零声教育出品 Mark 老师 QQ： 2548898954

RocksDB 特性

列族（column family）

> RocksDB的每个键值对都与唯一一个列族（column family）结合。如果没有指定 Column Family，键值对将会结合到 “default” 列族。
>
> 列族提供了一种从逻辑上给数据库分片的方法。他的一些有趣的特性包括：
>
> - 支持跨列族原子写。意味着你可以原子执行 `Write({cf1,key1,value1}, {cf2,key2,value2})`。
> - 跨列族的一致性视图。
> - 允许对不同的列族进行不同的配置
> - 即时添加 / 删除列族。两个操作都是非常快的。
>
> **实现：**
>
> 列族的主要实现思想是他们共享一个 WAL 日志，但是不共享 memtable 和 table 文件。通过共享 WAL 文件，我们实现了酷酷的原子写。通过隔离 memtable 和 table 文件，我们可以独立配置每个列族并且快速删除它们。
>
> 每当一个单独的列族刷盘，我们创建一个新的 WAL 文件。所有列族的所有新的写入都会去到新的 WAL文件。但是，我们还不能删除旧的 WAL，因为他还有一些对其他列族有用的数据。我们只能在所有的列族都把这个WAL 里的数据刷盘了，才能删除这个 WAL 文件。这带来了一些有趣的实现细节以及一些有趣的调优需求。确保你的所有列族都会有规律地刷盘。另外，看一下 `Options::max_total_wal_size`，通过配置他，过期的列族能自动被刷盘。

快照（snapshot）

> 一个快照会捕获在创建的时间点的 DB 的一致性视图。快照在DB重启之后将消失。
>
> 一个快照相当于一个 `SnapshotImpl` 类的小型对象。他只持有部分简单的字段，比如快照生成时候的`number_` 以及 `unix_time_`。
>
> **实现：**
>
> Snapshot 会存储在一个 `DBImpl` 持有的链表里。其中一个好处是，我们可以在获取 DB 互斥锁前分配好这个链表的节点。然后在持有互斥锁的时候，我们只需要更新链表指针。更进一步，`ReleaseSnapshot` 可以对所有的快照以任意顺序被调用。使用链表，我们不需要移动所有节点就可以删除一个节点了。
>
> **问题：**
>
> 快照具有伸缩性的问题；因为使用链表。尽管他是顺序排列的，他还是不可以使用二分查找。在 flush/comapction 的时候，如果我们需要找到一个 key 在最早的哪个 snapshot 中是可见的，我们必须逐个扫描快照链表。当许多快照存在的时候，这个扫描会非常明显的拖慢 flush/compaction 进程。当有成百上千个快照的时候就能观察到这种问题了。

迭代器

> 如果 `ReadOptions.snapshot` 被给出，那么迭代器会从一个快照里面返回数据。如果这是一个 nullptr，迭代器隐式创建一个迭代器创建的时间节点的快照，利用该隐式快照提供数据。隐式快照无法转换为显式快照。

事务

> 当使用 TransactionDB 或者 OptimisticTransactionDB 的时候，RocksDB 将支持事务。事务带有简单的 `BEGIN/COMMIT/ROLLBACK` API，并且允许应用并发地修改数据，具体的冲突检查，由 RocksDB 来处理。RocksDB 支持悲观和乐观的并发控制。
>
> 注意，当通过 `WriteBatch` 写入多个 key 的时候，RocksDB 提供原子化操作。事务提供了一个方法，来保证他们只会在没有冲突的时候被提交。与 `WriteBatch` 类似，只有当一个事务提交，其他线程才能看到被修改的内容（读 committed）。

悲观事务（TransactionDB）

> 当使用 TransactionDB 的时候，所有正在修改的 RocksDB 里的 key 都会被上锁，让 RocksDB 执行冲突检测。如果一个 key 发生锁冲突，操作会返回一个错误。当事务被提交，数据库保证这个事务是可以写入的。
>
> 一个 TransactionDB 在有大量并发工作压力的时候，相比 OptimisticTransactionDB 有更好的表现。然而，由于非常过激的上锁策略，使用 TransactionDB 会有一定的性能损耗。
> TransactionDB 会在所有写操作的时候做冲突检查，包括不使用事务写入的时候。

乐观事务（OptimisticTransactionDB）

> 乐观事务提供轻量级的乐观并发控制，用来给那些多个事务间不会有高的竞争或者干涉的工作场景。
>
> 乐观事务在预备写的时候不使用任何锁。作为替代，他们把这个操作推迟到在提交的时候检查，是否有其他人修改了正在进行的事务。如果和另一个写入有冲突（或者他无法做决定），提交会返回错误，并且没有任何 key 都不会被写入。
>
> 乐观的并发控制在处理那些偶尔出现的写冲突非常有效。然而，对于那些大量事务对同一个 key 写入导致写冲突频繁发生的场景，却不是一个好主意。对于这些场景，使用 TransactionDB 是更好的选择。OptimisticTransactionDB 在大量非事务写入，而少量事务写入的场景，会比 TransactionDB 性能更好。

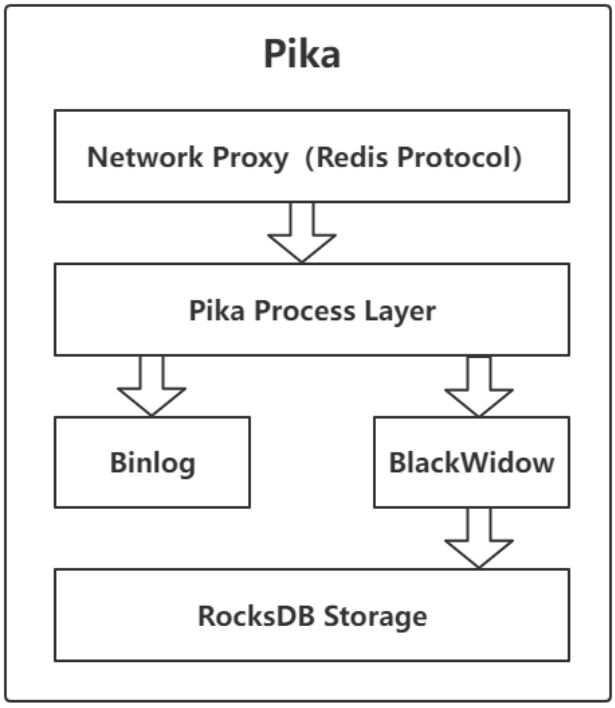
底层实现

> **读快照：**
>
> 每一个 RocksDB 的更新都是通过插入一个带有强制自增的序列号的项来实现的。给即将要被（事务或者非事务）DB 用来创建快照的 read_options。snapshot赋值一个序列号，可以只读到小于这个序列号的内容。
>
> **读写冲突检测：**
>
> 读写冲突可以通过升级为写写冲突来防止：通过 `GetForUpdate`（而不是`Get`）来做读操作。
>
> **写写冲突检测（悲观）：**
>
> 写写冲突在写入的时候用一个锁表来进行检测。非事务更新（put，merge，delete）在内部其实是以一个事务来运行的。
>
> **写写冲突检测（乐观）：**

> 写写冲突会在提交的时候检查其最后一个序列号，来检测冲突。
>
> 冲突检测的逻辑在 `TransactionUtil::CheckKeysForConflicts` 中实现；
>
> - 只检测在内存中出现的 key 的冲突和失败。
> - 冲突检测是通过比对每个 key 的最后的序列号（`DBImpl::GetLatestSequenceForKey`）与用于写入的序列号来实现的。

Pika

> Pika 是一个可持久化的大容量 Redis 存储服务，兼容 string、hash、list、zset、set 的绝大部分接口，解决 Redis 由于存储数据量巨大而导致内存不够用的容量瓶颈，并且可以像 Redis 一样，通过 slaveof 命令进行主从备份，支持全同步和部分同步，Pika 还可以用在 twemproxy 或者 codis 中来实现静态数据分片；



特点

- 容量大，支持百G数据量的存储
- 兼容 Redis ，不用修改代码即可平滑从 Redis 迁移到 Pika
- 支持主从 （slaveof）
- 完善的运维命令

pika 安装编译

CentOS

```
# 安装必要的 lib
sudo yum install gflags-devel snappy-devel glog-devel protobuf-devel
sudo yum install gcc-c++
# 安装可选的 lib
sudo yum install zlib-devel lz4-devel libzstd-devel
# 确保 gcc 在4.8或者以上
# 获取源代码
git clone https://github.com/OpenAtomFoundation/pika.git
cd pika
git submodule update --init
git checkout -b v3.4.0
make
```

Ubuntu

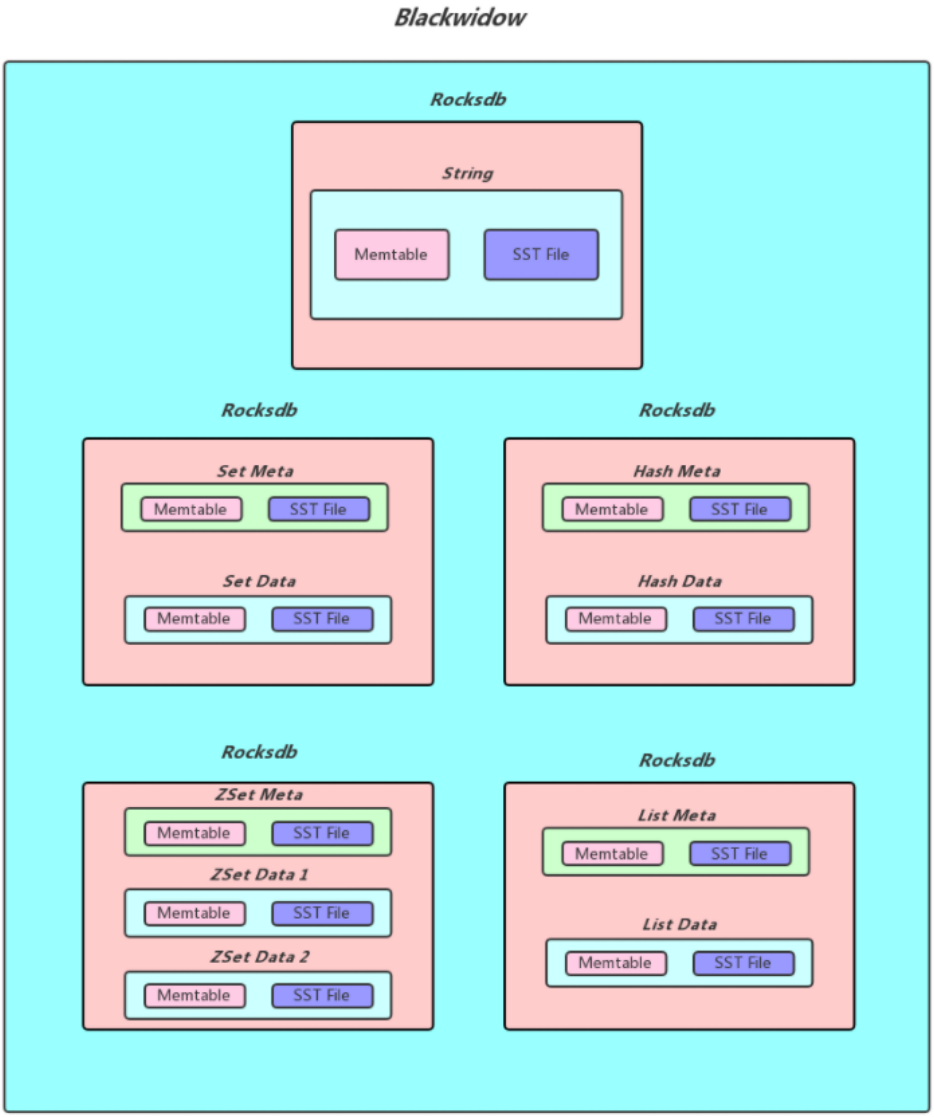
```
# 安装必要的 lib
sudo apt-get install libgflags-dev libsnappy-dev
sudo apt-get install libprotobuf-dev protobuf-compiler
sudo apt install libgoogle-glog-dev
# 确保 gcc 在4。8或者以上
# 获取源代码
git clone https://github.com/OpenAtomFoundation/pika.git
cd pika
git submodule update --init
git checkout -b v3.4.0
make
```

使用

```
./output/bin/pika -c ./conf/pika.conf
```

Blackwidow

> Blackwidow 本质上是基于 RocksDB 的封装，使本身只支持 kv 存储的 RocksDB 能够支持多种数据结构，目前 Blackwidow 支持五种数据结构的存储：String 结构（实际上就是存储 key，value），Hash 结构，List 结构，Set 结构和 ZSet 结构， 因为 RocksDB 的存储方式只有 kv 一种， 所以上述五种数据结构最终都要落盘到 RocksDB 的 kv 存储方式上，下面我们展示 Blackwidow 和 RocksDB 的关系并且说明我们是如何用 kv 来模拟多数据结构的。



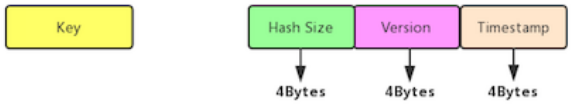
String 结构的存储

> String 本质上就是 Key，Value，我们知道 RocksDB 本身就是支持 kv 存储的，为了实现 Redis 中的 expire 功能，所以在 value 后面添加了 4 Bytes 用于存储 timestamp，作为最后 RocksDB 落盘的 kv 格式，下面是具体的实现方式：
>
> 
>
>
> 如果我们没有对该 String 对象设置超时时间，则 timestamp 存储的值就是默认值 0， 否则就是该对象过期时间的时间戳， 每次我们获取一个 String 对象的时候， 首先会解析 Value 部分的后四字节， 获取到timestamp 做出判断之后再返回结果。

Hash 结构的存储

> Blackwidow 中的 Hash 表由两部分构成，元数据 （meta_key，meta_value），和普通数据（data_key，data_value），元数据中存储的主要是 Hash 表的一些信息， 比如说当前 Hash 表的域的数量以及当前 hash 表的版本号和过期时间（用做秒删功能），而普通数据主要就是指的同一个 Hash 表中一一对应的 field 和value，作为具体最后 RocksDB 落盘的 kv 格式，下面是具体的实现方式：

1. 每个 Hash 表的 meta_key 和 meta_value 的落盘方式：



meta_key 实际上就是 hash 表的 key，而 meta_value 由三个部分构成：4Bytes 的 Hash size （用于存储当前hash表的大小） + 4Bytes 的 Version （用于秒删功能） + 4Bytes 的 Timestamp （用于记录我们给这个Hash 表设置的超时时间的时间戳， 默认为0）

2. Hash 表中 data_key 和 data_value 的落盘方式：

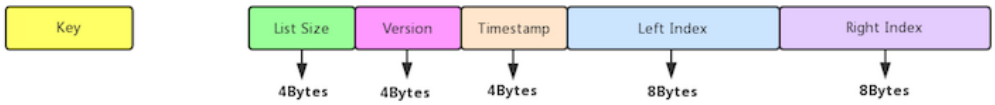
data_key由四个部分构成：4Bytes 的 Key size （用于记录后面追加的key的长度，便与解析） + key的内容 + 4Bytes 的 Version + Field 的内容， 而 data_value 就是 Hash 表某个 field 对应的 value。

如果我们需要查找一个 Hash 表中的某一个 field 对应的 value，我们首先会获取到 meta_value 解析出其中的 timestamp 判断这个 hash 表是否过期， 如果没有过期， 我们可以拿到其中的 version，然后我们使用key，version，和 field 拼出 data_key，进而找到对应的 data_value（如果存在的话）。

List 结构的存储

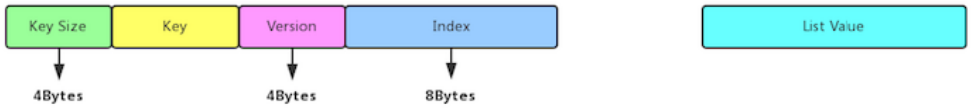
Blackwidow 中的 List 由两部分构成，元数据（meta_key，meta_value），和普通数据（data_key，data_value），元数据中存储的主要是 List 链表的一些信息， 比如说当前 List 链表结点的的数量以及当前 List 链表的版本号和过期时间（用做秒删功能），还有当前 List 链表的左右边界（由于 nemo 实现的链表结构被吐槽 lrange 效率低下，所以 Blackwidow 底层用数组来模拟链表，这样 lrange 速度会大大提升，因为结点存储都是有序的），普通数据实际上就是指的list中每一个结点中的数据，作为具体最后 RocksDB 落盘的 kv 格式，下面是具体的实现方式：

1. 每个 List 链表的 meta_key 和 meta_value 的落盘方式：



meta_key实际上就是 List 链表的 key，而 meta_value 由五个部分构成： 8Bytes 的 List size（用于存储当前链表中总共有多少个结点） + 4Bytes 的 Version （用于秒删功能） + 4Bytes 的 Timestamp （用于记录我们给这个List链表设置的超时时间的时间戳， 默认为0） + 8Bytes 的 Left Index（数组的左边界） + 8Bytes 的Right Index （数组的右边界）。

2. List 链表中 data_key 和 data_value 的落盘方式：

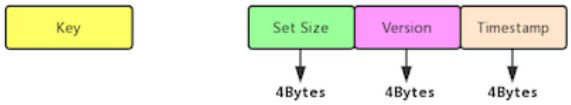


data_key由四个部分构成：4Bytes 的 Key size （用于记录后面追加的 key 的长度，便与解析） + key 的内容 + 4Bytes 的 Version + 8Bytes 的 Index （这个记录的就是当前结点的在这个 List 链表中的索引）， 而data_value 就是 List 链表该 node 中存储的值。

Set 结构的存储

Blackwidow 中的 Set 由两部分构成，元数据 （meta_key，meta_value），和普通数据（data_key，data_value），元数据中存储的主要是 Set 集合的一些信息， 比如说当前 Set 集合 member 的数量以及当前 Set 集合的版本号和过期时间（用做秒删功能），普通数据实际上就是指的 Set 集合中的 member，作为具体最后 RocksDB 落盘的 kv 格式，下面是具体的实现方式：

1. 每个 Set 集合的 meta_key 和 meta_value 的落盘方式：



meta_key 实际上就是 Set 集合的 key，而 meta_value 由三个部分构成：4Bytes 的 Set size （用于存储当前 Set 集合的大小） + 4Bytes 的 Version （用于秒删功能） + 4Bytes 的 Timestamp （用于记录我们给这个 Set 集合设置的超时时间的时间戳， 默认为0）。

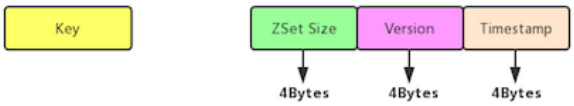
2. Set 集合中 data_key 和 data_value 的落盘方式：

data_key 由四个部分构成：4Bytes 的 Key size（用于记录后面追加的 key 的长度，便与解析） + key 的内容 + 4Bytes 的 Version + member 的内容， 由于 Set 集合只需要存储 member，所以 data_value 实际上就是空串。

ZSet 结构的存储

Blackwidow中 的 ZSet 由两部部分构成，元数据（meta_key，meta_value），和普通数据（data_key，data_value），元数据中存储的主要是 ZSet 集合的一些信息， 比如说当前 ZSet 集合 member 的数量以及当前 ZSet 集合的版本号和过期时间（用做秒删功能），而普通数据就是指的 ZSet 中每个 member 以及对应的 score，由于 ZSet 这种数据结构比较特殊，需要按照 member 进行排序，也需要按照 score 进行排序， 所以我们对于每一个 ZSet 我们会按照不同的格式存储两份普通数据，在这里我们称为 member to score 和score to member，作为具体最后 RocksDB 落盘的 kv 格式，下面是具体的实现方式：

1. 每个 ZSet 集合的 meta_key 和 meta_value 的落盘方式：

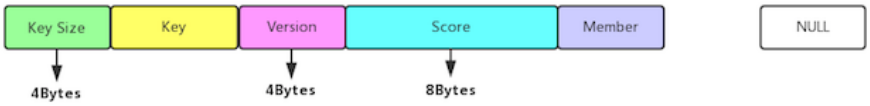


meta_key 实际上就是 ZSet 集合的 key，而 meta_value 由三个部分构成：4Bytes 的 ZSet size（用于存储当前zSet集合的大小） + 4Bytes 的 Version （用于秒删功能） + 4Bytes的 Timestamp （用于记录我们给这个Zset 集合设置的超时时间的时间戳， 默认为0）。

2. 每个 ZSet 集合的 data_key 和 data_value 的落盘方式 （member to score）：

member to socre的 data_key由四个部分构成：4Bytes 的 Key size （用于记录后面追加的 key 的长度，便与解析） + key 的内容 + 4Bytes的 Version + member 的内容， data_value 中存储的其 member 对应的 score 的值，大小为 8 个字节，由于 RocksDB 默认是按照字典序进行排列的，所以同一个zset中不同的 member 就是按照 member 的字典序来排列的（同一个 ZSet 的 key size，key，以及 version，也就是前缀都是一致的，不同的只有末端的 member）。

3. 每个 ZSet 集合的 data_key 和 data_value的落盘方式 （score to member）：



score to member 的 data_key 由五个部分构成：4Bytes 的 Key size（用于记录后面追加的 key 的长度，便与解析） + key 的内容 + 4Bytes 的 Version + 8Bytes 的 Score + member 的内容， 由于 score 和 member 都已经放在 data_key 中进行存储了所以 data_value 就是一个空串，无需存储其他内容了，对于 score to member中 的 data_key 我们自己实现了 RocksDB 的 comparetor，同一个 ZSet 中 score to member 的 data_key 会首先按照 score 来排序， 在 score 相同的情况下再按照 member 来排序。

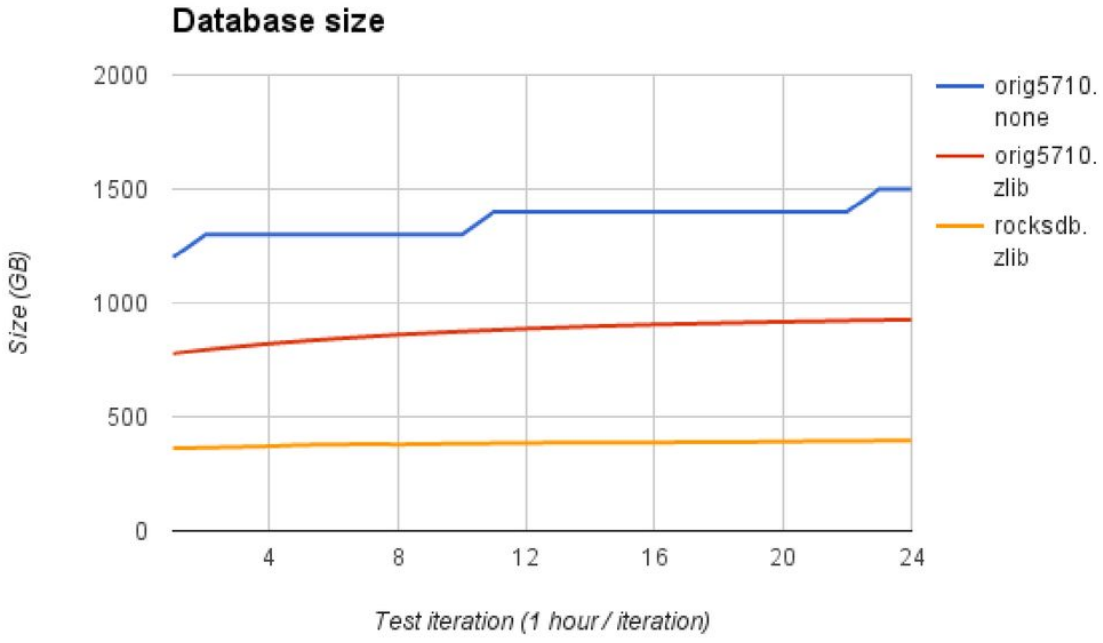
MyRocks

应用场景

大数据量业务

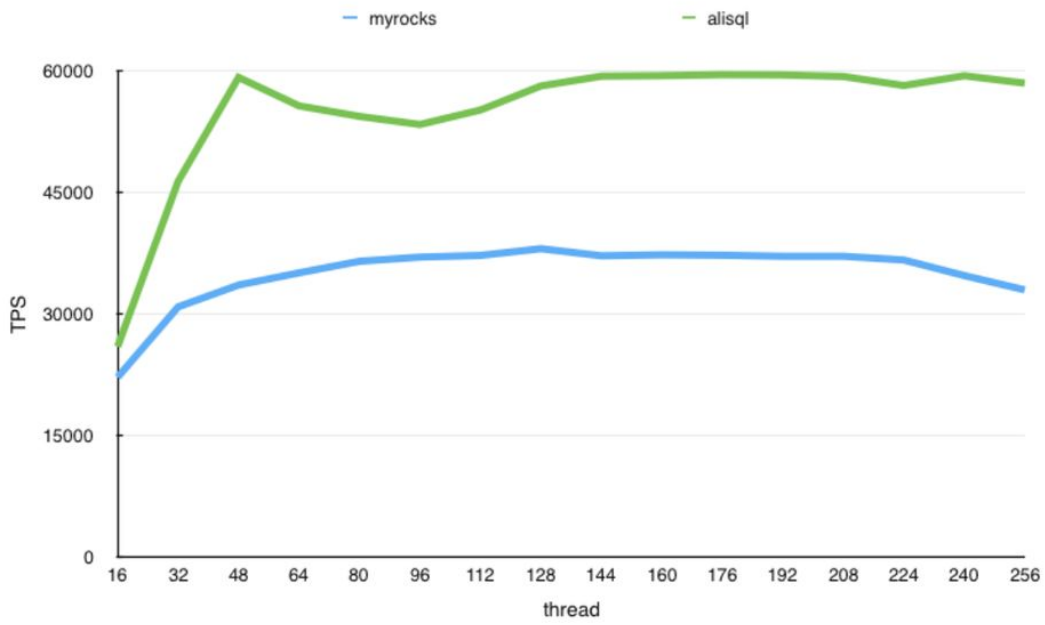
相比 InnoDB，RocksDB 占用更少的存储空间，还具备更高的压缩效率，非常适合大数据量的业务。

在网易内部的业务测试中，某个热门业务的DB实例由于数据量增长很快，DBA不得不频繁进行分表扩容操作。使用 MyRocks 替换 InnoDB 发现，启用压缩的 165GB 的 InnoDB 单表，在 MyRocks 压缩下仅为 51GB；总的存储空间由原来的 26 TB 降为不到 9TB。节省了大量的存储空间，同时延长了 DBA 需要分表扩容的周期。



写密集业务

RocksDB 采用追加的方式记录 DML 操作，将随机写变为顺序写；非常适合用在批量插入和更新频繁的业务场景。在阿里业务中：



事务实现

sequence number

> 谈到 RocksDB 事务，就必须提及 RocksDB 中的 sequence number 机制。RocksDB 中的每一条记录都有一个 sequence number，这个 sequence number 存储在记录的 key 中。
>
> `InternalKey:| User key (string) | sequence number (7 bytes) | value type (1 byte) |`
>
> 对于同样的 User key 记录，在 RocksDB 中可能存在多条，但他们的 sequence number 不同。sequence number 是实现事务处理的关键，同时也是 MVCC 的基础。

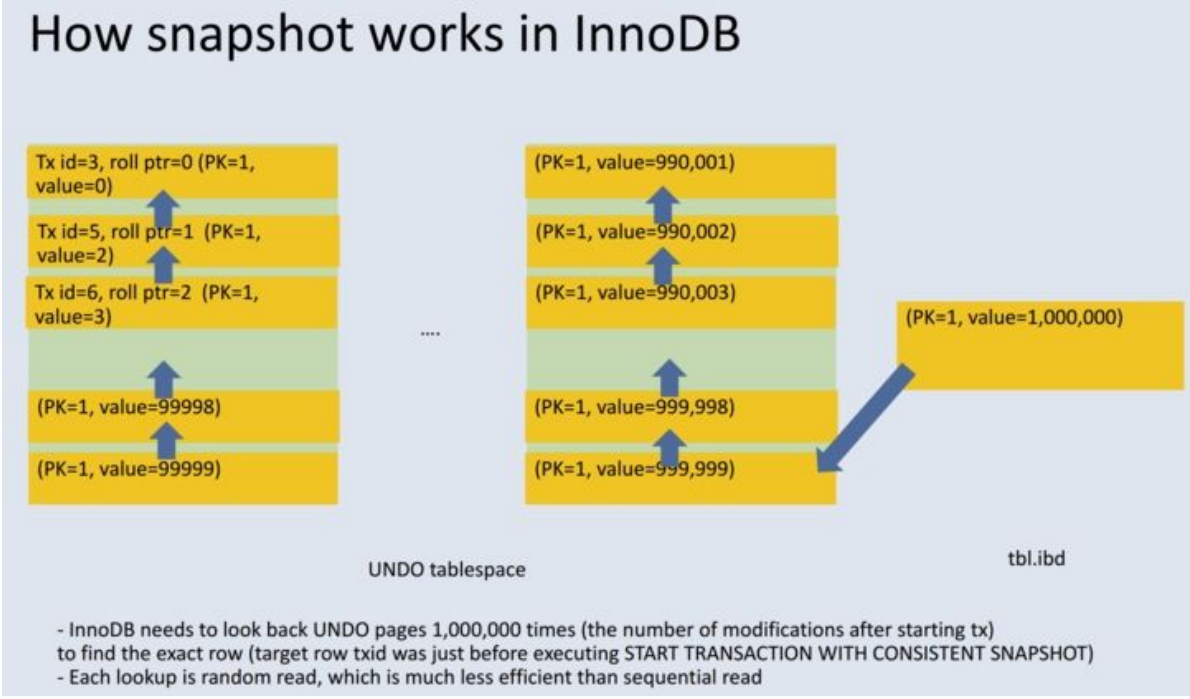
snapshot

> snapshot 是 RocksDB 的快照信息，snapshot 实际就是对应一个 sequence number。假设 snapshot 的 sequence number 为 Sa，那么对于此 snapshot 来说，只能看到 `sequence number <= Sa` 的记录，`sequence number > Sa` 的记录是不可见的。
>
> RocksDB 的 compact 操作与 snapshot 有紧密联系。以我们熟悉的 InnoDB 为例，RocksDB 的 compact 类似于 InnoDB 的 purge 操作， 而 snapshot 类似于 InnoDB 的 read view。 InnoDB 做 purge 操作时会根据已有的 read view 来判断哪些 undo log 可以 purge，而 RocksDB 的 compact 操作会根据已有 snapshot 信息即全局双向链表来判断哪些记录在 compact 时可以清理。
>
> 判断的大体原则是，从全局双向链表取出最小的 snapshot sequence number Sn。 如果已删除的老记录 `sequence number <= Sn`，那么这些老记录在 compact 时可以清理掉。

MVCC

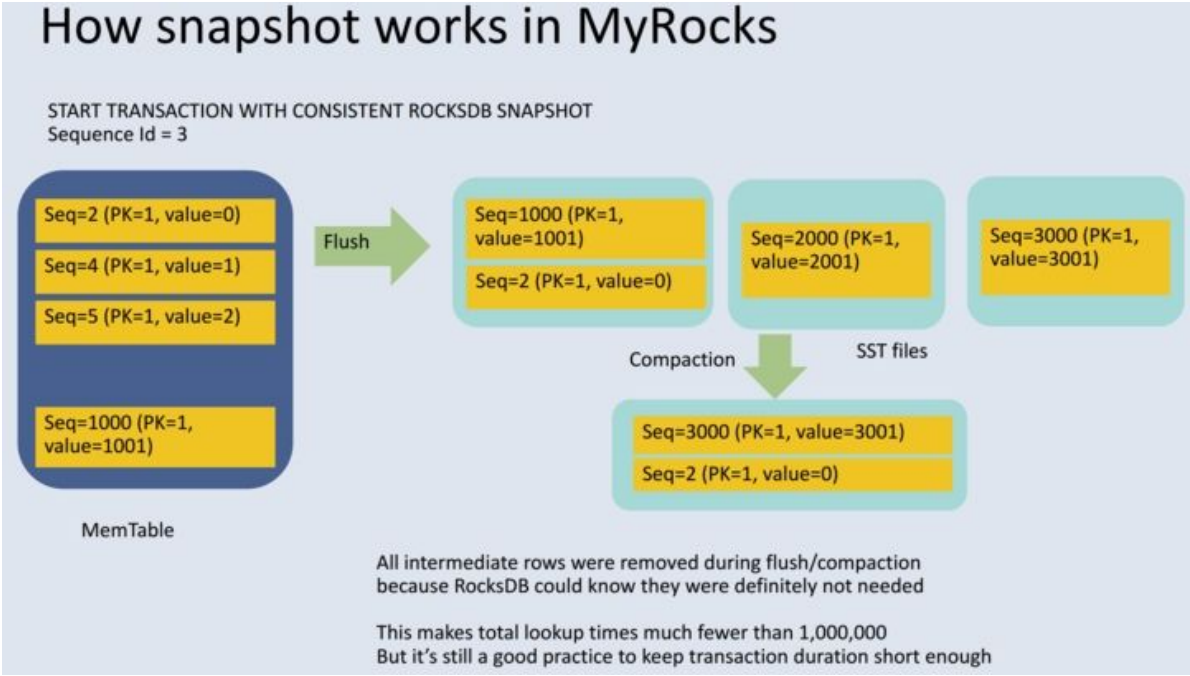
> 有了 snapshot，MVCC 实现起来就很顺利了。记录的 sequence number 天然的提供了记录的多版本信息。 每次查询用户记录时，并不需要加锁。而是根据当前的 sequence number Sn 创建一个 snapshot，查询过程中只取小于或等于 Sn 的最大 sequence number 的记录。查询结束时释放 snapshot。
>
> 详细代码：`DBIter::FindNextUserEntryInternal` 中；

undo



> 对于 InnoDB，由于备份事务 id 小于更新 100 万次增一的事务 id，因此，这 100 万个旧版本记录（即undo）都不会被 purge，这意味着在对该记录进行备份时，需要执行 100 万次版本回溯，每次都是基于记录上的 undo 指针对 undo 页进行随机读，效率很低。

snapshot



> RocksDB 针对 InnoDB 存在的旧版本记录 purge 问题进行了优化，假设原始记录的 sequence number 为2，该版本即为备份事务可见版本，对于比它更大的版本，在 RocksDB 将MemTable dump为 sst 文件，或对 sst 文件进行 Compaction 时会删除中间版本，仅保留当前活跃事务可见版本和记录最新的版本。这样既满足 MVCC 要求，又提高了快照读效率，同时也减少了需占用的存储空间。

隔离级别

> 隔离级别也是通过 snapshot 来实现的。在 InnoDB 中，隔离级别为 read-committed 时，事务中每个DML操作都会建立一个 read view，隔离级别为 repeatable-read 时，只在事务开启时建立一次 read view。 RocksDB 同 InnoDB类似，隔离级别为 read-committed 时，事务中每个 DML 操作都会建立一个 snapshot，隔离级别为 repeatable-read 时，只在事务开启时第一个 DML 操作建立一次 snapshot。
>
> 详细代码：`RocksDB_commit` 中；

锁

> MyRocks 目前只支持一种锁类型：排他锁（X锁），并且所有的锁信息都保存在内存中。在 RR 隔离界别下只在主键上实现 gap 锁；