IT 14 021
Examensarbete 30 hp
April 2014

Scalable Persisting and Querying of Streaming Data by Utilizing a NoSQL Data Store

Khalid Mahmood

Institutionen för informationsteknologi
Department of Information Technology

Abstract

Scalable Persisting and Querying of Streaming Data by Utilizing a NoSQL Data Store

Khalid Mahmood

Relational databases provide technology for scalable queries over persistent data. In many application scenarios a problem with conventional relational database technology is that loading large data logs produced at high rates into a database management system (DBMS) may not be fast enough, because of the high cost of indexing and converting data during loading. As an alternative a modern indexed parallel NoSQL data store, such as MongoDB, can be utilized. In this work, MongoDB was investigated for the performance of loading, indexing, and analyzing data logs of sensor readings. To investigate the trade-offs with the approach compared to relational database technology, a benchmark of log files from an industrial application was used for performance evaluation. For scalable query performance indexing is required. The evaluation covers both the loading time for the log files and the execution time of basic queries over loaded log data with and without indexes. As a comparison, we investigated the performance of using a popular open source relational DBMS and a DBMS from a major commercial vendor. The implementation, called AMI (Amos Mongo Interface), provides an interface between MongoDB and an extensible main-memory DBMS, Amos II, where different kinds of back-end storage managers and DBMSs can be interfaced. AMI enables general on-line analyzes through queries of data streams persisted in MongoDB as a back-end data store. It furthermore enables integration of NoSQL and SQL databases through queries to Amos II. The performance investigation used AMI to analyze the performance of MongoDB, while the relational DBMSs were analyzed by utilizing the existing relational DBMS interfaces of Amos II.

Handledare: Tore Risch
Ämnesgranskare: Tore Risch
Examinator: Ivan Christoff
IT 14 021
Sponsor: This work is supported by the Smart Vortex EU project.

Tryckt av: Reprocentralen ITC

Acknowledgements

First of all, I would like to thank my grandfather, Abul Hossain - my lifetime role-model who gives me inspiration in every step of my life to overcome challenges.

I am grateful to Swedish Institute for granting me Scholarship to study and research for two years in this wonderful country of Sweden.

Many thanks to my aunt, Jasmin Jahan, my parents and wife. I am grateful for everything that you gave me.

Finally, I would like to thank my supervisor Tore Risch at Department of Information Technology at Uppsala University for his continuous guidance to fulfill my research work.

Khalid Mahmood

April 10, 2014, Uppsala, Sweden

TABLE OF CONTENTS

Acronyms

ACID	Atomicity, Consistency, Isolation, Durability
AMI	Amos Mongo Interface
AMOS	Active Mediator Object System
AmosQL	Active Mediator Object System Query Language
ANSI	American National Standards Institute
API	Application Programming Interface
BSON	Binary JSON
CAP	Consistency Availability Partition-tolerance
CSV	Comma Separated Values
DBMS	Database Management System
DLL	Dynamic Link Library
JDBC	Java Database Connectivity
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
NoSQL	Not-only SQL
OID	Object Identifier
RDBMS	Relational Database Management System
SQL	Structured Query Language

1 Introduction

Relational databases are commonly used for large-scale analyses of historical data. Before data can be analyzed it has to be loaded into the database and indexed. If the data volume is high the applications require high performance bulk loading of data into the database. However, the load time for relational database may be time consuming. NoSQL data stores have been proposed [1] as a possible alternative approach to traditional relational databases for large scale data analysis. For providing high performance update, this type of database system generally sacrifices consistency by providing so called *eventual consistency* compare to ACID transactions of regular DBMSs. Unlike NoSQL data stores relational databases provide advanced query languages for the analytics and indexing is a major factor in providing scalable performance. Relational databases have a performance advantage compared to data stores that do not provide indexing to speed up the analytical task [2]. However, there are some NoSQL data stores, such as MongoDB [3], that provide a simple query language that uses indexing (also secondary B-tree indexes) to speed up data access and analytics.

The purpose of the project is to investigate whether a state-of-the-art NoSQL data store is suitable for storing and analyzing large scale numerical data logs compare to relational databases. The performance of three different back-end data stores for persisting and analyzing such data logs is investigated:

1 MongoDB is the leading NoSQL data store [4]. As MongoDB provides both indexing, a well-defined C interface [5], and a query language, it seems to be a good alternative to relational databases. It is investigated how well MongoDB enables high performance archival of numerical data logs and efficient subsequent log analytics queries.

2 As the most popular relational database, we investigate how well MySQL performs for loading and analyzing numerical data logs.

3 We furthermore compare the performance with that of a major commercial relational database vendor (DB-C).

We compared all three systems in terms of load-time, time to do analytics for a typical set of analytical queries, and the resource allocation for storing data logs. Our results revealed the trade-offs between loading and analyzing of log data for both kinds of systems. We discuss the cause of the performance differences and provide some issues that future system should consider when utilizing MongoDB as back-end storage for persisting and analyzing historical log data.

The implementation utilizes an extensible main-memory database system, Amos II [6], to which the three different data managers were interfaced as back-ends. In particular the *Amos-Mongo Interface (AMI)*, which is crucial in this project, was developed by utilizing the Amos II C foreign function interface [7]. AMI provides general query capabilities as well as high performance access of data source by enabling Amos Query Language (AmosQL) statements to operate over MongoDB databases. The interfaces of the two relational DBMSs were already present in Amos II, which simplified the performance comparisons with those systems.

The main contributions of this project are:

- It provides a benchmark to evaluate the performance of loading and analyzing numerical data logs.
- It provides a comparison of the suitability of the three database systems for large-scale historical data analysis based on the benchmark.
- It provides a flexible interface to access MongoDB data stores where general queries and command using the AmosQL query language can be expressed. The interface includes scalable and high-performing functionality for bulk loading data records into MongoDB.
- It overcomes the limitations of MongoDB to perform complex queries including joins and numerical operators by enabling relationally complete AmosQL queries to MongoDB databases.

- It solves a major bug of MongoDB’s C Driver API for Windows and contributed to the community forum by providing a fully functional driver.

The rest of the report is structured as follows. Section 2 describes and compares relevant background technologies for the project. It includes a presentation of our real-world historical log application. Section 3 presents the architecture and implementation of the Amos-Mongo Interface (AMI). Section 4, presents the basic benchmark for testing the performance of loading and analyzing data. The different systems are investigated using the benchmark. The section ends with a discussion of the performance results. Finally, Section 5 summarizes the work and indicates future research.

2 Background

In this section, we will first present a real world application scenario that requires persisting and analyzing of historical log data. After that the so called NoSQL database systems and their properties are discussed and contrasted with traditional DBMSs. In particular using MongoDB as a back-end NoSQL data store for persisting and querying historical data is discussed. Detailed discussions related to NoSQL database systems can be found in a survey by Rick Cattell [1] and a comprehensive report in [8]. As some of the topics discussed in this section have been motivated by these sources, the interested readers are requested also to read the original discussions. Finally the main functionality of the Amos II system [6] used in the project are overviewed.

2.1 Application scenario

Our application involves analyzing data logs from industrial equipments. In the scenario, a factory operates some machines and each machine has several sensors that measure various physical properties like power consumption, pressure, temperature, etc. For each machine, the sensors generate logs of measurements, where each log has timestamp *ts*, machine identifier *m*, sensor identifier *s*, and a measured value *mv*. Each measured value *mv* on machine *m* is associate with a *valid time* interval *bt* and *et* indicating the begin time and end time for *mv*, computed from the log time stamp *ts* [9]. The table (collection) *measures (<u>m, s, bt</u>, et, mv)* will contain the large volume of log data from many sensors on different machines. There is a composite key on *(m, s, bt)*.

These logs will be bulk loaded into MongoDB and two relational DBMSs (MySQL and DB-C) to compare the performance. Since the incoming sensor streams can be very voluminous it is important that the measurements can be loaded fast. After the log stream has been loaded into *measures,* the user can perform some query to analyze the status of the sensors and the corresponding values of *mv* to detect the anomalies of sensor readings. The performance of loading and analyzing logs will be investigated in Section 4. The rest of this section provides the necessary background.

2.2 NoSQL and relational DBMSs

In general, there are six key features of NoSQL data stores [1].

Horizontal partitioning: To achieve higher throughput in NoSQL data stores, the data can be partitioned over many different servers, whereas, for achieving high performance, traditionally, RDBMs are targeted towards improving more powerful hardware in a dedicated server. NoSQL systems like MongoDB support automatic sharding (fragmentation, partitioning) by distributing the data over many commodity servers.

Simple call level interface: In contrast to APIs such as JDBC to relational DBMSs, NoSQL data stores provide simple call level interfaces or protocols. In MongoDB, *BSON* (*JSON*-like documents) objects in binary format are used to store and exchange data, express queries and commands. MongoDB provides APIs in different programming languages to communicate with data servers

Weak constancy model: According to Eric Brewer's CAP theorem [10], a distribute system data store can only have at most two of three of the properties *consistency*, *availability* and *partition tolerance*. Based on this, NoSQL database systems give up the ACID transaction consistency of relational DBMSs to achieve the other two attributes. This is called *eventual consistency*. MongoDB does not provide global consistency among distributed data servers and *eventual consistency* is by default. As an option, atomic updates of a single MongoDB data server can be specified by the *findAndModify* command.

Replication: For providing redundancy and high availability, replication is used to distribute the same data over many servers. MongoDB provides master-slave replication over sharded data. Replication is asynchronous for higher performance; however, data loss can also occur in case of system failure.

Distributed index: For accessing the data from partitions, efficient use of distributed indexing is necessary. MongoDB can use either range based partitioning or hash based partitioning for distributed indexing.

Dynamic addition of attributes: In NoSQL, new attributes to data record can be dynamically added. MongoDB is a schema-less database system, which allows dynamic addition of attributes. In MongoDB, within a single collection, each record can be heterogeneous, while in a relational database table all records are uniform.

2.3 MongoDB

Being a NoSQL data store, MongoDB supports the above-mentioned key features. MongoDB has some extended features and different terminology compare to other DBMSs. Important features which can enhance our application are discussed here.

2.3.1 The MongoDB data model

Unlike *Memcached* [1], which is basic *NoSQL* key-value store that provides a main-memory cache of distributed objects, MongoDB is a *document* data store. A MongoDB database consists of a number of collections where each collection consists of a set of structured complex objects called *documents*, whereas a basic key-value store associates a void object with each key. A collection in MongoDB is a set of documents**,** which is similar to a table in RDBMs. However, MongoDB objects can have nested structures while a relational table row must be a record of atomic values. Unlike table rows, the attributes of each MongoDB document is not defined statically in the database schema, but are defined dynamically at runtime. Although a collection does not enforce a schema, documents within the same collection have similar purpose.

MongoDB represents objects in a *JSON*-like [11] binary representation called *BSON*. BSON supports the data types *Boolean*, *integer*, *float*, *date*, *string*, *binary,* and nested types as well as some MongoDB specific types [12]. Like JSON, a BSON object consists of atomic types, associative arrays, and sequences. The following is an example of a *BSON* document being a member of some collection:

```
{
  name: { first: 'John', last: 'McCarthy' },
  birth: new Date('Sep 04, 1927'),
  death: new Date('Dec 24, 2011'),
  contribs: [ 'Lisp', 'Artificial Intelligence', 'ALGOL' ],
  awards: [
            {
              award: 'Turing Award',
              year: 1971,
              by: 'ACM'
            },
            {
              award: 'Kyoto Prize',
              year: 1988,
              by: 'Inamori Foundation'
            },
            {
              award: 'National Medal of Science',
              year: 1990,
              by: 'National Science Foundation'
            }
          ]
}
```

***ObjecId*:**

In MongoDB, each object stored in a collection requires a unique *_id* field as key, which guarantees the uniqueness within the collection. This *_id* acts as a primary key which has a default B-Tree index defined on the field. If the client does not provide an *_id* field, the server will generate a 12-byte BSON object of type *ObjectId*, which combines following attributes:

- 4-bytes representing the seconds since the Unix epoch for the time when the object was created,
- a 3-bytes machine identifier,
- a 2-bytes process id, and
- a 3-bytes counter, starting with a random value.

The *ObjectId* *is* small and fast to generate [3]. Moreover, the ObjectId can also be generated in a client by using the client's driver APIs as machine identifier to ensure global uniqueness.

2.3.2 Dynamic query with index support

Like RDBMs, MongoDB supports dynamic queries with automatic use of indexes. It provides queries by field, range search, regular expression, as well as calls to user-defined JavaScript functions. The user can define indexes on object attributes and MongoDB will automatically exploit the indexes. MongoDB provide several kinds of indexes including single attribute index, compound index on multiple attributes, multi-dimensional indexes on arrays, geospatial indexes, and full text search indexes. The default B-Tree index on the primary key *_id* attribute can also be a compound index. This index could be utilized for representing the composite key of *measures()*. We have left utilizing this index for future work, since it would change the data representation and make queries more complicated.

2.3.3 Write concerns

MongoDB provides two different levels of *write concern*, which specifies whether inserts, updates, or deletes are guaranteed to be consistent or not [3]. With *unacknowledged* write concern, the write operation is not guaranteed to be persisted. In that case MongoDB does not acknowledge the receipt of the write operation in order to provide high performance. On the other hand, when consistency is required, *acknowledged* write concern can be used that confirms the receipt of the write operation, at the expense of a longer delay.

As a NoSQL data store, MongoDB also provides load balancing by automatic sharding and distributing data over servers. Replication can also be used for ensuring high availability with increased read performance. MapReduce can be utilized for applying complex analytics on all elements in collections in parallel. Aggregation is provided by GROUP BY functionality. MongoDB can also provide server side execution through JavaScript programs shipped to the servers.

2.4 MapReduce

A popular approach to large-scale data analysis is MapReduce [13], which provides a mechanism for parallel computations over data read from a distributed file system. With

MapReduce data is not loaded into a database before the analytics, so loading time is saved compared to relational DBMSs [2]. However, as scalable analyses of historical data often require efficient use of indexing, the absence of indexing in MapReduce (i.e. Apache Hadoop) can often provide significant performance degradation compared to relational databases for analytical queries [2]. Therefore, for scalable analytics and loading of historical log data, this approach might not be suitable alternative compare to relational DBMSs and NoSQL data stores. The purpose of this work is to investigate scalable loading, storage, and querying of numerical data logs, which is not directly provided by MapReduce.

2.5 Amos II

Amos II is an extensible main-memory DBMS that provides an object-oriented data model and a relationally complete functional query language, AmosQL [6]. In Amos II different back-end database systems can be queried by defining interfaces called *wrappers* [14]. A wrapper is an interface between the query processor of Amos II and the query processing capabilities of a particular kind of external data source.

The wrappers makes Amos II a *meditator* system that that can process and execute queries over data stored in different kinds of external data sources [15]. In particular, a general wrapper has been defined to enable transparent AmosQL queries to any relational database [6], which is used in the project to run the benchmark queries over MySQL and DB-C databases.

In the project an Amos II wrapper interface for MongoDB was developed. The interface is called AMI (Amos-Mongo Interface). It enables AmosQL queries that contain MongoDB queries as parameters represented by key-value pairs in Amos II. AMI uses the foreign function interface [7] of Amos II to access MongoDB data stores through the MongoDB C driver API [5]. The interface provides the necessary primitives for complex AmosQL queries over MongoDB collections for analyzing log files.

Although MongoDB has a dynamic queries capability with automatic use of indexes, it does not support relationally complete queries that join several collections and it does not provide numerical operators. Integrating MongoDB and Amos II provides relationally complete queries over MongoDB collection including numerical operators for log analysis.

Through AMI and Amos II, queries can be specified that combine data in MongoDB collections with data in other DBMSs, which enables integration of NoSQL and SQL databases. This enables to utilize several other kinds of data sources in addition to NoSQL and relational DBMSs for a single application. As this integration of heterogeneous data source poses numerous technical challenges, a wrapper-mediator approach such as Amos II for resolving these issues can be a well suited and viable solution [15].

2.5.1 The Amos II data model

The data model of AMOS II is an Object-Oriented (OO) extension of the DAPLEX functional data model [16]. The Amos II data model consists of three basic building blocks: *objects*, *types,* and *functions*. In this data model everything is an object, including types and functions, where types classify objects and functions define properties of objects. Objects can be classified into one or more *types*, which entail an object to be an *instance* of one or several types. The types are organized into a hierarchy where the type named O*bject* is the most general type.

In general, the representations of objects are of two types: *surrogates* and *literals*. Surrogate objects are represented by explicitly created and deleted *OIDs* (Object Identifiers) through AmosQL statements. Example of *surrogate* objects is real-world entities such as objects of type *Person*. By contrast, *literal* objects are self-described system maintained objects that do not have any explicit OID. Examples of this type of object are *numbers* and *strings*. Literal objects can also be a *collections* of other objects such as *vectors* (1-dimentional arrays of objects), *bags* (unordered collections of objects), and *records* (sets of key/value pairs). Records in Amos II correspond to BSON objects, which is the basis for the data model of MongoDB and extensively used in AMI.

Record Type:

The collection data type named *Record* represents dynamic associations of key-value pairs [6]. This is similar to *hash links* in Java, and *key-value pairs* or *BSON* objects in MongoDB. In the following manner a new *Record* can be instantiated in Amos II:

```
set :rec= { 'Greeting':      'Hello, I am Tore',
            'Email':         'Tore.Andersson@it.uu.se'
          }
```

Here, a record instance bound to the variable *:rec* is created that consists of two keys *Greeting* and *Email* with corresponding values 'Hello, I am Tore', and 'Tore.Andersson@it.uu.se', respectively. For developing an interface between Amos II and MongoDB, the efficient conversion between object of type *Record* and the corresponding MongoDB *BSON* objects is needed, which has been performed in this project.

Functions:

In Amos II, functions represent different kinds of properties of objects. Based on their implementations, the functions can be classified as *stored, derived, foreign,* or *procedural* functions. Except for procedural functions, Amos II functions are non-procedural without side effects [14].

Stored functions represent properties of object stored in the database, i.e. tables, for example:

```
create function age(Person)-> Integer
      as stored;
create function name(Person)-> Charstring
      as stored;
```

Here *age* and *name* are properties of objects of type *Person*, with type *Integer* and *Charstring,* respectively.

A *derived function* is defined in terms of an AmosQL query. A derived function is similar to a view in a relational database, but may be parameterized. For example:

```
create function age(Person p)->Integer as
       current_year() - born(p);
```

Here, the derived function *age()* is defined based on two other functions *current_year()* and *born()*. Derived functions are side effect free and the query optimizer is applied when they are defined.

A *foreign function* is implemented in an external programming language (C, Java, Python, or Lisp). Foreign functions are important for accessing external data stores and DBMSs. AMI is implemented using foreign functions in C [7] to access the MongoDB C driver API. Assume that we have an external hash table indexing strings implemented in Java. The following foreign function can be defined to get the string *v* for key *k*:

```
create function get_string(Charstring k)-> Charstring v
  as foreign "JAVA:Foreign/get_hash";
```

Here the foreign function *get_string()* is implemented as a Java method *get_hash* of the public Java class *Foreign*. The Java code is dynamically loaded when the function is defined. The Java Virtual Machine is interfaced with the Amos II kernel through the Java Native Interface to C [15].

A *procedural function* is defined using the procedural sub-language of AmosQL having side effects. The syntax of procedural functions is similar to *stored procedures* in SQL: 99 [17].

2.5.2 The query language AmosQL

The *select* statement provides a general and very flexible way of expressing AmosQL queries. The format of the select statement is as follows:

```
select <result>
      from <type extents>
      where <condition>;
```

An example of such a query is:

```
select name(p), age(p)
      from Person p
      where age(p)>34;
```

In this example, tuples with the properties *name* and *age* of those objects of type *Person* in the databases that are more than 34 years old will be returned.

Extensive query optimization is the key to execute a query efficiently. Naïve execution of the above query without a query optimizer, may lead to very inefficient execution. Thus, Amos II provides a query optimizer that transforms the query into an efficient execution strategy. The AMI interface between Amos II and MongoDB provides MongoDB specific primitives for executing queries to MongoDB databases.

2.5.3 Extensibility

Amos II is an extensible main memory database system. To extend Amos II for accessing any external data source like MongoDB, the Amos II foreign function interface has to be utilized. There are external interfaces between Amos II and several programming languages, such as ANSI C/C++, Java, Python, and Lisp [7], [18]. Although the Java interface is the most convenient way to write Amos II applications, for high performance and time critical application one should utilize the more advanced Amos II C interface [7]. In this project, as we aim to achieve high performance access to MongoDB databases, so the low level external C interface is utilized to extend the Amos II kernel.

For implementing foreign functions in C, the following steps of development are needed [7]:

- A C function *implementing* the foreign function has to be developed.
- A *binding* of the C function to a symbol in the Amos II database has to be defined in C.
- A *definition* of the foreign AmosQL function signature has to be defined in AmosQL.
- An optional *cost hint* to estimate the cost of executing the function can be defined.

A complete example of a AmosQL foreign function implemented in C has been provided in page 12-15 of [7]. Reader concerns about the detailed implementation of foreign functions are requested to follow this documentation.

2.6 Discussion

In this section, the properties of so called *NoSQL* databases were discussed. The general features of NoSQL data stores and some exclusive features of MongoDB and Amos II were discussed more specifically. To scale large scale data analysis, or in our case persisting and analyzing of historical log data, the features of MongoDB such as weak consistency model and dynamic query capability with support of indexes have been expected to enhance the performance of such applications. However, as one can also expect to utilize several other distributed data sources including NoSQL data stores and RDBMSs within a single application, a wrapper-mediator approach by extending the main memory database Amos II for accessing MongoDB was suggested.

MongoDB does not support numerical operators and complex queries that combine several collections (i.e. table) through joins. To save the programmer from having to implement such features (i.e. join), one of the key advantages of extending Amos II with MongoDB is to provide the ability to express powerful and relationally complete AmosQL queries over MongoDB databases.

AMI is implemented by a set of AmosQL foreign functions utilizing the foreign function C interface of Amos II. The implementation is described in the next section.

3 The Amos MongoDB Interface (AMI)

AMI is an interface between Amos II and the MongoDB DBMS. The purpose of this interface is to seamlessly integrate MongoDB with the Amos II extensible DBMS, which allows the AmosQL Query Language to operate over MongoDB databases.

3.1 Architecture

AMI utilizes the Amos II C interface [7] by defining a set of foreign functions in Amos II that can be called from queries and procedures. These foreign functions internally call the MongoDB client C Driver API [5]. Figure 3.1 illustrates the architecture of AMI having the following modules:

- The Amos II kernel provides general query processing functionality through the *Amos query processor*.
- The *Amos-Mongo wrapper* provides an interface for transparent and optimized queries to MongoDB data sources from the Amos query processor. It contains the *Amos-Mongo query processor* and the *Amos-Mongo interface*. This project implements the *Amos-Mongo interface,* which is as a set of foreign functions that calls the *MongoDB* client API, provided by the *C Driver.* The on-going Amos-Mongo query processor implements query optimization specialized for MongoDB. It will improve query performance by generating semantically equivalent queries for MongoDB data source for a given AmosQL query.
- The *MongoDB C Driver* 0.8.1 version has been used in this project to enable AMI to interact with MongoDB databases. The MongoDB C driver is developed by the vendor of MongoDB. Although the driver was in alpha stage of development, due to high performance and portability, it was used as a desirable driver. However, It has been found that the original driver had a major bug in Windows Socket communication, which has been fixed in this project and reported in the community forum [19]. The details are discussed in section 3.3.

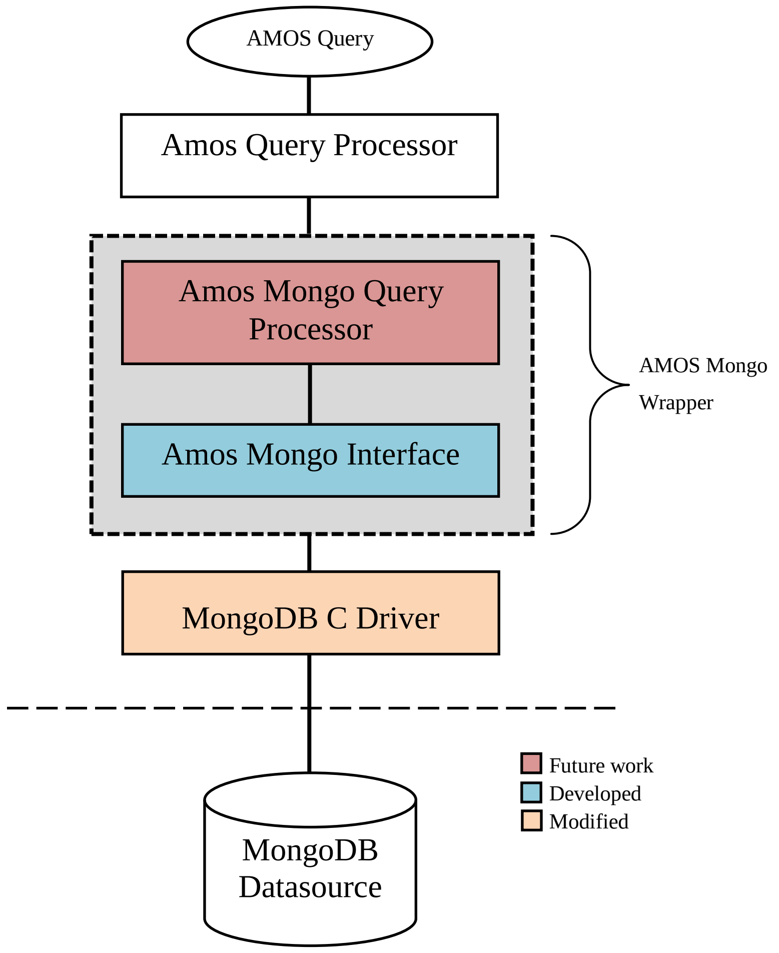


Figure 3.1. Architecture of AMI

3.2 AMI foreign functions

The interface functions to MongoDB are all implemented using the foreign function C interface of Amos II. It is using a lower level implementation compared to what has been discussed in section 2.5.3 and Amos II C Interfaces [7]. In this lower level API, unlike the function signature in [7], which has two arguments *a_callcontext cxt* and *a_tuple tpl*, only a single *a_callcontext cxt* argument is provided. Here *cxt* is an internal Amos II data structure for managing both call as well as efficiently representing *tpl* for actual arguments and results.

There are several kinds of foreign functions implemented in AMI:

- A foreign function that sends general MongoDB query expressions to the database server for execution.
- Foreign functions that add or delete objects to/from database collections. In particular a bulk insert function allows high-performance inserts of many objects.
- A foreign function that issues MongoDB database commands, e.g. to access meta-data or to create indexes. For example, as MongoDB creates a collection implicitly when it is referenced in a command the first time (i.e. when the *BSON* object is stored by an insert operation), a MongoDB database command can also be used for creating new collections explicitly [20].

A complete example related to these foreign functions has been provided in AMI tutorial section of 0.

3.2.1 Data type mappings

In general, the AMI client API represents data objects in local memory and sends it over a socket connection to the database server. In MongoDB, BSON objects are used to represent data objects, express queries, and express commands. The MongoDB C Driver provides APIs to create, read, and destroy BSON objects [5]. Similarly, Amos II provides corresponding data types to represent the contents of BSON objects. In particular, the datatype *Record* represents dynamic associative arrays [6]. A number of API functions in C are provided to access and store key-value pair in records. Therefore, to seamlessly integrate Amos II query language with MongoDB data store, it is important to provide an interface for data type mapping between *BSON* and *Record* objects. AMI provides the following two functions for data type mapping between *Record* and *BSON*.:

```
record_to_bson(bindtype env, oidtype *rec, bson *b, int conn_no)

bson_data_to_record(bindtype env, const char *data)
```

The datatype *oidtype* is a general handle to any kind of Amos II *object* and *env* provides the binding context for error handling. The functions are recursive as both *BSON* and *Record* can have nested key-value pairs.

Currently, AMI provides mappings between integers, floating point numbers, strings, binary types, and 1D arrays.

3.2.2 MongoDB object identifiers

In every MongoDB BSON document, the *_id* field is required to guarantee the uniqueness within the collection. The value of *_id* can be provided as a conventional data type like integer, floating point number, or string, which is unique in the collection. If a new document has been inserted without the *_id* field, the MongoDB server automatically creates the *_id* field as a 12-byte BSON object of type *ObjectId* that a uniquely identifies the new object in the data store. The object identifier is generated based on timestamp, machine ID, process ID, and a process-local incremental counter.

Although the MongoDB server will automatically create the object identifier if a *BSON* object is inserted without *_id* attribute, the MongoDB C driver API for insertion (`mongo_insert()`) does not return this object identifier. To enable the Amos user to further reference the created object after insertion, AMI uses a MongoDB C Driver API function to generate a globally unique *12-byte BSON* object identifier in the client side and add it to inserted record before sending it to the server for physical insertion. This mechanism to generate the object identifier in the client is a very efficient solution as it eliminates the communication overhead of the object creation taken place in the server.

3.2.3 Connecting to data sources

Before using any AMI functions, it is required to connect the MongoDB data source to Amos II. AMI provides the following foreign function implementation in C to access the data source.

```
create function mongo_connect(Charstring host) -> Integer conn_no
  as foreign 'mongo_connect+-';
```

Here, the host name can be defined as an IP address where a MongoDB server is running on a default port. It is possible to connect multiple server instances. In the following example, a new connection is created to the MongoDB data source running in *localhost* (IP 127.0.0.1) and the connection number is assigned to the Amos II variable *:c*. This variable can further be referenced by other interface functions to communicate with the data source.

```
set :c = mongo_connect("127.0.0.1");
```

Similarly, the following foreign function is provided to close the connection with a data source:

```
create function mongo_disconnect(Number conn_no)-> Boolean status
  as foreign 'mongo_disConnect-+';
```

In the following example the connection that was created by *mongo_connect()* earlier, is disconnected by calling *mongo_disconnect()*:

```
mongo_disconnect(:c);
```

3.2.4 Inserting a single object

For inserting an object (*BSON* Object) into a MongoDB data store, AMI provides the following foreign function interface that takes a MongoDB *connection* number, *database* name, *collection* name, and *Record* as parameters:

```
create function mongo_add(Number conn_no, Charstring  database,
                          Charstring collection, Record r)
                        -> Literal id
  as foreign 'mongo_add----+';
```

mongo_add() inserts a record *r* into the specified collection and returns the MongoDB object identifier *id* for the inserted object. As MongoDB is a schema-less, it is quite flexible to insert a record into a data store collection. For example, if the database or collection does not exist, MongoDB will automatically create the database and/or collection specified in the *mongo_add()* call. The *mongo_add()* implementation will convert the Amos II record *r* into an equivalent *BSON* Object. If the *_id* field is not provided in *r* a unique *12-Byte BSON* object identifier will be generated in the client side and returned (see section 2.3.1). Otherwise, the user provided value of attribute *_id* in *r* is returned. In the following example, a record is inserted into a MongoDB data store.

```
mongo_add(:c, "tutorial", "person",
            {
                  "Name": "Ville",
                  "age" :   54
            }
        );
```

Here, the MongoDB connection has been bound to variable *:c* by a previous *mongo_connect()* call (see section 3.2.3). The *mongo_add()* call creates a new MongoDB object stored in the collection "*person"* in the database "*tutorial"*. Since no attribute *_id* is provided, *mongo_add()* will generate and return a new MongoDB object identifier representing the inserted record in the MongoDB data store before inserting the new object into the data store. In this example, a record with two attributes *Name* and *age* is inserted to the data source.

Records with arbitrary new attributes can be inserted without modifying the database schema as relational databases require. In the following example a record with two additional attributes *_id* and *email* are provided without any modification of the schema. Here the object identifier *_id* has the value *10*, which will be returned from *mongo_add()*.

```
mongo_add(:c, "tutorial", "person",
            {
                  "_id"   : 10,
                  "Name"  : "George",
                  "age"   :  27,
                  "email": "george.85@it.uu.se"
            }
        );
```

3.2.5 Bulk inserting objects

With the same level of flexibility as with single document inserts using *mongo_add()*, AMI provides the following foreign function for bulk inserting multiple records:

```
create function mongo_add_batch(Integer  conn_no, Charstring  database,
                                Charstring  collection,
                                Vector vr, Integer writeConcern)
                                -> Boolean status
      as foreign 'mongo_add_batch-----+';
```

Here, the first three parameters are the same as for *mongo_add()*. The fourth parameter *vr* is a vector of records to be bulk inserted. If the parameter *writeConcern*=1 it specifies that an acknowledgement is returned after each inserted object, otherwise the insertion is unacknowledged and asynchronous. The *mongo_add_batch()* function returns true if the insertion succeeds.

The following convenience functions encapsulate the different write concerns:

```
create function mongo_add_batch_ack(Integer  conn_no, Charstring  database,
                                    Charstring  collection, Vector vr)
                                  -> Boolean status
      as mongo_add_batch(conn_no, database, collection, vr, 1);

create function mongo_add_batch_unack(Integer conn_no, Charstring database,
                                      Charstring  collection, Vector vr)
                                    -> Boolean status
      as mongo_add_batch(conn_no, database, collection, vr, 0);
```

With *mongo_add_batch_ack()* MongoDB confirms the receipt of each write operation to catch network, duplicate key, and other errors, while with *mongo_add_batch_unack()* MongoDB does not acknowledge the write operations and no errors are caught. With *mongo_add_batch_unack()* higher insertion rates are expected than with *mongo_add_batch_ack()*. In the following example, a vector of records are batch inserted with *mongo_add_batch_unack():*

```
      mongo_add_batch_unack(:c, "tutorial", "person",
                  {
                        {"Name": "Carl", "City": "Uppsala"},
                        {"Name": "Eve"},
                        {"Name": "George", "Country": "Sweden"},
                        {"Name": "Olof", "age": 65}
                  });
```

Notice that Amos uses the notation {…} to form vectors (while BSON uses […]) and that each record may have different attributes within the same bulk insert.

3.2.6 MongoDB queries

AMI provides a flexible foreign function *mongo_query()* for sending queries to MongoDB for evaluation. Since queries to MongoDB are expressed in *BSON* format, the corresponding record structure is used in the *mongo_query()* as well. It has the following definition:

```
create function mongo_query(Number  conn_no, Charstring  database,
                            Charstring collection, Record q)
                          -> Bag of Record r
      as foreign 'mongo_query----+';
```

The parameters *connection*, *database*, and *collection* name identify the queried collection, while the parameter *q* specifies the query as a record. The result from the evaluation by the MongoDB server is a bag of records that matches the query *q*.

In the following example, all the objects member of collection *person* in database *tutorial* database having attribute *age*=27 will be returned:

```
      mongo_query(:c, "tutorial", "person", {"age": 27});
```

To return all the documents from a collection, an empty record can be used:

```
      mongo_query(:c, "tutorial", "person", {:});
```

Range queries can also be expressed. For example the following query returns all objects in the *person* collection with *age* greater than *50* and less than *60*:

```
      mongo_query(:c , "tutorial", "person",
                   {
                         "age": { "$gt": 50, "$lt": 60 }
                   });
```

A comprehensive SQL to MongoDB Mapping chart can be found in [21]. Some more query examples are provided in AMI tutorial section of 0.

3.2.7 Deleting objects

The AMI function for deleting the documents from a MongoDB database is similar to *mongo_query()*:

```
create function mongo_del(Number  conn_no, Charstring  database,
                          Charstring collection, Record q)
                        -> Boolean status
      as foreign 'mongo_del----+';
```

The function *mongo_del()* deletes objects in a collection matching a query *q*.

In the following example, all the documents in the collection *person* having *age*=27 will be removed (if any). The *Boolean* value *true* will be returned upon success.

```
mongo_del(:c, "tutorial", "person", {"age": 27});
```

3.2.8 Indexing

An index is a special data structure supporting efficient execution of queries in databases. MongoDB provides a number of different indexing structures. In particular B-Tree indexes are definable for any top level object attribute(s) of the objects in a collection. Without indexing MongoDB must scan all objects in a queried collection to match the query filter against each object. To reduce this inefficiency, AMI provides the following function to define an index on one or several attributes for the objects in a given collection.

```
create function mongo_createIndex(Number  conn_no, Charstring  database,
                                  Charstring  collection,
                                  Charstring indexName, Record spec)
                                  -> Boolean status
      as foreign 'mongo_createIndex-----+';
```

The parameter *indexName* specifies a name for the index and the parameter spec specifies the properties of the index, such as its attributes, the sort order of the index, and what kind of index structure is created. By default B-tree indexes are created, but MongoDB supports several other kinds of indexes as well. Indexes can be defined on either a single attributes or multiple attributes. For example, the following function call creates a B-tree index on a single attribute *age*:

```
mongo_createIndex(:c, "tutorial", "person", "ageIndex", {"age": 1});
```

The number *1* specifies that the B-tree index for *age* will be stored in ascending order. To order the index in descending order *-1* is specified.

MongoDB supports compound (composite) indexes, where a single index structure holds references to multiple attributes of the objects in a collection [3]. The following example creates a compound index on *Name* and *age*. Here, first the values of the *Name* attribute will be stored in ascending order and the values of *age* will be stored in descending order.

```
mongo_createIndex(:c, "tutorial", "person", "ageIndex",
                         {
                               "Name": 1,
                               "age": -1
                         }
                   );
```

The following function removes an index named *indexName* from a collection:

```
create function mongo_dropIndex(Integer  conn_no, Charstring  database,
                                Charstring  collection,
                                Charstring indexName)
                                -> Record status
```

The function mongo_dropIndex() is defined using an API for issuing administrative commands on the database provided by *MongoDB C Driver* [5], to be explained next.

3.2.9 MongoDB database commands

MongoDB provides a set of *database commands* [20] that are issued on a data source. AMI provides the following function for issuing the MongoDB database commands *cmd*:

```
create function mongo_run_cmd(Integer  conn_no, Charstring  database,
                              Record cmd)
                            -> Record status
      as foreign 'mongo_run_cmd---+';
```

The database command *cmd* is represented as a record.

In the following example, *mongo_run_cmd()* is used to delete the index named *ageIndex* from collection *person*:

```
      mongo_run_cmd(conn_no, "tutorial",
                         {
                               "dropIndexes": "person",
                               "index"      : "ageIndex"
                         }
                   );
```

The syntax of a database command expressed as a record varies. For example, the attribute *dropIndexes* above specifies that an index in collection should be deleted and the attribute *index* specifies its name. As another example, *MongoDB C Driver* [5] implements *database user roles* [22] as database commands. All the database user roles can be expressed as records.

The function *mongo_dropIndex()* is defined as a procedural function in terms the *mongo_run_cmd()* as:

```
create function mongo_dropIndex(Integer  conn_no, Charstring  database,
                                Charstring  collection,
                                 Charstring indexName)
                               -> Record status
      as mongo_run_cmd(conn_no, database,
                   { "dropIndexes": collection, "index": indexName });
```

The following AMI functions are all defined using *mongo_run_cmd()*:

- *mongo_dropCollection()* - Removes the specified collection from the database.
- *mongo_dropDB()* - Removes the entire database.
- *mongo_dropIndex()* - Drop an index within a collection and database.
- *mongo_dropAllIndex()* - Drop all the indexes within a collection and database.
- *mongo_getPrevError()* - Returns status document containing all errors.
- *mongo_collStats()* - Reports storage utilization statics for a specified collection.
- *indexStats()* - Collect and aggregates statistics on all indexes.
- *mongo_dbStats()* - Reports storage utilization statistics for the specified database.
- *mongo_collNameSpaces()* - Get all the collection identifiers in a database.
- *mongo_indexNameSpaces()* - Get all the index identifiers in a database.

The detailed function signatures and examples of their usage can be found in 0.

3.3 MongoDB C driver issues

The MongoDB C Driver [5] provides a C-based client API for MongoDB database servers. There are several other client driver implementations for different other programming languages, e.g. Java, C#, or Python. To ensure high performance the C driver API was chosen in implementing AMI.

The current version 0.8.1 of the C driver is still in alpha stage and the project found that the driver had a major bug when being used under Windows. After building the driver with the recommended Python build utility *SCons* [23] and the *Microsoft*

Visual Studio 2010 compiler, the driver was unable to connect to the MongoDB server using the MongoDB C Driver function:

```
int mongo_client( mongo *conn , const char *host, int port );
```

It turned out that `conn->err` returned the error `MONGO_SOCKET_ERROR`, and the problem was found to be the absence of windows socket initialization, because the MongoDB C driver that was built for *Windows 7* environment, never initiated the *Winsock DLL* structure. We therefore added to the C driver code in `env.c` with proper Windows Socket initialization function *WSAStartup()*. The purpose of *WSAStartup()* is to allow an application or DLL to specify the version of Windows Sockets required and retrieve details of the specific Windows Sockets implementation. A DLL can only issue further Windows Sockets functions after successfully calling of *WSAStartup()* [24].

A Microsoft Visual Studio 2010 project with the proper implementation of the above mentioned windows socket initialization and modified source code of MongoDB C Driver 0.8.1 was developed in this project to build a functioning driver API. The detailed code as well as the settings of Visual Studio project is explained in 0. The problem description, solution, and the Microsoft Visual Studio 2010 project has been provided for the MongoDB bug report community forum [19].

It should be mentioned that the MongoDB C Driver 0.8.1 APIs can be successfully built and used under *Mac OS X 10.6 Snow Leopard* and *Scientific Linux for 64 bit* platform, without the above mentioned problem.

3.4 Discussion

In this section the architecture, implementation details as well as functionality of *Amos-Mongo Interface (AMI)* was described. In AMI the corrected *MongoDB C Driver 0.81* API is utilized to implement a set of foreign Amos II functions in C. AMI provides data type mappings between the associative arrays represented by *BSON* objects in MongoDB and *Record* objects in Amos II. As MongoDB uses BSON as the data storage and network transfer format for objects, this data-type mapping provides a flexible and convenient way of accessing MongoDB databases, expressing queries and commands from AmosQL, and extending the interface by defining AmosQL functions in terms of MongoDB database commands. The flexibility and performance of AMI provides a convenient and flexible solution for the ongoing development of a full-functional *Amos-Mongo Wrapper*. Finally, an executable tutorial as well as complete signatures of all functions in AMI is provided in 0**.**

4 Performance Evaluation

To evaluate NoSQL data stores compared with relational databases for storing and analyzing logged data streams (historical data), we compared the performance for data logs from real world industrial application of MongoDB, MySQL and a relational DBMS called DB-C from a major commercial vendor. Here, we compared these three systems in terms of load-time, time to do simple analytics, and the resource allocation. To perform analytics on persistent logs, we defined a benchmark consisting of a collection of basic tasks and queries required in analyzing numerical logs. For each task, we measured the query performance in terms of execution time for the systems. We also investigated the statistics about resource allocation (i.e. database and index size). Our results revealed the trade-offs between loading and analyzing data log for the systems. Although the process to load the data with indexing for scalable execution of queries took a lot of time in all systems, the observed performance for both MongoDB and DB-C was strikingly better compare to MySQL in both loading time and analytics. We have speculated about the cause of this dramatic performance differences and provide some insight issues that future system should consider when utilizing MongoDB as back-end storage for persisting and analyzing data logs.

4.1 Data set

The evaluation was made based on log files from a real-world industrial application in the Smart Vortex project [25]. The log measurements from time series for one kind of these time series was used in the performance evaluation. The chosen time series is plotted in Figure 4.1. It has approximately 111M measurements and occupied 6GB gigabytes.

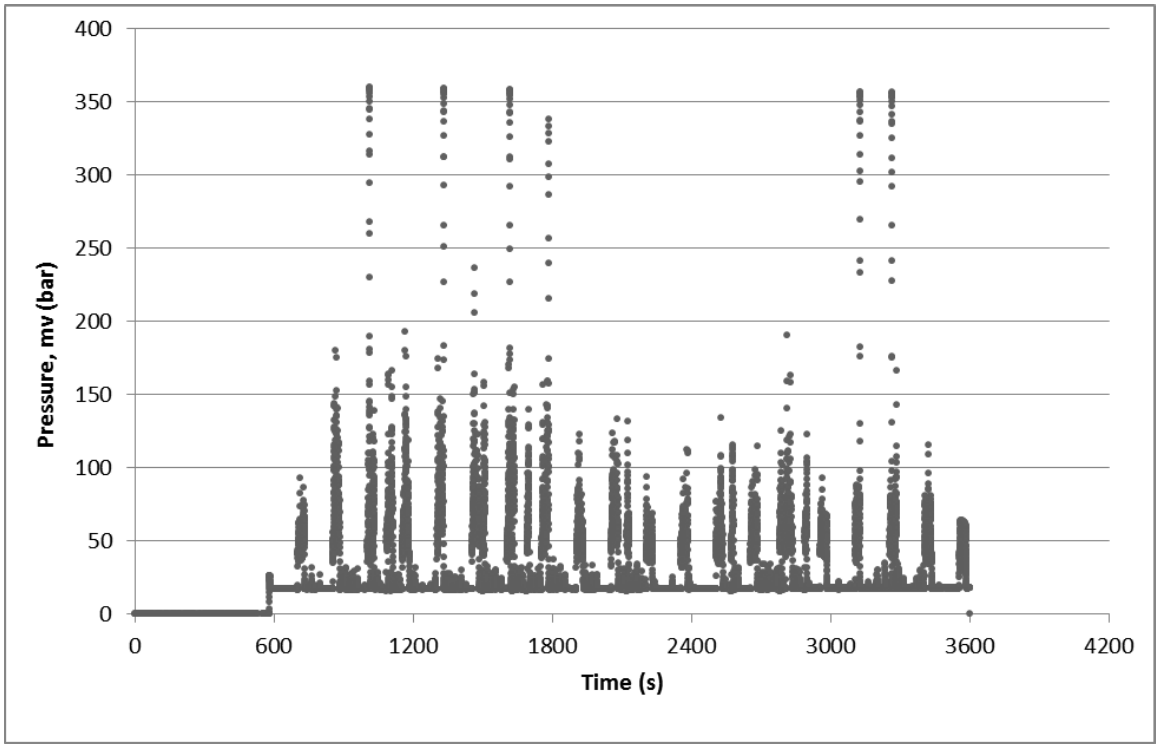


Figure 4.1. Pressure measurements of sensors for 1 hour

To investigate DBMS performance with growing raw data file size, parts of the raw log data file was loaded into the databases. The performance of load times, query times, and resource allocation was measured for the different DBMSs.

4.2 Benchmark queries

In this section, we define the tasks representing a set of queries that are basic to perform analytics over log data. There are many kinds of queries for analyzing log data, as discussed in [9]. Some of these queries require efficient evaluation of numerical expressions, which is supported by SQL but cannot be easily specified in a simple NoSQL data store or MongoDB. Therefore, we limit ourselves to those quires that are basic for log analytics and do not involve the use of complicated numerical operators or joins. Before executing the queries, the data has to be bulk loaded into the database.

4.2.1 Key lookup query Q1

This task involves finding measured value *mv* for a given machine *m*, sensor *s* and begin time, *bt*. The query in SQL or equivalent AmosQL by utilizing AMI is specified as follows:

```
SELECT * FROM measures va
WHERE va.m =?
AND va.s =? AND bt =?
```

```
mongo_query(:c, "LogDB", measures",
  {
     'm': ?, 's': ?, 'bt': ?
  }
)
```

In order to improve the performance of such a query, we need efficient use of indexing by B-trees or hash tables. In MySQL or DB-C, we index by defining a composite primary key on (*m, s, bt).* Since in MongoDB there is always a default B-tree index on attribute *_id*, we have to add a secondary compound index on (*m, s, bt).* All systems utilize B-tree indexes for performing such queries. In the later part of this section, we will provide several alternatives of bulk loading, indexing, and executing the key lookup task. Initially, the tasks will be performed without any index, which is expected to speed up the bulk loading but slows down the analytics. Then we will investigate two other alternatives by defining indexes. This is expected to slow down the bulk loading as it has to consider the update on indexes as well, while it will speed up the query execution task.

4.2.2 Basic analytical query Q2

This query involves finding anomaly of sensors by observing measured values, *mv*, deviating from an expected value. Here, the sensors with the measured value *mv* higher than the unexpected value is detected. Such query can be expressed in SQL, or equivalent AmosQL for MongoDB, as follows:

```
SELECT * FROM measures
WHERE mv>?
```

```
mongo_query(:c, "LogDB", "measures",
  {
    'mv': { $gt': x }
  }
)
```

Since the effectiveness of a secondary index is highly dependent on the selectivity, this query was executed for different query condition selectivities by providing the appropriate range of *mv*. To obtain different selectivities we measured the dependency between value of *mv* and actual selectivity of Q2, plotted in Figure 4.2

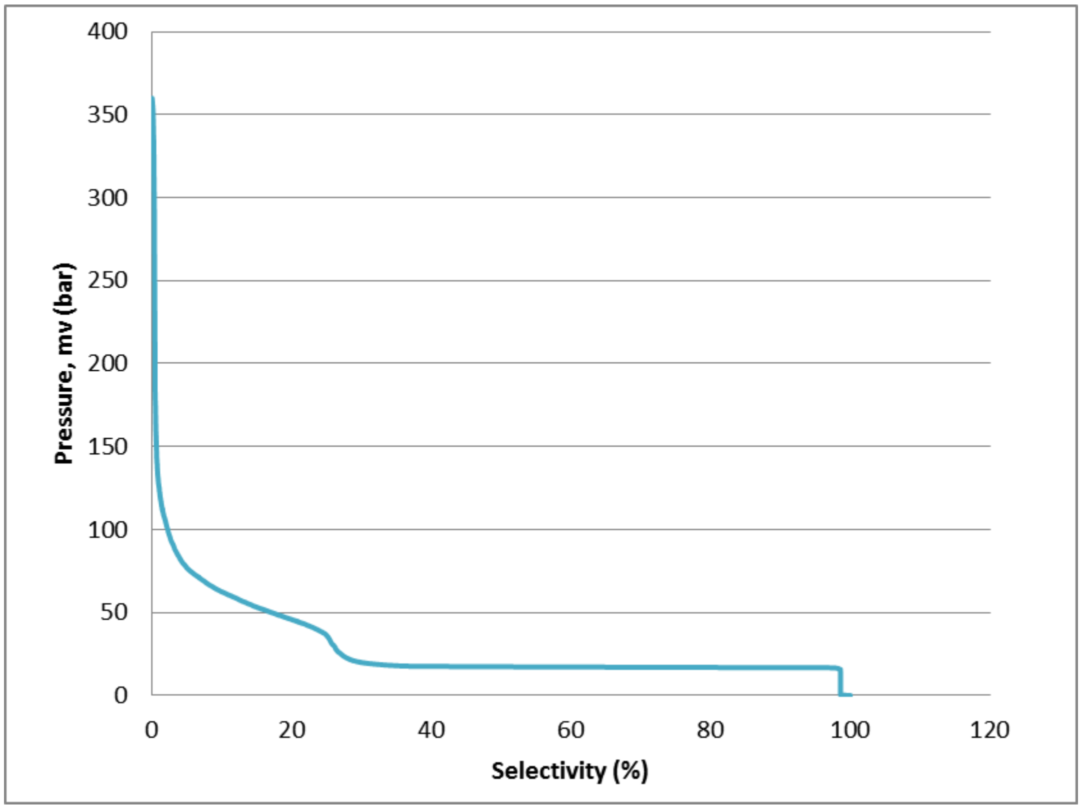


Figure 4.2. Measured value to selectivity mapping

Based on Figure 4.2, we executed Q2 for values of *mv* resulting in the selectivities 0.02%, 0.2%, 2%, 5% and 10%. In order to improve the performance of the query, we need the efficient use of a B-Tree index. We can utilize a secondary B-Tree index on *mv* for MySQL, DB-C, and MongoDB.

Query Q2 is an example of a very basic analytical query that involves inequality comparisons. Complex analytical queries usually involve inequalities and can often be rewritten into inequality queries like Q2 [9].

Another performance issue is that bulk loading gets slowed down by adding a secondary index on *mv*, which is also investigated.

4.3 Indexing alternatives

To investigate the impact of different indexing strategies and their trade-off with bulk-loading we investigated the following three kinds of indexing:

(1) ***No index:*** We performed bulk loading execution of the two queries without specifying any index. Here we expected to achieve fast loading time but the analytics should suffer without any index.

(2) ***Sensor key index:*** We created a composite index on machine id *m*, sensor id *s* and begin time *bt*. The data was bulk loaded with the index created and the corresponding queries were performed. We expected to observe degradation in loading performance compared to experiment (1). However, the key lookup query of Q1, was expected to have significant performance improvement, as it will utilize this index. Query Q2 does not utilize this index and should not be impacted.

(3) ***Sensor key and value indexes:*** We added an extra secondary index on *mv*. The data was then bulk loaded and the same queries as before were investigated. We expected to observe further degradation of the loading performance. Furthermore, the simple analytical task Q2 should perform significantly better, as it can utilize the secondary index on *mv*.

The indexing influences both bulk loading performance and storage utilization.

4.4 Benchmark environment

The benchmark was performed on a computer running Intel^R Core^{TM} i5-4670S, 3.1GHz CPU with Windows 7 Enterprise 64-bit operating system. The server has 16GB of physical memory. The MongoDB version used for empirical performance evaluation was v2.4.8 and for MySQL Server it was 5.6.12. For the MongoDB database AMI was used to execute the benchmark queries in AmosQL. There is an SQL interface for JavaAmos [18], which was used to perform the queries for MySQL and DB-C. Here, JavaAmos is a version of the Amos II kernel connected to the Java virtual machine (JVM). Although the SQL interface for JavaAmos utilizes Java, this should not impact significantly the performance compared to the C interface of AMI; according to [26], "*High-level* *languages are good and need not hurt performance”*. However, for providing unbiased comparison results, a SQL interface in C for Amos II was developed for MySQL performance evaluation in this project by utilizing the same Amos II C Interfaces [7] that is utilized in the AMI implementation.

4.4.1 MySQL configuration

In MySQL, the InnoDB engine was used in the benchmark, where the benchmark database and index size will occupy approximately 11.3GB and 6.3GB, respectively. According to the MySQL memory allocation recommendation [27], the *innodb_buffer_pool_size* should be set to 70% of available memory which is approximately 9.22GB for the chosen configuration. Furthermore, as the query cache biases the query execution speed, it was turned off by enabling the *SQL_NO_CACHE* option.

4.4.2 MongoDB configuration

As the attribute names are stored in each BSON object of a MongoDB collection, we have avoided long and descriptive attribute names, which might impact the final size of the database. For example, we avoided attribute names like *“mechineId”* and used *“m”* instead. No other optimizations and dedicated allocations of memory were performed for MongoDB.

4.4.3 DB-C configuration

We did not use any optimization and dedicated allocations of memory to evaluate the performance of DB-C; it was used out-of-the-box with default configuration. As for MySQL, the query cache was turned off.

4.4.4 Benchmark execution

For each of system we measured the load-time for 1, 2, 4, and 6 GB of data size. The raw data files were stored in CSV format where each individual row represents the sensor reading based on machine identifier *m*, sensor identifier *s*, begin time *bt*, end time *et*, and the measured value *mv*.

There are two ways to bulk load into MongoDB:

1. using command line utility *mongoimport*, and

2. using the client driver API function *mongo_insert_batch()*.

AMI utilizes the second alternative to provide greater flexibility since it provides different levels of write concerns for observing the tradeoff between write acknowledgement and bulk loading speed up. Unlike the *mongoimport* utility, *mongo_insert_batch()* also provides options to modify data before loading. We also analyzed the performance of command line loading, which has about the same performance compared to the acknowledged version of *mongo_insert_batch()*.

Bulk loading into MySQL was always performed utilizing the *LOAD DATA INFILE* SQL command for bulk loading CSV files[1]. For DB-C, we used its bulk loading utility. It should be noted that the *LOAD DATA INFILE* command of MySQL and bulk loading command for DB-C are not as flexible as the AMI bulk loading interface for MongoDB, which allows advanced pre-processing by AmosQL of log records from an input stream.

For the task executions, the key lookup Q1 for all three systems used the same data sizes starting from 1GB up to 6 GB in order to measure system scalability. By contrast, the simple analytical query Q2 was executed only with the largest data size of 6GB for all the systems since it evaluates the impact of using B-Tree based secondary index to speed up query performance, and this speed-up depends on the selectivity of the indexed condition and not the database size.

To enable incremental bulk loading of new data into exiting collections, the indexes are always predefined in all experiments, rather than building them after the bulk loading. Although one might consider the option of bulk loading first and then building the index, this will contradict the notion of our real scenario of applications where bulk

[1] The alternative to insertion rows by executing *INSERT INTO* commands was not used as it was significantly slower.

loading and analyzing the stream of log data is a continuous process that demands incremental loading of the data into a pre-existing log table or collection.

For providing stable results for each benchmark task, we made all the experiment starting with empty databases.

4.5 Experimental Results

In this sub-section, we present our benchmark results consisting of bulk loading and analyzing by scaling the size of the log data up to 6GB. For observing the tradeoff between bulk loading and speeding up the analytical tasks, the following three alternative experiments were conducted to investigate the impact of different indexing strategies discussed in section 4.3. In the following subsections we discuss these three alternative investigations in sequence.

4.5.1 No index

We loaded the data and made the performance measurements on all systems without defining any index. The loading performance for the systems is shown in Figure 4.3.

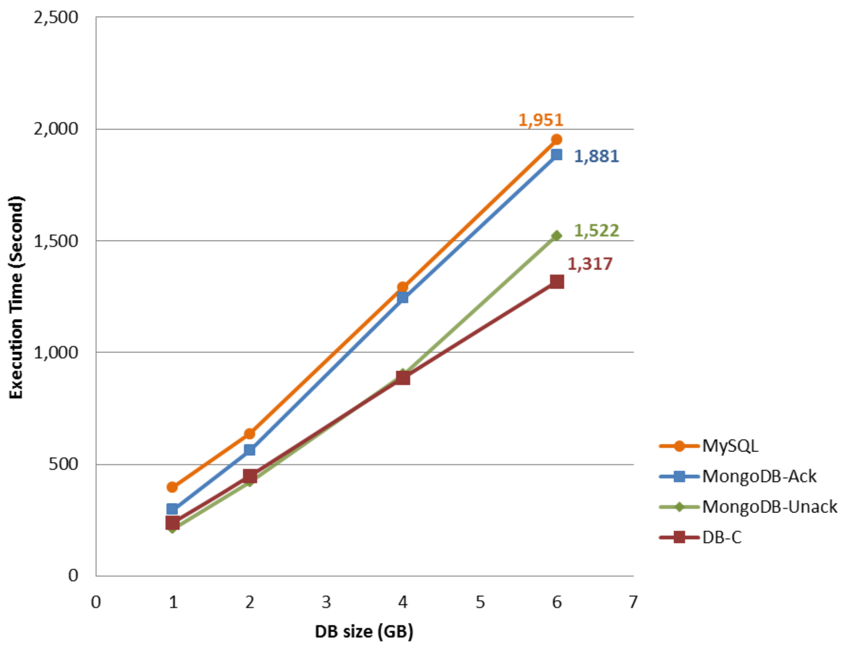


Figure 4.3. Performance of bulk loading without indexing

In the above figure, we can observe that all of the systems offer scalable loading performance by observing the linear increase in time as data size grows. DB-C was faster compared to all other systems and scaled linearly (around 4.3 MB/s), even faster than MongoDB without acknowledged write concern. Furthermore, the performance difference is small between MongoDB with acknowledged write concern and MySQL, despite that there is always a primary key index on *_id* in MongoDB, while MySQL has no index. As expected, for large data loadings (more than 4 GB) unacknowledged MongoDB is 19% and 22% better than both the acknowledged MongoDB and MySQL bulk loadings.

In Figure 4.4, the key lookup task Q1 is measured, to retrieve the particular record of a sensor. All three systems do not scale without indexing since every system performs a full table/collection scan. However, MongoDB seems to provide better performance compare to other systems.

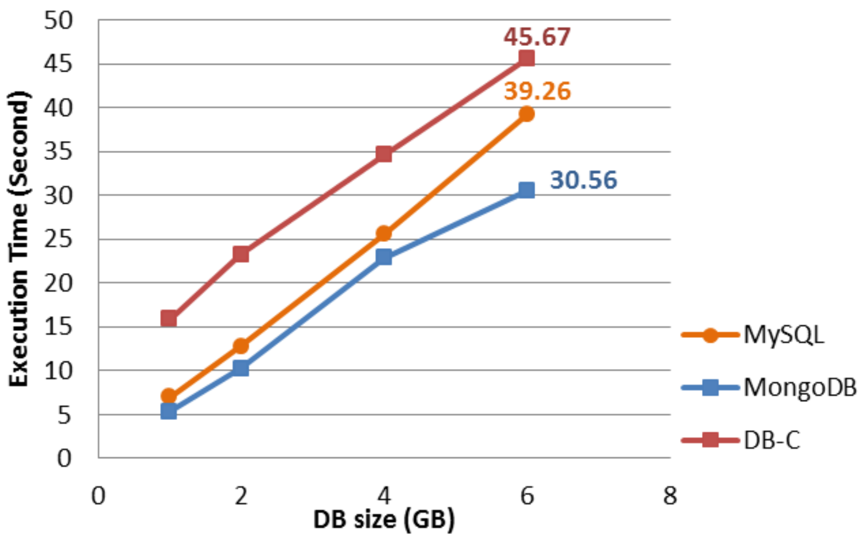


Figure 4.4. Performance of Q1 without indexing

Figure 4.5 provides the performance of the basic analytical task of Q2 without index for selectivities ranging from 0.02% up to 10% and 6GB of data. Unlike the key lookup task Q1, MongoDB here performed around 40% worse than both MySQL and DB-C for any.

As the key lookup task, an ordered B-Tree indexing is expected to speed up the performance of this analytical task.

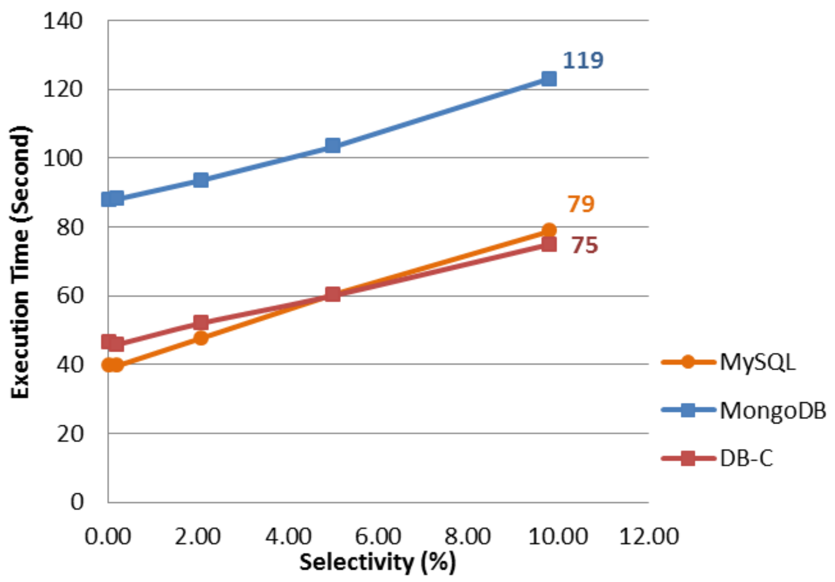


Figure 4.5. Performance of Q2 with varying selectivity without indexing

Discussion of results:

The performance results of bulk loading, key lookup task Q1 and basic analytical task Q2 were shown in Figure 4.3, Figure 4.4, Figure 4.5, respectively. DB-C demonstrated fastest bulk loading performance compare to other systems. Although there is a default primary index on the MongoDB, both of the bulk loading alternatives performed surprisingly better than MySQL.

For the key lookup task Q1 MongoDB is slightly faster, and the difference seems to increase with increasing database size. The lack of indexing makes none of the systems scale well. However, further investigations with a larger data set are needed for final conclusion.

For the basic analytical task Q2, both MySQL and DB-C is significantly faster than MongoDB for any chosen selectivity. Also in this case, the lack of indexing decreases scalability for all systems.

In the next section, we will add a composite index on machine id, sensor id and begin time to speed-up the key lookup task Q1 and observe the tradeoff between loading time and task exaction.

4.5.2 Sensor key index

In this experiment, we defined a composite index on machine id *m*, sensor id *s*, and begin time *bt*. In MySQL and DB-C, a composite primary key on these fields was defined. In MongoDB a composite secondary index was defined. The loading performance is shown for all systems in Figure 4.6. The most surprising outcome from the result is that MySQL scales significantly worse than MongoDB and DB-C.

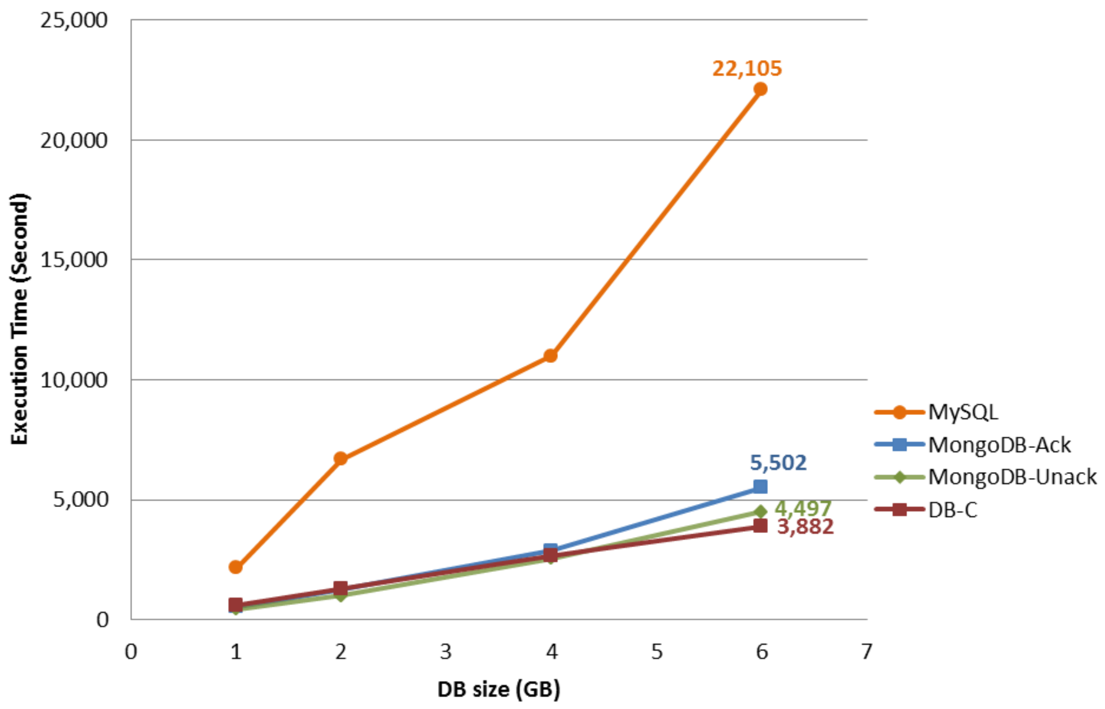


Figure 4.6. The performance of bulk loading with sensor key index

When the inserted data size is 4 GB or more, MySQL is more than 4 times slower compare to other systems. Both DB-C and MongoDB demonstrates scalable performance for bulk loading. DB-C scaled linearly (around 1.7 MB/s).

According to Figure 4.7, with the sensor key index, the key lookup task Q1 for a 6GB database takes 0.12 s with MySQL, 0.25 s with MongoDB and 0.076 s with DB-C, which is insignificant for all systems and shows that the indexing works.

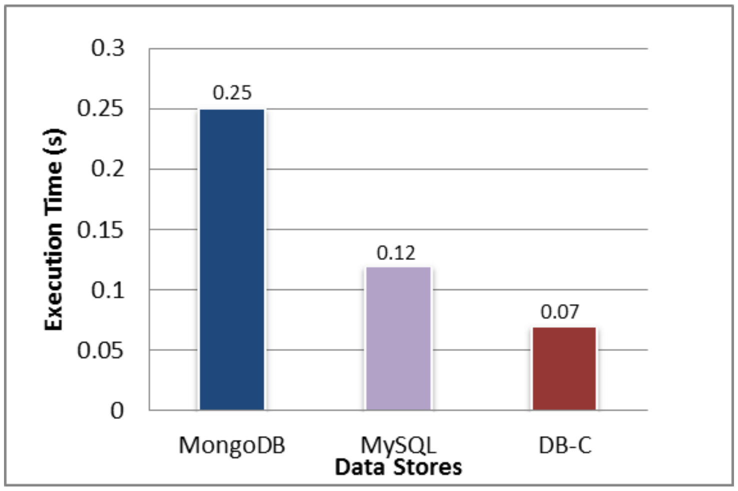


Figure 4.7. The performance of Q1 for 6GB data size with sensor key index

For MongoDB we also measured the access time of obtaining an object for a given MongoDB identifier, which was around 0.25 s, i.e. the access time for an identity lookup in MongoDB is the same as for a sensor key lookup. The reason is that both indexes are clustered B-Tree indexes.

The result of the simple analytical task Q2 for different selectivity is shown in Figure 4.8. Here, it turns out that MySQL performs much worse with a primary key index than without one, while MongoDB demonstrated scalability with less performance degradation. Whereas the performance of DB-C follows MongoDB's performance up to 5% selectivity and performed better for higher selectivity. With a senor key index MongoDB and DB-C become more than 12 and 32 times faster than MySQL, respectively.

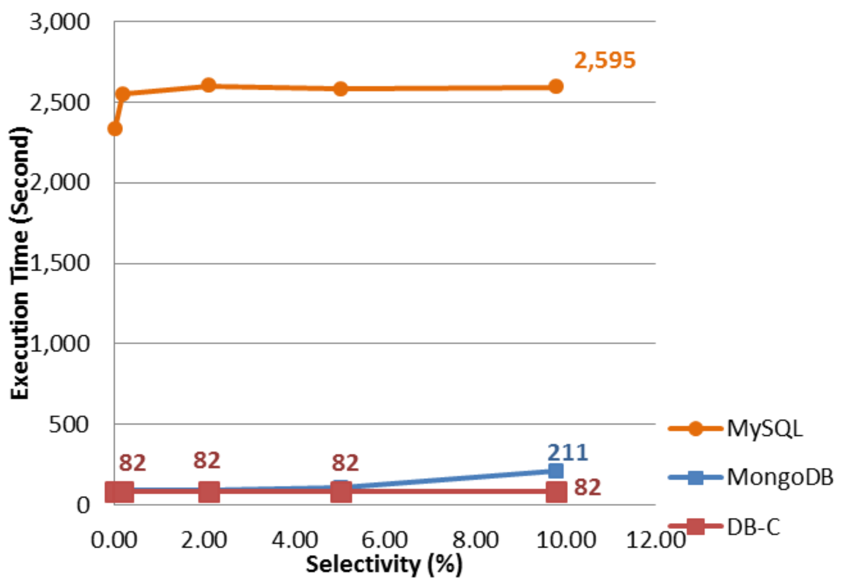


Figure 4.8. The performance of Q2 varying the selectivity with sensor key index

Discussion of results:

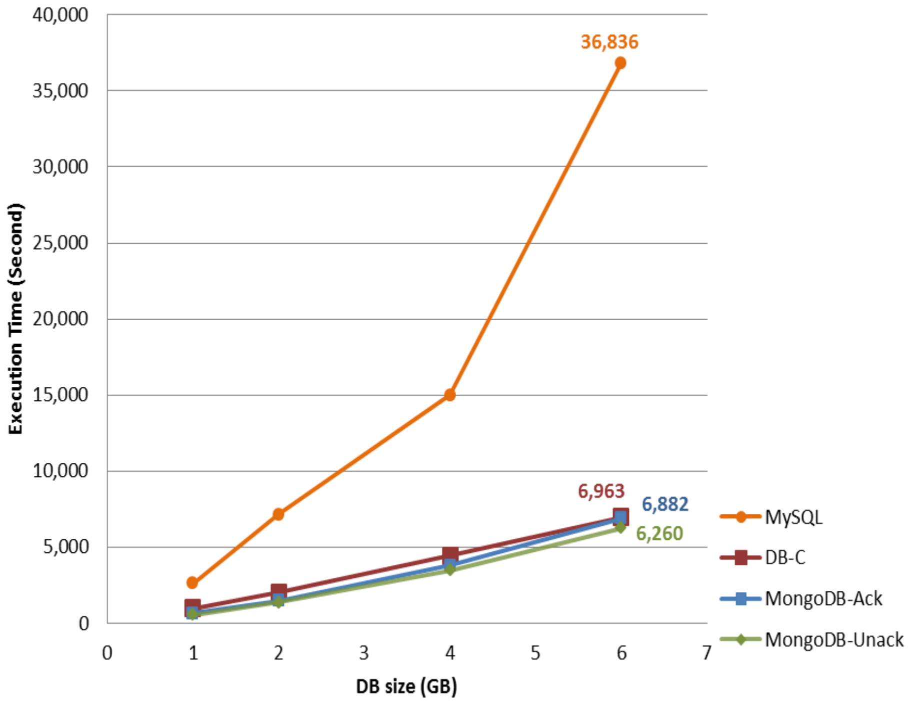
Even though MongoDB maintains both a primary index and a secondary composite index, it scales significantly better than MySQL and is comparable with DB-C for bulk loading of data logs. As expected, the sensor key index provides scalability for the key lookup task Q1 for all systems, while the simple analytic task Q2 is still slow.

In the next section, we added an index on measured value, *mv* to speed up the basic analytical task Q2 and observe the tradeoff between loading time and tasks exaction.

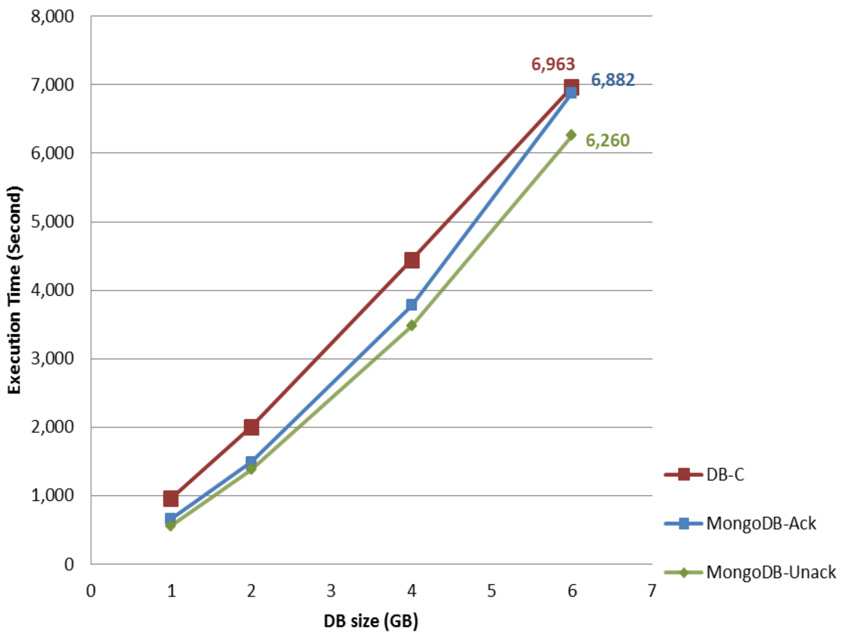
4.5.3 Sensor key and value indexes

In this experiment, we defined an ordered index on the measured value *mv*, for achieving scalable performance of Q2 and to evaluate the impact of selectivity when using that index. In every system, a secondary B-Tree index on *mv* field is defined.

The loading performance is shown in Figure 4.9.



(a)



(b)

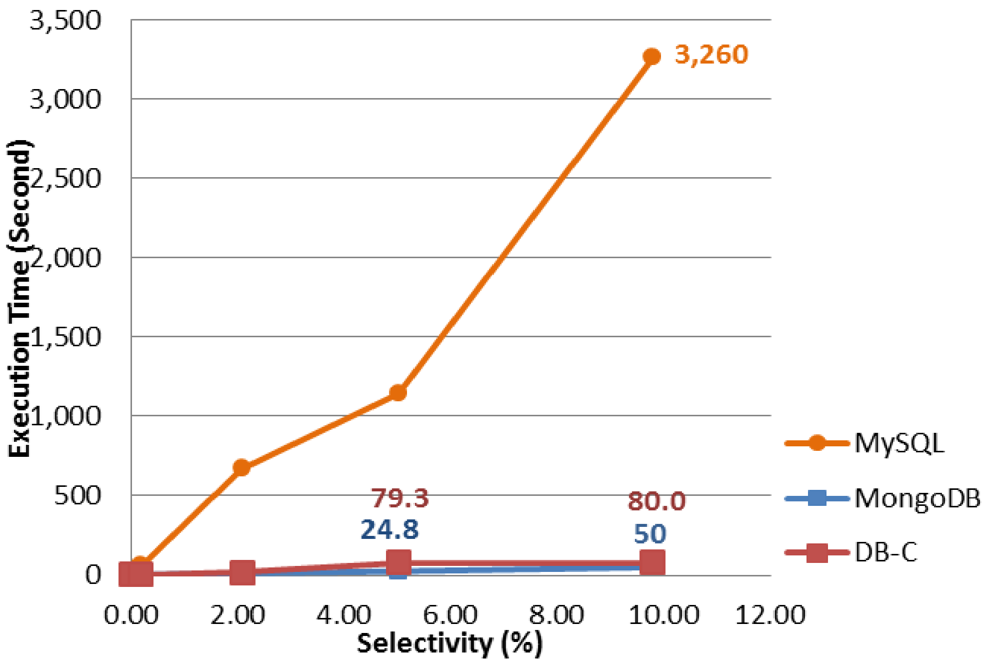
Figure 4.9. The performance of bulk loading with both sensor key and value indexes, (a) shows three systems (b) highlights only MongoDB and DB-C

The secondary value index decreases the loading performance of MySQL for a 6GB database around 60%, 25% for acknowledged MongoDB inserts, and 38% for unacknowledged MongoDB inserts. Thus the extra index has less impact on MongoDB performance than on MySQL.

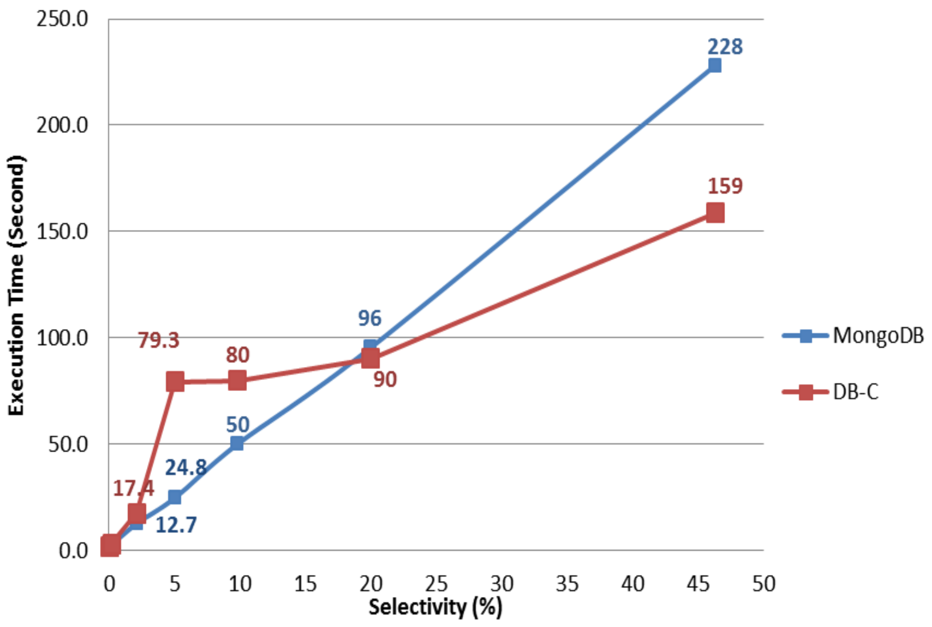
The bulk load time for DB-C is slightly slower than MongoDB scaling linearly (around 1.1 MB/s), while MySQL is very slow when both indexes are defined beforehand.

The performance of the key lookup task Q1 is the same as without the sensor value index, as expected.

Figure 4.10 shows the performance of the simple analytical task Q2 for different selectivities and a database size of 6 GB with both sensor key and value indexes. Clearly there is a problem with secondary index for inequality queries in MySQL. On the other hand, both MongoDB and DB-C scale well. Figure 4.10b compares the performance of Q2 for only MongoDB and DB-C.



(a)



(b)

Figure 4.10 The performance Q2 varying the selectivity with sensor key and value indexes a) shows three systems b) highlights only MongoDB and DB-C

This time we have decrease the selectivity up to 50%. Here it can be seen that for selectivities up to 2% DBC-C is slightly slower than MongoDB, between 2% and 20% Mongo-DB is faster, while above 20% DB-C is faster. The reason is that the query optimizer of DB-C changes from a non-clustered index scan to a full scan when the selectivity is somewhere between 2% and 5%, whereas MongoDB continues with in an index scan for growing selectivities.

Table 4.1 lists all the performances of the simple analytical task Q2 for a 6 GB database with varying selectivities, without indexing, with sensor key indexes, and with both sensor key and value indexes. It can be seen that for highly selective queries (0.02 %) the secondary index improves the performance of MongoDB from 88 s to 1.6 s (factor 55) , of MySQL from 40 s to 21 s (a factor 1.9) and of DB-C from 46 s to 1.8 s (factor 26) so the secondary index improves the performance much more for MongoDB and DB-C than for MySQL.

Selectivity	Without index			Sensor key index			Sensor key & value indexes		
%	MongoDB	MySQL	DB-C	MongoDB	MySQL	DB-C	MongoDB	MySQL	DB-C
0.02	88	40	46	88	2,335	82	1.6	21	1.8
0.2	88	40	44	90	2,549	83	2.7	57	3.3
2.00	94	48	52	95	2,602	82	12.7	672	17.4
5.00	103	60	60	104	2,582	82	24.8	1,141	79.3
10.00	119	79	75	211	2,595	82	50	3,260	80.0

Table 4.1. Analytical Task, Q2 with sensor key and value indexes

Discussion of results:

The results showed that MySQL was clearly slower than MongoDB and DB-C for both bulk loading and for utilizing secondary indexes in inequality queries, which are very important issues for log data analytics. Both DB-C and MongoDB scaled for the basic analytical task Q2 while MySQL did not. For selective queries, MongoDB performs better than DB-C while for non-selective queries DB-C switches to full scan thus provides better scalability.

4.5.4 Storage utilization

Figure 4.11 shows the database and index sizes for 6GB of log data loaded into the three systems. The total database size together with all the indexes in MongoDB, MySQL and DB-C are 20.3 GB, 17.7 GB 15GB, respectively. In MongoDB there is extra storage overhead of 3.4 GB for object identifier index.

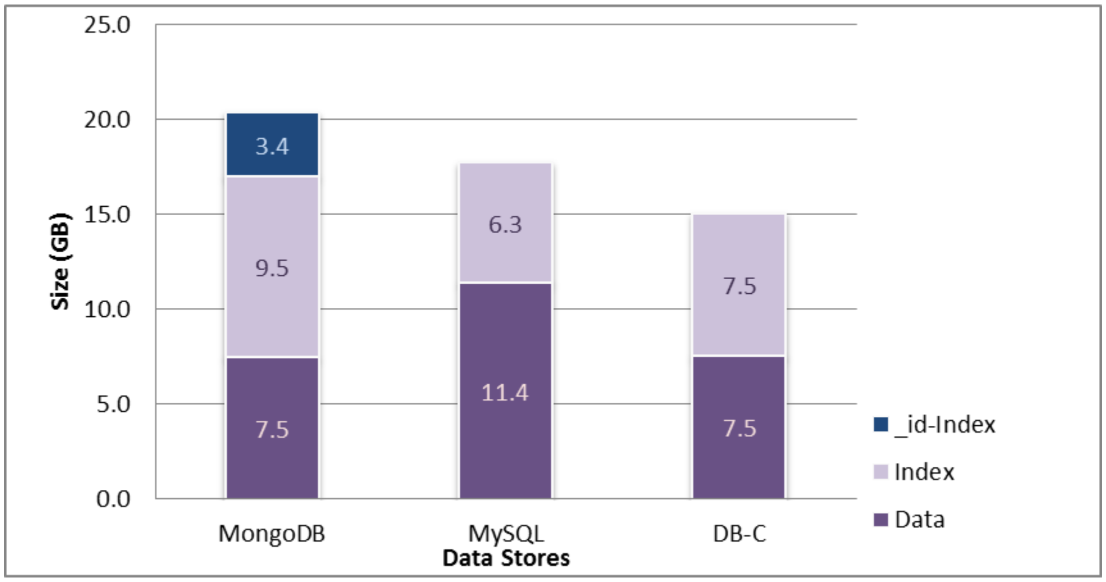


Figure 4.11. Database statistics for 6GB raw sensor data

For the data records MySQL consumes 52% more storage compare to MongoDB 11.4 GB. The reason is that our MongoDB representation is very compact because of the short attribute names (average 72 bytes), while the MySQL records have a fixed length of 95 bytes.

The index size for the combined sensor key and value indexes of MongoDB is larger (9.5 GB) than the corresponding MySQL (6.3 GB) and DB-C (7.5 GB) indexes.

4.6 Discussion

Although index utilization is crucial for efficient analysis of log data, indexing also slows down the process of bulk loading. For example, bulk loading was demonstrated to be scalable without having any index for the relational DBMSs, and with the default object identifier index of MongoDB. In spite of MongoDB having an object identifier index, its bulk loading performance was still 3% better than MySQL without any index. The bulk loading of the 6GB dataset into DB-C was fastest and linear at a speed of 4.3 MB/s, compared to both MySQL and MongoDB.

The introduction of indexes to speed up the analytical tasks demonstrated that MySQL's did not scale for our application compare to MongoDB and DB-C when indexing is used. For the largest data size of 6GB, both bulk loading alternatives of MongoDB and DB-C were more than 5 times faster compared to MySQL. For the analytical query, Q2 with 10% selectivity, MongoDB was 65 times faster and DB-C was 41 times faster compared to MySQL. The reason of performance degradation was due to MySQL's ineffective utilization of secondary B-Tree index. The good performance of DB-C show that relational databases are comparable to non-distributed NoSQL data stores for persisting and analyzing streaming logs.

The unacknowledged write concern of bulk loading with MongoDB is faster compared to acknowledge write concern (see Figure 4.9), so this option can be utilized for high performance loading of time-critical applications. However, the difference between unacknowledged bulk loading in MongoDB and bulk loading in DB-C was less than 10% at expense of a possibly inconsistent database. DB-C scaled best and linearly when only the key index was present (1.7 MB/s), while MongoDB was faster up to 6GB when both indexes were present while DB-C still scaled better and linearly than MongoDB (1.1 MB/s).

We also demonstrated MongoDB's default primary key index utilization by executing a lookup query and drew the conclusion that the utilization of this index can be highly efficient and its future application should be investigated.

The database and index sizes in MongoDB are comparable to the relational databases, with slight overhead for the unused primary index in MongoDB. MongoDB would be as compact as the other systems if the regular primary index can replaced with the unused index.

We also found that MongoDB is easy to use and could be run out-of-the box with minimal tuning while MySQL requires substantial tuning to be efficient. DB-C requires more complicated setup compared to the other system, but no tuning was necessary.

Furthermore, AMI's bulk loading was demonstrated to be flexible and having the same performance compared both to the bulk loading tools of the relational DBMSs and the command language interface in MongoDB. The flexibility makes AMI suitable for pre-processing of log data with insignificant overhead. For the experiments one large CSV log file of all sensor readings was first split into several smaller CSV log files with different sizes. These CSV files were then streamed into MongoDB by using a CSV file reader available in Amos II to feed the read data into AMI's bulk loader. The overhead of such streaming in AMI is that BSON records have to be generated and sent to the server. The overhead was found to be insignificant: for example, when data size is 6 GB, more than 111 million Amos II native objects of type *Record* were converted to the corresponding *BSON* objects of MongoDB before bulk loading them into the server. The difference in total bulk loading time between the bulk loading interface of AMI and MongoDB's command line bulk loading utility was insignificant.

5 Conclusion and Future Work

The performance of loading and analyzing of numerical log data using a NoSQL data store was investigated as an alternative to relational databases. The performance was evaluated on a benchmark consisting of real world application log files. The currently most popular NoSQL data store, MongoDB [4] and the most popular open source relational DBMS, MySQL [28] was compared with a state-of-the-art relational DBMS from a major commercial vendor. The evaluation was made by observing the tradeoff between load-time and basic analytic queries over numerical logs. Since error analyses often require selecting the sensor reading where an attribute value is larger than a threshold, the effectiveness of index selection was evaluated by varying the selectivity of a non-key attribute.

Although MySQL demonstrated similar bulk load performance of log data as the other systems when no index was defined, both MongoDB and the commercial relational DB-C scaled substantially better for bulk loading when an index(es) was present. Furthermore, MongoDB has the option to load data without confirming success of write operations (unacknowledged write concern). This option was slightly faster (around 10%) than DB-C when both indexes were defined. As historical data analysis can often tolerate weaker consistency for persistent loading, such an option may be useful for analyzing log data where full consistency is not required.

All system performed well for looking up records matching the key (query Q1) when a primary key index is present. For the analytical task of range comparisons between a non-key attribute and a constant (query Q2), both MongoDB and DB-C scaled substantially better than MySQL. A more careful comparison of DB-C and MongoDB revealed that DB-C scales better for non-selective queries, while MongoDB is faster for selective ones. The reason is that, unlike MongoDB and MySQL, DB-C switches from a non-clustered index scan to a full table scan when the selectivity is sufficiently low, while MongoDB (and MySQL) continues to use an index scan also for non-selective queries.

The extensible DBMS Amos II [6] was utilized for the comparison. An interface between Amos II and MongoDB called AMI (Amos Mongo Interface) was implemented to enable general queries and updates to MongoDB data store using MongoDB's query language. A corresponding interface already existed for querying relational databases from Amos II using SQL. While providing high performance access to MongoDB, AMI enables significant flexibility advantages compared to the MongoDB command line interface, including (1) expressing powerful and relationally complete AmosQL queries and updates including join and comprehensive numerical operator support not present in MongoDB, (2) comprehensive bulk loading data into MongoDB from AmosQL, and (3) execution of MongoDB database commands from AmosQL. Compared to the rigid bulk load interfaces of MySQL and DB-C, AMI's implementation of bulk inserts is flexible and opens us to perform on-line bulk loading of data streaming directly from sensors.

There are several critical issues that we observed in this project as future work:

- As MongoDB always maintains a default primary key index on an object identifier per record (document), this index should also be utilized as a clustered index when possible, which is being investigated as another alternative.
- MongoDB does not provide queries containing joins or numerical operators. Here AMI provides an interface for implementing such operators as post-processing operators in Amos II. The flexibility and performance of AMI provides an appropriate foundation for development of a fully-functional Amos-Mongo wrapper, which provides transparent relationally complete queries over MongoDB databases possibly combined with other kinds of data sources.
- Although the performance evaluation demonstrated that a system combining NoSQL features with dynamic query support by utilizing automatic indices can provide significant performance advantage for persistent loading and analyzing of historical log data, an elaborate

analysis of such applications having more complex queries should be developed.

- MongoDB provides automatic parallelization (sharding), where collections are distributed over several MongoDB servers based on the primary keys. The performance implications of sharding should be investigated. In particular, parallelization should improve bulk loading times. Even though MongoDB does not guarantee consistency between shards it could be acceptable for log analytics.

To conclude, in our chosen application scenario, MongoDB is shown to be a viable alternative for high performance loading and analysis of historical log data compared to relational databases.

References

[1] R. Cattell, “Scalable SQL and NoSQL data stores,” *ACM SIGMOD Rec.*, vol. 39, no. 4, p. 12, May 2011.

[2] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. Dewitt, S. Madden, and M. Stonebraker, “A Comparison of Approaches to Large-Scale Data Analysis,” in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, 2009, pp. 165–178.

[3] MongoDB Inc., “The MongoDB 2.4 Manual,” 2013. [Online]. Available: http://docs.mongodb.org/v2.4/. [Accessed: 14-Feb-2014].

[4] MongoDB Inc., “MongoDB – The Leading NoSQL Database,” 2014. [Online]. Available: http://www.mongodb.com/leading-nosql-database. [Accessed: 04-Mar-2014].

[5] MongoDB Inc., “MongoDB C Driver 0.8.1 Documentation,” 2013. [Online]. Available: http://api.mongodb.org/c/0.8.1/. [Accessed: 14-Feb-2014].

[6] S. Flodin, M. Hansson, V. Josifovski, T. Katchaounov, T. Risch, M. Sköld, and E. Zeitler, “Amos II Release 16 User’s Manual,” *Uppsala DataBase Laboratory, Department of Information Technology, Uppsala University, Sweden*, 2014. [Online]. Available: http://www.it.uu.se/research/group/udbl/amos/doc/amos_users_guide.html. [Accessed: 14-Feb-2014].

[7] T. Risch, “Amos II C Interfaces,” *Uppsala DataBase Laboratory, Department of Information Technology, Uppsala University, Sweden*, 2012. [Online]. Available: http://user.it.uu.se/~torer/publ/externalC.pdf. [Accessed: 14-Feb-2014].

[8] C. Strauch, “NoSQL Databases,” Stuttgart, 2011.

[9] T. Truong and T. Risch, “Scalable Numerical Queries by Algebraic Inequality Transformations,” in *The 19th International Conference on Database Systems for Advanced Applications, DASFAA 2014*, 2014.

[10] E. A. Brewer, “Towards Robust Distributed Systems,” in *PODC ’00 Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, 2000, pp. 7–19.

[11] "JavaScript Object Notation (JSON)," 2014. [Online]. Available: http://www.json.org/. [Accessed: 06-Mar-2014].

[12] MongoDB Inc., "BSON Types," 2013. [Online]. Available: http://docs.mongodb.org/manual/reference/bson-types/. [Accessed: 16-Feb-2014].

[13] J. Dean and S. Ghemawat, "MapReduce: Simplied Data Processing on Large Clusters," in *Proc of 6th Symposium on Operating Systems Design and Implementation*, 2004, pp. 137–149.

[14] T. Risch and V. Josifovski, "Distributed data integration by object-oriented mediator servers," *Concurr. Comput. Pract. Exp.*, vol. 13, no. 11, pp. 933–953, Sep. 2001.

[15] T. Risch, V. Josifovski, and T. Katchaounov, "Functional Data Integration in a Distributed Mediator System," *.M.D.Gray, L.Kerschberg, P.J.H.King, A.Poulovassilis Funct. Approach to Comput. with Data , Springer*, 2004.

[16] D. W. Shipman, "The functional data model and the data language DAPLEX," in *Proceedings of the 1979 ACM SIGMOD international conference on Management of data - SIGMOD '79*, 1979, p. 59.

[17] P. Gulutzan and T. Pelzer, *SQL-99 Complete, Really*. Miller Freeman, Lawrence, Kansas, 1999.

[18] D. Elin and T. Risch, "Amos II Java Interfaces," *Uppsala DataBase Laboratory, Department of Information Technology, Uppsala University, Sweden*, 2000. [Online]. Available: http://user.it.uu.se/~torer/publ/javaapi.pdf. [Accessed: 14-Feb-2014].

[19] K. Mahmood, "MongoDB C Driver 0.8.1 Socket Issues on Windows 7, Original Title: Socket initialization problem Windows 7," 2014. [Online]. Available: https://jira.mongodb.org/browse/CDRIVER-290. [Accessed: 16-Feb-2014].

[20] MongoDB Inc., "Database Commands," 2013. [Online]. Available: http://docs.mongodb.org/manual/reference/command/. [Accessed: 16-Feb-2014].

[21] MongoDB Inc., "SQL to MongoDB Mapping Chart," 2013. [Online]. Available: http://docs.mongodb.org/manual/reference/sql-comparison/. [Accessed: 19-Feb-2014].

[22] MongoDB Inc., "User Privilege Roles in MongoDB," 2013. [Online]. Available: http://docs.mongodb.org/manual/reference/user-privileges/. [Accessed: 16-Feb-2014].

[23] The SCons Foundation, “SCons: Open Source software construction tool,” 2013. [Online]. Available: http://www.scons.org/. [Accessed: 16-Feb-2014].

[24] Microsoft, “WSAStartup function: Microsoft Develoer Network (MSDN),” 2013. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/ms742213(v=vs.85).aspx. [Accessed: 16-Feb-2014].

[25] “Smart Vortex Project,” 2014. [Online]. Available: http://www.smartvortex.eu/. [Accessed: 26-Feb-2014].

[26] M. Stonebraker and R. Cattell, “10 Rules for Scalable Performance in ‘Simple Operation’ Datastores,” *Commun. ACM*, vol. 54, no. 6, pp. 72–80, Jun. 2011.

[27] R. James, “MySQL Memory Allocation,” 2012. [Online]. Available: http://mysql.rjweb.org/doc.php/memory. [Accessed: 27-Feb-2014].

[28] O. Corporation, “Market Share,” 2014. [Online]. Available: http://www.mysql.com/why-mysql/marketshare/. [Accessed: 04-Mar-2014].

Appendix A

Amos Mongo Interface (AMI) functions and examples

AMI interface functions

```
/* Connect to the MongoDB database server on a given IP host */
create function mongo_connect(Charstring host) -> Integer conn_no
  as foreign 'mongo_connect+-';

/* Disconnect from the MongoDB server */
create function mongo_disconnect(Number conn_no)-> Boolean status
  as foreign 'mongo_disConnect-+';

/* Add a record to a MongoDB collection and return its MongoDB identifier*/
create function mongo_add(Number conn_no, Charstring  database,
                          Charstring collection, Record o) -> Object id
  as foreign 'mongo_add----+';

/* Add batch of records to a MongoDB collection */
create function mongo_add_batch(Integer  conn_no, Charstring  database,
                                Charstring  collection,
                                Integer write_concern,  Vector oid)
                              -> Boolean status
  as foreign 'mongo_add_batch-----+';

/* Add batch of records to a MongoDB collection with write acknowledgment
*/
create function mongo_add_batch_ack(Integer  conn_no, Charstring  database,
                                Charstring  collection,
                                Vector oid)
                              -> Boolean status
as mongo_add_batch(conn_no, database, collection, 1,oid);

/* Add batch of records to a MongoDB collection without write
acknowledgment */
create function mongo_add_batch_unack(Integer  conn_no,
                         Charstring  database, Charstring  collection,
                          Vector oid) -> Boolean status
as mongo_add_batch(conn_no, database, collection, 0,oid);

/* Delete record(s) from a MongoDB collection and return Boolean status on
success */
create function mongo_del(Number  conn_no, Charstring  database,
                          Charstring collection, Record q)
                                -> Boolean status
  as foreign 'mongo_del----+';

/* Query a MongoDB collection */
create function mongo_query(Number  conn_no, Charstring  database,
                          Charstring collection, Record q)
                         -> Bag of Record x
  as foreign 'mongo_query----+';

/* Create index on collection */
create function mongo_createIndex(Number  conn_no, Charstring  database,
                                  Charstring  collection,
                                  Charstring indexName, Record field)
                                  -> Boolean status
  as foreign 'mongo_createIndex-----+';
```

AMI database commands

```
/* Generic function for issuing database command */
create function mongo_run_cmd(Integer  conn_no, Charstring  database,
                              Record cmd)
                            -> Record status
  as foreign 'mongo_run_cmd---+';

/* Derived function to drop an index */
create function mongo_dropIndex(Integer  conn_no, Charstring  database,
                                Charstring  collection,
                                Charstring indexName) -> Record status
  as mongo_run_cmd(conn_no, database,
                 { "dropIndexes": collection, "index": indexName });

/* Derived function to drop an index */
create function mongo_dropAllIndex(Integer  conn_no, Charstring  database,
                                   Charstring  collection)
                                 -> Record status

  as mongo_run_cmd(conn_no, database,
                 { "dropIndexes": collection, "index": "*" });

/* Return status object containing all errors */
create function mongo_getPrevError(Integer  conn_no, Charstring  database )
                                 -> Record status
  as mongo_run_cmd(conn_no, database, { "getPrevError": 1 });

/* Repors storage utilization statics for a specified collection */
create function mongo_collStats(Integer  conn_no, Charstring  database,
                                Charstring  collection) -> Record status
  as mongo_run_cmd(conn_no, database, { "collStats": collection});

/* Report the aggregate statistics for the B-tree data structure of a
MongoDB index.*/
create function indexStats(Integer  conn_no, Charstring  database,
                           Charstring  collection, Charstring indexName)
                         -> Record status
  as mongo_run_cmd(conn_no, database,
                 { "indexStats": collection, "index": indexName });

/* Report storage utilization statistics for the specified database */
create function mongo_dbStats(Integer  conn_no, Charstring  database,
                              Integer scale) -> Record status
  as mongo_run_cmd(conn_no, database, { "dbStats": 1, "scale": scale});
```

```
/* Count the number of documents in a collection */
create function mongo_count(Integer  conn_no, Charstring  database,
                            Charstring  collection, Record query)
                          -> Record
  as mongo_run_cmd(conn_no, database,{"count":collection ,"query": query});

/* Remove the specified collection from the database */
create function mongo_dropCollection(Integer  conn_no,Charstring  database,
                              Charstring  collection) -> Record status
  as mongo_run_cmd(conn_no, database, { "drop": collection});

/* Remove the entire database */
create function mongo_dropDB(Integer  conn_no, Charstring  database)
                           -> Record status
  as mongo_run_cmd(conn_no, database, { "dropDatabase": 1});

/* Get all the collections namespaces in a database */
create function mongo_collNameSpaces(Integer  conn_no,Charstring  database)
                                   -> Bag of Record x
  as mongo_query(conn_no, database, "system.namespaces", empty_record());

/* Get all the index name in a database */
create function mongo_indexNameSpaces(Integer  conn_no,Charstring database)
                                    -> Bag of Record x
  as mongo_query(conn_no, database, "system.indexes", empty_record());
```

AMI tutorial

```
/* Make a new connection and store in temporary (transient) variable :c */
set :c = mongo_connect("127.0.0.1");

/* Empty old collection 'person' in  database 'tutorial': */
mongo_del(:c, "tutorial", "person", {:});

/* Populate the database with some new records. */
mongo_add(:c, "tutorial", "person", {"Name": "Ville", "age": 54});
/* mongo_add() returns the unique MongoDB OID for the new database record
*/

/* Populate more: */
mongo_add(:c, "tutorial", "person", {"Name": "George", "age": 27});
mongo_add(:c, "tutorial", "person", {"Name": "Johan", "age": 27});
mongo_add(:c, "tutorial", "person", {"Name": "Kalle", "age": 13});

/* Get all objects in the database: */
mongo_query(:c, "tutorial", "person", empty_record());

/* Query the database to find specific records: */
mongo_query(:c, "tutorial", "person", {"age": 27});
/* Notice the field "_id" holding the OIDs */

/* Manually associate your own identifier (numbers or strings) with
records: */
mongo_add(:c, "tutorial", "person", {"_id":1, "Name": "Olle", "age": 55});
mongo_add(:c, "tutorial", "person", {"_id":"abc","Name":"Ulla","age": 55});
mongo_add(:c, "tutorial", "person", {"_id":1.3,"Name":"Kalle","age":55});

/* Get all objects in the database: */
mongo_query(:c, "tutorial", "person", empty_record());

/* Get the new objects with your manually added OIDs: */
mongo_query(:c, "tutorial", "person", {"_id":1});
mongo_query(:c, "tutorial", "person", {"age":55});

/* Delete some objects: */
mongo_del(:c, "tutorial", "person", {"age": 27});
mongo_del(:c, "tutorial", "person", {"Name": "Olle"});

/* Check who is left: */
mongo_query(:c, "tutorial", "person", empty_record());

/* Store first encountered object named "Kalle" in variable :kalle */
select r into :kalle
from Record r
where r in mongo_query(:c, "tutorial", "person", {"Name":"Kalle"});

/* Inspect variable :kalle */
:kalle;

/* Set :id_kalle to the Mongodb identifier of :kalle: */
set :id_kalle = :kalle["_id"];

/* Inspect :id_kalle */
:id_kalle;

/* Use :id_kalle to retrieve the record: */
mongo_query(:c, "tutorial", "person", {"_id": :id_kalle});

/* Do a bulk insert of several records without acknowledgements: */
mongo_add_batch_unack(:c, "tutorial", "person", {
                                {"Name": "Carl", "age": 38},
                                {"Name": "Eve", "age": 65},
                                {"Name": "Adam", "age": 68},
                                {"Name": "Olof"}});

/* Look at the database: */
mongo_query(:c, "tutorial", "person", empty_record());

/* Find persons with age>64: */
mongo_query(:c , "tutorial", "person", {"age": {"$gt": 64}});

/* Find persons with 50<age<60: */
mongo_query(:c , "tutorial", "person", {"age": {"$gt": 50, "$lt": 60}});

/* Find persons whith age<=40 or age>=60: */
mongo_query(:c , "tutorial", "person", {"$or": {{"age": {"$lte": 40}},
                                                {"age": {"$gte": 60}}}});

/* Find persons whith age<=40 or age>=60 and named Adam: */
set :mq = {"$and": {{"$or": {{"age": {"$lte": 40}},{"age": {"$gte": 60}}}},
           {"Name": "Adam"}}};
```

```
pp(:mq); /* Pretty print query */

mongo_query(:c , "tutorial", "person", :mq);

/* Find persons whith age<=40 or age>=60 and not named Adam: */
set :mq = {"$and": [{"$or": [{"age": {"$lte": 40}},{"age": {"$gte": 60}}]},
           {"Name": {"$ne": "Adam"}}]};

pp(:mq); /* Pretty print query */

mongo_query(:c , "tutorial", "person", :mq);

/* Find persons with age not <60): */
mongo_query(:c , "tutorial", "person", {"age": {"$not": {"$lt": 60}}});

/* Inspect collection statistics: */
pp(mongo_collstats(:c, "tutorial", "person"));

/* Disconnect from data source */
mongo_disconnect(:c);
```

Database utility commands tutorial

```
/* Make a new connection and store in temporary (transient) variable :c */
set :c = mongo_connect("127.0.0.1");

/* Get all the collection namespaces (including system colections) in
 database "tutorial"*/
mongo_collNameSpaces(:c, "tutorial");
/* Extract the namespace from the records */
mongo_collNameSpaces(:c, "tutorial")["name"];

/* See the storage utilization stats of "person" */
mongo_collStats(:c, "tutorial", "person");

/* Get all the index namespaces of "person" */
mongo_indexNameSpaces(:c, "tutorial");

/* Extract the namespace from the records */
mongo_indexNameSpaces(:c, "tutorial")["name"];

/* Create 2 B-Tree indexes */
mongo_createIndex(:c, "tutorial", "person", "ageIndex", {"age": 1});
mongo_createIndex(:c, "tutorial", "person", "NameIndex", {"Name": 1});

/* Inspect that there are 3 indexes now */
mongo_indexNameSpaces(:c, "tutorial")["name"];

/* The indexStats() command aggregates statistics for the B-tree data
structure of a MongoDB index.*/
/*run with mongod -enableExperimentalIndexStatsCmd*/
indexStats(:c, "tutorial", "person", "ageIndex" );

/* Utilize the B-Tree index on age */
mongo_get(:c, "tutorial", "person",
  {"age": { "$gt": 10, "$lte": 65 }} );

/* remove age index */
mongo_dropIndex(:c, "tutorial", "person", "ageIndex");

/* check it */
mongo_indexNameSpaces(:c, "tutorial")["name"];

/* remove all indexes */
mongo_dropAllIndex(:c, "tutorial", "person");

/* The indexes _id cannot be deleted: */
mongo_indexNameSpaces(:c, "tutorial")["name"];

/* Add a record with _id 1: */
mongo_add(:c, "tutorial", "person", {"_id":1, "Name": "Olle",
                                               "age": 55});
/* Add another record with _id 1 => noting returned */
mongo_add(:c, "tutorial", "person", {"_id":1, "Name": "Kalle",
                                               "age": 55});
/* Check the duplicate key error */
mongo_getPrevError(:c, "tutorial");

/* Report storage utilization statistics for the specified database.*/
/* The field “dataSize” is size in number of bytes */
mongo_dbStats(:c, "tutorial",  1 );

/* Here “dataSize” is in number of KB */
mongo_dbStats(:c, "tutorial",  1024 );

/* Remove person collection in tutorial database */
mongo_dropCollection(:c, "tutorial", "person");

/* See if it is deleted */
mongo_collNameSpaces(:c, "tutorial")["name"];

/* Remove "tutorial" database */
mongo_dropDB(:c, "tutorial");

/* Disconnect from data source */
mongo_disconnect(:c);
```

Appendix B

MongoDB C Driver 0.8.1 issues

As mentioned in section 3.3, to build a functioning driver, this project has made a Microsoft Visual Studio 2010 project with the implementation of windows socket initialization by restructuring the source code directories of MongoDB C Driver 0.8.1. The project builds and generates the *mongo_driver.dll* and *mongo_driver.dll* in the AmosNT/bin directory which can further be accessed by Amos-Mongo Interface, *mongo_wrapper.dll*. This project has used the preprocessor directives of `MONGO_USE__INT64` and `MONGO_ENV_STANDARD`. It also includes an additional linking dependency with `ws2_32.lib`. As discussed earlier the function `WSAStartup()` is added because an application or DLL can only issue further Windows Sockets functions after successfully calling `WSAStartup()`. The following lines of code have been embedded on line number 500 in `env.c` to resolve the issue.

```
WORD wVersionRequested;
WSADATA wsaData;
/* Use the MAKEWORD(lowbyte, highbyte) macro declared in Windef.h */
wVersionRequested = MAKEWORD(2, 2);

err = WSAStartup(wVersionRequested, &wsaData);
if (err != 0)
{
  /* Tell the user that we could not find a usable */
  /* Winsock DLL.                                  */
  printf("WSAStartup failed with error: %d\n", err);
}
```