# Data Integration over NoSQL Stores Using Access Path Based Mappings

Olivier Curé, Robin Hecht, Chan Le Duc, Myriam Lamolle

▶ **To cite this version:**

Olivier Curé, Robin Hecht, Chan Le Duc, Myriam Lamolle. Data Integration over NoSQL Stores Using Access Path Based Mappings. DEXA 2012, Aug 2011, Toulouse, France. pp.481-495, 10.1007/978-3-642-23088-2_36 . hal-00738356

**HAL Id: hal-00738356**

**https://hal-upec-upem.archives-ouvertes.fr/hal-00738356**

Submitted on 4 Oct 2012

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Data Integration over NoSQL Stores Using Access Path Based Mappings

Olivier Curé[1], Robin Hecht[2], Chan Le Duc[3], Myriam Lamolle[3]

[1] Université Paris-Est, LIGM, Marne-la-Vallée, France
ocure@univ-mlv.fr
[2] University of Bayreuth, Bayreuth, Germany
robin.hecht@uni-bayreuth.de
[3] LIASD Université Paris 8 - IUT de Montreuil
chan.leduc,myriam.lamolle@iut.univ-paris8.fr

**Abstract.** Due to the large amount of data generated by user interactions on the Web, some companies are currently innovating in the domain of data management by designing their own systems. Many of them are referred to as NoSQL databases, standing for 'Not only SQL'. With their wide adoption will emerge new needs and data integration will certainly be one of them. In this paper, we adapt a framework encountered for the integration of relational data to a broader context where both NoSQL and relational databases can be integrated. One important extension consists in the efficient answering of queries expressed over these data sources. The highly denormalized aspect of NoSQL databases results in varying performance costs for several possible query translations. Thus a data integration targeting NoSQL databases needs to generate an optimized translation for a given query. Our contributions are to propose (i) an access path based mapping solution that takes benefit of the design choices of each data source, (ii) integrate preferences to handle conflicts between sources and (iii) a query language that bridges the gap between the SQL query expressed by the user and the query language of the data sources. We also present a prototype implementation, where the target schema is represented as a set of relations and which enables the integration of two of the most popular NoSQL database models, namely document and a column family stores.

## 1 Introduction

In [8], several database experts argued that Relational Data Base Management Systems (RDBMS) can no longer handle all the data management issues encountered by many current applications. This is mostly due to (i) the high, and ever increasing, volume of data needed to be stored by many (web) companies, (ii) the extreme query workload required to access and analyze these data and (iii) the need for schema flexibility.

Several systems have already emerged to propose an alternative to RDBMS and many of them are categorized under the term NoSQL, standing for 'Not only SQL'. Many of these databases are based on the Distributed Hash Table (DHT)

model which provides a hash table access semantics. That is, in order to access or modify an object data, a client is required to provide the key of that object and then the database will lookup the object using an equality match to the required attribute key. The first implementations where developed by companies like Google and Amazon with respectively Bigtable [1] and Dynamo [3]. These systems influenced the implementation of several open source systems such as Cassandra[4], HBase[5], etc. Nowadays, the NoSQL ecosystem is relatively rich with several categories of databases: column family (e.g. Bigtable, HBase, Cassandra), key/value ( e.g. Dynamo, Riak[6]), document (e.g. MongoDB[7], CouchDB[8]) and graph oriented (e.g. InfiniteGrap[9], Neo4J[10]). Most of these systems share common characteristics by aiming to support scalability, availability, flexibility and to ensure fast access times for storage, data retrieval and analysis.

In order to meet some of these requirements, NoSQL database instances are designed to reply efficiently to the precise needs of a given application. Note that a similar approach, named denormalization [4], is frequently encountered for application using relational databases. Nevertheless, it may be required to combine the data stored in several NoSQL database instances into a single application and at the same time to leave them evolve with their own applications. This combination of data coming from different sources corresponds to the notion of a data integration system presented in [5]. Yet, several issues emerge due to the following NoSQL characteristics: (i) NoSQL categories are based on different data models and each implementation within a category may have its own specificities. (ii) There does not exist a common query language for all NoSQL databases. Moreover, most systems only support a procedural definition of queries. (iii) The NoSQL ecosystem is characterized by a set of heterogeneous data management systems, e.g. not all databases support indexing. (iv) The denormalized aspect of NoSQL databases makes query performance highly dependent on access paths.

In this paper, we present a data integration system which is based on the assumptions of Figure 1. The target schema corresponds to a standard relational model. This is motivated by the familiarity of most end-users with this data model and its possibility to be queried with the SQL language. The sources can either correspond to a set of column family, document and key/value stores as well as to standard RDBMS.

To enable the querying of NoSQL databases within a data integration framework we propose the following contributions. (1) We define a mapping language between the target and the sources which takes into account the denormalization aspect of NoSQL databases. This is materialized by storing preferred access

[4] http://cassandra.apache.org/
[5] http://hbase.apache.org/
[6] http://www.basho.com/
[7] http://www.mongodb.org/
[8] http://couchdb.apache.org/
[9] http://www.infinitegraph.com/
[10] http://neo4j.org/

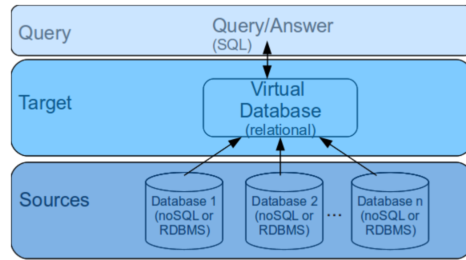


**Fig. 1.** Data integration overview

paths for a given mapping assertion. Moreover, this mapping language incorporates features dealing with conflicting data. (2) We propose a Bridge Query Language (BQL) that enables a transformation from an SQL query defined over the target to the query executed over a given source. (3) We present a prototype implementation which generates query programs for a popular document oriented database, namely MongoDB, and Cassandra, a column family store.

This paper is organized as follows. In Section 2, we present related works in the domain of querying NoSQL databases. In Section 3, we provide background knowledge on two feature rich and popular NoSQL databases: document and column family stores. Section 4 presents our data integration framework with a presentation of the syntax and semantics of the mapping language. In Section 5, query processing in our data integration system is presented and BQL is detailed. Section 6 concerns aspects of the prototype implementation. Finally, Section 7 concludes this paper.

## 2 Related work

In this section, we present some related works in the domain of querying non relational databases in the context of the cloud and Map/Reduce.

Decoupling query semantics from the underlying data store is a widely spread technique to support multiple data sources in one framework. Therefore, various systems offer a common abstraction layer on top of their data storage layer.

Hadoop[11] is a framework that supports data-intensive applications. On top of a distributed, scalable, and portable filesystem (HDFS, [1]), Hadoop provides a column-oriented database called HBase for real-time read and write access to very large datasets.

In order to support queries against these large datasets, a programming model called MapReduce [2] is provided by the system. MapReduce divides workloads into suitable units, which can be distributed over many nodes and therefore can be processed in parallel. However, the advantage of the fast processing of large datasets has also its catch, because writing MapReduce programs is a very time consuming business. There is a lot of overhead even for simple

[11] http://hadoop.apache.org/

tasks. Working out how to fit data processing into the MapReduce pattern can be a challenge. Therefore, Hadoop offers three different abstraction layers for its MapReduce implementation, called Hive, Pig and Cascading.

Hive[12] is a data warehouse infrastructure, which aims to bridge the gap between SQL and MapReduce queries. Therefore, it provides its own SQL like query language called HiveQL [9]. It has traditional SQL constructs like joins, GROUP BY, WHERE, SELECT and FROM clauses. These commands are translated into MapReduce functions afterwards. Hive insists that all data has to be stored in tables, with a schema under its management. Hive allows traditional MapReduce programs to be able to plug in their own mappers and reducers to do more sophisticated analysis.

Like Hive, Pig[13] tries to raise the level of abstraction for processing large data sets with Hadoop's MapReduce implementation. The Pig platform consists of a high level language called Pig Latin [6] for constructing data pipelines, where operations on an input relation are executed one after the other. These Pig Latin data pipelines are translated into a sequence of MapReduce jobs by a compiler, which is also included in the Pig framework.

Cascading[14] is an API for data processing on Hadoop clusters. It is not a new text based query syntax like Pig or another complex system that must be installed and maintained like Hive. Cascading offers a collection of operations like functions, filters and aggregators, which can be wired together into complex, scale-free and fault tolerant data processing workflows as opposed to directly implementing MapReduce algorithms.

In contrast to missing standards in a query language for NoSQL databases, standards for persisting java objects already exist. With the Java Persistence API (JPA[15]) and Java Domain Objects (JDO[16]) it is possible to map java objects into different databases. The Datanucleus implementation of these two standards provides a mapping layer on top of HBase, BigTable [1], Amazon S3[17], MongoDB and Cassandra. Googles App Engine[18] uses this framework for persistence.

A powerful data query and administration tool which is used extensively within the Oracle community is Quest Softwares Toad[19]. Since 2010, a prototype which also offers its support for column family stores is available. During the time of writing this paper, the beta version 1.2 can be connected to Azure Table Services[20], Cassandra, SimpleDB[21], HBase and every ODBC compliant relational database.

[12] http://hive.apache.org/
[13] http://pig.apache.org
[14] http://www.cascading.org/
[15] http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html
[16] http://www.oracle.com/technetwork/java/index-jsp-135919.html
[17] http://aws.amazon.com/de/s3/
[18] http://code.google.com/intl/de-DE/appengine/
[19] http://toadforcloud.com
[20] http://msdn.microsoft.com/en-us/library/dd179423.aspx
[21] http://aws.amazon.com/de/simpledb/

Toad for Cloud consists of two components. The first is the Toad client, which can be installed on a Microsoft Windows computer. It can be used to access different databases, the Amazon EC2 console, and to write and execute SQL statements. The second component is the Data Hub. It translates SQL statements submitted through the Toad client into a language understood by all supported databases and returns results in the familiar tabular row and column format.

In order to use SQL on column family stores like Cassandra and HBase, the column families, rows and columns have to be mapped into virtual tables. Afterwards, the user does have full MySQL support on these data, containing also inserts, updates and deletes. Furthermore, it is possible to do virtual data integration with different data sources.

One reason why only column family stores are supported by Toad is their easy mapping to relational databases. To do the same with a document store containing objects with a deep nested structure or squeezing a graph into a relational schema is a much more complicated task. Even if a suitable solution was found, powerful and easy to use query languages, tools and interfaces like Traverser API for Neo4J would be missing in the SQL layer of Toad, which queries the mapped tables.

Due to their different data models and their relatively young history, NoSQL databases still lack a common query language. However, Quest Software and Hadoop demonstrate that it is possible to use SQL (Toad) or a SQL like query language (Hive) on top of column family stores. A mapping to document stores and graph databases is still missing.

## 3 Background

We have seen that each category of NoSQL databases has its own data model. In this section, we present details concerning document oriented and column family categories.

### 3.1 Document oriented databases

Document oriented databases correspond to an extension of the well-known key-value concept where in this case the value consists of a structured document. A document contains hierarchically organized data similar to XML and JSON. This permits to represent one-to-one as well as one-to-many relationships in a single document. Therefore a complex document can be retrieved or stored without using joins. Since document oriented databases are aware of stored data, it enables to define document field indexes as well as to propose advanced query features. The most popular document oriented databases are MongoDB (10gen) and CouchDB (Apache).

**Example 1:** In the following a document oriented database stores drug information aimed at an application targeting the general public. According to features proposed by the application, two so-called collections are defined:

drugD and therapD. drugD includes documents describing drug related information whereas therapD contains documents with information about therapeutic classes and drugs used for it. Each drugD document is identified by a drug identifier. In this example, its attributes are limited to the name of the product, its price, pharmaceutical lab and a list of therapeutic classes. The key for therapD documents is a string corresponding to the therapeutic class name. It contains a single attribute corresponding to the list of drug identifiers treating this therapeutic class. Figure 2 presents an extract of this database. Finally, in order to ensure an efficient search to patients an index on the attribute name of the drugD document is defined.

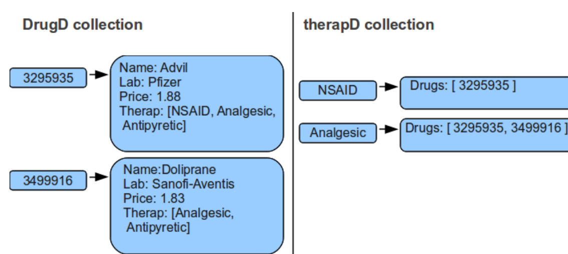


**Fig. 2.** Extract of drug document oriented database

### 3.2 Column-family databases

Column family stores correspond to persistent, sparse, distributed multilevel hash maps. In column family stores, arbitrary keys (rows) are applied to arbitrary key value pairs (columns). These columns can be extended with further arbitrary key value pairs. Afterwards, these key value pair lists can be organized into column families and keyspaces. Finally, column-family stores can appear in a very similar shape to relational databases on the surface. The most popular systems are HBase and Cassandra. All of them are influenced by Googles Bigtable.

**Example 2:** Figure 3 presents some column families defined in a medical application. Since Cassandra works best when its data model is denormalized, the data is divided on three column families: drugC, drugNameC and therapC. The columns drugName, contra, composition, lab are integrated into drugC and identified by row key drugId. drugNameC contains row key drugName and a drugId column in order to provide an efficient search for patients. Since end-users of this database need to search products by therapeutical classes, therapC contains therapName as row key and a column for each drugId with a timestamp as value.

**Fig. 3.** Extract of drug column family database

## 4 Data integration framework

This section presents the syntax and semantics of our data integration framework. Moreover, it focuses on the mapping language which supports the definition of correspondences between sources and target entities. Our mapping language integrates some original aspects by considering (i) query processing performances of the sources via access paths and (ii) dealing with contradicting information found between sources using preferences. In the rest of this paper, we consider the following example.

**Example 3:** A medical application needs to integrate drug data coming from two different NoSQL stores. The first database, corresponding to a document store, denoted `docDB`, and is used in a patient oriented application while the other database, a column family store, denoted `colDB`, contains information aimed at health care professionals. In this paper, we concentrate on some extracts of `docDB` and `colDB` which correspond to respectively Figures 2 and 3. The data stored in both databases present some overlapping as well as some discrepancies both at the tuple and schema level. For instance, at the schema level, both databases contain french drug identifiers, names, pharmaceutical companies and prices but only `colDB` proposes access to the composition of a drug product. Considering the tuple level, some drugs may be present in one database but not in the other. Moreover information concerning the same drug product (i.e. identified by the same identifier value) may contradict themselves in different sources. Given these source databases, the target schema is defined as follows. We consider that relation and attribute names are self-explanatory.

```
drug(drugId, drugName, lab, composition, price)
therapDrug (drugId, therapeuticName)
```

Obviously, our next step it to define correspondances between the sources and the target. This is supporteed by mapping assertions which are currently being defined manually by domain experts. In the near future, we aim to discover some of them automatically by analyzing extensions and intensions of both sources and the target. Nevertheless, we do not believe that all mapping assertions can be discovered automatically due to the lack of semantics contained in both the target and the sources. Next, we present the mapping language enabling the definitions of mapping assertions.

### 4.1 Mapping language

Our data integration system takes the form of a triple $\langle \mathcal{T}, \mathcal{S}, \mathcal{M} \rangle$ where $\mathcal{T}$ is the target schema, $\mathcal{S}$ is the source schema and $\mathcal{M}$ is the mapping between $\mathcal{T}$ and $\mathcal{S}$. In Section 1, we motivated the fact that the set $\mathcal{S}$ could correspond to both RDBMS and NoSQL stores and that $\mathcal{T}$ takes the form of a relational schema. We consider that the target schema is given, possibly defined by a team of domain experts or using schema matching techniques [7].

This mapping language adopts a GAV (Global As View) approach with sound sources [5]. The mapping assertions are thus of the following form: $\phi_{\mathcal{S}} \rightsquigarrow \phi_{\mathcal{T}}$ where $\phi_{\mathcal{S}}$ is a query over $\mathcal{S}$ and $\phi_{\mathcal{T}}$ is a relation of $\mathcal{T}$.

Our system must deal with the heterogeneity of the sources and the highly denormalized aspect of NoSQL database instances. In order to cope with this last aspect, our mapping language handles the different access paths proposed by a given source. This is due to the important performance differences one can observe between the processing of the same query through different access paths. For instance, in the context of `colDB`, retrieving the drug information from a drug name will be more effective using the `drugCName` column family rather than the `drugC` column family (which would require complete scan of all its tuples).

We believe that a mapping assertion corresponds to the ideal place to store the preferred access paths possible for a target relation. Hence each mapping assertion is associated with a list of attributes contained in its target relation. The mapping language enables the use of the '*' symbol which, like in SQL, denotes the complete list of attributes of a given relation. For a given mapping assertion, an access path with attribute 'a' is defined when the source entity offers an efficient access, either using a key or an index, to a collection, set of columns or tuple. Note that for a source corresponding to an RDBMS, the definition of access paths is not necessary since computing the most effective query execution plan will be performed by the system. Hence, definitions of access paths are mandatory only for mapping assertions whose right hand side corresponds to a NoSQL database.

**Definition 1:** General syntax of a mapping assertion with an access path specification on attribute 'a': `RelationT(a,b,c)` $\overset{\leftarrow}{a}$ `EntityS`$(< key; value >)$ where `RelationT` and `EntityS` respectively denote a relation of the target and a conjunction of collections, column families or relations of a source. In this mapping assertion, the attributes of `RelationT` follow the definition order of this relation. Due to the schema flexibility of NoSQL databases, we can not

rely upon any attribute ordering in a collection or column family. Hence, we must use attribute names to identify distinct portions of a tuple. In order to map `RelationT` and `EntityS` attributes, we introduce a '`AS`' keyword to define a correspondence between attribute symbols of the mapping assertion. Finally, an entry in `EntityS` is defined as a key/value structure using a `'<key ; value>'` syntax, where `key` is either (i) '`PKEY AS k`' or (ii) a variable name (previously defined in a `EntityS` couple of the same mapping) and `value` is either of the form (i) `nameS AS nameT` (where nameS and nameT are resp. attribute names from the source and the target) or (ii) of the form `FOREACH item AS name IN list` (where `item` corresponds to an element of the set denoted by `list` and `name` is an attribute identifier of the source). Finally, a keyword is introduced to denote the primary key of the structure (i.e. '`PKEY AS`') and to manipulate it, e.g. `IN KEY`.□

Note the possibility that some target relation attributes are never associated to a mapping assertion. This means that there is not an efficient way to filter a query by this attribute due to the denormalization of the source databases. For instance, a query searching for a given drug composition will be highly inefficient in the `colDB` database due to the absence of a predefined structure proposing a key-based access from this attribute. Finally, for a given source and target relation, there must be a single mapping assertion with a given attribute.

A second feature of our mapping language consists in handling the data structures of NoSQL databases that can be multivalued, nested and also contain some valuable information in the key of a key/value structure; e.g. in the `DrugNameC` column family of Figure 3, drug identifiers of a drug product are stored in the key position (i.e. left hand side of the record). A multivalued example is present in both `docDB` and `colDB` extracts for the `therap` attribute. This forces our mapping language to handle access to the information associated to these constructors, e.g. to enable iteration over lists. Nested attributes are handled by using the standard '.' notation found in object oriented programming. On the second hand, iterations over lists require the introduction of a '`FOREACH`' construct.

We now present the mapping assertions of our running example (a.p. denotes an access path):

1. $drug(i, l, n, c, p)$ $\overleftarrow{*}$ $drugD$(<PKEY AS i ; name AS n,
   lab AS l, price AS p>)
2. $drug(i, n, l, c, p)$ $\overleftarrow{i}$ $drugC$(<PKEY AS i ; name AS n
   lab AS l, compo AS c, price AS p>)
3. $drug(i, n, l, c, p)$ $\overleftarrow{n}$ $drugNameC$(<PKEY AS n ;
   FOREACH id AS i IN KEY>,
   $drugC$(<id ; lab AS l, compo AS c, price AS p>)
4. $therapDrug(i, t)$ $\overleftarrow{i}$ $drugD$(<PKEY AS i ; FOREACH the AS t IN Therap>)
5. $therapDrug(i, t)$ $\overleftarrow{t}$ $therapD$(<PKEY AS t ; FOREACH id AS i IN Drugs>)
6. $therapDrug(i, t)$ $\overleftarrow{i}$ $DrugC$(<PKEY AS i ; FOREACH the AS t IN Therap>)
7. $therapDrug(i, t)$ $\overleftarrow{t}$ $therapC$(<PKEY AS t ; FOREACH id AS i IN KEY>

This set of mapping assertions examplifies an important part of our mapping language features:

- assertion #1 has a '*' access path since we consider that all attributes of the `drugC` collection are indexed. Also note that on this mapping assertion, the `c` attribute is not mapped to any source attribute since that information is not available in the `docDB` database.
- Mapping assertion #3 introduces the fact that several source entities can be used in a single mapping (i.e. `drugNameC` and `drugC` column families). Intuitively, this query accesses a given `drugNameC` column family entry identified by a drug name and iterates over its drug identifiers, which are keys (using the '`IN KEY`' expression) then it uses these identifiers to access entries in to `drugC` column family (using `i` iterator variable in the key position of the `drugC`).

### 4.2 Dealing with conflicting data using attribute preferences

In general, data integration and data exchange solutions adopt the certain answers semantics for query answering, i.e. results of a query expressed over the target contain the intersection of data retrieved from the sources. We believe that this pessismistic approach is too restrictive and as a consequence, many valid results may be missing from final results.

At the other extreme of the query answering semantics spectrum, we find the possible answer semantics which provides as results over a target query the union of sources results. With this optimistic approach conflicting results may be proposed as a final result, leaving the end-users unsatisfied.

In this work, we propose a trade-off between these two semantics which is based on a preference-based approach. Intuitively, preferences provided over target attributes define a partial order over mapped sources. Hence, for a given data object, conflicting information on the same attribute among different data sources can be handled efficiently and the final result will contain the preferred values.

**Example 4:** Consider the queries over `docDB` and `colDB` asking for `lab` and `price` information for the drug identified by value 3295935. Given the information stored in both sources, respectively the column store (Figure 2) and column family store (Figure 3), conflicts arise on the prices, resp. 1.88 and 2.05 euros, and pharmaceuticals, resp. Pfizer and Wyeth.

When creating the mapping assertions, domain experts can express that drug prices are more accurate in the document store (`docDB`) and that information about pharmaceutical laboratory is more trustable in the column family (`colDB`). Hence the result of this query will contain a single tuple consisting of: {`Advil, Wyeth, 1.88`}, i.e. mixing values retrieved the different sources.

We now define the notion of preferences over mapping assertions.

**Definition 2:** Consider a set of source databases $\{DB_1, DB_2, ..DB_n\}$, a preference relation, denoted $\succ$, is a relation $\succ \subseteq DB_i \times DB_j$, with $i \neq j$, that is defined on each non primary key attribute of target relations. A preference $\succ$

is total on an attribute A if for every pair $\{DB_i, DB_j\}$ of sources that propose attribute A, either $DB_i \succ^* DB_j$ or $DB_j \succ^* DB_i$ with $\succ^*$ the transitive closure of $\succ$.□

**Example 5:** Consider the `drug` relation in our running example target schema. Its definition according to the preferences proposed in Example 3 are the following: `drug(drugId, drugName`$^{docDB \succ colDB}$`, lab`$^{colDB \succ docDB}$`, composition, price`$^{docDB \succ colDB}$`)`

That is, for a given drug, in case of conflict, its `docDB drugName` attribute is preferred to the one proposed by `colDB` and the preferred value for the `lab` attribute is `colDB` over `docDB`. Note that since the composition attribute can only be retrieved from the `colDB` source, it is not necessary to define a preference order over this attribute.

## 5 Query processing

Once a target relation schema and a set of mapping assertions have been defined, end-users can expressed queries in SQL over the target database. Since that database is virtual, i.e. it does not contain any data, data needs to be retrieved from the sources and processed to provide a final result. The presence of NoSQL databases in the set of sources imposes to transform the former SQL query into a query specifically tailored to each NoSQL source. This transformation is based on the peculiarities of the source database, e.g. whether a declarative query language exists or only procedural approach enables to query that database, and the mapping assertions. Since most NoSQL stores support only a procedural query approach, we have decided to implement a query language to bridge the gap between SQL and some code in a programming language. This section presents the Bridge Query Language (henceforth BQL) which is used internally by our data integration system and the query processing semantics.

### 5.1 Architecture

The overall architecture of query processing within our data integration system is presented in Figure 4. First, an end-user writes an SQL query over the target schema. The expressivity of accepted SQL queries corresponds to `Select Project Join` (SPJ) conjunctive queries, e.g. `GROUP BY` clauses are not accepted but we are planning to introduce them in future extensions. Note that this limitation is due to a common abstraction of the NoSQL databases we are studying in this paper (column family and document).

An end-user SQL query is then translated into the BQL internal query language of our data integration system. This transformation corresponds to a rewritting of the SQL into a BQL query using the mapping assertions. Note that this translation step is not needed for a RDBMS.

Then for each BQL, a second transformation is performed, this time to generate a query tailoring the NoSQL database system. Thus, for each supported NoSQL implementation, a set of rules is defined for the translation of a BQL

query. Most of the time, the BQL translation takes the form of a program and uses a specific API. In Section 6, we provide details on the translation from BQL to Java programs into MongoDB and Cassandra.

The results obtained from each query is later processed within the data integration system. Intuitively, each result set takes the form of a list containing the awaited target columns. In order to detect data conflicts, we need to efficiently identify similar objects. This step is performed by incorporating into the result set values corresponding to primary keys of target relations of the SQL query. So, even if primary keys are not supposed to be displayed in the final query result, they are temporarily stored in the result set. Hence objects returned from the union of the result sets are easily and unambiguously identified. Similar objects can then be analyzed using the preference orders defined over target attributes. The query result contains values retrieved from the preferred source attributes.

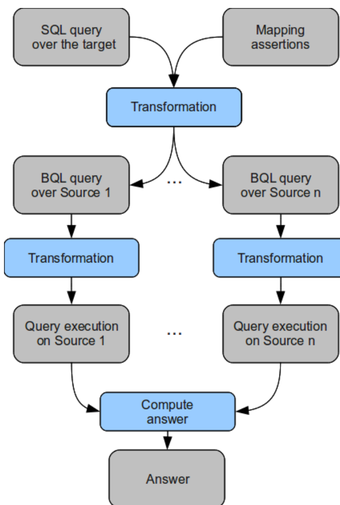


**Fig. 4.** Query processing

### 5.2 Bridge Query Language

BQL is the internal query language that bridges the gap between the SQL language of the target model and the different and the heterogenous query languages of the sources. The syntax of the query language follows the EBNF proposed in the companion web site. This language contains a set of reserved words whose semantics is obvious for a programmer. For instance, the `get` instruction enables to define a set of filter operations and to define the distinguished variables of the query, i.e. the values needed in the result. The `foreach in :` instruction is frequently encountered in many programming languages and their semantics

align. Intuitively, it supports an iteration over elements of a result set and the associated processing is performed after the `':'` symbol. We have implemented an SQL to BQL translator which parses an SQL query and generates a set of BQL queries, one for each NoSQL database mapped to the relation of the target query. This translator takes into account the mapping assertions defined over the data integration system.

**Example 6:** We now introduce a set of queries expressed over our running example. They correspond to different real case scenario and emphasize the different functionalities of our query language. For each target query (SQL), we present the BQL generated for both `colDB` and `docDB`.

- Query 1 accesses a single table via its primary key.
  SQL: `SELECT drugName, price FROM drug WHERE drugId=3295935;`
  docDB's BQL: `ans(drugName, price) = docDB.drugD.get({PKEY= 3295935},{name, price})` provides answer (Advil, 1.88)
  colDB's BQL: `ans(drugName, price) = colDB.drugC.get({PKEY= 3295935},{name, price})` provides answer (Advil, 2.05)
  Answer: Since the query identifies tuples with the primary key, the real world object of the answers is supposed to be the same and we apply the preferences over the union of the results. The processed result is (Advil, 1.88)
- Query 2 access single table over a non primary key but indexed attribute of the target.
  SQL: `SELECT drugId, price FROM drug WHERE drugName LIKE 'Advil';`
  docDB's BQL: `ans(drugId, price) = docDB.drugD.get({name='Advil'}, {PKEY, price})` with answer {(3295935, 1.88)}
  colDB's BQL: `temp(drugId) = colDB.drugNameC.get({name='Advil'}, {KEY})`
  `ans(drugId, price) = foreach id in temp(drugId):colDB.drugC.get( {KEY=id},{KEY, price}) colDB.drugC` with {(3295935, 2.05),(3575994, 2.98)}
  Answer: The final result set is {(3295935, 1.88),(3575994, 2.98)} thus mixing the results and taking advantage of the preference setting.
- Query 3 retrieves data from a single relation with a filter over a non-primary and non-indexed attribute of the target.
  SQL: `SELECT drugName FROM drug WHERE lab='Bayer';`
  docDB's BQL: `ans(drugName) = docDB.drugD.get({lab='Bayer'},{name})`
  colDB's BQL: No solution
  Answer: Since no queries are generated for the `colDB` store, the results are retrieved solely from `docDB`.
- Query 4 involves 2 relations and an access from a single primary key attribute of the target.
  SQL: `SELECT drugName FROM drug d, therapDrug td WHERE d.drugId=td.drugId AND therapId LIKE 'NSAID';`
  docDB's BQL: `temp(drugs) = docDB.therapD.get({PKEY='NSAID'},{drugs})`
  `ans(drugName) = foreach id in temp(drugs) : docDB.drugC.get( {PKEY=id},{name})`
  colDB's BQL: `temp(drugs) = colDB.therapC.get({PKEY='NSAID'},{KEY})`

```
ans(drugName)=foreach id in temp(drugs) :
coldDB.drugC.get({PKEY=id},{name})
```
Answer: provides the same result as in Query 2.

## 6 Implementation

In this section, we sketch our prototype implementation which tackles a document store (MongoDB) and a column store (Cassandra). Together they represent some of the most popular open source projects in the NoSQL ecosystem. The platform we have adopted corresponds to Java since both MongoDB and Cassandra propose APIs and enable the execution of this programming language. Moreover, Java is adapted to numerous other NoSQL stores. Nevertheless, in the near future, we are planning to tackle other systems, e.g. CouchDB or HBase, and consider other access methods, e.g. javascript or python.

An important task of our prototype is to handle the transformation modules found in Figure 4. That is to process the translation (i) from SQL to BQL and (ii) from BQL to the query language supported by each NoSQL stores. Due to the declarative nature of both query languages of the former translation, this task is easy to implement and is implemented in linear time on the length of the input SQL query. The latter translation task is more involved since BQL corresponds to a declarative language and the target query of our NoSQL stores corresponds Java methods. The high denormalization aspect of NoSQL stores imposes that only a limited set of queries can be efficiently processed. In fact, this results in having similarities between queries expressed over a given NoSQL database instance. We have extract these similarities into patterns which are implemented using Java methods. The main idea is to consider a set of finite BQL templates and to associate a Java method to each of these templates.

One can ask the following question: is this approach still valid and efficient when more complicated SQL queries, e.g. involving aggregate functions (min, max, etc.), group by and having or like constructors? A reply to this question necessarily needs to consider the particular features of each NoSQL database supported by the system since, up to now, a common framework for NoSQL stores does not exist. In the case of MongoDB, the support of regular expressions enables the execution of complex statements with minimal additional effort. On the other hand, Cassandra only supports a range query approach on primary keys. This results in having an inefficient support for aggregate operations and queries involving regular expressions.

## 7 Conclusions

This paper is a first approach to integrate data coming from NoSQL stores and relational databases into a single virtualized database. Due to the increasing popularity of this novel trend of databases, we consider that such data integration systems will be quite useful in the near future. Our system adopts a relational approach for the target schema which enables end-users to express queries in

the declarative SQL language. Several transformation steps are then required to obtain results from the data stored at each of the sources. Hence a bridge query language has been presented as the cornerstone of these transformations. Another important component of our system is the mapping language which (i) handles uncertainty and contradicting information at the sources by defining preferences over mapping assertions and (ii) supports the setting of access path information in order to generate an efficiently processable query plan.

On preliminary results, the overhead of these transformation steps does not impact the performance of query answering. Our list of future works is important and among others, it contains the support of NoSQL stores corresponding to a graph model and the (semi) automatic discovery of mapping assertions based on the analysis of value stored in each source.

## References


1. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data (awarded best paper!). In *OSDI*, pages 205–218, 2006.
2. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
3. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.
4. M. Kifer, A. Bernstein, and P. M. Lewis. *Database Systems: An Application Oriented Approach, Complete Version (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.
5. M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246, 2002.
6. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM.
7. E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10:334–350, December 2001.
8. M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.
9. A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.