Multi-model Databases: A New Journey to Handle the Variety of Data

JIAHENG LU, Department of Computer Science, University of Helsinki
IRENA HOLUBOVÁ, Department of Software Engineering, Charles University, Prague

The variety of data is one of the most challenging issues for the research and practice in data management systems. The data are naturally organized in different formats and models, including structured data, semi-structured data and unstructured data. In this survey, we introduce the area of multi-model DBMSs which build a single database platform to manage multi-model data. Even though multi-model databases are a newly emerging area, in recent years we have witnessed many database systems to embrace this category. We provide a general classification and multi-dimensional comparisons for the most popular multi-model databases. This comprehensive introduction on existing approaches and open problems, from the technique and application perspective, make this survey useful for motivating new multi-model database approaches, as well as serving as a technical reference for developing multi-model database applications.

CCS Concepts:•**Information systems** → **Database design and models; Data model extensions;** *Semi-structured data; Database query processing; Query languages for non-relational engines;* Extraction, transformation and loading; Object-relational mapping facilities;

Additional Key Words and Phrases: Big Data management, multi-model databases, NoSQL database management systems.

ACM Reference Format:
Jiaheng Lu and Irena Holubová, 2019. Multi-model Databases: A New Journey to Handle the Variety of Data. *ACM CSUR* 0, 0, Article 0 (2019), 38 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

As data with different types and formats are crucial for the optimal business decisions, we observe the substantial increase of demands to analyze and manipulate multi-model data, including structured, semi-structured and unstructured data. In particular, structured data includes relational, key/value, and graph data. Semi-structured data commonly refer to XML and JSON documents. Unstructured data are typically text files, containing dates, numbers and facts.

We illustrate the challenge of the variety of data with three examples as follows. First, let us consider *customer-360-view* [Kotorov 2003] to enable a holistic analysis on customer behaviors. This application demands to analyze the information from different data sources, such as product catalog (XML or JSON documents), customer social networks (graph data), social media (unstructured data) and relational tables of customer shopping records. Second, in the context of healthcare, high volumes of data are generated by multiple data sources [Aboudi and Benhlima 2018], including electrical health records (relational data), treatment plans and lab test reports (unstructured data), and health condition parameters for real-time patient health monitoring (key/value data). Finally, an oil & Gas company [Hems et al. 2013] might generate over

Author's addresses: Jiaheng Lu, Department of Computer Science, University of Helsinki, Gustaf Hällströmin katu 2b, FI-00014 Finland; I. Holubová, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Malostranské nám. 25, 118 00 Praha 1, Czech Republic.

DOI: http://dx.doi.org/10.1145/0000000.0000000

1.5 TB of diverse data every day [Baaziz and Quoniam 2014]. Those data come from diverse resources, such as sensors, GPS, and other instruments, and consequently have heterogeneous formats. Therefore, the above three examples demonstrate the emerging challenges to manipulate and analyze multi-model data in complex application scenarios.

We now exemplify the challenge of multi-model data management with a concrete small example from E-commerce in Figure 1, which contains customers, social network and orders information with four distinct data models. Customers information are stored in a relational table – their ID, name, and credit limits. Graph data bear information about mutual relationships between the customers, i.e. who knows whom. In JSON documents each order has an ID and a sequence of ordered items, each of which includes product number, name, and price. The fourth type of data, key/value pairs, bears a relationship between customers (their IDs) and orders (their IDs).

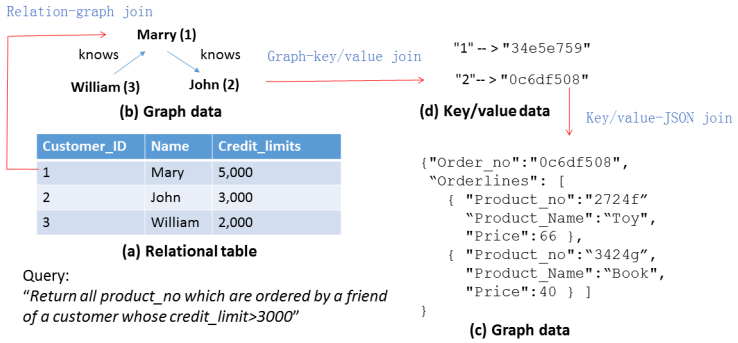


Fig. 1. Motivation example for multi-model data

In these multi-model data one may be interested in a recommendation query, which returns "*all product numbers ordered by a friend of a customer whose credit limit is greater than 3000*". Such a query can be evaluated using various approaches depending on the selected storage strategy. Either the data is stored in different database management systems (DBMSs) corresponding to the four data models, or the four types of data are transformed into a single format, e.g., the relational format, and stored in a relational database system. However, in the former case we need to solve the problems of (1) the installation and administration of multiple distinct systems and (2) joining data stored at distinct places. In the latter case, even though storing hierarchical or graph data in a relational DBMS is feasible, the efficiency of query evaluation is a bottleneck due to the inherent structural differences from flat relations.

A third option for the above task is to employ a single multi-model DBMS to exploit advantages of both the previous solutions: (1) The data is stored in the way optimal for the particular models and (2) only a single DBMS is employed to conveniently query across all the models. In Figure 2, we show two sample queries to return the requested result for two existing multi-model databases – ArangoDB [ArangoDB 2016] and OrientDB [OrientDB 2016] respectively.[1] A single data platform for multi-model data is beneficial to users by providing not only a unified query interface, but a single database platform to simplify query operations, reduce integration issues, and eliminate migration problems.

[1]We will introduce these two systems in a more detail (together with other related representatives) in Section 4.

ArangoDB:

```
LET customerIDs = (FOR customer IN customers
                   FILTER customer.credit_limit > 3000
                   RETURN customer.id)
LET friendIDs = (FOR customerID IN customerIDs
                 FOR friend IN 1..1 OUTBOUND customerID knows
                 RETURN friend.id)
FOR friend IN friendIDs
FOR order in 1..1 OUTBOUND friend customer2Order
RETURN order.orderlines[*].product_no
```

OrientDB:

```
select expand(out("knows").orders.orderlines.product_no)
from customers
where credit_limit > 3000
```

Result: `["2724f", "3424g"]`

Fig. 2. Sample queries for multi-model data in Figure 1

In general, there are two existing approaches to manipulate and query multi-model data: (1) *polyglot persistence* and (2) *multi-model databases* [Lu and Holubová 2017; Lu et al. 2018a]. First, the history of polyglot persistence can be traced back to multi-databases [Smith et al. 1981] and federation databases [Hammer and McLeod 1979], which were intensively studied during the 1980s. Their main strategy is to leverage different databases to store different models of data and then develop a mediator to integrate them together to answer queries. Recently, some research prototypes are developed on polyglot persistence platform. For example, DBMS+ [Lim et al. 2013] targets at embracing several processing and database platforms with a unified declarative processing. BigDAWG [Elmore et al. 2015] provides an architecture that supports for location transparency and a middleware that provides a uniform multi-island interface to run users' queries with three different integrated systems: PostgreSQL, SciDB and Accumulo.

The second kind of systems is to build one single database to manage different data models with a fully integrated backend to handle the system demands for performance, scalability, and fault tolerance [Lu et al. 2018b]. A framework of a fully integrated single management system can be traced back to the concept of ORDBMS (i.e., Object-Relational DataBase Management Systems), which borrow and adapt the object-oriented programming model into the relational databases. An ORDBMS can store and process various formats of data such as relational, text, XML, spatial and object by leveraging domain specific functions. But the salient difference between the ORDBMS and multi-model databases is that, in an ORDBMS framework, only the relational model is the first-class citizen, meaning all other models are developed on top of relational technology. But in multi-model databases, there is no indispensable model and every model is equally important. Compared with the first system of polyglot persistence, the second one manages multiple models with an integrated backend which can satisfy the growing requirements for scalability, high performance and fault tolerance. In this survey, we will focus on the second approach by building a single multi-model database. As for the first approach, interesting readers may refer to Appendix C.

Main Contributions. This survey reviews the representatives of multi-model databases and summarizes their major features and techniques. The comprehensive review and analysis make this article useful for motivating new multi-model process-

ing techniques, developing real-world multi-model database applications, as well as serving as a technique reference for selecting and comparing the existing multi-model database products. In particular, the main contributions are summed up as follows:

(1) We introduce the area of multi-model DBMSs and their relation to other database technologies. We provide historical background as well as a general classification of related approaches.
(2) We compare the existing multi-model DBMSs from various viewpoints and using distinct criteria. We also provide the timeline depicting their evolution and reflecting the historical needs for such systems.
(3) We provide a detailed overview and description of key features of existing representatives of multi-model DBMSs. Using examples we demonstrate their basic capabilities and differences.
(4) We discuss the remaining open problems and demonstrate that multi-model databases form a challenging research area where the solutions will find exploitation in a broad range of real-world use cases.

Related Work. Currently there exist several surveys dealing with efficient management and/or processing of Big Data. Paper [Sakr et al. 2015] describes existing Big Data processing systems, namely big SQL systems, graph management systems, and stream processing systems. Papers [Sakr et al. 2013; Li et al. 2014] focus on a detailed study of the MapReduce programming framework and approaches built on top of it. Considering Big Data DBMSs, there exist tens of papers which provide a general description and classification of NoSQL databases, experimental evaluations, comparative studies, and/or benchmarks of selected representatives of various types of NoSQL systems, eventually involving also relational DBMSs. For more specific studies, papers [Elshawi et al. 2015; Angles et al. 2017] survey graph DBMSs and their query languages. Paper [Cattell 2011] provides an overview and comparison of key/value, document, extensible record (i.e. column) and scalable relational DBMSs. There exists also a web page[2] focusing on ranking of various types of DBMSs, including NoSQL, which ranks database management systems according to their popularity which is evaluated[3] on the basis of number of mentions of the system on websites, frequency of technical discussions about the system etc. Recently, a general survey and a comparison of three multi-model databases has been published in [Płuciennik and Zgorzałek 2017]. However, to the best of our knowledge, there exists no paper solely dealing with multi-model databases in the extent and depth comparable to this survey.

It is worthy to mention the difference between multi-**modal** databases and multi-**model** databases. The former means the multi-media databases where the types of data may include speech, images, videos, handwritten text and fingerprints. But the latter stands for a system to manage data with different models such as relational, tree, graph and object models. The scope of this survey restricts to the latter one, i.e. multi-model databases.

Outline. The rest of this article is organized as follows: Section 2 presents a brief introduction of four common data models. Section 3 deals with classification and comparison of existing multi-model DBMSs from the view of both history and contemporary features. In Section 4 we provide a detailed description of particular multi-model systems. In Section 5 we discuss challenges and open problems, and finally we conclude this article in Section 6.

[2]http://db-engines.com/en/ranking
[3]http://db-engines.com/en/ranking_definition

2. PRELIMINARIES ON DATA MODELS

In this section, we briefly review the four data models which are supported by most of multi-model databases, including relational, semi-structured, key/value and graph model.

2.1. Relational Model

Relational model is based on the mathematical term *relation*, i.e. a subset of Cartesian product. The data are logically represented as tuples forming relations. Each record in a relation is uniquely identified by a *key*. The relational data can be both defined and queried using a declarative approach, which is currently mainly represented by the Structured Query Language (SQL) [ISO 2008]. The relational DBMS then ensures both storing and retrieving of the data. Examples of relational databases include financial and banking systems, computerized medical records, and on-line shopping.

2.2. Semi-structured Model for XML and JSON Documents

The semi-structured model is based on the idea of representing the data without explicit and separate definition of its schema. Instead, the particular pieces of information are interleaved with structural/semantic tags that define their structure, nesting etc. Such a representation enables more flexible processing and exchanging of the data. In the following, we introduce two representative semi-structured data types: XML and JSON.

The Extensible Markup Language (XML) [W3C 2008] is a human-readable and machine-readable markup language. Its format is textual and based on the exploitation of Unicode to enable the support of various languages. The data are expressed using elements delimited by tags which can contain simple text, subelements or their combination. Additional information can be stored in attributes of an element. XML is widely used for the representation of arbitrary data structures. such as those used in web services. The JSON (JavaScript Object Notation) [Ecma International 2013] is a human-readable open-standard format. It is based on the idea of an arbitrary combination of three basic data types used in most programming languages – key/value pairs, arrays and objects.

2.3. Key/value Model

The key/value model is definitely the simplest data model used in NoSQL databases. It corresponds to associative arrays, dictionaries, or hashes. Each record in the key/value model consists of an arbitrary value and its unique key which enables to store, retrieve, or modify the value. The simplicity of the model and respective operations enable efficient data processing (at the cost of non existence of a powerful query language).

2.4. Graph Model

The graph data model is based on the mathematical definition of a *graph*, i.e., a set of vertices (nodes) V and edges E corresponding to pairs of vertices from V. In the world of Big Data there exists a special type of databases, called *graph databases*, devoted for efficient storage and management of the graph data. We can further distinguish two types of graph databases which correspond to two main types of graph data use cases and differ in the respective usage [Sakr and Pardede 2011]. *Transactional databases* work with a large set of smaller graphs, such as a set of linguistic trees or chemical compounds. The respective operations usually search for supergraphs, subgraphs, or similar graphs. *Non-transactional databases* conversely target a single large graph (e.g., a social network), possibly having several components. The respective operations correspond to searching a (shortest) path, communities (i.e., subgraphs with specific

features) etc.

Interested readers may refer to other excellent surveys, such as [Angles and Gutiérrez 2008; Davoudian et al. 2018] for rigorous and comprehensive definitions on different data models in databases.

3. TAXONOMY AND COMPARATIVE STUDY

Starting with a brief history of the multi-model databases, in this section we provide a comparative study of existing multi-model DBMSs.

3.1. A brief History

In the mid-1960s, data was stored in file systems. Then, in the early 1980s, relational databases began to gain commercial traction for enterprise data management mainly owing to Edgar F. Codds relational model (which was described already in 1969). Later, in 1990s, enterprises identified a need to process non-relational data in many applications and thus a number of databases were developed to focus on a special type of applications, e.g. object-oriented databases, XML databases, spatial databases, or RDF databases. Today, the evolution continues to manage Big Data and cloud applications, i.e. to write, read, and distribute large scale of different types of data everywhere. In the early 2010s, a number of NoSQL databases are created, such as Cassandra, HBase, CouchDB, OrientDB, Neo4j, Asterix, ArangoDB, or MongoDB, to name a few.

By looking back at the history of databases above, one can identify a trend that more and more types of data are stored and processed in databases. Therefore, this calls for developing a multi-model database system to have the ability to manage different kinds of data simultaneously. We can observe a recent trend among NoSQL databases in moving towards multi-model databases. There are many databases which claim to be multi-model databases currently. However, the level of support for multi-model data varies greatly, with different ability to query across distinct models, to index the internal structure of a model, and to optimize query plans across models, which will be described in details in the following sections.

3.2. Taxonomy of multi-model databases

In this subsection, we discuss the taxonomy and the comparisons for diverse multi-model database systems. In particular, the current multi-model databases can be classified according to various criteria. One classification on the basis of their original (or core) data model is provided in Table I. As we can see, the table involves relational databases, all four types of NoSQL databases, and other types, such as object databases. We will use this basic classification in Section 4 where we describe particular DBMSs in a more detail.[4] In this section we focus on various other types of classification and comparative viewpoints.

First of all, in Fig. 3 we provide a timeline which depicts the journey where a system became multi-model, i.e. either when its original data format was extended towards additional ones, or when it was first released directly as a multi-model DBMS.

The evolution of the systems naturally corresponds to the growing popularity of particular formats. For example, we can see that the first main wave of multi-model databases has appeared soon after the beginning of the new millennium with the emergence of XML data. The key relational DBMSs were extended towards XML, usually via the SQL/XML standard or its variation, and thus they were transformed to so-called XML-enabled databases. The second wave can be observed after 2010 with the

[4] In Appendix A we also provide an overview of the top 5 DBMSs in the respective classes.

Table I. Classification of multi-model databases

Original Type	Representatives
Relational	PostgreSQL, SQL Server, IBM DB2, Oracle DB, Oracle MySQL, Sinew
Column	Cassandra, CrateDB, DynamoDB, HPE Vertica
Key/value	Riak, c-treeACE, Oracle NoSQL DB
Document	ArangoDB, Couchbase, MongoDB, Cosmos DB, MarkLogic
Graph	OrientDB
Object	Caché
Other	Not yet multi-model – NuoDB, Redis, Aerospike Multi-use-case – SAP HANA DB, Octopus DB

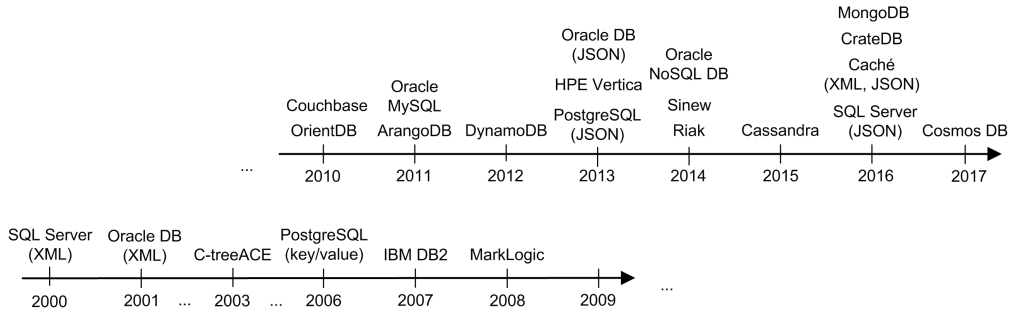


Fig. 3. Timeline of the support of multiple-models

arrival of the era of Big Data. The XML-enabled databases were often extended towards the JSON format and there have also appeared representatives of other types of DBMSs combining their original data format with other formats.

In Table II we classify the systems according to the strategy used to extend the original model to other models or to combine multiple models. We distinguish four types of approaches:

(1) adoption of a completely new storage strategy suitable for the new data model(s),
(2) extension of the original storage strategy for the purpose of the new data model(s),
(3) creating of a new interface for the original storage strategy, and
(4) no change in the original storage strategy.

Note that in some cases the approach can be clearly categorized, whereas mainly in case of the first and second group it is sometimes hard to decide where the particular DBMS belongs.

The typical representative of the first group are XML-enabled databases which use a native XML approach for their efficient storing and querying. An example of the second group is a document database *ArangoDB* where special edge collections are used to bear information about edges in a graph. Similarly *MongoDB* uses for this purpose references amongst documents. An example of the third group is *Sinew* which builds a new layer above traditional relational storage strategy. Another example can be *MarkLogic* which stores JSON documents in the same way as XML documents, but adds the support for Javascript to work with the data. And, it also supports processing of JSON data using XQuery [W3C 2015b]. Examples of the fourth group are all database systems which naturally involve storage and processing of data formats simpler than the original one. Hence, for example, all document databases can also be considered as key/value and column stores. Or, all column stores can be considered as key/value stores.

Table II. A strategy for extension towards multiple models

Approach	DBMS	Type
New storage strategy	PostgreSQL	relational
	SQL server	relational
	IBM DB2	relational
	Oracle DB	relational
	Cassandra	column
	CrateDB	column
	DynamoDB	column
	Riak	key/value
	Cosmos DB	document
Extension of the original storage strategy	MySQL	relational
	HPE Vertica	column
	ArangoDB	document
	MongoDB	document
	OrientDB	graph
	Caché	object
New interface for the original storage strategy	Sinew	relational
	c-treeACE	key/value
	Oracle NoSQL Database	key/value
	Couchbase	document
	MarkLogic	document

Next, in Table III we provide a matrix which visualizes the data models supported in the particular multi-model DBMSs. Note that in case of the document model we consider the most common JSON format or its variants, whereas there is a separate column for the XML format which has specific features and history of support. For the same reason we distinguish the general graph model and RDF [W3C 2014] data format. We also devote a separate column to object-like models (i.e., except for the classical object model we add here also distinct user-defined types and nested structures). The final column shows the popularity of different system on Nov. 2018 based on the statistics from the DB-Engines Ranking[5].

In case of the RDF model we have to point out its specific relation to this survey. Currently there exists a number of RDF triple stores. These systems are usually implemented as an extension of an existing DBMS, either as a part of it or as a module built on top of it. For example, a relational DBMS can be used as a back-end which stores RDF triples, not knowing anything about SPARQL [W3C 2013] etc. From the point of view of our survey this is not a multi-model database, but a possible use case of the respective DBMS; there is no cross-model query language, respective optimization of query evaluation etc. In this article we focus on extensions towards a new model which can be interlinked with other models supported by the DBMSs. Hence, in Table III we provide the indication of RDF support for DBMSs which are truly multi-model and which state the support for RDF directly as a part of the system. There exists a number of sources discussing various implementations of RDF support, such as, e.g. W3C wiki [W3C 2018a; 2018b] and comparative surveys focusing on triple stores [Wylot et al. 2018; Abdelaziz et al. 2017; Özsu 2016; Sakr and Al-Naymat 2010]. We refer an interested reader to them.

Tables IV, V and VI provide a closer look at the particular systems[6]. They overview the key characteristics of the systems divided according to their original type (i.e., relational, key/value, column etc.). In the first two tables we focus on:

(1) supported data formats,

[5]https://db-engines.com/en/ranking

[6]We deal with a more detailed description of each system in Section 4.

Table III. Overview of supported data models in multi-model DBMSs

Type	DBMS	Relational	Column	Key/value	JSON	XML	Graph	RDF	Nested data/UDT/object	Popularity (2018)
Relational	PostgreSQL	√		√	√	√			√	* * * * *
	SQL Server	√			√	√	√		√	* * * * *
	IBM DB2	√				√	√	√	√	* * * * *
	Oracle DB	√			√	√		√	√	* * * * *
	Oracle MySQL	√		√					√	* * * * *
	Sinew	√		√						*
Column	Cassandra		√				√		√	* * **
	CrateDB	√	√		√		√			*
	DynamoDB		√	√	√		√		√	* * *
	HPE Vertica		√		√		√			* * *
Key/value	Riak			√	√	√	√			**
	c-treeACE	√		√			√			*
	Oracle NoSQL DB	√		√			√	√		* * *
Document	ArangoDB			√	√		√			**
	Couchbase			√	√					* * **
	MongoDB			√	√				√	* * * * *
	Cosmos DB		√	√	√					* * *
	MarkLogic				√	√		√	√	* * * * *
Graph	OrientDB			√	√		√			* * *
Object	InterSystems Caché	√			√	√			√	*

(2) storage strategy used for the diverse data,
(3) what query language(s) it supports[7], and
(4) types of indices supported for the purpose of optimization of query evaluation.

In the third table we provide yes (√) / no (×) / unknown or unspecified (–) features informing:

(5) whether the database is distributed,
(6) whether the database requires schema definition for storing the data,
(7) whether the diverse data can be queried together using a single common language,
(8) whether there exists also a version for the cloud, and
(9) whether a special transaction management was introduced to handle the diverse data.

Characteristics (1) and (2) have already been described, while characteristics (4) are further analyzed and discussed later in this section. Considering characteristics (3), as we can see, query languages involve various approaches, both declarative and imperative. The options range from simple API (DynamoDB), full-text search (Riak), to extensions of popular standard query languages, such as SQL (e.g., PostgreSQL, Cassandra, or OrientDB) or XQuery (MarkLogic). Naturally SQL-extensions and SQL-like languages form the main approach (we devote to this aspect a separate Table VII).

[7]In Appendix B we also provide an overview of current query languages for popular data formats.

If we have a closer look at characteristics provided in Table VI. As we can see, most of the systems support data distribution. For the NoSQL databases, especially those of type key/value regardless the complexity of the value part (i.e., including column and document DBMSs) it is quite a natural feature. However, we can find this tendency also amongst other types of systems which reflects the general need for Big Data management. Flexible schema is not that common feature in general although we can find it, for example, also amongst relational databases which do not require schema for JSON or XML data. For NoSQL databases it is usually a common feature. Queries across multiple models are a kind of a must in multi-model databases, so most of the systems support them. In some cases, however, this information is unknown or irrelevant, depending on the type of the system. On the other hand, we have not managed to find any explicit information about existence of a special type of transaction management across diverse data models. This feature is however highly related to the way the system was extended towards multiple data models. Regarding cloud computing, we can witness a strong tendency of the DBMSs vendors towards the support of a version for the cloud. Again, this corresponds to the general trend in Big Data management, where the DaaS (Database as a Service) approach enables to create a solution for complex Big Data applications instantly.

Table VII is devoted to the overview of SQL extensions and SQL-like languages used in multi-model DBMSs. Again the systems are classified according to their original type in order to show that this probably most common and with regards to the popularity of SQL also logical approach can be found in all types of multi-model databases. At first sight, the least natural usage of SQL-like interface can probably be found amongst graph and document DBMSs. However, in this case the SQL clauses are simply extended towards the access of more complex data structures – in case of graph data the dot notation represents the edges, in case of nested document (JSON) data various operators enable to access deeper data levels including items of arrays. It is especially interesting to compare the latter approach with the way SQL/XML combines the access to relational and XML data via embedding XQuery.

Last but not least, in Table VIII we provide a summary of query optimization strategies used in the multi-model databases for the “non-native” formats. As expected, the most common type of query optimization is a kind B-tree/B+-tree index, especially in the case of relational databases which naturally exploit their most common and verified approach. Systems which support XML data also exploit a kind of native XML index, most commonly an ORDPATH-based approach which enables both efficient querying and data updates. A kind of hashing, a technique which can be used almost universally, is also a common approach in various types of DBMSs. However, in general there seems to be no universally acknowledged optimal or sub-optimal approach suitable for the multi-model query optimization. The distinct approaches are usually highly related to the way the system was extended towards other data models.

Summary. From the preceding discussion with regard to the varied aspects of multi-model databases, we summarize the observations in the following:

— The data models supported by multi-model databases include relational, column, key/value, document, XML, graph, and object.
— Multi-model databases employ cross-model languages based on the extension of SQL, XML, and graph languages.
— The data indices in multi-model databases include inverted index, B-tree, materialized view, hashing, and bitmap index. Most of them are based on an extension for relational or XML databases.

Table IV. Comparison of multi-model single-database DBMSs (part A).

Type	DBMS	Supported formats	Storage strategy	Query languages	Indices
Relational	PostgreSQL	relational, key/value, JSON, XML	relational tables – text or binary format + indices	extended SQL	inverted
	SQL Server	relational, XML, JSON, ...	text, relational tables	extended SQL	B-tree, full-text
	IBM DB2	relational, XML	native XML type	extended SQL/XML	XML paths / B+ tree, full-text
	Oracle DB	relational, XML, JSON, RDF	relational	SQL/XML or JSO extension of SQL	bitmap, B+tree, function-based, `XMLIndex`
	Oracle MySQL	relational, key/value	relational	SQL, memcached API	B-tree
	Sinew	relational, key/value, nested document, ...	logically a universal relation, physically partially materialized	SQL	–
Column	Cassandra	text, user-defined type	sparse tables	SQL-like CQL	inverted, B+ tree
	CrateDB	relational, JSON, BLOB, arrays	columnar store based on Lucene and Elasticsearch	SQL	Lucene
	DynamoDB	key/value, document (JSON)	column store	simple API (get/put/update) + simple queries over indices	hashing
	HPE Vertica	JSON, CSV	flex tables + map	SQL-like	for materialized data
Key/value	Riak	key/value, XML, JSON	key/value pairs in buckets	Solr	Solr
	c-treeACE	key/value + SQL API	record-oriented ISAM	SQL	ISAM
	Oracle NoSQL DB	key/value, (hierarchical) table API, RDF	key/value	SQL	B-tree
Document	ArangoDB	key/value, document, graph	document store allowing references	SQL-like AQL	mainly hash (eventually unique or sparse)
	Couchbase	key/value, document, distributed cache	document store + append-only write	SQL-based N_1QL	B+tree, B+trie
	MongoDB	document, graph	BSON format + indices	JSON-based query language	B-tree, hashed, geospatial
	Cosmos DB	document, key-value, graph, column	JSON format + indices	SQL-like query language	forward and inverted index mapping
	MarkLogic	XML, JSON, binary, text, ...	storing like hierarchical XML data	XPath, XQuery, SQL-like	inverted + native XML

Table V. Comparison of multi-model single-database DBMSs (part B). In the lower part of the table we include also systems which are not (yet) multi-model.

Type	DBMS	Supported formats	Storage strategy	Query languages	Indices
Graph	OrientDB	graph, document, key/value, object	key/value pairs + object-oriented links	Gremlin, extended SQL	SB-tree, extendible hashing, Lucene
Object	Caché	object, SQL or multi-dimensional, document (JSON, XML) API	multi-dimensional arrays	SQL with object extensions	bitmap, bit-slice, standard
Other	NuoDB	relational	key/value	–	–
	Redis	flat lists, sets, hash tables	key/value	–	–
	Aerospike	key/value	key/value	–	–

Table VI. Comparison of multi-model single-database DBMSs (yes/no features. In the lower part of the table we include also systems which are not (yet) multi-model.)

Type	DBMS	Data distribution	Flexible schema	Queries across models	Version for cloud	Multi-model transactions
Relational	PostgreSQL	×	√	√	√	×
	SQL Server	√	√	√	√	×
	IBM DB2	√	√	√	√	×
	Oracle DB	√	×	√	√	×
	Oracle MySQL	√	×	√	√	×
	Sinew	–	√	√	×	–
Column	Cassandra	√	×	√	√	×
	CrateDB	√	√	√	√	×
	DynamoDB	√	√	√	√	×
	HPE Vertica	√	√	√	√	×
Key/value	Riak	√	×	√	√	×
	c-treeACE	√	√	–	×	–
	Oracle NoSQL DB	√	×	√	√	×
Document	ArangoDB	√	√	√	√	×
	Couchbase	√	√	√	√	×
	MongoDB	√	√	√	√	×
	Cosmos DB	√	√	–	√	×
	MarkLogic	√	√	√	√	×
Graph	OrientDB	√	√	√	√	×
Object	Caché	√	√	–	√	–
Other	NuoDB	√	–	–	√	–
	Redis	√	–	–	√	–
	Aerospike	√	√	–	√	–

Table VII. Support of SQL extensions and SQL-like languages in multi-model databases)

Type	DBMS	SQL extension
Relational	PostgreSQL	Getting an array element by index, an object field by key, an object at a specified path, containment of values/paths, top-level key-existence, deleting a key/value pair / a string element / an array element with specified index / a field / an element with specified path, ...
	SQL Server	JSON: export relational data in the JSON format, test JSON format of a text value, JavaScript-like path queries SQLXML: SQL view of XML data + XML view of SQL relations
	IBM DB2	SQL/XML + embedding SQL queries to XQuery expressions
	Oracle DB	SQL/XML + JSON extensions (`JSON_VALUE`, `JSON_QUERY`, `JSON_EXISTS`, ...)
Document	Couchbase	Clauses `SELECT`, `FROM` (multiple buckets), ... for JSON
	Cosmos DB	Clauses `SELECT`, `FROM` (with inner join), `WHERE` and `ORDER BY` for JSON
	ArangoDB	key/value: insert, look-up, update document: simple QBE, complex joins, functions, ... graph: traversals, shortest path searches
Key/value	Oracle NoSQL DB	SQL-like, extended for nested data structures
	c-treeACE	Simple SQL-like language
Column	Cassandra	`SELECT`, `FROM`, `WHERE`, `ORDER BY`, `LIMIT` with limitations
	CrateDB	Standard ANSI SQL 92 + nested JSON attributes
Graph	OrientDB	Classical joins not supported, the links are simply navigated using dot notation; main SQL clauses + nested queries
Object	Caché	SQL + object extensions (e.g. object references instead of joins)

Table VIII. Query optimization strategies in multi-model databases)

Optimization	DBMS	Type
Inverted index	PostgreSQL	relational
	Cosmos DB	document
B-tree, B+-tree	SQL server	relational
	Oracle DB	relational
	Oracle MySQL	relational
	Cassandra	column
	Oracle NoSQL DB	key/value
	Couchbase	document
	MongoDB	document
Materialization	HPE Vertica	column
Hashing	DynamoDB	column
	ArangoDB	document
	MongoDB	document
	Cosmos DB	document
	OrientDB	graph
Bitmap index	Oracle DB	relational
	Caché	object
Function-based index	Oracle DB	relational
Native XML index	Oracle DB	relational
	SQL server	relational
	DB2	relational
	MarkLogic	document

— The existing multi-model databases have the features of data sharding, flexible schema, and a version for cloud. But they still lack of the support for multi-model transactions.

4. A CLOSER LOOK AT MULTI-MODEL DATABASE REPRESENTATIVES

In this section we explore in more detail different multi-model databases using the classification introduced at the beginning of Section 3. For each category we briefly describe key features of each of the representatives. We focus mainly on the aspects related to multi-model data management classified in the previous section. The aim is to provide the readers with a detailed look at each of the systems in the context of its competitors.

4.1. Relational Stores

One of the biggest sets of multi-model systems is naturally formed by relational stores. This is given by several reasons:

(1) Historically relational DBMSs are the most popular type of databases.
(2) The SQL standard has been extended towards other data formats (e.g., XML in SQL/XML) even before the arrival of Big Data and NoSQL DBMSs.
(3) The simplicity and universality of the relational model enables its extension towards other data models relatively easily.

PostgreSQL. The development of PostgreSQL[8] began in the mid-1980s aiming at a classical relational DBMS. The recent versions, however, bring many NoSQL features (like, e.g., materialized views enabling data duplicities for faster query evaluation or synchronous and asynchronous master-slave replication). There also exists a number of vendors of facilities to make it easy to set up, operate and scale PostgreSQL deployments in the cloud.

Following the support of the XML format, since 2006 it supports also storing of key/value pairs[9] in data type `HStore`. And since 2013 it supports storing of the JSON format in data types `json` and `jsonb`. In the former case an exact copy of the data is stored and it must be re-parsed on each access. Also not all operations are supported for data type `json` (such as containment and existence operators). In case of `jsonb` a decomposed binary format is used for data storage. It does not require re-parsing and supports indexing. However, the order of object keys, white space, and duplicate object keys are not preserved. The primitive types are mapped to native PostgreSQL types.

Both `json` and `jsonb` types can be used as other data types of PostgreSQL, such as in definition of table columns. There is no checking of schema of the stored JSON data; however, the documentation naturally recommends the JSON documents to have a somewhat fixed structure within a particular set stored at one place. An example of storing both relational and JSON data in PostgreSQL can be seen in Fig. 4.

Data stored in PostgreSQL data types `json` or `jsonb` can be queried using an SQL extension for JSON involving operators for getting an array element by index (`->int`), an object field by key (`->string`), or an object at a specified path (`#>text[]`).[10] Standard comparison operators are available only for `jsonb`. It also supports further operators like containment of values/paths in both directions (`@>` and `<@`), top-level key-existence for a string, any of the strings, or all of the strings (`?`, `?&` and `?|`), concatenation (`||`), and deleting either a key/value pair or a string element (`-text`), an array element with specified index (`-int`), or a field or element with specified path (`#-text[]`). PostgreSQL

[8] http://www.postgresql.org/
[9] Note that the first releases of NoSQL databases Redis and MongoDB are from 2009.
[10] Or, there exist their counterparts (with >> instead of >) returning the result in the form of text.

```
CREATE TABLE customer (
  id      INTEGER PRIMARY KEY,
  name    VARCHAR(50),
  address VARCHAR(50),
  orders  JSONB
);

INSERT INTO customer
VALUES (1, 'Mary', 'Prague',
 '{"Order_no":"0c6df508",
   "Orderlines":[
    {"Product_no":"2724f", "Product_Name":"Toy",  "Price":66},
    {"Product_no":"3424g", "Product_Name":"Book", "Price":40}]
  }');

INSERT INTO customer
VALUES (2, 'John', 'Helsinki',
 '{"Order_no":"0c6df511",
   "Orderlines":[
    {"Product_no":"2454f", "Product_Name":"Computer", "Price":34}]
  }');
```

id integer	name character varying (50)	address character varying (50)	orders jsonb
1	Mary	Prague	{"Orderlines":[{"Price":66,"Product_Name":"Toy","Product_no":"2724f"},{"Price":40,"Product_Name":...
2	John	Helsinki	{"Orderlines":[{"Price":34,"Product_Name":"Computer","Product_no":"2454f"}],"Order_no":"0c6df511"}

Fig. 4. An example of storing multi-model data in PostgreSQL

also provides functions for JSON creation, returning the length of an array, JSON object/array expansion, checking data types, transforming JSON data to records, or JSON data aggregation. An example of querying both relational and JSON data (defined in Fig. 4) can be seen in Fig. 5.

```
SELECT name,
  orders->>'Order_no' as Order_no,
  orders#>'{Orderlines,1}'->>'Product_Name' as Product_Name
FROM customer
WHERE orders->>'Order_no' <> '0c6df511';
```

name character varying (50)	order_no text	product_name text
Mary	0c6df508	Book

Fig. 5. An example of querying multi-model data in PostgreSQL

Data stored in `jsonb` can be indexed using the *Generalized Inverted Index* (GIN) corresponding to a set of pairs (key, posting list). GIN consists of a "B-tree index constructed over keys, where each key is an element of one or more indexed items and where each tuple in a leaf page contains either a pointer to a B-tree of heap pointers (*posting tree*), or a simple list of heap pointers (*posting list*) when the list is small enough".

By default the GIN index supports top-level key-exists operators (`?`, `?&` and `?|`, for a single string, all given strings, or any of the given strings respectively) and path/value-containment operator `@>`. Non-default GIN index supports only operator `@>`. The difference is that in case of default indexing for each key and value an independent index item is created. In case of non-default indexing for each value an index item is created as a hash of the value and all the related key(s).

SQL Server. Microsoft SQL Server[11] started in late 1980s as a relational DBMS. Since 2000 it supports XML and its access using SQLXML [Microsoft 2017c] (a deprecated Microsoft version of SQL extension for XML data), and thus is classified as an XML-enabled database. Since 2016 it supports also the JSON format [Popovic 2015], whereas the work with JSON data is quite similar to XML support. JSON data can be stored as a pure text in data type `NVARCHAR`. Or, function `OPENJSON` enables one to transform JSON text to a relational table, either with a pre-defined schema and mapping rules (JavaScript-like paths to JSON data), or without a schema as a set of key/value pairs.

In addition, thanks to Polybase [Microsoft 2017b], SQL Server 2016 can be considered also as a multi-model multi-database DBMS. Polybase is a technology that accesses both non-relational and relational data. In particular, it allows one to run SQL queries on external data in Hadoop[12] or Azure blob storage[13]. Microsoft Azure SQL database is a cloud database providing SQL Server functionality.

Regarding querying, SQLXML has the same aim as SQL/XML, but different syntax. [Holubova and Necasky 2009] Construct `OPENXML` enables to view XML data as SQL relations using a mapping which can be utilized using user-defined parameters. Construct `FOR XML` enables to view relational data as XML documents using four predefined modes denoting the complexity of the hierarchical structure.

In case of JSON data, SQL Server enables to export relational data in the JSON format (using clause `FOR JSON`), test whether a text value is in the JSON format (using function `ISJSON`), or parse a JSON text and on the specified JavaScript-like path extract a scalar value (using function `JSON_VALUE`) or an object/array (using clause `JSON_QUERY`). Function `JSON_MODIFY` enables one to update the value of a property.

Columns with XML data type can be indexed too. Using the ORDPATH schema [O'Neil et al. 2004] all tags, values and paths in the stored XML data are indexed within the *primary XML index*. Secondary indices can be created as well – a B+ tree can be built over pairs (path, value), tuples (primary_key_of_base_table, path, value), or pairs (value, path).

For the purpose of query optimization SQL Server does not support any special indexing technique for JSON data. Depending on its storage, either B-tree or full-text indices can be used.

IBM DB2. The first release of object-relational DBMS IBM DB2[14] dates back to early 1980s. IBM Db2 on Cloud is a fully managed database on cloud. Since 2007 it provides a support for XML (using the native XML storage feature called pureXML [Saracco et al. 2006]) and since 2012 it supports also RDF graphs (using extension called DB2-RDF [Bornea et al. 2013]).

XML data are stored [IBM Knowledge Center 2017b] in native XML data type columns in a parsed format reflecting the hierarchical structure, or using user-defined shredding into relational tables. The data are accessed [IBM Knowledge Center 2017a] using standard SQL/XML enhanced with several DB2-specific constructs, such as, e.g., embedding SQL queries to XQuery expressions.

In case of the XML data type DB2 supports several types of XML indices. [Holubova and Necasky 2009] The location (i.e., regions of storage) of each XML document is automatically stored in the *XML region index*. Unique XML paths and their IDs

[11] http://www.microsoft.com/en-us/server-cloud/products/sql-server/
[12] https://azure.microsoft.com/en-us/solutions/hadoop/
[13] https://azure.microsoft.com/en-us/services/storage/
[14] http://www.ibm.com/analytics/us/en/technology/db2/

are indexed automatically in the *XML column path index*. Query performance can be increased using user-defined *XML index* for selected XPath [W3C 2015a] expressions.

Oracle DB. Object-relational DBMS Oracle DB[15] was released in 1979 as the first commercial RDBMS based on SQL. Oracle8 was released in 1997 as the object-relational database. Oracle9i, released in 2001, introduced the ability to store and query XML. Oracle12c, released in 2013, was designed for the cloud, featuring an in-memory column store and support for JSON documents as well as RDF data (thanks to the Oracle Graph module).

XML data are stored similarly and in the case of DB2, i.e., either shredded into tables or in a native XML data type `XMLType` without the need to use the schema (but the validity can be checked if required). On the other hand, JSON data is stored as textual/binary data using `VARCHAR2`, `BLOB` (preferred, since it obviates the need for any character-set conversion), or `CLOB`. Also in this case a schema of the data is not required. Oracle only recommends to use the `is_json` CHECK constraint.

XML data are in Oracle DB accessed using standard SQL/XML. For the purpose of accessing JSON data Oracle extends SQL with SQL/JSON functions (`json_value` for selecting a scalar value, `json_query` for selecting one or more values, `json_table` for projecting JSON data to a virtual table), conditions (`json_exists`, `is (not) json`, `json_textcontains`), as well as a dot notation which acts similarly to a combination of `json_value` and `json_query` [Oracle 2017].

In case of XML data shredded into object-relational tables a B-tree index can be naturally used. For native XML storage the `XMLIndex` indexes paths, values, and relations parent-child, ancestor-descendant, and sibling. A variant of the ORDPATH numbering schema is exploited for storing positions of nodes.

can be created for SQL function `json_value`. For XML data it is denoted as deprecated.

MySQL. Open-source relational DBMS MySQL[16] was released in 1995. In 2008 it was acquired by SUN Microsystems and in 2010 by Oracle. In 2014 the first version of MySQL cluster enabling data sharding and replication was released. With the support of Memcached API[17] (since 2011) it enables to combine relational and key/value data access advantages. By default, pairs (key, value) are stored in the same table, i.e. no schema has to be defined. User-defined key prefix can however determine a pre-defined table and column where the value should be stored. [Keep 2011] Most MySQL indices are stored in B-trees, R-trees are used for spatial data types, `MEMORY` tables support hash indices.

Sinew. The DBMS Sinew [Tahara et al. 2014] is based on the idea of creating a new layer above a traditional relational DBMS that enables to query multi-model data (key/value, relational, nested document etc.) without a pre-defined schema. A logical view of the data is provided to the user in the form of a universal table. Columns of the table correspond to unique keys in the dataset (nested data is flattened).

Physically the data is stored in an underlying relational DBMS. Depending on the query workload a subset of the columns of the logical table is materialized, others are serialized in a single binary column. The storage schema is periodically adapted to the evolving workload.

[15]https://www.oracle.com/database/index.html
[16]https://www.oracle.com/mysql/index.html
[17]http://www.memcached.org/

4.2. Column Stores

Another large group of multi-model databases is represented by NoSQL column stores. Note that the term "column store" can be understood in two ways. (1) A *column-oriented* store is a DBMS (not necessarily NoSQL) that does not store data tables as rows, but as columns. These systems are usually used in analytics tools. An example is, e.g., HPE Vertica. (2) *Column-family* (or *wide-column*) stores represent a type of NoSQL databases which support tables having distinct numbers and types of columns, like, e.g., Cassandra. The underlying storage strategy can be arbitrary, including column-oriented, so these two groups can overlap. This section is devoted primarily to the second group of databases – column-family stores.

Cassandra. Apache Cassandra[18] (first released in 2008) is an open source NoSQL column-family store. DataStax Enterprise[19], a database for cloud applications, results from Cassandra. Using SQL-like *Cassandra Query Language* (CQL) it enables to store the data in sparse tables. Apart from scalar data types (like `text` or `int`), it supports three types of collections (`list`, `set` and `map`), tuples, and user defined data types (which can consist of any data types), together with respective operations for storing and retrieval of the data.

Internally the data are stored in *SSTables* (Sorted String Tables) originally proposed in Google system Bigtable [Chang et al. 2008]. An SSTable is "an ordered immutable map from keys to values, where both keys and values are arbitrary byte strings". It is further divided into blocks which are indexed to speed up data look up. Since SSTables are immutable, modified data are stored to a new SSTable and periodically merged using *compaction*.

Since 2015 Cassandra supports also the JSON format [DataStax, Inc. 2015]; however, the respective tables, i.e., the schema of the data, must be first specified. An example of storing both simple scalar and JSON data in Cassandra can be seen in Fig. 6.

The *Cassandra Query Language* (CQL) [The Apache Software Foundation 2017] can be considered as a subset of SQL. It consists of clauses `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `ORDER BY`, and `LIMIT`. However, only a single table can be queried in `FROM` clause and there are certain limitations for conditions in `WHERE` clause, such as restrictions only to the primary key or columns with a secondary index etc. Sorting is supported only according to the columns which determine how data are sorted and stored on disk. Clause `SELECT JSON` can be used to return each row as a single JSON encoded map; the mapping between JSON and Cassandra types is the same as in case of storing.

There are several types of indices in Cassandra. The primary key is always automatically indexed using an inverted index implemented using an auxiliary table. Secondary indices can be explicitly added for the columns according to which we want search data, including collections. The respective *SSTable Attached Secondary Indices* (SASI) are implemented using memory mapped B+ trees and thus allow also range queries. Indices are, however, not recommended for "high-cardinality columns, tables that use a counter column, a frequently updated or deleted column, and to look for a row in a large partition unless narrowly queried" [DataStax, Inc. 2013].

CrateDB. CrateDB[20] was released in 2016 after 3 years of development. It is a distributed column-oriented SQL database with a dynamic schema which can store also nested JSON documents, arrays, and BLOBs. It is built upon several existing open

[18] http://cassandra.apache.org/
[19] http://www.datastax.com/products/datastax-enterprise
[20] https://crate.io/

```
create keyspace myspace
WITH REPLICATION = { 'class' : 'SimpleStrategy',
                     'replication_factor' : 3 };

CREATE TYPE myspace.orderline (
   product_no text,
   product_name text,
   price float
   );

CREATE TYPE myspace.myorder (
   order_no text,
   orderlines list<frozen <orderline>>
   );

CREATE TABLE myspace.customer (
   id INT PRIMARY KEY,
   name text,
   address text,
   orders list<frozen <myorder>>
   );
```

```
INSERT INTO myspace.customer JSON
' {"id":1,
   "name":"Mary",
   "address":"Prague",
   "orders" : [
   { "order_no":"0c6df508",
     "orderlines":[
     { "product_no" : "2724f",
       "product_name" : "Toy",
       "price" : 66 },
       { "product_no" : "3424g",
         "product_name" :"Book",
         "price" : 40 } ] } ]
  }';

INSERT INTO myspace.customer JSON
' {"id":2,
   "name":"John",
   "address":"Helsinki",
   "orders" : [
   { "order_no":"0c6df511",
     "orderlines":[
     { "product_no" : "2454f",
       "product_name" : "Computer",
       "price" : 34 } ] } ]
  }';
```

Fig. 6. An example of storing multi-model data in Cassandra

source technologies, such as Elasticsearch[21] or Lucene[22]. CrateDB can be deployed to any operating system capable of running Java and thus also various cloud platforms.

Each row of a table in CrateDB is a semi-structured document. [Crate.io 2017] Every table in CrateDB is sharded across the nodes of a cluster, whereas each shard is a Lucene index. Operations on documents are atomic.

Data in CrateDB can be accessed via a standard ANSI SQL 92. Nested JSON attributes can be included in any SQL command. For this purpose, CrateDB added an SQL layer to a Lucene index-based data store using Elasticsearch interface to access the underlying Lucene indices.

DynamoDB. Amazon DynamoDB[23] was released in 2012 as a cloud database which supports both (JSON) documents and key/value flexible data models. In DynamoDB, a table is schemaless and it corresponds to a collection of items. An item is a collection of attributes and it is identified by a primary key. An attribute consists of a name, a data type, and a value. The data type can be a scalar value (string, number, Boolean etc.), a document (list or map), or a set of scalar values. The data items in a table do not have to have the same attributes. [Amazon 2017]

DynamoDB primarily supports a simple API for creating / updating / deleting / listing a table and putting / updating / getting / deleting an item. A bit more advanced feature enables to query over primary or secondary indices using comparison operators.

Two types of primary keys are supported in DynamoDB: The *partition key* determines the partition where a particular data item is stored. The *sort key* determines the order in which the data items are stored within a partition. DynamoDB also sup-

[21] https://www.elastic.co/
[22] http://lucene.apache.org/
[23] https://aws.amazon.com/dynamodb/

ports two types of secondary indices – global and local. A secondary index consists of a subset of attributes from a selected base table and a corresponding alternate key. Global secondary index can have the partition key different from the base table, local secondary index can not.

HPE Vertica. HPE Vertica[24] is a high-performance analytics engine which was designed to manage Big Data. Vertica offers two deployment modes for running in the clouds. The storage organization is column-oriented, whereas it supports standard SQL interface enriched by analytics capabilities. Since 2013 it was extended with *flex tables* [Hewlett Packard Enterprise 2018] which do not require schema definitions, enable to store also semi-structured data (e.g., JSON or CSV formats), and support SQL queries.

Creating flex tables is similar to creating classical tables, except column definitions are optional (if present, the table is denoted as *hybrid*). Vertica implicitly adds a `NOT NULL` column `__raw__` which stores the loaded semi-structured data. For a flex table without other column definitions, it also adds auto-incrementing column `__identity__` used for segmentation and sort order. The loaded data are stored in an internal map data format `VMap`, i.e., a set of key/value pairs, called *virtual columns*. Selected keys can be then materialized by promoting virtual columns to real table columns.

Besides the flex table itself, Vertica creates also associated keys table (with self-descriptive columns `key_name`, `frequency`, and `data_type_guess`) and a default view for the main flex table. The records under the `key_name` column of the table are used as view columns, along with any values for the key. If no values exist, the column value is `NULL`. Both the keys table and the default view enable to explore the data to determine its contents since the schema of the stored data is not required.

A flex table can be processed using SQL commands `SELECT`, `COPY`, `TRUNCATE`, and `DELETE`. Custom views can also be created. Both virtual and real columns can be queried using classical `SELECT` command. A `SELECT` query on a flex table or a flex table view invokes the `maplookup()` function to return information on virtual columns. Materializing virtual columns by *promoting* them to real columns improves query performance (at the cost of more space requirements). Promoting flex table columns results in a hybrid table so both raw and real data can still be queried together.

4.3. Key/value Stores

In general, key/value stores are considered as the least complex NoSQL DBMSs which support only a simple (but fast) API for storing and retrieving an item having a particular ID. These systems, however, usually provide more complex operations of the value part; hence, the convergence to multi-model systems is a relatively natural evolution step.

Riak. Riak[25] was first released in 2009 as a classical key/value DBMS. On top of it Riak CS provides a distributed cloud storage. Since 2014 two features – Riak Search and Riak Data Types – make it possible to use Riak also as a document store with querying capabilities [Basho Technologies, Inc. 2014]. Riak Data Types, based on a conflict-free replicated data type (CRDT), involve sets, maps (which enable embedding of any data type), counters etc. and can be indexed and searched through. Riak Search 2.0 is in fact an integration of Solr[26] for indexing and querying and Riak for storage and distribution. Riak Search must first be configured with a Solr schema (eventually the default one) so that Solr knows how to index value fields. Indices, e.g.,

[24] http://www.vertica.com/
[25] http://basho.com/products/riak-kv/
[26] http://lucene.apache.org/solr/

over particular fields of an XML or JSON document, are named Solr indices and must be associated with a bucket (i.e., a named set of key/value pairs) or a bucket type (i.e., a set of buckets). The fields to be indexed are extracted from the data using extractors. Riak currently supports JSON, XML, plain text, and Riak Data Types extractors, but it is possible to implement an own extractor as well.

As we have described before, using Solr, Riak enables to query over data that have been previously indexed. All distributed Solr queries are supported [Basho Technologies, Inc. 2017], including wild-cards, proximity search, range search, Boolean operators, grouping etc.

c-treeACE. FairCom c-treeACE[27] is denoted by its vendor as a *No+SQL* DBMS [Brown 2016], offering both NoSQL and SQL in a single database. c-treeACE supports both relational and non-relational APIs. It is based on an Indexed Sequential Access Method (ISAM) structure supporting operations with records, their sets, or files in which they are stored. The original version supported only the ISAM API; the SQL API was added in 2003.

Oracle NoSQL Database. Oracle NoSQL Database[28], first released in 2011, is a scalable, distributed NoSQL database built upon the Oracle Berkeley DB[29]. It can be also run as a fully managed cloud service using the Oracle Cloud. Contrary to Oracle MySQL, Oracle NoSQL Database is a key/value DBMS which (since release 3.0 in 2014) supports a table API, i.e., SQL. In addition, RDF support was added thanks to the Oracle Graph module. First, a definition of the tables must be provided, which includes table and attribute names, data types (involving scalar types, arrays, maps, records, and child tables corresponding to nested subtables), primary (and eventually shard) key, indices etc. (When using child tables, by default, child tables are not retrieved when retrieving a parent table, nor is the parent retrieved when a child table is retrieved.) An example of storing both relational and JSON data in Oracle NoSQL Database can be seen in Fig. 7; the structure of the resulting table can be seen in Fig. 8. An example of querying both relational and JSON data is provided in Fig. 9.

Oracle NoSQL Database secondary indices are implemented using distributed, shard-local B-trees [Oracle 2014]. The DBMS supports secondary indexing over simple, scalar as well as over non-scalar and nested data values.

4.4. Document Stores

Document DBMSs can be considered as advanced key/value stores with complex value part that can be queried. Hence, each document store can be considered as a kind of multi-model DBMS since it naturally supports also storing of key/value or column data.

ArangoDB. Contrary to most of the other DBMSs, ArangoDB was from the beginning created as a native multi-model system. Its first release is from 2011. It can be also run as a cloud-hosted database service. It supports key/value, document, and graph data. For the purpose of querying across all the data models it provides a common language [ArangoDB 2017]. ArangoDB however primarily serves documents to clients. Documents are represented in the JSON format and grouped in collections. A document contains a collection of attributes, each having a value of an atomic type or a compound type (an array or an embedded document/object).

[27] https://www.faircom.com/products/c-treeace
[28] http://www.oracle.com/technetwork/database/database-technologies/nosqldb/overview/index.html
[29] http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html

```
create table Customers (
  id integer,
  name string,
  address string,
  orders array (
    record (
      order_no string,
      orderlines array (
        record (
          product_no string,
          product_name string,
          price integer ) ) )
    ),
  primary key (id)
);

import -table Customers -file customer.json
```

Content of file **customer.json**:

```
{  "id":1,
   "name":"Mary",
   "address":"Prague",
   "orders" : [
     { "order_no":"0c6df508",
       "orderlines":[
         { "product_no" : "2724f",
           "product_name" : "Toy",
           "price" : 66 },
         { "product_no" : "3424g",
           "product_name"  :"Book",
           "price" : 40 } ] } ]
}
{  "id":2,
   "name":"John",
   "address":"Helsinki",
   "orders" : [
     {"order_no":"0c6df511",
      "orderlines":[
       { "product_no" : "2454f",
         "product_name" : "Computer",
         "price" : 34  } ] } ]
}
```

Fig. 7. An example of storing multi-model data in Oracle NoSQL Database

```
sql-> select * from Customers;
 +----+------+----------+-----------------------------+
 | id | name | address  |           orders            |
 +----+------+----------+-----------------------------+
 |  2 | John | Helsinki | order_no         | 0c6df511 |
 |    |      |          | orderlines                  |
 |    |      |          |     product_no   | 2454f    |
 |    |      |          |     product_name | Computer |
 |    |      |          |     price        | 34       |
 +----+------+----------+-----------------------------+
 |  1 | Mary | Prague   | order_no         | 0c6df508 |
 |    |      |          | orderlines                  |
 |    |      |          |     product_no   | 2724f    |
 |    |      |          |     product_name | Toy      |
 |    |      |          |     price        | 66       |
 |    |      |          |                             |
 |    |      |          |     product_no   | 3424g    |
 |    |      |          |     product_name | Book     |
 |    |      |          |     price        | 40       |
 +----+------+----------+-----------------------------+
```

Fig. 8. An example of storing multi-model data in Oracle NoSQL Database – the resulting table

A *document collection* always has a primary key attribute `_key` and in the absence of further secondary indices the document collection behaves like a simple key/value store. Special *edge collections* store documents as well, but they include two special attributes, `_from` and `_to`, which enable to create relations between documents. Hence two documents (vertices) stored in document collections are linked by a document (edge) stored in an edge collection. This is ArangoDB's graph data model.

ArangoDB query language (AQL) allows complex queries. Despite the different data models, it is similar to SQL. In case of the key/value store the only operations that are possible are single key lookups and key/value pair insertions and updates. In case of the document store queries can range from a simple "query by example" to complex "joins" using many collections, usage of functions (including user-defined ones) etc. For

```
sql-> SELECT c.name, c.orders.order_no,
c.orders.orderlines[0].product_name
   -> FROM customers c
   -> where c.orders.orderlines[0].price > 50;
 +------+----------+--------------+
 | name | order_no | product_name |
 +------+----------+--------------+
 | Mary | 0c6df508 | Toy          |
 +------+----------+--------------+
```

```
sql-> SELECT c.name, c.orders.order_no,
   -> [c.orders.orderlines[$element.price > 35]]
   -> FROM customers c;
 +------+----------+-------------------------+
 | name | order_no |        Column_3         |
 +------+----------+-------------------------+
 | Mary | 0c6df508 | product_no   | 2724f    |
 |      |          | product_name | Toy      |
 |      |          | price        | 66       |
 |      |          |                         |
 |      |          | product_no   | 3424g    |
 |      |          | product_name | Book     |
 |      |          | price        | 40       |
 +------+----------+-------------------------+
 | John | 0c6df511 |                         |
 +------+----------+-------------------------+
```

Fig. 9. An example of querying multi-model data in Oracle NoSQL Database

the purpose of graph data various types of traversing graph structures and shortest path searches are available. The most notable difference is probably the concept of loops borrowed from programming languages.

ArangoDB involves several types of indices. Some of them are created automatically, others which can be created on collection level are user-defined. For each collection there is a *primary index* which is a hash index for the document keys (attribute `_key`) of all documents in the collection. Every edge collection also has an automatically created edge index which provides quick access to documents by either their attributes `_from` or `_to`. It is also implemented as a hash index which stores a union of all the attributes. A user-defined index is also hash, in particular unsorted, so it supports equality lookups but no range queries or sorting. Optionally it can be declared as unique or sparse.

Another type of index is called a *skiplist*. It is a sorted index structure used for lookups, range queries and sorting. Optionally it can also be declared as unique or sparse. Other types of indices, such as persistent, full-text or geo, are available too.

Couchbase. Another document DBMS with a support for multiple data models is Couchbase[30], originally known as Membase, first released in 2010 and it can be easily deployed in the cloud. It is both key/value and document DBMS with an SQL-based query language. Documents (in JSON) are stored in data containers called *buckets* without any pre-defined schema. The storage approach is based on an append-only write model for each file for efficient writes which also requires regular compaction for cleanup. A special type of *memcached buckets* support caching of frequently-used data. Hence they reduce the number of queries a database server must perform. The server provides only in-RAM storage and data does not persist on disk. If it runs out of space in the buckets RAM quota, it uses the Least Recently Used (LRU) algorithm to evict items from the RAM.

The SQL-based query language of Couchbase, denoted as N_1QL, enables to access the JSON data. In addition, key/value API, MapReduce API, and spatial API for geographical data is provided. N_1QL involves classical clauses such as `SELECT`, `FROM` (targeting multiple buckets), `WHERE`, `GROUP BY`, and `ORDER BY`.

Two types of indices are supported in Couchbase – B+tree indices similar to those used in relational databases and B+trie (a hierarchical B+-tree based trie). B+trie provides a more efficient tree structure compared to B+trees and ensures a shallower tree hierarchy.

[30] http://www.couchbase.com/

MongoDB. Probably the most popular document DBMS MongoDB[31] (whose development began in 2007) has been declared as multi-model at the end of 2016. Its document model, that can naturally store also simple key/value pairs and table-like structures, has been extended towards graph data. In addition, MongoDB Atlas is a cloud-hosted database service.

In general, documents in MongoDB (expressed in JSON) have a flexible schema and hence the respective *collections* do not enforce document structure (except for field `_id` uniquely identifying each document). The user can decide whether to embed the data or to use references to other documents (which enable to form a graph). Operations are atomic at the document level.

MongoDB query language uses a JSON syntax. It supports both selection of documents using conditions (involving logical operators, comparison operators, field existence, regular expressions, bitwise operators etc.), projections of selected fields of the result, accessing of document fields in an arbitrary depth etc. MongoDB does not support joins. There are two methods for relating documents: (1) *Manual references* where one document contains field `_id` of another document and thus a second query must be always used to access the referenced data. (2) *DBRefs* references, where a document is referenced using field `_id`, collection name, and (optionally) database name, i.e. different document collections can be mutually linked. Also in this case a second query must be used to access the data, but there are drivers involving helper methods that form the query for the DBRefs automatically.

Documents are physically stored in BSON[32] – a binary representation of JSON documents. The maximum BSON document size is 16MB. MongoDB automatically creates a unique primary index on field `_id`. It also supports a number of secondary indices, such as single-field, compound (to index multiple fields), multikey (to index the content stored in arrays), geospatial, text, or hashed. Most types of MongoDB indices are based on a B-tree data structure [MongoDB, Inc. 2017].

Cosmos DB. Azure Cosmos DB[33] (before May 2017 called DocumentDB) from Microsoft is a cloud, schema-less, originally document database which supports ACID compliant transactions. It is multi-model and it supports document (JSON), key/value, graph, and columnar data models. For a new instance of Cosmos DB, the user chooses one of the data models and respective APIs to be used.

For accessing document, columnar, or key/value data Cosmos DB uses an SQL-like query language [Microsoft 2017a]. Every query consists of clause `SELECT` and optional clauses `FROM`, `WHERE` and `ORDER BY`. Clause `FROM` can involve inner joins whereas we join fields in JSON documents accessible via dot notation and positions of items in the arrays. Clause `WHERE` can involve arithmetic, logical, comparison, bitwise and string operators. For working with graph data the standard Gremlin [Rodriguez 2015] API is supported.

By default, Cosmos DB automatically indexes all documents in the database and it does not require any schema or creation of secondary indices. These defaults can be modified by setting an indexing policy specifying including/excluding documents and paths (selecting document fields) to/from index, configuring index types (hash/range/spatial for numbers/strings/points/polygons/linestrings and their required precision), and configuring index update modes (consistent/lazy/none). The indexing strategy in Cosmos DB [Shukla et al. 2015] is based on two strategies: (1) a map of

[31] https://www.mongodb.com/
[32] http://bsonspec.org/
[33] http://www.cosmosdb.com

tuples (document id, path) and (2) a map of tuples (path, document id). Particular path patterns can be excluded from the index.

4.4.1. XML Stores. XML stores can be considered as a special type of document databases. However, XML stores do not belong to the group of core NoSQL databases, so they are usually not intended for Big Data and respective distributed processing.

MarkLogic. The development of MarkLogic[34] began in 2001 as a native XML database, i.e., a system natively supporting hierarchical semi-structured XML data. Since 2008 it supports also the JSON format [MarkLogic Corporation 2017a] and currently also other data formats, like, e.g., RDF, binary, or textual. It can be deployed, managed and monitored in various cloud platforms.

As can be seen in Fig. 10, MarkLogic models a JSON document like an XML document, i.e., as a tree of nodes, rooted at an auxiliary document node. The nodes represent objects, arrays, text, number, Boolean, or null values. The name of a node corresponds to the property name if specified, otherwise unnamed nodes are supported. This similarity provides a unified way to manage and index documents of both types. MarkLogic indexes the structure of the data upon loading regardless their eventual schema. An example of storing both XML and JSON data in MarkLogic can be seen in Fig. 11.

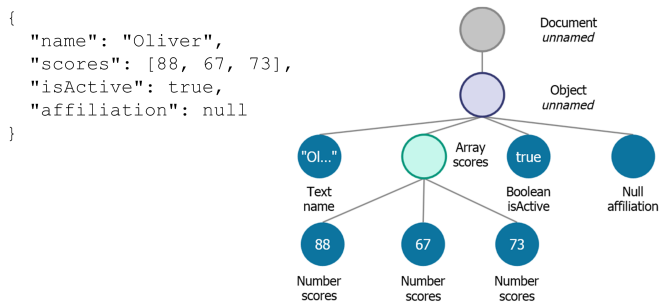


Fig. 10. An example of modeling JSON data as trees in MarkLogic (source: https://developer.marklogic.com/features/json)

JavaScript for JSON documents:

```
declareUpdate();
xdmp.documentInsert("/myJSON1.json",
{
  "Order_no":"0c6df508",
   "Orderlines":[
    { "Product_no":"2724f",
      "Product_Name":"Toy",
      "Price":66 },
    {"Product_no":"3424g",
     "Product_Name":"Book",
     "Price":40}]
}
);
```

XQuery update extension for XML documents:

```
xdmp:document-insert("/myXML1.xml",
<product no="3424g">
  <name>The King's Speech</name>
  <author>Mark Logue</author>
  <author>Peter Conradi</author>
</product>
);
```

Fig. 11. An example of storing multi-model data in MarkLogic

[34]http://www.marklogic.com/

Thanks to the tree representation, the JSON documents can be traversed using XPath queries which can be called also from the JavaScript and XQuery code. For querying using SQL MarkLogic enables to create a view which flattens the JSON/XML hierarchical data into tables. An example of querying both XML and JSON data using XQuery can be seen in Fig. 12.

```
let $product := fn:doc("/myXML1.xml")/product
let $order := fn:doc("/myJSON1.json")[Orderlines/Product_no = $product/@no]
return $order/Order_no

Result: 0c6df508
```

Fig. 12. An example of querying multi-model data in MarkLogic

Actually, MarkLogic stores, retrieves and indexes *document fragments*. By default a fragment is the whole document. But, MarkLogic also enables users to break large XML documents into document fragments. JSON documents are single-fragment; the maximum size of a JSON document is 512 MB for 64-bit machines.

MarkLogic maintains a default *universal index* [MarkLogic Corporation 2017b] to search the text, structure, and their combinations for XML and JSON data. It includes an inverted index for each word (or phrase), XML element and JSON property and their values (further optimized using hashing) and an index of parent-child relationships. Range indices for efficient evaluation of range queries can be further specified. A range index can be described as two data structures: (1) an array of pairs (document id, value) sorted by document ids and (2) an array of pairs (value, document id) sorted by values (whereas both are further optimized so that the values are stored only once). A path range index further enables to index JSON properties defined by an XPath expression. Last but not least, MarkLogic enables one to create *lexicons*, i.e., lists of unique words/values that enable identification of a word/value in the database and the number of its appearances. There are several types of lexicons, such as word, value, value co-occurrence, range etc.

4.5. Graph Stores

NoSQL graph databases enable to store the most complex data structures and involve a specific data access. Adding another type of data model thus increases the complexity of the problem. This is probably the reason why there seems to exist only a single representative of a graph multi-model database.

OrientDB. The first release of OrientDB[35] from 2010 was implemented on the basis of an object DBMS. Currently it is an open source NoSQL DBMS supporting graph, key/value, document, and object models. It can be deployed and managed in most cloud environments.

An element of storage [OrientDB 2017a] is a *record* having a unique ID and corresponding to a document (formed by a set of key/value pairs), a BLOB, a vertex, or an edge. *Classes* contain and define records; however, they can be schema-full, schema-less, or schema-mixed. Classes can inherit (all properties) from other classes. If class properties are defined, they can be further constrained or indexed.

Classes can have relationships of two types: (1) *Referenced* relationships are stored as physical links managed by storing the target record ID in the source record(s), similarly to storing pointers between two objects in memory. Four kinds of relationships are supported – `LINK` pointing to a single record and `LINKSET`, `LINKLIST`, or `LINKMAP`

[35] http://orientdb.com/orientdb/

pointing to several records. (2) *Embedded* relationships are stronger and stored within the record that embeds. Embedded records do not have their own record, they are only accessible through the container record and cannot exist without it. Similarly to links, four kinds of embedded links are supported: `EMBEDDED`, `EMBEDDEDSET`, `EMBEDDEDLIST`, and `EMBEDDEDMAP`. An example of storing both graph and JSON data in OrientDB together with a graphical visualization of the result can be seen in Fig. 13.

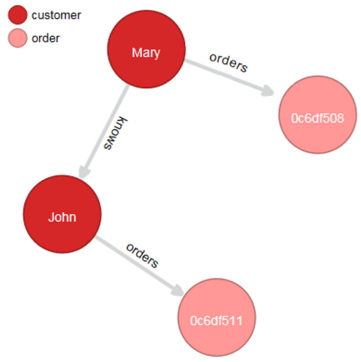
```
CREATE CLASS orderline EXTENDS V
CREATE PROPERTY orderline.product_no STRING
CREATE PROPERTY orderline.product_name STRING
CREATE PROPERTY orderline.price FLOAT

CREATE CLASS order EXTENDS V
CREATE PROPERTY order.order_no STRING
CREATE PROPERTY order.orderlines EMBEDDEDLIST orderline

CREATE CLASS customer EXTENDS V
CREATE PROPERTY customer.id INTEGER
CREATE PROPERTY customer.name STRING
CREATE PROPERTY customer.address STRING

CREATE CLASS orders EXTENDS E

CREATE CLASS knows EXTENDS E
```




```
CREATE VERTEX order CONTENT {
   "order_no":"0c6df508",
   "orderlines":[
    { "@type":"d",
      "@class":"orderline",
      "product_no":"2724f",
      "product_name":"Toy",
      "price":66 },
    { "@type":"d",
      "@class":"orderline",
      "product_no":"3424g",
      "product_name":"Book",
      "price":40}]
}
```

```
CREATE VERTEX order CONTENT {
   "order_no":"0c6df511",
   "orderlines":[
    { "@type":"d",
      "@class":"orderline",
      "product_no":"2454f",
      "product_name":"Computer",
      "price":34 }]
}
```

```
CREATE VERTEX customer CONTENT {
  "id" : 1,
  "name" : "Mary",
  "address" : "Prague"
  }

CREATE VERTEX customer CONTENT {
  "id" : 2,
  "name" : "John",
  "address" : "Helsinki"
  }
```

```
CREATE EDGE orders FROM
   (SELECT FROM customer WHERE name = "Mary")
   TO
   (SELECT FROM order WHERE order_no = "0c6df508")

CREATE EDGE orders FROM
   (SELECT FROM customer WHERE name = "John")
   TO
   (SELECT FROM order WHERE order_no = "0c6df511")

CREATE EDGE knows FROM
   (SELECT FROM customer WHERE name = "Mary")
   TO
   (SELECT FROM customer WHERE name = "John")
```

Fig. 13. An example of storing multi-model data in OrientDB

OrientDB supports querying the data with graph-traversal language Gremlin or SQL extended for graph traversal [OrientDB 2017b]. The main difference in SQL

commands is in class relationships represented by links. Classical joins are not supported and the links are simply navigated using dot notation. Otherwise the main SQL clauses as well as nested queries are supported.

OrientDB uses several indexing mechanisms. *SB-tree* [O'Neil 1992] is based on classical B-tree optimized for data insertions and range queries. It has variants (dis)allowing duplicities and for full text indexing. Significantly faster extendible hashing has the same variants but does not support range queries. Lucene full text and spatial indexing plugins are also available.

4.6. Other Stores

In this section we focus briefly on other types multi-model systems. We mention a representative of multi-model object stores and multi-use-case stores. And we also discuss systems which will probably soon become multi-model, as well as systems which are on the contrary no longer available.

4.6.1. Object Stores. With their emergence, object stores were expected to become the key database technology, similarly to object-oriented programming. Even though relational databases have maintained their leadership, there exist highly successful object DBMSs used in specific areas. Since object model enables to store any kind of data, a multi-model extension is a relatively straightforward step.

InterSystems Caché. DBMS Caché[36] from InterSystems was first launched in 1997 and recently transformed to the IRIS Data Platform[37]. It is an object database[38] which stores data in sparse, multidimensional arrays capable of carrying hierarchically structured data. The data can be accessed using several APIs – via objects based upon the ODMG standard (involving inheritance and polymorphism, embedded objects, collections etc.), SQL (including DDL, transactions, referential integrity, triggers, stored procedures etc. with various object enhancements), or direct (and highest-performance) manipulation of its multidimensional data structures. Hence, both schemaless and schema-based storage strategy is available. In addition, since 2016 it supports also documents in JSON or XML [InterSystems 2016].

Being an object database, Caché provides also an SQL API for data access enhanced with object features [InterSystems 2017], e.g. following object references using the operator -> instead of joins. In general, each instance of a persistent class has a "flattened" representation as a row in a table accessible via SQL.

The key important index structure in DBMS Caché is a *bitmap* index [InterSystems 2015] – a series of highly compressed bitstrings to represent the set of object IDs that correspond to a given indexed value. It is further extended with a *bitslice* index for a numeric data field when that field is used for an aggregate calculation SUM, COUNT, or AVG. It represents each numeric data value as a binary bit string and creates a bitmap for each digit in the binary value to record which rows have 1 for that binary digit. Finally, *standard* indices correspond to an array that associates the indexed values with the RowIds of the rows that contain the values.

4.6.2. Multi-Use-Case Stores. A related group of DBMSs can be denoted as *multi-use-case*. These systems do not aim at storing multiple data models and querying across them, but rather at systems suitable for various types of database applications. Hence the idea of one-size-fits-all is viewed from the viewpoint of use cases.

[36] http://www.intersystems.com/our-products/cache/

[37] https://www.intersystems.com/products/intersystems-iris/

[38] In fact, originally it was a key/value database – a long time before this term was introduced in the world of NoSQL databases. Currently it is however usually denoted as an object database.

For example SAP HANA DB[39] is an in-memory, column-oriented, relational DBMS. It exploits and combines the advantages of a row (OLTP) and columnar (OLAP) storage strategy together with in-memory processing in order to provide a highly efficient and universal data management tool.

Another example is OctopusDB[40] whose aim is to mimic OLTP, OLAP, streaming and other types of database systems. For this purpose it does not have any fixed hard coded (e.g., row or columnar) store, but it records all database operations to a sequential *primary log* by creating appropriate logical log records. It later creates arbitrary physical representations of the log (called *storage views*), depending on the workload.

4.6.3. Not (Yet) Multi-Model. Currently there also exists a number of DBMSs which cannot be denoted as multi-model. However, their current architecture enables this extension or such an extension is currently under development. Another set of DBMSs mentioned in this section involves systems whose support for multiple data models is highly limited. But in this case we can also assume that it will probably be (soon) extended.

NuoDB. NuoDB[41], released under version 1.0 in 2013, is a relational, or more specifically NewSQL DBMS which works in the cloud. As mentioned in [NuoDB 2013] "the NuoDB SQL engine is a personality for the atom layer", whereas the authors of NuoDB "are actively working on personalities other than the default SQL personality". Data is stored and managed using self-coordinating objects (*atoms*) representing data, indices, schemas, etc. Atomicity, consistency and isolation are ensured at the level of atom interaction without the knowledge of their SQL structure. Hence, replacing the SQL front-end would not influence the ACID semantics.

Redis. Redis[42] was first released in 2009 as a NoSQL key/value store. However, in the value part it supports not only strings, but also a list of strings, an (un)ordered set of strings, a hash table etc., together with respective operations for storing and retrieval of the data. Although the basic value types cannot be nested, the *Redis Modules*[43] are expected to turn Redis into multi-model database [Curtis 2016]. Redis Modules are add-ons to Redis which extend Redis to cover most of the popular use cases for any industry.

Aerospike. DBMSs Aerospike[44], first released in 2011, is a key/value store with the support for maps and lists in the value part that can nest. In addition, in 2012 Aerospike acquired AlchemyDB, "the first NewSQL database to integrate relational database management system, document store, and graph database capabilities on top of the Redis open-source key/value store" [Aerospike, Inc. 2012].

4.6.4. No More Available. Even in the dynamically evolving world of multi-model databases we can find also systems which are no longer maintained or available. The reasons are different. For example DBMS FoundationDB, supporting key/value, document, and object models, has been in 2015 acquired by Apple [Panzarino 2015] and it is no longer offering downloads. Similarly, Akiban Server which has the ability to treat groups of tables as objects and access them as JSON documents via SQL [The 451 Group 2013] was acquired by FoundationDB [Darrow 2013] in 2013.

[39] http://www.sap.com/product/technology-platform/hana.html
[40] https://infosys.uni-saarland.de/projects/octopusdb.php
[41] http://www.nuodb.com/
[42] http://redis.io/
[43] http://redismodules.com/
[44] http://www.aerospike.com/

5. CHALLENGES AND OPEN PROBLEMS

In this section, we show a compiled list of research challenges and open problems. We classify them into the following four categories: (1) multi-model query processing and optimization, (2) multi-model schema design and optimization, (3) multi-model evolution, and (4) multi-model extensibility.

—- ***Multi-model query processing and optimization***. Despite ORDBMSs are capable of storing data with various formats (models), they do not provide a cross-model data processing language, inter-model compilation or respective multi-model query optimization. In contrast, a multi-model database attempts to embrace this challenge by developing a unified query language to accommodate all the supported data models. As mentioned in the previous sections, there exist proposals of multi-model query languages. For example, AQL provided by ArangoDB enables one to access both graph and document data. However, the existing query languages are immature, and it is still an open challenge to develop a full-fledged query language for multi-model data.

A closely related problem is a proposal of an approach for identification of the optimal query plan for efficient evaluation of a given cross-model query [Lu 2017; Zhang et al. 2018]. *Wavelets* and *histograms* enable one to exploit the knowledge of distribution of data and thus optimize query evaluation strategies. However, the current techniques (e.g. [Alway and Nica 2016]) are developed for RDBMSs having a fixed relational schema, whereas multi-model DBMSs support both flexible and diverse schema. Thus, new dynamic techniques should be developed capable of adaptation to schema changes.

Currently the single-model DBMSs usually build a separate domain-specific index for different domains. Cross-domain queries are then evaluated by (1) separating index searches specifically for the individual domain, and (2) integrating the partial results to find all solutions. In the multi-model world we can use this approach too. For each of the models there exist verified types of indices, such as *B-tree* and *B+-tree* for relational data, *TreePi* [Zhang et al. 2007] and *gIndex* [Yan et al. 2004] for graph data, or *XB-tree* [Bruno et al. 2002] for hierarchical XML data. However, the efficiency of such approach is questionable. A natural hypothesis is that a universal index comprising various data models should quite probably be a better solution.

In addition, the cloud-based distributed technologies are going forward. Cloud data can be very diverse, including text, streaming data, unstructured and semi-structured data. And cloud users and developers may be in high numbers, but not DBMS experts. Therefore, one challenge is to extend the technology of distributed database management and parallel database programming to fulfill the requirement of the scalability, simplicity and flexibility of the cloud-based multi-model data management.

—- ***Multi-model schema design and optimization***. A good design of the database schema is a critical part influencing many aspects, such as efficiency of query processing, application extensibility etc. There are critical decisions about both the physical and logical schema of the data. For example, as shown in [Scherzinger et al. 2013] for the case of key/value stores, a naive schema design will result in 20–35% of database transactions failing for a certain workload, whereas this problem can be alleviated through the design of an appropriate schema. A similar paper [Mior 2014] provides a cost-based approach to schema optimization in column stores. Contrary to relational databases, NoSQL databases usually use significantly denormalized physical schema which requires additional space. Hence, in the world of multi-model systems we encounter contradictory requirements for the distinct models and thus it calls for a new solution for multi-model schema design to balance and trade-off the diverse requirement of multi-model data.

Even the question of existence of a schema differs significantly – traditional relational databases are based on existence of a pre-defined schema, whereas NoSQL databases are based on the assumption of schemalessness. A possible solution may find an inspiration, e.g., in the proposal of the *NoSQL AbstractModel* (NoAM) [Bugiotti et al. 2014], an abstract data model for NoSQL databases that specifies a system-independent data representation. However, the proposal covers only aggregate-oriented NoSQL databases (i.e., key/value, column, and document).

A closely related problem of schema inference from a sample set of data instances is another open issue in the multi-model context. There exists a number of approaches dealing with inference of, e.g., JSON [Baazizi et al. 2017] or XML [Mlýnková and Necaský 2013] schemas. Recently there have appeared approaches inferring a schema for NoSQL document stores [Gallinucci et al. 2018a], or in general for aggregate-oriented databases [Sevilla Ruiz et al. 2015; Chillón et al. 2017]. There are even methods which identify aggregation hierarchies in RDF data [Gallinucci et al. 2018b]. However, in the world of multi-model data we need to infer also references between the distinct models. In addition, the inference approaches may benefit from information extracted from related data with distinct models.

—- ***Multi-model evolution***. In general, it is a difficult task to efficiently manage data schema evolution and the propagation of the changes to the relevant portions in a database system, such as data instances, queries, indices, or even storage strategies. In some smaller applications a company can rely on a skilled database administrator to manage the data evolution and to propagate the modification to other impacted parts manually. But in most cases, it is a complicated and error-prone job.

In the context of multi-model databases, this task is more subtle and difficult. We can distinguish *intra-model* and *inter-model* changes. In the former case we can reuse the existing approaches for single models. In the latter case, however, they cannot be straightforwardly applied. The state-of-the-art solutions [Polak et al. 2015], using the classical Model-Driven Architecture, deal with multiple data models which represent distinct and overlapping views of a common model of the considered reality via which a change can be propagated to all affected parts. Then the change propagation can be solved within particular data models separately. In the case of multi-model databases the distinct models cover separate parts of the reality which are interconnected using references, foreign keys, or similar entities. Hence the evolution management has to be solved across all the supported data models. In addition, the challenge of query rewrite [Curino et al. 2008; Manousis et al. 2013], i.e. propagation of changes to queries, also becomes more complex in case of inter-model changes which require changes in data access constructs.

—- ***Multi-model extensibility***. The last but not least open problem is the challenge of *model extensibility*, which can be considered in several scopes. First, we may consider *intra-model* extensibility which means extending one of the models with new constructs, e.g., extending the XML model with the support for the query on IDs and IDREF(S). Second, we may consider *inter-model* extensibility which adds new constructs expressing relations between the models, e.g. the ability to express a CHECK constraint from the relational model across both relational and XML data. And third, we can provide *extra-model* extensibility which involves adding a whole new model, together with respective data and query, e.g. adding time series data with the support of time series analysis.

6. CONCLUSION

The specific V-characteristics of Big Data bring many challenging tasks to be solved to provide efficient and effective management of the data. In this survey we focus on

the variety challenge of Big Data which requires concurrent storage and management of distinct data types and formats. Multi-model DBMSs analyzed in this survey correspond to the "*one size fits a bunch*" viewpoint [Alsubaiee et al. 2014]. Considering the Gartner survey [Feinberg et al. 2015] which shows the high near-future representation and the existing large amount of multi-model systems, this approach has demonstrated its meaningfulness and practical applicability. On the other hand, this survey also shows that there still remains a long journey towards a mature and robust multi-model DBMS comparable with verified solutions from the world of relational databases. One intention of this survey is to promote research and industrial efforts to catch the opportunities and address challenges in developing a full-fledged multi-model database system.

ACKNOWLEDGMENTS

In part, this work was funded by the MŠMT ČR project SVV 260451 (I. Holubová) and Finnish Academy Project 310321 (J. Lu)

REFERENCES

Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, and Panos Kalnis. 2017. A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data. *Proc. VLDB Endow.* 10, 13 (Sept. 2017), 2049–2060.

Naoual El Aboudi and Laila Benhlima. 2018. Big Data Management for Healthcare Systems: Architecture, Requirements, and Implementation. *Adv. Bioinformatics* 2018 (2018), 4059018:1–4059018:10.

Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. 2009. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB* 2, 1 (2009), 922–933.

Aerospike, Inc. 2012. Aerospike Acquires AlchemyDB NewSQL Database. (2012). http://www.aerospike.com/uncategorized/aerospike-acquires-alchemydb-newsql-database-to-build-on-predictable-speed-and-web-scale-data-management-of-aerospike-real-time-nosql-database-2/

Divy Agrawal, Sanjay Chawla, Bertty Contreras-Rojas, Ahmed K. Elmagarmid, Yasser Idris, Zoi Kaoudi, Sebastian Kruse, Ji Lucas, Essam Mansour, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, Saravanan Thirumuruganathan, and Anis Troudi. 2018. RHEEM: Enabling Cross-Platform Data Processing - May The Big Data Be With You! -. *PVLDB* 11, 11 (2018), 1414–1427.

Rana Alotaibi, Damian Bursztyn, Alin Deutsch, Ioana Manolescu, and Stamatis Zampetakis. 2019. Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue. In *SIGMOD*.

Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: A Scalable, Open Source BDMS. *PVLDB* 7, 14 (2014), 1905–1916.

Kaleb Alway and Anisoara Nica. 2016. Constructing Join Histograms from Histograms with q-error Guarantees. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 2245–2246.

Amazon. 2017. Amazon DynamoDB – Developer Guide (API Version 2012-08-10). (2017). http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html

Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5, Article 68 (Sept. 2017), 40 pages.

Renzo Angles and Claudio Gutiérrez. 2008. Survey of graph database models. *ACM Comput. Surv.* 40, 1 (2008), 1:1–1:39.

ArangoDB. 2016. Three major NoSQL data models in one open-source database. https://www.arangodb.com/. (2016).

ArangoDB. 2017. ArangoDB v3.3 Documentation – Data Models and Modeling. (2017). https://docs.arangodb.com/3.3/Manual/DataModeling/

Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *ACM SIGMOD*. 1383–1394.

Abdelkader Baaziz and Luc Quoniam. 2014. How to use Big Data technologies to optimize operations in Upstream Petroleum Industry. *CoRR* abs/1412.0755 (2014).

Mohamed Amine Baazizi, Houssem Ben Lahmar, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2017. Schema Inference for Massive JSON Datasets. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*. 222–233.

Basho Technologies, Inc. 2014. Riak doc – Implementing a Document Store (version 2.2.0). (2014). http://docs.basho.com/riak/kv/2.2.0/developing/usage/document-store/

Basho Technologies, Inc. 2017. Riak doc – Using Search (version 2.2.3). (2017). https://docs.basho.com/riak/kv/2.2.3/developing/usage/search/

Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. 2013. Building an Efficient RDF Store over a Relational Database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 121–132.

Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoč. 2017. JSON: Data Model, Query Languages and Schema Specification. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '17)*. ACM, New York, NY, USA, 123–135. DOI: http://dx.doi.org/10.1145/3034786.3056120

Alysha Brown. 2016. Welcome to our eleventh major edition of c-treeACE database technology! (2016). https://www.faircom.com/insights/ctreeace-v11-announcement

Nicolas Bruno, Nick Koudas, and Divesh Srivastava. 2002. Holistic twig joins: optimal XML pattern matching. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*. 310–321.

Francesca Bugiotti, Damian Bursztyn, Alin Deutsch, Ioana Ileana, and Ioana Manolescu. 2015. Invisible Glue: Scalable Self-Tunning Multi-Stores. In *CIDR*.

Francesca Bugiotti, Luca Cabibbo, Paolo Atzeni, and Riccardo Torlone. 2014. Database Design for NoSQL Systems. In *Conceptual Modeling*, Eric Yu, Gillian Dobbie, Matthias Jarke, and Sandeep Purao (Eds.). Springer International Publishing, Cham, 223–231.

Rick Cattell. 2011. Scalable SQL and NoSQL Data Stores. *SIGMOD Rec.* 39, 4 (May 2011), 12–27.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (June 2008), 26 pages. DOI: http://dx.doi.org/10.1145/1365815.1365816

Alberto Hernández Chillón, Severino Feliciano Morales, Diego Sevilla, and Jesús García Molina. 2017. Exploring the Visualization of Schemas for Aggregate-Oriented NoSQL Databases. In *ER Forum/Demos (CEUR Workshop Proceedings)*, Vol. 1979. CEUR-WS.org, 72–85.

Crate.io. 2017. Crate.io – Storage and Consistency v. 1.0.1. (2017). https://crate.io/docs/crate/guide/en/latest/architecture/storage-consistency.html

Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. 2008. Graceful Database Schema Evolution: The PRISM Workbench. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 761–772. DOI: http://dx.doi.org/10.14778/1453856.1453939

James Curtis. 2016. With Modules, Redis Labs turns Redis into a multi-model database. (2016). https://451research.com/report-short?entityId=89003

Barb Darrow. 2013. FoundationDB Buys Akiban to Wed NoSQL and SQL Worlds. (2013). https://gigaom.com/2013/07/17/foundationdb-buys-akiban-to-wed-nosql-and-sql-worlds/

DataStax, Inc. 2013. Improving Secondary Index Write Performance in 1.2. (2013). http://www.datastax.com/dev/blog/improving-secondary-index-write-performance-in-1-2

DataStax, Inc. 2015. What's New in Cassandra 2.2: JSON Support. (2015). http://www.datastax.com/dev/blog/whats-new-in-cassandra-2-2-json-support

Ali Davoudian, Liu Chen, and Mengchi Liu. 2018. A Survey on NoSQL Stores. *ACM Comput. Surv.* 51, 2 (2018), 40:1–40:43.

David J. DeWitt, Alan Halverson, Rimma V. Nehme, Srinath Shankar, Josep Aguilar-Saborit, Artin Avanes, Miro Flasza, and Jim Gramling. 2013. Split query processing in polybase. In *SIGMOD*. 1255–1266.

Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, Magdalena Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stanley B. Zdonik. 2015. The BigDAWG Polystore System. *SIGMOD Record* 44, 2 (2015), 11–16.

Ecma International. 2013. ECMA-404 The JSON Data Interchange Standard. (2013). http://www.json.org/

Elmore et al. 2015. A Demonstration of the BigDAWG Polystore System. *PVLDB* 8, 12 (2015), 1908–1911.

Radwa Elshawi, Omar Batarfi, Ayman Fayoumi, Ahmed Barnawi, and Sherif Sakr. 2015. Big Graph Processing Systems: State-of-the-Art and Open Challenges. In *Big Data Computing Service and Applications (BigDataService), 2015 IEEE First International Conference on*. 24–33. DOI:http://dx.doi.org/10.1109/BigDataService.2015.11

Donald Feinberg, Merv Adrian, Nick Heudecker, Adam M. Ronthal, and Terilyn Palanca. 12 October 2015. Gartner Magic Quadrant for Operational Database Management Systems. (12 October 2015).

Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 1433–1445. DOI:http://dx.doi.org/10.1145/3183713.3190657

Enrico Gallinucci, Matteo Golfarelli, and Stefano Rizzi. 2018a. Schema profiling of document-oriented databases. *Inf. Syst.* 75 (2018), 13–25. DOI:http://dx.doi.org/10.1016/j.is.2018.02.007

Enrico Gallinucci, Matteo Golfarelli, Stefano Rizzi, Alberto Abelló, and Oscar Romero. 2018b. Interactive multidimensional modeling of linked data for exploratory OLAP. *Inf. Syst.* 77 (2018), 86–104.

Stefan Goessner. 2007. JSONPath – XPath for JSON. (2007). https://goessner.net/articles/JsonPath/

Michael Hammer and Dennis McLeod. 1979. *On Database Management System Architecture*. Mass. Inst. of Technology, Laboratory for Computer Science.

Adam Hems, Adil Soofi, and Ernie Perez. 2013. How innovative oil and gas companies are using big data to outmaneuver the competition. http://goo.gl/2IF6mz. (2013).

Hewlett Packard Enterprise. 2018. Using Flex Tables – Vertica Analytics Platform, Version 9.0.x Documentation. (2018). https://my.vertica.com/docs/9.0.x/HTML/index.htm#Authoring/FlexTables/FlexTableHandbook.htm

Irena Holubova and Martin Necasky. 2009. Current Support of XML by the "Big Three". In *Proceedings of the 4th International Conference XML Prague*. 251–268.

Jer-Wen Huang. 1994. MultiBase: a heterogeneous multidatabase management system. In *COMPSAC*. 332–339.

IBM Knowledge Center. 2017a. *DB2 11.1 for Linux, UNIX, and Windows – Querying XML Data*. http://www.ibm.com/support/knowledgecenter/SSEPGG_11.1.0/com.ibm.db2.luw.xml.doc/doc/c0023895.html

IBM Knowledge Center. 2017b. *DB2 11.1 for Linux, UNIX, and Windows – XML Data Type*. http://www.ibm.com/support/knowledgecenter/SSEPGG_11.1.0/com.ibm.db2.luw.xml.doc/doc/c0023366.html

InterSystems. 2015. Using Caché SQL – Defining and Building Indices. (2015). http://docs.intersystems.com/latest/csp/docbook/DocBook.UI.Page.cls?KEY=GSQLOPT_indices

InterSystems. 2016. Introducing the Document Data Model in Caché 2016.2. (2016). https://community.intersystems.com/post/introducing-document-data-model-cach\%C3\%A9-20162

InterSystems. 2017. Caché SQL Reference. (2017). http://docs.intersystems.com/latest/csp/docbook/DocBook.UI.Page.cls?KEY=RSQL

ISO. 2008. ISO/IEC 9075-1:2008 Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework). (2008). http://www.iso.org/iso/catalogue_detail.htm?csnumber=45498

jsoniq.org. 2013. JSONiq: The JSON Query Language. (2013). http://jsoniq.org/

Mat Keep. 2011. MySQL Cluster 7.2 (DMR2): NoSQL, Key/Value, Memcached. (2011). https://blogs.oracle.com/MySQL/entry/mysql_cluster_7_2_dmr2

Rado Kotorov. 2003. Customer relationship management: strategic lessons and future directions. *Business Process Management Journal* 9, 5 (2003), 566–571.

Feng Li, Beng Chin Ooi, M. Tamer Özsu, and Sai Wu. 2014. Distributed Data Management Using MapReduce. *ACM Comput. Surv.* 46, 3, Article 31 (Jan. 2014), 42 pages. DOI:http://dx.doi.org/10.1145/2503009

Harold Lim, Yuzhang Han, and Shivnath Babu. 2013. How to Fit when No One Size Fits. In *CIDR*. www.cidrdb.org.

Jiaheng Lu. 2017. Towards Benchmarking Multi-Model Databases. In *CIDR*.

Jiaheng Lu and Irena Holubová. 2017. Multi-model Data Management: What's New and What's Next?. In *EDBT*. 602–605.

Jiaheng Lu, Irena Holubová, and Bogdan Cautis. 2018a. Multi-model Databases and Tightly Integrated Polystores: Current Practices, Comparisons, and Open Challenges. In *CIKM*. 2301–2302.

Jiaheng Lu, Zhen Hua Liu, Pengfei Xu, and Chao Zhang. 2018b. UDBMS: Road to Unification for Multi-model Data Management. In *Advances in Conceptual Modeling - ER Workshops*. 285–294.

Petros Manousis, Panos Vassiliadis, and George Papastefanatos. 2013. Automating the Adaptation of Evolving Data-Intensive Ecosystems. In *ER*. 182–196.

MarkLogic Corporation. 2017a. Application Developer's Guide – Chapter 20 Working With JSON. (2017). https://docs.marklogic.com/guide/app-dev/json

MarkLogic Corporation. 2017b. Concepts Guide – Chapter 3 Indexing in MarkLogic. (2017). https://docs.marklogic.com/guide/concepts/indexing

Microsoft. 2016. LINQ (Language Integrated Query). (2016). https://docs.microsoft.com/en-us/dotnet/standard/using-linq

Microsoft. 2017a. Azure Cosmos DB SQL syntax reference. (2017). https://docs.microsoft.com/en-us/azure/cosmos-db/sql-api-sql-query-reference

Microsoft. 2017b. PolyBase Guide. (2017). https://msdn.microsoft.com/en-us/library/mt143171.aspx

Microsoft. 2017c. XML Data (SQL Server). (2017). https://docs.microsoft.com/en-us/sql/relational-databases/xml/xml-data-sql-server

Michael J. Mior. 2014. Automated Schema Design for NoSQL Databases. In *Proceedings of the 2014 SIGMOD PhD Symposium (SIGMOD'14 PhD Symposium)*. ACM, New York, NY, USA, 41–45. DOI:http://dx.doi.org/10.1145/2602622.2602624

Irena Mlýnková and Martin Necaský. 2013. Heuristic Methods for Inference of XML Schemas: Lessons Learned and Open Issues. *Informatica, Lith. Acad. Sci.* 24, 4 (2013), 577–602.

MongoDB, Inc. 2017. MongoDB Manual – Indexes. (2017). https://docs.mongodb.com/manual/indexes/

NuoDB. 2013. Multi-model databases: neither fish nor fowl but maybe a jigsaw puzzle? (2013). http://www.nuodb.com/blog/multi-model-databases-neither-fish-nor-fowl-maybe-jigsaw-puzzle

Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. 2004. ORDPATHs: Insert-friendly XML Node Labels. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*. ACM, New York, NY, USA, 903–908. DOI:http://dx.doi.org/10.1145/1007568.1007686

Patrick E. O'Neil. 1992. The SB-tree: An Index-sequential Structure for High-performance Sequential Access. *Acta Inf.* 29, 3 (June 1992), 241–265. DOI:http://dx.doi.org/10.1007/BF01185680

Oracle. 2014. Oracle NoSQL Database Compared to HBase. (2014). http://www.oracle.com/technetwork/products/nosqldb/documentation/nosql-vs-hbase-1961722.pdf

Oracle. 2017. JSON Developer's Guide. (2017). https://docs.oracle.com/en/database/oracle/oracle-database/12.2/adjsn/toc.htm

OrientDB. 2016. a 2nd Generation Distributed Graph Database. http://orientdb.com/orientdb/. (2016).

OrientDB. 2017a. OrientDB Manual – version 3.0 – Multi-Model Database. (2017). https://orientdb.com/docs/3.0.x/datamodeling/Tutorial-Document-and-graph-model.html

OrientDB. 2017b. OrientDB Manual - version 3.0.x – SQL Reference. (2017). https://orientdb.com/docs/3.0.x/sql/

M. Tamer Özsu. 2016. A Survey of RDF Data Management Systems. *Front. Comput. Sci.* 10, 3 (June 2016), 418–432. DOI:http://dx.doi.org/10.1007/s11704-016-5554-y

Matthew Panzarino. 2015. Apple Acquires Durable Database Company FoundationDB. (2015). https://techcrunch.com/2015/03/24/apple-acquires-durable-database-company-foundationdb/

Ewa Płuciennik and Kamil Zgorzałek. 2017. *The Multi-model Databases – A Review*. Springer International Publishing, Cham, 141–152. DOI:http://dx.doi.org/10.1007/978-3-319-58274-0_12

Marek Polak, Martin Chytil, Karel Jakubec, Vladimir Kudelas, Peter Pijak, Martin Necasky, and Irena Holubova. 2015. Data and Query Adaptation Using DaemonX. *COMPUTING AND INFORMATICS* 34, 1 (2015). http://www.cai.sk/ojs/index.php/cai/article/view/2040/688

Jovan Popovic. 2015. JSON Support in SQL Server 2016. (2015). https://blogs.msdn.microsoft.com/jocapc/2015/05/16/json-support-in-sql-server-2016/

Marko A. Rodriguez. 2015. The Gremlin Graph Traversal Machine and Language (Invited Talk). In *Proceedings of the 15th Symposium on Database Programming Languages (DBPL 2015)*. ACM, New York, NY, USA, 1–10. DOI:http://dx.doi.org/10.1145/2815072.2815073

Sherif Sakr and Ghazi Al-Naymat. 2010. Relational Processing of RDF Queries: A Survey. *SIGMOD Rec.* 38, 4 (June 2010), 23–28. DOI:http://dx.doi.org/10.1145/1815948.1815953

Sherif Sakr, Fuad Bajaber, Ahmed Barnawi, Abdulrahman Altalhi, Radwa Elshawi, and Omar Batarfi. 2015. Big Data Processing Systems: State-of-the-Art and Open Challenges. In *Cloud Computing (ICCC), 2015 International Conference on*. 1–8. DOI:http://dx.doi.org/10.1109/CLOUDCOMP.2015.7149633

Sherif Sakr, Anna Liu, and Ayman G. Fayoumi. 2013. The Family of Mapreduce and Large-scale Data Processing Systems. *ACM Comput. Surv.* 46, 1, Article 11 (July 2013), 44 pages. DOI:http://dx.doi.org/10.1145/2522968.2522979

Sherif Sakr and Eric Pardede (Eds.). 2011. *Graph Data Management: Techniques and Applications*. IGI Global. DOI: http://dx.doi.org/10.4018/978-1-61350-053-8

Cynthia M. Saracco, Don Chamberlin, and Rav Ahuja. 2006. *DB2 9: pureXML Overview and Fast Start*. RedBooks. http://www.redbooks.ibm.com/abstracts/sg247298.html?Open

Stefanie Scherzinger, Eduardo Cunha De Almeida, Felipe Ickert, and Marcos Didonet Del Fabro. 2013. On the Necessity of Model Checking NoSQL Database Schemas when Building SaaS Applications. In *Proceedings of the 2013 International Workshop on Testing the Cloud (TTC 2013)*. ACM, New York, NY, USA, 1–6. DOI: http://dx.doi.org/10.1145/2489295.2489297

Diego Sevilla Ruiz, Severino Feliciano Morales, and Jesús García Molina. 2015. Inferring Versioned Schemas from NoSQL Databases and Its Applications. In *Conceptual Modeling*, Paul Johannesson, Mong Li Lee, Stephen W. Liddle, Andreas L. Opdahl, and Óscar Pastor López (Eds.). Springer International Publishing, Cham, 467–480.

Dharma Shukla, Shireesh Thota, Karthik Raman, Madhan Gajendran, Ankur Shah, Sergii Ziuzin, Krishnan Sundaram, Miguel Gonzalez Guajardo, Anna Wawrzyniak, Samer Boshra, Renato Ferreira, Mohamed Nassar, Michael Koltachev, Ji Huang, Sudipta Sengupta, Justin Levandoski, and David Lomet. 2015. Schema-agnostic Indexing with Azure DocumentDB. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1668–1679.

John Miles Smith, Philip A. Bernstein, Umeshwar Dayal, Nathan Goodman, Terry Landers, Ken W. T. Lin, and Eugene Wong. 1981. Multibase: Integrating Heterogeneous Distributed Database Systems. In *Proceedings of the May 4-7, 1981, National Computer Conference (AFIPS '81)*. ACM, New York, NY, USA, 487–499. DOI: http://dx.doi.org/10.1145/1500412.1500483

Daniel Tahara, Thaddeus Diamond, and Daniel J. Abadi. 2014. Sinew: A SQL System for Multi-structured Data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 815–826. DOI: http://dx.doi.org/10.1145/2588555.2612183

Ran Tan, Rada Chirkova, Vijay Gadepally, and Timothy G. Mattson. 2017. Enabling query processing across heterogeneous data models: A survey. In *BigData*. 3211–3220.

The 451 Group. 2013. Neither Fish Nor Fowl: the Rise of Multi-Model Databases. (2013). https://blogs.the451group.com/information_management/2013/02/08/neither-fish-nor-fowl/

The Apache Software Foundation. 2017. The Cassandra Query Language (CQL). (2017). http://cassandra.apache.org/doc/latest/cql/

W3C. 2008. Extensible Markup Language (XML) 1.0 (Fifth Edition). (2008). http://www.w3.org/TR/xml/

W3C. 2013. SPARQL 1.1 Overview. (2013). http://www.w3.org/TR/sparql11-overview/

W3C. 2014. RDF 1.1 Concepts and Abstract Syntax. (2014). http://www.w3.org/TR/rdf11-concepts/

W3C. 2015a. XML Path Language (XPath) Version 1.0. (2015). http://www.w3.org/TR/xpath/

W3C. 2015b. XQuery 1.0: An XML Query Language (Second Edition). (2015). http://www.w3.org/TR/xquery/

W3C. 2018a. LargeTripleStores. (2018). https://www.w3.org/wiki/LargeTripleStores

W3C. 2018b. RdfStoreBenchmarking. (2018). https://www.w3.org/wiki/RdfStoreBenchmarking

Jingjing Wang, Tobin Baker, Magdalena Balazinska, Daniel Halperin, Brandon Haynes, Bill Howe, Dylan Hutchison, Shrainik Jain, Ryan Maas, Parmita Mehta, Dominik Moritz, Brandon Myers, Jennifer Ortiz, Dan Suciu, Andrew Whitaker, and Shengliang Xu. 2017. The Myria Big Data Management and Analytics System and Cloud Services. In *CIDR*.

Marcin Wylot, Manfred Hauswirth, Philippe Cudré-Mauroux, and Sherif Sakr. 2018. RDF Data Storage and Query Processing Schemes: A Survey. *ACM Comput. Surv.* 51, 4, Article 84 (Sept. 2018), 36 pages. DOI: http://dx.doi.org/10.1145/3177850

Xifeng Yan, Philip S. Yu, and Jiawei Han. 2004. Graph Indexing: A Frequent Structure-based Approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*. 335–346. DOI: http://dx.doi.org/10.1145/1007568.1007607

Chao Zhang, Jiaheng Lu, Pengfei Xu, and Yuxing Chen. 2018. UniBench: A Benchmark for Multi-model Database Management Systems. In *TPCTC*. 7–23.

Shijie Zhang, Meng Hu, and Jiong Yang. 2007. TreePi: A Novel Graph Indexing Method. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*. 966–975. DOI: http://dx.doi.org/10.1109/ICDE.2007.368955

APPENDIX A. The Top 5 DBMSs in the Particular Classes

To provide a broader context, in this appendix we overview the top 5 DBMSs[45] in the particular classes defined in Table I. As we can see in Table IX, where the multi-model

[45] https://db-engines.com/en/ranking [14.2. 2019]

DBMSs are in bold, most relational databases can support multiple models. On the other hand, two popular column stores, HBase[46] and MS Azure Table Storage[47], are not multi-model. The main features of HBase are based on ideas proposed in the Google BigTable [Chang et al. 2008], whereas it is a part of Apache Hadoop and based on the usage of HDFS. The data can be processed using MapReduce or an SQL extension provided by separate Apache projects. Contrary to (multi-model) MS Cosmos DB, MS Azure Table Storage is a pure column store with an emphasis on high throughput. The tables are schemaless and can be queried using MS LINQ [Microsoft 2016].

Table IX. The top 5 DBMSs in the particular classes according to DB-Engines Ranking

Class	Top 5 DBMSs
Relational	**Oracle DB**, **MySQL**, **MS SQL Server**, **PostgreSQL**, **IBM DB2**
Column	**Cassandra**, HBase, **Cosmos DB**, **Datastax Enterprise**, MS Azure Table Storage
Key/value	**Redis**, **DynamoDB**, Memcached, **Cosmos DB**, Hazelcast
Document	**MongoDB**, **DynamoDB**, **Couchbase**, **Cosmos DB**, CouchDB
Graph	Neo4j, **CosmosDB**, **Datastax Enterprise**, **OrientDB**, **ArangoDB**

Two key/value DBMSs are single model databases[48]. Memcached[49] is an in-memory store which was originally intended and is currently often used by other systems for caching data in RAM to speed up their processing. It can be described as a large hash table where the least-recetly used data are purged when necessary. The other representative, Hazelcast[50], is also an in-memory system where the elastically scalable data grid provides similar functionality and advantages. A popular single-model document store is Apache CouchDB[51]. It stores data in a JSON format, whereas a document can also have a set of binary attachments files. Querying of data is implemented using *views* which are generated on-demand to process data using MapReduce.

Finally, in the world of graph databases there is surprisingly one exception represented by the most popular DBMS of this kind – Neo4j[52]. Its logical model involves labeled nodes and edges which can have an arbitrary number of attributes. The graph data can be queried using the standard graph traversal language Gremlin, Java graph traversal interface, or own SQL-like graph query language Cypher [Francis et al. 2018]. Internally the data are stored in the form of adjacency lists, where adjoining nodes and edges point to each other. Neo4j High Availability enables a horizontally scaling read-mostly architecture.

APPENDIX B. Query Languages for Popular Data Formats

In this appendix, we overview query languages currently usually used for querying of the most popular data formats. These languages can be viewed as prospective candidates for extensions towards multiple models.

The simplest data model, i.e., key/value pairs, are usually accessed simply using methods get, put, and delete. In the world of relational data there is probably no other popular alternative to query the data than the SQL [ISO 2008]. In addition, as we have shown in Table VII, the usage of an SQL extension or an SQL-like query language is a common strategy across all types of multi-model DBMSs for various combinations of data models.

[46] https://hbase.apache.org/
[47] https://azure.microsoft.com/cs-cz/services/storage/tables/
[48] As we have discussed in Section 4.6.3, Redis will probably become a multi-model database soon.
[49] https://memcached.org/
[50] https://hazelcast.com/
[51] http://couchdb.apache.org/
[52] https://neo4j.com/

In the case of semi-structured formats we can identify two distinct situations. For XML data two W3C standards for querying, i.e., XPath [W3C 2015a] and XQuery [W3C 2015b], are currently used widely. For the JSON format there are currently several existing and quite distinct representatives (compared in detail in [Bourhis et al. 2017]), such as the JSON-based query language in MongoDB, XPath-based JSONPath [Goessner 2007], XQuery-based JSONiq [jsoniq.org 2013], or various proprietary SQL extensions. But, unfortunately, so far there is no generally acknowledged standard like in the case of XML.

Last but not least, in the world of graph data the main representatives (compared extensively in [Angles et al. 2017]) involve SPARQL [W3C 2013] primarily intended for Linked Data, Neo4j's graph query language Cypher [Francis et al. 2018], and Apache TinkerPop Gremlin [Rodriguez 2015].

APPENDIX C. Alternative Ways for Multi-Model Data Management

In this article, we have primarily surveyed single-store DBMSs to handle the challenge of multi-model data management. However, there is an alternative direction that supports different data models with multiple database engines. In this appendix, we give a brief introduction on these solutions and refer interested readers to other surveys and tutorials (e.g. [Tan et al. 2017; Lu et al. 2018a]) for the details.

The main ideas of these alternative solutions are to package together multiple query engines and combine different specialized stores, each with distinct (native) data model and different language and capabilities. Then the users rely on the middle-ware layer to process queries and data from different sources. [Tan et al. 2017] classify the existing solutions with four different types of systems as defined below:

— Federated system: multiple homogeneous data stores and one query interface.
— Polyglot system: multiple homogeneous data stores and multiple query interfaces.
— Multistore system: multiple heterogeneous data stores and one query interface.
— Polystore system: multiple heterogeneous data stores and multiple query interfaces.

First, federated systems were thoroughly researched during the periods of 1980s and 1990s. Their main strategy is to leverage different databases to store various models of data and then develop a middleware (called *mediator*) to integrate them together to answer queries. For example, one well-known system Multibase [Huang 1994] leverages a global schema and a single query interface. In order to process queries, the system decomposes the query to multiple local sub-queries based on the global schema and local schemata.

Second, polyglot systems address the need to handle complex data flows in cloud environment and distributed file systems, where the users' requests can be formulated with both complicated algorithms and declarative queries. For example, a representative system Spark SQL [Armbrust et al. 2015] provides APIs to allow users to process data with both DataFrames and SQL to access a number of data sources, such as JSON, JDBC, Hive, ORC and Parquet.

Third, multistore systems provide integrated accesses to a number of data stores including HDFS, RDBMS and NoSQL databases. They have an integrated query interface to process the data. The representative systems include HadoopDB [Abouzeid et al. 2009], Estocada [Bugiotti et al. 2015; Alotaibi et al. 2019] and Polybase [DeWitt et al. 2013].

Finally, polystore systems are built on top of multiple heterogeneous data storage engines. Users can choose from a number of queries to process data which are stored in a variety of data stores. The representative systems include BigDAWG [Duggan et al. 2015], RHEEM [Agrawal et al. 2018] and Myria [Wang et al. 2017].