Kudu：一个融合低延迟写入和高性能分析的存储系统

作者 siddontang (/u/1yJ3ge) ＋关注
2017.05.06 09:59 字数 6155 阅读 2644 评论 4 喜欢 14
(/u/1yJ3ge)

Kudu 是一个基于 Raft 的分布式存储系统，它致力于融合低延迟写入和高性能分析这两种场景，并且能很好的嵌入到 Hadoop 生态系统里面，跟其他系统譬如 Cloudera Impala，Apache Spark 等对接。

Kudu 很类似 TiDB。最开始，TiDB 是为了 OLTP 系统设计的，但后来发现我们 OLAP 的功能也越来越强大，所以就有了融合 OLTP 和 OLAP 的想法，当然这条路并不是那么容易，我们还有很多工作要做。因为 Kudu 的理念跟我们类似，所以我也很有兴趣去研究一下它，这里主要是依据 Kudu 在 2015 发布的 paper，因为 Kudu 是开源的，并且在不断的更新，所以现在代码里面一些实现可能还跟 paper 不一样了，但这里仅仅先说一下我对 paper 的理解，实际的代码我后续研究了在详细说明。

为什么需要 Kudu？

结构化数据存储系统在 Hadoop 生态系统里面，通常分为两类：

- 静态数据，数据通常都是使用二进制格式存放到 HDFS 上面，譬如 Apache Avro，Apache Parquet。但无论是 HDFS 还是相关的系统，都是为高吞吐连续访问数据这些场景设计的，都没有很好的支持单独 record 的更新，或者是提供好的随机访问的能力。
- 动态数据，数据通常都是使用半结构化的方式存储，譬如 Apache HBase，Apache Cassandra。这些系统都能低延迟的读写单独的 record，但是对于一些像 SQL 分析这样需要连续大量读取数据的场景，显得有点捉紧见拙。

上面的两种系统，各有自己的侧重点，一类是低延迟的随机访问特定数据，而另一类就是高吞吐的分析大量数据。之前，我们并没有这样的系统可以融合上面两种情况，所以通常的做法就是使用 pipeline，譬如我们非常熟悉的 Kafka，通常我们会将数据快速写到 HBase 等系统里面，然后通过 pipeline，在导出给其它分析系统。虽然我们在一定层面上面，我们其实通过 pipeline 来对整个系统进行了解耦，但总归要维护多套系统。而且数据更新之后，并不能直接实时的进行分析处理，有延迟的开销。所以在某些层面上面，并不是一个很好的解决方案。

Kudu 致力于解决上面的问题，它提供了简单的来处理数据的插入，更新和删除，同时提供了 table scan 来处理数据分析。通常如果一个系统要融合两个特性，很有可能就会陷入两边都做，两边都没做好的窘境，但 Kudu 很好的在融合上面取得了平衡，那么它是如何做到的呢？

Keyword

Tables 和 schemas

Kudu 提供了 table 的概念。用户可以建立多个 table，每个 table 都有一个预先定义好的 schema。Schema 里面定义了这个 table 多个 column，每个 column 都有名字，类型，是否允许 null 等。一些 columns 组成了 primary key。

可以看到，Kudu 的数据模型非常类似关系数据库，在使用之前，用户必须首先建立一个 table，访问不存在的 table 或者 column 都会报错。用户可以使用 DDL 语句添加或者删除 column，但不能删除包含 primary key 的 column。

但在 Paper 里面说到 Kudu 不支持二级索引以及除了 primary key 之外的唯一索引，这个后续可以通过更新的代码来确定下。

其实我这里非常关注的是 Kudu 的 Online DDL 是如何做的，只是 Paper 里面貌似没有提及，后面只能看代码了。

API

Kudu 提供了 Insert，Update 和 Delete 的 write API。不支持多行事务 API，这个不知道最新的能支持了没有，因为仅仅能对单行数据操作，还远远不够。

Kudu 提供了 Scan read API 让用户去读取数据。用户可以指定一些特定的条件来过滤结果，譬如用一个常量跟一个 column 里面的值比较，或者一段 primary key 的范围等条件。

提供 API 的好处在于实现简单，但对于用户来说，其实更好的使用方式仍然是 SQL，一些复杂的查询最好能通过 SQL 搞定，而不是让用户自己去 scan 数据，然后自己组装。

一致性模型

Kudu 提供两种一致性模型：snapshot consistency 和 external consistency。

默认 Kudu 提供 Snapshot consistency， 它具有更好的读性能，但可能会有 write skew 问题。而 External consistency 则能够完全保证整个系统的 linearizability，也就是当写入一条数据之后，后面的任何读取都一定能读到最新的数据。

为了实现 External consistency，Kudu 提供了几种方法：

- 在 clients 之间显示的传递时间戳。当写入一条数据之后，用户用要求 client 去拿一个时间戳作为 token，然后通过一个 external channel 的方式传递给另一个 client。然后另一个 client 就可以通过这个 token 去读取数据，这样就一定能保证读取到最新的数据了。不过这个方法实在是有点复杂。
- 提供类似 Spanner 的 commit-wait 机制。当写入一条数据之后，client 需要等待一段时间来确定写入成功。Kudu 并没有采用 Spanner TrueTime 的方案，而是使用了 HybridTime 的方案。HybridTime 依赖 NTP，这个可能导致 wait 的时间很长，但 Kudu 认为未来随着 read-time clock 的完善，这应该不是问题了。

Kudu 是我已知的第二个采用 HybridTime 来解决 External consistency 的产品，第一个当然就是 CockroachDB 了。TiDB 跟他们不一样，我们采用的是全局授时的方案，这个会简单很多，但其实也有跟 PD 交互的网络开销。后续TiDB 可能使用类似 Spanner 的 GPS + 原子钟，现阶段相关硬件的制造方式 Google 并没有说明，但其实难度不大。因为已经有很多硬件厂商主动找我们希望一起合作提供，只是比较贵，而现阶段我们大多数客户并没有跨全球事务这种场景。

Kudu 的一致性模型依赖时间戳，这应该是现在所有分布式系统通用的做法。Kudu 并没有给用户保留时间戳的概念，主要是觉得用户很可能会困惑，毕竟不是所有的用户都能很好的理解 MVCC 这些概念。当然，对于 read API，还是允许用户指定特定的一个时间戳，这样就能读取到历史数据。这个 TiDB 也是类似的做法，用户不知道时间戳，只是我们额外提供了一个设置 snapshot 的操作，让用户指定生成某个时间点的快照，读取那个时间点的数据。这个功能已经帮很多公司恢复了因为错误操作写坏的数据了。

架构

上面说了一些 Kudu 的 keyword， 现在来说说 Kudu 的整体架构。Kudu 类似 GFS，提供了一个单独的 Master 服务，用来管理整个集群的元信息，同时有多个 Tablet 服务，用来存储实际的数据。

分区

Kudu 支持对数据按照 Range 以及 Hash 的方式进行分区。 每个大的 table 都可以通过这种方式将数据分不到不同的 Tablet 上面。当用户创建一个表的时候，同时也可以指定特定的 partition schema，partition schema 会将 primary key 映射成对应的 partition key。每个 Tablet 上面会覆盖一段或者多段 partition keys 的range。当 client 需要操作数据的时候，它可以很方便的就知道这个数据在哪一个 Tablet 上面。

一个 partition schema 可以包括 0 或者多个 hash-partitioning 规则和最多一个 range-partitioning 规则。用户可以根据自己实际的场景来设置不同的 partition 规则。

譬如有一行数据是 `(host, metric, time, value)` ，time 是单调递增的，如果我们将 time 按照 hash 的方式分区，虽然能保证数据分散到不同的 Tablets 上面，但如果我们想查询某一段时间区间的数据，就得需要全部扫描所有的 Tablets 了。所以通常对于 time，我们都是采用 range 的分区方式。但 range 的方式会有 hot range 的问题，也就是同一个时间会有大量的数据写到一个 range 上面，而这个 hot range 是没法通过 scale out 来缓解的，所以我们可以将 `(host, metric)` 按照 hash 分区，这样就在 write 和 read 之间提供了一个平衡。

通过多个 partition 规则组合，能很好的应对一些场景，但同时这个这对用户的要求比较高，他们必须更加了解 Kudu，了解自己的整个系统数据会如何的写入以及查询。现在 TiDB 还只是单纯的支持 range 的分区方式，但未来不排除也引入 hash。

Raft

Kudu 使用 Raft 算法来保证分布式环境下面数据一致性，这里就不再详细的说明 Raft 算法了，因为有太多的资料了。

Kudu 的 heartbeat 是 500 毫秒，election timeout 是 1500 毫秒，这个时间其实很频繁，如果 Raft group 到了一定量级，网络开销会比较大。另外，Kudu 稍微做了一些 Raft 的改动：

- 使用了 exponential back-off 算法来处理 leader re-election 问题。
- 当一个新的 leader 跟 follower 进行交互的时候，Raft 会尝试先找到这两个节点的 log 分叉点，然后 leader 再从这个点去发送 log。Kudu 直接是通过 committedIndex 这个点来发送。

对于 membership change，Kudu 采用的是 one-by-one 算法，也就是每次只对一个节点进行变更。这个算法的好处是不像 joint consensus 那样复杂，容易实现，但其实还是会有一些在极端情况下面的 corner case (https://github.com/pingcap/tikv/issues/1468) 问

题。

当添加一个新的节点之后，Kudu 首先要走一个 remote bootstrap 流程。

1. 将新的节点加入到 Raft 的 configuration 里面
2. Leader 发送 StartEmoteBootstrap RPC，新的 follower 开始拉去 snapshot 和之后的 log
3. Follower 接受完所有数据并 apply 成功之后，开始响应 Raft RPC

可以看到，这个流程跟 TiKV 的做法类似，这个其实有一个缺陷的。假设我们有三个节点，加入第四个之后，如果新的节点还没 apply 完 snapshot，这时候挂掉了一个节点，那么整个集群其实是没法工作的。

为了解决这个问题，Kudu 引入了 `PRR_VOTER` 概念。当新的节点加入的时候，它是 `PRE_VOTE` 状态，这个节点不会参与到 Raft Vote 里面，只有当这个节点接受成功 snapshot 之后，才会变成 `VOTER` 。

当删除一个节点的时候，Leader 直接提交一个新的 configuration，删除这个节点，当这个 log 被 committed 之后，这个节点就把删除了。被删除的节点有可能不知道自己已经被删除了，如果它长时间没有收到其他的节点发过来的消息，就会问下 Master 自己还在不在，如果不在了，就自己干掉自己。这个做法跟 TiKV 也是类似的。

Master

Kudu 的 Master 是整个集群最核心的东西，类似于 TiKV 里面的 PD。在分布式系统里面，一些系统采用了无中心化的架构设计方案，但我个人觉得，有一个中心化的单点，能更好的用全局视角来控制和调度整个系统，而且实现起来很简单。

在 Kudu 里面，Master 自己也是一个单一的 Tablet table，只是对用户不可见。它保存了整个集群的元信息，并且为了性能，会将其全部缓存到内存上面。因为对于集群来说，元信息的量其实并不大，所以在很长一段时间，Master 都不会有 scale 的风险。同时 Master 也是采用 Raft 机制复制，来保证单点问题。

这个设计其实跟 PD 是一样的，PD 也将所有的元信息放到内存。同时，PD 内部集成 etcd，来保证整个系统的可用性。跟 Kudu Master 不一样的地方在于，PD 是一个独立的组件，而 Kudu 的 Master 其实还是集成在 Kudu 集群里面的。

Kudu 的 Master 主要负责以下几个事情：

Catalog manager

Master 的 catalog table 会管理所有 table 的一些元信息，譬如当前 table schema 的版本，table 的 state（creating，running，deleting 等），以及这个 table 在哪些 Tables 上面。

当用户要创建一个 table 的时候，首先 Master 在 catalog table 上面写入需要创建 table 的记录，table 的 state 为 CREATING。然后异步的去选择 Tablet servers 去创建相关的元信息。如果中间 Master 挂掉了，table 记录里面的 CREATING state 会表明这个 table 还在创建中，新的 Master leader 会继续这个流程。

Cluster coordinator

当 Tablet server 启动之后，会给 Master 注册，并且持续的给 Master 进行心跳汇报消后续的状态变化。

虽然 Master 是整个系统的中心，但它其实是一个观察者，它的很多信息都需要依赖 Tablet server 的上报，因为只有 Tablet server 自己知道当前自己有哪一些 tablet 在进行 Raft 复制，Raft 的操作是否执行成功，当前 tablet 的版本等。因为 Tablet 的状态变更依赖 Raft，每一次变更其实就在 Raft log 上面有一个对应的 index，所以上报给 Master 的消息一定是幂等的，因为 Master 自己会比较 tablet 上报的 log index 跟当前自己保存的 index，如果上报的 log index 是旧的，那么会直接丢弃。

这个设计的好处在于极大的简化了整个系统的设计，如果要 Master 自己去负责管理整个集群的状态变更，譬如 Master 给一个 tablet 发送增加副本的命令，然后等待这个操作完成，在继续处理后面的流程。整个系统光异常处理，都会变得特别复杂，譬如我们需要关注网络是不是断开了，超时了到底是成功了还是失败了，要不要再去 tablet 上面查一下？

相反，如果 Master 只是给 tablet 发送一个添加副本的命令，然后不管了，剩下的事情就是一段时间后让 tablet 自己上报回来，如果成功了继续后面的处理，不成功则尝试在加一次。虽然依赖 tablet 的上报会有延迟（通常情况，只要有变动，tablet 会及时的上报通知，所以这个延迟其实挺小的），整个架构简单了很多。

其实看到这里的时候，我觉得非常的熟悉，因为我们也是采用的这一套架构方案。最开始设计 PD 的时候，我们还设想的是 PD 主动去控制 TiKV，也就是我上面说的那套复杂的发命令流程。但后来发现实在是太复杂了，于是改成 TiKV 主动上报，这样 PD 其实就是一个无状态的服务了，无状态的服务好处就是如果挂了，新启动的 PD 能立刻恢复（当然，实际还是要做一些很多优化工作的）。

Tablet directory

因为 Master 知道集群所有的信息，所以当 client 需要读写数据的时候，它一定要先跟 Master 问一下对应的数据在哪一个 Tablet server 的 tablet 上面，然后才能发送对应的命令。

如果每次操作都从 Master 获取信息，那么 Master 铁定会成为一个性能瓶颈，鉴于 tablet 的变更不是特别的频繁，所以很多时候，client 会缓存访问的 tablet 信息，这样下次再访问的时候就不用从 Master 再次获取。

因为 tablet 也可能会变化，譬如 leader 跑到了另一个 server 上面，或者 tablet 已经不在当前 server 上面，client 会收到相关的错误，这时候，client 就重新再去 Master 获取一下最新的路由信息。

这个跟我们的做法仍然是一样的，client 缓存最近的路由信息，当路由失效的时候，重新去 PD 获取一下。当然，如果只是单纯的 leader 变更，其实返回的错误里面通常就会带上新的 leader 信息，这时候 client 直接刷新缓存，在直接访问了。

Tablet storage

Tablet server 是 Kudu 用来存放实际数据的服务，为了更好的性能，Kudu 自己实现了一套 tablet storage，而没有用现有的开源解决方案。Tablet storage 目标主要包括：

- 快速的按照 Column 扫描数据
- 低延迟的随机更新
- 一致的性能

RowSets

Tablets 在 Kudu 里面被切分成更小的单元，叫做 RowSets。一些 RowSets 只存在于内存，叫做 MemRowSets，而另一些则是使用 disk 和 memory 共享存放，叫做 DiskRowSets。任何一行数据只存在一个 RowSets 里面。

在任何时候，一个 tablet 仅有一个单独的 MemRowSet 用来保存最近插入的数据。后台有一个线程会定期的将 这些 MemRowSets 刷到 disk 上面。

当一个 MemRowSet 被刷到 disk 之后，一个新的空的 MemRowSet 被创建出来。之前的 MemRowSet 在刷到 disk 之后，就变成了 DiskRowSet。当刷的同时，如果有新的写入，仍然会写到这个正在刷的 MemRowSet 上面，Kudu 有一套机制能够保证新写入的数据也能一起被刷到 disk 上面。

MemRowSet

MemRowSet 是一个支持并发，提供锁优化的 B-tree，主要基于 MassTree，也有一些不同：

1. 因为 Kudu 使用的是 MVCC，所以任何的删除其实也是插入，所以这个 tree 没有删除操作。
2. 不支持任意的 in-place 数据变更操作，除非这次操作不会改变 value 的大小。
3. 将 Leaf link 起来，类似 B+-tree，这样对于 scan 会有明显的性能提升。
4. 并没有完全实现 `trie of trees` ，是只是使用了一个单一 tree，因为 Kudu 并没有太多高频随机访问的场景。

DiskRowSet

当 MemRowSets 被刷到 disk 之后，就变成了 DiskRowSets。当 MemRowSets 被刷到 disk 的时候，Kudu 发现超过 32 MB 了就滚动一个新的 DiskRowSet。因为 MemRowSet 是顺序的，所以 DiskRowSets 也是顺序的，各滚动的 DiskRowSet 里面的 primary keys 都是不相交的。

一个 DiskRowSet 包含 base data 和 delta data。Base data 按照 column 组织，也就是通常我们说的列存。各个 column 会被独立的写到 disk 里面一段连续的 block 上面，数据会被切分成多个 page，使用一个 B-tree 进行高效索引。

除了刷用户自定义的 column，Kudu 还默认将 primary key index 写到一个 column，同时使用 Bloom filter 来保证能快速通过找到 primary key。

为了简单，当 column 的数据刷到 disk，它就是默认 immutable 的了，但在刷的过程中，有可能有更新的数据，Kudu 将这些数据放到一个 delta stores 上面。Delta stores 可能在内存 DeltaMemStores，或者 disk DeltaFiles。

Delta store 维护的一个 map，key 是 `(row_offset, timestamp)` ，value 就是 RowChangeList 记录。Row offset 就是 row 在 RowSet 里面的索引，譬如，有最小 primary key 的 row 在 RowSet 里面是排在最前面的，它的 offset 就是 0。Timestamp 就是通常的 MVCC timestamp。

当需要给 DiskRowSet 更新数据的时候，Kudu 首先通过 primary key 找到对应的 row。通过 B-tree 索引，能知道哪一个 page 包含了这个 row，在 page 里面，可以计算 row 在整个 DiskRowSet 的 offset，然后就把这个 offset 插入到 DeltaMemStore 里面。

当 DeltaMemStore 超过了一个阀值，一个新的 DeltaMemStore 就会生成，原先的就会被刷到 disk，变成 immutable DeltaFile。

每个 DiskRowSet 都有一个 Bloom filter，便于快速的定位一个 key 是否存在于该 DiskRowSet 里面。DIskRowSet 还保存了最小和最大的 primary key，这样外面就能通过 key 落在哪一个 key range 里面，快速的定位到这个 key 属于哪一个 DiskRowSet。

Compaction

当做查询操作的时候，Kudu 也会从 DeltaStore 上面读取数据，所以如果 DeltaStore 太多，整个读性能会急剧下降。为了解决这个问题，Kudu 在后台会定期的将 delta data 做 compaction，merge 到 base data 里面。

同时，Kudu 还会定期的将一些 DIskRowSets 做 compaction，生成新的 DiskRowSets，对 RowSet 做 compaction 能直接去掉 deleted rows，同时也能减少重叠的 DiskRowSets，加速读操作。

总结

上面对 Kudu 大概进行了介绍，主要还是参考 Kudu 自己的论文。Kudu 在设计上面跟 TiKV 非常类似，所以对于很多设计，我是特别能理解为啥要这么做的，譬如 Master 的信息是通过 tablet 上报这种的。Kudu 对 Raft 在实现上面做了一些优化，以及在数据 partition 上面也有不错的做法，这些都是后面能借鉴的。

对于 Tablet Storage，虽然 Kudu 是自己实现的，但我发现，很多方面其实跟 RocksDB 差不了多少，类似 LSM 架构，只是可能这套系统专门为 Kudu 做了定制优化，而不像 RocksDB 那样具有普适性。对于 storage 来说，现在我们还是考虑使用 RocksDB。

另外，Kudu 采用的是列存，也就是每个列的数据单独聚合存放到一起，而 TiDB 这边还是主要使用的行存，也就是存储整行数据。列存对于 OLAP 非常友好，但在写入的时候压力可能会比较大，如果一个 table 有很多 column，写入性能影响会非常明显。行存则是对于 OLTP 比较友好，但在读取的时候会将整行数据全读出来，在一些分析场景下压力会有点大。但无论列存还是行存，都是为满足不同的业务场景而服务的，TiDB 后续其实可以考虑的是行列混存，这样就能适配不同的场景了，只是这个目标比较远大，希望感兴趣的同学一起加入来实现。