Why MySQL beats MongoDB in one big data scenario

Stephan Sommer-Schulz (Director Research)

searchmetrics

The Challenge

Backlink Database:

Build a graph of most of the backlinks in the internet.

- Store the data in a database
- Give the user the possibilities to
  - Perform queries with multiple where conditions
  - Aggregate fields where needed (grouping, counting)
  - Order results over multiple fields (asc, desc)
- Answer requests within seconds
- Keep costs and administration as low as possible

Definition:

A backlink is an internet link which points from one webpage to another, which is not in the same subdomain (means: no internal links).

The Data

Graph components:

The graph consists of nodes (webpages) and edges (links). Each node and each link can have a number of attributes.

No. of pages:	**9 800 000 000**
No. of links:	**112 000 000 000**
No. of daily new links & updates:	**1 000 000 000**

Because the links are the most interesting thing in our scenario, we will have at least one record per link in our database (> 112 billion).

MongoDB Approach (v2.0.0)

Good:

- Buildin sharding!
- Replication (not needed here)
- Rest and json (or bson)
- Schemaless / document oriented (we will see)
- Scalable (we will see)

Not so good:

- Poor datatypes (no tinyint, no smallint, ...)
- Large datasize, because of document-orientation, field names, ...
- Performance issues on aggregates (e.g. count distinct)
- Indexes must fit completely into memory
- Funny problems with sharding keys if a shard gets to big, but cannot be divided anymore.

MongoDB Tests 1

Test Sharding / Datainput / Index

- 3 Sharding-Servers (AMD Hexacore, 8 GByte, SSD)
- Start with one index
- Input 1 billion records (~300-400 million / shard)
- Create a second index ...

Problem 1:

- On one shard the second index didn't fit into memory (the eta was several days!)
- Do a db.killOp(...) for the index creation process – process is marked as killed, but no other reaction after 6 hours, system is blocked ☹
- Kill the mongod process, install more Memory (16 GByte) create index in 3 hours.

Problem 2:

- Sharding-Key has to be changed (no possibility of dividing any more in a segment)
- Export data – reimport data – waisting lots of time and resources
- Clue: Find a key that fits your needs over years, even if the requierments change sometimes (and they will) -> Impossible Mission in very large systems

Problem 3:

- The size of data and index (disk and memory) is much bigger than expected!?

MongoDB Tests 2

Test Advanced Queries

- 3 Sharding-Servers (AMD Hexacore, 8 GByte, SSD)
- Input 1 billion records (~300-400 million / shard)

Problem 1:

- Counting rows with distinct is damn slow
- Tried to calculate the counters over night doesn‘t fit all needs, because users can influence the queries.
- Map/Reduce is not an option for just–in-time results

Problem 2:

- Using query operators on none index fields is very slow, need to keep data in memory as well as the indexes (more memory!)
- It is not possible to catch every query-situation with an index, if we don‘t want to spend lots of money for
- Schemaless is fine, but the need of indexes in memory looks like a blocker, MySQL has not.

MySQL Approach (v5.5)

Good:

- Well known and powerfull SQL
- Replication (not needed here)
- Good datatypes
- Small data on disk because of relations
- Smaller indexes because of better datatypes

Not so good:

- No sharding!
- No rest, no json
- More development on application-site needed (watch for costs)

Why MySQL beats MongoDB in one big data scenario

Datasize Comparision (not relational)

Fields	Type (MySQL)	Avg. Bytes (MySQL)	Type (MongoDB)	Avg. Bytes (MongoDB)
6x Textfields	Varchar(3-1000)	346	Varchar (3-1000)	346
6x Tiny Attributes	Tinyint (each 1Byte)	6	64-Bit Integer	48
6x Small Attributes	Smallint (each 2 Bytes)	12	64-Bit Integer	48
10x Numbers	Integer (each 4 Bytes)	40	64-Bit Integer	80
3x Large Numbers	Bigints (each 8 Byte)	24	64-Bit Integer	24
2x Doubles	Double (each 8 Byte)	16	Float (each 8 Bytes)	16
6x Relational Ids	6 Ints, 2 Bigints	38	--	0
Sharding-Key	--	0	2x 64-Bit Integer	32
Fieldnames	--	0	33x 2 Bytes	66
	Sum per record:	**482 Bytes**		**660 Bytes**

Relational: MySQL relational datasize (4 tables): 148 Bytes per record!

Datasize Comparision (real world)

MySQL:

- One Record: 148 Byte (relational)
- No. of records: 112 Billion
- Datasize: 15.1 Terabyte
- Indexsize: 5.5 Terabyte

MongoDB:

- One Record: 660 Byte (document)
- No. of records: 112 Billion
- Datasize: 67.2 Terabyte
- Indexsize: 11.4 Terabyte

Looks like we will run into a little cost-problem with MongoDB ...

MongoDB Datasize

Fieldname Problem:

- Fieldnames are stored in MongoDB with every single datarecord – that's necessary, because it is schemaless!
- Fieldnames are stored in every row uncompressed
- If 1 datarecord consists of 33 fields in our scenario and the fieldnames are only 2 Bytes long („aa", „ab", „ac", ...) we will need:

One Datarecord – Fieldname-Size:

- No. Fields: 33
- Fieldbytes/Record: 66 (with only 2 Bytes / Field)

112 Billion Datarecords – Fieldname-Size:

- 66 Byte * 112 Billion = 6.72 Terabyte! (only for the names)

This is where part of your diskspace goes in MongoDB.
Maybe schemaless is not that big advantage ...

Why MySQL beats MongoDB in one big data scenario

Price on Amazon Cloud (EC2) in Ireland

MySQL:

- Datasize: 15.1 Terabyte
- Indexsize: 5.5 Terabyte
- Server: 107 (with 75% mem for index)
- $/Month: $ 158.360,- (admin & traffic costs come on top)

MongoDB:

- Datasize: 68.5 Terabyte
- Indexsize: 11.4 Terabyte
- Server: 221 (with 75% mem for index)
- $/Month: $ 327.080,- (admin & traffic costs come on top)

Amazon EC2 High-Memory Quadrupel Extra Large Instance

- Memory: 68,4 Gbyte
- Disk: 1.6 Terabyte
- $/Hour: $ 2,024
- $/Month: $ 1.480,-

MySQL Challenge

Assumption:

- Sharding in MySQL is not so difficult if we can avoid some traps:
  - No cross-shard joining
  - No cross-shard distinct counting
  - Generate unique keys in application

ToDo:

- Find a schema, which fits the above criteria (we luckily found one, after two weeks in the thinktank, but there will be usecases which never will work in such a setup).
- Setup shards as standalone MySQL servers, with no dependencies
- Build a data-deployment mechanism for the shards (Remind: unique ids)
- Build an application which is able to request data parallel from the shards and than make aggregations, sorts, ...

Problems:

- Network latency
- Caching in application level is finegrained, but can be tricky.

We are hiring

Mad Scientist* (m/w):

- You've trained a Support Vector Machine which selects your morning flakes by weather, milk temperature, blood pressure and thirty other features.
- TF/IDF and BM25 are so 90s. When you are looking for your doorkey, your searchengine ranks listwise.
- You didn't join the Yahoo „Learning to Rank" challenge because you wouldn't want to be a party pooper
- With the approach of map/reduce you gave a lol about the new funny name of the good old scatter/gather
- Your brain is parted in stack and heap, programming a doom-clone in whitespace was fun for an afternoon.
- The Mad Scientist is just another step on your way to Evil Genius.

Hundrets of servers, billions of datasets, sharding, replication, coffee, pizza, coke, more coffee and colleagues dressed in black.

Welcome to the real world, Neo.

*jobtitle may differ on business card

Thanks a lot!

searchmetrics