Chapter 4. # April 2016

Welcome to the April 2016 edition of MongoDB Developer's Notebook (MDB-DN). This month we answer the following question(s);

I get that MongoDB offers a polymorphic schema, and that part of the appeal of MongoDB is the "no schema migration" capability as new application requirements roll in. Still, I come from SQL and really miss SQL style data integrity check constraints. What can I do ?

Excellent question! Short version; we will demonstrate most or all of what you are used to seeing in your SQL environment as it relates to data integrity check constraints.

For fun we Googled "history of polymorphic schema" and found a 6 page paper from 1992 titled, `Schema Evolution in Database Systems - An Annotated Bibliography`*, by John F. Roddick. Source,*

```
https://www.researchgate.net/profile/John_Roddick/publication/215
744444_Schema_evolution_in_database_systems_-_an_annotated_biblio
graphy/links/09e4151352c6d325e2000000.pdf
```

Funny that we were already discussing schema polymorphism in 1992 (1986), and yet so little was delivered in enterprise grade systems until recently. Many of the (links) referenced in the paper above will resolve when they are Googled.

Software versions

The primary MongoDB software component used in this edition of MDB-DN is the MongoDB database server core, currently release 3.2.4. All of the software referenced is available for download at the URL's specified, in either trial or community editions.

All of these solutions were developed and tested on a single tier CentOS 7.0 operating system, running in a VMWare Fusion version 8.1 virtual machine. The MongoDB server software is version 3.2.3, and unless otherwise specified is running on one node, with no shards and no replicas. All software is 64 bit.

4.1 Terms and core concepts

Historically, structured query language (SQL) based database servers have stored data in a two dimensional, row and column format. (Similar to a single, well formatted TAB on a spreadsheet.) Generally, each row is semantically identical to every other row in the same table; the same number and names of columns for each row, and same data type for each column. The data values themselves can differ, but the table structure is rigidly enforced.

The primary issue presents itself when the structure of the table (rows and columns) must change. You have modelled a product catalog (for example, a table for sporting goods, skis, volleyballs, helmets), and now a new group of products must be offered with different attributes (columns); now you also sell warranties to products, which calls for new columns (service expiration date, service deductible).

Many, many years ago the SQL database administrator had to exclusively lock the table in the database (no application reads, no writes, no end user activity of any kind), and rebuild the table with this new structure. This action could take minutes, or days of application downtime. More recently, the SQL table could at least provide versioning; the administrator only needs an exclusive table lock on the table for sub-seconds to update the system catalogs that describe the table structure, and each end user application had to restart, re-initialize any cursors, prepared statements, and related. As each previously existing row with its older defined structure is read or updated, the database server would work to overcome any missing or altered columns.

> **Note:** Here is a Stack Overflow article from 2013 that offers a fair discussion of this challenge when using Microsoft SQL/Server,
>
> http://stackoverflow.com/questions/14463457/allowing-end-users-to -dynamically-add-columns-to-a-table
>
> And here is a 2016 Gartner Newsroom article that discusses application development trends,
>
> http://www.gartner.com/newsroom/id/3079718
>
> Gartner associates agile application development with the need to have a polymorphic schema.

SQL database systems are generally identified as, *schema on write*. As new data rows are written to SQL, everything is parsed and put into its specific bucket (one column in one row, one data type, and one name). Schema on write style systems generally offer higher data integrity, as you basically reject everything that doesn't meet a very specific set of pre-defined expectations.

NoSQL (not only SQL) database systems, like the Hadoop HBase database and others, are identified as *schema on read*. Generally, new data is written without bucketing; without specifying any kind of columns, identifiers, data types, etcetera. Only when you read do you try to bucket specific data into specifically identified and addressable columns.

Why ? Well, imagine you could screen scrape and ask questions of the entire Amazon.com movie catalog. Say you are in Amazon.com sales and marketing and have the responsibility to raise the average customer transaction amount (revenue per sale) by (n)%. Figure 4-1 below displays a single page from the Amazon.com catalog.

Figure 4-1   The Omega Man, an Amazon.com catalog offering, and a cool movie.

A SQL based system could easily model and store this catalog; movie title, available formats (VHS, DVD, BluRay, streaming), price, other. What if you wanted to query page formatting (style sheets) for page views that resulted in an item being added to a cart and purchased, versus those that were not ? Or, if you wanted to query page advertisement density, count and image size, relative to commit to purchase ? (Perhaps advertisement heavy pages reduce the sales closure rate ?) The point to these last few questions is that it would be easier to capture the page content, then magically parse content and then query, versus having to model each question up front, before capturing any content.

Note: **Schema on read, schema on write**

The following Url offered this comparison to schema on write, versus schema on read,

```
http://www.ibmbigdatahub.com/blog/why-schema-read-so-useful
```

Schema on write benefits:

– In traditional data ecosystems, most tools (and people) expect schemas and can get right to work once the schema is described.

– The approach is extremely useful in expressing relationships between data points.

– It can be a very efficient way to store "dense" data.

Schema on write consequences:

– Schemas are typically purpose-built and hard to change.

– Generally loses the raw/atomic data as a source.

– Requires considerable modeling/implementation effort before being able to work with the data.

– If a certain type of data can't be confined in the schema, you can't effectively store or use it (if you can store it at all).

– Unstructured and semi-structured (text, images, documents, other) data sources tend not to be a native fit.

Schema on read benefits:

– Gives you massive flexibility over how the data can be consumed.

– Your raw/atomic data can be stored for reference and consumption years into the future.

– The approach promotes experimentation, since the cost of getting it "wrong" is so low.

– Helps speed the time from data generation to availability.

– Gives you flexibility to store unstructured, semi-structured, and/or loosely or unorganized data.

Schema on read consequences:

– Can be "expensive" in terms of compute resources (then again, these big data engines were built to handle that).

– The data is not self-documenting (i.e., you can't look at a schema to figure out what the data is).

– You have to spend time creating the jobs that create the schema on read.

Is MongoDB schema on write, or schema on read

Well, perhaps Wikipedia.com says it best,

MongoDB is a cross-platform document-oriented database. Classified as a NoSQL database, MongoDB eschews the traditional table-based relational database structure in favor of JSON-like documents with dynamic schemas (MongoDB calls the format BSON), making the integration of data in certain types of applications easier and faster.

Source; `https://en.wikipedia.org/wiki/MongoDB`

MongoDB is schema on write, with collections (like SQL tables), documents (SQL like rows), keys and values (like SQL columns, with semantic integrity). And MongoDB is schema on read in that it supports rich data types, semi-structured and unstructured data, along with the functionality to read from that data.

Most importantly, MongoDB is polymorphic. You can insert new keys and values (columns) without redesigning or restructuring the database. MongoDB is strongly typed, meaning; columns are stored as character data when character, and integer when the value is an integer. A given (column) could start as integer, and then add support for character data as time evolves. MongoDB automatically and implicitly casts data from one data type to another, as needed. Need to run a query comparing character and integer data ? No problem.

What then is the implication to data integrity ?

Excellent question, and that is the question originally raised in this document.

MongoDB has most or all of the SQL style semantic integrity and check constraints. Considering the following, however;

> Suppose you model your `Customer_Orders` collection (table) such that every Order must have an accompanying `Order_Date`. Makes sense, how could you have an Order without an `Order_Date` ?

> Imagine that you become aware that (n)% of a given Item that was delivered to customers is defective, and will need to be replaced. You wish to reserve an amount of inventory on hand, to fulfill replacements orders. You could create an internal Order to reserve this amount of inventory, but a new Order requires an `Order_Date`, which begins the warehouse picking and shipping process. You could just delete documents (rows) from the Inventory table, but a rule exists there that all Inventory must be part of an existing Order before they are removed from inventory (and that Order will need an `Order_Date`).

The point is; you will need some rules, but not too many. On some level your business is a CEP system (complex event processing system, with rules), but is it optimal to put those rules in the database, or the application ?

And consider mobile, Web, and single page applications (SPA)

Another major trend to applications programming is the move to mobile and Web based applications. Wikipedia.com defines a single page application thusly,

> A single-page application (SPA) is a web application or web site that fits on a single web page with the goal of providing a more fluid user experience similar to a desktop application. In a SPA, either all necessary code, HTML, JavaScript, and CSS, is retrieved with a single page load, or the appropriate resources are dynamically loaded and added to the page as necessary, usually in response to user actions. The page does not reload at any point in the process, nor does the page control transfer to another page, although the location hash can be used to provide the perception and navigability of separate logical pages in the application, as can the HTML5 `pushState()` API.

> Source; `https://en.wikipedia.org/wiki/Single-page_application`

In effect, the only traffic between the end user application and application server and database after the initial page load, is data. Data validation rules are best enforced on the client, providing an immediate and optimal end user application experience. Client side data integrity checking or server side; the current trend places more checking on the client side.

> **Note:** One of the most popular MongoDB client side data validation application frameworks when using node.js arrives with Mongoose, and is detailed here.
>
> ```
> http://mongoosejs.com/
> ```
>
> ```
> http://mongoosejs.com/docs/index.html # quick start guide
> ```
>
> You have to register for the free tutorial below,
>
> ```
> https://masteringmean.com/lessons/196-Introduction-to-Mongoose-Mo
> del-Validation-and-Middleware
> ```

4.2 End to end MongoDB document validation example

In this section of this document, we fully implement and detail one example MongoDB data integrity check constraint (MongoDB document validation) end to end. At the end of this example, you should feel comfortable with all MongoDB document validation terms and conditions. The example we use is a given key value being inserted into a collection must equal the integer value 17. No other values should be accepted.

Key introductory points to consider:

– The Url heading all MongoDB on line documentation is,
  `https://docs.mongodb.org/manual/`

– MongoDB titles SQL style data integrity check constraints to be, *document validation*, and the head on line documentation page for this topic is at,
  `https://docs.mongodb.org/manual/core/document-validation/`

  As a topic, document validation is entirely unrelated to the MongoDB collection method titled, validate. Validate is detailed at the following Url,
  `https://docs.mongodb.org/manual/reference/command/validate/`

– MongoDB document validation can not be applied to the following MongoDB special databases and collections:

  • `admin` database

  • `local` database

  • `config` database

  • Any `system.*` collection in any database

– As with most database environments, MongoDB supports 4 primary data manipulation methods/verbs; find (SQL SELECT), insert (SQL INSERT), remove (SQL DELETE), and update (SQL UPDATE).

Document validation is enforced during the execution of the MongoDB command verbs, insert and update. The MongoDB command verb find does not modify data, so no document validation is enacted. And the MongoDB command verb remove, while subject to end user authorization constraints, does not itself modify data, and no document validation is enacted.

– MongoDB document validation constraint can be specified when a collection is created (`db.createCollection()`), or after the collection exists and does or does not contain documents.

Just to be brief, we detail creating document validation using db.createCollection(). To add document validation to an existing MongoDB collection, you use the `db.runCommand( { "collMod" : .. } )` command.

This topic is not expanded upon further here.

– While we detail using document validation when creating a collection, as a topic, document validation could be added to a collection that already contains documents. In this case, we are discussing application of a MongoDB collection modify method, and not insert.

Several MongoDB keywords are in play here:

• `bypassDocumentValidation`, is an access control, and is built into the pre-defined roles titled, `dbAdmin` and `restore`.

This topic is not expanded upon further here.

• `validationLevel`, either `off`, `strict` or `moderate`.

strict applies all document validation rules to all document inserts and updates.

moderate applies all document validation rules to document inserts only. In effect, if a *bad* document existed that you now update, any inherited violations will pass validation.

• `validationAction`, either `warn` or `error`.

error returns an error to the end user application, expecting that the end user application will take corrective action.

warn allows the given data manipulation method, and places an entry in the MongoDB write operation log.

– The MongoDB database method titled, `db.getCollectionInfos()` may be used to gather existing document validation rules in place. Usages include:

· `db.getCollectionInfos()`

Return data for all collections in the current database

· `db.getCollectionInfos( { "name" : " <collection name> " } )`

Return data for the named, <collection name> in the current database.

Example 4-1 offers the first MongoDB document validation example. A code review follows.

Example 4-1   First MongoDB document validation example.

```
db.my_coll.drop()

vc = { "$and" :
   [
   { "k1" : { "$exists" : "true" } },
   { "k1" : { "$eq"      : 17      } }
   ] }

db.createCollection( "my_coll",
   validator = vc, validationAction = "error" )

db.my_coll.insert( { "k1" : 17 } )
# db.my_coll.insert( { "k2" : 30 } )
# db.my_coll.insert( { "k1" : 32 } )
# db.my_coll.insert( { "k1" : "aaa" } )

db.my_coll.drop()

vc = { "$and" :
   [
   { "k1" : { "$eq"      : 17      } }
   ] }

db.createCollection( "my_coll",
   validator = vc, validationAction = "error" )

db.my_coll.insert( { "k1" : 17 } )
# db.my_coll.insert( { "k2" : 30 } )
```

Relative to Example 4-1, the following is offered:

- The db.my_coll.drop() methods are not required. These statements are in place so that we are certain we begin with a collection with no document validation rules in place.

- The lines that begin, vc = assign a value to a JavaScript string variable. This variable is then used in the line that follows, db.createCollection().

  - The first vc = example displays how document validation criteria can be chained using the $and clause.

    $and is a member of logical operators similar to, $or, $not, etcetera.

- The first example has an $exists clause, which in this context is not required; having the (must equal 17) clause is sufficient.

  – All of the db.my_coll.insert() methods succeed.

  – All of the db_my_coll.insert() methods that are commented out, would fail if allowed to execute.

4.3 Complete the following

In last month's edition of this document, *MongoDB Developer's Notebook March 2016 - - MongoDB Connector for Business Intelligence,* we detailed several activities including:

  – March 2016, Section 3.2.2, Download, install and configure a MongoDB database server

  – March 2016, Section 3.2.3, Start the MongoDB database server

  – March 2016, Section 3.2.4, Create a sample database, and collection, and add data

In this document, we will assume that all of the above steps have been completed. Further, we will assume you can or have already entered the MongoDB shell (binary program name, mongo), and are ready to enter MongoDB query language (MQL) commands.

In Section 4.2, "End to end MongoDB document validation example" on page 8 above, we detailed how to create a document validation constraint for (key == value). And we detailed a number of operating convention, rules and scopes.

In this section of this document, we move a little faster, displaying only the relevant MongoDB `db.createCollection()` method, and then `db.collection.insert()` methods that succeed or fail.

4.3.1 SQL style IS NULL, IS NOT NULL, MongoDB $exists

MongoDB does not have a concept of NULL equal to this same named concept in SQL. In SQL the following is true:

  – SQL has triple valued logic; an integer column could be equal to 17, or not equal to 17, or it could be NULL. Only one of these three conditions may be true. The SQL keyword NULL represents an unknown value. It is not zero, or an empty string. It is NULL.

  – If you have 10 rows, 6 with the value of 12 for a given column, and 4 with NULL values, a SQL GROUP BY will return 5 groups; the 6 rows equal to 12 in one group, and one group for each of the 4 unknown values.

MongoDB has a NULL value keyword, but it is a value, not an unknown value. Consider the following Example 4-2. A code review follows.

Example 4-2   MongoDB NULL value and unique index.

```
db.t1.drop()

db.t1.createIndex( { a : 1 }, { unique : true } )

db.t1.insert( { a : null } )   #  this insert succeeds
db.t1.insert( { a : null } )   #  this insert fails
```

Relative to Example 4-2, the following is offered:

– The second insert fails. NULL is a MongoDB keyword, and constant value.

– Unlike SQL, MongoDB does not represent a triple logic unknown value.

– The MongoDB keyword $exists supplies a SQL NULL like capability. If the MongoDB key value is missing, it is NULL. if the key value is present, its value is known.

Consider the following Example 4-3. A code review follows.

Example 4-3   MongoDB is not null ($exists)

```
db.my_coll.drop()

vc = { "$and" :
    [
    { "k1" : { "$exists" : "true" } }
    ] }

db.createCollection( "my_coll",
    validator = vc, validationAction = "error" )

db.my_coll.insert( { "k1" : 17 } )
# db.my_coll.insert( { "k2" : 17 } )
```

Relative to Example 4-3, the following is offered:

– The $exists clause causes a key value to be (not null).

– The second insert, which is commented out, would fail.

– There is a possible capability to measure the length of a given key value, ensure that is less than 1, but we didn't complete this example since it is close to nonsense. In effect, you could prevent the entry of a given key name, which offers little real functionality.

4.3.2 SQL style IN LIST, MongoDB $in

SQL offers a data integrity check constraint where a given column value must appear in a hard coded list. MongoDB offers this functionality.

Generally SQL does not offer a data integrity check constraint where a given column value must appear in the results of an embedded SQL SELECT. MongoDB also does not offer this capability.

Consider the following Example 4-4. A code review follows.

Example 4-4   SQL style IN LIST

```
db.my_coll.drop()

vc = { "$and" :
   [
   { "k1" : { "$in" : [ 1 , 2 ] } }
   ] }
      #
db.createCollection( "my_coll",
   validator = vc, validationAction = "error" )

db.my_coll.insert( { "k1" : 1, "k2" : 2 } )
# db.my_coll.insert( { "k1" : 3, "k2" : 2 } )
# db.my_coll.update( { "k1" : 2 }, { "k2" : 3 } )

db.my_coll.drop()
   #
vc = { "$and" :
   [
   { "k1" : { "$in"  : [ 1 , 2 ] } },
   { "k2" : { "$nin" : [ 5 , 6 ] } }
   ] }
      #
db.createCollection( "my_coll",
   validator = vc, validationAction = "error" )

db.my_coll.insert( { "k1" : 1, "k2" : 8 } )
# db.my_coll.insert( { "k1" : 1, "k2" : 6 } )
# db.my_coll.insert( { "k1" : 3, "k2" : 8 } )
# db.my_coll.insert( { "k1" : 3, "k2" : 6 } )
```

Relative to Example 4-4, the following is offered:

– As before, the MongoDB insert methods succeed, and the commented out insert methods would fail, as expected.

4.3.3 SQL Ship_Date >= Order_Date, MongoDB $where

We stated above that SQL generally does not offer the ability to create a data integrity check constraint comparing a column value to the results of a dynamic list, the results of a SQL sub-select. This is true.

SQL does offer the ability to compare a column value to a single or set of column values from the same data row. Not other data rows, but the same row. MongoDB offers this functionality as a standard MongoDB find method (similar to SQL SELECT), but not in a data integrity check constraint.

Consider the following Example 4-5. A code review follows.

Example 4-5   MongoDB $where

```
db.my_coll.drop()

#  The following block is not supported, and would fail

# vc = { "$and" :
#    [
#    { "$where" : "this.ship_date >= this.order_date" }
#    ] }
# db.createCollection( "my_coll",
#    validator = vc, validationAction = "error" )

db.my_coll.insert( { "order_date" : 18, "ship_date" : 19} )
db.my_coll.insert( { "order_date" : 18, "ship_date" : 10} )

db.my_coll.find( { "$where" : "this.ship_date >= this.order_date" } )

# Only the first document above is returned.
```

Relative to Example 4-5, the following is offered:

- – If this is a hard application requirement, you could:
  - • Enforce this requirement at the client tier.
  - • Execute periodic batch jobs to check data integrity at the server level using the find method with $where clause detailed above.
- – The nearest customer feature request is entered as a Jira with the following Url, `https://jira.mongodb.org/browse/SERVER-2702`
- – This capability is documented at the following Url, `https://docs.mongodb.org/manual/reference/command/collMod/`

  The most relevant clause from above states,

The validator option takes a document that specifies the validation rules or expressions. You can specify the expressions using the same operators as the query operators with the exception of : $geoNear, $near, $nearSphere, $text, and $where.

> **Note:** $lookup is also currently excluded from being a MongoDB clause in a document validation constraint.
>
> $lookup is not a *query operator*, and is instead a MongoDB *aggregation framework stage*. At this time, no aggregation framework stages may be used as document validation constraint clauses.

4.3.4 Off Topic: $lookup example (and also, foreign key constraints)

While it is off topic from MongoDB document validation, we have discussed the MongoDB aggregation framework stage titled, $lookup. An example might be in order.

Consider the following Example 4-6. A code review follows.

Example 4-6   MongoDB aggregation framework stage, $lookup

```
db.my_col1.drop()
db.my_col2.drop()

db.my_col1.insert( { "abbr" : "WI", "name" : "Wisconsin" } )
db.my_col2.insert( { "city" : "Madison", "st" : "WI" } )

db.my_col2.aggregate(
   [
   {
   "$lookup" :
      {
      "from"         : "my_col1",
      "localField"   : "st",
      "foreignField" : "abbr",
      "as"           : "state"
      }
   },
   {
   "$project" :
      {
      "_id"          : 0,
      "city"         : 1,
      "state"        : "$state.name"
      }
```

```
    }
    ] )
```

Relative to Example 4-6, the following is offered:

- The on line documentation page for this capability is located here, `https://docs.mongodb.org/manual/reference/operator/aggregation/lookup/`

- The data that is produced is,

    `[{u'city': u'Madison', u'state': [u'Wisconsin']}]`

    In effect, a document from the current collection, col2, joined with a second (reference) collection titled, col1.

- The $project stage is superfluous, and is used merely to restrict and rename output key values.

> **Note:** While not a data integrity check constraint in itself, this is how you most closely model foreign key constraints in MongoDB, using $lookup.
>
> As a document oriented database, MongoDB would inherently store detail data inside the parent document, much like an object relational database.

4.3.5 Semantic data integrity, MongoDB key value types

Semantic document validation (key value data type checking) is available in MongoDB. While this is a data integrity constraint checking type, in SQL all columns must be rigidly typed. Thus you might not have previously viewed this as an inherit feature to SQL.

Consider the following Example 4-7. A code review follows.

Example 4-7   Semantic document validation in MongoDB

```
db.my_coll.drop()

vc = { "$and" :
    [
    { "k1" : { "$type" : "int" } },
    { "k2" : { "$type" : "string" } }
    ] }
db.createCollection( "my_coll",
    validator = vc, validationAction = "error" )

db.my_coll.insert( { "k1" : 1, "k2" : "aaa" } )
# db.my_coll.insert( { "k1" : "aaa", "k2" : "aaa" } )
```

```
# db.my_coll.insert( { "k1" : 1, "k2" : 11 } )
```

Relative to Example 4-7, the following is offered:

– The on line document page for MongoDB key value types is located here, `https://docs.mongodb.org/manual/reference/operator/query/type/`

There are at least 18 or more tests there that we do not cover in this brief section.

– The first MongoDB insert method succeeds. The remaining MongoDB insert methods fail, as expected.

4.3.6 Complex constraint checking, length, other, MongoDB $regex

MongoDB supports regular expressions inside a document validation clause. This is hugely powerful, and offers nearly limitless constraint validation; validate e-mail addresses (at least from a formatting stand point), validate IP addresses, the list is nearly endless.

Consider the following Example 4-8 written in Python/pymongo. A code review follows.

Example 4-8   Complex constraint checking, MongoDB $regex

```
import pymongo
    #
my_conn = pymongo.MongoClient("mongodb://localhost")
db = my_conn.cc_db


################################################
################################################


db.my_coll.drop()

vc = { "$and" :
   [
   { "k1" : { "$type" : "string" } },
   { "k1" : { "$regex" : "([a-zA-Z]){8,8}" } }
   ] }

db.create_collection( "my_coll",
   validator = vc, validationAction = "error" )

db.my_coll.insert( { "k1" : "dddddddd" } )
# db.my_coll.insert( { "k1" : "dddddd" } )
```

Relative to Example 4-8, the following is offered:

– The on line document page for MongoDB $regex is located here,
  `https://docs.mongodb.org/manual/reference/operator/query/regex/`

  MongoDB uses Perl compatible regular expressions (i.e. "PCRE") version 8.38 with UTF-8 support.

– This example differs from all others in that this example is written in Python/pymongo.

  We did this because the regex value string in Python, `"([a-zA-Z]){8,8}"` is enclosed in parenthesis, not forward slashes as every normal regex example seems to imply.

  The regex value string states,

  • Accept any lowercase or uppercase value in the range A through Z.

  • And have its length be 8 characters long. If you wanted its length to be 3 to 7 characters, the curly brace pair would contain the value, {3-7}.

– The first insert succeeds, and the second fails.

– We didn't test to see if we actually needed the $type clause, and added it only because it supplies a given amount of self documentation.

4.3.7 MongoDB key value defaults

Again, not immediately apparent is SQL's default value clause for newly inserted data rows. Technically this is a data integrity check constraint. In effect, this constraint only applies to new row SQL inserts;

– If you supply SQL a value for a given column, then that is the value you write.

– If you omit the column from the SQL INSERT statement column list, then you write a default value.

MongoDB does support this capability, but not as a document validation clause. Each implicitly created MongoDB collection contains an automatically created and inserted key value titled, `_id`. Also automatically, a unique index is created on this key value.

To have default constraints on non `_id` key values, you must call the MongoDB update method when inserting, and call it with an $upsert parameter.

Consider the following Example 4-9. A code review follows.

Example 4-9   Default key value clause in MongoDB, $upsert, $set, $setOnInsert

```
db.my_coll.drop()

db.my_coll.insert( { "_id" : 10 , "k1" : 5, "k2" : 10 } )

db.my_coll.update( { "_id" : ObjectId() }, { "$set" :
   { "k1" : 10 } , "$setOnInsert" : { "k2" : 20 } }, { upsert : true } )

db.my_coll.update( { "_id" : 10        }, { "$set" :
   { "k1" : 10 } , "$setOnInsert" : { "k2" : 20 } }, { upsert : true } )

db.my_coll.find()

# { "_id" : 10, "k1" : 10, "k2" : 10 }
# { "_id" : ObjectId("56fca4b43de3c3c87e2ee4ae"), "k1" : 10, "k2" : 20 }
```

Relative to Example 4-9, the following is offered:

– The first insert is offered merely to give us a document that we may update later.

– The first update will actually cause an insert (actually, an upsert). This is because the MongoDB (equivalent to a SQL WHERE clause) specifies a new `ObjectID()`, which by definition, can not exist (will not be found to already exist in the collection).

Because this is an insert, both the `$set` and `$setOnInsert` will execute. We never programmatically call to update this document again, and the resultant document is displayed as the last line in Example 4-9.

– The second update does update a single document with the `_id` key value of 10. This is the first and only document we inserted above.

The `$set` will execute, and sets the key value of `k1` from 5 to 10. The `$setOnInsert` will not execute, since this is now an update and not an insert.

4.3.8 MongoDB unique key value constraints

MongoDB does offer unique indexes, thus unique constraints. Every MongoDB collection must have at least one unique constraint that is immutable, similar to ANSI standard SQL. And, you may have more than one unique constraint. We created a unique index above, in Example 4-2.

MongoDB can operate on a single or set of hosts. When multiple hosts are used for data replication (high availability), then each host and the data it contains is essentially the same, and there is no effect to unique indexes.

MongoDB can also operate across multiple hosts to support data partitioning, also called data sharding. When using multiple hosts in this manner, this is commonly referred to as a massively parallel processing architecture, or MPP. For example, you might have two hosts, one containing North America data, and a second host with data from Asia/Pacific. Any unique indexes must lead with the shard key, and forms the partitioning scheme for the given collection. In this case, unique data validation is per host. This is common practice in MPP architectures, and the challenge is not technical, merely one made for performance. On every new document insert, do you really wish to cross the ocean to determine if a record is unique to Asia, or unique to the whole planet ?

4.3.9 SQL stored procedures, triggers, MongoDB server side

Other than to bookmark the topic, we will not expand upon the MongoDB equivalent to SQL stored procedures, or triggers. SQL stored procedures and triggers are not clearly a data integrity check constraint topic anyway.

MongoDB does have a server side programming capability, which is documented here,

> `https://docs.mongodb.org/manual/core/server-side-javascript/`

Another day, more time perhaps.

4.4 In this document, we reviewed or created:

We detailed all steps necessary to create and test server side document validation with MongoDB, a full or near equivalent to SQL data integrity check constraints. Although we must state again, like all things this topic can be abused and approach something less than best practice:

– You do not want to lock your MongoDB database server down to the point that it becomes inflexible, and loses the inherit sweetness of a polymorphic schema supporting system.

– Best practice application programming had and still has the intent to perform most or all of this validation on the client tier, and there are many leading application development frameworks that support this.

However, now you can run server side document validation with MongoDB !

Persons who help this month.

Dave Lutz, Dylan Tong, and Patrick Sheehan.

Additional resources:

Free MongoDB training courses,

```
https://university.mongodb.com/
```

A really good two part tutorial blob post on this topic is available here,

```
https://www.mongodb.com/blog/post/document-validation-part-1-adding-
just-the-right-amount-of-control-over-your-documents
```

```
https://www.mongodb.com/blog/post/document-validation-part-2-putting
-it-all-together-a-tutorial
```