Omesairam

change streams allow applications to access real-time data changes without the complexity and risk of tailing the oplog

<https://developpaper.com/first-experience-of-mongodb-change-stream/>

**What is Change Stream?**

As the name implies, Change Stream is a change flow, which is a way for MongoDB to publish data changes to applications. That is, when any data changes in the database, the application can be notified. It can be understood as a trigger that executes in an application. The aggregation framework can be used to filter and transform what data an application wants and how it gets it. This will be discussed later.

**Principle of Change Stream**

Let’s first review how MongoDB replication sets work in general:

* The application initiates write requests to the database by driving.
* In the same transaction, MongoDB completes the oplog and set modifications.
* Oplog is pulled away by other slave nodes.
* The oplog obtained from node application also completes the modification of oplog and set in one transaction.

At this point, the replication set is synchronized. It can be found that the whole synchronization process depends on oplog. That is to say, oplog actually contains all the change data we need. If we observe the changes in oplog, can we get all the data of the changes? Yes, change stream is based on this principle. But it’s not that simple! Let’s see where the problem may be.

**How to Recovery from Breakpoint**

In the real world, no application can run uninterruptedly. Without considering the problems caused by bugs, normal application upgrades can also cause applications to interrupt. So where do you continue to get changes when the application is restored? Oplog can certainly help us do this, but you have to know enough about MongoDB to know that there are parameters such as oplog Replay, and some other issues.

**How to deal with subscriptions effectively**

Assuming that you need to subscribe to 10 different sets of changes in an application, do you need to open 10 tailable cursors to get oplog changes? What about 100 collections? Obviously this should not be done for efficiency reasons. Then the whole process becomes a producer-consumer model, with one thread responsible for getting changes from the oplog and the subscribed thread responsible for consuming those changes. Although implementations are not that complex and open source implementations are mostly available, multithreading is enough to give beginners a headache.  
To be fair, these are not serious problems, and the following ones may be even more troubling.

**How to manage authority**

To tail oplog, you mustlocal.oplog.rsRead permission. In fact, this is equivalent to having read access to the entire database, because all changes will be reflected here. DBA may prevent you from doing so, because it’s not a very safe way to do it.

**How to Roll Back Data**

In extreme cases, data rollback may occur in MongoDB if the application is not properly processed. If the oplog is tracked only, the notified changes are rolled back.

Fortunately, none of these issues is a problem now, because change stream helps us avoid these complex details.

**Usage method**

Because all drivers have different grammars and APIs, trying to use change stream from the shell is probably the easiest way. This does not prevent you from subsequently using the various drivers, because the functions that can be implemented in the shell must have the corresponding syntax in the driver. Let’s take shell as an example to see how change stream should be used.

**Open a shell and subscribe to the collection you need to focus on**  
For example:

var cursor = db.bar.watch();

For demonstration purposes, we iterate over the cursor in this shell to get new data:

while(true) {

if (cursor.hasNext()) {

print(JSON.stringify(cursor.next()));

}

}

**Open another shell and go tobarInsert a data into the collection:**

db.bar.insert({y: 1})

At this point, the change data is immediately output in the first shell:

{"\_id":{"\_data":{"$binary":"glzquiIAAAACRmRfaWQAZFzquiK0lDNo+K0DpwBaEARUMrm0ruVACoftuxjt1RtCBA==","$type":"00"}},"operationType":"insert","fullDocument":{"\_id":{"$oid":"5ceaba22b4943368f8ad03a7"},"y":1},"ns":{"db":"test","coll":"bar"},"documentKey":{"\_id":{"$oid":"5ceaba22b4943368f8ad03a7"}}}

A brief introduction to some of the fields here. For a more complete introduction, please refer to the document change events:

* \_idUsed to restore breakpoints. Knowing this value, changes can be restored from this breakpoint in the next restart after the application is disconnected.
* operationTypeOperational types, common values include:
  + insert
  + update
  + delete
* nsNamespaces in action
* fullDocumentComplete documentation

**Recovery from breakpoints**

var cursor = db.bar.watch([], {resumeAfter: <\\_id>})

Use at this timehasNext()/next()Subsequent changes can be obtained.

**Matters needing attention**

**{readConcern: ‘majority’}**

To avoid rollback updates being published, change stream chooses to publish these changes to the application only when a change reaches most nodes (which cannot be rolled back). The way in which it is used is{readConcern: "majority"}。 Therefore, the change stream will not notify the application of any changes in the following circumstances:

* DisabledreadConcern；
* Upgraded from the old version, but not updatedfeatureCompatibilityVersion；
* S downtime in PSA architecture;

**Breakpoint Recovery Time**

Because change stream relies on oplog, it naturally faces all the problems that oplog faces. One of the problems is that oplog is overwritten. Therefore, in order to ensure that breakpoints can be restored, it is necessary to ensure that breakpoints are requested within the time of oplog window application.

**Delete set**

If the collection is deleted during a subscription collection change, a message is receivedinvalidInformation notification indicates that the collection is no longer available:

{

"\_id" : {

"\_data" : BinData(0,"glzqxCcAAAACFFoQBFQyubSu5UAKh+27GO3VG0IE")

},

"operationType" : "invalidate"

}

**Reference material**

* Tailable cursor: https://docs.[mongodb](https://developpaper.com/tag/mongodb/).com/manual/core/tailable-cursors/
* Producer-consumer model: https://zh.wikipedia.org/wiki/%E7%94%9F%E4%BA%A7%E8%80%E6%88%E8%B4%B9%80%E9%85%E9%97%AE%E9%A2%98
* About rollback: https://docs.mongodb.com/manual/core/replica-set-rollbacks/
* Change events: https://docs.mongodb.com/manual/reference/change-events/
* Change Stream Introduction Document: https://docs.mongodb.com/manual/change Streams/

**Author brief introduction**

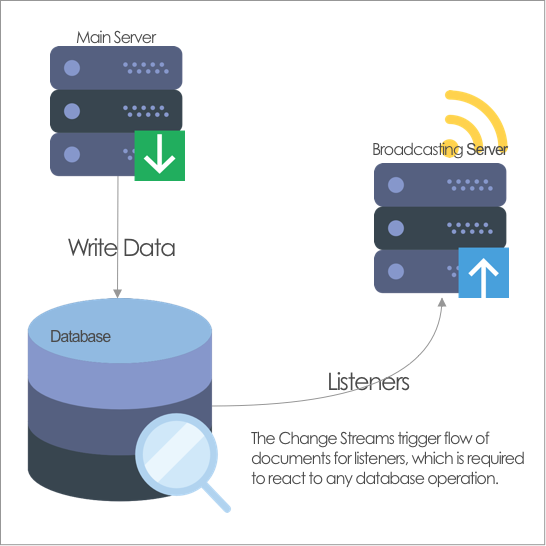
Zhang Yaoxing, Chief Technical Advisory Service Consultant for MongoDB Asia Pacific Region. I have many years of practical experience in the development, application and consulting services of MongoDB. As an expert in MongoDB certification, he has provided training, performance tuning, architecture design and other related MongoDB technology services for various large customers in different industries.

<https://medium.com/@mandalrajdeep/using-change-streams-in-mongodb-50ca3f44421a>

In the pursuit of making scalable systems, I have discovered that using listeners in your database makes your life many times easier. Here are some of the instances, when these listeners saved my life. Well, saved is a bit of an overstatement, but definitely made it more comfortable.

* **The building of reactive systems**
* **Data pipelines to other systems**
* **Updating parameters on UI and live notifications**

In a system where there is a frequent influx of data, which eventually gets dumped in a database, there are two alternate ways of tackling this. One way to do this is to tail the oplogs. The other way is to incorporate this logic somewhere before interaction with the database. I am not a fan of either, as these are what you call dirty ways of programming.



An Illustration of Having a Different Server for Listeners

*The idea here is to build an independent application, that subscribes to a particular category of DB changes, and immediately reacts to them.*

If you are already decently aware of what **Change Streams** are, you may directly jump to [Change Stream Output](https://medium.com/p/50ca3f44421a/) for detailed intricacies.

# Prerequisites

In order to use Change Streams, one must use a distributed database, referred to as the replica set, which is very well facilitated in MongoDB.

The database maintains an [oplog](https://docs.mongodb.com/manual/core/replica-set-oplog/" \t "_blank), which is a [capped collection](https://docs.mongodb.com/manual/core/capped-collections/) storing an ordered list of logical writes to the primary replica. This oplog facilitates the secondary replica to apply the changes to their own local copies.

Change Streams are applicable to replica sets in [MongoDB 3.6](https://www.mongodb.com/download-center" \t "_blank) onwards only. For utilizing the full potential [MongoDB 4.0+](https://www.mongodb.com/download-center" \t "_blank) is the most preferred option.

For creating replica sets from scratch, follow [this link](https://docs.mongodb.com/manual/tutorial/deploy-replica-set/), and for converting a standalone to a replica, follow [this link](https://docs.mongodb.com/manual/tutorial/convert-standalone-to-replica-set/).

# Opening a Change Stream

This is pretty straightforward. There are two ways of doing this in Nodejs, the first being a straightforward callback for an event.

const collection = db.collection(‘inventory’)  
const changeStream = collection.watch(**pipeline**)  
changeStream.on(‘change’, **next** => {  
 // process next document  
})

The second using async/await (in Nodejs)

const changeStreamIterator = collection.watch(**pipelie**)  
const **next** = await changeStreamIterator.next()

You can refer [here](https://docs.mongodb.com/manual/changeStreams/) for simple ways of doing the same in other languages

# Change Stream Event Document

The change stream output or the variable next in the previous section has the following structure. If you get a hand on this document structure, then you will always have a hundred degrees of freedom in working on these.

{  
 \_id : { <BSON Object> },  
 "**operationType**" : "<operation>",   
 "**fullDocument**" : { <document> },   
 "ns" : {  
 "db" : "<database>",  
 "coll" : "<collection"  
 },  
 "to" : {   
 "db" : "<database>",  
 "coll" : "<collection"  
 },  
 "documentKey" : { "\_id" : <value> },  
 "updateDescription" : {   
 "updatedFields" : { <document> },   
 "removedFields" : [ "<field>", ... ]  
 }  
 "clusterTime" : <Timestamp>,   
 "txnNumber" : <NumberLong>,   
 "lsid" : {  
 "id" : <UUID>,  
 "uid" : <BinData>  
 }  
}

The two most components are described as below

* **operationType** : Type of database operation, i.e. insert, delete, replace, update, drop, rename, dropDatabase, invalidate
* **fullDocument** : This is the entire document involved in the operation

The other components are practically quite useless, but listed as follows:

* **\_id** : Document containing metadata related to the operation, used when resuming a change stream
* **ns** : The namespace, i.e. the database and collection affected by the event
* **to** : The destination namespace, in case of rename
* **documentKey** : A document with just the **\_id** of the document by operationTypes insert, delete, replace and update
* **updateDescription** : A document describing the fields that were updated or removed by the update operation.
* **clusterTime** : Timestamp from oplog event
* **txnNumber** : Transaction identifier if a part of [multi-document](https://docs.mongodb.com/manual/core/transactions/) (which is a Mongo way to implement atomicity in multiple database operations)

# Pipelines

We have seen the change stream object, but we may choose to operate events only on selected documents (or stream objects) or modify some of the parameters in the object. Many of you might have used pipelines in [aggregation](https://docs.mongodb.com/manual/aggregation/), but even if you haven’t it really doesn’t matter. Pipelines are just a model for data aggregation and serve two purposes as mentioned already: selection and modification.

Pipelines are an array of stages, where the output of a stage is passed on to the next stage, and the output of that next stage is passed the succeeding stage, and so on. There are five types of stages in the pipeline way of handling change stream output:

## 1. $match

This is the most common and most useful and just lets you filter out events you might be interested in, rather than processing all events.

{ $match: { <query> } }

## 2. $project

This stage enables suppression of fields, addition, or even modification of the change stream output fields. More about it [here](https://docs.mongodb.com/manual/reference/operator/aggregation/project/#pipe._S_project).

{ $project: { <field>: <0 or 1>, <field>: <expression>} }

## 3. $addFields

Similar to **$project**, except that it just adds in new fields to the document, and lets all the existing fields be.

{ $addFields: { <newField>: <expression>} }

## 4. $replaceRoot

Useful for promoting an embedded (or new) document to the top level, replacing all other fields

{ $replaceRoot: { <newRoot>: <replacementDocument> } }

## 5. $redact

This one is a little advanced, it restricts the content of the change stream documents based on what information is stored in the documents. Presented below, is an indicative usage of this stage.

{ $redact: {  
 $cond: {  
 if: <**conditionalExpression**>,  
 then: "<**systemVariable**>",  
 else: "<**systemVariable**>"  
 }  
 }  
}

$redact supports three kinds of **systemVariable**

* **$$DESCEND**: Returns only current document level and excludes all embedded documents
* **$$PRUNE**: Excludes all fields at the current document level
* **$$KEEP**: Keeps all fields at the current document level

It is important to note that the current document is defined by the document level addressed by the **conditionalExpression.**

Let us now create a sample pipeline

const pipeline = [  
 { **$match**: { 'fullDocument.age': {$gte : 18}} },  
 { **$project**: {   
 'fullDocument.\_id : 0,   
 'fullDocument.fname': 1,   
 'fullDocument.lname': 1,   
 'fullDocument.sex' : 1  
 'fullDocument.title' : {  
 $conf : {  
 if: { $eq: [ "male", "$fullDocument.sex" ] },  
 then: "Mr",  
 else: "Ms"  
 }  
 }  
 } },  
 { **$addFields**: {   
 'fullDocument.name': {  
 $concat : ["$fullDocument.fname", " ", "fullDocument.lname"]  
 }   
 }}  
]

Instead of congesting your program with more code in order to wholly or selectively broadcast data or events to the external world, it will be a very simple affair to keep this functionality in a separate program altogether.

**package** com.example.demo.service;  
  
**import** com.mongodb.MongoClient;  
**import** com.mongodb.MongoClientURI;  
**import** com.mongodb.client.MongoCollection;  
**import** com.mongodb.client.MongoDatabase;  
**import** com.mongodb.client.model.Aggregates;  
**import** com.mongodb.client.model.Filters;  
**import** com.mongodb.client.model.changestream.ChangeStreamDocument;  
**import** org.bson.Document;  
**import** org.bson.conversions.Bson;  
**import** org.springframework.stereotype.Component;  
  
**import** java.util.ArrayList;  
**import** java.util.Collections;  
**import** java.util.List;  
  
@Component  
**public class** ChangeSrtreamService {  
  
 **public void** processChangeStreamWithnewCode() {  
 MongoClient mongoClient = **new** MongoClient(**new** MongoClientURI(**"mongodb://localhost:27017"**));  
  
*// Select the MongoDB database and collection to open the change stream against* MongoDatabase db = mongoClient.getDatabase(**"local"**);  
  
 MongoCollection<Document> collection = db.getCollection(**"users"**);  
  
*// Create $match pipeline stage.* List<Bson> pipeline = Collections.*singletonList*(Aggregates.*match*(  
 Filters.*in*(**"operationType"**, **"insert"**)));  
 List<Document> allInserts = **new** ArrayList<>();  
 **for** (ChangeStreamDocument<Document> doc : collection.watch(pipeline)) {  
 Document insertDoc = doc.getFullDocument();  
 System.***out***.println(insertDoc);  
 }  
 }  
}

**package** com.example.demo.controller;  
  
**import** com.mongodb.MongoClient;  
**import** com.mongodb.MongoClientURI;  
**import** com.mongodb.client.MongoCollection;  
**import** com.mongodb.client.MongoCursor;  
**import** com.mongodb.client.MongoDatabase;  
**import** com.mongodb.client.model.Aggregates;  
**import** com.mongodb.client.model.Filters;  
**import** com.mongodb.client.model.changestream.ChangeStreamDocument;  
**import** org.bson.Document;  
**import** org.bson.conversions.Bson;  
**import** org.springframework.beans.factory.annotation.Autowired;  
**import** org.springframework.data.mongodb.core.MongoTemplate;  
**import** org.springframework.stereotype.Component;  
  
**import** java.util.Arrays;  
**import** java.util.Collections;  
**import** java.util.List;  
  
@Component  
**public class** ChangeStremContl {  
  
 @Autowired  
 MongoTemplate **mongoTemplate**;  
 **public void** processChangeStream(){  
 MongoCollection<Document> users = **mongoTemplate**.getCollection(**"users"**);  
 System.***out***.println(**"=======>12344"**);  
 MongoCursor<ChangeStreamDocument<Document>> cursor = users.watch().iterator();  
 ChangeStreamDocument<Document> next = cursor.next();  
 System.***out***.println(**"=======>"**+next.getFullDocument());  
  
 }  
  
 **public void** processChangeStreamWithnewCode(){  
 MongoClient mongoClient = **new** MongoClient( **new** MongoClientURI(**"mongodb://localhost:27017"**));  
  
*// Select the MongoDB database and collection to open the change stream against* MongoDatabase db = mongoClient.getDatabase(**"local"**);  
  
 MongoCollection<Document> collection = db.getCollection(**"users"**);  
  
*// Create $match pipeline stage.* List<Bson> pipeline = Collections.*singletonList*(Aggregates.*match*(Filters.*or*(  
 Document.*parse*(**"{'fullDocument.name': 'string'}"**),  
 Filters.*in*(**"operationType"**, Arrays.*asList*(**"insert"**)))));  
  
*// Create the change stream cursor, passing the pipeline to the  
// collection.watch() method* MongoCursor<ChangeStreamDocument<Document>> iterator = collection.watch(pipeline).iterator();  
 System.***out***.println(**"popopopopopop====>"**+iterator.getServerAddress());  
 }  
}

**package** com.example.demo;  
  
**import** com.example.demo.controller.ChangeStremContl;  
**import** com.example.demo.service.ChangeSrtreamService;  
**import** org.springframework.beans.factory.annotation.Autowired;  
**import** org.springframework.boot.CommandLineRunner;  
**import** org.springframework.boot.SpringApplication;  
**import** org.springframework.boot.autoconfigure.SpringBootApplication;  
**import** org.springframework.data.mongodb.repository.config.EnableMongoRepositories;  
  
@SpringBootApplication  
**public class** DemoApplication **implements** CommandLineRunner {  
  
 @Autowired  
 ChangeStremContl **changeStremContl**;  
 @Autowired  
 ChangeSrtreamService **changeSrtreamService**;  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(DemoApplication.**class**, args);  
 }  
  
 @Override  
 **public void** run(String... args) **throws** Exception {  
 **changeSrtreamService**.processChangeStreamWithnewCode();  
 }  
}

**package** com.example.demo.service;  
  
**import** com.mongodb.MongoClient;  
**import** com.mongodb.MongoClientURI;  
**import** com.mongodb.client.MongoCollection;  
**import** com.mongodb.client.MongoDatabase;  
**import** com.mongodb.client.model.Aggregates;  
**import** com.mongodb.client.model.Filters;  
**import** com.mongodb.client.model.changestream.ChangeStreamDocument;  
**import** org.bson.Document;  
**import** org.bson.conversions.Bson;  
**import** org.springframework.stereotype.Component;  
  
**import** java.util.ArrayList;  
**import** java.util.Collections;  
**import** java.util.List;  
  
@Component  
**public class** ChangeSrtreamService {  
  
 **public void** processChangeStreamWithnewCode() {  
 MongoClient mongoClient = **new** MongoClient(**new** MongoClientURI(**"mongodb://localhost:27017"**));  
  
*// Select the MongoDB database and collection to open the change stream against* MongoDatabase db = mongoClient.getDatabase(**"local"**);  
  
 MongoCollection<Document> collection = db.getCollection(**"users"**);  
  
*// Create $match pipeline stage.* List<Bson> pipeline = Collections.*singletonList* (Aggregates.*match*(Filters.*and*(Document.*parse*(**"{'fullDocumentType.even':1}"**),Filters.*in*(**"operationType"**, **"insert"**))));  
 List<Document> allInserts = **new** ArrayList<>();  
 **for** (ChangeStreamDocument<Document> doc : collection.watch(pipeline)) {  
 Document insertDoc = doc.getFullDocument();  
 System.***out***.println(insertDoc);  
 }  
 }  
}

<https://docs.mongodb.com/manual/changeStreams/#use-cases>

cahnge streams allow applications to access real-time data changes without the complexity and risk of tailing the oplog

https://medium.com/riow/mongodb-data-collection-change-85b63d96ff76

Useful links:

https://docs.mongodb.com/manual/tutorial/convert-standalone-to-replica-set

https://docs.mongodb.com/manual/tutorial/change-streams-example

https://docs.mongodb.com/v3.6/tutorial/change-streams-example

http://plusnconsulting.com/post/MongoDB-Change-Streams

https://github.com/spring-projects/spring-data-mongodb/blob/master/src/main/asciidoc/reference/change-streams.adoc

https://medium.com/@mandalrajdeep/using-change-streams-in-mongodb-50ca3f44421a

https://www.bookstack.cn/read/mongodb-4.2-manual/3545c5026e465c9b.md

https://developpaper.com/first-experience-of-mongodb-change-stream/

replica set

<https://severalnines.com/database-blog/real-time-data-streaming-mongodb-change-streams>

**package** com.example.demo.service;  
  
**import** com.mongodb.MongoClient;  
**import** com.mongodb.MongoClientURI;  
**import** com.mongodb.client.MongoCollection;  
**import** com.mongodb.client.MongoCursor;  
**import** com.mongodb.client.MongoDatabase;  
**import** com.mongodb.client.model.Aggregates;  
**import** com.mongodb.client.model.Filters;  
**import** com.mongodb.client.model.changestream.ChangeStreamDocument;  
**import** org.bson.BsonDocument;  
**import** org.bson.Document;  
**import** org.bson.conversions.Bson;  
**import** org.springframework.stereotype.Component;  
  
**import** java.util.Arrays;  
**import** java.util.Collections;  
**import** java.util.List;  
  
  
@Component  
**public class** ChangeStreamInsert {  
  
  
 **public void** getInfo() {  
 MongoClient mongoClient = **new** MongoClient( **new** MongoClientURI(**"mongodb://localhost:27017"**));  
  
  
*// Select the MongoDB database and collection to open the change stream against* MongoDatabase db = mongoClient.getDatabase(**"local"**);  
  
 MongoCollection<Document> inventory = db.getCollection(**"users"**);  
  
 *// Create $match pipeline stage.* List<Bson> pipeline = Collections.*singletonList*(Aggregates.*match*(Filters.*or*(  
 Document.*parse*(**"{'fullDocument.name': 'string'}"**),  
 Filters.*in*(**"operationType"**, **"insert"**))));  
  
*// Create the change stream cursor, passing the pipeline to the  
// collection.watch() method* MongoCursor<ChangeStreamDocument<Document>> cursor1 = inventory.watch(pipeline).iterator();  
 ChangeStreamDocument<Document> next1 = cursor1.next();  
  
 BsonDocument resumeToken = next1.getResumeToken();  
  
  
 MongoCursor<ChangeStreamDocument<Document>> cursor2 = inventory.watch().resumeAfter(resumeToken).iterator();  
 ChangeStreamDocument<Document> next2 = cursor2.next();  
 System.***out***.println(**"===========>"** + next2.getFullDocument());  
  
 MongoCursor<ChangeStreamDocument<Document>> cursor3 = inventory.watch().startAfter(resumeToken).iterator();  
 ChangeStreamDocument<Document> next3 = cursor3.next();  
 System.***out***.println(**"===========555>"** + next3.getFullDocument());  
  
 }  
}