Practical Tips for Using MySQL as a Scalable Key-Value Store

Sunny Gleason
Distributed Systems Engineer
SunnyCloud
April 23, 2013
Who am I?

- Sunny Gleason
  - Distributed Systems Engineer
  - SunnyCloud, Boston MA

- Prior Web Services Work:
  - Amazon.com
  - Ning

- Focus: Scalable, Reliable Storage Systems for Structured & Unstructured Data
What’s this all about?

- NoSQL is getting a lot of love right now
- NoSQL core ideas: simplification and doing more with less
- Techniques that apply to any system: it is possible to create future-proof APIs while still enjoying the benefits of your favorite data store
Qualities of MySQL

• High-performance
• Transactional durability and recovery
• Replication for High Availability
• Battle-tested
• Well-known operational characteristics
Qualities of NoSQL

- High performance
- More targeted APIs
- Flexible schema design
- Horizontal scalability
- Simpler operational model (someday)
What’s in Store

• Practical Tips for Using MySQL as a Scalable Key–Value Store

• Key Take–Aways
  – Smaller Data is Better Data
  – Play to InnoDB’s Strengths
  – Live with more relaxed guarantees

• These techniques have been proven in production
NoSQL Recap

• What is a Key–Value store?
• What is a Document store?
• Why are these models important?
Key-Value Store

- Hash Table Model
  - Get(K) $\rightarrow$ V
  - MultiGet(K₁, K₂, ...) $\rightarrow$ [V₁, V₂, ...]
  - Put(K, V)
  - Delete(K)

- Examples:
  - In-Memory: Memcached, Redis*
  - Persistent: Dynamo, Voldemort, Riak*
Selected Key-Value Stores

- redis
- riak
Your New Key Value Store

- Percona Server
- MySQL
- MariaDB
Key-Value Architecture

- Every KV pair is **independent**: the storage system has major leeway in how it handles data
- Underlying innovation: DHT (Distributed Hash Table)
- Use **consistent hashing** to route data (value) placement based on key
- Enables construction of symmetric clusters of simple storage nodes
Consistent Hashing

Node 14 is responsible for keys whose hash = 11, 12, 13, 14

Node 1 is responsible for keys whose hash = 15, 0, 1

Node 3 is responsible for keys whose hash = 2, 3

Node 10 is responsible for keys whose hash = 9, 10

Node 8 is responsible for keys whose hash = 4, 5, 6, 7, 8

Key-Value Benefits

• By embracing the Key-Value model:
  – Flexible schema for application developers
  – Simplify the operational model (fewer, simpler tables)
  – Provide a path to specialized K–V stores when necessary
• At the cost of opaque values in the database
Document Store

- Start with Key-Value model
- Overlay structured Values
  - JSON-like object/property model
  - Ability to create secondary indexes (relations, range queries, full-text...)
- Examples
  - CouchDB, ElasticSearch*, Riak*, MongoDB*
Selected Document Stores

- Elasticsearch
- Riak
- CouchDB
- MongoDB
Your New Document Store

PERCONA Server

MySQL

MariaDB
Key Take-Aways

- Smaller Data is better Data
- Play to InnoDB’s Strengths
- Live with more relaxed guarantees
Small Data?!

- Embrace frugality with storage resources: keep rows narrow
- Each InnoDB page is precious: every time you use one it has a transactional, computational, storage, iops, and fragmentation cost
- By keeping rows narrow, we make better use of CPU, RAM, storage and iops since each page holds more rows
Making Data Smaller

- Compression
- Binary Representation
- Schema Extraction
Compression

- Instead of TEXT/LONGTEXT and huge JSON or XML values
- Use BLOB/LONGBLOB, and compress the value prior to storage:
  - GZip
  - Snappy
  - LZF
Compression Algorithms (Java)

Source: https://github.com/ning/jvm-compressor-benchmark/wiki (Thanks, Tatu!)
Compression Notes

• GZip, Bzip2 typically have great compression (70%+)
  – But, very high CPU utilization
• Snappy, LZF have good compression (30–50%) and lower CPU utilization
• However, all of these algorithms typically fall short for small values (< 1024 bytes)
But... What about my Tweets?!
For small documents, consider using a Huffman Coding library for compression*

Analyze a representative set of data beforehand to create a statistical data model

Can yield 30–50% compression or more for small documents (2–1024 bytes)

Tweet on!

*Also consider InnoDB page compression (however, its use of zlib puts CPU burden on the database)
Binary Serialization

• Look familiar?
  <element isAwesome="false"/>
  {“id”:23405, ...}
  JOHN\tSMITH\t17 XYZ LANE\t...

• Actual type sizes: int (4 bytes), boolean (1 byte*), enum (1 byte*)

• Use a binary encoding:
  – General: Msgpack, Avro, Thrift, Protobuf
  – JSON-based: Smile, BSON
  – XML-based: Fast Infoset
Binary Serialization

> require "msgpack"  
> msg = [1,2,3].to_msgpack  
> MessagePack.unpack(msg)  

Sources: http://msgpack.org/  http://martin.kleppmann.com/2012/12/compactprotocol.png
Binary Serialization Speed (Java)

Source: https://github.com/eishay/jvm-serializers/wiki
Binary Serialization Size (Java)

Source: https://github.com/eishay/jvm-serializers/wiki
• **Smile** is my personal favorite
  – JSON-based, fast, low-complexity
• **Msgpack** has advanced a *lot* recently
  – Could be the new winner based on size, performance and language support
• Other formats are still useful based on application-specific needs (already using Thrift, XML model, etc)
Possibly disturbing content follows!
Schema Extraction

• Idea:
  – Extract a subset of key–value pairs into a schema, represent as Array vs. Map
  – Additionally, use more efficient types to represent values (such as boolean -> bit, enum -> int)

• Benefits: explicit schema versioning, decouple schema evolution, more compact representation

• Downside: approaching relational DB complexity without benefits of querying (yet) & table–level consistency
Instead of:
{
"id":10,"first_name":"William",
"last_name":"Gates","yob":1955, ...
}
{
"id":11,"first_name":"Steve",
"last_name":"Jobs","yob":1955, ...
}

Schema:
[{{"id":"int"},{"first_name":"string"},
{{"last_name":"string"},{"yob":"int"}, ...]

Values:
[10,"William","Gates",1955, ...]
What about sparse objects?

- Include a value that encodes presence or absence of attribute values
- Use a bitmap int at beginning of array
- Sparse Values:
  - ["1010,10,"Gates"]
  - ["1101,11,"Steve",1955]
What about schema evolution?

- Include an int schema version in value array
- Remember previous versions of schema to decode values
- `[{"id":"int","v":1},{"first_name":"string","v":1}, {"last_name":"string","v":1},{"yob":"int","v":2}]`
- Versioned Values:
  [1,‘101,10,”Gates”]
  [2,‘1101,11,”Steve”,1955]
What about extension attributes?

- Include a map of undeclared attributes at the end of the value array
- Merge with declared attributes
- Extended Values:
  [1,'101,10,"Gates",{"home_town":"Redmond"}]
  [2,'1101,11,"Steve",{"company":"Apple"}]
Schema Extraction Summary

- Schema extraction can be a powerful technique
- Basis for validation as well as data minimization
- Combined with compression, provides strong data size reduction
- Use cases: log/event data, third-party data
- Cons:
  - Adds significant complexity to storage code
  - Can create friction for schema evolution
App-Defined Schema Philosophy

- Use the database for durable persistence and (possibly) transactions
- Let application-defined storage code manage the structure of values
- The pain of application-managed value formats is worth it compared to the pain of strict migrations (for better or worse)
Key Take-Aways

- Smaller Data is better Data
- Play to InnoDB’s Strengths
- Live with more relaxed guarantees
Playing to InnoDB’s Strengths

• Designing a Key–Value Schema
  – Defining Keys
  – Defining Values

• Useful Techniques
  – Sequence/ID Generation
  – Application–level Update Logs

• Addressing Operational Concerns
  – Fragmentation
  – Sharding and/or Multiple KV tables
InnoDB Structure Recap

B+Tree Structure

Levels are numbered starting from 0 at the leaf pages, incrementing up the tree. Pages on each level are doubly-linked with previous and next pointers in ascending order by key. Records within a page are singly-linked with a next pointer in ascending order by key. Infimum represents a value lower than any key on the page, and is always the first record in the singly-linked list of records. Supremum represents a value higher than any key on the page, and is always the last record in the singly-linked list of records. Non-leaf pages contain the minimum key of the child page and the child page number, called a “node pointer.”

Source: http://jcole.us/blog/files/innodb/20130109/72dpi/B_Tree_Structure.png (Thanks, Jeremy!)
Defining Keys

- Put as much thought into your Key design as possible
- Try to avoid end-user defined keys in favor of datastore-defined keys plus secondary indexes
- Avoid arbitrary-length keys (since secondary index rows will include that value, causing unnecessary duplication)
About UUIDs

• With UUIDs (or any random-looking value) as primary key, insert performance degrades significantly as database size grows beyond buffer pool
• Consider using `BINARY(16)` for UUIDs to conserve space (instead of `CHAR(36)`)
• Make the primary key a mostly-sequential value, and create an InnoDB secondary unique index on the “random” key
A Potentially Useful Trick

- Outside the data store, use an encoded/encrypted version of the integer (for example, 10 -> “ABCD1234”)
- Make sure the encoding function is a perfect mapping function (no collisions)
- Keep the encoding algorithm an internal design detail of the data store
- Within the data store, use the integer values as compact key identifiers
- Prevents outsider sequence guess attacks
Key Definition

• Ideally, something like BIGINT or INT
  – Possibly include INT _tenant_id for multi-tenant situations
  – Possibly include INT _type_id if you want to be able to operate on all instances of a given type (select all user profiles, etc)

• If using a composite primary key, pay careful attention to ordering
  – Left-to-right column ordering defines clustering placement in InnoDB
  – Do you want (_tenant_id, _key_id), or (_key_id, _tenant_id)?
Defining Values

• For future adoption of binary encoding and/or compression:
  – Consider adding CHAR(1) column(s) for “format” and/or “compression”
  – Consider BLOB/LONGBLOB even if storing text-based values like JSON or XML
  – Pay extra close attention to encoding of values when they arrive into the application

• BLOB Storage Requirements:
  – BLOB: 2+N bytes, N < 2^{16}
  – MEDIUMBLOB: 3+N bytes, N < 2^{24}
  – LONGBLOB: 4+N bytes, N < 2^{32}
Sequence Generation

- MySQL auto-increment columns
  - Have non-zero transactional overhead
  - In presence of sharding, schema management becomes more complex to work with

- External Sequence Services
  - Increment an AtomicLong in nanos
  - Decouples identity from data placement
  - Can make sharded operation easier
  - See https://github.com/twitter/snowflake and https://github.com/boundaryflake
Application-Level Update Logs

- How do we determine “recent updates”?
- Introduce columns for `_updated_dt` and `_created_dt` (for example, bigint utc millis)
- Brand new KV rows have `_created_dt = _updated_dt` (or `_version = 1`)
- Add a non-unique index on `_updated_dt` desc
- When used for asynchronous replication, presumes it’s ok to discard intermediate values (can make catch-up more efficient)
Addressing Fragmentation

• As rows are deleted and updated, InnoDB pages will become fragmented

• Consider logical deletion instead:
  ```
  update _key_values
  set _is_deleted = 'Y'
  where _key_id = ?
  ```

• In replicated configurations, batch deletion and defragmentation can be performed during failover/failback
Sharding & Multiple KV Tables

• In the Key–Value model, each entry is completely independent
• This makes the introduction of sharding easier (especially with externalized ID generation)
• Consider having “Active” and “Closed for Writing” KV tables to facilitate maintenance (shard migration, compaction)
Example Key–Value Schema

create table if not exists `_key_values` (  
  `_key_type`  smallint unsigned not null,  
  `_key_id`    bigint unsigned not null,  
  `_created_dt`  int unsigned not null,  
  `_updated_dt`  int unsigned not null,  
  `_version`  bigint unsigned not null,  
  `_is_deleted`  char(1) not null default 'N',  
  `_format`  char(1) not null default 'S',  
  `_compression`  char(1) not null default 'F',  
  `_value`  blob not null,  
PRIMARY KEY(`_key_type`, `_key_id`),  
INDEX(`_updated_dt`)  
) ENGINE=InnoDB ROW_FORMAT=DYNAMIC CHARACTER SET utf8;
Key Take-Aways

- Smaller Data is better Data
- Play to InnoDB’s Strengths
- Live with more relaxed guarantees
Live With More Relaxed Guarantees

- Atomic Transactions
- Versioning
  - Implementing Optimistic Updates
- Asynchronous Secondary Indexes
  - Range-based Indexes
  - Counters/Drill-Downs
  - External full-text indexes
Atomic Transactions: Pros & Cons

- When using MySQL as a Key-Value store, you have the option to expose multi-KV-pair transactions
- Multi-Put, Multi-Delete: can be very useful for debit/credit purchase and/or inventory systems
- Only works as long as KV pairs live in the same database (avoid cross-database transactions)
- If possible, avoid multi-mutators so that there is a clear path for sharding or alternatives
Implementing Optimistic Updates

- Techniques such as SELECT FOR UPDATE pessimistically lock rows which may not be contended
- With optimistic updates, the client provides the version of the object it wants to update
- The SQL looks roughly like:
  ```sql
  update _key_values
  set _value = ?, _version = _version + 1
  where _key_id = ? and _version = ?
  ```
Implementing Optimistic Updates

• After issuing the update statement, the client queries the number of rows updated
• If it’s 1, the update succeeded
• If it’s 0, the update failed
  – Return a CONFLICT error code to the client, signaling it to refresh the object and retry
  – If caches are employed, have the client refresh update the cache as well
Secondary Indexes

• 2-Query model plays to Key-Value store strengths: (1) **query for IDs**, followed by (2) **multi-get**

• Secondary indexes live in separate tables (potentially in different data stores)

• Enables extension of the Key-Value store with many features of Document Stores
  – Range-based queries
  – Materialized Aggregations
  – Full-text Indexes
Range-Based Secondary Indexes

• Logical Index Definition:
  `[{“zip_code”:”asc”},{“car_make”:”asc”},
   {“_key_id”:”asc”}]`

  CREATE TABLE zip_code_car_make (zip_code INT,
car_make VARCHAR(200), _key_id BIGINT UNIQUE);

  CREATE INDEX zip_code_car_make_ix
  ON zip_code_car_make (zip_code ASC, car_make ASC, _key_id ASC);
Materialized Aggregations

- Logical Counter Definition:
  ```json
  ["zip_code":"asc"],{"car_make":"asc"}
  ```

```sql
CREATE TABLE zip_code_car_make_counter (zip_code INT, car_make VARCHAR(200), _count BIGINT);
CREATE UNIQUE INDEX zip_code_car_make_counter_ix ON zip_code_car_make_counter (zip_code ASC, car_make ASC);
```
Full-Text Indexing

- JSON-based structured values lend themselves very well to ElasticSearch indexing
- Option 1 (push): update search index with every value update
- Option 2 (pull): async poll update log
- It is relatively easy to create an ElasticSearch “river” to propagate updates to the full-text index
Key Take-Aways

- Smaller Data is better Data
- Play to InnoDB’s Strengths
- Live with more relaxed guarantees
- Keep your options open
Keep Your Options Open

• **MySQL Extensions**
  – Techniques such as MySQL Memcached API (and previously, HandlerSocket) for fast object access

• **In-memory Key-Value stores:**
  – Redis and Memcached have high performance, high reliability, and easier clustering (compared to sharding)
  – Redis has powerful set and queue operations that can be useful for modeling graphs and permissions

• **Full-text Indexing:**
  – ElasticSearch has built-in clustering with consistent hashing, replication, and map well to JSON-based data
Wrap-Up

• NoSQL provides insight into scalable data store design: Key–Value & Document are powerful models
• With a bit of foresight, it is possible to apply these models to MySQL–based data services
• Obtain the scaling and performance characteristics of NoSQL with the reliability of MySQL
Mission Accomplished?

Thank You!