Tearing Down the Tower of Babel: Unified and Efficient Spatio-temporal Queries for NoSQL Stores

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

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*—NoSQL stores are used extensively for scalable storage and efficient querying of large spatio-temporal data collections in modern applications. Yet, despite their popularity, NoSQL systems have two main limitations when confronted with spatio-temporal data: (a) they do not offer optimized indexing methods, and (b) they still rely on heterogeneous languages and lack of standardization in data access, a situation bearing resemblance to the narrative of the tower of Babel. To address these limitations, we propose** NoDA**, a system for scalable querying of spatio-temporal data stored in different NoSQL stores in a unified way.** NoDA **relies on an abstraction layer that consists of data access operators with clear semantics, that provides a *unified* view of the underlying NoSQL stores. Furthermore,** NoDA **offers spatio-temporal operators that are internally implemented in an efficient way, by taking into advantage the individual features of each NoSQL store. Capitalizing on the query operators,** NoDA **provides a declarative interface based on a SQL-like language, allowing users to query different NoSQL stores using SQL. Our experiments demonstrate that** NoDA **significantly improves the performance of spatio-temporal querying over different types of NoSQL stores.**

***Index Terms*—NoSQL, spatio-temporal queries**

I. Introduction

NoSQL stores [1, 2] are increasingly adopted by modern applications and enterprises for scalable storage and efficient querying over vast data collections. Their advantages include support for flexible and schemaless data models [2], high availability and scalability, as well as faster time to market. As a result, they are extensively used for querying large spatio-temporal data collections and for mobility analytics [3].

Nevertheless, despite the popularity of NoSQL stores for scalable querying, two major limitations come up when confronted with spatio-temporal data: (a) they do not support optimized indexing methods, and (b) they use different languages for data access. Regarding indexing, most NoSQL stores offer (in the best case) limited support for spatial data (not spatio-temporal), which has an effect on the performance of spatio-temporal queries [4]. Therefore, it becomes imperative to propose optimized data access methods for spatio-temporal data that can be integrated in today's NoSQL stores.

Regarding different query languages, the current situation is that each NoSQL store provides its own query language or API. Consequently, big data developers need to acquire an in-depth understanding of the individual language/API of NoSQL stores and the supported features, in order to integrate NoSQL stores into their architectures. Perhaps more importantly, this hinders the seamless transition from one NoSQL store to another, as the application code is tightly coupled to the selected NoSQL store. In turn, this contrasts application development in relational databases, which is greatly facilitated by standardized data access, e.g., using ODBC/JDCB.

Motivated by these shortcomings, in this paper, we propose NoDA (**No**SQL **D**ata **A**ccess Operators), a system for unified and efficient querying of spatio-temporal data over different NoSQL stores. NoDA relies on an abstraction layer that consists of simple data access operators with clear semantics that can be easily combined by method chaining to form complex expressions. To further improve usability, we build a declarative SQL-like interface on top of NoDA's data access operators, thereby lifting the limitation of having to learn yet another language. In addition, NoDA incorporates spatio-temporal operators (range, k-NN) for querying data with spatial and temporal constraints. Even though it is not trivial to implement spatio-temporal queries efficiently for different NoSQL stores, we present design choices tailored for each category of NoSQL stores that boost the performance of query processing.

In summary, the contributions of this work include:

- We propose NoDA, an *abstraction layer* that consists of data access operators that provide unified access to NoSQL stores, as well as a declarative interface on top of the data access operators, which practically allows using SQL to query different NoSQL stores (Sect. III).
- We present the design and implementation of the spatio-temporal operators of NoDA over three widely different NoSQL stores (a document-oriented store, a wide-column store and a key-value store) that use diverse data models and languages (Sect. IV).
- We report on the performance gains of NoDA, by evaluating experimentally spatio-temporal queries over three NoSQL stores (MongoDB, HBase, and Redis) (Sect. V).

In addition, we review related work in Sect. II and we conclude in Sect. VI. The preliminary concept of NoDA has been demonstrated in [5, 6]. In this paper, we unveil the mature design and implementation of NoDA over three NoSQL stores

that belong to different categories, we present the declarative interface, and we provide an in-depth study of the spatio-temporal queries of NoDA.

II. Related Work

Scalable spatial data stores. Several research prototypes target scalable storage and indexing of spatial data, by exploiting a NoSQL store. However, most of these systems are tightly integrated with one specific NoSQL store [3], e.g., Pyro [7] is built on top of HBase, SECONDO's distributed version [8, 9] employs Cassandra, etc. In contrast, GeoMesa provides spatio-temporal indexing [10] over persistent storage, while supporting different backends (e.g., Accumulo, Redis, HBase, Bigtable and Cassandra). However, a critical difference is that NoDA exposes specific data access operators to the developer and a declarative language, thus aiming at easy integration with application code. Our work is also related to in-memory processing systems for spatial data, such as GeoSpark [11], LocationSpark [12], DITA [13] and Beast [14]. However, these systems mainly target efficient *in-memory* processing and analysis of spatial data, which is complementary to our work. In fact, they could benefit from coupling with NoDA as underlying layer for efficient data access from different NoSQL stores.

SQL on NoSQL stores. The majority of NoSQL systems offer their own native query languages for data access. Lately, some NoSQL systems, such as CouchDB, ArangoDB, Couchbase, OrientDB and DynamoDB, additionally support data access via an SQL-like language, apart from their native language. Clearly, the trend is to empower NoSQL stores with SQL support, e.g., Google's Spanner [15], Pig Latin [16], Hive [17], SparkSQL [18] and Zidian [19]. NoSQLBooster is a complete IDE for MongoDB from which a user can issue SQL queries, which are translated to the native language of MongoDB for query execution. Apache Phoenix can be added on HBase for running an SQL query, which is translated to a series of scans on the store. However, all these approaches are optimized for individual NoSQL stores, and do not address the problem of diversity in data models nor language heterogeneity. Facebook's Presto [20] (recently known as Trino) is an SQL-compliant query engine that operates on a wide variety of different data sources, such as RDBMS, NoSQL and stream processing systems. However, NoDA supports spatial and spatio-temporal data, whereas Presto is not optimized for this domain.

Polystores/Multistores. Our work also relates to the category of database management systems that are built on top of different, heterogeneous, integrated storage engines, e.g., BigDAWG [21, 22], Icarus [23], CloudMdsQL [24] and Optique [25] (a semantic polystore). Such systems offer a standard query interface and users formulate queries using a common (SQL-like) language, such as the one described in [26] for managing data from multiple stores. This hides the peculiarities of the underlying stores from the users, offering a single view of the system. In comparison with polystores, NoDA also targets efficient and seamless data access across NoSQL stores, but it is not tightly integrated with the underlying stores. Thus, NoDA can be extended in a much easier way to new NoSQL stores, by simply implementing its API for the new store. Apache Wayang [27] is an abstraction layer that supports the execution of data processing tasks over multiple data processing platforms. It integrates a set of operators and an optimizer [28] for cross-platform query execution. So far, Wayang does not operate upon any NoSQL store, covering Postgres and SQLite relational databases.

III. The NoDA Abstraction Layer

In this section, we describe the NoDA abstraction layer for uniform access to different NoSQL stores. Its objective is to offer a developer-friendly abstraction, which can be exploited to provide *simple* and *unified* access to scalable NoSQL stores. *Simple*, in terms of using a familiar vocabulary of generic operations, without mixing the data model and the query language of the individual NoSQL store in the application code. *Unified*, because the exact same operations are used for querying different NoSQL stores.

A. Concept

Figure 1 provides a graphical illustration of the NoDA abstraction layer on top of NoSQL stores. NoDA resides between application code and data storage as a bridge for data access, and aims at "hiding" the query language of the underlying store from the developer. Essentially, a big data developer expresses her code using the NoDA abstraction and attains two benefits: (a) NoDA hides the peculiarities and complexity of accessing the specific NoSQL store, and (b) the exact same code can be used to query data stored in different NoSQL stores, no matter how different their query languages are. NoDA is readily implemented over MongoDB, HBase and Redis, demonstrating its applicability over diverse NoSQL stores regarding the underlying data model.

In more technical terms, NoDA consists of a programming API and a SQL interface. The programming API provides a set of basic *data access operators*, which are commonly provided by all NoSQL stores. Examples of data access operators include `filter`, `project`, `groupBy`, `aggregate` and `sort`. In turn, this enables the provision of an SQL interface to end users, which translates an SQL query to a sequence of data access operators. Moreover, the retrieved data objects can populate a Spark Dataframe in order to undergo (more complex) in-memory, data-parallel processing.

B. The Programming API

By design, NoDA performs operations on sets of data objects, which are described by fields and corresponding values. The data objects are stored at a NoSQL store. Before applying operations, a specific set of objects must be selected. Practically, this selection is equivalent to determining a specific collection (in case of document-oriented stores), a table with multiple columns (in case of wide-column stores), or a set of key-value pairs (in case of a key-value store). After

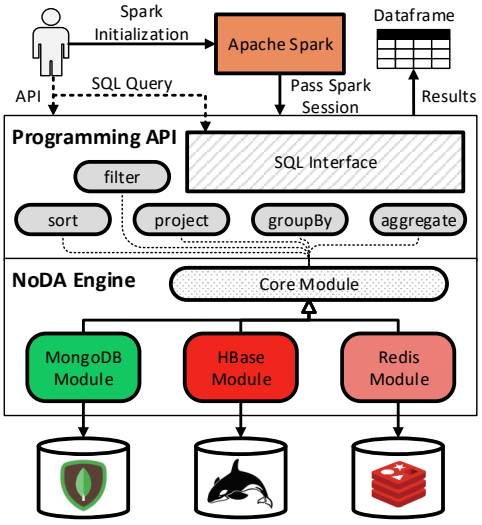


Fig. 1: The NoDA concept.

determining a specific set of data objects, operations can be applied on fields of these objects.

The programming API is used in the following way. First, a connection is established to the underlying NoSQL store and a concrete set of data objects is specified. Then, a query is formulated for execution, by specifying a sequence of data access operators, using *method chaining*.

Listing III.1: Code template for expressing queries using NoDA.

```
1  Dataset<Row> dataset = noSqlDbSystem.operateOn("table_name")
2      .filter( ... ).filter( ... ) //definition phase
3      .groupBy( ... ).sort( ... ) //definition phase
4      .project( ... ) //definition phase
5      .toDataframe(); //execution phase
```

Listing III.1 depicts the template code for expressing a query by means of a sequence of data access operators in NoDA. The `noSqlDBSystem` is an object reference (line 1) of `NoSqlDBSystem` object that has been instantiated (not shown in the Listing) with the connection details of a NoSQL store. This object can be optionally associated with a Spark session, which is useful for fetching the data objects in the form of a Spark Dataframe as well as for certain complex data processing tasks that cannot be "pushed-down" to the NoSQL store. Then, the collection name is specified using the `operateOn` method. Lines 2–5 show how the different operations can been specified. Inspired by Apache Spark's lazy execution model [18] that separates transformations from actions and invokes execution when an action is met, operations in NoDA are also separated in two phases: *definition* and *execution* phase. Essentially, the operators correspond to stages in a multi-stage pipeline. Multiple stages can be declared in the pipeline, which are executed only when an operator is found that belongs to the execution phase. All operators in the listing belong to the definition phase, except of `toDataFrame` which belongs to the execution phase. Apart from this, the execution phase also includes aggregate operations, such as `max`/`min`/`sum`/`avg` or `count`, which return a number. Also, the `limit`, `aggregate` and `distinct` operators belong to the definition phase. The `aggregate` operator allows a user to perform multiple aggregations within a single execution of operations, which is not feasible in the execution phase.

C. Implementation Aspects

1) Document-oriented Stores: MongoDB is a document-oriented, semi-structured NoSQL store, where data is modeled as *documents* (binary JSON objects) that contain fields associated with values. Documents are grouped in collections and the documents in a collection usually have the same interpretation. However, the schema of documents may vary.

The implementation of NoDA exploits the *aggregation pipeline* framework provided by MongoDB. The rationale behind this framework is that documents enter a multi-stage pipeline that transforms them. The pipeline supports various MongoDB. operators (e.g., *$match*, *$project*, *$group*, etc.), which can be combined so as to produce the desired results.

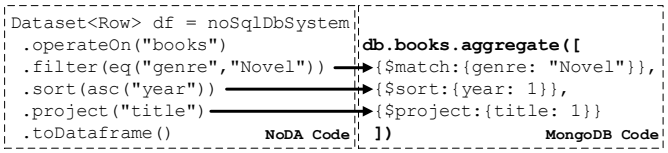


Fig. 2: Query transformation from NoDA to MongoDB.

Figure 2 shows how a query expressed as a sequence of data access operators is translated to the language of MongoDB. The expression on the left part of the figure is NoDA code, whereas the MongoDB-specific code that is eventually executed is depicted on the right. Essentially, NoDA provides concrete implementations of abstract operators using the language of the NoSQL store. In this example, `filter` is translated to MongoDB's *$match* operation, while `sort` and `project` are translated to their respective counterparts. With `operateOn`, we determine the collection of MongoDB which will be queried (*"books"*) and the aggregation pipeline is initiated.

2) Wide-column Stores: In this NoSQL category, data is stored in *tables* that contain multiple *columns*. A *row* is composed of a *row key* and columns associated with values. Rows are stored in lexicographic order by their key. Columns are grouped into *column families* and are referenced as *family:qualifier*. A column family has its own storage properties, and all columns underneath it are stored in the same storage file (HFile). A column represents an attribute, containing one or more *cells* that hold a value as a serialized object (arbitrary array of bytes). HBase supports efficient range scans based on the row key, whereas queries that do not use the row key are resolved using a full table scan. Also, a good practice is to form a column family with columns that are frequently accessed together by queries, for more efficient access.

HBase provides miscellaneous filters that operate at the level of row, row key, column family, column and cell. In NoDA,

a data access operator corresponds to one or more HBase filters. NoDA exploits these filters in the implementation of data access operators, by combining them in a *filter list*. A record is returned only if it complies with the individual filters in the filter list. In this way, we achieve to transform complex operations to a series of HBase filters for fetching the desired results. In more details, a comparison operator under the `filter` data access operator uses a *single column value* filter, which performs cell scan given a column's qualifier, family name and a value. A logical operator (`and`, `or`) is transformed to a filter list containing filters that must be all satisfied (conjunction) or at least one of them (disjunction). The `project` access operator may use a *family* filter for the projection of a whole column family, or a filter list containing a family filter and a *qualifier* filter for the projection of a specific column. The `limit` access operator uses the *page* filter to limit the results.

```
Dataset<Row> df = noSqlDbSystem.operateOn("orders")
        (S)       .filter(and(eq("customer:city","Athens"),
                              gt("order:price",100)))
      (F, Q)      .project("customer","order:id")
       (P)        .limit(500)
                  .toDataframe();                    NoDA Code
```

```
S = FilterList(
   SingleColumnValueFilter("customer:city",EQUAL,"Athens"),
   SingleColumnValueFilter("order:price",GREATER,100)
   ) -> MUST_PASS_ALL

F = FamilyFilter(EQUAL,"customer")      P = PageFilter(500)

Q = FilterList(
      FamilyFilter(EQUAL,"order")
      QualifierFilter(EQUAL,"id") ) -> MUST_PASS_ALL

                        Result:
          FilterList(S, F, Q, P) -> MUST_PASS_ALL
                                                   HBase Code
```

Fig. 3: Query transformation from NoDA to HBase.

Figure 3 explains the transformation of a query from NoDA to HBase filters by means of an example. The operation is performed on a table named *"orders"*, which is composed of two column families. The first column family holds customer-related information, while the other order-related information. The operation retrieves rows corresponding to customers that reside in the city of Athens and their order is worth more than 100 euros (`filter` operator). The fetched rows include all of the columns of *"customer"* family and the *"id"* column of the *"order"* family as well (`project` operator). The results are limited to the first 500 rows (`limit` operator). NoDA executes the operation by grouping the respective filters (*single column value* (*S*), *family* (*F*), *qualifier* (*Q*) and *page* (*P*) filters) under a single filter, whose constraints must be fulfilled by the retrieved rows. The *S* filter combines two single column value filters under a conjunctive condition. These filters operate on the columns *"city"* and *"price"* of the *"customer"* and *"order"* column families respectively. The *F* filter is related to the projection of the columns whose column family name is *"customer"*. The *Q* filter applies the projection with two filters under a conjunctive condition. The column qualifier should be *"id"* and the column family should be *"order"*. Finally, the *P* filter limits the number of retrieved results.

3) Key-values Stores: Redis is an in-memory, key-value store that inherently supports a data model of low expressive power. This makes the instantiation of NoDA more challenging. Objects in Redis are modeled as key-value pairs, and individual objects can be accessed by key efficiently.

Therefore, in order to implement NoDA over Redis, we need to design an effective data representation. To this end, an object (that consists of fields with associated values) is stored as a key-value entity, where the key is a unique identifier, while the value is a *hash* structure that associates fields with their corresponding values. In this way, we achieve retrieval by object identifier in constant time. Furthermore, to support efficient object retrieval by a field's value, we insert additional information in the key-value store. In more detail, we discern two cases: categorical and numerical fields. For a categorical field cf that takes values $\{v_1, v_2, \dots\}$, we generate keys $\{cf : v_1, cf : v_2, \dots\}$ that combine the field name and its values. Each of these keys is associated with a *set* structure as value. The set for key $cf : v_i$ contains all object identifiers that have value v_i in field cf. In this way, we can retrieve in constant time all object identifiers with a specific value in a field. Numerical fields are stored in a different way. Each numerical field nf is stored as a key associated with a *sorted set* as value. Each element of the sorted set consists of the object identifier and a *score* that keeps the value of the object for field nf. The elements of the sorted set are sorted by score, which provides retrieval of the object identifier by value in logarithmic time.

The execution of the data access operations on top of Redis is achieved by utilizing the *Redis pipelining*. In this way, we are able to send multiple commands to the server at a single step. The pipeline is materialized by scripts written in Lua, which perform server-side operations. When defining data access operators in NoDA, these are transformed into Lua scripts and are added to the pipeline, so as to be executed as a group. The scripts on the pipeline simulate a chain of operations. Each operation emits intermediate results, which are combined in order to reach the final results of the defined operations. The intermediate results are stored in temporary sets that expire eventually. Their key is randomly generated, and their value is a set that holds the object identifiers.

Figure 4 shows the transformation of an operation expressed in NoDA to Redis scripts, over a database that holds information about movies. The operation counts the number of movies that *Ron Howard* directed after 2010. For its execution, at first the targeted set is determined given the attribute (*"director"*) and a value (*"Ron Howard"*). The set contains all of the identifiers of the movies that have been directed by the specified director. Then, the comparison (*gte*) condition is performed on the sorted set that handles the *"year"* attribute. This is achieved by the *ZRANGEBYSCORE* command which retrieves the movies' identifiers that were released after 2010. The identifiers are stored in a temporary key-value entity (com-

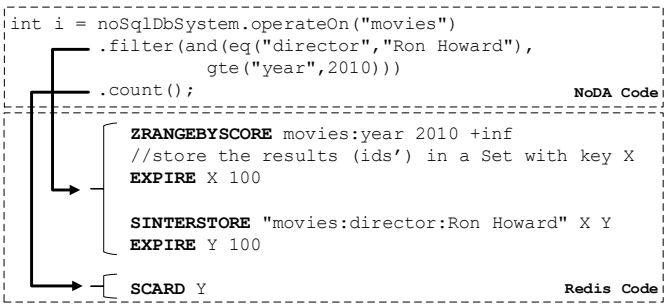


Fig. 4: Query transformation from NoDA to Redis.

mands in Lua were written for this purpose), whose key name is *"X"*, which will expire after 100 seconds. Subsequently, given the two key-value entities with key name *"director:Ron Howard"* and *"X"*, the *SINTERSTORE* command is executed, storing the common identifiers that are found in both sets, in a new temporary set named *"Y"*, which is also set to expire after 100 seconds. Finally, the *SCARD* command runs on *"Y"* set, returning the number of the contained elements.

D. The Declarative Interface

As already mentioned, the programming API offers numerous advantages, including the ease of understanding, as well as being simple and unified. However, the programming API of NoDA is yet another language that needs to be learnt by big data developers and data scientists. To remove this obstacle, NoDA incorporates a module which supports declarative querying using an SQL-like query language. We emphasize this novel feature of our approach that *it enables querying different NoSQL stores using SQL.*

Given a query expressed in SQL, we identify its constituent SQL clauses, and map each clause to a specific NoDA operation. Thus, the SQL query is translated into a sequence of NoDA operations, which can then be forwarded to the underlying NoSQL store, after translation to the language of the store. In our current implementation, the order of NoDA operations follows a rule-based approach. For example, selections and projections comprise the first operations in the sequence. This is because query optimization is left to the underlying NoSQL store. For example, stores such as MongoDB have their own query optimizer, whose task is to identify the best plan for query execution. For NoSQL stores that do not provide query optimization, our rule-based approach that pushes down operations that restrict the size of intermediate results works well in practice.

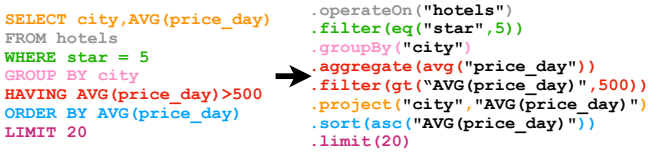


Fig. 5: Mapping of SQL query to NoDA operations.

Figure 5 illustrates how a query expressed in a language similar to SQL is mapped to NoDA operations. The query is performed on a data source (e.g., a table or a collection) named *hotels*, which is stored in a NoSQL store. The query retrieves the top-20 cities with the highest average price per day of their 5-star hotels, having a minimum price (500). The clauses of the SQL statement are coloured in order to clearly illustrate the correspondence to NoDA operations, which are indicated with the same colour. Similarly to the programming API, the SQL interface also supports user-defined functions (UDFs) in order to define customized operations such as spatial and spatio-temporal.

The implementation of the SQL module is based on *ANTLR* (https://www.antlr.org/), which is a parser generator for reading, processing, executing, or translating structured text or binary files. Given a grammar, ANTLR generates a parser, builds parse trees, and generates a listener interface (or visitor) that makes it feasible to recognise phrases and respond with specific actions. In our case, the actions that have been injected in those generated units, are related to the creation of the appropriate objects from NoDA's programming API, so as to form the respective sequence of operations. We use the grammar file of Presto [20].

IV. Spatio-temporal Queries in NoDA

In this section, we present the design choices and implementation techniques of NoDA for supporting efficient spatio-temporal queries over NoSQL stores.

A. Overview

Towards providing support for spatio-temporal operations, in addition to common operators such as boolean and comparison ones, NoDA supports operators oriented to spatial (2D) and spatio-temporal (3D) data, called *Geographical Operators* (or Geo-Operators in short).

Listing IV.1: Spatial k-NN query.

```
1  Coordinates c = Coordinates.newCoordinates(23.7613,37.9864);
2  Dataset<Row> dataset = noSqlDbOp
3              .filter(geoNearestNeighbors("location",c,2))
4              .toDataframe();
```

Listing IV.1 shows an example of a k-nearest neighbor (k-NN) query, expressed in NoDA. The query retrieves the 2 nearest objects to query point c. In practice, passing a Geo-Operator to the *filter* method is equivalent to a filtering operation on a specific type of spatial or spatio-temporal data.

Since k-NN queries are usually not optimally supported by NoSQL stores[1], NoDA transforms a k-NN query to a range query using a *radius estimation* technique. For this purpose, NoDA builds a QuadTree [29] over a sample of the data, and keeps the number of enclosed objects in its leaf nodes. The QuadTree is exported in serialized file on disk, and is loaded

[1]For example, MongoDB uses the *$geoNear* operator to support k-NN queries, but additionally requests to specify a maximum radius for search, which may fail to retrieve k results. If the maximum radius is not specified, all objects are accessed.

in-memory, when k-NN queries are issued. Then, at query time, we search for the leaf node that encloses the query point, and we compute the minimum radius for the range query that is guaranteed to enclose the leaf nodes that contain at least k objects.

B. NoDA *over a Document-oriented Database*

MongoDB provides built-in support for spatial data by means of the GeoJSON object type, which is used to represent various geometries, such as points and polygons. This kind of data is accessed by geospatial operators such as the *$geoNear*, *$geoWithin* and *$centerSphere*, which can take advantage of the built-in spatial indexes (*2d* and *2dsphere*) for efficient query execution. Nevertheless, spatio-temporal data is not inherently supported.

In order to effectively model spatio-temporal data in MongoDB, we model each spatio-temporal point as an individual document, which stores the location field (storing the coordinates) and the temporal field. We exploit the Hilbert space-filling curve [30] to derive an 1D approximation of the spatial location of each object, and we equip each document with a field (`hilbertIndex`) that keeps its Hilbert value.

The rationale behind this design decision is the following. In a MongoDB cluster, where data is partitioned to shards, the field that represents the spatial information cannot be used for partitioning, i.e., as shard key. This is a limitation of MongoDB as of now. In turn, this practically means that only temporal partitioning is supported, which is clearly sub-optimal for spatio-temporal queries as it affects data locality. To alleviate this shortcoming, we apply a custom solution for spatio-temporal partitioning, using as a shard key the combination of fields: {`hilbertIndex, date`}, which has been shown to improve the performance of spatio-temporal queries, because it preserves spatio-temporal data locality better [4].

```
db.collection.aggregate([
  { $match: { $and: [
    { $or: [
       { hilbertIndex: { $gte: 25436, $lte: 26684 } },
       { hilbertIndex: { $gte: 26976, $lte: 27112 } },
       { hilbertIndex: { $in : [25417, 25419, 25425, 25430] } }
    ] },
    {date: { $gte: ISODate("2021-02-18T12:34:27.673Z") } },
    {date: { $lte: ISODate("2021-02-24T03:29:54.439Z") } },
    { location: { $geoWithin: { $geometry: { type : Polygon,
       coordinates: [ [ [23.8142, 38.0264], [23.9330, 38.0264],
           [23.9330, 38.1251], [23.8142, 38.1251],
           [23.8142, 38.0264] ] ]
     } } } }
  ] } }
])
```

Fig. 6: Example of spatio-temporal query that NoDA sends to MongoDB for execution.

When a spatio-temporal query is issued for query execution, NoDA computes the Hilbert values associated with the spatial constraint of the query. Then, NoDA embeds these in the translated query to MongoDB as shown in Figure 6. Thus, the `hilbertIndex` field participates in the query's formation, in addition to the temporal constraint and the respective GeoJSON expression. In this way, the query is executed on the shards that contain the respective index values and date range. For the targeted shards, at first, the index is accessed for efficient retrieval of the documents that fulfill both the index values and the date condition. Then, in a refinement phase, the fetched documents are examined on their coordinates (GeoJSON objects), so as to exclude those that are not enclosed in the query's spatial constraint (false positives). The remaining documents of every shard are returned to the client.

C. NoDA *over a Wide-column Store*

HBase is a popular wide-column store that follows the design of BigTable [31], supporting tables with many columns, records associated with keys, as well as efficient range scans on keys. However, NoSQL stores such as HBase do not provide inherent support for spatio-temporal data and relevant operations. Thus, in order to implement the spatio-temporal operations of NoDA over HBase, we capitalize on its built-in support for efficient range scans.

Our approach to data modelling consists of encoding the spatio-temporal information in the row key of each record, in the same spirit as [32] for multidimensional data. This is achieved by obtaining the Geohash (https://en.wikipedia.org/wiki/Geohash) of the spatial coordinates (x, y) and concatenation with the time t of the record in Unix timestamp format, using the dash (-) as separator character, e.g., *sw8zf-1589354579100*. Since row keys must be unique, we add a randomly-generated string as suffix, in order to differentiate keys corresponding to the same Geohash value and timestamp. As a result, our row key has the following format: *GEOHASH-TIMESTAMP-RANDOMSTR*. Also, we define a column family *"location"* that consists of the columns with spatial and temporal information.

Given a spatio-temporal range query, we need to retrieve all rows with key starting with a specific prefix. This prefix is formed by (a) computing of the most precise Geohash value that encloses entirely the minimum bounding rectangle (MBR) of the spatial part of the query, and (b) determining the common prefix part of the starting and ending timestamps of the temporal interval of the query. For example, a spatio-temporal query referring to Northern Athens for the whole month of February of 2019 would be converted to a search for keys matching the expression (also called *mask*): *swbb?-158??????????-??????????*, where wildcard characters are used to match the unknown part of the row key.

At query time, the first step is to transform the spatio-temporal query to the above expression. Then, this expression is passed to a *fuzzy row filter*, which takes as input a mask that matches row keys that have arbitrary characters at specific positions. During the range scan, the fuzzy row filter performs fast-forwards for locating the row keys that match the mask. Obviously, the filter also retrieves false-positive query results. Subsequently, a *custom filter* is applied on the fetched rows, so as to perform refinement of the final result set. The custom filter is applied to the columns of each row that store the longi-

tude, latitude and date values. In our work, we implemented 6 custom filters in HBase supporting spatial and spatio-temporal filtering for rectangle, circle and polygon constraints. As a technical detail, notice that we implemented both the fuzzy row and the custom filters as server-side filters, hence their execution is efficiently performed on the *regionservers*.

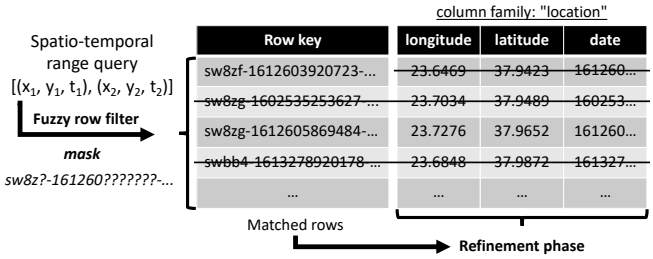


Fig. 7: Data modeling by NODA in wide-column store.

Figure 7 shows a spatio-temporal range query issued against HBase. The spatio-temporal constraints are used to compute the mask, which is subsequently used to filter records efficiently, based on row key. The row keys that do not match the mask are not accessed and are depicted with strikethrough text in the figure, including the row key and column family. The obtained matched rows undergo a refinement phase, where the exact values kept in the *"location"* column family are examined to exclude false positives. These are depicted with strikethrough text only at the column family of the row.

D. NODA *over a Key-value Store*

Supporting spatio-temporal queries over key-value stores is challenging due to the lack of range query support by the underlying store [33]. Thus, encoding the spatio-temporal information in the key is not feasible, as it would require the conversion of a continuous interval (corresponding to the spatio-temporal range) into an infinite number of discrete values.

Therefore, in order to implement the spatio-temporal operations of NODA over a key-value store (Redis), we opt for the following design. Each spatio-temporal record (x, y, t) is associated with a key-value pair. The key is a unique identifier of the record and it is randomly generated. The value field keeps the spatio-temporal information of a record. In addition, to support efficient spatio-temporal retrieval, we compute the Hilbert value h of each 3D spatio-temporal record (x, y, t), and we generate a dummy key named `location` whose value is a sorted set. The sorted set contains one entry for each record in our data set, and each entry consists of the record's unique identifier and its Hilbert value. The set is sorted by Hilbert value, effectively serving as an index, and thus supporting efficient retrieval in logarithmic time. With respect to additional, non-spatial fields of the data set, numeric fields are stored using the above approach, whereas categorical fields are stored using sets (instead of sorted sets).

Figure 8 illustrates the data modeling over the key-value store. In the center, the key-value pairs are depicted which

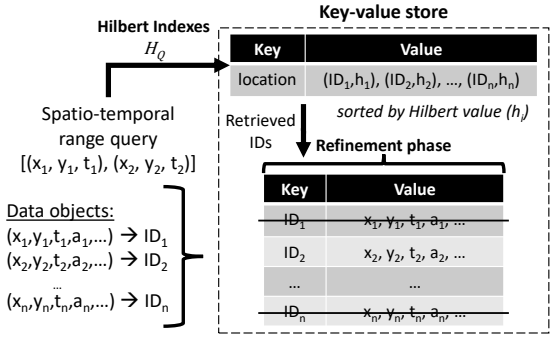


Fig. 8: Data modeling by NODA in key-value store.

correspond to the spatio-temporal data objects (values) and random identifiers (keys). At the top, the object identifiers are organized in a sorted set, ordered based on the Hilbert value of the corresponding spatio-temporal data object.

When a spatio-temporal range query Q is issued, query execution is performed in three steps. First, the spatio-temporal constraint is transformed to a set $\mathcal{H}_Q$ of Hilbert values, which correspond to the 3D cells that intersect with the query. Then, the sorted set is used in order to efficiently retrieve the identifiers of data objects whose Hilbert value h_i is in $\mathcal{H}_Q$, i.e., $h_i \in \mathcal{H}_Q$. Finally, we retrieve the objects based on the identifiers, which is efficiently performed in constant time. These objects undergo a refinement step, where false positives are discarded. Notice that the proposed design is generic and applicable over all modern key-value stores that typically provide support for sorted sets as values.

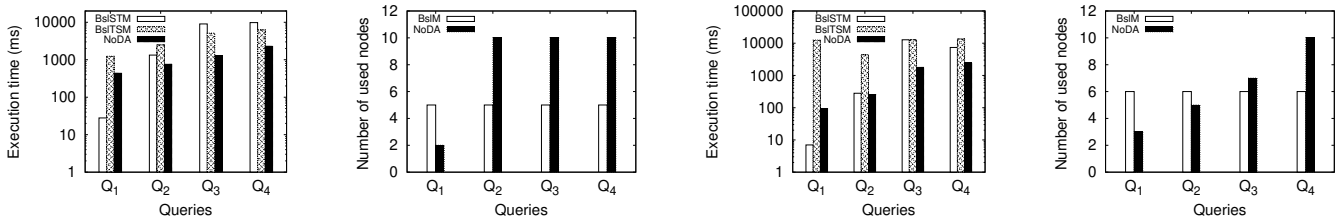
V. EXPERIMENTAL EVALUATION

In this section, we provide the results of the experimental study. Our code is available at: https://github.com/nkoutroumanis/NoSQL-Operators.

A. Experimental Setup

Baselines. We compare the performance of NODA against baseline solutions in the three NoSQL stores, denoted BSLM for MongoDB, BSLH for HBase, and BSLR for Redis (described in Sections V-B, V-C and V-D respectively). In brief, BSLM uses a compound index over space and time for efficient access, and partitions data to nodes based on time. Note that MongoDB does not support spatial partitioning (sharding) at the time of this writing. As HBase does not provide inherent support for spatial data, the baseline BSLH performs table scans without exploiting the row key, whereas in our approach we inject the spatio-temporal information in the row key. In the case of Redis, the baseline BSLR keeps two structures, one for spatial data (GEO type, supported by Redis) and one for temporal data, and merges the results of these queries.

Platform. Our experimental evaluation is performed in the Okeanos-Knossos IaaS platform, which is a cloud service supported by GRNET (https://grnet.gr) for research purposes, offering virtual computing and storage services. We have engaged 17 VMs running Ubuntu 16.04.6 LTS, 68 CPUs,



(a) Execution time (REAL) (b) Number of nodes (REAL) (c) Execution time (SYNTH) (d) Number of nodes (SYNTH)

Fig. 9: Performance of queries on MongoDB using REAL data set: (a), (b), and SYNTH data set: (c), (d).

136GB RAM and 2.00TB disk space. For uniformity, in each deployment, we use 12 nodes for data storage, 2 nodes as clients for data loading, and 3 nodes for handling the metadata (these 3 nodes are not used in the Redis cluster).

	Data size (GB) in stores				
Data set	MongoDB (Bsl*M)[2]	MongoDB (NoDA)	HBase	Redis (BslR)	Redis (NoDA)
REAL	40.5	40.8	92.8	7.6	6.0
SYNTH	3.6	4.1	8.7	14.6	11.6

TABLE I: Size of the stored data in the NoSQL stores.

Data sets. We use both real-life data as well as synthetic ones in our experiments. The real-life data set (REAL) is provided by a fleet management provider and consists of $15.2M$ GPS positions of vehicles moving in the Greek mainland and island territory for a period of five months (July to November 2018). Apart from the position in space and time, each record also contains information about the referring vehicle, the prevailing weather conditions, the road network and the nearest points-of-interest (POIs). The overall information spans 75 columns. The synthetic data set (SYNTH) is generated in a much smaller spatial area, covering approximately 1.54% of the spatial extent of REAL. The temporal extent of SYNTH is the half of REAL, i.e., 2.5 months, and contains twice the records. For the performance evaluation in Redis with the REAL set, only the values of 4 columns (spatio-temporal information and vehicle identifier) are inserted to the store, due to memory limitations.

Query generation. We generate spatio-temporal range queries by fixing the temporal constraint and increasing the spatial extent of the query, in order to study the effect of query selectivity on performance. We have also performed experiments varying the temporal extent of the query with similar results. Q_1 corresponds to the smaller spatial constraint, whereas Q_4 has the largest spatial constraint. In the case of MongoDB and HBase we use rectangular range queries, whereas we use circular range queries in the case of Redis.

Metrics. In the charts, we measure the execution time as the average of 10 executions, in order to factor out the effect of randomization. Also, we report the number of cluster nodes that participate in query execution, which often relates to the observed query execution time. In addition, when necessary, we also report on the number of accessed objects on the different nodes, which also affects the execution time.

[2]Bsl*M refers to the two baseline solutions in MongoDB case

B. Performance Evaluation in MongoDB

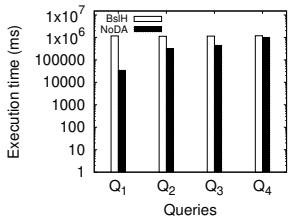
We evaluate the spatio-temporal extension of NoDA against a baseline solution (BslM) that uses a compound index in space and time. Notice that this is essentially the best practice for handling spatio-temporal data in MongoDB as of now. In fact, we evaluate two alternatives for the baseline; one that prioritizes space over time (BslSTM) in the compound index, and the opposite (BslTSM). For the baselines, the sharding key is the `date` field, since MongoDB does not support sharding based on a 2dsphere index. Regarding the available indexes, these include a 2dsphere index on the `location` field and an index on the `date` field, which is automatically built by MongoDB since `date` is set as sharding key.

Figures 9(a) and (b) show the results obtained using REAL for the different queries. Figure 9(a) shows the execution time (notice the log-scale on the y-axis), while Figure 9(b) shows the number of MongoDB nodes (out of 12) that participate in the query execution. The results show that almost always NoDA outperforms both baseline approaches. There are two reasons for this. First, NoDA partitions the data based on space and time, so a spatio-temporal query is processed by more nodes in NoDA (as shown in Figure 9(b)) that share the processing load, whereas the query in BslM targets fewer nodes. Second, at local level, NoDA typically examines fewer documents on the nodes. The only exception is for the query has high spatial selectivity (Q_1), where the baseline approach BslSTM that prioritizes space is faster than NoDA.

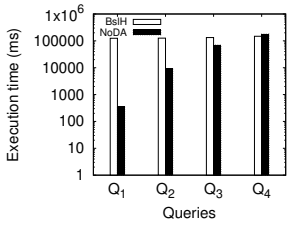
Figures 9(c) and (d) show the results obtained in the case of SYNTH data. Again, the results are consistent with the case of REAL data. NoDA outperforms the baseline solutions in all cases (except query Q_1), often by more than one order of magnitude. Notice that as the spatial extent of the query increases (Q_1–Q_4), Figure 9(d) shows that NoDA achieves to use more nodes for query processing, in contrast to BslM.

C. Performance Evaluation in HBase

For HBase (v 2.2.3), we use a baseline solution for comparative purposes, denoted BslH, which performs table scans without exploiting the row key for spatio-temporal querying. Recall that differently than BslH, NoDA performs pruning based on the row key, thus skipping irrelevant records.

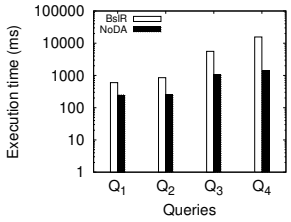


(a) Execution time (REAL)

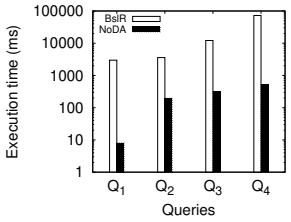


(b) Execution time (SYNTH)

Fig. 10: Performance of queries on HBase store.



(a) Execution time (REAL)



(b) Execution time (SYNTH)

Fig. 11: Performance of queries on Redis store.

Figure 10 shows the performance of NoDA against BslH both for REAL and SYNTH. Almost in all cases, NoDA outperforms BslH by a large margin. In general, for very selective queries (such as Q_1) NoDA is more than two orders of magnitude faster than BslH. For queries that return more results (e.g., Q_2 and Q_3), NoDA still outperforms BslH but the gain is reduced, and eventually the performance of the two approaches tends to converge for queries that return many results (Q_4). Notice that this is expected since it corresponds to the case that a full table scan is often faster than index-based access, when a large percentage of records are retrieved. In the case of SYNTH and only for Q_4, NoDA is slower than BslH due to the high number of matched rows that need to be examined in the refinement phase using the fuzzy filter. The takeaway message is that the more selective the query, the highest the performance gain of NoDA in HBase. This holds up to a certain point, i.e., for queries with selectivity around 5% (such as Q_4 on SYNTH).

D. Performance Evaluation in Redis

For the experiments in the Redis key-value store (v 6.0.8), we compare NoDA against a baseline solution (BslR) that handles two data structures (instead of one) on each node for spatio-temporal querying.

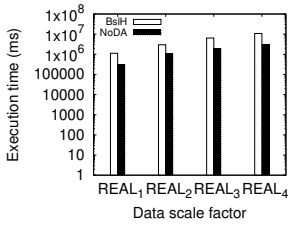
The first structure stores the spatial information with the identifier of each data object in a special (GEO) type of sorted set. This type is offered natively by Redis for querying spatial data, and it is based on geohashing. The other structure is a sorted set that stores the timestamp with the identifier of each data object. BslR executes the spatial part and the temporal part of the query separately, and performs set intersection over the matched object identifiers to return the result. Regarding partitioning, both NoDA and BslR use round-robin partitioning. Every node maintains the sorted sets for its locally stored data objects, which shares the load across nodes and increases the efficiency of query execution.

Figures 11(a) and 11(b) show the comparative performance of NoDA against BslR in terms of execution time for the REAL and SYNTH data sets respectively. In both data sets, for all queries, NoDA consistently outperforms BslR. Also, the general trend is that for queries that retrieve more results, this performance gain increases, reaching one order of magnitude (for REAL) and two orders of magnitude (for SYNTH). This is attributed to the modeling that NoDA adopts for spatio-temporal data. In particular, the sorted set on the `location` field enables fast retrieval of the object identifiers that are candidate results, which are then efficiently accessed by key in order to perform the refinement.

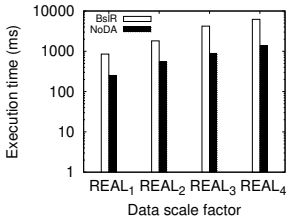
In summary, in the case of the key-value store, NoDA achieves to boost the performance of spatio-temporal queries, which cannot be processed efficiently due to the lack of range query support by the store. This is a strong argument in favour of our modeling technique, which enables efficient processing of spatio-temporal queries over key-value stores.

	REAL data set with scale factor			
Size (GB) in stores	REAL_1	REAL_2	REAL_3	REAL_4
HBase	92.8	190.4	387.19	583.95
Redis (BslR)	7.6	16.4	32.6	46.8
Redis (NoDA)	6.0	12.0	24.4	34.9
#documents (M)	15.2	31.4	63.9	96.4

TABLE II: Size (in GB) of the 4 instances of the REAL data set for the 3 NoSQL stores.



(a) Results on HBase



(b) Results on Redis

Fig. 12: Scalabilty study on (a) HBase, and (b) Redis, using the REAL data set.

E. Scalability Study

For our scalability study with increasing data set size, we have obtained GPS traces of more vehicles from the data provider, in the same spatio-temporal box. Thus, we use larger instances of the REAL data set, up to 6x larger. Table II reports the details of the 4 instances of the REAL data set used in our experiments, denoted REAL_1 – REAL_4.

We report the results on HBase and Redis only, due to lack of space (yet similar conclusions are drawn in the case of MongoDB too). We gradually increase the size of the data to assess its impact on performance of query execution, using

the same resources. We use query Q_2 in this experiment, as a middle case, and again we report the average of 10 executions. Figure 12(a) depicts the results obtained for HBase. Clearly, NoDA sustains the performance gain for larger data sets over BslH. It is noteworthy that the gain (in absolute values) increases for larger data sets, due to the log-scale used on y-axis. The increased execution time shows a linear relationship with the size of the data. Figure 12(b) shows the corresponding results for Redis. In this case, the absolute gain of NoDA over BslR increases more clearly for larger data sets.

VI. Conclusions and Future Work

In this paper, we presented NoDA, an abstraction layer for NoSQL stores that allows unified access to heterogeneous data models and storage systems. NoDA abstracts access to data stored in NoSQL stores, by providing a set of data access operators (with clear semantics) that are implemented for different NoSQL stores and capitalizes on them in order to provide a declarative interface. NoDA is optimized for spatial and spatio-temporal data, which are prevalent in modern applications. In our future work, we intend to extend NoDA to cover more types of spatio-temporal queries, different NoSQL stores, as well as for supporting updates.

Acknowledgments

This research work has received funding from the EU's Horizon 2020 research and innovation programme under grant agreement No 780754 (Track&Know project), and from the Hellenic Foundation for Research and Innovation (HFRI) and the General Secretariat for Research and Innovation (GSRI), under grant agreements No 1667 and No HFRI-FM17-81.

References

[1] R. Cattell, "Scalable SQL and NoSQL data stores," *SIGMOD Rec.*, vol. 39, no. 4, pp. 12–27, 2010.

[2] A. Davoudian, L. Chen, and M. Liu, "A survey on NoSQL stores," *ACM Comput. Surv.*, vol. 51, no. 2, pp. 40:1–40:43, 2018.

[3] C. Doulkeridis, A. Vlachou, N. Pelekis, and Y. Theodoridis, "A survey on big data processing frameworks for mobility analytics," *SIGMOD Rec.*, vol. 50, no. 2, pp. 18–29, 2021.

[4] N. Koutroumanis and C. Doulkeridis, "Scalable indexing and querying of spatio-temporal data in nosql stores," in *Proc. of EDBT*, 2021, pp. 611–622.

[5] N. Koutroumanis, P. Nikitopoulos, A. Vlachou, and C. Doulkeridis, "NoDA: Unified NoSQL data access operators for mobility data," in *Proc. of SSTD*, 2019, pp. 174–177.

[6] N. Koutroumanis, N. Kousathanas, C. Doulkeridis, and A. Vlachou, "A demonstration of NoDA: Unified access to NoSQL stores," *Proc. VLDB Endow.*, vol. 14, no. 12, pp. 2851–2854, 2021.

[7] S. Li, S. Hu, R. K. Ganti, M. Srivatsa, and T. F. Abdelzaher, "Pyro: A spatial-temporal big-data storage system," in *Proc. of USENIX*, 2015, pp. 97–109.

[8] J. K. Nidzwetzki and R. H. Güting, "Distributed SECONDO: A highly available and scalable system for spatial data processing," in *Proc. of SSTD*, 2015, pp. 491–496.

[9] J. K. Nidzwetzki and R. H. Güting, "Distributed secondo: an extensible and scalable database management system," *Distr. Paral. Datab.*, vol. 35, no. 3-4, pp. 197–248, 2017.

[10] A. D. Fox, C. N. Eichelberger, J. N. Hughes, and S. Lyon, "Spatio-temporal indexing in non-relational distributed databases," in *Proc. of IEEE Big Data*, 2013, pp. 291–299.

[11] J. Yu, Z. Zhang, and M. Sarwat, "Spatial data management in Apache Spark: The GeoSpark perspective and beyond," *GeoInformatica*, vol. 23, no. 1, pp. 37–78, 2019.

[12] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, and W. G. Aref, "LocationSpark: A distributed in-memory data management system for big spatial data," *Proc. VLDB Endow.*, vol. 9, no. 13, pp. 1565–1568, 2016.

[13] Z. Shang, G. Li, and Z. Bao, "DITA: distributed in-memory trajectory analytics," in *Proc. of SIGMOD*, 2018, pp. 725–740.

[14] A. Eldawy, V. Hristidis, S. Ghosh, M. Saeedan, A. Sevim, A. B. Siddique, S. Singla, G. Sivaram, T. Vu, and Y. Zhang, "Beast: Scalable exploratory analytics on spatio-temporal data," in *Proc. of CIKM*, 2021, pp. 3796–3807.

[15] D. F. Bacon, N. Bales, N. Bruno, B. F. Cooper, A. Dickinson, A. Fikes, C. Fraser, A. Gubarev, M. Joshi, E. Kogan, A. Lloyd, S. Melnik, R. Rao, D. Shue, C. Taylor, M. van der Holst, and D. Woodford, "Spanner: Becoming a SQL system," in *Proc. of SIGMOD*, 2017, pp. 331–343.

[16] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *Proc. of SIGMOD*, J. T. Wang, Ed., 2008, pp. 1099–1110.

[17] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy, "Hive - A petabyte scale data warehouse using hadoop," in *Proc. of ICDE*, 2010, pp. 996–1005.

[18] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: relational data processing in spark," in *Proc. of SIGMOD*, 2015, pp. 1383–1394.

[19] Y. Cao, W. Fan, and T. Yuan, "Block as a value for SQL over NoSQL," *Proc. VLDB Endow.*, vol. 12, no. 10, pp. 1153–1166, 2019.

[20] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner, "Presto: SQL on everything," in *Proc. of ICDE*, 2019, pp. 1802–1813.

[21] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. B. Zdonik, "The BigDAWG polystore system," *SIGMOD Rec.*, vol. 44, no. 2, pp. 11–16, 2015.

[22] A. J. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Çetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska, S. Madden, D. Maier, T. G. Mattson, S. Papadopoulos, J. Parkhurst, N. Tatbul, M. Vartak, and S. Zdonik, "A demonstration of the BigDAWG polystore system," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1908–1911, 2015.

[23] M. Vogt, A. Stiemer, and H. Schuldt, "Icarus: Towards a multistore database system," in *Proc. of IEEE BigData*, 2017, pp. 2490–2499.

[24] B. Kolev, C. Bondiombouy, P. Valduriez, R. Jiménez-Peris, R. Pau, and J. Pereira, "The CloudMdsQL multistore system," in *Proc. of SIGMOD*, 2016, pp. 2113–2116.

[25] E. Kharlamov, T. P. Mailis, K. Bereta, D. Bilidas, S. Brandt, E. Jiménez-Ruiz, S. Lamparter, C. Neuenstadt, Ö. L. Özçep, A. Soylu, C. Svingos, G. Xiao, D. Zheleznyakov, D. Calvanese, I. Horrocks, M. Giese, Y. E. Ioannidis, Y. Kotidis, R. Möller, and A. Waaler, "A semantic approach to polystores," in *Proc. of IEEE BigData*, 2016, pp. 2565–2573.

[26] B. Kolev *et al.*, "CloudMdsQL: Querying heterogeneous cloud data stores with a common language," *Distr. Paral. Datab.*, vol. 34, no. 4, pp. 463–503, 2016.

[27] Z. Kaoudi and J. Quiané-Ruiz, "Cross-platform data processing: Use cases and challenges," in *Proc. of ICDE*, 2018, pp. 1723–1726.

[28] D. Agrawal, S. Chawla, B. Contreras-Rojas, A. K. Elmagarmid, Y. Idris, Z. Kaoudi, S. Kruse, J. Lucas, E. Mansour, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, S. Thirumuruganathan, and A. Troudi, "RHEEM: enabling cross-platform data processing - may the big data be with you! -," *Proc. VLDB Endow.*, vol. 11, no. 11, pp. 1414–1427, 2018.

[29] R. A. Finkel and J. L. Bentley, "Quad Trees: A data structure for retrieval on composite keys," *Acta Informatica*, vol. 4, pp. 1–9, 1974.

[30] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz, "Analysis of the clustering properties of the Hilbert space-filling curve," *IEEE Trans. Knowl. Data Eng.*, vol. 13, no. 1, pp. 124–141, 2001.

[31] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, 2008.

[32] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi, "MD-HBase: A scalable multi-dimensional data infrastructure for location aware services," in *Proc. of MDM*, 2011, pp. 7–16.

[33] M. Pilman, K. Bocksrocker, L. Braun, R. Marroquin, and D. Kossmann, "Fast scans on key-value stores," *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1526–1537, 2017.