EVALUATION OF DATABASE TECHNOLOGIES FOR USAGE IN DYNAMIC DATA MODELS

A Comparison of Relational, Document Oriented and Graph Oriented Data Models

Alexander Wendt[1], Benjamin Dönz[1], Stephan Mantler[2], Dietmar Bruckner[1] and Alexander Mikula[3]

[1] *Institute of Computer Technology, Vienna University of Technology, Gusshausstrasse 27-29, A-1040 Vienna, Austria*

[2] *VRVis Zentrum für Virtual Reality und Visualisierung Forschungs-GmbH, Donau-City-Strasse 1, A-1220 Vienna, Austria*

[3] *Frequentis AG, Innovationsstrasse 1, A-1100 Vienna, Austria*

Keywords: Data model, RDF, Graph oriented, Document oriented, Relational, Query, Database, SPARQL, SQL.

Abstract: Database technologies are evaluated in respect to their performance in model extension, data integration, data access, querying and distributed data management. The structure of the data sources is partially unknown. Additional value is gained combination of data sources. Data models for a relational, a document and a graph oriented database are compared showing strengths and weaknesses of each data model.

1 INTRODUCTION

Since relational database systems were introduced they have been successfully used for innumerable applications. So successfully in fact, that for a long time no other paradigms were seriously considered. In 2002 the CAP (Consistency, Availability, Partition tolerance) theorem that was first proposed by Eric Brewer in (Brewer, 2000) was proved (Seth and Nancy, 2002). The theorem states, that in distributed systems one can only guarantee two of the three properties consistency, availability and partition tolerance (Brewer, 2000). Consistency, which is one of the ACID (Atomicity, Consistency, Isolation, Durability) properties on which relational database are founded, may not be as important for some applications as the other two properties are. Due to these circumstances and other specialized applications, a demand for exploring alternative paradigms has risen.

This has led to the development of several NoSQL databases (Levitt, 2010), (Stonebraker, 2010) that have been introduced by open source initiatives or highly successful companies such as Google’s Bigtable (Fay et al., 2006) or Amazon’s SimpleDB (Amazon, n. d.). These were originally developed for use in their own applications, but they have been made publically available in the mean time. When starting a new project today, one can therefore not only choose from different vendors of relational databases, but can also decide to use a totally different database paradigm altogether, depending on the requirements of the project at hand. In this paper, three different paradigms are compared in respect to their performance in model extension, data integration, data access, querying and distributed data management. The results of the experiments and a technology comparison of the reference implementations are presented in this paper.

The goal is to develop a generic common-use database application for a decision support system that allows the user to add data from arbitrary sources and integrate them into a common database. Additionally, the support of geospatial data types and queries was a requirement for this project. Five requirements, regarding the data model have to be taken into consideration:

- Model extension: Adding completely new data with unknown structure.
- Data integration: Integrating new data to an existing database by the user and combining data from various sources.
- Data access: Possibility for automatically creating queries using parameters from a user.
- Querying: Performing complex calculations in the database in a reasonable time.
- Distributed databases: Querying a set of distributed databases to return combined results.

Wendt A., Dönz B., Mantler S., Bruckner D. and Mikula A..
EVALUATION OF DATABASE TECHNOLOGIES FOR USAGE IN DYNAMIC DATA MODELS - A Comparison of Relational, Document Oriented and Graph Oriented Data Models.
DOI: 10.5220/0003714202190224
In *Proceedings of the 4th International Conference on Agents and Artificial Intelligence* (ICAART-2012), pages 219-224
ISBN: 978-989-8425-95-9
Copyright © 2012 SCITEPRESS (Science and Technology Publications, Lda.)

219

2 DATABASE DESCRIPTIONS

For a decision support system, a database-solution with a dynamic data model is needed that allows integrating unknown data sources and dynamic, on-the-fly query generation as well as data mapping for individual views on the data. For this purpose, three different database paradigms together with their respective implementations were explored: A relational database, a document oriented database and a graph oriented RDF database. As input data, tables with flat structures were used, i. e. a large flat file with several columns.

2.1 Relational Database

In the case of a relational database, the data model is defined before importing the data. Extension of tables may be possible after the database generation, but changing relations or keys in tables may make a rebuild necessary. In order to cover as many new data structures as possible, a generalized model has to be used. Further, input data has to be preprocessed and adapted with an ETL (Extraction, Transformation, and Loading) tool (SAS, (n. d.)), (Vassiliadis et al., 2002).

A higher normal form has to be reached, in order to generalize the data model. Approximately 1/3 of the attributes from the flat input tables were normalized into 3NF and 2/3 into 1NF to provide a tradeoff between high generalization and query complexity. The original six independent data tables were finally modeled with 43 independent tables.

An alternative generalized model, which could be created within the relational database, is based on the *Universal Data Model* pattern presented in (Hay, 1995) and (Hay, 1997). By using six tables, it allows the creation of classes and instances within the relational data model. In Figure 1, the basic pattern is shown.

The *Universal Data Model* pattern could be applied in the following way: In the table *Class*, class types are defined. Each class type has the some attributes, which are listed in the table *Attributes* and they are connected through the junction table *Attribute Assignment*. In the table *Instance*, the instance of the class types are saved. The actual corresponding attribute values of that instance are stored in the table *Values*. In the table *Relations*, it is possible to set a relation between subject and object instances with predicates like "superClassOf". Those predicates are defined in the table *Relation Types*. The classes within the *Universal Data Model* can be expanded with any type of attribute. New classes can be added. In this case, each domain (e. g. company data) could be modeled within one complete *Universal Data Model* pattern.

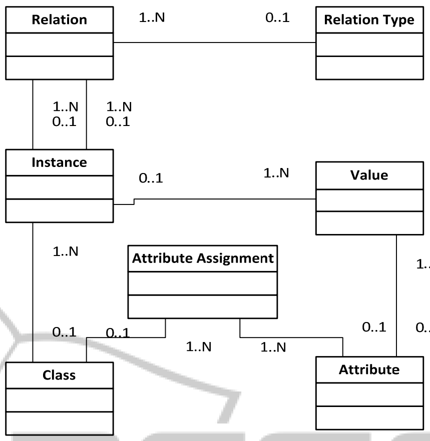

Figure 1: A data model based on the *Universal Data Model* pattern by D. C. Hay (Hay, 1995) modeled in UML notation.

Since geospatial data is a requirement in our project, Postgre (PostgreSQL, n. d.) with the add-on PostGIS (PostGIS, n. d.) was used for the relational database model. PostGIS contains additional functions for processing geographic data e. g. intersection.

Since no out of the box features for distributed databases are available for most relational databases and it would have to be done manually, database distribution was not implemented for PostGIS.

2.2 Document Oriented Database

A document oriented database was investigated for its ability to store and process hierarchical data elements in a single collection, while allowing individual database entries to have arbitrary structures. It would make adding new or appending information to selected existing entries trivial.

MongoDB (MongoDB, (n. d.)) was chosen, because of its wide implementation base, extensive documentation, native support for server-side JavaScript processing for map-reduce patterns, and for its scalability and resilience features.

Since MongoDB natively supports JSON (JavaScript Object Notation) and BSON (Binary JSON) data structures, importing new data is trivial. Existing tools were used to convert geospatial data such as Shape-files. They could then be imported as JSON files, representing each geospatial feature as a separate entity. Data can be imported as a separate collection, or added to existing collections. As the imported data can be arbitrarily structured and the end user is typically only interested in a subset of the attribute hierarchy, a separate metadata collection was used to track the desired attributes. This proved useful in reducing network bandwidth, and considerably simplified client-side processing and display.

Access to the data is provided through an interactive web page that allows the user to formulate geospatial queries towards a selected database collection. Metadata information is used to provide the user with automatically generated, meaningful filter criteria (sliders for numeric values, check boxes for enumerations and string searches for arbitrary text) and to specify the format of the result set. To minimize network latency and round-trip delays, only geospatial queries are processed within the database itself. The result set is then processed by JavaScript code within the web page, allowing near-instant refinement and updates as the user explores varying filter criteria. Unfortunately, support for geospatial queries was very limited at the time of this implementation, only allowing us to filter for a coarse bounding box (min/max tests for latitude and longitude) within the database query itself. Therefore the required precise distance-from-polygon metric is calculated in a separate pass through the result set before returning it to the client. The query language for MongoDB allows queries of moderate complexity but can be enhanced by java script code which is slow but may be executed in parallel following the map-reduce pattern.

Multiple instances can be combined to form database clusters for resilience against server or connection failure. Databases can also be split (sharded), using an arbitrary attribute as a key for distribution over several instances (shards). Each shard then only carries a fragment of the full database and the system automatically attempts to equalize the workload carried by all participating shards. Clustering and sharding are entirely transparent to the client, allowing the database backend to be rearranged and extended as required. MongoDB allows the distribution of large databases over multiple networked instances and queries are processed in parallel. This is desirable for compute intensive queries and aggregations (which can be formulated as Map-Reduce patterns).

2.3 Graph Oriented Database

Graph oriented databases were considered, because of the required functionality to add new information and links between existing data.

Of the several possible database systems available, the RDF (Resource Description Framework) database Allegrograph (Allegro, (n. d.)) was chosen, because it supports several programming interfaces, e.g. Java Jena, Java Sesame, Python, Lisp and others. Also geospatial types and distributed queries are supported. The reference implementation uses the Java Jena interface for importing data and the Java Sesame interface and SPARQL to query the database. Source files contain flat tables of data. The import supports three different operations. The most basic operation is an import of new entities, where each imported row is treated as a new instance of a definable type, the column header is the predicate and the actual value is the object of the triple. The second possibility is to append attributes to an existing entity by letting the user define which columns in the source file must match which columns in the existing database. The third mode is to add links between existing data. The user defines which columns in the source table must match which properties of the subject instance and the object instance. In all three modes the user can define a data type for each column. If possible, the value in the table is then automatically converted to this type and can hence be queried appropriately. If a conversion is not possible, e.g. the source field contains values like “15 million” instead of “15,000,000”, the value is still imported but without a type. These values can then be displayed in tables like all others but are not available for queries.

The user can access the data using an automatically generated query interface and two automatically generated report formats. SPARQL is used as a query language and is generated by setting filters However, in the implementation of Allegrograph, the extensions for geospatial queries are not yet available for SPARQL. Other functions have to be combined to filter the original query result. Two different implementations of user interfaces were implemented. In the first method, the query interface is comprised of a filter table that displays input fields for all predicates. The controls for entering the filter depend on the data type defined during the import, e.g. a textbox for texts or a checkbox for boolean values. The result of the preliminary search is a table containing all entities that match the search criteria. Values in the table that have additional triples associated to it with further information can be clicked on to show a more detailed report. This second report format shows all triples associated with the selected entity, both those where the entity is the subject as well as those where the entity is the object. Here, again, instances with additional information are shown as hyperlinks which link to a detailed report on that entity.

As a second query method, another solution was implemented, where the actual structure of the data is mapped against a manually created ontology. Both the ontology and the mapping can be modified at any time. Queries for this manual ontology are then transformed to fit the actual data structure. In this way, a custom view on the data can be defined and queried independently from the actual data. The model is reusable and by changing the mapping, the same query can be used for several sources, e.g. changed to a new source after importing it to the database.

Allegrograph's query engine allows federating multiple repositories into one virtual repository. This virtual repository can then be used to run queries using the data from all repositories. Compared with the document oriented approach, the federation of the databases is not done transparently but can be controlled by the developer. This allows including or excluding databases at query time depending on the user's privileges, available servers or user selection. In the test-setup, three repositories were hosted on the same server. A sample query returning approx. 2000 Triples from a database containing about 6 million triples took 6.9 seconds when only a single repository was included. The same query with data split into two repositories took 8.9 seconds. When split into three repositories it took 11.7 seconds to execute. The execution time in this setup therefore rises 30% for each additional database. Virtual repositories do not allow write operations for the obvious reason that it would not be clear to which physical repository the data should be written to. So, in our solution the user can decide where new data should be stored when importing a new file.

3 TECHNOLOGY COMPARSION

The described databases are compared regarding their possibilities for model creation and data import, data access and querying and the possibility of using multiple databases.

3.1 Model Creation and Data Import

The used relational database structure would probably be able to integrate most of the inputs in the domains. However, if something had to be changed in the schema, the whole database would have to be rebuilt. Errors in the schema due to insufficient information about the data sources demand a high effort to correct. All input data has to be adapted to the existing data model with an ETL tool or manually. Experience did show that the process of adapting the ETL tool for a complex data model on the first import is fault-prone and demands high effort, e. g. if data could not be completely imported as initially modeled, the model had to be revised. It is hard to exclude all errors and to consider unknown possibilities at the first import attempt. In order to create a stable model that allows integrating future unknown sources, several attempts are needed. However, it is impossible to exclude that new sources will again force a further database rebuild.

The document oriented approach performs well when importing the data since it does not have to be converted in any way. Data is imported as available, but the effort of organizing the data is actually only relocated to the query phase.

Allegrograph has tools to batch-import large files of triples, which can be used if data is already available in this format. When importing data from the original text files and splitting the tables into triples, the implementation of the functionality is not complicated, but it scales very badly. Importing a table with approx. 320k rows and 20 columns took 22 hours to complete. This is a rate of about 80 records (triples) per second, but since the number of records adds up to 6.4 million it takes a very long time to complete.

3.2 Data Access and Querying

In the relational database, with normalized tables, it was possible to create large complex queries. For instance, a query used 17 interconnected tables by including 16 sub-queries, taking in total approximately 2500 words (fields and operators) It took about 90s to execute it on around 60k records on a Intel® Core™ i5 CPU M560 2.67 GHz processor with 4 GB RAM. Materialized tables were used, in order to improve performance, which was executed in about 10s. As expected, in comparison to the other approaches, the relational database is superior in creating and executing complex queries on a large set of data. This is the main advantage of this type of system.

The usage of the *Universal Data Model* in the relational database would have the advantage that it would be completely generalized and adaptable for new data sources. However, the price would be paid with long complex queries, because of the many relations between instances. It would slow down the execution time. Further, no cardinalities can be set between the instances and its values would have to be a string for all values of the table *Value* (see Chapter 2.1).

Allegrograph offers several interfaces for querying the data. Experience did show that queries scale very badly. The execution time is proportional to database size, the structure of the query and the limitation. In our case, all queries must be limited to a fixed number of results; otherwise the query may take very long to complete (more than 4 min). Up to a limitation of around 300 results, the query time of a certain query was approximately the same. Above that, the time rises in discrete steps. The row-limited version of the query is satisfactorily fast for large result sets, since the query terminates after the defined number of results is reached. In this case, a query was tested with different limitations on an Intel® Xeon® 3.0 GHz processor with 4 GB RAM. From the keywords used in the queries, ORDER BY affects the performance significant, while DISTINCT, UNION and LIMIT do not. A remarkable experience was that the order of the statements in SPARQL plays a decisive role in the execution time. If the statement order is constructed optimally, the query executes in about 2 s, else the query does not complete within an acceptable time limit. The following tendencies were observed: Equal subjects should be grouped together if different subjects are used, within a group of subjects, the “simpler” statements should be placed first, statements with UNION should be put at the end of a group of subjects and finally, rdf:label and rdf:type should be placed as the first element of a group of subjects. However, for complex querying like the complex query example in the relational database, this approach is not suitable.

In MongoDB, the querying was similar to the Allegrograph. The idea of using metadata information to automatically generate filters worked well. The fact that joins are not available in MongoDB, poses limits on the complexity of the queries. To overcome this, java script code can be used in more complex situations. This method however slows down query performance in cases where parallel-execution is not possible.

3.3 The Usage of Multiple Databases

Since distributing data over several databases is not the focus of relational database systems and is also not supported natively by the used system (Postgre), this option was not explored.

Allegrograph supports federated virtual repositories that can be set up dynamically at execution time. This feature can be used to define which databases should be used for a query depending on user’s privileges, availability of servers or user-selection. The implementation was straightforward and only geospatial queries had to be adapted to the new circumstances. The execution time of a query is extended by about 30% for each additional repository.

In MongoDB, it was noticed that the apparent communication overhead involved in synchronizing clusters and shard management is substantial, and appeared to outweigh the benefits of parallelized execution for the relatively small database sets (~500.000 entries). Furthermore, recovery from failed or inaccessible cluster members proved to be nontrivial in some cases. Both facts may however be attributed to the use of pre-release development versions (which contained experimental functionality for geospatial queries).

4 CONCLUSIONS

Reference implementations were developed for three different database paradigms including their actual implementation in order to compare their performance in five different disciplines relevant to the project: Model extension, data integration, data access, querying and distributed databases.

It was possible to at least partially fulfill the requirements with all three approaches. It was shown that the results and effort during the implementation vary greatly.

Model extension: Relational databases are not as flexible as the database schema is predefined, which is different in the other two solutions. The amount of work needed for the implementation is also higher, but once the implementation is done it performs well. In the graph oriented approach, it is possible to extend the data model on the fly without affecting the actual data in the database. The same applies to document oriented databases.

Data integration: In order to be able to query data in the relational database, input data has to be preprocessed by an ETL tool, which demands a high initial effort. In the graph and document oriented approach, the possibility of importing data without transformation is given, but it affects the performance if the data structure has to be processed within the query. Therefore, it is strongly recommended to have some sort of ETL process, which is used on the data before importing or restructuring the data in the database. For importing data, the easiest approach however was the document oriented database because it does not need data transformation, performs well and is scalable to large data sources.

Data access: In the relational database, all data was accessible through the use of predefined queries. For simple queries, it would be possible to use graphical query creators. At some level of query complexity, experience has shown that it is easier to write the queries directly in SQL. The graph oriented approach uses predefined queries, either automatically derived from the available predicates or manually created in the dynamic data model. The document oriented approach uses queries that are automatically constructed according to the manually defined metadata.

Querying: The fast querying in the relational database on large amounts of data makes is well suited for the use in large databases, also for complex queries. The most flexible querying solution is the graph oriented approach. But this technology scales very badly and performance is the main issue in the graph-oriented approach. Therefore, only queries with low complexity are suited for use in large databases in this case. Performance is an issue in the document oriented approach, where the map-reduce patterns and sharding offer high performance for suitable queries, but not in the general case. The lack of JOINs reduces flexibility.

Distributed systems: The relational database did not natively support distributed queries and was therefore not tested. The graph and the document-oriented approaches both allow creating federated repositories, which can be queried in the same way as a single query. However, in the graph-oriented approach, a query takes about 30% longer to execute for each additional repository.

Table 1: Evaluation of results.

Requirement	Rel.	Doc.	Graph
Model Extension	-	++	++
Data integration	+	++	+
Data access	++	++	++
Querying	++	-	+
Distributed databases	NA	+	++

In Table 1, the fulfillment of the requirements by the different data models is summarized. In the table, the scale --, -, +, ++ and “NA” (here, not tested) is used.

ACKNOWLEDGEMENTS

This work was supported by the FFG - Österreichische Forschungsförderungsgesellschaft mbH [819065].

REFERENCES

AllegroGraph (n. d.), *AllegroGraph® RDFStore 4.2.1*. Retrieved June 14, 2011, from http://www.franz.com/agraph/allegrograph

Amazon (n. d.), *Amazon SimpleDB*. Retrieved June 14, 2011, from http://aws.amazon.com/de/simpledb/

Brewer E. (2000). *Towards robust distributed systems*. (Invited Talk) Principles of Distributed Computing, Portland, Oregon.

Fay C., Dean J., Ghemawat S., Hsieh W. C., Wallach D. A., Burrows M., Chandra T., Fikes A., Gruber R. E. (2006). Bigtable: A Distributed System for Structured Data. In *OSDI’06: Seventh Symposium on Operating System Design and Implementation*. Seattle, WA, November, 2006.

Hay, D. C. (1995). *Data Model Patterns: Conventions of Thought*, Dorset House, ISBN-13: 978-0932633293, ISBN-10: 0932633293, USA

Hay, D. C. (1997). *Advanced Data Model Patterns*, Essential Strategies, Inc. Retrieved June 20, 2011, from http://www.essentialstrategies.com/publications/modeling/advanceddm.htm

Levitt, N. (2010). Will NoSQL Databases Live Up to Their Promise? in *Computer 43 Issue:2*, doi:10.1109/MC.2010.58

MongoDB (n. d.). *MongoDB*, Retrieved July 22, 2011, from http://www.mongodb.org

PostGIS (n. d.). *PostGIS*, Retrieved June 21, 2011, from http://postgis.refractions.net

PostgreSQL (n. d.). *PostgreSQL*, Retrieved June 21, 2011, from http://www.postgresql.org

SAS (n. d.), *SAS® Enterprise Data Integration Server*. Retrieved July 25, 2011 from http://www.sas.com/software/data-management/entdiserver/index.html

Seth G., Nancy L. (2002). Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. In *ACM SIGACT News, Volume 33 Issue 2, June 2002*, ACM New York, NY, USA

Stonebraker, M. (2010). SQL databases v. NoSQL databases, in *Communications of the ACM Volume 53 Issue 4*, New York, NY, USA

Vassiliadis, P., Simitsis, A., Skiadopoulos, S. (2002). Conceptual Modeling for ETL Processes, in *Proc. 5th International Workshop on Data Warehousing and OLAP (DOLAP 2002)*, McLean, USA