**Google三驾马车读后感**

**19301158 王子坤**

**GFS**

**GFS的目标：**

1. 由于使用的机器是廉价的商业化机器，那么机器崩溃被认为是一种常态。
2. 系统存储以大文件为主，但也支持小文件。文件大小通常在100MB左右并且需要高效的操作几个GB的文件。
3. 系统需要支持大规模的连续读取和小规模的随机读取，以及大规模的追加写。
4. 高性能稳定的网络带宽比延迟更重要。
5. 以及最重要的，能在分布式的系统上运行。

**具体实现**

**Single Master**

Master中保存三种信息（MATEDATA）

1. 文件和chunk的namespace，即文件树形式的命名方式。
2. 文件到chunk的映射，即每个文件需要哪几个Chunk来记录。
3. 每一个chunk的具体位置。

这些信息都平时都存在内存中，以此提高响应速度。这似乎只能存储很少的信息，但是，Master只需要64byte的空间就能记录64MB的Chunk的相关信息，也就是说，128MB的内存能存储1PB的数据的相关信息。不过，为了保证Master能在崩溃后恢复，在执行改变前两种信息的操作前需要使用WAL的形式定期地保存到磁盘上。这样，在Master恢复的时候，就能通过磁盘上的WAL来重建前两种信息。而Chunk的位置则可以通过Heartbeats的形式查询并更新，这样不仅加速了Master的恢复，也能使GFS的配置更加灵活。

**写数据（Write）**

1. 客户端向 Master 询问目前哪个Chunk Server持有该Chunk的Lease
2. Master 向客户端返回Primary Replica和其他Replica的位置
3. 客户端将数据推送到所有的Replica上。Chunk Server会把这些数据保存在缓冲区中，等待所有 Replica 都接收到数据。
4. 客户端发送写请求给 Primary，Primary 为来自各个客户端的修改操作选定执行序列号，并按顺序地应用于其本地存储的数据。
5. Primary 将写请求转发给其他 Secondary Replica，Replica 们按照相同的顺序应用这些修改
6. Secondary Replica 响应 Primary，示意自己已经完成操作。
7. Primary 响应客户端，并返回该过程中发生的错误

这里要说明的是，每个Chunk被复制到多个Chunk Server中，以此避免单个节点崩溃可能造成的数据损失。而这些Chunk Server中负责相应写操作的Chunk Server被称为Primary Replica，其余的被称为Secondary Replica，接受来自Primary Replica的请求。而为了保证写入数据的一致性，只能有一个Primary Replica，这里的唯一性就由Lease来实现。当需要写入数据时，Master会将特定的Lease分配给某个Replica，拿到Lease的这个Replica就称为了Primary Replica。

可以注意到为了保证数据的一致性，一次写入只能有一个Primary Replica，同样地，在正式写入数据之前要求所有Secondary Replica缓存数据也是为了一致性。这里的思想类似于WAL，将要执行的操作先保存下来，这样万一崩溃了也能从磁盘读入数据，继续执行未完成的操作。等待所有Secondary Replica完成操作后再相应客户端也是为了保证数据的一致性。

和读操作类似，在写操作中，为了降低Master的压力，所有的数据由客户端发向Chunk Server。

**追加（Append）**

为了提升性能，GFS提供并推荐使用追加操作修改文件。它与写操作不同之处仅仅在于它向文件尾端添加数据而不是覆盖，它的流程也和写操作大同小异

1. 客户端将数据推送到每个 Replica，然后将请求发往 Primary
2. Primary 首先判断将数据追加到该块后是否会令块的大小超过上限：如果是，那么 Primary 会为该块写入填充至其大小达到上限，并通知其他 Replica 执行相同的操作，再响应客户端，通知其应在下一个块上重试该操作
3. 如果数据能够被放入到当前块中，那么 Primary 会把数据追加到自己的 Replica 中，拿到追加成功返回的偏移值，然后通知其他 Replica 将数据写入到该偏移位置中
4. 最后 Primary 再响应客户端

特别值得一提的是，GFS保证追加操作至少被执行一次（at least once），这意味着追加操作可能被执行多次。当追加操作失败时，为了保证偏移量，GFS会在对应的位置填充重复的数据，然后重试追加。也就是说，GFS不保证在每个副本中的数据完全一致，而仅仅保证数据被写入了。

**数据一致性**

在GFS中，由于分布式系统的原因，不同节点间处理请求和存储数据速度不一致导致了客户算读取数据时可能出现各种不同的情况。

文件的数据修改则相对复杂。在讲述接下来的内容前，首先明确，在文件的某一部分被修改后，它可能进入以下三种状态的其中之一：

* 客户端读取不同的 Replica 时可能会读取到不同的内容，那这部分文件是不一致的（Inconsistent）。
* 所有客户端无论读取哪个 Replica 都会读取到相同的内容，那这部分文件就是一致的（Consistent）。
* 所有客户端都能看到上一次修改的所有完整内容，且这部分文件是一致的，那么我们说这部分文件是确定的（Defined）。

也就是说，在一致性和强度上，Defined>Consistent>Inconsistent。

在修改后，一个文件的当前状态将取决于此次修改的类型以及修改是否成功。具体来说：

* 如果一次写入操作成功且没有与其他并发的写入操作发生重叠，那这部分的文件是确定的（同时也是一致的）。
* 如果有若干个写入操作并发地执行成功，那么这部分文件会是一致的但会是不确定的：在这种情况下，客户端所能看到的数据通常不能直接体现出其中的任何一次修改。也就是说，操作成功执行了，但是有的操作的数据改变被覆盖了，客户端看不到被覆盖的数据改变。
* 失败的写入操作会让文件进入不一致的状态。

至于为什么追加（Append）是Defined但是有可能是不一致是因为：在追加写操作失败时，为了保证数据的偏移，可能为填充重复的数据，此时导致了不一致。但是失败的操作会被再次执行，此时又保证了数据的一致性。而由于使用的是追加，所以任何数据的改动都可以观察到，所以是Defined。

**快照（Snapshot）**

这里的快照的目的不同于Raft中的压缩，它仅仅驶出为了生成一个新的Replica，可以看做是一个简单的复制操作。

1. 在 Master 接收到快照请求后，它首先会撤回相关Chunk Server的 Lease，保证在创建快照的过程中，客户端对不会对相关的Chunk Server进行写操作或者追加操作。
2. 当Lease收回后，Master会先将相关的改动写入日志，然后对自己管理的命名空间进行复制操作，复制产生的新记录指向原本的 Chunk。
3. 当有客户端尝试对新的Chunk Server进行写入时，Master 会注意到这个 Chunk 的引用计数大于1（可能是一个标记）。此时，Master 会为要读取的Chunk生成一个Handle，然后通知所有持有这些 Chunk 的 Chunk Server 在本地复制并使用出新的 Chunk，然后再返回给客户端

我想GFS提供快照的原因可能是为了在一个副本损坏时，从Primary Replica或者其他副本复制数据，然后用新的节点代替损坏的节点。

**垃圾回收**

当GFS收到删除文件的请求时，它并不直接删除文件，而是给文件打上删除的时间戳并将其命名为掩藏文件（文件开头加"."）。在周期性扫描过程中，当发现文件的删除时间超过设定期限后，才真正地将文件删除。Google认为其有以下优点

* 对于大规模的分布式系统来说，这样的机制更为**可靠**：在 Chunk 创建时，创建操作可能在某些 Chunk Server 上成功了，在其他 Chunk Server 上失败了，这导致某些 Chunk Server 上可能存在 Master 不知道的 Replica。除此以外，删除 Replica 的请求可能会发送失败，Master 会需要记得尝试重发。相比之下，由 Chunk Server 主动地删除 Replica 能够以一种更为统一的方式解决以上的问题
* 这样的删除机制将存储回收过程与 Master 日常的周期扫描过程合并在了一起，这就使得这些操作可以以批的形式进行处理，以减少资源损耗；除外，这样也得以让 Master 选择在相对空闲的时候进行这些操作
* 用户发送删除请求和数据被实际删除之间的延迟也有效避免了用户误操作的问题

**高可用**

论文中主要探讨了三个方面的高可用，分别是Master，Chunk Server和数据完整性。

当Master崩溃时，有两种情况，一是进程崩溃但是服务器没有，这种情况下，重开一个进程即可。另一种情况是整个机器崩溃了，在GFS还有被称为Shadow Master的机器，复制Master节点的信息。当Master机器崩溃后，Shadow Master会接替Master进行服务，但是仅提供读取操作的服务，不能更改信息。

当Chunk Server崩溃时，Master会安排新的Replica代替它，从其他Replica复制原始数据。而当Chunk Server恢复时，由于数据不同步，不应该提供服务，Master就需要区别新旧Chunk Server。GFS使用版本号来标记这个信息，每分配以此Lease，版本号就会增加并同步给其他Replica，而由于Chunk Server崩溃后不能更新，我们就能从版本号上辨别新旧。

由于写操作和追加操作可能不成功，所以数据可能会损坏。GFS使用检验和检查是否损坏，每次客户端读取数据时，Chunk Server都会检查检验和，一旦发现损坏，就会向Master报告。Master则会将请求发送给其他Replica，并从其他Replica复制数据到该机器。

**Map Reduce**

**基本思想**

MapReduce把所有的计算都拆分成两个基本的计算操作，即Map和Reduce。其中Map函数以一系列键值对作为输入，然后输出一个中间文件（Intermediate）。这个中间态是另一种形式的键值对。然后，Reduce函数将这个中间态作为输入，计算得出结果。其中，Map函数和Reduce函数的逻辑都是由开发人员自行定义的。

**分布式MapReduce流程**

上面提到的模型和思想都是单机的，想要在分布式系统中实现，还需要一些改动。在MapReduce中，他们选择将大任务拆分成小任务分配给多台机器，以此充分利用分布式系统的性能。

具体的流程如下

1. MapReduce客户端会将输入的文件会分为M个片段，每个片段的大小通常在 16~64 MB 之间。然后在多个机器上开始运行MapReduce程序。
2. 系统中会有一个机器被选为Master节点，整个 MapReduce 计算包含M个Map 任务和R个 Reduce 任务。Master节点会为空闲的 Worker节点分配Map任务和 Reduce 任务
3. 执行Map任务的 Worker开始读入自己对应的片段并将读入的数据解析为输入键值对。然后调用由用户定义的 Map任务。最后，Worker会将Map任务输出的结果存在内存中。
4. 在执行Map的同时，Map Worker根据Partition 函数将产生的中间结果分为R个部分，然后定期将内存中的中间文件存入到自己的本地磁盘中。任务完成时，Mapper 便会将中间文件在其本地磁盘上的存放位置报告给 Master。
5. Master会将中间文件存放位置通知给Reduce Work。Reduce Worker接收到这些信息后便会通过RPC读取中间文件。在读取完毕后，Reduce Worker会对读取到的数据进行排序，保证拥有相同键的键值对能够连续分布。
6. 最后，Reduce Worker会为每个键收集与其关联的值的集合，并调用用户定义的Reduce 函数。Reduce 函数的结果会被放入到对应的结果文件。
7. 当所有Map和Reduce都结束后，程序会换新客户端并返回结果。

整个流程非常清晰。首先，将输入文件分割成M个个片段，然后每个Map Worker读取对应的片段并执行Map函数，将结果存入中间文件。Reduce Work则通过Master得知中间文件的位置，然后读取其对应中间文件的内容并运行Reduce函数，最后把结果输出到结果文件中。

这里值得说明的是，无论是输入文件到Map Worker的映射还是中间文件到Reduce Worker的映射都可以通过自定义的哈希函数来确定，论文中默认使用**Hash(key) mod R**来确定。另外，M和R的值都是由用户指定的，应当比实际的机器数量要多一些，以此实现均衡负载。

**Fault-Tolerance**

因为使用了分布式系统，所以不可避免地要考虑容错的问题，在MapReduce中，容错也考虑Master和Work两种情况。

Master节点会定期地将当前运行状态存为快照，当Master节点崩溃，就从最近的快照恢复然后重新执行任务。

Master节点会定期地Ping每个Work节点，一旦发现Work节点不可达，针对其当前执行的是Map还是Reduce任务，会有不同的策略。

如果是Map任务，无论任务已完成或是未完成，都会废除当前节点的任务。之后，Master会将任务重新分配给其他节点，同时由于已经生成的中间文件不可访问，还会通知还未拿到中间文件的Reduce Worker去新的节点拿数据。

如果是Reduce任务，由于结果文件存在GFS中，文件的可用性和一致性由GFS保证，所以Master仅将未完成的任务重新分配。

**优化**

如果集群中有某个 Worker 花了特别长的时间来完成最后的几个 Map 或 Reduce 任务，整个 MapReduce 计算任务的耗时就会因此被拖长，这样的 Worker 也就成了落后者。MapReduce 在整个计算完成到一定程度时就会将剩余的任务即同时将其分配给其他空闲 Worker 来执行，并在其中一个 Worker 完成后将该任务视作已完成。

**总结**

MapReduce是一个相当简单的计算模型，它尝试将所有的计算任务都拆分成基础的Map和Reduce，以此降低实现的复杂度。

**Big Table**

Bigtable实现在Google File System的基础上，它关心数据的内容，根据的数据的内容建立数据模型，对外提供读写数据的接口。

**数据模型**

Bigtable基本的数据结构和关系型数据库类似，都是以行列构成的表，但是，它还另外增加了新的维度——时间。也就是说，在行列确定的情况下，一个单元格（Cell）中有多个以事件为版本的数据。Bigtable用(row:string, column:string, time:int64) → string表示映射关系。下图为论文中给出的一个例子。

如果想要在表中查询指定版本的内容，我们需要指出行，列，及版本。比如（“com.cnn.www”，“anchor：cnnsi.com”，t9）→ "CNN"。我个人猜想，增加时间这个维度是因为“三驾马车”被设计出来的时候主要是为了支持搜索引擎，搜索引擎可能需要保留多个时间段的网页数据，而GFS也使用追加（Append）作为数据的主要修改方式，所以增加时间戳作为版本既充分利用了GFS的特性，也能满足业务的需求。

另外，Bigtable还把多个Column Keys并入到被称为Column Family的集合中，并将Column Family作为访问控制的基础单元。我认为，这种方案其实是一种事务（Transaction）的实现方案。传统的事务以行为基本操作单位，在读写时对行上锁以实现隔离，而Bigtable则是以Column Family为单位，这里的访问控制其实就是锁的思想。

**相关组件**

在介绍系统的整体架构之前，我们要对Bigtable用到的两个重要组件有一些了解。由于Bigtable是分布式的数据库，在节点之间的协调上需要额外的处理，这里，Bigtable使用了Google内部的Chubby。我们可以把Chubby看做是Zookeeper，因为Zookeeper本质也就是Chubby的开源版本。另一方面，为了加快数据的查找和存储效率，Bigtable在存储数据之前都进行了排序，而此处用到的存储文件文论称之为SSTable（Sorted String Table）。

在Bigtable中，由于单个表（Table）存储的数据可能相当地多，那么读写的效率就会十分低下，于是Bigtable将Table分割为固定大小的Tablet，将其作为数据存储和查找的基本单位。每当Table增加了这里要说明的是，tablet是数据存储的基本单元，是用户感知不到的。而Column Family则是访问的基本单元，是编程时指定的，两者一前一后，不是一个概念。

**Tablet 定位**

因为是在分布式系统中，那么每个Tablet所在的机器不同，需要记录相关信息（METADATA）对其进行管理。而存储这些METADATA又需要分布式的系统，所以Bigtable又将这些METADATA的METADATA记录在一个文件中，并将这个文件的位置保存在Chubby中。总结一下，Bigtable有以下三层结构：

图示

描述已自动生成

1. 在Chubby中保存着Root Tablet的位置
2. Root Tablet中保存着METADATA Table中所有 Tablet 的位置
3. METADATA Table中保存着所有存储数据的Tablet的位置

这其中有几点值得注意。由于Root Tablet的特殊性，哪怕它的数据量再大，它也不允许被分割。METADATA tables被读取到内存中以加快速度，其中存储的是以开始和结尾的Row Key作为键，tablet位置作为值的映射。

如果客户端希望读取特定的数据，那么它会以此读取Chubby中的文件，Root Tablet，METADATA Tablet，最后读取存储改数据的Tablet。同时，为了加快读取的速度，它会将这些信息缓存到本地，直到信息失效。

**Tablet分配**

在谈Table分配之前，论文先讨论了怎么处理成员变更的问题。类似于GFS，Bigtable使用Master节点来管理这些相关的事情。

首先，Bigtable使用Chubby来检测Tablet Server的变化。这里的操作和Zookeeper的用法类似，当有新节点加入时，它需要在Chubby中新建一个对应的文件，并获取该文件的锁。由于所有的节点在Chubby中都有对应的文件，那么Master可以通过监听Chubby来获取所有Tablet Server的信息。这里有两种节点失效的情况，一种是仅仅回收了锁但是文件还在，这种情况很可能是节点崩溃了。由于节点不能自己退出，所以在Master节点得到该文件的锁后，它会将文件删除，以此表示节点退出。另一种情况是，文件已经被删除，这种情况说明节点是主动退出系统，那么可以直接重新分配Tablet给其他节点即可。

在正常的情况下，系统中会有大量数据写入，Master需要负责将这些数据分配到合适的Tablet Server。Bigtable并没有明确指出分配所使用的的算法，但是它提出了一个要求。为了保证数据的一致性，同一时间，一个 Tablet只能被分配给一个Tablet Server。Master通过向 Tablet Server 发送载入请求来分配 Tablet。如果该载入请求被Tablet Server接收到前Master仍是有效的，那么就可以认为此次 Tablet 分配操作已成功。

在这里，我们还要考虑Master崩溃的情况，论文中描述了Master恢复的步骤如下：

1. 在 Chubby 上获取 Master 独有的锁，确保不会有另一个 Master 同时启动
2. 从 Chubby 了解在工作的 Tablet Server
3. 从各个 Tablet Server 处获取其所负责的 Tablet 列表，并向其表明自己作为新 Master 的身份，确保 Tablet Server 的后续通信能发往这个新 Master
4. Master 确保 Root Tablet 及 METADATA 表的 Tablet 已完成分配
5. Master 扫描 METADATA 表获取集群中的所有 Tablet，并对未分配的 Tablet 重新进行分配

其中，第四步是为了第五步的正确执行。

**读写Tablet**

每个Tablet由若干个位于 GFS 上的 SSTable、一个位于内存内的MemTable以及一份Tablet Log组成。

为了保证系统可恢复，Google首先使用Table Log（即WAL）将客户端发出的写操作请求记录在磁盘中，那么，一旦系统崩溃，仍然可以从磁盘读取数据，继续执行命令。然后，相关的数据被放入位于内存中的Memtable中，因为内存的速度相当快，那么执行排序等操作就要快得多。当Memtable的大小达到设定的值后，它就会以SSTable的形式被存储到GFS中，这被称为Minor Compaction。

客户端的读操作请求则要综合考虑Memtable和SSTable中的数据，如果Memtable中已经有需要读的数据，就无需读取SSTable。由于Memtable和SSTable都是有序的，所以读取的速度都相当快。

在这里，虽然论文没有明确指出，我认为Memtable和SSTable的大小很可能是64MB。因为GFS将单个Chunk设置为64MB，那么为了最大化地利用磁盘空间，Memtable和SSTable的大小设置为这个值是相当合理的。

由于SSTable中的数据有可能被标记为删除，那么我们需要定期对其进行处理，Bigtable将其称为Major Compaction。在这个过程中，Bigtable会将过期或者被删除的数据删除，并合并多个SSTable。这里似乎和GFS的Garbage Collection有点类似，但是我认为这可能是两个层面的活动。Bigtable清理的是单个Chunk中的数据，而GFS清理的是磁盘中的单个Chunk。

**优化**

论文中提到，仅靠上述这些方法还不能达到要求的速度，因此，Bigtable还做了一些优化。

第一，为了提高读取的速度，Bigtable使用布隆过滤器判断数据是否在某个SSTable中。

第二，Tablet Server使用Scan Cache缓存SSTable返回的数据，在重复读时提高效率。使用Block Cache缓存从GFS读取的SSTable，这样在读取附近的数据时就无需从磁盘读取。

第三，Bigtable把所有的写入操作都写入到同一个Bigtable Log文件中，而不是每个Server分配一个。同时，因为这个文件相当大，恢复起来很费事。Bigtable会对其进行排序并进行切分，每个Tablet Sever只需读取自己的那部分就可以了。

第四，Bigtable允许针对特定的Column Family生成SSTable，同时进行压缩，以提高读取的效率。

**总结**

Bigtable重要的贡献是证明了在分布式的系统中，针对超大规模的数据量，使用排序大表的来设计数据库是可行的。这直接带动了LSM Tree的流行，在后来的HBase，LevelDB中都使用了这种方式处理数据。另外，Bigtable系统中Chubby的使用，还告诉工业界分布式协调组件的重要性，这也引导了Zookeeper的设计实现，而其仍然是今天的分布式系统中重要的组件。