Table of Contents

1 Introduction

Samson is a batch and streaming platform for BigData, not an external storage solution, like Cassandra or any other NoSQL database.

Though, in the intent to make Samson a stand-alone product, there is a need for a means to store data, both for incoming data and for the results obtained, and also a mechanism to query this data.

The Samson team has researched a number of different external data storage solutions and also the idea to implement the external storage from scratch, perfectly fitting Samson needs.

This latter plan was soon rejected because of the huge effort it would take to implement such a project.

Finally the decision has fallen on using MongoDB, as it is a distributed and fail-safe solution.

The fact that recent clients push for MongoDB has influenced in this decision.

2 MongoDB

MongoDB is a scalable, high-performance, open source, document-oriented (NoSQL) database, implemented in C++. It manages collections of BSON documents that can be nested in complex hierarchies and still be easy to query and index, which allows many applications to store data in a natural way that matches their native data types and structures.

2.1 Executables

MongoDB comes with lots of components and utilities, the most important components being:

- mongod The database process.
- mongos Sharding controller.
- mongo The database shell (uses interactive JavaScript).

Using these three components, any MongoDB configuration can be setup and tested.

2.2 A Simple MongoDB Setup

The first tests were performed using one single MongoDB server. This is a very simple setup and all that is needed is to execute a 'mongod' process in the machine supposed to serve the database.

Apart from MongoDB, a simple test program, especially implemented for this test was used. This test program is able to run in two different modes, either inserting data to a collection in a database of MongoDB, or querying the database.

To run a single server database:

```
$ mkdir /data/db  # create the directory where the data is to be stored - configurable, of course ...
```

```
$ ./mongod
```

The mongo JavaScript shell connects to localhost and test database by default:

```
$ ./mongo
```

```
> help
```

The test program, to exercise MongoDB is executed in two instances, one for inserting data and the other to send queries to MongoDB.

2.3 Redundancy and Fail-over

In order to protect the MongoDB data against node failure, any number of secondary *mongod* processes can be used,standing by to take over the execution in case the primary *mongod* process fails. In MongoDB this concept is referred to as *Replica Sets*.

To pick a primary, voting is used and at least three participants are necessary to be able to vote. Each of these secondary *mongod* processes should of course be running in separate computers, and this naturally adds to the total cost of the platform. Fortunately, MongoDB supports a concept called

Arbiter for this, so there is no need to 'waste' *three* different nodes per replica set.

Arbiters are nodes in a replica set that only participate in elections: they don't have a copy of the data and will never become the primary node (or even a readable secondary). They are mainly useful for breaking ties during elections (e.g. if a set only has two members).

Depending on how severe data-loss would be, it is normally enough to have a single backup node, accompanied by an arbiter, which doesn't need a separate machine. Various arbiters could run on a single server, even if this would make the platform a bit more vulnerable.

[In this pilot, the Samson nodes can be used as Arbiters, there are enough Samson nodes to run a single Arbiter in each Samson node]

2.4 Indexes

Indexes enhance query performance, often dramatically. Indexes in MongoDB are conceptually similar to those in RDBMSes like MySQL. The selection of the field/fields for the indexation is very important, making sure the fields to be used in the most common queries are a part of the index key.

2.5 Sharding

To increase MongoDB performance, a collection can be distributed among various nodes (computers). This concept has received the name *sharding* and has the advantage that inserts and queries are distributed over a number of nodes and are thus faster.

2.6 Sharded MongoDB Setup Example

The second, more complete test of MongoDB performance were performed using the Samson platform, including the MongoDB module implemented for the Passive Location Pilot.

For the MongoDB setup, four computers were used, forming a sharded cluster while another four computers were running the Samson platform. In a fifth computer, the necessary Samson tools was executed to initiate the tests:

• delilah	the Samson console
• passiveLocationPush	a tool to inject fake but syntactically correct data to the pilot

The details of this setup is beyond the scope of this document.

3 Samson

Samson is a modular platform able to execute third-party modules as a part of the platform. These modules are compiled as separate shared libraries and Samson links to them in run-time. This mechanism has been used to implement Samson modules that connect Samson with MongoDB in a seamless manner.

The idea with these Samson-MongoDB modules is to not include any MongoDB administration into Samson, but merely use the databases and collections that a Samson-user inputs as parameters to the platform. All MongoDB administration and decision, such as use shards, will be done outside of Samson, using standard MongoDB tools.

3.1 Samson modules to extend functionality

3.1.1. XML parser module for Arcanum input data

The incoming data from Arcanum is in XML format and each record occupies around 600 bytes of plain ASCII. In order for Samson to extract the interesting part of these records, a parser must be used. This parser will be implemented as a Samson module and the incoming 600 bytes of ASCII data is converted to around 20 bytes of binary data.

This module is also the responsible of translating the Cell Tower ID to Latitude and Longitude coordinates.

3.1.2. Geolocation module

The identifier of the cell towers must be directly converted to Latitude and Longitude coordinates as cell tower identifiers may change over time. Thus, it is not enough to save the cell tower identifier for later translation to geolocation coordinates. A Samson module has been implemented to translate a cell tower identifier to these coordinates, according to input tables provided by XXX. These input tables may of course change over time, but no information is lost as each document in MongoDB saves the coordinates (in the moment of the event) as well as the cell tower identifier.

3.1.3. MongoDB module

Samson needs a module to connect to and insert documents in MongoDB.

The MongoDB databases to be used by Samson must be configured outside Samson, concerning creation of the databases, creation of the collections and their corresponding indexes and sharding keys.

For more details on the implementation of this module, please refer to subchapters 4.1and 4.2.

4 Passive Location Pilot

The hardware to be used for the pilot are 13 nodes of *DELL PowerEdge R510*:

- Dual CPU Intel Xeon E5620 (2.40GHz)
- 24Gb of RAM
- 2 Tb of Hard disk (SATA, 3.5-in, 7.2K RPM)
- Red Hat Enterprise Linux 6.0 (which may be changed for a newer version)
- Raid 5 (Raid 10 should be considered – that's what the good people at 10gen recommends)

The 13 nodes will be used:

- Eight nodes for MongoDB (probably in two different MongoDB clusters)
- Five nodes for the Samson cluster

The Samson setup is pretty straightforward, all five nodes will simple form the Samson cluster, while the MongoDB cluster can be chosen in a number of ways, the most important factor being whether using replica-sets or not.

4.1 MongoDB Setup

For the Passive Location pilot, eight nodes will be used for the MongoDB cluster.

There are two collections for the pilot:

• History	all incoming records
• Last Known Location	only the newest record for each user

The most queried collection will without doubt be the 'Last Known Location' for the users and it makes sense to separate this collection to a separate MongoDB cluster. Also, as this data is just a subset of the 'History' collection, there is no real need to replicate it.

In case replicated MongoDB servers are needed (to assure no loss of data), a sharded cluster of three backed-up nodes would be used for the 'History' database (a total of six nodes) and the remaining two nodes would be used as a sharded cluster for the 'Last Known Location' database.

As mentioned earlier, using only one secondary node per replica set, an arbiter is needed for each replica set. The Samson nodes could be used to run the arbiters.

The 'mongos' and config servers will run in the same nodes as the 'mongod' processes. 10Gen recommends using three config servers, so that is what we'll do ...

If replication isn't needed, all six remaining nodes would be used for the 'History' database.

The above statements should be considered a recommendation and extensive testing would be needed to assure the optimal configuration.

[*Perhaps more that two nodes should be used for 'Last Known Location'* ...]

4.2 Details on the Collections

Both collections will contain only one single type of documents and the documents will have five fields:

- UserId (EMEI)
- Cell Tower ID
- Timestamp
- Latitude
- Longitude

In the 'Last Known Location' collection, for the special field '_id', the UserId will be used. The same thing could be done for the 'Hostory' collection as well, bet there is a special reason for it in the latter case:

In MongoDB, there exists a 'conditional upsert' command, which will be used in the 'Last Known Location' collection in the following way (pseudo code):

```
db.LastKnownLocation.update({ _id:<ID>, T : { $lt: <TS> }}, { _id:<ID>, T:<TS>, C:1, X:1, Y:1 }, true)
```

The first parameter to update (*`{_id:<ID>, T : { $lt: <TS> }})`*), the condition, makes sure the update is only performed if the identity is the same and if the timestamp of the document in the collection is less than the timestamp of the document to replace it.

The second parameter is the new contents of the document and the third parameter ensures that the document is inserted if not found.

Now, trying to update a document of this kind when there is already another document with the same identifier but with a newer timestamp, the condition will evaluate to false (no such document found) and an attempt to insert it (thanks to 'true' in the third parameter) but as a document with the very same _id already exists, the insert will fail and that's just what we want.

4.3 Indexing and Sharding

The searches in the collections will probably always be based on UserId, so it is important to distribute the records of a single user over all the nodes in the shard, to gain performance in searches. If only UserId is used as shard key, all records of a single user would reside on a single node, so this is not an option.

As *sharding key*, a combination of *UserId* and *Timestamp* seems like the most optimal choice. Timestamp is included to avoid that all records of a single user are stored on a single MongoDB node. Distributing the records for a user has the advantage that searches on that UserId are distributed over a number of MongoDB nodes and are thus faster (running in parallel in a number of nodes). Also, a shard key like this should ensure a well-balanced database.

As index, just *UserId* will be used as key. Searches and updates in the 'Last Known Location' collection will be much faster thanks to the index.

[Indexes make inserts slower (around 100% slower) so it is a pity we cannot skip the index for the 'History' database, but as searches will be performed also in this collection, we don't have much of a choice …]

4.4 Splitting collections

The larger a collection gets, the longer time inserts and searches take. This is not difficult to understand … But, knowing this, an easy way to gain throughput would be to split the collections (at least the 'History' collection that will grow very big) in time frames, for example saving the data of each month as a separate collection (or even down to days). The obvious advantage is that throughput is gained but there is also a catch … The implementation gets a little more complicated, as the collection used for insertion much be changed when the month/day changes. This is really not a problem; it is implemented once in the Samson modules and that's it. A bigger problem is that to search the entire collection, more than one search must be performed, one for each of the 'time-frame-splitted' collections and the outputs have to be merged into a single output.

This 'collection splitting method' will be used *only if needed*, depending on the throughput that the platform gives.

4.5 Samson setup

The Samson cluster will run in five nodes, all of which will connect to MongoDB and do inserts/updates to both MongoDB collections, via the Samson modules specifically implemented for this purpose.

4.6 Arcanum input for Samson

The data that is injected in Samson from ARCANUM is in XML form and needs to be parsed in order to extract the relevant data, which for one single record consists of the following fields:

- UserId (EMEI)
- Cell tower ID – to be translated to Longitude and Latitude coordinates
- Timestamp

As mentioned, a special Samson module will be used to parse the input Arcanum data.

4.7 Expected outcome

The maximum desired accessibility for the passive location pilot with O2 UK is for the Samson platform to be able to store 40k location input events per second in MongoDB and at the same time being able to serve at least a hundred queries per second. Two separate database tables (in MongoDB the corresponding concept is 'collections') will be used for this purpose.

The first of these MongoDB collections (the 'History' collection) will hold the raw input (but stripped down to only contain the relevant data, of course), while the second ('Last Known Location') will hold only one record (the corresponding MongoDB concept is 'document') per user/cell phone number, describing the last known location of this subscriber.

5 Test Results

Two different setups has been used to extract sufficient test results for this 'proof-of-concept':

1. One simple setup with only one MongoDB node, used with an external tool to upload data to MongoDB and to at the same time querying that same collection. The idea behind this simple test is to verify how much load a single (non-sharded) MongoDB node is able to cope with, at the same time as querying is performed against that collection. This may not be 100% relevant as main part of the queries will be against the database/collection that contains only the 'last known location' of each of the users in the pilot. However, the test gives us an initial idea on how many shards we will need to cope with 40000 KVs/s and 100 qps.
2. The second setup is as close to the real pilot setup as possible - four Samson nodes and four MongoDB nodes and all data uploads are executed using the Samson-MongoDB module implemented for the passive location pilot. The MongoDB secondary nodes aren't even started, and we must bear in mind that the replication of the data will slow down the entire system.

5.1 *A first insert and lookup stress test in a single MongoDB node*

The throughput tests were performed using the simple setup, with a single 'mongod' process. We're assuming that each shard will add more or less this same throughput.

The table below describes the number of queries per second at different insertion rates and starting with different sizes of the collection that is at the same time both inserted into and queried.

For the first row of the table, the collection was first injected with 207 million key-value pairs and without any simultaneous inserts, a throughput of 56 queries per second was achieved. When injecting 40 new key-values per second, the responsiveness surprisingly went up to 58 queries per second. However, 40 queries per second is not very much and this strange result is probably just fluctuations ...

Table 1. Test Results For the Simple Setup

rate	**0**	**40**	**80**	**200**	**400**	**800**	**4000**	**8000**	**16000**
load									
207M	56	58	64	-	-	-	-	X	X
103M	2790	2760	2750	-	-	-	-	X	X
69M	2900	2900	3000	2800	1900	1450	1050	X	X
52M	3200	3000	2900	2900	2700	2700	1500	X	X
41M	3000	3000	2900	3000	3150	3150	2950	2700	1700
21M	3000	3050	3150	3150	2950	3000	2800	2500	2250
10M	3000	3050	3050	3050	3100	3150	3000	2850	2100
2M	3150	3150	3180	3100	3050	3125	2900	2750	2400

The initial value of 207 million key values is taken from the total number of key-values that the database would contain after storing 40 events per second, eight hours a day, during six months:

40 eps * 3600 secs * 8 hours * 30 days * 6 months == 207 million key-value pairs

We have found a maximum throughput of a little more than 3000 queries per second, and this depends on the round-trip time of a TCP/IP message. In order to do a query, a message is sent to the mongod process and the response is waited for. 3000 queries per second implies that one query takes around 0.3 milliseconds and that's about as good as it gets, with Gigabit Ethernet, which is what was used during these tests.

5.2 Test results for Samson/MongoDB integration

[Test results of only inserts with 4 Samson nodes and a mongo shard of 3 nodes]

```
upload 100 rps:         1650 qps
upload 1000 rps:        1650/qps
upload 10000 rps:       1400/qps
upload 20000 rps:       1200/qps
upload 50000 rps:       840/qps
upload 100000 rps:      1.5/qps  (records in MongoDB: 25.932.391)
upload 75000 rps:       20/qps   (records in MongoDB: 31.986.846)
upload 50000 rps:       900/qps  (records in MongoDB: 39.094.596)
```

This test setup includes four nodes running a Samson cluster, another three nodes running a MongoDB cluster for the 'History' collection, and the two last nodes for the 'Last Known Location' collection.

No Replica sets are used, because of lack of nodes. The performance will logically drop when adding replication

Using the Samson utility *passiveLocationPush*, pushing 10000 key-values to the Samson platform, working in streaming mode, the (fake) records are inserted and the Samson platform is without any load worth mentioning.

Testing with a 'whopping' 80000 key-values per second for inserts, the Samson load goes up to 16 used cores (ALL cores) but still, all data is perfectly inserted and no records are lost.

Now, executing simultaneously *passiveLocationPush* (with 40 - 160000 key-values per second) and an external tool to query the database, we get the following results of queries per second.

40 Kvs/s	=>	X qps
1000 Kvs/s	=>	X qps
4000 Kvs/s	=>	X qps
10000 Kvs/s	=>	X qps
20000 Kvs/s	=>	X qps

40000 Kvs/s	=>	X qps
80000 Kvs/s	=>	X qps
160000 Kvs/s	=>	X qps

6 Conclusions

A setup consisting of four Samson nodes and eight MongoDB nodes (half of the MongoDB nodes running as secondary MongoDB servers for redundancy) the entire platform is easily able to store 40000 KVs/s and simultaneously respond to 100 queries per second (on the last know location database). When querying the 'Last Known Location' database collection, we reach XXX qps, while inserting 40000 Kvs/s to the 'History' collection and the corresponding updates/inserts in the 'Last Known Location' collection.

7 Remarks

- Journaling will not be used.
  Journaling cannot be used (around ten times slower!). If redundancy is needed, another node should be used: REPLICA SETS (primary/secondary). For this, an extra machine is needed per shard ... But also, this gives complete redundancy and availability. Tests should be performed putting the journal on a separate hard drive. According to the nice people at 10gen, this slowdown we've seen is far from normal ...

- Sharding will be used.

  We will not have enough with just ONE machine for MongoDB. Sharding is supposedly buggy (at least in MongoDB release 1.8) but we need to try it out ... I have the list from Theo about what kind of bugs he encountered with sharding (with MongoDB 1.8).

- Collections will be split month by month or in even smaller chunks.

  The smaller the collections are, the faster searches we will have ...

- Found a roof of around 3000 queries per second.

  After consulting with Mattias of 10gen, I'm pretty sure this is simply the network latency. Tests should be done querying from localhost to make sure ...

- Avoid having collections bigger than 50M records (7.5 Gigabytes on disk)

- Indexing makes the insert time almost 100% slower.

  But, searches are impossible without indexes, so indexing must be used. If the index is kept on a separate (and faster) disk, there is lots of throughput to gain.