The Multi-model Databases – A Review

Ewa Płuciennik(✉) and Kamil Zgorzałek

Institute of Computer Science, Silesian Technical University,
Akademicka 16, 44-100 Gliwice, Poland
Ewa.Pluciennik@polsl.pl,kamizgo506@student.polsl.pl
http://www.polsl.pl

Abstract. The following paper presents issues considering multi-model databases. A multi-model database can be understood as a database which is capable of storing data in different formats (relations, documents, graphs, objects, etc.) under one management system. This makes it possible to store related data in a most appropriate (dedicated) format as it comes to the structure of data itself and the processing performance. The idea is not new but since its rising in late 1980s it was not successfully and widely put into practice. The realm of storing and retrieving the data was dominated by the relational model. Nowadays this idea becomes again up-to-date because of the growing popularity of NoSQL movement and polyglot persistence. This article attempts to show the state-of-the-art in multi-model databases area and possibilities of this reconditioned idea.

Keywords: Database · Relational · NoSQL · Multi-model · Polyglot persistence

1 Introduction

Every time a commercial application is built it has to persist (store) some data so it can outlast even when application is not operating. So far choice of storage layer was quite simple if it comes to data structure model – a relational database. Of course most of the applications follow the object paradigm and to process relational data they need some additional layer – object-relational mapping, or force the programmer to use native methods to work with the underlying database. For the few years this choice is becoming more complicated. NoSQL movement has arisen. Now application designers should choose not only the technique to work with persistent data but the data model as well.

Right now there are four main categories of NoSQL databases: key-value, column family, document and graph ones. In relational databases elementary unit of data is a tuple (row). Tuples in relational model cannot be nested (it is impossible to create aggregates in sense of one object containing another object – for example customer and his/her orders, or treating complex object as a unit) and have very strict structure (list of fields). Relational data model does not define concept of complex field – relationship between objects are modelled through

S. Kozielski et al. (Eds.): BDAS 2017, CCIS 716, pp. 141–152, 2017.
DOI: 10.1007/978-3-319-58274-0_12

referential integrity. In contrast key-value, column-family and document databases are cut out for this kind of aggregates. In case of document databases these aggregates are also structured (documents/nested documents) and in column-family databases aggregates have two levels (column and row). Graph databases in turn are the best for entities with wide and complicated network of relationships [25]. The last model to mention is the object oriented one. This model follows the object oriented programming paradigm. Object databases emerged in 1990s. Because of strong integration with applications they did not gain much popularity but they are still on the market (e.g. Versant Object Database).

NoSQL Market Forecast 2015–2020 [16] states that key-value store is currently the best solution for scalable, high performance and robust large databases but the biggest obstacle to use NoSQL is transaction consistency and therefore relational databases will not fall into disuse. NoSQL databases should not be treated as replacement for relational databases but rather as its supplement. This report also forecasts graduated convergence of RDBMS and NoSQL into hybrid storage systems.

Having at their disposal such diversity of data models system architects face difficult decision of choosing which model would be the best and in what circumstances, if it comes to performance, security, consistency, etc. The multi-model databases seem good choice if flexibility in data models is our priority. This article attempts to review the state of the art in a multi-model data storage and more detailed comparison of chosen solutions, if it comes to functionality and performance.

2 Multi-model Databases

As was mentioned above the idea of combining different data models into one logical storage is not new. In article [29] from 1997 Garlic system was presented. Its architecture was based on wrappers that encapsulated the data sources (relational, object databases, images archive, etc.) and modelled the data as objects. Each object had an identifier composed of implementation ID (ID – defining an interface and a repository) and a key which identified an object within a given repository. Objects also had methods for searching and data manipulation. Query language was SQL, extended with such method invocation possibility and support for path expressions and nested collections. It is worth pointing out that nowadays such query language exists – JPQL, mentioned in the following part of this chapter. In 1998 patent no. US 5713014 A was published [27]. It defines "Multi-model database management system engine for database having complex data models". This system architecture is composed of physical storage layer, conceptual model layer (based on entity-relationship definition), logical data layer (consists of CODASYL, relational database and object oriented one) and external view layer (API/Language layer: SQL, C++, ODBC). Data unit is defined as a record. Records associations are stored in form of individual pointers or Dynamic Pointer Array (DPA). In article [26] from 2003 authors presented other approach to multi-model databases. The described system utilized different data models at different layers of database architecture: the highest, conceptual layer was based on object oriented and hierarchical, semi-structured models; underlying, logical layer was based on XNF2 (eXtended Non First Normal Form) model; physical layer consisted of RDBMS and MonetDB.

At present, if we look on the NoSQL market, there are quite a few databases which aspire to a multi-model type. ArangoDB stores documents in collections. Although documents can have different structure, documents which have the same attributes share common structure called "shape". This allows for reducing disk and memory space needed for storing data. ArangoDB uses JavaScript to write "actions" which can be compared to database stored procedures. To query the data ArrangoDB offers Query By Example (example document is in JSON[1] format) and its own query language AQL, which supports documents, graphs and joins [17].

Aerospike offers NoSQL key-value database (formerly Citrusleaf) which is optimized for flash memory to achieve real-time speed for managing terabytes of data. In late 2012 Aerospike has acquired AlchemyDB [1] which can be considered as a first (in modern NoSQL movement) database integrating different data models – relational, document and a graph on top of it. The basic unit of storage is a record uniquely identified by a so-called digest within a namespace (a top level container for the data). Aerospike query provides value-based lookup through the use of secondary indexes. Aerospike supports UDF (User-Defined Function) written in C or Lua scripting language [2].

OrientDB supports graph and document models. Documents consisting of key-value pairs (where a value can have simple or complex data type) are stored in classes and clusters (grouping documents together). A cluster or a class can be a vertex of a graph. Each record (e.g. document) has its own identifier (RID) which consists of a cluster ID and a position within a given cluster. RID describes physical position of a piece of data so it allows very fast data localisation. The database uses OrientDB SQL dialect to create, update or search for the data [19].

PostgreSQL from its beginning was considered as an object-relational database. Since version 9.2 it offers relational, object-relational, nested relational, array, key-value (hstore) and document (XML, JSON) store. What is more PostgreSQL offers possibility to nest other databases through Foreign Data Wrappers for relational and NoSQL (CouchDB, MongoDB, Redis, Neo4j) databases [9]. MarkLogic can store data in JSON, XML or RDF triples so it is considered multi-model. It serves as data integrator [15]. Virtuoso in turn offers RDF relational data management [18].

If it comes to relational databases, now they are all object-relational (Oracle, DB2, SQL Server, PostgreSQL) which means that user can define its own data type. What is more these databases also offer XML and JSON storage format. For example Oracle 12c offers JSON storage supported by SQL language extension allowing to use path expression and dedicated SQL functions (JSON_QUERY, JSON_VALUE and JSON_TABLE to name a few) [12].

[1] JSON - JavaScript Object Notation.

If we look at multi-modelness from programmer's point of view, we should consider languages/libraries that give us opportunity to work with different (if it comes to the model) databases from object application with minimum effort (the same interface and even query language). The solution with the widest list of cooperating databases is DataNucleus – implementation of JPA[2] and JDO[3] (which assumes arbitrary data model). At present it offers unified access by means of JPQL (Java Persistence Query Language – SQL-like query language for object data model defined in JPA) for the following databases: RDBMS, HBase, MongoDB, Cassandra, Neo4j, JSON, Amazon S3, GoogleStorage, LDAP, NeoDatis and among others JSON format (this list de facto covers all kinds of NoSQL databases) [8]. However, DataNucleus although quite universal suffers from a few practical disadvantages, for instance not all JPQL constructions work properly for all data stores [28].

Multi-modelness within the meaning of the polyglot persistence is now also appreciated by the big players in the commercial database and cloud processing markets. IBM have created a Next Generation Data Platform Architecture (based on Hadoop) which "combines the use of multiple persistence (storage) techniques to build, store, and manage data applications (apps) in the most efficient way possible" [31]. In this solution organizations that use cloud, dump their data into so-called landing zones without need to understand the schema or setup of the dedicated structures.

If we want to build a system that can work with multi-model data first we need to decide where to place a layer responsible for the data integration. We can put all the responsibility on the application, which then is obligated to dispatch the data into the fully separated so-called persistence lanes. Single persistence lane is responsible for cooperating with one database and contains an adequate data mapper. This architecture is quite simple to implement but (i) data should be divided into databases (and thereby data models) in advance, (ii) cross-lane queries are very hard to implement. Other solution is to complicate architecture but gain better functionality. The options are [24]:

- polyglot mapper placed between application and persistence lanes, which makes cross-lane queries possible and maintains a single object data model; as polyglot mapper DataNucleus, EclipseLink or Hibernate OGM can be used,
- nested database where there is one persistence lane but main database gives a possibility to map/connect other databases (e.g. Postgresql); cross lane queries are possible, but choice of databases is limited,
- omnipotent database which supports many models; main advantage of this approach is a single database with a single maintenance (backup, restore, etc.).

The idea of multi-model database seems to be a perfect solution for implementing polyglot persistence. But there are few challenges. Deciding which approach is the best is not straightforward. First one needs to consider (trade-off) two things:

[2] Java Persistence API.
[3] Java Data Objects.

– what kind of data model is needed – of course more models means more elastic (adaptive) data storage,
– how many different databases covers the requirements – each database needs to be maintained and it requires resources to operate.

Second decision to make is how to query this kind of structure. Of course one, universal language would be the best if it comes to cross-model queries but the question is:

– can already existing language/solution (e.g. JPQL) be used or a new one has to be created – which is not a simple task although is completely possible [17,26,29],
– are cross-model queries required; if not a better solution is to use native query languages for individual databases; if so existing multi-model databases have to be considered.

If it comes to unified query language over different data structures (models) multi-model databases are the best.

3 Comparison of Chosen Multi-model Databases

For more detailed comparison three databases were chosen: ArangoDB and OrientDB (key-value, documents and graphs) and Couchbase (key-value, document). This choice is based on number of models maintained by databases and DB-Engines popularity ranking[4]. The ranking covers all database models. OrientDB is the highest classified open source multi-model database (three models, position no. 45[5]), ArangoDB is classified on position no. 76 and constantly moving up (see footnote 5). Couchbase was chosen as two-model database (since version 2.0 it covers document and key-value models [30]) with position no. 23 (see footnote 5) – highest out of two-model databases.

3.1 Data Model

First data models will be described. In ArangoDB the basic data structure is a document which can consists of any number of attributes with values (of simple or complex type). Each document has three special attributes [3]:

– *_id* – unique, unequivocal identifier of a document within the confines of the database,
– *_key* – identifies a document within the confines of a collection,
– *_rev* – document's version.

[4] http://db-engines.com/en/ranking.
[5] As of 30 January 2017.

Documents are organized into collections which can be compared to tables in a relational database. Documents are schema-less so they can differ in terms of number and types of attributes. There are two kinds of collections: document (nodes) collection and edges collection. Edge is a special kind of document which has two peculiar attributes: *_from* and *_to* used to denote relation between documents. So documents are organized into a directed graph.

In OrientDB basic (the smallest) data unit is a record. There are four possible record types: byte record (BLOB), document, node (vertex) and edge. Document is the most flexible type of record because it can be schema-less. Alike ArangoDB, in OrientDB documents have special (obligatory) attributes [4]:

– *@class* – defines a class of a document (classes define types of records; they are schema-less, schema-full or mixed one),
– *@rid* – an automatically generated identifier for documents within the confines of the database,
– *@version* – version number.

Records are grouped into clusters. By default there is one cluster (physical or in-memory) per class. One class can be divided into more clusters (shards) to enable e.g. parallel query execution. Documents can be related in form of strong (embedded) or weak relationships. Nodes and edges have a form of documents but edges can be stored in a lightweight form. A lightweight edge is embedded into vertex record which can improve performance, but this type of edge can not have any additional properties [13].

Couchbase Server operates on two data models: document and key-value. Because these two models are very similar one can doubt if Couchbase can be perceived as multi-model. Nonetheless Couchbase uses key-value for searching to improve performance [22].

Table 1 contains summary of data model characteristics of covered databases. There are many similarities especially if it comes to JSON data format or indexing on key fields. Differences are more subtle and involve above all, data organization.

Table 1. Comparison of described databases

Characteristic	ArangoDB	OrientDB	Couchbase server
Data models	Key-value, document, graph		Key-value, document
Key-value model implementation	Automatically created index on key field		
Document model implementation	JSON data format		
Graph model implementation	Special edge document with attributes *_from* and *_to*		N/A
Data organization	Collections	Classes	Buckets
Special attributes	Identifier, version		Identifier

3.2 Query Language

This section presents the chosen databases query languages in comparison to SQL which is an unattainable ideal for many database designers.

ArangoDB creators are among them. AQL has a syntax very similar to SQL and it is declarative. The main difference is that it operates on collections and does not have data definition part (DDL).

OrientDB uses SQL operating on classes with some graph extensions added (SELECT phrase can be replaced with TRAVERSE phrase [21]). The main difference is that instead of joins, classes' associations are used.

Couchbase uses its own query language N1QL which is SQL for JSON [7]. There are two possible kinds of queries: indices management (DDL) and CRUD (DML).

Table 2 presents example basic CRUD queries for described databases. All languages are very similar to SQL if it comes to basic phrases (INSERT, SELECT, UPDATE, DELETE), all allow full CRUD functionality. There are of course some differences, especially in AQL where special loop and filter constructions are used for performing operation on elements of collection (*cars*).

Table 2. CRUD operation syntax comparison

AQL	OrientDB SQL	N1QL
Create data		
INSERT {make: "MakeA", model: "ModelB"} IN cars	INSERT INTO cars (make, model) VALUES ('MakeA', 'ModelB')	INSERT INTO cars (KEY, VALUE) VALUES (UUID(), {"make":"MakeA", "model":"ModelB"})
Retrieve data		
FOR t IN cars FILTER t.make == "MakeA" RETURN t	SELECT FROM cars WHERE make LIKE 'MakeA'	SELECT * FROM cars WHERE make = 'MakeA'
Update data		
UPDATE "1" WITH price: "3000" IN cars	UPDATE cars SET price = '3000' WHERE @rid = '1'	UPDATE cars USE KEYS "1" SET price = "3000"
Delete data		
FOR t IN cars FILTER t.make = 'MakeA' REMOVE t	DELETE FROM cars WHERE make = 'MakeA'	DELETE FROM cars t WHERE t.make = "MakeA"

3.3 Indices and Transaction

Indices and transactions are the elements which are very important in databases.

ArangoDB automatically creates primary hash index for each collection and for graph edges. This index is based on *_key* field and it is not sorted so it cannot be used for range selection. Except for primary index, user can create his/her own indices of types: hash, skiplist, geo, fulltext, sparse [11].

OrientDB uses indices in a similar way as in the relational databases. User has four types of indices to choose from: SB-Tree (based on B-tree with some optimization added), hash, auto-sharding (for distributed systems) and Lucene (fulltext and spatial) [10].

Couchbase offers two kinds of indices: global and local. Global indices minimize network processing during interactive query processing. There are four types of indices: Global Secondary Index, MapReduce views, spatial views and full text index [6].

Since version 1.3 ArangoDB offers transactions configurable by the user with full ACID at repeatable reads isolation level [14]. OrientDB also provides full ACID transaction with two isolation levels: read committed and repeatable reads [20]. Couchbase does not support transactions [23].

4 Performance Tests

Multi-model databases potentially have wider set of applicable use cases since they cover more than one data model under one management system. The question is if the diversity of maintained data models does not diminish the performance. One of the ArangoDB creators (Claus Weineberger) has worked out and conducted tests embracing ArangoDB, OrientDB, Neo4j, MongoDB and Postgres databases [5]. The main goal was to prove that native multi-model databases can successfully compete with one-model databases. Tests were performed on Pocked social network[6] with 1 632 803 nodes and 30 622 564 edges. Tests were divided in to two groups: read/write/ad-hoc aggregation queries and queries typical for a graph (shortest path and nearest neighbour). The tests results are very promising for multi-model databases – especially ArangoDB. For single read ArangoDB is slightly slower than MongoDB whereas OrientDB is 50% slower than ArangoDB. For single write both multi-model databases were faster than MongoDB. As for aggregation ArangoDB was 2.5 times faster than MongoDB and 18 times faster than OrientDB. Full tests results can be found at [5]. Of course these tests are vendor tests but all results and code (written in JavaScript) was published for public use and author explicitly states that ArangoDB currently works best when the data fits in the memory.

In our tests we have decided to concentrate on CRUD operation from Java developer perspective. We did not measure database performance in cluster or for large datasets. Because all three databases share ability to operate on document model, we wanted to answer the question if the multi-modelness diminishes performance of simple document operations (as a point of reference MongoDB was used). If so, to what degree? In tests we have measured number of operations per second. As for operations we have a single read, a single write, a single

[6] https://snap.stanford.edu/data/soc-pokec.html.

update, a single delete and an aggregation (number of documents satisfying some condition). All tests were performed in five series with 100 repeats (for example 500 document writes). No special database tuning was conducted. All databases have operated on default keys and indices. The dataset consisted of generated personal data (name, surname, profession, weight, etc.) with different age values.

Create operations were conducted by saving document using special methods offered by the databases. For retrieving filtering query was used (filtering about 1% of documents – 50 year old persons). Query was written in query language appropriate for a particular database:

– ArangoDB: *FOR t IN testCollection FILTER t.age == 50 RETURN t*,
– OrientDB: *SELECT FROM testDocument WHERE age = 50*,
– Couchbase: *SELECT * FROM default WHERE age == 50.*

Update operation was conducted by adding one field to each document using dedicated database methods. Similarly, for delete test each document was removed from a database. Aggregation test relied on counting documents representing persons 50 and more years old (about 50% of the dataset) using query:

– ArangoDB: *FOR t IN testCollection FILTER t.age >= @age RETURN COUNT(t)*,
– OrientDB: *SELECT COUNT(*) FROM testDocument WHERE key >= 50*,
– Couchbase: *SELECT COUNT(*) FROM default WHERE age >= 50.*

Table 3 summaries the test results – number of operation per second is presented.

Table 3. Test results – number of operation per second

Operation	ArangoDB	OrientDB	Couchbase	MongoDB
Create data	377	372	423	573
Retrieve data	415	235	261	492
Update data	639	581	681	695
Delete data	865	818	1792	956
Aggregation	421	175	303	452

For greater readability the results are presented in a form of bar chart in Fig. 1. Results for MongoDB are presented as reference line. Bars that are fitting under that line denote less performance than the reference database.

First, most conspicuous observation is that all multi-model databases CRUD operation were slower than in MongoDB except one case – document deletion in Couchbase. Documents in Couchbase are organized into buckets. A bucket is not only a logical unit but also physical storage. Remove method in Couchbase is performed not directly on document but throughout a bucket and probably is natively optimized. Second conclusion is that the slowest database was OrientDB and ArangoDB has results that are most evenly distributed and most similar to

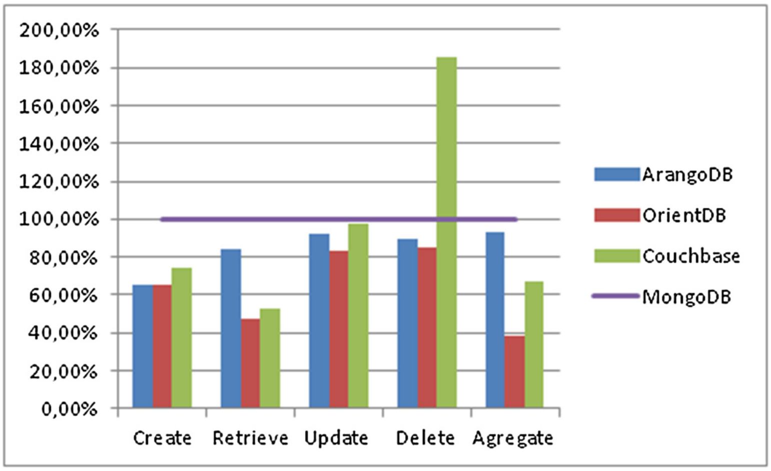

Fig. 1. CRUD test results - performance in relation to MongoDB

MongoDB. These tests are of course in their initial stage and need to be extended with additional scenarios because the subject is very interesting and very up-to-date.

5 Conclusion

There is no standard definition what multi-model means – support of more than one data model? It is obvious that this simple definition leads to one conclusion. Each database which supports UDT (User-Defined Type) or JSON is multi-model. Users can define types suited to their needs. In this case each modern database is multi-model, beginning from object-relational (Oracle, SQL Server, DB2) to ArangoDB for example. Perhaps we need stricter definition of multi-modelness of n-degree where n is a number of data models supported by a database out of the box. Right now it seems that maximum value of n is six.

If it comes to multi-model and adaptive database system, its designing and creating is complicated but possible and it already happens in commercial area of data storage market. Existing multi-model databases are promising technology to use when we need to create this kind of database system. It seems that it is worth to use them, provided they assure support for all or majority of the needed models. Using this kind of a database lighten resource and maintenance requirements and provide query language capable of operating on different data structures (for example documents and graphs). Multi-model databases are not as popular as other NoSQL solutions, but it seems that it will change although it is hard to anticipate in what degree. For now it can be stated that these databases are slightly slower that one-model solution – multi-model handling comes with a price. Is this price worth paying? To answer this question we certainly need more tests and to observe multi-model databases positions in popularity rankings.

Acknowledgement. This work was funded by statutory research of Faculty of Automatic Control, Electronics and Computer Science of Silesian University of Technology.

References

1. Aerospike Acquires AlchemyDB NewSQL Database. http://www.aerospike.com/press-releases/aerospike-acquires-alchemydb-newsql-database-to-build-on-predictable-speed-and-web-scale-data-management-of-aerospike-real-time-nosql-database-2/. Accessed 19 Nov 2016
2. Aerospike Documentation. http://www.aerospike.com/docs/. Accessed 19 Nov 2016
3. ArangoDB Data models and modelling. https://docs.arangodb.com/3.0/Manual/DataModeling/index.html. Accessed 19 Nov 2016
4. Basic Concepts OrientDB. http://orientdb.com/docs/2.0/orientdb.wiki/Concepts.html. Accessed 19 Nov 2016
5. Benchmark: PostgreSQL, MongoDB, Neo4j, OrientDB and ArangoDB. https://www.arangodb.com/2015/10/benchmark-postgresql-mongodb-arangodb/. Accessed 19 Nov 2016
6. Couchbase Server Indexing. http://developer.couchbase.com/documentation/server/4.5/indexes/indexing-overview.html. Accessed 19 Nov 2016
7. Database Querying with N1QL. http://www.couchbase.com/n1ql. Accessed 19 Nov 2016
8. DataNucleus AccessPlatform 5.0 Documentation. http://www.datanucleus.org/products/accessplatform_5_0/index.html. Accessed 19 Nov 2016
9. Foreign data wrappers. https://wiki.postgresql.org/wiki/Foreign_data_wrappers. Accessed 19 Nov 2016
10. Indexing - OrientDB Manual. http://orientdb.com/docs/last/Indexes.html. Accessed 19 Nov 2016
11. Indexing ArangoDB. https://docs.arangodb.com/3.0/Manual/Indexing/index.html. Accessed 19 Nov 2016
12. JSON in Oracle Database.https://docs.oracle.com/database/121/ADXDB/json.htm#ADXDB6246. Accessed 19 Nov 2016
13. Lightweight Edges - OrientDB. http://orientdb.com/docs/last/Lightweight-Edges.html. Accessed 19 Nov 2016
14. Locking and Isolation ArangoDB. https://docs.arangodb.com/3.1/Manual/Transactions/LockingAndIsolation.html. Accessed 19 Nov 2016
15. MarkLogic Semantics. http://www.marklogic.com/wp-content/uploads/2016/09/Semantics-Datasheet.pdf. Accessed 19 Nov 2016
16. NoSQL Market Forecast 2015–2020. http://www.marketresearchmedia.com/?p=568. Accessed 19 Nov 2016
17. On multi-model databases. Interview with Martin Schönert and Frank Celler. http://www.odbms.org/blog/2013/10/on-multi-model-databases-interview-with-martin-schonert-and-frank-celler/. Accessed 19 Nov 2016
18. Openlink Virtuoso Home. https://virtuoso.openlinksw.com/. Accessed 19 Nov 2016

19. OrientDB Manual - version 2.0, Document and Graph Models. http://www.orientechnologies.com/docs/last/orientdb.wiki/Tutorial-Document-and-graph-model.html. Accessed 19 Nov 2016
20. Transactions - OrientDB Manual. http://orientdb.com/docs/last/Transactions.html. Accessed 19 Nov 2016
21. Traverse - OrientDB Manual. http://orientdb.com/docs/last/SQL-Traverse.html. Accessed 19 Nov 2016
22. Why Couchbase? http://developer.couchbase.com/documentation/server/current/introduction/intro.html. Accessed 19 Nov 2016
23. Couchbase: View and query examples. http://developer.couchbase.com/documentation/server/4.1/developer-guide/views-query-sample.html. Accessed 19 Nov 2016
24. Engelschall, R.S.: Polyglot Persistence Boon and Bane for Software Architects. https://docs.arangodb.com/3.0/Manual/DataModeling/index.html. Accessed 19 Nov 2016
25. Fowler, M., Sadalage, P.: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley, Upper Saddle River (2012)
26. van Keulen, M., Vonk, J., de Vries, A.P., Flokstra, J., Blok, H.E.: Moa and the multi-model architecture: a new perspective on NF^2. In: Mařík, V., Retschitzegger, W., Štěpánková, O. (eds.) DEXA 2003. LNCS, vol. 2736, pp. 67–76. Springer, Heidelberg (2003). doi:10.1007/978-3-540-45227-0_8
27. NoSQL Market Forecast 2015–2020: Multi-model database management system engine for database having complex data models US 5713014 A. http://www.google.com/patents/US5713014. Accessed 28 Nov 2016
28. Płuciennik-Psota, E.: Object (not only) relational interfaces survey. Stud. Inform. **34**, 301–310 (2012)
29. Roth, M., Schwarz, P.: Don't scrap it, wrap it! a wrapper architecture for legacy data sources. In: VLDB 1997 Proceedings of the 23rd International Conference on Very Large Data Bases, pp. 266–275 (1997)
30. Wiederhold, B.: Key-value or document database? Couchbase 2.0 bridges the gap. https://blog.couchbase.com/key-value-or-document-database-couchbase-2-dot-0-bridges-gap. Accessed 30 Jan 2017
31. Zikopoulos, P., deRoos, D., Bienko, C., Buglio, R., Andrews, M.: Big Data Beyond the Hype. A Guide to Conversation for Today's Data Center. McGraw Hill Education, New York (2014)