Design and Implementation of a MongoDB Driver for Prolog

Sebastian Lundström

Abstract

To make MongoDB and Prolog work together, an intermediate layer is needed to facilitate the communication. Implementing such a layer, or driver, involves handling the network communication and encapsulating the functionality offered by MongoDB in native Prolog predicates. This thesis presents how the fundamentals of such a driver can be designed, developed and used, and provides a discussion on how to further develop the driver and make it ready for production.

Contents

1 Introduction

MongoDB is a document-oriented database system with a strong focus on flexibility, scalability and performance. Document-orientation involves leaving the row-centric concept of the relational database model, and introducing the much more flexible notion of a document. Document-orientation avoids rigid database schemas and also promotes a certain degree of denormalization which allows embedding documents into each other, leading to potentially much better performance by avoiding the need for expensive join operations.

Prolog, being an untyped language, agrees with the document-oriented approach of relaxing manifests in order to create more dynamic and flexible systems. Embedding terms in other terms is natural in Prolog, and embedding documents in other documents is natural in MongoDB.

In order to use MongoDB from a programming language, some kind of interface, or "driver", must sit in-between the database system and the language to facilitate the communication.

1.1 Purpose

The purpose of this thesis is to discuss the design and implementation of a MongoDB driver for Prolog.

1.2 Scope

The aim of this thesis is not to create a feature-complete production-ready driver, but to cover basic CRUD (Create, Read, Update, Delete) functionality and connection handling, laying the foundation for further development. More advanced MongoDB features such as replication and file storage are not covered.

Due to the lack of proper standardization within the Prolog community, maintaining portability between different Prolog implementations is difficult and not actively pursued.

2 Background

This chapter briefly discusses MongoDB and Prolog.

2.1 MongoDB

MongoDB is a document-oriented database management system that emphasizes speed, scalability and flexibility. This is accomplished by avoiding fixed database schemas and, by sacrificing joins, requiring a certain degree of denormalization by embedding documents into each other. See Chodorow and Dirolf (2010) for a more complete discussion.

2.1.1 Document-Orientation

MongoDB employs a document-oriented view of data [1]. Instead of focusing on the "row" as the primary data entity, document-orientation introduces the notion of a "document".

A document in MongoDB resides in a "collection" in which no predefined schema is enforced. Therefore, documents in a collection need not share the same structure. This allows for great flexibility, especially with data migrations, as individual documents can be modified freely without updating a schema and affecting all documents in a collection [1].

A document consists of key/value pairs, where keys are strings and values can be of many different types, including other documents (embedded documents). Allowing recursive structures in the database is a step away from the more traditional concepts used in relational databases, where a high degree of normalization is often desired. Relaxing the need for normalization makes it possible to create highly efficient systems by avoiding expensive join operations [1].

2.1.2 Main Features

Document-Orientation MongoDB is designed to be easy to scale, a goal which is easier to achieve when the relational model is not used. Document-oriented databases are more flexible and impose less restrictions on the layout of data, allowing for complex data structures to be stored within a single document [1].

Scalability The MongoDB approach to scaling involves "scaling out" – the process of adding more commodity servers to a cluster instead of replacing existing servers with better (and more expensive) ones [1]. Adding more machines when the need arises is straight-forward and the database system itself figures out how to best configure the machines.

Low Maintenance Because there are no database schemas, and databases and collections can be created on-the-fly without explicit commands, MongoDB requires very little manual maintenance. The database system is designed to be very automatic [1].

2.1.3 Interactive Shell

MongoDB includes an interactive JavaScript-based shell that can be used to conveniently access the database system [1]. In JavaScript, documents are represented as objects. The example in listing 1 illustrates the beginning of a shell session to create a simple blog database.

Listing 1: Inserting and finding documents.

```
1 $ mongo
2 MongoDB shell version: 1.8.1
3 connecting to: test
4 > use blog
5 switched to db blog
```

```
6  > db.posts.insert(
7      { "title" : "My first post",
8        "content" : "First post!" })
9  > db.posts.insert(
10     { "title" : "Second post",
11       "content" : "Post number 2." })
12 > db.posts.find()
13 { "_id" : ObjectId("4dfb6a5de8d172995e7874d7"),
14   "title" : "My first post",
15   "content" : "First post!" }
16 { "_id" : ObjectId("4dfcbec7e8d172995e7874d8"),
17   "title" : "Second post",
18   "content" : "Post number 2." }
```

Line 4 switches to the database called "blog", but does not create it. Lines 6 and 9 insert two documents into the collection "posts". The first insert is the first command to manipulate data inside the database, so this triggers the actual creation of both the database and the collection. Because no arguments are passed to the "find" call on line 12, it retrieves all documents in the collection, which are the ones we just inserted.

A new field called "_id" has been added to both documents. Unless a document already has such a field, one is automatically generated by the database when it is first inserted. This field must be able to uniquely identify documents within a collection [1].

To add comments to the first blog post, an update can be issued, illustrated in listing 2.

Listing 2: Adding a field to a document.

```
1 > db.posts.update(
2     { "_id" : ObjectId("4dfb6a5de8d172995e7874d7") },
3     { "$set" : { "comments" : [ "Good post!" ] } },
4     false,
```

```
5       false)
```

An update expects four arguments: a document indicating which document(s) to update, a document describing the update to perform, and two booleans indicating whether to perform an "upsert" (see 4.2.5) and if multiple documents should be updated if the query matches more than one.

The document describing the update to perform (the second argument) includes a special field called "$set". This field is a document that contains fields to be set (in this case added) in the updated document. Displaying all documents after performing the update is illustrated in listing 3.

Listing 3: Displaying documents of different structure.

```
1 > db.posts.find()
2 { "_id" : ObjectId("4dfcbec7e8d172995e7874d8"),
3   "title" : "Second post",
4   "content" : "Post number 2." }
5 { "_id" : ObjectId("4dfb6a5de8d172995e7874d7"),
6   "comments" : [ "Good post!" ],
7   "content" : "First post!",
8   "title" : "My first post" }
```

The updated document now contains an embedded array with a single comment. Listing 4 shows what happens if another comment is added. This time, a special field called "$push" is used which pushes a value onto the end of an array.

Listing 4: Adding to an array.

```
1 > db.posts.update(
2     { "_id" : ObjectId("4dfb6a5de8d172995e7874d7") },
3     { "$push" : { "comments" : "Interesting." } },
4     false,
5     false)
6 > db.posts.find()
```

```
7  ...
8  { "_id" : ObjectId("4dfb6a5de8d172995e7874d7"),
9    "comments" : [ "Good post!", "Interesting." ],
10   "content" : "First post!",
11   "title" : "My first post" }
```

2.1.4 BSON

At the core of MongoDB lies the BSON (Binary JSON) [8] data format which is used to communicate data as well as store data on disk. When a driver communicates with a MongoDB instance, documents are transmitted over the network as a series of BSON encoded bytes.

BSON is similar to and heavily influenced by JSON (JavaScript Object Notation) [9], the main difference being that BSON is a binary format whereas JSON is plain-text. The reason for using a binary format is efficiency. With JSON, numbers need to be converted to and from text in order to be used. This kind of conversion is generally slow, and BSON therefore extends the JSON model with a set of data types that can be parsed directly. For instance, a 32-bit integer is encoded in BSON as four consecutive little-endian bytes. This wastes some space for small integers, but is much faster to parse [8].

2.2 Prolog

Prolog is an interpreted and dynamic language with its roots in logic. It is also a declarative language, implying that the programmer specifies *what* the program is supposed to solve, not *how* [3].

This section gives a small introduction to Prolog in order to more easily appreciate the code examples in subsequent sections. See Clocksin and Mellish (1994) or Bratko (2001) for more complete discussions.

2.2.1 Type System

The type system used in Prolog is dynamic, and the fundamental data type is the "term", which can be any value [2]. Being dynamic means that variables can be bound to anything and need not be explicitly assigned a fixed type.

2.2.2 Data Structures

The main primitives in Prolog are *integer*s, *floats* and *atom*s. In SWI-Prolog, integers are unbounded and floats are represented as 64-bit IEEE 754 doubles. Atoms represent UTF-8 (Unicode Transformation Format, 8-bit) encoded text [10]. Listing 5 exemplifies the syntax for integers, floats and atoms.

Listing 5: Primitives in Prolog.

```
1 42
2 5.5
3 atom
4 'a longer atom'
```

A *structure* is a compound value with a fixed number of components and possibly a name [2]. The components are terms that need not share the same type and can be anything, including variables and other structures. Structures are often used to represent trees. Structures are shown in listing 6.

Listing 6: Structures in Prolog.

```
1 person(john, 32)
2 (nameless, structure)
```

A *list* is a sequence of zero or more terms which need not share the same type [2]. A non-empty list is a recursive structure that consists of a "head" which is the first element, and a "tail" which is itself a list of all elements except the first. If a list has only one element, its tail is the empty list. The empty list has neither head nor tail. Listing 7 shows an empty and a non-empty list.

Listing 7: Lists in Prolog.

```
1 []
2 [1, hello(world), 5.5]
```

2.2.3 Unification

In Prolog, there is no conventional "assignment" operation. Instead, value propagation is performed by a mechanism called "unification", or "matching". Unification is the process of trying to make two terms equal [2], and is usually performed by the "equals" infix operator.

Variables are denoted by an uppercase initial. A variable can be bound to any term, but once bound, it is immutable and cannot be rebound.

The unification `X = hello` succeeds, and the variable `X` is bound, or instantiated, to the atom `hello`. The unification `[1,2,3,4,5] = [H|T]` will result in `H` being bound to the integer `1`, and `T` being bound to the list `[2,3,4,5]`. The vertical bar is used to indicate that everything to the right of it represents the tail of the list, in other words, any and all elements that are not explicitly unified from the start of the list. This is necessary in order to work with lists of unknown length. The unification `A = B` binds the variables together, so as soon as one of them is bound to a value later on, the other will be bound to the same thing.

An unsuccessful unification is for instance `[a,b,c] = [a,b,d]`. The lengths of the lists match, but the atoms `c` and `d` are different and cannot be unified. The unification `foo(a,b,c) = foo(A,A,C)` fails because this would require `a` and `b` to be the same atom.

2.2.4 Program Structure

The fundamental unit of code in Prolog is the predicate (also called relation) [2]. A predicate consists of one or more clauses, together forming a logical disjunction. The value of any predicate is either true or false, indicating

whether the predicate succeeded. Any other output must be specified as output-arguments that become bound by the predicate before it returns. The convention used for argument order is input-arguments first, intermediate values in the middle, and output-arguments last [5].

A program to calculate the number of elements in a list is given in listing 8. This predicate follows the conventional argument order and uses an "accumulator" to construct the final result. An accumulator is a fabricated extra argument that holds the state of something produced "so far" [2].

Listing 8: A program relating a list to its length, using an accumulator.

```
1 length([], N, N).
2 length([_|T], N0, N) :-
3     N1 is N0 + 1,
4     length(T, N1, N).
```

The program in listing 8 takes three arguments: a list, an accumulator used to successively accumulate the length, and the length of the list. To calculate the length, the third argument is passed as an unbound variable. On the other hand, if the length is instead a value, the predicate verifies if the relation between the list and the given length holds, and fails (returns false) otherwise.

Line 1 holds the first clause, which is a base (non-recursive) case that states that the length of an empty list is the same as the value of the accumulator. This means that the initial value of the accumulator must be zero in order for the relation to be meaningful.

Line 2 starts the second clause, which deals with non-empty lists. The list construct `[_|T]` states that the head of the list (first element) is bound to an anonymous variable and the tail of the list (all but the first element) is bound to the variable `T`. The variable `N1` is bound to the value of the current accumulator plus one, using the *is*-operator to perform the arithmetic. The "plus one" indicates that we have successfully counted one element, the head of the list. The predicate then continues by invoking itself recursively, passing

the tail of the list, the updated accumulator, and the final length of the list. The final length is bound to the final value of the accumulator as soon as the tail of the list is empty and the base case can be matched.

Execution Example In an interactive Prolog environment, the predicate in listing 8 may be used as follows, providing a starting value of zero for the accumulator. The example in listing 9 is illustrated using a "trace", a debugging tool used extensively in Prolog to show each step taken to satisfy a query.

Listing 9: Trace of a call to *length/3*.

```
1  [trace]  ?- length([a,b,c], 0, Length).
2     Call: (6) length([a, b, c], 0, _G391) ? creep
3  ^  Call: (7) _G476 is 0+1 ? creep
4  ^  Exit: (7) 1 is 0+1 ? creep
5     Call: (7) length([b, c], 1, _G391) ? creep
6  ^  Call: (8) _G479 is 1+1 ? creep
7  ^  Exit: (8) 2 is 1+1 ? creep
8     Call: (8) length([c], 2, _G391) ? creep
9  ^  Call: (9) _G482 is 2+1 ? creep
10 ^  Exit: (9) 3 is 2+1 ? creep
11    Call: (9) length([], 3, _G391) ? creep
12    Exit: (9) length([], 3, 3) ? creep
13    Exit: (8) length([c], 2, 3) ? creep
14    Exit: (7) length([b, c], 1, 3) ? creep
15    Exit: (6) length([a, b, c], 0, 3) ? creep
16 Length = 3.
```

In listing 9, line 1 shows the query being posed, which is to find the length of a list of three elements. Line 2 shows the first call being made, which matches the second clause of the predicate due to the non-empty list. Variables are renamed internally by Prolog and can be identified by the "_G"

prefix. The initial accumulator of zero is incremented by one and bound to the variable `_G476` on line 3. Line 4 indicates with the word "exit" that this call succeeded. Line 5 makes a recursive call with the tail of the list, the incremented accumulator and the same unbound variable used for the final result. This recursive process is repeated until line 11, where the tail of the list contains no elements.

On line 11, the first clause (the base case) of the predicate matches the empty list and so the resulting length is bound to the value of the accumulator, which is 3. Because the base case has been reached, no more steps are needed to satisfy the query. The recursion unfolds, showing the final bindings for each successful call on the way back to the initial call. The last argument is bound to the value 3 which is the expected answer.

2.2.5 Anonymous Variables

It is often necessary to make a unification in order to access certain parts of a structure, but without caring for all the bindings created. In these cases, anonymous, or "don't care", variables can be used. These are denoted by an initial underscore. The predicate in listing 8 uses such a variable because the actual value of the list head is of no importance. It suffices to know that an element exists.

2.2.6 Pattern Matching and Branching

When invoking a predicate, a process called "pattern matching" is initiated [2]. Prolog tries to unify the given arguments with the formal parameters of the first clause of the predicate. If unification succeeds, Prolog proceeds to evaluate any premises present in the clause. If unification fails, Prolog reverts the effects of the unification so far, and repeats the process with the next clause of the predicate. This is called "backtracking" and is the Prolog equivalent to the *if*-statement.

Calling the predicate in listing 8 with a non-empty list will make Prolog first try to unify with the first clause. This fails, and the second clause is backtracked into. Should this clause also fail for some reason, the next clause after that would be tried, and so on, until no more clauses are found and the predicate fails.

2.2.7 Recursion

Repetition is performed by recursion [2]. The predicate in listing 8 is a recursive predicate because it is defined in terms of itself. To solve the problem of calculating the length of a list, the predicate solves a smaller problem, namely calculating the length of a shorter list: the tail. It works its way to the base case and eventually arrives at a solution.

2.2.8 Documentation Convention

When documenting Prolog predicates, each argument is usually prefixed with a symbol that indicates how the argument is supposed to behave [5]. The following documentation convention is employed by subsequent chapters.

+	*Input.* Argument must be already instantiated.
-	*Output.* Argument must not be already instantiated, but become instantiated by the predicate.
?	[XXX used?] Argument may or may not be instantiated. This is commonly used when the predicate can be used to either verify the argument or instantiate it.

A predicate is referred to by its name and arity (number of arguments), and, if applicable, its module: *module:predicate/arity*. Should the "length" predicate in listing 8 be defined in the module "lists", it would be referred to as *lists:length/3*.

3 Requirements

This chapter outlines the functionality implemented by the driver.

3.1 Suggested Features

The *MongoDB Driver Requirements* [6] list the following items as high priority for a driver to provide, in no particular order:

1. BSON serialization/deserialization,
2. full cursor support,
3. close exhausted cursors,
4. support for running database commands,
5. handle query errors,
6. convert all strings to UTF-8 (part of proper support for BSON),
7. hint, explain, count, $where,
8. database profiling,
9. advanced connection management, and
10. automatic reconnection.

3.2 Selected Features

To keep development focused on providing just enough functionality to use the database system as data storage and retrieval, the implementation focuses on items 1-6 and omits items 7-10 from the list in section 3.1. The selected requirements are further described in this section.

3.2.1 BSON Conversion

The driver must be able to convert back and forth between some idiomatic Prolog structure and BSON bytes.

3.2.2 Connection Management

The driver must be able to obtain a connection to a database server using a TCP (Transmission Control Protocol) socket. All communication with the database is performed through this socket.

3.2.3 Database Commands

The driver must be able to execute arbitrary database commands, such as creating and dropping collections.

3.2.4 Document Handling

The driver must expose functionality to find, insert, update and delete documents in a collection, providing basic CRUD coverage.

4 Design

This chapter discusses how documents are represented in Prolog, and describes the Application Programming Interface (API) of the driver.

4.1 Documents in Prolog

One advantage of working in a high-level language such as Prolog is the ability to express complex structures as literals. Most functions in MongoDB work with documents directly, so being able to easily construct and pass around document literals is helpful.

A BSON document is represented in Prolog as a list of key/value pairs. A pair is a structure named '-' (dash) with two arguments, a key and a value. Using a symbol as the name of the structure makes it possible to write it as an infix operator, avoiding the need for parentheses and making the key/value association more natural. Keys are UTF-8 encoded atoms, and values can be of several different types. Listing 10 illustrates a document that embeds other documents.

Listing 10: BSON document as a Prolog literal.

```
[
    '_id' - object_id('4dfb6a5de8d172995e7874d7'),
    name - 'Prolog',
    type - declarative,
    major_implementations -
        [
            [
                name - 'SWI-Prolog',
                web - 'http://www.swi-prolog.org/'
            ],
            [
                name - 'SICStus Prolog',
                web - 'http://www.sics.se/sicstus/'
            ]
        ]
]
```

4.2 Driver API

This section details the interface for the driver.

4.2.1 Server Connection

Obtaining a Connection A connection to a MongoDB server can be established by the predicate *new_ connection/1,3*. Two versions of the predicate are provided, one which assumes the default MongoDB host and port, and one which allows this to be specified. If an error occurs during the setup, an exception is thrown. The predicate interfaces are described in table 1 and table 2.

Argument	Description
-Connection	Connection established to the database server

Table 1: Interface to *new_ connection/1*.

Argument	Description
+Host	Host name of database server
+Port	Port number of database server
-Connection	Connection established to the database server

Table 2: Interface to *new_ connection/3*.

Releasing a Connection When a connection is no longer needed it must be properly released back to the system. The predicate *free_ connection/1* will release any resources associated with the given connection, rendering it unusable. Its interface is described in table 3.

Argument	Description
+Connection	Connection to release

Table 3: Interface to *free_ connection/1*.

4.2.2 Databases and Collections

A MongoDB server can contain multiple logical databases, and each database can contain multiple document collections. Once a connection to the server

has been established, a handle to a logical database can be produced by the predicate *get_ database/3*. The database need not exist beforehand as it will be created when first used. The predicate is described in table 4.

Argument	Description
+Connection	Connection handle
+DatabaseName	Name of the database to use
-Database	Database handle

Table 4: Interface to *get_ database/3*.

The database handle can be used to execute general database commands, but a collection handle must be retrieved when working with documents. This is accomplished by the predicate *get_ collection/3*, which returns a collection handle. The collection need not exist beforehand as it will be created when first used. Its interface is described in table 5.

Argument	Description
+Database	Database handle
+CollectionName	Name of the collection to use
-Collection	Collection handle

Table 5: Interface to *get_ collection/3*.

4.2.3 Find Documents

Querying the database for documents is accomplished through the family of "find" predicates. Depending on the nature of the query, different predicates can be used to retrieve, for example, a single document or all documents matching the query.

In general, a query returns documents in batches. A certain number of documents are returned with the query response, but if there are too many documents to fit in a single response, a "cursor" is established. The cursor can then be used to query for further documents.

The three main predicates for finding documents are *find/8*, *find_one/4* and *find_all/4*. The most general predicate is *find/8* which takes many arguments and offers very fine-grained control over the query. The interface to the predicate is described in table 6.

Argument	Description
+Collection	Collection handle
+Query	Document describing which documents to match
+ReturnFields	Document describing which fields to return
+Skip	Number of documents to skip
+Limit	Number of documents to return
+Options	List of option atoms
-Cursor	Cursor used to fetch more documents
-Docs	First batch of documents returned by the query

Table 6: Interface to *find/8*.

The predicate *find_one/4* is intended for the common case where you only expect one matching document. If no matching document is found, the returned document is the atom 'nil'. The interface is described in table 7.

Argument	Description
+Collection	Collection handle
+Query	Document describing which document to match
+ReturnFields	Document describing which fields to return
-Doc	Document returned by the query

Table 7: Interface to *find_one/4*.

The predicate *find_all/4* is intended for situations where all matching documents need processing and it is convenient to collect them into a single (potentially large) list. The interface is described in table 8.

Using Cursors When a query matches more documents than can be returned in a single response, a cursor is established and returned. A cursor is

Argument	Description
+Collection	Collection handle
+Query	Document describing which documents to match
+ReturnFields	Document describing which fields to return
-Docs	All documents returned by the query

Table 8: Interface to *find_all/4*.

created and maintained by the database server, and it represents a pointer to a subset of the documents matched by a query [1].

After making a query, the predicate *cursor_has_more/1* can be called on the given cursor. The predicate succeeds if more documents may be retrieved, and its interface is described in table 9.

Argument	Description
+Cursor	Cursor to investigate

Table 9: Interface to *cursor_has_more/1*.

If more documents need to be retrieved, the predicate *cursor_get_more/4* can be used. This predicate takes a cursor, returns a number of documents and also a new cursor handle representing the possibly modified state of the cursor. The interface is described in table 10.

Argument	Description
+Cursor	Cursor to investigate
+Limit	Number of documents to return
-Docs	Documents returned by the query
-NewCursor	New state of Cursor

Table 10: Interface to *cursor_has_more/1*.

When a cursor is no longer needed, it must be explicitly killed in order for the database to clean up any resources claimed by it. This is accomplished by the predicate *cursor_kill/1*, described in table 11.

Argument	Description
+Cursor	Cursor to kill

Table 11: Interface to *cursor_kill/1*.

If several cursors need to be killed, a more efficient version called *cursor_kill_batch/1* can be used to kill them in a single database call. This predicate is described in table 12.

Argument	Description
+Cursors	List of cursors to kill

Table 12: Interface to *cursor_kill_batch/1*.

Should all remaining documents matched by a query need to be retrieved in a single step, the predicate *cursor_exhaust/2* can be used. This is a convenience predicate that simply queries the cursor until all documents have been retrieved, and returns them as a single list. A cursor exhausted by this predicate need not be manually killed. The interface is described in table 13.

Argument	Description
+Cursor	Cursor to exhaust (and automatically kill)
-Docs	All remaining documents pointed to by the cursor

Table 13: Interface to *cursor_exhaust/2*.

4.2.4 Insert Documents

Two predicates are available to insert documents into a database collection. The simplest one is *insert/2*, which accepts a collection handle and a document to insert, as described in table 14.

If several documents need to be inserted, a more efficient predicate called *insert_batch/3* may be used to send them in a single database call. This predicate is described in table 15.

Argument	Description
+Collection	Collection handle
+Doc	Document to insert

Table 14: Interface to *insert/2*.

Argument	Description
+Collection	Collection handle
+Options	List of option atoms
+Docs	List of documents to insert

Table 15: Interface to *insert_batch/3*.

4.2.5 Update Documents

Updating a single document is done using the predicate *update/3*. It accepts a collection handle, a selector document and a document describing the update to perform on the matched document. If several documents match the selector, only the first one is used. The interface is described in table 16.

Argument	Description
+Collection	Collection handle
+Selector	Document describing which document to match
+Modifier	Document describing the update to perform

Table 16: Interface to *update/3*.

If all matching documents need to be updated, a similar predicate called *update_all/3* can be used. The interface to this predicate is identical to the one described in table 16, but it performs the update on all matching documents instead of just the first.

A common use case is the combination of an update and an insert, sometimes called an "upsert". This predicate is called *upsert/3*, and works like a normal update (described in table 16) except that if the document does not exist it is inserted instead of updated [1].

4.2.6 Delete Documents

Deleting documents from a collection is accomplished through the predicates *delete/2* and *delete/3*. Both predicates accept a collection handle and a "selector" document that describes which documents to match and delete. All matching documents are deleted unless the option 'single_remove' is supplied. The predicate interfaces are described in table 17 and table 18.

Argument	Description
+Collection	Collection handle
+Selector	Document describing which documents to delete

Table 17: Interface to *delete/3*.

Argument	Description
+Collection	Collection handle
+Selector	Document describing which documents to delete
+Options	List of option atoms

Table 18: Interface to *delete/3*.

4.2.7 Database Commands

To query a database for information that does not deal with individual documents, a predicate called *command/3* is provided. This predicate executes arbitrary database commands, such as dropping a collection, and is described in table 19.

Argument	Description
+Database	Database handle
+Query	Document describing the command to execute
-Doc	Document describing the result of the command

Table 19: Interface to *command/3*.

4.2.8 Error Handling

Errors issued by a failed database query is detected by the driver and an exception is thrown that contains the error document returned by the database. The error document contains a key called "$err" that maps to an atom describing the nature of the error.

It is sometimes useful to get more information on a query than returned by the query itself. The predicate *get_last_error/2* returns a document describing the status of the last query executed through the same connection, and it is described in table 20.

Argument	Description
+Database	Database handle
-Doc	Document describing the status of the last query

Table 20: Interface to *get_last_error/2*.

5 Usage Example

This chapter shows a small but complete example of how to use the driver by implementing a simple "to do" application, illustrated in listing 11.

Listing 11: Simple "to do" application.

```
1 :- use_module(mongo(mongo)).
2
3 todo :-
4     format('--- Simple Todo ---~n'),
5     mongo:new_connection(Connection),
6     mongo:get_database(Connection, todo, Database),
7     mongo:get_collection(Database, items, Collection),
8     action(list, Collection),
9     mongo:free_connection(Connection).
```

```
10
11 action(list, Collection) :- !,
12     list_items(Collection),
13     new_action(Collection).
14 action(add, Collection) :- !,
15     add_item(Collection),
16     new_action(Collection).
17 action(delete, Collection) :- !,
18     delete_item(Collection),
19     new_action(Collection).
20 action(quit, _Collection) :- !,
21     format('Bye!~n').
22 action(_Unknown, Collection) :-
23     format('Unknown alternative.~n'),
24     new_action(Collection).
25
26 new_action(Collection) :-
27     format('~nEnter list/add/delete/quit: '),
28     read(Action),
29     action(Action, Collection).
30
31 list_items(Collection) :-
32     mongo:find_all(Collection, [], [], Docs),
33     print_items(Docs).
34
35 print_items(Docs) :-
36     format('Id~26|Label~45|Priority~n'),
37     print_items_aux(Docs).
38
39 print_items_aux([]).
```

```
40 print_items_aux([Doc|Docs]) :-
41     bson:doc_get(Doc, '_id', object_id(Id)),
42     bson:doc_get(Doc, label, Label),
43     bson:doc_get(Doc, priority, Priority),
44     format('~w~26|~w~45|~w~n', [Id,Label,Priority]),
45     print_items_aux(Docs).
46
47 add_item(Collection) :-
48     format('Label: '),
49     read(Label),
50     format('Priority: '),
51     read(Priority),
52     Doc = [label-Label,priority-Priority],
53     mongo:insert(Collection, Doc).
54
55 delete_item(Collection) :-
56     format('Id: '),
57     read(Id),
58     mongo:delete(Collection, ['_id'-object_id(Id)]).
```

The first line makes sure that the driver is properly loaded into the Prolog environment. The *todo/0* predicate is used to start the program. It establishes a database connection and obtains a handle to the collection "items" residing in the database "todo". It then enters an "action" loop and eventually releases the connection when the loop stops.

The *action/2* predicate is the engine of the program, as it triggers the user to continuously enter new actions, until the "quit" action is given. Valid actions are "list", "add", "delete" and "quit."

The "list" action fetches all documents stored in the collection and displays them in a formatted table. The "add" action asks the user to detail a new item, and inserts it into the collection. The "delete" command asks the user

for an object identifier and deletes the corresponding document from the collection.

An example session is depicted in listing 12.

Listing 12: Using the "to do" application.

```
?- todo.
--- Simple Todo ---
Id                          Label              Priority

Enter list/add/delete/quit: add.
Label: 'Make tea'.
Priority: 1.

Enter list/add/delete/quit: add.
Label: 'Go for a walk'.
Priority: 2.

Enter list/add/delete/quit: list.
Id                          Label              Priority
4dff66bd4c594ffa3e17cb70  Make tea           1
4dff66eb4c594ffa3e17cb71  Go for a walk      2

Enter list/add/delete/quit: delete.
Id: '4dff66eb4c594ffa3e17cb71'.

Enter list/add/delete/quit: list.
Id                          Label              Priority
4dff66bd4c594ffa3e17cb70  Make tea           1

Enter list/add/delete/quit: quit.
Bye!
```

6 Implementation

This chapter describes how the design is implemented and what tools and techniques are used.

6.1 Module Organization

The driver is organized into two primary modules: *mongo* and *bson*. The mongo module is the main module which contains all the functionality needed to work with a MongoDB instance. This is the module that needs to be included in order to use the driver. The bson module exposes functionality to handle BSON conversions between documents and bytes, and also provides basic document manipulation predicates. The bson module is used internally by the mongo module, but implemented separately because BSON is not inherent to MongoDB and can be used on its own.

6.2 Grammar Rules

Most of the code that converts to or from lists is implemented using the Definite Clause Grammar (DCG) syntax of Prolog [2]. This syntax provides a convenient interface to "difference lists", which is a common technique in Prolog to avoid unnecessary (and inefficient) list concatenation [4].

6.3 Network Communication

Communication with a MongoDB server instance is conducted through a TCP socket using a certain protocol [7].

6.3.1 Sockets

Sockets are handled using the socket library provided by SWI-Prolog [10]. The predicate *new_connection/1,3* (described in section §4) creates a socket using *tcp_socket/1* and initializes it using *tcp_connect/4* which opens a

read stream as well as a write stream. If an error is encountered during this process, an exception is thrown and the socket is destroyed.

The predicate *free_connection/1* (described in section §4) closes both streams and releases the socket using the library predicate *close/2*.

6.3.2 Wire Protocol

The protocol used to facilitate the network communication defines the following message types [7]:

OP_REPLY Server reply to a client message.

OP_MSG Deprecated type for diagnostic messages.

OP_UPDATE Update documents.

OP_INSERT Insert documents.

OP_QUERY Query a collection for documents.

OP_GET_MORE Retrieve more documents from a cursor.

OP_DELETE Delete documents.

OP_KILL_CURSORS Kill cursors.

Message Header All messages start with a message header which is defined as four 32-bit integers:

1. length of entire message in bytes,
2. request identifier, to uniquely identify a particular message,
3. response identifier, sent only in database replies to associate with a request identifier, and,
4. operation code, describing the type of the message following the header.

The code for constructing a message header uses the DCG syntax and is defined in the internal module *mongo_bytes*. This module contains predicates for converting various values into byte sequences, such as *int32/1* for converting an integer into four little-endian bytes, and *c_string/1* for converting an atom into a zero-terminated byte list.

Listing 13: Constructing a message header.

```
1 header(RequestId, ResponseTo, OpCode) -->
2     [_,_,_,_], % Length of entire message.
3     int32(RequestId),
4     int32(ResponseTo),
5     int32(OpCode).
```

Line 2 in listing 13 creates a list of four (anonymous) variables. These represent the length of the entire message and are to be instantiated later as four bytes of a 32-bit little-endian integer. As soon as all bytes comprising the message have been created, they are counted and the length instantiated.

Example Message An example of how to construct an entire message is listed in listing 14. This example shows an *OP_QUERY* message.

Listing 14: Constructing an *OP_QUERY* message.

```
1  build_bytes_for_find(
2      Namespace, Query, ReturnFields,
3      Skip, Limit, Flags)
4  -->
5      mongo_bytes:header(0, 0, 2004),
6      mongo_bytes:int32(Flags),
7      mongo_bytes:c_string(Namespace),
8      mongo_bytes:int32(Skip),
9      mongo_bytes:int32(Limit),
10     mongo_bytes:bson_doc(Query),
```

```
11      mongo_bytes:bson_doc(ReturnFields).
```

Line 5 in listing 14 calls the predicate defined in listing 13, and passes as third argument an integer representing the message type *OP_QUERY*. The current implementation does not handle request or response identifiers, so these are simply filled with the value zero. The rest of the calls construct the actual contents of the message, converted to bytes in suitable ways.

When the message is constructed, its length is instantiated and the message can be sent over the connection socket.

6.4 BSON Handling [XXX rewrite this section]

At the core of MongoDB lies the BSON binary data format which is used to communicate data as well as store data on disk. When communicating with a MongoDB instance, documents are transmitted over the network as bytes of BSON. The driver converts Prolog structures into series of BSON bytes before sending them to the database, and database responses are converted back to Prolog structures again.

6.4.1 Document-BSON Conversion

When converting a document to BSON, a list of bytes is obtained.

When converting between structured documents and BSON encoded bytes, the relation BSON:DOC_BYTES/2 can be used. This is a two-way relation that accepts a document and returns bytes, or accepts bytes and returns a document. XXX

Given a list of BSON encoded bytes, the relation BSON:DOC_BYTES/2 can be used to convert it into its structured equivalent. The empty document is represented in BSON as five bytes, the first four being a 32-bit little-endian integer representing the length of the entire document, and the last byte signaling the end of the document with a zero.

```
?- bson:doc_bytes(Doc, [12,0,0,0,16,97,0,42,0,0,0,0]).
Doc = [a-42]
```

If a series of BSON bytes represents a concatenation of documents, the relation BSON:DOCS_BYTES/2 can be used. The following yields an empty and a non-empty document wrapped in a list.

```
?- bson:docs_bytes(
       Docs,
       [5,0,0,0,0,12,0,0,0,16,97,0,42,0,0,0,0]).
Docs = [[],[a-42]]
```

An exception is thrown if the conversion fails [XXX detail the exception?].

6.4.2 Document Manipulation

To make working with documents easier, a set of simple helper predicates are provided.

bson:doc_is_valid(+Doc) True if Doc is a valid document and can be converted into BSON encoded bytes.

bson:doc_empty(?Doc) True if Doc is an empty BSON document. Can be used to both check for emptiness as well as obtain an empty document.

bson:doc_get(+Doc, +Key, ?Value) True if Value is the value associated with Key in Doc or +null if Key cannot be found.

bson:doc_get_strict(+Doc, +Key, ?Value) True if Value is the value associated with Key in Doc. Fails if Key is not found or does not match Value.

bson:doc_put(+Doc, +Key, +Value, ?NewDoc) True if NewDoc is Doc with the addition or update of the association Key-Value.

bson:doc_delete(+Doc, +Key, ?NewDoc) True if NewDoc is Doc with the association removed that has Key as key. At most one association is removed. No change if Key is not found.

bson:doc_keys(+Doc, ?Keys) True if Keys is the keys for the associations in Doc.

bson:doc_values(+Doc, ?Values) True if Values is the values for the associations in Doc.

bson:doc_keys_values(+Doc, ?Keys, ?Values) True if Doc is the list of successive associations of Keys and Values.

6.5 C Extension

A small part of the BSON handling is written in C instead of Prolog. This part involves conversion of integers and floats to and from bytes. The two reasons for this are efficiency and ease of programming.

Being able to subvert the type system in C makes it trivial to populate a fixed-width number with individual bytes and then interpret them as a whole. This is especially convenient when converting between bytes and floating-point numbers.

BSON conversion is something the driver does very often, and pushing common operations down to a lower-level language is likely more efficient.

6.6 Test Suite

Unit testing is conducted using *PlUnit,* a unit testing framework built into SWI-Prolog and designed to be compatible with other Prolog implementa-

tions [11].

6.7 Environment

The technical environment used during development is as follows:

POSIX The driver is developed on a POSIX (Portable Operating System Interface for Unix) compliant system.

Prolog The driver is developed and tested on SWI-Prolog 5.10.2.

C Compiler A C compiler is required to build parts of the low-level BSON handling.

MongoDB The driver is tested on MongoDB versions 1.8.0 through 1.8.2.

7 Conclusions and Future Work

The driver enables MongoDB to be used from Prolog as a basic data storage system. The major functionality of finding, inserting, updating and deleting documents is present. Arbitrary database commands can be executed, providing the possibility to administrate the database system.

More advanced functionality offered by MongoDB such as dealing with "replica sets" [1] and automatic server reconnection is not implemented. This makes the current state of the driver only useful for small scale applications without need for great reliability and availability. This is the most obvious area for future development.

Making the driver compatible between different Prolog implementations requires some work. The UTF-8 handling makes extensive use of a facility in SWI-Prolog called "memory files" [10] which are unlikely to be similar or even exist in other implementations.

Another compatibility challenge is most likely the C extension. SWI-Prolog has a solid mechanism for working with C, including a compiler front-end to properly glue Prolog and C together. Other implementations might not provide this.

On the other hand, incorporating C code has the benefit of performance. The company behind MongoDB maintains a permissively licensed open source implementation of BSON written in C [8]. Replacing the Prolog implementation with this C library would lighten the code base considerably and probably make the driver faster.

The module system described in the current Prolog standard is considered flawed and therefore not implemented by most Prologs [10]. The driver is built around the module system present in SWI-Prolog, which is similar to those of other popular Prologs, such as SICStus Prolog, Ciao and YAP. Making the module usage completely compatible with these systems might require some work.

Another obvious path of improvement is making the driver thread-safe. XXXXXX

[XXX How to improve in the future? What is missing for it to be a "real" driver? Discuss choice of list as BSON document. Maybe use some map-like structure instead.]

References

[1] Chodorow, K. & Dirolf, M. (2010) *MongoDB: The Definitive Guide.* Sebastopol, United States of America: O'Reilly Media, Inc.

[2] Clocksin, W. F. & Mellish, C. S. (1994) *Programming in Prolog.* 4th ed. New York, United States of America: Springer-Verlag.

[3] Bratko, I. (2001) *Prolog Programming for Artificial Intelligence.* 3rd ed. Essex, England: Pearson Education Limited.

[4] O'Keefe, R. A. (1990) *The Craft of Prolog.* Cambridge, United States of America: The MIT Press.

[5] Covington, M. A. et al. "Some Coding Guidelines for Prolog." 3 Sep. 2009. Accessed 2011-06-19. `http://www.ai.uga.edu/mc/plcoding.pdf`

[6] Mongo Driver Requirements. Accessed 2011-06-19. `http://www.mongodb.org/display/DOCS/Mongo+Driver+Requirements`

[7] Mongo Wire Protocol. Accessed 2011-06-19. `http://www.mongodb.org/display/DOCS/Mongo+Wire+Protocol`

[8] BSON Specification. Accessed 2011-06-07. `http://bsonspec.org/`

[9] JSON Specification. Accessed 2011-06-19. `http://www.json.org/`

[10] SWI-Prolog Reference Manual. Accessed 2011-06-19. `http://www.swi-prolog.org/pldoc/refman/`

[11] Prolog Unit Tests. Accessed 2011-06-19. `http://www.swi-prolog.org/pldoc/package/plunit.html`