Application Modernization

Migrating Stored Procedures

MAY 2023

Introduction

"To gain greater flexibility and a powerful competitive edge when exploring data, companies are evolving to "schema-light" approaches, using denormalized data models, which have fewer physical tables, to organize data for maximum performance. This approach offers a host of benefits: agile data exploration, greater flexibility in storing structured and unstructured data, and reduced complexity, as data leaders no longer need to introduce additional abstraction layers, such as multiple "joins" between highly normalized tables."

- "How to build a data architecture to drive innovation - today and tomorrow", McKinsey

Every organization seeking transformation must enhance customer experiences, optimize internal operations, and ignite innovation across the board. This pressing need is driven not only by the ever-changing expectations of our customers but also by the dynamic landscape of data - its source, variety, volume, and velocity.

Legacy software, once built as monoliths, relied on relational systems best suited for row and table data. Back then, modeling general ledgers for a company was the primary goal. However, modern applications demand a complete reimagining of this model, breaking free from outdated relational schemas and embracing greater flexibility, streamlined management, and enhanced performance. Today's applications, accessible 24x7, are used by an array of devices, including desktops, laptops, IoT sensors, and handheld devices. In this always-on era, applications can't afford downtime — planned or unplanned. With social media's proliferation, handling diverse data types has become essential, necessitating agile deployment strategies across distributed landscapes.

Modernizing software applications requires a holistic approach, accounting for innovations in database technology, server platforms, data validation and ingestion pipelines, security, DevOps roles, and much more. Database modernization, a multi-phase process, encompasses refactoring stored procedures and triggers, business logic, platform modernization, partition and schema redesign, and performance tuning.

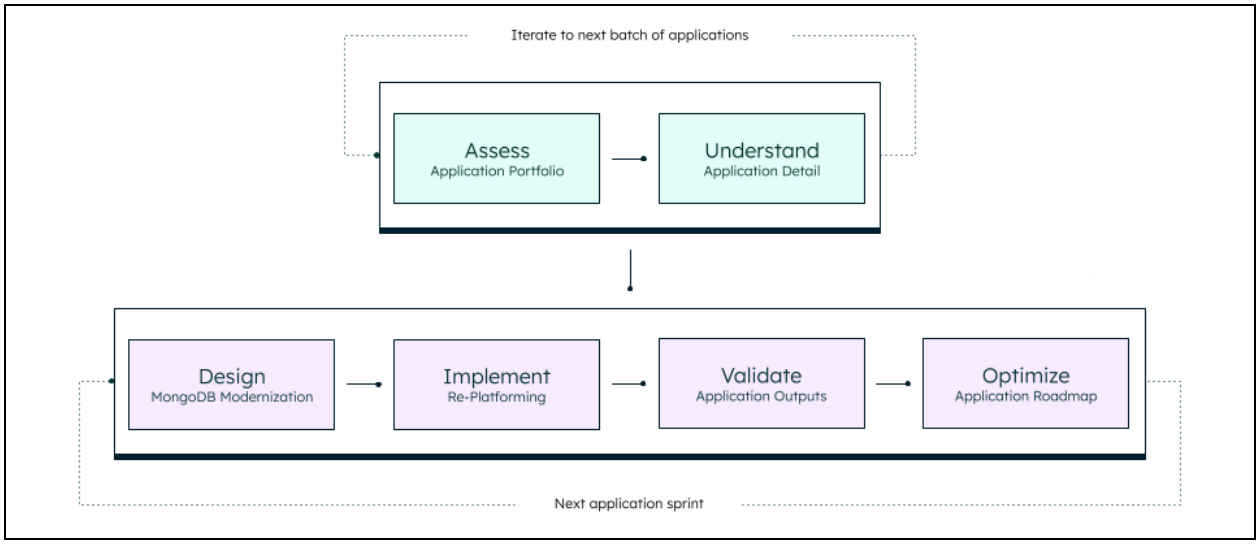


Figure 1: MongoDB App Modernization Factory accelerates and de-risks RDBMS Migration

To assist in this transformative journey, MongoDB offers comprehensive guidance on migrating through the App Modernization Factory. In this whitepaper, we will focus on the design stage of modernization; specifically on migrating stored procedure code from SQL-based relational schemas to aggregation pipelines on MongoDB.

Stored Procedures: Historical Perspective

Stored procedures have been essential to relational databases since their inception. They were introduced as a means to store and execute complex database operations, containing pre-compiled SQL statements that could be invoked on demand. Although they offered some benefits in the past, stored procedures now present several limitations compared to modern, agile development practices.

Historically, stored procedures emerged during the era of early client-server architectures in the 1970s and 1980s. Back then, business logic was stored in the database server rather than the client. When three-tier architectures became popular in the 1990s and 2000s, stored procedures continued to be part of the database layer.

What is a stored Procedure?

A stored procedure is a reusable SQL-based function registered to a relational database for executing complex query operations. Commonly used in RDBMS like Oracle, SQL Server, and DB2, stored procedures facilitate complex data manipulations across multiple tables.
However, modern databases like MongoDB utilize more advanced techniques, such as aggregation pipelines, that leverage the flexible document model and schema. This flexibility makes aggregation pipelines efficient and easy to understand, offering a superior alternative to stored procedures.

Stored procedures were initially perceived as fast, efficient, and secure. However, over time, their drawbacks became increasingly apparent. Today, stored procedures are considered less flexible and harder to maintain than their modern counterparts. They are also less portable and can be challenging to debug, test, and version. Furthermore, they blur the lines between developer and DBA roles regarding access rights and responsibilities.

While Stored Procedures may have had some advantages when initially proposed, they now simply offer large trade-offs for limited benefits in return.

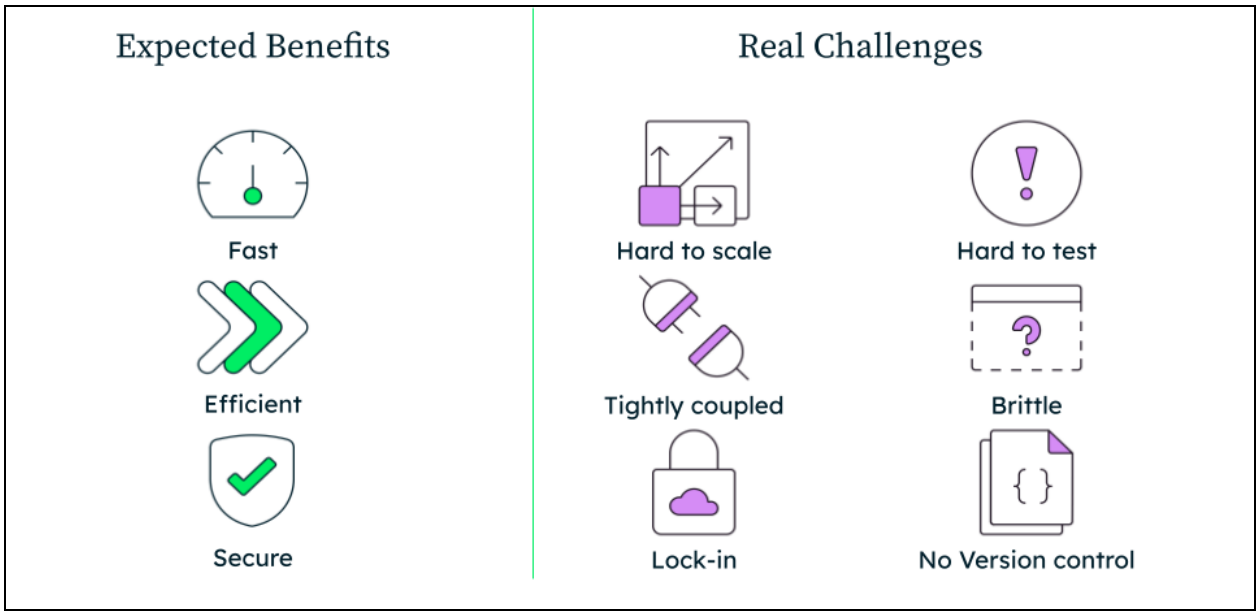


Figure 2: Benefits and challenges of using stored procedures

Fast - Stored Procedure calls can be quick due to the fact they are compiled once and stored in executable form. While they may have been performant at one time, that is never the case when

the database itself is under load. Also, it's hard to generate an optimal execution plan because not all arguments are supplied at compile time. So in terms of performance, results can be very mixed.

Efficient - While code reuse is a good thing, Stored Procedures can be hard to maintain, debug, test, and version. Neither are they very portable. Nowadays, they can often be the cause of the slowdown of other in-flight database operations. They can also introduce risk in terms of breaking changes and developers may shy away from changing them. This can lead to technical debt in many cases, and stifle innovation.

Secure - Nowadays they just blur the lines between developer and DBA roles regarding access rights and responsibilities. Business logic in modern implementations needs to reside on the middleware and run as an API. It is by far easier to deploy and run different versions of a web service than two versions of a set of stored procedures.

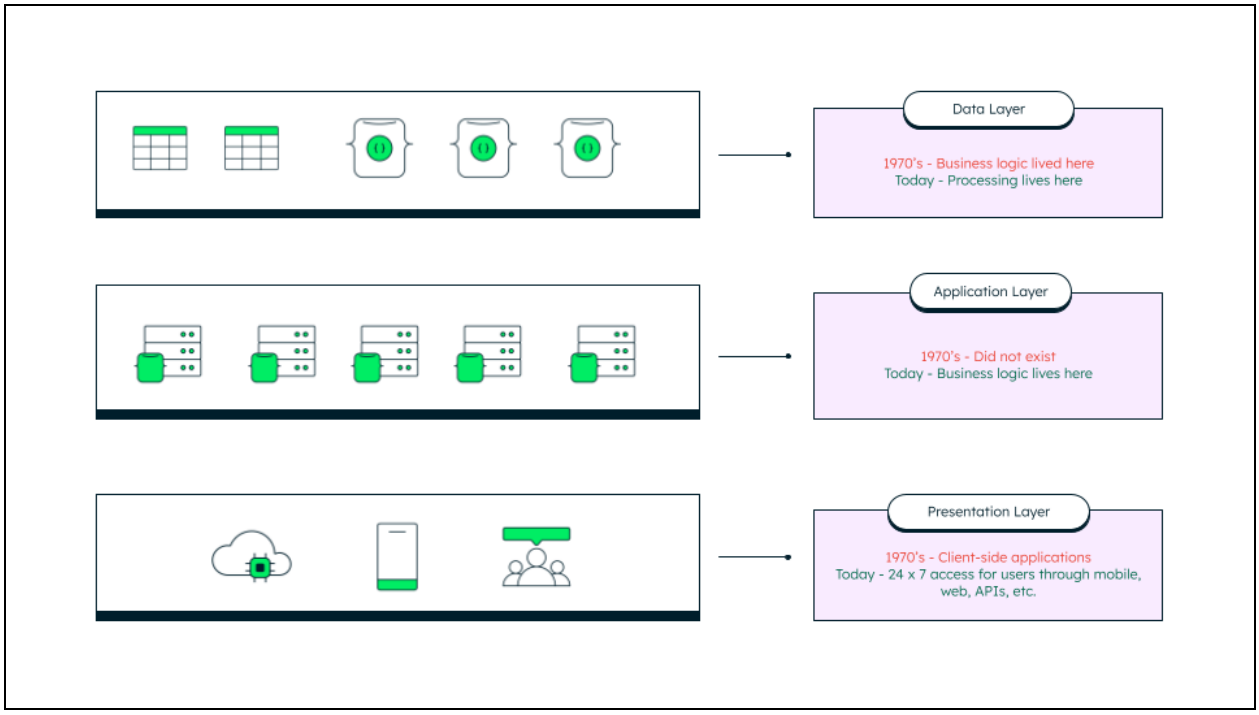


Figure 3: Stored procedures are misplaced in current architectures

Stored procedures are a technical hangover from early client-server architectures in the 70s and 80s, where business logic was stored in the database server rather than the client. Even when three-tier architectures became more popular in the 90s and 2000s, stored procedures were still part of the database layer. Current architectural paradigms call for stored procedures, or more accurately, their modern-day equivalents, to be made part of the application code. In doing so

they can then be part of the overall software development life cycle involving code review, collaborative software development, change management, and so on.

Migrating stored procedures to MongoDB

MongoDB replaces traditional stored procedures with powerful, flexible, and efficient alternatives tailored to the demands of today's digital landscape:

- Aggregation pipelines: Process and analyze data records, grouping values from multiple documents and performing operations on grouped data to deliver results in a single set.
- Atlas Functions: Server-side JavaScript functions for defining app behavior, allowing direct calls from client apps or integration with services for automatic execution. Ideal for low-latency tasks such as data movement, transformations, and validation.
- Atlas HTTPS endpoints: Customizable API routes or webhooks for app-specific integration with external services, utilizing serverless functions to handle incoming requests for specific URLs and HTTP methods.
- Atlas Triggers: Event-driven or scheduled application and database logic execution, running on a serverless compute layer that scales independently of the database server, unlike traditional SQL data triggers.
- Change streams: Subscribe to real-time data changes for single collections, databases, or entire deployments, with the ability to leverage aggregation pipelines for filtering or transforming notifications.

This document will focus exclusively on the use of the aggregation pipeline to replace SQL-based stored procedures, thereby allowing development teams to take greater advantage of the benefits already described.

Aggregation Pipeline Overview

An aggregation pipeline consists of one or more stages that process documents:

- Each stage performs an operation on the input documents. For example, a stage can filter documents, group documents, and calculate values.
- The documents that are output from a stage are passed to the next stage.
- An aggregation pipeline can calculate results for groups of documents. For example, compute the total, average, maximum, and minimum values.

Aggregations are best suited to:

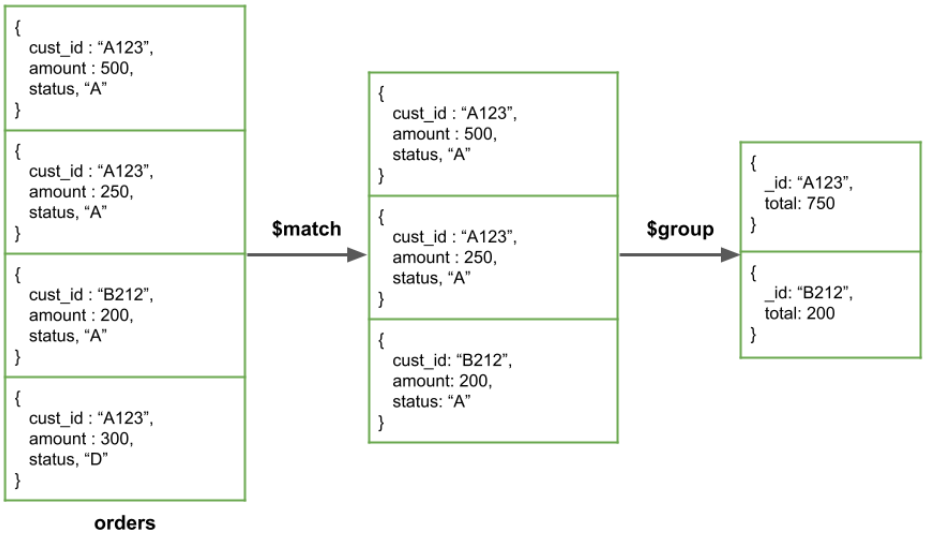
- Group values from multiple documents together.
- Perform operations on the grouped data to return a single result.
- Analyze data changes over time.

The example below shows how the match stage filters the orders collection, based on the document "status" field. The resulting documents are then grouped by the customer ID, or "cust_id" field. The "amount" values of each document are then used to calculate the sum of all the amounts per customer ID.

Where orders is the name of a collection...

```
db.orders.aggregate {[
                  { $match : {status : "A"},
                  { $group : { _id : "$cust_id", total : { $sum :
"$amount"} } }
]}
```

Aggregation pipeline in action:



Aggregation Stages

The following table provides an overview of common SQL aggregation terms, functions, and concepts and the corresponding MongoDB aggregation framework stages:

SQL command	**Aggregation Framework stage**
WHERE	$match
GROUP BY	$group
HAVING	$match
SELECT	$project
LIMIT	$limit
OFFSET	$skip
ORDER BY	$sort
SUM()	$sum
COUNT()	$sum and $sortByCount
JOIN	$lookup

SQL command	Aggregation Framework stage
SELECT INTO <NEW_TABLE>	$out
MERGE INTO <TABLE>	$merge
UNION ALL	$unionWith

In the event that a custom operator needs to be defined in order to implement behavior not supported by the above operators, then the $function operator can be used. The $function operator has the following syntax:

```
{
  $function: {
    body: <code>,
    args: <array expression>,
    lang: "js"
  }
}
```

Examples of the $function operator usage are given later.

The aggregation framework provides a comprehensive list of accumulator type operators such as $avg, $min, $max etc. However, the $accumulator operator also allows us to write custom JavaScript functions to implement behavior not supported by the MongoDB Aggregation Framework. With the help of this operator, we can create a bespoke accumulator function. Operators known as accumulators keep their state as documents move through the pipeline. The $accumulator operator follows many of the same guidelines as the $function operator.

```
{
  $accumulator: {
    init: <code>,
    initArgs: <array expression>,        // Optional
    accumulate: <code>,
    accumulateArgs: <array expression>,
    merge: <code>,
    finalize: <code>,                    // Optional
    lang: <string>
  }
}
```

The $function and $accumulator operators are offered to boost developer efficiency and enable MongoDB to naturally manage a lot of edge cases. Just keep in mind that even though they are powerful, these new operators should only be utilized if previous operators are unable to complete the task because they may degrade performance and are a potential security risk! In many environments server side scripting with javascript is disabled. Consideration must also be given to additional compute requirements around sizing when these operators are used.

For a more detailed comparison of SQL terminology and concepts to their MongoDB counterparts, take a look at the SQL to MongoDB Mapping Chart. It provides examples of various SQL syntax that are ordinarily applied to table data, and how they can be achieved in MongoDB to work on document-based data. In the following sections we will show some examples of SQL syntax commonly found in relational environments, and then the fully worked aggregation pipeline that can be used to achieve the same or similar result.

Conditionals

In SQL the IF statement allows you to either execute or skip a sequence of statements, depending on a condition. The Relational IF statement syntax is described in the left hand column of the table below, with the MongoDB aggregation shown on the right:

Relational/SQL Syntax	MongoDB Aggregation
IF condition_1 THEN statements_1 ELSEIF condition_2 THEN statements_2 [ELSIF condition_3 THEN statements_3] ... ... [ELSE else_statements] END IF;	var gte = [{ "$set": { "sRet": { "$cond": { "if": { "$gte": [{"$convert": { "input": "$param1", "to": "double" }}, {"$convert": { "input": "$param2", "to": "double" }}] }, "then": '1', "else": '0' } } } }, { "$project": { "_id": 0, "param1": 0, "param2": 0 } }]

On a collection containing the following data:

```
...
{
    _id: ObjectId("62d926bced578586b5fbc987"),
    param1: '3',
    param2: '1'
  },
  {
    _id: ObjectId("62d926bced578586b5fbc988"),
    param1: '1',
    param2: '0'
  },
  {
    _id: ObjectId("63737578e3beee5e3c864a72"),
    param1: '5',
    param2: '3'
  },
  {
    _id: ObjectId("637375bce3beee5e3c864a74"),
    param1: '2',
    param2: '4'
  },
  {
    _id: ObjectId("63737614e3beee5e3c864a76"),
    param1: '10',
    param2: '10'
  }
...
```

We can run the aggregation above against it:

```
db.test.aggregate(gte)
```

This returns the comparison of the two input parameters strings, converted to double precision numbers and the return variable is output as a string.

```
  { sRet: '1' },
  { sRet: '1' },
  { sRet: '1' },
  { sRet: '0' },
  { sRet: '1' }
```

The Aggregation pipelines support all conditional, boolean, arithmetic, and comparison operators. For a complete list of currently available operators always check the current Aggregation Pipeline Quick Reference.

CASE statement

The CASE expression goes through conditions and returns a value when the first condition is met (like an if-then-else statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the ELSE clause.
If there is no ELSE part and no conditions are true, it returns NULL.

Simple CASE statement in SQL (Example):	MongoDB Aggregation
DECLARE c_grade CHAR(1); c_rank VARCHAR2(20); BEGIN c_grade := 'B'; CASE c_grade WHEN 'A' THEN c_rank := 'Excellent' ; WHEN 'B' THEN c_rank := 'Very Good' ; WHEN 'C' THEN c_rank := 'Good' ; WHEN 'D' THEN c_rank := 'Fair' ; WHEN 'F' THEN c_rank := 'Poor' ; ELSE c_rank := 'No such grade' ; END CASE; DBMS_OUTPUT.PUT_LINE(c_rank); END	var pipeline = [{"$set": {"_id": 0, "student": 1, "grade": 1, "Comment": {"$switch": {"branches": [{'case': { $gte: ['$grade','8'] }, then: 'Excellent' }, {'case': { $eq: ['$grade','7'] }, then: 'Very Good' }, {'case': { $eq: ['$grade','6'] }, then: 'Good' }, {'case': { $eq: ['$grade','5'] }, then: 'Fair' }, {'case': {$eq: ['$grade','4'] }, then: 'Poor' }], 'default': 'No such grade' } }}}] db.test.aggregate(pipeline)

The MongoDB aggregation equivalent shown above makes use of the switch statement based on the following documents:

```
    {"student" : 'T. Smith', "grade": '7'},
    {"student" : 'J. Doe', "grade": '6'},
    {"student" : 'A. Hill', "grade": '8'},
    {"student" : 'M. Jones', "grade": '8'}
```

The aggregation stage employing the $switch statement works in the same way as its SQL counterpart: by finding an expression that evaluates to true and then breaks out of the control flow. The resulting documents are modified as shown here:

```
  { student: 'T. Smith', grade: '7', comment: 'Very Good' },
  { student: 'J. Doe', grade: '6', comment: 'Good' },
  { student: 'A. Hill', grade: '4', comment: 'Poor' },
  { student: 'M. Jones', grade: '8', comment: 'Excellent' }
```

Loops

SQL provides a number of mechanisms for iteration. We show the following basic pseudo code outlines of the various loop constructs:

```
LOOP
    IF condition THEN
        EXIT;
    END IF;
END LOOP;
```

```
FOR index IN lower_bound ..
upper_bound
LOOP
    statements;
END LOOP;
```

```
WHILE condition
LOOP
    statements;
END LOOP;
```

While the aggregation pipeline does not support loops directly, it does provide a number of ways that you can perform iterative operations on data, using the aggregation operators $filter, $map, and $reduce. As an example, by using the powerful $map operator, elements can be read from an array, and then be transformed by another operator.

Using the following records, we can employ a $map operation to obfuscate or partially mask a series of customer payment cards stored within an array.

```
db.test.insertMany([
    {
        "cust_name" : "Tom Bones",
        "cards": [
            {"provider": "AMEX", "number": "1234567890123456" },
            {"provider": "MC", "number": "0987654321098765"},
            {"provider": "Visa", "number": "6789012345678901"}
        ]

    },
```

```
    {
        "cust_name" : "Jerry Pacemaker",
        "cards": [
            {"provider": "AMEX", "number": "0987654321098765"},
            {"provider": "MC", "number": "1234567890123456" },
            {"provider": "Visa", "number": "4567891234570987"}
        ]
    }
])
```

We create our pipeline below and store it in the variable “mask.”

```
var pipeline = [
    {
        $project:
        {
            _id: 0,
            cust_name: 1,
            masked_number:
            {
                $map:
                {
                    input: '$cards.number',
                    as: 'masked',
                    'in': {
                     $concat: [ 'XXXXXXXXXXXX', {$substrCP: ['$$masked', 12, 4]} ]
                    }
                }
            }
        }
    }
]
```

The $map operation takes as input each “number” element of the “cards” array across all documents. Note the input needs to be an array.

```
"input" : "$cards.number"
```

A new local variable is invoked. This is optional. If not specified then the default variable name of “this” is used.

```
  "as": "masked"
```

The results are output to the output stream, and the local variable “masked” is used to hold the now masked card data and is used when constructing the now transformed output.

In this instance we are concatenating a string of “X” over the first 12 digits on the 16-digit card number. This leaves the final four digits visible. Notice that we reference the masked variable using the $$ notation ($$masked), which is how we invoke a local user variable.

```
 in: {"$concat": ["XXXXXXXXXXXX", {"$substrCP": ["$$masked", 12, 4]}]}
```

We run the aggregation then as follows:

```
db.test.aggregate(pipeline)
```

This results in the following:

```
 [
  {
    cust_name: 'Tom Bones',
    masked_number: [ 'XXXXXXXXXXXX3456', 'XXXXXXXXXXXX8765', 'XXXXXXXXXXXX8901' ]
  },
  {
    cust_name: 'Jerry Pacemaker',
    masked_number: [ 'XXXXXXXXXXXX8765', 'XXXXXXXXXXXX3456', 'XXXXXXXXXXXX0987' ]
  }
]
```

SELECT INTO

The SELECT INTO statement copies data from one table into a new one. In the basic example below, on the left is a generic Relational workflow while on the right you'll see how an aggregation pipeline might be used to write into a new collection.

Relational/SQL (Example)	MongoDB Aggregation
`DECLARE` `  l_customer_name customers.name%TYPE;` `BEGIN` `  -- get name of the customer 100 and assign it to l_customer_name` `  SELECT name INTO l_customer_name` `  FROM customers` `  WHERE customer_id = 100;` `  -- show the customer name` `  dbms_output.put_line( v_customer_name );` `END;`	`var outstage = [` `  {` `  "$set":{` `  "active_cards":` `  {` `  "$cond":` `  {` `  "if": { "$isArray": "$cards"},` `  "then": { "$size": "$cards"},` `  "else": "NA"` `  }` `  }` `  }` `  },` `  {` `  "$out": "active_cards"` `  }` `]`

The aggregation framework provides two stages, $merge and $out, for writing the results of the aggregation pipeline to a collection. A summary of the capabilities of the two stages can be found here.

In short, $out takes the documents returned by the aggregation pipeline and writes them to a specified collection. $out must be the last stage of the pipeline, whereas the $merge stage can write the aggregation output to a collection in the same or different database. The collection could be shared in the $merge instance. Additionally, the $merge stage supports the capability to insert new documents, merge documents, replace documents, keep existing documents, fail the operation, and process documents with a custom update pipeline into an existing collection. Lastly, $merge must also be the final stage in a pipeline.

Using the same documents as previously (in the LOOPS section), we show a brief example of how the $out operator can be used (right column in the table above). In this instance we add an additional field to the document via the $set operator. We do this by first verifying if "$cards" is an array. If so, it uses the $size operator to find the total number of array elements, which is recorded as the actual number of cards a customer has. We run the aggregation as follows:

```
db.test.aggregate(outstage)
```

This will write the output into a new collection called “active_cards” and the new field added to each document is shown when the collection is subsequently queried.

```
db.active_cards.find()
{
    _id: ObjectId("62eb94d146e72c879e57d3ca"),
    cust_name: 'Tom Bones',
    cards: [
      { provider: 'AMEX', number: '1234567890123456' },
      { provider: 'MC', number: '0987654321098765' },
      { provider: 'Visa', number: '6789012345678901' }
    ],
    active_cards: 3
  },
  {
    _id: ObjectId("62eb94d146e72c879e57d3cb"),
    cust_name: 'Jerry Pacemaker',
    cards: [
      { provider: 'AMEX', number: '0987654321098765' },
      { provider: 'MC', number: '1234567890123456' },
      { provider: 'Visa', number: '4567891234570987' }
    ],
    active_cards: 3
  }
```

Here's another example, but this time using the $merge operator, where we show how we can mask data using an aggregation pipeline. Consider the following payments records:

```
{
    _id: ObjectId("63065b65c7c064f934900ded"),
    card_name: 'Mrs. Jane A. Doe',
    card_num: '1234567890123456',
    card_expiry: ISODate("2023-08-31T23:59:59.000Z"),
    card_sec_code: '123',
    card_provider_name: 'Credit MasterCard Gold',
    card_type: 'CREDIT',
    transaction_id: 'eb1bd77836e8713656d9bf2debba8900',
    transaction_date: ISODate("2021-01-13T09:32:07.000Z"),
    transaction_curncy_code: 'GBP',
    transaction_amount: Decimal128("501.98"),
    settlement_id: '9ccb27aeb8394c2b3547521bcd52a367',
    settlement_date: ISODate("2021-01-21T14:03:53.000Z"),
    settlement_curncy_code: 'DKK',
    settlement_amount: Decimal128("4255.16"),
    reported: false,
    customer_info: { category: 'SENSITIVE', rating: 89, risk: 3 }
  },
  {
    _id: ObjectId("63065b65c7c064f934900dee"),
    card_name: 'Jim Smith',
    card_num: '9876543210987654',
    card_expiry: ISODate("2022-12-31T23:59:59.000Z"),
    card_sec_code: '987',
    card_provider_name: 'Debit Visa Platinum',
    card_type: 'DEBIT',
    transaction_id: '634c416a6fbcf060bb0ba90c4ad94f60',
    transaction_date: ISODate("2020-11-24T19:25:57.000Z"),
    transaction_curncy_code: 'EUR',
    transaction_amount: Decimal128("64.01"),
    settlement_id: 'd53799f94d7ad72f698c5a4f04c031a6',
    settlement_date: ISODate("2020-12-04T11:51:48.000Z"),
    settlement_curncy_code: 'USD',
    settlement_amount: Decimal128("76.87"),
    reported: true,
    customer_info: { category: 'NORMAL', rating: 78, risk: 55 }
  }
```

In this case, we want to encrypt the transaction_id of each record. We can do this by using the $function operator to apply javascript function code to generates a hash value using a custom algorithm on the documents transaction_id field in an aggregation stage:

```
var maskstage = {
  "transaction_id": {
            $function: {
                body: function(id) {
                    var hash = 0;
                    for (var i = 0; i < id.length; i++) {
                        hash = ((hash << 5) - hash) + id.charCodeAt(i);
                        hash |= 0;
                    }
                    return hash.toString(16);
                },
                args: ['$transaction_id'],
                lang: 'js'
            }
      }
}
```

We can then run the pipeline:

```
var pipeline = [
    {"$set": maskstage}
]

db.payments.aggregate(pipeline)
```

Comparing the transaction_ids for each document prior to the changes:

```
db.payments.find({},{_id:0, card_name: 1, transaction_id: 1})
[
  {
    card_name: 'Mrs. Jane A. Doe',
    transaction_id: 'eb1bd77836e8713656d9bf2debba8900'
  },
  {
    card_name: 'Jim Smith',
    transaction_id: '634c416a6fbcf060bb0ba90c4ad94f60'
  }
]
```

And then after the transformation has occurred with the pipeline:

```
[
  {
    _id: ObjectId("63065b65c7c064f934900ded"),
    card_name: 'Mrs. Jane A. Doe',
    card_num: '1234567890123456',
    card_expiry: ISODate("2023-08-31T23:59:59.000Z"),
    card_sec_code: '123',
    card_provider_name: 'Credit MasterCard Gold',
    card_type: 'CREDIT',
    transaction_id: 'f847de525f37118f39327768ffb8714c',
    transaction_date: ISODate("2021-01-13T09:32:07.000Z"),
    transaction_curncy_code: 'GBP',
    transaction_amount: Decimal128("501.98"),
    settlement_id: '9ccb27aeb8394c2b3547521bcd52a367',
    settlement_date: ISODate("2021-01-21T14:03:53.000Z"),
    settlement_curncy_code: 'DKK',
    settlement_amount: Decimal128("4255.16"),
    reported: false,
    customer_info: { category: 'SENSITIVE', rating: 89, risk: 3 }
  },
  {
    _id: ObjectId("63065b65c7c064f934900dee"),
    card_name: 'Jim Smith',
    card_num: '9876543210987654',
    card_expiry: ISODate("2022-12-31T23:59:59.000Z"),
    card_sec_code: '987',
    card_provider_name: 'Debit Visa Platinum',
    card_type: 'DEBIT',
    transaction_id: 'f847de525f37118f39327768ffb8714c',
    transaction_date: ISODate("2020-11-24T19:25:57.000Z"),
    transaction_curncy_code: 'EUR',
    transaction_amount: Decimal128("64.01"),
    settlement_id: 'd53799f94d7ad72f698c5a4f04c031a6',
    settlement_date: ISODate("2020-12-04T11:51:48.000Z"),
    settlement_curncy_code: 'USD',
    settlement_amount: Decimal128("76.87"),
    reported: true,
    customer_info: { category: 'NORMAL', rating: 78, risk: 55 }
  }
]
```

We now want to use the $merge operator to write the redacted or masked documents into a new collection, which will only contain documents with md5_hash()'ed transaction_id's. We can do this by creating a copy of the old pipeline, adding the additional $merge stage and running this new aggregation pipeline.

```
new_pipeline = [].concat(pipeline);      // copy original pipeline
```

```
new_pipeline.push(
...     {'$merge': {'into': { 'db': 'testdata', 'coll': 'payments_redacted'}, 'on':
'_id',  'whenMatched': 'fail', 'whenNotMatched': 'insert'}}
... );
2

db.payments.aggregate(new_pipeline);

db.payments_redacted.find();
[
  {
    _id: ObjectId("63065b65c7c064f934900ded"),
    card_name: 'Mrs. Jane A. Doe',
    card_num: '1234567890123456',
    card_expiry: ISODate("2023-08-31T23:59:59.000Z"),
    card_sec_code: '123',
    card_provider_name: 'Credit MasterCard Gold',
    card_type: 'CREDIT',
    transaction_id: 'f847de525f37118f39327768ffb8714c',
    transaction_date: ISODate("2021-01-13T09:32:07.000Z"),
    transaction_curncy_code: 'GBP',
    transaction_amount: Decimal128("501.98"),
    settlement_id: '9ccb27aeb8394c2b3547521bcd52a367',
    settlement_date: ISODate("2021-01-21T14:03:53.000Z"),
    settlement_curncy_code: 'DKK',
    settlement_amount: Decimal128("4255.16"),
    reported: false,
    customer_info: { category: 'SENSITIVE', rating: 89, risk: 3 }
  },
  {
    _id: ObjectId("63065b65c7c064f934900dee"),
    card_name: 'Jim Smith',
    card_num: '9876543210987654',
    card_expiry: ISODate("2022-12-31T23:59:59.000Z"),
    card_sec_code: '987',
    card_provider_name: 'Debit Visa Platinum',
    card_type: 'DEBIT',
    transaction_id: 'f847de525f37118f39327768ffb8714c',
    transaction_date: ISODate("2020-11-24T19:25:57.000Z"),
    transaction_curncy_code: 'EUR',
    transaction_amount: Decimal128("64.01"),
    settlement_id: 'd53799f94d7ad72f698c5a4f04c031a6',
    settlement_date: ISODate("2020-12-04T11:51:48.000Z"),
    settlement_curncy_code: 'USD',
    settlement_amount: Decimal128("76.87"),
    reported: true,
    customer_info: { category: 'NORMAL', rating: 78, risk: 55 }
```

```
  }
]
```

Materialized Views

In relational terms we can think of a Materialized View as persisting data from its view definition query and automatically being updated as data in the underlying tables changes. Such queries may typically consist of SQL-based joins and aggregations. In the MongoDB case, we can easily form complex queries based on aggregation pipelines.

Therefore, in MongoDB terms, an on-demand materialized view is a pre-computed aggregation pipeline result that is stored on and read from disk. Such on-demand materialized views are typically the results of a $merge or $out stage. This implies that rather than executing the expensive query that makes use of joins, functions, or subqueries, any user or application that needs to access this data can simply query the materialized view directly.

Relational/SQL Syntax (Example)	MongoDB Aggregation
`CREATE MATERIALIZED VIEW MV_Employee REFRESH COMPLETE ON COMMIT BUILD IMMEDIATE AS SELECT * FROM Employee;`	`highrisktotals = function(creditScore) { db.loans.aggregate([{ "$set": { "high_risk": { "$cond": { "if": {"$lt": ["$credit_score", creditScore]}, "then" : true, "else": false }} } },{ "$group": { "_id": "$high_risk", "count": { "$sum": 1 } } },{ $merge: { "into": "risky", "whenMatched": "replace"} }]) }`

Using our mortgage collection again to create a materialized view using the aggregation syntax of the right column in the table above. We can show the number of mortgage deals considered as high risk based on a particular credit score. The function “highrisktotals” takes a value for creditScore and uses an aggregation to obtain a count of loans that are considered both high or low risk.

Once the function has been run for a chosen creditScore value, the newly created “risky” collection can be queried to give an immediate response.

```
> highrisktotals(400)

> show collections
loans
risky

> db.risky.find()
[ { _id: true, count: 6 }, { _id: false, count: 18334638 } ]
```

Standard views

A view in SQL is a virtual table built from the results of a SQL statement. Like any database table, a view also has rows and columns. The fields of a view are populated from one or more actual database tables.

A view can be extended with SQL statements and functions to present data as though it were drawn from a single table. Similarly, a MongoDB view is a read-only queryable object whose contents are defined by an aggregation pipeline on other collections or views. MongoDB does not persist the view contents to disk. A view's content is computed on-demand when a client queries the view. In this instance we might want to create the MongoDB version of a standard view. Let us consider our previous US loans collection. We can create a view of the average credit score by state for “Bank of USA” customers using the following aggregation pipeline:

```
var pipeline = [{
 $match: {
  seller_name: 'BANK OF USA, N.A.'
 }
}, {
 $group: {
  _id: '$property_state',
  avg_credit_score: {
   $avg: '$credit_score'
  }
 }
}, {
```

```
 $sort: {
  avg_credit_score: -1
 }}]
```

We show how to create the view, which we are calling “avg_creditscore_state” using the following command:

```
db.createView("avg_creditscore_state", "loans", pipeline)
```

We can query the view (avg_creditscore_state) as follows:

```
db.avg_creditscore_state.find()
[
  { _id: 'OR', avg_credit_score: 760.812247983871 },
  { _id: 'AZ', avg_credit_score: 759.595966049101 },
  { _id: 'CA', avg_credit_score: 759.4811008635226 },
  { _id: 'CO', avg_credit_score: 759.0336244828266 },
  { _id: 'MI', avg_credit_score: 758.282131661442 },
  { _id: 'WA', avg_credit_score: 758.0439215686274 },
 ...
<snipped for brevity>
```

MongoDB documents and PL/SQL Records

A PL/SQL record is a composite data structure which consists of multiple fields; each has its own value. The following picture shows an example record that includes first name, last name, email, and phone number.

The following are examples of several kinds of SQL records that are available.

Table based:	Cursor based:
`DECLARE` `  record_name table_name%ROWTYPE;`	`DECLARE` `  record_name cursor_name%ROWTYPE;` `DECLARE` `  CURSOR c_contacts IS` `  SELECT first_name, last_name, phone` `  FROM contacts;`

	`r_contact c_contacts%ROWTYPE;`

Table-based records use a %ROWTYPE attribute with the table name to declare a record, whereas with a cursor based record the %ROWTYPE attribute is used with the cursor as shown. In the example above we:

- declare an explicit cursor that fetches data from the first_name, last_name, and phone columns of the contacts table.
- declare a record named r_contact whose structure is the same as the c_contacts cursor.

PL/SQL records are used when shifting from field-level to record-level operations. MongoDB already simplifies operations in this way with the use of its document model. This allows developers to change and manipulate data on either a per document or per field entry basis irrespective of the document shape. In the example below we show how we can simply manipulate a document by embedding additional fields to store the required number of phone numbers.

```
{
   “first_name” : “Arthur”,
   “last_name” : “Clarke”,
   “email_address” : “Arthur.C.Clarke@mysteriousworld.com”,
   “phone_number” : “408 123 623”
}
```

After update :

```
{
   “first_name” : “Arthur”,
   “last_name” : “Clarke”,
   “email_address” : “Arthur.C.Clarke@mysteriousworld.com”,
   “phone_number” : [
                    { “Home” : “408 123 623” } ,
                    { “Work” : “850 456 321” }
                    ]
}
```

We mentioned cursors previously and in particular we mentioned explicit cursors. Let's take some time to distinguish between the type of cursors available in PL/SQL and their MongoDB equivalents. Cursors in MongoDB are similar to those in the relational space. You can think of them as a type of pointer to the results of either a query or aggregation pipeline.

Implicit cursors

Whenever a relational database executes an SQL statement such as SELECT INTO, INSERT, UPDATE, and DELETE, it automatically creates an implicit cursor. This is internally managed across the whole execution cycle of an implicit cursor and reveals only the cursor’s information and statuses such as SQL%ROWCOUNT, SQL%ISOPEN, SQL%FOUND, and SQL%NOTFOUND. The

implicit cursor is not particularly elegant when the query returns zero or many rows, which cause NO_DATA_FOUND or TOO_MANY_ROWS exceptions respectively.

This is basically not that much different to how every MongoDB query and aggregation operates. You may already be familiar with how results are output in the MongoDB shell. In particular, when output via a cursor it is not saved into a variable. Then in such cases, the first twenty (20) results are iterated over and displayed. In order to see subsequent results you must “iterate” (type “it” in the mongo shell) again for the next 20 query or aggregation results and so on.

Example output below :

```
db.people.find()
[
  {
    _id: ObjectId("57d7a121fa937f710a7d4874"),
    last_name: 'Johnson',
    quote: 'Esse quaerat suscipit voluptate ipsam magni tempore
accusamus.',
    job: 'Sales executive',
    ssn: '081-78-3110',
    address: {
      city: 'Anthonyhaven',
      street: '41441 Galvan Lights Apt. 820',
      zip: '53666'
    },
    first_name: 'Martin',
    company_id: ObjectId("57d7a121fa937f710a7d486d"),
    employer: 'Terry and Sons',
    birthday: ISODate("2012-01-21T08:13:29.000Z"),
    email: 'sboyd@yahoo.com'
  },
< output snipped for brevity>
  {
    _id: ObjectId("57d7a121fa937f710a7d4888"),
    last_name: 'Anderson',
    quote: 'Quod commodi amet amet magni temporibus distinctio.',
    job: 'Set designer',
    ssn: '199-63-0879',
    address: {
      city: 'Port Elizabeth',
      state: 'Colorado',
      street: '0273 Brown Estates Suite 480',
      zip: '86332'
```

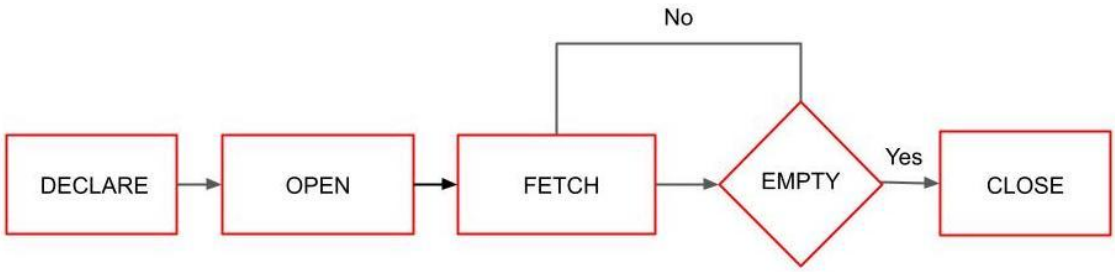
```
    },
    first_name: 'Justin',
    company_id: ObjectId("57d7a121fa937f710a7d486d"),
    employer: 'Terry and Sons',
    birthday: ISODate("2011-02-11T12:37:01.000Z"),
    email: 'pfields@jacobs.biz'
  }
]
Type "it" for more
```

The number of results displayed is controlled by the attribute config.set("displayBatchSize"). See below for how to change the attribute if required:

```
> config.set("displayBatchSize", 10)
```

Explicit cursors

An explicit cursor is a SELECT statement declared explicitly in the declaration section of the current block or a package specification. For an explicit cursor, you have control over its execution cycle from OPEN, FETCH, and CLOSE. Oracle defines an execution cycle that executes an SQL statement and associates a cursor with it. The following illustration shows the execution cycle of an explicit cursor:



To show a relatively simple way to see cursor behavior, we can start by saving a cursor from the Aggregation Framework into a variable. MongoDB allows you to manually iterate over that cursor in much the same way as you would for its SQL equivalent described above. Once saved into a variable we can iterate over the returned result set and access the resultant documents.

Here's our example of using cursors with an aggregation. In this instance we have a collection of U.S. mortgage record details. An individual record is shown below:

```
db.loans.findOne()
{
  _id: ObjectId("60d1e3d3105718f4c747e258"),
  credit_score: 790,
  first_payment_date: ISODate("2017-01-01T00:00:00.000Z"),
  first_time_homebuyer_flag: '9',
  maturity_date: ISODate("2046-12-01T00:00:00.000Z"),
  metropolitan_statistical_area_msa_or_metropolitan_division: 12580,
  mortgage_insurance_percentage_mi: 0,
  number_of_units: 1,
  occupancy_status: 'P',
  original_combined_loan_to_value_cltv: 69,
  original_debt_to_income_dti_ratio: 30,
  original_upb: 517000,
  original_loan_to_value_ltv: 69,
  original_interest_rate: 3.625,
  channel: 'C',
  prepayment_penalty_mortgage_ppm_flag: 'N',
  amortization_type_formerly_product_type: 'FRM',
  property_state: 'MD',
  property_type: 'SF',
  postal_code: 21000,
  loan_sequence_number: 'F16Q40000002',
  loan_purpose: 'N',
  original_loan_term: 360,
  number_of_borrowers: 2,
  seller_name: 'Other sellers',
  servicer_name: 'LAKEVIEW LOAN SERVICING, LLC',
  super_conforming_flag: 'Y',
  program_indicator: '9',
  property_valuation_method: 9,
  interest_only_io_indicator: 'N'
}
```

In the aggregation, we want to find the average credit score by the U.S. state for a particular mortgage lender. We have a $match stage that filters on the required mortgage vendor: "BANK OF USA, N.A." We then use a $group stage in order to group the filtered properties on the state where the properties are located (_id: "$property_state") and calculate the average credit score ("$credit_score") for that particular state. The newly calculated average is then stored in the avg_credit_score field. The average credit scores are then sorted, via the $sort stage, in descending order.

So, our aggregation pipeline looks like this:

```
var pipeline = [
    {
        $match:
        {
            seller_name: 'BANK OF USA, N.A.'
        }
    },
    {
        $group:
        {
            _id: '$property_state',
            avg_credit_score:
            {
                $avg: '$credit_score'
            }
        }
    },
    {
        $sort:
        {
            avg_credit_score: -1
        }
    }
]
```

In our example, this would be the declaration (DECLARE) stage:

```
var cursor = db.loans.aggregate(pipeline)
```

Opening, ie OPEN, the cursor with the next() function as follows:

```
cursor.next()
```

We can then read or FETCH documents using the next() cursor function:

```
printjson(cursor.next())
```

Regardless of how the cursor is created, it will automatically close due to either timeout or when the cursor has been exhausted. In order to explicitly perform the CLOSE stage however, we can call the close() function on the cursor. This will of course free the resources being used by the cursor.

```
cursor.close()
```

Putting the above altogether in the MongoDB shell might look like this:

```
> var cursor = db.loans.aggregate(pipeline)

> cursor.next()
{ _id: 'OR', avg_credit_score: 760.812247983871 }

Or alternatively,

> while (cursor.hasNext()) {
        printjson(cursor.next());
 }
{ _id: 'CO', avg_credit_score: 759.0336244828266 }
{ _id: 'WA', avg_credit_score: 758.0439215686274 }
{ _id: 'VA', avg_credit_score: 757.6168628878909 }
....

And optionally, if needed

> cursor.close()
```

Performance Considerations

There are two main focus areas when using aggregations. One is for real-time or near real-time processing, whereby we provide data for applications. In this case query performance is important in terms of latency or responsiveness. The other area is in batch processing where we provide data for analytics use cases. This area often has less stringent latency demands. Hence, we will concern ourselves for the most part, with real-time processing aspects in this section.

While a complete treatment of aggregation pipeline performance is beyond the scope of this document, there are some guidelines that, when followed while building aggregation pipelines, ensure that performance-related good practices are employed.

The two main components of performance to consider are index usage and avoiding memory constraints.

Index Usage

Recall that aggregations are able to make use of indexes in the same way as find(). However, bear in mind that once we use a stage that can't use indexes, then subsequent stages can no longer use indexes either. Note that the query optimizer attempts to move stages to the head of the pipeline, when possible, in order to take advantage of any indexes. The $match stage can make use of indexes — in particular when used at the head of the pipeline. The same goes for the $sort stage. While not true in every case, we should try to use such stages at the start of a pipeline and ensure

that they come where possible, before any stage that will reshape or transform the queried data. There are, of course, instances where data transform stages can be part of such stage rearrangement.

If you use a $limit stage with $sort, try to make sure that $limit comes after $sort where possible and again at the head of any pipeline. By putting $limit after $sort, and placing them together, it ensures the use of the Top-k sorting algorithm which is highly performant in these situations. The Top-k algorithm returns the k best answers of a query according to a given ranking.

Memory constraints

The 16-MB document limit of MongoDB applies to each document in a set of final aggregation results. This only applies to the amount of documents that are produced as a result, not the quantity of documents as they pass through the pipeline. In general, you can use $limit and $project stages appropriately to reduce or reshape your data and ensure you avoid hitting either the individual document limit, as well as any per stage memory limit, which we discuss in greater detail below.

The 100-MB per stage heap memory limit only really has an impact on what are termed blocking stages. Most stages are referred to as streaming stages, in that they will immediately pass data as it's processed in one stage on to the next stage of a pipeline. Blocking stages on the other hand are required to block and wait for all of the batches of data to arrive at that particular stage. Stages such as $sort and $group (and what we might term as $group-type stages) all block. If we take $sort as an example, then as a sort requires all of the data batches to be accumulated at that stage in order to sort all of the data, we can see why it has to block. A broader treatment and discussion of both streaming and blocking stages can be read here.

You can mitigate against blocking-type stages by again ensuring the use of indexes as well as the earlier recommendations of using previous pipeline stages to reduce data being passed to a blocking stage. If, after all the above attempts at reducing memory usage, you find you are still breaking the 100-MB stage limit, your data will spill to disk. This will not be as performant and is possibly only suitable when we consider aggregations for batch processing purposes, where performance is often not a key requirement.

Using explain() to understand Aggregation performance

Given that often more complex logic can be applied and run with aggregations, a useful and in many cases, critical tool to use for examining how your aggregation operates is to use an explain plan. Such a plan will allow you to understand how the various stages of your aggregation have been initially ordered and allow you to perform the required optimization to improve its performance.

Let's use the previous loans collection aggregation example (from the cursors section) to see how the explain function can help us to investigate the aggregation execution plan and where

necessary, apply our earlier performance advice in order to obtain improved aggregation query times.

For convenience sake, if you are going to repeatedly run explain plans then it it might be worth creating an explainable object and storing it in a variable.

```
exp = db.loans.explain("executionStats")
Explainable(loan_project.loans)
```

We can then run this against our previous aggregation query in order to generate the execution stats for our aggregation. Here's our original form before:

```
exp = db.loans.explain("executionStats")
Explainable(loan_project.loans)
```

Now, let's run the explain plan against this. Note that we have edited the output in order to remove entries not relevant to the current discussion. First, let's take a look at the winningPlan section

```
exp.aggregate(pipeline)
{
  explainVersion: '1',
  stages: [
    {
      '$cursor': {
        queryPlanner: {
          namespace: 'loan_project.loans',
          indexFilterSet: false,
          parsedQuery: { seller_name: { '$eq': 'BANK OF USA, N.A.' } },
          queryHash: '31812AAF',
          planCacheKey: '2607C515',
          maxIndexedOrSolutionsReached: false,
          maxIndexedAndSolutionsReached: false,
          maxScansToExplodeReached: false,
          winningPlan: {
            stage: 'PROJECTION_SIMPLE',
            transformBy: { credit_score: 1, property_state: 1, _id: 0 },
            inputStage: {
              stage: 'COLLSCAN',
              filter: { seller_name: { '$eq': 'BANK OF USA, N.A.' } },
              direction: 'forward'
            }
          },
```

Unsurprisingly, as we have no indexes on the collection currently, the winning plan includes a full collection scan, or COLLSCAN. This means we had to read every document in a huge collection consisting of 18,334,644 documents. This is followed by a PROJECTION_SIMPLE.

We can confirm this by looking at the executionStats section. This shows the number of documents that were read, how many documents were returned, and how many index keys were used. Ideally we would want all documents to be found via an index and the ratio of documents examined to returned to be as close to one as possible. Also note the query execution time (executionTimeMillis) of 31454 ms, or around 31 seconds.

```
executionStats: {
          executionSuccess: true,
          nReturned: 706404,
          executionTimeMillis: 31454,
          totalKeysExamined: 0,
          totalDocsExamined: 18334644
```

The output also shows we have a $sort stage, which, even while we don't spill to disk, could still be improved upon by using an appropriate index.

```
{
      '$sort': { sortKey: { avg_credit_score: -1 } },
      totalDataSizeSortedBytesEstimate: Long("12495"),
      usedDisk: false,
      nReturned: Long("51"),
      executionTimeMillisEstimate: Long("31444")
    }
```

Let's apply the performance best practice discussed earlier to the aggregation and see if we can improve the query time and pipeline performance in general. First, we need to move as many stages that could rely on an index to the start of the pipeline. We can also ensure we are only working with document fields that are of interest. Additionally, we could even limit the aggregation results set to return a smaller number of documents.

So, moving pipeline stages, or at least ensuring any added stages get added to the head of the pipeline ensures that when we add our index, these stages can take advantage of the index in terms of query filtering and sorting. Here's our original aggregation but with the necessary changes to ensure best practices:

```
var pipeline = [{
    $match: {
     seller_name: 'BANK OF USA, N.A.'
    }
   },
   {
    $project: {"_id": 0, "seller_name": 1, "property_state": 1,
"credit_score": 1}
   }, {
    $group: {
      _id: '$property_state',
      avg_credit_score: {
      $avg: '$credit_score'
     }
    }
   },{
    $sort: {
        avg_credit_score: -1
    }
   }]
```

The added $project stage ensures that only the document fields we will work on are returned. Another common mistake is to not provide a projection that filters out the _id field, which is returned by default. If the _id field is not a field in the index definition, it is not available, and the query system will need to fetch the full document to retrieve the value.

Don't forget the rule that by adding additional stages at the head of the pipeline they can also take advantage of any indexes that will be used by the other early stages such as $match, etc.

At this point, we need to create the index that we have been discussing. We will index on each of the document fields that our aggregation stages will perform some work on, and according to the ESR Rule. Hence, we create the following compound index:

```
db.loans.createIndex({"seller_name": 1, property_state: 1, credit_score :1})
seller_name_1_property_state_1_credit_score_1
```

To see how this improves our aggregation query we need to run it again, with the explain plan.

```
exp.aggregate(pipeline)
{
  ...
       winningPlan: {
            stage: 'PROJECTION_COVERED',
            transformBy: {
              seller_name: true,
              property_state: true,
              credit_score: true,
              _id: false
            },
            inputStage: {
              stage: 'IXSCAN',
              keyPattern: { seller_name: 1, property_state: 1,
credit_score: 1 },
              indexName: 'seller_name_1_property_state_1_credit_score_1',
              isMultiKey: false,
              multiKeyPaths: { seller_name: [], property_state: [],
credit_score: [] },
              isUnique: false,
              isSparse: false,
              isPartial: false,
              indexVersion: 2,
              direction: 'forward',
              indexBounds: {
                seller_name: [
                  '["BANK OF USA, N.A.", "BANK OF USA, N.A."]'
                ],
                property_state: [ '[MinKey, MaxKey]' ],
                credit_score: [ '[MinKey, MaxKey]' ]
              }
  ...
```

We can immediately see from our winning plan that both stages, IXSCAN and PROJECTION_COVERED, have been able to make use of the index we provided. This has canceled the previous requirement for a collection scan and made sure the “total documents/keys examined” to “documents returned (nReturned)” ratio is equal to one. We can verify this below in the executionsStats stage:

```
executionStats: {
         executionSuccess: true,
          nReturned: 706404,
          executionTimeMillis: 1320,
          totalKeysExamined: 706404,
          totalDocsExamined: 0,
```

All the pipeline stages that can take advantage of an index have done so, thereby ensuring documents did not have to be fetched from disk but instead read directly from the index. This has reduced the aggregation query execution time to a little over 1.3 ms.

For further performance discussion you are invited to read the chapter entitled Pipeline Performance Considerations in the Practical MongoDB Aggregations Book.

Worked Examples

Based on the following relational schema, we intend to show how a non-trivial relational stored procedures can be easily converted into a MongoDB aggregation framework using the readily available framework operators and constructs. For the stored procedure, we create the required relational schema as follows:

```
CREATE TABLE Customer (
   ID INT PRIMARY KEY,
   Name VARCHAR(50) NOT NULL,
   PhoneNumber VARCHAR(20),
   ZipCode VARCHAR(10),
   Occupation VARCHAR(50)
)

CREATE TABLE CustomerTransactionSummary (
   ID INT PRIMARY KEY,
   CustomerID INT NOT NULL,
   TransactionDate datetime NOT NULL,
   TransactionAmount money NOT NULL,
   CardNumber varchar(16) NOT NULL,
   CONSTRAINT FK_CustomerTransactionSummary_CustomerID FOREIGN KEY
(CustomerID) REFERENCES Customer(ID)
)
```

We then populate the table with the following row data:

```
INSERT INTO Customer (ID, Name, PhoneNumber, ZipCode, Occupation) VALUES
(1, 'John Smith', '555-1234', '90210', 'Lawyer'),
(2, 'Jane Doe', '555-5678', '60601', 'Doctor'),
(3, 'Mike Johnson', '555-4321', '90210', 'Lawyer'),
(4, 'Emily Wang', '555-8765', '60601', 'Doctor'),
(5, 'David Kim', '555-1111', '90210', 'Software Engineer'),
(6, 'Lisa Davis', '555-2222', '60601', 'Teacher'),
(7, 'Tom Lee', '555-3333', '90210', 'Software Engineer'),
(8, 'Karen Kim', '555-4444', '60601', 'Doctor'),
(9, 'Scott Johnson', '555-5555', '90210', 'Lawyer'),
(10, 'Susan Brown', '555-6666', '60601', 'Software Engineer');
```

```
INSERT INTO CustomerTransactionSummary (ID, CustomerID, TransactionDate,
TransactionAmount, CardNumber) VALUES
(1, 1, '2022-01-01 08:30:00', 100.00, '1234567890123456'),
(2, 1, '2022-01-02 12:30:00', 50.00, '2345678901234567'),
(3, 1, '2022-01-03 10:00:00', 75.00, '3456789012345678'),
(4, 2, '2022-01-04 14:30:00', 200.00, '4567890123456789'),
(5, 2, '2022-01-05 09:00:00', 300.00, '5678901234567890'),
(6, 3, '2022-01-06 15:00:00', 150.00, '6789012345678901'),
(7, 3, '2022-01-07 11:30:00', 100.00, '7890123456789012'),
(8, 4, '2022-01-08 12:00:00', 75.00, '8901234567890123'),
(9, 4, '2022-01-09 16:30:00', 100.00, '9012345678901234'),
(10, 5, '2022-01-10 10:00:00', 25.00, '0123456789012345'),
(11, 5, '2022-01-11 12:30:00', 50.00, '1234567890123456');
```

Our example stored procedure below, It employs a dynamic WHERE clause to retrieve the Name, Phone Number, and masked CardNumber of the top 10 Customers. The procedure can operate on all data or filter the results based on the input parameters for ZipCode and Occupation fields.

PL/SQL Stored Procedure (Example)

```
CREATE PROCEDURE TOPCUSTOMERS
   @ZipCode varchar(50) = null,
   @Occupation varchar(50) = null
AS
BEGIN
   IF (@ZipCode IS NOT NULL AND @Occupation IS NOT NULL)
   BEGIN
       SELECT TOP 10 C.Name, C.PhoneNumber,
       REPLACE(LEFT(T.CardNumber, 12), SUBSTRING(T.CardNumber, 1, 4),
'****') + RIGHT(T.CardNumber, 4) AS MaskedCardNumber
       FROM Customer C
       INNER JOIN (
           SELECT *
           FROM CustomerTransactionSummary
           WHERE CustomerID IN (
               SELECT ID
               FROM Customer
               WHERE ZipCode = @ZipCode AND Occupation = @Occupation
           )
       ) T ON C.ID = T.CustomerID
       ORDER BY C.Name
   END
   ELSE
   BEGIN
       SELECT TOP 10 C.Name, C.PhoneNumber,
       REPLACE(LEFT(T.CardNumber, 12), SUBSTRING(T.CardNumber, 1, 4),
'****') + RIGHT(T.CardNumber, 4) AS MaskedCardNumber
       FROM Customer C
       INNER JOIN CustomerTransactionSummary T ON C.ID = T.CustomerID
       ORDER BY C.Name
   END
END
```

When run from an SQL*Plus session, the stored procedure output for the sample dataset provided looks like this:

```
SQL> set serveroutput on
SQL> EXECUTE TOPCUSTOMERS('''60601''', '''Doctor''') ;

NAME                 PHONENUMBER     CARDNUMBER
-----------------------------------------------------
Dr. Jane Doe      555-5678           **** **** **** 1234
John Smith  555-8765                 **** **** **** 5678
Karen Kim   555-4444                 **** **** **** 9012
...                       ...                                  ...

PL/SQL procedure successfully completed.
```

For a MongoDB collection containing the same data, we can run an equivalent aggregation pipeline to generate the same output.

```
db.customer.aggregate([
 {
   $match: {
     Occupation: "Doctor",
     ZipCode: "60601"
   }
 },
 {
   $project: {
     _id: 0,
     Name: 1,
     PhoneNumber: 1,
     CardNumber: {
       $concat: [
         "************",
         {
           $substr: ["$CardNumber", 12, 4]
         }
       ]
     }
   }
 },
 {
   $sort: {
     Name: 1
   }
 },
 {
   $limit: 10
 }
])
```

The output results from the pipeline is shown below, and is the same as the stored procedure output above.


```
[
 {
   "name": "Dr. Jane Doe",
   "phoneNumber": "555-5678",
   "cardNumber": "**** **** **** 1234"
 },
 {
   "name": "John Smith",
   "phoneNumber": "555-8765",
   "cardNumber": "**** **** **** 5678"
 },
 {
   "name": "Karen Kim",
   "phoneNumber": "555-4444",
   "cardNumber": "**** **** **** 9012"
 }
]
```


Note: Customer and Customer Transaction Summary tables are modeled within the same collection, so there is no need for additional join and the $match stage in an aggregation pipeline can be omitted if a WHERE clause is unnecessary. If the aggregation pipeline is being frequently executed, it can be transformed into views. By doing so, we can utilize find queries on these views to efficiently access and analyze the data. Please refer to the documentation to gain a better understanding of how to enhance the performance of the aggregation pipeline.

Conclusion

The replacement of stored procedures with MongoDB's aggregation framework means aggregations are now an implicit part of your application code. They can be tested, maintained, and deployed as part of the release cycle. Developers can incorporate them within each step of the development process and dispense with outdated, hard-to-maintain code blocks that can stifle innovation. In this document we have covered how to implement MongoDB aggregation pipelines as a means to replace stored procedures based on SQL for relational schemas. By comparing the various aggregation operators and their SQL syntactic equivalents, we showed how aggregations can provide a modern, composable alternative means to process large amounts of data, both in terms of analytical and data processing. By covering aggregation best practice, we considered stage layout, ordering and co-location etc. We hope to ensure the avoidance of initial performance traps and allow more time to be spent on developing aggregations. This document is by no means an exhaustive treatment of the MongoDB aggregation framework. We have provided links, both within the document and in the reference section, to additional reading to further broaden your knowledge of aggregation framework pipelines and their many use cases.

Resources

Additional Reading

Practical MongoDB Aggregations Book
Stored Procedures in MongoDB

Useful Links

MongoDB: Developer Community
MongoDB: Atlas Triggers
MongoDB Manual: Change Streams
MongoDB: Aggregation
Generating Globally Unique Identifiers for Use with MongoDB

Safe Harbor

The development, release, and timing of any features or functionality described for MongoDB's products remains at MongoDB's sole discretion. This information is merely intended to outline MongoDB's general product direction and it should not be relied on in making a purchasing decision nor is this a commitment, promise or legal obligation to deliver any material, code, or functionality.

Every effort has been made to ensure the accuracy of statements made in this document with regards to third party organizations (and their products and those products' capabilities) at the time of publication. This includes verification and cross-referencing of the following sources: third party and independent documentation, webinars, live streams, events, articles, marketing collateral, reports, interviews and blogs, analyst engagements, user reviews, discussions with former employees & partners, and internal peer review. Third party product capabilities may evolve and shift subsequent to publication. Therefore it is recommended that readers verify whether stated capabilities and limitations are accurate at the time of reading.

Authors

Vittal Pai

Vittal Pai is a Senior Solutions Architect working for MongoDB. In his current role, he actively leads technical engagements with ISV and OEM partners, assist them in building solutions that adhere to the best practices of MongoDB. Prior to joining MongoDB, he spent nearly a decade at IBM as a full stack developer, working extensively with diverse technologies, including mobile and cloud platforms.

Ray Hassan

Ray is Partner Solutions Architect working with MongoDB's SI partners. With over 20 years industry experience, he has a track record of delivering innovative reference architecture solutions.

Vasanth Kumar

Vasanth is a Principal Solutions Architect with 20 Years of experience in building enterprise products across BFSI & IoT Sector. Working with MongoDB for the past three years helping ISVs to modernize their platform.