
ORACLE®

MySQL 5.7 + JSON

Morgan Tocker
MySQL Product Manager

ORACLE

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. Introduction
2. Core New JSON Features
3. To JSON or !JSON?

MySQL 5.7 - The Team

- **2x** Engineering Staff
- **3x** QA Staff
- **2x** Support Staff

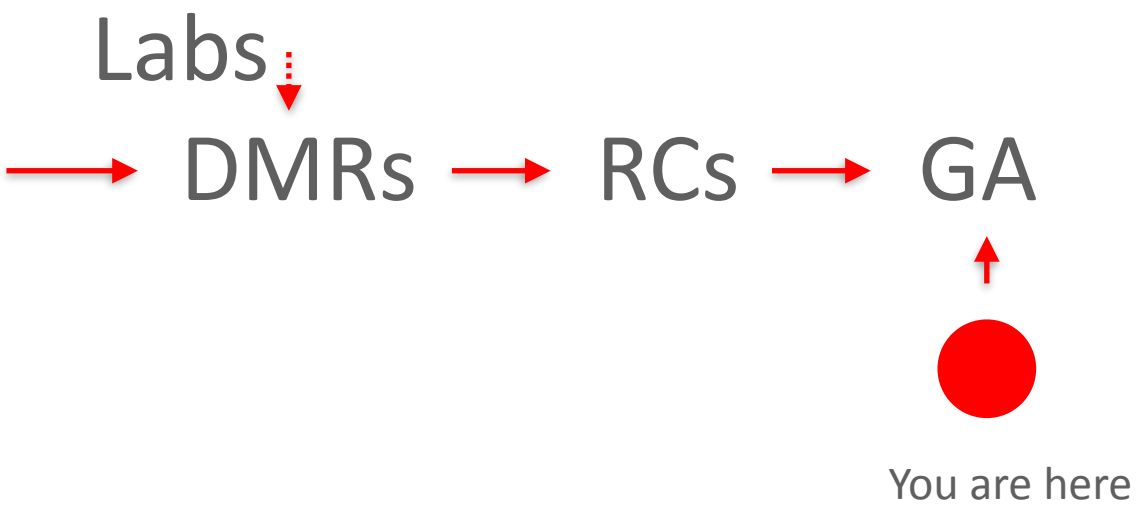
MySQL

Jan 2010

2015

ORACLE

Release Process



MySQL 5.7 is GA!

Performance & Scalability	Manageability
3 X Faster than MySQL 5.6	Native JSON Support
Enhanced InnoDB: faster online & bulk load operations	Improved Security: safer initialization, setup & management
Replication Improvements (incl. multi-source, multi-threaded slaves...)	Performance Schema Improvements
New Optimizer Cost Model: greater user control & better query performance	MySQL SYS Schema

THECOMPLETELISTOFFEATURES.COM

THECOMPLETELISTOFFEATURES.COM

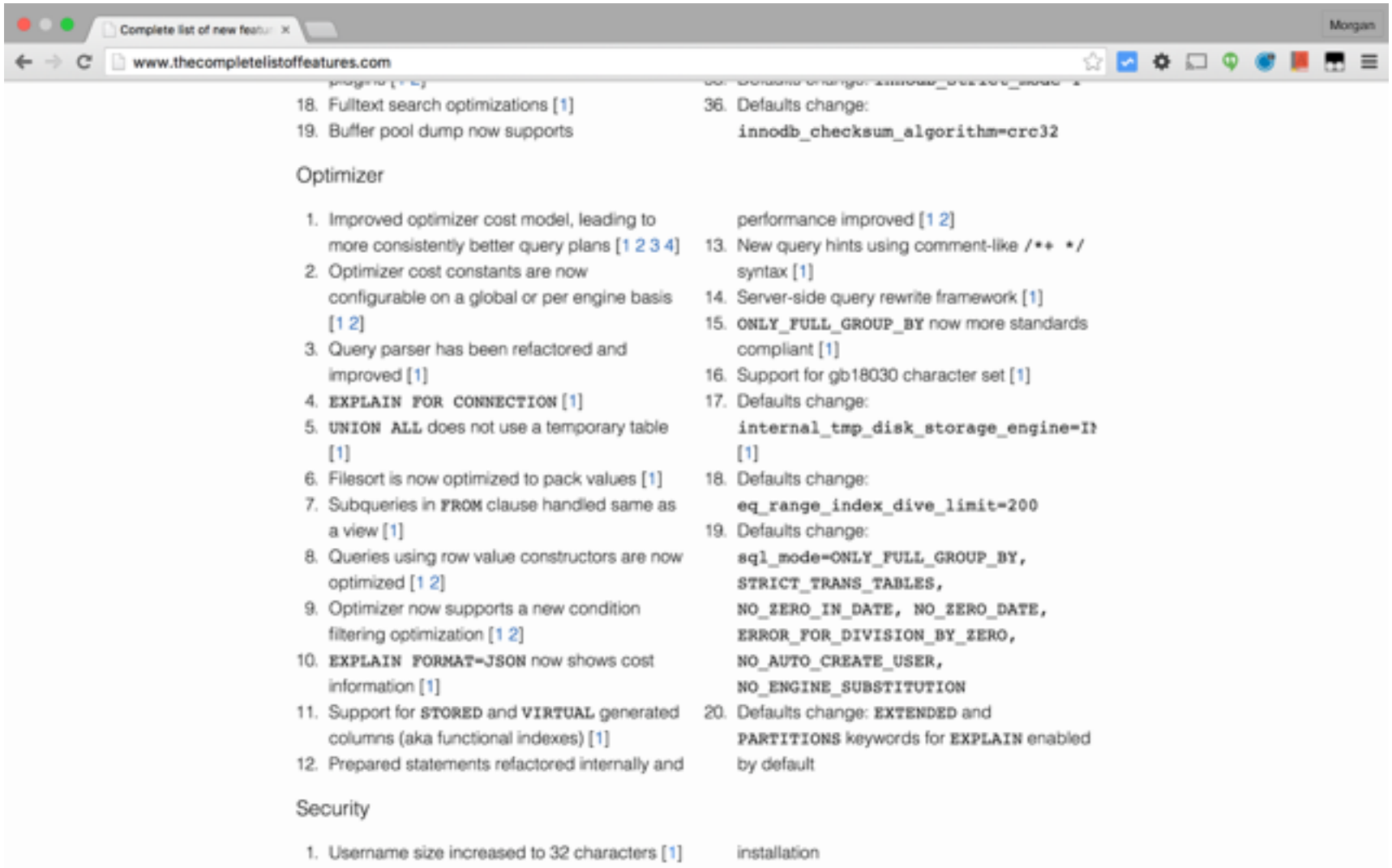
SQL thread [1]
11. Improved master dump thread performance

20. Defaults change: `slave_net_timeout=60`

InnoDB

1. Online buffer pool resize [1]
2. Improved crash recovery performance [1]
3. Improved read-only transaction scalability [1 2 3 4]
4. Improved read-write transaction scalability [1 2 3 4]
5. Several optimizations for high performance temporary tables [1 2 3 4 5]
6. `ALTER TABLE RENAME INDEX` only requires meta-data change [1]
7. Inplace `VARCHAR` only requires meta-data change [1]
8. `ALTER TABLE` performance improved [1]
9. Multiple page_cleaner threads [1]
10. Optimized buffer pool flushing [1]
11. New `innodb_log_checksum_algorithm` option [1]
12. Improved NUMA support [1]
13. General Tablespace support [1]
14. Transparent Page Compression [1]
15. `innodb_log_write_ahead_size` introduced to address potential 'read-on-write' with redo logs [1]
16. Fulltext indexes now support pluggable parsers [1]
17. Support for ngram and MeCab full-text parser `innodb_buffer_dump_pct` [1]
20. The doublewrite buffer is now disabled on filesystems that supports atomic writes (aka Fusion-io support) [1]
21. Page fill factor is now configurable [1]
22. Support for 32K and 64K pages [1]
23. Online undo log truncation [1]
24. `Update_time` meta data is now updated [1]
25. `TRUNCATE TABLE` is now atomic [1]
26. Memcached API performance improvements [1]
27. InnoDB now implements `information_schema.files` [1]
28. Defaults change: `innodb_file_format=Barracuda`
29. Defaults change: `innodb_large_prefix=1`
30. Defaults change: `innodb_page_cleaners=4`
31. Defaults change: `innodb_purge_threads=4`
32. Defaults change: `innodb_buffer_pool_dump_at_shutdowr`
33. Defaults change: `innodb_buffer_pool_load_at_startup=`
34. Defaults change: `innodb buffer pool dump pct=25`

THECOMPLETELISTOFFEATURES.COM



18. Fulltext search optimizations [1]
19. Buffer pool dump now supports

Optimizer

1. Improved optimizer cost model, leading to more consistently better query plans [1 2 3 4]
2. Optimizer cost constants are now configurable on a global or per engine basis [1 2]
3. Query parser has been refactored and improved [1]
4. `EXPLAIN FOR CONNECTION` [1]
5. `UNION ALL` does not use a temporary table [1]
6. Filesort is now optimized to pack values [1]
7. Subqueries in `FROM` clause handled same as a view [1]
8. Queries using row value constructors are now optimized [1 2]
9. Optimizer now supports a new condition filtering optimization [1 2]
10. `EXPLAIN FORMAT=JSON` now shows cost information [1]
11. Support for `STORED` and `VIRTUAL` generated columns (aka functional indexes) [1]
12. Prepared statements refactored internally and performance improved [1 2]
13. New query hints using comment-like `/*+ */` syntax [1]
14. Server-side query rewrite framework [1]
15. `ONLY_FULL_GROUP_BY` now more standards compliant [1]
16. Support for gb18030 character set [1]
17. Defaults change: `internal_tmp_disk_storage_engine=I` [1]
18. Defaults change: `eq_range_index_dive_limit=200`
19. Defaults change: `sql_mode=ONLY_FULL_GROUP_BY, STRICT_TRANS_TABLES, NO_ZERO_IN_DATE, NO_ZERO_DATE, ERROR_FOR_DIVISION_BY_ZERO, NO_AUTO_CREATE_USER, NO_ENGINE_SUBSTITUTION`
20. Defaults change: `EXTENDED` and `PARTITIONS` keywords for `EXPLAIN` enabled by default
36. Defaults change: `innodb_checksum_algorithm=crc32`

Security

1. Username size increased to 32 characters [1] installation

THECOMPLETELISTOFFEATURES.COM

Security

1. Username size increased to 32 characters [1]
2. Support for `IF [NOT] EXISTS` clause in `CREATE/DROP USER` [1]
3. Server option to require secure transport [1]
4. Support for multiple AES Encryption modes [1 2]
5. Support to `LOCK/UNLOCK` user accounts [1 2]
6. Support for password expiration policy [1 2]
7. Password strength enforcement
8. test database no longer created on installation
9. Anonymous users no longer created on installation
10. MySQL Firewall (commercial)
11. Random password generated by default on installation
12. New `ALTER USER` command
13. `SET password=''` now accepts a password instead of hash
14. Server now generates SSL keys by default
15. Insecure `old_password` hash removed
16. Ability to create utility users for stored programs that can not login [1]

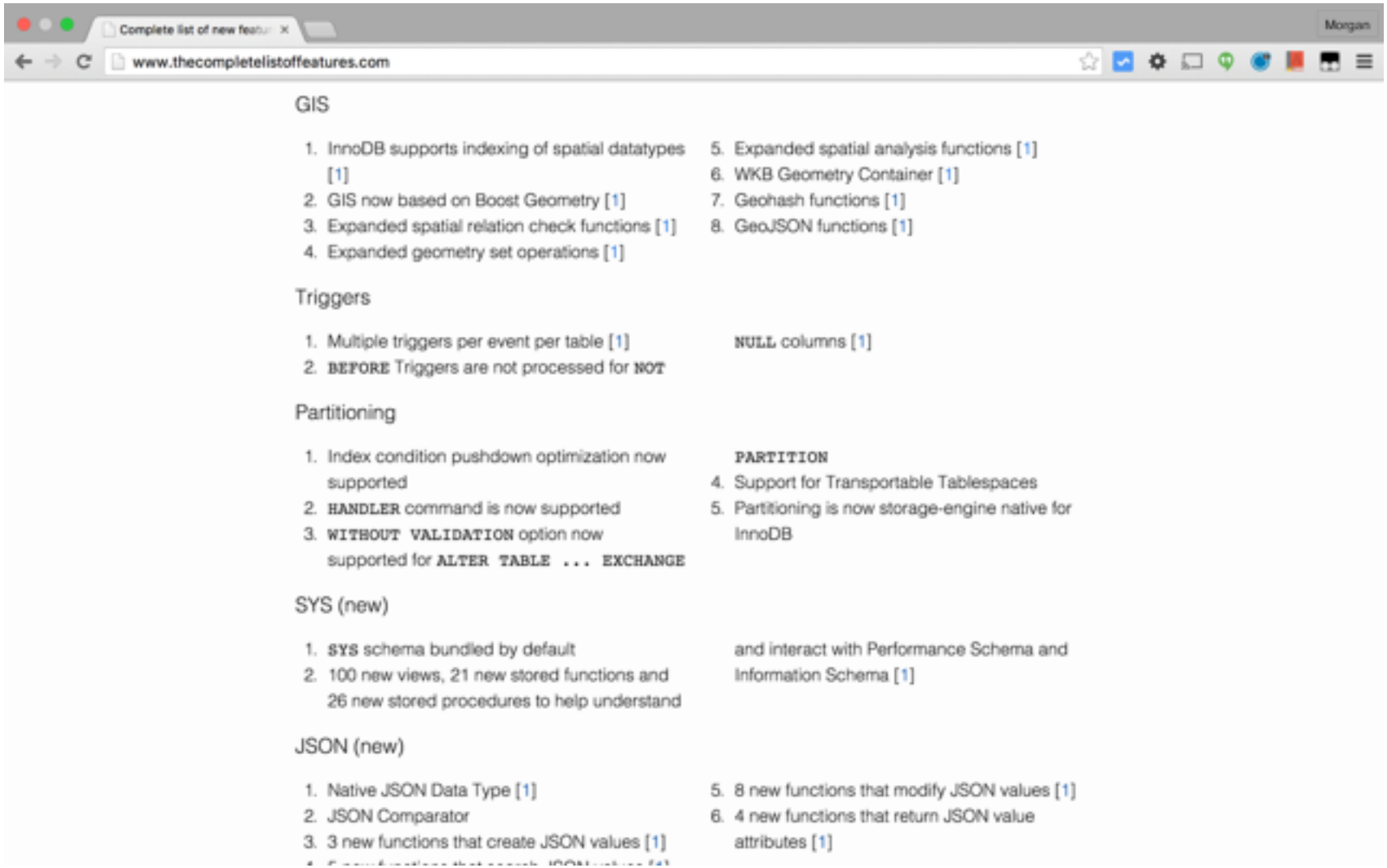
Performance Schema

1. Overhead has been reduced in client connect/disconnect phases
2. Memory footprint has been reduced
3. pfs_lock implementation has been improved
4. Table IO statistics are now batched for improved performance
5. Memory usage instrumentation
6. Stored programs instrumentation
7. Replication Instrumentation
8. Metadata Locking (MDL) Instrumentation
9. Transactions
10. Prepared Statements
11. Stage Progress
12. SX-lock and rw_lock
13. Thread status and variables
14. `Defaults change: performance-schema-consumer-events_statements_history=ON`
15. `Defaults change: performance-schema-consumer-events_transactions_history=ON`

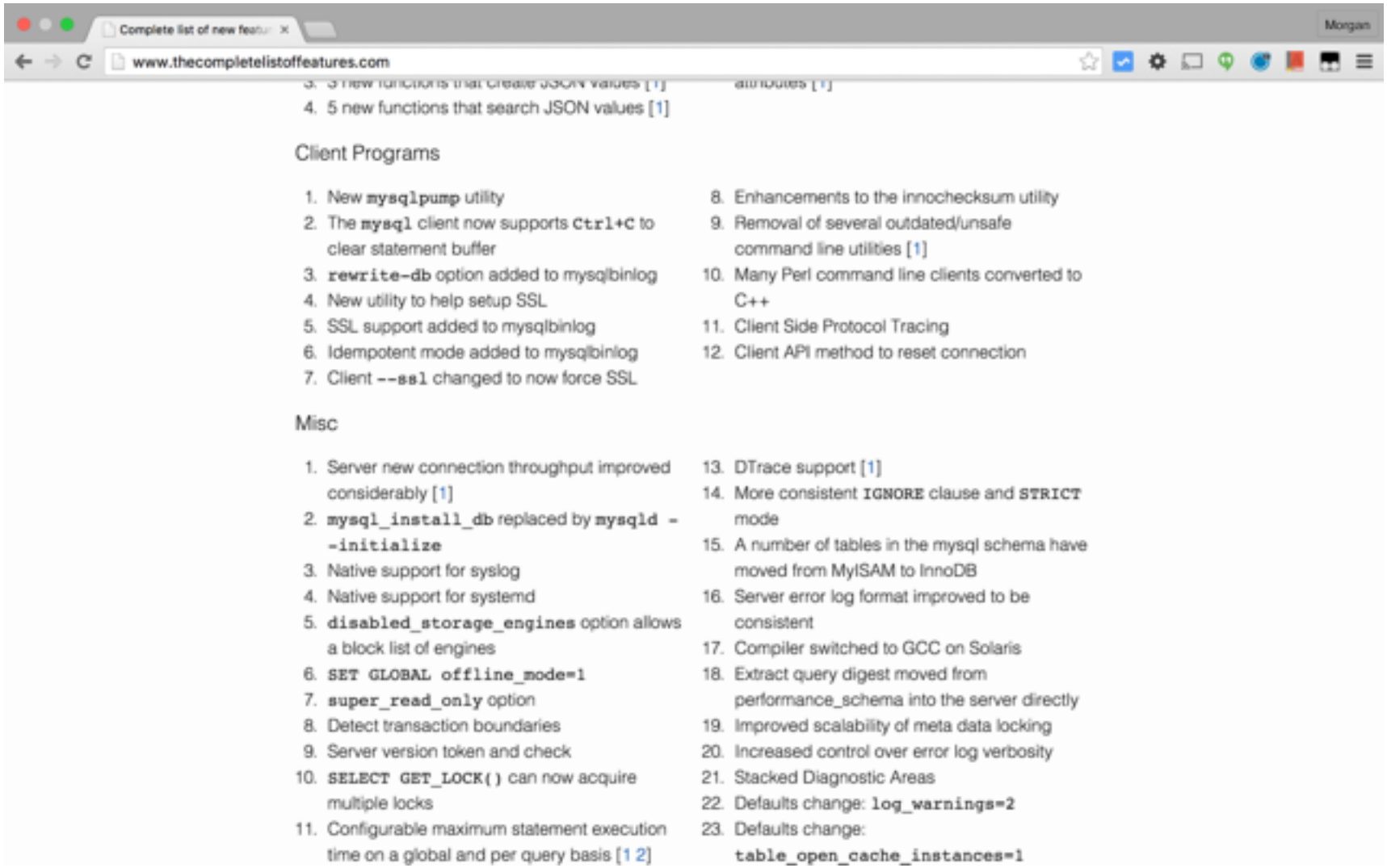
GIS

1. InnoDB supports indexing of spatial datatypes
5. Expanded spatial analysis functions [1]

THECOMPLETELISTOFFEATURES.COM



THECOMPLETELISTOFFEATURES.COM



4. 5 new functions that search JSON values [1]

Client Programs

1. New **mysqlpump** utility
2. The **mysql** client now supports **Ctrl+C** to clear statement buffer
3. **rewrite-db** option added to mysqlbinlog
4. New utility to help setup SSL
5. SSL support added to mysqlbinlog
6. Idempotent mode added to mysqlbinlog
7. Client **--ssl** changed to now force SSL
8. Enhancements to the innochecksum utility
9. Removal of several outdated/unsafe command line utilities [1]
10. Many Perl command line clients converted to C++
11. Client Side Protocol Tracing
12. Client API method to reset connection

Misc

1. Server new connection throughput improved considerably [1]
2. **mysql_install_db** replaced by **mysqld --initialize**
3. Native support for syslog
4. Native support for systemd
5. **disabled_storage_engines** option allows a block list of engines
6. **SET GLOBAL offline_mode=1**
7. **super_read_only** option
8. Detect transaction boundaries
9. Server version token and check
10. **SELECT GET_LOCK()** can now acquire multiple locks
11. Configurable maximum statement execution time on a global and per query basis [1 2]
13. DTrace support [1]
14. More consistent **IGNORE** clause and **STRICT** mode
15. A number of tables in the mysql schema have moved from MyISAM to InnoDB
16. Server error log format improved to be consistent
17. Compiler switched to GCC on Solaris
18. Extract query digest moved from performance_schema into the server directly
19. Improved scalability of meta data locking
20. Increased control over error log verbosity
21. Stacked Diagnostic Areas
22. Defaults change: **log_warnings=2**
23. Defaults change: **table_open_cache_instances=1**

Today's Agenda

1. Introduction
2. Core New JSON Features
3. To JSON or !JSON?

Core New JSON features in MySQL 5.7

- Native JSON datatype
- JSON Functions
- Generated Columns

The JSON Type

```
CREATE TABLE employees (data JSON);
INSERT INTO employees VALUES ('{"id": 1, "name": "Jane"}');
INSERT INTO employees VALUES ('{"id": 2, "name": "Joe"}');

SELECT * FROM employees;
+----------------------------+
| data                       |
+----------------------------+
| {"id": 1, "name": "Jane"} |
| {"id": 2, "name": "Joe"}  |
+----------------------------+
2 rows in set (0,00 sec)
```

JSON Type Tech Specs

- utf8mb4 character set
- Optimized for read intensive workload
- Parse and validation on insert only
- Dictionary
  - Sorted objects' keys
  - Fast access to array cells by index

JSON Type Tech Specs (cont.)

- Supports all native JSON types
  - Numbers, strings, bool
  - Objects, arrays
- Extended
  - Date, time, datetime, timestamp
  - Other

Advantages over TEXT/VARCHAR

1. Provides Document Validation:

```
INSERT INTO employees VALUES ('some random text');

ERROR 3130 (22032): Invalid JSON text: "Expect a
value here." at position 0 in value (or column)
'some random text'.
```

2. Efficient Binary Format
   Allows quicker access to object members and array elements

JSON Functions

```
SET @document = '[10, 20, [30, 40]]';

SELECT JSON_EXTRACT(@document, '$[1]');
+--------------------------------+
| JSON_EXTRACT(@document, '$[1]') |
+--------------------------------+
| 20                             |
+--------------------------------+
1 row in set (0.01 sec)
```

JSON_EXTRACT

- Accepts a ***JSON Path***, which is similar to a selector:


```
$("#type")
```


```
JSON_EXTRACT
(column_name, "$.type")
```

- JSON_EXTRACT also supports a short hand:
  column_name->"$.type"

Using Real Life Data

- Via **SF OpenData**
- 206K JSON objects representing subdivision parcels.


```
CREATE TABLE features (
id INT NOT NULL auto_increment primary key,
feature JSON NOT NULL
);
```

- Imported from https://github.com/zemirco/sf-city-lots-json + small tweaks

```
{
   "type":"Feature",
   "geometry":{
      "type":"Polygon",
      "coordinates":[
         [
            [-122.42200352825247,37.80848009696725,0],
            [-122.42207601332528,37.808835019815085,0],
            [-122.42110217434865,37.808803534992904,0],
            [-122.42106256906727,37.80860105681814,0],
            [-122.42200352825247,37.80848009696725,0]
         ]
      ]
   },
   "properties":{
      "TO_ST":"0",
      "BLKLOT":"0001001",
      "STREET":"UNKNOWN",
      "FROM_ST":"0",
      "LOT_NUM":"001",
      "ST_TYPE":null,
      "ODD_EVEN":"E",
      "BLOCK_NUM":"0001",
      "MAPBLKLOT":"0001001"
   }
}
```


JSON Search

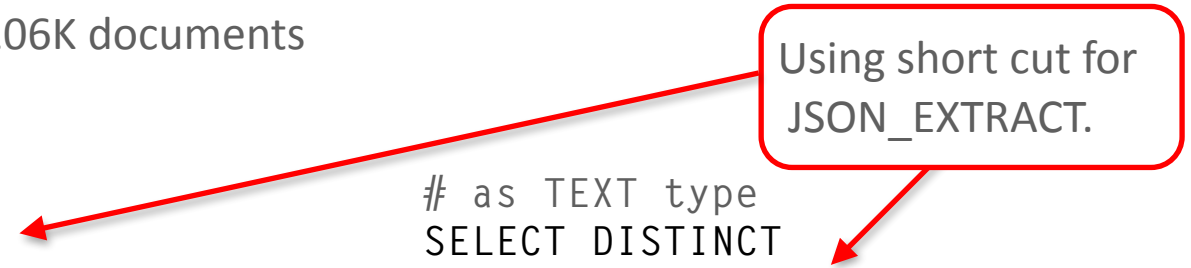
```
# Basic Find
SELECT * FROM features
WHERE feature->"$.properties.STREET" = 'MARKET'
LIMIT 1\G
*************************** 1. row ***************************
     id: 12250
feature: {"type": "Feature", "geometry": {"type": "Polygon",
"coordinates": [[[-122.39836263491878, 37.79189388899312,
0], [-122.39845248797837, 37.79233030084018, 0],
[-122.39768507706792, 37.7924280850133, 0],
[-122.39836263491878, 37.79189388899312, 0]]]},
"properties": {"TO_ST": "388", "BLKLOT": "0265003",
"STREET": "MARKET", "FROM_ST": "388", "LOT_NUM": "003",
"ST_TYPE": "ST", "ODD_EVEN": "E", "BLOCK_NUM": "0265",
"MAPBLKLOT": "0265003"}}
1 row in set (0.02 sec)
```

JSON Search

```
# Find where not exists
SELECT * FROM features
WHERE feature->"$.properties.STREET" IS NULL
LIMIT 1\G
Empty set (0.39 sec)
```

Naive Performance Comparison

Unindexed traversal of 206K documents




```
# as JSON type
SELECT DISTINCT
 feature->"$.type" as json_extract
FROM features;
+--------------+
| json_extract |
+--------------+
| "Feature"    |
+--------------+
1 row in set (1.25 sec)
```

```
# as TEXT type
SELECT DISTINCT
 feature->"$.type" as json_extract
FROM features;
+--------------+
| json_extract |
+--------------+
| "Feature"    |
+--------------+
1 row in set (12.85 sec)
```

Explanation: Binary format of JSON type is very efficient at searching. Storing as TEXT performs over 10x worse at traversal.

Introducing Generated Columns

id	my_integer	*my_integer_plus_one*
1	*10*	*11*
2	*20*	*21*
3	*30*	*31*
4	*40*	*41*

Column automatically maintained based on your specification.

```
CREATE TABLE t1 (
 id INT NOT NULL PRIMARY KEY auto_increment,
 my_integer INT,
 my_integer_plus_one INT AS (my_integer+1)
);
UPDATE t1 SET my_integer_plus_one = 10 WHERE id = 1;
ERROR 3105 (HY000): The value specified for generated
column 'my_integer_plus_one' in table 't1' is not
allowed.
```

Read-only of course

Generated Columns Support Indexes!

From table scan on 206K documents to index scan on 206K materialized values

```
ALTER TABLE features ADD feature_type VARCHAR(30) AS (feature-
>"$.type");
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

ALTER TABLE features ADD INDEX (feature_type);
Query OK, 0 rows affected (0.73 sec)
Records: 0  Duplicates: 0  Warnings: 0

SELECT DISTINCT feature_type FROM features;
+--------------+
| feature_type |
+--------------+
| "Feature"    |
+--------------+
1 row in set (0.06 sec)
```

Meta data change only (FAST). Does not need to touch table.

Creates index only. Does not modify table rows.

Down from 1.25 sec to 0.06 sec

Generated Columns (cont.)

- Used for “functional index”
- Available as either VIRTUAL *(default)* or STORED:

```
ALTER TABLE features ADD feature_type varchar(30) AS
(feature->"$.type") STORED;
Query OK, 206560 rows affected (4.70 sec)
Records: 206560  Duplicates: 0  Warnings: 0
```

- **Both** types of computed columns permit for indexes to be added.

Indexing Options Available

STORED	VIRTUAL
Primary and Secondary	Secondary Only
BTREE, Fulltext, GIS	BTREE Only
Mixed with fields	Mixed with fields
Requires table rebuild	No table rebuild
Not Online	INSTANT Alter
	Faster Insert

Bottom Line: Unless you need a PRIMARY KEY, FULLTEXT or GIS index VIRTUAL is probably better.

Virtual vs. Stored Performance

- Approximate *worst case* scenario via a table scan:

```
SELECT DISTINCT feature_type
FROM features;
+--------------+
| feature_type |
+--------------+
| "Feature"    |
+--------------+
```

```
VIRTUAL-TEXT (9.89 sec)
VIRTUAL-JSON (0.85 sec)
STORED-TEXT  (0.22 sec)
STORED-JSON  (0.24 sec)
```

Clarification: Since indexes are materialized (*stored*) themselves, the real-life case for STORED is when generating the column is computationally expensive *and* you can not use indexes effectively.

Unquote JSON String

```
SELECT
 DISTINCT JSON_UNQUOTE(feature->"$.type")
 as feature_type
FROM features;
+-----------------+
| feature_type    |
+-----------------+
| Feature         |
+-----------------+
1 row in set (1.22 sec)
```

JSON Path Search

- Provides a novice way to know the path. To retrieve via: *[[database.]table.]column->"$<path spec>"*

```
SELECT JSON_SEARCH(feature,
 'one', 'MARKET') AS
 extract_path
FROM features
WHERE id = 121254;
+------------------------+
| extract_path           |
+------------------------+
| "$.properties.STREET" |
+------------------------+
1 row in set (0.00 sec)
```

```
SELECT
feature->"$.properties.STREET"
 AS property_street
FROM features
WHERE id = 121254;
+-----------------+
| property_street |
+-----------------+
| "MARKET"        |
+-----------------+
1 row in set (0.00 sec)
```

JSON Array Creation

```
SELECT JSON_ARRAY(id,
 feature->"$.properties.STREET",
 feature->'$.type") AS json_array
FROM features ORDER BY RAND() LIMIT 3;
+---------------------------------+
| json_array                      |
+---------------------------------+
| [65298, "10TH", "Feature"]      |
| [122985, "08TH", "Feature"]     |
| [172884, "CURTIS", "Feature"]   |
+---------------------------------+
3 rows in set (2.66 sec)
```

JSON Object Creation

```
SELECT JSON_OBJECT('id', id,
 'street', feature->"$.properties.STREET",
 'type', feature->"$.type"
) AS json_object
FROM features ORDER BY RAND() LIMIT 3;
+-----------------------------------------------------------+
| json_object                                               |
+-----------------------------------------------------------+
| {"id": 122976, "type": "Feature", "street": "RAUSCH"}     |
| {"id": 148698, "type": "Feature", "street": "WALLACE"}    |
| {"id": 45214, "type": "Feature", "street": "HAIGHT"}      |
+-----------------------------------------------------------+
3 rows in set (3.11 sec)
```

JSON_REPLACE

```
SELECT JSON_REPLACE(feature, '$.type',
JSON_ARRAY('feature', 'bug')) as json_object FROM features
LIMIT 1;
+-----------------------------------------------------------+
| json_object                                               |
+-----------------------------------------------------------+
| {"type": ["feature", "bug"], "geometry": {"type": ..}} |
+-----------------------------------------------------------+
```

JSON Functions

- 5.7 supports functions to CREATE, SEARCH, MODIFY and RETURN JSON values:

JSON_ARRAY_APPEND()
JSON_ARRAY_INSERT()
JSON_ARRAY()
JSON_CONTAINS_PATH()
JSON_CONTAINS()
JSON_DEPTH()
JSON_EXTRACT()
JSON_INSERT()
JSON_KEYS()
JSON_LENGTH()
JSON_MERGE()
JSON_OBJECT()
JSON_QUOTE()
JSON_REMOVE()
JSON_REPLACE()
JSON_SEARCH()
JSON_SET()
JSON_TYPE()
JSON_UNQUOTE()
JSON_VALID()

https://dev.mysql.com/doc/refman/5.7/en/json-functions.html

JSON Comparator

```
SELECT CAST(1 AS JSON) = 1;
+-----------------------+
| CAST(1 AS JSON) = 1   |
+-----------------------+
|                     1 |
+-----------------------+
1 row in set (0.01 sec)
```

JSON value of 1 equals 1

JSON Objects Compare

```
SELECT CAST('{"num": 1.1}' AS JSON) = CAST('{"num": 1.1}' AS JSON);
+-------------------------------------------------------------------+
| CAST('{"num": 1.1}' AS JSON) = CAST('{"num": 1.1}' AS JSON)       |
+-------------------------------------------------------------------+
|                                                                 1 |
+-------------------------------------------------------------------+
1 row in set (0.00 sec)
```

Today's Agenda

1. Introduction
2. Core New JSON Features
3. To JSON or !JSON?

JSON or Column?

- Up to you!
- Advantages to both approaches

Storing as a *Column*

- Easier to apply a *schema* to your application
- *Schema* may make applications easier to maintain over time, as change is **controlled**;
  - Do not have to expect as many permutations
  - Allows some constraints over data

Storing as *JSON*

- More flexible way to represent data that is hard to model in *schema;*
  - Imagine you are a SaaS application serving many customers
  - Strong use-case to support **custom-fields**
  - Historically this may have used Entity–attribute–value model (EAV). Does not always perform well

JSON (cont.)

- Easier *denormalization*; an optimization that is important in some specific situations
- No painful schema changes*
- Easier prototyping
  - Fewer types to consider
  - No enforced schema, start storing values immediately

* MySQL 5.6 has Online DDL. This is not as large of an issue as it was historically.

Schema + Schemaless

```
CREATE TABLE pc_components (
 id INT NOT NULL PRIMARY KEY,
 description VARCHAR(60) NOT NULL,
 vendor VARCHAR(30) NOT NULL,
 serial_number VARCHAR(30) NOT NULL,
 attributes JSON NOT NULL
);
```

SSDs have *capacity_in_gb*, CPUs have a *core_count*. These attributes are not consistent across products.

Road Map

- In-place partial update of JSON/BLOB *(performance)*
- Partial streaming of JSON/BLOB (*replication*)
- Full text and GIS index on virtual columns
  - Currently works for "STORED"
- Multi-value Index
- Improved performance through condition pushdown

Resources

- http://mysqlserverteam.com/
- http://mysqlserverteam.com/tag/json/
- https://dev.mysql.com/doc/refman/5.7/en/mysql-nutshell.html
- http://dev.mysql.com/doc/relnotes/mysql/5.7/en/
- https://dev.mysql.com/doc/refman/5.7/en/json.html
- https://dev.mysql.com/doc/refman/5.7/en/json-functions.html
- http://www.thecompletelistoffeatures.com

Hardware and Software
Engineered to Work Together


ORACLE®