三篇文章了解 TiDB 技术内幕 - 说计算

申砾

2017-05-24

关系模型到 Key-Value 模型的映射

在这我们将关系模型简单理解为 Table 和 SQL 语句，那么问题变为如何在 KV 结构上保存 Table 以及如何在 KV 结构上运行 SQL 语句。 假设我们有这样一个表的定义：

```
CREATE TABLE User {
    ID int,
    Name varchar(20),
    Role varchar(20),
    Age int,
    PRIMARY KEY (ID),
    Key idxAge (age)
};
```

SQL 和 KV 结构之间存在巨大的区别，那么如何能够方便高效地进行映射，就成为一个很重要的问题。一个好的映射方案必须有利于对数据操作的需求。那么我们先看一下对数据的操作有哪些需求，分别有哪些特点。

对于一个 Table 来说，需要存储的数据包括三部分：

1. 表的元信息
2. Table 中的 Row
3. 索引数据

表的元信息我们暂时不讨论，会有专门的章节来介绍。 对于 Row，可以选择行存或者列存，这两种各有优缺点。TiDB 面向的首要目标是 OLTP 业务，这类业务需要支持快速地读取、保存、修改、删除一行数据，所以采用行存是比较合适的。

对于 Index，TiDB 不止需要支持 Primary Index，还需要支持 Secondary Index。Index 的作用的辅助查询，提升查询性能，以及保证某些 Constraint。查询的时候有两种模式，一种是点查，比如通过 Primary Key 或者 Unique Key 的等值条件进行查询，如 `select name from user where id=1;` ，这种需要通过索引快速定位到某一行数据；另一种是 Range 查询，如 `select name from user where age > 30 and age < 35;` ，这个时候需要通过 `idxAge` 索引查询 age 在 30 和 35 之间的那些数据。Index 还分为 Unique Index 和 非 Unique Index，这两种都需要支持。

分析完需要存储的数据的特点，我们再看看对这些数据的操作需求，主要考虑 Insert/Update/Delete/Select 这四种语句。

对于 Insert 语句，需要将 Row 写入 KV，并且建立好索引数据。

对于 Update 语句，需要将 Row 更新的同时，更新索引数据（如果有必要）。

对于 Delete 语句，需要在删除 Row 的同时，将索引也删除。

上面三个语句处理起来都很简单。对于 Select 语句，情况会复杂一些。首先我们需要能够简单快速地读取一行数据，所以每个 Row 需要有一个 ID （显示或隐式的 ID）。其次可能会读取连续多行数据，比如 `Select * from user;` 。最后还有通过索引读取数据的需求，对索引的使用可能是点查或者是范围查询。

大致的需求已经分析完了，现在让我们看看手里有什么可以用的：**一个全局有序的分布式 Key-Value 引擎**。全局有序这一点重要，可以帮助我们解决不少问题。比如对于快速获取一行数据，假设我们能够构造出某一个或者某几个 Key，定位到这一行，我们就能利用 TiKV 提供的 Seek 方法快速定位到这一行数据所在位置。再比如对于扫描全表的需求，如果能够映射为一个 Key 的 Range，从 StartKey 扫描到 EndKey，那么就可以简单的通过这种方式获得全表数据。操作 Index 数据也是类似的思路。接下来让我们看看 TiDB 是如何做的。

TiDB 对每个表分配一个 TableID，每一个索引都会分配一个 IndexID，每一行分配一个 RowID（如果表有整数型的 Primary Key，那么会用 Primary Key 的值当做 RowID），其中 TableID 在整个集群内唯一，IndexID/RowID 在表内唯一，这些 ID 都是 int64 类型。

每行数据按照如下规则进行编码成 Key-Value pair：

```
Key: tablePrefix{tableID}_recordPrefixSep{rowID}
Value: [col1, col2, col3, col4]
```

其中 Key 的 `tablePrefix` / `recordPrefixSep` 都是特定的字符串常量，用于在 KV 空间内区分其他数据。

对于 Index 数据，会按照如下规则编码成 Key-Value pair：

```
Key: tablePrefix{tableID}_indexPrefixSep{indexID}_indexedColumnsValue
Value: rowID
```

Index 数据还需要考虑 Unique Index 和非 Unique Index 两种情况，对于 Unique Index，可以按照上述编码规则。但是对于非 Unique Index，通过这种编码并不能构造出唯一的 Key，因为同一个 Index 的 `tablePrefix{tableID}_indexPrefixSep{indexID}` 都一样，可能有多行数据的 `ColumnsValue` 是一样的，所以对于非 Unique Index 的编码做了一点调整：

```
Key: tablePrefix{tableID}_indexPrefixSep{indexID}_indexedColumnsValue_rowID
Value: null
```

这样能够对索引中的每行数据构造出唯一的 Key。 注意上述编码规则中的 Key 里面的各种 xxPrefix 都是字符串常量，作用都是区分命名空间，以免不同类型的数据之间相互冲突，定义如下：

```
var(
    tablePrefix     = []byte{'t'}
    recordPrefixSep = []byte("_r")
    indexPrefixSep  = []byte("_i")
)
```

另外请大家注意，上述方案中，无论是 Row 还是 Index 的 Key 编码方案，一个 Table 内部所有的 Row 都有相同的前缀，一个 Index 的数据也都有相同的前缀。这样具体相同的前缀的数据，在 TiKV

的 Key 空间内，是排列在一起。同时只要我们小心地设计后缀部分的编码方案，保证编码前和编码后的比较关系不变，那么就可以将 Row 或者 Index 数据有序地保存在 TiKV 中。这种 保证编码前和编码后的比较关系不变 的方案我们称为 Memcomparable，对于任何类型的值，两个对象编码前的原始类型比较结果，和编码成 byte 数组后（注意，TiKV 中的 Key 和 Value 都是原始的 byte 数组）的比较结果保持一致。具体的编码方案参见 TiDB 的 codec 包 。采用这种编码后，一个表的所有 Row 数据就会按照 RowID 的顺序排列在 TiKV 的 Key 空间中，某一个 Index 的数据也会按照 Index 的 ColumnValue 顺序排列在 Key 空间内。

现在我们结合开始提到的需求以及 TiDB 的映射方案来看一下，这个方案是否能满足需求。首先我们通过这个映射方案，将 Row 和 Index 数据都转换为 Key-Value 数据，且每一行、每一条索引数据都是有唯一的 Key。其次，这种映射方案对于点查、范围查询都很友好，我们可以很容易地构造出某行、某条索引所对应的 Key，或者是某一块相邻的行、相邻的索引值所对应的 Key 范围。最后，在保证表中的一些 Constraint 的时候，可以通过构造并检查某个 Key 是否存在来判断是否能够满足相应的 Constraint。

至此我们已经聊完了如何将 Table 映射到 KV 上面，这里再举个简单的例子，便于大家理解，还是以上面的表结构为例。假设表中有 3 行数据：

```
1, "TiDB", "SQL Layer", 10
2, "TiKV", "KV Engine", 20
3, "PD", "Manager", 30
```

那么首先每行数据都会映射为一个 Key-Value pair，注意这个表有一个 Int 类型的 Primary Key，所以 RowID 的值即为这个 Primary Key 的值。假设这个表的 Table ID 为 10，其 Row 的数据为：

```
t10_r1 --> ["TiDB", "SQL Layer", 10]
t10_r2 --> ["TiKV", "KV Engine", 20]
t10_r3 --> ["PD", "Manager", 30]
```

除了 Primary Key 之外，这个表还有一个 Index，假设这个 Index 的 ID 为 1，则其数据为：

```
t10_i1_10_1 --> null
t10_i1_20_2 --> null
```

```
t10_i1_30_3 --> null
```

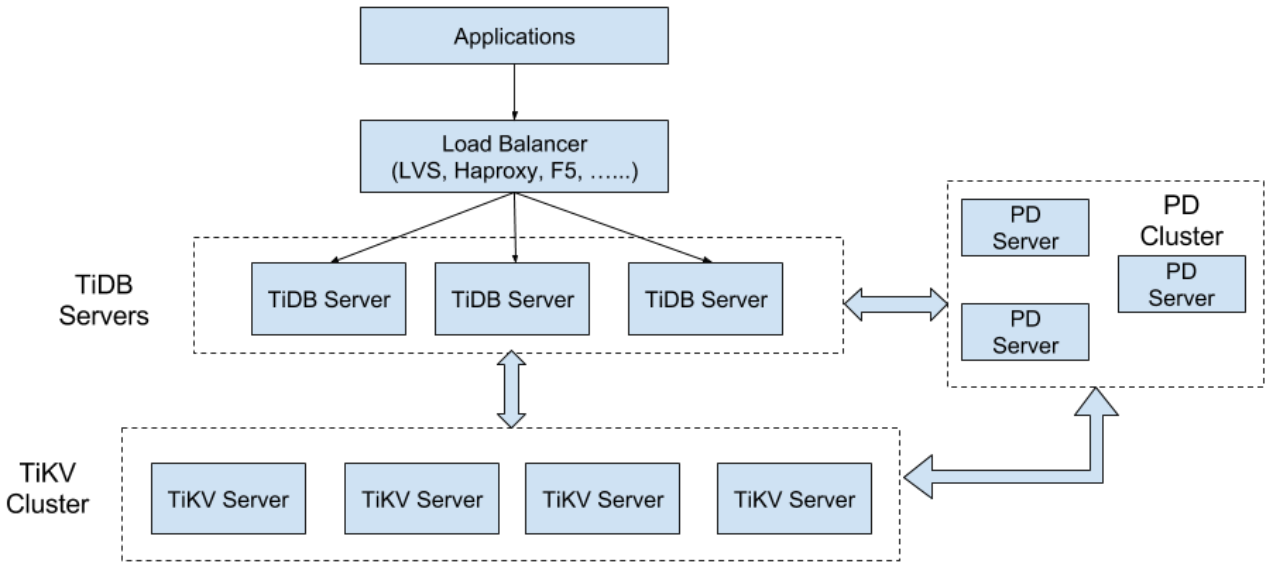
大家可以结合上面的编码规则来理解这个例子，希望大家能理解我们为什么选择了这个映射方案，这样做的目的是什么。

元信息管理

上节介绍了表中的数据和索引是如何映射为 KV，本节介绍一下元信息的存储。Database/Table 都有元信息，也就是其定义以及各项属性，这些信息也需要持久化，我们也将这些信息存储在 TiKV 中。每个 Database/Table 都被分配了一个唯一的 ID，这个 ID 作为唯一标识，并且在编码为 Key-Value 时，这个 ID 都会编码到 Key 中，再加上 `m_` 前缀。这样可以构造出一个 Key，Value 中存储的是序列化后的元信息。 除此之外，还有一个专门的 Key-Value 存储当前 Schema 信息的版本。TiDB 使用 Google F1 的 Online Schema 变更算法，有一个后台线程在不断的检查 TiKV 上面存储的 Schema 版本是否发生变化，并且保证在一定时间内一定能够获取版本的变化（如果确实发生了变化）。这部分的具体实现参见 TiDB 的异步 schema 变更实现 一文。

SQL on KV 架构

TiDB 的整体架构如下图所示



TiKV Cluster 主要作用是作为 KV 引擎存储数据，上篇文章已经介绍过了细节，这里不再敷述。本篇文章主要介绍 SQL 层，也就是 TiDB Servers 这一层，这一层的节点都是无状态的节点，本身并不存储数据，节点之间完全对等。TiDB Server 这一层最重要的工作是处理用户请求，执行 SQL 运算逻辑，接下来我们做一些简单的介绍。

SQL 运算

理解了 SQL 到 KV 的映射方案之后，我们可以理解关系数据是如何保存的，接下来我们要理解如何使用这些数据来满足用户的查询需求，也就是一个查询语句是如何操作底层存储的数据。 能想到的最简单的方案就是通过上一节所述的映射方案，将 SQL 查询映射为对 KV 的查询，再通过 KV 接口获取对应的数据，最后执行各种计算。 比如 `Select count(*) from user where name="TiDB";` 这样一个语句，我们需要读取表中所有的数据，然后检查 `Name` 字段是否是 `TiDB` ，如果是的话，则返回这一行。这样一个操作流程转换为 KV 操作流程：

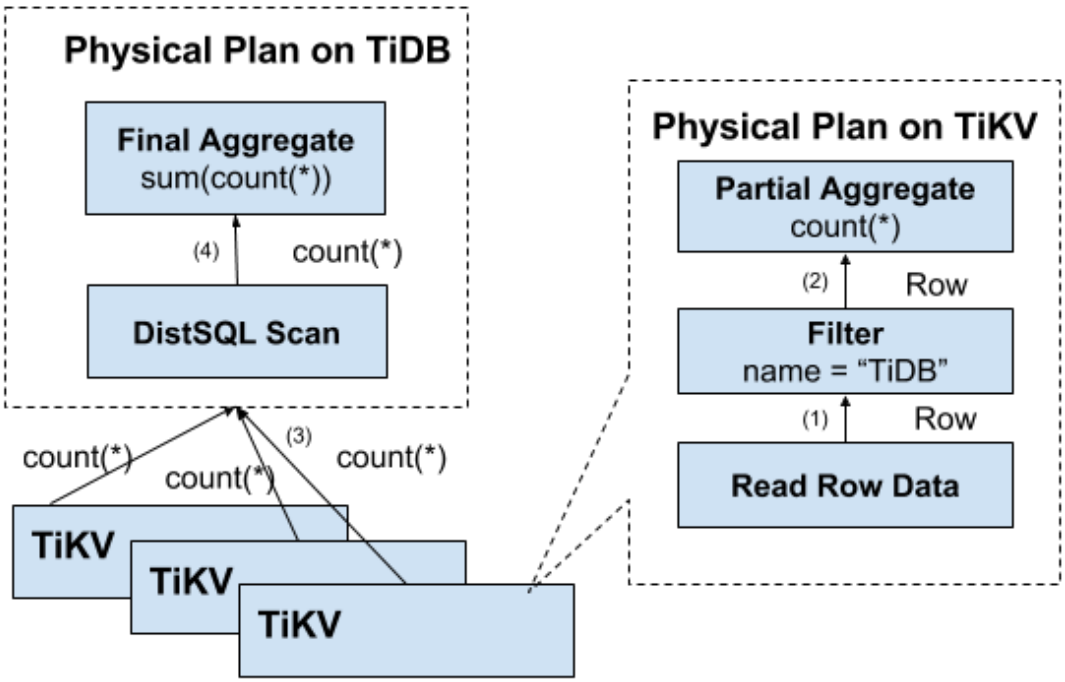
- 构造出 Key Range：一个表中所有的 RowID 都在 `[0, MaxInt64)` 这个范围内，那么我们用 0 和 MaxInt64 根据 Row 的 Key 编码规则，就能构造出一个 `[StartKey, EndKey)` 的左闭右开区间
- 扫描 Key Range：根据上面构造出的 Key Range，读取 TiKV 中的数据
- 过滤数据：对于读到的每一行数据，计算 `name="TiDB"` 这个表达式，如果为真，则向上返回这一行，否则丢弃这一行数据
- 计算 Count：对符合要求的每一行，累计到 Count 值上面

这个方案肯定是可以 Work 的，但是并不能 Work 的很好，原因是显而易见的：

1. 在扫描数据的时候，每一行都要通过 KV 操作同 TiKV 中读取出来，至少有一次 RPC 开销，如果需要扫描的数据很多，那么这个开销会非常大
2. 并不是所有的行都有用，如果不满足条件，其实可以不读取出来
3. 符合要求的行的值并没有什么意义，实际上这里只需要有几行数据这个信息就行

分布式 SQL 运算

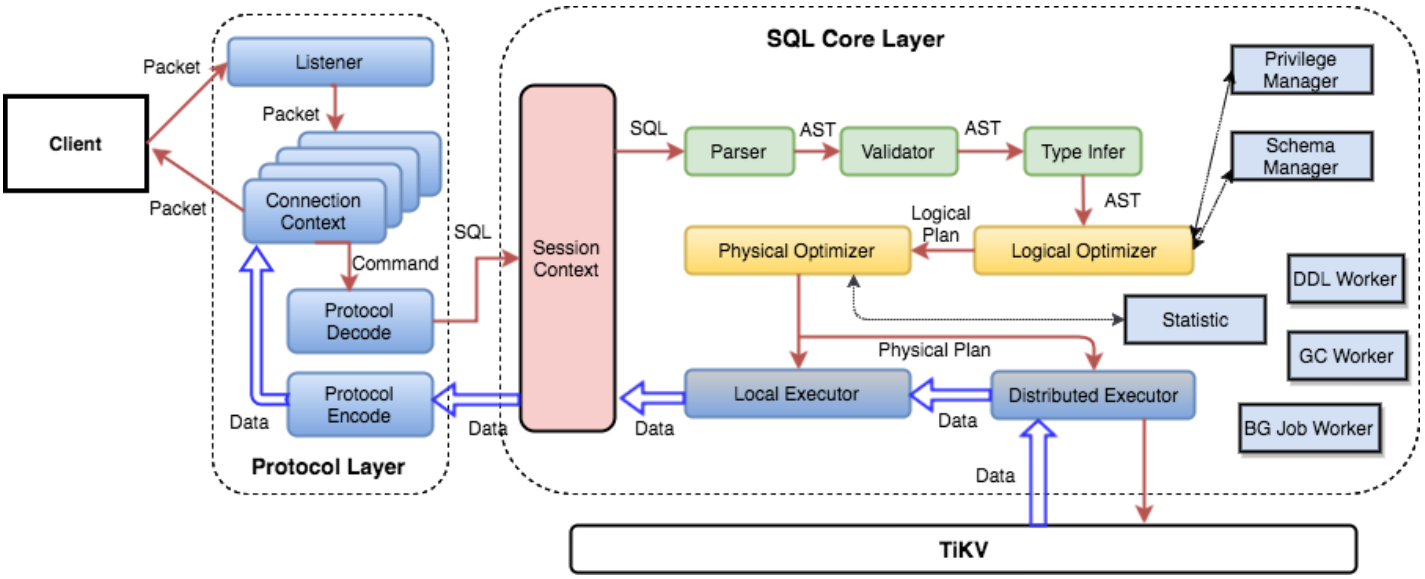
如何避免上述缺陷也是显而易见的，首先我们需要将计算尽量靠近存储节点，以避免大量的 RPC 调用。其次，我们需要将 Filter 也下推到存储节点进行计算，这样只需要返回有效的行，避免无意义的网络传输。最后，我们可以将聚合函数、GroupBy 也下推到存储节点，进行预聚合，每个节点只需要返回一个 Count 值即可，再由 tidb-server 将 Count 值 Sum 起来。 这里有一个数据逐层返回的示意图：



MPP and SMP in TiDB 这篇文章详细描述了 TiDB 是如何让 SQL 语句跑的更快，大家可以参考一下。

SQL 层架构

上面几节简要介绍了 SQL 层的一些功能，希望大家对 SQL 语句的处理有一个基本的了解。实际上 TiDB 的 SQL 层要复杂的多，模块以及层次非常多，下面这个图列出了重要的模块以及调用关系：



用户的 SQL 请求会直接或者通过 Load Balancer 发送到 tidb-server，tidb-server 会解析 MySQL Protocol Packet，获取请求内容，然后做语法解析、查询计划制定和优化、执行查询计划获取和处理数据。数据全部存储在 TiKV 集群中，所以在这个过程中 tidb-server 需要和 tikv-server 交互，获取数据。最后 tidb-server 需要将查询结果返回给用户。

小结

到这里，我们已经从 SQL 的角度了解了数据是如何存储，如何用于计算。SQL 层更详细的介绍会在今后的文章中给出，比如优化器的工作原理，分布式执行框架的细节。 下一篇文章我们将会介绍一些关于 PD 的信息，这部分会比较有意思，里面的很多东西是在使用 TiDB 过程中看不到，但是对整体集群又非常重要。主要会涉及到集群的管理和调度。

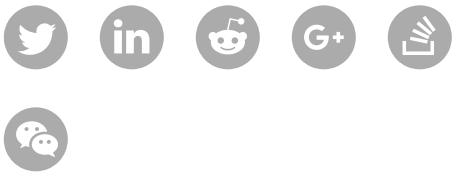
> 相关阅读： 三篇文章了解 *TiDB* 技术内幕 - 说存储； 三篇文章了解 *TiDB* 技术内幕 - 谈调度

商务咨询
4006790886
010-58400041
info@pingcap.com

前台总机
010-53326356

媒体合作
pr@pingcap.com

PingCAP 是业界领先的企业级开源分布式数据库企业，提供包括开源分布式数据库产品、解决方案与咨询、技术支持与培训认证服务，致力于为全球行业用户提供稳定高效、安全可靠、开放兼容的新型数据基础设施，解放企业生产力，加速企业数字化转型升级。



联系我们