A Demonstration of NoDA: Unified Access to NoSQL Stores

Nikolaos Koutroumanis
Dept. of Digital Systems
University of Piraeus
Piraeus, Greece
koutroumanis@unipi.gr

Nikolaos Kousathanas
Dept. of Digital Systems
University of Piraeus
Piraeus, Greece
nikolaos.kousathanas@gmail.com

Christos Doulkeridis
Dept. of Digital Systems
University of Piraeus
Piraeus, Greece
cdoulk@unipi.gr

Akrivi Vlachou
Dept.of Inf. & Com.Syst.Engineering
University of Aegean
Karlovasi, Greece
avlachou@aegean.gr

ABSTRACT

In this demo paper, we present a system prototype, called NoDA, that unifies access to NoSQL stores, by exposing a *single* interface to big data developers. This hides the heterogeneity of NoSQL stores, in terms of different query languages, non-standardized access, and different data models. NoDA comprises a layer positioned on top of NoSQL stores that defines a set of basic data access operators (filter, project, aggregate, etc.), implemented for different NoSQL engines. The provision of generic data access operators enables a declarative interface using SQL as query language. Furthermore, NoDA is extended to provide more complex operators, such as geospatial operators, which are only partially supported by NoSQL stores. We demonstrate NoDA by showcasing that the exact same query can be processed by different NoSQL stores, without any modification or transformation whatsoever.

1 INTRODUCTION

NoSQL stores [1, 2] support scalable data access using flexible data models, thus comprising the storage backend of choice for several modern applications and services. However, despite their popularity, NoSQL stores still rely on heterogeneous languages, different (non-standardized) data models, and properties. In turn, this delays the development of big data applications, since developers need to get acquainted with different languages, a factor that also hinders the portability of an application to a different NoSQL storage engine. This is in complete contrast with relational DBMSs

which use a common schema that allows for a clear separation and independence between application and storage.

Motivated by this limitation, we present the system architecture of NoDA [8], a prototype for unified data access to NoSQL stores. NoDA acts as an intermediate abstraction layer between application code and the underlying NoSQL store. Thus, NoDA offers a set of generic data access operators (e.g., filter, project, sort) using a familiar vocabulary for developers. These operators are internally implemented for different stores, thus hiding the heterogeneous languages from the developer. One advantage of using these operators is that the same application code becomes portable across NoSQL stores. In addition, we exploit the data access operators of NoDA, in order to provide a declarative interface (in SQL). In the background, SQL-like queries are translated in NoDA operators, which facilitates query execution over NoSQL stores that do not support SQL. As a result, data scientists and business analysts can query different NoSQL stores using a standardized query language.

Related work. Multi-model databases support different data models against a single, integrated backend [9]. ArangoDB integrates document, key-value and graph data models in a single system. It uses AQL as its query language which also supports geospatial functions for querying spatial data. OrientDB combines document, key-value, reactive and object-oriented models in a single system. In contrast to these systems, NoDA is designed to be used *on top of* any existing NoSQL store. Our work also relates to polystores, database management systems that are built on top of different, heterogeneous, integrated storage engines, e.g., Big-DAWG [3, 4] and semantic approaches [6]. Such systems offer a common (SQL-like) language like CloudMdsQL [7] for accessing data from multiple stores, hiding any peculiarities of the targeted databases. The integration of different databases is attained by using the native mechanisms of each database for querying, as in NoDA. Nonetheless, the main difference between polystores and NoDA is that polystores constitute composite systems whose components are orchestrated under a specific context for query optimization, execution, monitoring, etc. Their architecture is much more complex than NoDA, which makes extensibility to a new storage engine difficult. Instead, NoDA adopts a more lightweight approach where the incorporated components translate and execute the query on the underlying NoSQL store by using its native mechanisms. Last, but not least, GeoMesa provides spatio-temporal indexing [5] over
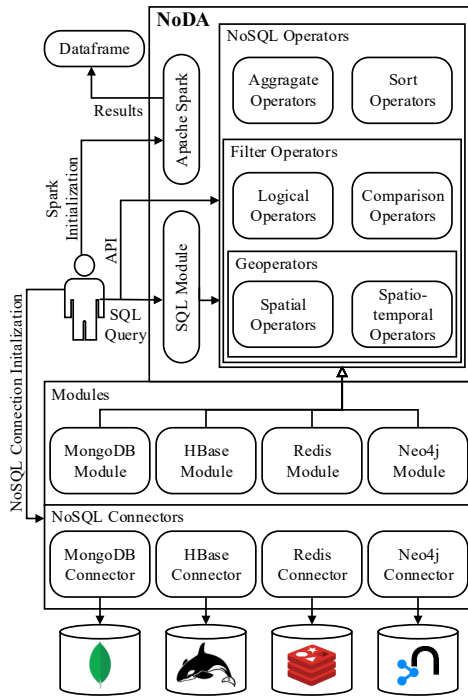
Figure 1: System architecture of NoDA.

persistent storage, while supporting different backends (e.g., Accumulo, Redis, HBase, Bigtable and Cassandra).

Contributions. In this demo paper, we present the system prototype of NoDA that: (i) supports the execution of simple and complex queries over *different* NoSQL stores by means of a set of generic data access operators, (ii) offers the capability of using declarative SQL querying, and (iii) visualizes the results on a map-based interactive web interface, for complex (spatio-temporal) queries.

2 THE NODA SYSTEM ARCHITECTURE

Figure 1 illustrates the high-level system architecture of NoDA. Its main constituent parts include: (i) the *NoSQL Connector*, which is responsible for establishing and managing the connection to a NoSQL store, (ii) the *NoSQL Operators*, which specify the abstraction of data access operators that need to be implemented as *modules* for different stores, (iii) an *Apache Spark session*, which facilitates the delivery of data in the form of *Dataframe* within a Spark context for further processing, and (iv) the *SQL module*, which allows declarative query formulation and its seamless translation to data operators that will be eventually executed. Consequently, the user has two options for querying a NoSQL store with NoDA: either by writing code using the generic data access operators (suitable for application developers), or by means of SQL queries (preferable for data scientists and business analysts).

2.1 NoSQL Connectors

NoDA provides functionality for establishing connections to NoSQL stores. For each NoSQL store, we develop a corresponding NoSQL

connector, which is initialized with information for establishing the connection. Although the exact information type differs among NoSQL stores, this typically includes the host's IP address, running port and database credentials (username, password).

A connector can be extended for encapsulating a specific type of information that is not supported in another NoSQL database. We use an object `NoSqlDbSystem` which corresponds to the initialized connection. For instance, Listings 2.1 and 2.2 show a connection setup for Redis and Neo4j respectively. The connection to Redis store is attempted with a threshold that represents the maximum waiting time of the client in case the connection is not available. Instead, for the connection establishment to Neo4j, a Base64 encoded ticket is set, passed in the kerberos authentication mechanism. Note that the JedisPoolConfig and AuthTokens objects in the corresponding Listings, are object types derived from the native Java client libraries of these databases.

```
Listing 2.1: Setting a connection to Redis store.

1  JedisPoolConfig jp = new JedisPoolConfig();
2  jpc.setMaxWaitMillis(10000);
3  NoSqlDbSystem noSqlDbSystem = NoSqlDbSystem.Redis()
4        .Builder(jp).host("192.168.1.1").port(6379).build();
```

```
Listing 2.2: Setting a connection to Neo4j store.

1  String ticket = ...
2  NoSqlDbSystem noSqlDbSystem = NoSqlDbSystem.Neo4j()
3        .Builder(AuthTokens.kerberos(ticket))
4        .host("192.168.1.1").port(7687).build();
```

2.2 NoSQL Operators

NoDA is designed to support generic data access operators, supporting both simple relational expressions as well as complex ones.

2.2.1 Simple Operators. These operators include `filter`, `project`, `sort`, `limit` and `aggregate`. The `filter` operators consist of three subcategories: logical, comparison and geo-operators. The logical operators are used to combine filter expressions in general. The comparison operators provide the fundamental filtering conditions (e.g., equality, inequality, etc.) on fields that handle either alphanumeric or numeric type values. The geo-operators provide filtering conditions that refer to spatial and temporal information. As a result, they can be exploited to apply spatial and spatio-temporal constraints. Regarding data types, we support numerical and alphanumerical data types, as well as more complex data types, such as 2D/3D spatial data types for geographical data.

Moreover, the `sort` operator applies a specific order on the records, based on a field or some fields. The `limit` operator restricts the number of records in the result set of a query based a user-specified value. The `aggregate` operators incorporate functions so as to support specific aggregations on a number of field(s). Apart from the aforementioned operators, additional operations on data are also supported. Concretely, the `project` operator intervenes in the shape of the fetched data when accessed, indicating the fields that are to be included in the query's results.

```
1  noSqlDbSystem.operateOn("hotels")
2              .filter(and(
3                  eq("star", 5), lte("price_per_day", 140)))
4              .count();
```

As an example, Listing 2.3 depicts a plain query on collection "hotels", which is specified by `operateOn`. The query retrieves the number of objects having 5 stars and price at most 140, using `filter`, `count` and constraints `eq` and `lte`. The operators are combined together using method chaining.

2.2.2 Complex Operators. The complex operators currently supported by NoDA include geo-operators, that enable filtering of spatial and spatio-temporal data. Unfortunately, not all NoSQL stores support spatial queries, whereas most stores do not support spatio-temporal queries. Therefore, we adopt the concept of space-filling curves to map the spatial information to 1D values.

In the case of spatial data, a mapping of locations to 1D values is achieved based on the Hilbert space-filling curve. Practically, the 2D space is partitioned in grid cells, and each cell is mapped to a 1D *key* based on its order on the space-filling curve. In this way, spatial objects are mapped to keys, and can be stored in *any* NoSQL store that supports key-based access. Considering that MongoDB and Neo4j provide built-in indexes for a single field and property accordingly, we exploit these indexes for efficient access to the generated keys. In the case of HBase, we inject the 1D value in the row key, in order to exploit its support for efficient range scans based on key. In the case of Redis, which is a key-value store, we put the 1D values in a sorted set, which serves as index for efficient retrieval of object identifiers in logarithmic time.

```
1  Coordinates c1 = Coordinates.newCoordinates(23.69, 37.93);
2  Coordinates c2 = Coordinates.newCoordinates(23.81, 38.01);
3  int count = noSqlDbSystem.operateOn("movingObjects")
4              .filter(inGeoRectangle("location", c1, c2))
5              .count();
```

Listing 2.4 depicts the code for a spatial box query, which retrieves spatial objects from collection "movingObjects" by applying the `inGeoRectangle` operator on field "location".

In the case of spatio-temporal data, the Hilbert-based approach is used too. In MongoDB, we build a compound index over the field that stores the Hilbert key together with the field that stores the temporal information. This enables spatio-temoral sharding which is not supported yet by the built-in spatial indexes (ver. 5.0). An alternative approach is adopted for Neo4j. Given a spatio-temporal point, an index key is generated by taking into account both the space and time information. This corresponds to 3D partitioning, where each cell refers to a specific spatial cell within a time interval. For HBase and Redis, the approach followed is similar to the case of spatial (2D) data.

2.2.3 Extensibility. Our prototype NoDA is readily extensible for operating over other NoSQL stores. The data access operators are defined in interfaces and abstract classes, whose functionality can be instantiated for different stores. In fact, they constitute a blueprint, which is implemented by developing a new module for each NoSQL store. The blueprint can be extended for operating on a specific store, through the Java or Scala client library of the NoSQL store. NoDA has been designed with the purpose of being extensible with limited effort. This is achieved by the integration of a new module under the NoDA project, which inherits from the core module the template. The template is filled with the appropriate commands, in the language of the new NoSQL store. In particular, the `filter`, `aggregate` and `sort` operators should be materialized by the database-oriented methods which apply the rationale of the operators on the store. For instance, for the case of MongoDB, the underlying methods that are utilized by the comparison operators are the respective operators of MongoDB (`$gte`, `$lte`).

2.3 Spark Session

Data access from NoSQL stores may result in large result sets, which require distributed processing. For this purpose, we integrate NoDA with a data-parallel processing framework, in order to enable parallel processing. To this end, we opt to use Apache Spark and its *Dataframe* abstraction. In more detail, the NoSQL connectors are able to encapsulate a Spark session, which can be used for loading the query results in Dataframe form. A Dataframe is a distributed collection of data composed of rows with typed columns, similar to a table. In the context of this demonstration, the Spark session also facilitates the transformation of data for visualization, after having been fetched from the data store. As the user interface presented in Section 3 requires a standardized form for visualizing the spatio-temporal information, we use the Dataframe for this purpose.

2.4 SQL Module

Even though NoDA provides a query language based on generic data access operators that are easy to learn, it is still yet another query language. To alleviate this shortcoming, we introduce a declarative interface based on a SQL-like language. This is in accordance with the current trend of providing declarative interfaces for big data processing and storage solutions (e.g., Hive, Pig, SparkSQL, Presto, etc.).

The SQL module takes an SQL statement as input, which is parsed to validate its syntax. The SQL module is based on the *ANTLR* tool (https://www.antlr.org/), which is used for accessing languages specified by a grammar. More specifically, given a grammar as an input, ANTLR generates the necessary source code (parser, tree parser, listener interface), which facilitates the recognition of phrases. The source code is the core of the SQL module, having been generated given the SQL grammar file of Presto (https://prestosql.io/).

After successful parsing and validation, the clauses are read individually, and the SQL query is transformed into a sequence of NoSQL operators that will be executed. Although we have not implemented a query optimizer yet, we apply some basic rules during this transformation. For example, we apply selection and projection push-down to avoid processing unnecessary records in the pipeline.

3 DEMONSTRATION SCENARIO

The web user interface runs as an application on Spring Boot and has been developed in the Angular web framework. It also uses the Leaflet Javascript library for the interactive map. Figure 2 shows
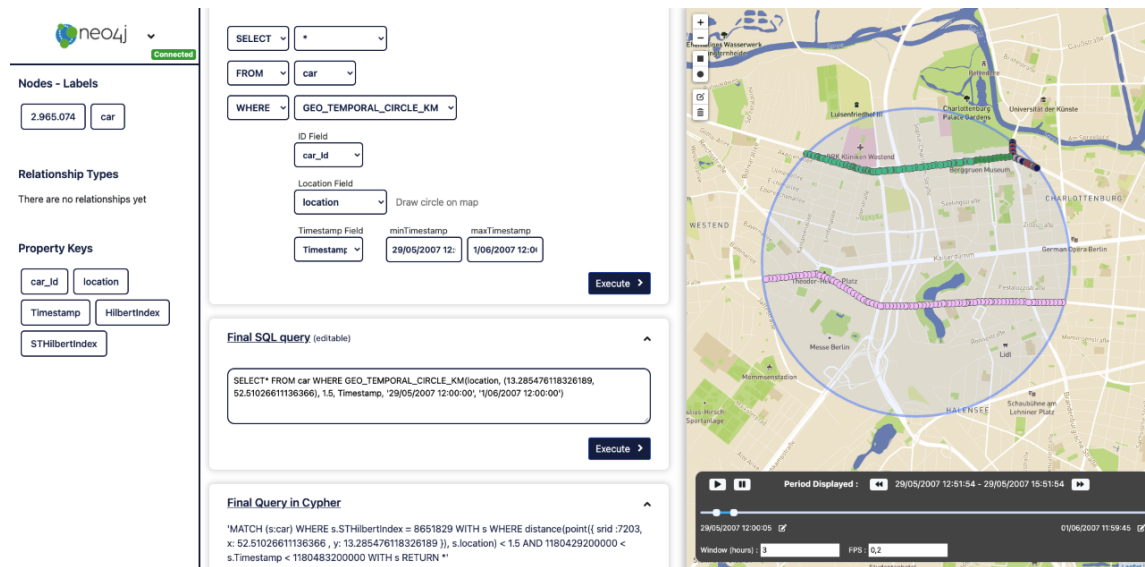
Figure 2: The graphical user interface of NoDA **for geospatial queries: connection details for the NoSQL store and database contents (left), query builder for easier query formulation, SQL editor, and illustration of the final query to be executed in the language of the NoSQL store (middle), map-based visualization for query results (right).**

the GUI of the demonstration, focusing on geospatial and spatio-temporal queries. It consists of three main parts: (a) the targeted NoSQL store, (b) the query builder, and (c) the map visualization.

Initially, the user can use the combo box (top left) to select the NoSQL that will be queried using NoDA (in the figure, Neo4J is selected). The user can switch easily from one store to another via the combo box, and the query will be sent to the corresponding store. In the background, NoDA establishes a connection to the selected NoSQL store and fetches some information regarding the data. In the depicted example, nodes and properties from Neo4J are shown on the screen, in order to assist the user during query formulation. As soon as the connection has been established, the user can formulate a query in two different ways: by means of the query builder or by writing a SQL query. The query builder currently supports a subset of the possible queries, mainly focusing on geospatial filter/project operators. However, the SQL editor supports the full repertoire offered by the declarative interface of NoDA. Also, note that the spatial constraint can be set by drawing a circle on the map, and this updates the query editor. When the execute button is pressed, the formulated SQL query is transformed to the query language of the NoSQL store. This final query is shown at the bottom for clarity.

Last, but not least, the retrieved results are rendered on the map, which supports basic functionality, such as zooming. Also, in case of spatio-temporal data (e.g., trajectories of moving objects), the GUI is equipped with a "player" that can be used to show how the locations change with time, thereby showing the movement of objects on the map.

4 CONCLUSIONS AND FUTURE WORK

In this paper, we presented a system prototype of NoDA that provides access to different NoSQL stores using a set of generic data access operators, thus hiding the heterogeneity of different stores. Also, we demonstrate the use of a declarative interface built on top of the operators. The design of our prototype is extensible to support other NoSQL stores in the future. In our future work, we intend to extend our prototype to support a larger variety of NoSQL stores. Furthermore, we will explore query processing over multiple NoSQL stores (in a polystore setting), by combining data from different stores.

REFERENCES

[1] Rick Cattell. 2010. Scalable SQL and NoSQL data stores. *SIGMOD Record* 39, 4 (2010), 12–27.
[2] Ali Davoudian, Liu Chen, and Mengchi Liu. 2018. A Survey on NoSQL Stores. *ACM Comput. Surv.* 51, 2 (2018), 40:1–40:43.
[3] Jennie Duggan et al. 2015. The BigDAWG Polystore System. *SIGMOD Record* 44, 2 (2015), 11–16.
[4] Aaron J. Elmore et al. 2015. A Demonstration of the BigDAWG Polystore System. *Proc. VLDB Endow.* 8, 12 (2015), 1908–1911.
[5] Anthony D. Fox, Christopher N. Eichelberger, James N. Hughes, and Skylar Lyon. 2013. Spatio-temporal Indexing in Non-relational Distributed Databases. In *Proc. of IEEE Big Data.* 291–299.
[6] Evgeny Kharlamov et al. 2016. A semantic approach to polystores. In *Proc. of IEEE Big Data.* 2565–2573.
[7] Boyan Kolev et al. 2016. CloudMdsQL: Querying Heterogeneous Cloud Data Stores with a Common Language. *Distributed Parallel Databases* 34, 4 (2016), 463–503.
[8] Nikolaos Koutroumanis et al. 2019. NoDA: Unified NoSQL Data Access Operators for Mobility Data. In *Proc. of SSTD.* 174–177.
[9] Jiaheng Lu and Irena Holubová. 2019. Multi-model Databases: A New Journey to Handle the Variety of Data. *ACM Comput. Surv.* 52, 3 (2019), 55:1–55:38.