facebook

DocStore: Document Database for MySQL at Facebook

Peng Tian, Tian Xia

04/14/2015

Agenda

Overview of DocStore

Document: A new column type to store JSON

New Built-in JSON functions

Document Path: A intuitive way to access JSON in SQL

FBSON: A binary JSON format and runtime parser

Indexing on JSON

Overview of DocStore

What is DocStore?

- Stands for “Document Store”
- A JSON document database built on top of MySQL

The goal of DocStore

- To provide a easy-to-use, flexible, and schema-less storage solution

Overview of DocStore

JSON: (JavaScript Object Notation)

```
{"name":"Tom",
 "age":30,
 "married":false,
 "address":{"houseNumber":1001,
            "streetName":"main",
            "zipcode":"98761",
            "state":"CA"
           },
 "cars":["F150",
         "Honda"
        ],
 "memo":null
}
```

Overview of DocStore

MySQL and its weaknesses

- Many good reasons to use MySQL, but…
- It is *not* developer friendly for rapid early development.
- Handles sparse tables inefficiently.

DocStore can resolve these issues with JSON

Overview of DocStore

Table 1: (before Online Schema Change)

Id	Name	Age	StreetName	StreetNumber	ZipCode	State	HomePhone
101	“Alex”	25	“Main”	“12345”	“94080”	“CA”	“6502343432”

Table 2: (after online schema change)

Id	Name	Age	StreetName	StreetNumber	ZipCode	State	HomePhone	WorkPhone	CellPhone
101	“Alex”	25	“Main”	“12345”	“94080”	“CA”	“6502343432”	NULL	NULL
202	“Tom”	35	“10th”	“777”	“94025”	“CA”	“6507734537”	“6508342356”	“6506628711”

Table 3: (if JSON was supported as a column type)

Id	JSON
101	{“name”:”Alex”, ”Age”:25, ”Address”:{”StreetName”:”Main”, ”StreetNumber”:”12345”, ”ZipCode”:”94080”, ”State”:”CA”}, ”HomePhone”:” 6502343432”}
202	{“name”:”Tom”, ”Age”:35, ”Address”:{”StreetName”: ”10th”,”StreetNumber”: ”777”,”ZipCode”: ”94025”,”State”: ”CA”}, ”HomePhone”:” 6502343432”, ”WorkPhone”:” 6508342356”, ”CellPhone”:” 6506628711”}

Overview of DocStore

Table 4: a sparse table

Id	K1	…	K50	…	K100	…	K150	…	K200
101	12345				true				“main”
202			67890				“CA”		

Table 5: If JSON was supported as a column type

Id	JSON
101	{“K1”:12345, ”K100”:true, ”K200”:”main”}
202	{“K50”:67890, ”K150”:”CA”}

So, the first thing we need for DocStore is …

Agenda

Overview of DocStore

Document: A new column type to store JSON

New Built-in JSON functions

Document Path: A intuitive way to access JSON in SQL

FBSON: A binary JSON format and runtime parser

Indexing on JSON

What next

Document: A new column type to store JSON

```
CREATE TABLE t (id int(8), doc document) ENGINE=innodb;

INSERT INTO t VALUES (100,
          ‘{"name":"Tom",
            "age":30,
            "married":false,
            "address":{"houseNumber":1001,"streetName":"main”,
                       "zipcode":"98761”,"state":"CA"},
            "cars":["F150","Honda"],
            "memo":null}’);
```

Document: A new column type to store JSON

What happens when inserting a JSON string into a Document column?

- Converted to FBSON & stored as BLOB in InnoDB
- Validations!
- Maximum size is 16MB – 1.
- All or nothing: Never get truncated!

Now, how to access the keys/values in JSON documents?

Agenda

Overview of DocStore

Document: A new column type to store JSON

New Built-in JSON functions

Document Path: A intuitive way to access JSON in SQL

FBSON: A binary JSON format and runtime parser

Indexing on JSON

What next

New Built-in JSON functions

- Introduced new Built-in JSON functions for DocStore
- For Document, Blob, Text, and Char column types

- `String json_extract(col,k1,k2…)`
- `Bool json_contains_key(col,k1,k2…)`
- `Bool json_valid(col,k1,k2…)`

```
SELECT json_extract('doc’,’name’)
FROM t
WEHRE json_extract('doc’,'address’,'zipcode’) like ‘98761’;

SELECT json_extract('doc’,’name’)
FROM t
WEHRE json_extract('doc’,’car’,’0’) like ‘F150’;

SELECT id
FROM t
WEHRE json_contains_key('doc’,'address’,'zipcode’);
```

Is this good enough?

Agenda

Overview of DocStore

Document: A new column type to store JSON

New built-in JSON functions

Document Path: A intuitive way to access JSON in SQL

FBSON: A binary JSON format and runtime parser

Indexing on JSON

What next

Document Path: A intuitive way to access JSON in SQL

- Starts with a column name whose type is Document, followed by a bunch of JSON keys separated by dot, e.g. `doc`.`address`.`zipcode`

- Also known as Virtual Column/Virtual Field

```
CREATE TABLE t (id int not null, doc document not null, primary key(id)
                unique key id_doc(id, doc.address.zipcode as int))
                engine=innodb;

SELECT id, doc.name
FROM t WHERE doc.address.zipcode like ‘98761’;

SELECT id, doc.name
FROM t WHERE doc.car.0 like ‘F150’;

SELECT id, doc.name
FROM t WHERE doc.age = 30;

SELECT id, doc.name
FROM t GROUP BY doc.address.streetName;

SELECT id, doc.name
FROM t ORDER BY cast(doc.address.houseNumber as unsigned);
```

Document Path: A intuitive way to access JSON in SQL

- Charsets: JSON “keys” vs. MySQL `identifers`

```
CREATE TABLE t (`~!();'"?,./\t` document not null) ENGINE=innodb;

INSERT INTO t VALUES('{"~!@#$%^&*()_":{"+-=:;\'<>?,./":"val"}}');

SELECT `~!();'"?,./\t`
FROM t
WHERE `~!();'"?,./\t`.`~!@#$%^&*()_`.`+-=:;'<>?,./` like "val";
```

- `doc.car.0` Is the number 0 a key name or an array index?

```
Is the value of `car` a JSON or an array?
```

- `foo.bar.baz` Any ambiguities?

```
database.table.column? Table.column.key? Column.key1.key2?
```

Document Path: A intuitive way to access JSON in SQL

- The type system: JSON/FBSON, MySQL, doc-paths with or without CAST, default type, and type conversions

```
CREATE TABLE t (id int not null, doc document not null, primary key(id)
                unique key id_doc(id, doc.address.zipcode as int)) engine=innodb;

SELECT id, doc.name
FROM t WEHRE doc.address.zipcode like ‘98761’;

SELECT id, doc.name
FROM t WEHRE doc.age = 30;

SELECT id, doc.name
FROM t GROUP BY doc.address.streetName;

SELECT id, doc.name
FROM t ORDER BY cast(doc.address.houseNumber as unsigned);
```

- NULL values: JSON/FBSON, MySQL, and nonexistent document paths

Blob column + json_extract() vs. Document column + Document Path

Blob column + json_extract()	Document column + Document Path
Storing JSON as string ☹	Automatically converting & storing as FBSON ☺
No validation so JSON can be invalid ☹	Automatically validating ☺
May be truncated (without strict mode) ☹	Never gets truncated ☺
Return type is string ☹	Return type is based on context, default is string ☺
Behaves as a MySQL built-in functions	Behaves like a table column
Indexes cannot be built on it ☹	Can be secondary key part ☺
Cannot be handled by query optimizer ☹	Can be handled by query optimizer ☺
Not very intuitive ☹	Very intuitive! ☺
All using low-level FBSON APIs	

Agenda

Overview of DocStore

Document: A new column type to store JSON

New Built-in JSON functions

Document Path: A intuitive way to access JSON in SQL

FBSON: A binary JSON format and runtime parser

Indexing on JSON

Why binary?

- Binary takes less space.
- String requires runtime parsing and conversion
  - “True” / “False” => 1/0
  - “12345” => 12345
  - “123.45” => 123.45
  - “Null”

Existing Binary Formats and Libraries

- BSON (used by MongoDB):
  - A lot of non-standard and MongoDB specific grammar
- Universal Binary JSON (inspired by CouchDB):
  - Less binary: array, object are enclosed by “[”, “]”, “{”, “}”.
- Both need to read objects in full to traverse next

FBSON format is simple and efficient for iterating and searching.

FBSON: A binary storage format for JSON strings

- Support all JSON types
- Richer and fine-grained types
- Size info is stored with all values
- Optionally, keys can be saved as IDs instead of strings

FBSON Library

FBSON library is a standalone, header-only, C++ library.

An incremental parser

- Parses JSON stream without loading full document.
- Reads JSON and writes FBSON binary at the same time

FBSON Library

Reading FBSON binary is very efficient:

- No memory allocation when de-serializing the binary bytes
- Search doesn’t need to read whole objects.
- A forward iterator to walk through FBSON objects.

FBSON Grammar

```
key  ::= 0x00 int8    //1-byte id in key dictionary
       | int8 (byte*) //int8 (non-zero) is the size of the key string

primitive_value ::= 0x00        //null value (0 byte)
                  | 0x01        //boolean true (0 byte)
                  | 0x02        //boolean false (0 byte)
                  | 0x03 int8   //char/int8 (1 byte)
                  | 0x04 int16  //int16 (2 bytes)
                  | 0x05 int32  //int32 (4 bytes)
                  | 0x06 int64  //int64 (8 bytes)
                  | 0x07 double //floating point (8 bytes)
                  | 0x08 string //variable length string
                  | 0x09 binary //variable length binary

string  ::= int32 (byte*) //int32 is the size of the string
binary  ::= int32 (byte*) //int32 is the size of the binary blob
```

FBSON Grammar

```
container       ::= 0x0A int32 key_value_list //object type
                  | 0x0B int32 value_list     //array type

key_value_list ::= key value key_value_list
value_list      ::= value value_list
value           ::= primitive_value
                  | container

document        ::= int8 container
```

Notes:

- The first byte stores version information.
- Empty container is encoded to a type byte and a size integer (0).

Agenda

Overview of DocStore

Document: A new column type to store JSON

New Built-in JSON functions

Document Path: A intuitive way to access JSON in SQL

FBSON: A binary JSON format and runtime parser

Indexing on JSON

Secondary Indexes on Documents

JSON doesn’t enforce type consistency.

```
{“zipcode”: 94025}
{“zipcode”: “94025”}
```

MySQL infers type at parsing time

- Select list
- Where constraints.
- Types of InnoDB secondary indexes

What’s the type of a document path if we want to build a secondary index on it?

Secondary Indexes on Documents

Document path type is explicitly defined in secondary indexes.

- int: 8-byte integers
- double: 8-byte double
- bool: 1-byte integers (0/1)
- string: prefix indexes, default size is 64 characters

```
CREATE TABLE t1 ( id int not null,
                  doc document not null,
                  b char(10),
                  c int not null,
                  primary key(id),
                  unique key doc_c(doc.address.zipcode as int, c)
                ) engine=innodb;
```

Secondary Indexes on Documents

Extracted values of document paths are stored in B-trees.

- Type conversion without precision loss.
  - Integers → Double
  - Integers/Double → String
- NULL will be stored in indexes if
  - a value doesn't match index type, and
  - type conversion is not possible.

Query Optimization

Basic optimizer support for document path secondary indexes.

- Single table retrieval
- **Covering index**: index-only scan

```
mysql> explain select c from t1 where doc.address.zipcode = 98761;

+----+-------------+-------+------+---------------+-------+---------+-------+------+--------------------------+
| id | select_type | table | type | possible_keys | key   | key_len | ref   | rows | Extra                    |
+----+-------------+-------+------+---------------+-------+---------+-------+------+--------------------------+
|  1 | SIMPLE      | t1    | ref  | doc_c         | doc_c | 9       | const |    1 | Using where; Using index |
+----+-------------+-------+------+---------------+-------+---------+-------+------+--------------------------+
```

Query Optimization

Basic optimizer support for document path secondary indexes.

- Single table retrieval
- **Non-covering index**: index is used to retrieve the row data

```
mysql> explain select b from t1 where doc.address.zipcode = 98761;
```

```
+----+-------------+-------+------+---------------+-------+---------+-------+------+-------------+
| id | select_type | table | type | possible_keys | key   | key_len | ref   | rows | Extra       |
+----+-------------+-------+------+---------------+-------+---------+-------+------+-------------+
|  1 | SIMPLE      | t1    | ref  | doc_c         | doc_c | 9       | const |    1 | Using where |
+----+-------------+-------+------+---------------+-------+---------+-------+------+-------------+
```

More covering index examples

- Implicit type conversion when covering index is found

```
mysql> explain select c from t1 where doc.address.zipcode = '98761';

+----+-------------+-------+------+---------------+-------+---------+-------+------+--------------------------+
| id | select_type | table | type | possible_keys | key   | key_len | ref   | rows | Extra                    |
+----+-------------+-------+------+---------------+-------+---------+-------+------+--------------------------+
|  1 | SIMPLE      | t1    | ref  | doc_c         | doc_c | 9       | const |    1 | Using where; Using index |
+----+-------------+-------+------+---------------+-------+---------+-------+------+--------------------------+
```

```
mysql> explain select c from t1 where doc.address.zipcode = trim(‘ 98761 ’);

+----+-------------+-------+------+---------------+-------+---------+-------+------+--------------------------+
| id | select_type | table | type | possible_keys | key   | key_len | ref   | rows | Extra                    |
+----+-------------+-------+------+---------------+-------+---------+-------+------+--------------------------+
|  1 | SIMPLE      | t1    | ref  | doc_c         | doc_c | 9       | const |    1 | Using where; Using index |
+----+-------------+-------+------+---------------+-------+---------+-------+------+--------------------------+
```

More covering index examples

- Index-only range scan

```
mysql> explain select id from t1 where doc.address.zipcode > 94025 and
doc.address.zipcode < 98761;

+----+-------------+-------+-------+---------------+-------+---------+------+------+--------------------------+
| id | select_type | table | type  | possible_keys | key   | key_len | ref  | rows | Extra                    |
+----+-------------+-------+-------+---------------+-------+---------+------+------+--------------------------+
|  1 | SIMPLE      | t1    | index | doc_c         | doc_c | 13      | NULL |    1 | Using where; Using index |
+----+-------------+-------+-------+---------------+-------+---------+------+------+--------------------------+
```

Hints for Document Path Secondary Indexes

Why hints?

- The variety of queries going through query optimization is large
- Adding comprehensive optimizer support takes time

New hints for document path secondary indexes

```
USE    DOCUMENT KEYS|INDEXES
IGNORE DOCUMENT KEYS|INDEXES

USE DOCUMENT KEYS USE INDEX (doc_path_key_1)
USE DOCUMENT KEYS IGNORE INDEX (doc_path_key_2)
```

Internals of Secondary Index

MySQL stores column information in the metadata file “.frm”.

For document path indexes, more information needs to be saved.

- Document path names (e.g. `doc.address.zipcode`)
- Document path type and value length (if prefix)

Similar information is also persisted in Innodb’s metadata table to extract values and save them into B-trees

Internals of Secondary Index

Document field object is inherited from blob.

To support secondary indexes, document field object will need to:

- Store key values to do index scan
- Output the extracted values directly from indexes (covering index)
- Get FBSON binary and extract the value (non-covering index)

Document field is a hybrid object!

Internals of Secondary Index

Previously, only columns can be key parts

With document paths, indexes could point to different document paths on the same column.

Columns are no-longer unique in document path secondary indexes

Facebook DocStore vs. MySQL 5.7.7 JSON Labs Release

Facebook DocStore	MySQL 5.7.7 JSON Labs Release
New column type "Document"	New column type "JSON"
Validated, converted, and stored in FBSON	Validated, converted, and stored in binary JSON format
Built-in JSON functions focusing on query	Built-in JSON function supporting query and manipulations.
Arbitrary document path in query, more ad-hoc	Virtual column tied with DDL
Secondary keys can include both regular column and document path	Secondary keys cannot include both regular column and document path

Thank you

facebook