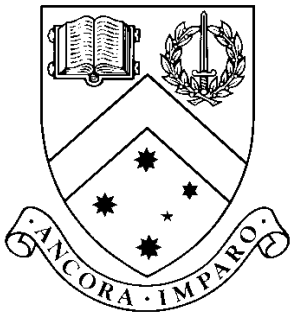

Monash University

Faculty of Information Technology

Computer Science Honours Reading Unit

Case Study of Inconsistent Railway Data With MongoDB

Author:
Jonathan Poltak Samosir

Supervisor:
Dr. Maria Indrawan-Santiago

May 25, 2015

Contents

1 Introduction

While relational database management systems (RDBMS) have been somewhat of a "go-to" solution for for a number of years for general data storage and management problems that many application developers face, we have noted a recent rise in the use of non-relational data management tools [8]. Most of these tools have traditionally fallen into the domain of big data analytics, with platforms such as the Hadoop ecosystem [1] being notably popular. However, outside of the domain of big data, looking more at general purpose data storage and management, what are now commonly referred to as "NoSQL" solutions are proving to be a popular solution.

NoSQL databases refer to those databases that are not built on top of the relational algebraic concepts, as laid out by Codd in 1970 [3], unlike the more commonly used RDBMS technologies, such as MySQL [2]. Being free of the strictness the relational model enforces on its data allows NoSQL databases to focus less on the overall structure of data, and more on factors such as scalability and performance [7].

While the relational model is a good fit for many data problems, its strictness in terms of flexibility of managing data eventually led to the introduction of the NoSQL model. The following characteristics can be given as a starting point for NoSQL databases in comparison to relational databases [6]:

- **Unstructured data support:** While the relational model would often force data to be stored in tabular formats, the NoSQL model does not force any kind of data schema.
- **Designed with distributed processing and horizontal scalability in mind:** Given the commoditisation of computer hardware in the last decade, support for horizontal scaling and processing among clusters is an important factor for adoption.
- **Less strict adherence to ACID principles:** While the relational model attempted to very much adhere to the transactional principles of data atomicity, consistency, isolation, and durability (ACID), this very much impacts performance in terms of distributed computing. Relaxing the strictness of adherence to these principles, allow many NoSQL databases to make the trade-off for higher performance.

Of course, these differences between the NoSQL model and relational model vary between each individual database technology's design, and trade-offs are often made depending on the goals and aims for that given database.

In this paper, we will look at the use of MongoDB [3], a popular NoSQL database solution, as a solution for a case study based upon a railway data problem using data from Monash University's Institute of Railway Technology (IRT). An overview of the case study in question will be given in §2. A small overview of the MongoDB database will be given in §3. Implementation and discussion details will be given in §4 and §5, respectively, before concluding in §6.

[1] https://hadoop.apache.org/
[2] https://www.mysql.com/
[3] https://www.mongodb.org/

2 Case Study Overview

The case study that is being looked at involves a project that has been worked at at the Monash University Institute of Railway technology (IRT). The project involves trains the operate in the Pilbara region of Western Australia, taking ore and minerals from loading points at mines to a specified unloading port. The data that is being dealt with comes from numerous categories of sensors being placed on certain specialised train cars to record data monitoring track and train car conditions [4, 5].

Data currently gets unloaded at sent back in large batches to remote servers once the car completes a trip and pulls back into port [10]. The project currently makes use of a RDBMS solution to manage data storage, however this solution is faced with many problems that currently require painful work-arounds. The most obvious of which involves unreliable data being received from sensors. For example, in the case of a damaged or failed sensor, reliable data cannot be guaranteed to be returned from such a sensor. Hence, data received by the remote server is often inconsistently structured, and thus the data has to go through a series of preprocessing work arounds to fit in within the strict schema that the RDBMS expects. As the only guaranteed consistency in the data that gets received is the timestamp and geocoordinates, the strictly structured relational model is not an appropriate solution.

The Monash IRT team are currently investigating further solutions in the NoSQL big data space, where they intend to replace the current system with an appropriately tested solution. While the scope of that project is much larger than what will be covered in this paper, we will look at the differences in what is possible when attempting to use a NoSQL database for the data storage and management.

For this paper, we will be looking at the possibility of using MongoDB, a popular NoSQL database that does away with the concept of the schemas and tables so commonly found in databases following the relational model. This is done with the expectation of better handling of the given inconsistent data.

3 MongoDB Overview

Note that most of the information relevant to MongoDB in this section is sourced from the official MongoDB manual. [2]

3.1 Data storage

MongoDB can be described as a documented-oriented NoSQL database, which does not rely on schemas or tables to structure its data [9]. Data within Mongo is stored using JSON-like structured documents, called BSON [1], which allows for strongly typed data fields and binary data. BSON documents are then stored within "collections" which reside in a given "database" [9]. To relate back to the relational model, these concepts within Mongo can be compared as shown in Table 1.

Table 1: MongoDB conceptual structures as they relate to the relational model concepts.

MongoDB	**Relational Model**
Database	Database
Collection	Table
BSON Document	Record/Row/Tuple
BSON Key	Column/Field
BSON Value	Value

While these concepts within Mongo can be related back to concepts within the relational model for ease of understanding, they are not exactly the same. For example, a table in the relational model is structured in such a way that all records within that table must adhere to. Conversely, in a Mongo collection, several differently structured BSON documents may coexist freely. BSON documents generally follow some sort of predefined structure for that collection, to maintain data structure sanity, for the user's sake, however none of this is enforced.

This sort of freedom in terms of structure allows for similarly structured documents in database collections, where differences may only be found in that certain documents may miss keys that are present in other documents. This is highly rational in the way that, depending on the use-case, data may not be present for certain fields all the time. Thus instead of placing null or default values for missing data, Mongo allows the user to just leave them out of the relevant document. Note that document structure in Mongo is also dynamic in the way that it can be changed at any time; BSON keys and values may be added or removed from existing documents at any time.

3.2 Data interaction

MongoDB allows a lot of the similar type functionality in queries and data manipulation that is also expected in relational databases. Generally interactions with data are performed through JavaScript functions on collections, or cursors (subsets of documents within a collection), allowing general querying operations through use of the `find()` function, and standard CRUD operations through the `create()`, `update()`, and `remove()` functions.

MongoDB officially provides support for many drivers allowing interaction with Mongo databases in different programming languages, including JavaScript, Python, and Java. As well as providing drivers, MongoDB also provides a default shell, called the *mongo shell*, which allows general JavaScript code to be written to interact with the standard MongoDB API and perform ad-hoc queries. No official graphical user interface exists to

interact with MongoDB, however there are numerous third party clients, which essentially wrap around mongo shell.

MongoDB also supports more advanced data manipulation operations through use of aggregation functions. Generally, aggregation functions are user defined operations to perform on documents in collections to return results. These include general aggregate operation models, such as MapReduce, and often involve grouping documents by given keys and various queries to limit the amount of documents to process.

3.3 Further database features

3.3.1 Lack of joins

MongoDB, like many other NoSQL databases, does not support the joins of multiple tables or, in MongoDB's case, documents, as is common in relational databases. Instead, MongoDB allows two main ways of relating different documents to each other. These are via referencing and nested documents. Nested documents are the main recommended way of join multiple documents, unless the documents have many-to-many relationships. To allow nested documents, a BSON value can store a whole other document, or arrays of documents. Example use-cases of where nested documents are appropriate include documents that have natural "child" documents that only it will ever need to reference. For example, a blog post document may have many nested comment documents as the comments are specific to a given blog post. For documents where many things may refer to it, it is recommended to use references via BSON values containing other document IDs.

3.3.2 Indexing

Indexing in MongoDB is supported in a similar ways that are present in relational databases. Mongo builds a B-Tree data structure on a particular collection's key, allowing all documents within that collection with that key to be accessed in $O(logn)$ time, as opposed to $O(n)$ time if unindexed. MongoDB also supports numerous types of indexes, such as single-key, multi-key, geospatial, and hashed indexes.

3.3.3 Replication

MongoDB allows replication and management of multiple copies of the same data for the purpose of increasing data redundancy and availability. Replication in Mongo is managed using a primary instance of data, with secondaries replicating the data in the primary using operation logs (*oplogs*) to mimic any operations on data that were performed in the primary. Replicas have fault tolerance in the way that if the primary fails, an election is started to promote any of the secondaries to become a primary. In general, a client will only interact with a primary, however operations can also be specified to be performed on specific secondaries. However, as replication data is asynchronously managed, any reads performed specifically on secondaries may not give up-to-date data.

3.3.4 Sharding

Sharding in MongoDB allows for horizontal scaling of a Mongo database, where a given database or collection is shared among different "shards", or multiple MongoDB instances, generally running on separate machines. Sharding is performed on an indexed document key, dividing documents containing that key based on either range or computed hashes. Ranged-based sharding is generally recommended where data is generally expected to be

accessed in some sort of sequence, while hash-based sharding offers a guaranteed random distribution of data across shards, and more balanced equally-sized shards.

4 Implementation

Implementation of the database for the given Monash IRT data was all performed using MongoDB version 3.0.3, the latest version as of 2015-05-18. Interaction with the MongoDB database was all performed using the mongo shell and corresponding official NodeJS MongoDB driver.

4.1 Overview of the data

The test data used for this case study was acquired from the Monash IRT team, and was an extract of real data they have received from the railway sensors. As only one set of data was given, simulated datasets based upon the extract have also been created to simulate the type of inconsistencies the IRT team are facing, including differently structured datasets and datasets with missing data (in the case of non-functional sensors).

The dataset received from the IRT team has the layout as shown in Table 2. It consists of 99 999 data points recorded at different times, and is received as raw data before being processed into a CSV format for easy manipulation and subsequent storage into their database.

Table 2: The different types of data received from the Monash University Institute of Railway Technology team.

Name	Data type	Description
Time	Floating point value	Time in seconds since the start of the test.
Speed	Floating point value	Speed in kilometres per hour.
LoadEmpty	-1, 0, 1	1 if load is present, 0 if empty, or -1 if unknown/in-port.
km	Floating point value	Elapsed kilometres travelled since the start of the test.
Lat	Floating point value from -90 to +90	Geographic latitude coordinate at given reading.
Lon	Floating point value from -180 to +180	Geographic longitude coordinate at given reading.
Track	Integer value from 1 to 213 841	Given track in which the train is located.
SND1	Floating point value	Spring nest deflection count in millimetres at car corner.
SND2	Floating point value	Spring nest deflection count in millimetres at car corner.
SND3	Floating point value	Spring nest deflection count in millimetres at car corner.
SND4	Floating point value	Spring nest deflection count in millimetres at car corner.
CouplerForce	Floating point value	The amount of force couple detected in the train car.
LateralAccel	Floating point value	The amount of lateral acceleration detected in the train car.
AccLeft	Floating point value	The amount of acceleration detected by the left sensor.
AccRight	Floating point value	The amount of acceleration detected by the right sensor.
BounceFront	Floating point value	The amount of bounce detected at the front of the car.
BounceRear	Floating point value	The amount of bounce detected at the rear of the car.
RockFront	Floating point value	The amount of rock detected at the front of the car.
RockRear	Floating point value	The amount of rock detected at the front of the car.

This dataset contains data from all the possible sensors. As previously stated, the datasets received are not always consistent, hence not all of these fields will always be present, or may be present and contain incorrect data readings. The only sensor data guaranteed to be received is the **Time**, **Lat**, and **Lon** sensor recordings.

4.2 Overview of data handling

None of the data returned from the sensors contain overly complicated structures, and hence MongoDB's BSON data storage format can easily handle all of the given data types. All given types can be handled using the BSON primitive types of Double and 32-bit Integer for floating point and integer data, respectively.

MongoDB comes with a built-in tool, called `mongoimport`, for the importing of multiple common data files, including CSV. Hence, the IRT data can be imported into a running MongoDB instance using the following command pattern:

```
mongoimport -d DATABASE_NAME -c COLLECTION_NAME --type csv --file FILE_NAME
    .csv --headerline --ignoreBlanks
```

The `-d` and `-c` flags enable the specifying of database and collection within that database in which to import the data into. The `--headerline` flag asks Mongo to use the CSV header to automatically generate BSON keys for each document. The `--ignoreBlanks` flag asks Mongo not to set empty CSV values to the empty string. Instead, these key-value pairs are left out of their respective documents. So each row in the CSV file will be converted to a MongoDB document, and stored in the specified database collection. By default, MongoDB tries to guess the data type for each value, with numbers represented as Doubles. Of course, storing every document as a flat container for different floating point values is not optimal, and pre-processing can be performed on Mongo to better structure the documents for later querying and processing.

4.3 Updating of data types

One of the first operations needed to be performed to restructure the data within MongoDB is to assign better data types fields where it is appropriate. With the given sensor data fields, the most obvious would be the **Track** and **LoadEmpty** fields, as the possible values they should be able to store are limited. With the **Track** field, the only values that should be present are integers representing the current track that the train car is located. Currently, the track numbers in the scope of the IRT project range from 1 to 213 841. These all fall into the 32-bit integer representation that BSON supports, hence rather than the default Double type, it more be more appropriate to change it to the 32-bit Integer type. This can be done iteratively with the following code:

```
var query = {Track: {$exists: true}};

db.train.find(query).forEach(function(doc) {
  // Cast Track to be Int32 type
  doc.Track = new NumberInt(doc.Track);

  // Write changes to MongoDB
  db.train.save(doc);
});
```

Listing 1: Performing an iterative operation on database collection “train”.

With the **LoadEmpty** field, the only possible values are -1, 0, and 1, each representing different states. While this would be a perfect case to use an enumerated type, MongoDB does not natively support enums. The two main choices for handling enums is by using either the String or 32-bit Integer types. As the String type allows more expressive values, and can be indexed to have integer-like performance, it would be most appropriate to change the field to be represented as a String type, mapping the existing numbers to appropriate strings. As the existing **LoadEmpty** value will need to be changed, this operation can be performed with the `db.collection.update()` function, as follows:

```
// Change LoadEmpty = 1 to 'loaded'
db.train.update(

  // query
  { "LoadEmpty" : 1 },

  // update
  { $set: {LoadEmpty: 'loaded'} },

  // options
  {
    "multi" : true,   // Update all documents
    "upsert" : false  // Insert a new document, if no existing document
        match the query
  }
```

```
);

// Change LoadEmpty = 0 to 'empty'
db.train.update(

  // query
  { "LoadEmpty" : 0 },

  // update
  { $set: {LoadEmpty: 'empty'} },

  // options
  {
    "multi" : true,   // Update all documents
    "upsert" : false  // Insert a new document, if no existing document
       match the query
  }
);

// Change LoadEmpty = -1 to 'unknown'
db.train.update(

  // query
  { "LoadEmpty" : -1 },

  // update
  { $set: {LoadEmpty: 'unknown'} },

  // options
  {
    "multi" : true,   // Update all documents
    "upsert" : false  // Insert a new document, if no existing document
       match the query
  }
);
```

Listing 2: Performing asynchronous `update()` operations on "train" collection.

Note that three different updates need to be performed on the collection, for each different value of **LoadEmpty**, as filtered out by the query argument. As each update will need to query the collection to retrieve a subset of documents to operate on, it would be wise to create a MongoDB index on the **LoadEmpty** key. This single-key index can be created using the following `db.collection.createIndex()` function:

```
// Create an ascending index on LoadEmpty field
db.train.createIndex({LoadEmpty: 1});
```

Listing 3: Creating an index on "LoadEmpty" key in "train" collection.

4.4 Restructuring of documents

As the documents now contain appropriately typed key-value pairs, further modifications to the documents can be performed to better structure the documents. By default, the CSV importer structures the documents generated from CSV rows as flat documents, with all row values being represented as a discrete key-value pair. The default flat structure can be seen as follows:

```
{
  "_id" : ObjectId("5561e94530f853aef8bbfd0c"),
  "Time" : 1094468487.0999999046325684,
  "Speed" : 4.4783997535999998,
  "LoadEmpty" : -1.0000000000000000,
```

```
  "km" : 5.0096998214999999,
  "Lat" : -20.6665750019999983,
  "Lon" : 117.1280816499999986,
  "Track" : 2.0000000000000000,
  "SND1" : 0.0118295024810000,
  "SND2" : -0.0173957574360000,
  "SND3" : -0.0295975576070000,
  "SND4" : -0.0327308348410000,
  "CouplerForce" : 23.1114991600000010,
  "LateralAccel" : 0.0033679292537000,
  "AccLeft" : 0.0000000000000000,
  "AccRight" : 0.0000000000000000,
  "BounceFront" : -0.0104506661800000,
  "BounceRear" : -0.0234966575210000,
  "RockFront" : 0.0445603373210000,
  "RockRear" : 0.0122018001710000
}
```

Listing 4: Original "flat" document structure.

From looking at and understanding the sensor data that is being dealt with, numerous sensors can be separated into distinct groups. For example, each of the **SND1**, **SND2**, **SND3**, and **SND4** values are from the same type of sensor located at different parts of the train car. Hence, rather than having four discrete key-pairs, these values can be grouped together in their own sub-document, or object, and pointed to by a new **SND** key. This is a more sane and appropriate way of structuring these values for later queries and operations.

Numerous other sensor values can be grouped together in this manner for more appropriate and saner structuring using sub-documents within the original BSON documents. Often nested documents are used where separate tables would be used in a relational modelling of the dataset, which also stretches to this case, where a separate table could possible hold all SND sensor values.

Further restructuring can be done to group the **Bounce**, **Rock**, and **Acc** values into sub-documents containing key-value pairs for each, finally ending up with the following document structure:

```
{
  "_id" : ObjectId("555d30ca33d9d7607772920f"),
  "Time" : 1094468487.0999999046325684,
  "Speed" : 4.4783997535999998,
  "LoadEmpty" : "unknown",
  "km" : 5.0096998214999999,
  "Track" : 2,
  "CouplerForce" : 23.1114991600000010,
  "LateralAccel" : 0.0033679292537000,
  "SND" : {
    "SND1" : 0.0118295024810000,
    "SND2" : -0.0173957574360000,
    "SND3" : -0.0295975576070000,
    "SND4" : -0.0327308348410000
  },
  "Loc" : {
    "Lat" : -20.6665750019999983,
    "Lon" : 117.1280816499999986
  },
  "Bounce" : {
    "Front" : -0.0104506661800000,
    "Rear" : -0.0234966575210000
  },
  "Rock" : {
    "Front" : 0.0445603373210000,
```

```
    "Rear" : 0.0122018001710000
  },
  "Acc" : {
    "Left" : 0.0000000000000000,
    "Right" : 0.0000000000000000
  }
}
```

Listing 5: Improved multi-layered document structure.

These restructurings could be performed with the following code, in mongo shell for example:

```
var updateDocsFunc = function(doc) {
  // Add location nested doc
  doc.Loc = {
    Lat: doc.Lat,
    Lon: doc.Lon
  };

  // Add Bounce nested doc
  doc.Bounce = {
    Front: doc.BounceFront,
    Rear: doc.BounceRear
  };

  // Add Rock nested doc
  doc.Rock = {
    Front: doc.RockFront,
    Rear: doc.RockRear
  };

  // Add SND nested doc
  doc.SND = {
    SND1: doc.SND1,
    SND2: doc.SND2,
    SND3: doc.SND3,
    SND4: doc.SND4
  };

  // Add Acc nested doc
  doc.Acc = {
    Left: doc.AccLeft,
    Right: doc.AccRight
  };

  // Delete old values
  delete doc.Lat;
  delete doc.Lon;
  delete doc.BounceFront;
  delete doc.BounceRear;
  delete doc.RockFront;
  delete doc.RockRear;
  delete doc.SND1;
  delete doc.SND2;
  delete doc.SND3;
  delete doc.SND4;
  delete doc.AccLeft;
  delete doc.AccRight;

  // Write changes
  db.train.save(doc);
};

// Iteratively perform operations on all records
```

```
db.train.find().snapshot().forEach(updateDocsFunc);
```

Listing 6: Restructuring of documents in "train" collection using iterative operation.

Finally, one more restructure could be performed on the existing document structure, which is grouping the **Lat** and **Lon** values into a single sub-document, which could be named **Loc**. This further enables future geospatial indexes being more easily created on the location data, for location-dependent queries and operations. This restructuring can be performed by the following code:

```
// Create a geospatial index on Loc nested document
db.train.createIndex({Loc: '2d'});
```

4.5 Handling missing values

For the test datasets missing data for particular sensors, MongoDB handles these cases. The datasets include CSV files with different value layouts, and often missing values in random rows, apart from the time and location fields as they are guaranteed.

For test datasets that are organised in different orders of fields, as Mongo's inner data storage format, BSON, does not have any ordering of its keys, documents are still generated with the appropriate data. For datasets with missing values in certain rows, the BSON document that is generated simply does not contain a key-value pair for the missed attribute. This is opposed to setting the key to a null value, or default value. Of course, a user may generate keys for the missing values and point them to null or default values, however there is no advantage to the extra processing.

When operations are performed on multiple documents in Mongo, if the documents contain a different amount of keys, Mongo effectively ignores this and continues to perform the operation on all documents. Generally, specific keys and values are passed to operations such as these, hence if the key used does not exist within certain documents, those documents simply get ignored from the operation. For example, if a collection contains 100 documents, where only four of those documents contain the **isAdmin** key, and an operation is called that depends on the key **isAdmin**, only those four documents within the collection will be operated on. This allows completely differently structured documents to be contained within the same collection (however, in a well-designed Mongo database, multiple collections would be used).

Hence, in the IRT case study, if certain sensors break down during operation and are unable to send back data, MongoDB will still allow queries and operations to be performed on the overall sensor data. However, if any operations depend on a particular sensor's data, such as the operations performed in the pre-processing of the data in §4.3, the documents missing those sensor's data will simply be left out of the operation. This avoids errors that are currently encountered with the currently set default values for broken sensors. For instance, if a sensor is broken and needs to be used in an operation, a lot less documents will be needed to be involved in the query, increasing query performance. Furthermore, any reducing operations performed on the datasets, will not be subject to involving unneeded default values in their calculations.

4.6 Filtering bad data

Often, the sensor readings received by the IRT team contains noisy data, defined by impossible values. For example, often the kilometre tracking sensor can return distance values consisting of negative numbers. These are deemed to be bad readings, and thus should not be included in any queries that are performed on the data. Hence, this data needs to be excluded from the overall dataset in the MongoDB collection.

To exclude such data, it is possible to use MongoDB's provided `update()` function, which allows the selection of noisy data based on a specified predicate. After data has been selected, `update()` allows the use of the `$unset` operator to remove specified fields from documents. This allows the other sensor readings in the same document to persist in the database, while only removing those readings specified as noisy by the predicate.

This is performed through the following code:

```
// Unset the bad km values (those with values less than 0)
db.train.update(
  { km: {$lt: 0} },       // predicate
  { $unset: {km: 1} },    // operation to perform
  { multi: true }         // option to specify operation on _all_ matching
      documents
);

// Unset all bad speed values (those with values less than 0)
db.train.update(
  { speed: {$lt: 0} },
  { $unset: {speed: 1} },
  { multi: true }
);

// Unset all bad LoadEmpty values (those not in valid enum values)
db.train.update(
  { LoadEmpty: {$not: {$in: ['unknown', 'empty', 'loaded']} } },
  { $unset: {LoadEmpty: 1} },
  { multi: true }
);

// Unset all bad Track values (those not in range of [0,213841])
db.train.update(
  { $or: [ {Track: {$lt: 0}}, {Track: {$gt: 213841}} ] },
  { $unset: {Track: 1} },
  { multi: true }
);
```

Listing 7: MongoDB `update()` functions to remove data deemed to be noisy from existing documents.

4.7 Ad-hoc adding of additional sensors

In the case of the IRT project, it is very possible that additional sensors could be added at any future stage. For a relational database, this would require modification to the table schemas, and possible restructuring operations to be performed on existing tables to set values in the columns for the additional sensors. In MongoDB, as documents are schema-less, new datasets containing new sensor's data can be serialised as BSON and inserted into existing collections at will, without requiring any modifications to the underlying database.

As stated previously, differently structured documents can co-exist in the same collection. Hence, adding datasets with additional sensors will simply result in documents being inserted that contain a different number of fields than those documents that were previously stored in that collection.

For testing, an additional dataset has been created, based off the original dataset, with the addition of two extra sensor fields: **Temp** and **Weight**. Documents with the addition of these new sensors simply have two additional key-value pairs that are not present in those documents consisting of data from the original dataset. Any queries performed that depend on either the **Temp** or **Weight** keys will simply only use those documents where such values exist for these keys, thus ignoring documents from the original dataset.

An example document containing these new fields is shown as follows:

```
{
  "_id" : ObjectId("556291145946aa793572b39a"),
  "Time" : 1.09447e+09,
  "Speed" : 4.4639997480000000,
  "LoadEmpty" : "unknown",
  "km" : 5.0096998220000000,
  "Track" : 2,
  "CouplerForce" : 23.1613006199999987,
  "Weight" : 2380,    // new pair
  "Temp" : 77,        // new pair
  "Loc" : {
    "Lat" : -20.6665759599999994,
    "Lon" : 117.1280811000000028
  },
  "Bounce" : {
    "Front" : -0.0303000000000000,
    "Rear" : -0.0029600000000000
  },
  "Rock" : {
    "Front" : 0.0383000000000000,
    "Rear" : 0.0062500000000000
  },
  "SND" : {
    "SND1" : -0.0111000000000000,
    "SND2" : 0.0001660000000000,
    "SND3" : undefined,
    "SND4" : -0.0494000000000000
  },
  "Acc" : {
    "Left" : 0.0000000000000000,
    "Right" : 0.0000000000000000
  }
}
```

Listing 8: Example of a document with extra sensor fields that do not exist in original dataset.

5 Discussion

6 Conclusion

References

[1] bsonspec.org/spec.html. `http://bsonspec.org/spec.html`. (Visited on 2015-05-13).

[2] The mongodb 3.0 manual — mongodb manual 3.0.3. `http://docs.mongodb.org/manual/`. (Visited on 2015-05-13).

[3] Codd, E. F. A relational model of data for large shared data banks. *Communications of the ACM 13*, 6 (1970), 377–387.

[4] Darby, M., Alvarez, E., McLeod, J., Tew, G., and Crew, G. The development of an instrumented wagon for continuously monitoring track condition. In *AusRAIL PLUS 2003, 17-19 November 2003, Sydney, NSW, Australia* (2003).

[5] Darby, M., Alvarez, E., McLeod, J., Tew, G., Crew, G., et al. Track condition monitoring: the next generation. In *Proceedings of 9th International Heavy Haul Association Conference* (2005), vol. 1, pp. 1–1.

[6] Indrawan-Santiago, M. Database research: Are we at a crossroad? reflection on nosql. In *Network-Based Information Systems (NBiS), 2012 15th International Conference on* (2012), IEEE, pp. 45–51.

[7] Leavitt, N. Will nosql databases live up to their promise? *Computer 43*, 2 (2010), 12–14.

[8] Padhy, R. P., Patra, M. R., and Satapathy, S. C. Rdbms to nosql: reviewing some next-generation non-relational databases. *International Journal of Advanced Engineering Science and Technologies 11*, 1 (2011), 15–30.

[9] Parker, Z., Poe, S., and Vrbsky, S. V. Comparing nosql mongodb to an sql db. In *Proceedings of the 51st ACM Southeast Conference* (2013), ACM, p. 5.

[10] Thomas, S., Hardie, G., and Thompson, C. Taking the guesswork out of speed restriction. In *CORE 2012: Global Perspectives; Conference on railway engineering, 10-12 September 2012, Brisbane, Australia* (2012), Engineers Australia, p. 707.