information

MDPI

Article

When Relational-Based Applications Go to NoSQL Databases: A Survey

Geomar A. Schreiner [1,*], Denio Duarte [2] and Ronaldo dos Santos Mello [1]

[1] Departamento de Informática e Estatística, Federal University of Santa Catarina, 88040-900 Florianópolis - SC, Brazil
[2] Campus Chapecó, Federal University of Fronteira Sul, 89815-899 Chapecó - SC, Brazil
* Correspondence: geomarschreiner@gmail.com

Received: 22 May 2019; Accepted: 12 July 2019; Published: 16 July 2019

Abstract: Several data-centric applications today produce and manipulate a large volume of data, the so-called Big Data. Traditional databases, in particular, relational databases, are not suitable for Big Data management. As a consequence, some approaches that allow the definition and manipulation of large relational data sets stored in NoSQL databases through an SQL interface have been proposed, focusing on scalability and availability. This paper presents a comparative analysis of these approaches based on an architectural classification that organizes them according to their system architectures. Our motivation is that wrapping is a relevant strategy for relational-based applications that intend to move relational data to NoSQL databases (usually maintained in the cloud). We also claim that this research area has some open issues, given that most approaches deal with only a subset of SQL operations or give support to specific target NoSQL databases. Our intention with this survey is, therefore, to contribute to the state-of-art in this research area and also provide a basis for choosing or even designing a relational-to-NoSQL data wrapping solution.

Keywords: big data; data interoperability; NoSQL databases; relational-to-NoSQL mapping

1. Introduction

Relational database systems (RDBMS) have been standing as the backbone of application systems in several domains, being the standard database solution for maintaining critical data. However, the continuous evolution of hardware, network, and software technologies, as well as the increase of digital data production, have posed new challenges for application development and database management. From the database point of view, the main challenges are related to high availability and scalability to deal with these big data-centric applications.

One of the reasons for RDBMS not coping with big data-centric applications is the need to conform to the atomicity, consistency, isolation, and durability (ACID) properties for data manipulation, which are orthogonal to the availability and scalability requirements. Indeed, the overhead to guarantee ACID transactions may be prohibitive when a large volume of data must be handled. In addition, the fixed record format of relational data also introduces a modeling and storage problem for data instances that do not respect a schema (relational databases follow the principle schema first, data later [1]). This variety of representation is a typical Big Data feature.

Additionally, RDBMS provide vertical scalability, i.e., data servers can be powered to improve performance, but such a task is usually complex and expensive [2]. On the other hand, cloud-based architectures offer a low-cost commodity infrastructure. In such a scenario, NoSQL databases (NoSQL DBs) have risen as a promising solution for big data processing with excellent scalability, since they relax the ACID properties to maximize availability [3,4].

NoSQL DBs are based on (not-relational) data models that better represent complex objects whose data instances may have heterogeneous structures and do not strictly need to respect a schema (following the principle data first, schema later [1]). In addition, they provide horizontal elasticity instead of the (limited) vertical elasticity supported by most relational databases. Horizontal elasticity guarantees a better performance for big data management since new machines can be added or removed based on the application storage needs [4]. It is important to notice that, if ACID guarantees are needed, RDBMS are still the right choice. However, if relaxed schema, high availability and horizontal elasticity are essential requirements, NoSQL DBs are a good solution.

The dichotomy between having a relational-based application and offering high scalability with a low-cost has directed some organizations to move their applications data to NoSQL DBs. Nevertheless, we have several problems related to this moving. The first one is that NoSQL and relational data models are different. NoSQL data models are flexible and commonly based on the notion of aggregates [4], allowing the definition of different data instances with possible complex structures. On the other hand, the relational data model represents only homogeneous and normalized data instances. Another problem is the database access interface: developers are used to defining and manipulating data using the SQL language, which is the standard for RDBMS. Instead, NoSQL DBs provide different access methods and access languages depending on their data model or a specific product and usually have limited (or not) support for the SQL language. As a consequence, the learning curve to start using NoSQL DBs is very steep due to these differences in terms of data representation and data accessing.

When we consider the adaptation of applications to new computational environments, the solutions can be organized into three categories [5]: (*i*) redevelopment, which rewrites the existing applications from scratch; (*ii*) wrapping, which provides a new interface to a software component, making it more easily accessible by other components; and (*iii*) migration, which moves the application to the new environment, while retaining the original systems data and functionality. The choice for one of these solutions depends on an evaluation of the costs, like the number of required changes, costs, as well as the associated risks. The first solution is more expensive since it requires a whole system reimplementation. The third one requires less effort than the first one considering that not all the system will be recoded. Instead, the second solution is the less costly one as it usually provides a faster-moving strategy. In this case, the wrapping component acts as an interface to a service that performs some processing required by an external client that does not need to know details about the implementation of the service.

Back to our specific problem of moving a relational-based application to a NoSQL-based application with a low startup cost, the purpose of this survey is to present and compare approaches that offer solutions based on the second category above, i.e., solutions that adopt the wrapping strategy to move applications from SQL to NoSQL databases. These approaches allow relational-based applications to access their data (stored in NoSQL databases) with low startup and maintenance costs since they still view and manipulate data in the relational format and, thus, SQL operations do not need to be explicitly translated to target NoSQL access methods or access languages by the application. Instead, this task is supported by the wrappers that provide the interoperability between SQL operations and NoSQL operations.

The surveyed works propose a SQL-to-NoSQL wrapping based on one of two development strategies: (*i*) a layer that receives, translates, and executes SQL operations over NoSQL databases and (*ii*) a storage engine that abstracts the storage of an RDBMS to access data in NoSQL databases, i.e., they map the logical relational data level to a physical data level in a NoSQL database. These strategies are detailed in Section 3.

1.1. Survey Scope: What Is Out

As stated before, this survey considers wrapping solutions for allowing relational-based applications to communicate with NoSQL databases. However, other approaches also deal with the widespread problem of relational data management in the cloud. We organize them into four categories:

(*i*) Cloud RDBMS; (*ii*) SQL-like approaches; (*iii*) database migration tools; and (*iv*) NoSQL-to-SQL interoperability. We briefly present them in the following:

The cloud RDBMS category comprises RDBMS that can manage data in the cloud. In this category, we have two main subcategories: (*i*) extensions of traditional RDBMS; and (*ii*) native cloud RDBMS. The first subcategory regards products that offer optional packages for such a management, like Oracle 12c [6] and PostgreSQL [7]. The second subcategory are RDBMS specially designed to manage data in the cloud, like Microsoft SQL Azure [8]. This second subcategory also comprises the NewSQL database systems. The NewSQL paradigm combines the high availability and scalability from NoSQL databases and the ACID support from relational databases [9]. They are usually in-memory RDBMS running on a cloud environment, and each node stores a slice (partition) of the data. Some examples are S-Store [10], H-Store [11], E-Store [12], Yesquel [13], Clay [14] and Rubato DB [15].

The SQL-like category regards solutions with the capability of executing operations, in an SQL-like dialect, in a set of NoSQL databases. The online analytical processing (OLAP) is the focus in this category, e.g., AsterixDB [16], SparkSQL [17], Scope [18], SQL to Flink Translator [19], Dryad [20], Pig [21], SQL++ [22,23], CloudMdsQL [24], Hive [25], MuSQLe [26], among others.

The database migration tools category comprises solutions that do not implement a system to access data. Instead, they offer a tool that helps users to migrate relational data to NoSQL databases. These solutions accomplish a map between the relational data to a specific NoSQL database, but only to migrate the data. The developer is further responsible for modifying his/her system to access the NoSQL database. Some examples are the extensions of DigiBrowser [27] and ERwin HAWK [28].

Finally, the NoSQL-to-SQL interoperability category is related to industry and academic solutions that work on the opposite direction of this survey intention, i.e., approaches that develop solutions to store and/or access NoSQL data from relational databases. Their motivation is the support of the SQL standard and transaction management provided by an RDBMS to manage complex NoSQL data, as well as flexible NoSQL schemas. Most solutions consider the mapping of JSON data to relational data. In this context, several RDBMS vendors support JSON data management, like Oracle [1,29], DB2 [30], SQL Server [31] and PostgreSQL [32]. Their capabilities usually include: (*i*) a binary storage format, like Oracle OSON and DB2 BSON; (*ii*) SQL extensions that allow the generation and retrieval of data in JSON format, like Oracle SQL/JSON; (*iii*) APIs to manipulate JSON data, like DB2 JSON API; and (*iv*) indexing strategies for JSON data. We also have some academic proposals, like Argo [33], Sinew [34] and the work of DiScala and Abadi [35]. Argo proposes a layer for mapping JSON data to a relational database and an SQL extension (Argo/SQL) for manipulating JSON data. Sinew is also a layer that manages relational views over typical NoSQL data (arbitrary documents of key-value pairs) against which SQL queries and updates can be issued. The work of DiScala and Abadi introduces an algorithm that converts JSON data into relational data. Basically, the algorithm discovers relationships among the attributes of denormalized NoSQL datasets in order to organize these attributes into relational tables.

Different from all of these categories, this survey focuses on wrapping approaches that map relational data and SQL operations to data and operations of NoSQL databases, respectively. These approaches can be considered to move relational-based applications to the NoSQL world with low startup and maintenance costs. It is also important to observe here that we consider approaches that offer at least an elementary SQL DML interface, and we do not consider SQL extensions for SQL-to-NoSQL interoperability purposes, like, for example, the SQL support for XML data and data analytics.

In addition, this survey does not point out what NoSQL database model and NoSQL database product are the most suitable to move relational data. This problem is context sensitive, having each product specific advantages and disadvantages. The best choice among them depends on the data nature, application workload, among other features.

Given the survey scope, we initially review the data models of NoSQL databases and propose a basic architectural classification to categorize the related approaches. In the following, all approaches

in the same category are detailed and compared to find out similarities and differences among them. Last but not least, we provide an overall comparative analysis to highlight tendencies and limitations. We intend to contribute to the state-of-art on this subject as well as to raise some open issues and opportunities. We also believe that all provided overview and discussion of the approaches can be useful for developers to choose the most suitable one for their big data applications on top of a relational database, and even as a basis for the design and development of new SQL-to-NoSQL solutions. To the best of our knowledge, no other survey is focusing on this scope.

1.2. Paper Organization

The rest of this paper is organized as follows. The next section gives an overview of the data models of NoSQL databases. Section 3 proposes an architectural classification to organize the related work. Section 4 details and compares approaches in a same classification. Section 5 provides an overall comparative analysis and Section 6 is dedicated to the conclusion.

2. NoSQL Background

NoSQL database is a class of DBMS conceived to fulfill some data management requirements that relational databases do not adequately satisfy. They can be defined as databases that do not adopt the relational data model and have six properties [3,36]: (*i*) horizontal scaling; (*ii*) ability to store complex data in a distributed way; (*iii*) simple access interface or protocol for data manipulation; (*iv*) relaxed/non-existent ACID support; (*v*) high availability; and (*vi*) optional and flexible schema.

NoSQL databases have independent designs, each one with schemaless capabilities as well as specific data models that support complex data. In the literature, we find out different taxonomies about the data models of the NoSQL databases [3,4,36]. In this paper, we consider the four categories of NoSQL data models defined in [4]. These categories are preferred since they are more compatible with the NoSQL foundation presented in the interoperability approaches available in the literature and considered in this survey. The NoSQL data model categories are: (*i*) key-value; (*ii*) document-oriented; (*iii*) column-oriented; and (*iv*) graph. However, we had identified related work only for the first three categories, which are described in the following.

The key-value data model is the simplest NoSQL data model. It is composed of a set of key-value pairs, being the value accessed through a unique atomic key. A value can maintain a simple or complex content, but this content cannot be queried, i.e., it is a "black-box" content. Due to it, we assume that any value in a key-value data model has an atomic domain.

The document-oriented data model specializes the key-value data model. A document encompasses a set of key-value pairs, and a unique atomic key accesses each document. However, document content is composed of a set of simple or complex attributes. A simple attribute holds an atomic value and a complex attribute holds a list of values, a set of values, or another document.

Finally, the column-oriented (or simply columnar) data model represents data properties based on a column-distributed schema. It is composed of a keyspace, a column family, a column set accessed by a unique atomic key, columns, and values [4]. This data model has been motivated by the Google Big Table, and it is also known as extensible record stores or wide column stores [3,36].

3. Architectural Classification

We propose an architectural classification for organizing the current related work based on the system architecture followed by the approaches. Figure 1 pictorially shows the proposed classification. As stated before, our classification groups the approaches into two categories: Layer and Storage Engine.

In short, all of the considered approaches map relational schemata and SQL operations to the data models and operations of the NoSQL databases, respectively. Their main intention is to combine the best of both worlds, i.e., the power of the SQL standard and the big data capabilities of NoSQL

databases, by introducing a conversion mechanism. The motivation here is to minimize migration costs for applications that access relational databases and intend to move to NoSQL databases.

The Layer category denotes approaches that support a relational layer over one or more NoSQL databases (Figure 1a). This layer acts as a middleware between a relational-based application and a NoSQL solution, being able to receive SQL operations and translate them into specific NoSQL database access methods or access languages. On the other hand, the Storage Engine category comprises approaches that modify the physical layer of an RDBMS to provide the storage of relational data in a NoSQL database (Figure 1b). Different from the Layer category, the mapping component is tightly coupled to the source RDBMS.

The next section details and compares the approaches on each one of these wrapping categories.

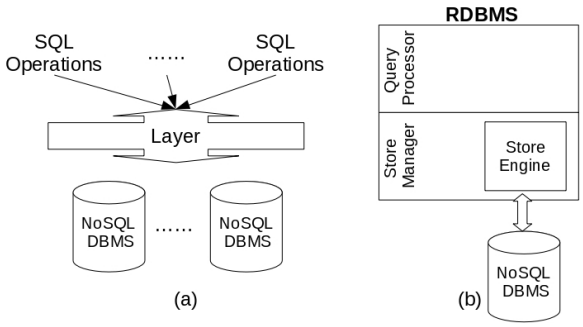

Figure 1. Overview of the proposed architectural classification for the related work.

4. SQL-to-NoSQL Wrapping Approaches

This section presents approaches that provide SQL-to-NoSQL database wrapping-based interoperability based on the architectural representation of Section 3. As a running example, we consider a subset of the *Internet Movie Database (IMDb)* [37]. The database is composed of six tables: *title_basics*, *title_crew*, *title_episode*, *title_principals*, *title_ratings*, and *name_basics* (see Figure 2). The *title_basics* table maintains basic information about 5,956,101 titles (movies, novels, series, among others). Table *title_crew* contains the directors and writers for all the *titles_basics* table (5,958,788 tuples). Information about TV and series episodes are stored in *title_episodes* table (7,453,095 tuples). Table *title_principals* contains the principal cast/crew for each titles (34,334,609 tuples). Table *title_ratings* contains the IMDb rating and votes information for titles (1,702,114 tuples). Finally, table *name_basics* stores basic information about people (e.g., actors, writers, and directors) (9,409,237 tuples).

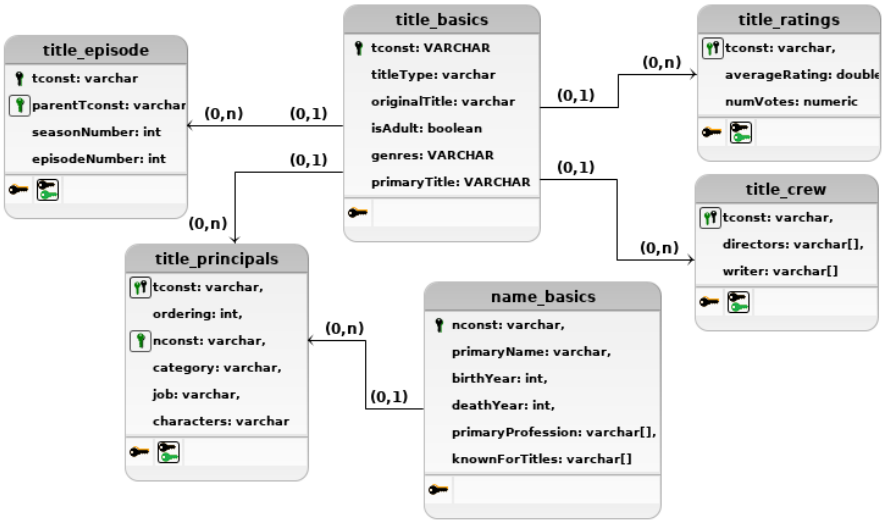

Figure 2. *IMDb* logic data model.

4.1. Layer Approach

Approaches in the *Layer* category provide a mapping infrastructure that allows systems or end users to define and/or manipulate NoSQL databases from SQL operations. There exist ten approaches in this category: the work of Zhang et al. maps SQL operations to document-oriented and column-oriented NoSQL databases [38]; *SimpleSQL* is a layer on the SimpleDB NoSQL database [39]; *JackHare* is a layer on the HBase NoSQL database [40]; *Unity* is a layer that accesses data in relational and NoSQL data sources, using MongoDB as the target NoSQL database [41]; the work of Rith et al. allows the execution of SQL operations over relational databases (trivial case) and some NoSQL databases, specifically, MongoDB and Cassandra [42]; *Apache Phoenix* [43] is a layer that also executes SQL operations over HBase; *SQLToKeyNoSQL* [44,45] is a layer that executes SQL operations over key-based NoSQL databases (column-oriented, document-oriented and key-value databases); *MIDAS 1.8* [46] proposes a layer that provides a transparent interoperability among different data management services in the cloud; *NoSQL*^{2} [47] is a middleware for running SQL commands into NoSQL databases with emphasis on the commands related to database administration; and *MSI* [48] is an integration middleware that support hybrid database architectures allowing users to query data simultaneously from relational SQL and NoSQL DBMSs. As the approaches proposed in [38,42] have no name, for the sake of simplicity, we call them, from now on, *RithWork* and *ZhangWork*, respectively.

In the following, we present some architectural issues, as well as schema mapping and join processing strategies for each approach.

4.1.1. Architectural Issues

The system architecture of the approaches presents, in general, the same anatomy. They usually comprise three modules: (*i*) *access interface*; (*ii*) *translation*; and (*iii*) *execution*. The first one is responsible for receiving and validating SQL operations. In most approaches, a subset of the main SQL Data Definition Language (DDL) and Data Manipulation Language (DML) operations is supported. *Unity*, *RithWork* and *MSI* are the only ones that do not consider DDL operations.

The translation module is responsible for identifying and mapping SQL operations and their components (e.g., tables and attributes) to corresponding operations in the target NoSQL database. The translation is usually supported by a metadata dictionary and specific mapping rules between the relational and the NoSQL data models. *SimpleSQL* translates SQL operations to SimpleDB API methods, like *get()* and *put()*, while *JackHare* translates them to *map-reduce* jobs [49] that access data in HBase. *Unity*, *Apache Phoenix* and *SQLToKeyNoSQL* additionally provide a query optimizer that determines join ordering and portions of the query plan to be executed on each considered data source. *RithWork*, *ZhangWork*, *MSI* and *NoSQL*^{2} translate the SQL operation directly to the specific data source access language through wrappers called *connectors*.

The execution module is responsible for sending the translated operation to the target NoSQL database, as well as for receiving the result set and mapping it to tuples to be output, in case of a query operation. *SimpleDB* defines correspondences between a set of documents and a set of tuples. Instead, *JackHare* and *Apache Phoenix* define correspondences between a set of key-value pairs and a set of tuples, and specific connectors support mappings in *RithWork*, *ZhangWork* and *Unity*. Currently, these three last approaches offer connectors that provide correspondences between MongoDB documents and Cassandra column families to tuples. *SQLToKeyNoSQL* and *MIDAS 1.8* consider an intermediary canonical data model that maps tuples to each specific NoSQL data model (and vice versa) through wrappers. We remark that *MIDAS 1.8* uses the canonical model proposed in *SQLToKeyNoSQL* and works only for the document-oriented data model.

4.1.2. Schema Mapping

This section details the strategies for mapping relational schemata to the data models of the target NoSQL databases. We start by the *SimpleSQL* approach. *SimpleSQL* allows the execution of SQL

operations on the SimpleDB NoSQL database. SimpleDB supports the document-oriented data model, which represents complex objects, i.e., objects with structured or multivalued attributes. Its particular data model is composed of the following concepts: *domains*, *items*, *attributes* and *values*. A domain consists of a name *dom* and a set of items it_i with the form $(dom, \{it_1, \ldots, it_n\})$. Each item, in turn, is composed of a name n and a set of attributes with the form $(n, ((key, value)_1, \ldots, (key, value)_n))$, where *key* is the attribute name (attribute key) and $value$ is the *value* of *key*. Multivalued attributes are allowed. *SimpleSQL* maps a relational database *db* into a SimpleDB domain with the same name of *db*. Each item name in SimpleDB denotes the primary key *pk* of a table $t \in db$. *SimpleSQL* also defines a special attribute called *SimpleSQL_TableName* whose value is the name of t. For example, Figure 3 shows an extract from the IMDb mapped to SimpleSQL approach. The *IMDd* is mapped to a SimpleDB domain and each tuple is mapped to an *item* (document) in SimpleDB. Considering the cardinality of each table in *IMDb*, the final cardinality of SimpleSQL is 64,813,944 items (i.e., documents).

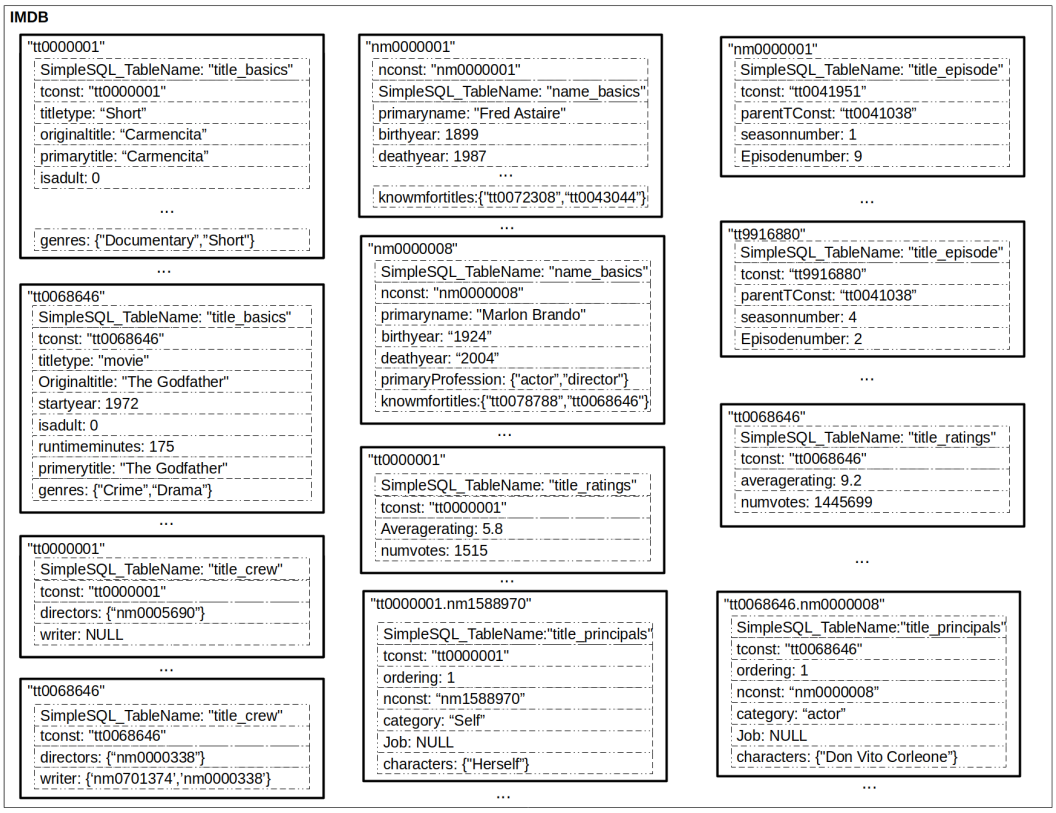

Figure 3. *IMDb* mapped to SimlpeSQL approach.

JackHare is a framework that allows the execution of SQL DDL and DML (except some aggregate functions) operations over HBase. HBase is a column-oriented NoSQL database, and it is composed of one keyspace. A keyspace has a set of HTables, and each HTable maintains a set of columns' families. Each column family holds a set of keys, being each key, in turn, composed of columns and values.

JackHare maps a relational database to the HBase data model by applying the following rules: (*i*) each relational database *db* is mapped to an HTable HT with the name of *db*; (*ii*) each table $t \in db$ is mapped to a component of the key (as detailed in the next rule); (*iii*) each row $r \in t$ is mapped to a set of key-value pairs K, being each key obtained by concatenating the primary key of r with the name of t; and (*iv*) a column $c \in r$ is mapped to a column $c_{HB} \in HT$. The name and value of c become the c_{HB} key and value, respectively. Figure 4 shows the mapping of a subset of IMDb to HBase. The database is mapped to an HTable (HTable: IMDB), and each relational table to a column family. The column family groups the attributes of a row from a given table. A row key id identifies each row. For example, the tuple that represents the movie *The Godfather* from table *title_basics* is grouped in the

title_basics column family, and its row key is `title_basics:tt0068646`. Thus, the resulting HTable is composed of 69,581,116 rowkeys pointing to the same number of rows, and six column families (one for each table).

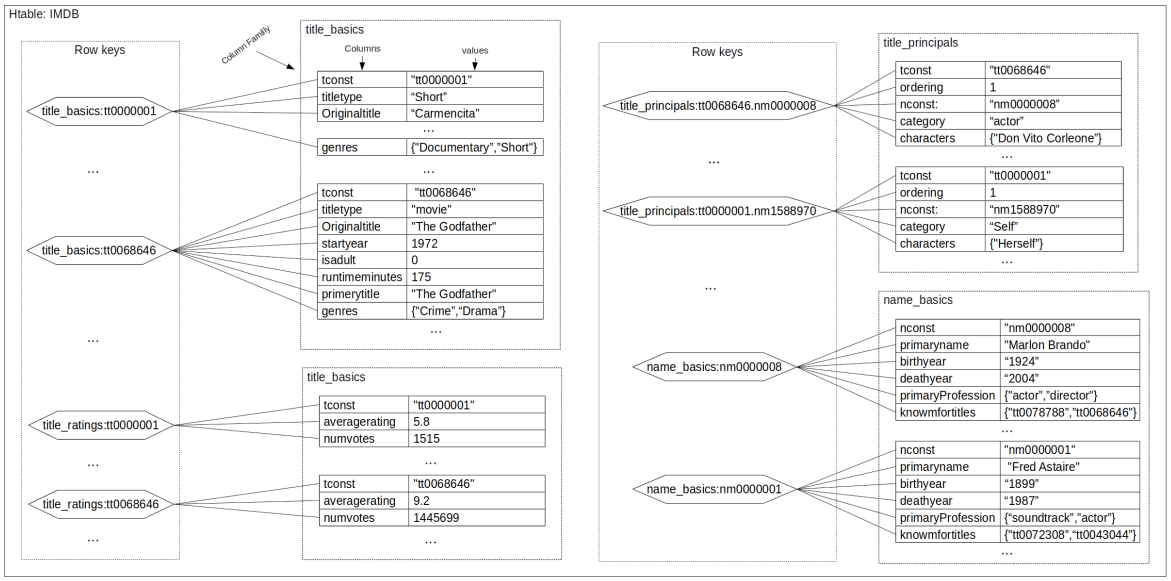

Figure 4. *IMDb* mapped to JackHare.

Like *JackHare*, *Apache Phoenix* provides access to SQL DDL and DML operations over HBase. However, *Apache Phoenix* implements a one-to-one mapping of relational tables to HTables. Each row of the relational table is mapped to a row in the HTable, and the *rowId* of an HTable row is composed of the concatenation of the primary key values of the table row. Column families of HTable are used in *Apache Phoenix* for schema versioning. Figure 5 shows the mapping of the IMDb. Notice that, for each table, one HTable is created. In addition, the total number of rowkeys generated by JackHare and Apache Phoenix are the same, but Apache Phoenix divides rowkeys into different key stets, one for each HTable.

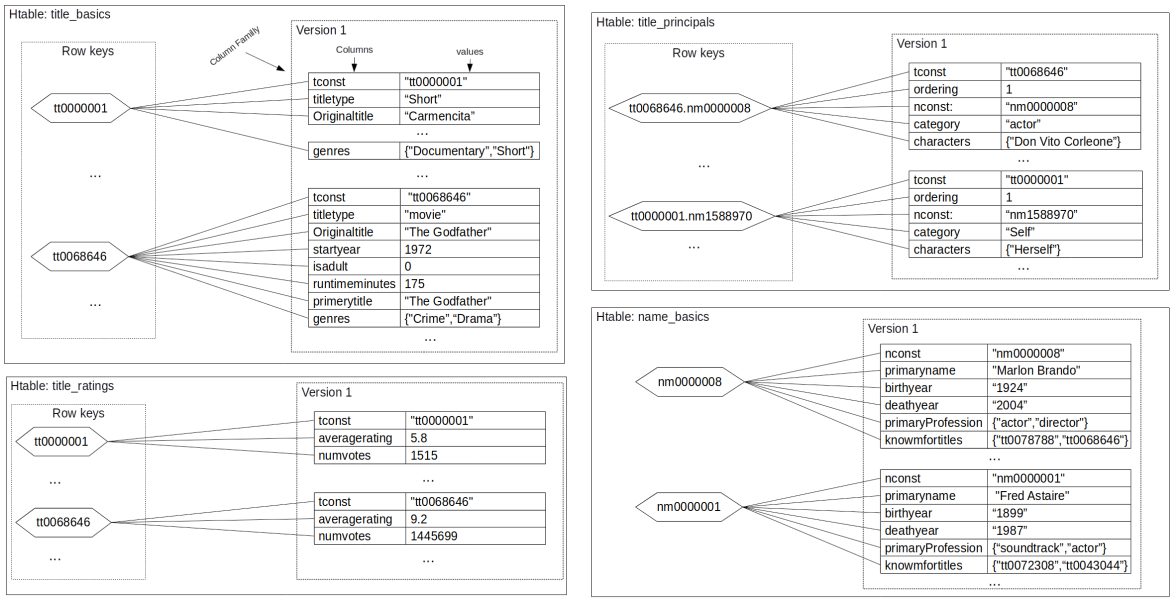

Figure 5. *IMDd* mapped to Apache Phoenix.

The *Unity* approach provides the mapping of a limited set of SQL operations over multiple data sources, including the MongoDB NoSQL database. MongoDB is also a document-oriented database whose data model is composed of a *database*, *collections* and *documents*. A collection maintains a set of documents and a document is composed of a set of key-value pairs. *Unity* maps a relational database *db* to a MongoDB database M_{db} with the same name of *db*. Each collection $M_c \in M_{db}$ maps one table $t \in db$. M_c name is the name of t. Each row $r \in t$ is a document $m_d \in M_c$. Key-value pairs represent attributes and values from r. For example, the *title_basics* table is mapped to a collection in MongoDB (Figure 6). The collection *title_basics* is composed of 5,956,101 documents since every tuple become a document. *Unity* also claims to have support for Cassandra, but mapping details are not available.

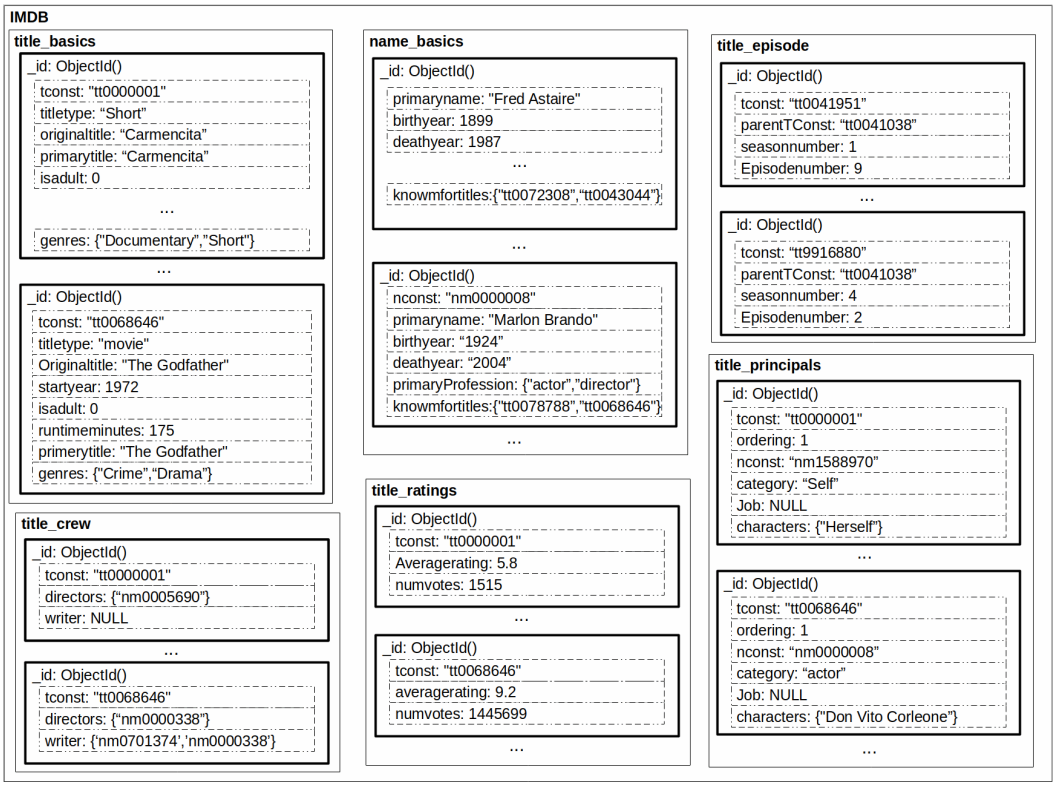

Figure 6. *IMDb* mapped to Unity.

The *SQLToKeyNoSQL* approach allows the execution of some SQL DDL and DML operations over multiple key-based NoSQL databases. To provide this kind of access, it defines a canonical model that abstracts the three key-based NoSQL data models. This canonical model is a set of keys organized hierarchically. It can be seen as a tree T where the root node holds the relational database. Figure 7 shows the *IMDb* represented in the canonical model. Besides the root, T has three levels of keys: the first level maps a relational table t, the second level maps tuples from t represented by the concatenation of primary keys of t, and the third level maps the columns of each tuple of t. The leaf nodes maintain the values of each column. A relational schema represented in the canonical model is further mapped to the target data model of a key-based NoSQL database. Figure 8 shows the canonical model mapped to a document-oriented NoSQL DB (MongoDB). The first level keys of the canonical model are mapped to document collections. The second level keys (tuple identifiers) are mapped to documents, and, finally, third level keys are mapped to attributes. Consideration of the title *Carmencita* in the canonical model (second level key *tt0000001*) is represented in the MongoDB model (Figure 8) by the document *_id: tt0000001* in *title_basics* document collection. Again, we remark that *MIDAS 1.8* is based on the same canonical model. More details about these mappings are given in Section 5.

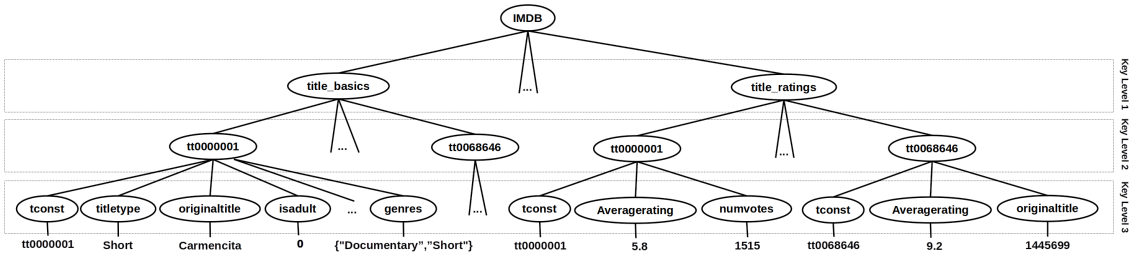

Figure 7. *IMDb* represented in the SQLToKeyNoSQL canonical model.

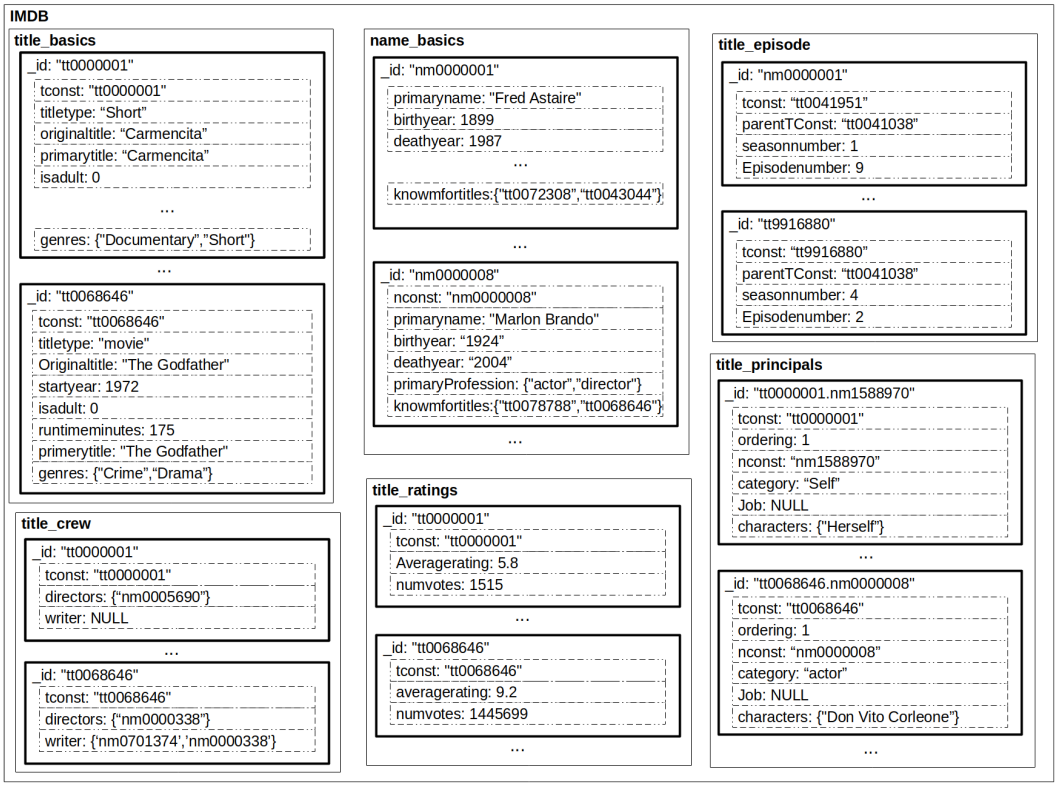

Figure 8. *IMDb* mapped to MongoDB by the *SQLToKyeNoSQL* approach.

The *RithWork* approach also provides the execution of a limited set of SQL operations in MongoDB and Cassandra. It translates some SQL DML to API methods of these NoSQL databases through predefined wrappers called *connectors*. Connectors parse SQL queries, send them to the target database, and return the result to the requester. In [42], no further details about data and operation mappings are presented. Despite that, in the case of Cassandra, we conclude that it probably accomplishes a trivial mapping for query operations over a single table, given that Cassandra has an SQL-like query language.

NoSQL^2 and *MSI* mappings are defined manually through configuration files. *NoSQL*^{2} allows only DDL commands since it aims to help a DBA to manage objects and rules in the NoSQL database. On the other hand, *MSI* offers only a subset of DML commands.

Finally, *ZhangWork* gives support for document-oriented and column-oriented NoSQL databases. Like *RithWork*, this approach is based on wrappers for data accessing. Each wrapper maps SQL requests to the target NoSQL database API. Unfortunately, authors do not also provide details of the mapping process between the relational model and these two NoSQL data models.

4.1.3. Join Processing

Operations on NoSQL databases are not set-oriented. Instead, they are record-oriented. Thus, queries with join operations are not supported by NoSQL databases, as it happens in relational databases. To overcome this problem, most *Layer* approaches provide the processing of complex queries with join operations.

The *SimpleSQL* approach supports joins by decomposing the SQL query into a list of attributes, tables, joins, and predicates. If joins are detected, it splits the query into single queries with related predicates. After retrieving each single result set, a *dataTable* is created with the schema of the expected result. The list of retrieved items is joined item by item based on the join conditions and the relational-to-SimpleDB schema mappings. The *JackHare* approach processes a join through parallel map-reduce jobs. It starts by determining the size of the tables. Then, in the *map* phase, the smallest one is converted to a *key list* and stored in HBase. In the following, it matches the values of the biggest table with this key list and updates the key list. This process is also executed through map-reduce jobs to obtain better performance. After all key list entries are analyzed, *JackHare* returns the list of combined tuples, which constitutes the *reduce* phase. This strategy allows the manipulation of large data sets since the processing can be distributed over several nodes.

Apache Phoenix, *Unity* and *SQLToKeyNoSQL* support join operations between NoSQL data sources by adopting a *hash join* strategy based on record keys. *Unity* performs joins based on the size of datasets and how the datasets are distributed over the servers (if it is the case). It starts from datasets that fit in main memory. If none of the datasets fit in main memory, it performs the join operation in a parallel way.

Apache Phoenix implements a variant of the *hash join* algorithm for data that can fit in the main memory, and a *hash-merge* join otherwise. Both algorithms use the attributes of the join operation as keys to combine the tuples.

SQLToKeyNoSQL support joins data in the same data source or between different data sources. The implemented join algorithm is a standard *hash join*, where the keys for joining tuples are extracted from the join predicate. This approach also gives the user the possibility of changing the join algorithm. In the current version, *SQLToKeyNoSQL* supports *hash-join* (*default*) and *merge-join*. In addition, the developer can plug other join algorithms in the middleware.

MIDAS 1.8 executes join operations when data come from different types of data sources. It scans both result sets (a *nested loop* algorithm) to merge data from the two datasets and generate the final result. On the other hand, *MSI* proposes two choices to perform joins: (*i*) to transform a NoSQL object into a MySQL table and execute the join inside MySQL; or (*ii*) to execute *nested loop* joins between the datasets.

Finally, *RithWork*, *ZhangWork* and *NoSQL*^{2} approaches do not support join operations.

4.1.4. Comparison

Based on the discussion in the previous section, we group the features to be compared in Table 1 as follows: (*i*) *Data Model* refers to the target NoSQL data model; (*ii*) *SQL Support* highlights the supported SQL operations; (*iii*) *SQL Mapping* presents the approach to map input SQL operations to corresponding operations at the target NoSQL database; (*iv*) *Dictionary (Dict.)* indicates whether or not the approach works with a metadata catalog to map data and operations; and (*v*) *Join Processing* denotes the strategy to perform join operations.

Most approaches take into account metadata to support their wrapping strategy. *JackHare*, *SimpleSQL*, *MIDAS 1.8*, *SQLToKeyNoSQL* and *Unity* accomplish a semantic and a syntactic verification of SQL operations with the aid of a dictionary. In the first four approaches, a NoSQL schema must be available a priori, being the mapping pointed to this target schema. In the case of *Unity*, if the NoSQL source has schema information, then the dictionary is also considered. Otherwise, mapping rules at the data model level are considered, i.e., relational data are mapped to similar concepts in the target NoSQL data model. *ZhangWork* and *Apache Phoenix* use the dictionary to store metadata about the

target NoSQL database, e.g., table mapping, NoSQL database alias, among others. *SQLToKeyNoSQL* also underlies information about mapping tables to objects in the target NoSQL database through metadata in a dictionary, like the NoSQL database location, primary keys, and foreign keys. *RithWork* does not mention the use of a dictionary probably because the mapping is coded by hand through new connectors. *NoSQL*^{2} underlies the mapping strategy to configuration files built by the user.

Table 1. Comparison of layer approaches.

Approach	Data Model	SQL Support	SQL Mapping	Dict.	Join Processing
ZhangWork	Col./Doc.	DDL + DML Subset	API Methods	Yes	-
SimpleSQL	Document	DDL + DML Subset	API Methods	Yes	Similarity
JackHare	Columnar	DDL + DML Subset	Map-Reduce Jobs	Yes	Map-Reduce Jobs
Unity	Col./Doc.	DML Subset	API Methods	Yes	Hash Join
RithWork	Col./Doc.	DML Subset	API Meth./Access Com.	-	-
APache Phoenix	Columnar	DDL + DML	API Methods	Yes	Hash Join
SQLToKeyNoSQL	Key-based	DDL + DML Subset	API Methods	Yes	Hash Join
MIDAS 1.8	Document	DML Subset	API Methods	Yes	Nested Loop
NoSQL^2	Independent	DDL Subset	Manually Configured	No	-
MSI	Key-based	DML Subset	API Methods	Yes	Nested Loop

DML—Data Manipulation Language; DDL—Data Definition Language.

As shown in Table 1, most approaches work on specific NoSQL data models and NoSQL products. This lack of generality could be better addressed. For example, MongoDB and CouchDB are both document-oriented NoSQL data models, and *SimpleSQL* could provide support for both since they implement the same data model. In fact, *Layer* approaches offer, in general, a set of mapping rules to a specific NoSQL solution. *SQLToKeyNoSQL* is the only approach that provides wrappers for three different NoSQL data models (although *MIDAS 1.8* claims that support several types of data sources because it uses the same *SQLToKeyNoSQL*'s canonical model, regarding NoSQL data models, in the current version, it supports only document-oriented data model). *NoSQL*^{2} is the only work that claims to be data model independent, but the specification of mappings is entirely manual, which is a hard drawback.

According to Table 1, some works deal with a subset of SQL DDL and DML operations, while other ones are more limited (*MIDAS 1.8*, *MSI*, *NoSQL*^{2}, *Unity*, *RithWork*). *SimpleSQL* and *SQLToKeyNoSQL* deal with simplified versions of DDL operations, and DML operations cannot have nested queries or aggregate functions. Instead, *MSI* supports nested queries. *JackHare* additionally supports ordering, grouping, and aggregate functions in query operations. The DML capabilities of *Unity* are very similar to *SimpleSQL*. With respect to DDL operations, *SimpleSQL* considers only table and index creation. *Unity* and *RithWork* offer only support for a subset of SQL DML operations, and *Apache Phoenix* is the most comprehensive approach concerning SQL support. *ZhangWork* claims that it deals with a subset of SQL DDL and DML operations, but we did not find detailed information about it.

Considering the execution flow of each SQL operation, *Unity* and *RithWork* use the *MongoDB* physical operators to execute their operations since they propose a translation of SQL to *Mongo* query language. The *NoSQL*^{2} also uses *MongoDB* operators but do not support DML. Similarly, the *SimpleSQL* maps all the SQL statement to *SimpleDB* SQL-like query language. The mapping uses *SimpleDB* internal operators for selections and projections. *JackHare* implements map-reduce jobs as physical operators, and filtering actions are executed in the reduce phase. *SQLToKeyNoSQL* implements an internal representation model for query execution, and each SQL operation is decomposed and translated into primitives methods: put, get, delete from *REST API*. Similar to *SQLToKeyNoSQL*, *MIDAS 1.8* applies a process that decomposes the queries in an internal language that is translated to the specific API language.

On considering join processing, we see that the majority of approaches adopt different strategies, in particular, *hash join* implementations, which achieve better performance than *nested loop*-based solutions. Despite this lack of agreement, some of them provide distributed or parallel join processing, like *Apache Phoenix*, *Unity* and *JackHare*.

4.2. Storage Engine

Approaches in the storage engine category change the physical layer of an RDBMS (the storage manager of an RDBMS, in particular) to allow persistence and retrieval of data into/from NoSQL databases. The query processing module of the RDBMS makes requests to the modified storage manager that, in turn, maps relational data to data in compliance with a NoSQL data model. Three works fit into this category: *Phoenix* [50], *CloudyStore* [51] and *DQE* [52]. They are also detailed in the following in terms of architectural issues, schema mapping, and join processing.

4.2.1. Architectural Issues

The system architecture of *Phoenix* and *CloudyStore* are similar since they both implement a storage engine for the MySQL RDBMS. MySQL was chosen because its architecture allows the user to create pluggable storage engines. Storage engines must have support for the basic CRUD (create, read, update, and delete) operations. In this case, MySQL query processor calls the NoSQL storage engine to get and set data, i.e., *Phoenix* and *CloudyStore* translate the SQL requests to the native access method or access language of the target NoSQL database. The main difference between the two approaches is that *CloudyStore* uses the MySQL internal key to optimize data access, and *Phoenix* uses an intermediary data model to provide the mappings.

The distributed query engine (*DQE*) approach implements a storage engine for *Derby* RDBMS, allowing data storage in the HBase NoSQL database. *DQE* architecture is composed of three layers: *query engine*, *storage system* and *file system*. The first one receives and processes SQL operations. Then, it sends requests to the storage system, which, in turn, performs the requests (over the file system) and returns the results. To support this processing, *DQE* modifies the following Derby modules: *parser*, *optimizer* and *store management*. The first two modules are redefined to provide optimizations based on HBase store model system. The latter implements, in fact, the mapping between the relational and HBase data models.

4.2.2. Schema Mapping

Phoenix implements a MySQL storage engine to *Scalaris*. Scalaris is a key-value NoSQL database. In the key-value data model, the key is a unique identifier of a data item (value), and the value is retrieved as a single block. The value can be simple or complex in Scalaris. Simple values have atomic domains (e.g., *string* or *integer* data types), and complex values hold a set of key-value pairs.

Phoenix defines an intermediary data model to map relational data to the key-value data model called *VOEM* (*Value-based OEM*). This intermediary data model is used as a mapping schema between the two data models, and the idea is that it may be extended to provide mappings to other NoSQL data models. VOEM is an extension of OEM (*Object Exchange Model*), which is a complex object model whose instances have a unique atomic identifier (OID). As relational data are identified by values instead of OIDs, VOEM extends OEM with the notion of a *key*, i.e., the value identification.

A VOEM object is a tuple $v = < OID, \lambda, \tau, \nu, k >$, where OID is the identifier of the object, λ is a label that describes the object, τ is a type that indicates if the object is simple or complex, and ν is the object value. If the object has a complex type, then ν contains an OID set. Finally, k is a key that identifies the key-value pair and maps the tuple to the corresponding key-value pair in the key-value data model. This key is obtained by concatenating values in the VOEM graph, as detailed in the following. Figure 9a shows part of table *IMDb* represented in VOEM.

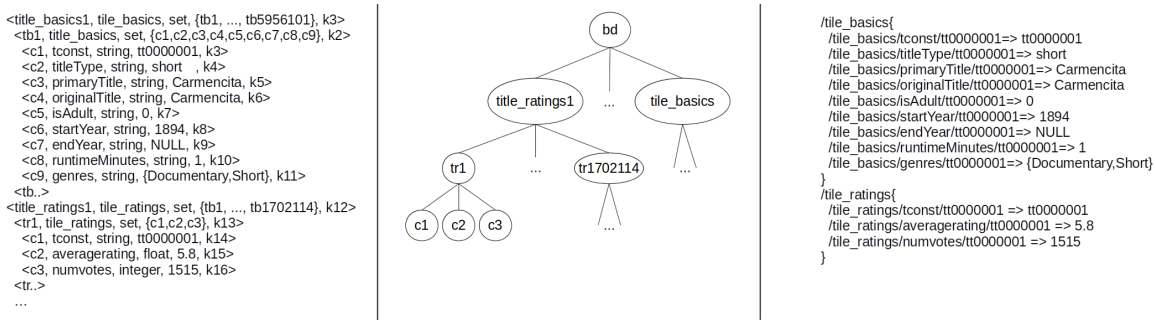

Figure 9. Table Movie represented in a VOEM data model.

The mapping of the relational data model to VOEM respects the following rules: (*i*) a tuple *t* of a relational table is represented by a VOEM object o_i with label *l* (the value of the primary key of *t*), and their values are a collection of *OIDs* that maps all the columns of *t* and (*ii*) each column $c \in t$ with type τ is mapped to a VOEM object o_j with label *c* (column name) and type τ.

A VOEM object collection can be seen as a labeled graph *V* (Figure 9b) where each node of *V* is an object and the edges are relationships between objects. This graph view is useful to define the key of each VOEM object o_i. In fact, a key is given by the concatenation of the labels of each object in a *depth-in* graph traversal and the primary key of the row that o_i represents. For example, in the key-value pair `"/tile_ratings/averagerating/tt0000001 => 5.8` of Figure 9c), the key is defined by the concatenation of the path labels *tr*1 (tile_ratings) and *c*2 (averagerating) (see Figure 9b), and the primary key with value *tt0000001* comes from the object *c*1 that identifies the mapped row.

Based on this path expression scheme for representing VOEM objects, the mapping to the key-value data model is straightforward: a VOEM object o_i is mapped to a key-value pair *kv*. The key of *kv* is the key of o_i. If the value of o_i is atomic, then the value of *kv* is the same of o_i. If the value is complex, then the value of *kv* is the set of VOEM objects that defines the value of o_i. For example, the first row of the table *tile_ratings* is mapped to the object "title_ratings" (Figure 9a,b), which is mapped to the key-value instance in Figure 9c). In the key-value instance, each line is a key-value pair: the first value is the key, and the second one is the value.

The *CloudyStore* approach maps relational data to a column-oriented NoSQL database named *Cloudy*. Cloudy has a data model similar to HBase: keyspaces, families of columns, keys, columns, and values.

CloudyStore proposes three rules to map a relational database to Cloudy: (i) a database *db* is mapped to a keyspace *K* with the same name of *db*; (ii) a table $tab \in db$ is mapped to a column family $CF \in K$ with the same name of *tab*; and (iii) for each tuple $t \in tab$ is generated a key *tk* which is the internal MySQL key for *t* (called *rowid*), and each column $c \in t$ is mapped to a column *ck* assigned to the key *tk*, with the same name and value of *c*.

Figure 10 shows how the *IMDb* is stored in Cloudy according to the *CloudyStore* approach. Each table is mapped to a column families (e.g., *title_basics*). Each table row is mapped to a set of *column-value* pairs. For example, the *Godfather* title is mapped as a set of columns in the *title_basics* column family, being identified by the MySQL *rowid* `A`.

In *DQE* approach, the mapping rules are the following: (i) a database *db* is mapped to an HBase keyspace *K* named *db*; (ii) a table $t \in db$ is mapped to an HTable *ht* with the same name of *t*; (iii) a row $r \in t$ is mapped to a row $rk \in ht$; and (iv) each column $c \in r$ is mapped to a column $ck \in rk$, with the same name and value of *c*.

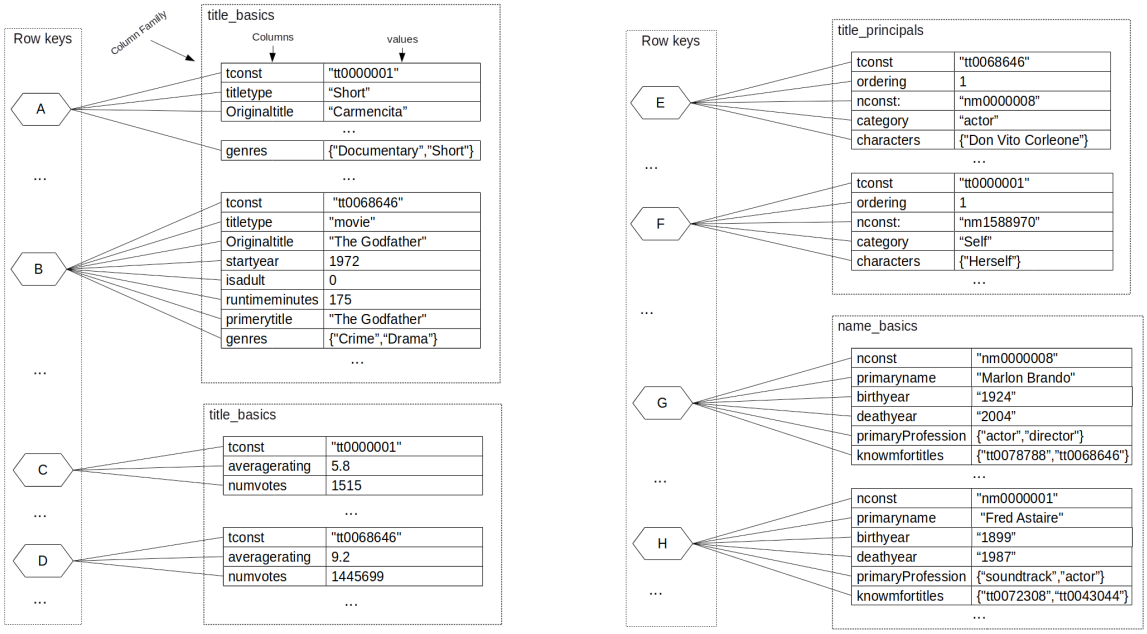

Figure 10. *IMDb* mapped to Cloudy according to the *CloudyStore* approach.

If a table has indexed attributes, *DQE* creates, for each indexed attribute, a new HTable. The attribute values are mapped to entries (keys) in this HTable and, for each entry, their values are the keys in the corresponding HTable that holds the attribute values. If an entry has more than one correspondent key, a set of keys is stored as a value in the HTable index.

Figure 11 shows the mapping of the *IMDb* to HBase according to the *DQE* approach. Notice that the map generated by *DQE* is the same of Apache Phoenix (also same Figure 5). The only difference between the mapping approach of Apache Phoenix and DQE is that DQE also creates HTables for index structures. DQE creates one HTable for each relational table (with the same number of column and rows) and additionally creates on HTable for each index of the relational system.

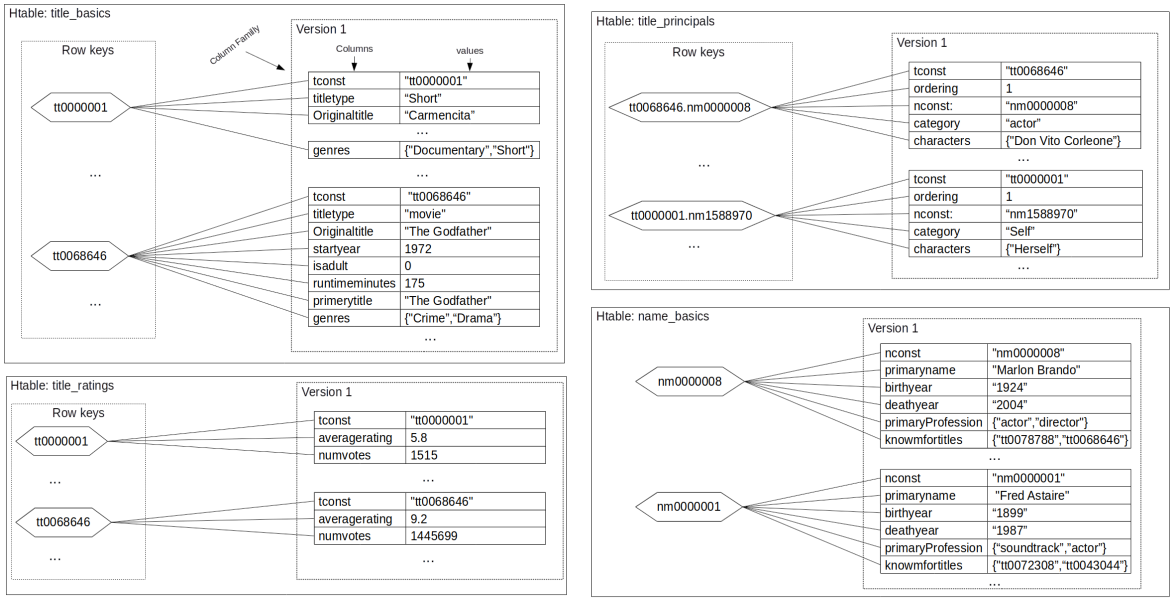

Figure 11. *IMDb* mapped to HBase according to the *DQE* approach.

4.2.3. Join Processing

Storage Engine approaches do not develop specific join strategies because the host RDBMS already has this capability. In this case, the approach is only responsible for getting data from the target NoSQL database, the join being processed by the RDBMS query processing module.

4.2.4. Comparison

As we did for *Layer* approaches, we built a table (Table 2) with the most important features of the approaches in the *Storage Engine* category. The column *Data Model* refers to the data model of the target NoSQL database. The column *RDB* denotes the RDBMS considered by the approach. It means that the approach can translate any SQL DDL or DML operation supported by this RDBMS and execute it in the target NoSQL database. The column *Mapping Information* presents the strategy for maintaining mapping information.

Table 2. Comparison of storage engine approaches.

Approach	Data Model	RDB	Mapping Information
Phoenix	Key-value	MySQL	VOEM
CloudyStore	Columnar	MySQL	Dictionary
DQE	Columnar	Derby	Dictionary

Table 2 reveals that the approaches consider only two NoSQL data models: key-value and column-oriented. It can be justified by the fact that they are the less complex NoSQL data models. We also note that all RDBMS are open source, which is expected since their source codes are available for developers that intend to modify their components.

On concerning mapping information, we observe that *Phoenix* follows a different strategy: it considers an intermediate data model (VOEM) for representing mapping information. As VOEM is a general graph-based model, it can be used for mapping relational data to any other NoSQL data model. On the other hand, *CloudyStore* and *DQE* implement a single and straightforward mapping to the data model of the target NoSQL database through a specific mapping schema defined in a dictionary, which is less flexible.

5. Comparative Analysis

This section presents a comparative analysis of all approaches described in this paper. Table 3 summarizes their general features and differences. Besides the *Approach* name and *Category* (according to the architectural classification presented in Section 3), it shows to which *NoSQL Data Model* the approach provides wrapping, the SQL capabilities (*SQL Support*), and if the approach deals with one or more data sources (*Multiple Sources*).

Table 3. Comparison of related work.

Approach	Category	NoSQL Data Model	SQL Support	Multiple Sources
ZhangWork	Layer	Document/Column	DDL + DML subset	Yes
Phoenix	Storage Engine	Key-value	Full SQL	No
Apache Phoenix	Layer	Document	DML + DDL	No
CloudyStore	Storage Engine	Column	Full SQL	No
DQE	Storage Engine	Column	DDL + DML	No
JackHare	Layer	Column	DML + DDL subset	No
SimpleSQL	Layer	Document	DDL + DML subset	No
RithWork	Layer	Document/Column	DML subset	Yes
Unity	Layer	Document/Column	DML subset	Yes
SQLToKeyNoSQL	Layer	Key-based	DML + DDL subset	Yes
NoSQL^2	Layer	Independent	DDL subset	Yes
MIDAS 1.8	Layer	Document	DML subset	Yes
MSI	Layer	Key-based	DML subset	Yes

Notice that only the features that make sense to all related work are highlighted in Table 3. For example, we do not consider here the comparison of the RDBMS on which the mapping strategy was implemented because this feature is relevant only for approaches in the *Storage Engine* category.

First of all,, we argue that the choice for a specific approach depends on its category and the user needs. Each category has its advantages and disadvantages. *Layer* approaches are more flexible than *Storage Engine* approaches since they can be designed to handle multiple sources. However, there is an overhead to process SQL operations due to the required mapping to a NoSQL database performed by the approach. On the other hand, *Storage Engine* minimizes this overhead since the mappings are coded in the kernel of an RDBMS. However, they are developed to specific RDBMS and coupled to specific NoSQL databases, being less flexible.

Most of the *Layer* approaches provide access to multiple sources. However, they are usually based on wrappers that must be manually coded to implement the translation of SQL operations to the API access methods of the NoSQL databases. Different from them, *SQLToKeyNoSQL* supports the three key-based NoSQL databases, and its mapping strategy is based on a canonical model that abstracts all key-based NoSQL models. The canonical model provides a more straightforward mapping to the data models of the NoSQL databases, as well as (also generic) primitive methods generated from SQL operations that are further translated to each specific NoSQL database access method. It reduces considerably the manual effort to define the mappings. *MIDAS 1.8* also considers an intermediate canonical model, but it is restricted to the document data model.

The other two points that arise from Table 3 analysis are the absence of a comprehensive approach concerning the support of mappings for all NoSQL data models, as well as full support for SQL operations, which is offered by few works. The only exception w.r.t. NoSQL data model support is $NoSQL^2$, but its limitation to DDL operations makes it a non-promising solution. In fact, the first point is a hard task for *Storage Engine* approaches because they usually focus on a single target NoSQL database at the physical level, and the second point is a hard task for *Layer* approaches because it requires a robust SQL processor. Thus, if the focus is flexibility, *Layer* approaches are recommended. Otherwise, if the focus is a full SQL support, *Storage Engine* approaches are recommended.

Table 4 shows another view of the relational-to-NoSQL schema mapping approaches. It organizes the related work according to the target NoSQL data model supported. Approaches that provide mappings for more than one NoSQL data model are duplicated in the table. In addition, approaches that do not detail a schema mapping strategy (*ZhangWork*, *RithWork* and $NoSQL^2$) were omitted from Table 4.

Table 4. Schema mapping strategy comparison.

Approach	Target NoSQL Data Model	Relational-to-NoSQL Mapping Strategy				
		Database	Table	Tuple	Attribute	Value
Phoenix	Key-Value	database	key component	-	key component	value
SQLToKeyNoSQL		database	key component	value	value component	value component
MSI		database	key component	value	value component	value component
JackHare	Columnar	HTable	key component	column set	column	value
DQE		keyspace	HTable	column set	column	value
Apache Phoenix		keyspace	HTable	column set	column	value
CloudyStore		keyspace	column family	column set	column	value
SQLToKeyNoSQL		keyspace	column family	column set	column	value
SimpleSQL	Document	collection	attribute	document	attribute	value
Unity		database	collection	document	attribute	value
Midas 1.8		DBaaS and DaaS	collection	document	attribute	value
MSI		database	collection	document	attribute	value
SQLToKeyNoSQL		database	collection	document	attribute	value

As previously stated, no approach maps the relational model to a NoSQL solution based on the graph data model, this subject being an open issue. Concerning the other NoSQL data models, we consider, in this comparison, the set of common concepts described in Section 2 and specific features of the data models of the mentioned target NoSQL databases in the related work. For the key-value data model, the database is a repository of a set of keys. For the column-oriented data model, generally, we have a keyspace, a column family, and a column set accessed by a unique key. For the document-oriented data model, the database is a repository of document collections, a document collection is a set of documents, and a document is composed of a set of attributes and their values.

On considering the target key-value data model, we see two different mapping strategies, as shown in Table 4. *Phoenix* defines a key for the key-value schema as a composition of table name, attribute name, and the primary key value. Instead, *SQLToKeyNoSQL* considers only the table name and the primary key value of a tuple to define a key. For example, the first tuple of the *Movie* table generates the following key-value pairs {"/movie/name/1", "Psycho"}, {"/movie/director/1", "1"} and {"/movie/year/1", "1960"} in *Phoenix*, and the following key-value pair {"movie.1", "id:1; name:Psycho; director:1; year:1960;"} in *SQLToKeyNoSQL*. *SQLToKeyNoSQL* seems to be a more interesting approach since it generates fewer data entries in the key-value database in a more compact way. Keys in *Phoenix* are larger because of the redundant table and attribute information for each mapped tuple.

As shown in Table 4, several approaches provide mappings to the NoSQL column-oriented data model. In this case, the differences in the proposed strategies regard the treatment of database and table information. We argue that approaches that convert a database to a keyspace and a table to a column family (*SQLToKeyNoSQL* and *CloudyStore*) better respect the hierarchy of concepts of both data models: relational and column-oriented. In the same way, approaches that map relational concepts directly to HTables (*JackHare*, *Apache Phoenix* and *DQE*) provide a specific mapping strategy since HTable is a particular concept of the HBase NoSQL data model. As a keyspace is a repository of HTables for HBase, it seems to be more manageable to distribute different HTables (various relational tables) in different data nodes, if necessary, for elasticity purposes, i.e., data from the same table tend to be stored together. *JackHare* maintains data from several tables in the same HTable, being more complex to manage data distribution. However, it concatenates the primary key values and table names to build the keys of every set of key-value pairs. This strategy improves the performance of queries over the source database.

For the NoSQL document-oriented data model, we also notice different mapping strategies for database and table information. Following the same reasoning for the NoSQL column-oriented data model, we argue that to map a database to the equivalent concept in the document data model, as well as tables to collections, are more suitable to the hierarchy of concepts of the relational model and, thus, tend to facilitate the mapping management. All of the works except *SimpleSQL* adopt this mapping strategy.

Table 5 shows how the approaches deal with the mapping of SQL operations to NoSQL operations. Based on the way the approaches work, we classify the mapping strategies in two categories: (*i*) *SQL compiler*: approaches in this category accomplish a lexical, syntactic and semantic analysis of the SQL operation, and further translate the SQL operation directly to the specific NoSQL target access language to be executed and (*ii*) *SQL parser + hand-coded wrappers*: approaches in this category accomplish a lexical and syntactic analysis of the SQL operation followed by the invocation of specific wrappers that translate such operation to a target NoSQL database operation.

Approaches that communicate with one (or more) specific NoSQL database usually adopt the SQL *compiler* strategy because it translates SQL operations to the specific access language operations of the target NoSQL database. However, this strategy leads the approaches to work with a specific NoSQL database. On the other hand, approaches that follow the *SQL parser + hand-coded wrappers* strategy are more flexible since they can connect to new NoSQL databases through the implementation of new wrappers.

In some cases, the approach is able to generate an optimized access plan for the input SQL query before translating it to the access method or access language of the target NoSQL database. *CloudyStore*, *DQE*, *Unity*, *SQLToKeyNoSQL* and *MSI* support this additional capability (*SQL query optimizer* column in Table 5).

Table 5. Comparison of SQL operation mapping strategies.

Approach	SQL Compiler	SQL Parser + Hand-Coded Wrappers	SQL Query Optimizer
ZhangWork	-	X	-
Phoenix	X	-	-
Apache Phoenix	X	-	-
CloudyStore	X	-	X
DQE	X	-	X
JackHare	X	-	-
SimpleSQL	X	-	-
RithWork	-	X	-
Unity	-	X	X
SQLToKeyNoSQL	-	X	X
NoSQL^2	-	X	-
MSI	-	X	X
Midas 1.8	-	X	-

Approaches that follow the *SQL compiler* strategy are suitable for applications: (*i*) that use the access interface of a specific RDBMS (*Storage Engine* approaches) because they do not have to modify their access interfaces or (*ii*) whose data fits well with a specific NoSQL data model, providing a better access and/or storage performance.

Instead, approaches that follow the *SQL parser + hand-coded wrappers* strategy are suitable for dynamic applications that deal with heterogeneous data or data with flexible schemas. This kind of application may require a flexible usage of several (different) NoSQL databases to represent data, like polyglot persistence systems [4].

Final Remarks

The problem addressed in this survey is how to move relational-based applications to NoSQL-based applications with a low startup cost. Although all the approaches can be used for this purpose, we argue that most of them do not fulfill the needs of a real data-centric application. We can point out two reasons for that: lack of support for (*i*) transactions and (*ii*) full DDL and DML operations support.

Store Engine approaches are more suitable to achieve the requirement (*ii*). However, the applications are restricted to work with a specific RDBMS and a specific NoSQL data model. In turn, *Layer* approaches do not restrict applications to work with a given RDBMS, yet they do not support transactions (only single ones) and deal only with a subset of DML and DDL operations. It denotes that there is still space for improvements in this research topic.

6. Conclusions

The traditional RDBMS architecture is inadequate for big data-centric applications, mainly due to the overhead to guarantee data and schema consistency. To cope with this problem, a new generation of RDBMS products has been proposed for the cloud. Even then, some applications are not concerned about ACID properties because scalability and availability are more important than the consistency. In this case, NoSQL databases are the right choice.

NoSQL databases are designed to be scalable with big data and available most of the time. However, the majority of them do not support SQL operations, so the migration costs from relational-based applications to NoSQL databases are prohibitive since the applications need to understand new database interfaces, new access language paradigms, and new data models.

The purpose of this survey is to bring together all approaches that implement wrapping solutions to access NoSQL databases through SQL operations. We propose an architectural classification to classify the related work and a set of features to compare them. The features allow us to highlight the weaknesses and strengths of each approach. These are the most significant contributions of this survey.

The general comparison shows the advantages of some approaches, like the flexibility of *Layer* approaches and the optimized access of *Storage Engine* approaches. These issues must be carefully considered when developers intend to use this kind of wrapping solution. In turn, the specific comparisons are useful for evaluating features of a given solution and thus helping developers. Therefore, we hope that this survey can be used as a guide for practitioners that want to move the relational-based applications to an NoSQL environment with low startup cost.

Finally, on giving focus to relational-to-NoSQL wrapping proposals, we identified some open issues by the time this survey was written. We emphasize here the absence of works that make a full (and efficient) translation of SQL operations, including integrity constraints and transactions, as well as a detailed translation of relational data for all existing data models of NoSQL databases. These issues are relevant for future research efforts.

Author Contributions: G.S., D.D., and R.M. conceived of the presented idea and co-wrote the paper. G.S. contributed to the research design. All authors discussed the results and contributed to the writing of the paper.

Funding: This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior—Brasil (CAPES)—Finance Code 001.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Liu, Z.H.; Hammerschmidt, B.C.; McMahon, D. JSON data management: Supporting schema-less development in RDBMS. In Proceedings of the International Conference on Management of Data (SIGMOD), Snowbird, UT, USA, 22–27 June 2014.
2. Nance, C.; Losser, T.; Iype, R.; Harmon, G. NoSQL vs RDBMS—Why There Is Room for Both. In Proceedings of the Southern Association for Information Systems (SAIS), Atlanta, GA, USA, 23–24 March 2013.
3. Cattell, R. Scalable SQL and NoSQL Data Stores. *SIGMOD Rec.* **2011**, *39*, 12–27. [CrossRef]
4. Sadalage, P.J.; Fowler, M. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*; Pearson Education: London, UK, 2012.
5. Bisbal, J.; Lawless, D.; Wu, B.; Grimson, J. Legacy information systems: Issues and directions. *IEEE Softw.* **1999**, *16*, 103–111. [CrossRef]
6. Llewellyn, B. *Oracle Multitenant*; White paper; Oracle: Redwood City, CA, USA, 2013.
7. EnterpriseDB. *A Cloud-Ready Enterprise Database Checklist*; White Paper; PostgreSQL Global Development Group: Bedford, MA, USA, 2012.
8. Lee, J.; Malcolm, G.; Matthews, A.; Negrin, R.; Owens, Z.; Robinson, D.; Azure, S. *Overview of Microsoft SQL Azure Database*; White Paper; Microsoft: Redmond, WA, USA, 2009.
9. Stonebraker, M. New Opportunities for New SQL. *Commun. ACM* **2012**, *55*, 10–11. [CrossRef]
10. Cetintemel, U.; Du, J.; Kraska, T.; Madden, S.; Maier, D.; Meehan, J.; Pavlo, A.; Stonebraker, M.; Sutherland, E.; Tatbul, N.; et al. S-Store: A Streaming NewSQL System for Big Velocity Applications. *VLDB* **2014**, *7*, 1633–1636. [CrossRef]
11. Kallman, R.; Kimura, H.; Natkins, J.; Pavlo, A.; Rasin, A.; Zdonik, S.; Jones, E.P.; Madden, S.; Stonebraker, M.; Zhang, Y.; et al. H-store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.* **2008**, *1*, 1496–1499. [CrossRef]
12. Taft, R.; Mansour, E.; Serafini, M.; Duggan, J.; Elmore, A.J.; Aboulnaga, A.; Pavlo, A.; Stonebraker, M. E-store: Fine-grained Elastic Partitioning for Distributed Transaction Processing Systems. *Proc. VLDB Endow.* **2014**, *8*, 245–256. [CrossRef]
13. Aguilera, M.K.; Leners, J.B.; Walfish, M. Yesquel: Scalable Sql Storage for Web Applications. In Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15), Monterey, CA, USA, 4–7 October 2015.

14. Serafini, M.; Taft, R.; Elmore, A.J.; Pavlo, A.; Aboulnaga, A.; Stonebraker, M. Clay: Fine-grained Adaptive Partitioning for General Database Schemas. *Proc. VLDB Endow.* **2016**, *10*, 445–456. [CrossRef]
15. Yuan, L.Y.; Wu, L.; You, J.H.; Chi, Y. Rubato DB: A Highly Scalable Staged Grid Database System for OLTP and Big Data Applications. In Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, Shanghai, China, 3–7 November 2014; ACM: New York, NY, USA, 2014; pp. 1–10.
16. Alsubaiee, S.; Altowim, Y.; Altwaijry, H.; Behm, A.; Borkar, V.; Bu, Y.; Carey, M.; Cetindil, I.; Cheelangi, M.; Faraaz, K.; et al. AsterixDB: A Scalable, Open Source BDMS. *Proc. VLDB Endow.* **2014**, *7*, 1905–1916. [CrossRef]
17. Armbrust, M.; Xin, R.S.; Lian, C.; Huai, Y.; Liu, D.; Bradley, J.K.; Meng, X.; Kaftan, T.; Franklin, M.J.; Ghodsi, A.; et al. Spark SQL: Relational Data Processing in Spark. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15), Melbourne, VIC, Australia, 31 May–4 June 2015.
18. Chaiken, R.; Jenkins, B.; Larson, P.; Ramsey, B.; Shakib, D.; Weaver, S.; Zhou, J. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proc. VLDB Endow.* **2008**, *1*, 1265–1276. [CrossRef]
19. Gouda, Y.G.; Mohammed, H.S.; Khafagy, M.H. Advanced SQL Query To Flink Translator. *Int. J. Appl. Inf. Syst.* **2016**, *10*, 11–15.
20. Isard, M.; Yu, Y. Distributed Data-parallel Computing Using a High-level Programming Language. In Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09), Providence, RI, USA, 29 June–2 July 2009.
21. Olston, C.; Reed, B.; Srivastava, U.; Kumar, R.; Tomkins, A. Pig Latin: A Not-so-foreign Language for Data Processing. In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08), Vancouver, BC, Canada, 10–12 June 2008.
22. Ong, K.W.; Papakonstantinou, Y.; Vernoux, R. The SQL++ unifying semi-structured query language, and an expressiveness benchmark of SQL-on-Hadoop, NoSQL and NewSQL databases. *arXiv* **2014**, arXiv:1405.3631.
23. Vathy-Fogarassy, Á.; Hugyák, T. Uniform data access platform for SQL and NoSQL database systems. *Inf. Syst.* **2017**, *69*, 93–105. [CrossRef]
24. Kolev, B.; Bondiombouy, C.; Valduriez, P.; Jimenez-Peris, R.; Pau, R.; Pereira, J. The CloudMdsQL Multistore System. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16), San Francisco, CA, USA, 26 June–1 July 2016.
25. Thusoo, A.; Sarma, J.S.; Jain, N.; Shao, Z.; Chakka, P.; Anthony, S.; Liu, H.; Wyckoff, P.; Murthy, R. Hive: A Warehousing Solution over a Map-reduce Framework. *Proc. VLDB Endow.* **2009**, *2*, 1626–1629. [CrossRef]
26. Giannakouris, V.; Papailiou, N.; Tsoumakos, D.; Koziris, N. MuSQLE: Distributed SQL query execution over multiple engine environments. In Proceedings of the 2016 IEEE International Conference on Big Data (Big Data), Washington, DC, USA, 5–8 December 2016.
27. Alami, A.E.; Bahaj, M. Migration of a relational databases to NoSQL: The way forward. In Proceedings of the 2016 5th International Conference on Multimedia Computing and Systems (ICMCS), Marrakech, Morocco, 29 September–1 October 2016.
28. Jia, T.; Zhao, X.; Wang, Z.; Gong, D.; Ding, G. Model Transformation and Data Migration from Relational Database to MongoDB. In Proceedings of the 2016 IEEE International Congress on Big Data (BigData Congress), San Francisco, CA, USA, 27 June–2 July 2016.
29. Liu, Z.H.; Hammerschmidt, B.C.; McMahon, D.; Liu, Y.; Chang, H.J. Closing the functional and Performance Gap between SQL and NoSQL. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16), San Francisco, CA, USA, 26 June–1 July 2016.
30. DB, I. *DB2 JSON Capabilities*; White Paper; IBM: Armonk, NY, USA, 2016.
31. Microsoft. *JSON Support in SQL Server 2016*; White Paper; Microsoft: Redmond, WA, USA, 2016. Available online: https://blogs.msdn.microsoft.com/jocapc/2015/05/16/json-support-in-sql-server-2016/ (accessed on 13 January 2017).
32. PostgreSQL. JSON Types. 2016. Available online: https://www.postgresql.org/docs/9.4/static/datatype-json.html (accessed on 14 February 2017).
33. Chasseur, C.; Li, Y.; Patel, J.M. Enabling JSON Document Stores in Relational Systems. In Proceedings of the 16th International Workshop on the Web and Databases, New York, NY, USA, 23 June 2013.

34. Tahara, D.; Diamond, T.; Abadi, D.J. Sinew: A SQL system for multi-structured data. In Proceedings of the International Conference on Management of Data (SIGMOD' 14), Snowbird, UT, USA, 22–27 June 2014.
35. DiScala, M.; Abadi, D.J. Automatic Generation of Normalized Relational Schemas from Nested Key-Value Data. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16), San Francisco, CA, USA, 26 June–1 July 2016.
36. Corbellini, A.; Mateos, C.; Zunino, A.; Godoy, D.; Schiaffino, S. Persisting big-data: The NoSQL landscape. *Inf. Syst.* **2017**, *63*, 1–23. [CrossRef]
37. IMDb. Internet Movie Database. 2019. Available online: www.imdb.com/interfaces (accessed on 12 July 2019).
38. Zhang, H.; Wang, Y.; Han, J. Middleware design for integrating relational database and NOSQL based on data dictionary. In Proceedings of the International Conference on Transportation, Mechanical, and Electrical Engineering (TMEE), ChangChun, China, 16–18 December 2011; pp. 1469–1472.
39. Dos Santos Ferreira, G.; Calil, A.; dos Santos Mello, R. On Providing DDL Support for a Relational Layer over a Document NoSQL Database. In Proceedings of the International Conference on Information Integration and Web-based Applications & Services, Vienna, Austria, 2–4 December 2013.
40. Chung, W.C.; Lin, H.P.; Chen, S.C.; Jiang, M.F.; Chung, Y.C. JackHare: A framework for SQL to NoSQL translation using MapReduce. *Autom. Softw. Eng.* **2014**, *21*, 489–508. [CrossRef]
41. Lawrence, R. Integration and Virtualization of Relational SQL and NoSQL Systems Including MySQL and MongoDB. In Proceedings of the International Conference on Computational Science and Computational Intelligence (CSCI), Las Vegas, NV, USA, 10–13 March 2014.
42. Rith, J.; Lehmayr, P.S.; Meyer-Wegener, K. Speaking in Tongues: SQL Access to NoSQL Systems. In Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC'14), Gyeongju, Korea, 24–28 March 2014.
43. Apache. White Paper: Apache Phoenix. 2017. Availlable online: http://phoenix.apache.org/ (accessed on 2 March 2017).
44. Schreiner, G.A.; Duarte, D.; dos Santos Mello, R. SQLtoKeyNoSQL: A Layer for Relational to Key-based NoSQL Database Mapping. In Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services, Brussels, Belgium, 11–13 December 2015.
45. Schreiner, G.A.; Duarte, D.; dos Santos Mello, R. Bringing SQL databases to key-based NoSQL databases: A canonical approach. *Computing* **2019**, doi:10.1007/s00607-019-00736-1. [CrossRef]
46. Ribeiro, E.L.F.; Vieira, M.A.; Claro, D.B.; Silva, N. Transparent Interoperability Middleware between Data and Service Cloud Layers. In Proceedings of the 2018 International Conference on Cloud Computing and Services Science, Funchal, Madeira, Portugal, 19–21 March 2018.
47. Adriana, J.; Holanda, M. NoSQL^2: SQL to NoSQL Databases. In Proceedings of the 2018 Trends and Advances in Information Systems and Technologies (WorldCist), Naples, Italy, 27–29 March 2018.
48. Li, C.; Gu, J. An integration approach of hybrid databases based on SQL in cloud computing environment. *Softw. Pract. Exp.* **2019**, *49*, 401–422. [CrossRef]
49. Dean, J.; Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* **2008**, *51*, 107–113. [CrossRef]
50. Arnaut, D.E.; Schroeder, R.; Hara, C.S. Phoenix: A Relational Storage Component for the Cloud. In Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing, Washington, DC, USA, 28 June–3 July 2013; IEEE Computer Society: Washington, DC, USA, 2013.
51. Egger, D. SQL in the Cloud. Master's Thesis, ETH Zurich, Zurich, Switzerland, 2009.
52. Vilaça, R.; Cruz, F.; Pereira, J.; Oliveira, R. An Effective Scalable SQL Engine for NoSQL Databases. In *Distributed Applications and Interoperable Systems: IFIP International Conference on Distributed Applications and Interoperable Systems*; Dowling, J., Taïani, F., Eds.; Springer: Berlin/Heidelberg, Germany, 2013.