新型分布式数据库中的关键算法与模型探究
——以 TiKV 与 TiDB 为例

陈典　　陈佳阳　　陈梓钿　　邓倍铭

日期：June 30, 2022

1 前言

新型分布式数据库（在本报告中具体指云原生时代的新型数据库产品架构）目前正处于蓬勃发展阶段。以近几年提出的 NewSQL 为例，这类架构需具备对海量数据的存储管理能力，在许多业务场景下还要保持传统数据库的 SQL 模型和事务特性。

在本报告中，我们对新型分布式数据库中的四种关键算法/模型做了初步探究，包括：分布式共识算法 Raft（第 2 节）、分布式事务模型 Percolator（第 3 节）、Volcano 查询优化模型（第 4 节）、F1 异步模式变更算法（第 5 节），它们都被学术界和工业界广泛认可。其中，前两部分属于偏底层的分布式机制，后两部分属于偏上层的关系型数据模型。

为了更好地了解上述算法和模型在实际工程中的应用，我们选择了 modb 排行榜[1] 中近两年蝉联首位的分布式关系型数据库 TiDB 作为分析案例。TiDB 也属于 NewSQL 范畴，其底层依赖于分布式键值型数据库 TiKV，二者均为开源项目，且已在国内金融、电商领域取得广泛应用，具有较强的代表性。在每一节的结尾，我们将讨论特定算法或模型在 TiKV 或 TiDB 中的一些实现、优化细节。

2 Raft：分布式共识算法

2.1 Raft 与一致性

分布式存储系统通常通过维护多个副本来进行容错，提高系统的可用性。要实现此目标，就必须要解决分布式存储系统的最核心问题：*维护多个副本的一致性*。一致性（consensus）是构建具有容错性（fault-tolerant）的分布式系统的基础。在一个具有一致性的集群里面，同一时刻所有的结点对存储在其中的某个值都有相同的结果，即对其共享的存储保持一致。集群还应该具有自动恢复的性质，当少数结点失效的时候不影响集群的正常工作；当大多数结点失效的时候，则会停止服务，避免返回错误的结果。

Raft 是斯坦福大学的 Diego Ongaro、John Ousterhout 以易理解为目标设计的一致性算法，于 2013 年提出 [6]，Diego Ongaro 后来在其博士学位论文中做出了进一步优化 [5]。

[1] https://www.modb.pro/dbRank

2.2 Raft 的基础设计

2.2.1 复制状态机

一致性协议通常基于复制状态机（Replicated State Machine），即所有结点都从同一个状态出发，都经过同样的一些指令，最后到达同一个状态。复制状态机用于解决分布式系统中的各种容错问题，例如具有单个 Leader 的大规模系统（GFS、HDFS 等）通常使用单独的复制状态机来进行 Leader 的选举和存储重新选举需要的配置信息。如图 1 所示，复制状态机要能容忍很多错误场景。

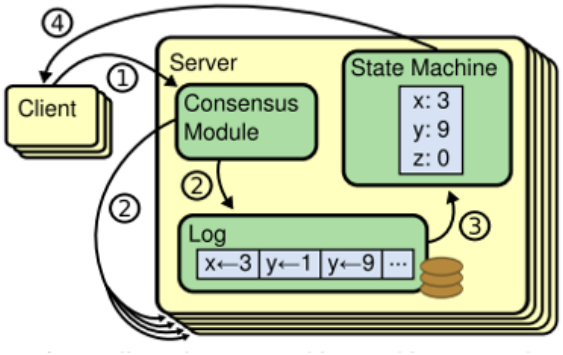

图 1: Raft 复制状态机的架构

复制状态机的典型实现是基于复制日志，其中包含三个组件：

- 状态机 *(State Machine)*：状态机会从日志里面取出所有的命令，然后顺序执行一遍，从而对外提供具有一致性保证的数据。
- 日志 *(Log)*：保存了所有修改指令的日志项。
- 一致性模块 *(Consensus Module)*：用于保证写入日志的一致性，这也是 Raft 算法核心内容。

状态机可以理解为一个确定的应用程序，即相同的输入必然产生相同的输出。每个节点上存储了一个包含一系列指令的日志，复制状态机按序执行这些指令，由于每一个日志在相同位置上的指令是相同的，所以每个状态机具有一致的执行序列。至于如何实现日志完全一致的复制，则是一致性模块的职责，也就是我们下面要介绍的 Raft 算法。

2.2.2 Raft 节点状态

Raft 算法中的节点状态十分关键，可以说是后续一系列机制的基石，在任何时候有三种可能的取值，其转换逻辑如图 2 所示：

- *Leader*：处于该状态的节点负责处理客户端的请求，同时还需要协调日志的复制。在任意时刻，最多允许存在 1 个 *Leader*，其他节点都是 *Follower*。集群在选举期间可能短暂处于没有 *Leader* 的场景。
- *Follower*：处于该状态的节点是被动的，不主动提出请求，只是响应 *Leader* 和 *Candidate* 的请求（节点之间的通信通过 RPC 进行）。

- *Candidate*：从 *Follower* 转变为 *Leader* 的过渡状态。因为 *Follower* 是一个完全被动的状态，所以当重新选举时，需要将自己提升为 *Candidate*，再发起选举。

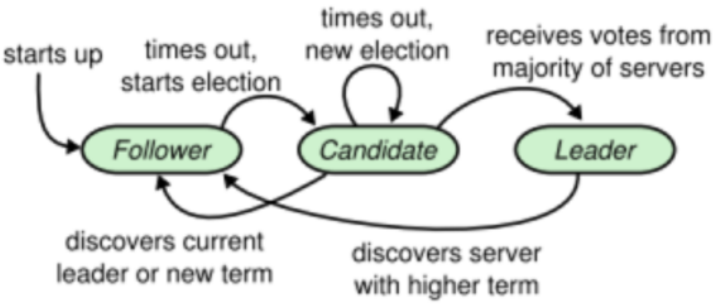

图 2: Raft 节点状态的转换

由图 2 可知，所有的节点都是从 *Follower* 开始，如果经过一段时间后收不到来自 *Leader* 的信号，那么就认定 *Leader* 已经崩溃了，需要进行新一轮的选举，将状态变更为 *Candidate*。*Candidate* 有可能接收多数节点的投票，被选举为 *Leader*，也有可能因选举失败或超时回退为 *Follower*。最后，如果 *Leader* 发现自己的任期（在 2.2.3中介绍）不是最新的，则会主动变更为 *Follower*。

2.2.3 Term

任期（Term）是 Raft 节点的另一个关键属性。在分布式系统中，由于节点的物理时间戳都不统一，因此需要一个逻辑时间戳来表明事件发生的先后顺序，Term 正是起到了逻辑时间戳的作用。如图 3 所示，Raft 的运行过程可以被抽象划分为一系列任意长度的 Term。

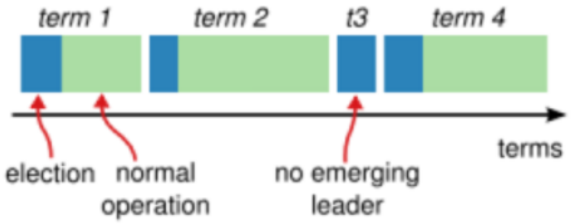

图 3: 关于 Term 的过程抽象

每个 Term 开始于选举，这一阶段每个 *Candidate* 都试图成为 *Leader*，如果选举成功，它就在该 Term 的有效期内履行处理客户端请求、协调日志备份的职责。Term 在 Raft 中始终保持单调递增，一个节点的 Term 越大，那么它所拥有的日志就越准确。

每个节点都会维护自己当前的任期（Current Term），并且持久化存储。当 *Leader* 宕机后再恢复，会利用 Current Term 判断自己是否已经过时，若是，就会回退为 *Follower*。节点的 Term 通常会在节点之间通信时改变：如果某个节点发现 Current Term 小于其它节点，那么这个节点必须更新该值，与集群其他节点保持一致；如果 *Candidate* 或者 *Leader* 发现自己的 Term 不是最新的，它就必须降级为 *Follower*；如果某个节点收到一个过时的请求（拥有较小的 Term），它会拒绝该请求。

2.3 一致性机制的实现

Raft 的一致性机制是基于日志由 *Leader* 到 *Follower* 的单向传递。也就是说 *Leader* 相当于一个总控节点，负责接受客户端的请求，并且把这一过程产生的日志发送给各个 *Follower* 进行复制。单向的特性保证了只有 *Leader* 能决定何时提交一个日志。通过这一机制，Raft 将一致性难题分解为三个相对独立的子问题：

- *Leader* 选举：一开始每个节点都是 *Follower*，需要通过选举决定由谁来做 *Leader*，当 *Leader* 宕机的时候也需要重新进行选举。
- 日志复制：当 *Leader* 选举出来之后，需要把后续产生的日志复制到每个 *Follower* 节点上，需要保证所有日志项是有序且正确的。
- 安全性：当 *Leader* 把一个日志项复制到绝大多数的 *Follower* 后，需要提交（Commit）这个日志项，并执行（Apply）其包含的指令。这里的安全性是指所有的节点按照相同的顺序执行相同的日志项。

简单来说，一次 Raft 流程有如下操作：

1. *Leader* 收到客户端发送的请求；
2. *Leader* 将该请求中的指令和数据追加到自己的日志中；
3. *Leader* 将对应的日志项发送给集群中的 *Follower*；
4. *Leader* 等待 *Follower* 的结果，如果大多数节点提交了这些日志项，则告知上层应用可以进行相应操作（Apply）；
5. *Leader* 将结果返回给客户端，继续处理下一次请求。

2.4 TiKV 对 Raft 流程的优化

2.4.1 异步 Apply

当一日志项被大部分节点追加到本地的存储层后，就认为这个日志项被提交了，提交后的日志项不会被修改或删除，此时数据的一致性已经得到了保证。基于这一认识，TiKV 将日志的提交与执行过程分开，用另一个线程去异步地执行日志，从而减少后续请求的阻塞时间。在这种情况下，Raft 流程改为：

1. *Leader* 接受一个客户端发送的请求；
2. *Leader* 将该请求中的指令和数据追加到自己的日志中并发送给 *Follower*；
3. *Leader* 继续接受其他客户端的请求，持续进行步骤 2；
4. *Leader* 发现日志项已经被提交，在另一个线程中 Apply；
5. *Leader* 异步 Apply 之后，返回结果给对应的客户端。

异步 Apply 的改进令 TiKV 得以完全并行处理日志的复制和执行。虽然对于一个特定的客户端来说，一次请求仍然要走完完整的 Raft 流程，但在并发场景下，整个系统的吞吐量得到了大幅提高。

2.4.2 批处理与管道通信

- 批处理（*Batch*）：对于 Raft 来说，*Leader* 可以一次收集多个请求，然后按批发送给 *Follower*。当然，这里需要一个预先设置的参数来限制批处理的数据量上限。
- 管道（*Pipeline*）：在 Raft 的初始设计中，*Leader* 会维护一个 NextIndex 的变量来表示下一个给 *Follower* 发送的日志位置，通常情况下，只要 *Leader* 跟 *Follower* 建立起了连接，我们都会认为网络是稳定互通的。所以 TiKV 做出了如下优化：当 *Leader* 给 *Follower* 发送了一批日志项之后，它可以直接更新 NextIndex，并且立刻发送后面的日志项，不需要等待 *Follower* 的返回。如果网络出现了错误，或者 *Follower* 返回一些错误，*Leader* 才需要调整 NextIndex 然后重新发送。

3 Percolator：分布式事务模型

3.1 背景介绍

Percolator 是 Google 在 2010 年发表的论文《Large-scale Incremental Processing Using Distributed Transactions and Notifications》[7] 中提出的一种分布式事务解决方案，在超大规模的数据集，且数据变化多、小范围增量更新的情况下（即 OLTP 场景）满足业务对一致性的要求。Percolator 支持 ACID 语义，并实现了快照隔离（Snapshot Isolation）的事务隔离级别，可以将其看作是一种通用的分布式事务解决方案。在实现上，Percolator 是基于 Google 更早之前提出的 BigTable [1]，其本质上是一个二阶段提交协议，利用 BigTable 的行事务机制协助处理分布式事务。

3.2 架构组成

在 Percolator 分布式事务解决方案中，主要包含三个组件，分别是 Client、Timestamp Oracle 和 BigTable，下面将展开介绍这三个组件。

3.2.1 Client

在两阶段提交算法中有两种角色，即协调者和参与者，在 Percolator 中，Client 充当协调者的角色，负责发起和提交事务。换言之，Client 是整个 Percolator 协议的控制中心，是两阶段提交的协调者。

3.3 Timestamp Oracle

Timestamp Oracle 是一个全局的授时服务，用于向服务中的其他组件提供全局唯一且递增的时间戳。在 Percolator 中，系统需要依赖于 Timestamp Oracle 提供的全局唯一且递增的时间戳来实现快照隔离级别（即根据数据的时间戳来确保事务的可重复读）。在事务的发起和提交阶段，Client 也需要从 Timestamp Oracle 中获取一个时间戳，用于保证顺利协调不同事务。

Percolator 基于 Timestamp Oracle 实现的快照隔离机制具体如下：

Column	Use
c:lock	An uncommitted transaction is writing this cell; contains the location of primary lock.
c:write	Committed data present; stores the BigTable timestamp of the data.
c:data	Stores the data itself.
c:notify	Hint: observers may need to run.
c:ack_O	Observer "O" has run; stores start timestamp of successful last run.

表 1: Percolator 元数据的列名及其含义

- 事务中所有的读操作都会读到一个一致性快照（Consistent Snapshot）的数据，保证了可重复读（Repeated Read）；
- 两个并发事务同时对同一个单元（Cell）写入时，只会有一个事务能够提交成功；
- 当一个事务提交时，如果发现本事务更新的一些数据，被其他具有更大的起始时间戳（*start_ts*）的事务修改了，则执行回滚（失败），否则提交（成功）；
- 快照隔离存在写倾斜（Write Skew）问题，即两个事务读的数据集有重叠，但是写入的数据集没有重叠，此时虽然两个事务都可以成功提交，但是相互都没有看见对方写入的新数据，不符合可序列化（Serializable）的隔离级别。但是快照隔离相对可序列化有更好的读性能（读操作不需要加锁），在现实中应用更广泛。

3.3.1 BigTable

BigTable 同样是 Google 研发的分布式数据，用于解决海量数据存储的问题，相关论文发表在了 2006 年的 OSDI 大会上 [1]。BigTable 从数据模型上可以理解为一个多维度的有序字典，其中 Key 由三元组 {*row, column, timestamp*} 组成，Value 则是任意的字节序列。BigTable 提供了单行的跨多列的事务能力，Percolator 利用这个特性来保证对同一行的多个列的操作是具有原子性的。以 BigTable 为基础，Percolator 将元数据存储在特殊的列中，具体如表1所示。

3.4 事务处理

分布式事务处理主要分为写入逻辑和读取逻辑。

3.4.1 写入逻辑

Percolator 采用两阶段提交算法来提交事务，写入逻辑被分为 Prewrite 阶段和 Commit 阶段。

Prewrite 阶段 主要流程如下：

1. 事务 T_1 开始，从全局时间授权模块 Timestamp Oracle 获取 *start_ts*，在所有需要写操作的行中选一个作为 primary，其他的均为 secondary；
2. 将 primary 行写入 c:lock 列（上锁），写之前会检查是否有冲突；
3. primary 上锁成功后，写入 c:data 列更新新版本的数据，并以起始时间戳为版本号；

4. primary 上锁流程结束后，开始 secondary 行的 Prewrite 流程，流程与 primary 行上锁流程相似，但是锁类型不一样，且锁的内容会多出对 primary 行的指向（*lock_type=>secondary_lock*，*primary_key=>primary_key*）；
5. Prewrite 流程任何一步发生错误，都会回滚，删除新加的 c:lock 列和新加的 c:data 列。

Commit 阶段 主要流程如下：

1. Commit 阶段开始时，事务 T_1 从 Timestamp Oracle 中获取 commit 时间戳（要求大于其起始时间戳）；
2. 提交 primary 行，写入对应的 c:write 列数据 {*key, commit_ts, start_ts*}，表明 *start_ts* 对应的数据已提交；
3. 删除 primary 行的 c:clock 列（释放锁）；
4. 如果 primary 行提交失败，整个事务进行回滚，删除新加的 c:lock 列和新加的 c:data 列；
5. 如果 primary 行提交成功，则可异步提交其余的 secondary 行。

3.4.2 读取逻辑

1. 事务 T_1 获取 *start_ts*；
2. 检查当前我们要读取的数据是否存在一个时间戳在 [0, *start_ts*] 范围内的锁：
   - 如果存在一个时间戳在 [0, *start_ts*] 范围的锁，那么意味着当前的数据被一个更早启动的事务 T_2 锁定了，但是此时 T_2 还没有提交。因为当前无法判断这个锁定数据的事务是否会被提交，所以当前的读请求不能被满足，只能等待锁被释放之后，再尝试读取数据；
   - 如果没有锁，或者锁的时间戳大于 *start_ts*，那么读请求可以被满足；
3. 从 c:write 列中获取在 [0, *start_ts*] 范围内的最大 *commit_ts* 的记录，然后依此获取到对应的 *start_ts*；
4. 根据上一步获取的 *start_ts*，从 c:data 列获取对应的记录。

3.5 Percolator 在 TiKV 中的实现

TiKV 底层的存储引擎使用的是 RocksDB。RocksDB 提供了原子性的批量写操作，可以实现如上所述 Percolator 对行事务的要求。在 RocksDB 中，数组的组织方式是基于列族（CF）的，一个 RocksDB 实例可以有多个 CF，每个 CF 是一个隔离的 key 命名空间，并且拥有独立的 LSM-tree 存储结构。同一个 RocksDB 实例中的多个 CF 通过共享一个预写日志（Write-Ahead Log）来保证多个 CF 写操作的原子性。

在 TiKV 中，一个 RocksDB 实例有三个 CF：CF_DEFAULT、CF_LOCK、CF_WRITE，分别对应着 Percolator 的 c:data、c:lock、c:write。要对每个 key 存储多个版本的数据，只需简单地将 key 和 timestamp 结合成一个 internal key 存储在 RocksDB 中即可。

在 TiKV 中，CF_WRITE 中有 4 中不同类型的数据：

1. Put：将 CF_DEFAULT 中的一条对应的数据写入到 CF_WRITE 中；

2. Delete：表示写入操作是一个 Delete 操作；
3. Rollback：当回滚一个事务的时候，在 CF_WRITE 中插入一条 Rollback 的记录；
4. Lock。

3.6 Percolator 在 TiKV 中的优化

3.6.1 Parallel Prewrite

在 TiKV 中，一个事务不会以 one-by-one 的形式进行 Prewrite。当存在多个 TiKV 节点时，系统会在多个节点上并行地执行 Prewrite。在 TiKV 的实现中，当提交一个事务时，事务中涉及的 Key 会根据分区（Region）情况被分成多个 batch，每个 batch 在 Prewrite 阶段会并行地执行，不需要关注 primary 是否第一个 Prewrite 成功。

如果事务在 Prewrite 阶段发生冲突，则会被回滚。在执行回滚时，TiKV 是在 CF_WRITE 中插入一条 Rollback 记录，而不是如 Percolator 论文中描述的那样删除对应的锁记录。这条 Rollback 记录表示对应的事务已经回滚了，如果后续收到它的其他 Prewrite 请求（这种情况可能在网络异常的时候出现），则直接拒绝；如果让这些延迟的 Prewrite 请求成功，正确性还是可以保证，但是这个 key 会被持续锁定直到过期。

3.6.2 Short Value in Write Column

当访问一个 value 时，系统需要先从 CF_WRITE 中找到 key 对应的最新版本（即 *start_ts*），然后从 CF_DEFAULT 中找到具体的记录。如果 value 比较小的话（称为 short value），在 RocksDB 上执行两次查询的开销相对来说有点大。针对这一问题，TiKV 做了一个优化：如果 value 比较小，在 Prewrite 阶段，系统不会将 value 放到 CF_DEFAULT 中，而是将其放在 CF_LOCK 中；在 commit 阶段，这个 value 会从 CF_LOCK 移动到 CF_WRITE 中。因此，在访问这个 short value 时，只需访问 CF_WRITE 就可以了，减少了一次 RocksDB 查询，从而减小开销。

3.6.3 Point Read Without Timestamp

对于每个事务，系统需要先分配一个 *start_ts*，然后保证事务只能看到在 *start_ts* 之前提交的数据。如果一个事务只读取一个 key 的数据，则并不需要为其分配一个 *start_ts*，只需要读取这个 key 的最新数据即可。

3.6.4 Calculated Commit Timestamp

为了保证实现快照隔离级别，系统需要让 *commit_ts* 足够大，不会出现一个事务在一次有效读操作之前被提交，否则将无法保证可重复读操作。

3.7 小结

优点 Percolator 的优点主要有以下两方面：

1. 事务管理建立在存储系统之上，整体系统架构清晰，系统扩展性好，实现简单；

2. 在事务冲突较少的场景下，读写性能良好。

不足 Percolator 的不足主要有以下两方面：

1. 在事务冲突较多的场景下，性能较差，因为出现了冲突之后需要不断重试，开销很大；
2. 在采用多版本并发控制（MVCC）算法的情况下也会出现读等待的情况，当存在读写冲突时，对读性能有较大影响。

4 Volcano / Cascades：SQL 查询优化模型

4.1 SQL 优化过程

在 TiDB 中，SQL 优化的过程可以分为逻辑优化和物理优化两个部分。

- 逻辑优化：主要是基于规则的优化（Rule-Based Optimization，RBO），依次遍历内部定义实现的优化规则，不断地调整 SQL 的逻辑执行计划，对查询做逻辑上的等价变化。可以视为将一棵逻辑算子树（Logical Plan Tree）进行逻辑等价的变化，最后的结果是一棵更优的逻辑算子树。
- 物理优化：是将一棵逻辑算子树转换成一棵物理算子树（Physical Plan Tree），会为逻辑查询计划中的算子选择某个具体的实现。这一优化过程需要依靠统计信息以决定哪一种方式代价最低，是基于代价的优化（Cost-Based Optimization，CBO），可以视为在逻辑优化的基础上，为优化过后的算子树中的算子选择更优的具体实现。

4.2 Volcano 框架

Volcano 是一个包含了优化框架和执行框架的项目，其目标是构建具有完全扩展性的优化器生成器 [4]。Volcano 并非具体的优化器实现，而是提供了一个实现、构建优化器的框架，同时也是 Cascades 优化器的前身。

Volcano Optimizer Generator 本身的定位是一个优化器的 “生成器”，其核心贡献是提供了一个搜索引擎。作者提供了一个数据库查询优化器的基本框架，而数据库实现者要为自己的数据模型实现相应的接口，之后便可以生成一个查询优化器。

Volcano Optimizer Generator 在优化器方向提出了一些方法：

1. 使用 “Logical Algebra” 来表示各种关系代数算子，而使用 “Physical Algebra” 来表示各种关系代数算子的实现算法。Logical Algebra 之间使用 Transformation 来完成变换，而 Logical Algebra 到 Physical Algebra 之间的转换使用基于代价（CBO）的选择。
2. 使用 “Rule” 来表示各种变化。例如 Logical Algebra 之间的变化使用 Transformation Rule；而 Logical Algebra 到 Physical Algebra 之间的转换使用 Implementation Rule。
3. 使用 “Property” 来表示各个算子、表达式的结果。Logical Propery 主要包括算子的模式、统计信息等；Physical Property 表示算子所产生的数的物理属性，比如按照某个键排序、按照某个键分布在集群中等。
4. 优化器搜索过程采用自顶向下的记忆化搜索提高效率。

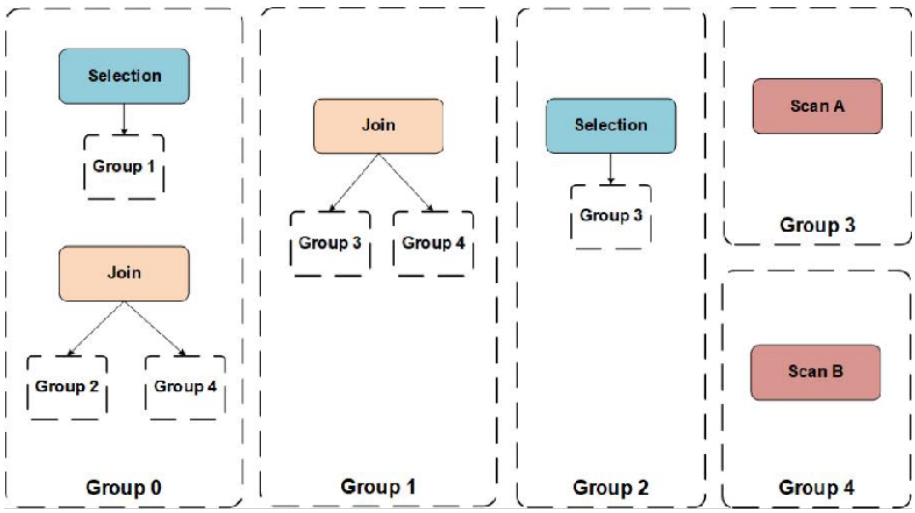

图 4: Memo 数据结构

4.3 Cascades 优化器

Cascades 优化器是 Volcano 的后续作品，这一优化器在后者的基础上做了进一步的优化，主要提出了 Memo、Rule、Pattern、Searching Algorithm 等重要概念 [3]。

Memo Cascades 优化器的搜索空间是一个由关系代数算子树所组成的森林，Memo 为保存这个森林的数据结构，可以高效地保存搜索状态（图 4 ）。Memo 的 Group 设计可以避免存储冗余的算子，每个 Group 中会存在多个逻辑等价的 Expression。

Rule 在 Volcano 优化器中，Rule 被分为了 Transformation Rule 和 Implementation Rule（4.2）。Transformation Rule 用来在 Memo 中添加逻辑等价的 Group Expression，其具有原子性，只作用于算子树的一个局部小片段，每个 Transformation Rule 都有自己的匹配条件，Cascades 通过不停应用可以匹配上的 Transformation Rule 来扩展搜索的空间，寻找可能的最优解。Implementation Rule 是为 Group Expression 选择物理算子。Cascades 没有显式对两者做区分。

Pattern 用于描述 Group Expression 的局部特征。每个 Rule 都有自己的 Pattern，只有满足了相应 Pattern 的 Group Expression 才能够应用该 Rule，如图 5 所示。

Searching Algorithm 优化规则和搜索过程是 Cascades 的核心，也是优化器的工作重心。在传统的优化器实现中，往往是面向过程地应用优化规则，对算子树进行变换。这种方式往往难以扩展，需要考虑规则之间的应用顺序。而 Cascades 在处理这一问题时，将搜索过程与具体的规则解耦，用面向对象的方式对优化规则进行建模，规则的编写不需要关心搜索过程。具体地，Cascades 为 Rule 的应用顺序做了很细致的设计，每个 Rule 都有 *promise* 和 *condition* 两个方法，其中 *promise* 用来表示 Rule 在当前搜索过程中的重要性，而 *condition* 直接通过返回一个布尔值决定一个 Rule 是否可以在当前过程中被应用。当一个 Rule 被成功应用之后，会计算下一步有可能会被应用的 Rule 的集合。另外，Cascades 使用 Branch-And-Bound 方法对搜索空间进行剪枝，在搜索的过程中为算子设置其代价上限，如果搜索中超过了预设的上限，将对这个搜索分支预先进行剪枝。

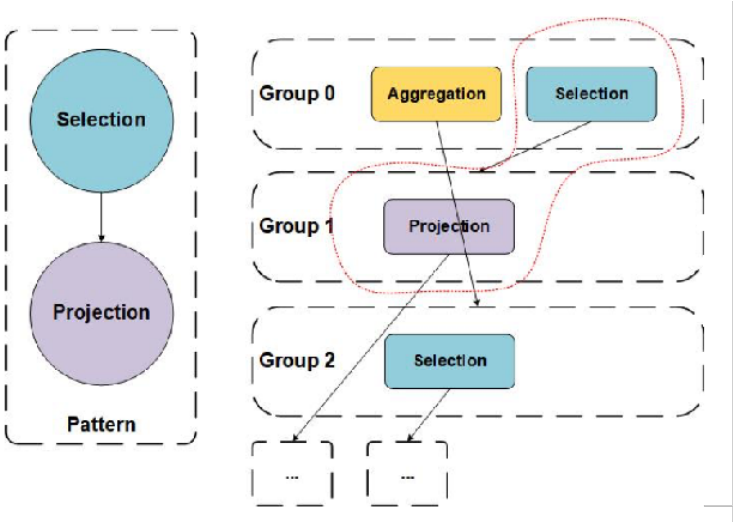

图 5: 一个 Selection → Projection 的 Pattern

4.4 TiDB 中 Cascades Planner 的搜索过程

1. 预处理阶段：对原始的逻辑算子树做“一定更优”的逻辑变换，例如列裁剪、最大最小消除、投影消除、谓词下推等。
2. 逻辑搜索阶段：对输入的逻辑算子树做逻辑上的等价变换。但不同的是，此处先将 Logical Plan Tree 转换成 Group Tree，通过在 Group Tree 应用预定义的转换规则来实现逻辑上的等价变化。
3. 物理实现阶段：将逻辑计划转化成代价最小的物理计划，Cascades Planner 会为一个 Group Tree（一组逻辑等价的 Logical Plan Tree）选择物理计划。

5 Google F1：异步模式变更算法

5.1 F1 特性与基础设计

F1 是 Google 研发的分布式关系型数据库管理系统 [9]，在保证一致性、高可用性的基础上，提供一种新型的关系型数据模式演化（schema evolution）协议。模式演化是指在不丢失数据的情况下改变数据库定义的能力，已经被业界研究了近三十年。Google 提出 F1 的动机是为了让其广告系统具备更好的可扩展性，该系统的存储层最初是基于分片的 MySQL 构建的，难以应付日益增长的业务数据和相对频繁的模式变更需求。

5.1.1 特性及约束

F1 的模式演化协议以如下所述的系统特性为基础 [8]：

1. 大规模的分布式。一个 F1 实例由数百个独立的服务器组成，运行在分布在全球许多数据中心的共享机器集群上。
2. 关系型模式。每个 F1 服务器都有一个描述表、列、索引和约束的关系型模式的副本，对模式的任何修改都需要分布式机制来更新所有服务器。

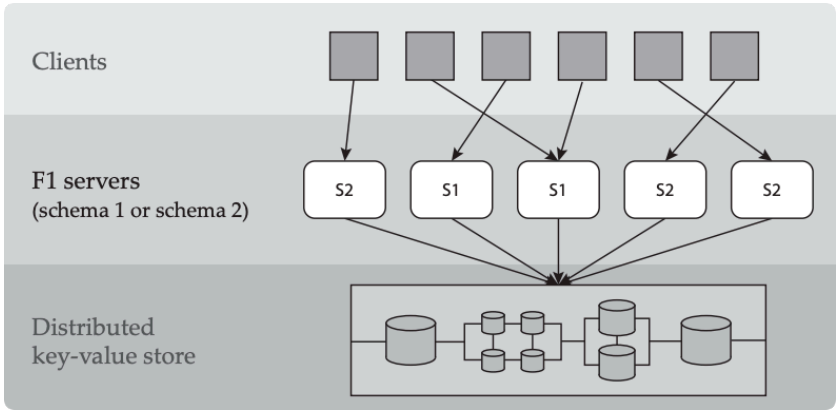

图 6: F1 模式变更基础设计

3. *共享数据存储*。所有数据中心的 F1 服务器都可以访问下层的键值存储系统（这里是 Spanner）中的所有数据，服务器之间不存在数据分区。
4. *无状态服务器*。F1 服务器必须能够容忍机器故障、抢占、网络资源的不可用，因此被设计成无状态的——客户端可以连接到任何 F1 服务器，同一事务的不同语句也可以由不同的 F1 服务器执行。
5. *无全局成员*。由于 F1 服务器的无状态特性，F1 不需要实现全局成员关系协议。没有可靠的机制来确定当前运行的是哪个服务器，无法（同时也不需要）实现显式的全局同步。

此外，F1 的模式演化协议还受到如下的业务需求约束：

1. *完全的数据可用性*。F1 是 Google 业务基础设施的关键部分，数据的不可用会直接影响经济效益，因此，对模式的修改（例如，锁定一个列以建立新索引）不能使数据库的任一部分脱机。
2. *最小化对性能的影响*。上面提到，F1 的主要任务是支持 Google 的广告系统。由于该系统在多个团队和项目之间共享，模式更改非常频繁，如果每次更改都需要以明显的性能降级为代价，势必会削弱整个系统的可用性。
3. *异步的模式变更*。F1 中不存在全局成员关系，因此无法对所有服务器进行同步的模式更改，不同的 F1 服务器会在不同的时间过渡到新模式。换言之，同一时间的不同 F1 服务器可能应用着不同的模式，如图 6 所示。

5.1.2 存储引擎与行表示

前面提到，F1 服务器需要的状态都放置在下层的键值存储系统中，Google 实际上选择 Spanner 作为这个存储系统的实现，这是一个 Google 在更早的时期提出的可扩展、多版本、全局同步复制的数据库，能提供外部一致的读写操作 [2]。Spanner 的外部一致性保证主要是源于 TrueTime API，直接暴露分布式系统中时钟的不确定性。得益于良好的抽象设计，在 F1 的视角中，Spanner 就是一个提供外部一致性的键值存储引擎，无需考虑额外的实现细节。F1 对关系型数据到键值数据的映射方式如图 7 所示，这一过程在实现中又被称为编码。

Example

first_name*	last_name*	age	phone_number
John	Doe	24	555-123-4567
Jane	Doe	35	555-456-7890

(a) Relational representation.

key	value
Example.John.Doe.*exists*	
Example.John.Doe.*age*	24
Example.John.Doe.*phone_number*	555-123-4567
Example.Jane.Doe.*exists*	
Example.Jane.Doe.*age*	35
Example.Jane.Doe.*phone_number*	555-456-7890

(b) Key–value representation.

图 7: F1 行表示方式

5.1.3 并发控制

F1 使用一种基于时间戳的乐观并发控制机制，类似于本文第 3 节介绍的 Percolator（同样构建在分布式键值存储之上）。在 F1 的模式中，每个表中的每一列都与一个乐观锁关联，这个乐观锁控制着不同事务对该行数据的并发访问。

当客户端在一次事务中读取列值时，会从覆盖这些列的锁中获取最后修改的时间戳；在后续提交时，会将这些时间戳交给服务器进行验证，以确保它们没有更改。如果验证成功，与这些列相关联的所有锁的时间戳将被更新为当前时间戳。这一策略保证了事务的原子性。

由于用户可以向表添加新锁并将其与该表中的任意列关联，因此 F1 必须考虑锁机制对模式变更的影响。

5.2 异步模式变更的难点与对策

5.2.1 异步模式变更的过程

如 5.1.1 所述，F1 实例中的所有服务器共享一组位于键值存储中的键值对。为了将这些键值对解码为行，每个 F1 服务器需要在其内存中维护一个特定版本的模式实例，并使用该模式将关系操作符转换为键值存储所支持的操作。换言之，模式决定了 F1 服务器如何将下层的键值数据解释为业务端需要的关系型数据。

模式本身也会被编码为特殊的键值对，对所有的 F1 服务器均可见。当使用新版本的模式替换旧版本的模式时，模式变更就启动了，F1 开始将新模式传播到实例中的所有服务器。由于其大规模的分布式特性，在这些服务器之间进行同步是无法实现的，不同的服务器可能在不同的时间转换到新模式。只有当实例中的所有 F1 服务器都加载了新模式时，模式更改才算完成。

5.2.2 异步模式变更的难点

由于所有 F1 服务器共享底层存储，因此，不正确地执行异步模式更改可能会破坏数据库。例如，如果新版本的模式中为表 T_i 添加了一个新的索引 $I_{i,k}$，仍然在旧模式上运行的服务器在对 T_i 中的数据进行写操作时，无法同时对 $I_{i,k}$ 进行维护。最终，任何执行索引 $I_{i,k}$ 读取的查询都将返回错误的结果。

这种错误的根本原因是对模式的变更过于突然。旧模式上的服务器不知道索引，而新模式上的服务器在所有操作中使用它，两种行为的差异破坏了数据的一致性。尽管这里以添加索引为例，但是系统中所有基本的模式更改操作都会出现类似的问题，这正是异步模式变更的难点

所在。

5.2.3 异步模式变更的安全执行方案

为了解决上述问题，Google 的工程师设计了一个通过中间状态来执行模式更改的协议。在该协议中，模式的元素（表、列、索引等）可以被标记为若干种中间状态，从而限制对它们的操作。这种方式将单个危险的模式更改分解为一系列安全的模式更改，最终确保数据的一致性。

具体而言，F1 为模式的元素规定了两种非中间状态：1）公有（*public*），此时表、列或索引存在，且可以接受任意的读写操作；2）不存在（*absent*），此时表、列或索引已被移除，不接受任何操作。同时，为了在模式变更期间限制对元素的某一特定操作，F1 又为模式的元素规定了两种中间状态：1）只删（*delete-only*），此时表、列或索引关联的键值对只能被删除，不允许插入新的键值对，也不能被读取；2）只写（*write-only*），此时表、列或索引关联的键值对可以接受任何写操作，但也不能被读取。

这里我们以添加一个必需（required）元素为例，介绍这四种状态在模式变更过程中的应用。元素的初状态为 *absent*，末状态为 *public*，通过 5.2.2 的分析可知，直接进行这两种状态的转换会破坏数据的一致性，需要借助中间状态来安全过渡。四种状态的转换次序如下：

$$absent \rightarrow delete\ only \rightarrow write\ only \rightarrow public$$

文献 [8] 对上述任意两个相邻状态的转换安全性进行了形式化的推导，证明了：规定 F1 实例中的服务器最多只能有两个不同状态，这一转换次序能保证模式变更前后数据的一致性。

此外，针对宕机、网络连接丢失的情况，F1 设计了模式租约（schema lease）机制。每个 F1 服务器需要在规定的租期（例如几分钟）结束后从底层存储中重新读取模式来更新租期，这一行为称为续签。续签失败的服务器将自动终止，稍后在其他正常运行的节点上重新启动。

5.3 F1 在 TiDB 中的实现

TiDB 与 F1 有很多相似之处：应用于大规模的分布式、关系型数据模式、无状态、依赖于底层的共享键值存储（正如前文所述，TiDB 构建于 TiKV 之上）……于是在 TiDB 中引入 F1 的机制是一件很自然的事，只不过 TiDB 设计了自己的抽象层次，如图 8 所示。

各组件的职责分别为：

1. *load schema*：在租约过期时加载最新的模式，若加载失败将关闭相应的 TiDB 服务器。
2. *start job*：在接收到 DDL（即模式变更）请求后，创建一个作业并分配一个唯一的 ID，将其存储在 TiKV 中。
3. *worker*：定期检查是否有模式变更的作业等待处理，如果存在未处理的作业且是本节点创建的，则将其取出并异步执行处理。

以删除一个列为例，我们能在 TiDB 的源码中看到 5.2.3 所述各个状态的过渡处理，如图 9 所示。

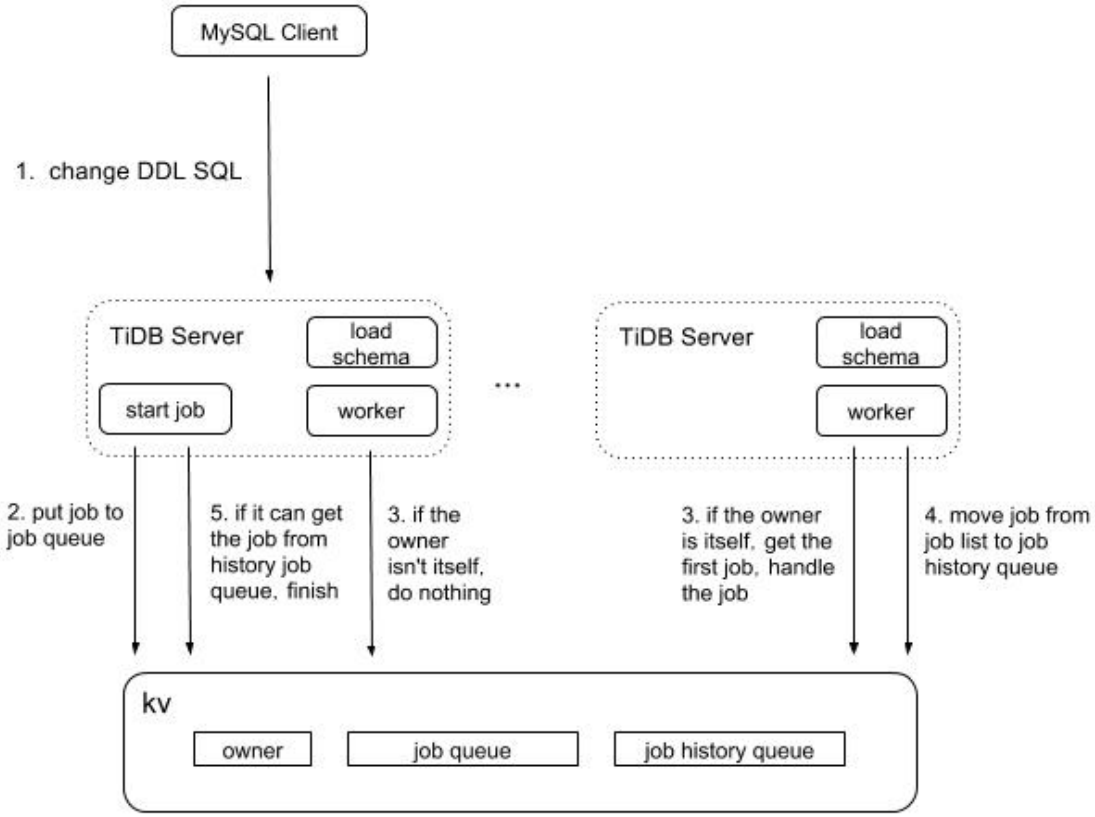

图 8: TiDB 的模式变更处理流程

```go
func onDropColumn(t *meta.Meta, job *model.Job) (ver int64, _ error) {
    tblInfo, colInfo, err := checkDropColumn(t, job)
    if err ≠ nil : ver, errors.Trace(err) ⤴

    originalState := colInfo.State
    /.../
    switch colInfo.State {
    case model.StatePublic:
        /.../
        adjustColumnInfoInDropColumn(tblInfo, colInfo.Offset)
        job.SchemaState = model.StateWriteOnly
        colInfo.State = model.StateWriteOnly
        ver, err = updateVersionAndTableInfoWithCheck(t, job, tblInfo, originalState ≠ colInfo.State)
    case model.StateWriteOnly:...
    case model.StateDeleteOnly:...
    case model.StateDeleteReorganization:
        /.../
        tblInfo.Columns = tblInfo.Columns[:len(tblInfo.Columns)-1]
        colInfo.State = model.StateNone
        ver, err = updateVersionAndTableInfo(t, job, tblInfo, originalState ≠ colInfo.State)
        if err ≠ nil : ver, errors.Trace(err) ⤴

        // Finish this job.
        if job.IsRollingback() {...} else {
            job.FinishTableJob(model.JobStateDone, model.StateNone, ver, tblInfo)
        }
    default:...
    }
    return ver, errors.Trace(err)
}
```

图 9: TiDB 的模式变更源码片段

6 结语

本报告围绕分布式数据库技术这一主题，按照自下向上的系统架构顺序，针对共识、事务、SQL 优化、模式变更四类难点探究了相应的解决方案，也介绍了实际工程中相关的一些实现细节。通过此次报告，我们对分布式数据库（这一在计算机科学领域极具综合性、挑战性的方向）的理论基础和前沿工程实践有了更系统性的认识。

参考文献

[1] Fay Chang et al. “Bigtable: A Distributed Storage System for Structured Data”. In: *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2006, pp. 205–218.

[2] James C. Corbett et al. “Spanner: Google’s Globally Distributed Database”. In: *ACM Trans. Comput. Syst.* (Aug. 2013). DOI: 10.1145/2491245. URL: https://doi.org/10.1145/2491245.

[3] Goetz Graefe. “The cascades framework for query optimization”. In: *IEEE Data Eng. Bull.* 18.3 (1995), pp. 19–29.

[4] Goetz Graefe and William J McKenna. “The volcano optimizer generator: Extensibility and efficient search”. In: *Proceedings of IEEE 9th international conference on data engineering*. IEEE. 1993, pp. 209–218.

[5] Diego Ongaro. “Consensus: Bridging Theory and Practice”. PhD thesis. Stanford, CA, USA, 2014. ISBN: 9798662514218.

[6] Diego Ongaro and John Ousterhout. “In search of an understandable consensus algorithm”. In: *2014 USENIX Annual Technical Conference (Usenix ATC 14)*. 2014, pp. 305–319.

[7] Daniel Peng and Frank Dabek. “Large-scale Incremental Processing Using Distributed Transactions and Notifications”. In: *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. Vancouver, BC: USENIX Association, Oct. 2010. URL: https://www.usenix.org/conference/osdi10/large-scale-incremental-processing-using-distributed-transactions-and.

[8] Ian Rae et al. “Online, Asynchronous Schema Change in F1”. In: *Proc. VLDB Endow.* 6.11 (Aug. 2013), pp. 1045–1056. ISSN: 2150-8097. DOI: 10.14778/2536222.2536230. URL: https://doi.org/10.14778/2536222.2536230.

[9] Jeff Shute et al. “F1: A distributed SQL database that scales”. In: *Proceedings of the VLDB Endowment* 6 (Aug. 2013), pp. 1068–1079. DOI: 10.14778/2536222.2536232.