Introducing the MySQL Document Store

Mike Frank / マイク フランク
MySQL Global Business Unit
Product Management Director
Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle’s products remains at the sole discretion of Oracle.
Today’s Agenda

1. Relational Databases, Document Databases and MySQL
2. MySQL JSON Support
3. Document Use Cases
4. The X DevAPI
5. Getting it all working together
Relational Databases

• Data Integrity
  – Normalization
  – Constraints (foreign keys etc)

• Atomicity, Consistency, Isolation, Durability - ACID
  – Transactions

• SQL
  – Powerful, Optimizable Query Language
  – Declare what you want and the DB will find out the most efficient way to get it to you
Plus...

- MySQL has been around since 1995
- Ubiquitous
- Pretty much a standard
- Scalable
- When there are issues, they are known and understood
- Large body of knowledge, from small to BIG deployments
Document Databases

• Schemaless
  – no schema design, normalization, foreign keys, constraints, data types etc
  – faster initial development

• Flexible data structures
  – nested arrays and objects
  – some data is simply naturally unstructured or cannot be modeled efficiently in the relational model (hierarchies, product DB etc)
  – persist objects without ORMs
Document Databases (Cont.)

• JSON
  – Closer to the frontend
  – "native" in JavaScript
  – Node.js and full stack JavaScript

• Easy to learn, easy to use
Relational vs Document Databases

Why not both?
1. Relational Databases, Document Databases and MySQL
2. MySQL JSON Support
3. Document Use Cases
4. The X DevAPI
5. Getting it all working together
## The New JSON Datatype

```sql
mysql> CREATE TABLE employees (data JSON);
mysql> INSERT INTO employees VALUES
    ('{"id": 1, "name": "Jane"}'),
    ('{"id": 2, "name": "Joe"}');
mysql> SELECT * FROM employees;
<table>
<thead>
<tr>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>{&quot;id&quot;: 1, &quot;name&quot;: &quot;Jane&quot;}</td>
</tr>
<tr>
<td>{&quot;id&quot;: 2, &quot;name&quot;: &quot;Joe&quot;}</td>
</tr>
</tbody>
</table>
```

- Validation on INSERT
- No reparsing on SELECT
- Optimized for read
- Dictionary of sorted keys
- Can compare JSON/SQL
- Can convert JSON/SQL
- Supports all native JSON datatypes
- Also supports date, time, timestamp etc.
MySQL 5.7: JSON Support

• Native JSON datatype
• Store JSON values (objects, arrays and simple values) in MySQL tables
• Binary JSON storage format
• Conversion from "native" SQL types to and from JSON values
• JSON Manipulation functions
  – Extract contents (JSON_EXTRACT, JSON_KEYS etc)
  – Inspect contents (JSON_CONTAINS etc)
  – Modify contents (JSON_SET, JSON_INSERT, JSON_REMOVE etc)
  – Create arrays and objects (JSON_ARRAY, JSON_OBJECT)
  – Search objects (JSON_SEARCH)
MySQL 5.7: JSON Support (cont.)

• Inline SQL JSON path expressions
  SELECT doc->{'$.object.array[0].item'} FROM some_table

• Boolean operators (compare JSON values etc)
  – foo = doc->'$.field'
Generated Columns

CREATE TABLE order_lines
(orderno integer,
  lineno integer,
  price decimal(10,2),
  qty integer,
  sum_price decimal(10,2) GENERATED ALWAYS AS (qty * price) STORED );

• Column generated from the expression
• VIRTUAL: computed when read, not stored, indexable
• STORED: computed when inserted/updated, stored in SE, indexable
• Useful for:
  – Functional index
  – Materialized cache for complex conditions
  – Simplify query expression
Functional Index

```
CREATE TABLE order_lines
(orderno integer,
_lineno integer,
 price decimal(10,2),
 qty  integer,
 sum_price decimal(10,2) GENERATED ALWAYS AS (qty * price) VIRTUAL);

ALTER TABLE order_lines ADD INDEX idx (sum_price);
```

- **Online** index creation
- Composite index on a mix of ordinary, virtual and stored columns
Indexing JSON data

CREATE TABLE employees (data JSON);

ALTER TABLE employees
ADD COLUMN name VARCHAR(30) AS (JSON_UNQUOTE(data->"$.name"))
VIRTUAL,
ADD INDEX name_idx (name);

• Functional index approach
• Use inlined JSON path or JSON_EXTRACT to specify field to be indexed
• Support both VIRTUAL and STORED generated columns
### Generated column: STORED vs VIRTUAL

<table>
<thead>
<tr>
<th></th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORED</td>
<td>• Fast retrieval</td>
<td>• Require table rebuild at creation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Update table data at INSERT/UPDATE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Require more storage space</td>
</tr>
<tr>
<td>VIRTUAL</td>
<td>• Metadata change only, instant</td>
<td>• Compute when read, slower retrieval</td>
</tr>
<tr>
<td></td>
<td>• Faster INSERT/UPDATE, no change to table</td>
<td></td>
</tr>
</tbody>
</table>

*Oracle*
## Indexing Generated Column: STORED vs VIRTUAL

<table>
<thead>
<tr>
<th></th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORED</td>
<td>• Primary &amp; secondary index</td>
<td>• Duplication of data in base table and index</td>
</tr>
<tr>
<td></td>
<td>• B-TREE, Full text, R-TREE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Independent of SE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Online operation</td>
<td></td>
</tr>
<tr>
<td>VIRTUAL</td>
<td>• Less storage</td>
<td>• Secondary index only</td>
</tr>
<tr>
<td></td>
<td>• Online operation</td>
<td>• B-TREE only</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Require SE support</td>
</tr>
</tbody>
</table>
1. Relational Databases, Document Databases and MySQL
2. MySQL JSON Support
3. Document Use Cases
4. The X DevAPI
5. Getting it all working together
Extracting JSON from a Relational DB

Relational In, Relational + Document Out

• Data stored in relational tables, but frontend uses JSON
• JSON directly maps to native data structures in many languages
  – Often easier for application code to use
  – JavaScript, Python, Ruby etc
  – In browser JavaScript
Extracting JSON from a Relational DB

Relational In, Relational + Document Out

• SQL Functions to construct JSON
  – JSON_OBJECT(), JSON_ARRAY()

• Ex.:

  SELECT JSON_OBJECT('cust_id', id, 'name', name, 'email', email) FROM customer;

  CREATE VIEW customer_json AS
    SELECT JSON_OBJECT('cust_id', id, 'name', name, 'email', email) as doc
    FROM customer;
  SELECT * FROM customer_json;

• Updates and inserts still happen through the table columns
Using MySQL as a JSON Document Container

Document In, Relational + Document Out

• Virtually Schemaless
  – Unstructured data
  – No clear, fixed structure for the data... records can have different fields
  – Often data that is not involved in business rules
  – Examples: "product_info", "properties", "options" etc

• Data does not map cleanly into a relational model (arrays, hierarchical data etc)

• Prototyping
MySQL as a JSON Document Container

Example: "properties" table

WIKIPEDIA

https://www.mediawiki.org/wiki/Manual:Database_layout
### MySQL as a JSON Document Container

**Example: "product_info" table**

<table>
<thead>
<tr>
<th>product_id</th>
<th>attribute</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>size</td>
<td>M</td>
</tr>
<tr>
<td>9</td>
<td>color</td>
<td>red</td>
</tr>
<tr>
<td>9</td>
<td>fabric</td>
<td>cotton</td>
</tr>
<tr>
<td>11</td>
<td>flavour</td>
<td>strawberry</td>
</tr>
<tr>
<td>12</td>
<td>capacity</td>
<td>128GB</td>
</tr>
<tr>
<td>12</td>
<td>speed class</td>
<td>class 10</td>
</tr>
<tr>
<td>13</td>
<td>connectivity</td>
<td>Wi-Fi</td>
</tr>
<tr>
<td>13</td>
<td>storage</td>
<td>64GB</td>
</tr>
<tr>
<td>13</td>
<td>screen size</td>
<td>8.9&quot;</td>
</tr>
<tr>
<td>13</td>
<td>resolution</td>
<td>2560 x 1600 (339 ppi)</td>
</tr>
<tr>
<td>13</td>
<td>battery life</td>
<td>12 hours</td>
</tr>
</tbody>
</table>

```json
{
  "product_id": 9,
  "size": "M",
  "color": "red",
  "fabric": "cotton"
},
{
  "product_id": 11,
  "flavour": "strawberry"
},
{
  "product_id": 12,
  "capacity": "128GB",
  "speed class": "class 10"
},
```
MySQL as a JSON Document Container

Document In, Relational + Document Out

• An ordinary MySQL table with a single JSON data column
• Generated columns allow SQL engine to look inside the JSON data
  – Virtual columns
  – Primary Keys
  – Indexes
  – Foreign Keys
• Writes on the JSON column
• Reads primarily from the JSON columns
Hybrid Relational and JSON

Relational + Document In, Relational + Document Out

- Database is mostly relational
- Some parts of the database are unstructured or does not model cleanly as relational
- JSON columns in relational tables
- Queries can mix and match JSON and column data
- Evolution path for Document based applications
Hybrid Relational and JSON

Relational + Document In, Relational + Document Out

```
product (product_id, name, category_id, ...);
product_info (product_id, attribute, value);
```
1. Relational Databases, Document Databases and MySQL
2. MySQL JSON Support
3. Document Use Cases
4. The X DevAPI
5. Getting it all working together
Document Operations via SQL

• Powerful
• Allows complex queries
• But... still difficult to use
Document Operations via SQL

```
CREATE TABLE product (  
id VARCHAR(32) GENERATED ALWAYS AS (JSON_EXTRACT(doc, '$.id')) STORED,
doc JSON  
);  

INSERT INTO product VALUES (1, '{...}');  
SELECT * FROM product WHERE JSON_EXTRACT(doc, '$.field') = value;  
```
The X DevAPI

- Abstraction over SQL
- Focused on 4 basic CRUD operations (Create, Read, Update, Delete)
- Fluent, Native Language API
- No knowledge of SQL needed
- X Protocol
  - CRUD requests encoded at protocol level
  - Request details "visible" (vs "opaque" SQL strings)
Collection and Schema Operations

• Get a handle to a Schema
  
  ```
  mydb = session.getSchema("mydb");
  ```

• Create a Collection
  
  ```
  mydb.createCollection("products");
  ```

• Get a (local) reference to a Collection
  
  ```
  products = mydb.getCollection("products");
  ```
products.add({"name":"bananas", "color":"yellow"}).execute();
products.find("color = 'yellow'").sort(["name"]).execute();
Modify Documents

```java
products.modify("product_id = 123").set("color", "red").execute();
```
products.remove("product_id = 123").execute();
X DevAPI Sessions

• X Session
  – Stateless
  – CRUD only, no SQL
  – Abstracts the connection

• Node Session
  – Direct connection to a database node
  – Allows CRUD and SQL
Other Operations on Collections

• Create an Index

```javascript
db.post.createIndex("email").field("author.email", "text(30)", false)
```
CRUD Operations – NoSQL/Document and SQL/Relational

<table>
<thead>
<tr>
<th>Operation</th>
<th>Document</th>
<th>Relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>Collection.add()</td>
<td>Table.insert()</td>
</tr>
<tr>
<td>Read</td>
<td>Collection.find()</td>
<td>Table.select()</td>
</tr>
<tr>
<td>Update</td>
<td>Collection.modify()</td>
<td>Table.update()</td>
</tr>
<tr>
<td>Delete</td>
<td>Collection.remove()</td>
<td>Table.delete()</td>
</tr>
</tbody>
</table>
1. Relational Databases, Document Databases and MySQL
2. MySQL JSON Support
3. Document Use Cases
4. The X DevAPI
5. Getting it all working together
5.7.12 Development Preview Release

• MySQL 5.7.12 with Document Store plugin
• MySQL Shell 1.0.3
• Connector/J 7.0
• Connector/Net 7.0
• Connector/Node.js 1.0
MySQL 5.7, Connectors, Drivers, and Protocols

MySQL Connectors and Drivers
- SQL API
- Std Protocol
- CRUD and SQL APIs
- X Protocol

MySQL Shell

Memcached driver

Core

Plugins
- X Protocol Plugin
- Memcached Plugin

MySQL

X and Std Protocols

Memcache Protocol

Std Protocol 3306

X Protocol 33060
uri = "mysql:x://localhost:33060/test?user=user&password=mypwd";
XSession session = new MysqlxSessionFactory().getSession(uri);
Schema schema = session.getDefaultSchema();
// document walkthrough
Collection coll = schema.createCollection("myBooks", true);
DbDoc newDoc = new DbDoc().add("isbn",
   new JsonString().setValue("12345"));
newDoc.add("title",
   new JsonString().setValue("Effi Briest"));
newDoc.add("author",
   new JsonString().setValue("Theodor Fontane"));
newDoc.add("currentlyReadingPage",
   new JsonNumber().setValue(String.valueOf(42)));
coll.add(newDoc).execute();

DocResult docs = coll.find("title = 'Effi Briest' and currentlyReadingPage > 10").execute();

DbDoc book = docs.fetchOne();
System.err.println("Currently reading "
   + ((JsonString)book.get("title")).getString()
   + " on page "
   + ((JsonNumber)book.get("currentlyReadingPage")).getInteger());

// increment the page number and fetch it again

coll.modify("isbn = 12345")
   .set("currentlyReadingPage",
      ((JsonNumber)book.get("currentlyReadingPage")).getInteger() + 1).execute();

docs = coll.find("title = 'Effi Briest' and currentlyReadingPage > 10").execute();

book = docs.fetchOne();
System.err.println("Currently reading "
   + ((JsonString)book.get("title")).getString()
   + " on page "
   + ((JsonNumber)book.get("currentlyReadingPage")).getInteger());
X DevAPI Connectors – MySQL Connector/Net 7.0

```csharp
using (XSession session = MySQLX.GetSession("mysqlx://test:test@localhost:33060"))
{
    string schemaName = "test";
    Schema testSchema = session.GetSchema(schemaName);
    if (testSchema.ExistsInDatabase())
        session.DropSchema(schemaName);
    session.CreateSchema(schemaName);

    // insert some docs
    Collection coll = testSchema.CreateCollection("myDocs");
    var docs = new[]
    {
        new { _id = 1, title = "Book 1", pages = 20 },
        new { _id = 2, title = "Book 2", pages = 30 },
        new { _id = 3, title = "Book 3", pages = 40 },
        new { _id = 4, title = "Book 4", pages = 50 },
    };
    Result r = coll.Add(docs).Execute();
    Console.WriteLine("Docs added: " + r.RecordsAffected);

    // modify some values
    r = coll.Modify("_id = :ID").
        Bind("Id", 2).Set("pages", "25").Execute();
    Console.WriteLine("Docs modified: " + r.RecordsAffected);

    // remove a book
    r = coll.Remove("_id = :ID").Bind("Id", 4).Execute();
    Console.WriteLine("Docs removed: " + r.RecordsAffected);

    // list the results
    var result30orMore = coll.Find("pages > 20").
        OrderBy("pages DESC").Execute().FetchAll();
    foreach (var doc in result30orMore)
    {
        Console.WriteLine(doc.ToString());
    }
}
```
X DevAPI Connectors – MySQL Connector/Node.js 1.0 NEW!

```javascript
const mysqlx = require('mysqlx');

mysqlx.getSession({
  host: 'localhost',
  dbUser: 'myuser',
  dbPassword: 'secret'
}).then(session => {
  const collection = session.getSchema('myschema').getCollection('mycollection');
  return Promise.all([
    collection.add({ foo: "bar", something: { nested: [1,2,3,4] } }).execute();
    session.close();
  ]).catch(err => {
    console.log(err);
  });
});

const collection = mysqlx.getSchema('myschema').getCollection('questions');

const collection = collection.find("answer == 42")
  .orderBy("foo DESC")
  .limit(10)
  .execute(doc => console.log(doc))
  .then(() => console.log("All done") // Promise resolves when all data is there
    .catch((err) => console.log("Oups, an error", err);
```

Copyright © 2016 Oracle and/or its affiliates. All rights reserved.
MySQL Shell

```
Copyright (c) 2016, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type '\help', '\h' or '\?' for help.

Currently in JavaScript mode. Use \sql to switch to SQL mode and execute queries.

mysql-js> db.getCollections()
{
   "CountryInfo": <Collection:CountryInfo>
}
mysql-js> db.CountryInfo.find().limit(1)
[
{
   "GNP": 828,
   "IndepYear": null,
   "Name": "Aruba",
   "_id": "ABW",
   "demographics": {
      "LifeExpectancy": 78.4000015258789,
      "Population": 103000
   },
   "geography": {
      "Continent": "North America",
      "Region": "Caribbean",
      "SurfaceArea": 193
   },
   "government": {
      "GovernmentForm": "Nonmetropolitan Territory of The Netherlands",
      "HeadOfState": "Brentijn"
   }
}
```
MySQL Plugin for VisualStudio
# Resources

<table>
<thead>
<tr>
<th>Topic</th>
<th>Link(s)</th>
</tr>
</thead>
</table>
Thank You!