TiDB 的正确使用姿势

> 最近这几个月，特别是 TiDB RC1 发布后，越来越多的用户已经开始测试起来，也有很多朋友已经在生产环境中使用，我们这边也陆续的收到了很多用户的测试和使用反馈。非常感谢各位小伙伴和早期用户的厚爱，而且看了这么多场景后，也总结出了一些 TiDB 的使用实践 (其实 Spanner 的最佳实践大部分在 TiDB 中也是适用的，MySQL 最佳实践也是），也是借着 Google Cloud Spanner 发布的东风，看了一下 Spanner 官方的一些最佳实践文档，写篇文章讲讲 TiDB 以及分布式关系型数据库的一些正确的使用姿势，当然，时代也在一直发展，TiDB 也在不停的进化，这篇文章基本上只代表近期的一些观察。

首先谈谈 Schema 设计的一些比较好的经验。由于 TiDB 是一个分布式的数据库，可能在表结构设计的时候需要考虑的事情和传统的单机数据库不太一样，需要开发者能够带着「这个表的数据会分散在不同的机器上」这个前提，才能做更好的设计。

和 Spanner 一样，TiDB 中的一张表的行（Rows）是按照主键的字节序排序的（整数类型的主键我们会使用特定的编码使其字节序和按大小排序一致），即使在 CREATE TABLE 语句中不显式的创建主键，TiDB 也会分配一个隐式的。
有四点需要记住：

1. 按照字节序的顺序扫描的效率是比较高的；
2. 连续的行大概率会存储在同一台机器的邻近位置，每次批量的读取和写入的效率会高；
3. 索引是有序的（主键也是一种索引），一行的每一列的索引都会占用一个 KV Pair，比如，某个表除了主键有 3 个索引，那么在这个表中插入一行，对应在底层存储就是 4 个 KV Pairs 的写入：数据行以及 3 个索引行。
4. 一行的数据都是存在一个 KV Pair 中，不会被切分，这点和类 BigTable 的列式存储很不一样。

表的数据在 TiDB 内部会被底层存储 TiKV 切分成很多 64M 的 Region（对应 Spanner 的 Splits 的概念），每个 Region 里面存储的都是连续的行，Region 是 TiDB 进行数据调度的单位，随着一个 Region 的数据量越来越大和时间的推移，Region 会分裂/合并，或者移动到集群中不同的物理机上，使得整个集群能够水平扩展。

- 建议：

  1. 尽可能批量写入，但是一次写入总大小不要超过 Region 的分裂阈值（64M），另外 TiDB 也对单个事务有大小的限制。

  2. 存储超宽表是比较不合适的，特别是一行的列非常多，同时不是太稀疏，一个经验是最好单行的总数据大小不要超过 64K，越小越好。大的数据最好拆到多张表中。

  3. 对于高并发且访问频繁的数据，尽可能一次访问只命中一个 Region，这个也很好理解，比如一个模糊查询或者一个没有索引的表扫描操作，可能会发生在多个物理节点上，一来会有更大的网络开销，二来访问的 Region 越多，遇到 stale region 然后重试的概率也越大（可以理解为 TiDB 会经常做 Region 的移动，客户端的路由信息可能更新不那么及时），这些可能会影响 .99 延迟；另一方面，小事务（在一个 Region 的范围内）的写入的延迟会更低，TiDB 针对同一个 Region 内的跨行事务是有优化的。另外 TiDB 对通过主键精准的点查询（结果集只有一条）效率更高。

关于索引

除了使用主键查询外，TiDB 允许用户创建二级索引以加速访问，就像上面提到过的，在 TiKV 的层面，TiDB 这边的表里面的行数据和索引的数据看起来都是 TiKV 中的 KV Pair，所以很多适用于表数据的原则也适用于索引。和 Spanner 有点不一样的是，TiDB 只支持全局索引，也就是 Spanner 中默认的 Non-interleaved indexes。全局索引的好处是对使用者没有限制，可以 scale 到任意大小，不过这意味着，索引信息不一定和实际的数据在一个 Region 内。

- 建议：
  对于大海捞针式的查询来说 (海量数据中精准定位某条或者某几条)，务

必通过索引。

当然也不要盲目的创建索引，创建太多索引会影响写入的性能。

反模式(最好别这么干！)

其实 Spanner 的白皮书已经写得很清楚了，我再赘述一下：

第一种，过度依赖单调递增的主键，AUTO INCREMENT ID
在传统的关系型数据库中，开发者经常会依赖自增 ID 来作为 PRIMARY KEY，但是其实大多数场景大家想要的只是一个不重复的 ID 而已，至于是不是自增其实无所谓，但是这个对于分布式数据库来说是不推荐的，随着插入的压力增大，会在这张表的尾部 Region 形成热点，而且这个热点并没有办法分散到多台机器。TiDB 在 GA 的版本中会对非自增 ID 主键进行优化，让 insert workload 尽可能分散。

- 建议：
  如果业务没有必要使用单调递增 ID 作为主键，就别用，使用真正有意义的列作为主键（一般来说，例如：邮箱、用户名等）
  使用随机的 UUID 或者对单调递增的 ID 进行 bit-reverse （位反转）

第二种，单调递增的索引(比如时间戳)
很多日志类型的业务，因为经常需要按照时间的维度查询，所以很自然需要对 timestamp 创建索引，但是这类索引的问题本质上和单调递增主键是一样的，因为在 TiDB 的内部实现里，索引也是一堆连续的 KV Pairs，不断的插入单调递增的时间戳会造成索引尾部的 Region 形成热点，导致写入的吞吐受到影响。

- 建议：
  因为不可避免的，很多用户在使用 TiDB 存储日志，毕竟 TiDB 的弹性伸缩能力和 MySQL 兼容的查询特性是很适合这类业务的。另一方面，如果发现写入的压力实在扛不住，但是又非常想用 TiDB 来存储这种类型的数据，可以像 Spanner 建议的那样做 Application 层面的 Sharding，以存储日志为例，原来的可能在 TiDB 上创建一个 log 表，更好的模式是可以创建多个 log 表，如：log_1, log_2 ⋯ log_N，然后业务层插入的时候根据时间戳进行 hash ，随机分配到 1..N 这几个分片表

中的一个。

相应的，查询的时候需要将查询请求分发到各个分片上，最后在业务层汇总结果。

查询优化

TiDB 的优化分为基于规则的优化（Rule Based Optimization）和基于代价的优化（Cost Based Optimization）, 本质上 TiDB 的 SQL 引擎更像是一个分布式计算框架，对于大表的数据因为本身 TiDB 会将数据分散到多个存储节点上，能将查询逻辑下推，会大大的提升查询的效率。

TiDB 基于规则的优化有：
谓词下推

谓词下推会将 **where/on**/having 条件推到离数据表尽可能近的地方，比如：

```
select * from t join s on t.id = s.id where t.c1 < 10
```

可以被 TiDB 自动改写成

```
select * from (select * from t where t.c1 < 10) as t join s on
t.id = s.id
```

关联子查询消除

关联子查询可能被 TiDB 改写成 Join，例如：

```
select * from t where t.id in (select id from s where s.c1 < 10
and s.name = t.name)
```

可以被改写成：

```
select * from t semi join s on t.id = s.id and s.name = t.name
and s.c1 < 10
```

聚合下推
聚合函数可以被推过 Join，所以类似带等值连接的 Join 的效率会比较高，例

如：

```
select count(s.id) from t join s on t.id = s.t_id
```

可以被改写成：

```
select sum(agg0) from t join (select count(id) as agg0, t_id from
s group by t_id) as s on t.id = s.t_id
```

基于规则的优化有时可以组合以产生意想不到的效果，例如：

```
select s.c2 from s where 0 = (select count(id) from t where
t.s_id = s.id)
```

在TiDB中，这个语句会先通过关联子查询消除的优化，变成：

```
select s.c2 from s left outer join t on t.s_id = s.id group by
s.id where 0 = count(t.id)
```

然后这个语句会通过聚合下推的优化，变成：

```
select s.c2 from s left outer join (select count(t.id) as agg0
from t group by t.s_id) t on t.s_id = s.id group by s.id where 0
= sum(agg0)
```

再经过聚合消除的判断，语句可以优化成：

```
select s.c2 from s left outer join (select count(t.id) as agg0
from t group by t.s_id) t on t.s_id = s.id where 0 = agg0
```

基于代价的优化有：

读取表时，如果有多条索引可以选择，我们可以通过统计信息选择最优的索引。例如:

```
select * from t where age = 30 and name in ( '小明', '小强')
```

对于包含 Join 的操作，我们可以区分大小表，TiDB 的对于一个大表和一个小表的 Join 会有特殊的优化。
例如

```
select * from t join s on s.id = t.id
```

优化器会通过对表大小的估计来选择 Join 的算法：即选择把较小的表装入内存中。
对于多种方案，利用动态规划算法选择最优者，例如:

```
(select * from t where c1 < 10) union all (select * from s where
c2 < 10) order by c3 limit 10
```

t 和 s 可以根据索引的数据分布来确定选择索引 c3 还是 c2。

总之正确使用 TiDB 的姿势，或者说 TiDB 的典型的应用场景是:

大数据量下，MySQL 复杂查询很慢;

大数据量下，数据增长很快，接近单机处理的极限，不想分库分表或者使用数据库中间件等对业务侵入性较大，架构反过来约束业务的 Sharding 方案;

大数据量下，有高并发实时写入、实时查询、实时统计分析的需求;

有分布式事务、多数据中心的数据 100% 强一致性、auto-failover 的高可用的需求。

如果整篇文章你只想记住一句话，那就是数据条数少于 5000w 的场景下通常用不到 TiDB，TiDB 是为大规模的数据场景设计的。如果还想记住一句话，那就是单机 MySQL 能满足的场景也用不到 TiDB。