TPC-H applied to MongoDB: How a NoSQL database performs

Informatik Vertiefung
Nico Rutishauser
09-706-821
University Zurich
25 February 2012

supervised by : Prof. Dr. Michael Böhlen
Amr Noureldin

Abstract

This work compares the database benchmark TPC-H applied on PostgreSQL and MongoDB. Comparing a classical SQL with a NoSQL database is delicate and needs some closer look on the schema and queries. A short introduction shows the basics of MongoDB, the main differences to a RDBMS and how to bridge the gap. Six TPC-H queries are selected and modified for MongoDB. The benchmarks, comparing runtimes of PostgreSQL with the original queries and MongoDB with the adapted queries, reveal that the NoSQL database is ways slower when it comes to elaborate server-side calculations.

Contents

Introduction

In a modern society, nearly every person runs across databases - wittingly or unwittingly. For this interaction, people want always a good performance. Database designers are confronted with the task to get reasonable speed and efficiency. That's where the non-profit-corporation TPC comes into play. They offer database benchmarks for different business uses like inventory control or banking systems. This paper goes into the TPC-H benchmark, a "decision support benchmark". "It consists of a suite of business oriented ad-hoc queries and concurrent data modifications." [1]. The 22 queries in the TPC-H benchmark are written in SQL, which are easy to execute on relational databases. This paper analyzes the performances of six different TPC-H queries using a non-relational database.

In contrast to relational databases, non-relational databases don't use a determinate schema with the goal to avoid joins. Beside Google BigTable and Amazon Dynamo, the two of the big players, there exist a handful of other NoSQL databases. One of them is MongoDB, which I will use in this research. Its performance will be compared with PostgreSQL.

The first part of this paper introduces MongoDB and some interesting features. I will show how to adapt a SQL schema and how queries can be mapped to MongoDB. It is not a full guide for MongoDB, rather a short and crisp lead-in to understand the "TPC-H in MongoDB" section. This shows the full MongoDB schema and some picked TPC-H queries mapped to MongoDB. The benchmark results and a corresponding discussion rounds off the paper.

MongoDB

MongoDB is an open-source and document-oriented and schema-free DBMS, storing data in BSON, the binary encoded JSON format [2]. The database is developed under four focuses: "Flexibility", "power", "speed/scaling" and "ease of use" [3]. This paper investigates primary the "speed" attribute, however, we make use of the "flexibility" while designing the schema and of the "ease of use" for our fitted queries. The mapping chart (Table 1) helps creating a common base.

SQL term	MongoDB term
database	database
table	collection
index	index
row	document
column	field
join	embedding and linking
primary key	_id field
group by	aggregation

Table 1: Mapping chart between SQL and MongoDB [4]

"[...] MongoDB does not support joins and so, at times, requires bit of denormalization." [5]. To get around the problem, one could link the objects and compute the joins client-side using a driver. However, to compare the performance of the database, this is no option for us. The task is to find a schema performing with all queries, which would work in SQL, too. The only way to avoid joins is to put all the data into a single collection, where "embedding" is the keyword. A simple example (Figure 1) helps for a better understanding.

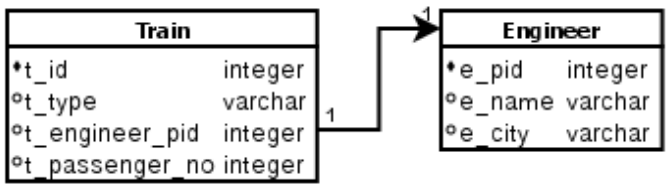

Figure 1: Example of a SQL schema

t_id	t_type	t_engineer_pid	t_passenger_no
10	EuroStar	2	37
28	Turbo	17	129

Table 2: Sample data for "train" table

e_pid	e_name	e_city
12	Sparrow	Effretikon
2	Meyer	Bern
17	Kingsley	Luzern

Table 3: Sample data for "engineer" table

One train has exactly one engineer and vice versa, which is called a 1:1 relationship. When the user likes to know how the engineer of a certain train (e.g. with id=28) is called, he would do a join of the two tables:

```
SELECT e_name FROM Train, Engineer WHERE t_id = 28 AND t_engineer_pid = e_pid;
```

As already mentioned above, MongoDB does not provide joins, that's why we have to modify the given schema. Instead of two tables, we put all information into one collection. This means, that we embed the engineer into the train object (it would naturally also work to embed the train-object into the engineer-object, but it is less intuitive). The unused engineer with e_pid = 12 is dropped.

The general schema:

```
{
        _id : integer,
        type : string,
        passenger_no : integer,
        engineer : {
                pid : integer,
                name : string,
                city : string }
}
```

The sample data looks like this:

```
{
        _id : 10,
        type : Eurostar,
        passenger_no : 37,
        engineer : {
                pid : 2,
                name : Meyer,
                city : Bern }
}
{
        _id : 28,
        type : Turbo,
        passenger_no : 129,
        engineer : {
                pid : 17,
                name : Kingsley,
                city : Luzern }
}
```

Note for getting a better readability, I renamed some fields. For the queries, MongoDB uses JSON-like syntax. The same query as before for MongoDB would now look like this:

```
db.train.find( { "_id" : 28}, { "engineer.name" : 1} );
```

The content of the first pair of curly brackets represents the condition, while the second part specifies, that only the name should be returned. Enclosed objects are simply accessed using the dot-notation.
We can modify the schema (Figure 2) and replace the "engineer" table by a "passenger" table, so that it contains now a 1:n relationship between train and passenger.

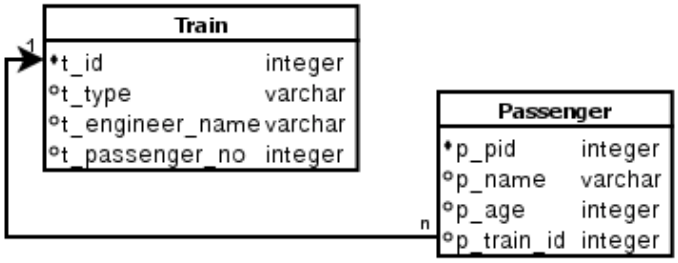

Figure 2: The modified schema

The 1:n relationship can be mapped to two very different MongoDB schemata:

```
{
        _id : integer,
        type : string,
        passenger_no : integer,
        engineer_name : string,
        passengers : [
                {       pid : integer,
                        name :string,
                        age : integer },
                {       pid : integer,
                        name :string,
                        age : integer },
                ...
                ]
}
```

```
{
        _id : integer,
        name : string,
        age : integer,
        train {
                id : integer,
                type : string,
                engineer_name : string,
                passenger_no : integer }
}
```

On the left, probably the more visceral, all passengers are stored as an array, embedded into the train-object. At the second schema, we embed the train in each passenger. The first schema is the more memory-efficient way because there is no data-duplication. But an array makes querying more difficult, since we need to work with position-markers to dig into an array. The schema on the right side stores the trains multiple times (as many as passengers exist) and therefore contains many duplicate objects, which is adverse when a field in the train-object has to be updated. Update-queries are not more difficult to design, but more demanding for the DBMS. For example if one wants to update the type of the train with id = 10, the query could be:

```
db.passengers.update({ “train.id” : 10 }, { $set : { “train.type” : “Intercity” } });
```

This query touches all passengers where the train id is 10 (first curly brackets), and updates the type of this objects, using the “$set” operator which sets the value of the field “type” to “Intercity”.
On the other hand, when we use the schema on the right side, we can use the simple dot-notation unrestricted on for all fields in the data. Which schema to take really depends on what the data is for and what is going to be queried. If there are no updates on the embedded documents and the performance is more important than memory efficiency, the schema on the right side is the better solution.
When we want to deal with numbers, there is no way around aggregation. Hitherto, MongoDB offers not so many possibilities as a “classical” SQL database does. Statements like “group by”, “avg(...)” or “sum(...)” are more complicated or need to be sidestepped by using the Map-Reduce framework [6]. Google developed and introduced the Map-Reduce model in 2004. The big advantage of this programming model is its possibility to distribute tasks which a single computer could not process [7][8]. As its name already discloses, Map-Reduce contains two functions: A map and a reduce function. The map function runs first and executes a user-defined function parallel for all entries. It stores the results in as a list of (key, value) items. The items are then grouped by the key. The reduce function collects these groups and calculates one final result. We continue the example above and look for the total sum of passengers in all trains with type “Turbo”. The SQL query would be:

```
SELECT sum(t_passenger_no) FROM Train where t_type = 'Turbo'
```

The same query in MongoDB using Map-Reduce could look like this:

```
// map function
map = function() {
      emit("sum", this.passenger_no);
};

// reduce function
red = function(k,v) {
      var i, sum = 0;
      for (i in v) {
            sum += v[i];
      }
      return sum;
};

db.train.mapReduce(map, red, { query : { "type" : "Turbo"}, out: "result"});
```

The map-function goes through all documents and emits the number of passengers under the keyword "sum". This output is the input for the reduce-function, which summarizes the number of passengers. The final result is stored in a new collection called "result".
Developers are currently designing an "aggregation framework", which is a tool to calculate aggregates without using Map-Reduce [9]. For simple cases like totaling or averaging, using the powerful Map-Reduce is like taking a sledgehammer to crack a nut. For people acquainted with SQL, group-by or distinct statements can be imitated using the aggregation framework. The same totaling query as above using the aggregation framework:

```
db.runCommand( { aggregate : "train", pipeline : [
      { $match : { "type" : "Turbo"} },
      { $group : {
            _id : "sum_all_passengers",
            sum : { $sum : "$passenger_no" }
      }}
]});
```

All the items are pushed through the pipeline. The "match" condition eliminates non-matching documents and the "group" statement summarizes the passengers.

TPC-H in MongoDB

Schema

With the 22 queries, TPC offers the matching schema (Figure 3).

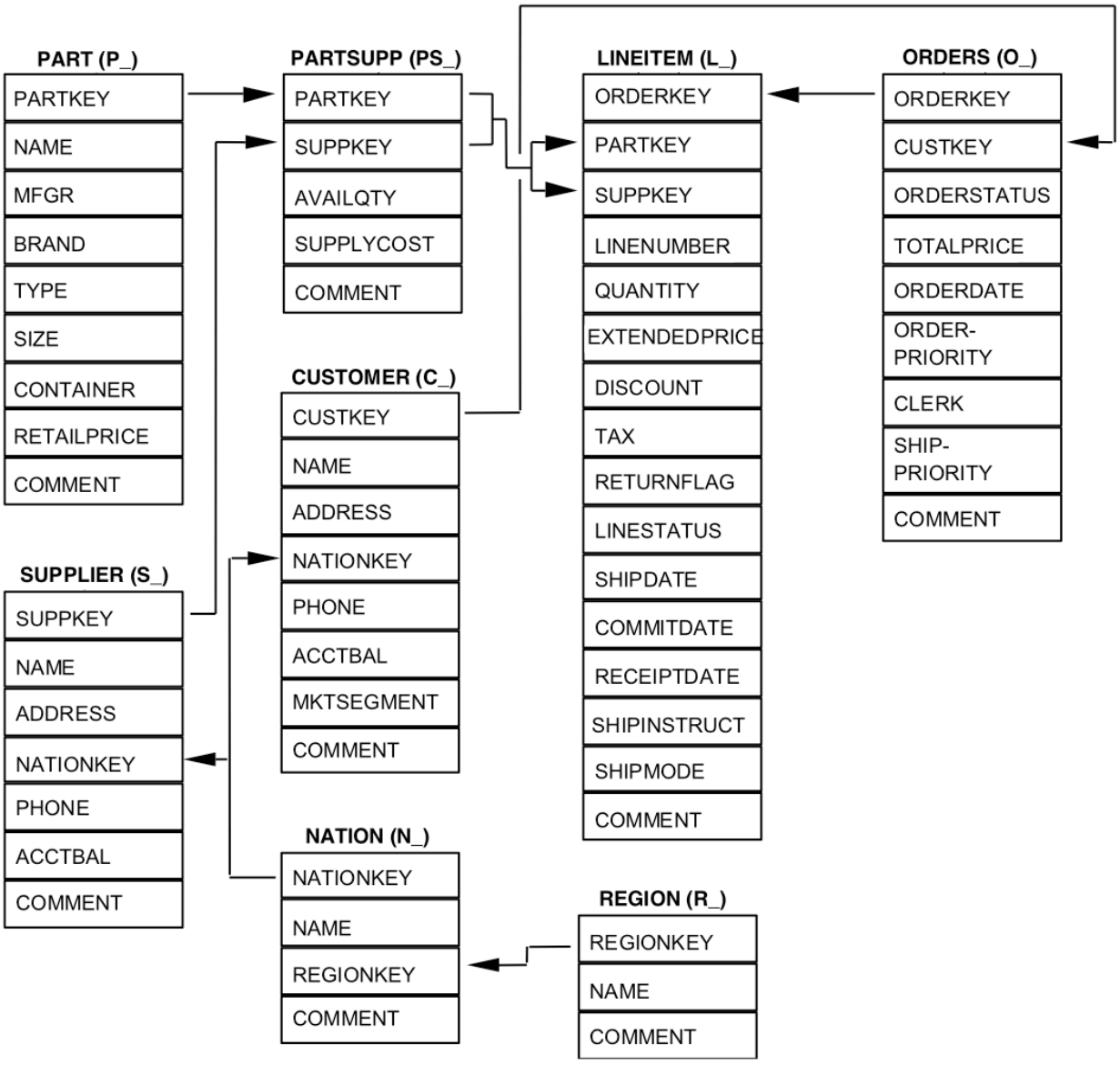

Figure 3: Original TPC-H schema [10]

Using this schema as it is in MongoDB would be possible only for those queries touching exactly one table. Like in the example above, we modify the schema and embed all tables into a single collection.
As we seek for performance and the data is not going to be updated, we do not use arrays to map 1:n relation. The connection “orders”:”lineitem” is for example a 1:n relation. Instead of embedding all “lineitems” into an array in “orders”, which would withal look very messy, we embed for each “lineitem” an “order”. One order may be stored multiple times now.
We can unfold the SQL schema starting with the “lineitem” object as it does not contain a primary key, and therefore no other table is dependant from it (no foreign key possible). We replace now successively all foreign keys of “lineitem”. Let’s start with the “orderkey”: Searching for the order with the given key returns a full order object. This object we put into the “lineitem” object. As the found order has the foreign key “custkey”, we search for the customer with the key and insert the whole customer-object into the order-object. Continuing this technique recursively for all foreign keys results in our final schema (JSON):

```
{
	_id : ObjectID,
	linenumber,
	quantity,
	extende dprice,
	discount
	tax,
	returnflag,
	linestatus,
	shipdate,
	commitdate,
	receiptdate,
	shipinstruct
	shipmode,
	comment,
	order : {
		orderkey,
		orderstatus,
		totalprice,
		orderdate,
		orderpriority,
		clerk,
		shippriority,
		comment,
		customer : {
			custkey,
			name,
			address,
			phone,
			acctbal,
			mktsegment,
			comment,
			nation : {
				nationkey,
				name,
				comment,
				region : {
					regionkey,
					name,
					comment } } } },
	partsupp : {
		availqty,
		supplycost,
		comment,
		part : {
			partkey,
			name,
			mfgr,
			brand,
			type,
			size,
			container,
			retailprice,
			comment },
		supplier : {
			suppkey,
			name,
			address,
			phone,
			acctbal,
			comment,
			nation : {
				nationkey,
				name,
				comment,
				region : {
					regionkey,
					name,
					comment } } } }
}
```

As an advantage, this schema is clearer than the SQL counterpart: one object represents one “deal”. While searching for a whole deal, there is no need to touch multiple tables. The assembled schema has its drawback on higher memory usage. For example a customer with all its details is stored twice when he orders two parts. As that data is stored multiple times, updating is more demanding than it is in the initial schema. Depending on the business, updating former deals is not required and it’s even better to have a snapshot from thence. As we do not update the data, this schema is legal in our case.
During my work, converting the data from the, SQL-styled format with multiple tables into one single collection turned out to be more demanding as I thought. Starting with an ambitious data set of 6 million lineitems, I had to admit that it exceeded my available computation power. Every foreign key in the table needs to be replaced by the full entry. As there are nine embedded documents in the “deal” collection, just as many queries have to be done for each deal. Lowering the count of lineitems to 600’000 (10% of the original data size) resolved the problem but maybe also one’s sights concerning the performance comparability.

Queries

As already mentioned, to benchmark the server performance, the queries should run completely on the MongoDB server. I realized that the new aggregation framework offers in many cases a suitable function. Since it is still in development process, it did not make it to the stable release of MongoDB yet. That’s the reason why the minimum version to run the following queries is 2.1.0. The original TPC-H queries, written in SQL, can be found in the attachments section on the end of this paper.

Pricing Summary Record Query (Q1)

„The Pricing Summary Report Query provides a summary pricing report for all lineitems shipped as of a given date. The date is within 60 - 120 days of the greatest ship date contained in the database. The query lists totals for extended price, discounted extended price, discounted extended price plus tax, average quantity, average extended price, and average discount. These aggregates are grouped by RETURNFLAG and LINESTATUS, and listed in ascending order of RETURNFLAG and LINESTATUS. A count of the number of lineitems in each group is included.“ [10]

```
// TPC-H Query 1 for MongoDB using the native group-statement

// reduce function
var red = function(doc, out) {
      out.count_order++;
      out.sum_qty += doc.quantity;
      out.sum_base_price += doc.extendedprice;
      out.sum_disc_price += doc.extendedprice * (1 - doc.discount);
      out.sum_charge += doc.extendedprice * (1 - doc.discount) * (1 + doc.tax);
      out.avg_disc += doc.discount // sum the discount first
};

// finalize function
var avg = function(out) {
      out.avg_qty = out.sum_qty / out.count_order;
      out.avg_price = out.sum_base_price / out.count_order;
      out.avg_disc = out.avg_disc / out.count_order // calculate the average of the discount
};

db.deals.group( {
      key : { returnflag : true, linestatus : true},
      cond : { "shipdate" : {$lte: new Date(1998, 8, 1)}}, // month is 0-indexed
      initial: { count_order : 0, sum_qty : 0, sum_base_price : 0, sum_disc_price : 0,
            sum_charge : 0, avg_disc : 0},
      reduce : red,
      finalize : avg
});
```

The first TPC-H query does not contain complex statements. All it does is totaling and averaging some fields of the deals. Exempted from the impossibility of group and order commonly, Map-Reduce offers a good focal point. “Grouped data should simply be ordered client-side”, proposes the MongoDB guide [6]. We therefore simply forgo the order-statement.
Alternatively, we can use the aggregation framework, where it is possible to group and order in combination.

```
// TPC-H Query 1 for MongoDB using the aggregation framework

// use aggregation framework
db.runCommand( { aggregate : "deals", pipeline : [
      { $match : { "shipdate" : { $lte: new Date(1998, 8, 1) }} },
      { $group : {
            _id : { "returnflag" :1, "linestatus" : 1},
            sum_qty : { $sum : "$quantity"},
            sum_base_price : { $sum : "$extendedprice"},
            sum_disc_price : { $sum : { $multiply : [ "$extendedprice",
                              { $subtract : [1, "$discount"]}] }},
            sum_charge : { $sum : {$multiply : [ "$extendedprice",
                              { $subtract : [1, "$discount"]}, {$add : [1, "$tax"]}] }},
            avg_qty : { $avg : "$quantity"},
            avg_price : { $avg : "$extendedprice"},
            avg_disc : { $avg : "$discount"},
            count_order : { $sum : 1}
            } },
      { $sort : {"_id.returnflag" : 1, "_id.linestatus" : 1}}
]});
```

National Market Share Query (Q8)

*„The market share for a given nation within a given region is defined as the fraction of the revenue, the sum of [l_extendedprice * (1-l_discount)], from the products of a specified type in that region that was supplied by suppliers from the given nation. The query determines this for the years 1995 and 1996 presented in this order.“ [10]*

```
// TPC-H Query 8 for MongoDB

// subquery
var start = new Date(1995, 0, 1); // month is 0-indexed
var end = new Date(1996, 11, 31);

var subquery = { $and : [
      {"order.customer.nation.region.name" : { $regex : '^AMERICA'}},
      {"partsupp.part.type" : "ECONOMY ANODIZED STEEL" },
      {"order.orderdate" : { "$gte" : start, "$lt" : end } }
]};

var volume_each_nation = db.runCommand(
      { aggregate : "deals", pipeline : [
            { $match : subquery}, // eliminate items which are not matching
            { $project : {
                  "_id" : 0, // remove the id field
                  "o_year" : {$year : "$order.orderdate"}, // extract the year
                  "volume" : {$multiply : ["$extendedprice", {$subtract : [1, "$discount"] }] },
                  "nation" : "$partsupp.supplier.nation.name"
                  }
            }
      ]});

// cache the result temporarily in the database
db.tmp.insert(volume_each_nation.result);

// process result of subquery (stored in the database)
var red = function(doc, out) {
      out.o_year = doc.o_year;
      out.total_sum += doc.volume; // helper field to calculate market share
      if (doc.nation == "BRAZIL") { // sum mkt_share of the country
            out.mkt_share += doc.volume;
      }
};

var share = function(out) {
      out.mkt_share = out.mkt_share / out.total_sum
      delete out.total_sum; // remove the total_sum field
};

db.tmp.group( {
      key : { o_year : true },
      initial : { total_sum : 0, mkt_share : 0 },
      reduce : red,
      finalize : share
});
```

The so-called “National Market Share Query” touches all tables except “partsupp”. The inner SQL query determines the volume of each nation over two years. I designed the MongoDB query in the same matter. The interim result is calculated by using the aggregation framework. A better performance can be reached when only matching items come to demanding computation. That’s why we put the “match” statement before the “project” statement. Since the aggregation is still in development progress, there is no support for directly storing the result in a collection, yet. We have to cache the interim result manually in a collection called “tmp”. The second part, analog to the outer SQL statements, groups this interim results in the same line as we have seen it in Query 1. We omit the order-statement once again.

Top Supplier Query (Q15)

„The Top Supplier Query finds the supplier who contributed the most to the overall revenue for parts shipped during a given quarter of a given year. In case of a tie, the query lists all suppliers whose contribution was equal to the maximum, presented in supplier number order.“ [10]

```
// TPC-H Query 15 for MongoDB

// month is 0-indexed
var subquery = {"shipdate" : { "$gte" : new Date(1996, 0, 1), "$lt" : new Date(1996, 3, 1) } };

// extracts the total revenue of each supplier
var eachsupp = db.runCommand(
      { aggregate : "deals", pipeline : [
            { $match : subquery },
            { $project : {
                  "_id" : 0, // remove _id field
                  "revenue" : {$multiply : ["$extendedprice", {$subtract : [1, "$discount"] }] },
                  "supplier_no" : "$partsupp.supplier.suppkey",
                  "name" : "$partsupp.supplier.name",
                  "address" : "$partsupp.supplier.address",
                  "phone" : "$partsupp.supplier.phone" }},
            { $group : {
                  _id : "$supplier_no",
                  total_revenue : { $sum : "$revenue" },
                  name : {$first : "$name"},
                  address : {$first : "$address"},
                  phone : {$first : "$phone"} }}
      ]});


// store the result in the database
db.tmp.insert(eachsupp.result);

// find the top supplier
db.tmp.find().sort({total_revenue : -1}).limit(1);
```

Once more, I used the aggregation framework to implement this query. We sum the revenue for suppliers and group him or her by their unique supplier number. The provisional result is again stored in a temporary collection, which we query to find the maximum revenue. Doing this with the “sort and limit-by-one” technique restricts us to one single result, even if two suppliers would have the same total revenue.

Potential Part Promotion Query (Q20)

„The Potential Part Promotion query identifies suppliers who have an excess of a given part available; an excess is defined to be more than 50% of the parts like the given part that the supplier shipped in a given year for a given nation. Only parts whose names share a certain naming convention are considered.“ [10]

```
// TPC-H Query 20 for MongoDB

var query_part = {
      "shipdate" : { "$gte" : new Date(1994, 0, 1), "$lt" : new Date(1995, 0, 1) },
      "partsupp.supplier.nation.name" :
            { $regex : '^CANADA'}, // use regex because of whitespaces in the end
      "partsupp.part.name" :
            { $regex : '^forest', $options : 'i' } // option i makes case insensitive
};

var red = function(doc, out) {
      out.sum += doc.quantity;
}

// calculate the the total quantity first
var half_total_quantity = db.deals.group( {
      key : "sum_quantity",
      cond : query_part,
      initial : { sum : 0 },
      reduce : red
})[0].sum / 2;


db.deals.find( { "$and" : [
      query_part,
      {"partsupp.availqty" : { "$gt" : half_total_quantity } } ]},
      {"_id" : 0, "partsupp.supplier.name" : 1, "partsupp.supplier.address" : 1}
).sort({"partsupp.supplier.name" : 1});
```

The main pitfall of the relatively facile “Potential Part Promotion Query” is the calculation of the total quantity of the country. It’s a single summation, which we can do with Map-Reduce. In the following you find a step-by-step explanation to comprehend the code above:

Step 0: The original SQL code snipped to translate is

```
1     select
2           0.5 * sum(l_quantity)
3     from
4           lineitem
5     where
6           l_partkey = ps_partkey
7           and l_suppkey = ps_suppkey
8           and l_shipdate >= date('1994-01-01')
9           and l_shipdate < date('1995-01-01')
```

Step 1: Lines 6 and 7 can be ignored because of the embedded schema; No joins have to be made. Lines 8 and 9 are the condition to match only a given year. We rewrite it in JSON:

```
{ "shipdate" : { "$gte" : new Date(1994, 0, 1), "$lt" : new Date(1995, 0, 1) } }
```

“$gte” is the operator for “>=”, “$lt” stands for “<”. The month is 0-indexed. This condition, we put (together with other conditions occurring in the query) into the variable “query_part”.

Step 2: To sum the quantity (line 2), we use the group-statement with a reduce function. The reduce function helps us to iterate over all entries and summing the quantity:

```
var red = function(doc, out) {
      out.sum += doc.quantity;
}
```

It takes a deal “doc” and reads the quantity, which is, according to our schema, located on the top level. This value is added to the variable “sum”, which is written into a new document called “out”. Going through

all deals, the “sum” of “out” increases. Note that the reduce function is not executed, only stored to the variable “red”.

Step 3: Now we just have to apply our prepared reduce function to the collection. The easiest way to do so is in my opinion the group-statement. We do not really need to group anything, but it helps combining the query (step 1) and the reduce function.

```
db.deals.group( {
      key : "sum_quantity",
      cond : query_part,
      initial : { sum : 0 },
      reduce : red
})
```

The “key” tag is the key to group by. As we do not need to group, we can use a constant “sum_quantity” instead. This ensures that the result is a single group. The prepared query is applied using the “cond” tag. To apply the reduce function, we have to define the used “sum” variable first. Using the “initial” tag, we initially set it to zero. Last but not least, the reduce function is added.

Step 4: The result of a group-statement is an array of all groups. As we only have a single group with the key “sum_quantity”, we can access the group with index 0. Using the dot-notation ushers us to the total sum, which we divide by 2.

```
var half_total_quantity = db.deals.group({...})[0].sum / 2;
```

Step 5: Having this value, we can put it into the “find”-condition and sort the concluding result.

Suppliers Who Kept Orders Waiting Query (Q21)

"The Suppliers Who Kept Orders Waiting query identifies suppliers, for a given nation, whose product was part of a multi-supplier order (with current status of 'F') where they were the only supplier who failed to meet the committed delivery date." [10]

```
// TPC-H Query 21 for MongoDB

var query = { $and : [
      {"order.orderstatus" : "F"},
      {"partsupp.supplier.nation.name" : { $regex : '^SAUDI ARABIA'} },
      {$where : "this.receiptdate > this.commitdate"}
]};

// check if there are other suppliers with same order
var multisupp = {
      $where : function() {
            return db.deals.findOne({ $and : [
                  { "order.orderkey" : this.order.orderkey },
                  { "partsupp.supplier.suppkey" : { "$ne" : this.partsupp.supplier.suppkey} }
                  ]}) != null;
            }
};

// make sure that no other supplier failed
var onlyfail = {
      $where : function() {
            return db.deals.findOne({ $and : [
                  { "order.orderkey" : this.order.orderkey },
                  { "partsupp.supplier.suppkey" : { "$ne" : this.partsupp.supplier.suppkey} },
                  { $where : "this.receiptdate > this.commitdate" }
                  ]}) == null;
            }
};


res = db.runCommand( { aggregate : "deals", pipeline : [
      { $match : query},
      { $match : multisupp},
      { $match : onlyfail},
      { $project : {
            _id : 0,
            s_name : "$partsupp.supplier.name" } },
      { $group : {
            _id : "$s_name",
            numwait : { $sum : 1} } },
      { $sort : { numwait : -1, _id : 1} }
]});
```

For the first time, the statements "exists" and "not exists" appear in the query, which are not supported in MongoDB. There is a "exist" flag, but it only checks if a field of a document exists. I had to write my own functions, which can be used as query conditions:

- "multisupp" returns true if there are other suppliers for a certain order
- "onlyfail" returns true if the supplier is the only one who failed delivering the part

Global Sales Opportunity Query (Q22)

„This query counts how many customers within a specific range of country codes have not placed orders for 7 years but who have a greater than average "positive" account balance. It also reflects the magnitude of that balance. Country code is defined as the first two characters of c_phone." [10]

This query is in our context very interesting. One condition for the wanted customers is that they never made an order. Using our modified MongoDB schema, this query is impossible, because we are storing whole deals. Hence, a customer that never made a deal does not find itself in the database. But why not just skipping this condition to find attractive regions? Here is the MongoDB query:

```
// TPC-H Query 22 for MongoDB

// query to match country code
var phone = { $or : [
      { "order.customer.phone" : { $regex : '^13' } },
      { "order.customer.phone" : { $regex : '^31' } },
      { "order.customer.phone" : { $regex : '^23' } },
      { "order.customer.phone" : { $regex : '^29' } },
      { "order.customer.phone" : { $regex : '^30' } },
      { "order.customer.phone" : { $regex : '^18' } },
      { "order.customer.phone" : { $regex : '^17' } }
]};

// calculate the average account balance
var avgAccBal = db.runCommand( { aggregate : "deals", pipeline : [
      { $match : phone },
      { $match : { "order.customer.acctbal" : { "$gt" : 0 } }},
      { $group : {
            _id : "avg_acc",
            avg : { $avg : "$order.customer.acctbal"} } },
]}).result[0].avg;

db.runCommand( { aggregate : "deals", pipeline : [
      { $match : phone },
      { $match : { "order.customer.acctbal" : { "$gt" : avgAccBal } }},
      { $group : { // group to have distinct customers
            _id : "$order.customer.custkey",
            bal : { $first : "$order.customer.acctbal" },
            phone : { $first : "$order.customer.phone" } }},
      { $group : {
            _id :  { "$substr" : [ "$phone", 0, 2] },
            numcust : { "$sum" : 1 },
            totacctbal : { "$sum" : "$bal"} }},
      { $sort : { "_id" : 1 }} // _id = cntrycode
]});
```

First, we calculate the average balance of all customers in the matching region. Having this value, the customers with a higher than average account balance are extracted and grouped by the region, so that we have a list of attractive regions in conclusion. Because of the schema, customers are probably stored multiple times. To prevent that a customer is factored more than once, we have to group first, due the fact that the aggregation framework does not support "distinct" out of the box.

Benchmark results

All benchmarks are running on a Mac Book Air, Intel i5 2557M, 4GB 1333Mhz, 128GB SSD (SM128C). The results are an average value of five measurements. The data contains 600'572 "deals" (MongoDB), respectively "lineitems" (PostgreSQL) and does not have any indexes.

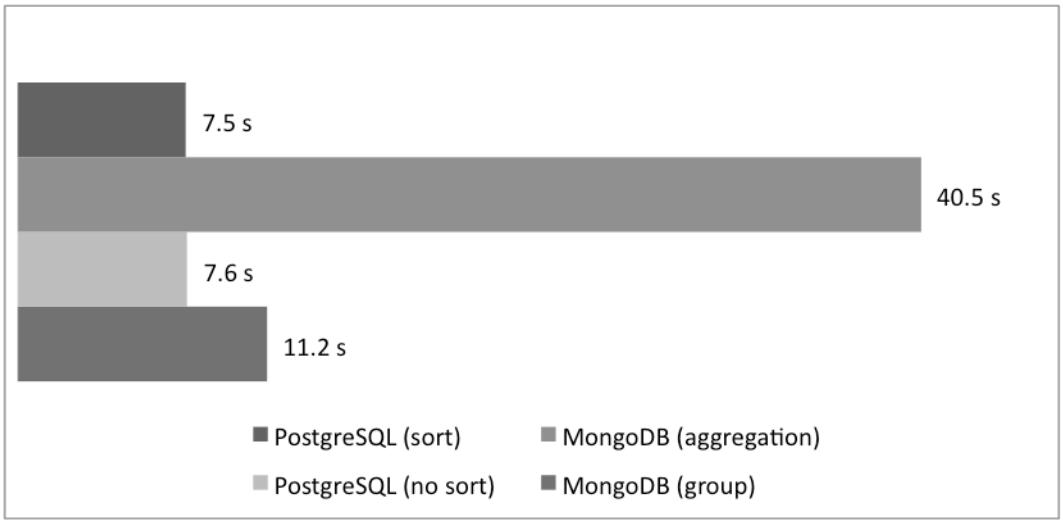

Chart 1: Pricing Summary Record Query (Q1) with parameter [Delta] = 90.

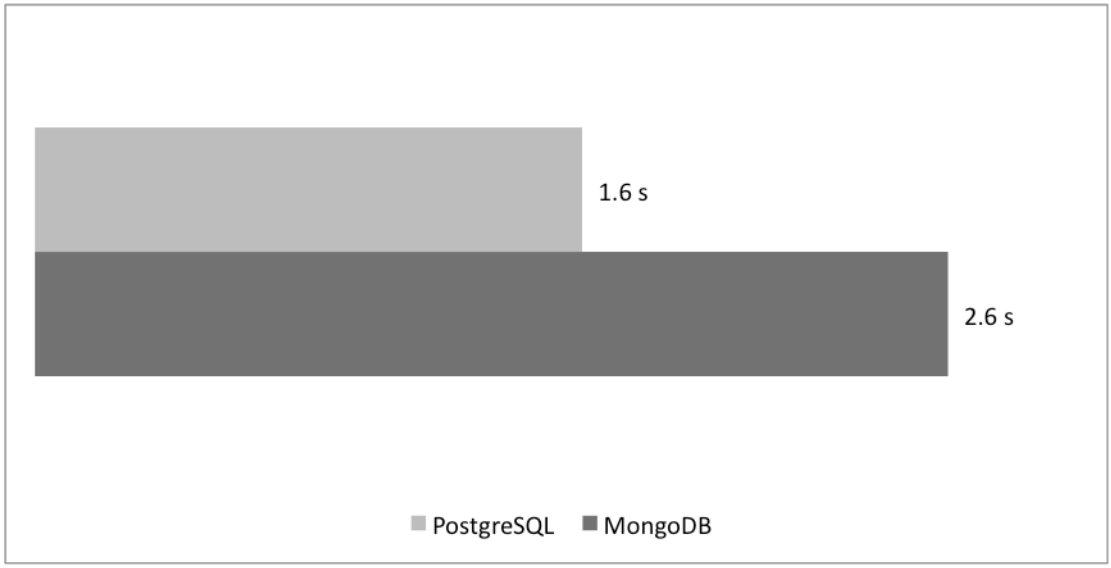

Chart 2: National Market Share Query (Q8) with parameters [Nation] = BRAZIL, [Region] = AMERICA and [TYPE] = ECONOMY ANODIZED STEEL.

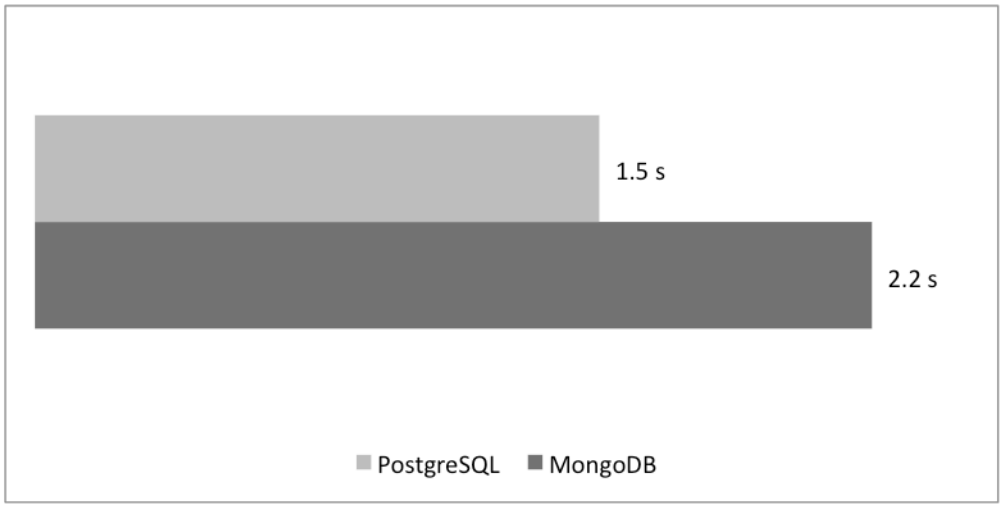

Chart 3: Top Supplier Query (Q15) with parameter [Date] = 1996-01-01

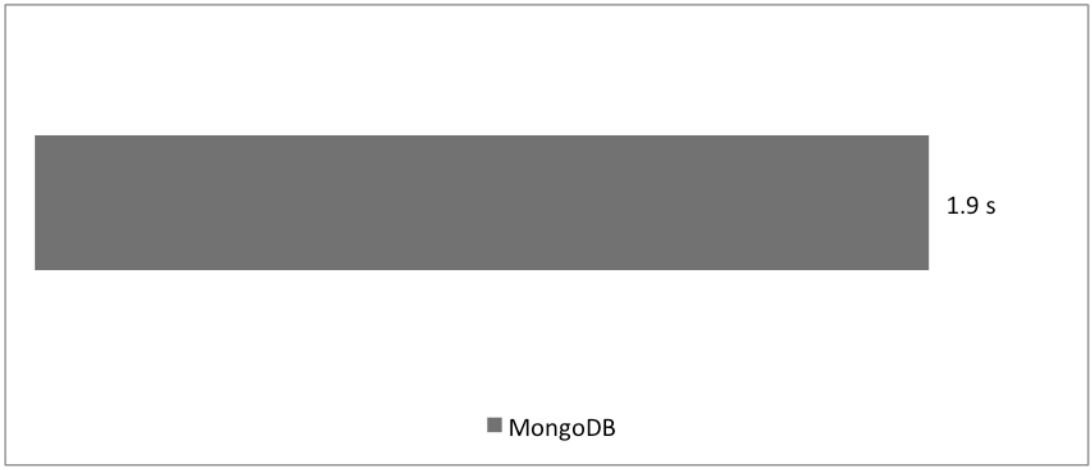

Chart 4: Potential Part Promotion Query (Q20) with parameters [Color] = forest, [Date] = 1994-01-01 and [Nation] = CANADA. PostgreSQL did not finish after 48h.

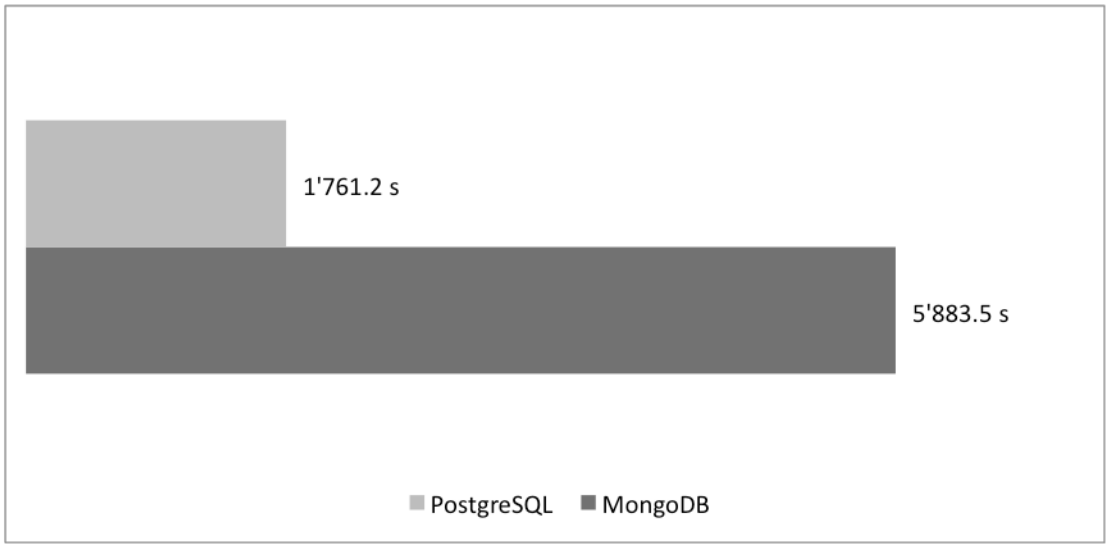

Chart 5: Suppliers Who Kept Waiting Query (Q21) with parameter [Nation] = SAUDI ARABIA. Average of only three measurements

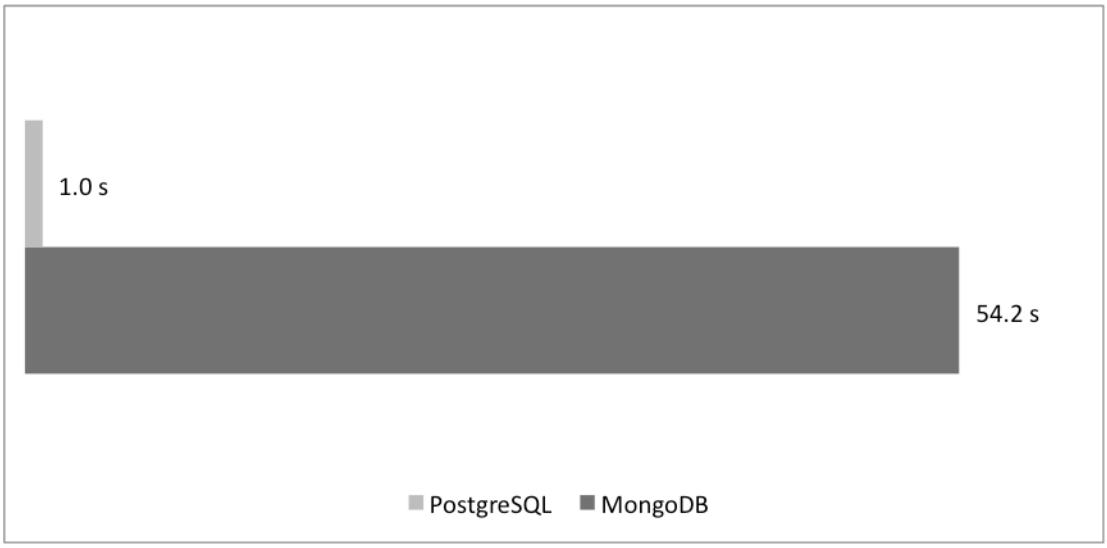

Chart 6: Global Sales Opportunity Query (Q22) with parameters [I1] = 13, [I2] = 31, [I3] = 23, [I4] = 29, [I5] = 30, [I6] = 18 and [I7] = 17

Discussion

According to the measurements above, it is conspicuous that MongoDB lags far behind PostgreSQL. When we look at Chart 1, the "Pricing Summary Report Query (Q1)", at first sight, the outlier "MongoDB (aggregation framework)" catches one's eye. Compared with the Map-Reduce method, it is almost four times slower. Other MongoDB users, as we can read in Internet forums, came up with the same conclusion. As it is still in development, it is not optimized for speed yet, rather experimental to explore the constraints. This observation strengthens when we know that the Map-Reduce concept is not optimized for performance but for distributed calculation. With the "Global Sales Opportunity Query (Q22)" we see in Chart 6 that using the aggregation framework multiple times consecutively enlarges the gap between PostgreSQL and MongoDB even further (factor *50*). In MongoDB, we need to group four times, compared with one single group-statement in PostgreSQL. From Q1 we learn that grouping with the aggregation framework is at a rough estimate five times slower than grouping in PostgreSQL. In Q22 we see four such group-statements in MongoDB, which should make the query 4x5=20 times slower than the PostgreSQL version. To this factor we can add the slower summation and averaging. The result can be seen in the benchmark.
Because of the high flexibility of the aggregation framework, I used it on almost every query. This restricts the benchmarks and the aggregation framework seems to be the bottleneck, causing that MongoDB comes off badly on all comparisons. A glimmer of light could be the "Potential Part Promotion Query (Q20)", which does not use the aggregation framework and was the fastest of all MongoDB queries. Regretfully, it does not return any result within decent time in PostgreSQL. I tried the same SQL query having indexes on "suppkey" and "partkey", but the query did not return any result, neither.
The difference on Query 21 is principally called forth by my own-written "exist" and "not exist" functions. They are not as optimized as those in PostgreSQL, when we regard that for every data entry, two such complicated queries have to be performed. Let's look closer at the "not exists" statement in PostgreSQL on a small example:

```
SELECT    l.id, l.value
FROM      t_left l
WHERE     NOT EXISTS (
                    SELECT    value
                    FROM      t_right r
                    WHERE     r.value = l.value )
```

PostgreSQL hashes the values of t_right in a hash table, which is looked up against all rows of t_left. [11] Having such a hash table is much faster than querying for every item all over again, as we do it in MongoDB.

Summary

Looking isolated at the benchmark charts, one could say that PostgreSQL is way out in front of MongoDB. Under these circumstances it is true, but one has to mention the constraints in the same breath.
First of all, it's a design problem. Having a defined RDBMS schema and optimized, but demanding queries is suboptimal and hazy to transfer to a NoSQL database. Joining in SQL is an ordinary matter. In NoSQL databases, the task is to design a schema where joins are something seldom and processed client-side. And that is already the second point: Server-side query processing is in the nature of SQL queries. [12] describes it as following: "Queries allow the user to describe desired data, leaving the database management system (DBMS) responsible for planning, optimizing, and performing the physical operations necessary to produce that result as it chooses." In contrast, MongoDB is not designed perform every operation for the user. After my extensive usage, I have more the impression that it is a very flexible storage for all kinds of data and not an analysis tool at the same time. It is pedestrian to write MongoDB queries that deliver the same results as the SQL queries do. The third tender spot to mention is that the aggregation framework, which allows coming closest to SQL queries, is not mature yet.
My concern of the data size was unjustified; I think even the reduced data gave some stable performance comparison. Running the benchmarks on a more powerful machine would maybe require larger data.
To conclude, we can lay down the prejudice that these benchmarks were a David vs. Goliath case. It is more like comparing databases designed for two totally different fields of duties. In this paper, MongoDB dabbled in the field of PostgreSQL, without having much success.

Advantages	**Disadvantages**
Flexible schema, i.e. embedding of documents, so that no joins are necessary anymore. No need for null-values because the schema can vary between two entries of the same collection.	No "perfect" schema, always depending on what the database is for, what should be queried and on the work balance between client-side and server-side execution.
Map-Reduce framework can be used to distribute long-running calculation.	Aggregation framework is still in early stages of development and therefore slow.
Uses JSON as the query language which is straightforward to understand	Complex SQL queries cannot be simply converted to MongoDB because they depend on the different schemata.

Table 2: Digest over the Advantages and Disadvantages of using MongoDB instead of a RDBMS.

Literature

[1] TPC: Home - TPC-H. http://www.tpc.org/tpch/, 2012. [Online; accessed 23-January-2012].

[2] 10gen.com: BSON - MongoDB. http://www.mongodb.org/display/DOCS/ BSON, 2012. [Online; acessed 2-February-2012].

[3] 10gen.com: Philosophy - MongoDB. http://www.mongodb.org/display/DOCS/Philosophy, 2012. [Online; accessed 2-February-2012].

[4] 10gen.com: SQL to Mongo Mapping Chart - MongoDB. http://www.mongodb.org/display/DOCS/ SQL+to+Mongo+Mapping+Chart, 2012. [Online; accessed 3-February-2012].

[5] 10gen.com: Data Modeling and Rails - MongoDB. http://www.mongodb.org/display/DOCS/ MongoDB+Data+Modeling+and+Rails, 2012. [Online; accessed 26-January-2012].

[6] 10gen.com: Aggregation - MongoDB. http://www.mongodb.org/display/DOCS/Aggregation. [Online; accessed 4-February-2012].

[7] Wikipedia: MapReduce Wikipedia. http://en.wikipedia.org/wiki/MapReduce. [Online; accessed 17-February-2012].

[8] Orend, Kai: Analysis and Classification of NoSQL Databases and Evaluation of their Ability to Replace an Object-relational Persistence Layer. TU München, 2010

[9] 10gen.com: Aggregation Framework - MongoDB. http://www.mongodb.org/display/DOCS/ Aggregation+Framework. [Online; accessed 9-February-2012].

[10] TPC: Documentation - TPC-H. http://www.tpc.org/tpch/spec/tpch2.14.2.pdf, 2011. [Online; acessed 20-December-2011].

[11] Explain Extended: NOT IN vs. NOT EXISTS vs. LEFT JOIN / IS NULL: PostgreSQL. http://explainextended.com/2009/09/16/not-in-vs-not-exists-vs-left-join-is-null-postgresql/, 2009. [Online; accessed 17-January-2012].

[12] Wikipedia: SQL Wikipedia. http://en.wikipedia.org/wiki/SQL. [Online; accessed 18-February-2012].

Attachments

TPC-H original queries

They are already filled with the same parameters as the MongoDB counterparts in this paper.

Pricing Summary Report Query (Q1)

```
select
      l_returnflag,
      l_linestatus,
      sum(l_quantity) as sum_qty,
      sum(l_extendedprice) as sum_base_price,
      sum(l_extendedprice*(1-l_discount)) as sum_disc_price,
      sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge,
      avg(l_quantity) as avg_qty,
      avg(l_extendedprice) as avg_price,
      avg(l_discount) as avg_disc,
      count(*) as count_order
from
      lineitem
where
      l_shipdate <= date '1998-12-01' -  90
group by
      l_returnflag,
      l_linestatus
order by
      l_returnflag,
      l_linestatus;
```

National Market Share Query (Q8)

```
select
      o_year,
      sum(case
            when nation = 'BRAZIL'
                  then volume
            else 0
      end) / sum(volume) as mkt_share
from (
      select
            extract(year from o_orderdate) as o_year,
            l_extendedprice * (1-l_discount) as volume,
            n2.n_name as nation
      from
            part,
            supplier,
            lineitem,
            orders,
            customer,
            nation n1,
            nation n2,
            region
      where
            p_partkey = l_partkey
            and s_suppkey = l_suppkey
            and l_orderkey = o_orderkey
            and o_custkey = c_custkey
            and c_nationkey = n1.n_nationkey
            and n1.n_regionkey = r_regionkey
            and r_name = 'AMERICA'
            and s_nationkey = n2.n_nationkey
            and o_orderdate >= date '1995-01-01'
            and o_orderdate < date '1996-12-31'
            and p_type = 'ECONOMY ANODIZED STEEL'
      ) as all_nations
```

```
group by
      o_year
order by
      o_year;
```

Top Supplier Query (Q15)

```
create view revenue (supplier_no, total_revenue) as
      select
            l_suppkey,
            sum(l_extendedprice * (1 - l_discount))
      from
            lineitem
      where
            l_shipdate >= date '1996-01-01'
            and l_shipdate < date '1996-01-01' + interval '3' month
      group by
            l_suppkey;

select
      s_suppkey,
      s_name,
      s_address,
      s_phone,
      total_revenue
from
      supplier,
      revenue
where
      s_suppkey = supplier_no
      and total_revenue = (
            select
                  max(total_revenue)
            from
                  revenue
            )
order by
      s_suppkey;
```

Potential Part Promotion Query (Q20)

```
select
      s_name,
      s_address
from
      supplier,
      nation
where
      s_suppkey in (
            select
                  ps_suppkey
            from
                  partsupp
            where
                  ps_partkey in (
                        select
                              p_partkey
                        from
                              part
                        where
                              p_name like 'forest%'
                  )
            and ps_availqty > (
                  select
                        0.5 * sum(l_quantity)
                  from
                        lineitem
                  where
                        l_partkey = ps_partkey
                        and l_suppkey = ps_suppkey
                        and l_shipdate >= date('1994-01-01')
```

```
                        and l_shipdate < date('1995-01-01')
            )
      )
      and s_nationkey = n_nationkey
      and n_name = 'CANADA'
order by
      s_name;
```

Suppliers Who Kept Orders Waiting Query (Q21)

```
select
      s_name,
      count(*) as numwait
from
      supplier,
      lineitem l1,
      orders,
      nation
where
      s_suppkey = l1.l_suppkey
      and o_orderkey = l1.l_orderkey
      and o_orderstatus = 'F'
      and l1.l_receiptdate > l1.l_commitdate
      and exists (
            select
                  *
            from
                  lineitem l2
            where
                  l2.l_orderkey = l1.l_orderkey
                  and l2.l_suppkey <> l1.l_suppkey
            )
      and not exists (
            select
                  *
            from
                  lineitem l3
            where
                  l3.l_orderkey = l1.l_orderkey
                  and l3.l_suppkey <> l1.l_suppkey
                  and l3.l_receiptdate > l3.l_commitdate
            )
      and s_nationkey = n_nationkey
      and n_name = 'SAUDI ARABIA'
group by
      s_name
order by
      numwait desc,
        s_name;
```

Global Sales Opportunity Query (Q22)

```
select
      cntrycode,
      count(*) as numcust,
      sum(c_acctbal) as totacctbal
from (
      select
            substring(c_phone from 1 for 2) as cntrycode,
            c_acctbal
      from
            customer
      where
            substring(c_phone from 1 for 2) in ('13','31','23','29','30','18','17')
            and c_acctbal > (
                  select
                        avg(c_acctbal)
                  from
                        customer
                  where
                        c_acctbal > 0.00
```

```
                        and substring (c_phone from 1 for 2) in
                                        ('13','31','23','29','30','18','17')
                )
      ) as custsale
group by cntrycode
order by cntrycode;
```

MongoDB schema mapping

The code I used to insert the data into MongoDB.

1. Map 1-to-1 into MongoDB

First, I inserted the raw data into the MongoDB in the same schema as SQL has. It was done using Java. The data is located in files with format (nation):

```
7|GERMANY|3|blithely ironic foxes grow. quickly pending accounts are b
```

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FilenameFilter;
import java.net.UnknownHostException;
import java.util.List;
import java.util.Scanner;

import com.mongodb.BasicDBObject;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.DBObject;
import com.mongodb.MongoException;

/**
 * Maps the entries just 1 to 1 to the MongoDB. There is no resolution about
 * keys.
 *
 * @author Nico Rutishauser
 *
 */
public class Map1to1 {

	private static final String DATABASE_NAME = "1to1";
	private static final String HOST = "host";
	private static final int PORT = 27017;

	public static void main(String[] args) throws UnknownHostException, MongoException {
		ConnectorHelper ch = new ConnectorHelper(); // connect to MongoDB
		DB db = ch.connectMongo(HOST, PORT, DATABASE_NAME);

		File folder = new File("resources");
		FilenameFilter filter = new FilenameFilter() {
			@Override
			public boolean accept(File arg0, String arg1) {
				return arg1.endsWith(".tbl");
			}
		};
		// read all resources files
		for (String fileName : folder.list(filter)) {
			File toRead = new File(folder, fileName);
			String collName;
			List<String> keys;
			if (fileName.startsWith("customer")) {
				keys = new Constants().MAP_CUSTOMER_1TO1;
				collName = "customer";
			} else if (fileName.startsWith("lineitem")) {
				keys = new Constants().MAP_LINEITEM_1TO1;
				collName = "lineitem";
			} else if (fileName.startsWith("nation")) {
				keys = new Constants().MAP_NATION_1TO1;
				collName = "nation";
			} else if (fileName.startsWith("order")) {
```

```
                                keys = new Constants().MAP_ORDERS_1TO1;
                                collName = "orders";
                        } else if (fileName.startsWith("part.")) {
                                keys = new Constants().MAP_PART_1TO1;
                                collName = "part";
                        } else if (fileName.startsWith("partsupp")) {
                                keys = new Constants().MAP_PARTSUPP_1TO1;
                                collName = "partsupp";
                        } else if (fileName.startsWith("region")) {
                                keys = new Constants().MAP_REGION_1TO1;
                                collName = "region";
                        } else if (fileName.startsWith("supplier")) {
                                keys = new Constants().MAP_SUPPLIER_1TO1;
                                collName = "supplier";
                        } else {
                                System.out.println("unmappable file " + fileName);
                                System.out.println("process stopped");
                                break;
                        }

                        try {
                                DBCollection collection = db.getCollection(collName);
                                collection.drop();
                                Scanner scanner = new Scanner(toRead);
                                while (scanner.hasNext()) {
                                        String line = scanner.nextLine();
                                        DBObject mongoEntry = new BasicDBObject();
                                        String[] elements = line.split("\\|");

                                        for (int i = 0; i < elements.length; i++) {
                                                mongoEntry.put(keys.get(i),
                                                        CastHelper.autoCast(elements[i]));
                                        }

                                        collection.insert(mongoEntry);
                                }
                        } catch (FileNotFoundException e) {
                                e.printStackTrace();
                        }
                }
        }
}
```

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class CastHelper {
        private final static SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
        public static Object autoCast(String entry) {
                try {
                        int asInt = Integer.valueOf(entry);
                        return asInt;
                } catch (NumberFormatException e) {
                        try {
                                double asDouble = Double.valueOf(entry);
                                return asDouble;
                        } catch (Exception e1) {
                                try {
                                        if  (entry.matches("[0-9][0-9]-[0-9][0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]-
                                                [0-9][0-9][0-9][0-9]")) {
                                                // it's a phone number that should not be parsed into a
                                                // date
                                                return entry;
                                        } else {
                                                Date date = (Date) sdf.parse(entry);
                                                return date;
                                        }
                                } catch (ParseException e2) {
                                        return entry;
                                }
```

```
                        }
                }
        }
}
```

2. Convert the data to final schema

In a second step, I converted the data to the final schema used in this paper. It may be useful to add indexes on the primary keys. The following JavaScript code can be executed directly through the MongoDB shell. Starting at "lineitem", all foreign keys are replaced by their full object.

```
dbSource = db.getSiblingDB("1to1");
dbDest = db.getSiblingDB("final");
dbDest.deals.drop();

dbSource.lineitem.find().batchSize(1000).forEach(
      function(lineitem) {
            {
                  // replace order
                  order = dbSource.orders.findOne( {"orderkey" : lineitem.orderkey} );
                  lineitem.order = order;
                  delete(lineitem.orderkey);
                  {
                        // replace customer
                        customer = dbSource.customer.findOne( {"custkey" : order.custkey} );
                        lineitem.order.customer = customer;
                        delete(lineitem.order.custkey);
                        delete(lineitem.order._id);
                        {
                              // replace nation
                              nation = dbSource.nation.findOne( {"nationkey" : customer.nationkey} );
                              lineitem.order.customer.nation = nation;
                              delete(lineitem.order.customer.nationkey);
                              delete(lineitem.order.customer._id);
                              {
                                    // replace region
                                    region = dbSource.region.findOne(
                                                {"regionkey" : nation.regionkey} );
                                    lineitem.order.customer.nation.region = region;
                                    delete(lineitem.order.customer.nation.regionkey);
                                    delete(lineitem.order.customer.nation._id);
                                    delete(lineitem.order.customer.nation.region._id);
                              }
                        }
                  }
            }

            {
                  // replace partsupp
                  partsupp = dbSource.partsupp.findOne(
                              {"partkey" : lineitem.partkey, "suppkey" : lineitem.suppkey} );
                  lineitem.partsupp = partsupp;
                  delete(lineitem.partkey);
                  delete(lineitem.suppkey);
                  delete(lineitem.partsupp._id);
                  {
                        // replace part
                        part = dbSource.part.findOne( {"partkey" : partsupp.partkey} );
                        lineitem.partsupp.part = part;
                        delete(lineitem.partsupp.partkey);
                        delete(lineitem.partsupp.part._id);
                  }

                  {
                        // replace supplier
                        supplier = dbSource.supplier.findOne( {"suppkey" : partsupp.suppkey} );
                        lineitem.partsupp.supplier = supplier;
                        delete(lineitem.partsupp.suppkey);
                        delete(lineitem.partsupp.supplier._id);
                        {
                              // replace nation
```

```
                              nation = dbSource.nation.findOne( {"nationkey" : supplier.nationkey} );
                              lineitem.partsupp.supplier.nation = nation;
                              delete(lineitem.partsupp.supplier.nationkey);
                              delete(lineitem.partsupp.supplier._id);
                              {
                                    // replace region
                                    region = dbSource.region.findOne(
                                                {"regionkey" :nation.regionkey} );
                                    lineitem.partsupp.supplier.nation.region = region;
                                    delete(lineitem.partsupp.supplier.nation.regionkey);
                                    delete(lineitem.partsupp.supplier.nation._id);
                                    delete(lineitem.partsupp.supplier.nation.region._id);
                              }
                        }
                  }
            }

            dbDest.deals.insert( lineitem );
      }
);
```