Softsys HW 03 Writeup

Alison Berkowitz and Nathan Lintz

March 2014

1 Introduction

In this report, we discuss our use of MongoDB, a non-relational database, and how we used it to prototype a 6-degrees of separation from Kevin Bacon program. Our source code can be found here: https://github.com/alisonberkowitz/LKMHW3.

2 Installation

For our database management system, we use MongoDB, which is free and licensed under Free Software Foundation's GNU AGPL v3.0. We decided to use Pymongo to work with MongoDB from Python, which is easy to install with a simple pip install command. Before deciding on Pymongo, we attempted to work with MongoDB from C and used the MongoDB C driver, which can be cloned from https://github.com/mongodb/mongo-c-driver.git.

3 Immersion

Both the C interface and the Python interface were easy to get started with. The Python interface was simpler because there was a lot of clear example code in the docs. Moreover, the python driver uses dictionaries to store Mongo documents which provided us with a simple interface to marshall our objects and save them. The C driver was more confusing than the python interface because there were fewer code samples on the internet. We also had to follow a fairly confusing process to store our documents in the database which involved using BSON string building methods.

3.1 Python Code Example

```
class Node(object):
        def __init__(self, name=None, nodeType="actor", children=[],
            numberChildren=0):
                self._name = name
                self._nodeType = nodeType
                self._children = children
                self._numberChildren = numberChildren
```

```
class DatabaseService(object):
        def __init__(self, dbName=""):
                self._dbName = dbName
                client = MongoClient('localhost', 27017)
                self._db = client[dbName]

        def createNode(self, node):
                nodes = self._db.nodes
                nodes.insert({'name':node.name, 'children':node.
                    children, 'type':node.nodeType, 'numberChildren':
                    node.numberChildren})
```

3.2 C Code Example

```
typedef struct
{
  char *type;
  char *name;
  char **children;
  int numberChildren;
} Node;

% nodes is an array of nodes which we omitted here for brevity

ps = ( bson ** )malloc( sizeof( bson * ) * n);
for ( i = 0; i < n; i++ ) {
p = ( bson * )malloc( sizeof( bson ) );
bson_init( p );

bson_append_new_oid( p, "_id" );
bson_append_string( p, "type", nodes[i].type );
bson_append_string( p, "name", nodes[i].name );
bson_append_int( p, "numberChildren", nodes[i].numberChildren );

bson_append_start_array(p, "children");
for (j = 0; j<(nodes[i].numberChildren); j++)
{
  char str[15];
  sprintf(str, "%d", j);
  bson_append_string(p,str,nodes[i].children[j]);
}
bson_append_finish_array(p);

bson_finish( p );
ps[i] = p;
}
```

```
  mongo_insert_batch( conn, "test.seeddb", (const bson **) ps, n, 0, 0
      );

  for ( i = 0; i < n; i++ ) {
    bson_destroy( ps[i] );
    free( ps[i] );
  }
```

4 Semantics

In Mongo, you retrieve documents using the command "find"[1]. You can run simple queries by searching a document for a record's attribute attribute. For example, we wrote the following interface to query for actors by their name.

```
bson_init( query );
bson_append_string( query, "name", name );

bson_finish( query );

mongo_cursor_init( cursor, conn, "test.nodes" );
mongo_cursor_set_query( cursor, query );

while( mongo_cursor_next( cursor ) == MONGO_OK ) {
  bson_iterator iterator[1];
  if ( bson_find( iterator, mongo_cursor_bson( cursor ), "name" )) {
  printf("found name: \%s", bson_iterator_string(iterator))
        }
}
```

In the code snippet above, we create a query string which queried for an actor based on their name. Then, we pass the query string to the bson iterator which goes through our database and looks for objects which match the query.

While simple queries provide a lot of Mongo's power, complex queries are what make Mongo really shine. One type of complex queries are and/or operators. These queries are like simple queries but instead of matching on a single condition e.g. matching to a document name, they are used to match documents which meet multiple conditions. For example, they could be used to find all objects in our database whose type is actor and whose name is Kevin Bacon. This would be helpful if we had a movie model in our collection and there was a movie titled Kevin Bacon since our simple query on name alone would return both the actor and the movie objects named Kevin Bacon.

The last type of complex query supported by Mongo are queries which use the Mongo query operators such as the $in operator. As a non relational database, Mongo doesn't have the powerful join feature found in relational databases and instead it encourages you to nest related documents in the same document. The $in operator lets you go query the subdocuments effectively removing any needs for joins. For example, we didn't want to use a separate model to store movies and

[1] http://docs.mongodb.org/manual/tutorial/query-documents/

actors because our data source only had the actor to movie relationship. To solve this problem we each actor store a list of movies they are in as a subdocument of the actor document. To query for a movie based on its title, we can take advantage of the $in operator and find all subdocuments whose name matches a movie title.

```
bson_init( query );
bson_append_start_object(query, "children");
bson_append_start_array(query, "$in");
bson_append_string( query, "name", name );
bson_append_finish_array(query);
bson_append_finish_object(query);
bson_finish( query );
```

5 Performance

Queries in MongoDB are much faster than in a relational database because it keeps related data together instead of separating it into multiple tables and joining it later. It is also easy to scale up because MongoDB uses autosharding to add more machines. When a data set is so large it exceeds the storage capacity for one machine, the data is divided and distributed over multiple servers, or shards. This allows for more concurrent queries because each shard only processes the queries directed to them by Mongo's query routers. Also, indexing is supported by MongoDB to process queries efficiently. This way, it does not need to inspect all of the data to execute each query. The index stores the value of only a specific field so we can check if that value matches a specific query, instead of traversing all of the fields.

6 ACIDity

Mongo DB is not ACID compliant for the following reasons [2].

(a) Atomic: Mongo is not atomic for multi document updates but it is atomic for single document updates. This means that as you update a single document, you are guaranteed that the update will either fail or succeed for the entire document. This guarantee is not true for updates across multiple documents. However, the nature of Mongo is that you want to denormalize your data and store relationships inside a single nested document. Therefore, if you follow Mongo conventions, you should only have to guarantee atomic updates for a single document anyways.

(b) Consistent: If you have a single Mongo server running, you are guaranteed that it will be consistent. However, when you want to scale your application to use multiple servers, slave nodes to your primary server are eventually consistent but there's no guarantee that they will always be consistent.

(c) Isolated: Mongo is isolated because there is a server-wide write lock. This means that writes to the database are guaranteed to be isolated because two writes to the database cannot occur at the same time. Therefore, concurrent and serially executed writes will result in the same

[2] http://css.dzone.com/articles/how-acid-mongodb

behavior. Reads are guaranteed to be isolated so long as no one is writing meaning that they are isolated as well.

(d) Durable: Mongo is not durable but can be configured to keep your data fairly safe. MySQL is durable because the journal of each transaction is committed so if there is a power loss or a crash, you can know that everything up to the last write will be saved. This is because the journal stores enough information to restore the database state in case of failure. In contrast, Mongo is configured to commit a journal of operations every 100 ms. Therefore, if there is a powerloss or crash, any writes after the last journal commit will be lost. By configuring your mongo server to commit the journal more frequently, you can reduce the likelihood of data loss.

7 C Interface

6) The C driver for Mongo is somewhat easy but the sample code for it is sparse and the documentation is missing a number of core features. For example, they barely talk about complex queries and don't explain how to perform a query with Mongo query operations. We would not recommend it to implement the server code because trying to figure out how to perform these operations would slow down development time considerably.

In terms of API design, the Mongo C driver needs a lot of work. In the pymongo API, you can save documents as a dictionary and query documents to retrieve a dictionary. In C, you need to use BSON string builders to store and query your objects which adds an unnecessary layer of complexity to the developer.

We didn't run into any issues of unimplemented Mongo features so the API is complete and reliable but it still seems pretty raw when compared to the pymongo driver. I would at least wait until they can build some higher level abstractions to interact with the database before using this C driver.

While the C driver for Mongo has a number of problems, our code provides a very clean API for developers who want to run queries on an actor database like the one needed for the six degrees of separation problem. We provide three different high level queries which make database access simple. The first two queries are actorNode and movieNode. These let you pass in an actor or movie name as well as a Node object and fill the Node with the name of of the movie or actor as well as the actors the movie has for movie nodes and the movies the actor has been in for actor nodes. The third query is BFS. This query takes in a start and an end actor name and returns a chain of movies and actors which connect the start and end nodes. This method even supports movie to movie paths as well as actor to movie and movie to actor paths. Thus, our API for the mongo C driver provides a lot of powerful features for solving the 6 degrees of separation from Kevin Bacon problem.