# JUST: JD Urban Spatio-Temporal Data Engine

Ruiyuan Li[1,2], Huajun He[3,2], Rubin Wang[3,2], Yuchuan Huang[2], Junwen Liu[2],
Sijie Ruan[1,2], Tianfu He[4,2], Jie Bao[2], Yu Zheng[1,2]
[1] *Xidian University, Xi'an, China* [2] *JD Intelligent Cities Research, Beijing, China*
[3] *Southwest Jiaotong University, Chengdu, China* [4] *Harbin Institute of Technology, Harbin, China*
{ruiyuan.li, hehuajun3, wangrubin3, huangyuchuan, liujunwen8, ruansijie, hetianfu3, baojie3, zheng.yu}@jd.com

***Abstract*—With the prevalence of positioning techniques, a prodigious number of spatio-temporal data is generated constantly. To effectively support sophisticated urban applications, e.g., location-based services, based on spatio-temporal data, it is desirable for an efficient, scalable, update-enabled, and easy-to-use spatio-temporal data management system.**

**This paper presents JUST, i.e., JD Urban Spatio-Temporal data engine, which can efficiently manage big spatio-temporal data in a convenient way. JUST incorporates the distributed NoSQL data store, i.e., Apache HBase, as the underlying storage, GeoMesa as the spatio-temporal data indexing tool, and Apache Spark as the execution engine. We creatively design two indexing techniques, i.e., Z2T and XZ2T, which accelerates spatio-temporal queries tremendously. Furthermore, we introduce a compression mechanism, which not only greatly reduces the storage cost, but also improves the query efficiency. To make JUST easy-to-use, we design and implement a complete SQL engine, with which all operations can be performed through a SQL-like query language, i.e., JustQL. JUST also supports inherently new data insertions and historical data updates without index reconstruction. JUST is deployed as a PaaS in JD with multi-users support. Many applications have been developed based on the SDKs provided by JUST. Extensive experiments are carried out with six state-of-the-art distributed spatio-temporal data management systems based on two real datasets and one synthetic dataset. The results show that JUST has a competitive query performance and is much more scalable than them.**

## I. Introduction

With the ubiquity of positioning techniques, a myriad of spatio-temporal data is generated constantly. For example, there are about 1T GPS logs generated by over 60,000 couriers each day in JD[1]. Besides, there are more than 30 million orders per day in JD online mall, where each order is associated with a delivery address. These spatio-temporal data is very useful for many urban applications. For example, the GPS logs of couriers can help recovering digital maps in living areas [1], which is essential for path planning and package dispatching. The purchase orders give us insight into the economy or functions of an urban area, which is important for location selection and urban planning.

However, it is non-trivial to manage spatio-temporal data. One reason is that, the spatio-temporal data is generated continuously and can have huge size. Another reason is that, the spatio-temporal data is inherently complex with at least three dimensions, i.e., the latitude, the longitude, and the time information. Traditional relational databases, such as Oracle Spatial, MySQL Spatial and PostGIS, support spatio-temporal data management, but they usually fail when the data gets large. In the last decade, distributed frameworks such as Apache Hadoop [2] and Apache Spark [3] have been proved to be scalable for big data storage and processing. The spatio-temporal data management systems based on Hadoop, e.g., [4–8], could face the efficiency problem, as Hadoop stores its intermediate result on the disk, resulting in multiple disk I/Os even for a single job. Spark caches data in memory with an abstraction of RDD (Resilient Distributed Dataset), therefore it is more efficient for big data computing. Most of the Spark-based spatio-temporal data management systems, e.g., [9–15], build huge R-tree, KD-tree, or grid indexes in memory, thus can achieve efficient spatio-temporal data management. However, these systems load all data into memory, which requires high-performance clusters with much memory. Hence, their scalability is limited. Besides, when a spatio-temporal request comes, they need to scan huge indexes, which is time-consuming. Most of the Hadoop-based or Spark-based systems do not support data updates, as the indexes they build contain the correlations among different records. If there comes new data, they will reconstruct indexes from scratch.

Distributed NoSQL (Not Only SQL) data stores, such as HBase [16], achieve millions of updates per second. However, due to lack of secondary indexes, these NoSQL data stores do not natively support spatio-temporal data management. There are many spatio-temporal data management systems [17–22] based on NoSQL data stores. For example, GeoMesa [17], an open-source indexing tool, transforms multi-dimensional data into one-dimensional keys. As a result, it can manage large-scale spatio-temporal data on the top of distributed NoSQL data stores. However, these systems are mainly for data storage. They do not provide data processing methods or SQL language. As a result, it is hard to use these systems, since we need to dive into their development manuals, and implement our own spatio-temporal analysis methods and predicates.

This paper presents JUST [23], i.e., JD Urban Spatio-Temporal data engine, which can efficiently manage big spatio-temporal data in a convenient way based on GeoMesa. JUST has several notable characteristics: 1) **Efficiency**. We find that the indexes built by the native GeoMesa are not suitable for spatio-temporal range queries. To this end, we design two novel indexing strategies, i.e., Z2T and XZ2T, which expedites the query efficiency tremendously. We also introduce a compression mechanism, which greatly reduces the storage cost and improves the query efficiency by reducing disk IOs; 2) **Scalability**. JUST is based on the distributed

Yu Zheng and Jie bao are the corresponding authors of this paper.
[1]https://en.wikipedia.org/wiki/JD.com

TABLE I
COMPARING JUST AGAINST OTHER SYSTEMS

| Features | JUST | Simba | STARK | ST-Hadoop | SparkGIS | Hadoop-GIS | SpatiolHadoop | GeoSpark | LocationSpark | SpatialSpark | MD-HBase | BBoxDB |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Category | NoSQL | Spark | Spark | Hadoop | Spark | MR/Hive | Hadoop | Spark | Spark | Spark | NoSQL | NoSQL |
| Scalability[a] | Yes | Limited | Limited | Yes | Limited | Yes | Yes | Limited | Limited | Limited | Yes | Yes |
| SQL | Yes | Yes | Yes | Yes | No | Yes | Yes | No | No | No | No | No |
| Data Update | Yes | No | No | Limited[b] | No | No | No | No | Yes | No | Yes | Yes |
| Data Processing | Yes | No | No | No | No | Yes | No | Yes | Yes | No | No | No |
| S or ST[c] | S/ST | S | S/ST | S/ST | S | S | S | S | S | S | S | S |
| Non-point data | Yes | Not Present | No | No | No | No | No | Yes | Yes | No | No | Yes |

[a] We test most of these systems in Section VIII. [b] ST-Hadoop only supports data updates in future time. For historical data insertions, it fails.
[c] S and ST mean spatial and spatio-temporal data support, respectively.

NoSQL data store HBase, and uses Spark as execution engine. Consequently, JUST can manage massive spatio-temporal data with limited resources; 3) **Update-enabled**. The spatio-temporal information is encoded in the keys of HBase, where the key of a record has nothing to do with that of others. As a result, JUST supports new data insertions or historical data updates; 4) **Easy to use**. We design and implement a complete SQL engine, and preset plenty of out-of-the-box spatio-temporal analysis functions. All operations of JUST can be done with a SQL-like query language, i.e., JustQL.

The contributions of this paper are summarized as follows:

(1) We design and implement a holistic distributed system, i.e., JUST, with which users can efficiently manage big spatio-temporal data in a convenient way.

(2) We propose two novel indexing techniques, i.e., Z2T and XZ2T, in NoSQL environments, which accelerates spatio-temporal queries tremendously. A compression mechanism is introduced, which not only greatly reduces the storage cost, but also improves the query efficiency.

(3) We implement a complete SQL engine with many out-of-the-box operations preset, based on which all operations can be performed through a SQL-like query language, i.e., JustQL. The SQL engine makes JUST extremely easy to use.

(4) JUST is deployed as a PaaS in JD with multi-users support. Many urban applications have been developed based on the SDKs provided by JUST. Extensive experiments using two real datasets and one synthetic dataset show that JUST outperforms six state-of-the-art systems in terms both of efficiency and scalability. Readers can experience JUST in [23].

JUST is under active development now, and significant new features are added frequently. This paper presents JUST of version 1.1.0, which is released in Oct, 2019. The remainder of this paper is organized as follows. We first review the related spatio-temporal data management systems in Section II. The architecture and data flow of JUST are presented in Section III. Section IV to Section VII describe each layer of JUST, respectively. We present the evaluation results in Section VIII, and conclude this paper with future works in Section IX.

## II. RELATED WORKS

In this section, we review related works from four aspects: 1) Relational Databases, 2) Hadoop-based Systems; 3) Spark-based Systems; and 4) NoSQL-based Systems.

**Relational Databases.** Traditional relational databases, e.g., Oracle Spatial, MySQL Spatial, and PostGIS, integrate spatial data types and operations to manage spatial data. These systems build spatial indexes, such as R-tree, K-D tree, or Quad-tree for fast spatio-temporal queries. However, they always encounter bottlenecks in big data scenarios. For example, they usually lack effective spatial partition mechanisms to achieve load balance. As a result, they suffer from a scalability issue, especially for the complex spatial data types.

**Hadoop-based Systems.** Hadoop is a disk-based distributed framework, providing high scalability, high availability and fault tolerance for big data storage and processing. There are many spatial or spatio-temporal systems built upon Hadoop [4–8]. For example, SpatialHadoop [5, 6] extends Hadoop to support spatial data types and functions. It proposes spatial partitioning and indexing methods, and supports spatial range queries, $k$-NN queries, and spatial joins. Hadoop-GIS [7] extends Apache Hive with a unified grid index to support spatial data queries. It proposes a spatial query engine with an optimizer, i.e., RESQUE, and a multi-layer index to optimize spatial partitioning and parallel processing over MapReduce. Neither SpatialHadoop nor Hadoop-GIS supports spatio-temporal queries. ST-Hadoop [7] is an extension of SpatialHadoop that integrates spatio-temporal concepts in each layer of SpatialHadoop, thus it can support spatio-temporal range queries and joins. Although these systems exhibit high scalability for spatial/spatio-temporal data management, Hadoop will read and write the disks many times even for a single job, which cuts down the system efficiency.

**Spark-based Systems.** Apache Spark is a distributed in-memory computing system, which is built on the top of RDD [3]. Comparing Hadoop, Spark is more efficient as it caches data in memory as much as possible. However, Spark itself does not support spatial/spatio-temporal data management inherently. GeoSpark [9, 10] extends Spark by providing Spatial RDDs (SRDDs) to support spatial range queries, $k$-NN queries, and spatial joins. Nevertheless, SRDDs can only retain one certain geometric type. Besides, GeoSpark lacks of a global index, which limits its performance. SpatialSpark [11] adopts fixed grid partitioning, binary space partitioning, and tile partitioning on spatial data, and supports spatial range queries. LocationSpark [12] provides a set of spatial query operators, including spatial range queries, $k$-NN queries, and spatial joins. It employs various spatial indexes, e.g., grid, R-tree, Quad-tree, and IR-tree. It also designs a dynamic memory caching framework, which flushes less frequently accessed data into disk. To avoid out of memory exceptions, SparkGIS [13] employs dynamic query rewriting to

manage large spatial query workflows that exceed available resources. STARK [14] adds spatio-temporal support to Spark. It includes spatial partitioners, various indexes, and multiple operations. Simba [15] is built on SparkSQL that supports SQL and DataFrame to manipulate spatial data. These Spark-based systems load all data into memory, which requires high-performance clusters with much memory. As a result, their scalability is limited. Besides, for each spatio-temporal request, they need to scan huge indexes, which is costly.

**NoSQL-based Systems.** NoSQL data stores, such as Cassandra [24] and HBase [16], achieve millions of updates per second, and provide high scalability, and random-access data management. Many works [17–22] provide spatial or spatio-temporal data access based on NoSQL. MD-HBase [18] extends HBase to support spatial queries. It provides two index structures, i.e., KD-tree and Quad-tree, to support spatial range and $k$-NN queries. BBoxDB [19] designs a two-level index on the top of NoSQL data stores, in which the global index applies a KD-tree to map which node stores data, and the local index uses an R-tree to locate the partition of each data item in each node. However, these frameworks do not support spatio-temporal data. GeoMesa [17] manages massive spatio-temporal data over distributed NoSQL data stores. The main idea behind GeoMesa is to transform multi-dimensional data into one-dimensional linear keys using space filling curves [25]. As a result, records closed in spatio-temporal dimensions are converted into keys closed in lexicographical order, thus these data can be stored and loaded efficiently in a batch. Most of these NoSQL-based systems is hard to use. Users need to delve into the handbooks and implement their own spatio-temporal query predicates and operations.

Most existing spatio-temporal systems mentioned above do not support data updates. They need to rebuild the indexes from scratch with new data coming. JUST supports data updates, and encapsulates various out-of-the-box data processing methods. Table I compares the key features of JUST with existing spatio-temporal systems.

## III. Architecture of JUST System

JUST adopts HBase as the underlying storage, GeoMesa as the indexing tool, and Spark as the execution engine. Figure 1 presents the overall architecture of JUST, where the new proposed modules are highlighted by orange boxes. As shown in Figure 1, JUST comes with six layers:

**Data Source Layer.** JUST can load spatio-temporal data from multiple data sources. Currently, JUST can load data directly from 1) disk files of CSV/GPX/KML/GeoJson format in a single machine or HDFS; 2) tables in a Apache Hive [26]; and 3) tables in Apache HBase [16].

**Indexing & Storing Layer.** In this layer, JUST builds spatial or spatio-temporal indexes over the loaded data. We propose various novel indexing strategies to accelerate spatio-temporal range queries. Besides, we innovatively identify three types of data tables and one type of meta table. Moreover, we devise a novel compression mechanism, which not only reduces the underlying storage size, but also expedites the query efficiency by reducing disk IOs (detailed in Section IV).

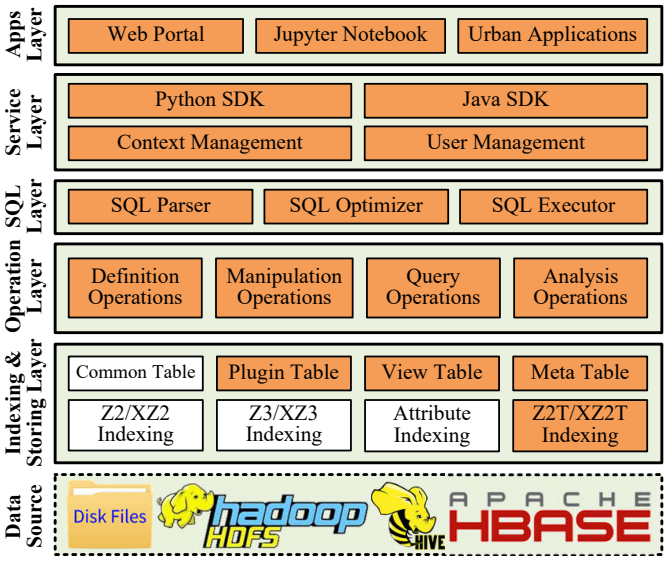



Fig. 1. Architecture of JUST.

**Operation Layer.** This layer presets four types of operations: 1) *Definition Operation*, which creates/drops tables or views in JUST; 2) *Manipulation Operation*, which inserts data into JUST tables; 3) *Query Operation*, which retrieves data from tables or views; and 4) *Analysis Operation*, which encapsulates various spatio-temporal data mining and processing functions, such as trajectory preprocessing, trajectory map-matching, and spatial clustering (detailed in Section V).

**SQL Layer.** This layer seamlessly integrates with Spark SQL. In other words, all functions of Spark SQL are supported by JUST. It first parses a query statement into a parse tree, then generates a logical plan. We propose a logical optimizer to convert the logical plan into a better one. Finally, JUST intelligently transforms the logical plan into Spark SQL operations or JUST operations (detailed in Section VI).

**Service Layer.** To support multiple users, while eliminating the cost of Spark context construction for each user, JUST maintains a Spark context shared by all users, which speeds up the query processing tremendously. Besides, JUST provides multiple programming language SDKs, including Java SDK and Python SDK (detailed in Section VII).

**Application Layer.** Users can use JUST to manage spatio-temporal data through either JUST web portal or JUST jupyter notebook [23]. In addition, developers can build their own ur-

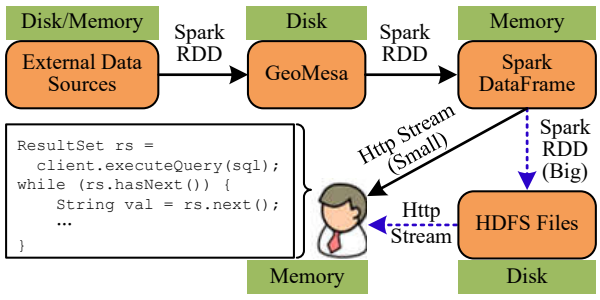



Fig. 2. Data Flow of JUST.

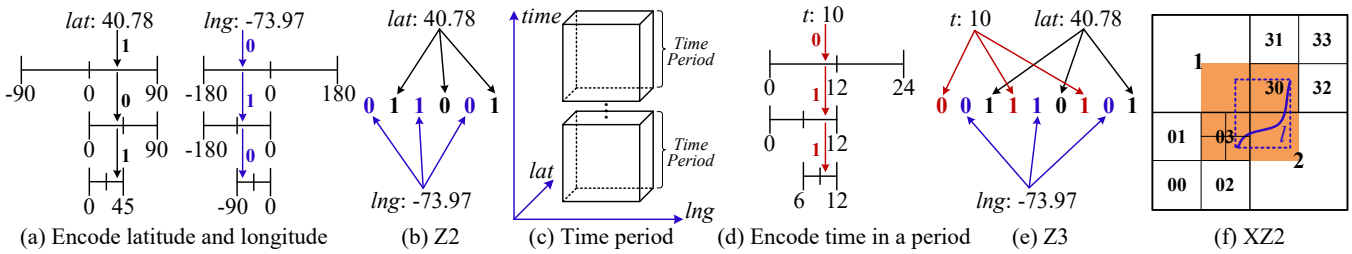



Fig. 3. Indexing Strategies of Native GeoMesa.

ban spatio-temporal applications based on the SDKs provided by JUST (detailed in Section VII).

Figure 2 illustrates the complete data flow of JUST. The spatio-temporal data is loaded from external data sources into GeoMesa using Spark (in the rest of this paper, we call GeoMesa along with its underlying storage as GeoMesa). GeoMesa would build spatio-temporal indexes for these data according to the configuration of users. These two processes need to be done only once. To answer a spatio-temporal query, JUST first performs spatio-temporal operations over GeoMesa using Spark, then transforms the result into Spark `DataFrame`. Thus we can leverage the complex aggregate operations, e.g., `GROUP BY`, and other powerful functions of Spark SQL. If the final result is smaller than a configurable parameter, it would be directly returned to users. Otherwise, to avoid an out-of-memory exception of the driver program, the result will be returned to users in multiple transmissions. Specifically, JUST would split the result to a set of small parts, and store them into HDFS. Driver program would read these HDFS files one by one. This process is handled by the JUST SDKs, and transparent to end users. Users can traverse the result in a way like the database cursor.

## IV. Indexing & Storing Layer

In this section, we first briefly introduce some existing indexing strategies provided by GeoMesa. However, as existing indexing strategies are not fit for spatio-temporal range queries in most cases, we propose various novel indexing strategies to accelerate spatio-temporal queries tremendously. Finally, we elaborate the underlying storage data models of JUST.

### A. Existing Indexing Strategies of GeoMesa

GeoMesa provides various indexing strategies, some of which that are related to this paper including:

**Z2 and Z3 Indexing Strategies.** Z2 and Z3 indexing strategies are used to index point-based data. Z2 is based on Z-ordering [27] function to project two-dimensional geographical coordinates onto one-dimensional keys, which support spatial range query efficiently. As shown in Figure 3a, it first respectively encodes the latitude $lat$ and longitude $lng$ of a record into a binary vector $B$, which is in a way like a binary search. If the value is located in the left half of the search space, it appends "0" to $B$, otherwise, it appends "1". This process is repeated until the length of $B$ is reached to a specified value $\alpha$. After that, it crosswise combines the two binary codes into a single one, as shown in Figure 3b.

Z3 is proposed to support spatio-temporal range query. It regards the time as the third dimension. As the time dimension is unbounded, Z3 breaks it into disjoint time periods (e.g., a day, a week, a month, or a year), as shown in Figure 3c. The time period number is calculated by Equ (1):

$$Num(t) = \lfloor (t - RefTime) \div TimePeriodLen \rfloor \quad (1)$$

where $RefTime$ is the reference time (e.g., 1970-01-01T00:00:00Z), and $TimePeriodLen$ is the time span of a time period. For each time period, Z3 encodes the time using the same technique. Figure 3d gives an example of time encoding with the time period of a day. Finally, Z3 crosses spatio-temporal codes into a single one, as shown in Figure 3e.

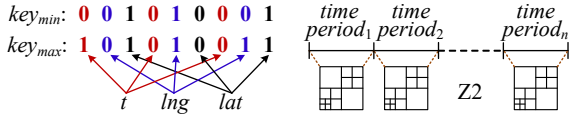
**XZ2 and XZ3 Indexing Strategies.** XZ2 and XZ3 are for non-point data (e.g. polygons or lines) based on XZ-ordering [28], an extension of Z-ordering. The main idea of XZ2 is to find a square that *just* contains the spatial MBR (Minimum Bounding Rectangle) of a record. For example, in Figure 3f, XZ2 finds the orange square to represent the line $l$. Likewise, by taking the similar technique of Z3, XZ3 wants to find a cube to stand for a spatio-temporal range.

For each indexing strategy, GeoMesa adds a random prefix to the generated keys, which distributes records across region servers and achieves load balance. For more details about the indexing strategies of GeoMesa, please refer to [17, 28, 29].

### B. Z2T Indexing Strategy

**Motivation.** We find that in most cases, the spatio-temporal range query efficiency is much slow with Z3 indexing strategy. For example, if we want to retrieve the data records in a $1km \times 1km$ spatial area from 01:00 to 13:00 in one day (this is a very generic query in many urban applications), we might get a key range shown as Figure 4a. It covers most part of a time period, where most of them are beyond the interested spatial area. As a result, we need to scan most of the data records, regardless of the fact that we are only interested in a small spatial region.

After taking a closer look at it, we find the primary reason is that, the scale of spatial dimension is different from that of temporal dimension. The ratio of interested time range with respect to the time period (e.g., $(13-1)/24$) is much greater than the ratio of interested spatial range with respect




(a) Example of Key Range in Z3 (b) Z2T Indexing Strategy

Fig. 4. Motivation for Z2T.

to the Earth area (e.g., $(1 \times 1)/510,100,000$). Combining the spatio-temporal codes together from scratch may result in the invalidation of spatial filtering. To mitigate this issue, one may select a longer time period (e.g., a month or a year provided by GeoMesa). However, even if we select the longest time period provided by GeoMesa, i.e., a year, the time dimension still plays the major role (i.e., $(13-1)/(365\times24)$ vs $(1\times1)/510,100,000$), thus makes the spatial filtering useless. One may add even longer time periods, e.g., a decade or a century, to GeoMesa. It does not work because: 1) the temporal range of a query varies from several minutes to several months, a fixed time period cannot cope with all of these situations; and 2) if the time period is set too long, we cannot make full use of the time period number to perform temporal filtering, as all data records fall into the same time period.

**Z2T Indexing Strategy.** Borrowing the idea from [30], we propose Z2T indexing strategy, which first divides the time dimension into multiple disjoint time periods, then builds an individual Z2 index for the data records falling in each time period, as shown in Figure 4b. Specifically, to index a point-based spatio-temporal record $r = (lng, lat, t)$, we first calculate the time period number according to Equ (1), then calculate its spatial code using the Z2 function $Z2(lng, lat)$. In short, the key combination of Z2T indexing strategy for a record $r = (lng, lat, t)$ is shown as Equ (2):

$$Num(t) :: Z2(lng, lat) \tag{2}$$

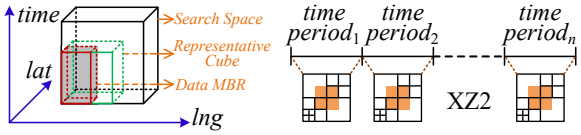
where "::" means a concatenation operation.

To answer a spatio-temporal range query $(lng_{min}, lat_{min}, t_{min}, lng_{max}, lat_{max}, t_{max})$ using Z2T indexing strategy, there are three steps. 1) We find all time periods that intersect with $[t_{min}, t_{max}]$. Suppose $m = Num(t_{min})$, $n = Num(t_{max})$, then all of the time periods $TimePeriod_i$, $m \leq i \leq n$, are qualified. 2) For each qualified time period $TimePeriod_i$ with a time period number $i$, we generate a key range $[key_{min}, key_{max}]$, where $key_{min} = i :: Z2(lng_{min}, lat_{min})$, and $key_{max} = i :: Z2(lng_{max}, lat_{max})$. 3) We trigger `SCAN` operations over the underlying key-value data store in parallel using the key ranges.

**Discussion.** The difference between Z2T and Z3 is that, Z2T builds a Z2 index instead of Z3 index in each time period. We can select a relatively small time period, e.g., a day or a week, in Z2T, thus we can use the time period to perform temporal filtering, meanwhile keep the spatial filtering ability with Z2.

### C. XZ2T Indexing Strategy

**Motivation.** XZ3 is designed for non-point-based spatio-temporal data (e.g., lines or polygons). However, like Z3, as the different scales between spatial dimension and temporal dimension, XZ3 would cause the spatial filtering ability to be lost, too. For example, as shown in Figure 5a, XZ3 aims to find a cube (marked in green) in the search space (marked in black) to represent the spatio-temporal data (whose MBR are marked in red). In most cases, the time dimension of the data is much larger than the spatial dimension, i.e., the spatial information does not take effect. As a result, it makes the spatial filtering invalid, which further results in scanning all data in each qualified time period.



(a) Example of XZ3 (b) XZ2T Indexing Strategy

Fig. 5. Motivation for XZ2T.

**XZ2T Indexing Strategy.** To address above issue, this paper proposes a new indexing strategy, i.e., XZ2T. Similar to Z2T, XZ2T first splits the time dimension into multiple disjoint time periods, then constructs an individual XZ2 index in each time period, as presented in Figure 5b. To index a non-point-based spatio-temporal record $r$, we first get its MBR $r.mbr = (lng_{min}, lat_{min}, t_{min}, lng_{max}, lat_{max}, t_{max})$, then calculate the time period that $r$ falls in according to Equ (1). Note here we use $t_{min}$ to calculate the time period number $Num(t_{min})$. Finally, we compute the spatial code using XZ2 function $XZ2(lng_{min}, lat_{min}, lng_{max}, lat_{max})$.

In summary, the key combination of XZ2T indexing strategy for the record $r$ with the MBR $r.mbr = (lng_{min}, lat_{min}, t_{min}, lng_{max}, lat_{max}, t_{max})$ is shown as Equ (3):

$$Num(t_{min}) :: XZ2(lng_{min}, lat_{min}, lng_{max}, lat_{max}) \tag{3}$$

The process to answer a spatio-temporal range query using XZ2T is similar to that of Z2T.

### D. Storage Data Models

The spatio-temporal data is stored in the underlying key-value NoSQL data store of GeoMesa, with the keys generated using one of the indexing strategies introduced above. We categorize the tables in JUST into three types of data tables and one type of meta table: *Common Table*, *Plugin Table*, *View Table*, and *Meta Table*.

**Common Table.** We call the table supported by GeoMesa itself as common table. Common table is the most flexible. Users can store most primitive types, geometric types, and our new proposed data types, e.g., `st_series` and `t_series`, in a common table. For example, users can create a common table using the following JustQL statement in JUST:

```
CREATE TABLE <tableName> (
  fid integer:primary key,
  name string,
  time date,
  geom point:srid=4326,
  gpsList st_series:compress=gzip|zip
) [USERDATA {'geomesa.indices.enabled':'z3'}]
```

where the contents in "[∗]" are optional. `USERDATA`, followed by a JSON string, is the hint to configure GeoMesa, e.g., to tell GeoMesa to build only Z3 index. We creatively propose a compression mechanism for the field with huge bytes. For example, we compress the field of `gpsList` with the gzip or zip method. The compression of fields not only reduces the storage cost, but also accelerates the query efficiency through reducing the disk IOs.

**Plugin Table.** For easy of use, JUST introduces a new concept, i.e., plugin table, which predefines the underlying storage

schema and the default adopted indexes of a data structure. In this way, users can reuse the codes to the maximum extent, and focus their minds on the application logic. Another necessity of plugin tables is that, some operations regard each row of a plugin table as a complete entity. The projection of partial fields of a plugin table could be meaningless. We add an implicit field `item` for the plugin table, representing the complete entity. For example, JUST presets a plugin table of "trajectory" [31], whose fields are predefined as Figure 6. If we want to perform a map-matching operation [32, 33] over trajectories, we only need to retrieve the `item` field, and feed it into a map-matching function.

| Key | Value | | | | | | |
|---|---|---|---|---|---|---|---|
| XZ2T | MBR | $\text{Point}_{\text{start}}$ | $\text{Point}_{\text{end}}$ | $\text{Time}_{\text{start}}$ | $\text{Time}_{\text{end}}$ | GPS List | ⋯ |
| $key_1$ | $mbr_1$ | $p_{s1}$ | $p_{e1}$ | $t_{s1}$ | $t_{e1}$ | gzip(⋯) | ⋯ |
| ⋯ | ⋯ | ⋯ | ⋯ | ⋯ | ⋯ | ⋯ | ⋯ |

Fig. 6. Partial Fields of Trajectory Plugin Table.

To create a plugin table of trajectory, one can simply input the following JustQL statement.

```
CREATE TABLE <tableName>
AS trajectory [USERDATA {...}]
```

**View Table.** Recall that in Figure 2, to answer a spatio-temporal query, we should first retrieve data from the disk in GeoMesa. The query results might be used for further processing and analysis. To avoid retrieving data from the disk for every data analysis, we can cache the intermediate results in memory as Spark `DataFrame`, called view tables, using the following JustQL statement.

```
CREATE VIEW <viewName> AS SELECT ...
```

Once the user sessions are time out, their view tables would be cleared up from the memory. Users can store their view tables into common tables or plugin tables, using the following statement. If the common tables or plugin tables do not exist, JUST will create them automatically. View tables greatly expedite the analysis efficiency of JUST, achieving an objective of "one query, multiple usages".

```
STORE VIEW <viewName> TO TABLE <tableName>
```

**Meta Table.** Meta tables record the meta information of data tables in JUST, which is very useful in data exploration and query optimization. For example, meta tables indicate whether a data table is a common table or a plugin table, record the fields of a data table, and so on. Meta tables are transparent to normal users, thus normal users cannot manipulate meta tables directly. Meta tables are stored in a MySQL database, as Apache Hive [26] does, because the sizes of meta tables would not be too large, and we can benefit from the remarkable advantages of the relational database, such as full transaction support, powerful integrity constraint, and so on.

In the following, we call common tables and plugin tables as **tables**, but view tables as **views**.

## V. Operation Layer

JUST provides a set of operations, which can be divided into four categories: *Definition Operations*, *Manipulation Operations*, *Query Operations*, and *Analysis Operations*.

### A. Definition Operations

This type of operations changes the structure of tables or views in JUST. There are three types of definition operations:

(1) Create or drop tables or views in JUST. Examples of creating tables and views are given in Section IV-D. Here gives an example of dropping tables or views.

```
DROP TABLE|VIEW <tableName|viewName>
```

JUST will delete corresponding records in the meta table, and drop the given table in disk, or the given view in memory.

(2) List the authorized tables or views. It is efficient as we need only to visit the meta table in MySQL.

```
SHOW TABLES|VIEWS
```

(3) Describe the fields of a given table or view. As the field information of a table is stored in the relational MySQL database, we can get the results efficiently.

```
DESC TABLE|VIEW <tableName|viewName>
```

### B. Manipulation Operations

This type of operations stores data into JUST tables. We have described the method to store data into tables from views in Section IV-D. We can also load data from external data sources. For example, to load data from Hive data warehouse, we can use the following statement:

```
LOAD hive:<hiveDatabaseName.hiveTableName>
TO geomesa:<tableName>
CONFIG {
  'fid': 'trajId',
  'time': 'long_to_date_ms(timestamp)',
  'geom': 'lng_lat_to_point(lng, lat)',
  ...
}
[FILTER 'trajId="1068" limit 10']
```

where the JSON string following `CONFIG` is the field mapping relationship between the hive table and the JUST table. We preset a bunch of functions to transform the data types of Hive to that of JUST. The SQL filtering string, following `FILTER`, allows users to select partial data in Hive tables to be loaded.

### C. Query Operations

Query operations select data from tables or views. Currently, JUST supports spatial or spatio-temporal filtering and field value filtering operations over tables. Other complex queries like aggregate operations and `JOIN` operations, are also supported on views by leveraging Spark SQL. In this paper, we mainly focus on the most widely used three types of spatial or spatio-temporal query operations: *Spatial Range Query*, *Spatio-Temporal Range Query*, and *k-NN Query*.

**Spatial Range Query.** Given a spatial dataset $D$ and a rectangular spatial range $S = (lng_{min}, lat_{min}, lng_{max}, lat_{max})$, a spatial range query aims to find all records $r \in D$, where $r$ locates in the spatial range $S$. Spatial range query is very useful in many location-based services. For example, we can use a spatial range query to answer the questions like: "what are the restaurants within three kilometres of me?"

By default, JUST builds a Z2 index for point-based spatial data, and XZ2 index for non-point-based spatial data. Users can use the following statement to perform a spatial range query. JUST would first generate key ranges according to the

given MBR, using the Z2 or XZ2 function, then trigger SCAN operations over the underlying NoSQL data store.

```
SELECT fid, name, time, geom
FROM <tableName>
WHERE geom WITHIN st_makeMBR(lng_min, lat_min,
  lng_max, lat_max)
```

where geom is the name of geometry field (the same below).

**Spatio-Temporal Range Query.** Given a spatio-temporal dataset $D$, a rectangular spatial range $S = (lng_{min}, lat_{min}, lng_{max}, lat_{max})$, and a temporal range $T = (t_{min}, t_{max})$, a spatio-temporal range query returns all records $r \in D$, where $r$ is generated during $T$ and locates in the spatial range $S$. Spatio-temporal range query can be used in many urban applications, e.g., traffic flow prediction [34], where we use spatio-temporal range query to get the trajectories passing a specified spatial area during a certain time interval.

JUST builds a Z2T index (for point-based data) or ZX2T index (for non-point-based data) for the data with spatio-temporal fields by default. To perform a spatio-temporal range query in JUST, users need only to input the following JustQL statement. JUST would first calculate key ranges by the given spatio-temporal parameters, then perform SCAN operations on the underlying NoSQL data store.

```
SELECT fid, name, time, geom
FROM <tableName>
WHERE geom WITHIN st_makeMBR(lng_min, lat_min,
  lng_max, lat_max) AND time BETWEEN t_min AND t_max
```

**$k$-NN (Nearest Neighbor) Query.** Given a spatial dataset $D$, a query point $q = (lng, lat)$, and a positive integer number $k$, $k$-NN query finds a set of records $D' \subseteq D$, where $|D'| = k$, and for each $r_i \in D'$, $r_j \in D \setminus D'$, $d(q, r_i) < d(q, r_j)$. Here $d(*, *)$ is a distance function between two geometries. This paper focuses on the point-based records for $k$-NN query. Other non-point-based records, such as lines or polygons are also supported by JUST. We adopt Euclidean distance for simplicity in this paper. $k$-NN query is widely used in dispatch systems. For example, taxi companies use this function to find the nearest taxi cab to pick up a passenger.

```
SELECT fid, name, time, geom
FROM <tableName>
WHERE geom IN st_KNN(st_makePoint(lng, lat), k)
```

Users can use above statement to trigger a $k$-NN query. JUST regards spatial range query as a building block, whose main idea is to iteratively expand the query spatial range, until the most $k$ nearest records are found. Algorithm 1 gives the pseudo-code of $k$-NN query, which contains two steps:

(1) *Initialization* (Line 1-3). This step initializes some variables. Here, $cq$ is a priority queue that stores candidate records; $aq$ is a priority queue to record the areas that need to be queried; and $d_{max}$ is the currently maximum distance between $q$ and the records in $cq$. We define the distance between a point $q$ and an area $a$ as the minimum distance between $q$ and any point $p \in a$, shown as Equ (4).

$$d_A(q, a) = \min_{p \in a} d(q, p) \tag{4}$$

(2) *Expansion* (Line 4-11). This step pops an area $a$ from $aq$. If there are $k$ records in $cq$ and the distance between $q$ and $a$ is greater than $d_{max}$, the query process is terminated (Lemma 1, denoted as **Area Pruning**), and the records in $cq$ are returned (Line 12). If the size of $a$ is greater than a certain one, we partition $a$ into four equal size areas and add them to $aq$, then continue to check the next area. Otherwise, we trigger a spatial range query by $a$. Finally, we add all records $p \in D_{SR}$ to $dq$, and update $d_{max}$.

**Algorithm 1:** $k$-NN query

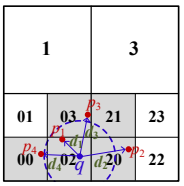
**Input:** Dateset $D$, query point $q$, positive integer $k$.
**Output:** $k$-NN query result $D'$.

1 Initial a priority queue $cq$ with a max size $k$, whose elements $r$ are ordered by $d(q, r)$;
2 Initial a priority queue $aq$ with the whole spatial area, whose elements $a$ are ordered by $d_A(q, a)$;
3 $d_{max} = 0$; // Maximum distance currently
4 **while** *$aq$ is not empty* **do**
5  $a = aq.pop()$;
6  **if** $cq.size() = k \wedge d_A(q, a) > d_{max}$ **then**
7  **break**; // Area Pruning
   // $g = 1km \times 1km$ is a system parameter, defining the minimum size of $a$
8  **if** *the size of $a > g$* **then**
9  Add the four children of $a$ to $aq$; **continue**;
10  $D_{SR}$ = Spatial range query by $a$;
11  Add all $p \in D_{SR}$ to $cq$; $d_{max} = d(q, cq.last())$;
12 **return** $cq$ as $\mathcal{T}_{knn}$;

**Lemma 1.** *If $d_A(q, a) > d_{max}$ and $|cq| = k$, then we can safely stop the expansion process for $k$-NN query.*

*Proof.* For any record $r$ that locates in $a$, we have $d(q, r) \geq d_A(q, a) = \min_{p \in a} d(q, p) > d_{max}$. That is to say, $r$ would not be in the $k$-NN result. Moreover, $a$ is the nearest area in $aq$ to $q$, hence for all records $r' \in a'$, $a' \in aq$, we have $d(q, r') > d_{max}$. Thus, we can safely stop the expansion process if $d_A(q, a) > d_{max}$ and $|cq| = k$. □

**Example**. Figure 7 gives an example of $k$-NN point query with $k = 3$, where the critical steps are shown in Figure 7b.




(a) 3-NN query

(1) Initially, $aq$=<02,2,03,00,01,1,3>,$cq$=<>,$d_{max}$=0;
(2) Query 02, $aq$=<20,03,21,00,01,22,23,1,3>,$cq$=<$p_1$>,$d_{max}$=$d_1$;
(3) Query 20, $aq$=<03,21,00,01,22,23,1,3>,$cq$=<$p_1$,$p_2$>,$d_{max}$=$d_2$;
(4) Query 03, $aq$=<21,00,01,22,23,1,3>,$cq$=<$p_1$,$p_3$,$p_2$>,$d_{max}$=$d_2$;
(5) Query 21, $aq$=<00,01,22,23,1,3>,$cq$={$p_1$,$p_3$,$p_2$},$d_{max}$=$d_2$;
(6) Query 00, $aq$=<01,22,23,1,3>,$cq$={$p_1$,$p_4$,$p_3$},$d_{max}$=$d_3$;
(7) Because $d$(01, $q$)>$d_{max}$ and $|cq|$=3, $cq$ is the result. Stop.

(b) Critical Steps of 3-NN query processing

Fig. 7. Example of $k$-NN Query ($k = 3$).

### D. Analysis Operations

JUST encapsulates various spatio-temporal analysis operations, which consist of three types:

**1-1 Analysis Operations.** This type of operations transforms each row of a view into another one record, which is implemented through Spark SQL UDF. For example, we provide a series of coordinate transformation functions among different standards. To transform a point from WGS84 to GCJ02, we can use the following statement:

```
SELECT st_WGS84ToGCJ02(lng, lat)
```

```
FROM <tableName|viewName>
```

**1-N Analysis Operations.** This type of operations converts each row of a view to multiple records. The UDF mechanism of Spark SQL is not supported for this case. We implement our own executors for 1-N analysis operations. For example, we provide variety of trajectory processing operations [33], e.g., `st_trajNoiseFilter`, `st_trajSegmentation`, `st_trajStayPoint`, and `st_trajMapMatching`. An example of trajectory noise filtering is shown as follows.

```
SELECT st_trajNoiseFilter(item)
FROM <tableName|viewName>
```

where `item` is the implicit field of a trajectory plugin table.

**N-M Analysis Operations.** This type of operations converts multiple rows of a view to one or more records. We implement our new executors for this type of operations, as Spark SQL does not support it. For example, JUST provides a spatial clustering method DBSCAN [35] for users:

```
SELECT st_DBSCAN(geom, minPts, radius)
FROM <tableName|viewName>
```

where $minPts$ and $radius$ are the parameters of DBSCAN.

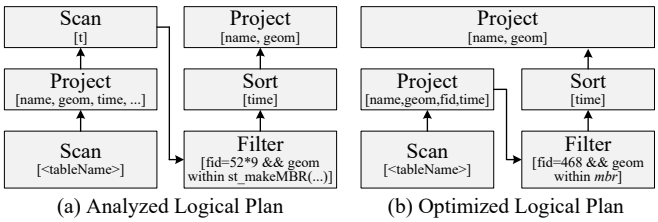
## VI. SQL Layer

One of the main goals of JUST is to make the system easy to use. We realize a complete SQL engine based on ANTLR [36], a powerful parser generator for structured text or binary files. All operations in JUST can be done using a standard SQL-like query language, i.e, JustQL, which greatly reduces the learning curves of users. For example, if a user issues the following statement, our SQL engine mainly performs three tasks: 1) SQL Parse, 2) SQL Optimize, and 3) SQL Execute.

```
SELECT name, geom
FROM (
  SELECT * FROM <tableName>
) t
WHERE fid=52*9 AND geom WITHIN
  st_makeMBR(lng_min, lat_min, lng_max, lat_max)
ORDER BY time
```

**SQL Parse.** JUST first converts the SQL statement into a syntax tree using an ANTLR-based parser. After that, JUST retrieves schema information of the input tables from the meta table. Using this information, JUST verifies field names, expands `select *` and checks data types of the syntax tree. Finally, we can get an analyzed logical plan. The analyzed logical plan of the above SQL statement is shown as Figure 8a, where each node represents a logical operation, and the children of a node are its inputs.

**SQL Optimize.** The SQL optimizer transforms the logical plan into a better one, by leveraging the following rules:




(a) Analyzed Logical Plan (b) Optimized Logical Plan

Fig. 8. Optimization of Logical Plan.

(1) *Calculate constant expressions.* For constant expressions, JUST will calculate them first, and replace them with their values, which avoids calculating expressions for many times. For example, we can replace `fid = 52 * 9 && geom within st_makeMBR(`$lng_{min}, lat_{min}, lng_{max}, lat_{max}$`)` with `fid = 468 && geo within` $mbr$, where $mbr$ is calculated by the user-given parameters.

(2) *Push down selections.* JUST pushes filtering predicates, e.g., spatio-temporal range predicates or equality predicate, closer to the table scan operators, which improves filtering efficiency greatly. For example, JUST pushes `fid = 468 && geo within` $mbr$ to the scan of `<tableName>`.

(3) *Push down projections.* JUST pushes field selections to the table scan operators as closer as possible. Besides, JUST intelligently identifies the necessary fields for filtering, ordering, grouping, and projection, and prunes the unnecessary fields to minimize the amount of data transferred between operators. In the given example, JUST only retrieves the fields of `name, geo, time,` and `fid` from the underlying table.

After optimized through multiple algebra rules, the logical plan of Figure 8a is transformed into Figure 8b.

**SQL Execute.** In this step, JUST translates the optimized logical plan into GeoMesa operations or Spark SQL operations. Specifically, the predicates of spatial or spatio-temporal filtering and field value filtering, as well as projections, are performed by the underlying GeoMesa. The intermediate result forms a Spark `DataFrame`, with which complex query predicates, like aggregate operations, `JOIN` operations, or analysis operations, are performed by Spark SQL. Integrating Spark SQL into our SQL layer allows us realize complex operations and analysis with less efforts.

## VII. Service Layer and Application Layer

In this section, we first briefly review the implementation of context & user management in the service layer, then exhibit two applications that are developed based on JUST in JD.

### A. Context & User Management

GeoMesa and Spark themselves do not support multiple thread-level users. To run a spatio-temporal operation, Spark will create a single Spark session, which is time-consuming. Besides, GeoMesa does not distinguish tables of different users. In service layer, JUST maintains a Spark context shared by all users based on Spark Job Server [37], thus can eliminate the cost of Spark session construction. Moreover, we build a namespace for each user, i.e., for the tables or views of a user, we add a unique prefix to their names, which is transparent to users. As a result, JUST supports multiple users simultaneously, while these users do not affect each other.

### B. SDKs & Applications

JUST provides multiple programming language SDKs, i.e., Java SDK and Python SDK. These SDKs communicate with JUST through HTTP protocol, where JUST acts as a PaaS (Platform as a Service). Many urban applications have been developed based on the JUST SDKs in JD.


(a) Urban Block Indicator System (b) Map Recovery System

Fig. 9. Two Applications of JUST in JD.

**Urban Block Indicator System.** This system is built to give a general overview of any urban area (a.k.a. address portrait). It first partitions the spatial space into about $150m \times 150m$ grids (where the GeoHash code[2] has a length of 7), then calculates multiple indicators of each grid, e.g., purchasing power, air quality, traffic flow, etc. The indicators of fine-grained urban areas are very helpful for billboard placement, location selection, and urban planning. However, there are about $2,400,000km^2$ urban areas in China[3]. If there are only ten records in each grid each day, there are over ten million records per day, which can be hardly supported by traditional spatial database. To this end, Urban block indicator system adopts JUST as the underlying storage, using the XZ2T indexing strategy. Users can search the indicators of any area using a spatio-temporal range query, as shown in Figure 9a.

**Map Recovery System.** This system [1] utilizes the trajectories of couriers in JD Logistics to recover the road networks of living areas, which could be missing in commercial maps. A complete map is essential for path planning and package dispatching. The GPS logs of over 60,000 couriers are loaded into JUST in batches each day. With the help of JUST to manage and process trajectories, the map recovery system can repair the road networks in living areas, and infer the speed and traffic mode (e.g., riding or walking) of each road segment with much fewer efforts, as shown in Figure 9b.

## VIII. Evaluations

In this section, we first depict the datasets and experimental settings, then present the evaluation results.

### A. Datasets & Settings

**Datasets.** To evaluate the performance of JUST, we use two real spatio-temporal datasets and one synthetic dataset: 1) **Traj**, which contains the trajectories of 48,813 lorries in JD Logistics; 2) **Order**, which includes the purchase orders from JD Mall. Each order is associated with an order time $t$, and a biased delivery address $point$ for the purpose of privacy protection; and 3) **Synthetic**, which is generated by copying & sampling the Traj dataset up to 1T to test the scalability of JUST. The statistics of the datasets are shown in Table II.

**Settings.** We test the efficiency of spatial range query, spatio-temporal range query, and $k$-NN query of JUST. Table III shows the storage settings. We set the time period of Z2T and XZ2T as a day, and compress the `GPSList` field of the Traj dataset with GZip, as it could have hundreds of GPS points in a trajectory. Table IV summarizes the query parameters, where the default values are in bold. Table V gives the softwares and their versions used in our experiments. To eliminate the HBase cache[4], we randomly select 100 different query parameters, perform each query only once, and take the median response time of all queries as the final results. All experiments are conducted on a cluster of 5 nodes, with each node equipped with CentOS 7.4, 8-core CPU, 32GB RAM, and 1T disk.

[2]https://en.wikipedia.org/wiki/Geohash

[3]https://dwz.cn/zowUkwDy

TABLE II
Statistics of Datasets

| Attributes | Traj | Order | Synthetic |
|---|---|---|---|
| # Points | 886,593,200 | 71,007,530 | 8,865,932,000 |
| # Records | 314,086 | 71,007,530 | 3,140,860 |
| Raw Size | 136GB | 10GB | 1360GB |
| Time Span | 2014/03/01 - 2014/03/31 | 2018/10/01 - 2018/11/30 | 2014/03/01 - 2014/12/31 |

TABLE III
Storage Settings

| Datasets | Indexes | Data Model |
|---|---|---|
| Traj | XZ2 on $MBR$, XZ2T on $MBR$ and $Time_{start}$ | Plugin Table |
| Order | Z2 on $point$, Z2T on $point$ and $t$ | Common Table |
| Synthetic | XZ2 on $MBR$, XZ2T on $MBR$ and $Time_{start}$ | Plugin Table |

TABLE IV
Query Settings

| Parameters | Settings |
|---|---|
| Data Size (%) | 20, 40, 60, 80, **100** |
| Time Window | 1h, 6h, **1d**, 1w, 1m |
| Spatial Window ($km^2$) | $1 \times 1$, $2 \times 2$, $\mathbf{3 \times 3}$, $4 \times 4$, $5 \times 5$ |
| $k$ | 50, 100, **150**, 200, 250 |

TABLE V
Softwares in the Experiments

| Software | Version | Software | Version | Software | Version |
|---|---|---|---|---|---|
| Hadoop | 2.7.6 | GeoMesa | 2.3.0 | JDK | 1.8 |
| Spark | 2.3.3 | HBase | 1.4.9 | Scala | 2.11 |

**Baselines.** We compare JUST with six state-of-the-art spatial/spatio-temporal systems[5], where their supported queries are shown as Table VI. To verify the effectiveness of compression mechanism and Z2T/XZ2T indexing strategies proposed by this paper, we compare two variants:

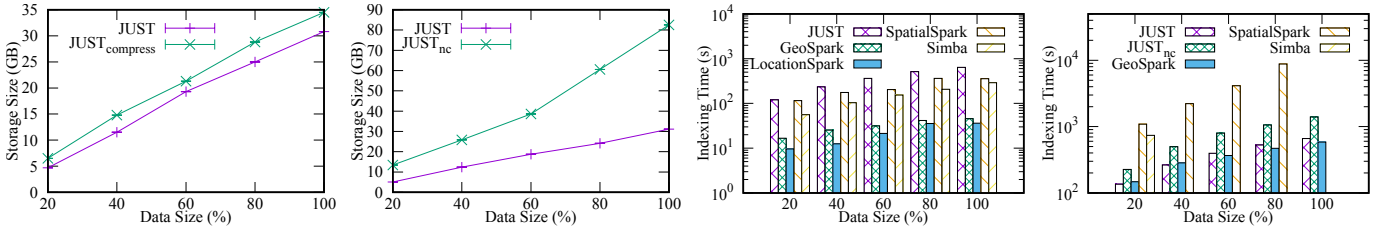
- **JUST**$_{nc}$, which does not use the compression method for the field of `GPSList` in the Traj dataset;
- **JUST**$_d$**/JUST**$_y$**/JUST**$_c$, which use Z3/XZ3 indexing strategies with the time period as a day, a year, or a century, respectively (we extend a century of time period as GeoMesa does not support it).

### B. Performance of Storage & Indexing

**Storage Performance.** Figure 10a and Figure 10b show the storage costs of JUST with different raw data sizes. Note here that the storage costs include both of the index structures

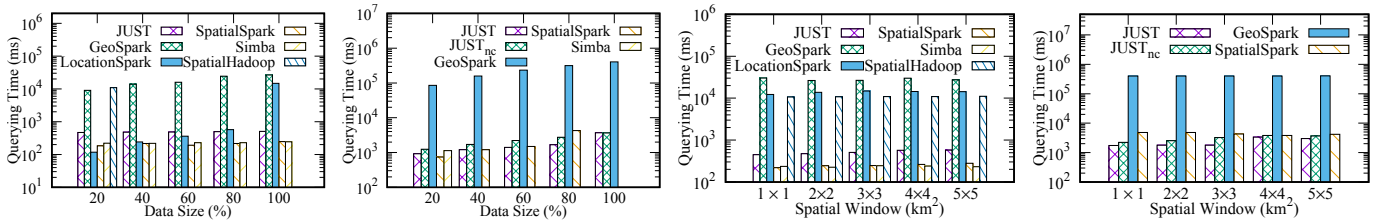
[4]HBase will cache results in memory to expedite the same queries.

[5]We test most available systems. [19] runs failed in our experiments, though we have resorted to its authors. [14] runs out of memory even for 20% of the Order dataset. [18] and [7] are out of date.




(a) Storage Size (Order). (b) Storage Size (Traj). (c) Indexing Time (Order). (d) Indexing Time (Traj).

Fig. 10. Performance of Storage & Indexing.




(a) Data Size (Order). (b) Data Size (Traj). (c) Spatial Window (Order). (d) Spatial Window (Traj).

Fig. 11. Performance of Spatial Range Query (for SpatialSpark, the Data Size of Traj is 80%).

TABLE VI
COMPARING SYSTEMS AND THEIR SUPPORTED QUERIES

| Systems | S | ST | $k$-NN | Systems | S | ST | $k$-NN |
|---|---|---|---|---|---|---|---|
| ST-Hadoop [4] | √ | √ | √ | SpatialHadoop [5, 6] | √ | × | √ |
| Simba [15] | √ | × | √ | GeoSpark [9, 10] | √ | × | √ |
| SpatialSpark [11] | √ | × | × | LocationSpark [12] | √ | × | √ |

(i.e., the keys) and the data (i.e., the values) itself. As we can see, with more raw data, it needs more storage space for both datasets. For the Traj dataset, JUST takes much less space than JUST$_{nc}$, which verifies the effectiveness of our proposed compression mechanism. It is interesting to see that storing 136GB trajectories only needs about 30GB storage space. However, for the Order dataset, if we compress its fields (i.e., the JUST$_{compress}$ line in Figure 10a), it surprisingly takes a little more space than that we do not. Because for small data with only several bytes, its compressed version is usually bigger than itself. The lesson we can learn is that, our compression mechanism is only suitable for big fields (e.g., the `GPSList` field of the Traj dataset). To this end, in the following experiments, we do not adopt the compression techniques for the Order dataset.

**Indexing Performance.** Figure 10c and Figure 10d illustrate the indexing performance of different systems. Note that for JUST and its variants, the indexing time includes both the cost of indexing and storing. As a result, for the Order dataset, JUST takes more time than the Spark-based systems. However, for the Traj data, JUST takes much less time than SpatialSpark and Simba. The reason could be that SpatialSpark and Simba build huge indexes in memory. When the memory is about to run out, they incur disk thrashing. Indeed, Simba throws an out of memory exception when the data size of Traj is 40%, and SpatialSpark fails when the data size of Traj is 100%. The results of Hadoop-based systems, i.e., SpatialHadoop and ST-Hadoop, are not presented, because they takes more than three hours even for 40% of Order data (we find that they spend too much time to serialize and store the indexes). Moreover, JUST takes less time than JUST$_{nc}$, because the compressed Traj dataset has a smaller size, which incurs less disk IOs.

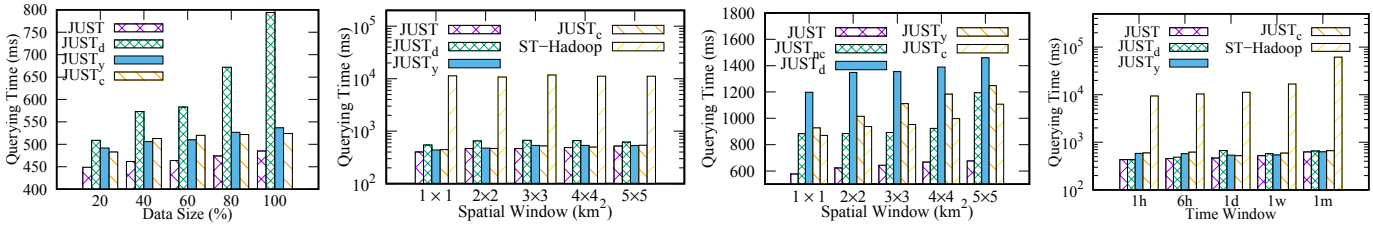
### C. Performance of Spatial Range Query

**Different Data Sizes.** As shown in Figure 11a and Figure 11b, with a bigger dataset, all systems need more time to answer a spatial range query, as more data is scanned and returned. We do not present ST-Hadoop here because ST-Hadoop is based on SpatialHadoop, thus shows the same results with SpatialHadoop. JUST is much faster than SpatialHadoop, and can achieve similar query performance to the Spark-based systems, as JUST encodes the spatial information into the keys of the NoSQL data store, which allows us to locate the qualified records directly. However, JUST is much more scalable than the Spark-based systems. For example, Simba runs out of memory when the data size of Traj is over 20%, and LocationSpark has the same problem even for 20% of Traj data. JUST is faster than JUST$_{nc}$, although JUST would decompress the Traj data, which indicates that the disk IOs play major role when answering a spatial range query.

**Different Spatial Windows.** Figure 11c and Figure 11d show that, with a bigger spatial window, most systems spend more time for spatial range query. Because with a bigger spatial window, more data will be returned, which triggers more disk IOs. Simba and SpatialSpark seem faster than JUST for the Order data, as they store all data in memory. However, for the Traj data, JUST runs faster than SpatialSpark, although SpatialSpark stores only 80% of the data. The reason could be SpatialSpark builds huge indexes in memory. For each spatial range query, it needs to scan the huge indexes, which is costly.

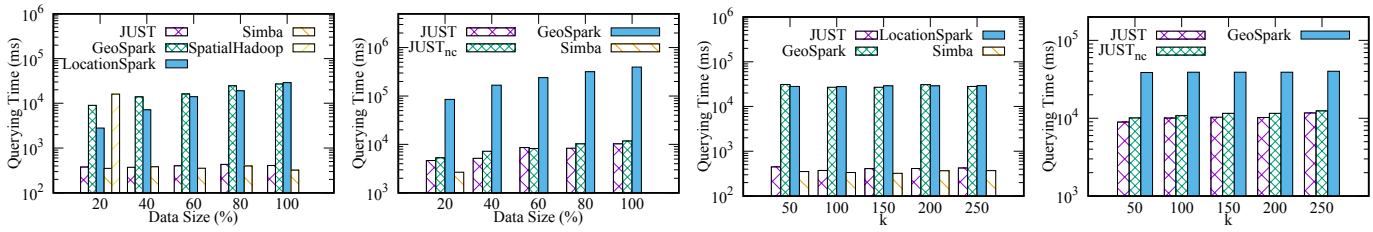
### D. Performance of Spatio-Temporal Query

**Different Data Sizes.** Figure 12a shows the query efficiency varies with different data sizes of the Order dataset. The results of the Traj dataset are not presented as it shows similar




(a) Data Size (Order). (b) Spatial Window (Order). (c) Spatial Window (Traj). (d) Time Window (Order).

Fig. 12. Performance of Spatio-Temporal Range Query (for ST-Hadoop, the Data Size of Order is 20%).




(a) Data Size (Order). (b) Data Size (Traj). (c) $k$ Value (Order). (d) $k$ Value (Traj).

Fig. 13. Performance of $k$-NN Query.

results with that of Order data, as well as the page limitation. The results of ST-Hadoop are not shown here, because ST-Hadoop runs too slowly even for 20% of the Order data (the querying time is as high as 11,330ms). There are three observations: 1) with more data, the time of spatio-temporal query gets larger for all methods, because it scans and returns more data; 2) JUST is faster than other methods, because the Z2T indexing strategy proposed by this paper avoids the invalidation of spatial filtering caused by the different scales between spatial dimension and time dimension; 3) the spatio-temporal query performance with a bigger time period is better than that with a smaller one, which demonstrates the correctness of our analysis.

**Different Spatial Windows.** Figure 12b and Figure 12c present the query time with different spatial windows for the two real datasets. It is observed that: 1) with a bigger spatial window, all methods need more time to answer a spatio-temporal range query, as we retrieve and return more data; 2) although JUST and its variants are based on the 100% of Order dataset, they achieve an order of magnitude improvement with regard to ST-Hadoop (with only 20% of Order data), because it is expensive for ST-Hadoop to start a MapReduce job; 3) for the Traj dataset, JUST is faster than JUST$_{nc}$, as its storage size is much smaller than that of JUST$_{nc}$, which reduces the disk IOs; 4) JUST is faster than JUST$_d$, JUST$_y$, and JUST$_c$, which is due to the proposed XZ2T indexing strategy; 5) the cost of querying the Traj dataset is a little more than that of Order dataset, because the size of Traj is bigger than that of Order. It would scan and return more data for Traj dataset.

**Different Time Windows.** Figure 12d compares the spatio-temporal range query time with different time windows for Order data. The result for Traj dataset is not presented, as it is similar with that of Order data. With a bigger time window, all methods need more time, because a bigger time window means more qualified records. ST-Hadoop is much slower than JUST and its variants, for the bottleneck of disk IOs.

### *E. Performance of $k$-NN Query*

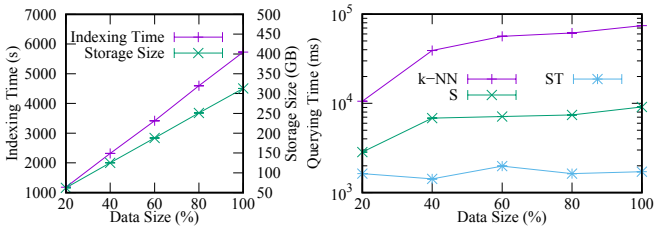
**Different Data Sizes.** As shown in Figure 13a and Figure 13b, with a bigger data size, it takes more time to answer a $k$-NN query, because in each expansion process of $k$-NN query, we trigger a spatial range query, which scans more records. For the smaller Order data, JUST shows a competitive performance comparing Simba, but for the bigger Traj data, Simba runs an out of memory exception when the data size is 40%.

**Different $k$ Values.** Figure 13c and Figure 13d depict the performance of $k$-NN query varies with different $k$ values. As we can see, with a bigger $k$, all systems need slightly more time, because we should expand more times and trigger more spatial range queries to get the most $k$ nearest records. JUST is much more efficient than GeoSpark and LocationSpark, as JUST locates the qualified records directly, and triggers `SCAN` operations in parallel. JUST is a little better than JUST$_{nc}$, owing to the proposed compression mechanism.

### *F. Scalability of JUST*

To test the scalability of JUST, we conduct a set of experiments using the Synthetic dataset, whose size is over 1T. As shown in Figure 14a, both indexing time and storage size increase linearly with an increasing data size from 20% to 100%, as we need to process more data. JUST only takes about 1.5 hours and 313GB disk space to index over 1T data, which is owed to the proposed compression mechanism.

Figure 14b shows that, both $k$-NN query and spatial range query take more time with a bigger dataset, because there is more data scanned and returned. However, the efficiency of spatio-temporal query has nothing to do with the data size, as it scans and returns the same results no matter how big the dataset is. For a spatio-temporal query, JUST can locate the qualified time periods directly. In each qualified time period, the amount of records is not affected by new added data.




(a) Data Size (Synthetic). (b) Data Size (Synthetic).

Fig. 14. Scalability of JUST.

## IX. Conclusion and Future Works

This paper presents JUST, a distributed spatio-temporal data management system which is efficient, scalable, update-enabled and easy to use. JUST employs two novel indexes, i.e., Z2T and XZ2T, and introduces a compression mechanism to improve spatio-temporal query performance. Besides, a SQL-like query language, i.e, JustQL, is designed and various spatio-temporal operations are preset to improve the usage convenience. Experiments based on two real datasets and one synthetic dataset verify the powerful scalability and efficiency of JUST. Currently, JUST is deployed as a PaaS in JD, supporting various urban spatio-temporal applications.

There are many important avenues to perfect JUST: 1) We are working towards supporting more data sources, especially the streaming data sources such as Kafka. 2) We will add more spatio-temporal data types as plugin tables, and more analysis operations to JUST, which can make JUST more convenient to manage spatio-temporal data. 3) JUST currently has a naive rule-based optimizer with a small number of simple rules. We plan to build a cost-based optimizer to make JUST more efficient. 4) At present, JUST is more suitable for OLAP. For each query request, JUST sets up a Spark job, which is costly for scheduling cluster resources. If JUST intelligently selects a single-machine version for a small data request, we can achieve a goal of combining OLAP and OLTP together.

## Acknowledgment

We would like to thank our users and developers for their contributions, with special thanks to Yuan Sui, Wei Wu, Jian Hu, Haowen Zhu, Meng Sheng, Tao Gong, Jinghui Liu, Yannan Liu, Peng Wang, Qin Chen, and Jiuqun He. This work was supported by the National Key R&D Program of China (2019YFB2101805).

## References

[1] S. Ruan, C. Long, J. Bao, C. Li, Z. Yu, R. Li, Y. Liang, T. He, and Y. Zheng, "Learning to generate maps from trajectories," in *AAAI*, 2020.
[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*. USENIX Association, 2012, pp. 2–2.
[4] L. Alarabi, M. F. Mokbel, and M. Musleh, "St-hadoop: A mapreduce framework for spatio-temporal data," *GeoInformatica*, vol. 22, no. 4, pp. 785–813, 2018.
[5] A. Eldawy and M. F. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data," in *ICDE*. IEEE, 2015, pp. 1352–1363.
[6] A. Eldawy, L. Alarabi, and M. F. Mokbel, "Spatial partitioning techniques in spatialhadoop," *VLDB*, vol. 8, no. 12, pp. 1602–1605, 2015.
[7] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz, "Hadoop gis: a high performance spatial data warehousing system over mapreduce," *VLDB*, vol. 6, no. 11, pp. 1009–1020, 2013.
[8] R. T. Whitman, M. B. Park, S. M. Ambrose, and E. G. Hoel, "Spatial indexing and analytics on hadoop," in *SIGSPATIAL*. ACM, 2014, pp. 73–82.
[9] J. Yu, J. Wu, and M. Sarwat, "A demonstration of geospark: A cluster computing framework for processing big spatial data," in *ICDE*. IEEE, 2016, pp. 1410–1413.
[10] J. Yu and J. Wu, "Geospark: A cluster computing framework for processing large-scale spatial data," in *SIGSPATIAL*. ACM, 2015, p. 70.
[11] S. You, J. Zhang, and L. Gruenwald, "Large-scale spatial join query processing in cloud," in *ICDE Workshops*. IEEE, 2015, pp. 34–41.
[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," *VLDB*, vol. 9, no. 13, pp. 1565–1568, 2016.
[13] F. Baig, H. Vo, T. Kurc, J. Saltz, and F. Wang, "Sparkgis: Resource aware efficient in-memory spatial query processing," in *SIGSPATIAL*. ACM, 2017, p. 28.
[14] S. Hagedorn, P. Gotze, and K.-U. Sattler, "The stark framework for spatio-temporal data analytics on spark," *BTW*, 2017.
[15] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *SIGMOD*. ACM, 2016, pp. 1071–1085.
[16] "Hbase," https://hbase.apache.org/, 2019.
[17] "Geomesa," https://www.geomesa.org/, 2019.
[18] S. Nishimura, S. Das, D. Agrawal, and A. El Abbadi, "Md-hbase: A scalable multi-dimensional data infrastructure for location aware services," in *MDM*, vol. 1. IEEE, 2011, pp. 7–16.
[19] J. K. Nidzwetzki and R. H. Güting, "Bboxdb-a scalable data store for multi-dimensional big data," in *CIKM*. ACM, 2018, pp. 1867–1870.
[20] N. Du, J. Zhan, M. Zhao, D. Xiao, and Y. Xie, "Spatio-temporal data index model of moving objects on fixed networks using hbase," in *CICT*. IEEE, 2015, pp. 247–251.
[21] Y.-T. Hsu, Y.-C. Pan, L.-Y. Wei, W.-C. Peng, and W.-C. Lee, "Key formulation schemes for spatial index in cloud data managements," in *MDM*. IEEE, 2012, pp. 21–26.
[22] X. Tang, B. Han, and H. Chen, "A hybrid index for multi-dimensional query in hbase," in *CCIS*. IEEE, 2016, pp. 332–336.
[23] "Academic homepage of just," http://just.urban-computing.com/, 2019.
[24] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.
[25] H. S, *Space-filling curves*. Springer Science & Business Media, 2012.
[26] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *VLDB*, vol. 2, no. 2, pp. 1626–1629, 2009.
[27] J. A. Orenstein and T. H. Merrett, "A class of data structures for associative searching," in *SIGMOD*. ACM, 1984, pp. 181–190.
[28] C. B, G. K, and H.-P. K, "Xz-ordering: A space-filling curve for objects with spatial extension," in *SSD*. Springer, 1999, pp. 75–90.
[29] J. N. Hughes, A. Annex, and et al., "Geomesa: a distributed architecture for spatio-temporal fusion," in *Geospatial Informatics, Fusion, and Motion Video Analytics V*, vol. 9473. SPIE, 2015, p. 94730F.
[30] Y. Zheng, "Trajectory data mining: an overview," *TIST*, vol. 6, no. 3, p. 29, 2015.
[31] R. Li, H. He, R. Wang, S. Ruan, Y. Sui, J. Bao, and Y. Zheng, "Trajmesa: A distributed nosql storage engine for big trajectory data," in *ICDE*. IEEE, 2020.
[32] R. Li, S. Ruan, J. Bao, and Y. Zheng, "A cloud-based trajectory data management system," in *SIGSPATIAL*. ACM, 2017, p. 96.
[33] S. Ruan, R. Li, J. Bao, T. He, and Y. Zheng, "Cloudtp: A cloud-based flexible trajectory preprocessing framework," in *ICDE*. IEEE, 2018, pp. 1601–1604.
[34] J. Zhang, Y. Zheng, D. Qi, R. Li, X. Yi, and T. Li, "Predicting citywide crowd flows using deep spatio-temporal residual networks," *Artificial Intelligence*, vol. 259, pp. 147–166, 2018.
[35] M. Ester, H.-P. Kriegel, J. Sander, X. Xu *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise." in *KDD*, vol. 96, no. 34, 1996, pp. 226–231.
[36] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.
[37] "Spark job server," https://github.com/spark-jobserver/spark-jobserver, 2019.