facebook

MyRocks Deep Dive

Yoshinori Matsunobu
Production Database Engineer, Facebook
Apr 18, 2016 – Percona Live Tutorial

Agenda

- MyRocks overview
- Getting Started
- Architecture
  - Data structure and table definition
  - Query optimizer and statistics
  - Row Locking and Concurrency
- Replication, Backup and Recovery
- Performance Tuning
- Monitoring

Target audiences

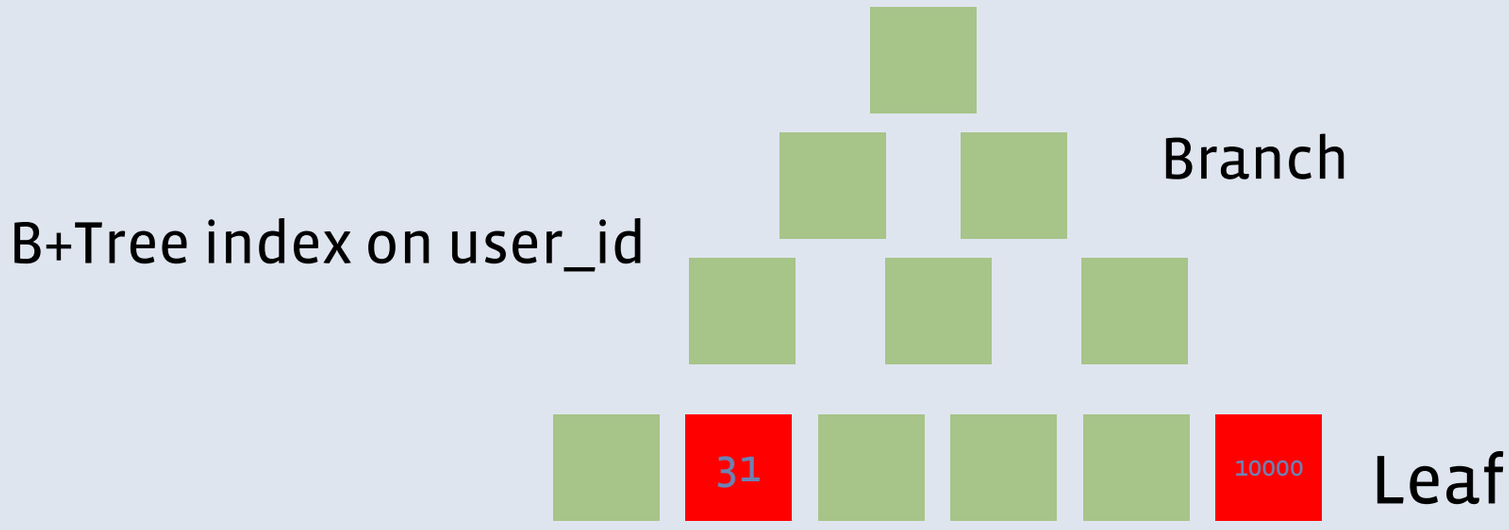
- Interested in efficiency -- Reducing the number of MySQL servers
- Already familiar with MySQL
  - InnoDB and MySQL Replication
  - It is ok you don't know anything about RocksDB or LSM – they're covered in this tutorial!

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

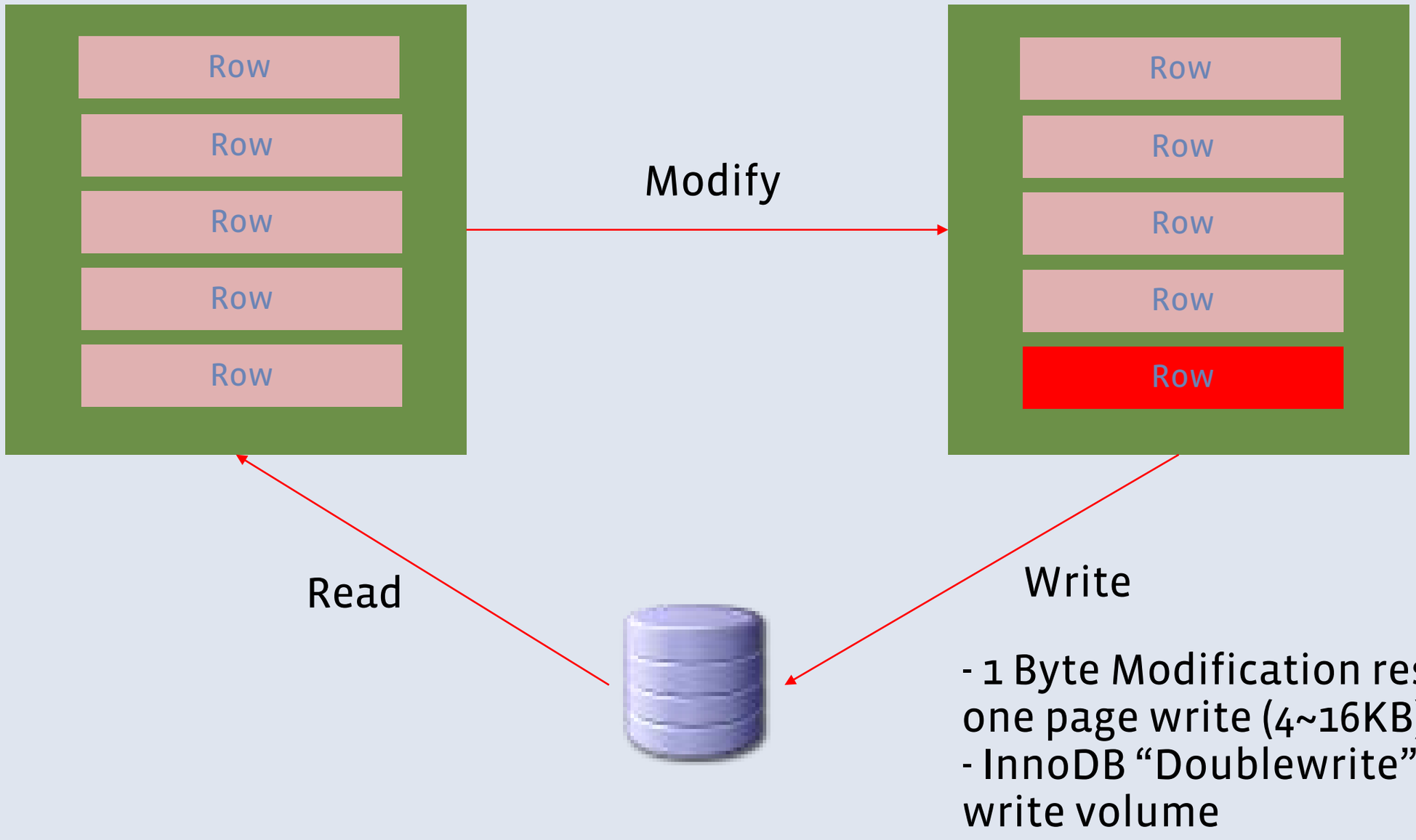
Random Write on B+Tree

```
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

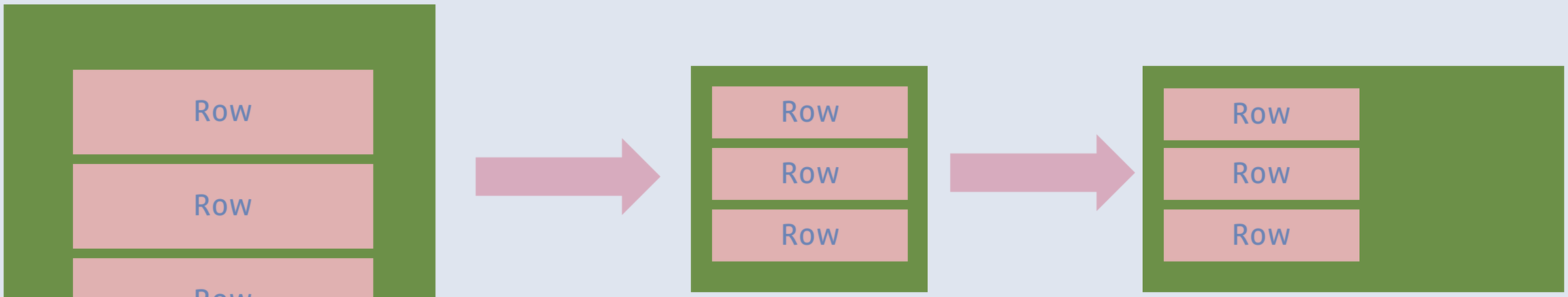
```
INSERT INTO message_table (user_id) VALUES (31)
```

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

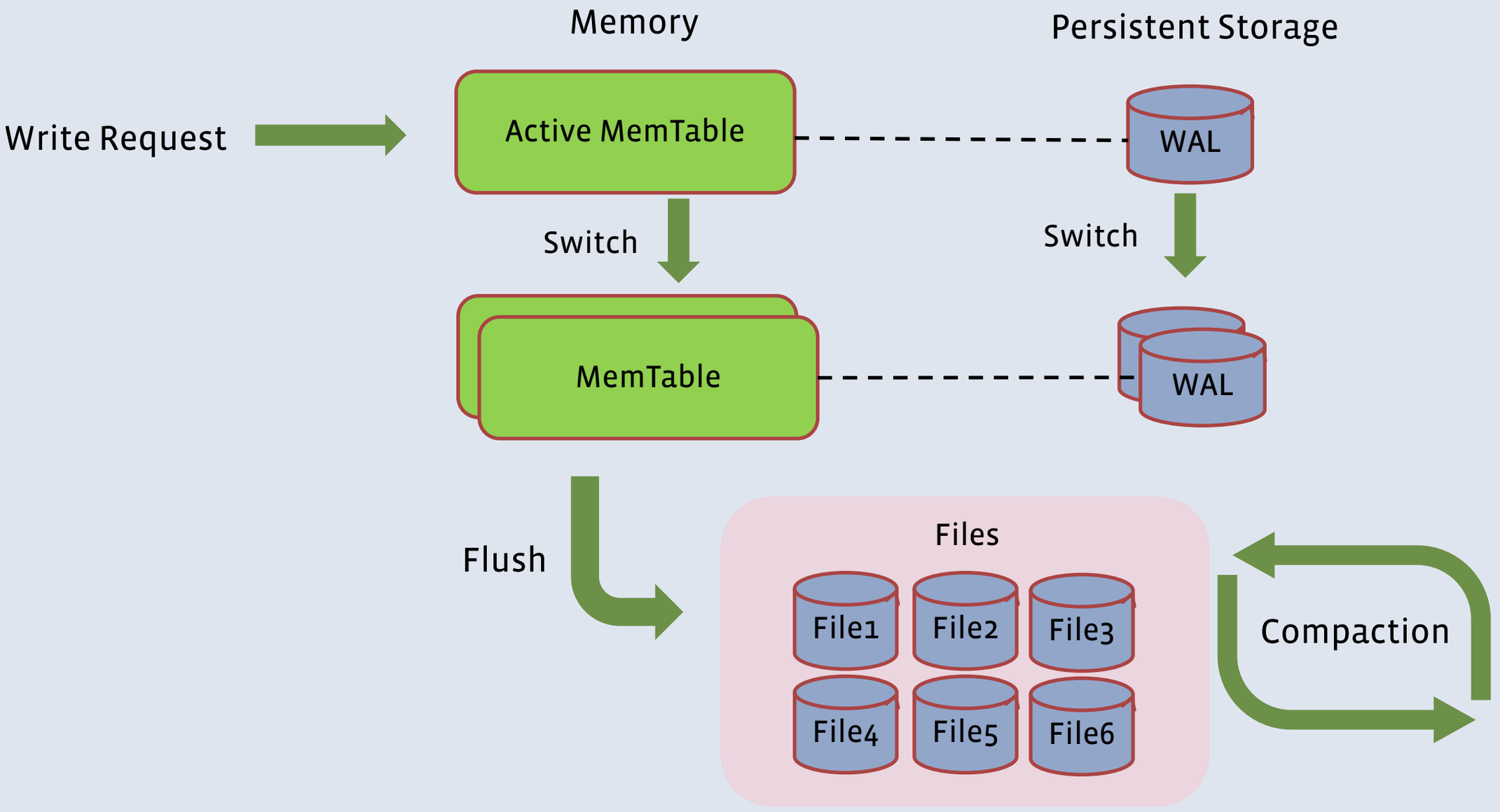
- http://rocksdb.org/
- Forked from LevelDB
  - Key-Value LSM persistent store
  - Embedded
  - Data stored locally
  - Optimized for fast storage

- 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

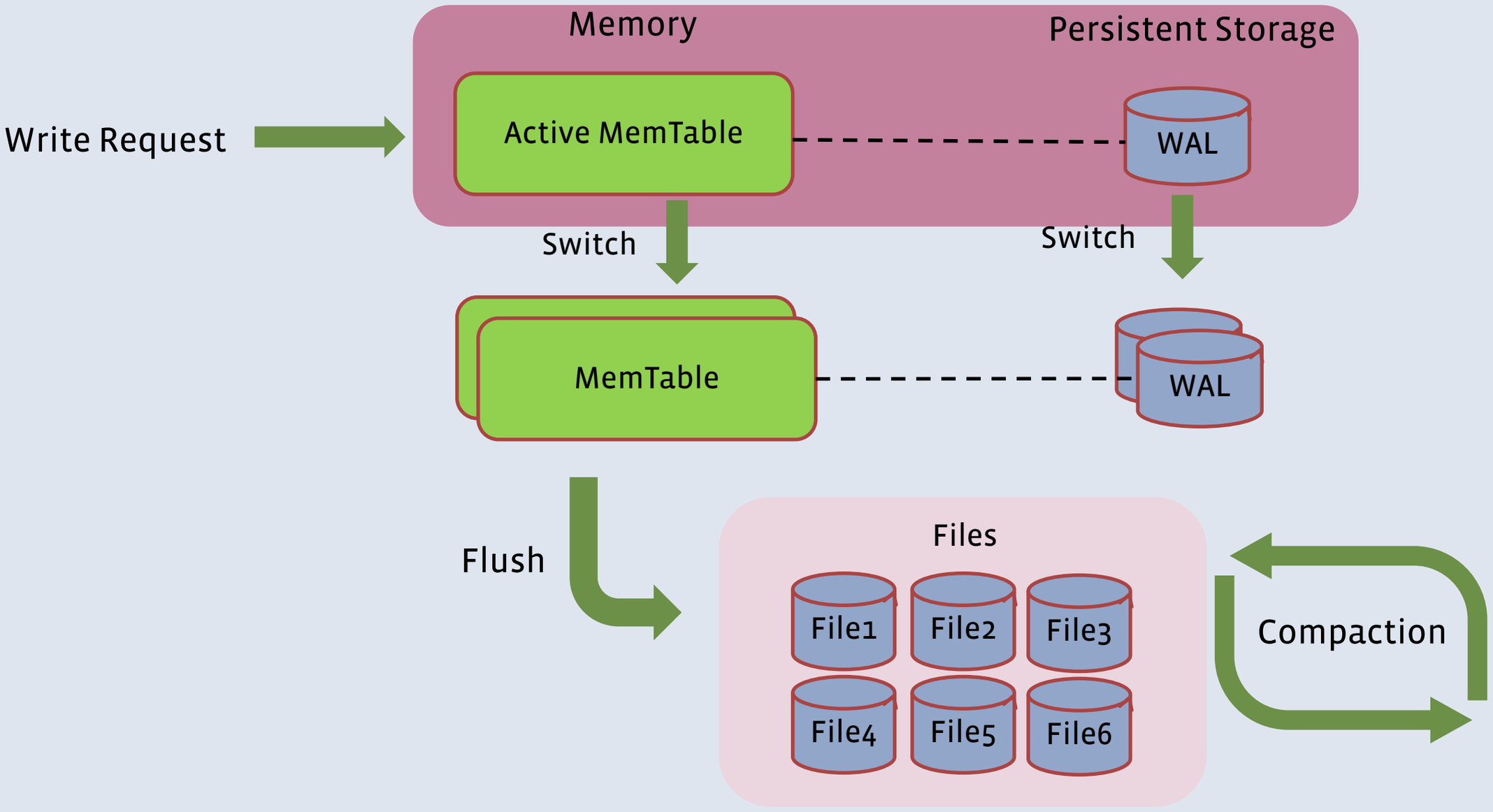
RocksDB architecture overview

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

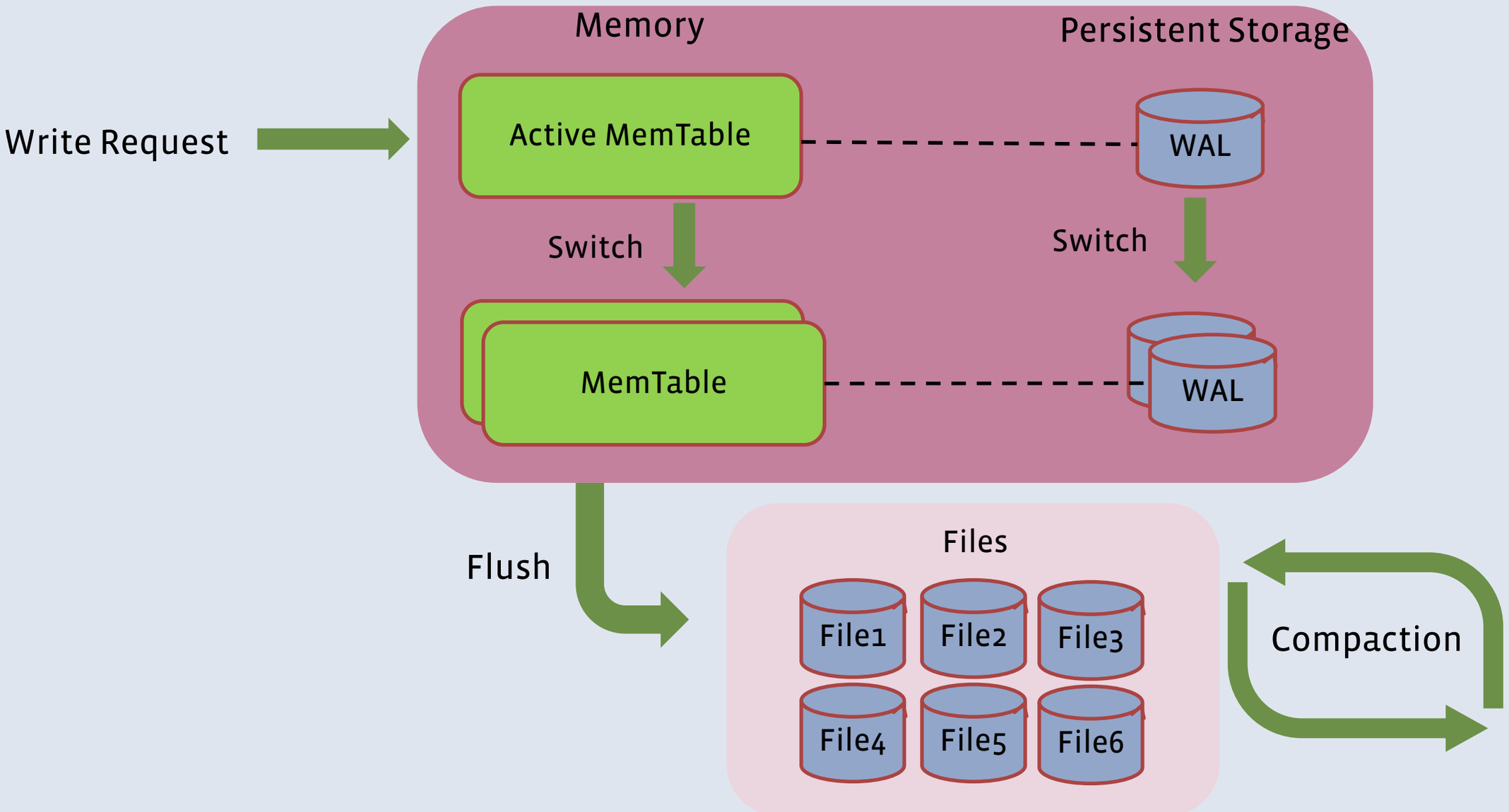
RocksDB Architecture



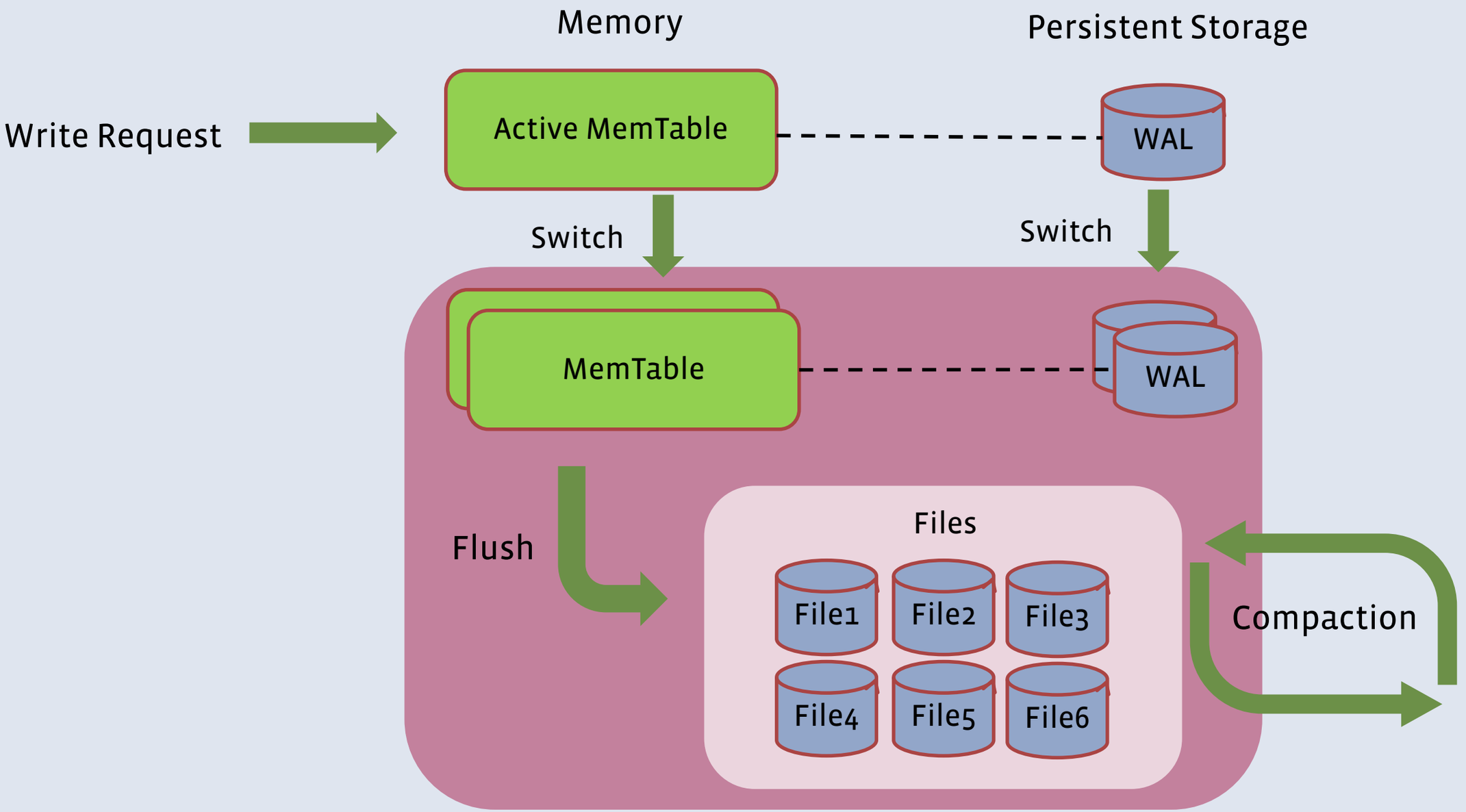
Write Path (1)


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

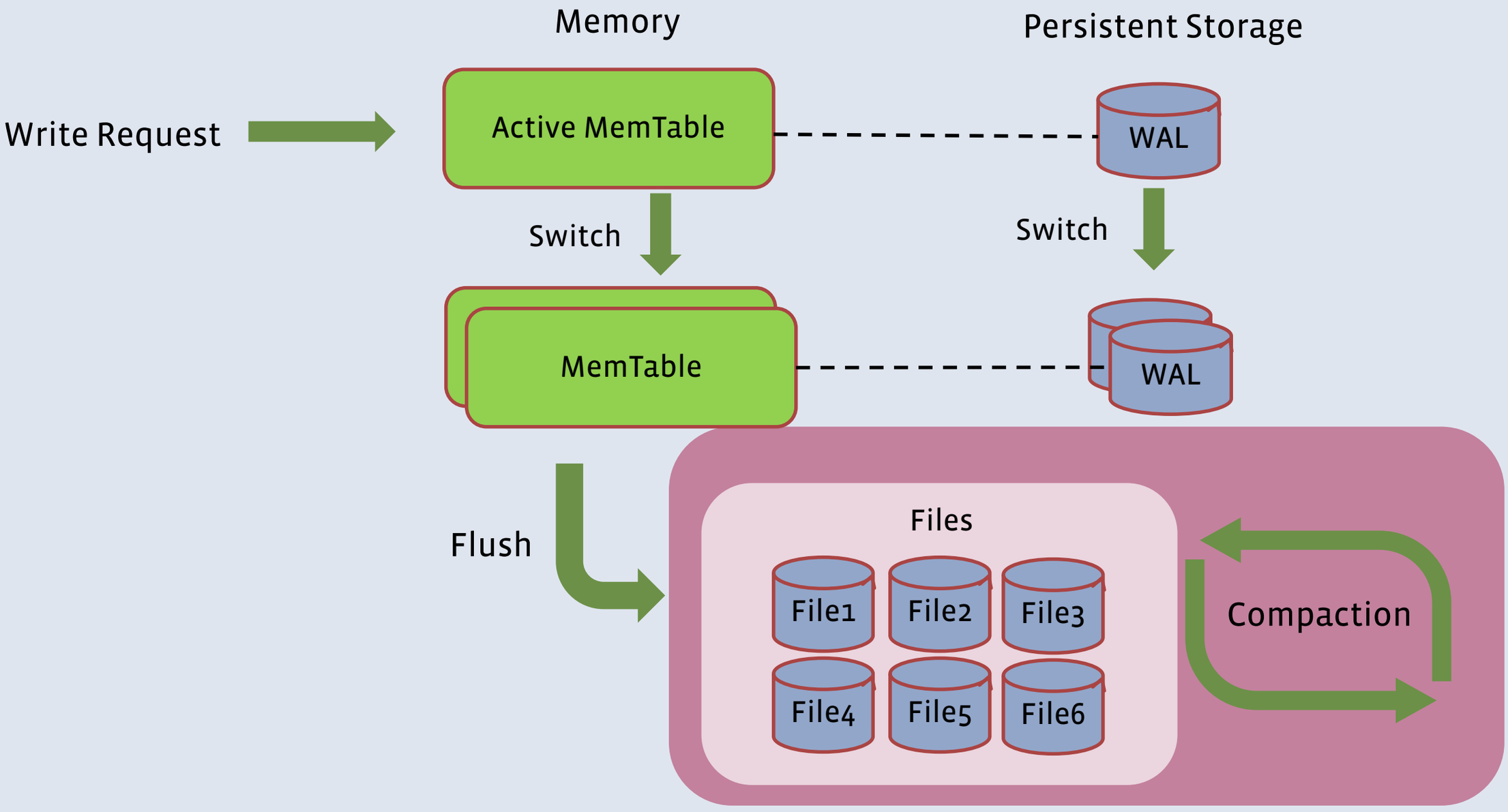
Write Path (2)


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

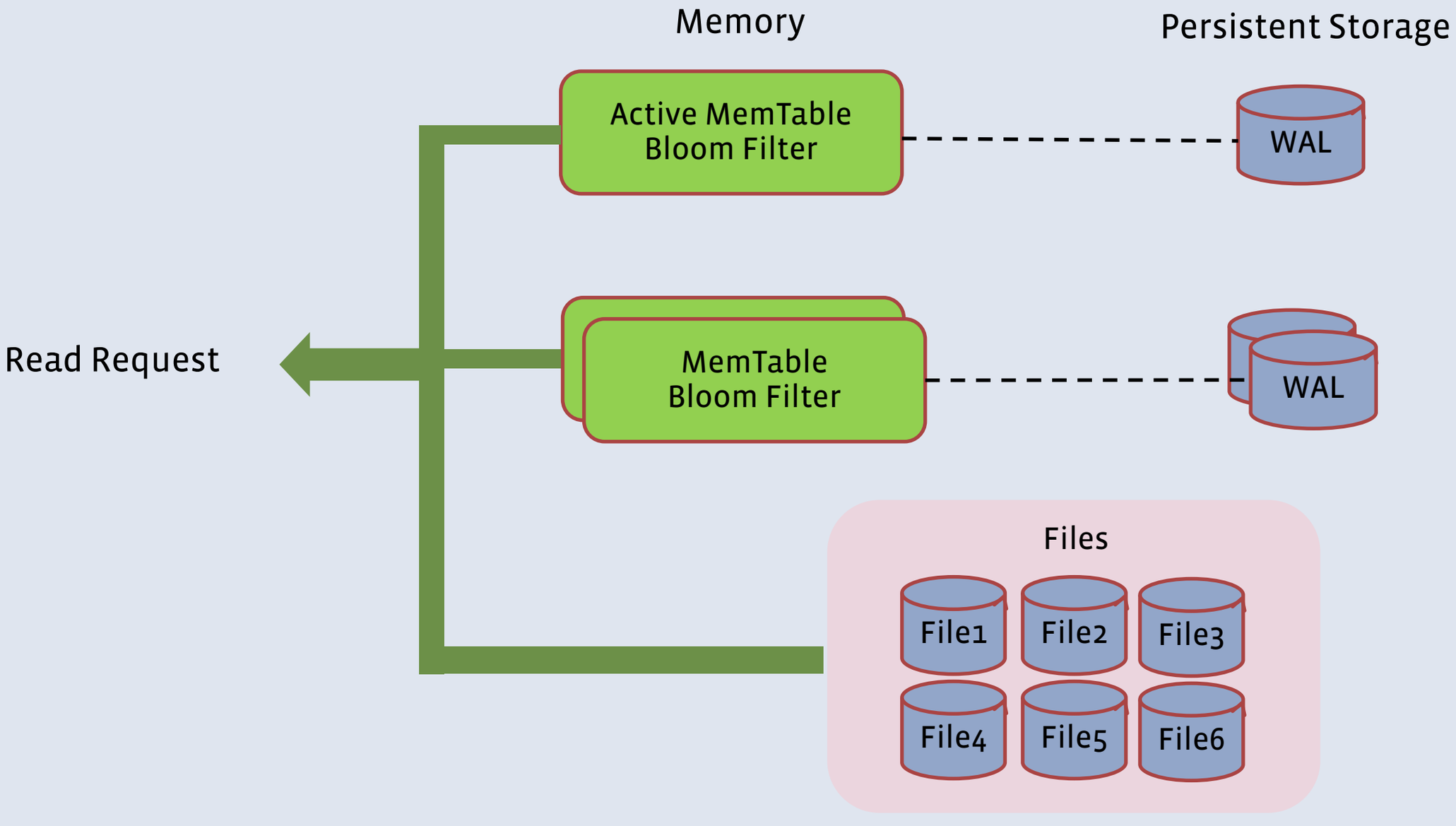
Write Path (3)


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

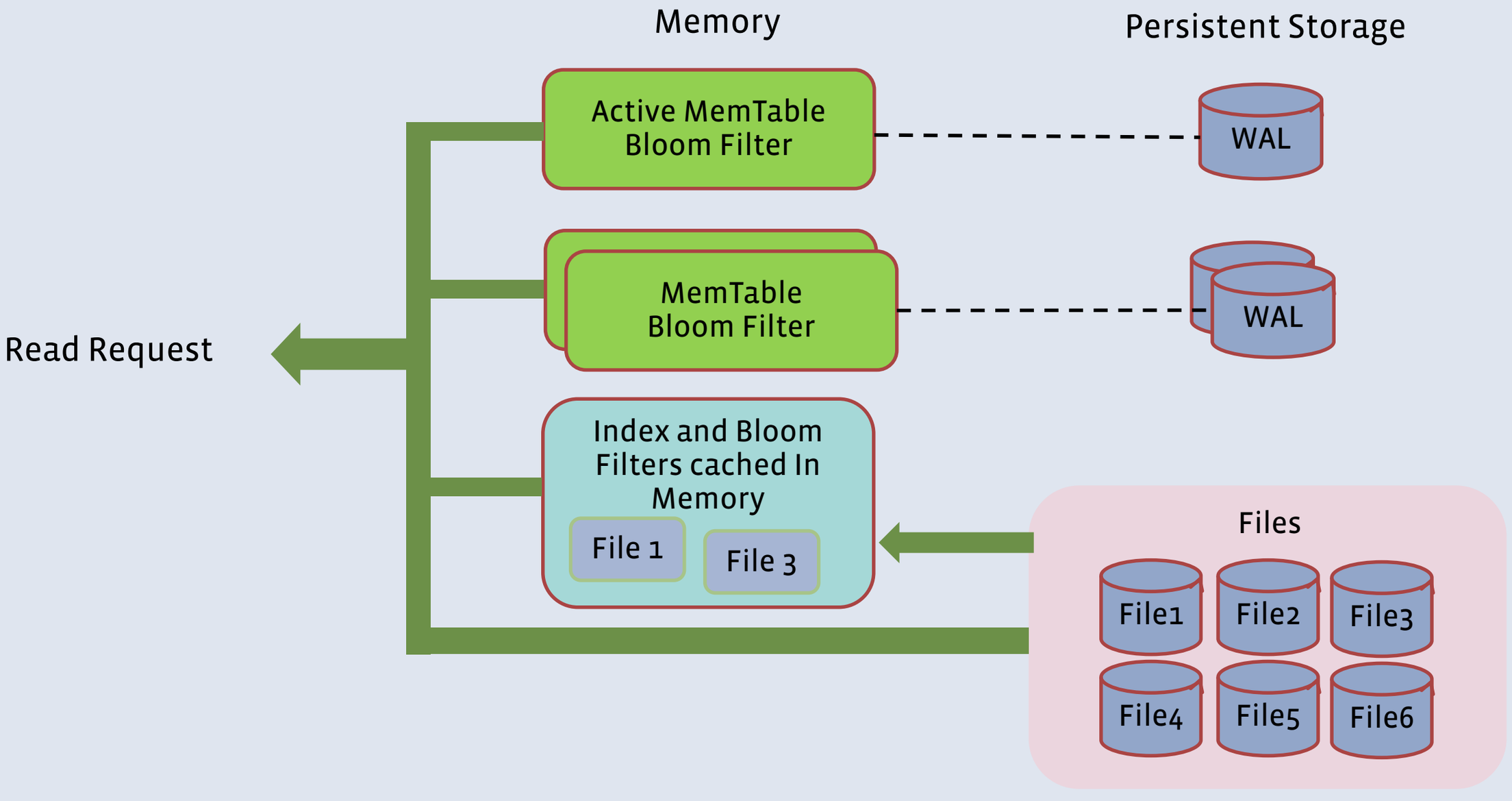
Write Path (4)


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

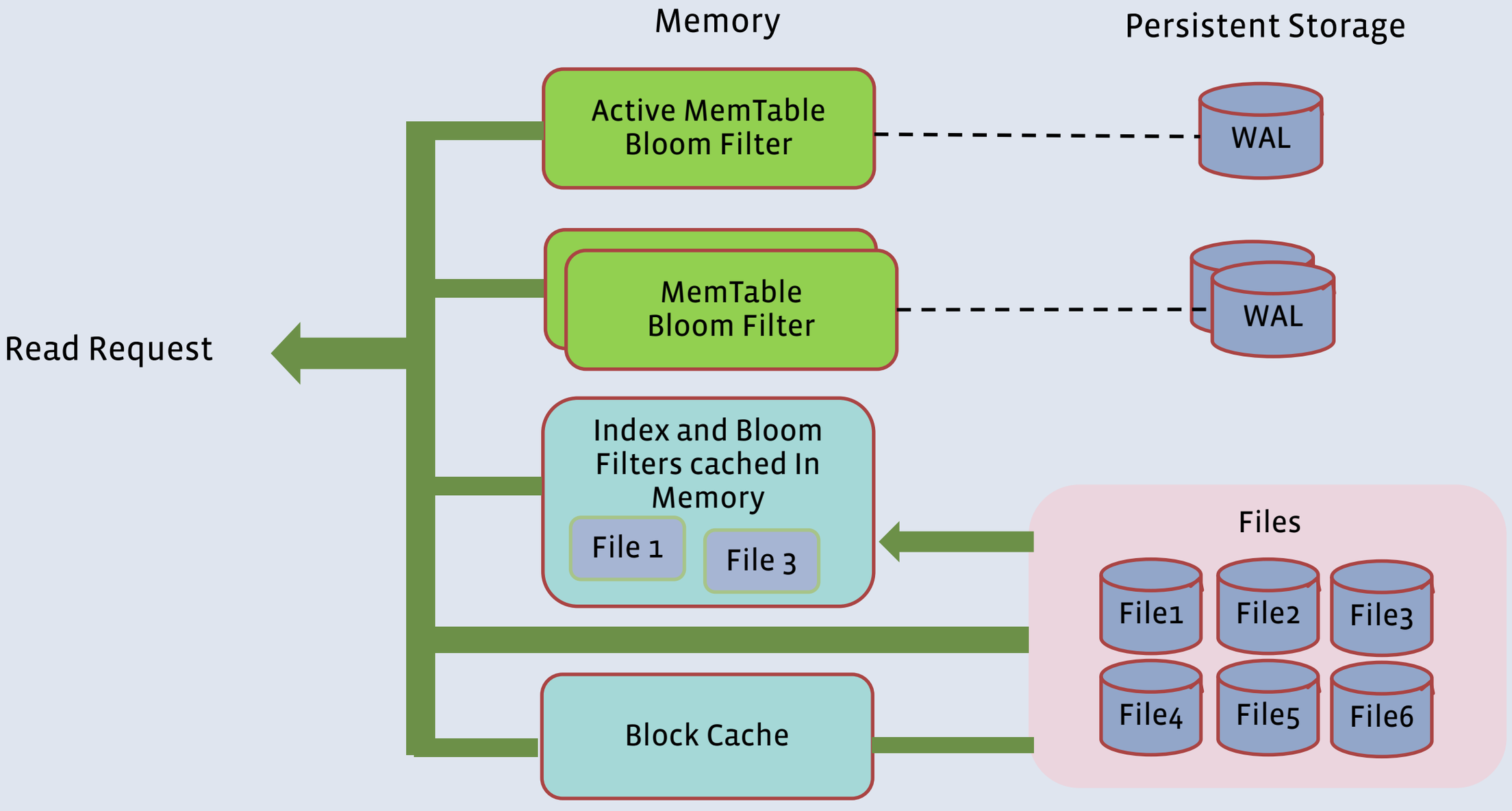
Read Path


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

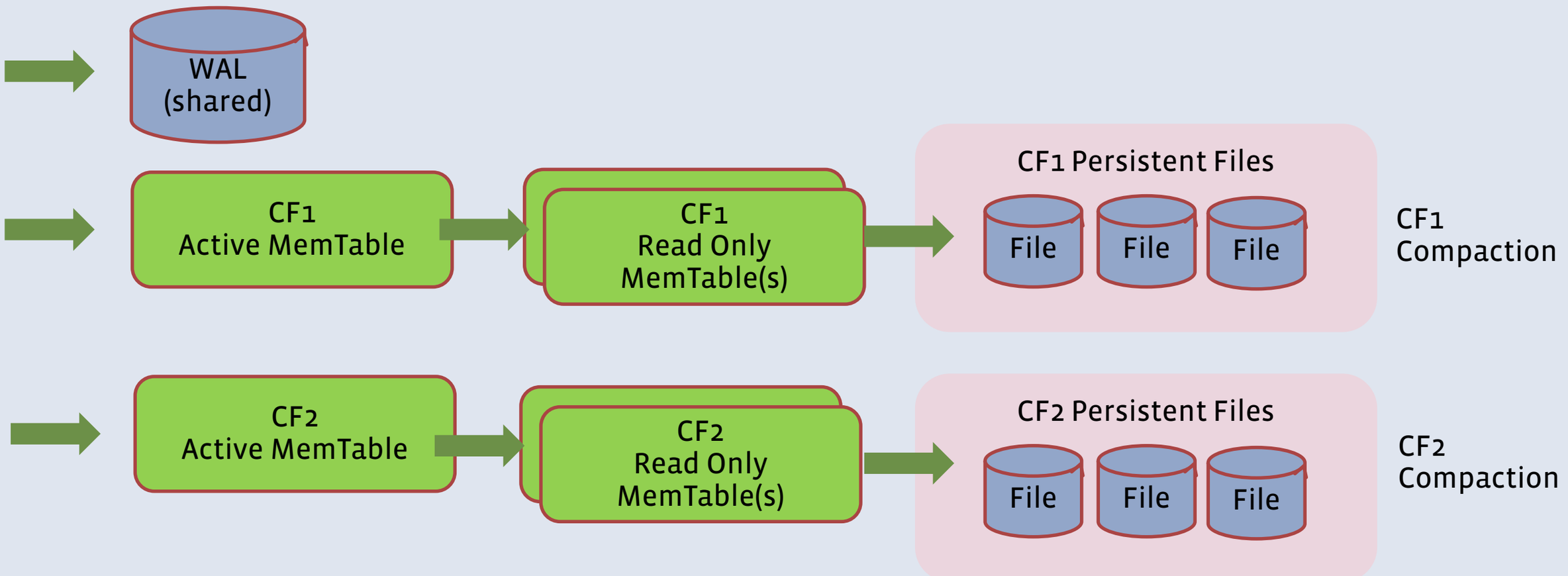
Read Path


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

Column Family

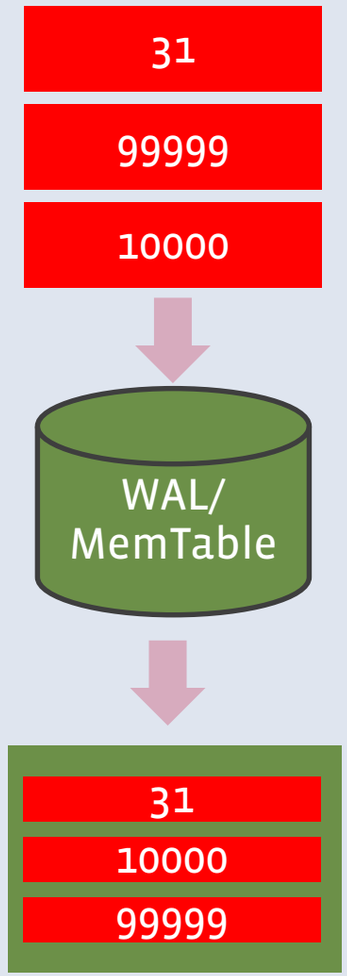
Query atomicity across different key spaces.

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

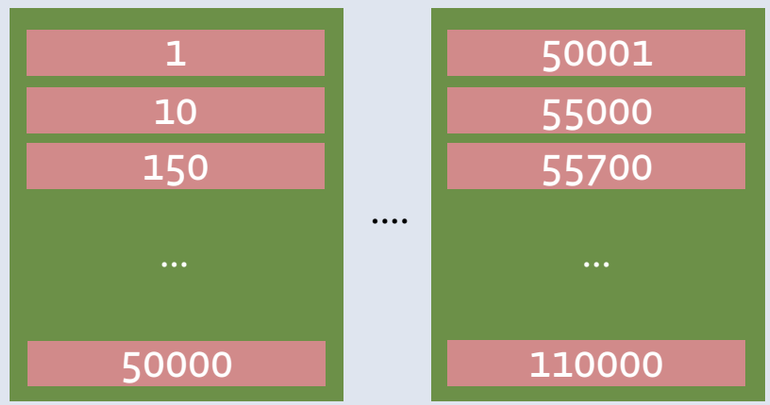


How LSM works

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

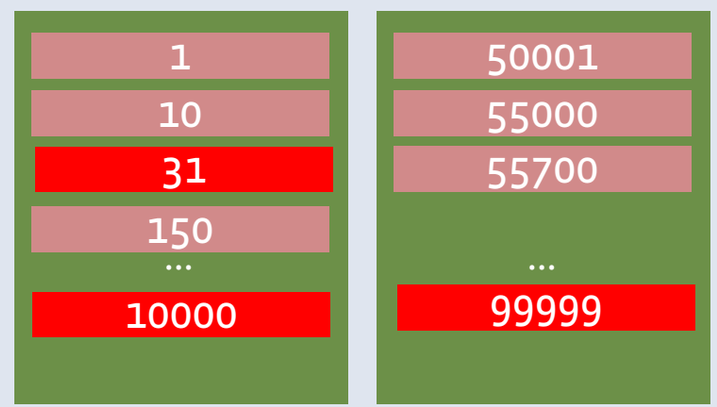


New SST



existing SSTs

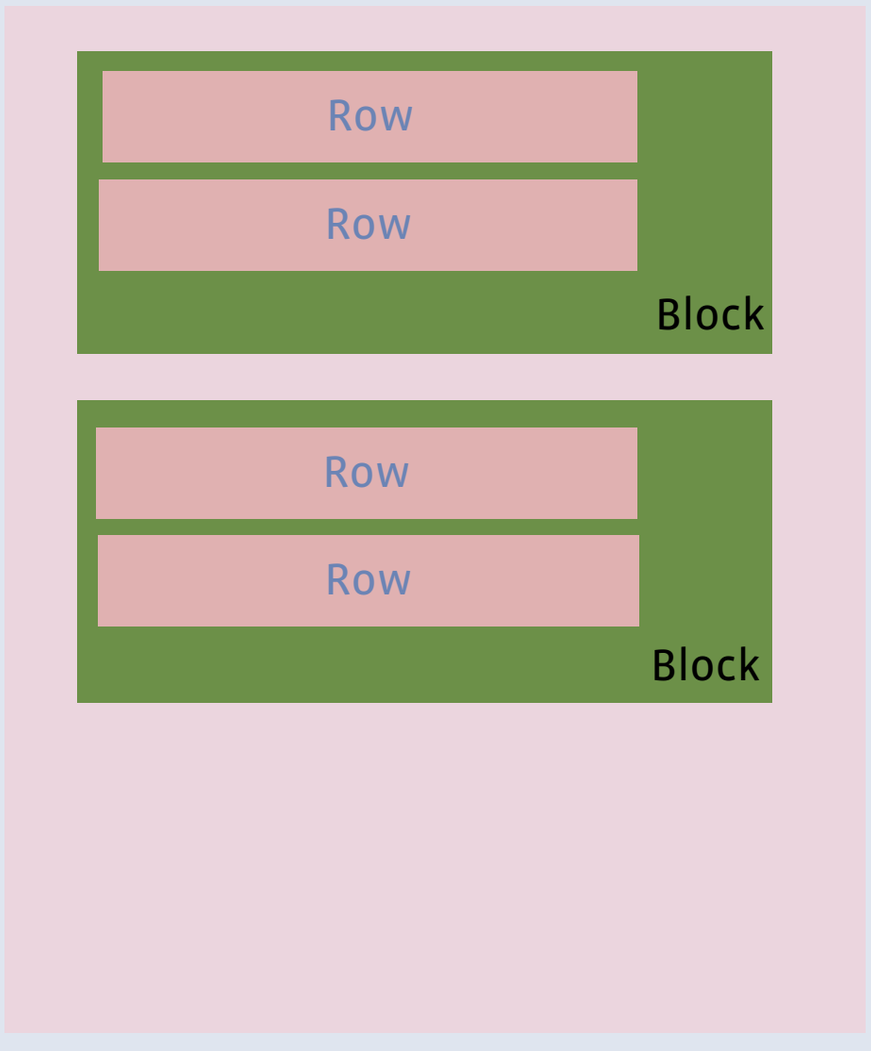
Compaction



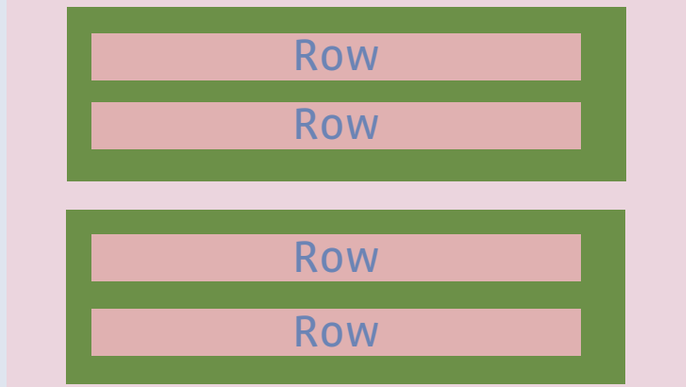
Writing new SSTs sequentially

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

LSM handles compression better



16MB SST File

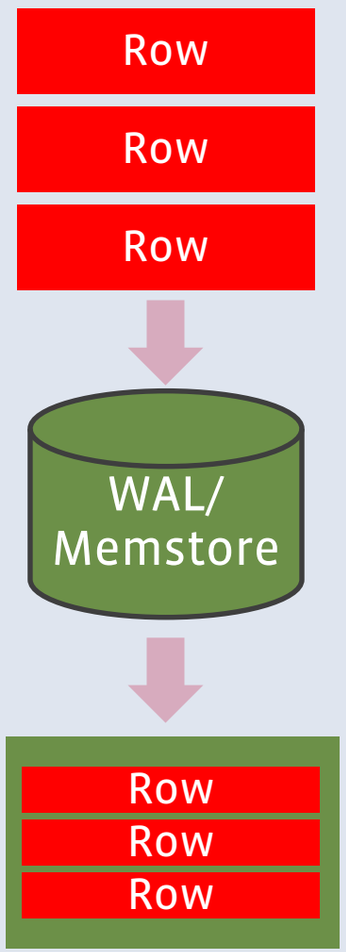


5MB SST File

=> 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

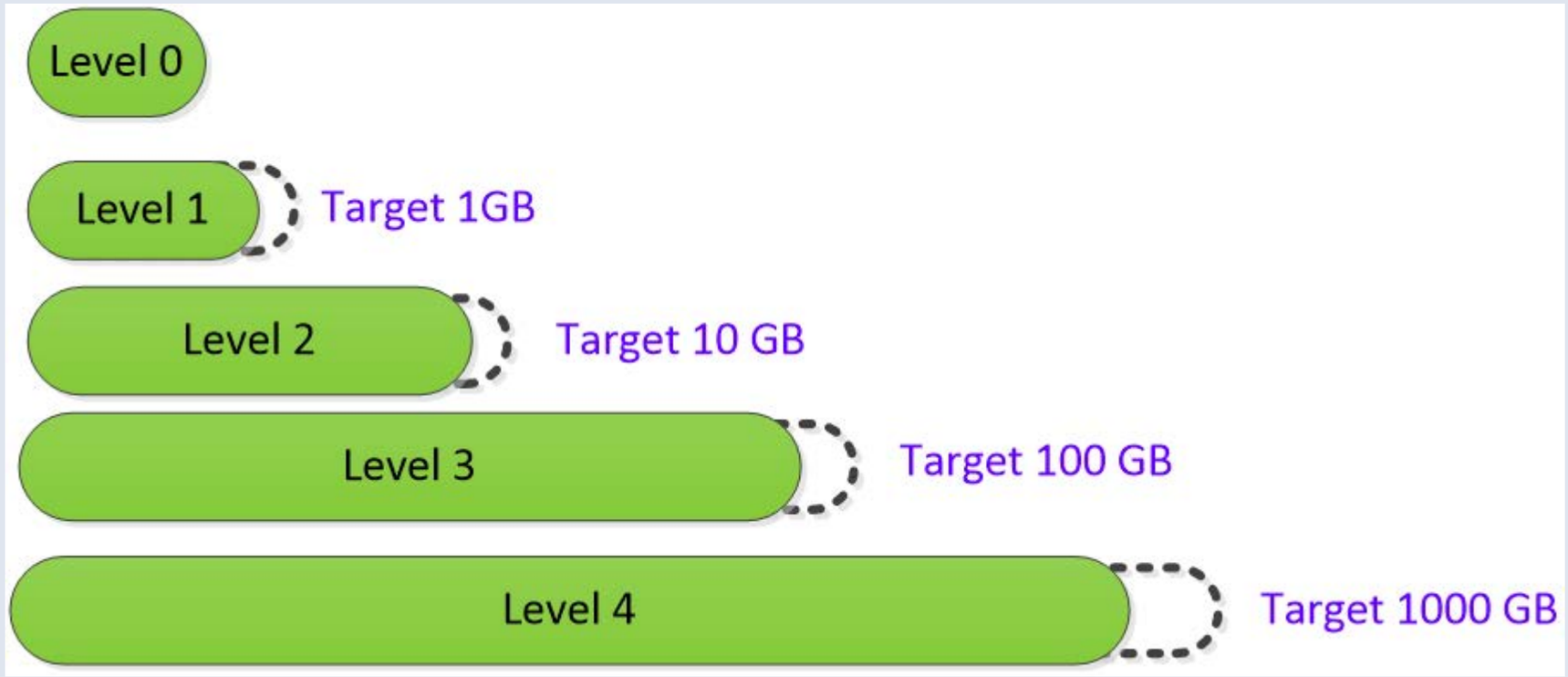
Zero-Filling metadata

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

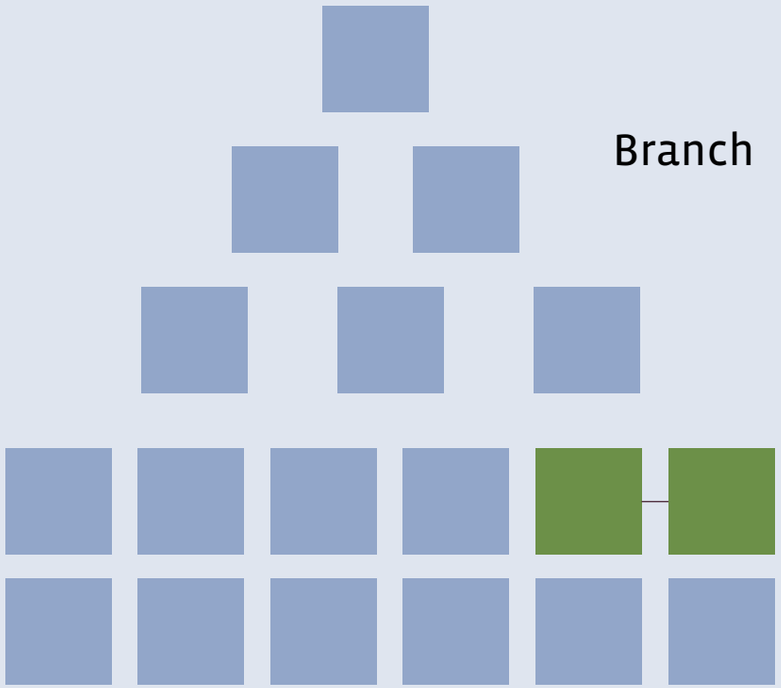


- For each level, data is sorted by key
- Read Amplification: 1 ~ number of levels (depending on cache -- L0~L3 are usually cached)
- Write Amplification: 1 + 1 + fanout * (number of levels – 2) / 2
- Space Amplification: 1.11
  - 11% is much smaller than B+Tree's fragmentation

Read Penalty on LSM

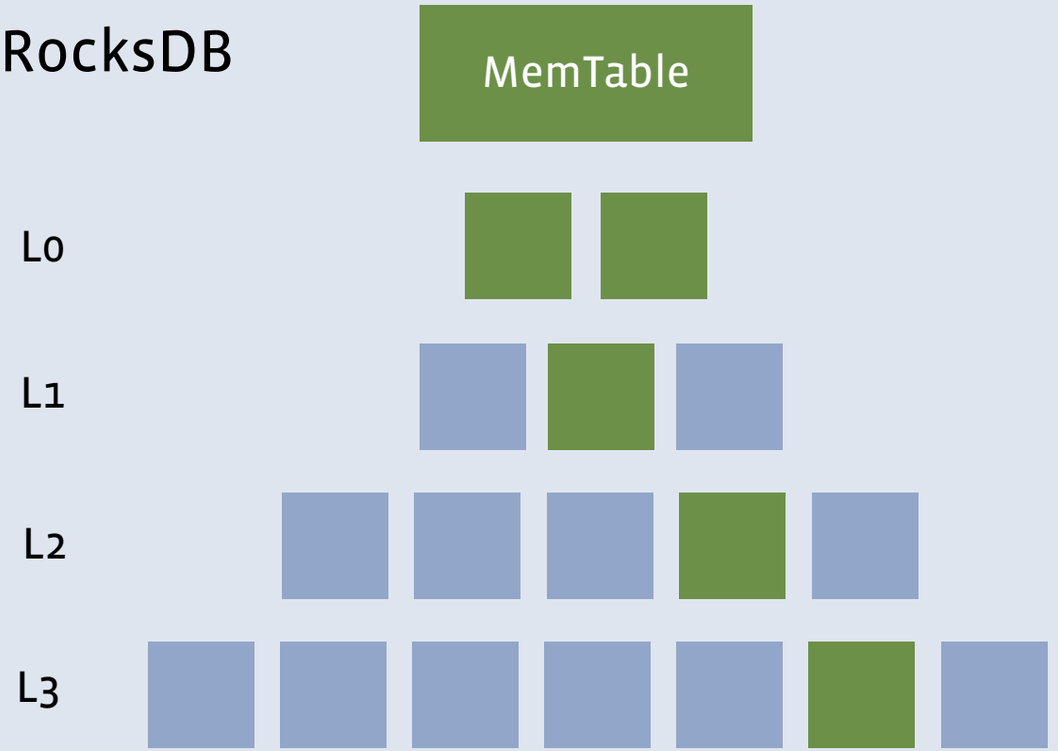
SELECT id1, id2, time FROM t WHERE id1=100 AND id2=100 ORDER BY time DESC LIMIT 1000;
Index on (id1, id2, time)

InnoDB



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

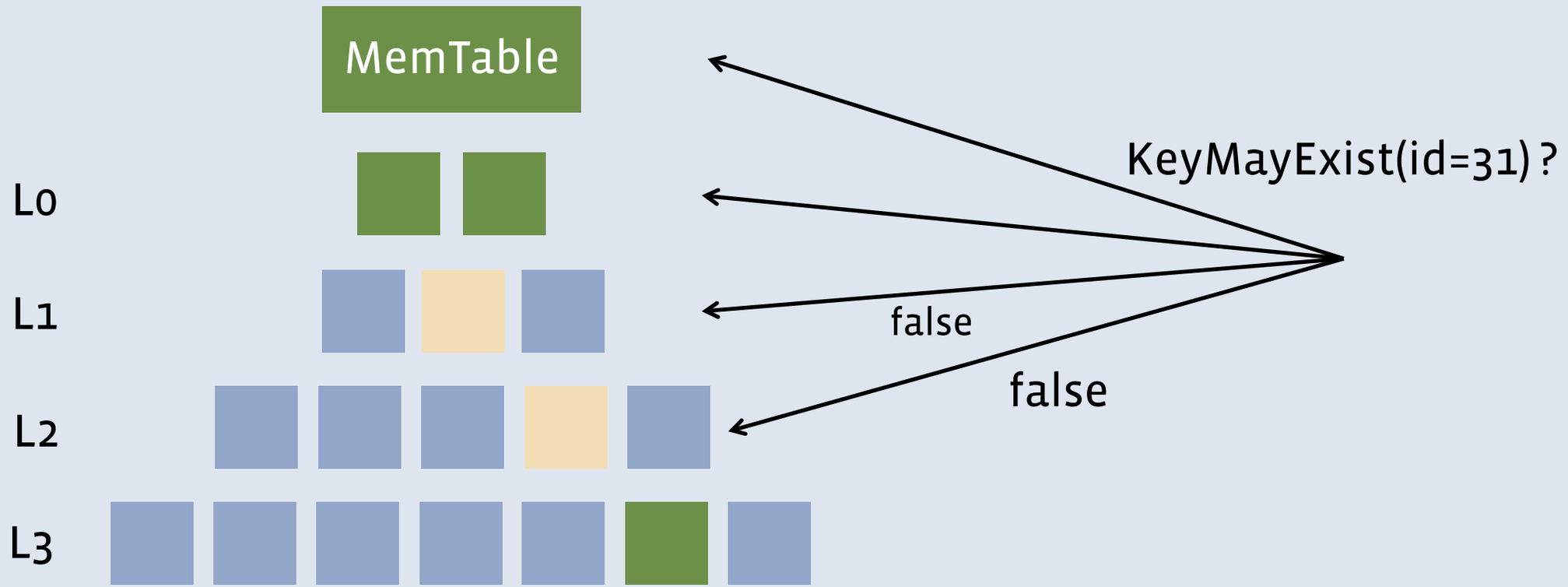
RocksDB



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)

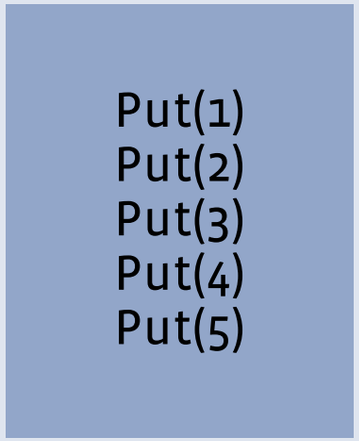
Bloom Filter

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

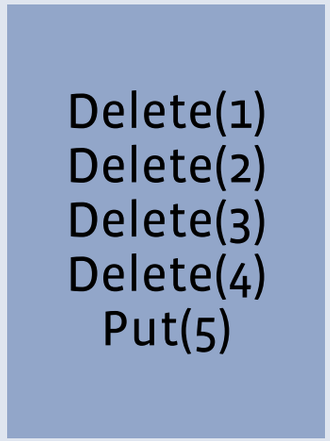


Delete penalty

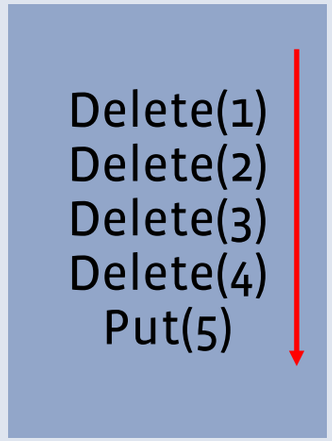
INSERT INTO t
VALUES (1),(2),(3),(4),(5);



DELETE FROM t WHERE
id <= 4;



SELECT COUNT(*) FROM t;



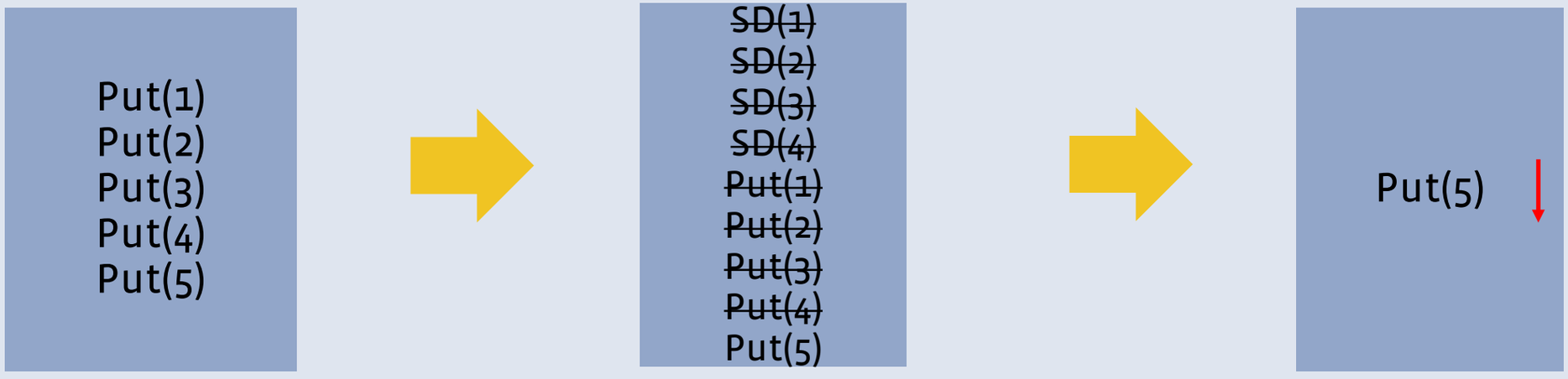
- “Delete” adds a tombstone
  - When reading, ignoring *all* Puts for the same key
- Tombstones can’t disappear until bottom level compaction happens
- Some reads need to scan lots of tombstones => inefficient
  - In this example, reading 5 entries is needed just for getting one row

“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
  - Working with MariaDB Company
  - Currently RC stage
  - 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

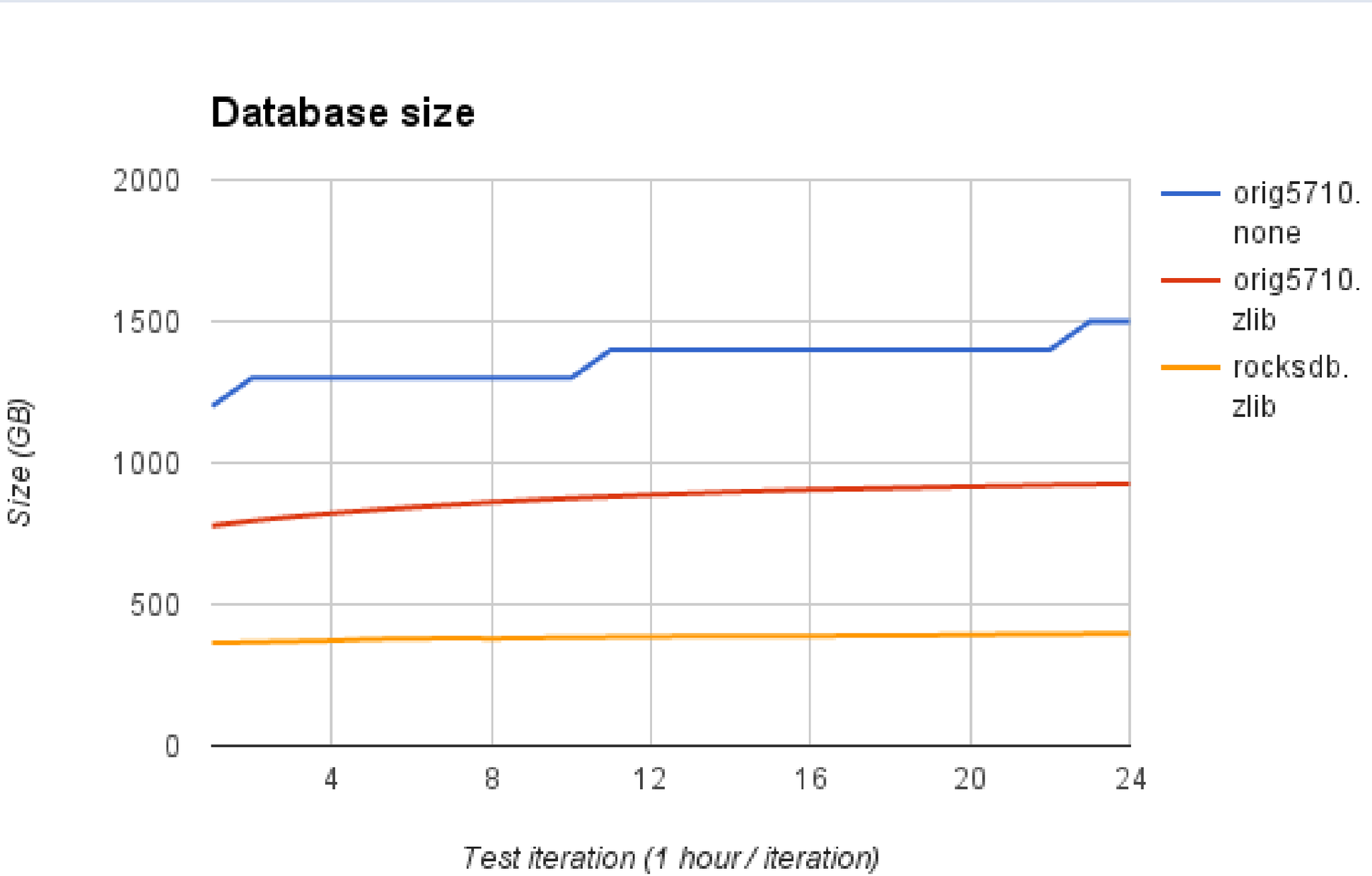
- Much smaller space and write amplification compared to InnoDB
- Reverse order index (Reverse Column Family)
- SingleDelete
- Prefix bloom filter
  - “SELECT … WHERE id=1 and time >= X” => using bloom filter for id
- Mem-comparable keys when using case sensitive collations
- Optimizer statistics without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

Performance (LinkBench)

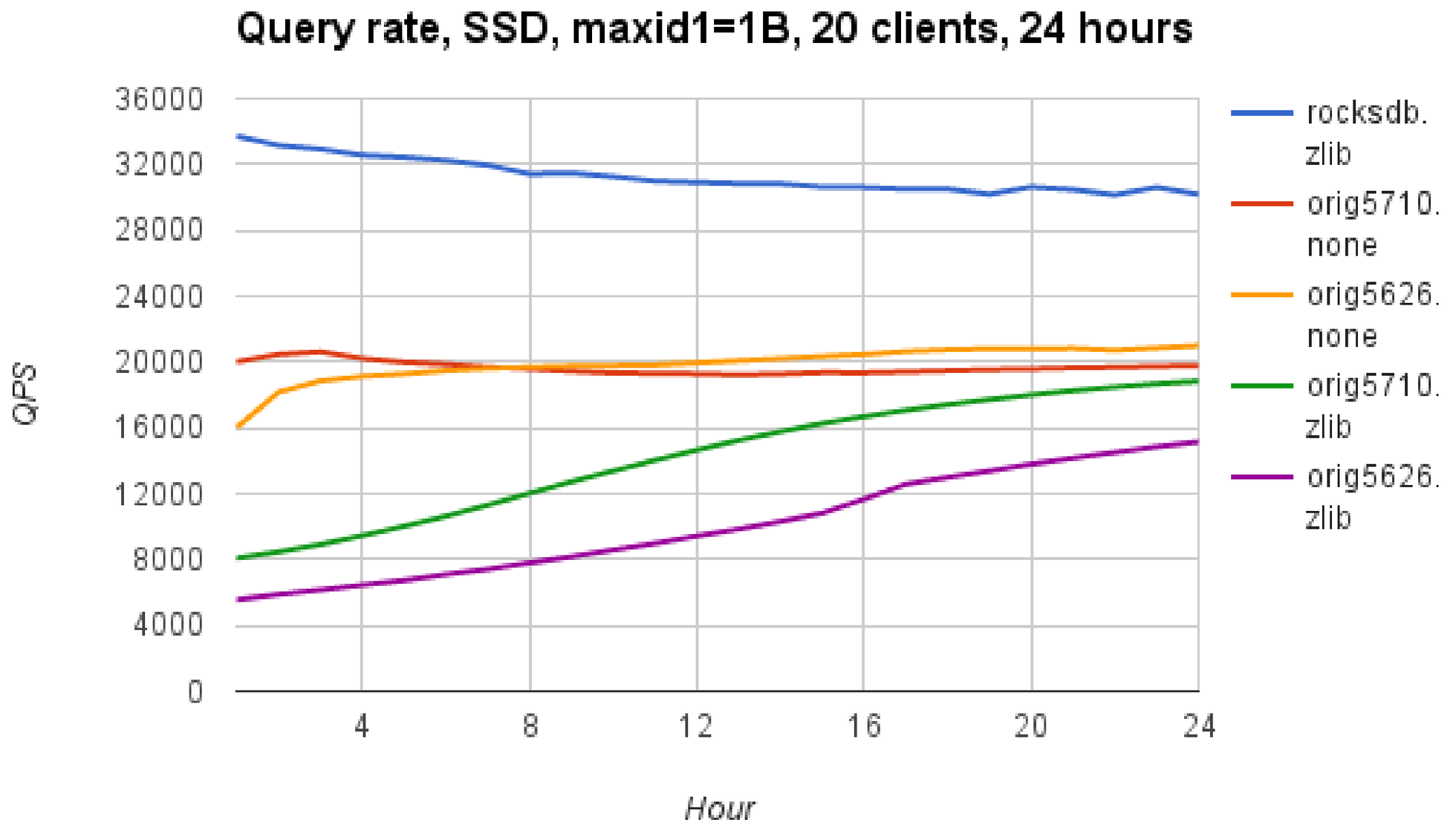
- Space Usage
- QPS
- Flash reads per query
- Flash writes per query
- Data Loading
- Latency
- HDD

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

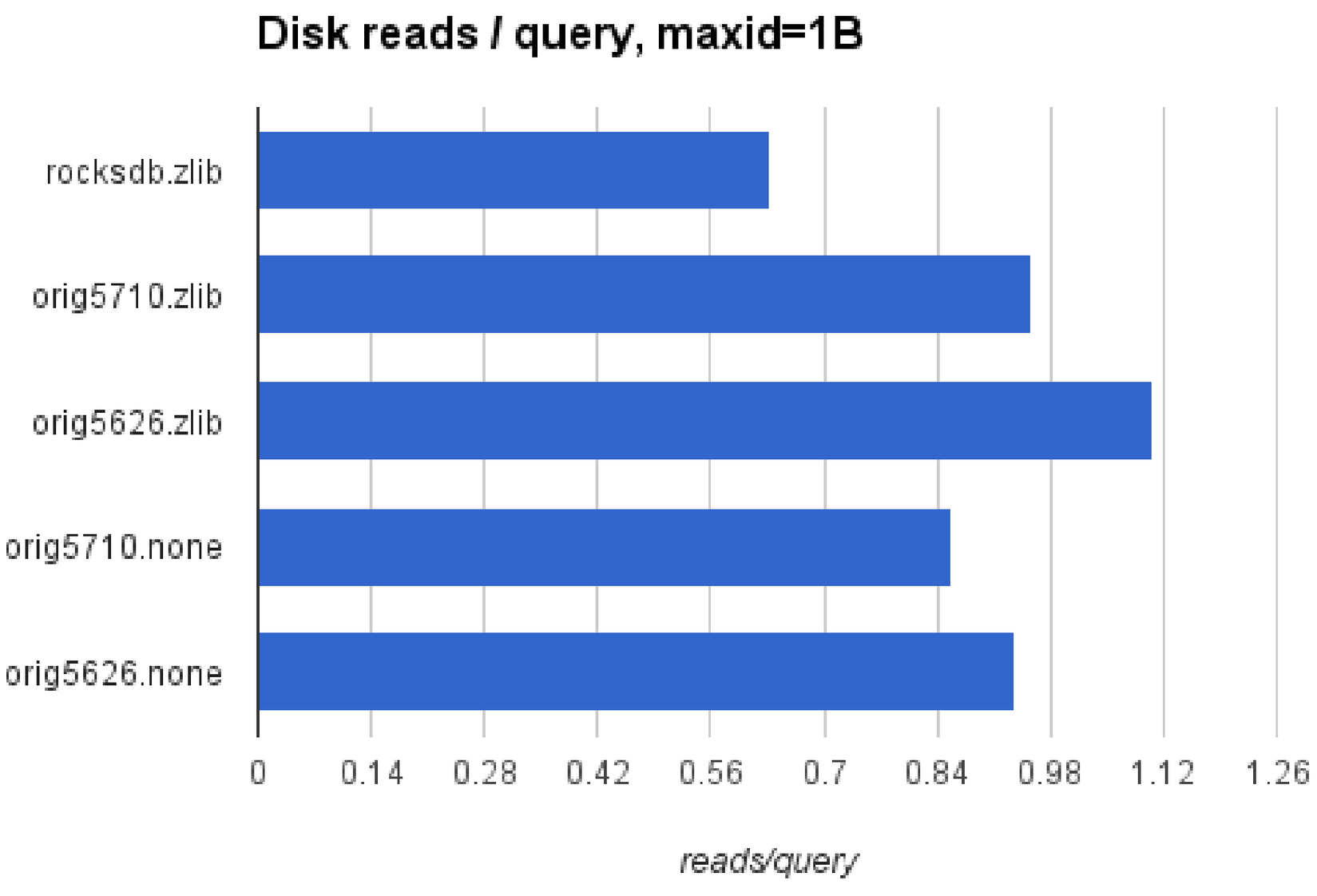
Database Size (Compression)



QPS

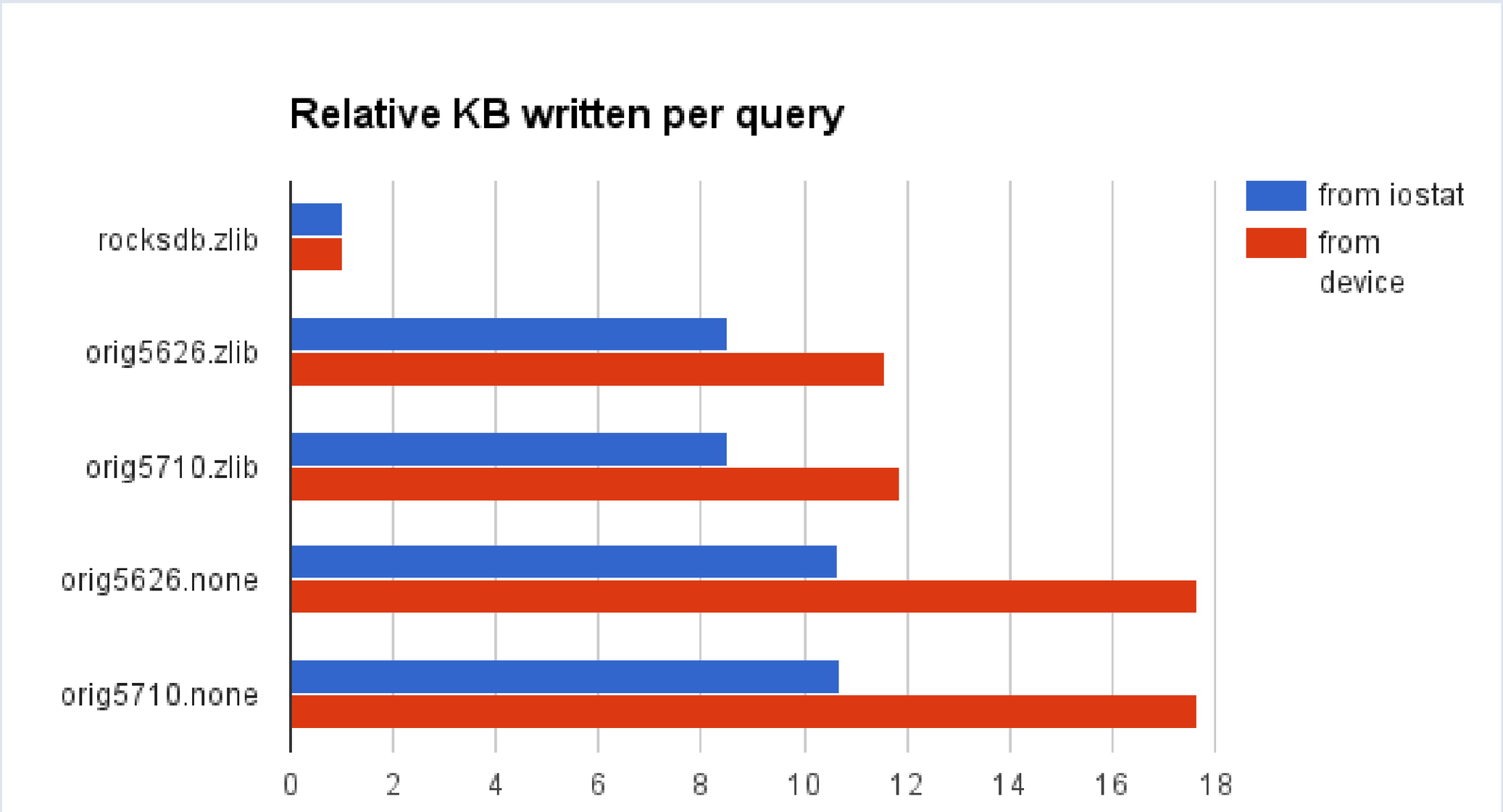

Query rate, SSD, maxid1=1B, 20 clients, 24 hours
36000
32000
28000
24000
20000
16000
12000
8000
4000
0
QPS
4
8
12
16
20
24
Hour
rocksdb.
zlib
orig5710.
none
orig5626.
none
orig5710.
zlib
orig5626.
zlib

Flash Reads per query



Smaller Space == Better Cache Hit Rate

Flush writes per query



Data Loading (migration)

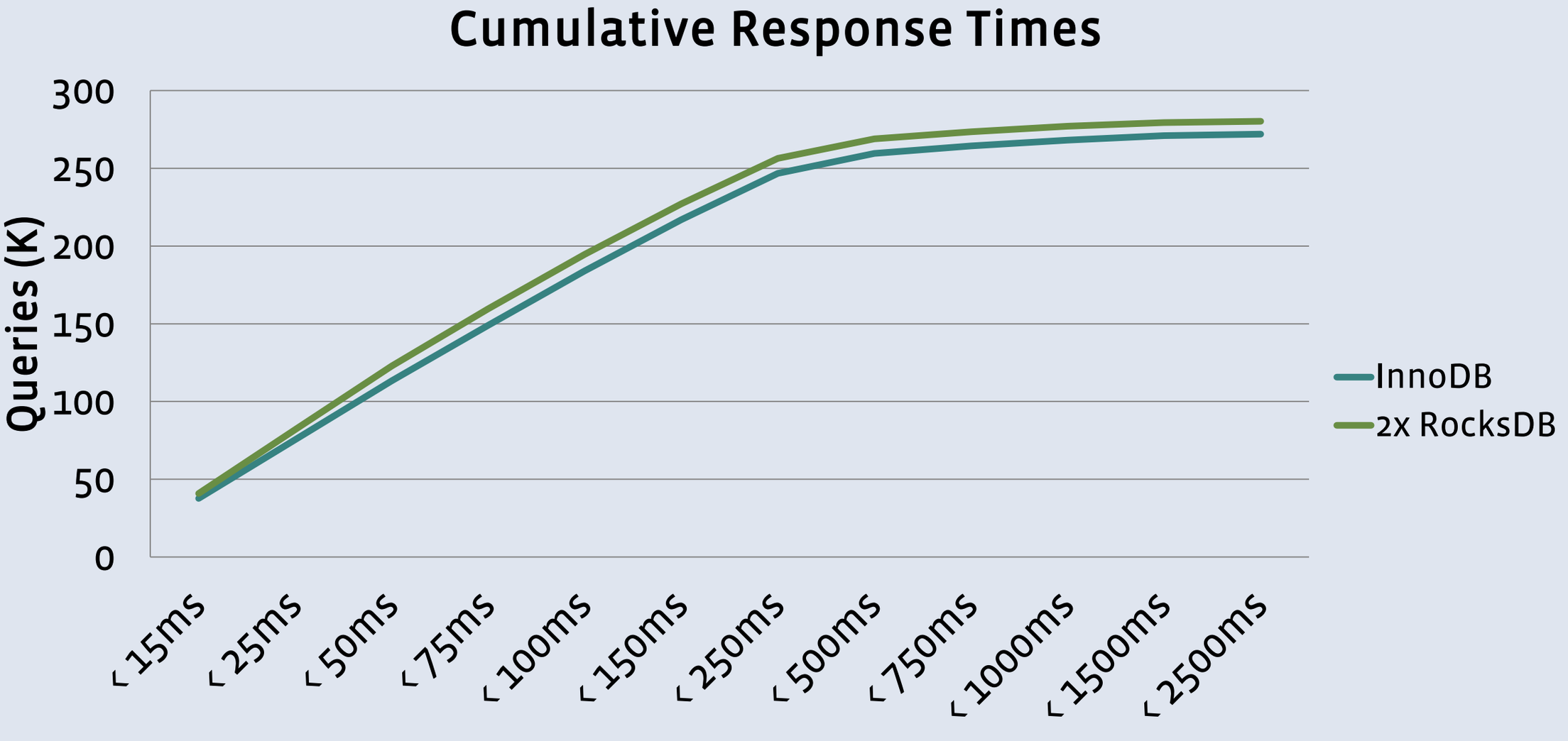
- Dump and Reload by mysqldump | mysql

- InnoDB to InnoDB: Logical Copy completed in 6:07:03
  - 1276GB to 1044GB (uncompressed)

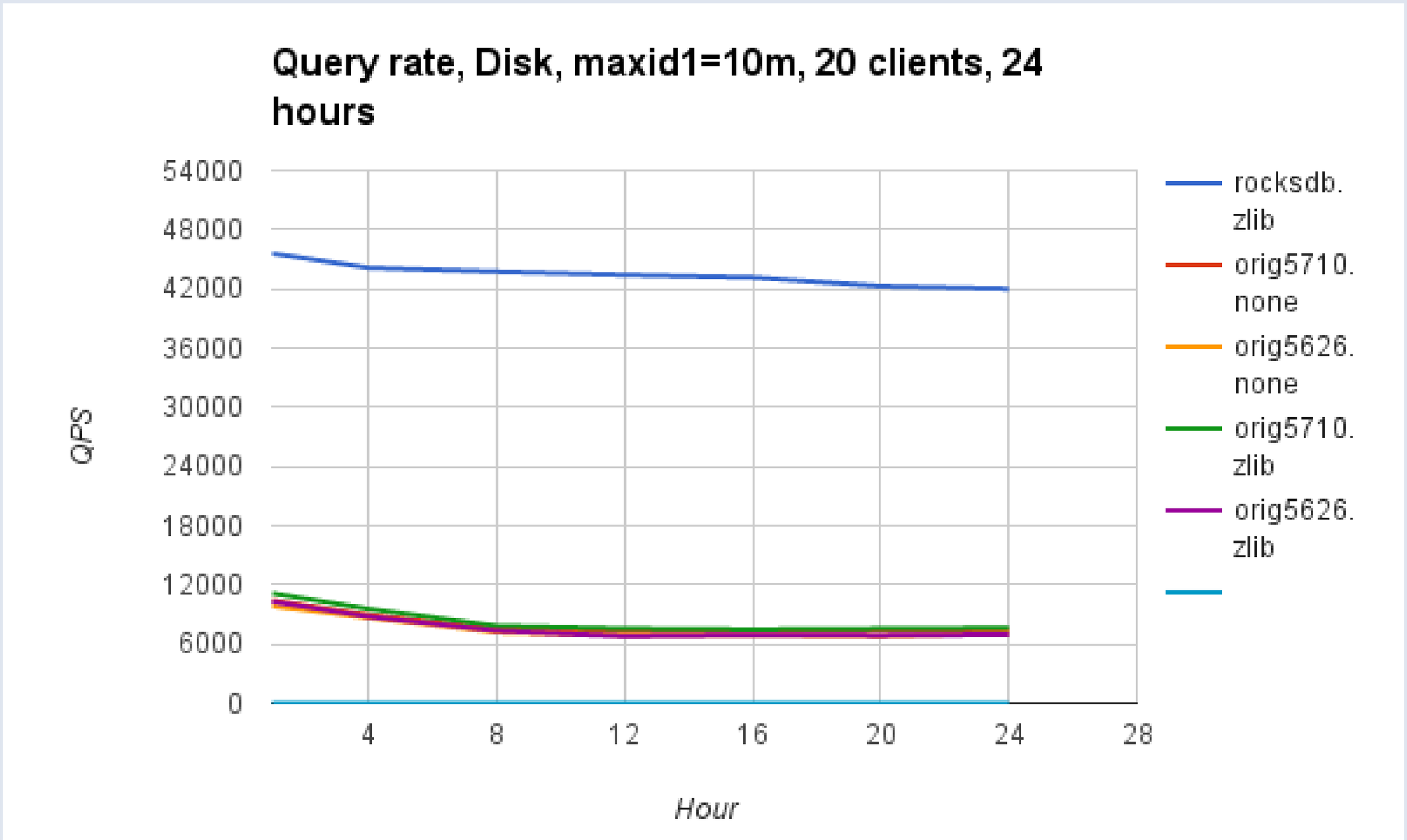
- InnoDB to MyRocks: Logical Copy completed in 4:20:17
  - 1276GB to 233GB (zlib L1 compressed)

Latency

- Slow query log to capture query times



On HDD workloads



Agenda

- MyRocks overview
- Getting Started
- Architecture
  - Data structure and table definition
  - Query optimizer and statistics
  - Row Locking and Concurrency
- Replication, Backup and Recovery
- Performance Tuning
- Monitoring

Getting Started

- Downloading Facebook MySQL 5.6 source code from GitHub
- Building MySQL binary by cmake
- 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

Downloading Facebook MySQL 5.6

- Everything is published as Open Source Software at https://github.com/facebook/mysql-5.6
- Added many features/enhancements
  - MyRocks, crash safe master/gtid, admission control, RBR, and many more
- Forked from official (Oracle) MySQL 5.6, rebased regularly
  - Currently rebased from 5.6.27 (three revisions older than the latest – not very old)
- Actively developed
- Not based on MySQL 5.7
  - Most features in MySQL 5.7 were not interesting to us, so we decided to skip. Will revisit in 5.8
  - We backported some really useful features for us – like Loss-Less Semisync

Building MySQL

```
$ git clone https://github.com/facebook/mysql-5.6.git
$ cd mysql-5.6
$ git submodule init
$ git submodule update
$ cmake . -DCMAKE_BUILD_TYPE=RelWithDebInfo -DWITH_SSL=system -DWITH_ZLIB=bundled -
DMYSQL_MAINTAINER_MODE=0 -DENABLED_LOCAL_INFILE=1
$ make –j24
$ make install
```

- See https://github.com/facebook/mysql-5.6/wiki/Build-Steps for details
- RocksDB (https://github.com/facebook/rocksdb/) is managed as submodule
- Optionally Snappy, LZ4, ZSTD and Bzip2 compression libraries can be added
  - “export WITH_SNAPPY=/usr” if libsnappy.a is located at /usr/lib/.
  - See storage/rocksdb/CMakeLists.txt for details

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

MyRocks in Depth

- MyRocks data structure
- Query Optimizer and Optimizer Statistics
- Row Locking and concurrency
- Backup
- Crash Recovery

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

RocksDB Key		RocksDB Value
Internal Index ID	Primary Key	The rest columns

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

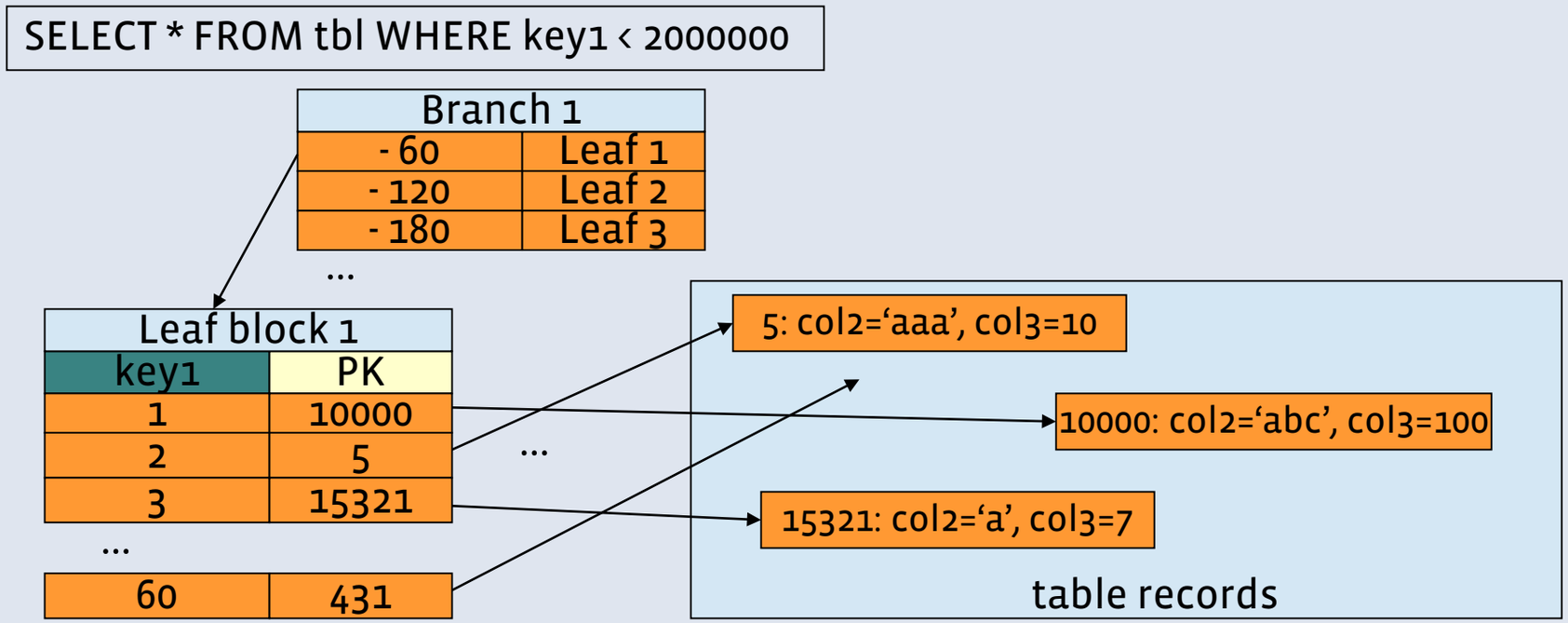
RocksDB Key			Value
Internal Index id	SK	PK	Value
257	10	1	Null
257	20	2	Null
257	30	3	Null
257	40	4	Null
257	50	5	Null

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

Index lookup/scan efficiency

- It's very similar to InnoDB since both InnoDB and MyRocks use clustered index
  - Single step read if looking up by primary key (i.e. WHERE pk = 1)
  - Covering Index: If queries using secondary key touch only secondary key + primary key fields, they don't read Primary Keys (no extra lookup)
    - Both InnoDB and MyRocks support covering index

Avoid non-covering secondary index scan



- When doing index scan, index leaf blocks can be fetched by *sequential* reads,
but *random* reads for table records are required
- The overhead is very high for both InnoDB and MyRocks
- Try to rewrite queries to use covering index range scan

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

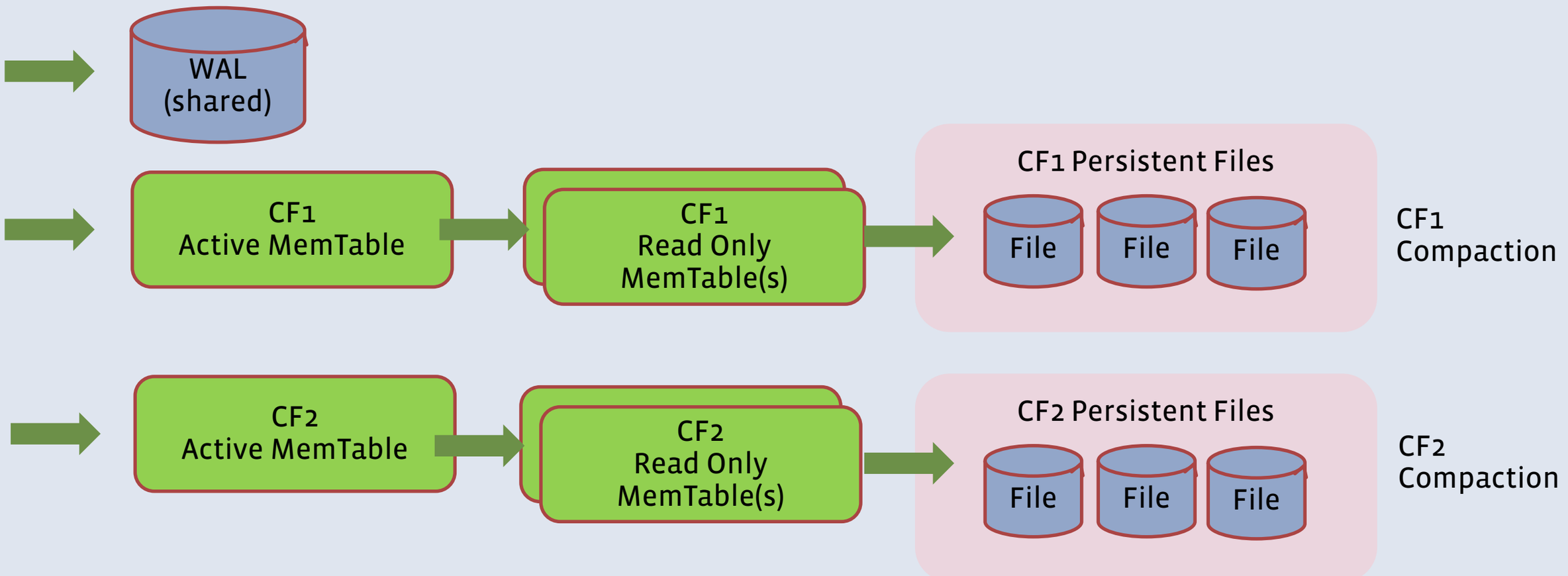
Index and Column Family

- 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
- Most RocksDB configuration parameters are per Column Family
  - MemTable, Bloom Filter, etc
- Different types of indexes should 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

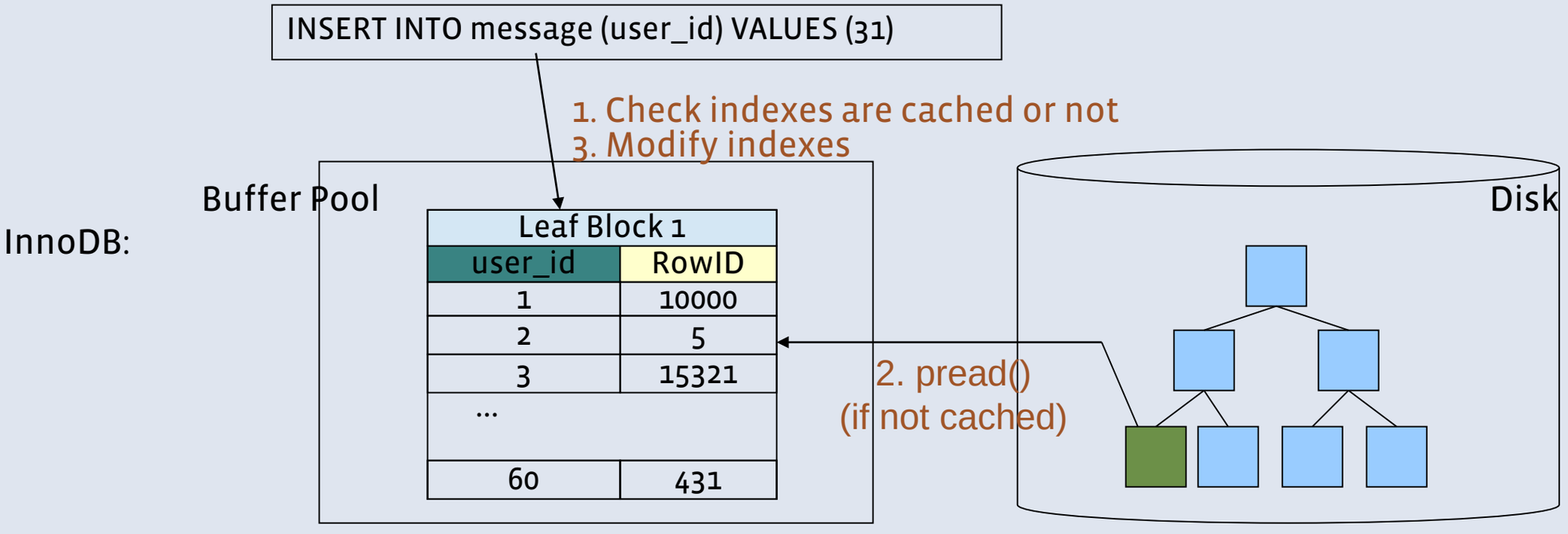
Query atomicity across different key spaces.

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



Index maintenance efficiency

- In MyRocks, non-unique Index does not need reads on index writes
  - Better write QPS especially if index does not fit in memory



INSERT INTO message (user_id) VALUES (31)

MyRocks:

MemTable/WAL 1. Put (user_id=31, pk=...)

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”

Write path

- Example: “UPDATE t SET c1=100 WHERE c1=1;”
  - There were 4 indexes – Primary key, i1 (c1), i2 (c2), i3(c2, c1)
  - Only 1 row was affected
- These write operations were issued into RocksDB at transaction commit
  - Put (primary key, c1=100) -- overwrite
  - SingleDelete (i1, c1=1)
  - Put (i1, c1=100)
  - SingleDelete(i3, c2=X, c1=1)
  - Put (i3, c2=X, c1=100)
  - Put (new binlog state) -- overwrite
  - Put (new relay log state) -- overwrite
- i2 was not affected because i2 didn’t include c1

Mem-comparable Keys and Case Sensitiveness

- MyRocks is optimized for CHAR/VARCHAR indexes using case sensitive collations, such as latin1_bin, utf8_bin and binary.
- Set collation-server=latin1_bin (or utf8_bin, binary) in my.cnf
- Default collation in MySQL is latin1_swedish_ci, which is not case sensitive
- MyRocks by default does not allow to have indexes with case insensitive collations
- You can optionally allow by setting rocksdb_strict_collation_exceptions='.*' in my.cnf

Case Sensitive collations

- Schema Definition
  - CREATE TABLE … ENGINE=ROCKSDB COLLATE latin1_bin;
- Less strict unique constraint
  - ‘AAA’, ‘aaa’, ‘aAa’ are treated different characters. Unique key error is not returned
- WHERE key=‘a’ no longer matches key ‘A’
- Sort ordering is changed
  - Case sensitive: ‘AAA’ -> ‘ABA’ -> ‘AaB’
  - Case insensitive: ‘AAA’ -> ‘AaB’ -> ‘ABA’
  - (A=0x41, B=0x42, a=0x61)
- Replication may stop if replicating from case insensitive master to case sensitive slaves
  - Storing ‘A’ -> DELETE FROM t WHERE key=‘a’ (deleted on master, ignored on slave) -> INSERT INTO t (key) VALUES (‘A’) (inserted on master, duplicate key error on slave)

Data Dictionary

- MyRocks stores some metadata information into RocksDB's dedicated column family named __system__
- Dictionary Examples
  - Table name => internal index id mapping
  - Internal index id => index metadata, Column Family id
  - Column Family id => Column Family flags (i.e. reverse order or not)
  - Binlog state
    - Binlog name, position, and GTID
    - Written at transaction commit
  - Index statistics
- Can be read via information_schema

Query optimizer, table and index statistics

- MyRocks automatically stores index statistics, so that MySQL can choose proper query execution plans

- MyRocks stores following estimated statistics
  - For each index:
    - Index name
    - Index size
    - Number of rows
    - Actual disk space
    - Number of deletes (tombstones)
    - For each index prefixes:
      - Distinct number of keys (for cardinality)

Where statistics are stored

- SST files
  - Table Property
    - RocksDB's extension format, allocated for each sst file
  - Written when SST files are created (flush and compaction)
    - Calculating statistics during flush/compaction, for all associated indexes for each sst
- Data Dictionary
  - Summary of these statistics, for each index

When index statistics are updated

- At MemTable Flush (automatic)
- At Compaction (automatic)
- ANALYZE TABLE (manual)

Index statistics in small tables

- Until MemTable is flushed, index statistics are not written
  - Typical MemTable size is 128MB, allocated for each Column Family
- If your MySQL instance is not updated, index statistics are not written
  - SHOW INDEX reports zero index cardinality
- ANALYZE TABLE triggers MemTable flush, which updates cardinality on small tables too

How MyRocks estimates range scan efficiency

- “SELECT * FROM t WHERE idx < 10000000”
  - How MyRocks decides to do full index scan or range scan?
- RocksDB has an API GetApproximateSizes() for specified ranges. MyRocks uses this function
  - Range Scan: GetApproximateSizes(begin, end) / average_row_length (from data dictionary) => Estimated number of rows scanned
  - Full Scan: Estimated total number of rows (from data dictionary)
  - MySQL optimizer uses these information

MyRocks specific optimizer features/limitations

- Need to execute ANALYZE TABLE on small tables
- No “index dive” overhead
- Index Condition Pushdown is supported
- Multi Range Read (MRR) is not supported yet

MyRocks in Depth

- MyRocks data structure
- Query Optimizer and Optimizer Statistics
- Row Locking and concurrency
- Backup
- Crash Recovery

MyRocks row locking

- Lock granularity in MyRocks is row
- Locks are released at the end of the transaction (commit or rollback)
- All locking states are kept in memory
  - Can’t modify billions of rows within one transaction
- Shared level lock is not supported yet
  - SELECT .. LOCK IN SHARE MODE holds exclusive lock (as .. FOR UPDATE)
- Gap lock support is very limited
  - Rows not found are not locked, except when using all primary keys in WHERE condition
- Row based binary logging (binlog_format=ROW) must be used
- Read Uncommitted and Serializable isolation levels are not supported
- Automatic deadlock detection is not supported
- SAVEPOINT is not supported

Primary key locking reads

- SELECT * FROM t WHERE id = 1 FOR UPDATE;
- GetForUpdate(id=1) – reserving a row lock for id=1 only
- Nobody else can do locking read or insert for the same id
- id1=1 is locked even if the row does not exist
- This is the only Gap Lock pattern that MyRocks supports
  - MyRocks holds gap lock when using all primary key columns in WHERE clause
- Used for unique key checking

Releasing locks for rows scanned but not matched

- "SELECT * FROM t WHERE non_indexed_column = 1 FOR UPDATE"
- InnoDB with Statement based Binlog: Does full table scan, lock all rows
- InnoDB with Row based Binlog + Read Committed Isolation: Does full table scan, lock rows, release rows immediately if non_indexed_column != 1
  - After the SELECT completes, only rows with non_indexed_column = 1 are locked
- MyRocks: Same as InnoDB with RBR + RC

Secondary index locking reads

- SELECT sk FROM t WHERE sk = 1 FOR UPDATE;
- Locking ordering is Secondary key -> Primary key
  - Locking the secondary key where sk=1
  - Reading primary key from the secondary key (extended key)
  - Locking the primary key so that nobody can modify the row
  - Reading primary key values is skipped thanks to covering index
  - Other secondary keys are not locked
- All locking reads lock primary key
- Unique Constraint + Secondary Key is also supported

Transaction Isolation Differences compared to InnoDB/PostgreSQL

- MyRocks transaction isolation implementation is close to PostgreSQL's style – snapshot isolation
  - Behavior is very close, though there are some minor differences
- There are some behavior differences between InnoDB and PostgreSQL/MyRocks
- Locking reads (UPDATE, SELECT FOR UPDATE, etc) read current data in InnoDB, regardless of isolation levels
- Locking reads in MyRocks and PostgreSQL read from snapshot
  - Read Committed: Snapshot is created at each statement
  - Repeatable Read (PostgreSQL): Snapshot is created at the beginning of the transaction
  - Repeatable Read (MyRocks): Snapshot is created at the first statement of the transaction
- https://github.com/ept/hermitage is a great resource to understand with examples

Multi-Statement Transactions

- Client 1
  BEGIN;
  INSERT INTO linktable (id1=1, id2=1, link_type=1)

  Client 2
  BEGIN;
  INSERT INTO linktable (id1=1, id2=2, link_type=1)

  Client 1
  INSERT INTO counttable (id1=1, link_type=1) ON DUPLICATE KEY UPDATE…
  COMMIT;

  Client 2
  INSERT INTO counttable (id1=1, link_type=1) ON DUPLICATE KEY UPDATE
  ???

  Trying to update a row that was changed by somebody after starting my transaction
  => Allowed with MyRocks RC, but not allowed with MyRocks. InnoDB allows both

- InnoDB RC/RR: Client 2 overwrites client 1
  PostgreSQL RC: Client 2 overwrites client 1
  MyRocks RC: Client 2 overwrites client 1
- PostgreSQL RR: ERROR: could not serialize access due to concurrent update
- MyRocks RR: ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction

Gap Lock

- MyRocks has very limited Gap Lock support – only when using all primary keys in WHERE condition
- InnoDB vs MyRocks major locking differences
  - Update with Select (INSERT/CREATE .. SELECT, UPDATE .. SELECT etc) locks rows scanned by select in InnoDB. MyRocks does not by default
  - Locking reads in InnoDB lock rows that don't exist. MyRocks has limited support for it.
    - InnoDB can lock in range – i.e. blocking to insert any id greater than 10. MyRocks can't do that
- MyRocks needs Row Based Binary Logging
  - It returns errors on update when using SBR, except SQL threads
  - Set "rocksdb_unsafe_for_binlog=true" if you're sure your queries are 100% safe with SBR

Why lack of Gap Lock support needs RBR

1) DELETE FROM t WHERE value = 9;

id	value
1	0
10	9
100	1
1000	0

2) UPDATE t SET value = 9 WHERE id = 1;

1) Checked id = 1 first, not matched, unlocking if gap lock is not supported
2) Updated value = 9 where id = 1, committed, written to binlog
1) Finished delete statement. id=10 was deleted. Written to binlog
Final Result: id=1 exists, value=9. id=10 was deleted

Binary log:
1. UPDATE t SET value = 9 WHERE user_id=1;
2. DELETE FROM t WHERE status = 9;
=> Both id=1 and 10 are deleted on slaves

Data Consistency is broken. Row based Binary logging & Replication (RBR) can prevent this issue

Gap Lock

- When migrating from InnoDB to MyRocks:
  - Fewer row lock contentions than InnoDB
  - For some queries relying on gap lock, you may get “Deadlock” errors in MyRocks
  - For some queries relying on gap lock, MyRocks does not work at all
  - We added session variables for handling gap lock gracefully
    - “gap_lock_raise_error” -- raising errors if queries relying on gap lock
    - “gap_lock_write_log” – writing into log file for queries using gap lock

Queries relying on Gap Lock – empty check

link table has primary key on (id1, type, id2)
user table has primary key on (id)
If matching (id1, type) does not exist in link, set link_exist=0 on user
After inserting into link, set link_exist=1 on user

MyRocks:

InnoDB:

```
BEGIN;
SELECT * FROM link WHERE id1=2 AND type=2 LOCK IN SHARE MODE;
 => Empty Set
```

```
BEGIN;
INSERT INTO link (id1, type, id2) VALUES (2, 2, 3);
=> Not blocked in MyRocks, because of no Gap Lock
UPDATE user SET link_exist=1 WHERE id=2;
COMMIT;
```

```
UPDATE user SET link_exist=0 WHERE id=2;
=> Getting “Deadlock” error in MyRocks, with Repeatable Read
(if UPDATE was done, the row became inconsistent)
```

```
BEGIN;
INSERT INTO link (id1, type, id2) VALUES (2, 2, 3);
 => Blocked in InnoDB, because of Gap Lock
```

```
UPDATE user SET link_exist=0 WHERE id=2;
COMMIT;
=> INSERT proceeded
```

```
UPDATE user SET link_exist=1 WHERE id=2;
COMMIT;
```

- MyRocks does not implicitly overwrite changes after creating a snapshot, with Repeatable Read isolation. This rarely causes unexpected data inconsistency
- With MyRocks, applications will get more errors than InnoDB

Queries relying on Gap Lock – empty check

link table has primary key on (id1, type, id2)
user table has primary key on (id)
If matching (id1, type) does not exist in link, set link_exist=0 on user
After inserting into link, set link_exist=1 on user

MyRocks:

```
BEGIN;
SELECT * FROM link WHERE id1=2 AND type=2 LOCK IN SHARE MODE;
 => Empty Set

BEGIN;
INSERT INTO link (id1, type, id2) VALUES (2, 2, 3);
=> Not blocked in MyRocks, because of no Gap Lock

UPDATE user SET link_exist=0 WHERE id=2;
COMMIT;

UPDATE user SET link_exist=1 WHERE id=2;
=> Getting “Deadlock” error in MyRocks, with Repeatable Read
```

Queries relying on Gap Lock – prefix uniqueness

link table has primary key on (id1, type, id2)
Want uniqueness on prefix (id1=1 and type=10)

InnoDB:

```
BEGIN;
SELECT * FROM link WHERE id1=1 AND type=10 FOR UPDATE;
=> Returning Empty Set, holding gap lock on (id1=1, type=10)
=> Nobody else can insert a row starting with (id1=1, type=10) until transaction ends, thanks to Gap Lock

INSERT INTO link (id1, type, id2) VALUES (1, 10, 150);
COMMIT;
```

Other transactions:

```
BEGIN;
SELECT * FROM link WHERE id1=1 AND type=10 FOR UPDATE;
=> Row exists
UPDATE link SET … WHERE id1=1 AND type=10 AND id2=150;
COMMIT;
```

This does NOT work with MyRocks – will end up inserting many rows starting with (id1=1, type=10)

Queries relying on Gap Lock – prefix uniqueness

link table has primary key on (id1, type, id2)
Want uniqueness on prefix (id1=1 and type=10)

MyRocks holds gap lock if using ALL primary key columns. Unique prefix can be done by this.

MyRocks:

```
BEGIN;
SELECT * FROM link WHERE id1=1 AND type=10 AND id2=0 FOR UPDATE;
=> Returning Empty Set, holding gap lock on (id1=1, type=10, id2=0)
=> Nobody else can proceed SELECT FOR UPDATE with (id1=1, type=10, id2=0). id2 =0 is fixed dummy id.

INSERT INTO link (id1, type, id2) VALUES (1, 10, 150);
COMMIT;
```

Other transactions:

```
BEGIN;
SELECT * FROM link WHERE id1=1 AND type=10 AND id2=0 FOR UPDATE;
=> Row exists
UPDATE link SET … WHERE id1=1 AND type=10 AND id2=150;
COMMIT;
```

Queries relying on Gap Lock – blocking queue

- Gap Lock can be used to block inserting any value greater (or smaller) than X

link table has primary key on (id1, type, id2)

```
BEGIN;
SELECT * FROM link WHERE id1=100 AND type=10 ORDER BY id2 DESC LIMIT 1 FOR UPDATE;
=> Returning id2 = 20


BEGIN;
INSERT INTO link (id1, type, id2) VALUES (100, 10, 21);
=> Blocked
(can't insert id=22 or greater either)
```

MyRocks does not support this

Queries relying on Gap Lock – pt-table-checksum

- REPLACE INTO percona.checksums SELECT * FROM app_tables WHERE key > X and key < Y;
- Running the same statement on both master and slaves via replication, then comparing results (checksums)
- This relies on Gap Lock so it doesn't work with MyRocks
- We're checking table checksum with different algorithm at Facebook

Locking rows that do not exist (1)

- create table to (id1 int, id2 int, value int, primary key(id1, id2));
  insert into to values (1,1,0),(3,3,0),(4,4,0),(6,6,0);

  T1:
  begin;
  select * from to where id1=1 for update;

  T2:
  begin;
  select * from to where id1=1 and id2=4 for update;
  insert into to values (1,5,0);

- MyRocks RR/RC: succeeds
  InnoDB RBR+RC: succeeds
  InnoDB RR: T2 blocked by T1
  PostgreSQL RR/RC: succeeds

  Only InnoDB supports this (Gap Locking). MyRocks and PostgreSQL behavior are expected.

Locking rows that do not exist (2)

- create table to (id1 int, id2 int, value int, primary key(id1, id2));
  insert into to values (1,1,0),(3,3,0),(4,4,0),(6,6,0);

  T1:
  begin;
  select * from to where id1=1 and id2=5 for update;

  T2:
  begin;
  insert into to values (1,5,0);

  T3:
  begin;
  select * from to where id1=1 and id2=5 for update;

- PostgreSQL RC/RR: succeeds
  InnoDB RR: T2/T3 blocked by T1
  InnoDB RBR+RC: succeeds
  MyRocks RC/RR: T2/T3 blocked by T1

After creating a snapshot, other clients updating rows

- create table t (id int primary key, value int);
  (t had 10M rows, all values were zero)

- T1:
  select * from t where value > 0 for update;
  (taking long time)

T2:
update t set value=value+1 where id=5000000;
commit;
(starting T2 just after starting T1. finishing before T1 touches the row)

- InnoDB:
  RC/RR: T1 returned id=5000000 (returning rows updated after creating a snapshot at the beginning of the long running select for update)
- PostgreSQL:
  RC: T1 returned id=5000000
  RR: ERROR: could not serialize access due to concurrent update
- MyRocks:
  RC/RR: ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction

Replication, Backup and Recovery

- Replication
  - Row Based Binary Logging and Replication (RBR)
  - Read Free Replication
  - Facebook's Replication Enhancements
- Crash Safety
  - Slave recovery
  - Master failover
- Backup
  - Online logical backup
  - Consistent Snapshot and long running transactions
  - Online binary backup

RBR overview

- MyRocks needs binlog_format=ROW
  - Statement based binary logging (SBR) may cause data mismatch between master and slaves without full Gap Lock support, so MyRocks raises errors when using SBR
- It is possible to use binlog_format=STATEMENT on slaves
  - Can be useful when replicating from InnoDB with statement based binary logging
- MyRocks returns errors on SBR, except from replication threads
  - Set rocksdb_unsafe_for_binlog=1 if you are sure it is safe

RBR advantages

- Generally better replication speed because no query parsing is needed
- No statement is unsafe for row based binary logging
- Costly queries that change little are efficiently applied
- Data can be consumed by non-MySQL consumers

RBR disadvantages

- Many statements are currently unsafe for RBR
  - small queries producing large changes
  - queries on tables without a usable key
- lack of visibility into replication progress
  - events contain incomplete type information
    - external consumers don't have necessary data available
    - schema mismatches between master and slave
- binary logs may be substantially larger
- internal type changes may cause incompatibility
- triggers don't run on slaves when applying changes
- modifying no row won't be written to binlog

Notable differences between SBR and RBR

- data mismatches between master and slave are unacceptable
  - default -- both duplicate key error and no data found become error
  - SBR -- duplicate key error stops slave, no data found does not
  - slave_exec_mode=IDEMPOTENT -- both duplicate key error and no data found are skipped
  - slave_exec_mode=SEMI_STRICT (Facebook extension) -- only no data found is ignored (compatible with SBR)
- dropping columns on slave will cause replication to stop
  - Errors like "Column 2 of table 'test.a' cannot be converted from type 'enum' to type 'int(11)'" (FB extension: log_column_names=ON)

RBR configurations

- binlog_format=ROW
- binlog_rows_query_log_events=ON
  - “mysqlbinlog –vv --base64-output=decode-rows” can print original statements
- binlog_row_image=FULL
- slave_type_conversions=ALL_NON_LOSSY
- log_only_query_comments=OFF (FB extension)
- slave_exec_mode=SEMI_STRICT (FB extension)
- log_column_names=ON (FB extension)
  - Changes binlog format
- slave_run_triggers_for_rbr=YES (MariaDB extension, FB backported)
- sql_log_bin_triggers=OFF (MariaDB extension, FB backported)

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 setting “rocksdb_skip_unique_check=1”

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
  - Delete/SingleDelete (BI keys)
  - Put (AI)
- Eliminates random read overhead on update/delete
- Can be configured by rocksdb_rpl_lookup_rows parameter

Facebook's Replication Extensions

- GTID + Multi-Threaded Slave (MTS) + Reduced Durability
  - Replication states are written to mysql.slave_gtid_info table (transactional table, installed by mysql_install_db in fb-mysql)
  - Made crash safe slave work with GTID
  - slave-transaction-retries working with MTS
  - Changed Binlog Index format so that GTID auto positioning faster
  - RBR (described before)
  - Semisync (backporting Loss-Less from 5.7, semisync mysqlbinlog)
  - And many more…

Crash Safety

- How to recover MyRocks instances when they crash, without restoring entire data from other instances
  - Restoring TBs instances is painful
  - We want to recover by 1. restarting mysqld, then 2. start slave
  - We call it as “Crash Safe” – can be recovered without full restore
- MyRocks is designed to be crash safe

Crash Scenario and how to recover MyRocks

- Slave Failure
  - MyRocks supports “Crash Safe Slave”. You can just restart mysqld instance then MySQL recovers everything
  - my.cnf
    - relay_log_recovery=1
- Master Failure
  - This is much more complex than Slave Failure, but it’s possible to make crash safe

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)

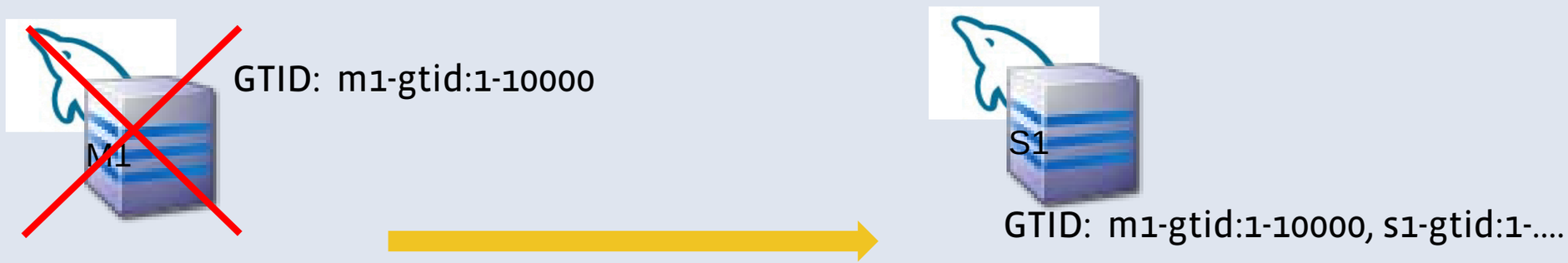
WAL entry 3

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 Safe Master with GTID and semisync

- What is crash safe master?
  - When master is down and promoting a slave, after the crashed master's recovery, we want to add the crashed master as a new slave without rebuilding the whole crashed master instance
  - Recovery (recovering from OS reboot etc) shouldn't take days. Applying a few minutes/hours of binlogs is much faster than copying and rebuilding the entire instance



After crashed master's recovery, continue replication by
CHANGE MASTER TO MASTER_HOST='S1', MASTER_AUTO_POSITION=1;

Difficulties of the crash safe master

GTID: m1-gtid:1-10000, s1-gtid:1-....

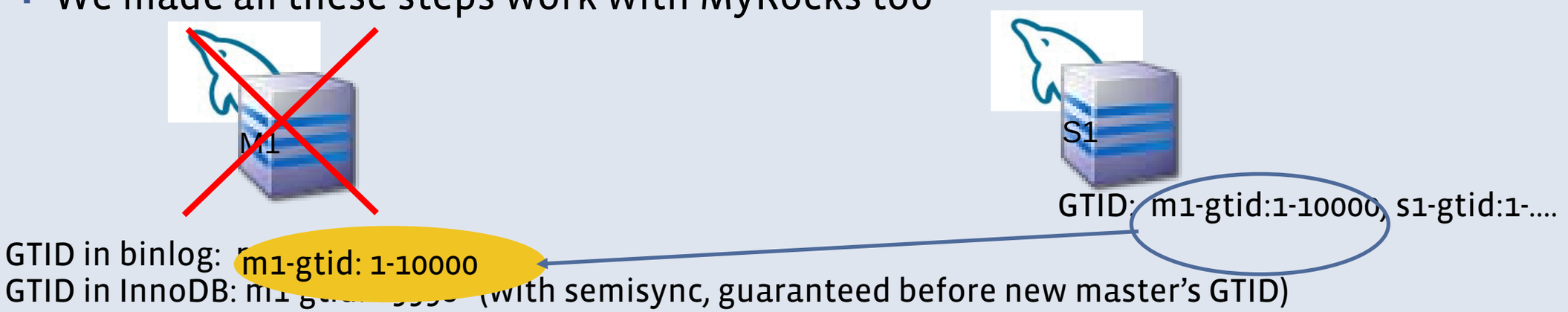
GTID in binlog: m1-gtid:1-10005
GTID in MyRocks: with loss-less semisync, guaranteed before new master's GTID

- If orig master's GTID is ahead new master's, the orig master can't join as a slave
  - Because new master doesn't have the data

- GTID in binlog may ahead semisync slaves (even with loss-less semisync), and crash recovery is done by GTID in binlog
  - During crash recovery, prepared state transactions are rolled forward
  - We needed to trim orig master's binary logs, so that prepared state transactions are rolled back

Detailed crashed master recovery steps

- Before restarting crashed master's mysqld process, remove binary logs
  - InnoDB has prepared state transactions that were not sent to any slave yet. If matching binlog events exist, they will be rolled forward
- Find last committed GTID on the crashed master, and set GTID_PURGED
  - We parse mysql.err log by Python script, then finding the last committed GTID
- Apply missing binlog events from new master
  - Either by mysqlbinlog or replication with --replicate-same-server-id
- We made all these steps work with MyRocks too



MyRocks limitations around crash safety

- MyRocks currently does not support XA between binary logs and RocksDB, so you may lose data or cause data inconsistency on master machine failure
  - MyRocks by default does not call fsync() at commit. By setting rocksdb_use_fsync=1, it calls fsync(), but it doesn't help to prevent inconsistency because of lack of XA support
  - All metadata operations (i.e. adding indexes) are always synchronized
- With GTID and Loss-Less Semi-Synchronous Replication, you can failover without data loss, without data inconsistency
  - You need to promote a slave. Do not wait for the dead master to recover
  - Regular (non-Semisync) failover may cause data loss but can prevent data inconsistency
- We're working on supporting XA and full durability with good enough performance in MyRocks. We're going to implement:
  - 2PC in RocksDB
  - Group commit support in MyRocks

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

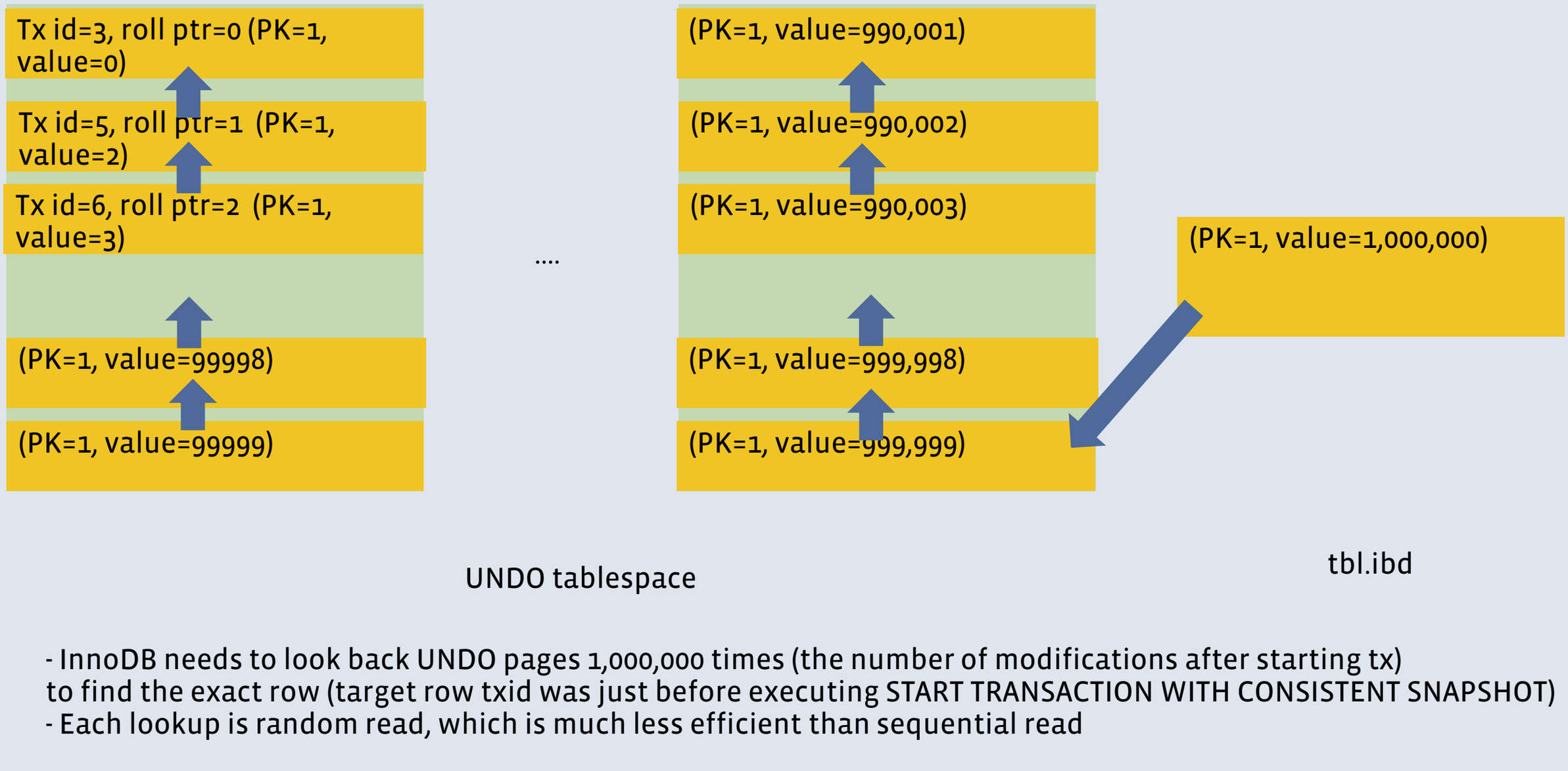
- SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
- START TRANSACTION WITH CONSISTENT ROCKSDB SNAPSHOT
  - New syntax (Facebook patch)
  - Get an internal binlog lock so that nobody can write to binlog
    - Much lighter than FLUSH TABLES WITH READ LOCK
  - Create RocksDB snapshot
  - Get current binlog 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		

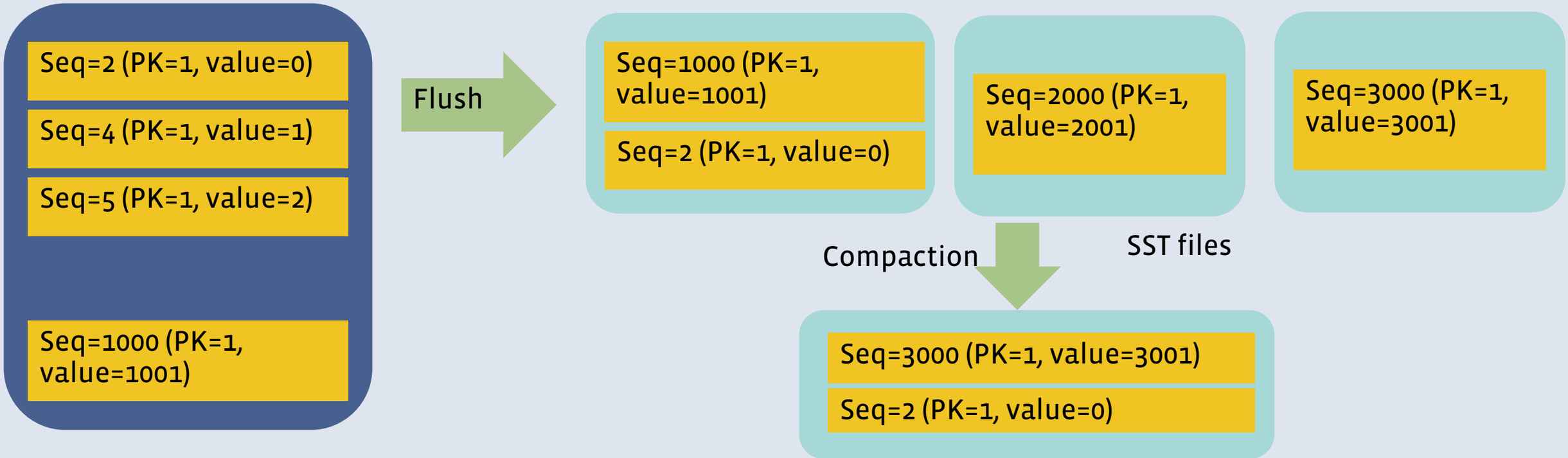
How snapshot works in InnoDB



- 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



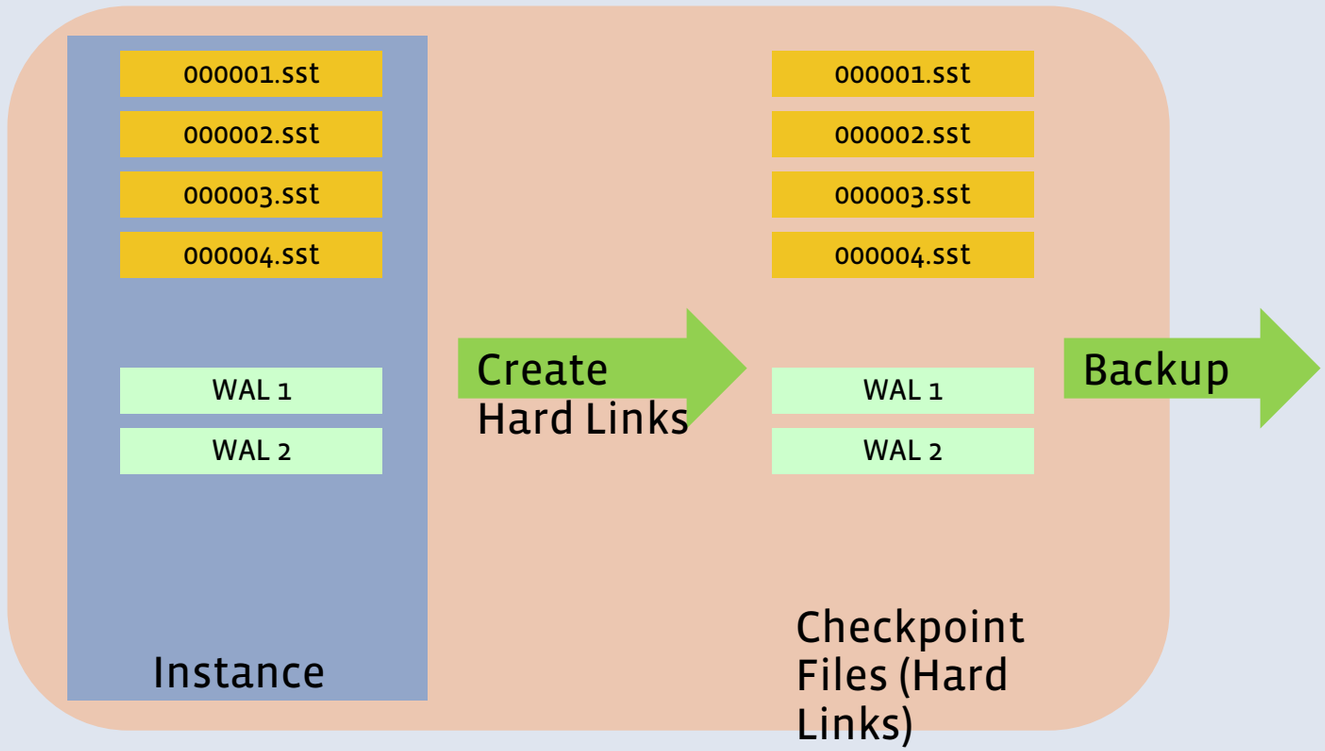
All intermediate rows were removed during flush/compaction
because RocksDB could know they were definitely not needed

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

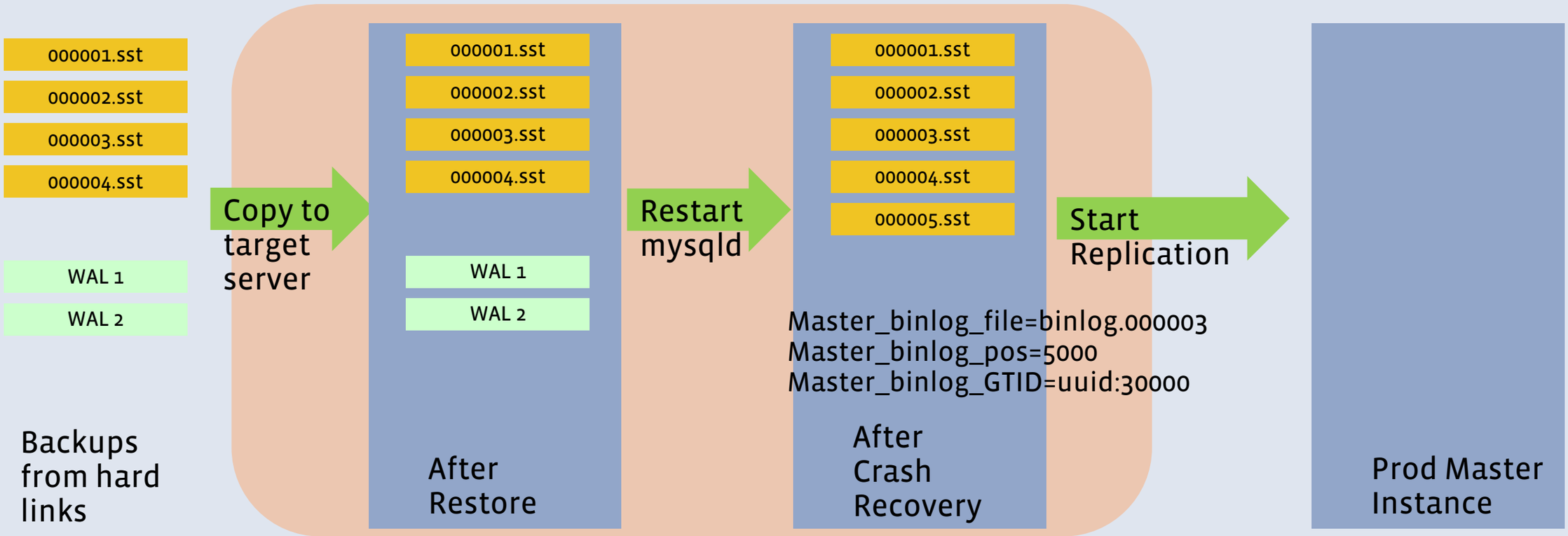
- MyRocks provides online binary backup solutions and tools
- Online binary backup is useful for:
  - Creating new slave instances much faster than logical backup
  - Restoring from backup much faster than logical backup
  - Currently only full binary backup is possible. Partial or incremental binary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

How MyRocks online backup works



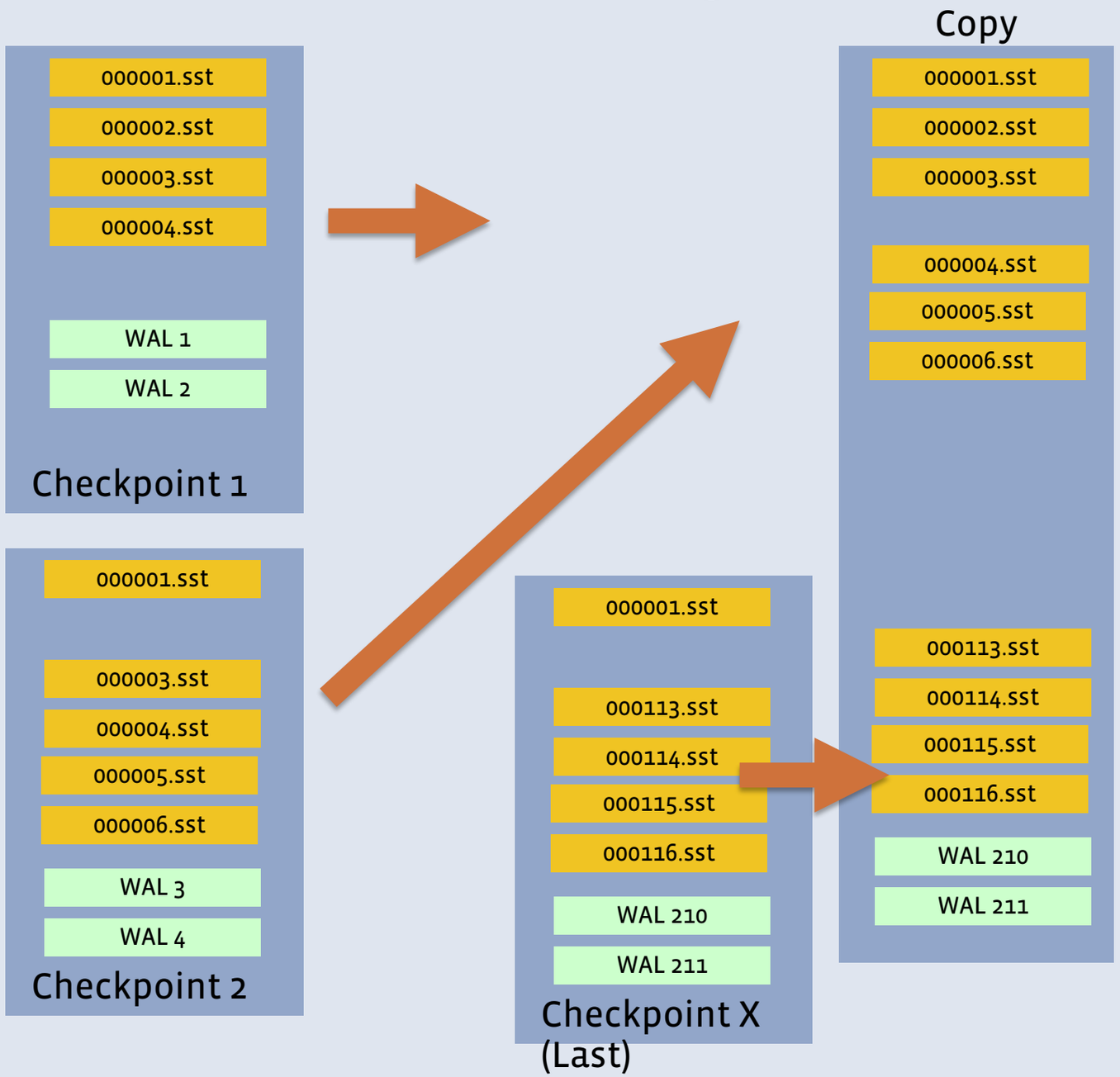
- Create hard links for sst files -- on RocksDB directory ($datadir/.rocksdb/*.sst)
- Then backup all Hard Link files and WAL files somewhere (local or remote storage)
  - Hard Link files (.sst) are immutable. So source instances do not have to be stopped
- MyRocks has a special syntax “SET GLOBAL rocksdb_create_checkpoint = /path/to/backup” to create hard links and to copy WAL files
  - WAL files are mutable (append only) but become consistent after starting instance

How to restore and recover from backups



- MyRocks writes binlog state and replication state into WAL at commit
- After crash recovery, the last replication state is reflected at mysql.slave_relay_log_info or mysql.slave_gtid_info
- You can restart replication from the last replication state
- The binlog state is the last snapshot time. If long time has passed after that, it will take very long time to sync up with master. You can't recover if master doesn't have necessary binlogs.

Tricks: Renewing Checkpoints



- After enough time has passed since last checkpoint, destroy it and create another checkpoint, then copy from the new checkpoint
- Copy only sst files at intermediate checkpoints
- Skip copying sst files if they were already copied during previous checkpoints
- At the last checkpoint, copy the rest files (WAL, manifest, etc)
- Binlog/Replication state is at the last checkpoint time. Replication sync up time can be much shorter and not dependent on instance size
- Automating these steps will help

myrocks_hotbackup: MyRocks online backup tool

- myrocks_hotbackup is a tool to take online binary backup
- Automates all of the algorithms described in previous slides
- Included in fb-mysql 5.6 (under scripts/), Open Source, written in Python
- Works as “Streaming Backup”
  - Can send backup files to remote servers, without writing locally
  - After streaming backup, “move-back” step (locating WAL, SST, frm, and metadata files properly) is needed
  - “tar” and “xbstream” are supported streaming options. xbstream is more recommended to prevent burst writes on destination
- Supports “WDT”, faster network transfer method
  - https://github.com/facebook/wdt

Online binary backup with myrocks_hotbackup

- Major restrictions
  - Source MySQL instance must be running
  - You can’t take backup from other than RocksDB storage engine
- Major benefits
  - You don’t need much extra space on source machine
  - Apply-log phase is extremely short
- Example usage:
  - [src]$ myrocks_hotbackup --user=root --port=3306 --checkpoint_dir=/data/backup --stream=xbstream | ssh $dst ‘xbstream –x /data/backup’
  - [dst]$ myrocks_hotbackup --move_back --datadir=/data/mysql --rocksdb_datadir=/data/mysql/.rocksdb –rocksdb_waldir=/txlogs --backup_dir=/data/backup

How myrocks_hotbackup works (1)

- Create a checkpoint, start backup round
  - SET GLOBAL rocksdb_create_checkpoint= '$checkpoint_dir/$i'
    - RocksDB creates hard links from data directory to checkpoint directory
  - Backup SST files one by one
    - Taking advantage that SST file is immutable
    - Streaming backup (tar or xbstream with ssh) is supported
- Delete a checkpoint if
  - Some time has passed (every --interval seconds)
  - Backed up all files
- If SST backup has not finished, re-create a checkpoint and repeat the same steps
  - myrocks_hotbackup remembers previously sent SST files, and skips sending the same file
  - As far as backup speed is faster than SST generation speed, backup will end eventually

How myrocks_hotbackup works (2)

- Repeating “Create checkpoint” and “Delete checkpoint” cycles
- After backup all SST files, backup the rest files
  - WAL files, *.frm files, other metadata files
  - These are way smaller than SST files
- Impact of multiple checkpoints
  - Necessary WAL files do not depend on data size (much shorter apply-log time)
  - May send some SST files that are no longer used (i.e. deleted by compaction)
- Move-back
  - Put WAL files under rocksdb_wal_dir
  - Put *.frm files under datadir
  - Put SST and Manifest files under rocksdb_datadir

Starting instance

- Starting mysqld
  - Crash recovery happens – applying WAL files
    - By moving checkpoint frequently, the number of WAL files can be reduced
    - Smaller number of WAL files can reduce crash recovery time
  - Unnecessary SST files are automatically removed at mysqld start
  - Binlog state is printed into mysql *.err log

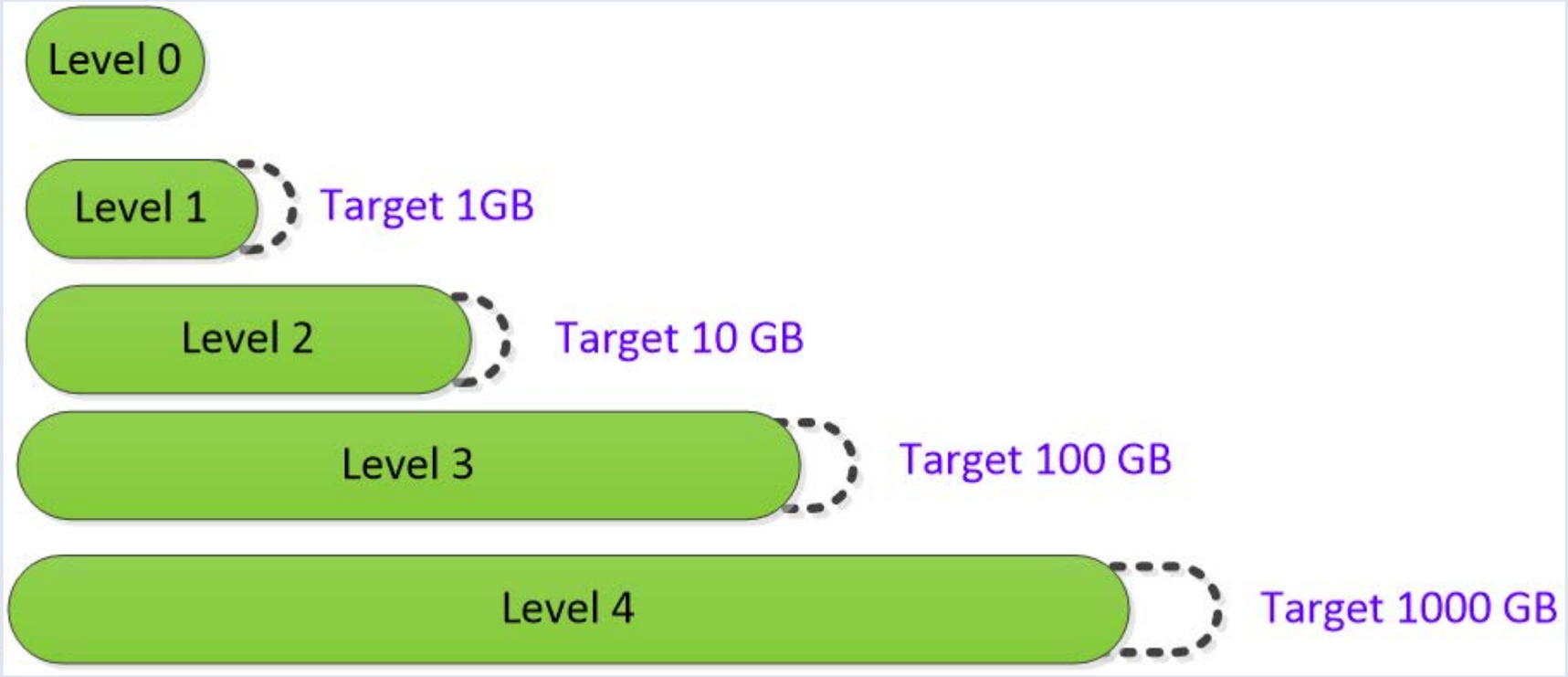
Configuring replication from backup

- Backup source's binlog state is written to *.err log on destination
- Connect to source MySQL, then run
  - SHOW GTID EXECUTED IN '$binlog_file' FROM $binlog_pos;
    - Returns GTID position where the destination instance should start replication
- Configure replication on destination
  - SET GLOBAL GTID_PURGED='$gtid';
  - CHANGE MASTER TO MASTER_HOST='$master', MASTER_AUTO_POSITION=1;

Performance Tuning

- Reverse order column family
  - Useful if the index is intensively used for descending range scan
- Space and Compression
- Bloom Filter
- Data Loading
- my.cnf configuration examples

Space and Compression



- Estimated steady-state total size is 1.11 * (size of the bottommost level)
- Compression algorithm can be configured per level

Leveled LSM and Compression basics

- First of all, compression and decompression is different☺
- Compression happens at Flush and Compaction only. Both are done by background threads. Not user facing.
- Decompression happens whenever reading compressed data
- Data blocks are cached in RocksDB block cache, with uncompressed format
- Decompression must be fast and efficient, since it happens many more often than compression
- Level 0 and 1 sst files are very frequently written but doesn't occupy space
- It is possible to use stronger compression level than InnoDB
  - InnoDB compression happens on user facing operations
  - zlib compression level1 for InnoDB => Can use zlib compression level 6 for MyRocks

Space and Compression

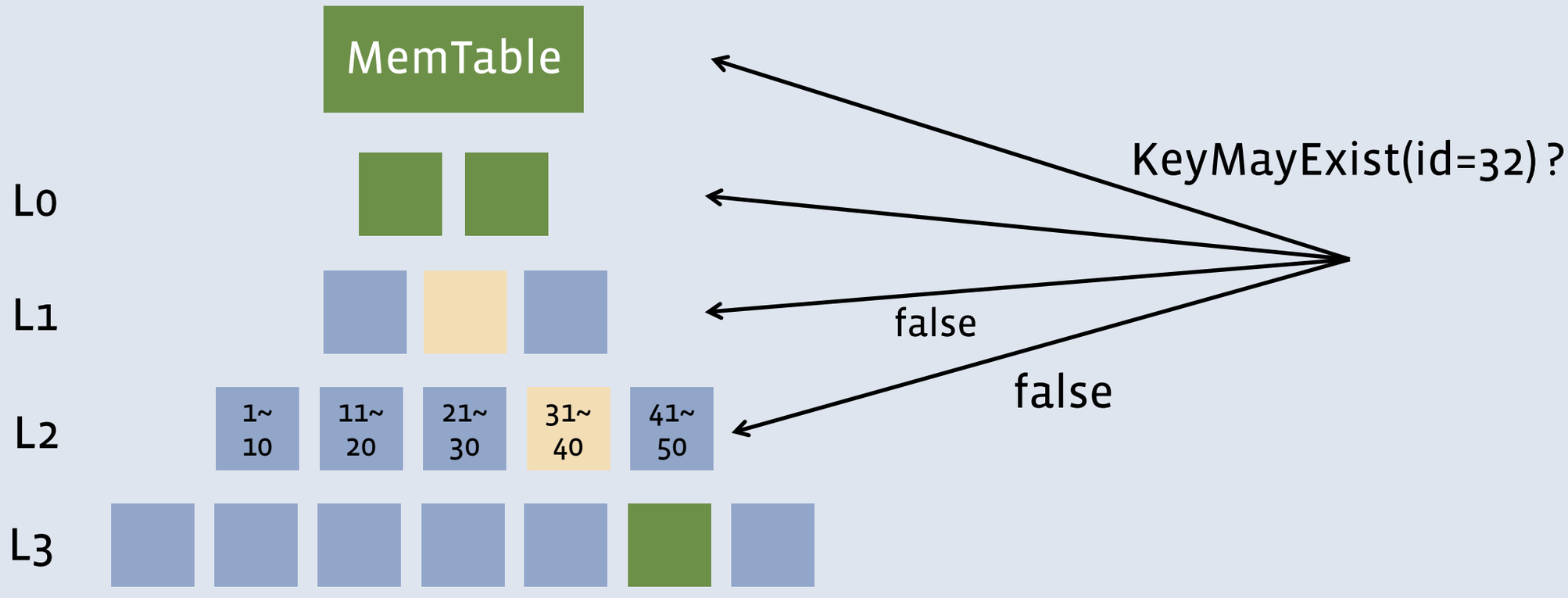
- Use compression. Default is Snappy
- zlib is also recommended since it gives better space savings than Snappy/LZ4, and decompression is fast enough
- zstd gives better compression rate and speed than zlib. It's new and actively developed
- Compressing Level 0 ~ Level 2 sst files don't give much space benefits. Not using compression for these levels makes more sense to reduce CPU usage

Size optimizations for leveled placement

- level_compaction_dynamic_level_bytes=true
  - http://rocksdb.org/blog/2207/dynamic-level/

Bloom Filter

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



- Rows are sorted by key for each level
- SST file stores min key and max key at header so out of range sst files are skipped regardless of bloom filter
- Bloom filter is useful if keys are within min/max range in the sst, and if keys do not exist there

How to configure bloom filter

- By default, bloom filter is disabled
- Can be set in my.cnf
  - rocksdb_default_cf_options=prefix_extractor=capped:12;
    block_based_table_factory={cache_index_and_filter_blocks=1;filter_policy=bloomfilter:10:false;
    whole_key_filtering=0}
  - Define bloom filter length -- Internal 4 byte index id + major condition length
- Condition length must be larger than or equal to bloom filter length
  - Example: “WHERE id_bigint=1” => bloom filter length has to be shorter than or equal to 12
  - MyRocks checks WHERE condition length then skips using bloom filter if can’t be used
- Can be set per column family
- Can’t change bloom filter settings without dump & restore
  - Because bloom filter information is stored in sst files persistently

When bloom filter can be used (1)

- MyRocks automatically decides to use bloom filter if possible, based on WHERE conditions and CF parameters
- Can be used for equal conditions in WHERE clause
  - WHERE id=1 (index id)
- Can be used for AND conditions
  - WHERE id1 = 1 and id2=1 (index id1,id2)
- Can be used for prefix index lookup
  - WHERE id1=1 (index id1, id2)
- Can be used for prefix range scan with equal predicates
  - WHERE id1 = 1 AND time < now()
  - Bloom filter is used for filtering id1=1 only. Can't be used for range conditions

When bloom filter can be used (2)

- WHERE key IN ('01234567890123456789', '1'), and BF length 16
  - BF used for the first condition, not used for the second condition

- MyRocks bloom filter supports both point lookup and prefix lookup
  - Prefix bloom filter is not used on descending range/full scan (ascending scan when using reverse column family)

Bloom filter size overhead

- SST file size increases approximately 2~3%
- If your equal lookup rarely returns empty set, it makes sense to disable bloom filter on the bottommost level
  - All data reside on the bottommost level
  - If your equal lookups always find data (i.e. using negative cache in front), bloom filter on the bottommost level is useless
- CF option “optimize_filters_for_hits=true” can turn off bloom filter on the bottommost level

Data Loading

- There are some session variables to make data loading faster
  - rocksdb_skip_unique_check=1
  - rocksdb_commit_in_the_middle=1

Deleting large number of rows

- SET session sql_log_bin=0; SET session rocksdb_commit_in_the_middle=1;
  DELETE FROM x WHERE id < 10000000;
  - Run these on both master and slaves
  - The DELETE is not written to binlog so taking long time is ok
  - “rocksdb_commit_in_the_middle=1” helps not to maintain lots of uncommitted changes in memory (WriteBatch)
  - Since MyRocks doesn’t hold gap lock, this won’t block inserts

- Repeating deletes with LIMIT is not recommended
  - i.e. (for 1..100,000 DELETE FROM x WHERE id < 10000000 LIMIT 100)
  - It has to gradually scan many tombstones

Other configurations (DB options)

- rocksdb_block_size
  - I/O unit (not fully aligned). Default is 4KB. 16KB gives better space savings but needs extra CPU for decompression. Measure trade-offs between 4K, 8K, 16K and 32K.
- rocksdb_block_cache_size
  - RocksDB's internal cache. Less important than innodb_buffer_pool_size since RocksDB relies on OS cache too
- rocksdb_max_total_wal_size
  - Controls maximum WAL size. Setting as large as total InnoDB log size would be fine
- rocksdb_base_background_compactions
- rocksdb_max_background_compactions
- rocksdb_max_background_flushes
- rocksdb_lock_wait_timeout
- rocksdb_max_open_files=-1
  - Increase file descriptor limit for mysqld process (Increase nofile in /etc/security/limits.conf)
- rocksdb_rpl_lookup_rows=0

Other configurations (CF options) (1)

- write_buffer_size
- max_write_buffer_number
- levelo_file_num_compaction_trigger
- levelo_slowdown_writes_trigger
- levelo_stop_writes_trigger

- compression_per_level
- compression_opts

- level_compaction_dynamic_level_bytes

Other configurations (CF options) (2)

- block_based_table_factory
  - cache_index_and_filter_blocks
  - filter_policy
  - whole_key_filtering
- prefix_extractor
  - optimize_filters_for_hits

Configuration Example

```
[mysqld]
rocksdb
default-storage-engine=rocksdb
skip-innodb
default-tmp-storage-engine=MyISAM
collation-server=latin1_bin

rocksdb_max_open_files=-1
rocksdb_base_background_compactions=1
rocksdb_max_background_compactions=8
rocksdb_max_total_wal_size=4G
rocksdb_max_background_flushes=4
rocksdb_block_size=16384
rocksdb_block_cache_size=12G
rocksdb_lock_wait_timeout=2
rocksdb_rpl_lookup_rows=0

rocksdb_default_cf_options=write_buffer_size=128m;target_file_size_base=32m;max_bytes_for_level_base
=512m;level0_file_num_compaction_trigger=4;level0_slowdown_writes_trigger=10;level0_stop_writes_trigg
er=15;max_write_buffer_number=4;compression_per_level=kNoCompression:kNoCompression:kNoCompres
sion:kZlibCompression:kZlibCompression:kZlibCompression:kZlibCompression;compression_opts=-
14:6:0;block_based_table_factory={cache_index_and_filter_blocks=1;filter_policy=bloomfilter:10:false;whole
_key_filtering=0};prefix_extractor=capped:12;level_compaction_dynamic_level_bytes=true;optimize_filters_
for_hits=true
```

Monitoring

- MyRocks files
- SHOW ENGINE ROCKSDB STATUS
- SHOW GLOBAL STATUS
- information_schema
- sst_dump
- Perf Context
- Checking data consistency
  between InnoDB and MyRocks

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 section. 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
---------------------------------------
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=1;”

- 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

Checking data consistency

- Comparing data between InnoDB and MyRocks instances, without stopping traffics
  - Or for multiple InnoDB instances, or multiple MyRocks instances too
- Take consistent snapshots at the same binlog position, then run any non-locking selects, compare results
- pt-table-checksum does not work for MyRocks, because of lack of Gap Lock support

Writing SELECT correctness tools

- Suppose comparing two slaves
  - slave 2) STOP SLAVE SQL_THREAD; SET GLOBAL slave_parallel_workers=0; sleep 1 second,
  - Slave 1) START TRANSACTION WITH CONSISTENT INNODB|ROCKSDB SNAPSHOT; => getting binlog GTID X
  - Slave2) START SLAVE UNTIL SQL_AFTER_GTIDS = X;
    SELECT @@global.gtid_executed => Y
    Compare X and Y and confirm Y is subset of X (Y is not exceeding X)
    SELECT WAIT_UNTIL_SQL_THREAD_AFTER_GTIDS(X);
  - slave2) START TRANSACTION WITH CONSISTENT INNODB|ROCKSDB SNAPSHOT;
  - slave2) SET GLOBAL slave_parallel_workers=N; START SLAVE SQL_THREAD;
  - Slave1, slave2) Run any SELECTs to compare results
  - slave1, slave2) ROLLBACK; -- release snapshots. They're expensive to hold for a long time
- START SLAVE does implicit commit in 5.7. Make sure to run via different thread
- Make snapshot retention period short. Beware of wait_timeout.

Memory Management

- Jemalloc is recommended memory allocator in MyRocks
  - Memory (RSS) is less fragmented than glibc
- Major memory components
  - Block Cache
    - rocksdb_block_cache_size
  - MemTable
    - write_buffer_size * max_write_buffer_number * number of column families
    - SHOW ENGINE ROCKSDB STAUS prints memory usage from MemTable
  - WriteBatch
- O_DIRECT is not supported (yet)

Memory usage

- All modifications and row lock state by transactions are kept in memory
- They’re released at transaction commit
- Maximum number of row locks or modifications per transaction can be controlled by rocksdb_max_row_locks session variable
- If you are running huge inserts/updates/deletes, consider using rocksdb_commit_in_the_middle session variable
  - Commits with regular intervals

Mixing both InnoDB and MyRocks

- Add allow-multiple-engines in my.cnf
- Not recommended using in production
  - May be useful on some test experiments
  - We haven’t tested enough
  - Transaction can not cross multiple storage engines

Contributing to MyRocks

- Bug Reports
  - https://github.com/facebook/mysql-5.6/issues
- Documentation
  - MyRocks: https://github.com/facebook/mysql-5.6/wiki
  - RocksDB: https://github.com/facebook/rocksdb/wiki
- Development
  - Test Cases
  - New Features

Thanks!

- https://github.com/facebook/mysql-5.6