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

Nikolaos Koutroumanis[1], Christos Doulkeridis[1], Akrivi Vlachou[2]

[1]Department of Digital Systems
University of Piraeus, Greece

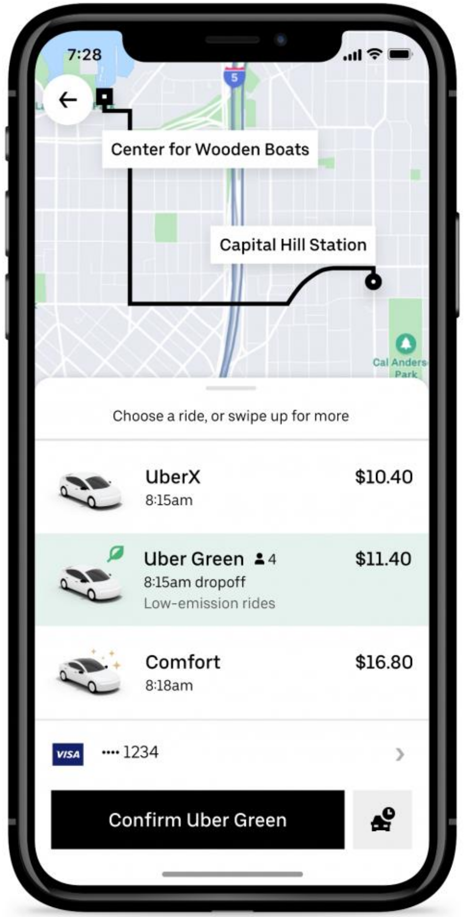
[2]Department of Information & Communication Systems Engineering
University of the Aegean, Greece

This work has received funding from This research work has received funding from the European Union'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 Technology (GSRT), under grant agreements No 1667 and No HFRI-FM17-81

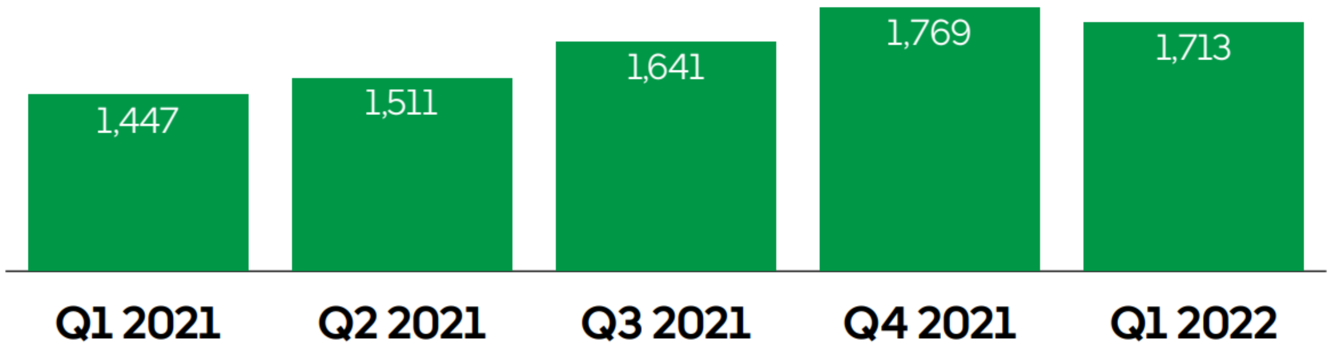
Contents

Introduction

- NoSQL stores today
  - increasingly adopted by modern applications and enterprises
  - for scalable storage and efficient querying over vast data collections
- Strong features:
  - support for schemaless data models, high availability and scalability
- Uber reports thousand of million trips per quarter year



Uber Technologies, Inc. Q1 2022 Earnings, Supplemental Data

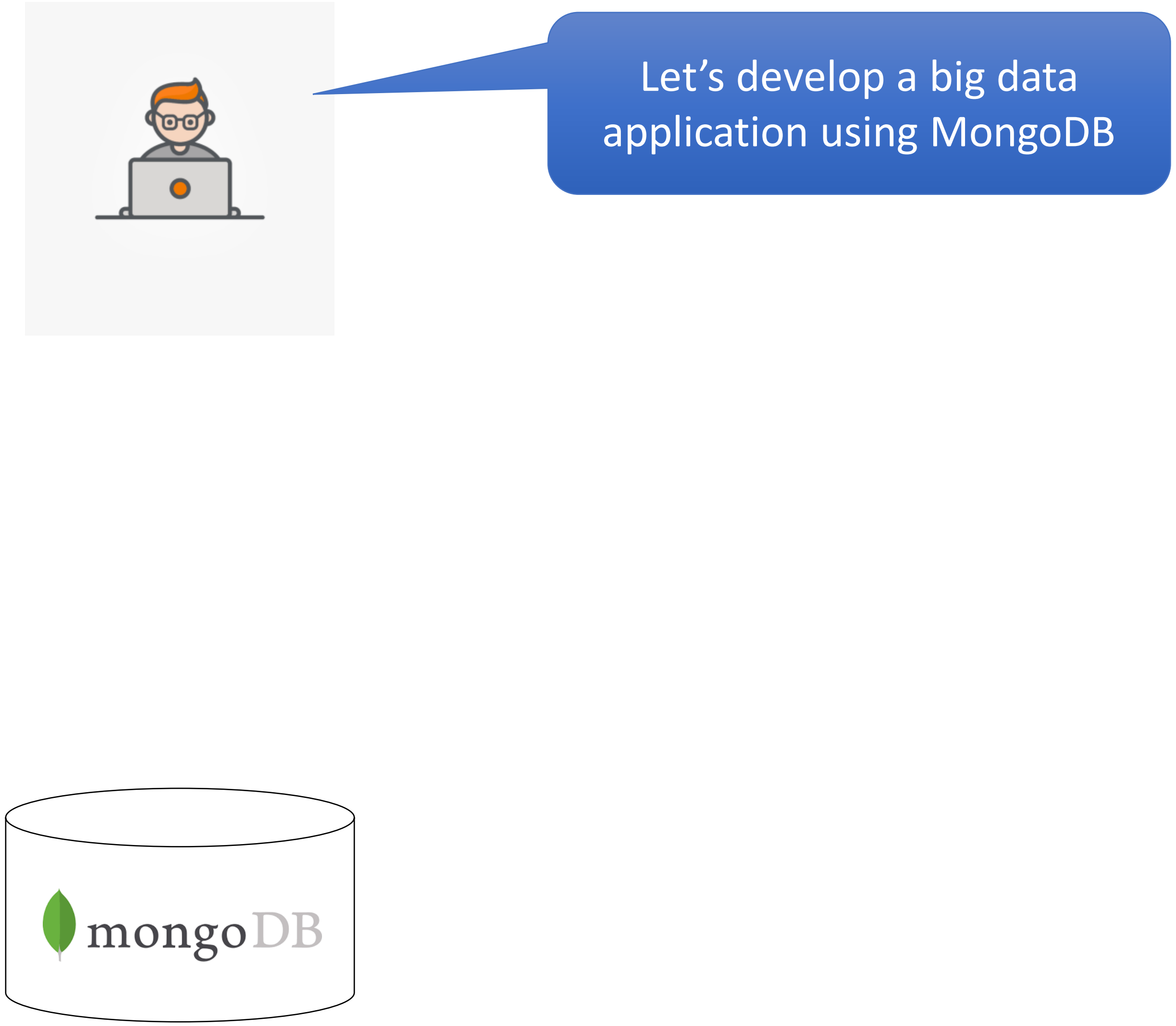


Motivation

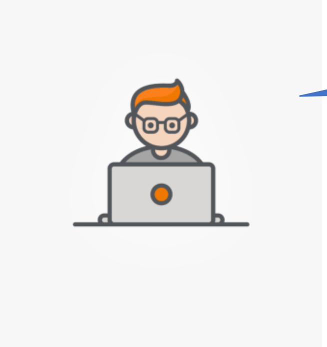
- Despite the popularity of NoSQL systems, they are not optimized for spatial data

- Major limitations:
  - No optimized spatio-temporal indexing methods (only limited support for spatial data)
  - No support for declarative querying (such as SQL)
  - Different (heterogeneous) query languages

Motivation



Motivation

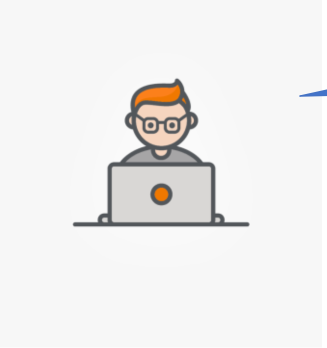


Let's develop a big data application using MongoDB

Listing 1.1: Typical code for a MongoDB filter query.

```
MongoClient mongoClient = new MongoClient();
MongoCollection m = mongoClient
    .getDatabase("test").getCollection("collection");
FindIterable r = m.find(and(gte("price",50),lte("price",80)));
mongoClient.close();
```

Motivation

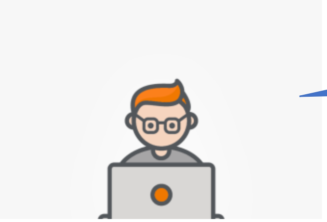


Let’s now develop the same application using HBase

Listing 1.1: Typical code for a MongoDB filter query.

```
MongoClient mongoClient = new MongoClient();
MongoCollection m = mongoClient
    .getDatabase("test").getCollection("collection");
FindIterable r = m.find(and(gte("price",50),lte("price",80)));
mongoClient.close();
```

Motivation

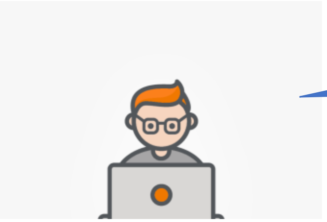


Let’s now develop the same application using HBase

Listing 1.2: Typical code for an HBase filter query.

```
Configuration c = HBaseConfiguration.create();
Connection connection = ConnectionFactory.createConnection(c);
Table table = connection.getTable(TableName.valueOf("test"));
Scan scan = new Scan();
byte[] columnFamily = Bytes.toBytes("products");
byte[] qualifier = Bytes.toBytes("price");
FilterBase f1 =  new SingleColumnValueFilter(columnFamily,
                 qualifier, CompareOperator.GREATER_OR_EQUAL,
                 Bytes.toBytes(50));
FilterBase f2 =  new SingleColumnValueFilter(columnFamily,
                 qualifier, CompareOperator.LESS_OR_EQUAL,
                 Bytes.toBytes(80));
scan.setFilter(new FilterList(f1,f2));
ResultScanner resultScanner = table.getScanner(scan);
table.close();
```

Motivation



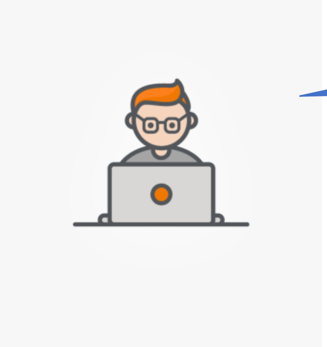
Let’s now develop the same application using Redis

Listing 1.2: Typical code for an HBase filter query.

```
Configuration c = HBaseConfiguration.create();
Connection connection = ConnectionFactory.createConnection(c);
Table table = connection.getTable(TableName.valueOf("test"));
Scan scan = new Scan();
byte[] columnFamily = Bytes.toBytes("products");
byte[] qualifier = Bytes.toBytes("price");
FilterBase f1 =  new SingleColumnValueFilter(columnFamily,
                 qualifier, CompareOperator.GREATER_OR_EQUAL,
                 Bytes.toBytes(50));
FilterBase f2 =  new SingleColumnValueFilter(columnFamily,
                 qualifier, CompareOperator.LESS_OR_EQUAL,
                 Bytes.toBytes(80));
scan.setFilter(new FilterList(f1,f2));
ResultScanner resultScanner = table.getScanner(scan);
table.close();
```

redis

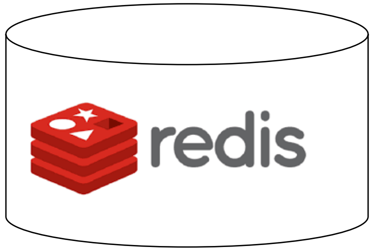
Motivation



Let’s now develop the same application using Redis

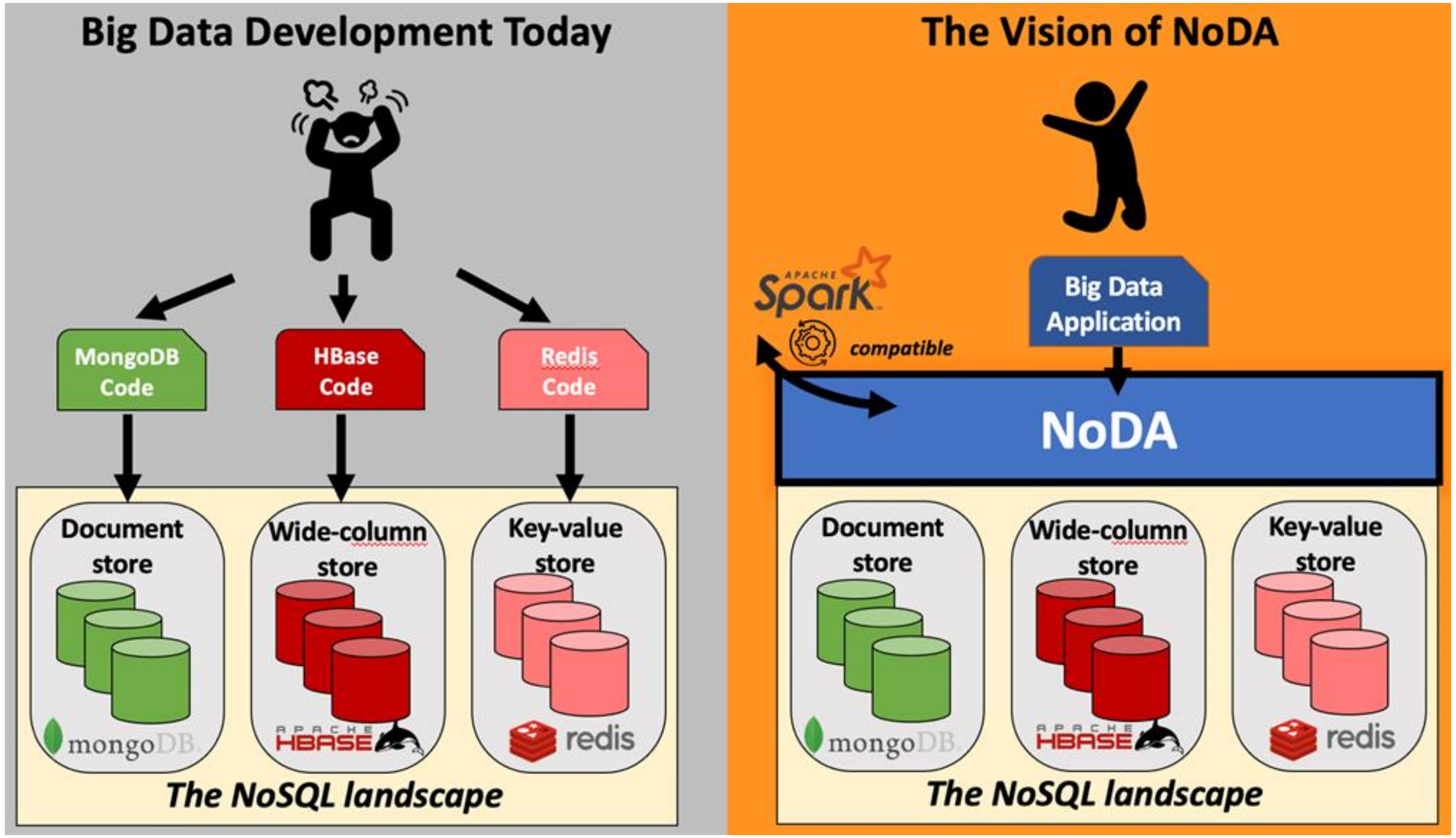
Listing 1.3: Typical code for a Redis filter query.

```
Jedis jedis = new Jedis();
Set<Tuple> rs = jedis.zrangeByScoreWithScores("price",50,80);
jedis.close();
```



The NoDA abstraction layer

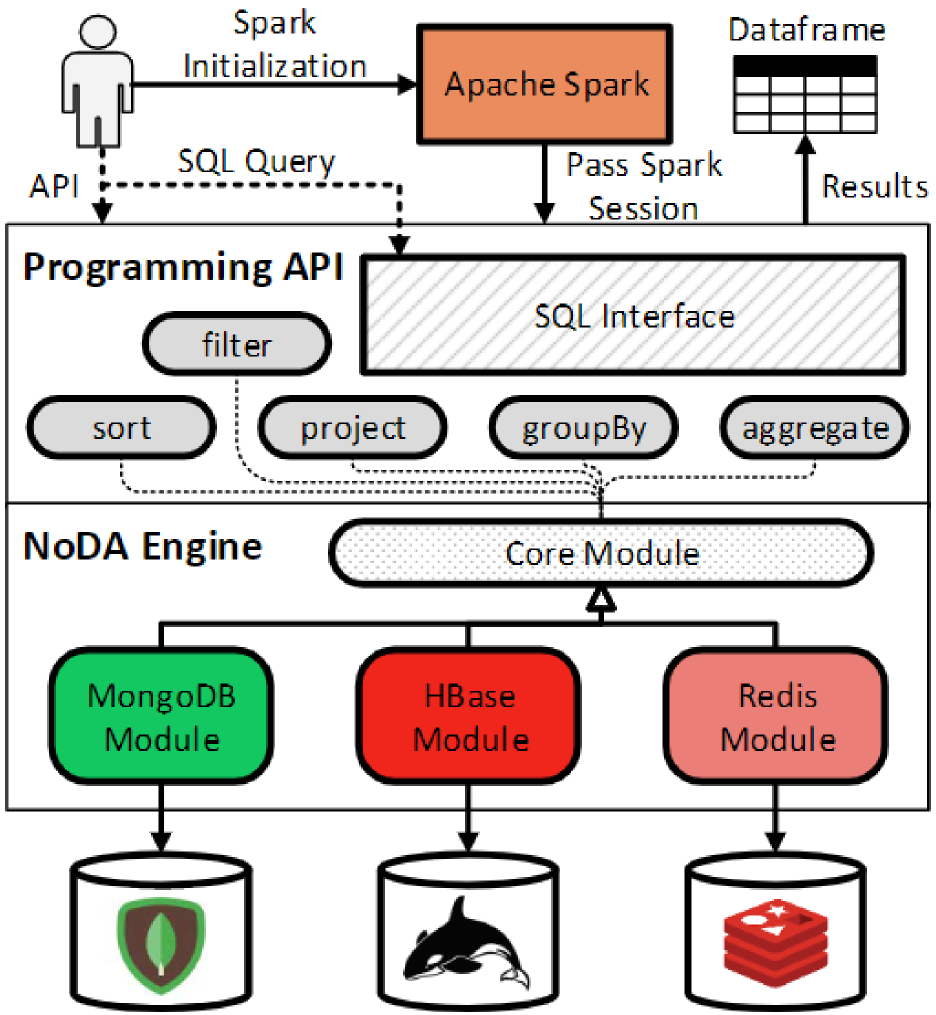
- We propose NoDA, an abstraction layer for simple and uniform access to different NoSQL stores



The NoDA Architecture

- It is composed of two components
  - The Programming API for big data developers
  - The Declarative (SQL) interface for data scientists

- The programming API provides a set of basic data access operators, supported by all NoSQL stores
- Examples of data access operators include filter, project, groupBy, aggregate and sort
- This enables the provision of an SQL interface to end users
- An SQL query is translated to a sequence of data access operators



NoDA: The programming API

- Access operators are expressed in the following way:

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

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

- At first, a connection is instantiated
  - noSqlDBSystem is an object reference that handles the connection
- Then, a query is formulated for execution
  - by specifying a sequence of data access operators, using method chaining
- NoDA can be optionally associated with a Spark session
  - useful for fetching the data objects in the form of a Spark Dataframe

NoDA: The declarative interface

- Additionally supports an SQL-like query language
- Every SQL clause is mapped to a specific NoDA operation, e.g.,

```
SELECT city,AVG(price_day)
FROM hotels
WHERE star = 5
GROUP BY city
HAVING AVG(price_day)>500
ORDER BY AVG(price_day)
LIMIT 20
```

→

```
.operateOn("hotels")
.filter(eq("star",5))
.groupBy("city")
.aggregate(avg("price_day"))
.filter(gt("AVG(price_day)",500))
.project("city","AVG(price_day)")
.sort(asc("AVG(price_day)"))
.limit(20)
```

NoDA Implementation on top of HBase

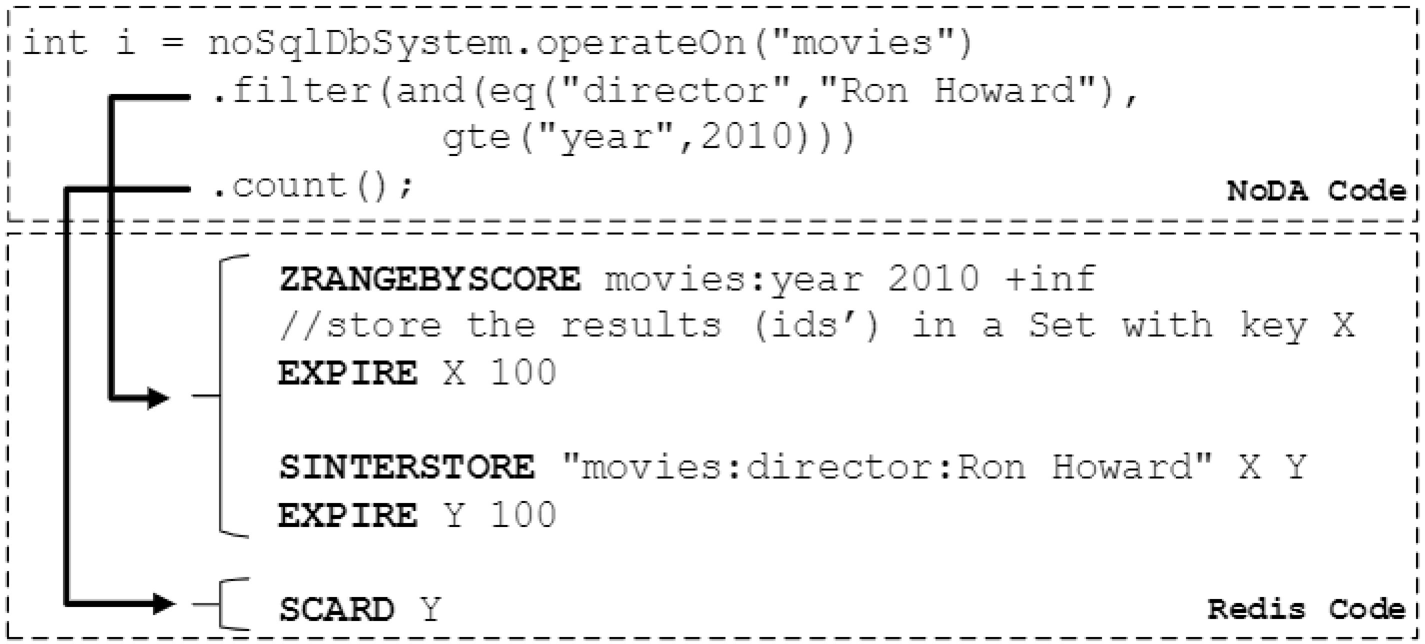
- Under the hood, NoDA capitalizes the libraries of each NoSQL store
- It uses its native query language for performing its abstract operations
- The implementation on top of HBase (wide-column store) store exploits its filters
- Complex operations in NoDA are transformed to a series of HBase filters

```
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
```

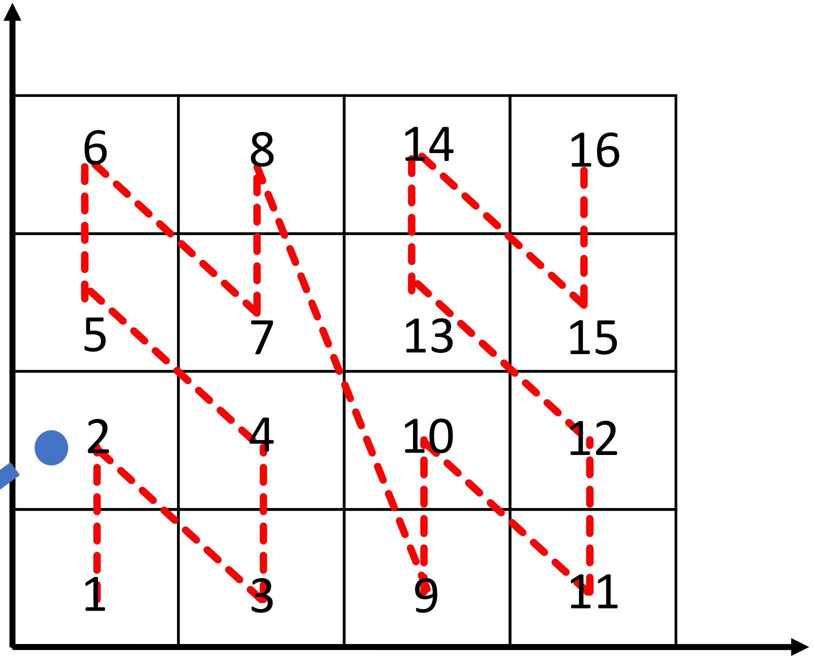
NoDA Implementation on top of Redis

- In Redis (key-value) store, data are modeled as key-value pairs
- This data model has a lower expressive power
- Redis pipelining is utilized, handling set structures in which set operations are performed



Spatio-temporal queries in NoDA/HBase (1/2)

- Wide-column store (HBase) case
- We encode the spatio-temporal information in the row key of each record
- We exploit the Geohash of the spatial coordinates *(x, y)*
- The Geohash is concatenated with the time (*t)* of the record in Unix timestamp format
- The final expression is then concatenated with a random string to ensure the key's uniqueness

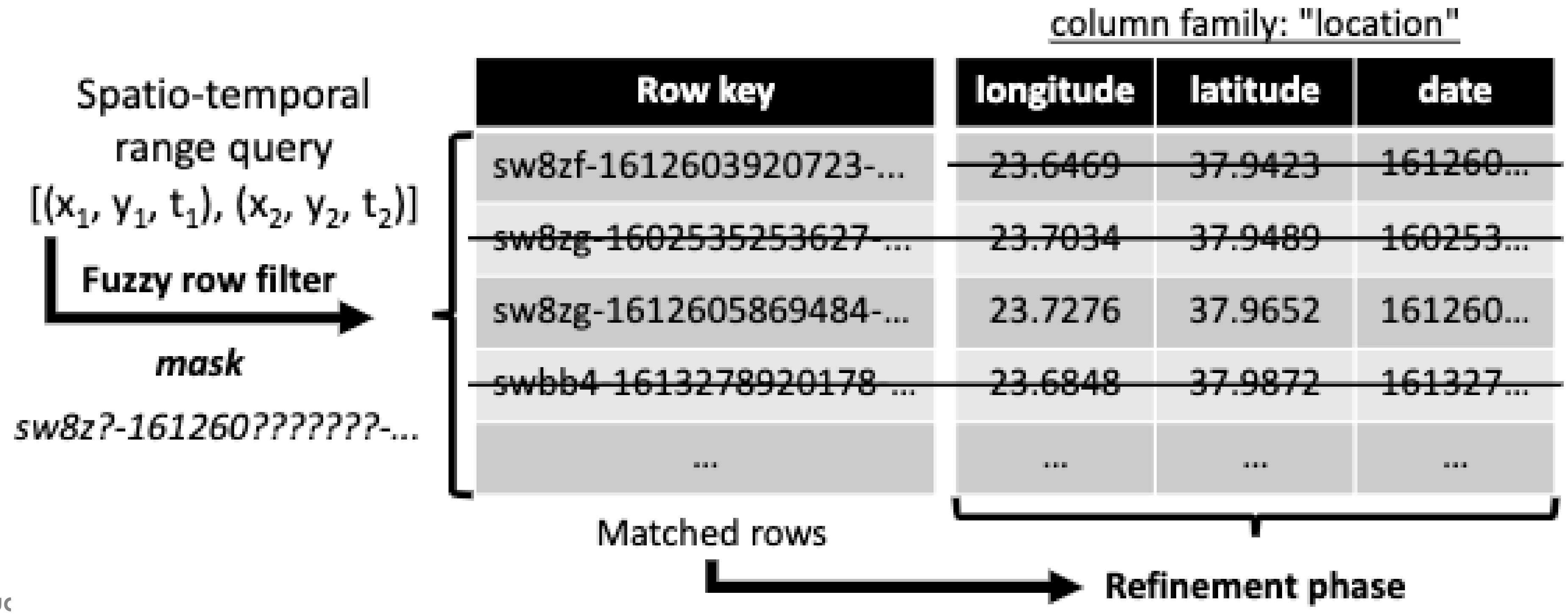


Z-order curve

suj4 – 1652432507473 – RANDOMSTRING

Spatio-temporal queries in NoDA/HBase (2/2)

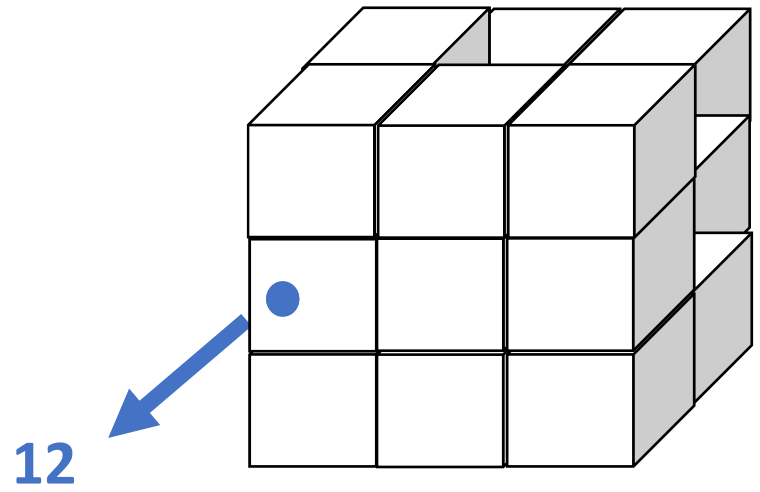
- Spatio-temporal queries are implemented via fuzzy row and the custom filters as server-side filters for filtering and refining the records
- The filters are executed on the regionservers (server-side)



Spatio-temporal queries in NoDA/Redis (1/2)

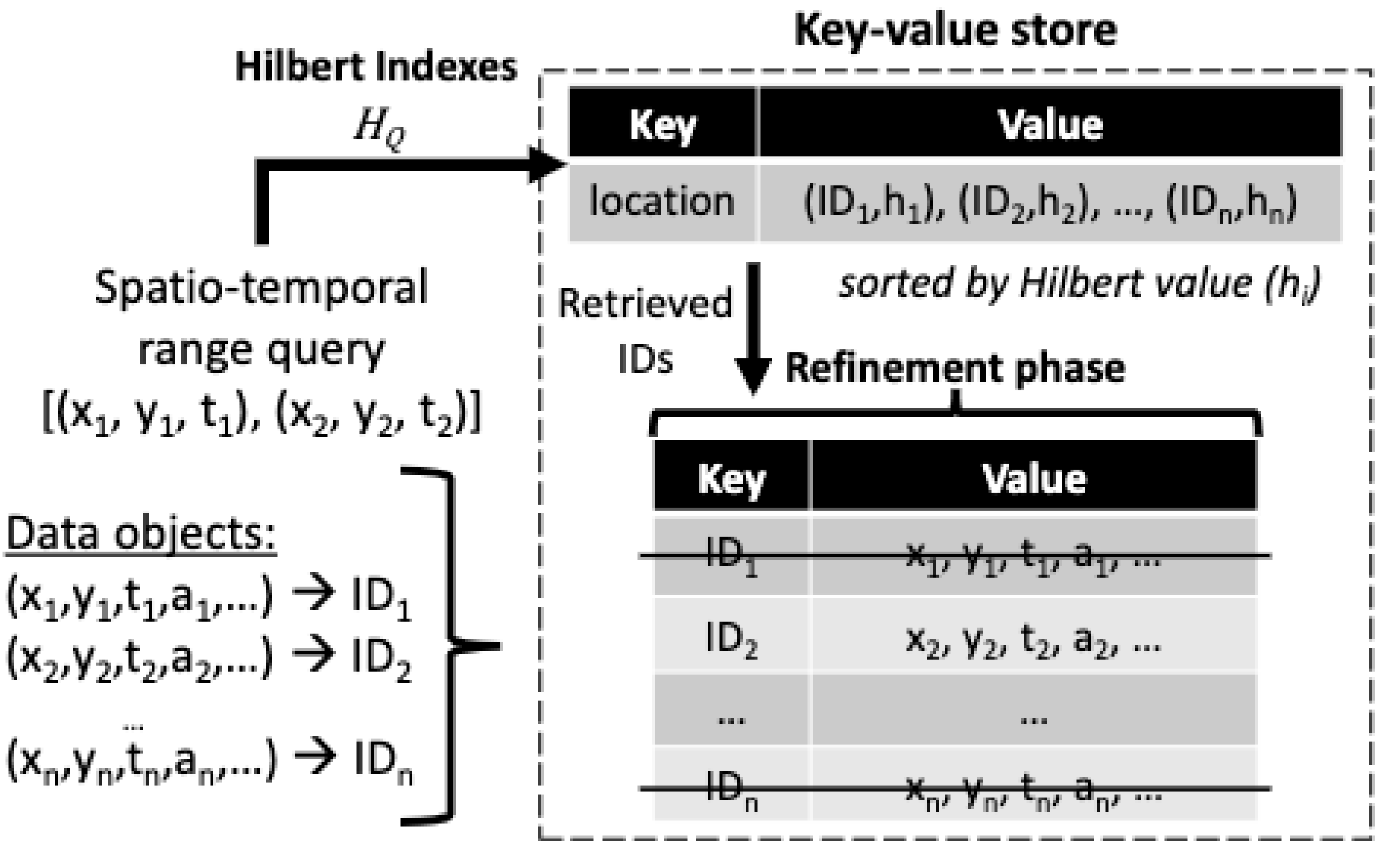
- Key-value store (Redis) case
- Each spatio-temporal record is stored in a key-value entry where the key is its *identifier* (ID)
- The value is a hash that stores the object's information
- Range queries are not supported in these stores
- We exploit the structures that are supported in the key-value entries
- The Hilbert value (*h)* of each 3D spatio-temporal record (x, y, t) is computed

Key	Value
ID_1	$x_1, y_1, t_1, a_1, \ldots$
ID_2	$x_2, y_2, t_2, a_2, \ldots$
…	…
ID_n	$x_n, y_n, t_n, a_n, \ldots$



Spatio-temporal queries in NoDA/Redis (2/2)

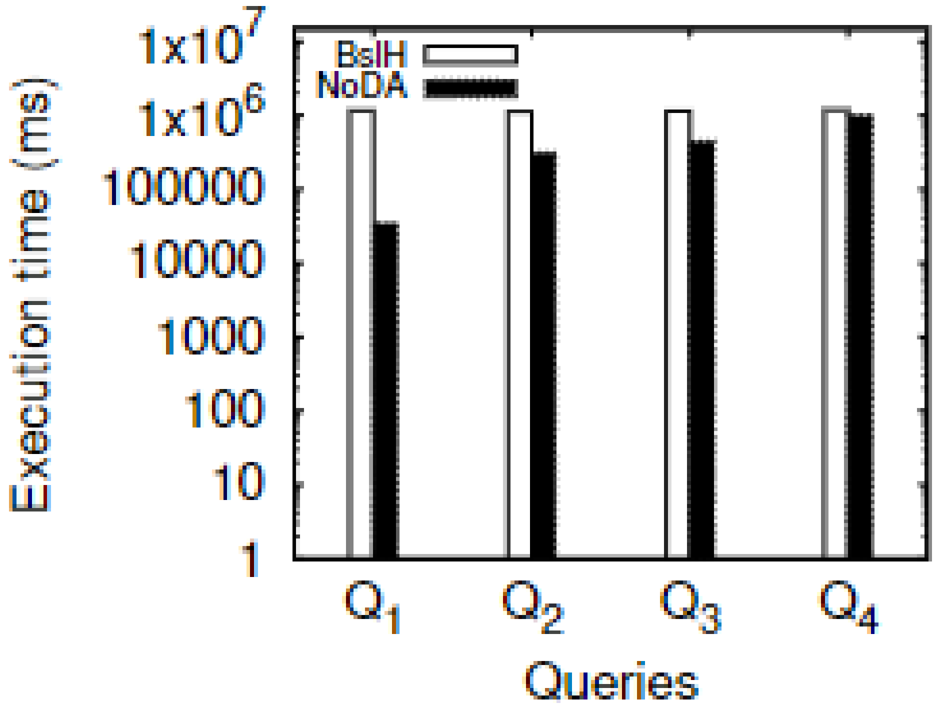
- The Hilbert values are stored with the record's identifier in a Sorted Set structure
- The structure is accessed by a dummy key, named "location"
- The set is sorted by the Hilbert value, effectively serving as an index



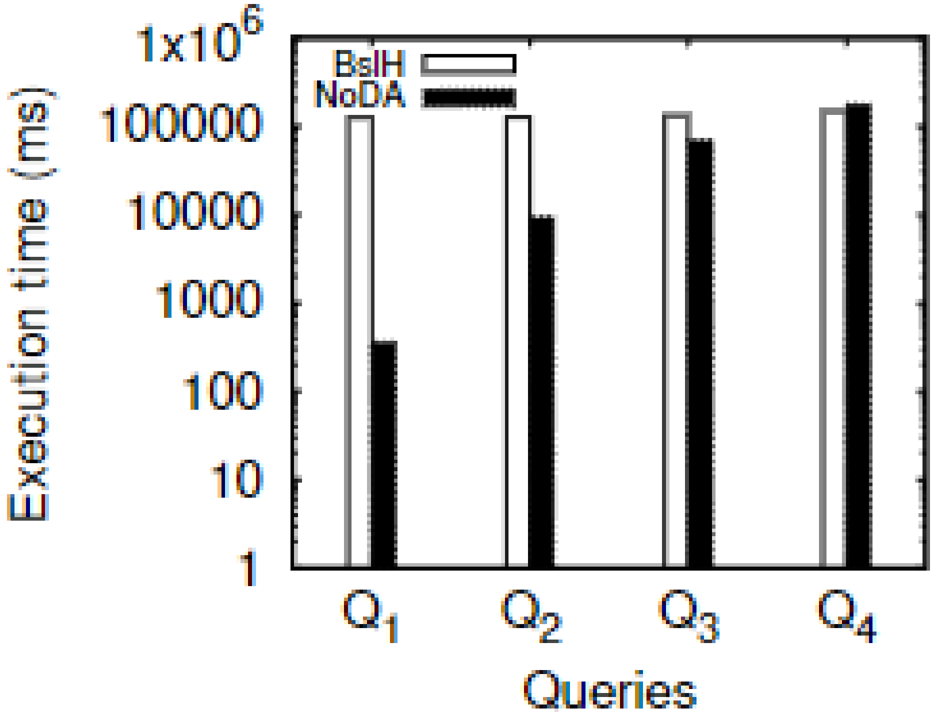
Experiments – Setup

- Carried out in a cluster of 17 virtual machines (VMs)
  - 12 VMs for data storage (shards, regionservers)
  - 2 VMs for data insertion and querying
  - 3 VMs for services (e.g., config servers for MongoDB and Namenode, Zookeeper for HBase)
- Real-life **(REAL)** data and synthetic **(SYNTH)** data sets are used
- Spatio-temporal queries
  - Used 4 queries with fixed temporal interval while varying the spatial selectivity
- Comparative execution time is measured
  - NoDA vs. a baseline approach in each store

Experiments – HBase

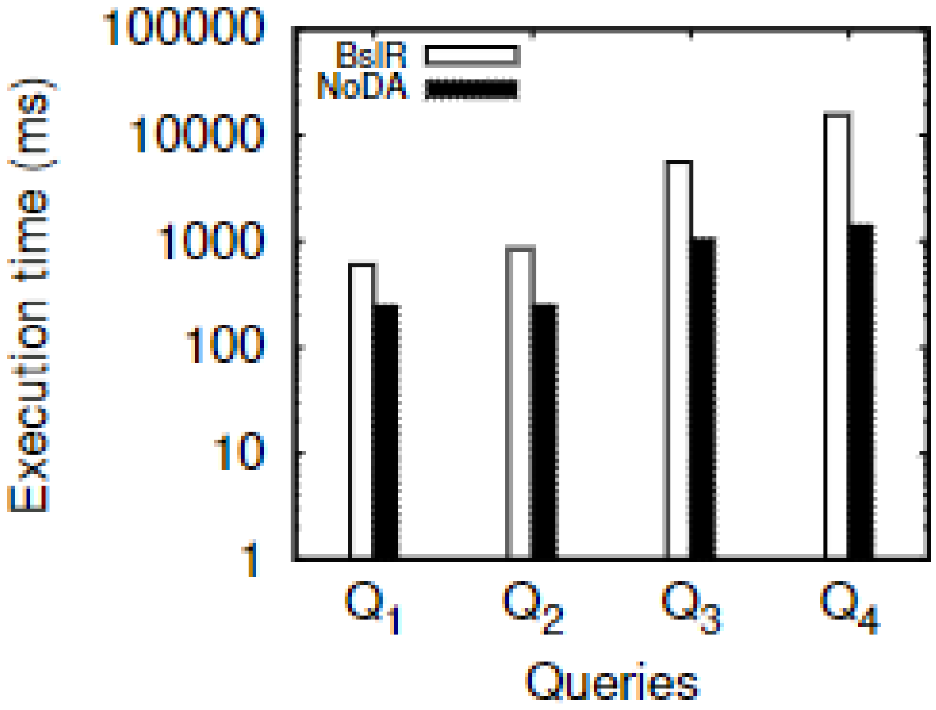


REAL

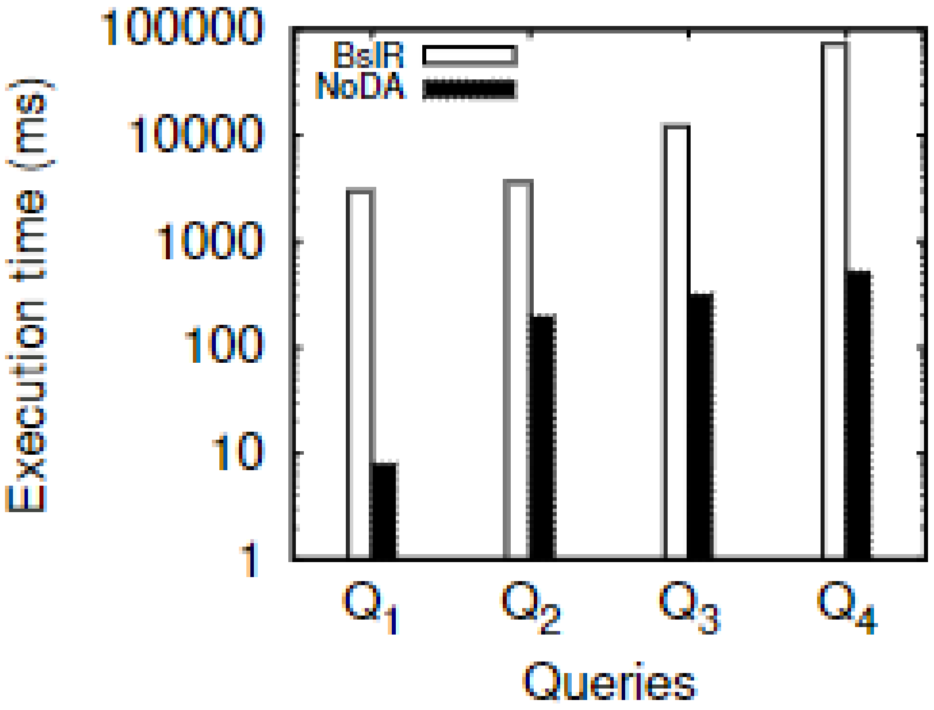


SYNTH

Experiments – Redis

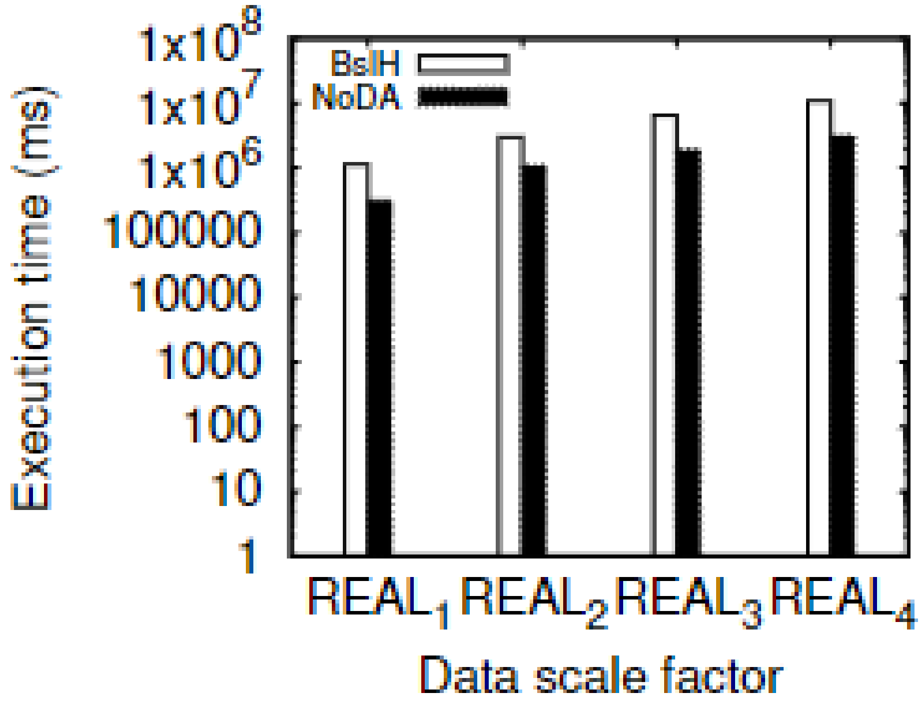


REAL

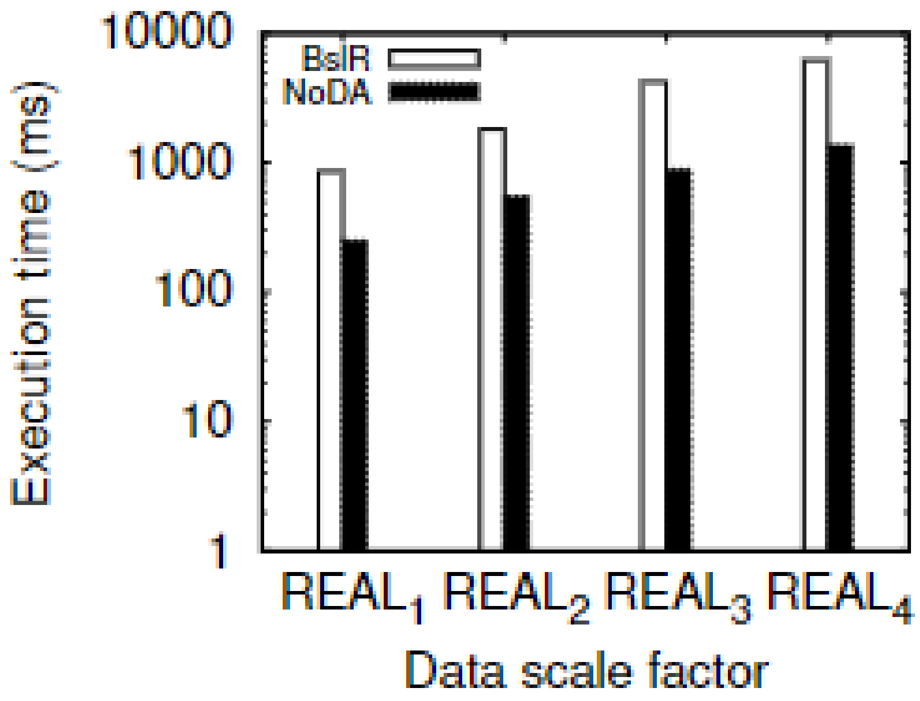


SYNTH

Experiments – Scalability w/ size of data set



HBase



Redis

Related Work

- Our work, relates to **polystores** (BigDAWG, Icarus, CloudMdsQL)
- Also, it relates to **data access query engines** (Pig Latin, Presto, Hive)
- NoDA is not tightly coupled with the data sources it supports
- NoDA provides data access in a unified manner (no matter the underlying store)
- Can be easily extended to other stores
- It resembles the JDBC interface, used for accessing relational databases

Conclusions

- We introduced NoDA for unified data access on top of NoSQL stores
- NoDA focuses on spatio-temporal data, providing spatio-temporal operators
- NoDA exposes both a programming API and an SQL interface
- Under the hood, the languages and native libraries of stores are exploited
- So far, it has been implemented on top of MongoDB, HBase and Redis

- Future work
  - Augment the set of mobility-oriented operators (e.g., add trajectory operators)
  - Integrate new operator types, e.g., support spatio-textual data
  - Implement a mechanism for operating on two or more stores simultaneously

Thank you for your attention

More info:

our group: *http://www.datastories.org/*

project Track & Know: https://trackandknowproject.eu/

project Spades: https://www.ds.unipi.gr/spades/

project Chorologos: https://www.ds.unipi.gr/chorologos/

e-mail: koutroumanis@unipi.gr