A Benchmark for Multidimensional Statistical Data

Philipp Baumgärtel*, Gregor Endler, and Richard Lenz

Institute of Computer Science 6,
University of Erlangen-Nuremberg
{philipp.baumgaertel,gregor.endler,richard.lenz}@fau.de

Abstract. ProHTA (Prospective Health Technology Assessment) is a simulation project that aims at estimating the outcome of new medical innovations at an early stage. To this end, hybrid and modular simulations are employed. For this large scale simulation project, efficient management of multidimensional statistical data is important. Therefore, we propose a benchmark to evaluate query processing of this kind of data in relational and non-relational databases. We compare our benchmark with existing approaches and point out differences. This paper presents a mapping to a flexible relational model, JSON documents and RDF. The queries defined for our benchmark are mapped to SQL, SPARQL, the MongoDB query language and MapReduce. Using our benchmark, we evaluate these different systems and discuss differences between them.

1 Introduction

ProHTA (Prospective Health Technology Assessment) is a large scale simulation project within the Cluster of Excellence for Medical Technology – Medical Valley European Metropolitan Region Nuremberg (EMN).The objective of this interdisciplinary research project is to study the effects of new innovative medical technologies at a very early stage [7]. At the core of the project is an incrementally growing set of healthcare simulation modules, which are configured and calibrated with data from various sources. Typical data sources are multidimensional statistical data, like cancer registries (e.g. SEER[1]), population statistics or geographical databases. Not all data sources are initially known, though. Moreover, adding new dimensions to a multidimensional classification is common.

All these data are collected and stored in a central ProHTA database, which is required to support uncertainty management, availability and performance [1]. Consequently, the ProHTA database must have a generic general purpose database schema which allows deferred semantic annotations and incremental growth.

* On behalf of the ProHTA Research Group
[1] http://seer.cancer.gov/

A straight forward idea to store heterogeneous continuously evolving data sets with varying semantics is to use RDF triplestores. Arbitrary types of information can be expressed through sets of RDF-triples, which simply represent statements of the form subject-predicate-object. In a first prototype we use the RDF triplestore Jena, which is flexible enough to store an arbitrary number of classifications per fact [1]. However, there are many other options to store continuously growing data sets in a generic database. In order to compare the performance of these very different approaches for this problem we developed a benchmark for multidimensional statistical data.

One of the alternatives to RDF is the relational EAV (Entity-Attribute-Value) approach [11], which stores arbitrary attributes for each entity in RDF-like triples. Other possible solutions are document stores like MongoDB and CouchDB. These systems allow for storing arbitrary JSON documents and querying them with system specific query languages or MapReduce [6]. Another alternative are XML databases like BaseX. XML and document stores do not require a schema to be known upfront and entities may contain lists of arbitrary attributes.

We propose a benchmark to compare the alternatives of storing multidimensional data with an arbitrary number of dimensions per fact. To create this benchmark we used Jain's methodology [9]. We present a conceptual model and queries that are based on the requirements of the ProHTA simulation project and map these to different data management systems. We exemplify our benchmark by evaluating PostgreSQL, SQLite, MongoDB and Jena. We chose Jena because our current solution already uses it as data management system. PostgreSQL and SQLite were chosen arbitrarily and we decided to use MongoDB because of it's interesting MongoDB query language and it's MapReduce ability. So far, we have evaluated the MapReduce approach on a single processor only, in order to have a fair comparison with other non-parallel solutions. However, note that the true strength of MapReduce only lies in parallel execution, which is not taken into account in this paper. Commercial systems were excluded to be able to publish our results without restrictions.

In Sect. 2, we discuss related work. Then, we define our benchmark in Sect. 3. Sect. 4 and Sect. 5 present the mapping of the conceptual model and the queries to the specific systems. We evaluate the systems using our benchmark in Sect. 6 and conclude with a short summary and a perspective on future work.

2 Related Work

There are many existing approaches to evaluate data warehouse solutions. One of the best known is the TPC-H[2], which measures the performance for a given data warehouse schema. Another approach is the data warehouse benchmark by Darmont et. al. [5], which is able to generate arbitrary data warehouse schemata. Both approaches rely on schemata with a fixed number of dimensions. Therefore, these solutions are not suitable for the data management problem in ProHTA.

[2] `http://www.tpc.org/tpch/`

There are also approaches to evaluate NoSQL systems. The YCSB (Yahoo! Cloud Serving Benchmark) [3] evaluated CRUD (Create, Read, Update, Delete) operations for distributed data management systems. Pavlo et al. [12] compared filter, aggregation and join operations for relational and non-relational systems. They mapped their queries to MapReduce and SQL. Floratou et al. [8] utilized the YCSB to compare a distributed relational DBMS to MongoDB. Additionally, they compared the relational DBMS to Hive[3] using TPC-H. Tudorica and Bucur [14] evaluated the read and write performance of various NoSQL systems and compared the features of these systems. Cudre-Mauroux et al. [4] evaluated SciDB and MySQL for scientific applications by creating a set of 9 scientific queries based on astronomy workloads.

All of these approaches are not suitable for dynamically evolving statistical databases, as they do not evaluate generic multidimensional schemata. Additionally, some of these benchmarks only evaluate read and write operations. For ProHTA, we need to evaluate approaches for managing statistical data cubes with an arbitrary number of dimensions. Stonebraker et al. [13] argue that for each problem domain, a specialized solution performs best. They prove their point by evaluating different systems for scientific applications, data warehousing and data stream processing with application specific workloads. Therefore, we have developed a new benchmark as no existing benchmark covers our specific problem domain.

3 Definition of the Benchmark

In this section, we define the conceptual model, exemplary data, and queries of our benchmark. The data and queries are based on the requirements of the ProHTA simulation project.

3.1 Conceptual Model

This model is a simplified version of the actual ProHTA data model for heterogeneous multidimensional statistical data [1]. Each fact is a tuple consisting of an identifier, the name of a data cube, a numerical value and a set of classifications.

$$\text{fact}_i = (\text{id}_i, \text{cube}, \text{value}_i, \{\text{classification}_{i,1}, \text{classification}_{i,2}, \dots\}) \tag{1}$$

We do not support hierarchical classifications for the benchmark, as this is of no importance for performance evaluations. Each classification is a tuple consisting of a number representing the dimension and a numerical value that classifies the fact in this dimension.

$$\text{classification}_{i,j} = (\text{dimension}_{i,j}, \text{value}_{i,j}) \tag{2}$$

We use data cubes with d dimensions and n possible classification values per dimension as test data. Each data cube is dense and contains n^d facts. Dense

[3] `http://hive.apache.org/`

data cubes are common for multidimensional statistical data in the ProHTA setting. Each fact contains a random value uniformly sampled from the interval $[0, 1]$.

3.2 Query Definition

The queries are based on OLAP operators and typical queries from ProHTA simulations. We assume a data cube with d dimensions $0, 1, \ldots, d-1$ and n classification values $0, 1, \ldots, n-1$ in each dimension.
Insert measures the time to insert n^d facts.
Dice queries all facts with a value $\leq \lfloor n/2 \rfloor$ in each classification. That represents a selectivity of approximately 50% for each dimension, which is common for queries in ProHTA.
Roll Up groups the facts by the first $\max(\lfloor d/2 \rfloor, 1)$ dimensions and calculates the sum of the facts per group.
With **Add Dimension** the user can extend the classification scheme on demand. This is required for heterogeneous multidimensional data, as we don't know the classification scheme in advance. For the benchmark query, we add a classification with the value 0 and the dimension d to each fact.
Cube Join correlates the facts from data cubes c_1 and c_2 with n^d facts in each cube. For each fact i_1 from c_1 we search for the fact i_2 from c_2 with the same classifications. The resulting fact i' has the same classifications as i_1 and i_2. The value of fact i_1 is contained in i' as $\text{leftvalue}_{i'}$ and the value of the fact i_2 is contained in i' as $\text{rightvalue}_{i'}$. The scheme of the resulting facts in the cube $c_{1,2}$ is:

$$\text{joinedfact}_{i'} = (\text{id}_{i'}, c_{1,2}, \text{leftvalue}_{i'}, \text{rightvalue}_{i'}, \{\text{classification}_{i',1}, \ldots\}) \quad (3)$$

The **Cube Join** can correlate facts from different sources to either compare them or to enrich the facts from one source with information from another source.

4 Mapping

In this section, we present a mapping from our conceptual model (Sect. 3.1) to the evaluated systems.

4.1 Relational

The relational schema is based on the EAV approach:

```
fact (id, value, cube)
classification (id, fact_id[fact], value, dimension)
```

We evaluated multiple alternatives to create indexes for this mapping. We created indexes for the columns `id` (for both relations), `cube`, `factid` and for the combination of the columns (`value`, `dimension`, `factid`). These indexes showed the best performance.

Composite types and arrays are a PostgreSQL-specific alternative to the EAV approach. The classification can be modelled as a composite type and an array of classifications can be stored for each fact. However, this is not suitable for our problem as PostgreSQL does not support searching the content of arrays[4].

Additionally, we evaluated a denormalized version of this mapping to be able to compare the performance to an efficient ROLAP schema. The denormalized solution requires only one table and contains a separate column for each dimension. Despite being not flexible enough for statistical simulation data, we include this solution in our benchmark as baseline.

4.2 Document Store

A document store offers two alternatives for mapping our conceptual model to JSON documents. We can either store facts and classifications in separate documents or we can store the classifications as sub-documents. We decided to use the sub-document approach despite it's redundancy. That allows us to take advantage of the MongoDB query language, as this query language does not support joining documents.

As there is no standard schema definition language for JSON documents, we give an example for $d = 2$:

```
{id : 1,
 value: 0.5,
 cube: "Test",
 classifications : [
     {dimension : 0, value : 0},
     {dimension : 1, value : 0}]}
```

As MongoDB provides indexes, we evaluated different alternatives and decided to create indexes on `cube` and
(`classifications.value`, `classifications.dimension`).
For evaluating MapReduce with MongoDB, we did not create any indexes, as MapReduce is not able to utilize them.

4.3 RDF

The mapping to RDF is similar to the document store mapping. However, Jena does not support custom indexes. Each fact is linked to an arbitrary number of classifications. We present an exemplary fact for RDF in Turtle[5] notation for $d = 2$.

```
:f1 a              :Fact ;
    :value         0.5 ;
    :cube          "Test" ;
    :classification [a :Classification; :dimension 0; :value 0] ;
    :classification [a :Classification; :dimension 1; :value 0] .
```

[4] http://www.postgresql.org/docs/9.1/static/arrays.html
[5] http://www.w3.org/TR/turtle/

5 Query Mapping

In this section, we present a translation of exemplary most interesting queries from Sect. 3.2 for each system. Additionally, we discuss how the query complexity depends on the number of dimensions d. We assume $n = 10$ and $d = 2$ for all exemplary queries except for **Roll Up** with $d = 4$.

5.1 Relational

For example, the query for **Cube Join** is:

```
SELECT lf.value AS leftvalue, rf.value AS rightvalue,
    lc0.value AS dimvalue0 , lc1.value AS dimvalue1
FROM fact lf, fact rf, classification lc0, classification rc0,
    classification lc1, classification rc1
WHERE lf.cube = 'Test' AND rf.cube = 'Test2'
AND lc0.factid = lf.id AND rc0.factid = rf.id
AND lc0.dimension =  0 AND rc0.dimension =  0
AND lc0.value = rc0.value
AND lc1.factid = lf.id AND rc1.factid = rf.id
AND lc1.dimension =  1 AND rc1.dimension =  1
AND lc1.value = rc1.value;
```

Here, we join the facts `lf` and `rf`. As $d = 2$ for this example, we have to compare the classifications `lc0` und `lc1` of the fact `lf` with the classifications `rc0` and `rc1` of the fact `rf`. Therefore, we need $2d + 1$ joins and $5d + 2$ filter conditions.

Dice requires d Joins and $3d + 1$ filter conditions. **Roll Up** requires d Joins and $2d+1$ filter conditions. The complexity of inserting additional classifications (**Add Dimension**) does not depend on d.

For the denormalized relational solution, only the number of filter conditions for **Dice** and **Cube Join** depends on d and only **Cube Join** requires a join.

5.2 MapReduce

The MapReduce solution for **Roll Up** in pseudocode is :

```
map(fact):
    if fact.cube == 'Test':
        key_classes = []
        for classification in fact.classifications:
            if classification['dimension'] in [0,1]:
                key_classes.append(classification)
        emit({classifications : key_classes, cube: fact.cube},fact)

reduce(key, facts):
    result = {cube : key.cube, value: 0,
              classifications: key.classifications};
    for fact in facts:
        result.value += fact.value;
    return result;
```

For **Roll Up**, we generate the key in the map function to group the facts from the first classifications with dimension $\leq \max(\lfloor d/2 \rfloor, 1)$. Then, the reduce function calculates the sum of the facts. We use the map function to filter the facts for **Dice** and to add a classification to each fact for **Add Dimension**. For **Cube Join**, we use all classifications of a fact as key in the map function. Therefore, the reduce function gets two fitting facts – one from each cube – and produces the joined fact. This is known as the Standard Repartition Join [2]. The complexity of the MapReduce queries does not depend on d except for the number of filter conditions for **Dice** and the key construction of **Roll Up**.

5.3 MongoDB Query Language

Besides MapReduce, MongoDB offers a custom query language[6]. However, this language does not support joins and advanced aggregation features. Therefore, **Cube Join** and **Roll Up** can not be mapped to the MongoDB query language. However, the Aggregation Framework[7] extends the MongoDB query language. This enables us to perform the **Cube Join** and **Roll Up** queries.

Dice is mapped using the filter functionality of the MongoDB query language:

```
{'$and': [{'cube': 'Test'},
          {'classifications': {'$elemMatch':
              {'dimension': 0, 'value': {'$lte': 5}}}},
          {'classifications': {'$elemMatch':
              {'dimension': 1, 'value': {'$lte': 5}}}}
         ]}
```

The `$elemMatch` operator enables conditions for subdocuments. The number of conditions in this query is $d + 1$.

Roll Up is mapped using the Aggregation Framework, which uses pipelining of operators:

```
{'$match' : {'cube' : 'Test'}},
{'$unwind' : '$classifications'},
{'$match' : { 'classifications.dimension' :
    { '$in' : [0,1]}}},
{'$group' : { '_id' : '$_id', 'classifications' :
    {'$push' : '$classifications'},
        'value' : { '$first' : '$value'}}},
{'$group' : { '_id' : "$classifications",
    'value' : { '$sum' : '$value'}}},
{'$project' : {'value' : 1, 'classifications' : '$_id'}}
```

Here, we use `$match` to find the facts of the desired cube. Then, we split up the array containing the classifying subdocuments using `$unwind`. This produces copies of the fact document for each element of the array. These documents

[6] `http://www.mongodb.org/display/DOCS/Querying`
[7] `http://docs.mongodb.org/manual/applications/aggregation/`

contain only one classification instead of the classification array. Then, `$match` finds the classifications to group by. After that, `$group` reverses the `$unwind` operation. Now, each document contains only the classifications we want to group by. The final `$group` is the actual aggregation and `$project` produces the right output format. The number of classifications to group by is $\max(\lfloor d/2 \rfloor, 1)$.

Add Dimension filters the facts and adds a classification with `$push`. The complexity of **Add Dimension** does not depend on d.

For **Cube Join**, we utilize the Aggregation Framework again:

```
{'$match' : {'$or' : [{"cube" : 'Test'},
                      {"cube" : 'Test2'}]}},
{'$project' : {'classifications' : 1,
    'leftvalue' :
      {'$cond' : [{'$eq' : ['$cube', 'Test']}, '$value', 0]},
    'rightvalue' :
      {'$cond' : [{'$eq' : ['$cube', 'Test2']}, '$value', 0]}}},
{'$group' : {'_id' : "$classifications",
    'leftvalue' : {'$sum' : '$leftvalue'},
    'rightvalue' : {'$sum' : '$rightvalue'}}},
{'$project' : {'leftvalue' : 1, 'rightvalue' : 1,
    'classifications' : '$_id'}}
```

Conditional values (`$cond`) split up the value attributes of facts and store them in new attributes `leftvalue` and `rightvalue`. Then, we group by all classifications to match facts from one cube to the corresponding facts from the other cube. Therefore, each group contains two documents. Then, we can sum up the `leftvalue` and `rightvalue` attributes, because they either contain the desired value or 0. Again, the complexity does not depend on d.

5.4 RDF

Since version 1.1, SPARQL has supported aggregation and updates. Therefore, we are able to map all queries to SPARQL.

Dice needs d `FILTER` operators to find the desired facts. The number of triples in this query is $3d + 2$:

```
SELECT ?fact ?value ?dimvalue0 ?dimvalue1 WHERE {
?fact cube:value ?value ; cube:cube "Test" .
?fact cube:classification [:dimension 0; :value ?dimvalue0] .
FILTER( ?dimvalue0 <= 5 )
?fact cube:classification [:dimension 1; :value ?dimvalue1] .
FILTER( ?dimvalue1 <= 5 )
}
```

Roll Up uses `GROUP BY` to aggregate values and needs $3d + 2$ triples for the query. For **Add Dimension**, the `INSERT` statement generates and stores 4 additional triples for each fact in the respective cube. Finally, **Cube Join** requires $6d + 4$ triples in the query.

6 Evaluation

In this section, we present the results of our evaluation for PostgreSQL, SQLite, MongoDB and Jena. For MongoDB, we evaluated it's query language as well as MapReduce.

A Python implementation of our benchmarking framework is available for download on our Homepage[8]. Data sets and queries are generated automatically for a desired problem size. As this framework is based on a simple data model and well defined queries, it can be extended to evaluate other solutions. Additionally, unit tests guarantee that all evaluated solutions adhere to the defined semantics.

6.1 Hardware and Software Configuration

We used a quad-core computer[9] with a 2.5 inch hard drive[10] and two 4 GB DDR3 RAM modules with 1333 MHz for the evaluation. The operating system was Ubuntu 12.04 64 bit with the OpenJDK 64 bit server VM (Java version 1.6.0). We evaluated MongoDB version 2.2.0, PostgreSQL version 9.1.5, SQLite version 3.7.9 and Jena Fuseki version 0.2.3 with the TDB back end. We used 64 bit versions of the systems if they were available and did not modify the standard configuration. As journaling is activated in the standard configuration of MongoDB, all systems guarantee the durability of stored data.

6.2 Evaluation Results

We evaluated data cubes with sizes n^d ranging from 10^3 to 10^6. These cube sizes cover most of the data sets in ProHTA. For **Insert**, we measured the time to store a cube of the desired size in an empty database. We executed all other queries on a database containing two cubes of the desired size. SQLite does not start the query execution until the results are fetched. Therefore, we measured the time for executing the query and returning the results.

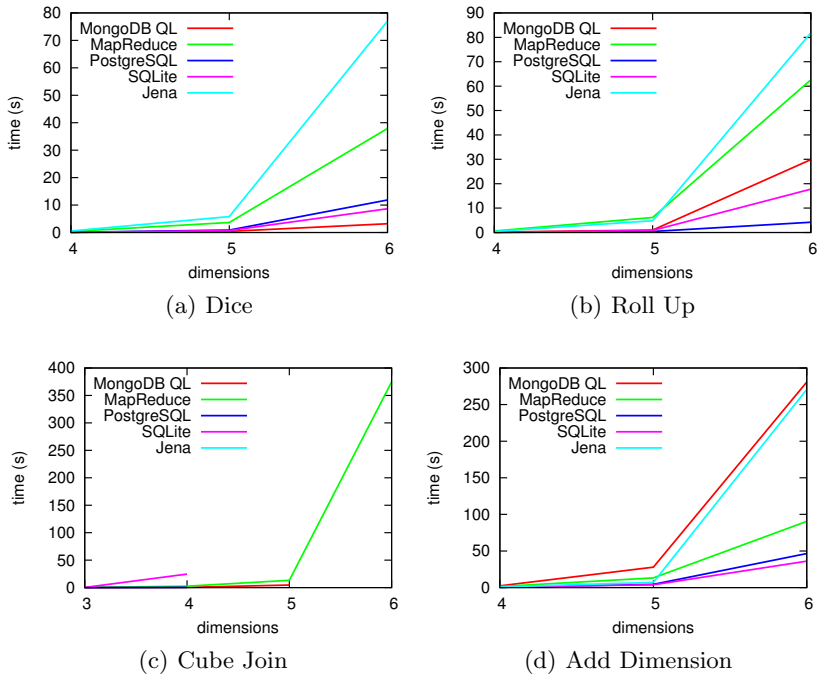
We ran each test 25 times (if possible) and compared the results to rule out caching effects. Tests with a long run time were executed only three times. We compared the results and used the first one for the evaluation if there were no significant differences.

With PostgreSQL, subsequent test runs showed significantly different results. This can be attributed to optimizations that were employed after a certain number of queries. We eliminated this behavior by running `ANALYZE` before each query. That way, PostgreSQL employed the optimizations for every query. We did not include the time for running `ANALYZE` in the results. This is valid, as we assume that statistical data for optimizations exists in our data management system.

[8] `http://www6.cs.fau.de/pb/`
[9] Intel(R) Core(TM) i5-2540M CPU @ 2.60GHz
[10] Seagate Momentus / max.: 7.200 rpm / buffer: 16 MB / bus: S-ATA II (S-ATA 300)

Query	Dimensions	MongoDB QL	MapReduce	PostgreSQL	SQLite	Jena	PostgreSQL 1T	SQLite 1T
Insert	3	**0.011** (0.003)	See	1.830 (0.024)	0.291 (0.016)	1.818 (0.126)	0.540 (0.010)	0.165 (0.022)
	4	**0.290** (0.067)	MongoDB QL	22.71 (0.255)	1.213 (0.105)	18.86 (0.910)	5.323 (0.075)	0.247 (0.051)
	5	11.86 (3.164)		276.3 (1.887)	**5.515** (1.118)	264.8	53.88 (0.503)	1.103 (0.020)
	6	215.3		2631	**61.80**	8193	510.1	8.676
Dice	3	**0.004** (2.9e-4)	0.044 (0.008)	0.012 (0.004)	0.010 (0.003)	0.068 (0.016)	0.003 (0.001)	8.6e-4 (4.4e-4)
	4	**0.048** (0.006)	0.370 (0.051)	0.072 (0.003)	0.057 (0.003)	0.528 (0.009)	0.016 (0.004)	0.007 (9.8e-5)
	5	**0.403** (0.219)	3.652 (0.050)	0.897 (0.008)	0.699 (0.005)	5.809 (0.071)	0.074 (0.004)	0.075 (3.2e-4)
	6	**3.202** (0.370)	37.90 (0.341)	11.86 (0.133)	8.668 (0.074)	77.01 (7.103)	0.494 (0.022)	0.803 (0.004)
Roll Up	3	0.015 (0.003)	0.060 (0.008)	0.004 (0.002)	**0.003** (3.0e-5)	0.036 (0.011)	0.002 (8.7e-4)	0.001 (6.8e-4)
	4	0.188 (0.005)	0.594 (0.004)	**0.041** (0.004)	0.064 (3.8e-4)	0.433 (0.010)	0.009 (0.004)	0.015 (2.8e-4)
	5	0.998 (0.665)	6.154 (0.096)	**0.408** (0.003)	0.860 (0.003)	4.844 (0.079)	0.050 (0.004)	0.163 (5.9e-4)
	6	29.86 (8.629)	62.60 (0.244)	**4.227** (0.283)	17.77 (0.257)	81.80 (4.824)	0.351 (0.003)	4.749 (0.027)
Cube Join	3	**0.045** (0.006)	0.199 (0.024)	**0.044** (0.003)	0.220 (0.007)	15.82 (0.027)	0.016 (0.005)	0.003 (5.3e-5)
	4	**0.425** (0.007)	2.597 (0.194)	0.502 (0.008)	24.68 (0.301)	–	0.105 (0.006)	0.039 (0.003)
	5	**4.570** (0.071)	13.26 (0.727)	–	–	–	0.940 (0.040)	0.441 (8.5e-4)
	6	– (Doc. Sz.)	**374.6**	–	–	–	10.77 (0.044)	5.247 (0.031)
Add Dim.	3	0.180 (0.066)	0.133 (0.021)	**0.043** (0.009)	0.309 (0.026)	0.391 (0.015)	0.240 (0.010)	0.160 (0.008)
	4	2.317 (0.392)	1.365 (0.247)	**0.319** (0.050)	0.990 (0.288)	0.910 (0.019)	0.405 (0.018)	0.178 (0.027)
	5	27.96 (6.667)	13.27 (3.263)	4.436 (0.211)	**3.819** (0.344)	7.023 (0.734)	1.751 (0.351)	0.190 (0.026)
	6	280.7	90.42	46.48	**36.04**	270.7	16.98 (0.619)	0.166 (0.008)

Table 1. Time (s) and standard deviation for the queries



(a) Dice (b) Roll Up

(c) Cube Join (d) Add Dimension

Fig. 1. Evaluation of the queries

Table 1 and Fig. 1 show the results of our evaluation. For each test run, we present in Table 1 the average time (in seconds) and the standard deviation if we were able to run the test 25 times. We aborted each query except **Insert** aft 600 seconds ("–" in the table). "PostgreSQL 1T" and "SQLite 1T" are the denormalized relational solutions. We did include these solutions in our evaluation despite being not flexible enough to manage heterogeneous data. The fastest solution for each problem size and query is printed in boldface (excluding the denormalized solutions). As an overview, we depicted the results in Fig. 1. Because of space limitations, we omitted **Insert** and for clarity reasons, we did not plot the results for the denormalized relational solutions.

PostgreSQL performs the **Cube Join** very fast by using hash joins. However, for larger cubes ($d > 4$) the available memory was not sufficient. Therefore, PostgreSQL was not able to perform the **Cube Join** in less than 600 seconds. MongoDB was not able to perform the **Cube Join** for $d > 5$. This was because the Aggregation Framework stores all results in one single document and the result exceeded the hard coded maximum document size ("– Doc. Sz." in the table). To be able to perform the cube join for $d = 5$, we increased the maximum document size from 16MB to 256MB in the MongoDB code. However, we were not able to increase this limit further because of integer overflows. For Jena, we extended the Java heap space to 4GB to be able to evaluate **Add Dimension** for $d = 6$.

In conclusion, MongoDB seems to be most suitable for managing simulation data in ProHTA. The MongoDB query language is fast and MapReduce is the only solution that allows for large **Cube Joins**. PostgreSQL is very fast for **Roll Up** queries and for adding dimensions but is slow for **Insert**. Jena is too slow and can not perform **Cube Joins** for cubes with more than 10^3 facts.

6.3 Evaluating a Prefilled Database

In this section, we evaluate the dependency between the performance and the amount of data in the data management system. To this end, we evaluated **Dice** for cubes with 10^4 facts for an empty database and for a database prefilled with 100 cubes containing 10^4 facts each. This amount of data is realistic for a large healthcare simulation project.

Load	0	$10^4 \cdot 100$
MongoDB QL	0.048 (0.006)	0.040 (0.007)
MapReduce	0.370 (0.051)	5.882 (0.054)
PostgreSQL	0.072 (0.003)	0.064 (0.005)
PostgreSQL 1T	0.016 (0.004)	0.016 (0.006)
SQLite	0.057 (0.003)	0.081 (0.004)
SQLite 1T	0.007 (9.8e-5)	0.018 (0.005)
Jena	0.528 (0.009)	0.595 (0.052)

Table 2. Time (s) and standard deviation of the dice query

Table 2 presents the results. The difference for the MongoDB query language is within the limits of the error of measurement. However, MapReduce depends heavily on the amount of data in the database. This is because MapReduce has to process each document in the database. In MongoDB, MapReduce and the filter from the MongoDB query language can be combined. That way, the amount of data in the database does not influence the performance of MapReduce queries. PostgreSQL showed no significant difference between the two experiments. Jena and SQLite showed a small dependency on the amount of data in the database. Therefore, the amount of data in the ProHTA simulation project does not influence whether or not the solutions are suitable.

7 Conclusions and Future Work

This paper proposed a benchmark to evaluate solutions for storing heterogeneous multidimensional statistical data. The benchmark is based on the data management of a large scale healthcare simulation project. The data model in this project is based on the EAV approach. As EAV is a widespread solution to create generic data models [10], our benchmark is valid for a large number of applications besides statistical data.

We simplified the data model to be able to map it easily to a large number of different data management solutions. However, the simplified data model is still close enough to the real model to be able to use the evaluated solutions in the ProHTA project.

We created a set of well-defined queries, which can be mapped to various query languages. These queries are based on realistic and typical queries from healthcare simulations. The mapping of queries enables evaluating the expressiveness of the target query languages as well as their performance.

No clear winner has emerged from our evaluation. As expected, all solutions were far slower than the denormalized relational approach. However, as a generic data model is required, we can not rely on this solution. SQLite was even faster than MongoDB for inserting the facts; PostgreSQL and Jena were very slow. For **Dice**, the MongoDB query language was faster than the relational solutions and more than 20 times faster than Jena. For **Roll Up**, PostgreSQL performed fastest. Despite being slow, MapReduce was the only solution that was able to perform the **Cube Join** for $d > 4$. For MapReduce, there was no need to create indexes, therefore it performed **AddDimension** faster than the MongoDB query language. Jena showed the slowest performance for each query and was not able to perform the **Cube Join** for $d > 3$.

In future work, we need to evaluate other systems to be able to decide which solution suits the ProHTA data management problem best. The dependency between performance and amount of data in the database should be evaluated more thoroughly. In addition to that, we need to evaluate how the queries can be performed in parallel. This is because we did not utilize this strength of MapReduce and MongoDB in our benchmark yet. Also, sparse data cubes with a high number of dimensions need to be evaluated.

Using this benchmark, we can define a catalog of criteria to find the best system for a wide range of data management problems with certain characteristics. Additionally, we can evaluate optimization strategies for each solution.

Acknowledgements

This project is supported by the German Federal Ministry of Education and Research (BMBF), project grant No. 13EX1013B.

References

1. Baumgärtel, P., Lenz, R.: Towards data and data quality management for large scale healthcare simulations. In: Conchon, E., Correia, C., Fred, A., Gamboa, H. (eds.) Proceedings of the International Conference on Health Informatics. pp. 275–280. SciTePress - Science and Technology Publications (2012), iSBN: 978-989-8425-88-1
2. Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., Shekita, E.J., Tian, Y.: A comparison of join algorithms for log processing in mapreduce. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. pp. 975–986. SIGMOD '10, ACM, New York, NY, USA (2010)

3. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM symposium on Cloud computing. pp. 143–154. SoCC '10, ACM, New York, NY, USA (2010)
4. Cudre-Mauroux, P., Kimura, H., Lim, K.T., Rogers, J., Madden, S., Stonebraker, M., Zdonik, S.B., Brown, P.G.: Ss-db: A standard science dbms benchmark (2012), (submitted for publication)
5. Darmont, J., Bentayeb, F., Boussad, O.: Dweb: A data warehouse engineering benchmark. In: Tjoa, A., Trujillo, J. (eds.) Data Warehousing and Knowledge Discovery, Lecture Notes in Computer Science, vol. 3589, pp. 85–94. Springer Berlin / Heidelberg (2005), 10.1007/115468499
6. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (Jan 2008)
7. Djanatliev, A., Kolominsky-Rabas, P., Hofmann, B.M., Aisenbrey, A., German, R.: Hybrid simulation approach for prospective assessment of mobile stroke units. In: SIMULTECH 2012 - Proceedings of the 2nd International Conference on Simulation and Modeling Methodologies, Technologies and Applications. pp. 357 – 366 (2012)
8. Floratou, A., Teletia, N., Dewitt, D.J., Patel, J.M., Zhang, D.: Can the elephants handle the nosql onslaught? In: Proceedings of the VLDB Endowment. vol. Volume 5 (2012)
9. Jain, R.: The art of computer systems performance analysis. John Wiley & Sons, Inc. (1991)
10. Lenz, R., Elstner, T., Siegele, H., Kuhn, K.A.: A practical approach to process support in health information systems. Journal of the American Medical Informatics Association 9(6), 571–585 (2002)
11. Nadkarni, P.M., Marenco, L., Chen, R., Skoufos, E., Shepherd, G., Miller, P.: Organization of heterogeneous scientific data using the eav/cr representation. Journal of the American Medical Informatics Association 6(6), 478–493 (1999)
12. Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., DeWitt, D.J., Madden, S., Stonebraker, M.: A comparison of approaches to large-scale data analysis. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of data. pp. 165–178. SIGMOD '09, ACM, New York, NY, USA (2009)
13. Stonebraker, M., Bear, C., Çetintemel, U., Cherniack, M., Ge, T., Hachem, N., Harizopoulos, S., Lifter, J., Rogers, J., Zdonik, S.: One size fits all? - part 2: benchmarking results. In: Proceedings of the 3rd Conference on Innovative Data Systems Research (CIDR) (2007)
14. Tudorica, B., Bucur, C.: A comparison between several nosql databases with comments and notes. In: Roedunet International Conference (RoEduNet), 2011 10th. pp. 1 –5 (june 2011)