Design and Implementation of a MongoDB Driver for Prolog

Sebastian Lundström

June 18, 2011

Abstract

[XXX Prolog is good stuff, and MongoDB ...]

Flesh out MongoDB, examples are nice (make big references to Mongo and Prolog)

Make Design into API Design, very straight-forward

Insert Usage Example

Make Implementation discuss lots more on how things are implemented, like BSON parsing, socket communication etc.

Contents

1 Introduction

[XXX Is the following more suitable as Abstract material? If so, what should the introduction contain?]

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. [XXX ref: MongoDB: The Definitive Guide, p 4]

2.1.1 Document-Orientation

[XXX lots of references missing here right now]

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

A document in MongoDB resides in a "collection", and no predefined schema is enforced on the collection. 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.

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.

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. [**?**][XXX ref? Mongo book p 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. [xxx ref mongodb p 2] Adding more machines when the need arises is straight-forward and the database system itself figures out how to best configure the machines. [xxx ref mongodb p 3]

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. [xxx ref mongodb]

2.1.3 Interactive Shell

MongoDB includes an interactive JavaScript-based shell that can be used to conveniently access the database system [XXX ref mongo shell]. The example in listing 1 illustrates the beginning of a shell session to create a simple blog database.

Listing 1: Inserting and finding documents.

```
$ mongo
MongoDB shell version: 1.8.1
connecting to: test
> use blog
switched to db blog
> db.posts.insert(
    { "title" : "My first post",
      "content" : "First post!" })
> db.posts.insert(
    { "title" : "Second post",
      "content" : "Post number 2." })
> db.posts.find()
{ "_id" : ObjectId("4dfb6a5de8d172995e7874d7"),
  "title" : "My first post",
  "content" : "First post!" }
{ "_id" : ObjectId("4dfcbec7e8d172995e7874d8"),
  "title" : "Second post",
  "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.

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" [XXX footnote?] 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.

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) [XXX ref bson] 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) [XXX ref json], 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. [xxx ref bsonspec.org faq?]

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*.

This section gives a small introduction to Prolog in order to more easily appreciate the code examples in subsequent sections.

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. A variable can be bound to any term, but once bound, it is immutable and cannot be rebound.

2.2.2 Data Structures

[XXX lots of missing refs]

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 encoded text [xxx ref SWI].

A *structure* is a compound value [xxxxxxxxxx]

A *list* is a sequence of zero or more terms which need not share the same type. 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.

2.2.3 Unification

XXXXXXXXXXXXX

Variables are recognized by an uppercase initial.

xxxxxxxxxxx Anonymous variables, or "don't care" variables, are denoted by an initial underscore. [xxx ref] Anonymous variables are used when xx struct forces xxx

XXX rewrite this:

The first step in determining which clause to use for a certain set of input is accomplished through "unification", sometimes called "pattern matching". Prolog tries to unify the input with the formal parameters of the first clause of the predicate. If the unification fails, the process is repeated with the second clause, and so on. Invoking the predicate LENGTH shown in 2.2.4 with a non-empty list causes unification to fail in the first clause (an empty list cannot be unified with a non-empty list), but succeed in the second (the first element is selected but ignored, and the tail is bound to the variable T).

2.2.4 Program Structure

The fundamental unit of code in Prolog is the predicate (also called relation). 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 [xxx ref prolog guidelines].

A program to calculate the number of elements in a list is given in listing 5. 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" [XXX ref].

Listing 5: 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 5 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 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 addition. 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 5 may be used as follows, providing a starting value of zero for

the accumulator. The example in listing 6 is illustrated using a "trace", a debugging tool used extensively in Prolog to show each step taken to satisfy a query.

Listing 6: 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 6, 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 Control Flow

Branching is performed by "backtracking". Prolog works in a top-down, depth-first fashion when evaluating predicates [XXX ref]. Calling the predicate *length/3* in listing 5 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.

Iteration is performed by recursion. The predicate *length/3* in listing 5 is a recursive predicate because it is defined in terms of itself [XXX ref]. 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.

2.2.6 Documentation Convention

When documenting Prolog predicates, each argument is usually prefixed with a symbol that indicates how the argument is supposed to behave. 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.

? 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 usually referred to by its name and arity (number of arguments), and, if applicable, its module: MODULE:PREDICATE/ARITY. If the LENGTH predicate in section 2.2.4 would be defined in the module LISTS, it would be referred to as LISTS:LENGTH/3.

3 Requirements

This chapter discusses the functionality required by the driver.

[XXX main ref: http://www.mongodb.org/display/DOCS/Mongo+Driver+Requirements]

3.1 BSON Conversion

[xxx ref: http://bsonspec.org/]

The data representation format used by MongoDB is a binary encoded variation of JSON, called BSON (Binary JSON). In MongoDB, BSON is the format used to store data on disk as well as transmit data over the network.

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

3.2 Connection Management

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

3.3 Document Handling

The driver must expose functionality to find, insert, update and delete documents in a database.

4 API Design

This chapter discusses the overall design of the driver and describes its Application Programming Interface (API).

4.1 Driver Organization

The driver is organized into two modules: *mongo* and *bson*. The mongo module is the main module which contains all the functionality needed to work with a MongoDB instance. The bson module exposes functionality to handle BSON conversions between documents and bytes, and also 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 without it.

4.2 BSON Handling

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.

A BSON document is represented in Prolog as a list of key/value pairs, also called fields. 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 the name as an infix operator, avoiding the need for

parentheses. Keys are UTF-8 encoded atoms, and values can be of several different types.

4.2.1 Document-to-BSON Conversion

[XXX 6.2.1 and 6.2.2 will probably be rewritten and merged into one section. The conversion predicate works ways and could be described as such.]

A document with two fields may look like this:

```
[name-'MongoDB', type-db]
```

4.2.2 BSON-to-Document 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. xxxxxxxxxxxxxx

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].

4.2.3 Document Manipulation

[XXX This section got very manual-ish. Is this relevant? Should I keep it, rewrite or omit?]

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.

4.3 Network Communication

Communication with a MongoDB instance is conducted through a Transmission Control Protocol (TCP) socket. [xxx ref mongo wire protocol]

4.3.1 Obtaining a Connection

A connection can be established by the predicate MONGO: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.

```
?- mongo:new_connection(Connection).
Connection = [opaque connection handle]
```

The following will connect to a MongoDB instance running on localhost attached to the port 1234.

```
?- mongo:new_connection(localhost, 1234, Connection).
Connection = [opaque connection handle]
```

4.3.2 Releasing a Connection

When a connection is no longer needed it must be properly released back to the system. Assuming *Connection* is a connection obtained by the predicate MONGO:NEW_CONNECTION/1,3, the following will release any resources associated with it, rendering it unusable.

```
?- mongo:free_connection(Connection).
[Connection may no longer be used]
```

4.4 Working with Data

This section gives an overview of the main functionality needed to work with databases, collections and documents.

4.4.1 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 retrieved by the predicate MONGO:GET_DATABASE/3. The predicate accepts a connection handle and the name of the desired database as an atom, and returns a handle to the database. The database need not exist beforehand, and will be created when it is first used.

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 MONGO:GET_COLLECTION/3, which accepts a database handle and the name of the collection, and returns a collection handle. The collection need not exist beforehand, and will be created when it is first used.

4.4.2 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 *mongo:find/8*, *mongo:find_one/4* and *mongo:find_all/4*. The most general predicate is *find* which takes many arguments and offers very fine-grained control over the query. The interface to the predicate is described in table 1.

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	Documents returned by the query

Table 1: Interface to mongo:find/8.

The predicate *find_one* 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 2.

The predicate *find_all* 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 3.

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 2: Interface to mongo:find_one/4.

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

Table 3: Interface to mongo:find_all/4.

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 created and maintained by the database instance, and it represents a pointer to a subset of the documents matched by a query [xxx ref and maybe rewrite].

After making a query, the predicate *mongo:cursor_has_more/*1 can be called on a cursor. The predicate succeeds if more documents can be retrieved, and its interface is described in table 4.

Argument	Description
+Cursor	Cursor to investigate

Table 4: Interface to mongo:cursor_has_more/1.

If more documents need to be retrieved, the predicate *mongo: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 5.

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 *mongo:cursor_kill/1*, described in table 6. If several cursors

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

Table 5: Interface to mongo:cursor_has_more/1.

need to be killed, a more efficient version called *mongo:cursor_kill_batch/1* can be used to kill them in a single database call. This predicate is described in table 7.

Argument	Description
+Cursor	Cursor to kill

Table 6: Interface to mongo:cursor_kill/1.

Argument	Description
+Cursors	List of cursors to kill

Table 7: Interface to mongo:cursor_kill_batch/1.

Should all remaining documents matched by a query need to be retrieved in a single action, the predicate *mongo: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 8.

4.4.3 Insert Documents

Two predicates are available to insert documents into a database collection. The simplest one is *mongo:insert/2*, which accepts a collection handle and a document to insert, as described in table 9.

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

Table 8: Interface to mongo:cursor_exhaust/2.

Argument	Description
+Collection	Collection handle
+Doc	Document to insert

Table 9: Interface to mongo:insert/2.

If several documents need to be inserted, a more efficient predicate called *mongo:insert_batch/3* may be used. This predicate is described in table 10.

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

Table 10: Interface to mongo:insert_batch/3.

4.4.4 Update Documents

Updating a single document is done using the predicate *mongo: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 11.

If all matching documents need to be updated, a similar predicate called *mongo:update_all/3* can be used. The interface to this predicate is identical to the one described in table 11, 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 *mongo:upsert/3*, and works

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

Table 11: Interface to mongo:update/3.

like a normal update (described in table 11) except that if the document does not exist it is inserted instead of updated [XXX ref].

4.4.5 Delete Documents

Deleting documents from a collection is accomplished through the predicates *mongo:delete/2* and *mongo: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 12 and table 13.

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

Table 12: Interface to mongo:delete/3.

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

Table 13: Interface to mongo:delete/3.

4.4.6 General Database Commands

Some commands deal not with documents directly, but with whole databases, collections or meta data.

4.5 Error Handling

Errors issued by a failed 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.

5 Usage Example

This chapter shows an example of how to use the driver.

6 Implementation

[XXX Talk a lot more about how things actually are implemented. DCGs, sockets, wire bytes, ...]

[XXX What is this chapter supposed to contain, really? Discuss code organization, modules? Discuss portability? Discuss efficiency (relevant?)?]

6.1 Definite Clause Grammar

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

6.2 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.

6.3 Unit Testing

Unit testing is conducted using *PlUnit,* a unit testing framework built into SWI-Prolog and designed to be compatible with other Prolog implementations [XXX ref].

6.4 Technical 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 An ANSI C compiler is required to build parts of the low-level BSON handling.

MongoDB The driver is tested on MongoDB versions 1.8.0 and 1.8.1.

7 Evaluation

[xxx Did it work? Is it usable? What should have been done differently?]

8 Related Work

[xxx Compare to existing drivers? Erlang? And something completely different? Will probably skip this chapter. Doesn't feel that interesting.]

9 Conclusion/Future Work [merge or split?]

[xxx Not sure how much this section relates to Evaluation above.

Portability (Tested on Mac, SWI, GCC/Clang, etc.)? Efficiency? How to improve in the future? Write a C extension with more/most functionality? Incorporate BSON C implementation? Don't know how portable that would be though, but SWI (and probably SICStus?) has a mature interface to C. What is missing for it to be a "real" driver? Discuss choice of list as BSON document. Maybe use some map-like structure instead.]

[Should I include a chapter with some kind of usage example, with a longer code snippet that shows what modules to include and how to actually interact with the driver? Is that Appendix material?]

References

[1] [xxx how should these be ordered? alphabetical last names? publish date? in order of first referenced in text?]

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

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

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

[5] Some Coding Guidelines for Prolog
Covington, M. A. et al. Retrieved 2011-06-07. [xxx how to format web refs?]
http://www.ai.uga.edu/mc/plcoding.pdf

[6] [xxx unreferenced] MongoDB Driver Documentation
http://www.mongodb.org/display/DOCS/Drivers

[7] [xxx unreferenced] BSON Specification, Retrieved 2011-06-07.
http://bsonspec.org/

[8] [xxx unreferenced] SWI-Prolog Documentation
http://www.swi-prolog.org/pldoc/