳进行索引。BigTable 的时间戳是 64-bit 整数，它们由 BigTable 隐式赋值，代表“实时时间”的微秒值，也可由客户端应用程序显式赋值。需要避免崩溃的应用程序必须自己生成独一无二的时间戳。一个 cell 中不同版本的数据按照时间戳降序存储，使得最新版本的数据能被最先读取。

为了减轻多个版本数据的管理负担,我们对每一个列族配有两个设置参数,以此实现 BigTable 对多版本数据的自动垃圾回收。客户端可以指定只保存最后 n 个版本的数据，或者“足够新”的数据（比如只保存最近 7 天写入的记录）。

2 Building Blocks（构件）

BigTable 是构建在其他几个 Google 基础组件之上的。BigTable 集群通常运行在一个共享机器池中，这些机器上运行着各种分布式应用程序。BigTable 依赖于 Google 集群管理系统，该系统负责共享机器的作业调度，资源管理，机器状态监控，以及机器故障处理。BigTable 进程通常和其他应用程序进程共享相同的机器。

BigTable 使用 GFS 存储日志和数据文件。GFS 是一个分布式文件系统，它通过为每个文件维护多个副本的方式来实现更高的可靠性和可用性。

BigTable 数据文件使用 SSTable 文件存储。SSTable 实现了一种持久化、不可修改的有序 KV 映射结构，key 和 value 都可以是任意字符串。SSTable 提供了 KV 数据查询和遍历操作。SSTable 包含了一组 block 序列（每个 block 的默认大小是 64KB，但是大小是可配置的）。block index（存储在 SSTable 的末尾）用于定位 block；当 SSTable 被打开时，index 被载入内存。SSTable 的查询操作是一次磁盘随机读：首先对内存中的 index 进行二分查找，找到合适的 block，然后从磁盘读出相应的 block。当然，也可以选择将整个 SSTable 映射到内存，这样就无需进行磁盘操作了。

BigTable 依赖于一个高可用的持久化分布式锁服务 Chubby。Chubby 服务由 5 个活动副本组成，其中一个被选举为 master，用于处理请求。只有在大多数副本正常运行，并且彼此能够正常通信的前提下，Chubby 服务才是可用的。在面临故障时，Chubby 使用 Paxos 算法保证副本一致性。Chubby 提供了一个由文件夹和小文件组成的命名空间。每个文件夹和文件都可被用作一个锁，并且对每一个文件的读写是原子性的。Chubby 客户端库提供了 Chubby 文件的一致性缓存。每个 Chubby 客户端都维持一个与 Chubby 服务的会话 (session)。如果一个客户端会话无法在租期内重新签订会话租约，这个会话到期后就失效了。当客户端会话到期后，它将丢失持有的锁和打开的句柄。Chubby 客户端也可以向 Chubby 文件和文件夹注册回调函数，一旦它们的状态发生改变，或者会话到期，Chubby 客户端就可以得到通知。

BigTable 使用 Chubby 实现各种任务：确保在任何时间都至少有一个活跃的 master；存储 BigTable 数据的引导程序 (bootstrap) 的位置；发现 tablet 服务器，并在 tablet 服务器失效后采取措施；存储 BigTable 模式信息。如果 Chubby 长时间内不可用，BigTable 就会失效。

3 Implementation（实现）

BigTable 的实现包括 3 个主要组件：链接到客户端的库，一个 master 服务器，多个 tablet 服务器。为了适应工作负载的变化，可以向集群中动态添加（或删除）Tablet 服务器。

master 负责向 tablet 服务器分配 tablets，检测新加入的或过期失效的 tablet 服务器，对 tablet 服务器进行负载均衡，对 GFS 中的文件进行垃圾回收。另外，master 还负责处理模式变化，比如 table 和列族的创建/删除。

每个 tablet 服务器都管理着一组 tablets（每个 tablet 服务器通常管理 10 到 1000 个 tablets）。tablet 服务器负责处理针对已加载的 table 的读写请求，以及在 tablet 变得过大时，对其进行分裂。

和其他许多 single-master 类型的分布式存储系统一样，客户端数据不会经过 master：客户端直接与 tablet 服务器通信来完成读写请求。因为 BigTable 客户端在获取 tablet 位置信息时不依赖于 master，所以 BigTable 客户端甚至无需与 master 通信。因此，在实际应用中 master 的负载是很轻的。

BigTable 集群存储了许多 tables，每个 table 由一组 tablets 组成，而每个 tablet 又包含了某个范围内的行的所有相关数据。初始时，每个 table 只拥有一个 tablet。随着 table 的增长，它自动分裂成多个 tablets，每个 tablet 的缺省大小为 1GB。

虽然我们的模型支持任意大小的数据，但是当前的 BigTable 实现并不支持极其庞大的数据。由于 tablet 无法从一行的中间进行分裂，所以我们建议用户在一行内最多存储的数据量不要超过数百 GB。

在本节接下来的内容中，我们会描述一些 BigTable 的实现细节。

3.1 Tablet Location（Tablet 的位置）

我们使用一种类似于 B+ 树的三层架构来存储 tablet 的位置信息（图 1）。第一层是一个存储在 Chubby 内的文件，保存了 root tablet 的位置信息。root tablet 保存了 METADATA 表里的所有 tablets 的位置信息。每个 METADATA 子表保存了一组用户子表 (user tablets) 的位置信息。root tablet 需要特殊对待——它永不分裂——以此保证 tablet 的位置层次结构不会超过 3 层。

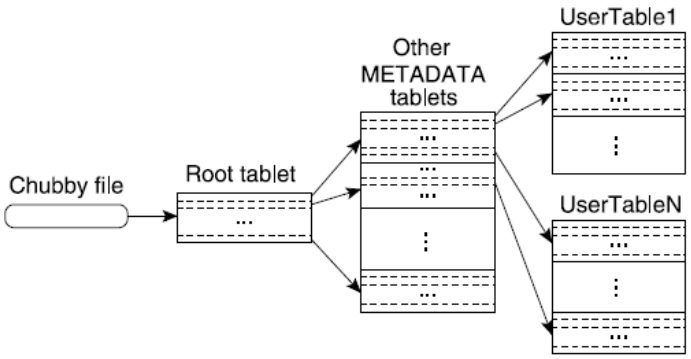

图 1: tablet location hierarchy

METADATA 表将 tablet 的位置信息存储在行关键字下，而这个行关键字是由 tablet 所属 table 的标识符和 tablet 的最后一行编码而成。METADATA 的每一行存储了内存中大约 1 KB 的数据。在 METADATA 子表 128 MB 的保守存储限制下，采用这种三层位置存储模式足以标识 2^{34} 个 tablets 的位置。

客户端软件库通过遍历位置层次结构来定位 tablets，然后缓存找到的 tablets 的位置信息。如果客户端不知道某个 tablet 的位置，或者发现它缓存的位置信息不正确，那么客户端就会递归地在树状层次结构中查询 tablet 的位置。如果客户端的缓存为空，定位算法就需要 3 次网络往返通信才能确定 tablet 的位置，包括对 Chubby 的一次读操作。如果客户端缓存过期了 (stale)，定位算法就需要 6 次往返通信（3 次往返通信发现缓存过期，3 次往返通信更新缓存），因为过期的缓存项只有在 cache miss 的时候才会被发现（我们期望这种情况不要太频繁，因为 METADATA 子表不应该被频繁地移动）。虽然 tablet 位置信息存储在内存中，因此对 tablet 的查询操作无需访问 GFS，但我们仍可以通过在客户端软件库预取 tablet 位置来进一步减小开销：当客户端读取 METADATA 表的时候一次读出多条 tablet 的元数据。

我们也将次级信息 (secondary information) 存储在 METADATA 表，包括与每个 tablet 关联的所有事件日志（例如：一个 tablet 服务器何时启动为其提供服务）。这些信息对于调试和性能分析都非常有用。

3.2 Tablet Assignment（Tablet 分配）

在任意时刻，每个 tablet 最多被分配给一个 tablet 服务器。master 跟踪记录当前活跃的 tablet 服务器、tablets 的分配情况，以及哪些 tablets 未被分配给 tablet 服务器。当一个 tablet 未被分配时，如果存在一个空间充足且可用的 tablet 服务器，master 就向这个 tablet 服务器发送 tablet 加载请求 (tablet load requests)，从而将 tablet 分配给这个服务器。只有在下一个 master 故障转移之前 tablet 服务器未接受到 tablet 加载请求的情况下，这种分配才会失败：tablet 服务器只接收来自当前 master 的 tablet 加载请求。因此，一旦 master 发送了 tablet 加载请求，它就有理由认为：直到 tablet 服务器失效前，tablet 一直处于已被分配的状态；否则，tablet 服务器通知 master，它尚未加载 tablet。

BigTable 使用 Chubby 跟踪记录 tablet 服务器状态。当 tablet 服务器启动时，它会在一个特定的 Chubby 文件夹下新建一个唯一命名的文件，然后在这个文件上创建一个互斥锁并持有它。master 通过实时监控该文件夹（服务器文件夹）的状态来发现新加入的 tablet 服务器。如果 tablet 服务器丢失了它的互斥锁，它就会停止对 tablet 提供服务：例如，网络分区可能会造成服务器丢失其 Chubby 会话。（Chubby 提供了一种高效机制，允许 tablet 服务器在不引起网络拥塞的前提下检查自身是否还持有互斥锁。）只要文件还存在，tablet 服务器就会尝试重新获得文件的互斥锁。如果文件不存在了，tablet 服务器将无法再提供服务，于是它就会自行终止。无论 tablet 服务器何时终止（比如，因为集群管理系统从集群中移除该 tablet 服务器主机），它都会试图释放它所持有的锁，以便 master 能够快速地重新分配 tablets。

master 负责监控 tablet 何时不再为 tablets 提供服务，并尽快将那些 tablets 重新分配给新的 tablet 服务器。为了监控 tablet 服务器何时不再为 tablets 提供服务，master 轮询每一个 tablet 服务器所持有的锁的状态。如果 tablet 服务器报告 master 它丢失了它的锁，或者 master 在经过数次尝试后仍无法连接到 tablet 服务器，master 就会尝试获取该 tablet 服务器文件上

的互斥锁。如果 master 能够获得该锁，就说明 Chubby 运行正常，而 tablet 服务器或是宕机，或是无法连接到 Chubby，那么 master 就要通过删除 tablet 服务器在 Chubby 上的文件以确保它不再给 tablet 提供服务。一旦服务器文件被删除，master 就将之前分配给该 tablet 服务器的 tablets 放进未分配的 tablets 的集合中。为了确保 BigTable 集群不受 master 和 Chubby 之间网络通信问题的影响，master 在 Chubby 会话到期时自行终止。master 的故障不会改变 tablet 在 tablet 服务器上的分配状态。

当 master 在集群管理系统中启动后，它必须先了解当前 tablet 的分配状态，然后才能改变分配状态。master 启动时的执行步骤如下：

(1) master 在 Chubby 中获取一个唯一的 *master* 锁，用来阻止并发的 master 实例。

(2) master 扫描 Chubby 上的服务器文件夹，探知存活的服务器。

(3) master 和所有 tablet 服务器通信，探知哪些 tablets 已经被分配给 tablet 服务器，并更新这些 tablet 服务器上记录的 master 状态信息（如果 tablet 服务器后续又接收到从先前的 master 发送来的 tablet 加载请求，它将拒绝这些请求）。

(4) master 扫描 METADATA 表，学习 tablets 集合。如果扫描过程中遇到未分配的 tablet，master 就将其添加到未分配的 tablets 集合中以等待合适的分配时机。

一个复杂之处在于，只有等到所有 METADATA 子表被分配出去后，对 METADATA 表的扫描才得以进行。因此，在开始扫描之前 (Step (4))，如果在 Step (3) 中发现 root tablet 处于未分配状态，master 就把 root tablet 添加到未分配的 tablets 集合中，保证 root tablet 一定会被分配给 tablet 服务器。因为 root tablet 包含所有 METADATA 子表的名字，所以 master 完成对 root tablet 的扫描后便可知道它们的名字。

保存现有的 tablets 的集合只有在以下情形中会发生改变：创建或删除 table，把两个现有的 tablets 合并成一个更大的 tablet，或者把一个现有的 tablet 分裂成两个更小的 tablets。master 可以跟踪记录这些变化，因为除了最后一种情形，其余情形都是由 master 引发的。tablet 分裂操作需要特殊处理，因为 tablets 的初始化是由 tablet 服务器完成的。tablet 服务器在完成一次分裂操作后，通过在 METADATA 表里记录新的 tablet 的信息来提交本次操作。提交分裂操作之后，tablet 服务器会通知 master。如果通知信息丢失（tablet 服务器或 master 宕机所致），master 在要求 tablet 服务器加载刚分裂出来的新 tablet 之时，就会检测到这个新的 tablet。由于 tablet 服务器在 METADATA 表中找到的 tablet entry 只是 master 要求它加载的 tablet 的一部分，它会将分裂操作通知给 master（译者的理解：tablet 在接受到 master 的 tablet 加载请求到执行请求的这段时间内，要求加载的 tablet 被分裂了，那么 tablet 服务器准备执行加载请求时就会发现，它只能找到该 tablet 的一部分，所以它就会向 master 发送 tablet 已被分裂的通知）。

3.3 Tablet Serving（Tablet 服务）

如图 2所示，tablet 的持久化状态信息存储在 GFS 中。更新被提交至存储恢复记录 (redo records) 的提交日志 (commit log) 中。最近几次的已提交的更新存储在内存中名为 *memtable*

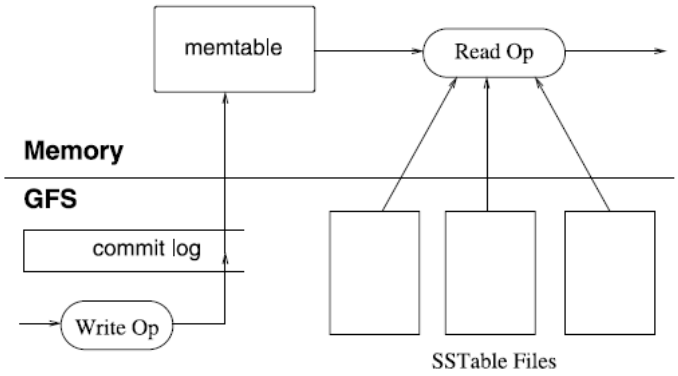

图 2: tablet representation

的排序缓存中。memtable 以行为单位维护这些更新，每一行使用 copy-on-write 机制保持行级 (row-level) 一致性。旧的更新存储在一系列不可修改的 SSTable 中。

tablet 服务器通过从 MATADATA 表读取 tablet 元信息来恢复 tablet。这些元信息包括组成 tablet 的 SSTable 列表，以及一组恢复点 (redo points)，这些 redo points 指向任何可能包含 tablet 数据的提交日志。服务器把 SSTables 的索引读入内存，应用 redo points 之后以提交的更新，以此重建 memtable。

当写操作到达 tablet 服务器时，服务器检查其格式是否正确，以及写者是否有执行该操作的权限——有一个 Chubby 文件保存了拥有写权限的操作者列表，tablet 服务器通过读取该文件即可验证某个写者是否拥有写权限。对于有效的写操作，服务器将其记录到提交日志中。可以采用批量提交来提高包含大量小的修改操作的客户端的吞吐量。写操作被提交后，写的内容就会被插入到 memtable。

当读操作到底 tablet 服务器时，类似与写操作，服务器检查其格式和权限。一个有效的读操作在一系列 SSTables 和 memtable 的合并视图 (merged view) 中执行。由于 SSTables 和 memtable 都是基于字典序的数据存储结构，所以可以高效地构建合并视图。

当 tablets 正在被分裂或合并时，新到来的读写操作都是能够正常继续的。当 tablets 正在被压缩的时候，读写操作也是可能会出现的；压缩的具体内容在下一节讲述。

3.4 Compactions（压缩）

随着写操作的进行，memtable 的大小不断增加。当 memtable 的大小达到阈值时，就将其冻结并创建一个新的 memtable，然后被冻结的 memtable 被转换为 SSTable 并写入 GFS。该过程称为 *minor compation*，其目标有二：缩减 tablet 服务器的内存占用；如果服务器宕机，可以减少服务器恢复期间需从提交日志中读取的数据量。

每次 minor compaction 都会创建一个新的 SSTable。如果 minor compation 一直得不到检查，读操作可能需要合并来自多个 SSTables 的更新。因此，我们通过在后台周期性地执行 *merging compaction* 来限制此类 SSTables 的数量。merging compaction 读取若干 SSTables 和 memtable，将其重新写入一个新的 SSTable。merging compaction 结束后，作为输入的 SSTables 和 memtable 立即被废弃。

将所有 SSTables 重新写入一个 SSTable 的 merging compaction 称为 *major compaction*。由 non-major compactions 生成的 SSTables 可能包含特殊的删除项，这些删除项隐藏了仍存在于旧 SSTables 中的已删除的数据。换句话说，major compaction 生成的 SSTables 中不包含删除信息或已删除的数据（译者记：结合 LevelDB，删除都是标记删除，真正的删除发生在 major compaction 合并 SSTables 之时）。BigTable 轮询所有 tablets，定期对其执行 major compaction。这些 major compaction 使得 BigTable 得以回收那些被标记删除的数据占用的资源，保证被标记删除的数据及时从系统中消失——这对于存储敏感数据的服务而言很重要。

BigTable 优异的读性能得益于 GFS 的局部性优化。写文件的时候，GFS 尝试在同一个写者机器上放置一份数据副本。读文件的时候，由距读者最近的一个文件副本提供服务。因此，在 tablet 服务器和 GFS 服务器共享主机的通常情况下，tablet 服务器会把数据压缩到 SSTables，并在本地磁盘上为其创建一个副本，以此来保证后续针对 SSTables 的读请求能够被快速处理。

3.5 Schema Management（模式管理）

BigTable 模式信息存储在 Chubby 中。Chubby 为 BigTable 模式提供了有效的通信基础组件，因为它提供了针对整个文件的原子写操作，以及小文件的一致性缓存。例如，假设客户端想删除一个 table 的列族中的某些列。master 执行访问控制检查，并在客户端的删除操作结束后检查模式的格式正确性，然后通过重写 Chubby 中相应的模式文件来建立新模式。不管 tablet 服务器何时需要确定当前有哪些列族，它只需从 Chubby 读取相关的模式文件即可，这些文件存在于服务器的 Chubby 客户端缓存中，并且几乎总是可用的。因为 Chubby 缓存的一致性，文件的所有更改都对 tablet 服务器可见。

4 REFINEMENTS（优化）

上一节中描述的 BigTable 实现需要做一些优化才能达到用户要求的高性能、高可用和高可靠性。为了强调这些优化内容，本节将从细节上更加深入地描述 BigTable 的部分实现。

4.1 Locality Groups（局部性群组）

每一个列族都被分配给一个客户端自定义的局部性群组——一个能让客户端控制其数据存储布局的抽象描述。在 compaction 期间，对于 tablet 中的每个局部性群组都会单独生成一个 SSTable。将通常不会一起访问的列族隔离到单独的局部性群组中可以提高读操作的效率。

另外，可以为每个局部性群组制定一些有用的配置参数（tuning parameters）。例如：可以将一个局部性设定为存储在内存中。对于内存中的局部性群组而言，SSTables 以延迟加载的方式加载到 tablet 服务器内存中。由于 SSTables 是不可更改的，所以不存在一致性问题。一旦加载完毕，读取局部性群组中的列族时就不必访问磁盘了。这一特性对于小块数据的频繁访问是很有用的；we use it internally for the tablet-location column family in the METADATA tablet.

4.2 Compression（压缩）

客户端可以控制是否需要对一个局部性群组的 SSTables 进行压缩，并指定压缩格式。用户指定的压缩格式会被应用到每一个 SSTable 的 block（通过局部性群组的配置参数可以控制 block 的大小）。单独压缩每一个 block 会损失一些磁盘空间（译者注：分块压缩的压缩比率要比整体压缩的低），但是我们无需解压缩整个文件便可读取 SSTable 的一小部分数据。许多 BigTable 客户端使用的是两趟自定义压缩模式。第一趟使用 Bentley and Mcllroy's 模式，对一个大窗口中的长公共字符串 (long common strings) 进行压缩。第二趟使用一种快速压缩算法，该算法在一个包含 16KB 数据的小窗口中搜索重复数据。两趟压缩都是极快的——在现代机器上，编码速率可达 100-200MB/s，解码数据可达 400-1000MB/s。

尽管我们在选择压缩算法时强调速度而不是空间，但是上述的两趟压缩模式在这两点上都异常出色。比如在 Webtable 中，我们使用该压缩模式来存储网页内容。在一次实验中，我们在一个经过压缩的局部性群组中存储了大量的文档。出于实验目的，我们在存储文档时存储了对我们可用的一个版本，而非全部版本。该模式实现了 10:1 的空间压缩比，明显优于传统 Gzip 针对 HTML 页面的 3:1 或 4:1 的压缩比，这得益于 Webtable 的行数据布局方式：来自单个主机的页面彼此紧邻存储。这使得 Bentley-Mcllroy 算法能够识别出来自同一个主机的网页中的大量重复内容。许多应用程序，不只是 BigTable，在选择行关键字的时候都尽可能让相似的数据聚集在一起，因此能够实现非常理想的压缩比率。如果我们在 BigTable 中存储相同数据的多个版本，则可获得更高的压缩比率（译者注：利用 BigTable 的 timestamp，可以存储相同数据的多个版本）。

4.3 Caching for Read Performance（通过缓存技术提高读操作的性能）

tablet 服务器使用两级缓存来提高读操作的性能。Scan Cache 是一级缓存，缓存的是 tablet 服务器通过 SSTable 接口返回的 key-value 对。Block Cache 是二级缓存，缓存的是从 GFS 读入的 SSTable 的 blocks。Scan Cache 对于经常反复读取相同数据的应用程序来说非常有用。Block Cache 对于经常读取和最近读取的数据临近的数据的应用程序来说非常有用（比如，针对一个 hot row 中的相同局部性群组的不同列的顺序读 (sequential reads) 或随机读 (random read)）。

4.4 Bloom Filters

一次读操作必须读取构成一个 tablet 状态的所有 SSTables。如果这些 SSTables 尚未装入内存，我们可能需要进行多次磁盘访问。通过让客户端为特定的局部性群组中的 SSTables 指定 Bloom Filters，我们就能够减少磁盘访问次数。Bloom Filter 允许我们询问一个 SSTable 是否包含指定的行/列数据对。对于特定的应用程序，只需在 tablet 服务器内存中开辟一小块空间来存储 Bloom Filters，就可以大幅减少读操作所需要随机磁盘访问。Bloom Filters 的使用亦可避免大多数针对不存在的行/列数据的查询而导致的磁盘访问。

4.5 Commit-Log Implementation（提交日志的实现）

如果我们为每个 tablet 的提交日志都保存为一个日志文件，就需要在 GFS 中对大量文件进行并发写。基于每个 GFS 服务器的底层文件系统实现，这些写操作可能需要大量的磁盘随机

访问才能写入不同的磁盘日志文件。另外，由于批量提交 (group commit) 的操作数目一般比较小，为每个 tablet 设置单独的日志文件也会降低批量操作本应具有的效率。为了解决这些问题，我们把每一个 tablet 服务器的修改日志都已追加方式写入同一个提交日志中，这样一来，同一个日志文件中就混杂了不同 tablet 服务器的修改操作的日志记录。

对于常规操作而言，仅适用一个日志文件无疑带来了极高的性能突破，但这使得恢复操作复杂化。当 tablet 服务器宕机时，它所服务的 tablets 会被重新分配到大量其他的 tablet 服务器；这些服务器一般只会加载最初宕机的服务器上的少数 tablets。为了恢复某个 tablet 的状态，新的 tablet 服务器需要重新从最初宕机的服务器写入的提交日志中为该 tablet 应用修改记录。然而，这些 tablets 的修改日志是混杂在同一个磁盘文件中的。一种方案是，每个新的 tablet 服务器读入完整的提交日志，但是仅应用那些需要用于恢复它所加载的 tablets 的日志项。然而在该模式下，如果 100 台机器中每台只从宕机服务器那里分配到一个 tablet，那么就需要读取同一个日志文件 100 次（每个服务器读一次）。

我们通过提交日志项的 key<table, row name, log sequence number> 对日志项进行排序，以此避免对同一日志文件的重复读。在排序输出结果中，对于特定 tablet 的所有修改记录都是连续的，因此可以在一次磁盘随机读之后进行顺序读，从而提高读性能。为了实现排序的并行化，我们把日志文件分成多个 64MB 的片段，在不同的 tablet 服务器上对每个片段并行排序。排序进程由 master 进行调度；当一个 tablet 服务器需要从提交日志文件中恢复对 tablets 的修改操作时，排序进程即刻启动。

由于各种原因，在 GFS 中写提交日志有时会产生性能瓶颈（比如，GFS 服务器主机发生了写操作崩溃，负载过高，与一组特定的 GFS 服务器之间的网络通信受阻）。为了让修改操作避开 GFS 服务器的延迟峰值，每个 tablet 服务器配有两个写线程，每个线程都只写自己的日志文件；在任意时刻只有一个线程出入活跃状态。如果一个线程对日志文件的写性能明显下降，就切换到另一个线程，在提交日志队列中的修改操作由当前活跃的线程写入日志。日志项包含的序列号允许恢复线程忽略重复日志项——这是由线程切换产生的。

4.6 Speeding Up Tablet Recovery（加速 Tablet 恢复）

卸载 tablet 之前，tablet 服务器首先对 tablet 做一个 minor compaction，减少了服务器的提交日志中未压缩的日志项，从而减小了后续的恢复时间。完成 minor compaction 之后，tablet 服务器就停止为 tablet 提供服务。在 tablet 服务器真正卸载 tablet 之前，它会再做一次 minor compation（通常很快），清除服务器日志中尚未压缩的残留数据，这些数据是第一次 minor compaction 执行期间到达服务器的。第二次 minor compaction 完成后，tablet 就能够被加载到另一个 tablet 服务器而无需恢复任何日志项。

4.7 Exploiting Immutability

除了 SSTables 缓存，BigTable 系统的其他各部分都已得到简化，这是基于这样一个事实——SSTables 一经生成就不再修改。例如，读取 SSTables 时无需对文件系统的访问进行任何同步（译者注：对同一文件的并发读不需要同步）。所以，我们可以高效地实现基于行 (row) 的并发控制。唯一可被修改的数据结构——可读可写——是 memtable。为了减少 memtable 的读竞争，我们为 memtable 的每一行设置 copy-on-write，使得读写得以并发执行。

由于 SSTable 不可修改，“永久移除标记删除的数据”这一问题就转变为对废弃的 SSTables 进行垃圾回收的问题了。每个 tablet 的 SSTables 信息都注册在 METADATA 表内。master 在移除废弃的 SSTables 时，采用“标记-清除 (mark-and-sweep)”的垃圾回收方式，METADATA 表保存了这些 SSTables 的根节点信息。

最后，SSTables 不可修改的特性让我们能够快速分裂 tablets。与其为每一个新分裂出来的 tablets(child tablets) 生成一组新的 SSTables，我们不如让这些新 tablets 与原来的 tablet(parent tablet) 共享 SSTables。