facebook

MyRocks Introduction

Yoshinori Matsunobu
Production Database Engineer, Facebook
Apr 2017– Percona Live Tutorial

Agenda

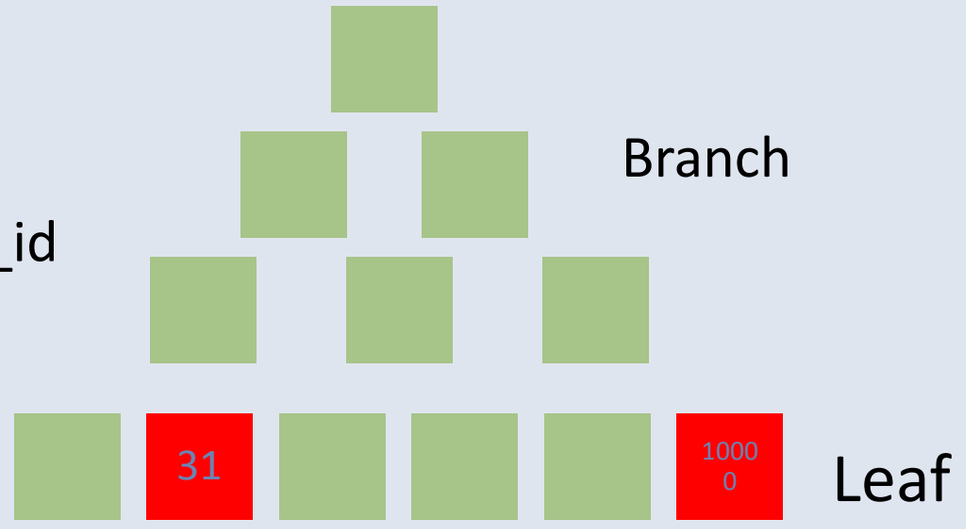
- MyRocks overview
- Getting Started

H/W trends and limitations

- SSD/Flash is getting affordable, but MLC Flash is still a bit expensive
- HDD: Large enough capacity but very limited IOPS
  - Reducing read/write IOPS is very important -- Reducing write is harder
- SSD/Flash: Great read iops but limited space and write endurance
  - Reducing space is higher priority

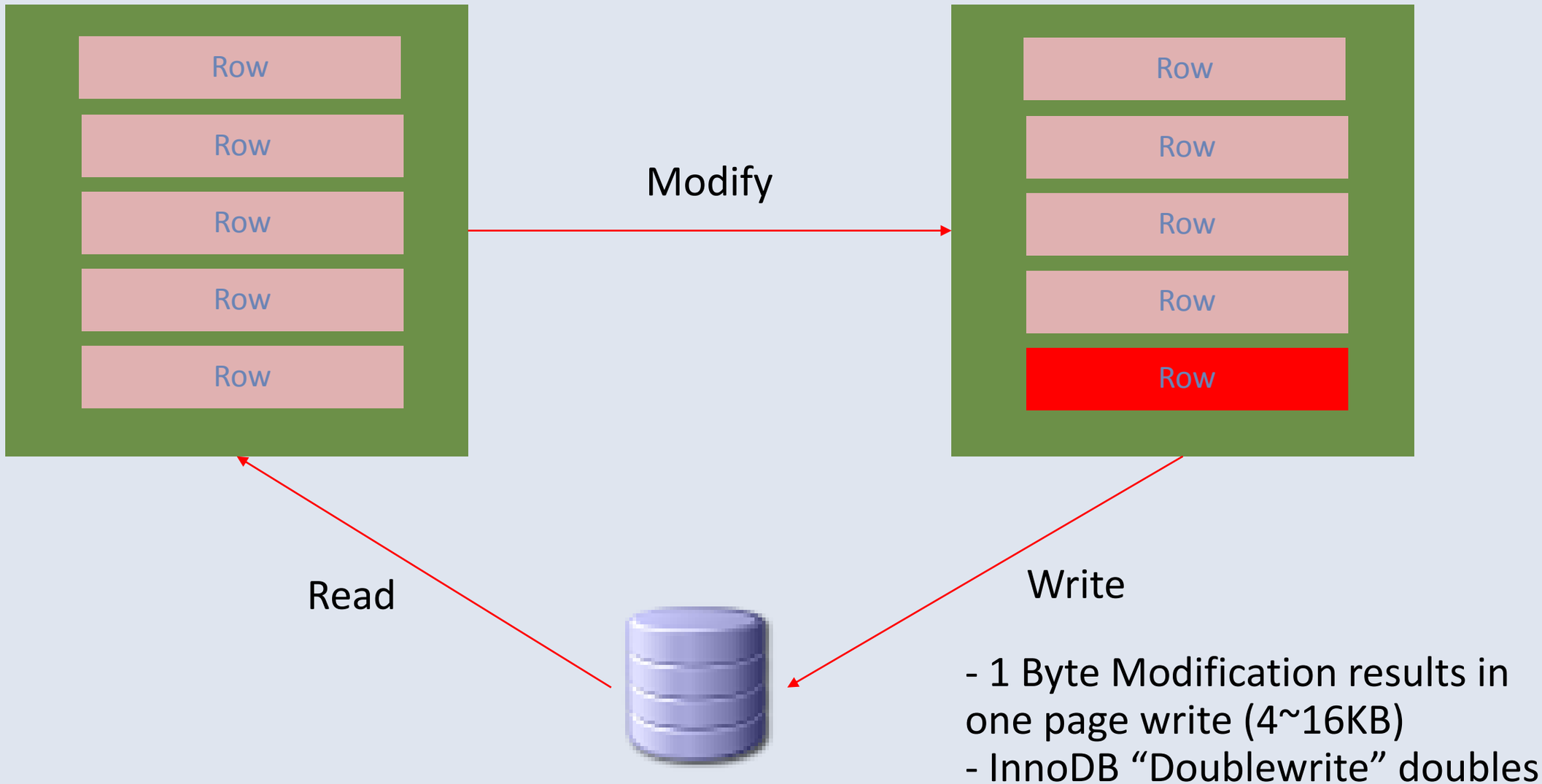
Random Write on B+Tree

```
INSERT INTO message (user_id) VALUES (31);
INSERT INTO message (user_id) VALUES (10000);
…..
```



- B+Tree index leaf page size is small (16KB in InnoDB)
- Modifications in random order => Random Writes, and Random Reads if not cached
- N rows modification => In the worst case N different random page reads and writes per index

Write Amplification on B+Tree



- 1 Byte Modification results in one page write (4~16KB)
- InnoDB “Doublewrite” doubles write volume

B+Tree Fragmentation increases Space

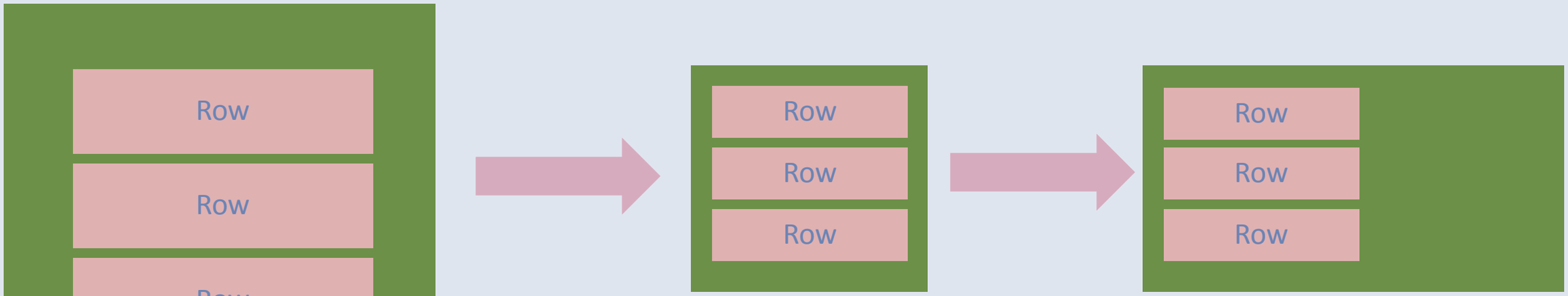
```
INSERT INTO message_table (user_id) VALUES (31)
```

Leaf Block 1	
user_id	RowID
1	10000
2	5
3	15321
…	
60	431

Leaf Block 1	
user_id	RowID
1	10000
…	
30	333
Empty	

Leaf Block 2	
user_id	RowID
31	345
…	
60	431
Empty	

Compression issues in InnoDB



Uncompressed
16KB page

Compressed
to 5KB

Using 8KB space
on storage

0~4KB => 4KB
4~8KB => 8KB
8~16KB => 16KB

New (5.7~) punch-hole compression has similar issue

RocksDB

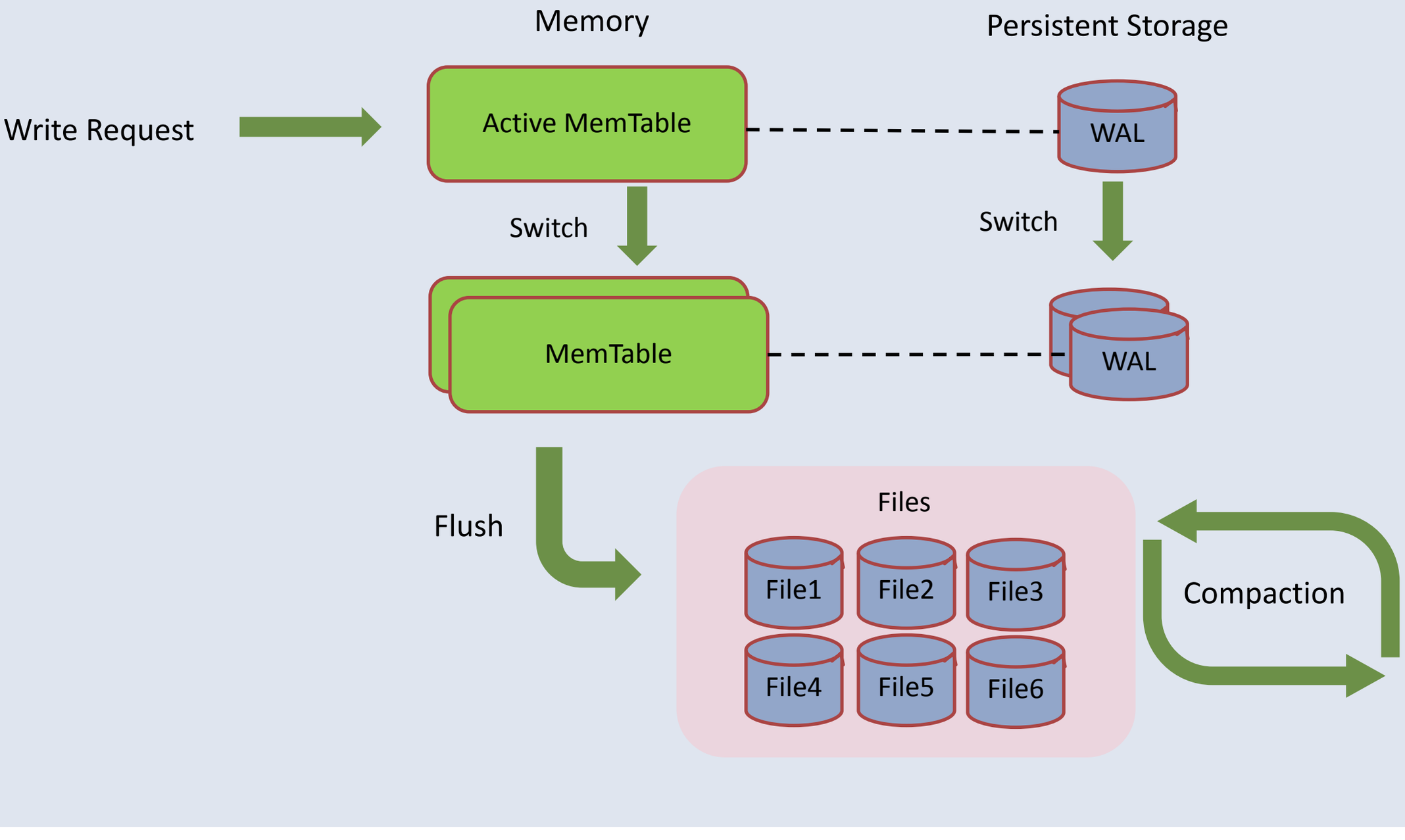
- http://rocksdb.org/
- Forked from LevelDB
  - Key-Value LSM persistent store
  - Embedded
  - Data stored locally
  - Optimized for fast storage

- LevelDB was created by Google
- Facebook forked and developed RocksDB
- Used at many backend services at Facebook, and many external large services
- MyRocks == MySQL with RocksDB Storage Engine

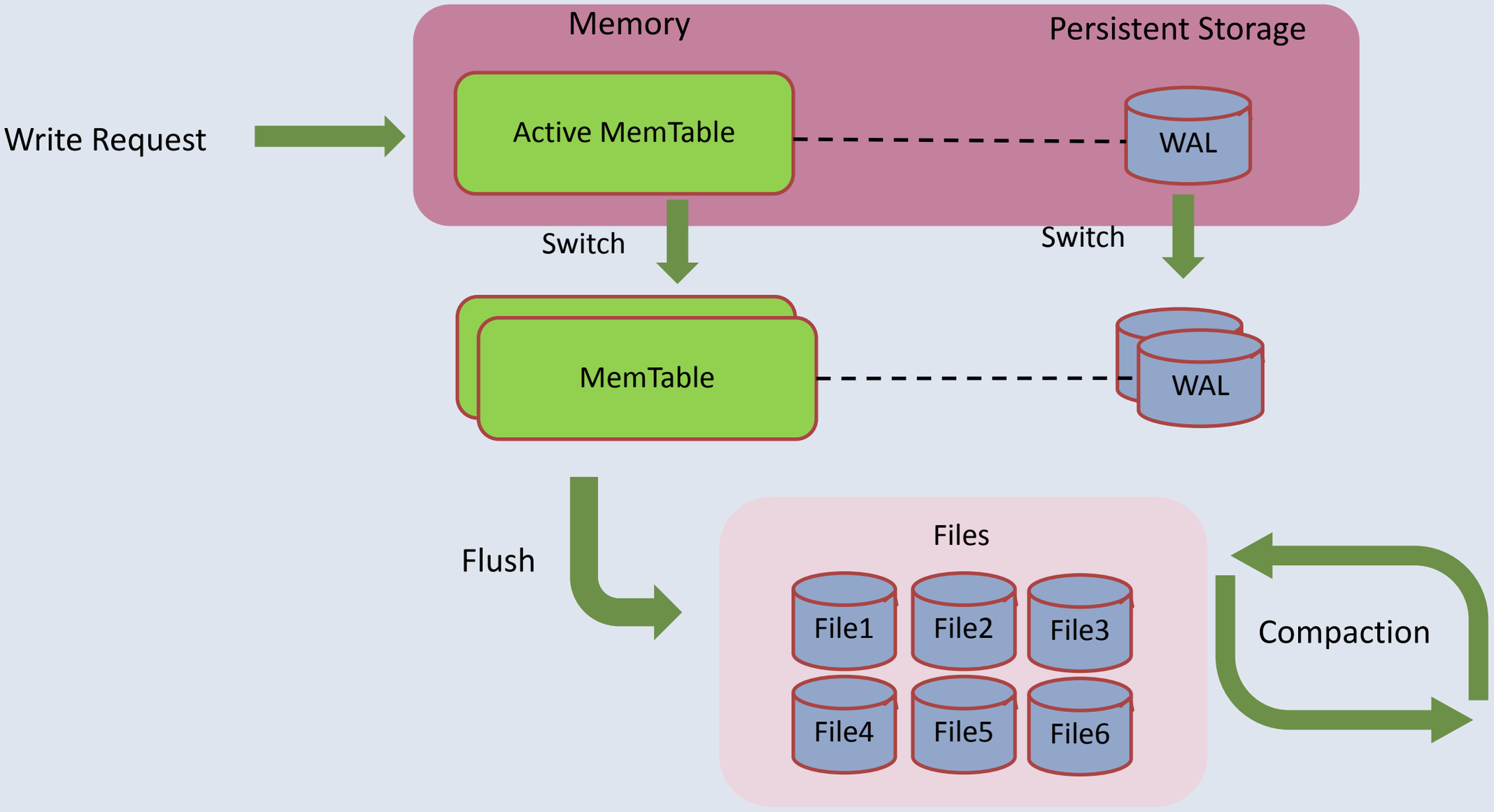
RocksDB architecture overview

- Leveled LSM Structure
- MemTable
- WAL (Write Ahead Log)
- Compaction
- Column Family

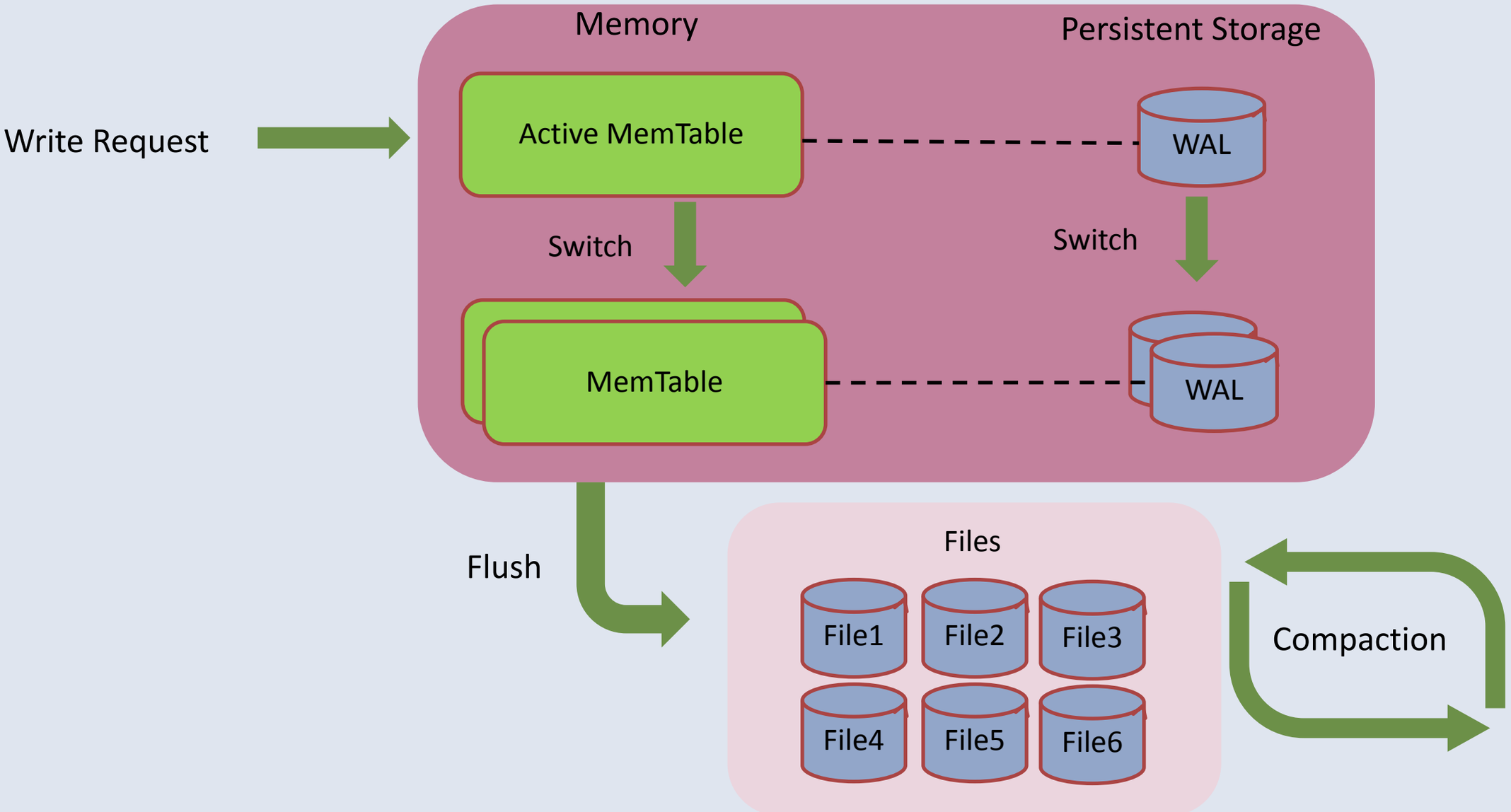
RocksDB Architecture



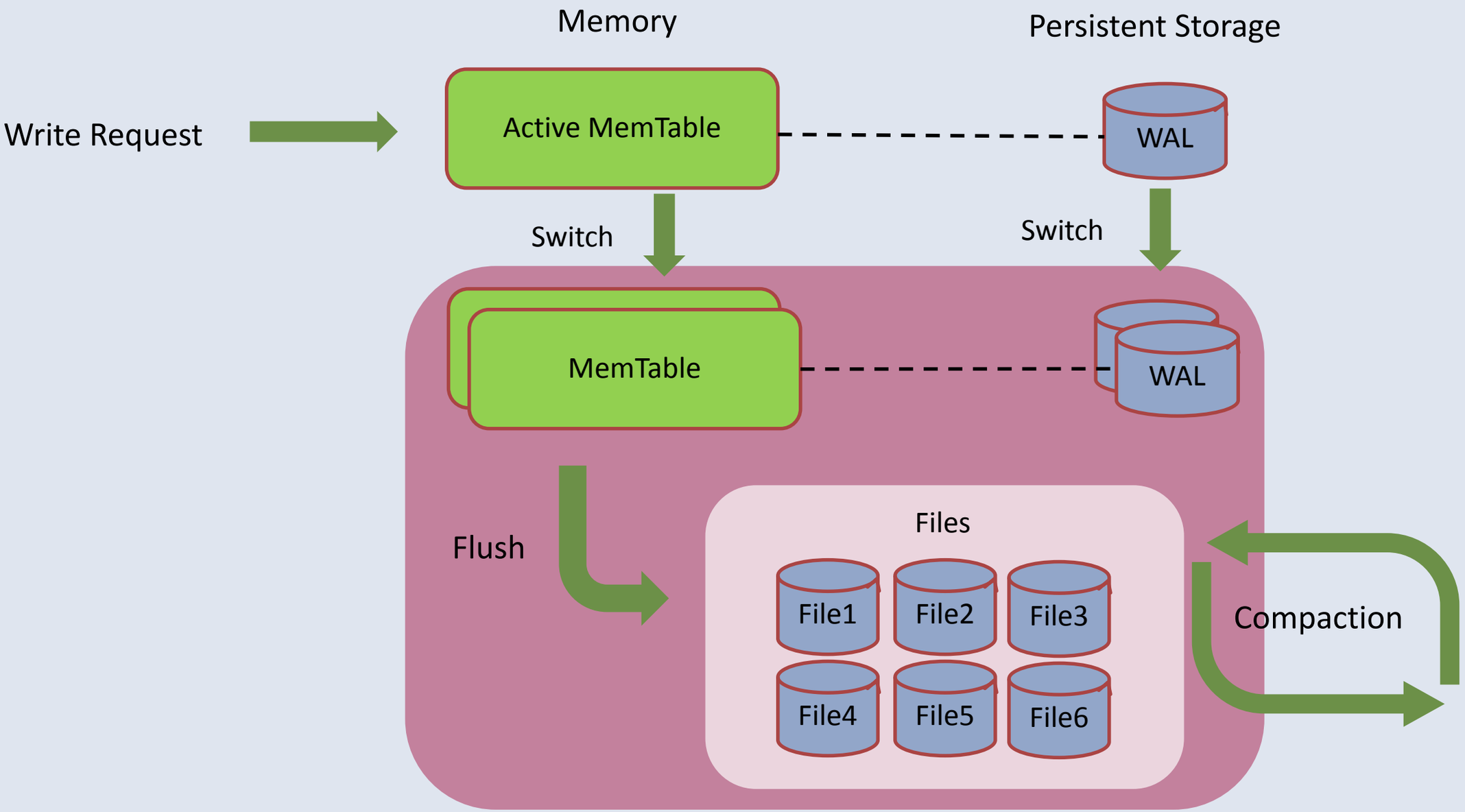
Write Path (1)


Memory
Persistent Storage
Write Request
Active MemTable
WAL
Switch
Switch
MemTable
WAL
Flush
Files
File1
File2
File3
File4
File5
File6
Compaction

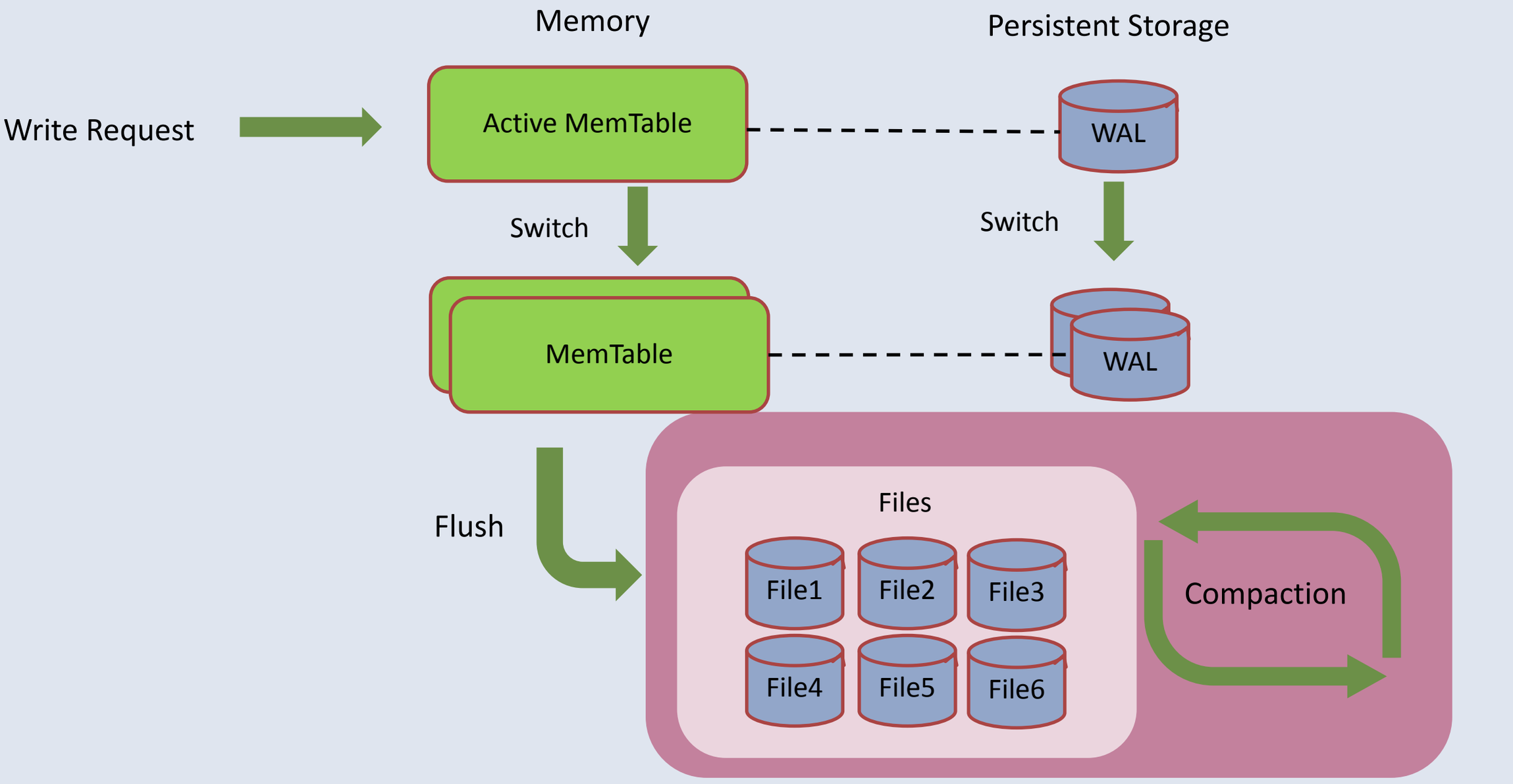
Write Path (2)


Memory
Persistent Storage
Write Request
Active MemTable
WAL
Switch
Switch
MemTable
WAL
Flush
Files
File1
File2
File3
File4
File5
File6
Compaction

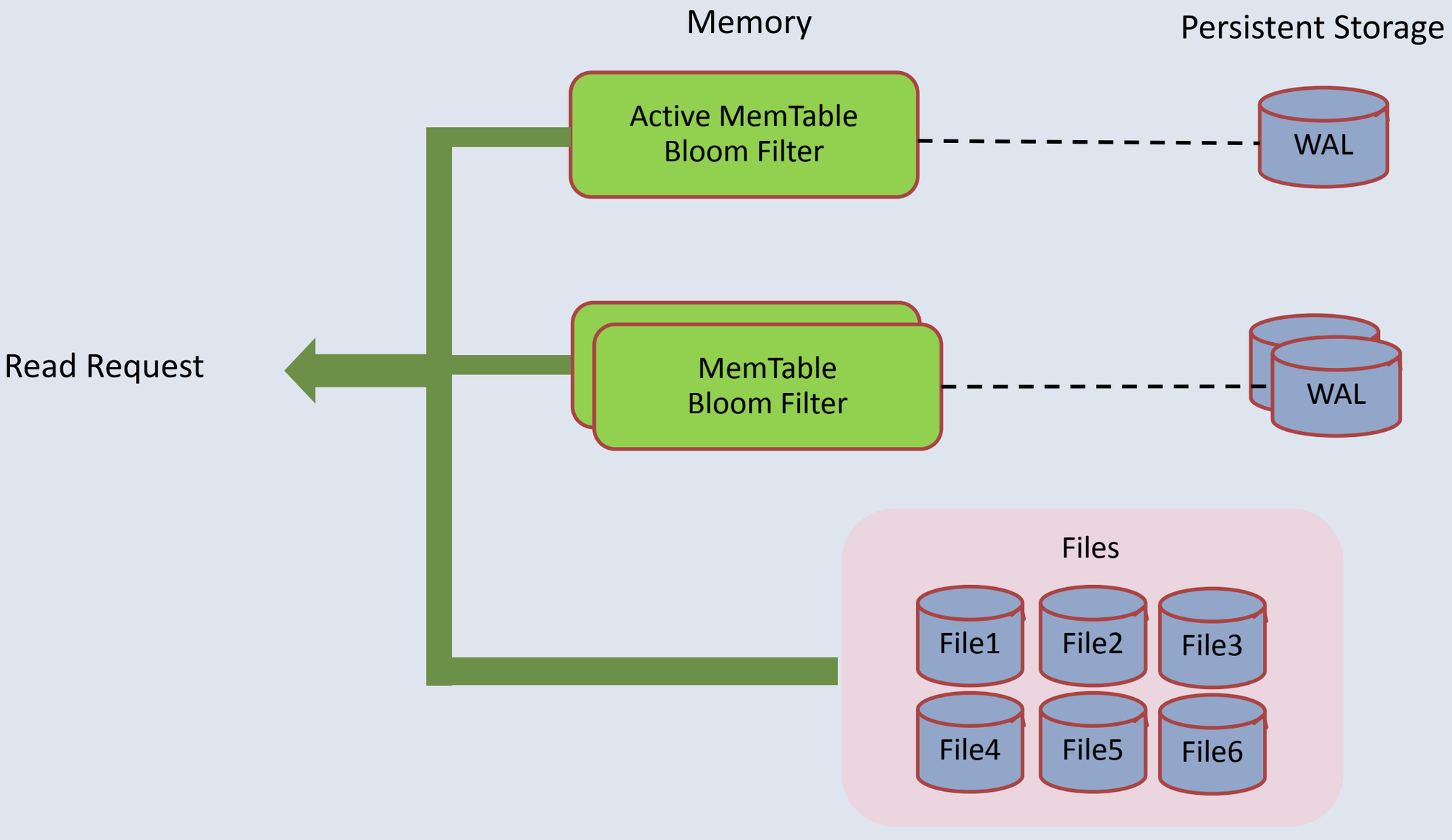
Write Path (3)


Memory
Persistent Storage
Write Request
Active MemTable
WAL
Switch
Switch
MemTable
WAL
Flush
Files
File1
File2
File3
File4
File5
File6
Compaction

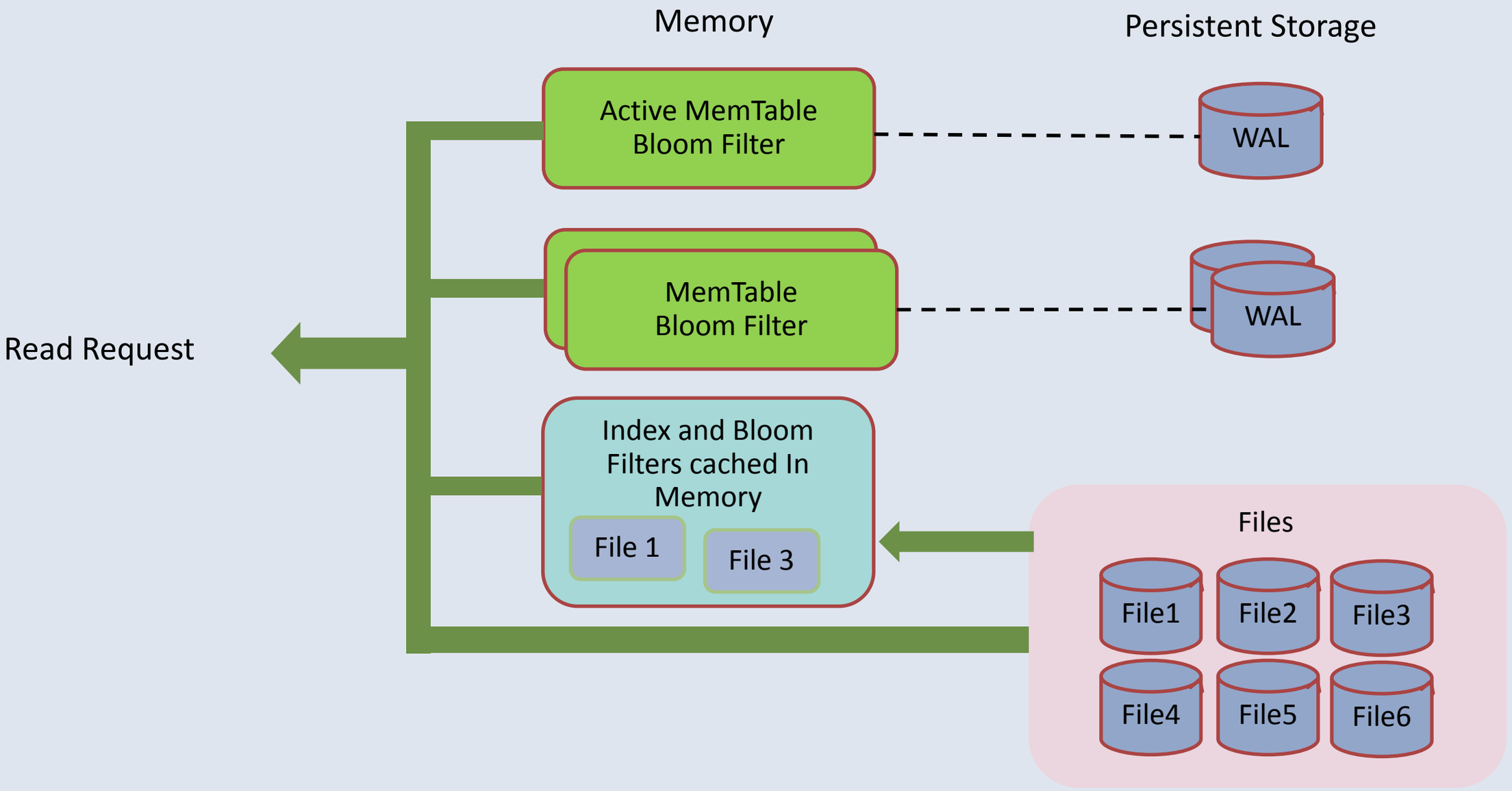
Write Path (4)


Memory
Persistent Storage
Write Request
Active MemTable
WAL
Switch
Switch
MemTable
WAL
Flush
Files
File1
File2
File3
File4
File5
File6
Compaction

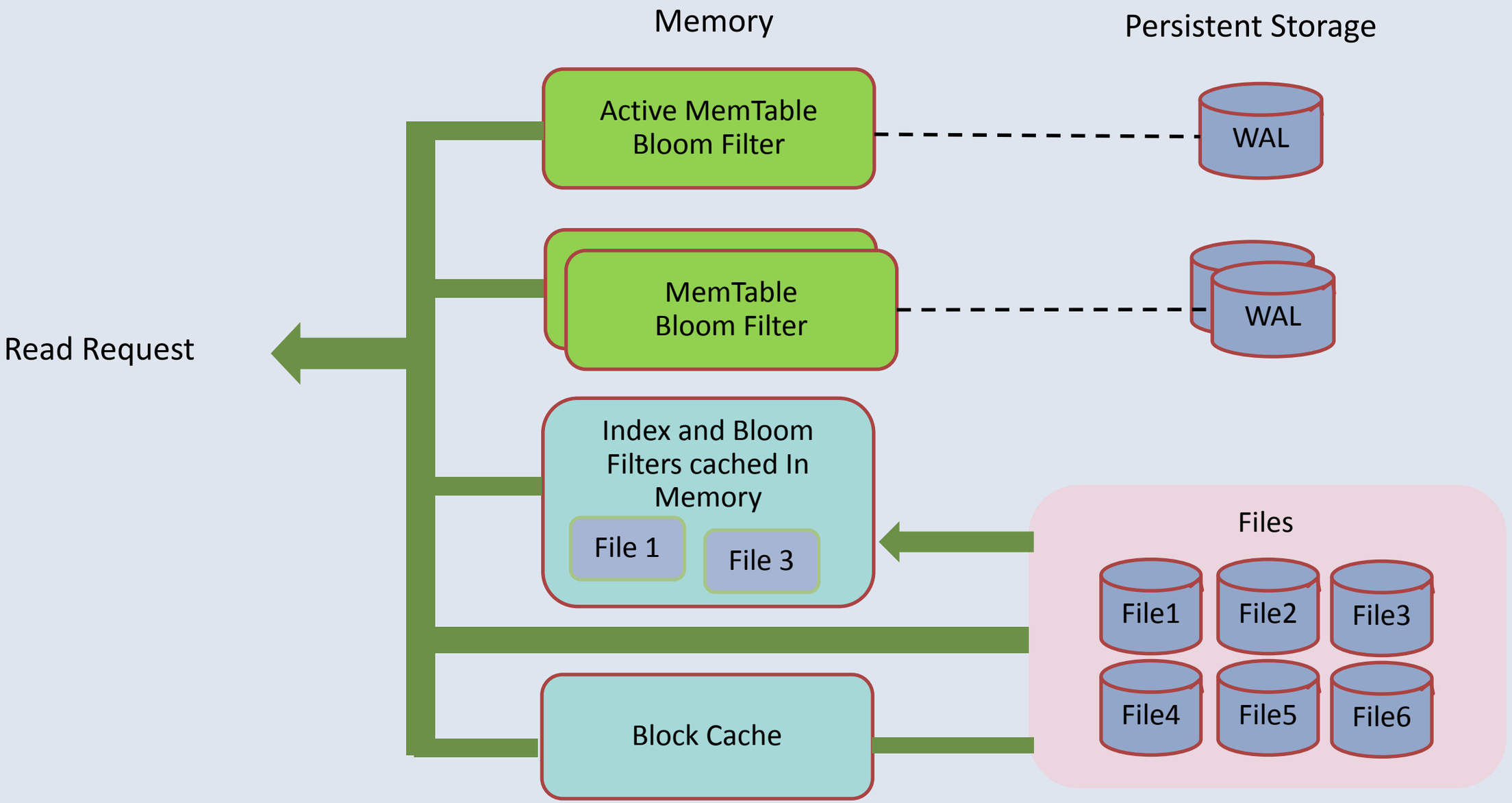
Read Path


Memory
Persistent Storage
Active MemTable
Bloom Filter
WAL
Read Request
MemTable
Bloom Filter
WAL
Files
File1
File2
File3
File4
File5
File6

Read Path

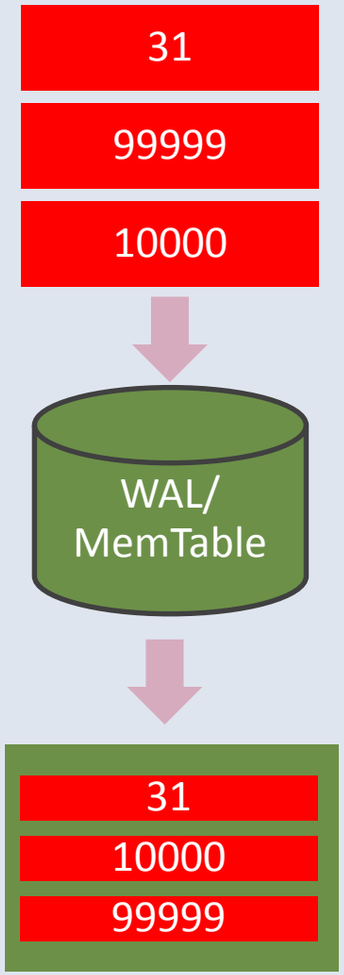

Memory
Persistent Storage
Active MemTable
Bloom Filter
WAL
MemTable
Bloom Filter
WAL
Read Request
Index and Bloom
Filters cached In
Memory
File 1
File 3
Files
File1
File2
File3
File4
File5
File6

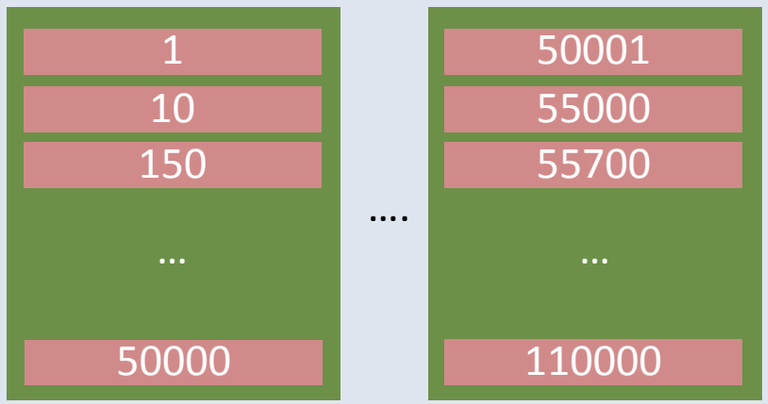
Read Path

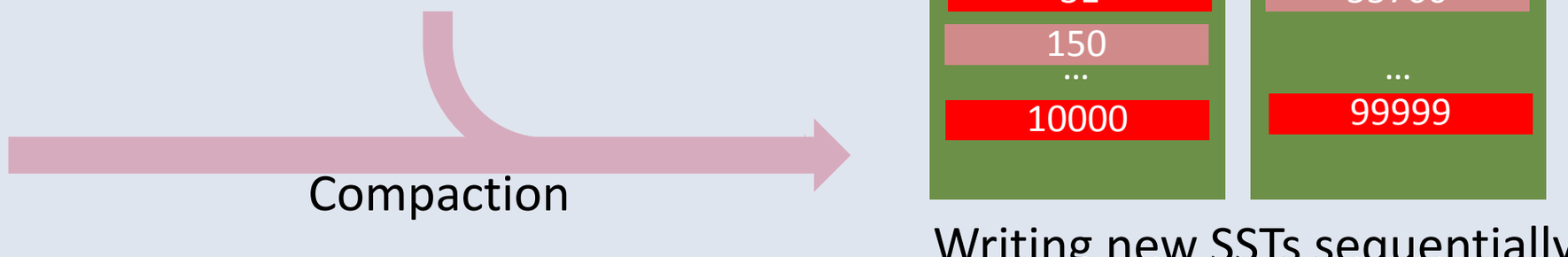

Memory
Persistent Storage
Active MemTable
Bloom Filter
WAL
MemTable
Bloom Filter
WAL
Read Request
Index and Bloom
Filters cached In
Memory
File 1
File 3
Files
File1
File2
File3
File4
File5
File6
Block Cache

How LSM works

```
INSERT INTO message (user_id) VALUES (31);
INSERT INTO message (user_id) VALUES (99999);
INSERT INTO message (user_id) VALUES (10000);
```

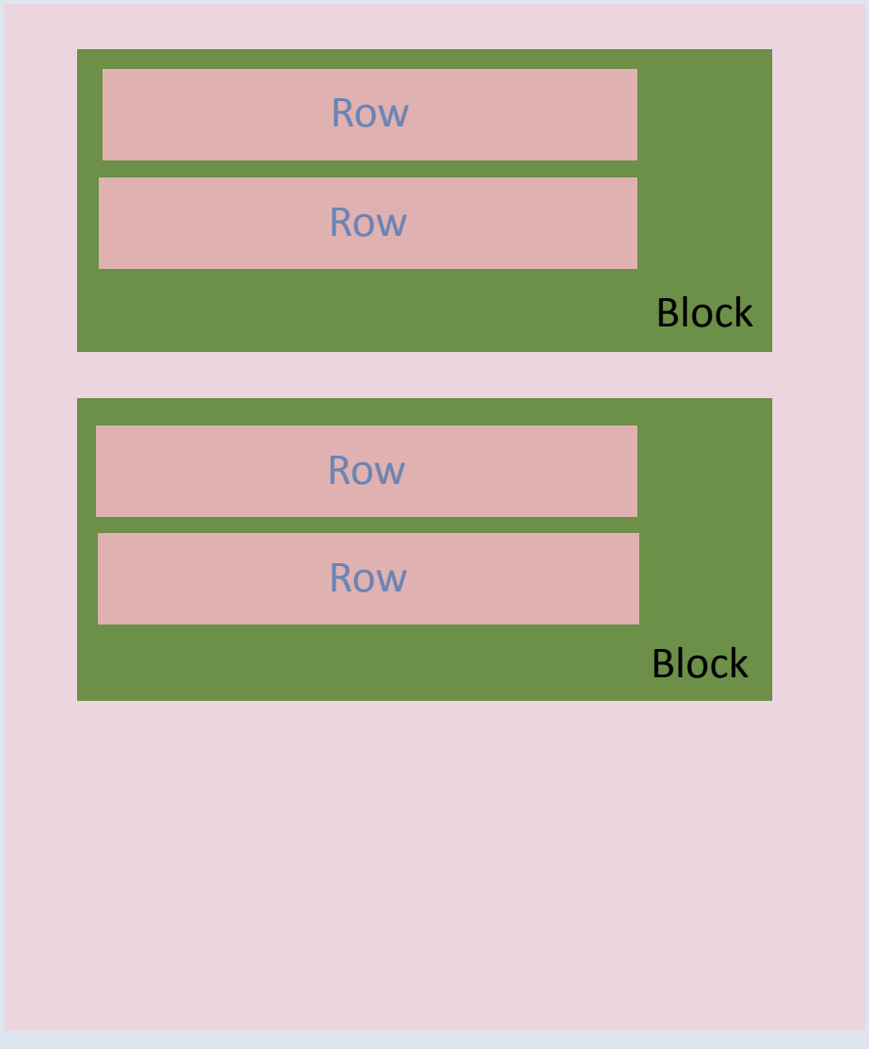




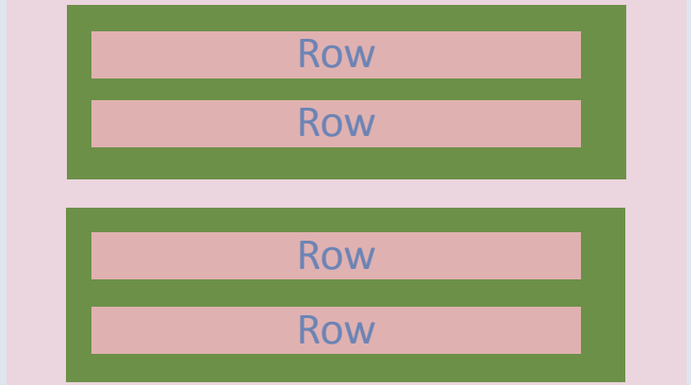


N random row modifications => A few sequential reads & writes

LSM handles compression better



16MB SST File

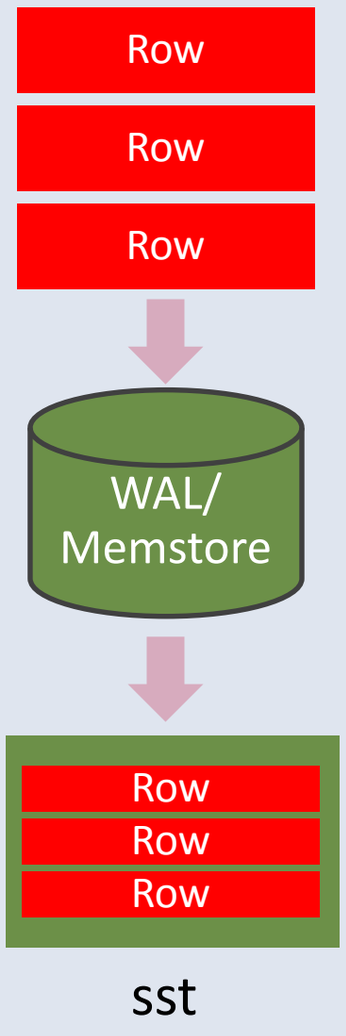


5MB SST File

=> Aligned to OS sector (4KB unit)
=> Negligible OS page alignment overhead

Reducing Space/Write Amplification

Append Only



Prefix Key Encoding

id1	id2	id3
100	200	1
100	200	2
100	200	3
100	200	4

id1	id2	id3
100	200	1
		2
		3
		4

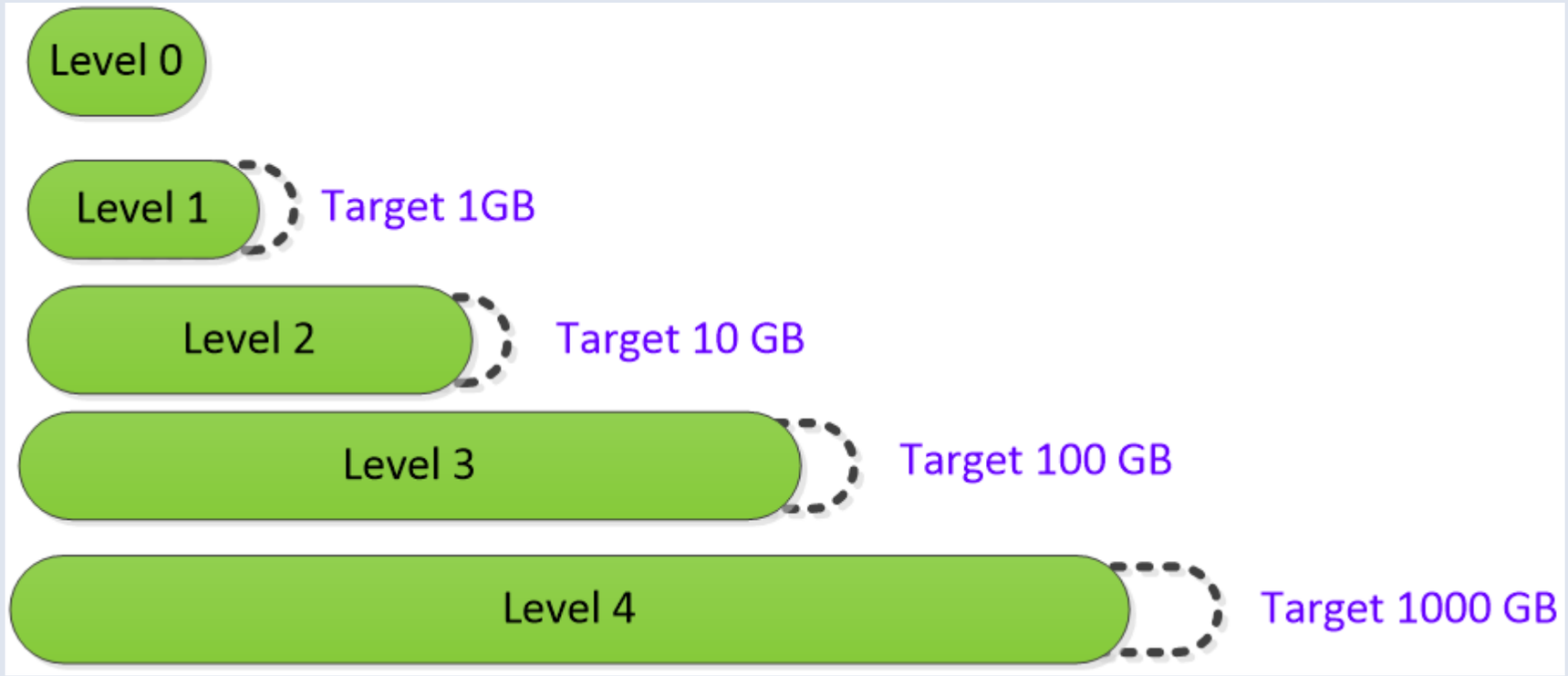
Zero-Filling metadata

key	value	seq id	flag
k1	v1	1234561	W
k2	v2	1234562	W
k3	v3	1234563	W
k4	v4	1234564	W

key	value	seq id	flag
k1	v1	0	W
k2	v2	0	W
k3	v3	0	W
k4	v4	0	W

Seq id is 7 bytes in RocksDB. After compression, “0” uses very little space

LSM Compaction Algorithm -- Level

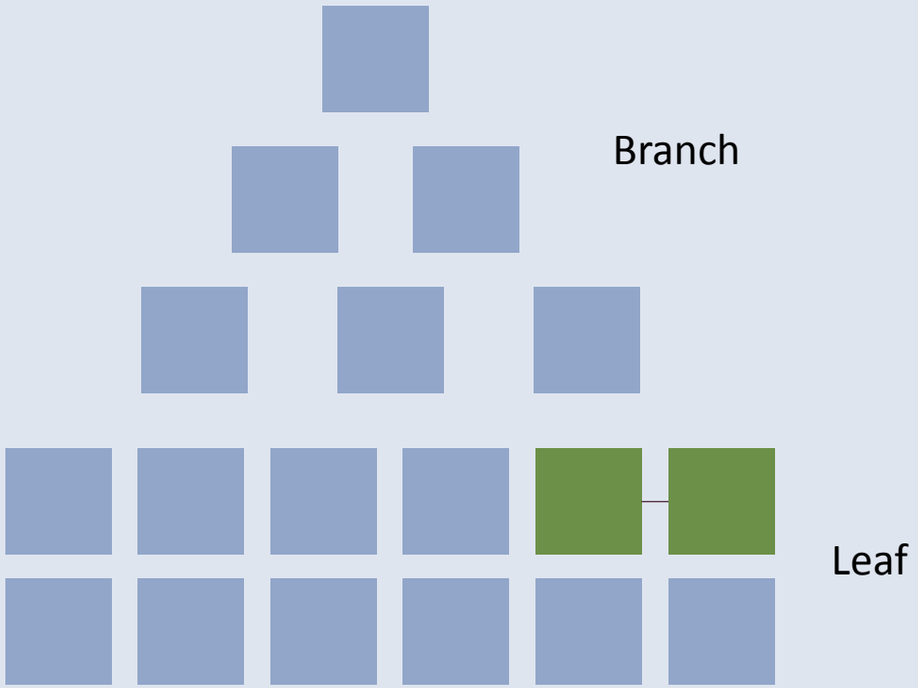


- For each level, data is sorted by key
- Read Amplification: 1 ~ number of levels (depending on cache -- L0~L3 are usually cached)
- Write Amplification: 1 + 1 + fanout * (number of levels – 2) / 2
- Space Amplification: 1.11
  - 11% is much smaller than B+Tree's fragmentation

Read Penalty on LSM

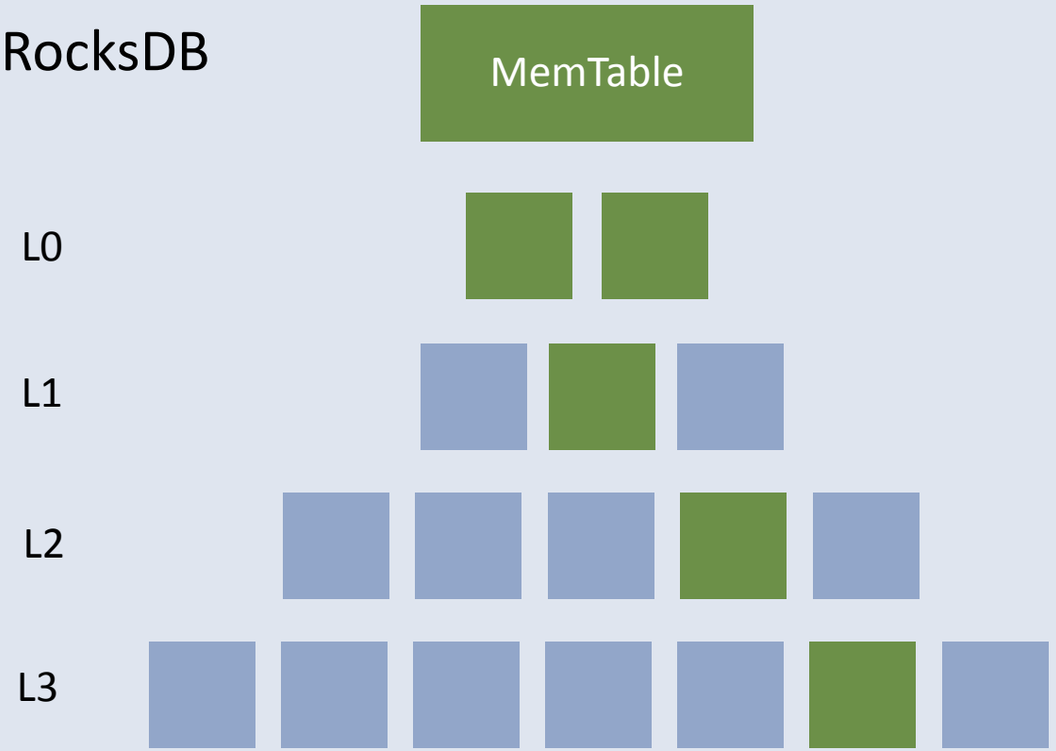
SELECT id1, id2, time FROM t WHERE id1=100 AND id2=100 ORDER BY time DESC LIMIT 1000;
Index on (id1, id2, time)

InnoDB



Range Scan with covering index is
done by just reading leaves sequentially,
and ordering is guaranteed (very efficient)

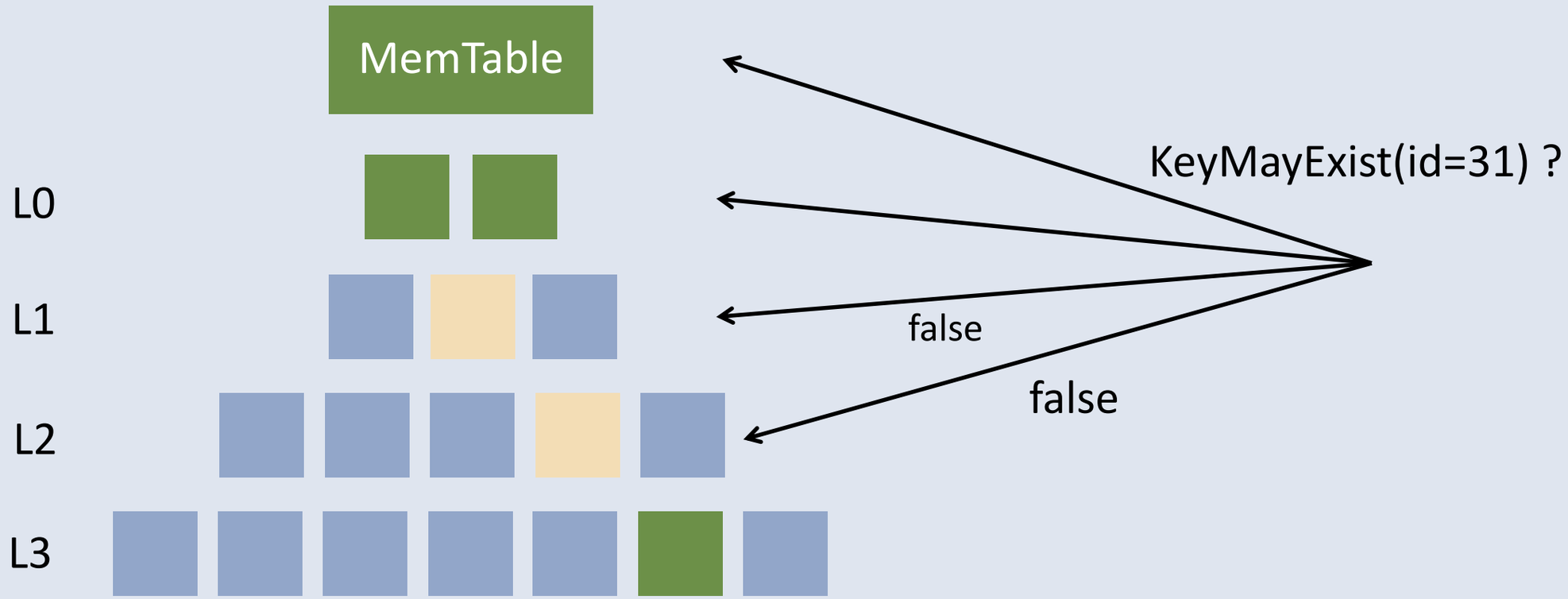
RocksDB



Merge is needed to do range scan with ORDER BY
(L0-L2 are usually cached, but in total it needs
more CPU cycles than InnoDB)

Bloom Filter

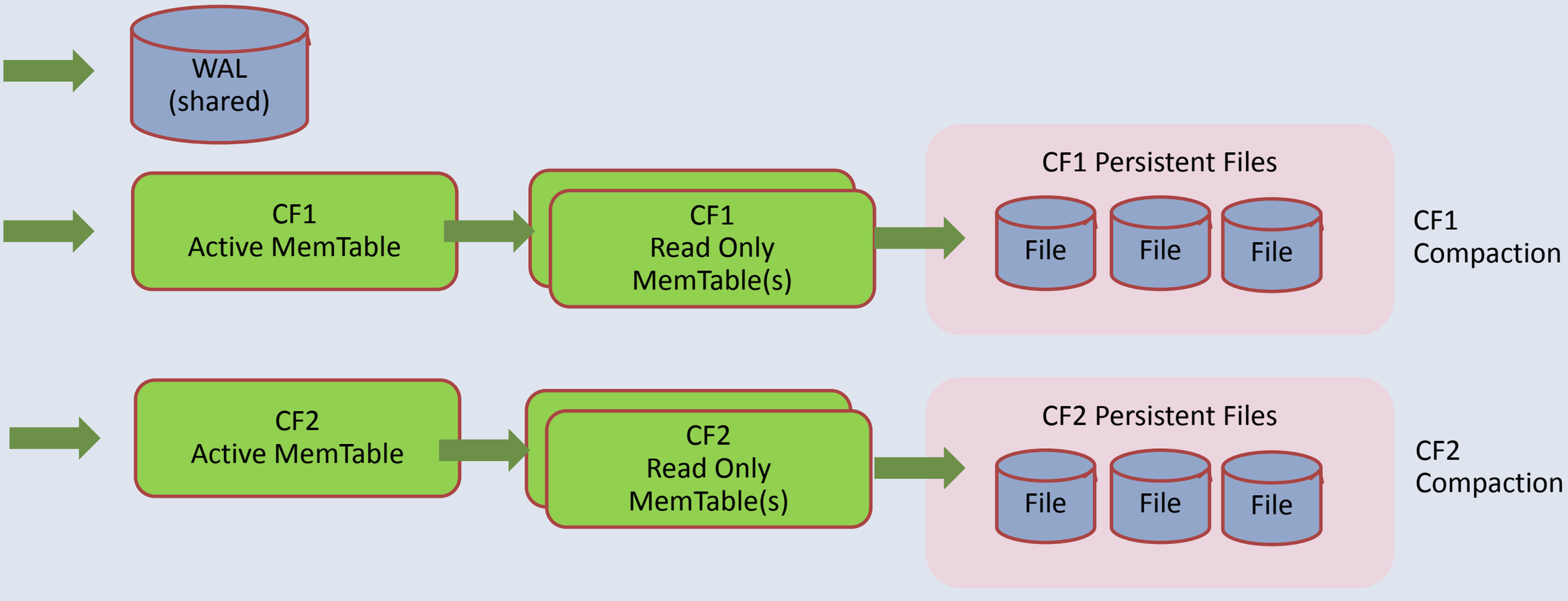
Checking key may exist or not without reading data,
and skipping read i/o if it definitely does not exist



Column Family

Query atomicity across different key spaces.

- Column families:
  - Separate MemTables and SST files
  - Share transactional logs

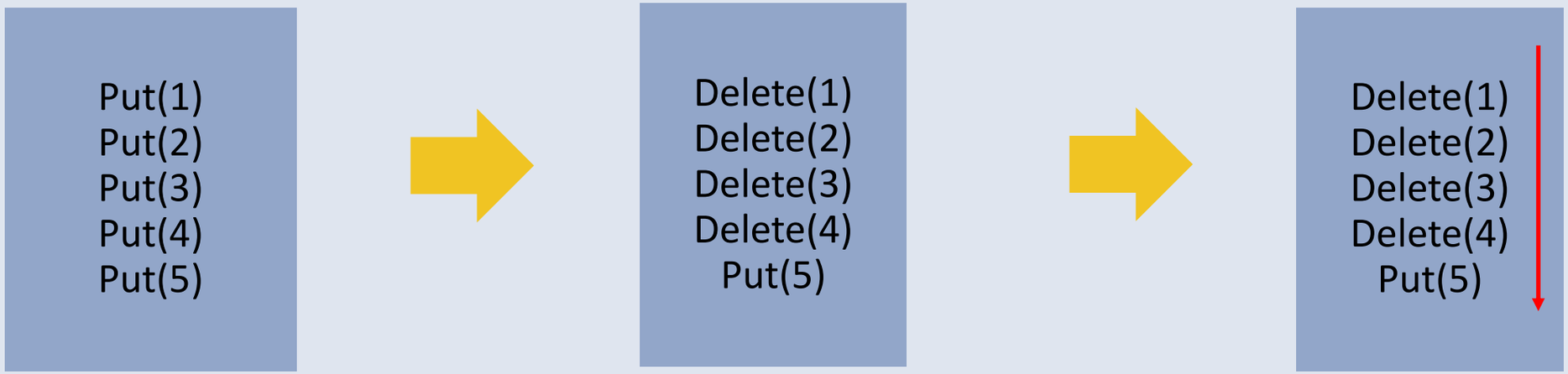


Delete penalty

INSERT INTO t
VALUES (1),(2),(3),(4),(5);

DELETE FROM t WHERE
id <= 4;

SELECT COUNT(*) FROM t;



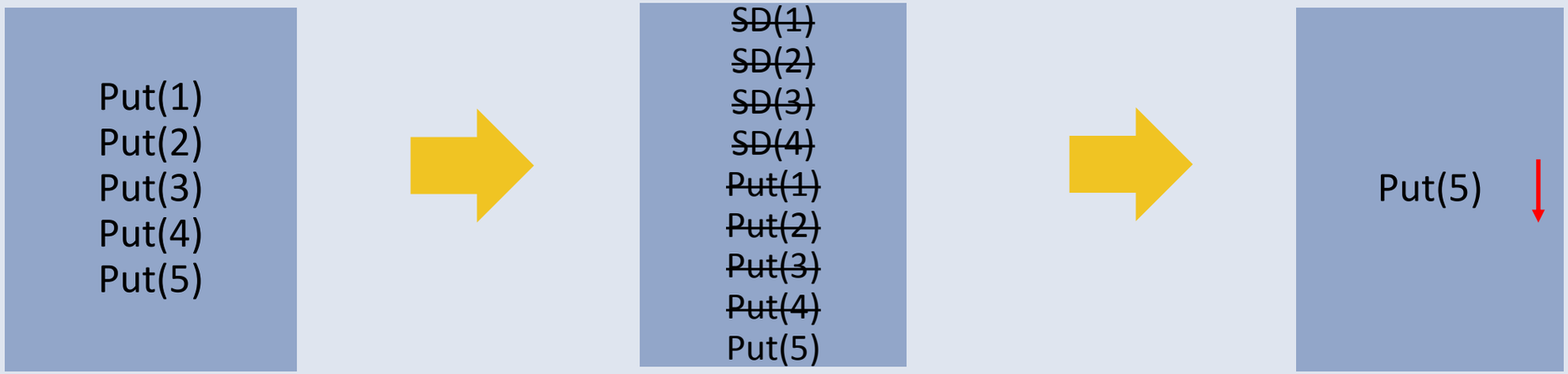
- “Delete” adds a tombstone
  - When reading, ignoring *all* Puts for the same key
- Tombstones can’t disappear until bottom level compaction happens
- Some reads need to scan lots of tombstones => inefficient
  - In this example, reading 5 entries is needed just for getting one row

“SingleDelete” optimization in RocksDB

INSERT INTO t
VALUES (1),(2),(3),(4),(5);

DELETE FROM t WHERE
id <= 4;

SELECT COUNT(*) FROM t;



- If Put for the same key is guaranteed to happen only once, SingleDelete can remove Put and itself
  - Reading just one entry is ok – more efficient than reading 5 entries
- MyRocks uses SingleDelete whenever possible

LSM on Disk

- Main Advantage
  - Lower write penalty
- Main Disadvantage
  - Higher read penalty

- Good fit for write heavy applications

LSM on Flash

- Main Advantages
  - Smaller space with compression
  - Lower write amplification
- Main Disadvantage
  - Higher read penalty

MyRocks (RocksDB storage engine for MySQL)

- Taking both LSM advantages and MySQL features
  - LSM advantage: Smaller space and lower write amplification
  - MySQL features: SQL, Replication, Connectors and many tools
- Fully Open Source
  - https://github.com/facebook/mysql-5.6/

Major feature sets in MyRocks

- Similar feature sets as InnoDB
- Transaction
  - Atomicity
  - MVCC / Non locking consistent reads
  - Read Committed, Repeatable Read (PostgreSQL-style)
  - Crash safe slave and master
- Online Backup
  - Logical backup by mysqldump
  - Binary backup by myrocks_hotbackup

Performance and Efficiency in MyRocks

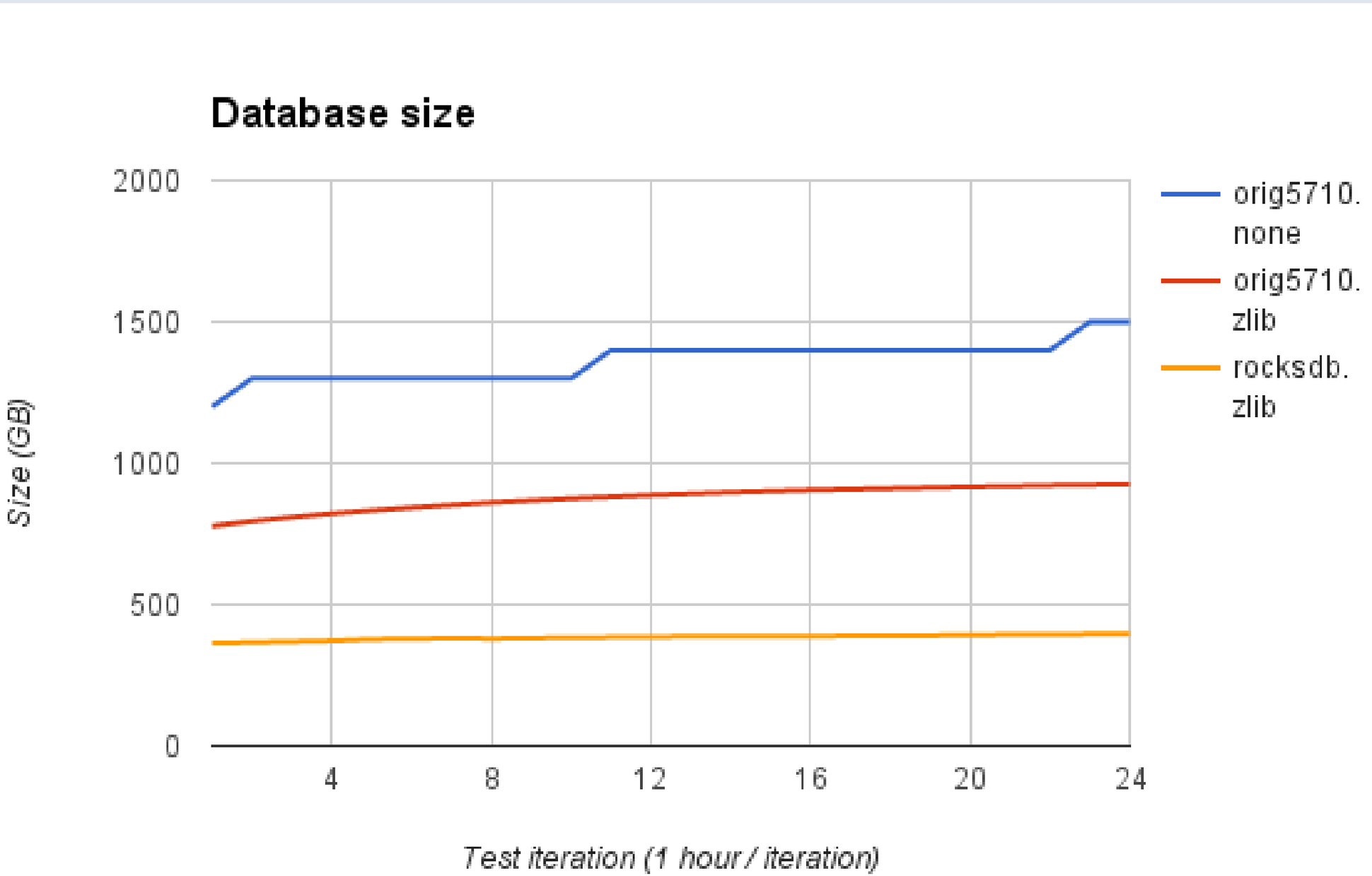
- Much smaller space and write amplification compared to InnoDB
- Faster Replication
- Faster Data Loading
- Reverse order index (Reverse Column Family)
- SingleDelete
- Prefix bloom filter
  - “SELECT … WHERE id=1 and time >= X” => using bloom filter for id
- Mem-comparable keys when using case sensitive collations
- Optimizer statistics without diving into pages

Performance (LinkBench)

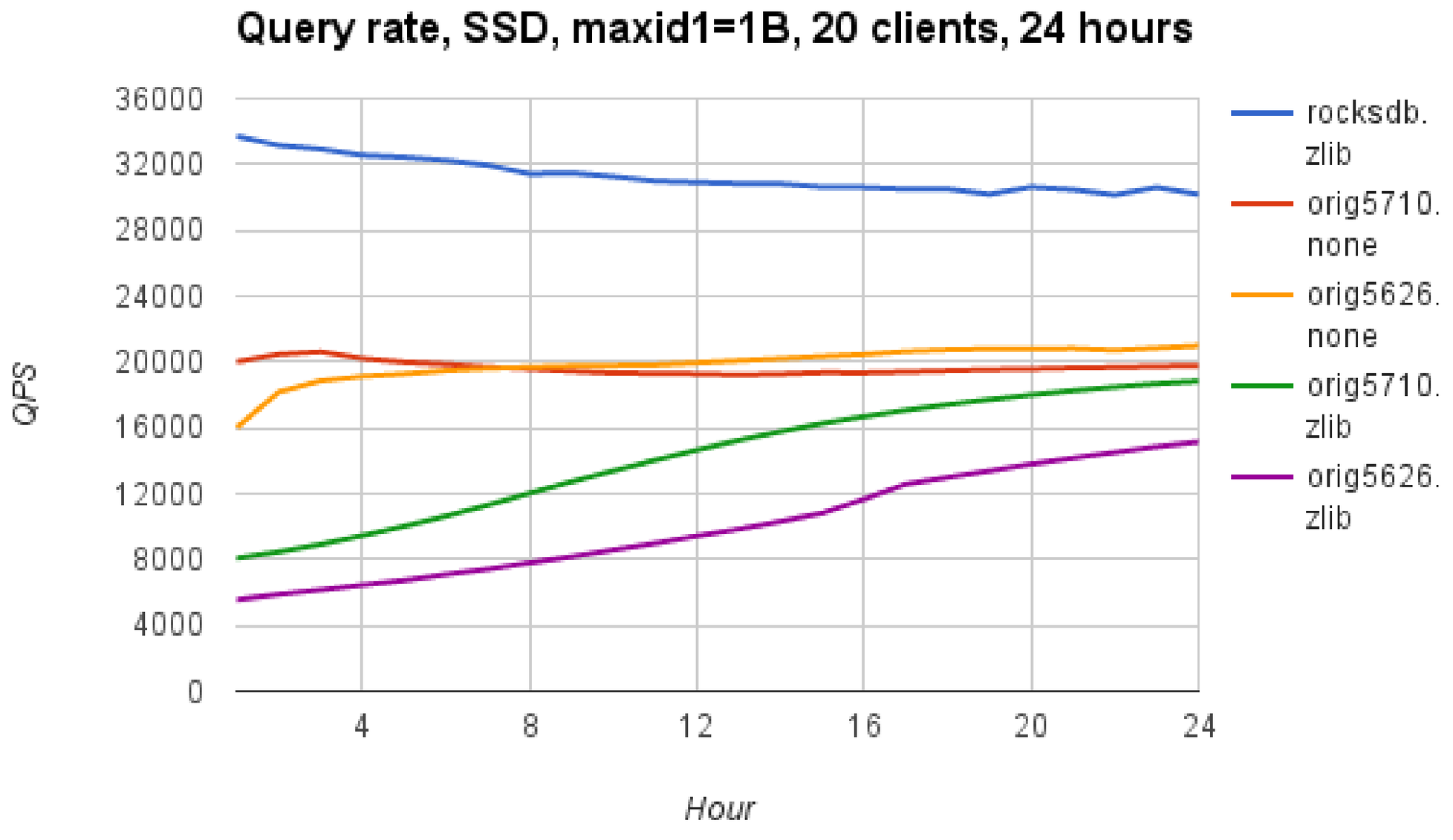
- Space Usage
- QPS
- Flash reads per query
- Flash writes per query
- Data Loading
- Latency
- HDD

- http://smalldatum.blogspot.com/2016/01/myrocks-vs-innodb-with-linkbench-over-7.html

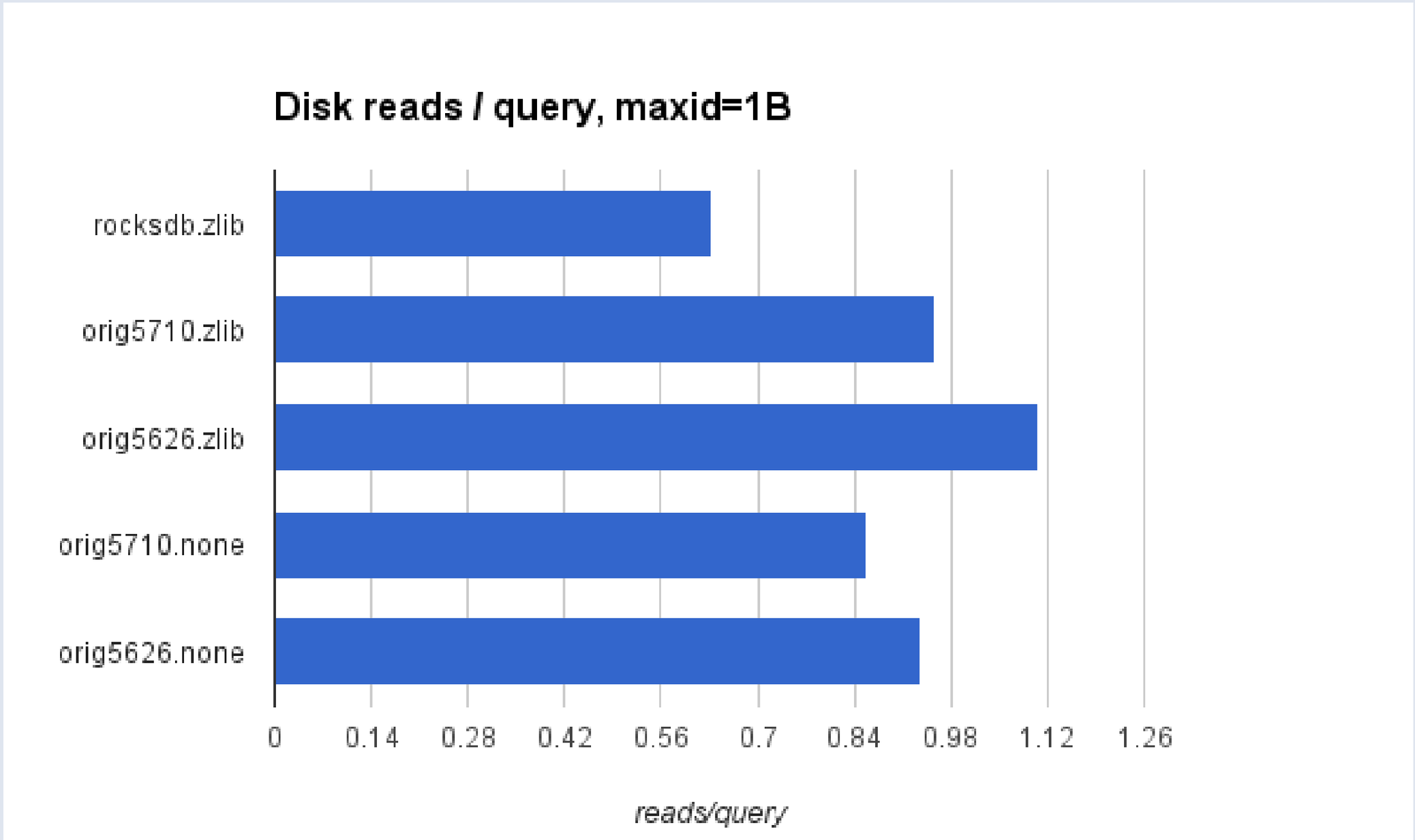
Database Size (Compression)



QPS

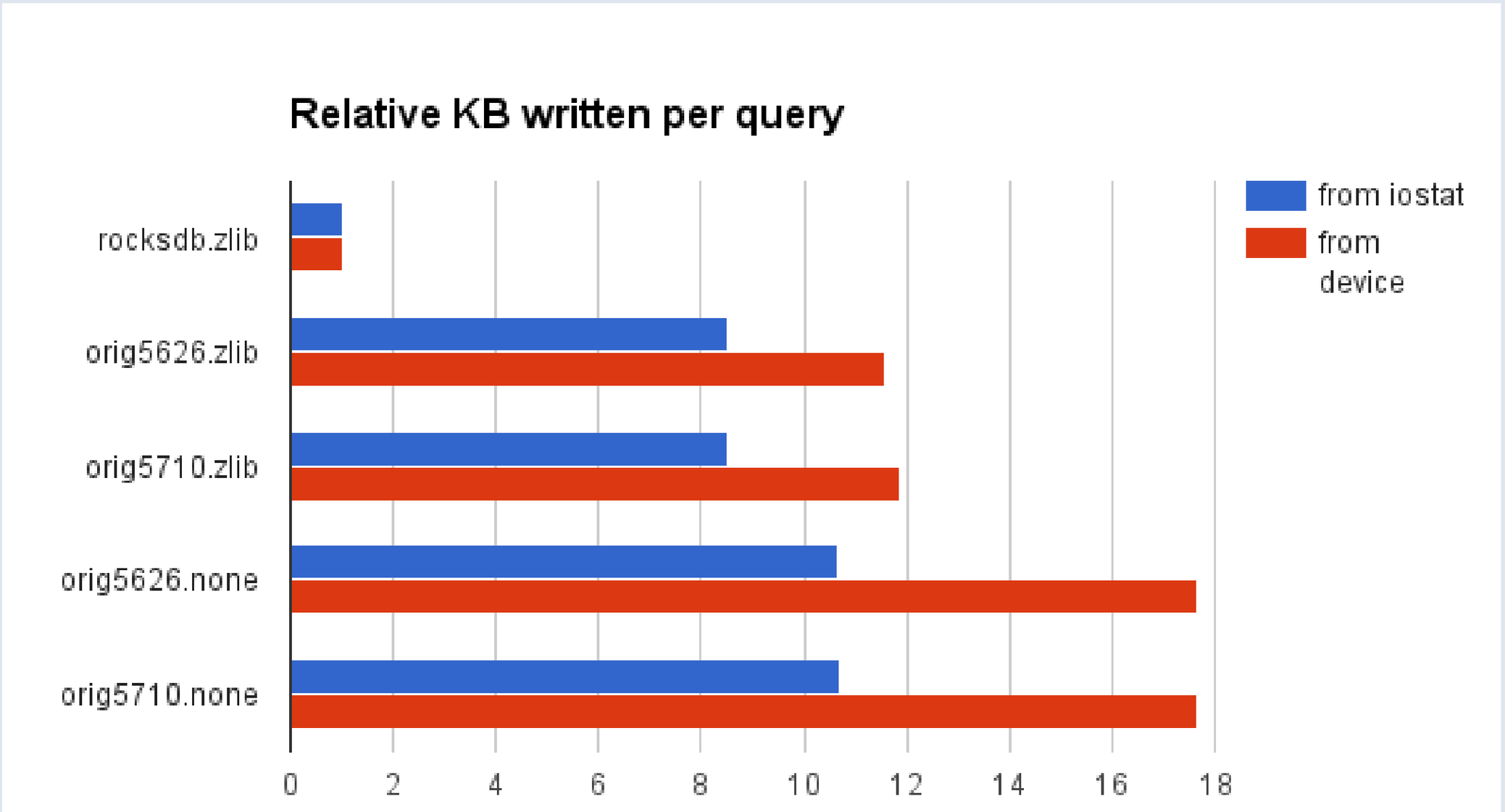

Query rate, SSD, maxid1=1B, 20 clients, 24 hours
36000
32000
28000
24000
20000
16000
12000
8000
4000
0
QPS
4
8
12
16
20
24
Hour
rocksdb.
zlib
orig5710.
none
orig5626.
none
orig5710.
zlib
orig5626.
zlib

Flash Reads per query

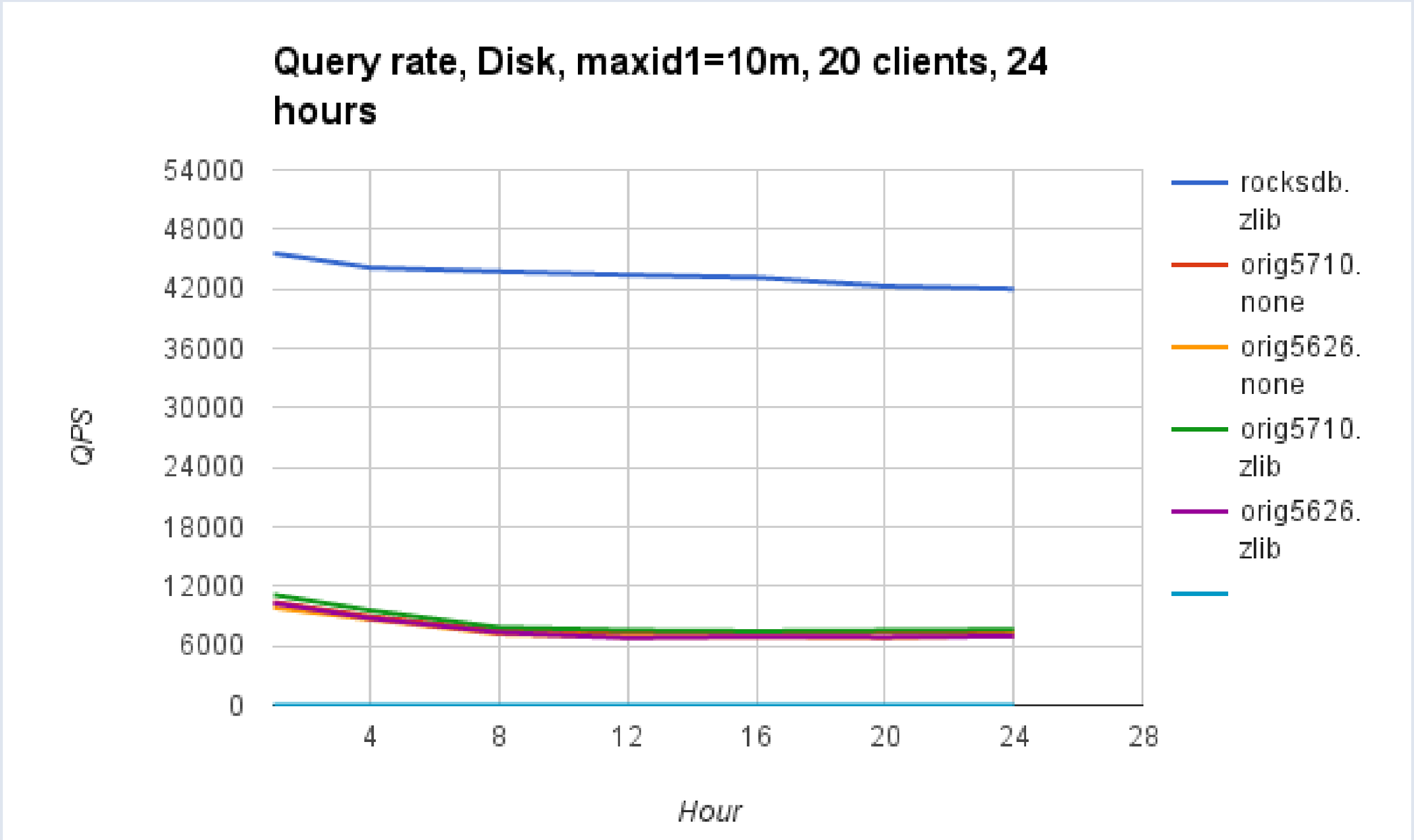


Smaller Space == Better Cache Hit Rate

Flush writes per query



On HDD workloads



Getting Started

- Downloading MySQL with MyRocks support (MariaDB, Percona Server)
- Configuring my.cnf
- Installing MySQL and initializing data directory
- Starting mysqld
- Creating and manipulating some tables
- Shutting down mysqld

- https://github.com/facebook/mysql-5.6/wiki/Getting-Started-with-MyRocks

my.cnf (minimal configuration)

```
[mysqld]
rocksdb
default-storage-engine=rocksdb
skip-innodb
default-tmp-storage-engine=MyISAM
collation-server=latin1_bin (or utf8_bin, binary)

log-bin
binlog-format=ROW
```

- You shouldn’t mix multiple transactional storage engines within the same instance
  - Not transactional across engines, Not tested at all
- Add “allow-multiple-engines” in my.cnf, if you really want to mix InnoDB and MyRocks

Creating tables

```
CREATE TABLE t (
  id INT PRIMARY KEY,
  value1 INT,
  value2 VARCHAR (100),
  INDEX (value1)
) ENGINE=RocksDB COLLATE latin1_bin;
```

- “ENGINE=RocksDB” is a syntax to create RocksDB tables
- Setting “default-storage-engine=RocksDB” in my.cnf is fine too
- It is generally recommended to have a PRIMARY KEY, though MyRocks allows tables without PRIMARY KEYs
- Tables are automatically compressed, without any configuration in DDL. Compression algorithm can be configured via my.cnf

facebook

MyRocks Features

Yoshinori Matsunobu
Facebook
Apr/2017

What is MyRocks

- MySQL on top of RocksDB (RocksDB storage engine)
- RocksDB is an open source LSM key value store forked from LevelDB
- Actively developed
- Used in production at FB
- Mailing List
  - https://groups.google.com/forum/#!forum/myrocks-dev

MyRocks goals

- Smaller space usage
  - 50% compared to compressed InnoDB at FB
- Better write amplification
  - Can use more affordable flash storage
- Fast, and small enough CPU usage with general purpose workloads
  - Large enough data that don’t fit in RAM
  - Point lookup, range scan, full index/table scan
  - Insert, update, delete by point/range, and occasional bulk inserts
  - Same or even smaller CPU usage compared to InnoDB at FB
  - Made it possible to consolidate 2x more instances per machine

MyRocks features

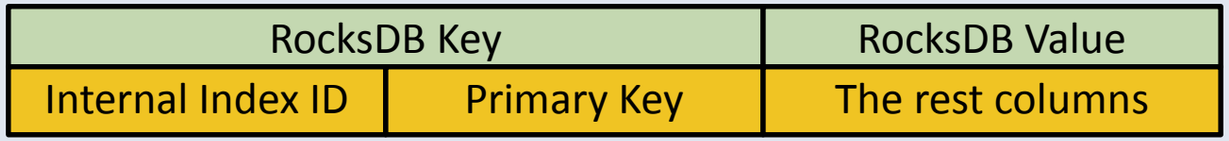
- Clustered Index (same as InnoDB)
- Bloom Filter and Column Family
- Transactions
- Faster data loading, deletes and replication
- Dynamic Options
- TTL
- Crash Safety with XA
- Online logical and binary backup
- Direct IO
- Transactional DDL (upcoming)
- Graceful sst file deletes

MyRocks Data Structure and Schema Design

- Supports Primary Key and Secondary Key
- Primary Key is clustered
  - Similar to InnoDB
  - Primary key lookup can be done by single step
  - Good range scan performance with index-only reads
- “Index comment” specifies Column Family
  - MySQL has a syntax to add a comment for each index
- Fulltext, Foreign, Spatial indexes are not supported
- Tablespace is not supported
- Online DDL has not been supported yet

Internal Index ID

- MyRocks assigns internal 4 byte index id for each index
- Internal index id is used for all MyRocks internal operations, such as reading/writing/updating/deleting rows, dropping indexes
- You don’t have to be aware of index ids, unless debugging internal data structures



Internal Key/Value format

- Primary Key
  - Key:
    - Internal 4 byte index id (auto-assigned)
    - Packed primary key columns
  - Value:
    - Packed other columns
    - Record Checksum (optional)
- Secondary Key
  - Key:
    - Internal 4 byte index id
    - Packed secondary key columns
    - Packed primary key columns (excluding duplicate columns)
  - Value:
    - Record Checksum (optional)

Primary Key:

RocksDB Key		RocksDB Value		Rocks Metadata
Internal Index ID	Primary Key	The rest columns	Checksum	SeqID, Flag

Secondary Key:

RocksDB Key			Rocks Value	Rocks Metadata
Internal Index ID	Secondary Key	Primary Key	Checksum	SeqID, Flag

Secondary Key structure is called Extended Key – containing both Secondary Key and Primary Key

Example

- CREATE TABLE t1 (id INT PRIMARY KEY, c1 INT, c2 INT, INDEX i1 (c1));
- INSERT INTO t1 VALUES (1, 10, 100), (2, 20, 200), (3, 30, 300),(4,40, 400),(5, 50, 500)
- Primary key index id was 256, Secondary key (i1) index id was 257

RocksDB Key		Value
Internal Index id	PK	Value
256	1	10,100
256	2	20,200
256	3	30,300
256	4	40,400
256	5	50,500

RocksDB Key			Value
Internal Index id	SK	PK	Value
257	10	1	Null
257	20	2	Null
257	30	3	Null
257	40	4	Null
257	50	5	Null

Space overhead by Internal Index ID is very small, thanks to Prefix Key Encoding feature of RocksDB

Tables without primary key

- Hidden Primary Key
  - Key:
    - Internal 4 byte index id
    - Internal 8 byte auto generated id (internal primary key)
    - Hidden keys are not visible from applications
  - Value:
    - All columns (packed format)
    - Record Checksum (optional)
- Secondary Key
  - Same as tables with primary key

Hidden Primary Key:

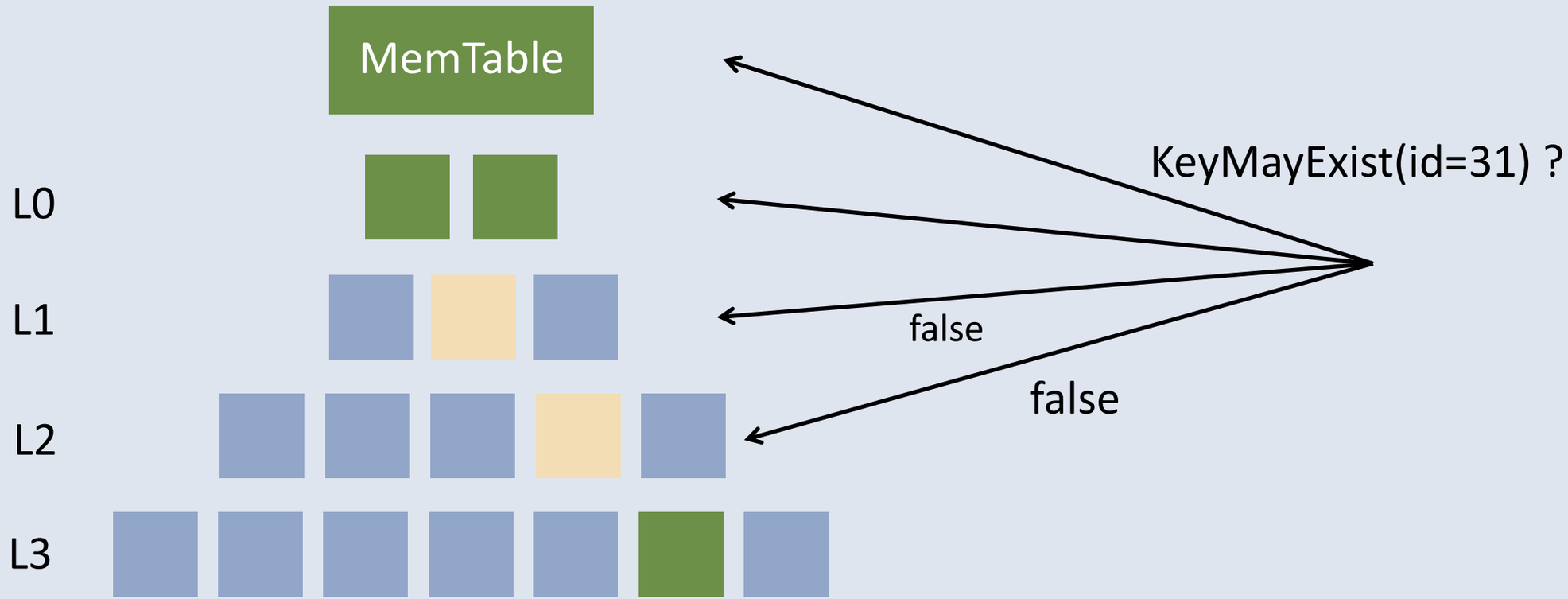
RocksDB Key		RocksDB Value		Rocks Metadata
Internal Index ID	Hidden PK	All columns	Checksum	SeqID, Flag

Secondary Key:

RocksDB Key			Rocks Value	Rocks Metadata
Internal Index ID	Secondary Key	Hidden PK	Checksum	SeqID, Flag

Bloom Filter

- Checking key may exist or not without reading data,
and skipping read i/o if it definitely does not exist
- Bloom filter length needs defined
  (must be equal to or less than equal condition length)

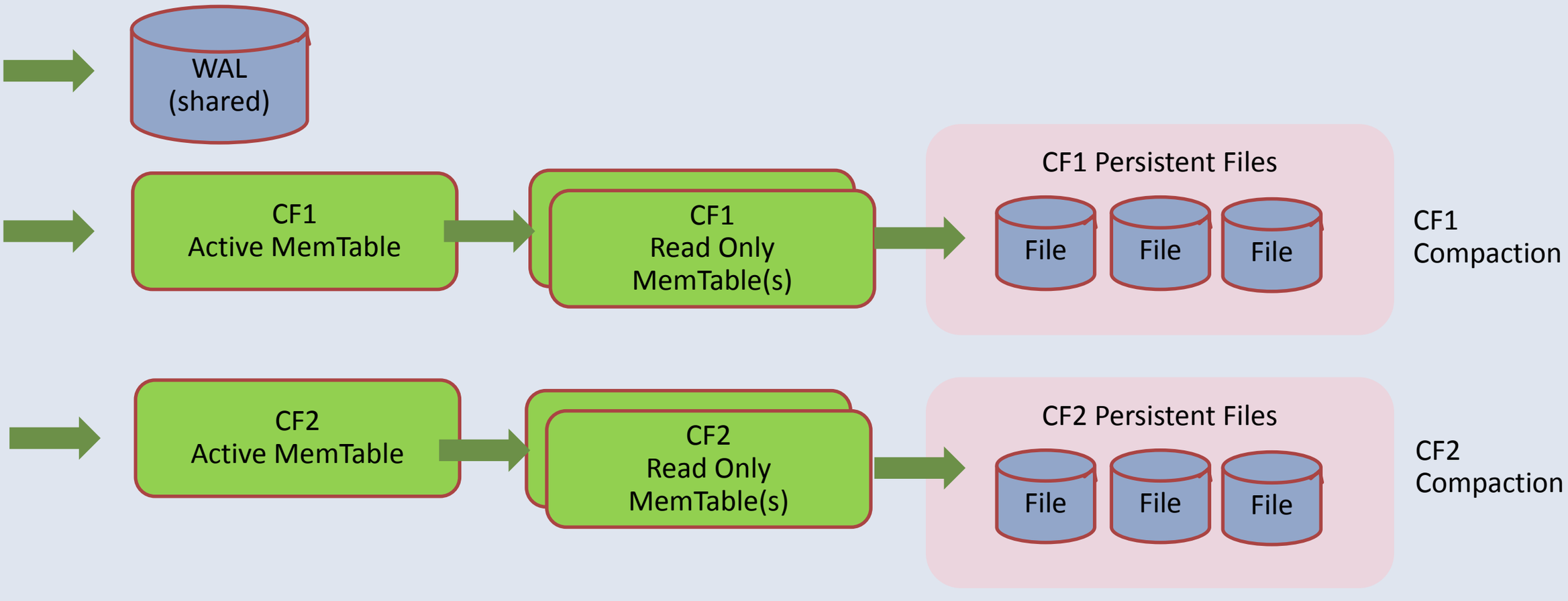


Configuration Example: cf_link_pk={prefix_extractor=capped:20}

Column Family

Query atomicity across different key spaces.

- Column families:
  - Separate MemTables and SST files
  - Share transactional logs



Index and Column Family

- Column Family and MyRocks Index mapping is 1:N
  - Each MyRocks index belongs to one Column Family
  - Multiple indexes can belong to the same Column Family
  - If not specified in DDL, the index belongs to “default” Column Family
- Different types of indexes can be allocated to different Column Families
  - Do not create too many Column Families
    - ~20 would be good enough
  - INDEX COMMENT specifies associated column family

Column Family specific configurations

- Index ordering
  - Reverse order CF for descending scan-mostly indexes
- Bloom Filter
- And many more (block size, compression, MemTable size, etc)

- We classified indexes on our main database for similar CF types at Facebook
  - Node
  - Link
  - Link reverse order
  - Others

Reverse column families

- RocksDB is great at scanning forward
- But ORDER BY DESC queries are slow
- Reverse column families make descending scan a lot faster

id1	id2	id3
100	200	4
		3
		2
		1

Prefix Key Encoding

id1	id2	id3
100	200	1
100	200	2
100	200	3
100	200	4

id1	id2	id3
100	200	1
		2
		3
		4

Table Definition Example

```
CREATE TABLE `linktable` (
    `id1` bigint unsigned,
    `id1_type` int unsigned,
    `id2` bigint unsigned,
    `id2_type` int unsigned,
    `link_type` bigint unsigned,
    `visibility` tinyint NOT NULL,
    `data` varchar NOT NULL,
    `time` bigint unsigned NOT NULL,
    `version` int unsigned NOT NULL,
     PRIMARY KEY (link_type, `id1`,`id2`) COMMENT 'cf_link_pk',
     KEY `id1_type` (`id1`,`link_type`,`visibility`,`time`,`version`,`data`) COMMENT 'rev:cf_link_id1_type'
   ) ENGINE=RocksDB DEFAULT COLLATE=latin1_bin;
```

- Index Comment specifies Column Family name
  - If the column family does not exist, RocksDB automatically creates it
- “rev:” is a syntax to create reverse order column family
- Column Family statistics can be viewed via “SHOW ENGINE ROCKSDB STATUS\G”

Partitions and column family

- MyRocks supports MySQL partitioning
  - All of MyRocks data files are stored under $datadir/.rocksdb
  - But each partition belongs to different index and stored logically separated
- RANGE and LIST partitions can have different column families per partition

```
CREATE TABLE t1 (
  c1 INT,
  c2 INT,
  name VARCHAR(25) NOT NULL,
  event DATE,
  PRIMARY KEY (`c1`, `c2`) COMMENT 'p0_cfname=foo;p1_cfname=my_custom_cf;p2_cfname=baz'
) ENGINE=ROCKSDB
PARTITION BY LIST(c1) (
  PARTITION p0 VALUES IN (1, 4, 7),
  PARTITION p1 VALUES IN (2, 5, 8),
  PARTITION p2 VALUES IN (3, 6, 9)
);
```

Transactions

- General purpose transaction supports are implemented
  - BEGIN; COMMIT; ROLLBACK
    - SAVEPOINT (modifying rows) is not supported
  - START TRANSACTION WITH CONSISTENT SNAPSHOT;
    - Read Committed and Repeatable Read isolation levels
    - SELECT FOR UPDATE and SELECT LOCK IN SHARE MODE
    - Non locking reads do not hold read locks
    - Controllable durability (2pc with binlog, no 2pc, sync/async per commit)
- Gap Lock is not implemented
  - Important Limitation: Row Based Binary Logging (RBR) must be used on master

Faster data loading

- General write path in MyRocks (RocksDB):
  - Written to WAL/binlog, written to MemTable -> flushed to L0 -> compacted to L1..Lmax
  - X bytes to binlog => will result in X * N (typically 5~20) bytes
- Fast and write efficient data loading in MyRocks:
  - Directly creating data files in the bottom-most RocksDB levels, avoiding MemTable writes and compactions
  - Primary Key
    - SET SESSION rocksdb_bulk_load=1; then bulk insert or LOAD DATA
    - Source data must be pre-sorted by primary keys
      - mysqldump --order-by-primary | mysql --init-command="set rocksdb_bulk_load=1"
  - Secondary Key
    - You need to add secondary keys after data loading, to do bulk loading into secondary keys
    - ALTER TABLE … ADD INDEX or CREATE INDEX automatically uses bulk loading

Faster deletes

- MyRocks provides session variables to make deletes faster, by sacrificing some transaction semantics
  - Blind deletes by primary key (rocksdb-blind-delete-primary-key)
    - Skipping to check if rows exist or not, and just mark deletes
    - Effective statements: DELETE FROM t WHERE id = 1 (or id IN (1, 2, 3..))
    - Not effective if tables have secondary keys
  - Implicit commits on range deletes (rocksdb-commit-in-the-middle)
    - Improving concurrency because locks are released earlier
    - Rollback overhead can be significantly smaller
    - If you terminate statements, rows may be partially committed
  - Skip holding row locks (rocksdb-master-skip-tx-api)

Skipping using RocksDB's transaction API

- Fast Path for modifying rows
  - MyRocks uses RocksDB's transaction API on regular operations
  - RocksDB has non-transactional API too, that only provides atomicity
    - "WriteBatch" API
  - Not holding any row locks. It's users' responsibility to confirm there is no concurrent access to the same rows
- Can be safely used on slaves, if not modified outside of replication
  - SET global rpl_skip_tx_api=1; STOP/START SLAVE SQL_THREAD;
- We provide an option for master too
  - SET SESSION rocksdb_master_skip_tx_api = 1;

Faster Replication

- Skipping holding row locks (skipping using transaction api)
  - rpl_skip_tx_api variable, as described in the previous slide
- Read Free Replication

Read Free Replication

- Read Free Replication is a feature to skip random reads on slaves
  - Skipping unique constraint checking on INSERT
  - Skipping row checking on UPDATE/DELETE
- RBR and Append Only database (LSM, Fractal) make it possible
- TokuDB implemented this feature. We implemented in MyRocks based on its idea and codebase
- Unique key check can be skipped by “SET SESSION unique_checks=0”

Avoiding look-up rows

- Update and Delete path on Slave, with RBR, without Read Free Repl:
  - Relay log has “Before Row Image” (BI)
  - Getting primary key from BI
  - Point lookup by the primary key (random read)
  - If not exist, skip or raise error, depending on slave_exec_mode. If exists, update by “After Row Image (AI)” or delete
- Read Free Repl:
  - Blind Delete/SingleDelete (BI keys)
  - Blind Put (AI)
- Eliminates random read overhead on update/delete
- WARNING: If you modify outside of repl and if the table has sec keys, indexes may be inconsistent
- Can be configured by rocksdb_read_free_rpl_tables parameter
  - Configuring target tables by regex format. “.*” will activate for all tables

Dynamic Options

- Most DB and CF options can now be changed online
- Example Command
  - SET @@global.rocksdb_update_cf_options = 'cf1={write_buffer_size=8m;target_file_size_base=2m};cf2={write_buffer_size=16m;max_bytes_for_level_multiplier=8};cf3={target_file_size_base=4m};';
- Can be viewed new CF setting from information_schema
  - SELECT * FROM ROCKSDB_CF_OPTIONS WHERE CF_NAME='cf1' AND OPTION_TYPE='WRITE_BUFFER_SIZE';
  - SELECT * FROM ROCKSDB_CF_OPTIONS WHERE CF_NAME='cf1' AND OPTION_TYPE='TARGET_FILE_SIZE_BAS';
- Values are not persisted. You need to update my.cnf to get persisted

TTL

- Time To Live – deleting rows if reaching specified timestamp
- Matching HBase TTL feature
- More efficient than scheduled deletes
  - After reaching specified timestamp, rows are purged during compactions. No tombstone is generated (that's why very efficient than batch deletes!)
  - Rows are not visible when reaching specified timestamp, even before compactions
- Definition Example

```
Keeping 10 days:
CREATE TABLE t1 (
 a INT PRIMARY KEY (a)
) ENGINE=rocksdb
COMMENT='ttl_duration=864000;';
```

How crash recovery works

- All modifications are written to WAL (transaction log files) at commit
- Flushed to kernel buffer, but fsync() is not called at commit
  - On mysqld process down
    - All committed transactions are written to WAL file / kernel buffer. No data loss
  - On OS/machine down
    - Some of the committed transactions may be lost
    - Need to catch up missing transactions from master
- Binlog and Replication State are also written to WAL at commit, in atomic

Crash Safe MyRocks on slaves

Master_binlog_file= mysql-bin.000100
Master_binlog_pos = 1000
Put (key=1111, value=100)

WAL entry 1

Master_binlog_file= mysql-bin.000100
Master_binlog_pos = 2000
Put (key=1112, value=200)

WAL entry 2

Master_binlog_file= mysql-bin.000100
Master_binlog_pos = 3000
Put (key=1113, value=300)

WAL entry 3

Slave Instance

- WAL is append only, and it has internal checksum. On crash recovery, if detecting any broken WAL entry, RocksDB discards the broken enrty and all WAL entries after that. So state after crash recovery is consistent
- Even if WAL entry 3 is lost, after crash recovery, replication state becomes “master_binlog_pos=2000” so it can fetch binlog events for WAL entry 3 from master

Crash safety settings in depth

	sync-binlog	rocksdb-flush-log-at-trx-commit	rocksdb-enable-2pc	rocksdb-wal-recovery-mode
No data loss on unplanned machine reboot	1 (default)	1 (default)	1 (default)	1 (default)
No data loss on mysqld crash & recovery	0	2	1	1
No data loss if always failover	0	2	0	2

- Need Loss-Less Semisync to prevent data loss on failover
- sync-binlog: fsync() at every N commits
- rocksdb-flush-log-at-trx-commit: 1: fsync() at each commit, 0/2: fsync per second
- rocksdb-enable-2pc: synching binary logs and rocksdb WAL at recovery
- rocksdb-wal-recovery-mode: trimming corrupted WAL at recovery if needed, if not 1

- If you do not run master failover on machine reboot, you should make fully durable configurations (default).
- If you always run master failover on machine reboot but may restart on mysqld crash (e.g. mysqld_safe), you do not need fsync on commit but still need 2pc and wal protection
- If you always run master failover, you can run without durability and 2pc

Backup

- Logical Backup by mysqldump
- Consistent Snapshot and long running transactions
- Binary Backup by myrocks_hotbackup

Logical Backup by mysqldump

- Facebook mysql-5.6 extended mysqldump to support MyRocks
- mysqldump --single-transaction
- Can take either InnoDB or MyRocks consistent snapshot, not both
  - Checks default-storage-engine
  - default-storage-engine=RocksDB => taking consistent MyRocks dump
  - default-storage-engine=InnoDB => taking consistent InnoDB dump

How consistent backup works

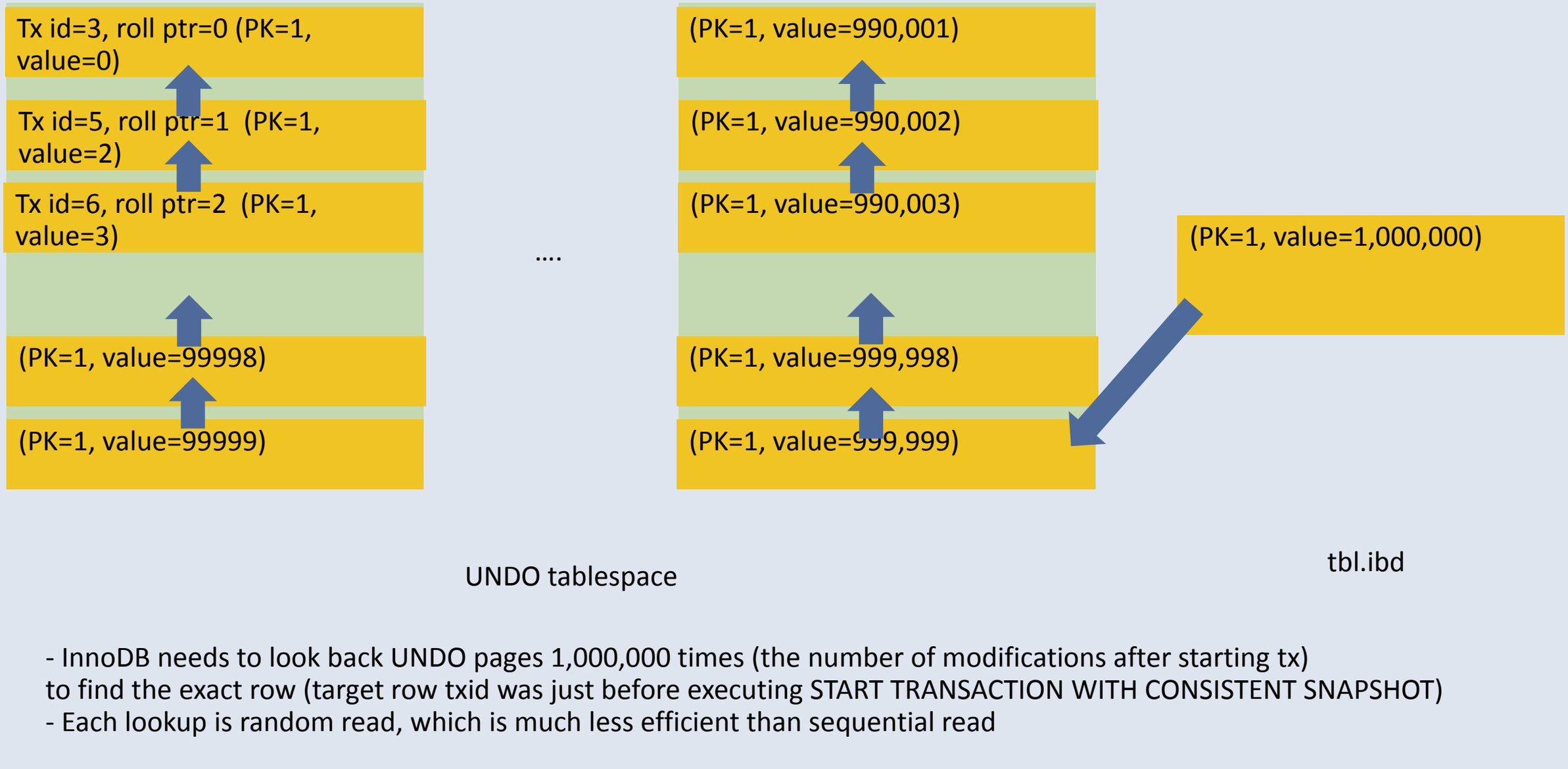
- SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
- START TRANSACTION WITH CONSISTENT ROCKSDB SNAPSHOT
  - New syntax (Facebook patch)
  - Get an internal binlog lock so that nobody can write to binlog
    - Much lighter than FLUSH TABLES WITH READ LOCK
  - Create RocksDB snapshot
  - Get current binlog position and GTID
  - Unlock internal binlog lock
- Dump all tables (SELECT …)
  - Repeatable Read guarantees all dump data is based on the acquired snapshot

Snapshot and long running transactions

- Logical backup holds snapshots for a very long time. This is bad from performance point of view
- The overhead is high in MyRocks too, but not as high as InnoDB

Backup Client	Applications	
START TRANSACTION WITH CONSISTENT SNAPSHOT;		
		PK=1, value=0
	UPDATE t SET value=value+1 WHERE PK=1;	PK=1, value=1
	UPDATE t SET value=value+1 WHERE PK=1;	PK=1, value=2
	UPDATE t SET value=value+1 WHERE PK=1;	PK=1, value=3
	….. (modified 1,000,000 times)	
		PK=1, value=1,000,000
SELECT * FROM t; => returning value=0		

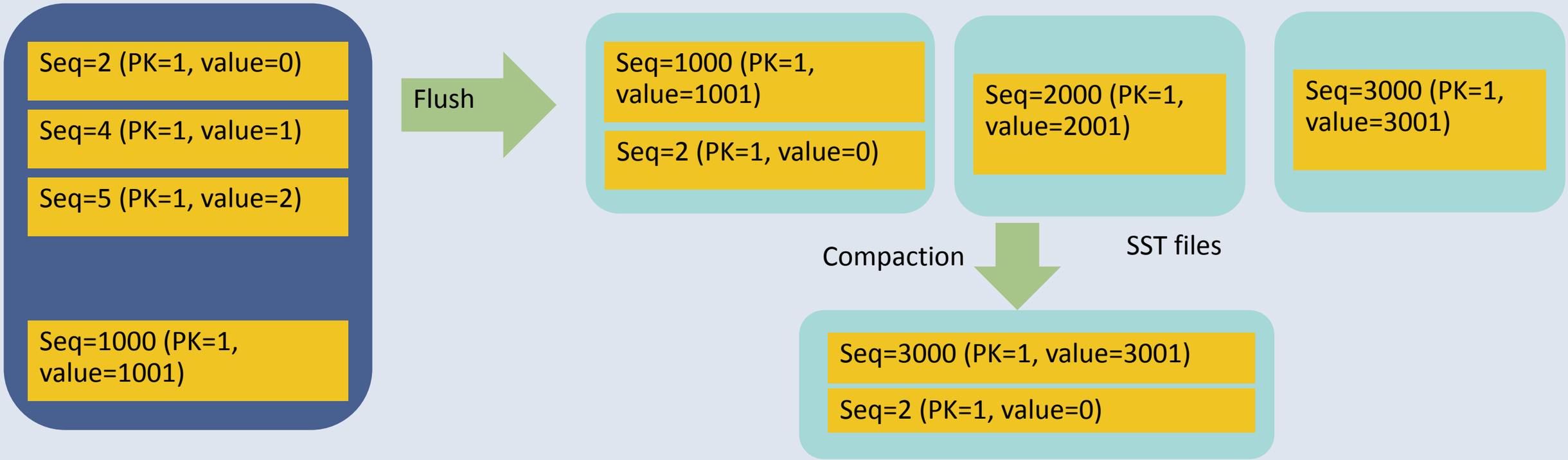
How snapshot works in InnoDB



- InnoDB needs to look back UNDO pages 1,000,000 times (the number of modifications after starting tx) to find the exact row (target row txid was just before executing START TRANSACTION WITH CONSISTENT SNAPSHOT)
- Each lookup is random read, which is much less efficient than sequential read

How snapshot works in MyRocks

START TRANSACTION WITH CONSISTENT ROCKSDB SNAPSHOT
Sequence Id = 3



All intermediate rows were removed during flush/compaction
because RocksDB could know they were definitely not needed

This makes total lookup times much fewer than 1,000,000
But it's still a good practice to keep transaction duration short enough

Online Binary Backup

- MyRocks provides online binary backup solutions and tools
- Online binary backup is useful for:
  - Creating new slave instances much faster than logical backup
  - Restoring from backup much faster than logical backup
  - Currently only full binary backup is possible. Partial or incremental binary backup has not been supported yet

Direct IO

- MyRocks has not fully supported Direct IO yet
- Storage <==> OS page cache <==> RocksDB's LRU cache
- Relies on Linux kernel a lot
  - Linux kernel 4.6~ works efficiently (saw ~10% better qps with iibench than 4.0)
  - Older kernels may hit VM allocation stalls and use higher %sys CPU
- Direct IO will reduce kernel dependency and will resolve page cache allocation issues
- Upcoming configuration options
  - Compressed block cache size
  - Read ahead (prefetch) size for full index/table scan

Transactional DDL (upcoming)

- Updating *.frm is not transactional in MySQL
  - Table definitions can be inconsistent on crash during DDL
- MySQL 8.0 started supporting transactional ddl (no more frm)
  - All table metadata are stored in storage engine's data dictionary
  - MyRocks is going to support transactional ddl too
  - MyRocks is going to check consistency between frm and data dictionary at instance startup
  - Follow https://github.com/facebook/mysql-5.6/issues/609 for details

Graceful sst file deletes on Flash

- Compactions read X number of files, create Y number of files then delete source files
  - Typical config is 32MB * 25 files per compaction (800MB in total)
- Compactions can run in parallel
- Deleting many files without any interval causes problems on Flash
  - TRIM stalls
- rocksdb-sst-mgr-rate-bytes-per-sec can control deletion speed
  - By default is 64MB/s
  - Background deletion is throttled. Compactions are not blocked

Summary -- MyRocks features

- Clustered Index (same as InnoDB)
- Bloom Filter and Column Family
- Transactions
- Faster data loading, deletes and replication
- Dynamic Options
- TTL
- Crash Safety with XA
- Online logical and binary backup
- Direct IO
- Transactional DDL (upcoming)
- Graceful sst file deletes

facebook

facebook

MyRocks in production at Facebook

Yoshinori Matsunobu
Facebook
Apr/2017

Agenda

- MySQL at Facebook
- MyRocks overview
- Production Deployment
  - Defining Schema
  - Fast data migration from InnoDB to MyRocks
  - Data verification
  - Creating MyRocks copies (replicas)
  - Crash safety settings
  - Preventing Stalls
  - Monitoring

“Main MySQL Database” at Facebook

- Storing Social Graph
- Massively Sharded
- Petabytes scale
- Low latency
- Automated Operations
- Pure Flash Storage

What is MyRocks

- MySQL on top of RocksDB (RocksDB storage engine)
- RocksDB is an open source LSM key value store forked from LevelDB
- Actively developed
- Used in production at FB
- Mailing List
  - https://groups.google.com/forum/#!forum/myrocks-dev

MyRocks goals

- Smaller space usage
  - 50% compared to compressed InnoDB at FB
- Better write amplification
  - Can use more affordable flash storage
- Fast, and small enough CPU usage with general purpose workloads
  - Large enough data that don’t fit in RAM
  - Point lookup, range scan, full index/table scan
  - Insert, update, delete by point/range, and occasional bulk inserts
  - Same or even smaller CPU usage compared to InnoDB at FB
  - Make it possible to consolidate 2x more instances per machine

MyRocks features

- Clustered Index (same as InnoDB)
- Bloom Filter and Column Family
- Transactions
- Faster data loading, deletes and replication
- Dynamic Options
- TTL
- Crash Safety with XA
- Online logical and binary backup
- More concurrent writes (upcoming)
- Direct IO
- Graceful sst file deletes

MyRocks migration -- Technical Challenges

- Initial Migration
  - Defining MyRocks-optimized Schema
  - Finding out optimal Column Family options
  - Creating MyRocks instances without downtime
  - Loading into MyRocks tables within reasonable time
  - Verifying data consistency between InnoDB and MyRocks
- Continuous Monitoring
  - Resource Usage like space, iops, cpu and memory
  - Query plan outliers
  - Stalls and crashes

MyRocks migration -- Technical Challenges (2)

- When running MyRocks on master
  - RBR
  - Removing queries relying on InnoDB Gap Lock
  - Robust XA support (binlog and RocksDB)

MyRocks tables with optimal CF options

- Index ordering
  - Reverse order CF for descending scan-mostly indexes
- Bloom Filter
  - Bloom filter length should be equal to or less than equal condition length
- Most MyRocks configuration options are per column family
  - Each MyRocks index belongs to a specified CF
  - We classified indexes on our main database for similar CF types
    - Node
    - Link
    - Link reverse order
    - Others

Creating first MyRocks instance without downtime

- Picking one of the InnoDB slave instances, then starting logical dump and restore
  - Stopping one slave does not affect services
- Our MySQL replication environment
  - Single Master and multiple Slaves
  - Slaves are distributed across long distant regions
  - We use Loss Less Semi-Synchronous Replication for local mysqlbinlog (tailing binlogs), Asynchronous Replication for cross-region slaves
  - We have multiple databases within the instance (Multi-Threaded Slave helps to reduce lag)

Faster data loading

- General write path in MyRocks (RocksDB):
  - Written to WAL/binlog, written to MemTable -> flushed to L0 -> compacted to L1..Lmax
  - X bytes to binlog => will result in X * N (typically 5~20) bytes
- Fast and write efficient data loading in MyRocks:
  - Directly creating data files in the bottom-most RocksDB levels, avoiding MemTable writes and compactions
  - Primary Key
    - SET SESSION rocksdb_bulk_load=1; then bulk insert or LOAD DATA
    - Source data must be pre-sorted by primary keys
      - mysqldump --order-by-primary | mysql --init-command="set rocksdb_bulk_load=1"
  - Secondary Key
    - You need to add secondary keys after data loading, to do bulk loading into secondary keys
    - ALTER TABLE … ADD INDEX or CREATE INDEX automatically uses bulk loading

Data migration steps

- Dst) Create table … ENGINE=ROCKSDB; (creating MyRocks tables with proper column families)
- Dst) ALTER TABLE DROP INDEX; (dropping secondary keys)
- Src) STOP SLAVE;
- mysqldump –host=innodb | mysql –host=myrocks –init-command=“set sql_log_bin=0; set rocksdb_bulk_load=1”
- Dst) ALTER TABLE ADD INDEX; (adding secondary keys)
- Src, Dst) START SLAVE;

Data Verification

- MyRocks/RocksDB is relatively new database technology
- Might have more bugs than robust InnoDB
- Ensuring data consistency helps avoid showing conflicting results

Verification tests

- Index count check between primary key and secondary keys
  - If any index is broken, it can be detected
  - SELECT ‘PRIMARY’, COUNT(*) FROM t FORCE INDEX (PRIMARY)
    UNION SELECT ‘idx1’, COUNT(*) FROM t FORCE INDEX (idx1)
  - Can’t be used if there is no secondary key
- Index stats check
  - Checking if “rows” show SHOW TABLE STATUS is not far different from actual row count
- Checksum tests w/ InnoDB
  - Comparing between InnoDB instance and MyRocks instance
  - Creating a transaction consistent snapshot at the same GTID position, scan, then compare checksum
- Shadow correctness check
  - Capturing read traffics

Shadow traffic tests

- We have a shadow test framework
  - Capturing read/write queries from production instances
  - Replaying them into a shadow master instance
- Shadow master tests
  - Client errors
  - Rewriting queries relying on Gap Lock
    - gap_lock_raise_error=1, gap_lock_write_log=1

Creating MyRocks copies

- We provide an open source MyRocks binary copy utility “myrocks_hotbackup”
- Once you create first MyRocks instance, you can create other MyRocks instances by myrocks_hotbackup
  - Binary copy is much faster than logical dump and load
  - Source instance is not stopped

myrocks hotbackup

- myrocks hotbackup
  - Python script
  - Creates binary backups
  - Requires running server
  - Crash recovery happens at startup
  - Does not set any locks when copies frm files!
    - You must ensure there is no parallel DDL while backup is running

Mixing InnoDB and MyRocks during migration

- It’s possible to mix InnoDB and MyRocks in the same replica set
  - e.g InnoDB master + 3 InnoDB slaves + 2 MyRocks slaves
  - Because replication is engine independent, MyRocks can be replicated from InnoDB (and vice versa)
  - But it’s not recommended mixing InnoDB and MyRocks within an instance
    - Atomic transaction can not be atomic
- It’s highly recommended having at least two MyRocks instances
  - Second MyRocks instance can be copied in binary format (myrocks_hotbackup)
    - Fast and not stopping source
  - First MyRocks instance has to be dumped from InnoDB
    - Slow and stopping source

Notable issues fixed during migration

- Lots of “Snapshot Conflict” errors
  - Because of implementation differences in MyRocks (PostgreSQL Style snapshot isolation)
  - Setting tx-isolation=READ-COMMITTED solved the problem
- Slaves stopped with I/O errors on reads
  - We switched to make MyRocks abort on I/O errors for both reads and writes
- Index statistics bugs
  - Inaccurate row counts/length on bulk loading -> fixed
  - Cardinality was not updated on fast index creation -> fixed
- Some crash bugs in MyRocks/RocksDB

Crash Safety

- Crash Safety makes operations much easier
  - Just restarting failed instances can restart replication
  - No need to rebuild entire instances, as long as data is there
- Crash Safe Slave
  - Crash Safe Master
  - 2PC (binlog and WAL)

Crash safety and XA

- We use Loss Less Semi-Synchronous Replication for Fast Master Failover
  - We do failover on machine failure
  - On mysqld crash, master may restart without failover
  - XA support is still needed to make binlog state and tx log (WAL) state consistent
- We fixed lots of XA bugs during preliminary deployment tests
  - Lost data on checkpoint (myrocks_hotbackup)
  - WAL size exceeded far more than max_wal_space_limit

Crash safety settings in depth

	sync-binlog	rocksdb-flush-log-at-trx-commit	rocksdb-enable-2pc	rocksdb-wal-recovery-mode
No data loss on unplanned machine reboot	1 (default)	1 (default)	1 (default)	1 (default)
No data loss on mysqld crash & recovery	0	2	1	1
No data loss if always failover	0	2	0	2

- Need Loss-Less Semisync to prevent data loss on failover
- sync-binlog: fsync() at every N commits
- rocksdb-flush-log-at-trx-commit: 1: fsync() at each commit, 0/2: fsync per second
- rocksdb-enable-2pc: synching binary logs and rocksdb WAL at recovery
- rocksdb-wal-recovery-mode: trimming corrupted WAL at recovery if needed, if not 1

- If you do not run master failover on machine reboot, you should make fully durable configurations (default).
- If you always run master failover on machine reboot but may restart on mysqld crash (e.g. mysqld_safe), you do not need fsync on commit but still need 2pc and wal protection
- If you always run master failover, you can run without durability and 2pc

Preventing stalls

- Heavy writes cause lots of compactions, which may cause stalls

- Typical write stall cases
  - Online schema change/migration that write lots of data
  - Massive data migration jobs

- Use rocksdb_bulk_load and fast index creation whenever possible
  - Online schema change can utilize this technique
  - Can be harder for data migration jobs that write into existing tables

When write stalls happen

- Estimated number of pending compaction bytes exceeded X bytes
  - soft|hard_pending_compaction_bytes, default 64GB
- Number of L0 files exceeded level0_slowdown|stop_writes_trigger (default 10)
- Number of unflushed number of MemTables exceeded max_write_buffer_number (default 4)

- All of these incidents are written to LOG as WARN level
- All of these options apply to each column family

What happens on write stalls

- Soft stalls
  - COMMIT takes longer time than usual
  - Total estimated written bytes to MemTable is capped to rocksdb_delayed_write_rate, until slowdown conditions are cleared
    - Default is 16MB/s (previously 2MB/s)
- Hard stalls
  - All writes are blocked at COMMIT, until stop conditions are cleared

Stalls because of pending compactions

- Estimated number of pending compaction bytes exceeded X bytes
  - soft|hard_pending_compaction_bytes, default 64GB
- SHOW ENGINE ROCKSDB STATUS prints “slowdown for pending_compaction_bytes”

Stalls because of too many L0 files

- Number of L0 files exceeded level0_slowdown_writes_trigger (default 10)

- SHOW ENGINE ROCKSDB STATUS prints current number of L0 files for each CF

- Possible causes and workarounds
  - Compactions might not be able to keep up with write ingestion
  - Don’t flush MemTables too often
    - Avoid too many implicit MemTable flushes
      - ANALYZE TABLE (Set rocksdb_flush_memtable_on_analyze=OFF)
      - Avoid dropping too many tables/indexes within short time (dropping non-empty indexes cause implicit MemTable flush)

Stalls because of slow MemTable flush

- Number of unflushed number of MemTables exceeded max_write_buffer_number (default 4)
  - Flush (MemTable->L0) couldn’t keep up with writes from app. Less likely to happen
  - Don’t use too strong compression algorithm in L0 (use NoCompression or LZ4 in L0)
  - Increase rocksdb_max_background_flushes if needed (4~8 are sane numbers)
  - Don’t increase max_write_buffer_number too many
    - Estimated peak memory usage in MemTable is write_buffer_size * (number of CF) * max_write_buffer_number

Mitigating Write Stalls

- Reduce total bytes written
  - But avoid using too strong compression algorithm on upper levels
    - Zlib or ZSTD compression in the bottommost level
    - LZ4 in all other levels
  - Use more write efficient compaction algorithm
    - compaction_pri=kMinOverlappingRatio
- Delete files slowly on Flash
  - Deleting too many large files cause TRIM stalls on Flash
  - MyRocks throttles sst file deletes by 64MB/s by default
  - Binlog file deletions should be slowed down as well

Speeding up compactions

- Common tips to speed up compactions
  - Using faster compression algorithm
  - Increasing rocksdb_max_background_compactions
  - Reducing total number of bytes written by compactions (making compactions smarter)

Reducing write amplification and time

- How to get Write Amplification
  - rocksdb_compact_write_bytes / rocksdb_wal_bytes from SHOW GLOBAL STATUS
- Common Configurations
  - compression_per_level=kLZ4Compression;bottommost_compression=kZSTD
  - compaction_pri=kMinOverlappingRatio

Monitoring

- MyRocks files
- SHOW ENGINE ROCKSDB STATUS
- SHOW ENGINE ROCKSDB TRANSACTION STATUS
- LOG files
- information_schema tables
- sst_dump tool
- ldb tool

MyRocks/RocksDB files

- Data Files (*.sst)
- WAL files (*.log)
- Manifest files
- Options files
- LOG files

- All files are created at $datadir/.rocksdb by default
  - Can be changed by rocksdb_datadir and rocksdb_wal_dir

SHOW ENGINE ROCKSDB STATUS

- Column Family Statistics, including size, read and write amp per level
- Memory usage

```
*************************** 7. row ***************************
  Type: CF_COMPACTION
  Name: default
Status:
** Compaction Stats [default] **
Level    Files   Size(MB) Score Read(GB)   Rn(GB) Rnp1(GB) Write(GB) Wnew(GB) Moved(GB) W-Amp Rd(MB/s) Wr(MB/s) Comp(sec) Comp(cnt) Avg(sec) KeyIn KeyDrop
----------------------------------------------------------------------------------------------------------------------------------------------------------
  L0      2/0      51.58   0.5      0.0      0.0       0.0       0.3       0.3       0.0   0.0      0.0     40.3          7         10    0.669       0       0
  L3      6/0     109.36   0.9      0.7      0.7       0.0       0.6       0.6       0.0   0.9     43.8     40.7         16          3    5.172    7494K   297K
  L4     61/0    1247.31   1.0      2.0      0.3       1.7       2.0       0.2       0.0   6.9     49.7     48.5         41          9    4.593     15M   176K
  L5    989/0   12592.86   1.0      2.0      0.3       1.8       1.9       0.1       0.0   7.4      8.1      7.4        258          8   32.209     17M   726K
  L6   4271/0  127363.51   0.0      0.0      0.0       0.0       0.0       0.0       0.0   0.0      0.0      0.0          0          0    0.000       0       0
 Sum   5329/0  141364.62   0.0      4.7      1.2       3.5       4.7       1.2       0.0  17.9     15.0     15.0        321         30   10.707     41M  1200K
```

SHOW ENGINE ROCKSDB TRANSACTION STATUS

- Similar to SHOW ENGINE INNODB STATUS for transaction secti0n.
  Useful to find out long running sessions

```
mysql> show engine rocksdb transaction status\G
*************************** 1. row ***************************
  Type: SNAPSHOTS
  Name: rocksdb
Status:
=====================================================
2016-04-14 14:29:46 ROCKSDB TRANSACTION MONITOR OUTPUT
=====================================================
---------
SNAPSHOTS
---------
LIST OF SNAPSHOTS FOR EACH SESSION:
---SNAPSHOT, ACTIVE 27 sec
MySQL thread id 9, OS thread handle 0x7fbbfcc0c000  query id 11276587  root
lock count 3, write count 1
---------------------------------------
END OF ROCKSDB TRANSACTION MONITOR OUTPUT
=======================================
```

SHOW GLOBAL STATUS

```
mysql> show global status like 'rocksdb%';
+---------------------------------------+-------------+
| Variable_name                         | Value       |
+---------------------------------------+-------------+
| rocksdb_rows_deleted                  | 216223      |
| rocksdb_rows_inserted                 | 1318158     |
| rocksdb_rows_read                     | 7102838     |
| rocksdb_rows_updated                  | 1997116     |
....
| rocksdb_bloom_filter_prefix_checked   | 773124      |
| rocksdb_bloom_filter_prefix_useful    | 308445      |
| rocksdb_bloom_filter_useful           | 10108448    |
....
```

information_schema

```
mysql> select f.index_number, f.sst_name from information_schema.rocksdb_index_file_map
f, information_schema.rocksdb_ddl d where f.column_family = d.column_family and
f.index_number = d.index_number and d.table_schema= ‘test’ and
d.table_name= ‘linktable’ and d.cf=’rev:cf_id1_type’ order by 1, 2;
+--------------+------------+
| index_number | sst_name   |
+--------------+------------+
|         2822 | 156068.sst |
|         2822 | 156119.sst |
|         2822 | 156164.sst |
|         2822 | 156191.sst |
|         2822 | 156240.sst |
|         2822 | 156294.sst |
|         2822 | 156333.sst |
|         2822 | 259093.sst |
|         2822 | 268721.sst |
|         2822 | 268764.sst |
|         2822 | 270503.sst |
|         2822 | 270722.sst |
|         2822 | 270971.sst |
|         2822 | 271147.sst |
+--------------+------------+
14 rows in set (0.41 sec)
```

This is an example returning all sst file names that specified db.table.cf_name has

sst_dump

- sst_dump is leveldb/rocksdb tool to parse a SST file. This is useful for debugging purposes, if you want to investigate SST files
- Automatically built and installed on fb-mysql

```
# sst_dump --command=scan --output_hex --file=/data/mysql/.rocksdb/000020.sst
```

Perf Context

- RocksDB exposes many internal statistics
- It’s disabled in MyRocks by default, since it’s relatively expensive
- Can be enabled by “set global rocksdb_perf_context_level=2;”

- Global level context
  - select * from information_schema.rocksdb_perf_context_global;
- Per table context
  - select * from information_schema.rocksdb_perf_context where table_schema=‘test’ and table_name=‘t1’;
- If “INTERNAL_DELETE_SKIPPED_COUNT” is very high, it’s a sign that there are many tombstones

Summary

- We started migrating from InnoDB to MyRocks in our main database
- Major motivation was to save space
- Online data correctness check tool helped to find lots of data integrity bugs and prevented from deploying inconsistent instances in production
- Bulk loading and fast index creation helped to avoid compaction stalls
- Crash safe features like Transactions, XA are supported

facebook