HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

MongoDB

An introduction and performance analysis

Seminar Thesis

Master of Science in Engineering
Major Software and Systems
HSR Hochschule für Technik Rapperswil
www.hsr.ch/mse

Advisor: **Prof. Stefan Keller**
Author: **Rico Suter**

Rapperswil, January 2012

Version 1.0

Abstract

This paper consists of two parts: An introduction to MongoDB and a performance analysis. The paper starts with a short explanation what MongoDB is, showing its features and limitations. After this introduction, the user gets a guide on how to install MongoDB on his computer. When the database is installed the reader is introduced to the built-in command line shell where he can run first queries. Every database is accessed by other languages which need their own client libraries. Therefore the reader finds a simple introduction to PyMongo for Python and the official .NET client library. The second part is the performance analysis against PostgreSQL. There are three benchmarks: “bulk data insert”, “tag search” a table joining benchmark and “bounding box location search” benchmark testing the geospatial performance of MongoDB, PostgreSQL and PostgreSQL with the PostGIS extension. The paper ends with notes about when to use MongoDB, a conclusion and the writer’s opinion of MongoDB.

Keywords

MongoDB, introduction, tutorial, installation, queries, client library, PyMongo, .NET, C#, performance, insert, tags, bouding box, usage scenarios, schema-free, data processing, geospatial, PostgreSQL, PostGIS

Contents

1 Introduction

1.1 Overview

MongoDB is an open-source NoSQL-Database developed by 10gen [1] in C++. NoSQL is a new trend in database development and refers generally to databases without fixed schema [2]. Such databases usually have a lower transaction safety but are faster in accessing data and scale better than relational databases. For more information on NoSQL databases I recommend reading the NoSQL book written by S. Edlich et. al. [3]. MongoDB is one of the newer NoSQL databases developed in 2009. The database belongs to the category of the document-based databases. The origin of the name "Mongo" is not very clear: Some people think it comes from the English word "hu*mongo*us" (gigantic) [4], other assume its name is based on the character "Mongo" from the movie Blazzing Saddles [5].

This paper is divided into an installation guide, followed by an explanation how to use the MongoDB database. After this introductory chapter two client libraries are presented: PyMongo for Python and the official .NET MongoDB library. The main part of this paper is the performance benchmark. At the end you will find some usage scenarios, a summary and a conclusion.

1.2 What is MongoDB?

The MongoDB database consists of a set of databases in which each database contains multiple collections. Because MongoDB works with dynamic schemas, every collection can contain different types of objects. Every object – also called document – is represented as a JSON structure: a list of key-value pairs. The value can be of three types: a primitive value, an array of documents or again a list of key-value-pairs (document).

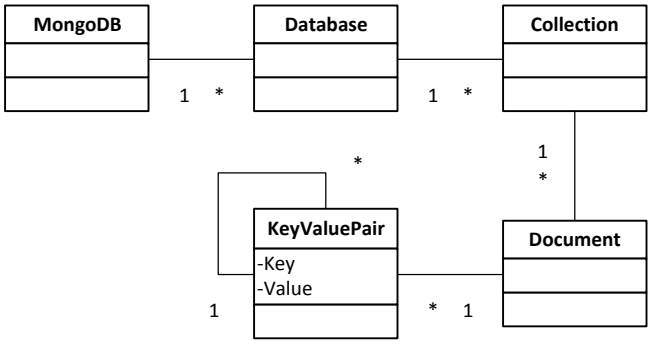

Figure 1: Model of the MongoDB system

To query these objects, the client can set filters on the collections expressed as a list of key-value pairs. It is even possible to query nested fields. The queries are also JSON structured; hence a complex query can take much more space than the same query for a relational database in SQL syntax. If the built-in queries are too limited, it is possible to send JavaScript logic to the server for much more complex queries. MongoDB requires using always the correct type: If you insert an integer value into a document, you have to query for it also with an integer value. Using its string representation does not yield the same result.

The database supports indexes [6]: It is possible to create ascending, descending, unique and geospatial indexes. These indexes are implemented as B-Tree indexes. The "_id" column which can be found in every root document is always indexed.

MongoDB supports two types of replication: master-slave and replica sets. In the master-slave replication, the master has full data access and writes every change to its slaves. The slaves can only be used to read data. Replica sets work the same as master-slave replications, but it is possible to elect a new master if the original master went down. Another feature supported by MongoDB is automatic sharding [7]. Using this feature the data can be partitioned over multiple nodes. The administrator only has to define a sharding key for each collection which defines how to partition the contained documents. In such an environment, the clients connect to a special master node called *mongos* process which analyses the query and redirects it to the appropriate node or nodes. To avoid data losses, every logical node can consist of multiple physical servers which act as a replica set. Using this node infrastructure it is also possible to use Map/Reduce [8] to work on the available data set having a very good performance.

Transactions are not directly supported by MongoDB. Though there are two workarounds: atomic operations and two-phase commits. Atomic operations allow performing multiple operations in one call. An example is *findAndModify* [9] or the *$inc* [10] operator used in updates.

There are several other limitations. For example, if you use the 32-bit version of MongoDB the data set is limited to a size of 2.5 gigabytes [11]. MongoDB does not support full single server durability which means you need multiple replications to avoid data losses if one server suffers a power loss or crash [12]. Another drawback is the fact that it uses much more storage space for the same data then for example PostgreSQL. Because – as opposed to relational databases – every document can have different keys [13] the whole document has to be stored, not only the values. That's why it is recommended to use short key names.

1.3 Installation

MongoDB can be installed on nearly every operation system. This chapter shows the steps to install the database on Ubuntu and Windows. More information on the installation process can be found on the MongoDB website [14]. If you do not want to install MongoDB, you can use the online shell as well: Go to http://www.mongodb.org and click on "Try it out".

Ubuntu

First install MongoDB with apt:

```
sudo apt-get install mongodb
```

Create the default database storage directory:

```
sudo mkdir -p /data/db/
sudo chown `id -u` /data/db
```

After creating the directory, the MongoDB server can be started:

```
mongod
```

If the port is already in use, try starting MongoDB with another port using the *--port* option. Use the "service" command to start the server as a service:

```
service mongodb start
```

Windows

To install MongoDB on Microsoft Windows, simply install the MongoDB installer. Before starting the server, create the directories "C:/data/db" on your system hard drive. To start MongoDB, open a new command line shell, "cd" to the "bin" directory in the MongoDB directory and start the server with the "mongod" command. The server is now running as long as the command line is opened.

1.4 First steps

The server is now up and running and can be used by clients. There is a client shell which can be started with the "mongo" command. The following call will start the client using the database "benchmark":

```
mongo benchmark
```

To use the "mongo" command in Windows, start a new command line shell and browse to the "/bin" directory within your MongoDB installation.

To store the first MongoDB document, use this command:

```
db.foo.insert({"key" : "value"})
```

Now you can check if the document has been inserted:

```
db.foo.find()
```

The "find" command's output will be something similar to:

```
{ "_id" : ObjectId("4ebe79742033f11f07fe742e"), "key" : "value" }
```

1.5 Queries

Queries are built using JSON or in client libraries with a BSON data structure. The following samples show queries written in JavaScript which can be used in the "mongo" shell. On the MongoDB website there is a mapping chart which maps a lot of relational queries to its MongoDB equivalents [15].

Insert

Use the "insert" method to insert a new tuple in a database:

```
db.myCollection.insert({key1: "value1", key2: "value2"})
```

If the database “db” or the collection “myCollection” does not exist, it will automatically be created. For every new object an object ID will be generated and stored in the key “_id”. Of course it is possible to insert more complex objects (e.g. nested data structures):

```
db.myCollection.insert({name: "Foo Bar", address: {Street: "Foo", City: "Bar"}})
```

Read

The stored objects can be read with the “find” method. The following commands return all objects saved in the “myCollection” collection.

```
db.myCollection.find()
```

To search for a specific object, limit the number of the result items and get only one column of each document, check out the query below.

```
db.myCollection.find({lastname: "Meier"}, {firstname: true}).limit(10).skip(20)
```

The SQL query for the above search looks like:

```
SELECT firstname FROM myTable WHERE lastname = 'Meier' LIMIT 20,10
```

MongoDB allows querying the documents with very simple queries. The following query searches in documents which contain an array of loved objects stored in the key “loves”. The query returns all persons who love either apples or oranges or both.

```
db.persons.insert(firstname: "Meier", loves: ['apple', 'orange', 'tomato'])
db.persons.find($or: [{loves: 'apple'}, {loves: 'orange'}])
```

It is very easy to query the data if they have the right structure. In MongoDB there are no joins. References to other documents or objects are usually dereferenced on the client side by lazily querying the referenced documents. There is also a way to do this transparently using *DBRefs* [16]. It is possible to add references using the *ObjectId* type:

```
db.person.insert({firstname: "Meier", mother: ObjectId("...")})
```

If you access the field “mother” the referenced document will be lazily loaded by the MongoDB driver. This functionality must be supported by the used driver (client library).

Update

The update command works like its relational equivalent: The first argument chooses the tuples to update and the second sets the new values.

```
db.myCollection.update({id: 123}, {$set : {a : 4}})
```

Delete

To following command will remove all tuples whose first name is “Hans”.

```
db.myCollection.remove({firstname: "Hans"});
```

2 Client libraries

During the implementation of the performance benchmark, I used two client libraries:

- Pymongo for Python [17]
- Mongo Csharp Driver for .NET [18]

There are much more libraries for almost every programming language [19].

2.1 PyMongo for Python

The client library for Python is called PyMongo and can be installed on Ubuntu using "apt":

```
sudo apt-get install python-pymongo
```

To set up a connection to the database "db" and get access to the collection "myCollection" use the following Python code:

```
connection = pymongo.Connection("localhost", 27017)
database = connection["db"]
collection = database["myCollection"]
collection.ensure_index("store")
```

The variable "collection" contains the logic to query, insert, update and delete objects:

```
tuple = {{"firstname", "Rico"}, {"lastname", "Suter"}}
collection.insert(tuple)
```

To get a list of objects whose firstname equals "Rico" use the following query. Queries can be written using the Pythons built-in associative arrays:

```
for person in collection.find({"firstname", "Rico"}):
      person
```

The code to initialize the objects and queries is very elegant because Python is not typed and the initialization of new dictionary data structures is a built in feature. The automatic creation of new databases and collections if they don't exist is a good feature to reduce the amount of typed code.

2.2 MongoDB driver for .NET

Because C# is a typed language the library usage is not as elegant as in Python: The library uses BSON documents which have to be initialized to insert new objects or to query the database.

These are the programming bits to set up a connection and get a collection object:

```
var server = MongoServer.Create("mongodb://localhost:27017");
var database = server.GetDatabase("db");
var collection = database.GetCollection("myCollection");
collection.EnsureIndex("myKey");
```

To insert data you have to create a new BsonDocument and insert it into a MongoDB collection. MongoDB uses BSON to transmit all objects over the wire. BSON [20] is the binary representation of a

JSON data structure. The advantages over the usual JSON data structure are its minimized data usage and better processing performance.

```
var tuple = new BsonDocument
{
       {"firstname", "Rico"},
       {"lastname", "Suter"}
};
collection.Insert(tuple);
```

To query a collection you can simply use the “find” method already mentioned in a previous chapter:

```
var query = new BsonDocument { { "firstname", "Rico" } };
var result = collection.Find(query);

foreach (BsonDocument obj in result)
{
       var lastname = obj.GetElement("lastname").Value.AsString;
       Console.WriteLine(lastname);
}
```

This query will return all objects whose first name equals to “Rico”.

The MongoDB driver for .NET is a nice piece of code. It is easy to use and does not take a lot of learning time. However it is sometimes difficult to construct complex queries with the available BSON classes.

3 Performance

To test the performance of the MongoDB I developed a test suite in .NET using C#. Every benchmark will be tested on the following systems:

- PostgreSQL (32-bit because PostGIS is only available for 32-bit PostgreSQL)
- MongoDB (32-bit)

Every benchmark will be run using indexes and without indexes. To see how the databases scale, the benchmarks will be run on different object sets with different sizes. The benchmarks are run on a Microsoft Windows operation system on a quad-core notebook computer with a SDD harddrive. The first idea was to extend the benchmark by Michel Ott [21]. Because I had to change a lot of the original benchmark, I decided to switch to .NET and rewrite the whole benchmark in C#.

In every benchmark, lower numbers are better which means the query used less time to execute.

3.1 Benchmark settings

The tasks are generated randomly, thus every query differs and it is very hard for the database to cache any query. The tests are always running under the same user using the default database settings.

Every benchmark will be run with three parallel threads on a new created table. The benchmark creates the table only once per benchmark run. This means that this is a hot start benchmark. At the end of the test, the average of the all queries will be computed and displayed.

The structure of the used data is explained in the chapter “bulk data insert” and the other benchmark scenarios. The benchmarks are run with 1000, 10000 and 100000 of these data objects.

For more information on benchmark setups check out the HSR GIS benchmark site [22].

3.2 Bulk data insert

Benchmark

This benchmark tests the speed of inserting a lot of data objects. In this test case the benchmark client should insert every tuple with a single call – it must not use bulk inserts. Because some databases cannot save the designated object in a single row, they need to insert multiple tuples for one object. This will result in a huge disadvantage for PostgreSQL without any extensions.

Data

The inserted objects are used for the next benchmarks. It is a simple structure: an object with a location and multiple tags (key-value-pairs).

The test on the “plain” PostgreSQL database will be done using two tables: an “objects” table and a “tags” table with a foreign key to the “objects” table. This leads to multiple rows per object and a disadvantage over the other databases.

To have accurate test results for MongoDB, the insert call has to be modified because the driver's default insert behavior is asynchronous. To use synchronous insert operations, the client has to enable the safe mode. This can be done while creating the MongoDB connection:

```
var settings = new MongoServerSettings();
settings.SafeMode = SafeMode.True;
settings.Server = new MongoServerAddress(hostname, port);
settings.DefaultCredentials = new MongoCredentials(username, password);
var server = MongoServer.Create(settings);
```

Results

There are big differences in the four test cases. The data insertion into the PostgreSQL without hstore ("PostgreSQL" and "PostgreSQL with geo") is the slowest because there are a lot more tuples to insert then in PostgreSQL with hstore or MongoDB. Nevertheless MongoDB is much faster than hstore. Consult the table below for more details:

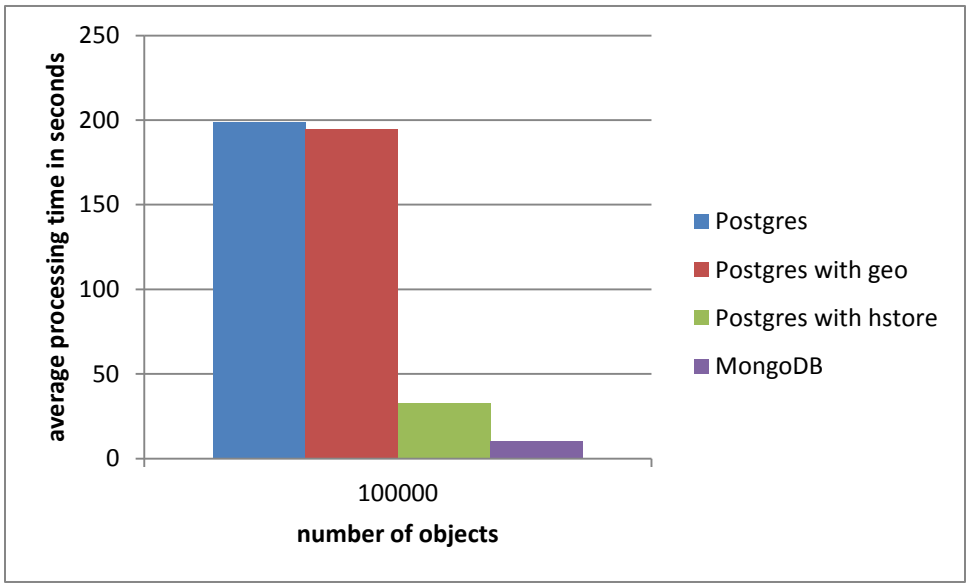

Figure 2: Data insertion benchmark results

This diagram shows the duration in seconds to create 100000 objects. I think the reason why MongoDB is faster is because it does not use transactions or ensures durable writes whereas PostgreSQL does the two things by default.

3.3 Tag search scenario

Benchmark

This benchmark will measure the time to query objects with a specific tag and a specific value. The 10000 tasks will be created randomly. Every task searches for one tag with one value. In relational databases this will test the join behavior and in document based databases the speed of querying nested objects.

Data

This scenario tests the speed of looking up tags on objects. Every object has multiple tags. For every tag there is a value associated. The following diagram shows the "domain" of this test.

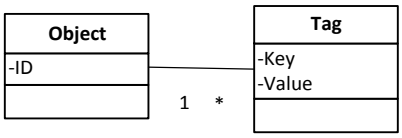

Figure 3: Domain of the tag search scenario

Every system has its own way to represent this data structure.

PostgreSQL

The relational approach used by PostgreSQL without hstore uses two simple tables: One for the objects and one for the tags associated with the objects.

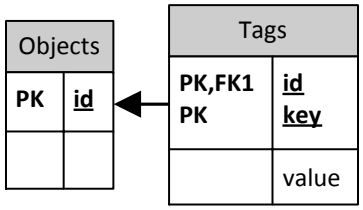

Figure 4: PostgreSQL tables for the tag search scenario

The SQL query to fetch the desired rows matching two tags and values looks like:

```
SELECT * FROM objects_benchmark o
       LEFT OUTER JOIN objects_tags_benchmark t1 ON (t1.id = o.id)
       LEFT OUTER JOIN objects_tags_benchmark t2 ON (t2.id = o.id)
       WHERE t1.key = 'tag1' AND t1.value = 'value1' AND
             t2.key = 'tag2' AND t2.value = 'value2';
```

Using the hstore type, the data structure can be represented only with one table. The tags are stored in the “location” column whose type is hstore. The type hstore contains a list of key-value-pairs.

To create the hstore data type for the database “benchmark” first start the PostgreSQL console using the “psql” command (see “C:\Program Files (x86)\PostgreSQL\9.1\bin”) in a command line shell and type in the password for the user “postgres”[1].

```
psql –U postgres –d benchmark
```

The following command will create the hstore data type for the current database:

```
CREATE EXTENSION hstore;
```

[1] Tip: First create the database “benchmark” with PgAdmin

The following SQL query is used to look up all objects with the requested two tags and values. The first query is used together with indexes as it is faster than the second SQL statement.

```
SELECT * FROM objects WHERE
       hstore(tags)->'"tag1"=>"value1"' AND
       hstore(tags)->'"tag2"=>"value2"';
```

The second SQL query yield the same results as the previous query. This query is used without indexes because this is the faster query if there are no indexes.

```
SELECT * FROM objects WHERE
       tags @> hstore('tag1', 'value1') AND
       tags @> hstore('tag2', 'value2');
```

MongoDB

In MongoDB the objects are stored the same way as with the hstore, because MongoDB can only store key-value-pair lists and does not support joins.

```
collection.Insert({"id": 1, "tags":
        [{"key": "key1", "value": "value1"},
         {"key": "key2", "value": "value2"}]});
```

Use the following query to lookup all objects with the designated tags:

```
collection.Find({"$and": [
       { "tags" : {"$elemMatch" : { "key" : "key1", "value" : "value1"}}},
       { "tags" : {"$elemMatch" : { "key" : "key2", "value" : "value2"}}}
]}
```

Results

As you can see in the next diagrams, MongoDB is always slower than PostgreSQL even without hstore.

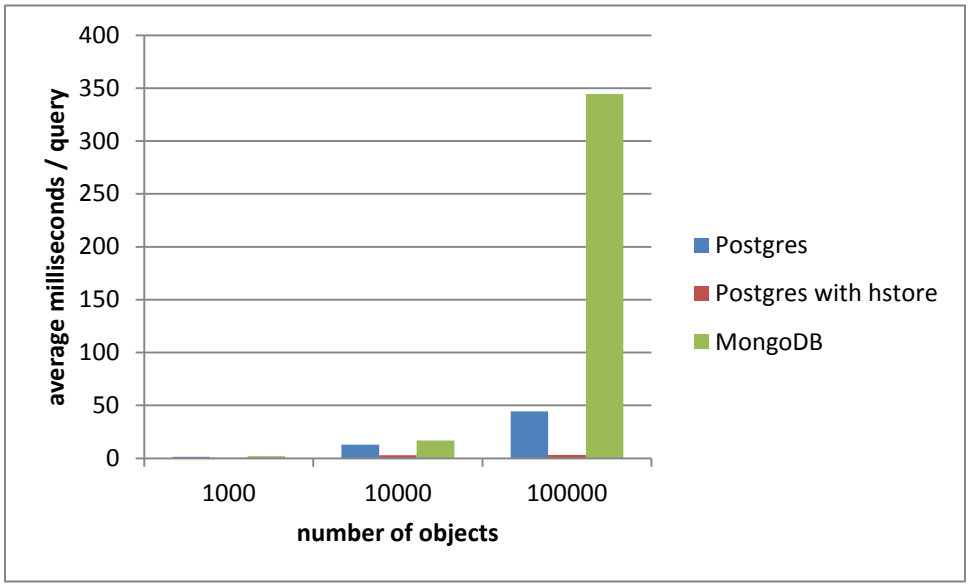

Figure 5: Benchmark results of the tag search scenario (without indexes)

If you use indexes, which is the default, MongoDB is nearly four times slower than both PostgreSQL benchmarks. This benchmark shows a strange result: PostgreSQL with hstore yield approximately the same results with indexes (GIN or GIST) and without indexes; removing indexes is even a bit faster.

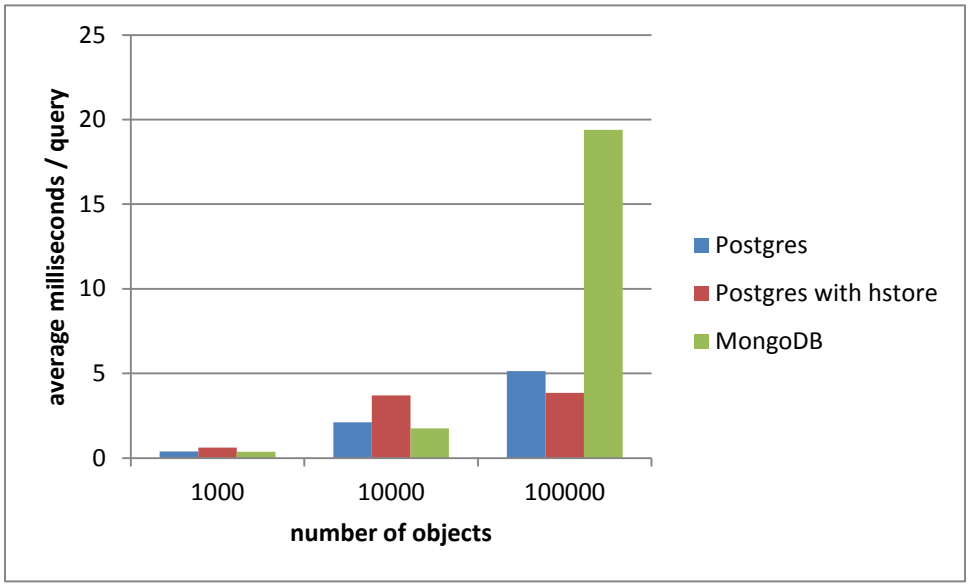

Figure 6: Benchmark results of the tag search scenario (with indexes)

This benchmark shows that MongoDB is generally not intended to search for tags over a lot of data objects. I think, the reason why MongoDB is the fastest database for a small amount of data, is because MongoDB stores – if it is working on a small set of objects - all objects in memory instead on the hard drive. If you heavily rely on scenarios like this "tag search" with big object sets it is better to use PostgreSQL. If you use sharding, which is one of MongoDB strengths, MongoDB can eventually improve the results compared to his competitors. However this scenario has not been benchmarked.

3.4 Bounding box location search scenario

Benchmark

The intention of this scenario is to test the databases for their geospatial usage. In every database the duration of a bounding box range query will be measured. Every run of the test specifies a rectangle and asks for all objects within this rectangle.

Data

Every object has an x- and a y-value which can be seen as their latitude and longitude. For each database – PostgreSQL and MongoDB – there are two possibilities: using default integers and using tailored geo-spatial data types or indexes.

PostgreSQL

The PostgreSQL database will first be tested with big integers. This can be done using the following simple SQL query:

```
SELECT * FROM objects t WHERE
        (x BETWEEN 5132926 AND 5392525) AND
        (y BETWEEN 4385840 AND 5111316);
```

In a second step the same benchmark is made with the geospatial data types from PostGIS. To use this data type, first install PostGIS: Start the "PostgreSQL Stack Builder" and install PostGIS[2].

After creation of the table without a "location" column, the column can be created using the following query[3]:

```
SELECT AddGeometryColumn('public', 'objects_benchmark', 'location', 2178, 'POINT', 2);
```

To create an index on this column run the following query:

```
CREATE INDEX index_objects ON objects USING GIST (location);
```

New tuples can now be inserted into the table:

```
INSERT INTO objects VALUES(1, GeometryFromText('POINT(50.13157 43.1735)', 2178))
```

To find all tuples within a specific bounding box use this query:

```
SELECT * FROM objects t WHERE location && ST_SetSRID(
ST_MakeBox2D(ST_Point(49.41101, 46.84283), ST_Point(59.74656, 59.85466)), 2178);
```

MongoDb

In MongoDB there is no dedicated geospatial data type. But MongoDB supports special geospatial indexes [23]. If such an index is set on a specific key of objects in a collection, the index is defined over the first two fields contained in the property. Define your data structure as follows:

```
collection.insert({ id: 1, location : [ 45.88663, 56.87652 ] })
```

To create the index on the location property in this document, use this command:

```
collection.ensureIndex({ location : "2d" })
```

The geospatial "within" query is shown below. This query is only available if a two-dimensional index has been created on the "location" key.

```
collection.find({"location" : {"$within" : {"$box" : [[45.8, 56.8], [46.5, 50.3]]}}})
```

The usual integer query looks like the next query [24]:

```
collection.find( {x: [{$gte: 10}, {$lte: 30}], y: [{$gte: 20}, {$lte: 40}] });
```

[2] Important: PostGIS is only available for 32-bit PostgreSQL installations
[3] "2178" is the SRID and can be chosen randomly but must be the same for all queries

Results

The winner of this benchmark is PostgreSQL without any additions. I think PostGIS data types are only faster if used with more advanced geospatial query methods, like polygonal areas.

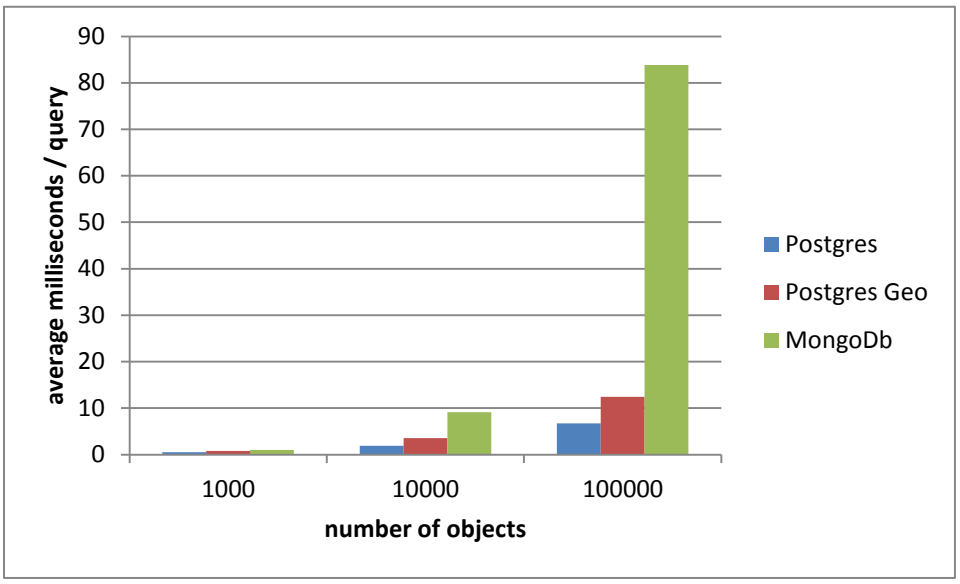

Figure 7: Bounding box benchmark results (without indexes)

MongoDB with geospatial data types is missing in the first diagram because geospatial queries can only be used with indexes. Queries using geospatial indexes are about 15 percent faster than querying bounding boxes on integers in MongoDB. It has to be mentioned that MongoDB supports geospatial indexes over sharded collections which can eventually be faster than PostgreSQL partitioned indexes.

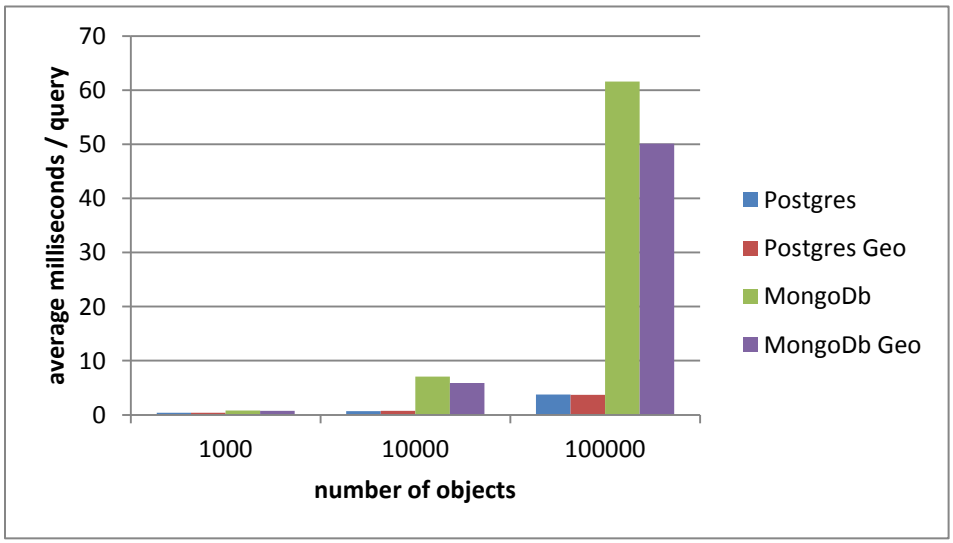

Figure 8: Bounding box benchmark results (with indexes)

When to use MongoDB

3.5 Schema-free

Because MongoDB is schema, free every object can easily be serialized without changing the database. If you want to serialize an object, simply convert it to a JSON representation and store it in the database. No change in the schema is needed and therefore the development of the logic for persisting new domain objects can be done very fast.

3.6 Data processing

If you want to process a lot of data with complex queries, I recommend using a traditional optimized database. The performance results have shown that MongoDB is the slowest database in every query test. However a benchmark by Peter Bengtsson [25] shows that MongoDB is very fast for simple CRUD (create, read, update, delete) operations: If you generally use your database for simple operations and less complex queries MongoDB should be a good choice.

3.7 Geospatial

In general, the geospatial performance is much slower than using PostGIS. However if you need a simple and lightweight solution with easy to understand queries MongoDB can be a solution for you.

4 Conclusion

MongoDB is a fresh new way to persist data. It is easy to learn and fits well for small projects. If you have a bigger project with an always changing domain model, I think it is better to use a relational database. The reason is that MongoDB does not support foreign keys and joins. This forces you to define which your “base objects” are and embed all other objects in it. If you want to use embedded objects as base objects you have a big problem. A good example is a blog: You have your blog entries and comments on every entry. This can easily be represented in MongoDB. But if you want to have a page with all your comments which are embedded in various other objects, you have a problem…

In most cases the performance of MongoDB is good enough but if you want to process a lot of data with complex queries, it’s better to use another database. However the data insertion, updating and deletion performance is very good. This makes MongoDB a good database for projects with simple data access. One good sample scenario is logging which only uses simple inserts and because MongoDB is schemaless the logged information can easily be extended.

In my opinion the future of MongoDB looks good because there is a trend toward web based applications which have generally a lot but simple queries and MongoDB has a very good JavaScript integration. I think in the future developers also tend to look for simpler and more comfortable APIs and MongoDB can surely benefit from this trend as it has a simple API and is very easy to use.

All in all I think MongoDB is worth a look: This doesn’t take a lot of time and maybe you enjoy the flexible way MongoDB manages the persistence of your data.

5 References

[1] [Online]. Available: http://www.10gen.com/.

[2] [Online]. Available: http://de.wikipedia.org/wiki/NoSQL.

[3] S. Edlich, A. Friedland, J. Hampe, B. Brauer and M. Brückner, NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken, Carl Hanser Verlag GmbH & CO. KG, 2011.

[4] [Online]. Available: http://en.wikipedia.org/wiki/MongoDB.

[5] [Online]. Available: http://bigdatalowlatency.com/mongodb.html.

[6] [Online]. Available: http://www.mongodb.org/display/DOCS/Indexes.

[7] [Online]. Available: http://www.mongodb.org/display/DOCS/Sharding.

[8] [Online]. Available: http://www.mongodb.org/display/DOCS/MapReduce.

[9] [Online]. Available: http://www.mongodb.org/display/DOCS/findAndModify+Command.

[10] [Online]. Available: http://www.mongodb.org/display/DOCS/Updating#Updating-%24inc.

[11] [Online]. Available: http://blog.mongodb.org/post/137788967/32-bit-limitations.

[12] [Online]. Available: http://blog.boxedice.com/2010/02/28/notes-from-a-production-mongodb-deployment/.

[13] [Online]. Available: http://lemire.me/blog/archives/2011/12/19/compressing-document-oriented-databases-by-rewriting-your-documents/.

[14] [Online]. Available: http://www.mongodb.org/display/DOCS/Quickstart+Unix.

[15] [Online]. Available: http://www.mongodb.org/display/DOCS/SQL+to+Mongo+Mapping+Chart.

[16] [Online]. Available: http://www.mongodb.org/display/DOCS/Database+References.

[17] [Online]. Available: http://api.mongodb.org/python/.

[18] [Online]. Available: https://github.com/mongodb/mongo-csharp-driver.

[19] [Online]. Available: http://www.mongodb.org/display/DOCS/Drivers.

[20] [Online]. Available: http://en.wikipedia.org/wiki/BSON.

[21] [Online]. Available: http://wiki.hsr.ch/Datenbanken/SeminarDatenbanksystemeFS11.

[22] [Online]. Available: http://www.gis.hsr.ch/wiki/Benchmark.

[23] [Online]. Available: http://www.mongodb.org/display/DOCS/Geospatial+Indexing.

[24] [Online]. Available: http://www.mongodb.org/display/DOCS/Advanced+Queries.

[25] [Online]. Available: http://www.peterbe.com/plog/speed-test-between-django_mongokit-and-postgresql_psycopg2.

5.1 Additional sources

MongoDB paper
http://openmymind.net/mongodb.pdf

Interesting article
http://blog.boxedice.com/2009/07/25/choosing-a-non-relational-database-why-we-migrated-from-mysql-to-mongodb/

.NET Benchmark used in the “Performance” chapter
http://www.rsuter.com/Upload/MongoDBBenchmark.zip

6 Appendix

6.1 Figures

6.2 Benchmark results

bulk data insert

	1000	10000	100000
Postgres	1.81	18.87	199.11
Postgres with geo	1.92	19.11	194.87
Postgres with hstore	0.32	3.15	32.35
MongoDB	0.12	1.04	10.16

tag search scenario

	1000	10000	100000	**without**
Postgres	1.31	12.85	44.43	**indexes**
Postgres with hstore	0.53	3.02	3.11	
MongoDB	1.88	16.65	344.58	
	1000	10000	100000	**with indexes**
Postgres	0.39	2.11	5.14	
Postgres with hstore	0.62	3.69	3.84	
MongoDB	0.36	1.75	19.39	

bounding box scenario

	1000	10000	100000	**without**
Postgres	0.49	1.91	6.73	**indexes**
Postgres Geo	0.79	3.55	12.44	
MongoDb	1.01	9.11	83.81	
	1000	10000	100000	**with indexes**
Postgres	0.37	0.71	3.77	
Postgres Geo	0.41	0.72	3.69	
MongoDb	0.81	7.05	61.55	
MongoDb Geo	0.74	5.91	50.12	

6.3 Tag search PostgreSQL SQL queries

Following is a list of all sample SQL queries used for the “tag search scenario” benchmark using PostgreSQL.

no hstore, no geometry, no index

```
--INITIALIZATION

CREATE TABLE objects_benchmark (id BIGINT PRIMARY KEY, x BIGINT, y BIGINT);
CREATE TABLE objects_tags_benchmark (id BIGINT REFERENCES
objects_benchmark(id), key TEXT NOT NULL, value TEXT NOT NULL);

--INSERT

INSERT INTO objects_benchmark VALUES(1, 5009507, 5704796)
INSERT INTO objects_tags_benchmark VALUES(1, 'a', '32')
...

--BENCHMARK

SELECT * FROM objects_benchmark o
LEFT OUTER JOIN objects_tags_benchmark t1
ON (t1.id = o.id) WHERE TRUE AND t1.key = 'c' AND t1.value = '5';
...

--CLEANUP

DROP TABLE IF EXISTS objects_benchmark CASCADE
DROP INDEX IF EXISTS index_objects_benchmark CASCADE
DROP INDEX IF EXISTS index_objects_tags_benchmark CASCADE
DROP TABLE IF EXISTS objects_tags_benchmark CASCADE
```

no hstore, no geometry, index

```
--INITIALIZATION

CREATE TABLE objects_benchmark (id BIGINT PRIMARY KEY, x BIGINT, y BIGINT);
CREATE TABLE objects_tags_benchmark (id BIGINT REFERENCES
objects_benchmark(id), key TEXT NOT NULL, value TEXT NOT NULL);

--INSERT

INSERT INTO objects_benchmark VALUES(1, 5009507, 5704796)
INSERT INTO objects_tags_benchmark VALUES(1, 'a', '32')
...

CREATE INDEX index_objects_benchmark ON objects_benchmark (x, y);
CREATE INDEX index_objects_tags_benchmark ON objects_tags_benchmark (key, value);

--BENCHMARK

SELECT * FROM objects_benchmark o
LEFT OUTER JOIN objects_tags_benchmark t1
ON (t1.id = o.id) WHERE TRUE AND t1.key = 'c' AND t1.value = '5';
...

--CLEANUP

DROP TABLE IF EXISTS objects_benchmark CASCADE
DROP INDEX IF EXISTS index_objects_benchmark CASCADE
```

```
DROP INDEX IF EXISTS index_objects_tags_benchmark CASCADE
DROP TABLE IF EXISTS objects_tags_benchmark CASCADE
```

hstore, no geometry, no index

```
--INITIALIZATION

CREATE TABLE objects_benchmark (id BIGINT PRIMARY KEY, x BIGINT,
y BIGINT, tags HSTORE NOT NULL);

--INSERT

INSERT INTO  objects_benchmark VALUES(1, 5009507, 5704796,
hstore('\"a\" => \"32\",\"b\" => \"46\",\"c\" => \"64\",
\"d\" => \"87\",\"e\" => \"23\"'))
...

--BENCHMARK

SELECT * FROM objects_benchmark WHERE true  AND tags @> hstore('a', '32');
...

--CLEANUP

DROP TABLE IF EXISTS objects_benchmark CASCADE
DROP INDEX IF EXISTS index_objects_benchmark CASCADE
DROP INDEX IF EXISTS index_objects_tags_benchmark CASCADE
```

hstore, no geometry, index

```
--INITIALIZATION

CREATE TABLE objects_benchmark (id BIGINT PRIMARY KEY, x BIGINT,
y BIGINT, tags HSTORE NOT NULL);

--INSERT

INSERT INTO  objects_benchmark VALUES(1, 5009507, 5704796,
hstore('\"a\" => \"32\",\"b\" => \"46\",\"c\" => \"64\",
\"d\" => \"87\",\"e\" => \"23\"'))
...

CREATE INDEX index_objects_benchmark ON objects_benchmark (x, y);
CREATE INDEX index_objects_tags_benchmark ON objects_benchmark USING GIST (tags);
-- GIN is also possible: The GIN index is a little bit slower than the GIST index

--BENCHMARK

SELECT * FROM objects_benchmark WHERE true AND hstore(tags)->'c' = '5';
...

--CLEANUP

DROP TABLE IF EXISTS objects_benchmark CASCADE
DROP INDEX IF EXISTS index_objects_benchmark CASCADE
DROP INDEX IF EXISTS index_objects_tags_benchmark CASCADE
```