How to Use JSON in MySQL Wrong

Bill Karwin, Square Inc.

Santa Clara, California | April 23th – 25th, 2018

Me

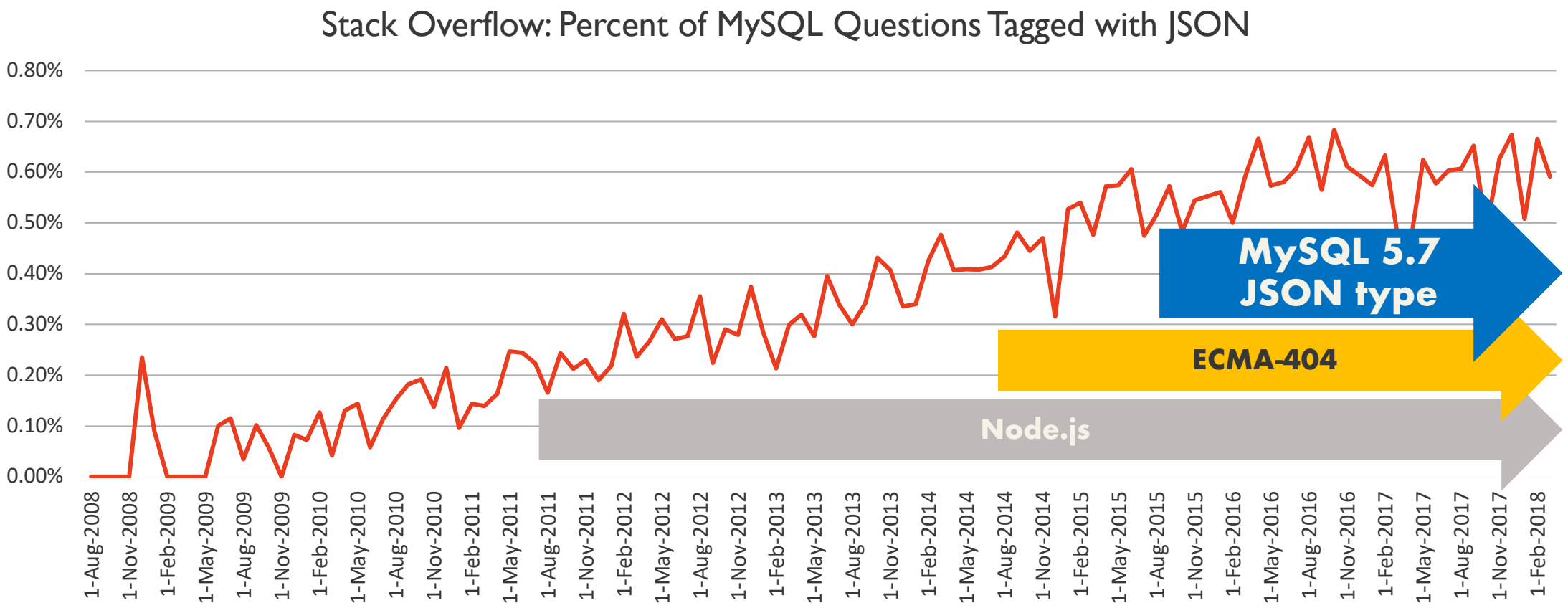
- Database Developer at Square Inc.
- MySQL Quality Contributor
- Oracle Ace Director
- Author of "SQL Antipatterns:
  Avoiding the Pitfalls of Database Programming"

Outline

- Why JSON?
- How do we load JSON data?
- What about LOAD JSON INFILE?
- What about performance?
- What's that about "generated" columns?
- What about searching multi-valued attributes?
- What about storage size?
- What about client interfaces?
- How to Use JSON in MySQL *Right*

Why JSON?

Interest in JSON Is Growing



https://data.stackexchange.com/stackoverflow/query/834289/mysql-and-json-tags-by-month

Why JSON?

- Portable data interchange format
  - Easy for humans to read
  - Easy for code to use
  - It's not XML
- Flexible schema in an SQL database
  - Semi-structured data
  - Like a document database
  - No more ALTER TABLE?

How do we load JSON data?

Test Data

- Data dump for dba.StackExchange.com
  - 987MB of XML
  - "All user content contributed to the Stack Exchange network is cc-by-sa 3.0 licensed, intended to be shared and remixed."
- LOAD XML INFILE to import data
- Then copy into equivalent JSON tables
- Let's see what trouble we find!

Test Data: https://archive.org/details/stackexchange
My code: https://github.com/billkarwin/bk-tools/tree/master/stackexchange

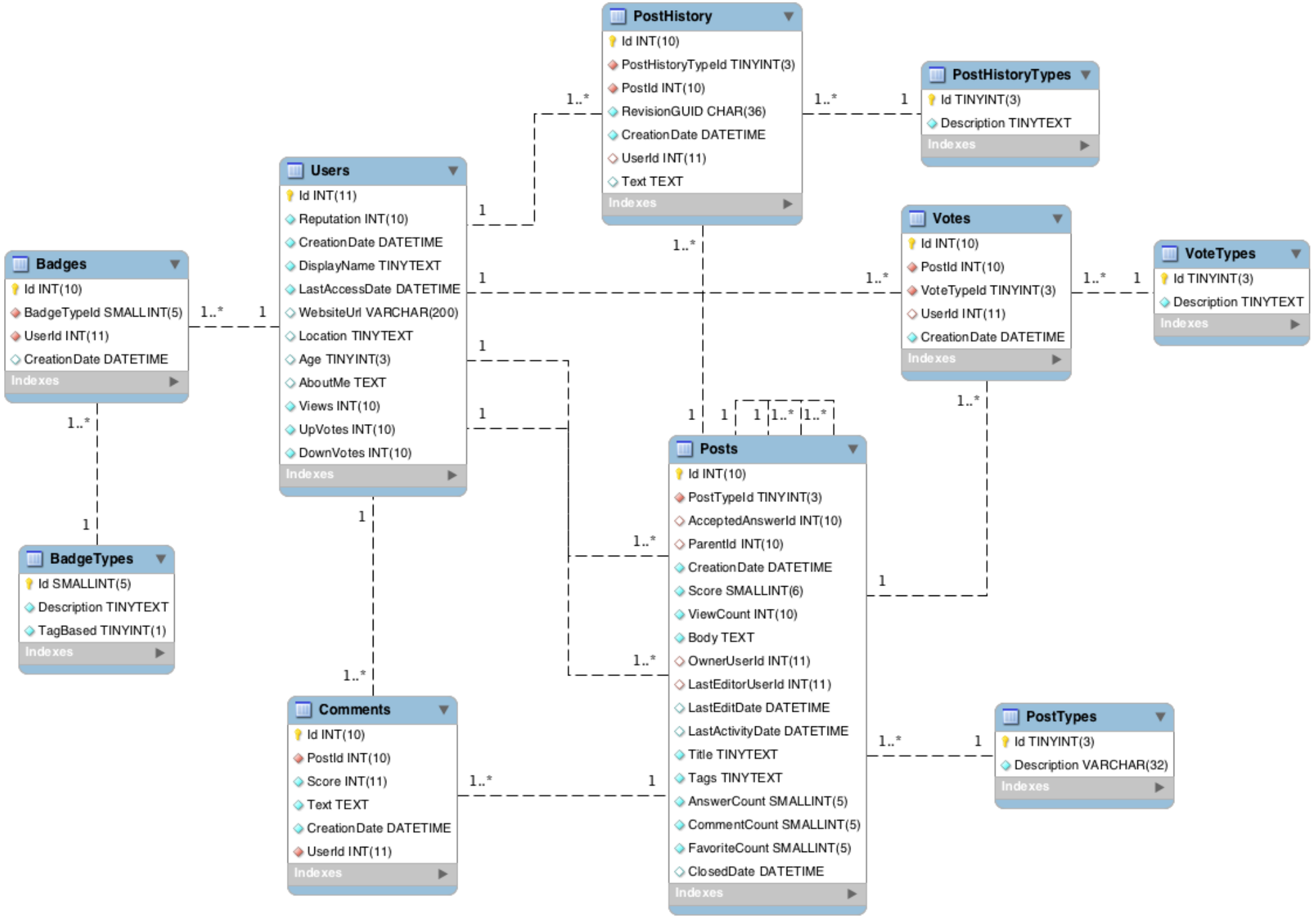

PostHistory
Id INT(10)
PostHistoryTypeId TINYINT(3)
PostId INT(10)
RevisionGUID CHAR(36)
CreationDate DATETIME
UserId INT(11)
Text TEXT
Indexes
PostHistoryTypes
Id TINYINT(3)
Description TINYTEXT
Indexes
Users
Id INT(11)
Reputation INT(10)
CreationDate DATETIME
DisplayName TINYTEXT
LastAccessDate DATETIME
WebsiteUrl VARCHAR(200)
Location TINYTEXT
Age TINYINT(3)
AboutMe TEXT
Views INT(10)
UpVotes INT(10)
DownVotes INT(10)
Indexes
Votes
Id INT(10)
PostId INT(10)
VoteTypeId TINYINT(3)
UserId INT(11)
CreationDate DATETIME
Indexes
VoteTypes
Id TINYINT(3)
Description TINYTEXT
Indexes
Badges
Id INT(10)
BadgeTypeId SMALLINT(5)
UserId INT(11)
CreationDate DATETIME
Indexes
BadgeTypes
Id SMALLINT(5)
Description TINYTEXT
TagBased TINYINT(1)
Indexes
Posts
Id INT(10)
PostTypeId TINYINT(3)
AcceptedAnswerId INT(10)
ParentId INT(10)
CreationDate DATETIME
Score SMALLINT(6)
ViewCount INT(10)
Body TEXT
OwnerUserId INT(11)
LastEditorUserId INT(11)
LastEditDate DATETIME
LastActivityDate DATETIME
Title TINYTEXT
Tags TINYTEXT
AnswerCount SMALLINT(5)
CommentCount SMALLINT(5)
FavoriteCount SMALLINT(5)
ClosedDate DATETIME
Indexes
Comments
Id INT(10)
PostId INT(10)
Score INT(11)
Text TEXT
CreationDate DATETIME
UserId INT(11)
Indexes
PostTypes
Id TINYINT(3)
Description VARCHAR(32)
Indexes

Table: Posts in Traditional Columns

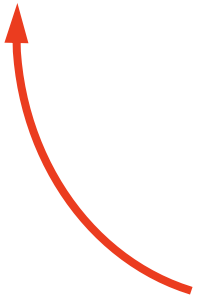
```
CREATE TABLE Posts (
  Id               INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  PostTypeId       TINYINT UNSIGNED NOT NULL,
  AcceptedAnswerId INT UNSIGNED NULL COMMENT 'if PostTypeId=1',
  ParentId         INT UNSIGNED NULL COMMENT 'if PostTypeId=2',
  CreationDate     DATETIME NOT NULL,
  Score            SMALLINT NOT NULL DEFAULT 0,
  ViewCount        INT UNSIGNED NOT NULL DEFAULT 0,
  Body             TEXT NOT NULL,
  OwnerUserId      INT NULL,
  LastEditorUserId INT NULL,
  LastEditDate     DATETIME NULL,
  LastActivityDate DATETIME NULL,
  Title            TINYTEXT NOT NULL,
  Tags             TINYTEXT NOT NULL,
  AnswerCount      SMALLINT UNSIGNED NOT NULL DEFAULT 0,
  CommentCount     SMALLINT UNSIGNED NOT NULL DEFAULT 0,
  FavoriteCount    SMALLINT UNSIGNED NOT NULL DEFAULT 0,
  ClosedDate       DATETIME NULL
);
```

Import from XML Source Data

```
LOAD XML LOCAL INFILE 'Posts.xml' INTO TABLE Posts
(
  Id, PostTypeId, AcceptedAnswerId,
  ParentId, @CreationDate, Score,
  ViewCount, Body, OwnerUserId,
  LastEditorUserId, @LastEditDate, @LastActivityDate,
  Title, Tags, AnswerCount,
  CommentCount, FavoriteCount, @ClosedDate
)
SET CreationDate     = STR_TO_DATE(@CreationDate, @DATETIME_ISO8601),
    LastEditDate     = STR_TO_DATE(@LastEditDate, @DATETIME_ISO8601),
    LastActivityDate = STR_TO_DATE(@LastActivityDate, @DATETIME_ISO8601),
    ClosedDate       = STR_TO_DATE(@ClosedDate, @DATETIME_ISO8601);
```

Table: PostsJson to Store Copy of Data in JSON

```
CREATE TABLE PostsJson (
  Id                INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  Data              JSON NOT NULL
);
```



I'll copy all the attributes
after the primary key
into a JSON column

A JSON Object Should Look Something Like This

```
{
  "PostTypeId": 1,
  "Title": "What are the main differences between InnoDB and MyISAM?",
  "CreationDate": "2011-01-03 20:46:03.000000",
  "Score": 180,
  ... more ...
}
```

What about LOAD JSON INFILE?

LOAD JSON INFILE?

There is no LOAD JSON INFILE statement (yet).

Please vote for these bug report / feature requests!

- LOAD DATA will not load a file into a JSON column unless converted
  - https://bugs.mysql.com/bug.php?id=79066
- Need a LOAD JSON statement
  - https://bugs.mysql.com/bug.php?id=79209

How to Convert Columns into JSON Fields?

```
INSERT INTO PostsJson (Id, Data)
SELECT Id,
  ...some magic...
FROM Posts;
```

Format JSON Using String Concatenation? Can You Spot the Mistakes?

```
INSERT INTO PostsJson (Id, Data)
SELECT Id, CONCAT('{',
  '"PostTypeId": "', PostTypeId, '", ',
  '"AcceptedAnswerId": "', AcceptedAnswerId, '", ',
  '"ParentId": "', ParentId, '" ',
  '"CreationDate": "', CreationDate, '", ',
  '"Score": "', Score, '", ',
  '"ViewCount": "', ViewCount, ', ',
  '"Body": "', Body, '", ',
  '"OwnerUserId" "', OwnerUserId, '", ',
  '"LastEditorUserId": "', LastEditorUserId, '", ',
  '"LastEditDate": "', LastEditDate, '", ',
  '"LastActivityDate": "', LastActivityDate,
  '"Title": "', Title, '", ',
  '"Tags": "', Tags, '", ',
  '"AnswerCount": "', AnswerCount, '", '
  '"CommentCount": "', CommentCount, '", ',
  '"FavoriteCount": "', FavoriteCount, '", ',
  '"ClosedDate": "', ClosedDate '", '
'}')
FROM Posts;
```

missing comma in JSON

missing double-quote in JSON

missing colon in JSON

missing termination

missing comma in CONCAT

It's Easy to Write Invalid JSON

Fix Mistakes:

- use [ ] around array
- use "key": "value", not "key", "value"
- use double-quotes, not single-quotes

Use JSON_OBJECT() or JSON_ARRAY() to Produce Valid JSON More Easily

```
INSERT INTO PostsJson (Id, Data)
SELECT Id, JSON_OBJECT(
   'PostTypeId', PostTypeId,
   'AcceptedAnswerId', AcceptedAnswerId,
   'ParentId', ParentId,
   'CreationDate', CreationDate,
   'Score', Score,
   'ViewCount', ViewCount,
   'Body', Body,
   'OwnerUserId', OwnerUserId,
   'LastEditorUserId', LastEditorUserId,
   'LastEditDate', LastEditDate,
   'LastActivityDate', LastActivityDate,
   'Title', Title,
   'Tags', Tags,
   'AnswerCount', AnswerCount,
   'CommentCount', CommentCount,
   'FavoriteCount', FavoriteCount,
   'ClosedDate', ClosedDate
)
FROM Posts;
```

JSON Extraction Function

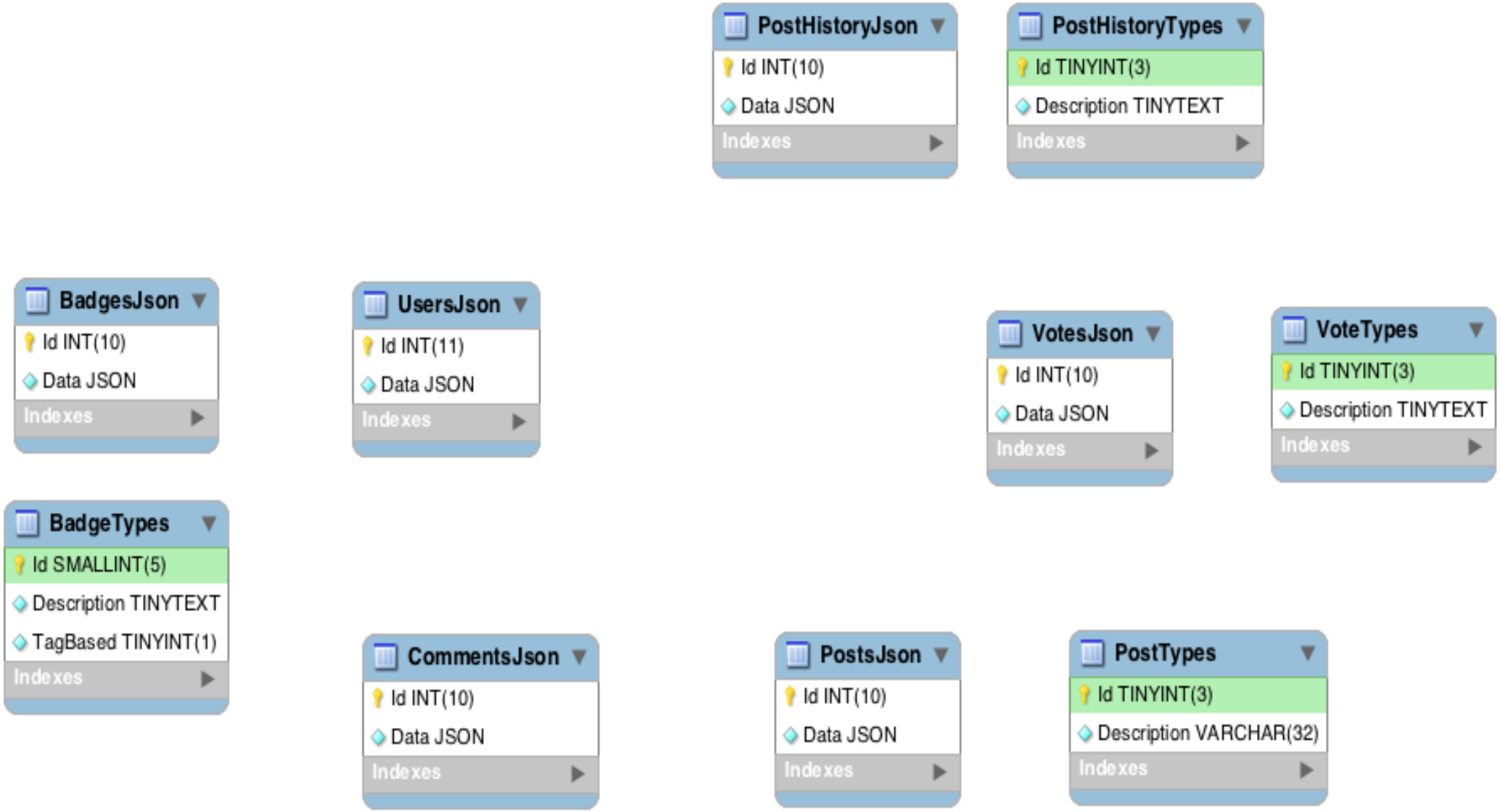
```
CREATE TABLE PostsJson (
  Id                INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  Data              JSON NOT NULL
);


SELECT Id,
  JSON_EXTRACT(Data, '$.Title'),
  JSON_EXTRACT(Data, '$.ParentId'),
  JSON_EXTRACT(Data, '$.Body')
FROM PostsJson
WHERE Id = 12828
```

JSON Extraction Operator

```
SELECT Id,
  Data->'$.Title',
  Data->'$.ParentId',
  Data->'$.Body'
FROM PostsJson
WHERE Id = 12828
```

One Down, Five to Go...

- ✓ Posts
- Badges
- Comments
- PostHistory
- Users
- Votes


PostHistoryJson
Id INT(10)
Data JSON
Indexes
PostHistoryTypes
Id TINYINT(3)
Description TINYTEXT
Indexes
BadgesJson
Id INT(10)
Data JSON
Indexes
UsersJson
Id INT(11)
Data JSON
Indexes
VotesJson
Id INT(10)
Data JSON
Indexes
VoteTypes
Id TINYINT(3)
Description TINYTEXT
Indexes
BadgeTypes
Id SMALLINT(5)
Description TINYTEXT
TagBased TINYINT(1)
Indexes
CommentsJson
Id INT(10)
Data JSON
Indexes
PostsJson
Id INT(10)
Data JSON
Indexes
PostTypes
Id TINYINT(3)
Description VARCHAR(32)
Indexes

What about performance?

Indexes for Optimization

- Avoid a table-scan – use an index to find matching rows

```
EXPLAIN SELECT * FROM PostHistory WHERE UserId = 2703;

           id: 1
  select_type: SIMPLE
        table: PostHistory
   partitions: NULL
         type: ref
possible_keys: UserId
          key: UserId
      key_len: 5
          ref: const
         rows: 138
     filtered: 100.00
        Extra: Using index
```

index on UserId

small number – close to the actual number of matching rows

No Support for Indexes

- Like any expression, a search on a JSON function can't use an index

```
EXPLAIN SELECT * FROM PostHistoryJson WHERE Data->'$.UserId' = 2703;

           id: 1
  select_type: SIMPLE
        table: PostHistoryJson
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 459294
     filtered: 100.00
        Extra: Using where
```

table-scan reads
ALL rows in the table

large number

How Does That Perform?

With Index on traditional table

Status	Duration
starting	0.000124
checking permissions	0.000012
Opening tables	0.000057
init	0.000010
System lock	0.000014
optimizing	0.000015
statistics	0.000094
preparing	0.000020
executing	0.000006
Sending data	0.000090
end	0.000009
query end	0.000012
closing tables	0.000015
freeing items	0.000027
cleaning up	0.000014

With Table-Scan on JSON table

Status	Duration
starting	0.000076
checking permissions	0.000008
Opening tables	0.000042
init	0.000007
System lock	0.000009
optimizing	0.000013
statistics	0.000019
preparing	0.000015
executing	0.000004
Sending data	0.694767
end	0.000012
query end	0.000009
closing tables	0.000011
freeing items	0.000017
cleaning up	0.000020

All Right—Can We Make an Index on JSON?

- No, JSON columns don't support indexes directly

```
ALTER TABLE PostsJson ADD INDEX (Data);

ERROR 3152 (42000): JSON column 'Data' supports indexing only via
generated columns on a specified JSON path.
```

What's that about "generated" columns?

Generated Columns

- Define a column as an expression using other columns in the same row

```
ALTER TABLE Posts ADD COLUMN CreationMonth TINYINT UNSIGNED
  AS (MONTH(CreationDate));
```

- You can then query it, like a VIEW at the column level

```
SELECT * FROM Posts WHERE CreationMonth = 4;
```

- It's still a table-scan so far

```
EXPLAIN SELECT * FROM Posts WHERE CreationMonth = 4;
```

```
+----+-------------+-------+------------+------+---------------+------+---------+------+--------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows   | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+--------+----------+-------------+
|  1 | SIMPLE      | Posts | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 145232 |    10.00 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+--------+----------+-------------+
```

Generated Columns

- Index this virtual column to optimize

```
ALTER TABLE Posts ADD KEY (CreationMonth);
EXPLAIN SELECT * FROM Posts WHERE CreationMonth = 4;
```

```
+----+-------------+-------+------------+------+---------------+---------------+---------+-------+-------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key           | key_len | ref   | rows  | filtered | Extra |
+----+-------------+-------+------------+------+---------------+---------------+---------+-------+-------+----------+-------+
|  1 | SIMPLE      | Posts | NULL       | ref  | CreationMonth | CreationMonth | 2       | const | 11658 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+---------------+---------+-------+-------+----------+-------+
```

- The index is also used if you use the expression

```
EXPLAIN SELECT * FROM Posts WHERE MONTH(CreationDate) = 4;
```

```
+----+-------------+-------+------------+------+---------------+---------------+---------+-------+-------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key           | key_len | ref   | rows  | filtered | Extra |
+----+-------------+-------+------------+------+---------------+---------------+---------+-------+-------+----------+-------+
|  1 | SIMPLE      | Posts | NULL       | ref  | CreationMonth | CreationMonth | 2       | const | 11658 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+---------------+---------+-------+-------+----------+-------+
```

Generated Columns Using JSON

- You can use any scalar expression—including JSON functions

```
ALTER TABLE PostsJson ADD COLUMN CreationDate DATETIME
  AS (Data->'$.CreationDate');
```

- Add index to optimize

```
ALTER TABLE PostsJson ADD KEY (CreationDate);
EXPLAIN SELECT * FROM PostsJson WHERE CreationDate = '2018-04-20';
```

```
+----+-------------+-----------+------------+------+---------------+--------------+---------+-------+------+----------+-------+
| id | select_type | table     | partitions | type | possible_keys | key          | key_len | ref   | rows | filtered | Extra |
+----+-------------+-----------+------------+------+---------------+--------------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | PostsJson | NULL       | ref  | CreationDate  | CreationDate | 6       | const |    1 |   100.00 | NULL  |
+----+-------------+-----------+------------+------+---------------+--------------+---------+-------+------+----------+-------+
```

- But with JSON, using the expression doesn't cue the use of the index

```
EXPLAIN SELECT * FROM PostsJson WHERE Data->'$.CreationDate' = '2018-04-20';
```

```
+----+-------------+-----------+------------+------+---------------+------+---------+-------+--------+----------+-------------+
| id | select_type | table     | partitions | type | possible_keys | key  | key_len | ref   | rows   | filtered | Extra       |
+----+-------------+-----------+------------+------+---------------+------+---------+-------+--------+----------+-------------+
|  1 | SIMPLE      | PostsJson | NULL       | ALL  | NULL          | NULL | NULL    | NULL  | 119795 |   100.00 | Using where |
+----+-------------+-----------+------------+------+---------------+------+---------+-------+--------+----------+-------------+
```

Declare a Foreign Key on a Generated Column

```
ALTER TABLE PostsJson
  ADD COLUMN PostTypeId TINYINT UNSIGNED
    AS (Data->'$.PostTypeId'),
  ADD FOREIGN KEY (PostTypeId)
    REFERENCES PostTypes(Id);
```

```
ERROR 1215 (HY000): Cannot add foreign key constraint
```

```
ALTER TABLE PostsJson
  ADD COLUMN PostTypeId TINYINT UNSIGNED
    AS (Data->'$.PostTypeId') STORED,
  ADD FOREIGN KEY (PostTypeId)
    REFERENCES PostTypes(Id);
```

foreign keys use STORED generated columns, not VIRTUAL

But the Next Foreign Key Doesn't Work?

```
ALTER TABLE PostsJson
  ADD COLUMN AcceptedAnswerId INT UNSIGNED
    AS (Data->'$.AcceptedAnswerId') STORED,
  ADD FOREIGN KEY (AcceptedAnswerId)
    REFERENCES Posts(Id);
```

```
ERROR 3156 (22018): Invalid JSON value for CAST to INTEGER
from column json_extract at row 1
```

Naturally, Some Posts Don't Have an Accepted Answer

```
SELECT JSON_PRETTY(Data) FROM PostsJson LIMIT 1;
```


```
{
  "Tags": "<mysql><innodb><myisam>",
  "Score": 180,
  "Title": "What are the main differences between InnoDB and MyISAM?",
  "ParentId": null,
  "ViewCount": 172059,
  "ClosedDate": null,
  "PostTypeId": 1,
  "AnswerCount": 10,
  "OwnerUserId": 8,
  "CommentCount": 1,
  "CreationDate": "2011-01-03 20:46:03.000000",
  "LastEditDate": null,
  "FavoriteCount": 105,
  "AcceptedAnswerId": null,
  "LastActivityDate": "2017-03-09 13:33:48.000000",
  "LastEditorUserId": null
}
```

Is That a SQL NULL? No...

```
SELECT IFNULL(Data->'$.AcceptedAnswerId', 'missing')
  AS AcceptedAnswerId
FROM PostsJson WHERE Id = 12828;

+--------------------+
| AcceptedAnswerId |
+--------------------+
| null               |
+--------------------+
```

a real SQL NULL would have
defaulted to the second argument;
it would also be spelled in caps

Is That a String 'null'? No...

```
SELECT Data->'$.AcceptedAnswerId' = 'null'
  AS AcceptedAnswerId
FROM PostsJson WHERE Id = 12828;

+--------------------+
| AcceptedAnswerId   |
+--------------------+
|                  0 |
+--------------------+
```

how can
'null' != 'null'?

It's Actually a Very Small JSON Document: 'null'

```
CREATE TABLE WhatIsIt AS
  SELECT Data->'$.AcceptedAnswerId'
    AS AcceptedAnswerId
  FROM PostsJson WHERE Id = 12828;
```

the type is revealed

```
CREATE TABLE `WhatIsIt` (
  `AcceptedAnswerId` json DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

https://bugs.mysql.com/bug.php?id=85755

Get a Scalar Value with JSON_UNQUOTE() or the Operator

```
SELECT Data->>'$.AcceptedAnswerId' = 'null'
  AS AcceptedAnswerId
FROM PostsJson WHERE Id = 12828;

+--------------------+
| AcceptedAnswerId   |
+--------------------+
|                  1 |
+--------------------+
```

now it's *string*
'null' = 'null'

But This Still Doesn't Work

```
ALTER TABLE PostsJson
  ADD COLUMN AcceptedAnswerId INT UNSIGNED
    AS (Data->>'$.AcceptedAnswerId') STORED,
  ADD FOREIGN KEY (AcceptedAnswerId)
    REFERENCES Posts(Id);
```

```
ERROR 1366 (HY000): Incorrect integer value: 'null' for
column 'AcceptedAnswerId' at row 1
```

strict mode is on by default,
so implicit type conversions
are errors

Disable Strict Mode? Bad Idea...

integer value of string 'null' = 0
but there is no Posts.Id = 0

```
ALTER TABLE PostsJson
  ADD COLUMN AcceptedAnswerId INT UNSIGNED
    AS (Data->>'$.AcceptedAnswerId') STORED,
  ADD FOREIGN KEY (AcceptedAnswerId)
    REFERENCES Posts(Id);
```

```
ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint
fails (`stackexchange`.`#sql-6182_7d`, CONSTRAINT `postsjson_ibfk_2` FOREIGN KEY
(`AcceptedAnswerId`) REFERENCES `posts` (`id`))
```

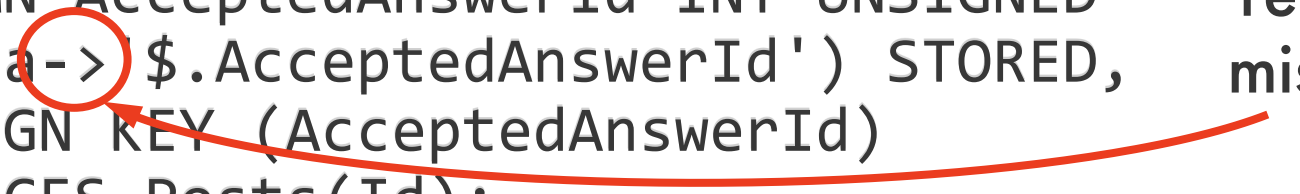
Instead, Convert the String 'null' to SQL NULL

```
ALTER TABLE PostsJson
  ADD COLUMN AcceptedAnswerId INT UNSIGNED
    AS (NULLIF(Data->>'$.AcceptedAnswerId', 'null')) STORED,
  ADD FOREIGN KEY (AcceptedAnswerId)
    REFERENCES Posts(Id);


Query OK, 150657 rows affected (4.04 sec)
Records: 150657  Duplicates: 0  Warnings: 0
```

Alternative: Remove Each Attribute That Is 'null'

```
UPDATE PostsJson
SET Data = JSON_REMOVE(Data, '$.AcceptedAnswerId')
WHERE Data->>'$.AcceptedAnswerId' = 'null';

Query OK, 120030 rows affected (7.64 sec)
Rows matched: 120030  Changed: 120030  Warnings: 0
```

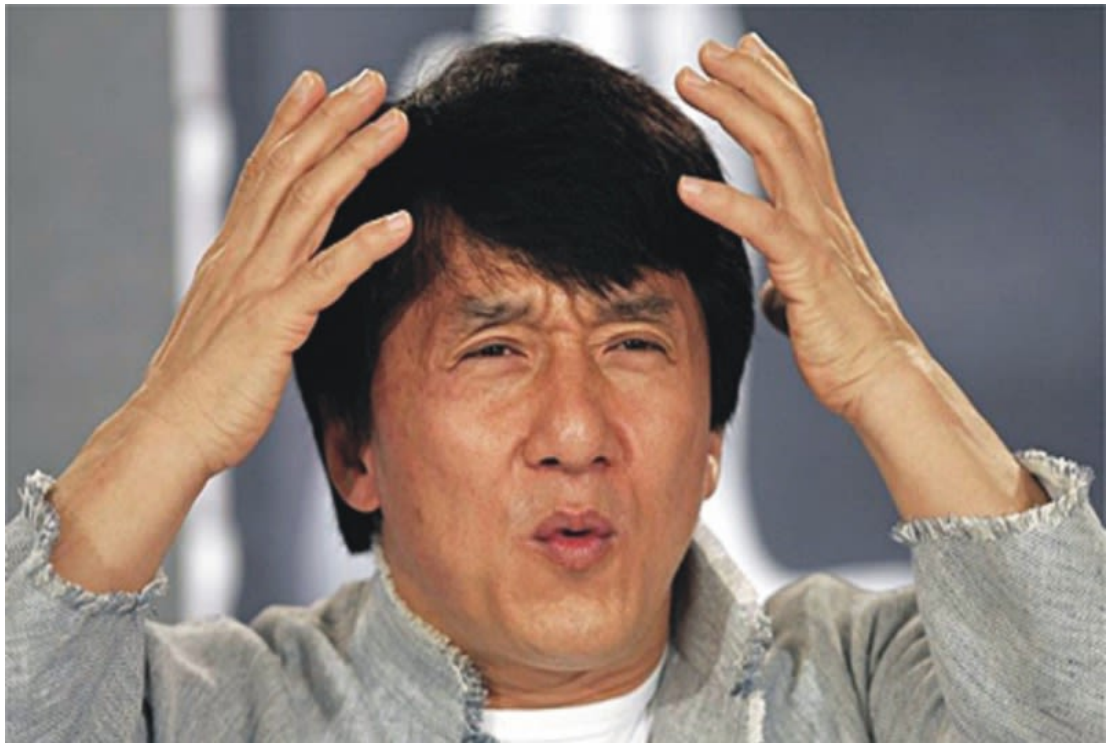
```
ALTER TABLE PostsJson
  ADD COLUMN AcceptedAnswerId INT UNSIGNED
    AS (Data->'$.AcceptedAnswerId') STORED,
  ADD FOREIGN KEY (AcceptedAnswerId)
    REFERENCES Posts(Id);

Query OK, 150657 rows affected (4.71 sec)
Records: 150657  Duplicates: 0  Warnings: 0
```



simple extract operator returns SQL NULL for missing JSON attribute

How to Index JSON Attributes, Really

- ALTER TABLE to add a generated columns with expressions to extract the JSON attributes
- A foreign key requires generated columns to be STORED, not VIRTUAL
  - Adding a VIRTUAL generated column is an online DDL change
  - Adding a STORED generated column must perform a table-copy
- Nullable attributes must be either:
  - Removed from the JSON document, so JSON_EXTRACT() returns an SQL NULL
  - Extracted, unquoted, and then converted to SQL NULL in the generated as expression
- Finally, declare KEY or FOREIGN KEY on the generated columns



What about searching multi-valued attributes?

Some Attributes Are Multi-Valued

```
SELECT Data FROM PostsJson LIMIT 1;
{
  "Tags": "<mysql><innodb><myisam>",
  ...
```

```
SELECT SUBSTRING_INDEX(
  SUBSTRING_INDEX(Data->>'$.Tags', '<', 2), '>', -2) AS Tag1
FROM PostsJson LIMIT 1;
```

```
+---------+
| Tag1    |
+---------+
| <mysql> |
+---------+
```

Convert a List into a JSON Array

```
UPDATE PostsJson
SET Data = JSON_SET(Data, '$.Tags', JSON_ARRAY(
  SUBSTRING_INDEX(SUBSTRING_INDEX(Data->>'$.Tags', '<', 2), '>', -2),
  SUBSTRING_INDEX(SUBSTRING_INDEX(Data->>'$.Tags', '<', 3), '>', -2),
  SUBSTRING_INDEX(SUBSTRING_INDEX(Data->>'$.Tags', '<', 4), '>', -2),
  SUBSTRING_INDEX(SUBSTRING_INDEX(Data->>'$.Tags', '<', 5), '>', -2),
  SUBSTRING_INDEX(SUBSTRING_INDEX(Data->>'$.Tags', '<', 6), '>', -2)));

SELECT Data->'$.Tags' AS Tags FROM PostsJson LIMIT 1;

+-----------------------------------------------------------+
| Tags                                                      |
+-----------------------------------------------------------+
| ["<mysql>", "<innodb>", "<myisam>", "<myisam>", "<myisam>"] |
+-----------------------------------------------------------+
```

Search the Array with JSON_SEARCH()

```
SELECT Id FROM PostsJson
WHERE JSON_SEARCH(Data->'$.Tags', 'one', '<innodb>') IS NOT NULL
LIMIT 1;
```

```
+-------+
| Id    |
+-------+
| 19298 |
+-------+
```

Can We Index That? Yes—But Only for One Specific Tag

```
ALTER TABLE PostsJson
  ADD COLUMN TagInnodb BOOLEAN AS
    (JSON_SEARCH(Data->'$.Tags', 'one', '<innodb>') IS NOT NULL),
  ADD KEY (TagInnoDB);
```

```
EXPLAIN SELECT * FROM PostsJson WHERE TagInnoDB = 1;
```

```
+----+-------------+-----------+------------+------+---------------+-----------+---------+-------+------+----------+-------+
| id | select_type | table     | partitions | type | possible_keys | key       | key_len | ref   | rows | filtered | Extra |
+----+-------------+-----------+------------+------+---------------+-----------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | PostsJson | NULL       | ref  | TagInnodb     | TagInnodb | 2       | const |    1 |   100.00 | NULL  |
+----+-------------+-----------+------------+------+---------------+-----------+---------+-------+------+----------+-------+
```

Can We Index Every Possible Tag Value? Probably Not...

```
ALTER TABLE PostsJson
  ADD COLUMN TagMysql BOOLEAN AS (JSON_SEARCH(Data->'$.Tags', 'one', '<mysql>') IS NOT NULL),
  ADD KEY (TagMysql);

ALTER TABLE PostsJson
  ADD COLUMN TagMongodb BOOLEAN AS (JSON_SEARCH(Data->'$.Tags', 'one', '<mongodb>') IS NOT NULL),
  ADD KEY (TagMongodb);

ALTER TABLE PostsJson
  ADD COLUMN TagOracle BOOLEAN AS (JSON_SEARCH(Data->'$.Tags', 'one', '<oracle>') IS NOT NULL),
  ADD KEY (TagOracle);

ALTER TABLE PostsJson
  ADD COLUMN TagSqlite BOOLEAN AS (JSON_SEARCH(Data->'$.Tags', 'one', '<sqlite>') IS NOT NULL),
  ADD KEY (TagSqlite);

...
```

```
ERROR 1069 (42000): Too many keys specified; max 64 keys allowed
```

How Can We Index Any Tag?

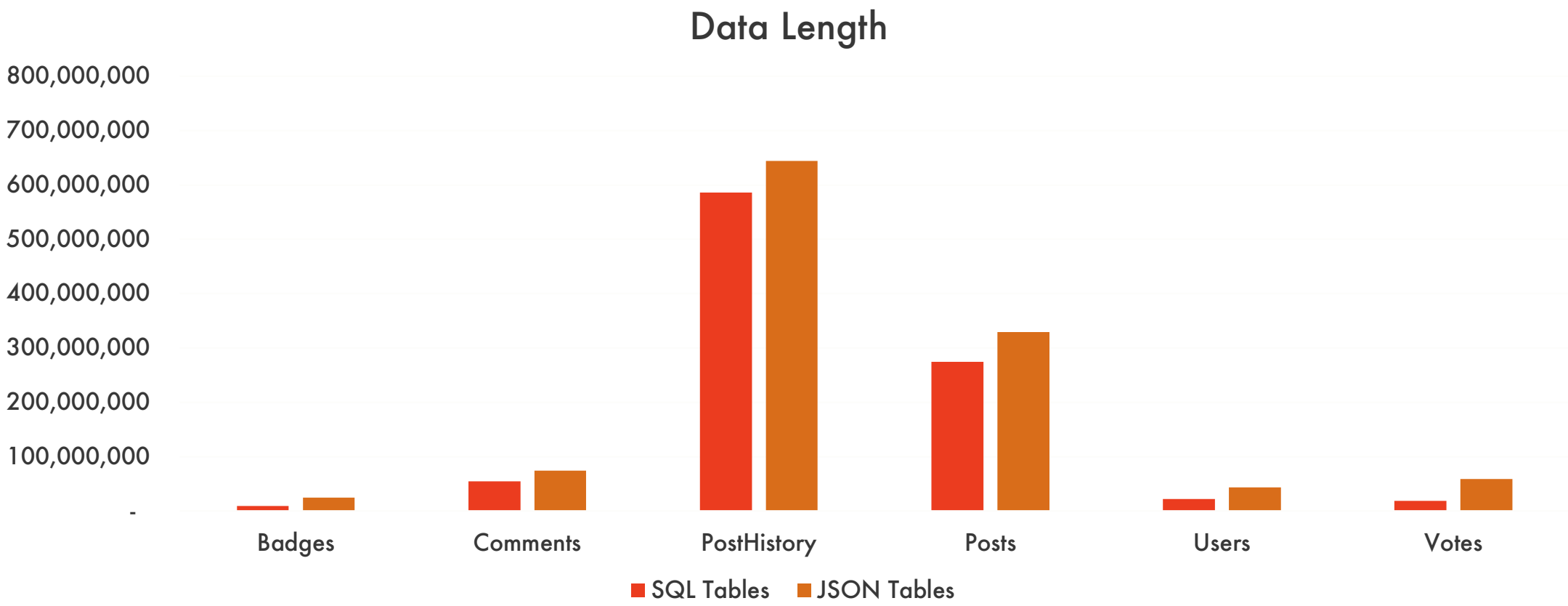
- Many-to-many relationship between Posts and Tags needs its own table:

```
CREATE TABLE PostsTags (
  PostId INT UNSIGNED,
  TagId INT UNSIGNED,
  PRIMARY KEY (PostId, TagId),
  FOREIGN KEY (PostId) REFERENCES PostsJson,
  FOREIGN KEY (TagId) REFERENCES Tags
);
```

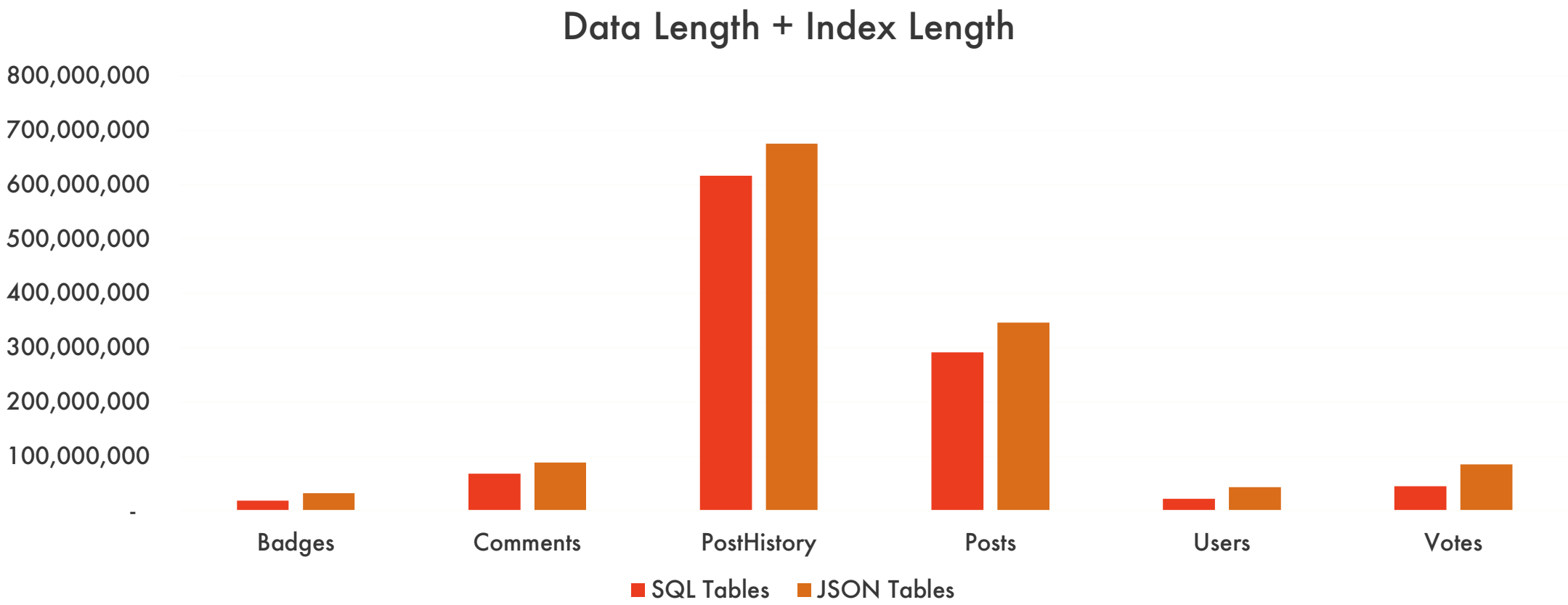
- Fill this table with one row per pairing
- Use one index to search for any tag!

What about storage size?

JSON Data Takes 120% – 317% Space (average 194%)



JSON Data + Indexes Takes 110% – 202% Space (average 154%)



Your Mileage May Vary

- The increased size of JSON depends on several factors:
  - Number of indexes
  - Generated columns as STORED vs. VIRTUAL
  - Data types of attributes
  - Length of attribute values
  - Length of attribute names

Length of INT Values Matters

```
CREATE TABLE IntLengthTest1 (id SERIAL PRIMARY KEY, d JSON);
INSERT INTO IntLengthTest1 SET d = JSON_OBJECT('a', 1234567890);

CREATE TABLE IntLengthTest2 LIKE IntLengthTest1;
INSERT INTO IntLengthTest2 SET d = JSON_OBJECT('a', '1234567890');
```

Double the rows until both tables have 1048576 rows

```
INSERT INTO IntLengthTest1 (d)
SELECT d FROM IntLengthTest1; /* repeat 20 times */

INSERT INTO IntLengthTest2 (d)
SELECT d FROM IntLengthTest2; /* repeat 20 times */
```

Length of INT Values Matters

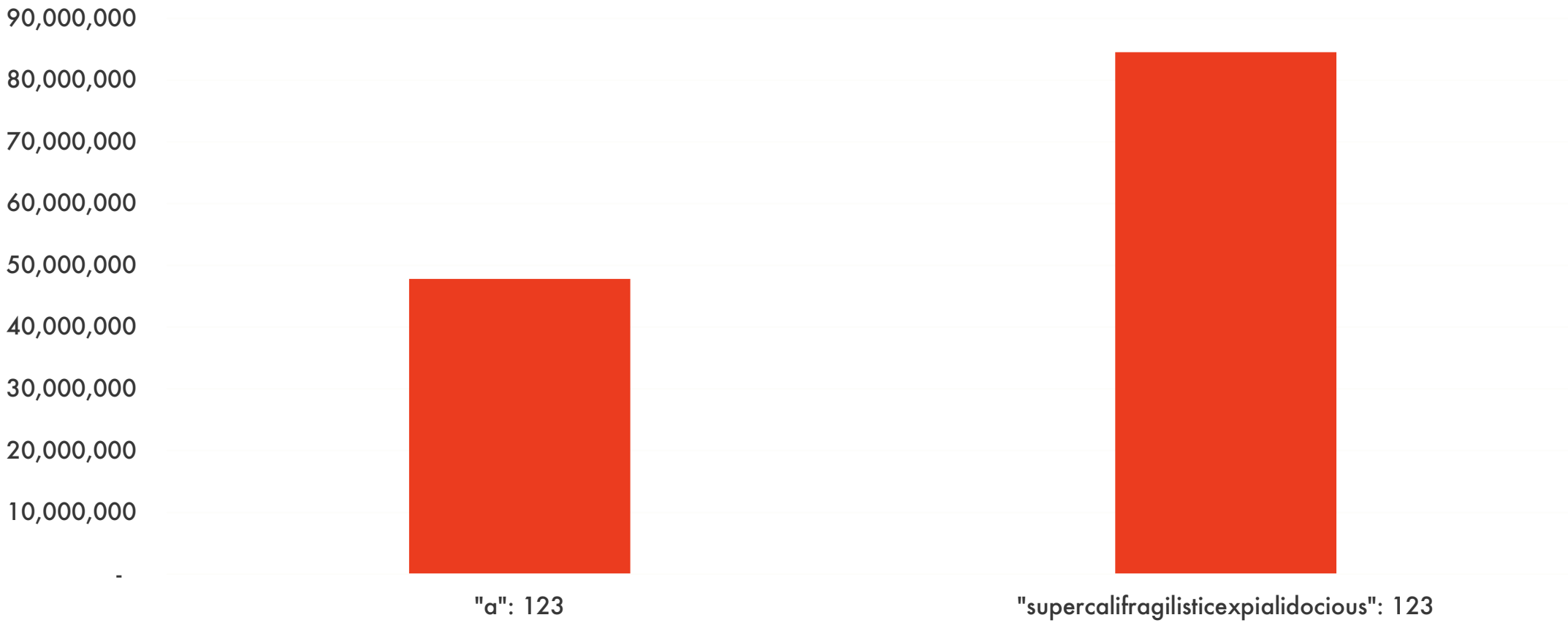
Length of Attribute Names Matters

```
CREATE TABLE AttrLengthTest1 (id SERIAL PRIMARY KEY, d JSON);
INSERT INTO AttrLengthTest1 SET d = JSON_OBJECT('a', 123);


CREATE TABLE AttrLengthTest2 LIKE AttrLengthTest1;
INSERT INTO AttrLengthTest1 SET d =
JSON_OBJECT('supercalifragilisticexpialidocious', 123);
```

Double the rows until both tables have 1048576 rows

```
INSERT INTO AttrLengthTest1 (d)
SELECT d FROM AttrLengthTest1; /* repeat 20 times */
INSERT INTO AttrLengthTest2 (d)
SELECT d FROM AttrLengthTest2; /* repeat 20 times */
```

Length of Attribute Names Matters



What about client interfaces?

Java

- JSON data is returned as java.lang.String
- Use a library to parse a JSON string into a Java object, or format an object into JSON
  - JSON.simple: https://github.com/fangyidong/json-simple
  - FasterXML Jackson: https://github.com/FasterXML/jackson
  - Google GSON: https://github.com/google/gson
  - Oracle JSONP: https://jsonp.java.net/
  - Good article with performance comparisons:
    - *https://blog.takipi.com/the-ultimate-json-library-json-simple-vs-gson-vs-jackson-vs-json/*

Go

- JSON data is returned as a string
- Use the standard json package
  - json.Unmarshal() to parse a JSON string into a Go array or map
  - json.Marshal() to convert an array or an object to JSON string

PHP

- JSON data is returned as a string
- Use builtin functions to convert from JSON string to/from PHP structures
  - json_decode()
  - json_encode()

How to Use JSON in MySQL *Right*

Stability vs. Maneuverability

https://commons.wikimedia.org/wiki/Category:Cessna_landings#/media/File:Mainland_Air_Cessna_152_ZK-FCQ_Dunedin,_NZ.jpg

https://commons.wikimedia.org/wiki/Lockheed_Martin_F-22_Raptor#/media/File:Raptor_F-22_27th.jpg

Use JSON Like a Document Store

- Search by the PRIMARY KEY where possible

```
SELECT * FROM PostsJson WHERE Id = 19298;
```

- Good to use indexed generated columns in WHERE or ORDER BY

```
SELECT * FROM PostsJson WHERE OwnerUserId = 2703
ORDER BY CreatedDate;
```

- Extracting fields is fine when only displaying them in the SELECT-list

```
SELECT Data->>'$.Title' AS Title FROM PostsJson WHERE Id = 19298;
```

Use JSON Like a Document Store

- X DevAPI to use MySQL like a document store
  - No need to install another NoSQL product
  - https://dev.mysql.com/doc/refman/5.7/en/document-store.html
  - Go to the presentation "MySQL 8.0: a Document Store with all the benefits of a transactional RDBMS"

Use SQL and Normalization

- Use JSON as flexible schema only when you need it
  - User-defined fields
  - Alternative to EAV
  - Log-type data
- Storing nested arrays or objects is denormalized design
- Use dependent tables for multi-valued attributes
  if you want to search or sort by them
- Use traditional columns instead of JSON fields for constraints

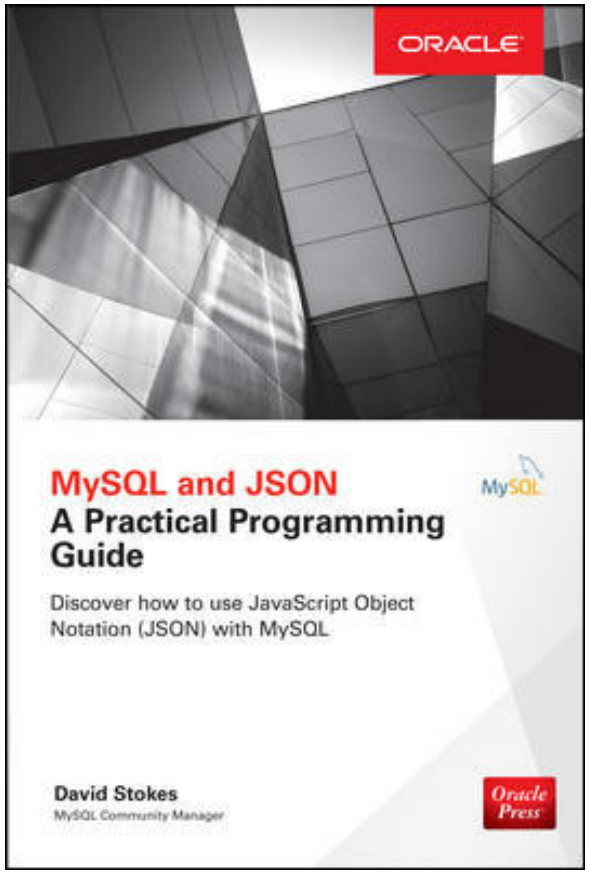
Capacity Planning

- Allocate 2x – 3x storage and buffer pool for JSON data
- Good reason to use JSON only for a subset of your data

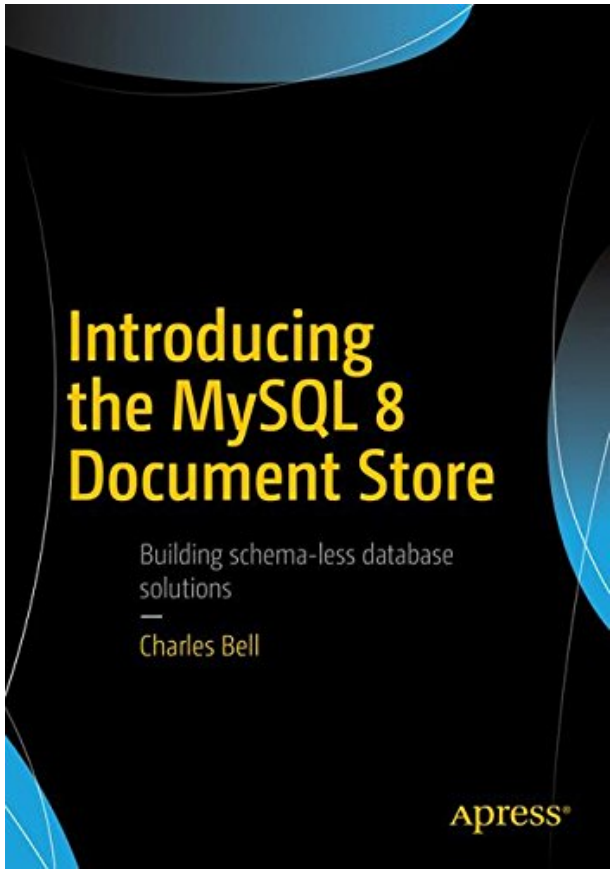
Application Design

- Prefer to encode & decode JSON in your app, not in SQL
  - Move that computation out to edge servers to scale out the load
  - SQL should treat JSON as a "black box," i.e. irreducible strings
- Test performance cost of JSON encoding & decoding functions

See Upcoming Books

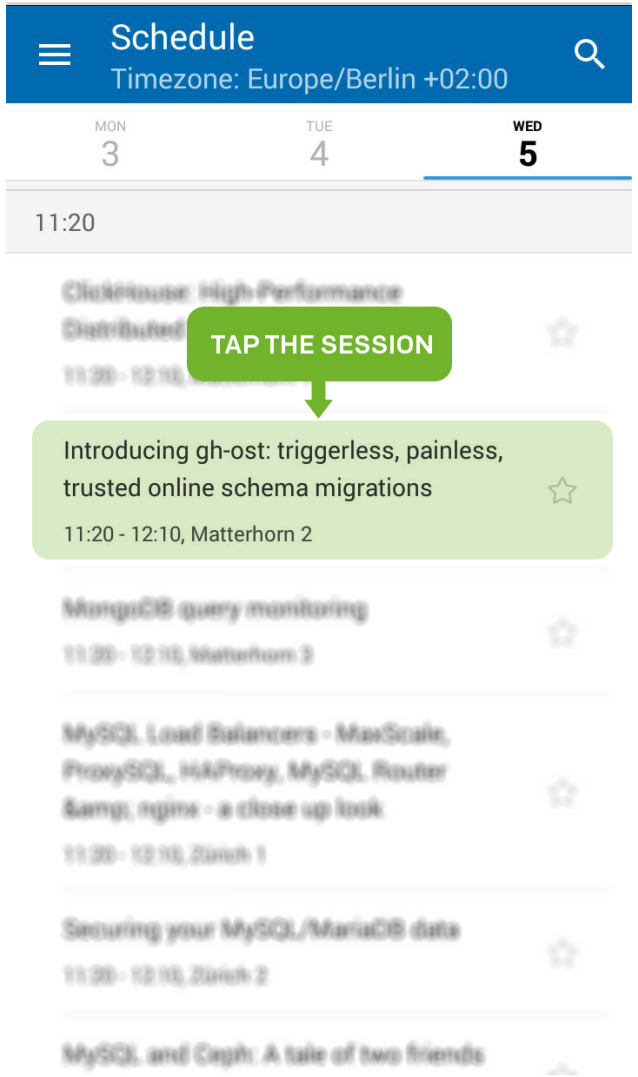


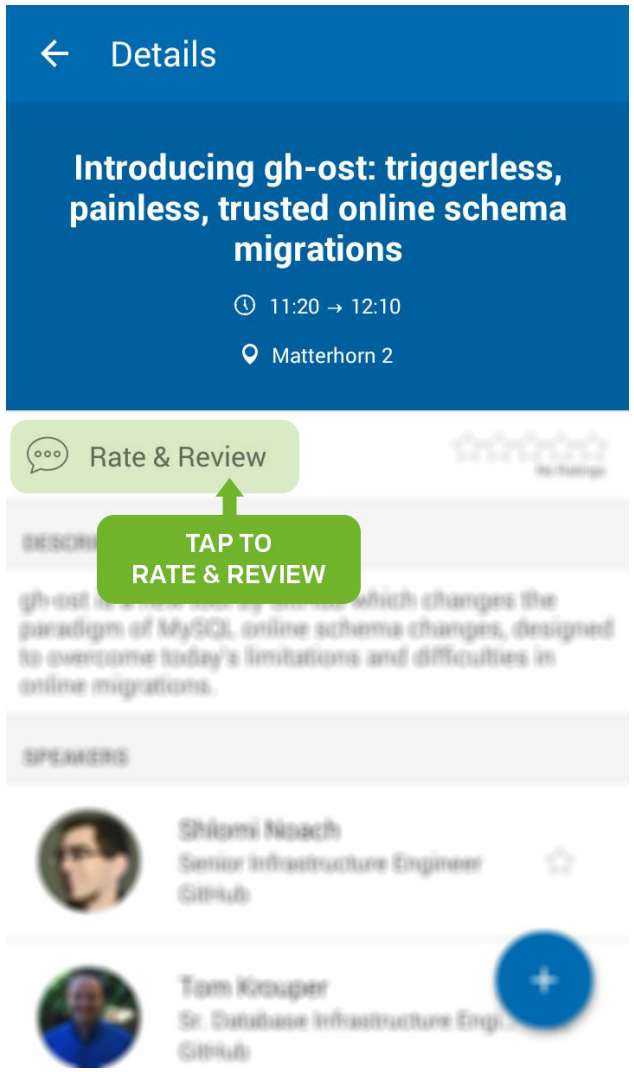
MySQL and JSON: A Practical Programming Guide (2018-06-08) by David Stokes
https://www.mhprofessional.com/mysql-and-json-a-practical-programming-guide

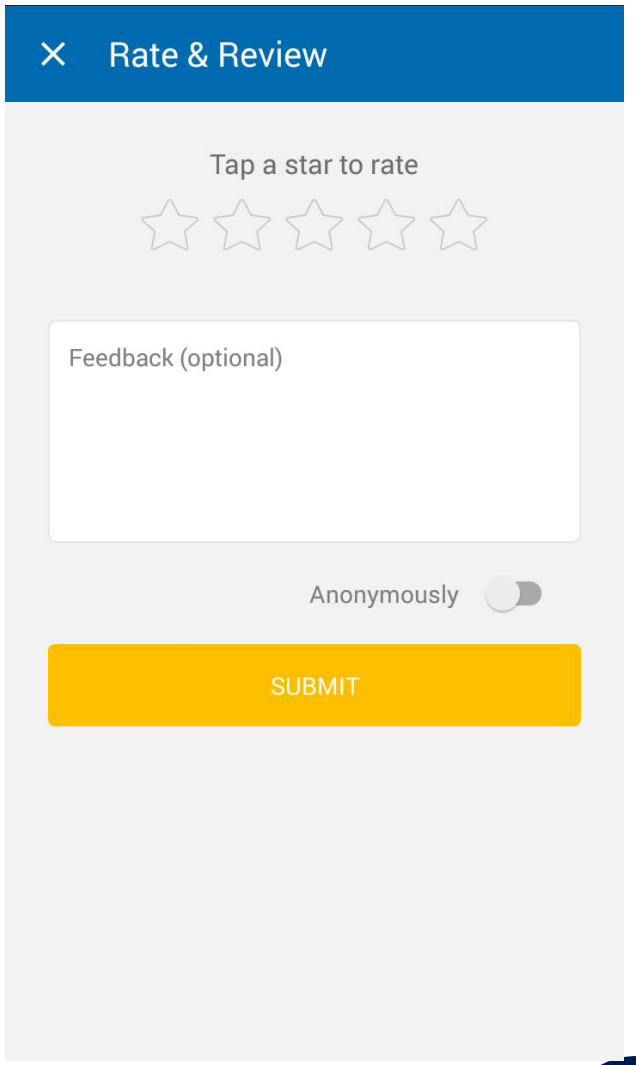


Introducing the MySQL 8 Document Store (2018-07-31) by Charles Bell
https://www.apress.com/us/book/9781484227244

Rate My Session







License and Copyright

Attribution.	**Noncommercial.**	**No Derivative Works.**
You must attribute this work to Bill Karwin.	You may not use this work for commercial purposes.	You may not alter, transform, or build upon this work.