RocksDB

# 一个进程只读打开rocksdb，一个进程可写打开rocksdb，数据能否更新?

一个进程只读模式打开rocksdb，另外一个进程可写模式打开同一个rocksdb，可写进程更新了一个key的data，只读模式的进程获取不到这个更新的data，这是为什么?



雷鹏
曾经的 Terark，现今的 Topling，更快，更好



你要的功能，在 RocksDB 中叫做 Secondary Instance。

Topling 的 Todis(外存版Redis) 的从库机制就使用了 Secondary Instance。

Todis@github 底层使用的存储引擎是 ToplingDB，ToplingDB fork 自 RocksDB，进行了很多增强和改进，例如，通过 SidePlugin 旁路插件化机制，就连 DB 的打开方式，也只需要通过修改配置文件(json/yaml) 就可以了，Todis 本身的代码不需要对 Secondary Instance 进行任何专门的处理。

因为 RocksDB 的 Secondary Instance 已经实现很长时间了，我们使用它的时候，就直接用了，没有专门去进行调研、测试，于是，这一用，就用出问题了：

当我们写完代码进行集成测试的时候，奇怪的事情发生了：从从库不断查询之前写入主库的数据，竟然是时有时无的！一开始一直以为是我们自己的问题，然而——

经过反复的排查，我们最终确定这是 RocksDB 本身实现的问题：

首先， RocksDB 提供了一个方法 DBImplSecondary::TryCatchUpWithPrimary 用于从库和主库之间的同步。我们先调用 TryCatchUpWithPrimary 进行同步，再调用从库的 Get 进行查询，就可以拿到新写入主库的数据。这个方法应当在任何情况有效，然而我们发现，在同一个线程依次调用 TryCatchUpWithPrimary 和 Get 方法可以正常工作；但如果是有一个线程专门调用 TryCatchUpWithPrimary ，另

外一个线程执行 Get 进行查询，新写入主库的数据就不会被读取到！
我们立刻提交了一个 issue，并提供了一个稳定复现此 bug 的代码片段。一个月后， RocksDB 的开发人员修复了这个 bug 。 RocksDB 在储存 Key Value 数据时，还会额外储存一个 LogSequenceNumber 。这个 SequenceNumber 在 MemTable 中是单调递增的。而当 RocksDB 读取数据时，会先储存当前的 SequenceNumber 作为快照 snapshot，对于这一次的读操作，只有记录中 LogSequenceNumber 不大于 snapshot 的 Key 可以被读取到。在这个 bug 中， RocksDB 近期的修改导致了从库调用 TryCatchUpWithPrimary 时记录的当前 SequenceNumber 有时会产生回退，此时新写入主库的 Key 的 LogSequenceNumber 大于从库当前的 SequenceNumber ，自然是无法读取出来的。

发布于 2022-03-18 20:04



**刘风**



首先RocksDB中的每次更新行为都有一个LogSequenceNumber，

从最初的0开始，每次写入都会++

这个num在memtable中单向递增。

当以以只读模式打开时，会记录打开时的num, 大于该num的所有操作不可见。

发布于 2019-08-16 13:45



**flaneur**
おやすみプンプン



你经常看 TA 的内容

有个 Secondary Instance 特性，大约原理是只读实例去 tail WAL 跟 manifest 文件，重放到自己的内存里，是个异步的复制过程。

编辑于 2022-04-26 09:55

# Read only and Secondary instances

logesh056 edited this page on May 26 · 15 revisions

## Feature Overview

RocksDB database can be opened in read-write mode (aka. **Primary Instance**) or can be opened in read-only mode. RocksDB supports two variations of read-only mode:

- **Read-only Instance** - Opens the database in read-only mode. When the Read-only instance is created, it gets a static read-only view of the Primary Instance's database contents
- **Secondary Instance** – Opens the database in read-only mode. Supports extra ability to dynamically catch-up with the Primary instance (through a manual call by the user – based on their delay/frequency requirements)

The Primary Instance is a regular RocksDB instance capable of read, write, flush and compaction. The Read-only and Secondary Instances supports read operations alone.

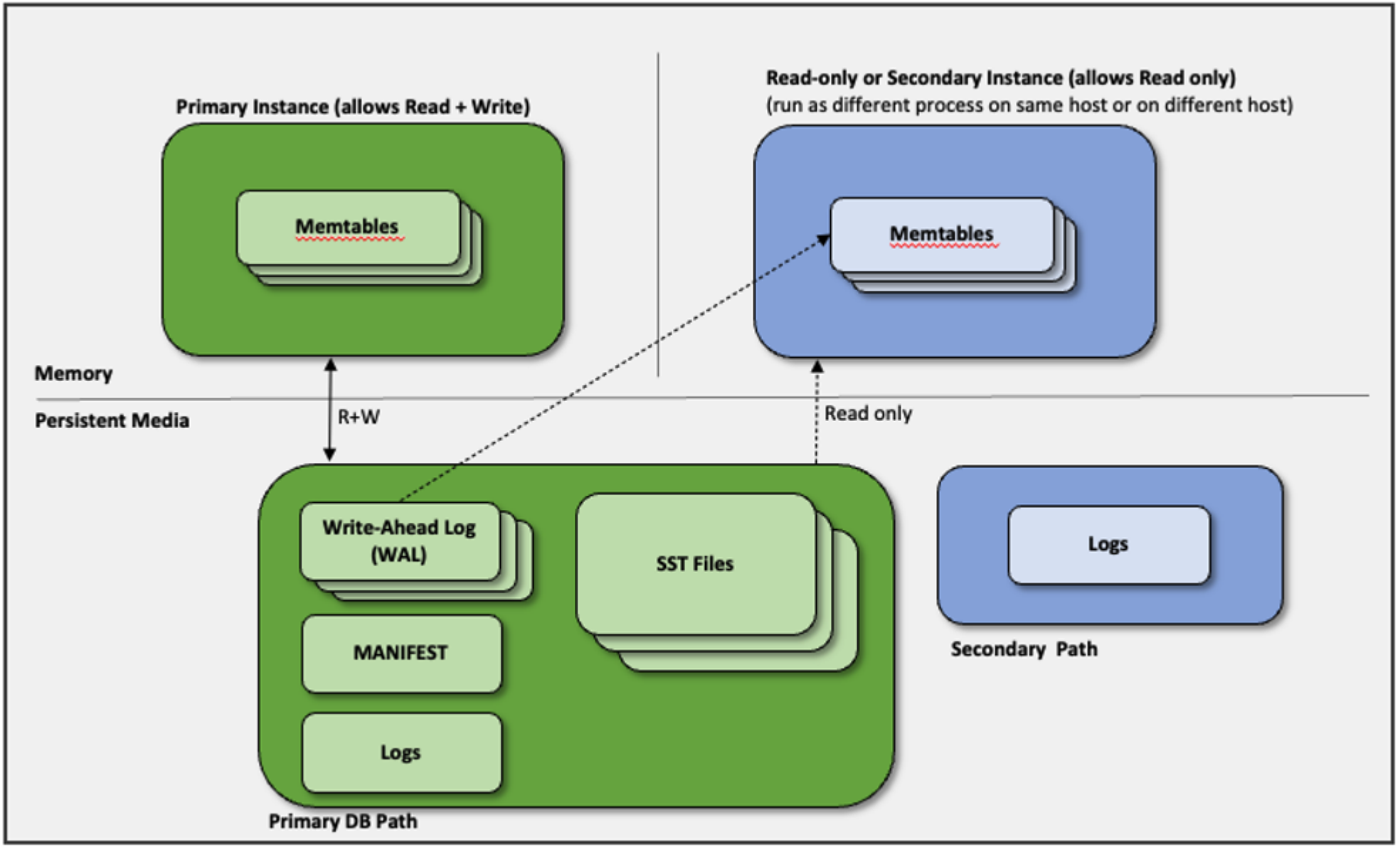
**Only single instance of Primary is allowed; but many concurrent Read-only and Secondary Instances are allowed.**

The Read-only/Secondary instances can be run as a different process on the same host (as Primary) or run on a different host (when Primary DB resides in a distributed File System). The Read-only/Secondary instances opens all files in the primary DB Path in read-only mode and never creates any new files in that directory – hence, these instances can be run with read-only user credentials to that directory/its contents.

Read-only and Secondary instances can be used to distribute read workloads and to scale read performance.

The Secondary Instance is also used as the building block for the new RocksDB feature called "Remote Compaction" which allows distributing/offloading the Compaction operation to a different host. Remote Compaction is triggered using a call (on a remote host) to `DB::OpenAndCompact()`, which internally creates a Secondary instance and runs compaction **without installing the compaction result in the primary's MANIFEST**.

On the Primary Instance, all the flushed data will be in the SST files and the unflushed data will be in the memtables (with corresponding log entry in the WAL file (if WAL is enabled)). The Read-Only/Secondary instance can access the SST files (under the Primary DB Path specified in the `OpenForReadOnly()/OpenAsSecondary()` call) – but, may not have access to memtables of the primary instance (when it is in different process space or in a different host). So, during the Read-only/Secondary instance setup, their memtables are constructed by replaying the log entries in the WAL files. These memtables are never flushed.



Read-only Instance does not have support to catch-up with the Primary Instance.

Secondary Instance supports ability to catch-up with the Primary - users have to call `TryCatchupWithPrimary()` to catchup with the Primary. On this call, the Secondary Instance:

- Catches up with the MANIFEST changes (SST file additions/deletions). Based on the latest MANIFEST information, it closes obsolete SST files and opens the new SST files.
- Drops old memtables and recreates its memtables based on the current WAL contents
- Checks for Column families that were dropped on the Primary after the Secondary was created or after previous invocation of this call. However, if the Secondary does not delete the corresponding column family handle(s), the data of that column family is still accessible to the Secondary

Also, see Checkpoints

## Feature Comparison

| | Primary Instance | Read-Only Instance | Secondary Instance |
|---|---|---|---|
| Multiple concurrent instances allowed? | No | Yes | Yes |
| Read Operations | Yes | Yes | Yes |
| Write Operations and Automatic/Manual Flush/Compaction | Yes | No | No (but, supports a special form of compaction used for Remote Compaction feature(see notes in “Feature overview” section)) |
| Support for Open() with subset of Column-Families | No | Yes (but – default CF (CF-0) cannot be skipped) | Yes (but – default CF (CF-0) cannot be skipped) |
| Ability to Catch-up with Primary | N/A | No | Yes |
| "`options.max_open_files = -1`" mandatory? | No | No | Yes |

| | Primary Instance | Read-Only Instance | Secondary Instance |
|---|---|---|---|
| Support for failing `Open` if WAL is present | N/A | Yes | No |
| Support for Tailing Iterators | Yes | N/A | No |
| Support for Snapshots based read (`read_options.snapshot`) | Yes | Yes | No |
| Default location where info logs are created | Under the Primary DB Path Directory | No info log files created | Under the Secondary DB Path Directory |

## Example usage (Read-Only Instance)

```
const std::string kDbPath = "/tmp/rocksdbtest";
...
// Assume we have already opened a regular
// whose database directory is kDbPath.
  DB* ro_db = nullptr;
  Options options;

  /* Open Readonly with default CF alone */
  auto s = DB::OpenForReadOnly(options, kDBPath, &ro_db);
  assert(s.ok() && ro_db);

  ReadOptions ropts;

  Iterator* iter1 = ro_db->NewIterator(ropts);
  iter1->SeekToFirst();

  auto key_cnt = 0;
  for ( ;iter1->Valid();  iter1->Next(), key_cnt++) {
  }
  fprintf(stdout, "read %d keys\n", key_cnt);
```

```
...
```

## Example usage (Secondary Instance)

```
const std::string kDbPath = "/tmp/rocksdbtest";
...
// Assume we have already opened a regular RocksDB instance db_primary
// whose database directory is kDbPath.
assert(db_primary);

Options options;
options.max_open_files = -1;

// Secondary instance needs its own directory to store info logs (LOG)
const std::string kSecondaryPath = "/tmp/rocksdb_secondary/";
DB* db_secondary = nullptr;

Status s = DB::OpenAsSecondary(options, kDbPath, kSecondaryPath, &db_secondary);
assert(!s.ok() || db_secondary);

// Let secondary **try** to catch up with primary
s = db_secondary->TryCatchUpWithPrimary();
assert(s.ok());

// Read operations
std::string value;
s = db_secondary->Get(ReadOptions(), "foo", &value);
...
```

```
...
RocksDB secDb = null;
File dbDir = new File("/tmp/rocksdbtest");
// Secondary instance needs its own directory to store info logs (LOG)
File secDir = new File("/tmp/rocksdb_secondary/");
```

```
try {
      Files.createDirectories(dbDir.getAbsoluteFile().toPath());
      Files.createDirectories(secDir.getAbsoluteFile().toPath());
      Options options;
      options.max_open_files = -1;
      secDb = Rocksdb.openAsSecondary(options,
                    dbDir.getAbsolutePath(),
                    secDir.getAbsolutePath());
     }
catch (IOException | RocksDBException ex) {
    throw new RuntimeException("Exception during open Secondary db", ex);
 }


// Let secondary **try** to catch up with primary
secDb..tryCatchUpWithPrimary();


// Read operations
byte[] value = secDb.get("foo".getBytes());
...
```

More detailed example can be found in [multi_processes_example.cc](https://github.com/facebook/rocksdb/blob/main/examples/multi_processes_example.cc).

## Current Limitations and Caveats

- If the writes on Primary Instance does not have WAL enabled ( `WriteOptions.disableWAL == true` ), the Read-only/Secondary Instances will not have visibility of data residing in Primary's memtables – resulting in partial view of the database.
- RocksDB relies heavily on compaction to improve read performance. If `TryCatchUpWithPrimary()` was invoked on the Secondary right before a compaction (on Primary), then Secondary could achieve lower read performance (compared to Primary) until next invocation of `TryCatchUpWithPrimary()` .
- The Secondary Instance specific limitations:
  - Secondary must be opened with `max_open_files = -1` , indicating it must keep all file descriptors open to prevent them from becoming inaccessible after the primary unlinks them, which does not work on some non-POSIX file systems. We have a plan to relax this limitation in

the future.

- Reads from a specific snapshot ( `ReadOptions.snapshot` ) is not currently supported
- Tailing Iterators are not supported.
- Column families created on the Primary after the Secondary Instance starts will be ignored unless the Secondary Instance closes and restarts with the newly created column families..