MySQL 8.0

a Document Store with all the benefits of a transactional RDBMS

Dave Stokes - MySQL Community Manager - Oracle

Frédéric Descamps - MySQL Community Manager - Oracle
Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purpose 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 product remains at the sole discretion of Oracle.
about us

Who are we?
Disclaimer

We are not Mike Zinner and Alfredo Kojima

Mike and Alfredo, team leaders of the MySQL Document Store were not able to come to the conference and they are sorry about that.
Dave Stokes

- @stoker
- MySQL Evangelist
- New Book: MySQL & JSON -> Pre Order Now on Amazon
- http://elephantdolphin.blogspot.com
Frédéric Descamps

- @lefred
- MySQL Evangelist
- Hacking MySQL since 3.23
- devops believer
- living in Belgium 🇧🇪
- http://lefred.be
Relational Databases

- Data Integrity
Relational Databases

- Data Integrity
  - normalization
Relational Databases

- Data Integrity
  - normalization
  - constraints (foreign keys, ...)
  
  
  

Relational Databases

- Data Integrity
  - normalization
  - constraints (foreign keys, ...)
- Atomicity, Consistency, Isolation, Durability
Relational Databases

- Data Integrity
  - normalization
  - constraints (foreign keys, ...)
- Atomicity, Consistency, Isolation, Durability
  - ACID compliant
Relational Databases

- Data Integrity
  - normalization
  - constraints (foreign keys, ...)
- Atomicity, Consistency, Isolation, Durability
  - ACID compliant
  - transactions
Relational Databases

- Data Integrity
  - normalization
  - constraints (foreign keys, ...)
- Atomicity, Consistency, Isolation, Durability
  - ACID compliant
  - transactions
- SQL
Relational Databases

- Data Integrity
  - normalization
  - constraints (foreign keys, ...)
- Atomicity, Consistency, Isolation, Durability
  - ACID compliant
  - transactions
- SQL
  - powerful query language
NoSQL & MySQL

NoSQL Databases
NoSQL or Document Store

- Schemaless
NoSQL or Document Store

- Schemaless
  - no schema design, no normalization, no foreign keys, no data types, ...
NoSQL or Document Store

- Schemaless
  - no schema design, no normalization, no foreign keys, no data types, ...
  - very quick initial development
NoSQL or Document Store

- Schemaless
  - no schema design, no normalization, no foreign keys, no data types, ...
  - very quick initial development
- Flexible data structure
NoSQL or Document Store

- Schemaless
  - no schema design, no normalization, no foreign keys, no data types, ...
  - very quick initial development
- Flexible data structure
  - embedded arrays or objects
NoSQL or Document Store

- Schemaless
  - no schema design, no normalization, no foreign keys, no data types, ...
  - very quick initial development
- Flexible data structure
  - embedded arrays or objects
  - valid solution when natural data can’t be modelized optimally into a relational model
NoSQL or Document Store

• Schemaless
  ○ no schema design, no normalization, no foreign keys, no data types, ...
  ○ very quick initial development

• Flexible data structure
  ○ embedded arrays or objects
  ○ valid solution when natural data can’t be modelized optimally into a relational model
  ○ objects persistence without the use of any ORM - *mapping object-oriented*
NoSQL or Document Store

- **Schemaless**
  - no schema design, no normalization, no foreign keys, no data types, ...
  - very quick initial development
- **Flexible data structure**
  - embedded arrays or objects
  - valid solution when natural data can’t be modelized optimally into a relational model
  - objects persistence without the use of any ORM - *mapping object-oriented*
- **JSON**
NoSQL or Document Store

- Schemaless
  - no schema design, no normalization, no foreign keys, no data types, ...
  - very quick initial development
- Flexible data structure
  - embedded arrays or objects
  - valid solution when natural data can’t be modeled optimally into a relational model
  - objects persistence without the use of any ORM - mapping object-oriented
- JSON
  - close to frontend
NoSQL or Document Store

- Schemaless
  - no schema design, no normalization, no foreign keys, no data types, ...
  - very quick initial development

- Flexible data structure
  - embedded arrays or objects
  - valid solution when natural data can’t be modelized optimally into a relational model
  - objects persistence without the use of any ORM - mapping object-oriented

- JSON
  - close to frontend
  - native in JS
NoSQL or Document Store

- Schemaless
  - no schema design, no normalization, no foreign keys, no data types, ...
  - very quick initial development
- Flexible data structure
  - embedded arrays or objects
  - valid solution when natural data can’t be modelized optimally into a relational model
  - objects persistence without the use of any ORM - *mapping object-oriented*
- JSON
  - close to frontend
  - native in JS
  - easy to learn
How DBAs see data
How DBAs see data

How Developers see data

```json
{
  "GNP" : 249704,
  "Name" : "Belgium",
  "government" : {
    "GovernmentForm" : "Constitutional Monarchy, Federation",
    "HeadOfState" : "Philippe I"
  },
  "_id" : "BEL",
  "IndepYear" : 1830,
  "demographics" : {
    "Population" : 10239000,
    "LifeExpectancy" : 77.8000030517578
  },
  "geography" : {
    "Region" : "Western Europe",
    "SurfaceArea" : 30518,
    "Continent" : "Europe"
  }
}
```
How DBAs see data

How Developers see data

{  
  "GNP" : 249704,  
  "Name" : "Belgium",  
  "government" : {  
    "GovernmentForm" : "Constitutional Monarchy, Federation",  
    "HeadOfState" : "Philippe I"  
  },  
  "id" : "BEL",  
  "IndepYear" : 1830,  
  "demographics" : {  
    "Population" : 10239000,  
    "LifeExpectancy" : 77.8000030517578  
  },  
  "geography" : {  
    "Region" : "Western Europe",  
    "SurfaceArea" : 30518,  
    "Continent" : "Europe"  
  }  
}
How DBAs see data

How Developers see data

```json
{
  "GNP" : 249704,
  "Name" : "Belgium",
  "government" : {
    "GovernmentForm" : "Constitutional Monarchy, Federation",
    "HeadOfState" : "Philippe I"
  },
  "id" : "BEL",
  "IndepYear" : 1830,
  "demographics" : {
    "Population" : 10239000,
    "LifeExpectancy" : 77.8000030517578
  },
  "geography" : {
    "Region" : "Western Europe",
    "SurfaceArea" : 30518,
    "Continent" : "Europe"
  }
}
```
How DBAs see data

How Developers see data

```json
{
  "GNP" : 249704,
  "Name" : "Belgium",
  "government" : {
    "GovernmentForm" : "Constitutional Monarchy, Federation",
    "HeadOfState" : "Philippe I"
  },
  "_id" : "BEL",
  "IndepYear" : 1830,
  "demographics" : {
    "Population" : 10239000,
    "LifeExpectancy" : 77.8000030517578
  },
  "geography" : {
    "Region" : "Western Europe",
    "SurfaceArea" : 30518,
    "Continent" : "Europe"
  }
}
```
What if there was a way to provide both SQL and NoSQL on one stable platform that has proven stability on well know technology with a large Community and a diverse ecosystem?
DMBS or NoSQL?
DMBS or NoSQL?

Why not both?
The MySQL Document Store!

SQL is now optional!
A solution for all

Developers:
- [x] schemaless
- [x] rapid prototyping/simpler APIs
- [x] document model
- [x] transactions

Operations:
- [x] performance management/visibility
- [x] robust replication, backup, restore
- [x] comprehensive tooling ecosystem
- [x] simpler application schema upgrades

Business Owner:
- [x] don't lose my data == ACID trx
- [x] capture all my data = extensible/schemaless
- [x] product on schedule/time to market = rapid development
MySQL Document Store

the Solution
Built on the MySQL JSON Data type and Proven MySQL Server Technology

- Provides a schema flexible JSON Document Store
- **No SQL** required
- No need to define all possible attributes, tables, etc.
- Uses new X DevAPI
- Can leverage generated column to extract JSON values into materialized columns that can be indexed for fast SQL searches.
- Document can be ~1GB
  - It's a column in a row of a table
- Allows use of modern programming styles
  - No more embedded strings of SQL in your code
  - Easy to read
- Also works with relational Tables
- Proven MySQL Technology

Copyright @ 2018 Oracle and/or its affiliates. All rights reserved.
X DevAPI

- Connectors for
  - C++, Java, .Net, Node.js, Python, PHP
  - working with Communities to help them supporting it too
- New MySQL Shell
  - Command Completion
  - Python, JavaScrips & SQL modes
  - Admin functions
  - New Util object
    - A new high-level session concept that can scale from single MySQL Server to a multiple server environment
- Non-blocking, asynchronous calls follow common language patterns
- Supports CRUD operations
Starting using **MySQL** DS in few minutes

```sql
MySQL => JS  \c root@localhost
Creating a session to 'root@localhost'
Enter password:
Fetching schema names for autocompletion... Press ^C to stop.
Your MySQL connection id is 13 (X protocol)
Server version: 8.0.11 MySQL Community Server - GPL
No default schema selected; type \use <schema> to set one.

MySQL => localhost:33060+ JS  session.createSchema('docstore')
<Schema:docstore>

MySQL => localhost:33060+ JS  \use docstore
Default schema `docstore` accessible through db.
```
Migration from MongoDB to MySQL DS

For this example, I will use the well known restaurants collection:
Migration from MongoDB to MySQL DS

For this example, I will use the well known restaurants collection:

We need to dump the data to a file and we will use MySQL Shell with the Python interpreter to load the data.
Migration from MongoDB to **MySQL DS**

For this example, I will use the well known *restaurants* collection:

We need to dump the data to a file and we will use **MySQL Shell** with the Python interpreter to load the data.
Dump and load using **MySQL** Shell & Python

this example is inspired by @datacharmer's work: [https://www.slideshare.net/datacharmer/mysql-documentstore](https://www.slideshare.net/datacharmer/mysql-documentstore)

```
$ mongo --quiet --eval 'DBQuery.shellBatchSize=30000;
   db.restaurants.find().shellPrint()'
| perl -pe 's/(?:ObjectId|ISODate)\("[^"]*"\)/ $1/g' > all_recs.json
```
Dump and load using **MySQL Shell & Python**

this example is inspired by @datacharmer's work: [https://www.slideshare.net/datacharmer/mysql-documentstore](https://www.slideshare.net/datacharmer/mysql-documentstore)

```bash
$ mongo --quiet --eval 'DBQuery.shellBatchSize=30000;
    db.restaurants.find().shellPrint()' \
| perl -pe 's/(?:ObjectId|ISODate)\(("[^"]+")\)/ $1/g' > all_recs.json
```

```python
import json
import re

with open('all_recs.json', 'r') as json_data:
    for line in json_data:
        skip = re.match('Type', line)
        if not skip:
            rec = json.loads(line)
            db.restaurants.add(rec).execute()
```
DON'T ACT LIKE YOU'RE NOT IMPRESSED
Let's query

MySQL

127.0.0.1:33060

JS

restaurants.find()
Let’s query

That’s too much records to show it here... let’s limit it
MySQL [127.0.0.1:33060] JS restaurants.find().limit(1)

[
  {
    "_id": "5943c83d1adc26055941640c",
    "address": {
      "building": "351",
      "coord": [
        -73.9851,
        40.7677
      ],
      "street": "West 57 Street",
      "zipcode": "10019"
    },
    "borough": "Manhattan",
    "cuisine": "Irish",
    "grades": [
      {
        "date": "2014-09-06T00:00:00Z",
        "grade": "A",
        "score": 2
      },
      {
        "date": "2013-07-22T00:00:00Z",
        "grade": "A",
        "score": 11
      },
      {
        "date": "2012-07-31T00:00:00Z",
        "grade": "A",
        "score": 12
      },
      {
        "date": "2011-12-29T00:00:00Z",
        "grade": "A",
        "score": 10
      }
    ]
  }
]
Some more examples

```sql
MySQL> 127.0.0.1:33060 > JS
restaurants.find().fields(["name","cuisine"]).limit(2)
```

```json
[
  {
    "cuisine": "Irish",
    "name": "Dj Reynolds Pub And Restaurant"
  },
  {
    "cuisine": "American",
    "name": "Riviera Caterer"
  }
]
```

2 documents in set (0.00 sec)
Some more examples

```
MySQL > 127.0.0.1:33060 > JS restaurants.find().fields(["name","cuisine"]).limit(2)
[
  {
    "cuisine": "Irish",
    "name": "Dj Reynolds Pub And Restaurant"
  },
  {
    "cuisine": "American",
    "name": "Riviera Caterer"
  }
]
2 documents in set (0.00 sec)
```

Let's add a selection criteria:
Syntax slightly different than MongoDB

```sql
MySQL
127.0.0.1:33060
JS

restaurants.find("cuisine='French' AND borough!='Manhattan'").
fields(['name', 'cuisine', 'borough']).limit(2)
[
  {
    "borough": "Queens",
    "cuisine": "French",
    "name": "La Baraka Restaurant"
  },
  {
    "borough": "Queens",
    "cuisine": "French",
    "name": "Air France Lounge"
  }
]
2 documents in set (0.00 sec)
```
Syntax slightly different than MongoDB

```
MySQL> 127.0.0.1:33060 JS restaurants.find("cuisine='French' AND borough!='Manhattan'").fields(['name','cuisine','borough']).limit(2)
[
    {
        "cuisine": "French",
        "name": "La Baraka Restaurant"
    },
    {
        "cuisine": "French",
        "name": "Air France Lounge"
    }
]
2 documents in set (0.00 sec)
```
CRUD operations

```
MySQL $127.0.0.1:33060 $ JS restaurants.remove("cuisine='French' AND borough!='Manhattan'").limit(2)
Query OK, 2 items affected (0.16 sec)

MySQL $127.0.0.1:33060 $ JS restaurants.find("cuisine='French' AND borough!='Manhattan'").fields(["name","cuisine","_id"]).limit(2)
[
  {
    "_id": "5943c83e1adc2605594170aa",
    "cuisine": "French",
    "name": "Bar Tabac"
  },
  {
    "_id": "5943c83e1adc260559417255",
    "cuisine": "French",
    "name": "Tournesol"
  }
]
2 documents in set (0.01 sec)
```
CRUD operations for collections

Add a document
CRUD operations for collections

Modify a document
CRUD operations for collections

Remove a document
MySQL Document Store Objects Summary
Document Store Full ACID!

MySQL Document Store supports transactions
Document Store Full ACID!

MySQL Document Store supports transactions

```
MySQL localhost:33060+  f fred JS session.rollback()
Query OK, 0 rows affected (0.0992 sec)
MySQL localhost:33060+  f fred JS test.find()
[
    {
        "id": "00005ade55110000000000000001",
        "name": "fred"
    }
]
1 document in set (0.0130 sec)
```
what about my old SQL?

The hidden part of the iceberg
JSON datatype is behind the scene

- native datatype (since 5.7.8)
JSON datatype is behind the scene

- native datatype (since 5.7.8)
- JSON values are stored in MySQL tables using UTF8MB4
JSON datatype is behind the scene

- native datatype (since 5.7.8)
- JSON values are stored in MySQL tables using UTF8MB4
- Conversion from "native" SQL types to JSON values
JSON datatype is behind the scene

- native datatype (since 5.7.8)
- JSON values are stored in MySQL tables using UTF8MB4
- Conversion from "native" SQL types to JSON values
- JSON manipulation functions (JSON_EXTRACT, JSON_KEYS, JSON_SEARCH, JSON_TABLES, ...)

Copyright @ 2018 Oracle and/or its affiliates. All rights reserved.
JSON datatype is behind the scene

- native datatype (since 5.7.8)
- JSON values are stored in MySQL tables using UTF8MB4
- Conversion from "native" SQL types to JSON values
- JSON manipulation functions (JSON_EXTRACT, JSON_KEYS, JSON_SEARCH, JSON_TABLES, ...)
- Generated/virtual columns
JSON datatype is behind the scene

- native datatype (since 5.7.8)
- JSON values are stored in MySQL tables using UTF8MB4
- Conversion from "native" SQL types to JSON values
- JSON manipulation functions (JSON_EXTRACT, JSON_KEYS, JSON_SEARCH, JSON_TABLES, ...)
- Generated/virtual columns
  - Indexing JSON data
JSON datatype is behind the scene

- native datatype (since 5.7.8)
- JSON values are stored in MySQL tables using UTF8MB4
- Conversion from "native" SQL types to JSON values
- JSON manipulation functions (JSON_EXTRACT, JSON_KEYS, JSON_SEARCH, JSON_TABLES, ...)
- Generated/virtual columns
  - Indexing JSON data
  - Foreign Keys to JSON data
JSON datatype is behind the scene

- native datatype (since 5.7.8)
- JSON values are stored in MySQL tables using UTF8MB4
- Conversion from "native" SQL types to JSON values
- JSON manipulation functions (JSON_EXTRACT, JSON_KEYS, JSON_SEARCH, JSON_TABLES, ...)
- Generated/virtual columns
  - Indexing JSON data
  - Foreign Keys to JSON data
  - SQL Views to JSON data
How does it work?

- MySQL Shell
- Node.js Application
- Windows Application
- MySQL Database
- MySQL Shell

MySQL Server 5.7 & 8.0

* Existing applications are not touched
  - Developers adopt Docker only if they want to
* X Protocol allows SQL and CRUD
  - SQL: SELECT * FROM myTable WHERE id = 12;
  - XDevAPI: db.myTable.find({id: 12})
* Directly hooked into the optimizer
  - Bypass MySQL connection handling
What does a collection look like on the server?

```
<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>doc</td>
<td>json</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>_id</td>
<td>varbinary(32)</td>
<td>NO</td>
<td>PRI</td>
<td>NULL</td>
<td>STORED GENERATED</td>
</tr>
</tbody>
</table>
```

2 rows in set (0.0668 sec)

```
show create table restaurants

Table: restaurants
Create Table: CREATE TABLE `restaurants` (   `doc` json DEFAULT NULL,   `_id` varbinary(32) GENERATED ALWAYS AS (json_unquote(json_extract(`doc`, `_utf8mb4$_id`))) STORED NOT NULL,   PRIMARY KEY (`_id`) ) ENGINE=INNODB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```
Every document has a unique identifier called the *document ID*, which can be thought of as the equivalent of a table’s primary key. The document ID value can be manually assigned when adding a document. If no value is assigned, a document ID is generated and assigned to the document automatically!

Use `getDocumentId()` or `getDocumentIds()` to get `_ids(s)`
Mapping to SQL examples

- `createCollection('mycollection')`
Mapping to SQL examples

- `createCollection('mycollection')`

```sql
CREATE TABLE `test`.`mycoll` (  
doc JSON,  
_id VARCHAR(32)  
    GENERATED ALWAYS AS (doc->>'$.id') STORED  
    PRIMARY KEY  
) CHARSET utf8mb4;
```
Mapping to SQL examples

- `createCollection('mycollection')`

```sql
CREATE TABLE `test`.`mycoll` (  
doc JSON,  
_id VARCHAR(32)  
  GENERATED ALWAYS AS (doc->>'$.id') STORED  
  PRIMARY KEY  
) CHARSET utf8mb4;
```

- `mycollection.add({'test': 1234})`
Mapping to SQL examples

- `createCollection('mycollection')`

```sql
CREATE TABLE `test`.`mycoll` (
  doc JSON,
  _id VARCHAR(32)
    GENERATED ALWAYS AS (doc->>'$.id') STORED
  PRIMARY KEY
) CHARSET utf8mb4;
```

- `mycollection.add({'test': 1234})`

```sql
INSERT INTO `test`.`mycoll` (doc) VALUES (JSON_OBJECT('_id','66307fe367ee6114e0e548bdac28bf', 'test',1234) );
```
Mapping to SQL examples (2)

- `mycollection.find("test > 100")`
Mapping to SQL examples (2)

- `mycollection.find("test > 100")`

```sql
SELECT doc
FROM `test`.`mycoll`
WHERE (JSON_EXTRACT(doc,'$.test') > 100);
```
SQL and JSON Example

```sql
MySQL> localhost:33060+|docstore|SQL
ALTER TABLE restaurants ADD COLUMN borough VARCHAR(20) GENERATED ALWAYS AS (json_unquote(json_extract('doc', '$.borough'))) VIRTUAL;
```
**SQL and JSON Example**

```sql
MySQL localhost:33060+ docstore SQL ALTER TABLE restaurants ADD COLUMN borough VARCHAR(20) GENERATED ALWAYS AS (json_unquote(json_extract('doc', '$.borough'))) VIRTUAL;
```

same as:

```sql
MySQL localhost:33060+ docstore SQL ALTER TABLE restaurants ADD COLUMN borough VARCHAR(20) GENERATED ALWAYS AS (doc->>'$.borough') VIRTUAL;
```
SQL and JSON Example

we can use it in SQL:

```
SELECT _id, borough FROM restaurants LIMIT 5;
```
Indexes in Document Store

It's also possible to create indexes without using SQL syntax:

```javascript
db.restaurants.createIndex('cuisine_idx',
  {fields:{field: "$_.cuisine", required: false, type: "text(20)"}})
```
Indexes in Document Store

It’s also possible to create indexes without using SQL syntax:

```sql
db.restaurants.CreateIndex('cuisine_idx',
{fields:{field: "$_.cuisine", required: false, type: "text(20)"}})
```

```sql
show create table restaurants
```

```
Table: restaurants
Create Table: CREATE TABLE `restaurants` (  
`doc` json DEFAULT NULL,
`_id` varbinary(32) GENERATED ALWAYS AS (json_unquote(json_extract(`doc`, `_utf8mb4\'.$_id' ))) STORED NOT NULL,
`$ix_t20_BC26D4DF1273E3F7412529AEE9E95A0CC8475CEB` text GENERATED ALWAYS AS (json_unquote(json_extract(`doc`, `_utf8mb4\'.$.cuisine' ))) VIRTUAL,
PRIMARY KEY (`_id`),
KEY `cuisine_idx` (`$ix_t20_BC26D4DF1273E3F7412529AEE9E95A0CC8475CEB'(20))
) ENGINE=TRAM DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```
SQL and JSON Example (2): validation

```
MySQL > localhost:33060+ docstore SQL
ALTER TABLE restaurants ADD COLUMN cuisine VARCHAR(20) GENERATED ALWAYS AS (doc->>'$.cuisine') VIRTUAL, WITH VALIDATION;
ERROR: 1406: Data too long for column 'cuisine' at row 10
```
SQL and JSON Example (2): validation

```
MySQL> localhost:33060+ docstore SQL ALTER TABLE restaurants ADD COLUMN cuisine VARCHAR(20) GENERATED ALWAYS AS (doc->>"$.cuisine") VIRTUAL, WITH VALIDATION;
ERROR: 1406: Data too long for column 'cuisine' at row 10

MySQL> localhost:33060+ docstore SQL ALTER TABLE restaurants ADD COLUMN cuisine VARCHAR(20) GENERATED ALWAYS AS (LEFT(doc->>"$.cuisine",20)) VIRTUAL, WITH VALIDATION;
Query OK, 25359 rows affected (0.00 sec)
```
SQL and JSON Example (2): validation

```
MySQL > localhost:33060+ docstore SQL ALTER TABLE restaurants ADD COLUMN cuisine VARCHAR(20) GENERATED ALWAYS AS (doc->"$.cuisine") VIRTUAL, WITH VALIDATION;
ERROR: 1406: Data too long for column 'cuisine' at row 10
```

```
MySQL > localhost:33060+ docstore SQL ALTER TABLE restaurants ADD COLUMN cuisine VARCHAR(20) GENERATED ALWAYS AS (LEFT(doc->"$.cuisine",20)) VIRTUAL, WITH VALIDATION;
Query OK, 25359 rows affected (0.00 sec)
```
SQL and JSON Example (3): explain

```sql
EXPLAIN SELECT doc->>'$.name' AS name, cuisine, borough FROM restaurants WHERE cuisine='Italian' AND borough='Brooklyn' LIMIT 2;
```

```
id: 1
select_type: SIMPLE
table: restaurants
partitions: NULL
type: ALL
possible_keys: NULL
key: NULL
key_len: NULL
ref: NULL
rows: 24890
filtered: 1.0000000000000000
Extra: Using where
1 row in set, 1 warning (0.00 sec)
```
SQL and JSON Example (3): explain

```sql
EXPLAIN SELECT doc->>'$.name' AS name, cuisine, borough FROM restaurants WHERE cuisine='Italian' AND borough='Brooklyn' LIMIT 2
```

**1. row**

- **id**: 1
- **select_type**: SIMPLE
- **table**: restaurants
- **partitions**: NULL
- **type**: ALL
- **possible_keys**: NULL
- **key**: NULL
- **key_len**: NULL
- **ref**: NULL
- **rows**: 24890
- **filtered**: 1.00000001192092896

Extra: Using where

1 row in set, 1 warning (0.00 sec)

Note (code 1003): /* select#1 */ select json_unquote(json_extract('docstore'.'restaurants'.'doc','$.name')) AS 'name', 'docstore'.'restaurants'.'cuisine' AS 'cuisine', 'docstore'.'restaurants'.'borough' AS 'borough' from 'docstore'.'restaurants' where (('docstore'.'restaurants'.'borough' = 'Brooklyn') and ('docstore'.'restaurants'.'cuisine' = 'Italian')) limit 2
SQL and JSON Example (4): add index
SQL and JSON Example (4): add index
SQL and JSON Example (5): arrays

```
MySQL  localhost:33060+  docstore  SQL  select doc->>'$.grades' from restaurants limit 1

doc->>'$.grades': [{"date": "2014-03-03T00:00:00Z", "grade": "A", "score": 2}, {"date": "2013-09-11T00:00:00Z", "grade": "A", "score": 6}, {"date": "2013-01-24T00:00:00Z", "grade": "A", "score": 10}, {"date": "2011-11-23T00:00:00Z", "grade": "A", "score": 9}, {"date": "2011-03-10T00:00:00Z", "grade": "B", "score": 14}]
```
SQL and JSON Example (5): arrays

```
MySQL> localhost:33060+ docstore SQL> select doc->>'$.grades' from restaurants limit 1

1. row

doc->>'$.grades': [{"date": "2014-03-03T00:00:00Z", "grade": "A", "score": 2}, {"date": "2013-09-11T00:00:00Z", "grade": "A", "score": 6}, {"date": "2013-01-24T00:00:00Z", "grade": "A", "score": 10}, {"date": "2011-11-23T00:00:00Z", "grade": "A", "score": 9}, {"date": "2011-03-10T00:00:00Z", "grade": "B", "score": 14}]

MySQL> localhost:33060+ docstore SQL> select doc->>'$.grades[0]' from restaurants limit 1

1. row

doc->>'$.grades[0]': {"date": "2014-03-03T00:00:00Z", "grade": "A", "score": 2}
```
SQL and JSON Example (5): arrays

```sql
MySQL> select doc->>'$.grades' from restaurants limit 1

1. row

[{
    "date": "2014-03-03T00:00:00Z",
    "grade": "A",
    "score": 2
},
{
    "date": "2013-09-11T00:00:00Z",
    "grade": "A",
    "score": 6
},
{
    "date": "2013-01-24T00:00:00Z",
    "grade": "A",
    "score": 10
},
{
    "date": "2011-11-23T00:00:00Z",
    "grade": "A",
    "score": 9
},
{
    "date": "2011-03-10T00:00:00Z",
    "grade": "B",
    "score": 14
}]
```

```sql
MySQL> select doc->>'$.grades[0]' from restaurants limit 1

1. row

[{
    "date": "2014-03-03T00:00:00Z",
    "grade": "A",
    "score": 2
}]
```

```sql
MySQL> select doc->>'$.grades[last]' from restaurants limit 1

1. row

[{
    "date": "2011-03-10T00:00:00Z",
    "grade": "B",
    "score": 14
}]
```
SQL and JSON Example (5): arrays

MySQL: localhost:33060+ docstore SQL: select doc->>'$.grades' from restaurants limit 1

```
1. row

{["date": "2014-03-03T00:00:00Z", "grade": "A", "score": 2}, {["date": "2013-09-11T0
0:00:00Z", "grade": "A", "score": 6}, {["date": "2013-01-24T00:00:00Z", "grade": "A", "score": 10}, {["date": "2011-11-23T00:00:00Z", "grade": "A", "score": 9}, {["date": "2011-03-10T00:00:00Z", "grade": "B", "score": 14]}
```

MySQL: localhost:33060+ docstore SQL: select doc->>'$.grades[0]' from restaurants limit 1

```
1. row

{"date": "2014-03-03T00:00:00Z", "grade": "A", "score": 2}]
```

MySQL: localhost:33060+ docstore SQL: select doc->>'$.grades[last]' from restaurants limit 1

```
1. row

{"date": "2011-03-10T00:00:00Z", "grade": "B", "score": 14}]
```
NoSQL as SQL

### JSON_TABLE

```sql
SELECT _id, borough, cuisine, name
FROM restaurants, JSON_TABLE(doc, "$"
COLUMNS(name CHAR(40) PATH ".name"))
t2 LIMIT 2;
```

<table>
<thead>
<tr>
<th>_id</th>
<th>borough</th>
<th>cuisine</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>59c3a6946273b7d975cf3b39</td>
<td>Bronx</td>
<td>Bakery</td>
<td>Morris Park Bake Shop</td>
</tr>
<tr>
<td>59c3a6946273b7d975cf3b3a</td>
<td>Brooklyn</td>
<td>Hamburgers</td>
<td>Wendy'S</td>
</tr>
</tbody>
</table>
NoSQL as SQL (2)

JSON_TABLE

```
SELECT name, borough, cuisine, street, zipcode
FROM restaurants, JSON_TABLE(doc, "$" COLUMNS(
    name CHAR(40) PATH ".name",
    street CHAR(20) PATH ".address.street",
    zipcode CHAR(10) PATH ".address.zipcode") AS t2)
LIMIT 2;
```

<table>
<thead>
<tr>
<th>name</th>
<th>borough</th>
<th>cuisine</th>
<th>street</th>
<th>zipcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Morris Park Bake Shop</td>
<td>Bronx</td>
<td>Bakery</td>
<td>Morris Park Ave</td>
<td>10462</td>
</tr>
<tr>
<td>Wendy's</td>
<td>Brooklyn</td>
<td>Hamburgers</td>
<td>Flatbush Avenue</td>
<td>11225</td>
</tr>
</tbody>
</table>
SQL and NoSQL

```sql
MySQL localhost:33060+ docstore SQL
create table cuisine (id int auto_increment primary key,
cuisine varchar(20), idolike boolean) COLLATE=utf8mb4_0900_ai_ci;
Query OK, 0 rows affected (0.00 sec)
MySQL localhost:33060+ docstore SQL
insert into cuisine (select distinct 0, cuisine, 0 FROM restaurants);
Query OK, 85 rows affected (0.00 sec)
```
SQL and NoSQL

```
MySQL  localhost:33060+  docstore  SQL  create table cuisine (id int auto_increment primary key, cuisine varchar(20), idolike boolean) COLLATE=utf8mb4_0900_ai_ci;
Query OK, 0 rows affected (0.00 sec)
MySQL  localhost:33060+  docstore  SQL  insert into cuisine (select distinct 0, cuisine, 0 FROM restaurants ) ;
Query OK, 85 rows affected (0.00 sec)
```

```
MySQL  localhost:33060+  docstore  SQL  update cuisine set idolike=1 where id=47;
Query OK, 1 row affected (0.00 sec)
MySQL  localhost:33060+  docstore  SQL  update cuisine set idolike=1 where id=48;
Query OK, 1 row affected (0.00 sec)
MySQL  localhost:33060+  docstore  SQL  update cuisine set idolike=1 where id=82;
Query OK, 1 row affected (0.00 sec)
```
SQL and NoSQL

MySQL
localhost:33060+ docstore SQL
create table cuisine (id int auto_increment primary key, cuisine varchar(20), idolike boolean) COLLATE=utf8mb4_0900_ai_ci;
Query OK, 0 rows affected (0.00 sec)

MySQL
docstore SQL
insert into cuisine (select distinct 0, cuisine, 0 FROM restaurants ) ;
Query OK, 85 rows affected (0.00 sec)

MySQL
docstore SQL
update cuisine set idolike=1 where id=47;
Query OK, 1 row affected (0.00 sec)

MySQL
docstore SQL
update cuisine set idolike=1 where id=48;
Query OK, 1 row affected (0.00 sec)

MySQL
docstore SQL
update cuisine set idolike=1 where id=82;
Query OK, 1 row affected (0.00 sec)
### SQL and NoSQL (2)

```sql
SELECT doc->>'$.name' AS name, t1.cuisine, borough FROM restaurants t1 join cuisine t2 on t2.cuisine = t1.cuisine WHERE idolike=1 limit 5;
```

<table>
<thead>
<tr>
<th>name</th>
<th>cuisine</th>
<th>borough</th>
</tr>
</thead>
<tbody>
<tr>
<td>Venice Restaurant</td>
<td>Italian</td>
<td>Bronx</td>
</tr>
<tr>
<td>Spoto'S Restaurant</td>
<td>Italian</td>
<td>Bronx</td>
</tr>
<tr>
<td>Rino'S</td>
<td>Italian</td>
<td>Bronx</td>
</tr>
<tr>
<td>Dominick'S Bar &amp; Restaurant</td>
<td>Italian</td>
<td>Bronx</td>
</tr>
<tr>
<td>Enzo Cafe</td>
<td>Italian</td>
<td>Bronx</td>
</tr>
</tbody>
</table>
SQL and NoSQL (2)

```sql
SELECT doc->>'$.name' AS name, t1.cuisine, borough FROM restaurants t1 JOIN cuisine t2 ON t2.cuisine = t1.cuisine WHERE idolike=1 limit 5;
```

<table>
<thead>
<tr>
<th>name</th>
<th>cuisine</th>
<th>borough</th>
</tr>
</thead>
<tbody>
<tr>
<td>Venice Restaurant</td>
<td>Italian</td>
<td>Bronx</td>
</tr>
<tr>
<td>Spoto'S Restaurant</td>
<td>Italian</td>
<td>Bronx</td>
</tr>
<tr>
<td>Rino'S</td>
<td>Italian</td>
<td>Bronx</td>
</tr>
<tr>
<td>Dominick'S Bar &amp; Restaurant</td>
<td>Italian</td>
<td>Bronx</td>
</tr>
<tr>
<td>Enzo Cafe</td>
<td>Italian</td>
<td>Bronx</td>
</tr>
</tbody>
</table>
### SQL and NoSQL (2)

```sql
SELECT doc->>'$.name' AS name, t1.cuisine, borough FROM restaurants t1 JOIN cuisine t2 on t2.cuisine = t1.cuisine WHERE idolike=1 limit 5;
```

<table>
<thead>
<tr>
<th>name</th>
<th>cuisine</th>
<th>borough</th>
</tr>
</thead>
<tbody>
<tr>
<td>Venice Restaurant</td>
<td>Italian</td>
<td>Bronx</td>
</tr>
<tr>
<td>Spoto'S Restaurant</td>
<td>Italian</td>
<td>Bronx</td>
</tr>
<tr>
<td>Rino'S</td>
<td>Italian</td>
<td>Bronx</td>
</tr>
<tr>
<td>Dominick'S Bar &amp; Restaurant</td>
<td>Italian</td>
<td>Bronx</td>
</tr>
<tr>
<td>Enzo Cafe</td>
<td>Italian</td>
<td>Bronx</td>
</tr>
</tbody>
</table>
what do I gain?

Conclusion
Conclusion

This is the best of the two worlds in one product!

- Data integrity
- ACID Compliant
- Transactions
- SQL

- schemaless
- flexible data structure
- easy to start (CRUD)
Thank you!

Any Questions?

don’t forget to rate the session and share you 🤍 for @MySQL on social media