Extensible Data Modeling with MySQL

Bill Karwin
Percona Live MySQL Conference & Expo
“I need to add a new column—but I don’t want ALTER TABLE to lock the application for a long time.”
How MySQL Does ALTER TABLE

1. Lock the table.
2. Make a new, empty the table like the original.
3. Modify the columns of the new empty table.
4. Copy all rows of data from original to new table... *no matter how long it takes.*
5. Swap the old and new tables.
6. Unlock the tables & drop the original.
Extensibility

- How can we add new attributes without the pain of schema changes?
  - Object-oriented modeling
  - Sparse columns
- Especially to support user-defined attributes at runtime or after deployment:
  - Content management systems
  - E-commerce frameworks
  - Games
Solutions

• “Extra Columns”
• Entity-Attribute-Value
• Class Table Inheritance
• Serialized LOB & Inverted Indexes
• Online Schema Changes
• Non-Relational Databases
EXTRA COLUMNS
# Table with Fixed Columns

```sql
CREATE TABLE Title (
    id int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    title text NOT NULL,
    imdb_index varchar(12) DEFAULT NULL,
    kind_id int(11) NOT NULL,
    production_year int(11) DEFAULT NULL,
    imdb_id int(11) DEFAULT NULL,
    phonetic_code varchar(5) DEFAULT NULL,
    episode_of_id int(11) DEFAULT NULL,
    season_nr int(11) DEFAULT NULL,
    episode_nr int(11) DEFAULT NULL,
    series_years varchar(49) DEFAULT NULL,
    title_crc32 int(10) unsigned DEFAULT NULL
);
```
Table with Extra Columns

CREATE TABLE Title (  
id int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,  
title text NOT NULL,  
imdb_index varchar(12) DEFAULT NULL,  
kind_id int(11) NOT NULL,  
production_year int(11) DEFAULT NULL,  
imdb_id int(11) DEFAULT NULL,  
phonetic_code varchar(5) DEFAULT NULL,  
extra_data1 TEXT DEFAULT NULL,  
extra_data2 TEXT DEFAULT NULL,  
extra_data3 TEXT DEFAULT NULL,  
extra_data4 TEXT DEFAULT NULL,  
extra_data5 TEXT DEFAULT NULL,  
extra_data6 TEXT DEFAULT NULL,  
);
Adding a New Attribute

UPDATE Title
SET extra_data3 = 'PG-13'
WHERE id = 207468;

remember which column you used for each new attribute!
Pros and Cons

• Good solution:
  – No ALTER TABLE necessary to use a column for a new attribute—only a project decision is needed.
  – Related to Single Table Inheritance (STI)
    http://martinfowler.com/eaaCatalog/singleTableInheritance.html
Pros and Cons

• Bad solution:
  – If you run out of extra columns, then you’re back to ALTER TABLE.
  – Anyone can put any data in the columns—you can’t assume consistent usage on every row.
  – Columns lack descriptive names or the right data type.
EAV

- Store each attribute in a row instead of a column.

```sql
CREATE TABLE Attributes (  
etity INT NOT NULL,  
attribute VARCHAR(20) NOT NULL,  
value TEXT,  
FOREIGN KEY (entity) REFERENCES Title (id) );
```
Example EAV Data

```sql
SELECT * FROM Attributes;
```

<table>
<thead>
<tr>
<th>entity</th>
<th>attribute</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>207468</td>
<td>title</td>
<td>Goldfinger</td>
</tr>
<tr>
<td>207468</td>
<td>production_year</td>
<td>1964</td>
</tr>
<tr>
<td>207468</td>
<td>rating</td>
<td>7.8</td>
</tr>
<tr>
<td>207468</td>
<td>length</td>
<td>110 min</td>
</tr>
</tbody>
</table>
Adding a New Attribute

• Simply use INSERT with a new attribute name.

```sql
INSERT INTO Attributes (entity, attribute, value)
VALUES (207468, 'budget', '$3,000,000');
```
Query EAV as a Pivot

```sql
SELECT a.entity AS id,
    a.value AS title,
    y.value AS production_year,
    r.value AS rating,
    b.value AS budget
FROM Attributes AS a
JOIN Attributes AS y USING (entity)
JOIN Attributes AS r USING (entity)
JOIN Attributes AS b USING (entity)
WHERE a.attribute = 'title'
    AND y.attribute = 'production_year'
    AND r.attribute = 'rating'
    AND b.attribute = 'budget';
```

| id     | title     | production_year | rating | budget     |
|--------+-----------+-----------------+--------+------------|
| 207468 | Goldfinger | 1964            | 7.8    | $3,000,000 |

another join required for each additional attribute
NEW CUYAMA

Population 562
Ft. above sea level 2150
Established 1951

TOTAL 4663
Sounds Simple Enough, But…

• NOT NULL doesn’t work
• FOREIGN KEY doesn’t work
• UNIQUE KEY doesn’t work
• Data types don’t work
• Searches don’t scale
• Indexes and storage are inefficient
Constraints Don’t Work

CREATE TABLE Attributes (  
    entity INT NOT NULL,  
    attribute VARCHAR(20) NOT NULL,  
    value TEXT NOT NULL,  
    FOREIGN KEY (entity)  
    REFERENCES Title (id)  
    FOREIGN KEY (value)  
    REFERENCES Ratings (rating)  
) ;

constraints apply to all rows, not just rows for a specific attribute type
Data Types Don’t Work

INSERT INTO Attributes (entity, attribute, value) VALUES (207468, 'budget', 'banana');

database cannot prevent application storing nonsensical data
Add Typed Value Columns?

CREATE TABLE Attributes (  
etity INT NOT NULL,  
attribute VARCHAR(20) NOT NULL,  
intval BIGINT,  
floatval FLOAT,  
textval TEXT,  
dateval DATE,  
datetimew DATE,  
FOREIGN KEY (entity) REFERENCES Title (id)  
);  

now my application needs to know which data type column to use for each attribute when inserting and querying
Searches Don’t Scale

• You must hard-code each attribute name,
  – One JOIN per attribute!

• Alternatively, you can query all attributes, but the result is one attribute per row:
  
  ```sql
  SELECT attribute, value
  FROM Attributes
  WHERE entity = 207468;
  – ...and sort it out in your application code.
  ```
Indexes and Storage Are Inefficient

• Many rows, with few distinct attribute names.
  – Poor index cardinality.

• The entity and attribute columns use extra space for every attribute of every “row.”
  – In a conventional table, the entity is the primary key, so it’s stored only once per row.
  – The attribute name is in the table definition, so it’s stored only once per table.
Pros and Cons

• Good solution:
  – No ALTER TABLE needed again—ever!
  – Supports ultimate flexibility, potentially any “row” can have its own distinct set of attributes.
Pros and Cons

• Bad solution:
  – SQL operations become more complex.
  – Lots of application code required to reinvent features that an RDBMS already provides.
  – Doesn’t scale well—pivots required.
CLASS TABLE INHERITANCE
Subtypes

• Titles includes:
  – Films
  – TV shows
  – TV episodes
  – Video games

• Some attributes apply to all, other attributes apply to one subtype or the other.
Title Table

```sql
CREATE TABLE Title (  
id int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
title text NOT NULL,
imdb_index varchar(12) DEFAULT NULL,
kind_id int(11) NOT NULL,
production_year int(11) DEFAULT NULL,
imdb_id int(11) DEFAULT NULL,
phonetic_code varchar(5) DEFAULT NULL,
episode_of_id int(11) DEFAULT NULL,
season_nr int(11) DEFAULT NULL,
episode_nr int(11) DEFAULT NULL,
series_years varchar(49) DEFAULT NULL,
title_crc32 int(10) unsigned DEFAULT NULL
);
```

only for tv shows
Title Table with Subtype Tables

CREATE TABLE Title (  
id int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,  
title text NOT NULL,  
imdb_index varchar(12) DEFAULT NULL,  
kind_id int(11) NOT NULL,  
production_year int(11) DEFAULT NULL,  
imdb_id int(11) DEFAULT NULL,  
phonetic_code varchar(5) DEFAULT NULL,  
title_crc32 int(10) unsigned DEFAULT NULL,  
PRIMARY KEY (id)  
);  

CREATE TABLE Film (  
id int(11) NOT NULL PRIMARY KEY,  
aspect_ratio varchar(20),  
FOREIGN KEY (id) REFERENCES Title(id)  
);  

CREATE TABLE TVShow (  
id int(11) NOT NULL PRIMARY KEY,  
episode_of_id int(11) DEFAULT NULL,  
season_nr int(11) DEFAULT NULL,  
episode_nr int(11) DEFAULT NULL,  
series_years varchar(49) DEFAULT NULL,  
FOREIGN KEY (id) REFERENCES Title(id)  
);
Adding a New Subtype

• Create a new table—without locking existing tables.

  CREATE TABLE VideoGames (  
    id int(11) NOT NULL PRIMARY KEY,  
    platforms varchar(100) NOT NULL,  
    FOREIGN KEY (id)  
    REFERENCES Title(id)  
  );
Pros and Cons

• Good solution:
  – Best to support a finite set of subtypes, which are likely unchanging after creation.
  – Data types and constraints work normally.
  – Easy to create or drop subtype tables.
  – Easy to query attributes common to all subtypes.
  – Subtype tables are shorter, indexes are smaller.
Pros and Cons

• Bad solution:
  – Adding one entry takes two INSERT statements.
  – Querying attributes of subtypes requires a join.
  – Querying all types with subtype attributes requires multiple joins (as many as subtypes).
  – Adding a common attribute locks a large table.
  – Adding an attribute to a populated subtype locks a smaller table.
SERIALIZED LOB
What is Serializing?

• Objects in your applications can be represented in *serialized* form—i.e., convert the object to a scalar string that you can save and load back as an object.
  – Java objects implementing `Serializable` and processed with `writeObject()`
  – PHP variables processed with `serialize()`
  – Python objects processed with `pickle.dump()`
  – Data encoded with XML, JSON, YAML, etc.
What Is a LOB?

• The BLOB or TEXT datatypes can store long sequences of bytes or characters, such as a string.
• You can store the string representing your object into a single BLOB or TEXT column.
  – You don’t need to define SQL columns for each field of your object.
Title Table with Serialized LOB

```sql
CREATE TABLE Title (  id int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
  title text NOT NULL,
  imdb_index varchar(12) DEFAULT NULL,
  kind_id int(11) NOT NULL,
  production_year int(11) DEFAULT NULL,
  imdb_id int(11) DEFAULT NULL,
  phonetic_code varchar(5) DEFAULT NULL,
  title_crc32 int(10) unsigned DEFAULT NULL,
  extra_info TEXT)
);```

holds everything else, plus anything we didn’t think of
Adding a New Attribute

UPDATE Title
SET extra_info =
'{
  "episode_of_id": "1291895",
  "season_nr": "5",
  "episode_nr": "6"
}'
WHERE id = 1292057;

JSON example
Using XML in MySQL

• MySQL has limited support for XML.

```sql
SELECT id, title,
  ExtractValue(extra_info, '/episode_nr')
AS episode_nr
FROM Title
WHERE ExtractValue(extra_info,
  '/episode_of_id') = 1292057;
```

• Forces table-scans, not possible to use indexes.

Dynamic Columns in MariaDB

CREATE TABLE Title (  
id int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,  
title text NOT NULL,  
...  
extra_info BLOB
);

INSERT INTO Title (title, extra_info)  
VALUES ('Trials and Tribble-ations',  
COLUMN_CREATE('episode_of_id', '1291895',  
'episode_nr', '5',  
'season_nr', '6'));

https://kb.askmonty.org/en/dynamic-columns/
Pros and Cons

• Good solution:
  – Store any object and add new custom fields at any time.
  – No need to do ALTER TABLE to add custom fields.
Pros and Cons

• Bad solution:
  – Not indexable.
  – Must return the whole object, not an individual field.
  – Must write the whole object to update a single field.
  – Hard to use a custom field in a `WHERE` clause, `GROUP BY` or `ORDER BY`.
  – No support in the database for data types or constraints, e.g. `NOT NULL`, `UNIQUE`, `FOREIGN KEY`. 
INVERTED INDEXES
Use This with Serialized LOB

• Helps to mitigate some of the weaknesses.
How This Works

• Create a new table for each field of the LOB that you want to address individually:

```sql
CREATE TABLE Title_EpisodeOf (
  episode_of_id INT NOT NULL,
  id INT NOT NULL,
  PRIMARY KEY (episode_of_id, id),
  FOREIGN KEY (id) REFERENCES Title (id)
);
```

here’s where you get the index support
How This Works

• For each LOB containing an “episode_of_id” field, insert a row to the attribute table with its value.

  INSERT INTO Title_EpisodeOf
  VALUES (1291895, 1292057);

• If another title doesn’t have this field, then you don’t create a referencing row.
Query for Recent Users

SELECT u.*
FROM Title_EpisodeOf AS e
JOIN Title AS t USING (id)
WHERE e.episode_of_id = '1291895';

This is a *primary key lookup*. It matches only titles that have such a field, *and* whose value matches the condition.
Pros and Cons

• Good solution:
  – Preserves the advantage of Serialized LOB.
  – Adds support for SQL data types, and UNIQUE and FOREIGN KEY constraints.
  – You can index any custom field—without locking the master table.
Pros and Cons

• Bad solution:
  – Redundant storage.
  – It’s up to you to keep attribute tables in sync manually (or with triggers).
  – Requires JOIN to fetch the master row.
  – You must plan which columns you want to be indexed (but this is true of conventional columns too).
  – Still no support for NOT NULL constraint.
ONLINE SCHEMA CHANGES
pt-online-schema-change

• Performs online, non-blocking ALTER TABLE.
  – Captures concurrent updates to a table while restructuring.
  – Some risks and caveats exist; please read the manual and test carefully.

• Free tool—part of Percona Toolkit.
How MySQL Does ALTER TABLE

1. Lock the table.
2. Make a new, empty the table like the original.
3. Modify the columns of the new empty table.
4. Copy all rows of data from original to new table.
5. Swap the old and new tables.
6. Unlock the tables & drop the original.
How pt-osc Does ALTER TABLE

Lock the table:

1. Make a new, empty the table like the original.
2. Modify the columns of the new empty table.
3. Copy all rows of data from original to new table.
   a. Iterate over the table in chunks, in primary key order.
   b. Use triggers to capture ongoing changes in the original, and apply them to the new table.
4. Swap the tables, then drop the original.

Unlock the tables.
Visualize This (1)

cast_info

after trigger

cast_info new
Visualize This (2)

cast_info → after trigger → cast_info new
Visualize This (3)
Visualize This (4)
Visualize This (5)
Visualize This (6)

cast_info

cast_info old

DROP
Adding a New Attribute

• Design the ALTER TABLE statement, but don’t execute it yet.

```sql
mysql> ALTER TABLE cast_info
    ADD COLUMN source INT NOT NULL;
```

• Equivalent pt-online-schema-change command:

```bash
$ pt-online-schema-change
  h=localhost,D=imdb,t=cast_info
  --alter "ADD COLUMN source INT NOT NULL"
```
$ pt-online-schema-change h=localhost,D=imdb,t=cast_info --alter "ADD COLUMN source INT NOT NULL" --execute

Alterating `imdb`.`cast_info`...
Creating new table...
Created new table imdb._cast_info_new OK.
Altering new table...
Altered `imdb`._cast_info_new OK.
Creating triggers...
Created triggers OK.
Copying approximately 22545051 rows...
Copying `imdb`.`cast_info`: 10% 04:05 remain
Copying `imdb`.`cast_info`: 19% 04:07 remain
Copying `imdb`.`cast_info`: 28% 03:44 remain
Copying `imdb`.`cast_info`: 37% 03:16 remain
Copying `imdb`.`cast_info`: 47% 02:47 remain
Copying `imdb`.`cast_info`: 56% 02:18 remain
Copying `imdb`.`cast_info`: 64% 01:53 remain
Copying `imdb`.`cast_info`: 73% 01:28 remain
Copying `imdb`.`cast_info`: 82% 00:55 remain
Copying `imdb`.`cast_info`: 91% 00:26 remain
Copied rows OK.
Swapping tables...
Swapped original and new tables OK.
Dropping old table...
Dropped old table `imdb`._cast_info_old OK.
Dropping triggers...
Dropped triggers OK.
Successfully altered `imdb`._cast_info.
Self-Adjusting

• Copies rows in “chunks” which the tool sizes dynamically.
• The tool throttles back if it increases load too much or if it causes any replication slaves to lag.
• The tool tries to set its lock timeouts to let applications be more likely to succeed.
Why Shouldn’t I Use This?

- Is your table small enough that `ALTER` is already quick enough?
- Is your change already very quick, for example `DROP KEY` in InnoDB?
- Will `pt-online-schema-change` take too long or increase the load too much?
Pros and Cons

• Good solution:
  – `ALTER TABLE` to add conventional columns without the pain of locking.
Pros and Cons

• Bad solution:
  – Can take up to $4 \times$ more time than ALTER TABLE.
  – Table must have a PRIMARY key.
  – Table must not have triggers.
  – No need if your table is small and ALTER TABLE already runs quickly enough.
  – No need for some ALTER TABLE operations that don’t restructure the table (e.g. dropping indexes, adding comments).
NON-RELATIONAL DATABASES
No Rules to Break

• To be *relational*, a table must have a fixed set of columns on every row.
• No such rule exists in a *non-relational* model; you can store a distinct set of fields per record.
• No schema makes NoSQL more flexible.
Adding a New Attribute

• Document-oriented databases are designed to support defining distinct attributes per document.
• But you lose advantages of relational databases:
  – Data types
  – Constraints
  – Uniform structure of records
SUMMARY
## Summary

<table>
<thead>
<tr>
<th>Solution</th>
<th>Lock-free</th>
<th>Flexible</th>
<th>Select</th>
<th>Filter</th>
<th>Indexed</th>
<th>Data Types</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extra Columns</td>
<td>no*</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes*</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>EAV</td>
<td>yes</td>
<td>yes</td>
<td>yes*</td>
<td>yes</td>
<td>yes*</td>
<td>no*</td>
<td>no</td>
</tr>
<tr>
<td>CTI</td>
<td>no*</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>LOB</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Inverted Index</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>OSC</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>NoSQL</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no*</td>
<td>no</td>
</tr>
</tbody>
</table>

* conditions or exceptions apply.
Senior Industry Experts
In-Person and Online Classes
Custom Onsite Training

http://percona.com/training
“Rate This Session”

SQL Antipatterns: Avoiding the Pitfalls of Database Programming by Bill Karwin

Available in print, epub, mobi, pdf. Delivery options for Kindle or Dropbox.