Chapter 6. June 2016

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

> My team and I have been tasked with learning MongoDB, writing queries and getting a new project out of the door in 3 weeks. I know how to write most queries in SQL, and have been doing that for years. Can you detail for me when to use MongoDB find ( ), versus aggregate ( ); pretty much give me a primer on the whole subject area?
>
> *Excellent question! MongoDB find ( ) and aggregate ( ) are very much like SQL SELECT in that you can write your first one in minutes, and then spend days or longer mastering the topic; performance tuning, advanced syntax, other.*
>
> *In prior editions of this document we covered-*
>
> – *March 2016, running SQL directly against MongoDB. Useful for business intelligence (BI) reporting and application development support, but likely not what you'd deliver natively in your new application.*
> – *May 2016, index utilization and query optimizers. The May 2016 document would be the follow on topic to this month's, "how to write queries" discussion.*

For source queries, we pulled the TPC.org TPC-C and TPC-H specifications (we'll write those queries in MongoDB MQL), and we pulled a sample database from one of our favorite SQL database servers, Informix Dynamic Server (IDS). We also produce a number of queries using the MongoDB University course, M-101(P), using the zips data set.

Software versions

The primary MongoDB software component used in this edition of MDB-DN is the MongoDB database server core, currently release 3.2.6. 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 3.2.6, and unless otherwise specified is running on one node, with no shards and no replicas. All software is 64 bit.

6.1 Terms and core concepts

We label this document as a *primer* on MongoDB queries, since there really is no substitute for reading the entire MongoDB (query) documentation. In short, we will state;

- There is the find ( ) method, which fulfills a subset of the clauses to a SQL SELECT; the column list (a projection), the WHERE clause, and ORDER BY.
- There is the aggregate ( ) method, which fulfills all functionality of the SQL SELECT statement, and more.
- And there are a small number of (helper) functions; shorthand to COUNT ( ), DISTINCT ( ), and similar.

The find ( ) method

- Generally allows the SQL equivalent to a WHERE clause and certain amount of key (SQL column) manipulation. find ( ) can return zero, one, or more documents, just like SQL SELECT.

  The (WHERE clause) is passed as the first argument to find ( ), in the form of a JSON document. This first argument is referred to as the *query document*.

  Example 101:

  ```
  db.zips.find( { “state” : “CO” } )
  ```

  Returns all states from the zips collection where the value of the state key is CO. The query document is that data contained within curly braces.

  The second argument to find ( ) is the *projection document*, also a JSON document, and is used to suppress or include the return of keys to the calling function, rename keys, and more.

  Example 102:

  ```
  db.zips.find( { “state” : “CO” }, { “_id” : 0 } )
  ```

  Returns all states from the zips collections where the value of the state key is “CO”, and returns all keys (columns) except the default key titled, “_id”.

- There are about 2 dozen modifiers to find ( ), called *cursor methods*. Examples include:
  - Example 103: count ( )

    ```
    db.zips.find( { "state" : "CO" } ).count( )
    ```

    Returns a count of the result set, versus the results themselves.
  - Example 104: sort ( )

```
db.zips.find( { "state" : "CO" } ).sort( { "pop" : 1 } )
```

Returns the result set in sorted order; in this case, sorted ascending by pop (population).

The documentation page for sort ( ) is located here,

```
https://docs.mongodb.com/manual/reference/method/cursor.sort/
```

- Example 105: limit ( )

```
db.zips.find( { "state" : "CO" } ).sort( { "pop" : -1 }
).limit( 5 )
```

Returns only the first 5 results to the query. Since we changed the sort to be on pop (population) descending (-1), this query returns the documents with the 5 highest populations.

limit ( ) is similar to the SQL statement, SELECT FIRST (n).

- The documentation page for all of the cursor methods is located here,

```
https://docs.mongodb.com/manual/reference/method/js-cursor/
```

– find ( ) offers a subset of functionality to aggregate ( ), and then also SQL SELECT.

find ( ) will either return documents via a collection scan retrieval (sequential scan), or indexed retrieval. The choice between collection scan retrieval or index retrieval is called the *collection access method,* similar to table access method in SQL.

In either collection access method (collection scan, or indexed), the results are pipelined, that is; as soon as documents qualify the selection criteria (pass the criteria specified in the query document), they are passed to the calling function, which is generally a client side program.

The exception to the above may occur when a sort ( ) modifier (cursor method) is present.

Note: If the index used to process the query document can also serve the sort ( ) method, then documents arrive pre-sorted, the same as in SQL. The MongoDB query optimizer will favor this manner of processing since it returns documents in the shortest time, using the least amount of resource. This is also referred to as a *pipelined sort,* (aka, a non-blocking sort).

If there is no index available to support the sort ( ) method, or there is other conflict, then an internal sort package will automatically be invoked to process the query, resulting in a non-pipelined, or blocking sort.

MongoDB limits find ( ) and blocking sorts to 32 megabytes of memory, or the query will fail.

This is actually a feature. We want find ( ) to execute in the low (single digit) millisecond range. We want the more consumptive queries to use aggregate ( ).

sort ( ) and memory limits are documented here,

```
https://docs.mongodb.com/manual/reference/method/cursor.sort/
```

Off topic, but other (system) MongoDB limits are documented here,

```
https://docs.mongodb.com/manual/reference/limits/
```

- In last month's edition of this document (May 2016), we detailed query optimization and index utilization, including producing and reading query plans.

  Example 106:

  ```
  db.zips.find( { "state" : "CO" }, { "_id" : 0 } ).sort( { "pop"
  : -1 } ).limit(5).explain( "executionStats" )
  ```

  Returns the query plan for the find ( ) statement, and not the results themselves. From this output we see two lines out of many,

  ```
  "memUsage" : 590
  ```

  ```
  "memLimit" : 33554432
  ```

  memLimit is the memory available (32 megabytes, 33554432 / 1024 (KB) / 1024 (MB)), and memUsage is the amount of memory consumed. All values are in bytes.

- There is a findOne ( ) method, which is a helper to find ( ), limit (1).

  This topic is not expanded upon further here.

- And there is a pretty ( ) method, which formats the output to a more human readable format.

This topic is not expanded upon further here.

– The documentation page for find ( ) is located here,

```
https://docs.mongodb.com/v3.2/reference/method/db.collection.find/
```

– The documentation pages for find ( ), and more specifically projection, are located here,

```
https://docs.mongodb.com/v3.2/reference/operator/projection/
```

```
https://docs.mongodb.com/v3.0/reference/operator/projection/positional/
```

– Lastly, the logical operators to find ( ), (and, or, not, in, not in, etcetera), are called *projection operators*, and are documented here,

```
https://docs.mongodb.com/v3.0/reference/operator/query/
```

The aggregate ( ) method, when to use

As stated above, the aggregate ( ) method offers a superset of the functionality to find ( ), and the full or larger set of capabilities to SQL SELECT. aggregate ( ) can return zero, one, or more documents, just like SQL SELECT. The question then easily becomes, when should you use aggregate( ) versus find ( ):

– Use aggregate ( ) when you can't use find ( ). For example-

- When you need to calculate aggregate expressions; E.g., count the number of cities in each state or province.
- When you need to lookup (join).
- When you want to out (output) the results of the query to another collection without client side intervention.
- When you need to unwind, (or wind) elements of an array. (These are similar to ETL pivots.)
- Other.

– Use an aggregate if your sort ( ) will ever exceed 32 megabytes.

aggregate ( ) allows 100 megabytes per stage, and also overflow to disk. This detail is documented here,

```
https://docs.mongodb.com/manual/reference/operator/aggregation/group/
```

https://docs.mongodb.com/manual/core/aggregation-pipeline-limits/

Example 107 displays an aggregate query calling for disk overflow support:

```
db.zips.aggregate(
```



```
[
{ "$group" :
   {
   "_id" : "$state",
      //
   "totalPop" : { "$sum" : "$pop" },
   "cityCount" : { "$sum" : 1 }
   }
} ,
{ "$sort" : { "_id" : 1 } }
],
{
"allowDiskUse" : true
} )
```

Note: Example 107 above produces two new, never before existing key values titled, cityCount and totalPop.

totalPop is calculated by the expression "sum (pop)", a total of all of the pop key values for a given group of states.

cityCount is calculated by the expression "sum (1)". sum (1) is the means to count when using aggregate ( ) method. Unlike find ( ), aggregate ( ) does not have a dedicated count ( ) operator.

The aggregate ( ) method, stages

At this point we know heuristically when to use aggregate ( ) over find ( ), but what does aggregate ( ) really do ?

aggregate ( ) is the heavy lifter query method to MongoDB. Similar to Linux pipes (cat file | sed | sort | awk | ..), you can assemble any number of *stages* (one or more) using aggregate ( ) that execute in a linked, sequential fashion. Each stage type can appear multiple times and in any order, allowing for a *programmable query framework*. Each of the stages ingest, perform a specific operation, and then output to the next (subsequent) stage. The list of aggregation framework stages is listed here,

```
https://docs.mongodb.com/v3.2/reference/operator/aggregation-pipelin
e/
```


An overview of a number of aggregation framework stages is detailed here:

- match

  match is the aggregation framework stage where you filter, like an SQL WHERE clause, or like the find ( ) method's query document.

  Example 108: match

```
db.zips.aggregate([
   { "$match" :
      {
      "state" : "CO"
      }
   }
   ])
```

  In this example, we call to return only documents from this named collection that have a key value for state equal to CO.

- sort

  sort is the aggregation framework stage where you call to sort, like the SQL ORDER BY clause, or like the find ( ) method's sort ( ) cursor method.

  Example 109: sort

```
db.zips.aggregate([
   { "$match" :
      {
      "state" : "CO"
      }
   },
   { "$sort" :
      {
      "city"  : 1,
      "pop"   : -1
      }
   }
   ])
```

  In this sort example, we added to the match example detailed above. The point is to detail how aggregation framework stages can be combined.

And, we detailed a sort on multiple keys, some ascending, some descending.

– group

group is the aggregation framework stage where you can call to (group), perform aggregate calculations, and more. group is like the SQL GROUP BY clause. find ( ) has no similar function to aggregate ( ) and group.

Example 110: group

```
db.zips.aggregate([
{ "$group" :
   {
   "_id" : "$state",
   "totalPop" : { "$sum" : "$pop" },
   "zipCount" : { "$sum" : 1 }
   }
} ,
{ "$sort" :
   {
   "_id" : 1
   }
}
] )
```

This example has 2 stages, group, then sort. The sort is not required, and was added only to assist in reading the output. (The states come out in alphabetic sequence; totally optional, totally unrelated to group.)

The source collection has 6 keys (not pictured). These are: _id, city, loc, pop (population), state, and zip.

The _id key in the zips collection is completely unrelated to the first argument to group, which is also titled, _id. The first argument to group, _id, serves as a keyword, and specifies the keys from the input stream that we wish to group on. You may group on one or more keys. In this example, we group by one key, state. The dollar sign serves to identify, *group on the value of state*, not a literal string value, state.

Note: What processing is in place behind the scenes for a group stage; did group call to sort the input data stream (call to sort on state) ?

No. group buckets by key value as it receives the input stream. This makes group a blocking stage. (No worries there; sorts are also by their nature, blocking.)

If an index is in place, will group make use of same ?

The closest documentation on this topic is located here,

```
https://docs.mongodb.com/manual/core/aggregation-pipeline/#aggregation-pipeline-behavior
```

match and sort are mentioned as making use of an existing index. group is not specifically mentioned as making use of an index. It is mentioned that group *makes no assumptions on the state of its input stream*. This implies the group is not currently aware that it is receiving pre-sorted data.

All of the above is relative to release 3.2.6. There are tech talks relative to changes in the product post release 3.2.6 where this behavior might change.

The next 2 expressions, totalPop, and zipCount are aggregation expressions (aggregate calculations):

- totalPop calls to produce a group total (a sum) of pop, thus returning total population by state, our group key.
- zipCount calls to produce a count of documents by group, by state, our group key.

Note: As mentioned above, summing on (1) is a common/general technique used within MongoDB, a means to count.

– limit

limit is the aggregation framework stage to limit output, similar to the SQL FIRST (n) modifier. By coupling limit with sort, you can call to return the first (n) records, or last; you return last by sorting in descending order.

Example 111: limit

```
db.zips.aggregate([
{ "$group" :
   {
   "_id"       : "$state",
```

```
   "totalPop"  : { "$sum" : "$pop" }
   }
} ,
{ "$sort" :
   {
   "totalPop" : -1
   }
},
{ "$limit" : 5
}
] )
```

Because this example sorts by a sum of population descending, then limits 5, we return the 5 highest populations by state.

– out

out is the aggregation framework stage that calls to output (write/insert) the documents produced by this aggregate calculation to a (new) collection. out is similar to the SQL SELECT INTO statement.

The collection will be replaced, if it previously existed. Any indexes in place will be preserved. The output collection can not be sharded, nor capped.

As a termination stage, out must be the last stage in an aggregation flow.

Example 112: out

```
db.zips.aggregate([
{ "$match" :
   {
   "state"  : "CO"
   }
} ,
{ "$out" : "zipsCO"
}
] )
```

– sample

sample is the aggregation framework stage to (sample), and not read the entire contents of the named collection. A number of algorithms are in place that determine what and how documents are scanned. See,

https://docs.mongodb.com/manual/reference/operator/aggregation/sample/#pipe._S_sample

Example 113: sample

```
db.zips.aggregate([
{ "$sample" :
   { "size" : 5 }
}
] )
```

– geoNear

geoNear is the aggregation framework stage that returns an ordered stream of documents based on the proximity to a geospatial point. geoNear incorporates the functionality of match, sort, and limit for geospatial data. The output documents include an additional distance field and can include a location identifier field.

This topic is not expanded upon further here.

– lookup

lookup is the aggregation framework stage to perform left outer joins. An example use case would be given the state/province abbreviation, retrieve the state name, governor, other; basically, data enrichment.

Note: While lookup currently supports a left outer join, a new aggregation pipeline stage titled graphLookup supports recursive join, commonly used in parts explosions, parent-of and child-of type queries, and more. graphLookup is currently discussed for the 3.4 release of MongoDB.

Example 114: lookup

```
db.states.insert( { "abbr" : "WI", "name" : "Wisconsin" } )
db.zips.aggregate( [
{
"$match" :
   { "state" : "WI" }
},
{
```



```
"$lookup" :
   {
   "from"         : "states",
   "localField"   : "state",
   "foreignField" : "abbr",
   "as"           : "state"
   }
},
{
"$project" :
   {
   "_id"        : 0,
   "city"       : 1,
   "state"      : "$state.name"
   }
} ] )
```

The initial insert statement is made simply to give us a second collection with data to lookup into.

The match stage is not required, and is used simply to reduce output.

The project stage is also not required, and is used simply to rename output key, and reduce the number of output keys; purely style.

– indexStats

While the indexStats stage is part of the aggregation framework, in our minds this is more of a tuning and optimization routine, and not a stage used for queries. Perhaps we're wrong.

Example 115: indexStats

```
db.zips.aggregate( [ { "$indexStats" : {} } ] ).pretty( )
```

Not covered thus far: unwind, and project

Besides group (or in conjunction with group), our two favorite stages to the aggregation framework are unwind and project. Rather than cover these stages here, we will cover them below via specific (and highly useful) examples.

If you master the group, unwind, and project stages, and the push and addToSet operators, then there is nothing you can't functionally do when querying with MongoDB.

6.2 Complete the following

In this section of this document, we take the www.tpc.org TPC-C and TPC-H database benchmarks, and write them in MongoDB Query Language (MQL). We also take the demonstration database from IBM/Informix Dynamic Server (IDS), and produce that in MQL. As MongoDB does not shred data on insert in the typical relational database method, we've modelled the data to match a document data model. And, we present only those queries that require new techniques. E.g., after we cover AND and OR query predicates, we do not cover them again.

TPC-C

The TPC-C benchmark contains 27 statements worthy of study; 19 SELECTS, and 9 UPDATES. When you abstract the query shapes (does it matters if you have 2 AND predicates, or 3, or 4 ?), the TPC-C contains 6 distinct queries. Missing from these shapes are an OR query, an IN query, a range query, and others. So, we'll add these throughout this section.

zips.json

The sample data set we use in these examples is the zips.json data set from many of the online MongoDB training courses. Url's include,

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

```
http://media.mongodb.org/zips.json
```

The second Url above offers a download of the `zips.json` data file. A sample schema from `zips.json` is listed below in Example 6-1. All work from this point forward in this document is done in the MongoDB command shell titled, mongo.

Example 6-1 Sample document from the zips collection.

```
> db.zips.findOne( )
{
   "_id" : ObjectId("57042dfb4395c1c26641c1f2"),
   "city" : "ACMAR",
   "zip" : "35004",
   "loc" : {
      "y" : 33.584132,
      "x" : 86.51557
   },
```



```
   "pop" : 6055,
   "state" : "AL"
}
```

6.2.1 TPC-C: multiple predicates, AND and OR

Example 201:

```
db.zips.find( { "state" : "CO", "city" : "BUENA VISTA" } )
```

Example 201 executes a query with multiple additive predicates, similar to two predicates in a SQL WHERE clause combined with an AND operator; documents satisfying the query criteria must be both state = CO AND city = BUENA VSIA.

Example 202:

```
db.zips.find(
  {
    "$or" : [ { "state" : "CO" }, { "city" : "BUENA VISTA" } ]
  }
  )
```

Example 202 executes a query with an OR condition; documents satisfying the query criteria must be either state = CO OR city = BUENA VISTA.

Example 203:

```
db.zips.find(
   {
   "$or" :
      [
      { "state" : "CO" , "city" : "BUENA VISTA" },
      { "pop" : { "$lte" : 4 } }
      ]
   }
   )
```

And example 203 uses both ANDs and ORs; documents satisfying the query criteria must be state = CO AND city = BUENA VISTA, OR pop less than or equal to 4.

There are more ANDs, ORs, and related to be had. Generally titled, operators, this topic is detailed here,

```
https://docs.mongodb.com/manual/reference/operator/query/
```

ANDs and ORs are called *logical query operators*. lte (less than or equal to) is called a *comparison query operator*.

6.2.2 TPC-C: key value modification on read

We did all of the queries above using find ( ) because we could; we did not require any functionality not supported by find ( ). There are a small number of expressions in the TPC-C that return or evaluate a modified key value. E.g.,

```
SELECT col + 5 FROM ..
```

```
UPDATE .. SET col = col + 5
```

This are very uncommon routines to perform server side with MongoDB. One of the three driving design goals behind MongoDB was that it be blindingly fast. Thus, everything we can do on the client is done on the client and not on the server tier.

Note: According to The Definitive Guide to MongoDB - Second Edition,

```
www.amazon.com/David-Hows-Definitive-Guide-MongoDB/dp/B00JW90F1E
```

The three goals behind MongoDB was that it be blindingly fast, massively scalable, and easy to use.

Still, we can perform this routine using MongoDB and an aggregate query with a projection stage. These operators are detailed here,

```
https://docs.mongodb.com/manual/reference/operator/aggregation-arithmetic/
```

Example 204:

```
db.zips.aggregate(
   [
   { "$match" :
      {
      "pop" : 4
      }
   },
```



```
{ "$project" :
   {
   "_id"        : 0,
   "state"      : 1,
   "city"       : 1,
   "pop"        : { "$add" : [ "$pop" , 2 ] }
   }
}
]
)
```

Example 204 executes an aggregate query with two stages; match and project. The match is not required, and was added only for style. Notice we query for documents with a pop (population) value equal to 4, and later output the value of 6, as we add 2 to 4.

The project stage suppresses return of the _id field, purely for style, and calls to return the keys titled city and state. project also changes the value of population, by adding 2 to it.

Above we give the link to aggregate arithmetic operators. There are operators for string, arrays, dates, and more. These operators are detailed here,

```
https://docs.mongodb.com/manual/reference/operator/aggregation/
```

6.2.3 TPC-C: count ( )

Like SQL, MongoDB has operators for count ( ), sum ( ), max ( ), avg ( ), and many, many more. In MongoDB, you can count with find ( ), and with aggregate ( ). The expressions sum ( ), max ( ), avg ( ), etcetera, are all supported via aggregate ( ), and not with find ( ).

Counting with aggregate ( ) makes use of the group stage, and then some single or set of aggregate expressions; in other words, you can do a lot more than count when counting. The count ( ) method to find ( ) is called a collection operator, and is detailed here,

```
https://docs.mongodb.com/manual/reference/method/db.collection.count
/
```

Example 205:

```
db.zips.find( { "state" : "CO" } ).count( )
```

We detailed how to (count) with aggregate ( ) and group, in Example 110 above.

6.2.4 TPC-C: distinct ( )

The TPC-C does not have a SELECT DISTINCT( ) statement. MongoDB does have a distinct ( ) collection operator, detailed here,

```
https://docs.mongodb.com/manual/reference/method/db.collection.distinct/
```

The MongoDB distinct ( ) collection operator differs from SQL SELECT DISTINCT in that it can only (distinct) by one key; SQL SELECT can distinct by one or more columns. To (distinct) by multiple keys with MongoDB, use the MongoDB group stage to the aggregation framework.

Example 206:

```
db.zips.distinct( "city" )
db.zips.distinct( "city" , { "pop" : { "$lt" : 100 } } )
```

In example 206, we have two separate queries. The first query returns the distinct city names from the zips collection. The second query returns distinct city names where the population is less than 100. The second argument to the second query is a *query document*, and can be as complex as any query document we have seen thus far.

Example 207:

```
db.zips.aggregate(
   [
   { "$group" :
      {
      "_id" : { "state" : "$state" , "city" : "$city" }
      }
   }
   ]
   )
```

In example 207, we see for the first time the means to group on multiple keys. We also see for the first time, a group stage without any aggregate calculations. Without aggregate calculations, the only output from group will be the key value.

We are not currently outputting any non-keys because we are using group to perform a distinct; to return only unique key values.

What can you output here ?

You can output group by key columns, and aggregate expressions; (count), sum ( ), max ( ), avg ( ), and many, many others. (count is in parenthesis because there is no count operator when using aggregate ( ); to count with aggregate ( ), we sum (1)).

This is the group stage. group collapses and performs calculations on detail records; that's what group does, in MongoDB and SQL both.Thus, you can not continue to output the individual documents that are contained in that group *without using a technique*. The technique we use is to fold the documents from the group into an array.

Note: We have a cool use case demonstrating this technique below-

If you wished to query and return the city name most, or least common across all states, you would group by city name and count; done, problem solved.

If however, you wished to recover that single or set of state names for the target city, you would have normally lost this detail, having grouped on city (common across states). What we will do below, is preserve that single or set of state names in an array, which will be part of each aggregated city document.

Huh ? There are 4 (count) Buena Vistas, as show below,

```
Example 208:
   db.zips.find( { "city" : "BUENA VISTA" },
      { "_id" : 0, "city" : 1, "state" : 1 } )
   { "city" : "BUENA VISTA", "state" : "CO" }
   { "city" : "BUENA VISTA", "state" : "PA" }
   { "city" : "BUENA VISTA", "state" : "TN" }
   { "city" : "BUENA VISTA", "state" : "VA" }
```

Counting Buena Vista is easy, and we've already seen numerous examples.

When we wish to recover the collapsed detail records, we can (push) or (addToSet) and create a grouped document similar to,

```
   {
   '_id': {'city': 'BUENA VISTA'},
   'countOf': 4,
   {'memberStates': [{'state': 'CO'},
   {'state': 'PA'}, {'state': 'TN'}, {'state': 'VA'}]
   }
```

We cover the technique to complete this task below.

How does SQL complete this same task ? SQL could either write to a temporary table, and then join, or include a nested join (an embedded query or co-related sub-query).

Both of these operations are very costly.

Again, all of the MongoDB collection operators are detailed here,

https://docs.mongodb.com/manual/reference/method/js-collection/

Note: The zips collection offers other unique challenges. There can be 1 zip for a given city, or several zips for a given city. (Large cities like Los Angeles, California have many zips. Small cities like East Troy, Wisconsin have only one.)

When you are counting cities from a zips collections, you must first group and remove all but unique city/state key combinations, otherwise you will count Houston, Texas once for every zip code it has.

6.2.5 TPC-C: more, in, nested documents, and positional querying

From all of the content above, you have examples to recreate each of the queries from the TPC-C and more. Still, there are a few more examples we wish to add.

Query with IN

Example 209: query with IN

```
db.zips.find( { "state" : { "$in" : [ "CO" , "WI" ] } } )
```

Example 209 details use of a query document with an IN expression, similar to an OR expression with many key values.

Nested documents

The documents in the zips collection do not contain any nested documents; it has an array of values for the loc (location) key, but still no nested documents. To detail using nested document notation (dot notation), we will create a nested document and then remove it.

Example 210: nested documents

```
db.zips.find( { "zip" : "81211" }, { "_id" : 0 } )
   {'city': 'BUENA VISTA', 'state': 'CO', 'zip': '81211',
   'pop': 5220, 'loc': [-106.147121, 38.838003]}
db.zips.update( { "zip" : "81211" }, { "$set" : { "thingsToDo.water"
: "raft", "thingsToDo.food" : "Pizza Works" } } )
   WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
db.zips.find( { "thingsToDo.water" : "raft" }, { "_id" : 0 } )
   { "city" : "BUENA VISTA", "loc" : [ -106.147121, 38.838003 ],
   "pop" : 5220, "state" : "CO", "zip" : "81211",
```

```
   "thingsToDo" : { "water" : "raft", "food" : "Pizza Works" } }
db.zips.update( { "zip" : "81211" }, { "$unset" : { "thingsToDo" : ""
} } )
   WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
db.zips.find( { "zip" : "81211" }, { "_id" : 0 } )
   {'city': 'BUENA VISTA', 'state': 'CO', 'zip': '81211',
   'pop': 5220, 'loc': [-106.147121, 38.838003]}
```

In example 210, we have three find ( ) methods, and two update ( ) methods:

- The first find ( ) is to show our before state.
- The first update ( ) adds an embedded document to a single document in the zips collection; previously, no documents in zips had an embedded document.
- The second find ( ) details how to write a query document with *dot notation*, making reference to a key in an embedded document, the whole point of this example. The embedded document is titled, thingsToDo, and it has two keys; water and food.
- The last update ( ) method merely puts our data back to its original state.
- And the last find ( ) method proves that fact.

Positional querying (updating)

In example 210, we queried and updated a key in a nested document using dot notation. In part, example 210 works because each key, embedded or not, is distinctly named. All keys are always distinctly named; that is a rule for keys.

However, when using arrays, a key may have zero, one or more values. A common challenge is how to update a specific value in an array, and not other values in an array. We say update because the challenge is less so when querying.

The zips collection has an array for loc (location), however, we'll add a new array with data that is more easily read.

Example 211: positional querying (updating)

```
db.zips.find( { "zip" : "81211" }, { "_id" : 0 } )
   { "city" : "BUENA VISTA", "loc" : [ -106.147121, 38.838003 ],
   "pop" : 5220, "state" : "CO", "zip" : "81211" }
db.zips.update( { "zip" : "81211" }, { "$set" : { "thingsToDo" : [
"raft", "Pizza Works" ] } } )
```

```
   WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
db.zips.find( { "thingsToDo" : "Pizza Works" }, { "_id" : 0 } )
   { "city" : "BUENA VISTA", "loc" : [ -106.147121, 38.838003 ],
   "pop" : 5220, "state" : "CO", "zip" : "81211", "thingsToDo" : [
   "raft", "Pizza Works" ] }
db.zips.find( { "thingsToDo.0" : "Pizza Works" }, { "_id" : 0 } )
   // No data returned
db.zips.find( { "thingsToDo.1" : "Pizza Works" }, { "_id" : 0 } )
   { "city" : "BUENA VISTA", "loc" : [ -106.147121, 38.838003 ],
   "pop" : 5220, "state" : "CO", "zip" : "81211", "thingsToDo" : [
   "raft", "Pizza Works" ] }
db.zips.update( { "zip" : "81211" }, { "$set" : { "thingsToDo.1" :
"Pizza Works 3000" } } )
   WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
db.zips.find( { "zip" : "81211" }, { "_id" : 0 } )
   { "city" : "BUENA VISTA", "loc" : [ -106.147121, 38.838003 ],
   "pop" : 5220, "state" : "CO", "zip" : "81211", "thingsToDo" : [
   "raft", "Pizza Works 3000" ] }
db.zips.update( { "zip" : "81211" }, { "$unset" : { "thingsToDo" : ""
} } )
   WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
db.zips.find( { "zip" : "81211" }, { "_id" : 0 } )
   { "city" : "BUENA VISTA", "loc" : [ -106.147121, 38.838003 ],
   "pop" : 5220, "state" : "CO", "zip" : "81211" }
```

Example 211 contains 6 find ( ) methods, and 3 update ( ) methods:

- The first find ( ) method is to show our before state.
- The first update adds an array titled, thingsToDo, with two values; raft, and Pizza Works.
- The next find ( ) method displays that a query into the array is, by default non-positional; if any of the array values match our query document, then the document is returned.
- The next two find ( ) methods display how to query a specific element of an array using ordinal notation. Arrays are indexed starting from the value, zero. Thus, thingsToDo.1 (the second array element) equals Pizza Works, and not thingsToDo.0.

- The update ( ) method id really why we are here; updating a value in an array by position is a common task-

  Imagine a movie entry on the Amazon.com Web site with 4 comments; How do you call to update just comment number 2 ?

- The remainder of the methods in this example are used to remove any changes we made to the zips collection.

The following Url offers the document page for positional querying/updating,

```
https://docs.mongodb.com/manual/reference/operator/update/positional
/
```

There is a much larger number of operators and modifiers for just arrays documented here,

```
https://docs.mongodb.com/manual/reference/operator/update-array/#upd
ate-operator-modifiers
```

6.2.6 TPC-H

The TPC.org TPC-C benchmark is very old, and centered largely on online transaction processing (OLTP) style queries. The TPC.org TPC-H benchmark is more modern and more difficult, containing 42 or more test queries, and other routines.

That being stated, there is not that more much content related to query construction techniques that we have to introduce. A lot of the TPC-H queries just have more predicates using techniques we have already covered. By means of example, here is the first query in the TPC-H:

```
select
        l_returnflag,
        l_linestatus,
        sum(l_quantity) as sum_qty,
        sum(l_extendedprice) as sum_base_price,
        sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
        sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as
sum_charge,
        avg(l_quantity) as avg_qty,
        avg(l_extendedprice) as avg_price,
        avg(l_discount) as avg_disc,
        count(*) as count_order
```



```
from
        lineitem
where
        l_shipdate <= date ('1998-12-01') - :1 day
group by
        l_returnflag,
        l_linestatus
order by
        l_returnflag,
        l_linestatus;
```

We've already done each of the following:

- sort (order by)
- aggregate, group (group by)
- query documents (SQL where clause)
- sum ( ) and count ( )
- Modifying key value on read

We haven't done avg ( ), but its the same concept as sum ( ).

In this section of this document then, we cover only the new techniques required to program a TPC-H in MongoDB MQL.

6.2.7 TPC-H: nested select, sub-select, co-related subquery

The TPC-H has a number of nested selects (also known as sub-selects), and then also co-related subqueries. Because MongoDB does not shred data on input (a new customer order is kept as one document in one collection, not 3 to (n) records in 3 or more tables), MongoDB rarely needs to join. MongoDB can lookup (join), but even this operation is not common.

Example 208 above makes reference to a technique wherein MongoDB can perform aggregate calculations and still preserve detail data. This is another means by which MongoDB avoids needing nested selects, and co-related subqueries.

The TPC-H does have a number of case statements (MongoDB calls the case statement a switch statement), so we'll cover that next.

Example 212: switch, case


```
// This example needs MongoDB version 3.3.5 or higher.
db.zips.aggregate(
   [
   { "$match" :
      { "state" : { "$in" : [ "CO" , "WI", "TX", "NV" ] } }
   },
   { "$group" :
      { "_id" : "$state" }
   },
   { "$project" :
      {
      "state"         : 1,
      "stateCoolness" :
         {
         "$switch" :
            {
               "branches" :
               [
               {
               "case" :
                  { "$or" : [
                     { "$eq" : [ "$_id" , "CO" ] },
                     { "$eq" : [ "$_id" , "WI" ] }
                  ] } ,
               "then" : "Very Cool"
               },
               ],
               "default" : "Not Cool"
            }
         }
      }
   }
```

```
    ] )
{ "_id" : "TX", "stateCoolness" : "Not Cool" }
{ "_id" : "WI", "stateCoolness" : "Very Cool" }
{ "_id" : "NV", "stateCoolness" : "Not Cool" }
{ "_id" : "CO", "stateCoolness" : "Very Cool" }
```

Relative to example 212, the following is offered:

- This example has a single aggregate function with three stages; match, group, and project.
- The match and project are purely for style, to reduce the number of documents and keys that needs to be read for testing.
- The project stage is the only stage that matters-
  - We project (return) the state key (the state abbreviation).
  - Then we return a new, calculated key titled, stateCoolness.
  - The case statement is on the _id field, which contains the value from state, due to the work of the group stage above.

    The expression in the case statement must return a boolean: true or false. The eq expression is one such expression titled, logical query operators. These operators are detailed here,

    `https://docs.mongodb.com/v3.0/reference/operator/query-logical/`
  - If the _id key equals CO or WI, then we return "Very Cool". All other _id key values return "Not Cool".

There is a simpler form of switch titled cond, which has been in MongoDB since version 2.6. The cond project stage expression is detailed here,

`https://docs.mongodb.com/manual/reference/operator/aggregation/cond/`

6.2.8 IBM/Informix Dynamic Server (IDS)

We include Informix IDS purely for sentimental reasons. A review of the sample database from this relational database server reminded us to cover:

- Regex, or wild card string queries.
- A match stage after any group stage. (Equal to a SQL HAVING clause.)

Regex query example

In the zips collection, the zip key is stored as a character string. While Regex works on many key value types, it is perhaps best suited for delivering complex string evaluations.

Example 213: Regex

```
db.zips.find( { "zip" : { "$in" : [ /^53/, /^80/ ] } } )
```

Example 213 returns zip documents in the range 53000 to 53999 and 80000 to 80999.

Example 214: match after group

```
db.zips.aggregate(
   [
   { "$group" :
      { "_id" : "$state" }
   },
   { "$match" :
      { "_id" : { "$in" : [ "CO" , "WI", "TX", "NV" ] } }
   },
   ] )
```

Perhaps the biggest lesson in example 214 is that the state key gets renamed to _id as the result of the group stage. If this annoys you, you can rename the _id field after the group with a project stage.

6.2.9 Final example: push and addToSet (wind), and unwind

This section of this document starts our final set of examples.

In reality, this whole document started with the exercise to query and determine which city name in the United States was most common across states. As stated above, the zips collection has a primary key of zip (zipcode). Thus, a large city like Los Angeles, CA, or Houston TX has many zip documents. Before you can count cities, you have to remove the duplicate city names per each state.

Note: This is why we like this example so much.

To deliver the necessary data, you must run two group stages; one to remove duplicate cities per state, and a second to count unique city names across states.

Having multiple (and different) group by clauses in SQL is just not possible.

Example 215: Most common city in US

```
db.zips.aggregate(
   [
      {
      "$group" :
         {
         "_id"     :
            {
            "city"  : "$city" ,
            "state" : "$state"
            }
         }
      } ,
      {
      "$group" :
         {
         "_id"     : "$_id.city",
         "countOf" : { "$sum" : 1 }
         }
      } ,
      {
      "$sort" :
         {
        "countOf" : -1
         }
```

```
        },
        { "$limit" : 10 }
    ] )
{countOf': 24, _id': u'CLINTON'}
{countOf': 24, _id': u'FRANKLIN'}
{countOf': 23, _id': u'MADISON'}
{countOf': 22, _id': u'GREENVILLE'}
{countOf': 22, _id': u'ARLINGTON'}
{countOf': 21, _id': u'SALEM'}
{countOf': 21, _id': u'CHESTER'}
{countOf': 20, _id': u'SPRINGFIELD'}
{countOf': 19, _id': u'PRINCETON'}
{countOf': 19, _id': u'TROY'}
```

Relative to example 215, the following is offered:

- There isn't anything terribly new in example 215; a group, a group, a sort, and a limit.
- The first group stage is used to collapse multiple zip codes found to exist for one city/state combination. We do this because we wish to count by city/state, and not by zipcode. Without this stage, a city like Houston, Texas, with numerous zip codes, would be counted many times in error.
- The second group stage is our money group. This group groups by cities across states, and then counts.
- The sort stage is used to sort, descending, returning the highest counts of cities across states first.
- And the limit stage is used ti limit output to the first 10 cities.

So again, there isn't anything terribly new in example 215. What if we wanted the data from example 215, and also the member states from which the most common cities are found ? Who are all of these states with a city named Clinton ?

The challenge is the group does not want to preserve detail records; not in MongoDB, and not in SQL. You can output one document per group, no more. To preserve detail (in effect, output multiple documents), we stuff the detail we wish to preserve into an array, which is then attached to the single document output per group. Its easier than it may initially sound.

Example 216: addToSet (push)

```
db.zips.aggregate(
   [
      {
      "$group" :
         {
         "_id"       :
            {
            "city"   : "$city" ,
            "state"  : "$state"
            }
         }
      } ,
      {
      "$group" :
         {
         "_id"       : "$_id.city",
         "countOf"   : { "$sum" : 1 },
         "stateArr"  : { "$addToSet" : "$_id.state" }
         }
      } ,
      {
      "$sort" :
         {
        "countOf" : -1
         }
      },
      {
      "$limit" : 10
      }
   ] )
```

```
{ "_id" : "CLINTON", "countOf" : 24, "stateArr" : [ "MS", "IA", "SC",
"NC", "LA", "ME", "AR", "OK", "OH", "PA", "WI", "TN", "MN", "WA",
"MI", "IL", "NY", "IN", "CT", "NJ", "MT", "KY", "MD", "MA" ] }
```

Relative to example 216, the following is offered:

- Example 216 builds from example 215. We only added one line, in the project stage, beginning with the text, stateArr.
- stateArr is a calculated output key. stateArr is populated via an addToSet *array update modifier*. The document page for addToSet is located here,

  `https://docs.mongodb.com/manual/reference/operator/update/addToSet/`

  addToSet will insert an element into an array. Our array is named stateArr. addToSet is very similar to another array update modifier titled, push. All of the array update modifiers are detailed here,

  `https://docs.mongodb.com/manual/reference/operator/update-array/`

  The only difference between addToSet and push is that addToSet will not add duplicate values to the array. We don't need this feature, and could use push, but felt any potential extra overhead was worth the self-documenting nature of using a routine that prevents duplicates.
- The last line in example 216 displays the first line of sample output.

Example 217: countOf

```
db.zips.aggregate(
   [
      {
      "$group" :
         {
         "_id"       :
            {
            "city"   : "$city" ,
            "state"  : "$state"
            }
         }
      } ,
      {
```

```
    "$group" :
       {
       "_id"       : "$_id.city",
       "stateArr"  : { "$addToSet" : "$_id.state" }
       }
    } ,
    {
    "$project" :
       {
       "_id"       : "$_id",
       "countOf"   :
          {
          "$size"  : "$stateArr"
          },
       "stateArr"  : "$stateArr"
       }
    },
    {
    "$sort" :
       {
      "countOf" : -1
       }
    },
    {
    "$limit" : 10
    }
  ] )
```

Relative to example 217, the following is offered:

- Example 217 offers a slight modification to example 216, possibly for efficiency.

– Where example 216 counts the number of states per city, example 217 does not. Example 217 calculates this same number by the number of elements in the array.

  This might be more efficient since we calculate this count later in the pipeline. Is someone modifies this code later, perhaps inserting a match after the last group and before output, we could save processing.

unwind

So far in this section of this document we used addToSet to build an array from detail records that would normally have been discarded during the group stage of the aggregate collection method. This is a very cool technique to have if for no other reason, it helps us solve queries that SQL doesn't solve, or at least, doesn't solve easily.

The opposite of addToSet is unwind. Where addToSet builds an array, unwind (unbuilds) an array, outputting a new document from each element of the array. A document with an array with 7 elements, outputs 7 documents post unwind. unwind is documented here,

```
https://docs.mongodb.com/manual/reference/operator/aggregation/unwind/
```

unwind would be used anytime you wish to unwind an array. The document object model used by MongoDB normally gives us many arrays we may wish to unwind. Since we have been using one collection, zips, with its one array of boring loc data, we will use unwind now to unwind that data we previously built throughout this section. We will use unwind to unwind our newly created stateArr array.

Example 218: unwind

```
db.zips.aggregate(
   [
      {
      "$group" :
         {
         "_id"       :
            {
            "city"   : "$city" ,
            "state"  : "$state"
            }
```



```
        }
      } ,
      {
      "$group" :
        {
        "_id"       : "$_id.city",
        "countOf"   : { "$sum" : 1 },
        "stateArr"  : { "$addToSet" : "$_id.state" }
        }
      } ,
      {
      "$sort" :
        {
       "countOf" : -1
        }
      },
      {
      "$limit" : 10
      },
      {
      "$unwind" : "$stateArr"
      }
   ] )
{'stateArr': 'MS', 'countOf': 24, '_id': 'CLINTON'}
{'stateArr': 'IA', 'countOf': 24, '_id': 'CLINTON'}
{'stateArr': 'SC', 'countOf': 24, '_id': 'CLINTON'}
```

Relative to example 218, the following is offered:

- From example 218, you see we've added one new stage to the aggregate cursor method titled, unwind.
- unwind takes one argument, the name of the array to unwind.
- Sample data is output at the end of the flow.

6.2.10 Different example, same techniques

So far we have been focused on solving the query: which city name is most common across states. Just to offer a different query (allow us to be certain we are getting this material), we now change the query: which state has the largest number of unique city names (city names found in this state but not found in other states) ?

Example 219: db.zips.aggregate(

```
db.zips.aggregate(
   [
      {
      "$group" :
         {
         "_id"     :
            {
            "city"  : "$city" ,
            "state" : "$state"
            }
         }
      } ,
      {
      "$group" :
         {
         "_id"     : "$_id.city",
         "countOf" : { "$sum" : 1 } ,
        "memberStates" : { "$push" : { "stateName" : "$_id.state" } }
         }
      } ,
      {
      "$match" :
         {
         "countOf" : 1
         }
```



```
        },
        {
        "$group" :
           {
           "_id"     : "$memberStates.stateName",
           "countOf" : { "$sum" : 1 }
           }
        },
        {
        "$sort" :
           {
           "countOf" : -1
           }
        }
     ] )
{ "_id" : [ "PA" ], "countOf" : 856 }
{ "_id" : [ "NY" ], "countOf" : 785 }
{ "_id" : [ "CA" ], "countOf" : 730 }
...
{ "_id" : [ "RI" ], "countOf" : 24 }
{ "_id" : [ "DE" ], "countOf" : 15 }
{ "_id" : [ "DC" ], "countOf" : 1 }
```

Relative to example 219, the following is offered:

- Example 219 has 3 group stages, a match stage, and a sort stage.
- The first 2 group stages are the same we have seen throughout this section;
  - group on city/state to remove the multiple zip codes for a given large city.
  - Then our real group; group by city, count and collect the state names.
- The match says, I only want cities found in 1 state.

  Since the city is found in only one state, there is only one state in the stateArr array.

- The third group says, now group by state and count.

  Since the data stream contained only cities found in one state, and the output from this stage groups by state, this stage outputs the count of all cities in that state not found in other states.
- And we sort descending.
- Looking at the data, Pennsylvania has the most number of unique city names. Delaware and the District of Columbia have the fewest. (Delaware is kind of small, and DC is not really a state, just kind of one large city.)

6.2.11 Top 5, last 5

The next two queries detail how to produce a single, grand total result from aggregate ( ), and how to produce a first (n) last (n) type result.

Example 301: produce a single, grand total result

```
db.zips.aggregate(
   [
      {
      "$group" :
         {
         "_id"           :
            {
            "_id2"       : { "$literal" : 1 }
            },
         "grandTotal"    : { "$sum" : "$pop" }
         }
      }
   ] )
```

```
{ "_id" : { "_id2" : 1 }, "grandTotal" : 248709873 }
```

Relative to example 301, the following is offered:

- We generate a value for the group by key titled, _id._id2, equal to the numeric literal 1.
- And we calculate a sum.

Many years ago SQL added a nice, SELECT FIRST 5 LAST 5 .., (where 5 is a numeric constant of your choosing) style functionality. Here is how you perform this task in MongoDB.

Example 302: first (n) last (n)

```
db.zips.aggregate(
   [
      {
      "$group"                    :
         {
         "_id"                    :
            {
            "state"               : "$state"
            },
         "pop"                    : { "$sum" : "$pop" }
         }
      },
      {
      "$project"                  :
         {
         "_id"                    : 0,
         "state"                  : "$_id.state",
         "pop"                    : 1,
         "nextGroupId"            :
            {
            "$literal"            : 1
            }
         }
      },
      {
      "$sort"                     :
         {
         "pop"                    : -1
```

```
      }
   },
   {
   "$group"                    :
      {
      "_id"                    :
         {
          "id2"                : "$nextGroupId"
         },
      "stateArr"               :
         {
         "$push"               :
            {
            "state"            : "$state",
            "pop"              : "$pop"
            }
         }
      }
   },
{
   "$project"                  :
      {
      "_id"                    : 0,
      "stateArrTop"            :
         {
         "$slice" : [ "$stateArr"  , 5 ]
         },
      "stateArrBottom"         :
         {
         "$slice" : [ "$stateArr" , -5 ]
         },
      }
```

```
        },
        {
        "$project"                :
           {
           "allStates"            :
              {
              "$setUnion"         :
              [
              "$stateArrTop",
              "$stateArrBottom"
              ]
              }
           }
        },
        {
        "$unwind"                 : "$allStates"
        },
        {
        "$project"                :
           {
           "state"                : "$allStates.state",
           "pop"                  : "$allStates.pop",
           }
        }
     ] )
{'state': 'AK', 'pop': 550043}
{'state': 'DC', 'pop': 606900}
{'state': 'WY','pop': 453588}
{'state': 'ND', 'pop': 638800}
{'state': 'VT', 'pop': 562758}
{'state': 'PA', 'pop': 11881643}
{'state': 'FL', 'pop': 12937926}
```

```
{'state': 'TX', 'pop': 16986510}
{'state': 'NY', 'pop': 17990455}
{'state': 'CA', 'pop': 29760021}
```

Relative to example 302, the following is offered:

- The first group stage calculates pop (population) totals by state.

  We would be done processing right here, if we did not wish to output only the first (n) and last (n) documents.

- The first project stage calls to suppress or output given keys, and adds a new key that we group on below; a numeric constant equal to 1.

  Why have this numeric constant, Why group on 1 ?

  We are going to take all of the documents that we produce and put them into one array for top (first n) populations, and one array for bottom (last n) populations. Grouping is how we can group documents.

- The sort stage sorts what were random sequence documents.

- The second group groups all state records and places them into an array titled, stateArr. These array records arrive sorted, as the result of the sort stage above.

  Just to be verbose, notice we have created an array of documents. By creating a document with state and pop (population), we can keep these two keys associated with one another.

- The second project stage copies the stateArr array into two arrays titled, stateArrTop and stateArrBottom. The slice *array update modifier* calls to populate these two new arrays with the first 5 and last 5 values.

  The slice array update modifier is detailed here,

  `https://docs.mongodb.com/manual/reference/operator/projection/slice/`

Note: slice works in both the aggregate collection method, and on the update ( ) method.

The result is you can create *capped arrays* in both collections (tables), and while processing aggregate calculations.

Very cool.

An article describing using slice with update ( ) is located here,

`http://blog.mongodb.org/post/58782996153/push-to-sorted-array`

- The next project stage merges these two array, top and bottom, using the setUnion array update modifier.

  The setUnion array update modifier is detailed here,

  ```
  https://docs.mongodb.com/manual/reference/operator/aggregation
  /setUnion/#exp._S_setUnion
  ```

- The unwind stage pivots our data from many array elements in one document, to several documents.
- And the last project stage simply cleans up our keys names, purely style.

6.3 In this document, we reviewed or created:

We provided a primer on the topic of writing queries with MongoDB.

- All of the basic AND, OR, and IN type stuff.
- Differences between find ( ) and aggregate ( ).
- aggregate ( ) with a number of array techniques, including how to output detail after a group stage.
- How to complete a FIRST (n)LAST (n) query.

Persons who help this month.

Dave Lutz, and Lori Still.

Additional resources:

Free MongoDB training courses,

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

This document is located here,

```
https://github.com/farrell0/MongoDB-Developers-Notebook
```

