International Journal of Computer Engineering and Data Science (ISSN: 2737-8543)
Volume 2– Issue 1, January - March 2022

Big vector spatial data storage, indexing and query processing using the NoSQL Accumulo data store

El Hassane Nassif[1], Hajji Hicham[2], Reda Yaagoubi[3] , Hassan Badir[4]

[1,2,3]Ecole ESGIT, IAV H2, Rabat, Morocco
[4]System and Data Engineering Team, Abdelmalek Essaadi University, Tangier, Morocco
E-mail : nassif.hassane@gmail.com, h.hajji@iav.ac.ma, r.yaagoubi@iav.ac.ma, hbadir@uae.ac.ma

Article history
Received Nov 01, 2021
Revised Jan 10, 2022
Accepted Feb 05, 2022
Published April 01, 2022

ABSTRACT

Recent advent of connected objects, omnipresence of social networks, success of the smartphone market, mobile telecommunications technology generalization, all these factors contribute to generating more and more spatial data with big data characteristics. Relational databases are widely used to store, retrieve and manage spatial data but they reach their limits in the presence of big data constraints. NoSQL Key/Value stores are a new approach that has emerged to support data big data optimally. In this research work, we are interested in the Accumulo database. A NoSQL database that uses a key value store model. Through this paper, we propose a spatial version of Accumulo to handle big vector spatial data. First, we analyse Accumulo's storage and indexing logic. Next, we design a storage and indexing model specific to vector-type spatial data. Finally, we propose an application of this new model through an example of spatial queries.

Keywords: ***Accumulo, big data, spatial, vector, index***

I. INTRODUCTION

At the time of writing this paper, the term "Big Data" is almost ubiquitous in articles and reports published by computer practitioners and researchers. This is the reason why there are several definitions of "Big Data". The most common one comes from Gartner [1] which suggests that big data is characterized by very high volumes, varieties and velocities requiring innovative forms of processing for better understandings, for better decision-making and for better process automation.

In the early years of the digital revolution, primarily computers produced data. Nowadays, the advent of connected objects, the omnipresence of social networks, the success of the smartphones market, the widespread deployment of mobile telecommunications technologies are causing an explosion of data volumes generated by these activities [2].

The spatial field is also concerned with the advent of Big Data insofar as part of this data are spatial resulting from neogeography [3] which is a new movement that has experienced a real boom these recent years thanks to the success of participatory cartographic data creation platforms. OpenStreetMaps/GoogleMap and Waez are a significant example of this movement by allowing any person connected to the internet to produce, disseminate and consume spatial data from any area or region of the globe. In addition, humankind is equipping itself more and more with connected objects, smartphones, drones, vehicles and uses more and more internet georeferenced content sharing services (text, image or video) which generates more and more georeferenced data.

This considerable success of neogeography and georeferenced services [4] will lead to an explosion in the production of spatial data and consequently it will be difficult to manage this kind of data in a conventional manner through geographic information systems and relational databases. Indeed, even if these systems have the capacity to assemble, store and manipulate geographic information. However, they have various limits when spatial data gradually becomes big data. In such a context and given the absence of big data solutions adapted to spatial data, we propose through this paper, we propose a spatial version of Accumulo to handle big vector spatial data. First, we analyse Accumulo's storage and indexing logic. Next, we design a storage and indexing model specific to vector-type spatial data. Finally, we propose an application of this new model through an example of spatial queries.

II. CONCEPTS AND PREVIOUS WORKS

A. Relational Database Limitations

Relational databases have been widely used to store, track and manage data since the 1970s [5] [6]. However, even if it is

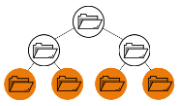
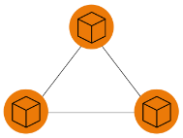
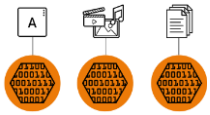
the foundation of any classical information system, it reaches its limits due to big data constraints mainly the 3 Vs constraints: Volume, Velocity, and Variety.

- Volume: In the presence of very large volumes of data (thousands of Terabytes), the relational model reaches its limits.
- Variety: In the relational model, the data schema (i.e. fields, tables, integrity constraints and relationships) is predefined at the time of setting up the use case. Any subsequent changes can be complex, especially if the structure of the data varies all the time. Frequent changes to the structure of a relational database can lead to errors and data loss if changes are frequent.
- Velocity: New internet trends have created the need to implement "Real Time" applications that handle massive data. Relational databases are not suitable for this type of application, because each "Real Time" operation has its own time constraints, impossible to meet in a context of massive data while respecting the integrity constraints of relational databases.

B. Distributed Storage

Data storage is an information system process used to persist, organize and share data. It helps meet application demands when manipulated data cannot fit fully into RAM[1]. Initially and through the development of broadband computing networks, data storage was centralized on remote servers so that all applications could access them at the same time [7]. Then, because of data volume growth and the advent of the big data era, this centralized model evolved into a storage model distributed over several machines. Three storage modes exists and are used by practitioners [8] to organize and to present data in different ways (TABLE 1)

TABLE I. STORAGE MODES [REDHAT INC, 2018]

File mode storage	
	Organizes and represents data in the form of a hierarchy of files contained in folders. Each single file has an extension type that's determined by the application used to create the file.
Block mode	
	Groups data into separate volumes and equal sizes called Blocks. Each Block has its own identifier and is managed by a storage system, which allows you to select the most suitable locations.
Object mode	
	In this storage mode, data is partitioned into small units called objects and stored in a single, stand-alone repository containing data and it's describing metadata.

[1] Random Access Memory

In 2003, Google published a research paper describing the architecture of a new distributed file storage system called GFS dedicated to big data use cases. GFS was developed in C-language [9]. The publication of this work allowed Yahoo's lab to produce a new distributed file system called Hadoop Distributed File System [10] developed with java. Previously, HDFS was part of the Nutch research project that aimed to build an open-source search engine. HDFS is inspired by GFS and has almost the same characteristics; it is based on the same principles that rely on two separate services: NameNode and DataNode for HDFS and master and ChunkServer for GFS.

- Name node/ master services: Run on dedicated nodes of the cluster. Their responsibility is to store the metadata of the file system in order to keep track of all the data as well as all the nodes that compose the distributed storage cluster.
- Data node/Chunk services, run on slave nodes responsible for storing all the data ingested by the cluster.

HDFS and GFS makes easier to process large data sets distributed on clusters of servers while assuring Data Reliability; High availability (Fault tolerance); High throughput and scalability.

C. NoSQL paradigm

The advent of Big Data and consequently the increased constraints of the 3 Vs have contributed significantly to the emergence of new approaches to data management, including NoSQL databases. The term NoSQL is the acronym for Not Only Sql, it was first used in 2009 at a conference on distributed databases [11] [12]. Beyond the term, NoSQL means a new approach to databases that can effectively manage big data by ensuring high service availability and scaling ability. NoSQL databases follow a complete break with the relational model, as it abandons certain ACID [13] properties in favour of data distribution and in favour of horizontal scaling and performance [14] [15]. In recent years, primarily practitioners and web companies to adapt to their specific performance and maintenance requirements have developed varieties of NoSQL databases. Some of these databases have taken ideas from Amazon's Dynamo [16] or Google BigTable [17]. Others have pursued very different approaches such as Neo4j or Hypergraph DB. Due to the variety of these approaches and the overlapping needs and functionality, it could be difficult to obtain and maintain an overview of the NoSQL database scene. Therefore, there were

different attempts to categorize them, the most common classification was proposed by [18] who divided the NoSQL databases into four families: column-oriented, key-value-oriented, graph-oriented, and document-oriented. Each of these four database families has its own data model and data storage service.

D. Apache Accumulo

Accumulo is a column family NoSQL database that uses the same Google BigTable design. Nevertheless, Accumulo has several features that distinguish it from other existing BigTable implementations. One interesting feature is the iterator framework that iterate over key-value pairs allowing developers to make stored big data exploitable trough distributed search and custom aggregations and summarization. The second main feature of Accumulo is the capability to enable fine-grained data access control.
Accumulo database can store data with various types, mainly numerical and textual, in a single table. A table can contain millions of columns and can support a new column without impacting already established applications.
Each Accumulo table contains key value pairs that we can represent in the form of bytes tables with the structure above (timestamps field is an exception as its type is "Long")

TABLE II. ACCUMULO NOSQL COLUMN

Key					**Value**
RowID	***Column***			***TimeStamp***	
	Family	Name	Visibility		

Acumulo Tables consist of a set of lexicographic[2] sorted key-value pairs. Sort is done based on keys in an ascending order and based on Timestamps in a descending order so that later versions of the same key appear first in case of sequential scans. The combination of: RowID, Column Family, Column Name and TimeStamp gives a unique value that matches a single record in the table.
Despite all of its advantages, Accumulo is not able to represent and manage spatial data whether they are vector or raster. In order to overcome this limitation, we extend Acumulo in order to support vector spatial data. Because of the multidimensional nature of vector spatial data, the challenge will be to find a way to index them using Accumulo RowId field.

E. Previous works

Some NoSQL database editors have continued to develop their products to cover the spatial domain. In the following we give a survey of some of the big available NoSQL databases that supports spatial data.

1) Redis:

Redis is a Key Value database that supports vector spatial data through a native geometry Point object that represents's a single location in space with an x-coordinate (longitude) and a y-coordinate (latitude). Just like standard data types, storage of spatial data is based on sorted datasets called GeoSet. Sort order is maintained by using a Hash function that transforms the two spatial dimensions into one value named score. It's a double precision number used for sorting and can represent any different decimal or integer value between -/+ 2^53. But when representing much larger numbers, elements will be added to the GeoSet with the same score. This problem was solved by implementing a lexicography technique that considers strings as binary data and compares the raw values of their bytes, byte after byte. Spatial data inserts and updates are done using the GeoAdd method, deletes are done by GeoRem method and searching through GeoPos or GeoHash methods. GeoPos return X, Y coordinates, and GeoHash return the linearized result using the hash function previously described. For spatial queries, Redis has another method called GeoRadius that allows the end user to query the database in order to render all the included points in a circle.

2) Elastic search:

Elastic search is a document oriented database that supports several types of vector spatial data through two spatial objects: geo_point (supports latitude and longitude pairs) and geo_shape (supports Point, MultiPoint, LineString, MultiLineString, Polygon with holes and MultiPolygon with holes). Spatial Elastic search implementation follows a specific indexing logic by encoding latitude and longitude using Geohash base32 and Bkd-Tree algorithms . Longitude/latitude coordinates default size is 16Bytes X 2. When indexed as Geohash, they will be base32 encoded strings. The default precision is ~2 cm base on 12 characters in the encoded hash string, each character in a GeoHash adds additional 5 bits to the precision. So the longer the hash string is, the more precise the location will be. Because of memory allocation, precision can be a bottleneck for Elasticsearch. Depending on the use case, the precision should be handled carefully by specifying how precise we need our point to be instead of the default full 12 levels of precision. Elastic search offers several spatial queries. Geo_Bounding_Box query finds documents with Geo-Points that fall into the specified rectangle. Geo_Distance query, finds documents with Geo-Points within the specified distance of a central point. Geo_Polygon query Find documents with Geo-Points within a specified polygon.

3) MongoDB:

Is another NoSQL document-oriented database, based on JSON documents. MongoDB supports vector spatial data and stores them as GeoJSON objects or as legacy coordinate pairs. The first option is used to calculate geometry over an Earth-like sphere. The second option is used to calculate distances on a Euclidean plane. MongoDB supports a set of standard vector spatial data types (Point, LineString, Polygon,

2 is a generalization of the alphabetical order for any element of a totally ordered set.

MultiPoint, MultiLineString, MultiPolygon, and GeometryCollection) and offers a subset of spatial operations (inclusion, intersection, and proximity). MongoDB also offers 2d vector indexes a 2dsphere index that supports only spherical queries (i.e. queries that interpret geometries on a spherical surface) and 2d index that supports flat queries (i.e. queries that interpret geometries on a flat surface)

4) Hbase:

Hbase is a column-oriented database that works in the same way as Accumulo, in fact its another implementation of Google BigTable design [17]. Hbase does not support spatial data but there is some research works that proposes a spatial extension. MD-HBase [24] offers an extension based on two layers. An indexing layer using standard K-d tree and the Quad tree index structures. And a storage layer that stores the items sorted by their key and range-partitions the key space. The keys correspond to the Z-value of the dimensions being indexed mainly: location and timestamp. MD-HBase proposes two spatial queries KNN and RangeQuery. A second extension of Hbase [25] proposes another spatial index structure using Geohash encoding based on a latitude and longitude geocode system, which can convert the latitude and longitude information into a one-dimensional encoding string. Based on this Geohash index, the author designed a KNN query and a rectangular range queries.

III. STORAGE LAYER

A. Spatial data storage model

Since Accumulo does not support spatial data, we propose in this work to extend its column-oriented data model in order to support vector spatial data. In this section, we present the logical layer of our extension. We also describe the system in terms of its conceptual organization: classes, interfaces and relations.

Figure 1 illustrates the class diagram used in our system, it's a static high-level structure that illustrates vector spatial objects with various geometries including point, line, polygon, multiline, multipoint, multi-polygon as well as their attributes, dependencies and relationships. This class model has the particularity to be universal insofar as it can be adapted to any other big table implementation such as Hbase or others and can take in charge several types of big spatial data sources.

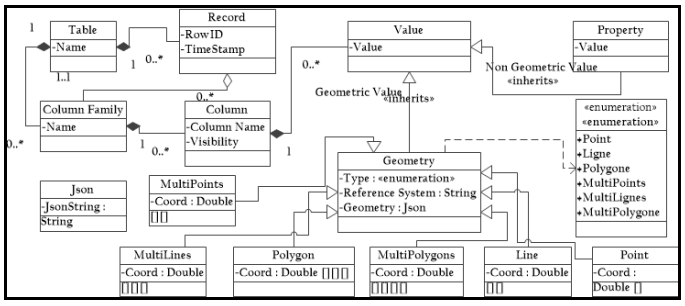


Figure 1. Accumulo spatial model class diagrama

The diagram shows two groups of objects, a first group representing the basic structure of Accumulo notably: "Table", "Record", "Column Family", "Column" and "Value". All these objects are linked by composition and aggregation association since the main purpose is to emphasize the dependency of the contained class on the life cycle of the container class and to refer to the "Record" class construction, which is the result of the "Column Family" class being aggregated or built as a collection. The second group represent spatial data through a central class named "Value" in the diagram. "Value" is a kind of super-class that defines the main features for both child classes: "Property" and "Geometry". These subclasses inherits "Value" class definition and modify it to their specific properties. The class "Property" is dedicated to non-geometric attributes and refers to the basic Accumulo cell field. The class "Geometry" is dedicated for spatial data with a main attribute "enumeration" covering all types (point, line, polygon, multiline, multipoint and multipolygon). TABLE III. gives an example of values for each type of geometry.

TABLE III. JSON GEOMETRY STRUCTURE

Geometry	Value
Point	"geometry": {"type": "Point","coordinates": [-5.44921875, 32.39851580247402]}
Polygon	"geometry":{ "type":"Polygon", "coordinates":[[[-5.44921875, 33.7243396617476],[-8.4375,32.39851580247402],[-8.96484375,29.075375179558346],[-5.9765625, 30.2970178833],[-3.1640625,31.50362930577303],[-5.44921875,33.7243396617476]]]}
Line	"geometry": {"type": "LineString", "coordinates": [[102.0, 0.0],[103.0, 1.0], [104.0, 0.0], [105.0, 1.0]}
Multi-Lines	"geometry": {"type": "MultiLineString", "coordinates": [[[170.0, 45.0], [180.0, 45.0]], [[-180.0, 45.0], [-170.0, 45.0]]] }
Multi-Polygons	"geometry": {"type": "MultiPolygon", "coordinates": [[[[180.0, 40.0], [180.0, 50.0], [170.0, 50.0], [170.0, 40.0], [180.0, 40.0]]], [[[-170.0, 40.0], [-180.0, 50.0], [-180.0, 40.0], [-170.0, 40.0]]]] }
Multi-Points	"geometry": { "type": "MultiPoint","coordinates": [[100.0, 0.0],[101.0, 1.0]] }

Note that this way of representing vector-type big spatial big data is valid, regardless of the data source. It is also possible to merge various big spatial data coming from multiple data sources.

B. Multi sources oriented spatial data storage

In the following, we will use our previous model (Figure 1.) to store spatial data coming from different big spatial data sources. We consider two examples, a first one related to Telecom, precisely a subscribers location tracking use case. The second example deals with vehicles pollution spatial monitoring. These two use cases deals with two completely different data sources and yet we will see at the end of this paper how we can reconcile them to be stored according to one and only one Accumulo spatial model using our precedent structure.

Concerning the Telecom use case, we consider that the data source at the entry of Accumulo is a localized voice call with the structure above: Origin phone number, destination phone, Longitude, Latitude, Call duration, Call date and Call Time. Then, we partition data in input to two different structures, a first Json part to represent subscriber's location using Point geometry type and a second part to represent the rest of data, in particular the origin and destination telephone numbers. This gives the structure displayed on the figure below:

Key					Value
RowID	*Column*			*TimeStamp*	
	Family	Name	Visibility		
Correspondence with the Accumulo model ⬇					
RowID	*Column Family*	*Column Name*	*Column Visibility*	*TimeStamp*	*Value*
Rowid1	Event	Origine Phone Number	Private	11905013	Tel a
Rowid1	Event	Destination Phone Number	Private	11905013	Tel b
Rowid1	Event	Geometry	Public	11905013	"geometry": { "type": "Point", "coordinates": [-3.123865, 23.3456]}
Rowid1	Event	Call duration in seconds	Public	11905014	34
Rowid1	Event	Date	Public	11905015	21/10/2020
Rowid1	Event	Hour	Public	11905018	15:01:09

Figure 2. Accumulo table structure for storing calls with their location

Note: "visibility" column on Figure 2. is a feature available in Accumulo that helps developers to meet strict data access requirements to comply with security requirements and allows implementing a wide variety of data access scenarios. We introduce this feature in the present paper for information only and we will not develop it any more than that.

The second use case concerns vehicles pollution monitoring using intelligent sensors for measuring pollution rates and generating various localized measures based on density of some chemical components known to contribute to air pollution. We will use the same Accumulo structure already defined for the Telecom use case using the same data model shown on the class diagram (see Figure 1.). Since the two data sets concerns different domains: Telecom subscribers location tracking versus vehicles pollution monitoring, we have added a new column family to the second set of data (Figure 3.).

RowID	*Column Family*	*Column Name*	*Column Visibility*	*TimeStamp*	*Value*
Rowid1	Event	Origine Phone Number	Private	11905013	Tel a
Rowid1	Event	Destination Phone Number	Private	11905013	Tel b
Rowid1	Event	Geometry	Public	11905013	"geometry": { "type": "Point", "coordinates": [-3.123865, 23.3456]}
Rowid1	Event	Call duration in seconds	Public	11905014	34
Rowid1	Event	Date	Public	11905015	21/10/2020
Rowid1	Event	Hour	Public	11905018	15:01:09
Rowid2	Pollution	ozone	Public	11908002	100
Rowid2	Pollution	Particles	Public	11908003	54
Rowid2	Pollution	Carbon Monoxide	Public	11908003	46
Rowid2	Pollution	Sulfure Dioxyde	Public	11908005	80
Rowid2	Pollution	Nitrogene Dioxide	Public	11908006	67
Rowid2	Pollution	Géometrie	Public	11908007	"geometry":{"type":"Point", "coordinates": [[-5.44921875, 20.3456]}

Figure 3. Multi sources example

Indeed, in addition to logical data separation, column families design has a significant impact on performance. Storing two sets of data in the same family implies that during queries will have to scan all existing columns, even those not affected by the query in question. This is a costly operation in terms of resources consumption and in case of increase of data sources number, Accumulo offers the possibility to store a subset of column families together on disk. This approach improves performance when columns from different column families are accessed in the same scan and allows some sets of column families to be read from disk without having to read data from all the other column families. In our case, we consider that the two sets of data deals with two different use cases and at no time, will there be a need to do analysis on both data sets in an aggregate way.

We have seen in this section that through the flexibility of Accumulo's column-oriented structure and thanks to our new spatial model, we guarantee continuous data storage and integration as new spatial data is collected. Our model can support any number of columns and column families without specifying them beforehand, ensuring support for big spatial data for a wide variety of sources.

C. GeoJson view

In this work, we use Json to format spatial data before storing them on Accumulo. The reason of choosing Json is that it plays an intermediary role by allowing us to transform data to GeoJson without major transformations, thus combining spatial data with other no spatial attributes. Of course, the goal sought here is the standardization in spatial information transmission with third-party applications, notably for visualization purposes. Figure 4 gives the logical structuring of our GeoJson conversion model with a central class "Feature" that represent both spatial and non-spatial attributes.

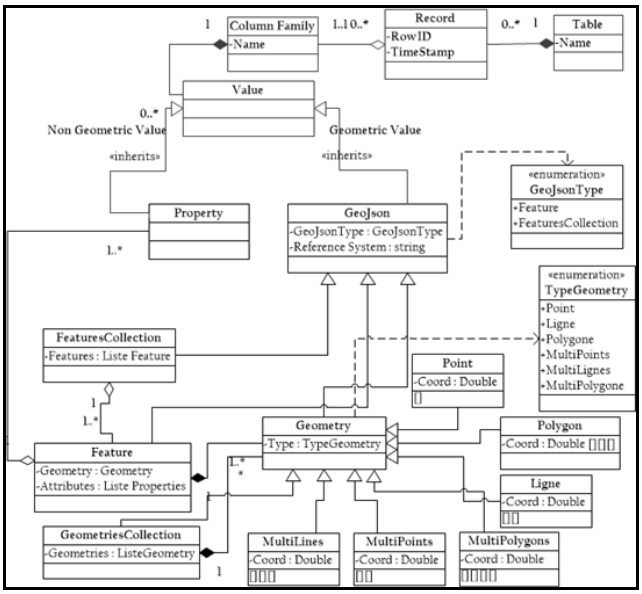


Figure 4. Accumulo GeoJson view class diagram

Figure 5 gives an example of transforming source data based on the basic spatial structure without major transformations and explains how we combine spatial and non-spatial attributes to form a unique structure using GeoJson.

Key					Value
RowID	*Column*			*TimeStamp*	
	Family	Name	Visibility		
Correspondence with the Accumulo model					
RowID	***Column Family***	***Column Name***	***Column Visibility***	***TimeStamp***	***Value***
Rowid1	Event	Origine Phone Number	Private	11905013	Tel a
Rowid1	Event	Destination Phone Number	Private	11905013	Tel b
Rowid1	Event	Geometry	Public	11905013	"geometry": { "type": "Point", "coordinates": [-3.123865, 23.3456]}
Rowid1	Event	Call duration in seconds	Public	11905014	34
Rowid1	Event	Date	Public	11905015	21/10/2020
Rowid1	Event	Hour	Public	11905018	15:01:09
Correspondence with the Data Model at the output of Accumulo					
{"type":"Feature", "geometry": { "type": "Point", "coordinates": [-3.123865, 23.3456]}, "properties":{" Origine Phone Number":" Tel a", " Destination Phone Number":" Tel b", " Call duration in seconds":" 34","Date":" 21/10/2020","Hour": "15:01:09"}}					

Figure 5. A GeoJson view example using Accumulo basic structure

IV. INDEXING LAYER

Accumulo stores data on tables using columns organized into families, it also offers data indexing using a single and unique field named RowId. These characteristics are similar, semantically speaking, to relational databases properties. Nevertheless, there is a major difference with regard to their indexing logic and data models differences. Relational database auto calculate indexes, whereas Accumulo does not generate indexes but gives full power to client applications to calculate them according to their own and specific logic using a Java API. This means that Accumulo indexing approach is flexible and can be adapted to new use cases such as handling big vector spatial data by calculating spatial data indexes based on big vector spatial data already stored on the database.

The particularity of vector spatial data is that they are multidimensional comprising both spatial coordinates and eventually temporal dimension; especially in case of a continuous use case that observes data over a specific time window. On the other hand, we know that Accumulo database manages indexes through a single and unique field named RowID. This field exists on any Accumulo table and is dedicated to store the table index. Due to this constraint, we apply a linearization function to transform several data dimensions inputs into one single value that we will store into the RowID field. Many linearization methods exist; we can classify them into three main families: regular and non-regular grids indexing, space-filling curves and geohash functions.

According to our research, only two Accumulo spatial index implementations exist, especially GeoMesa and GeoWave. GeoMesa supports space filling curve indexes while GeoWave supports Hilbert curve. This is not the case for Hbase (another implementation of Google Big Table). Ref. [19] proposes a spatial index structure using Geohash encoding. , Ref. [20] proposes a spatial index, based on a hybrid index structure, combining a quad-tree and a regular grid. Ref. [21] introduces a method to construct an Hbase spatial index by employing a Hilbert space-filling curve.

In this paper and since there is no extension of hexagonal grid indexing approach on a column-oriented database, we have decided to adapt this indexing approach to Accumulo and to associate it with our spatial model described above to fully manage spatial vector data.

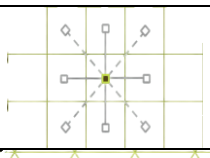
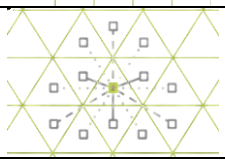
A. *Hexagonal tessellation*

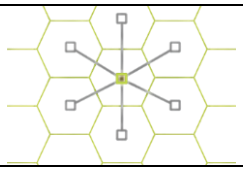
In the domain of planar tessellation, hexagons have several properties that make them more efficient that triangles or squares. Indeed, hexagons match circular shapes better than triangular and square shapes. If we consider that, we have a circle which radius is R circumscribed by a triangle, a square and a hexagon then the area of the circumscribed circle for each shape gives the following values (TABLE IV.):

TABLE IV. COMPARISION BETWEEN TRIANGLE, SQUARE AND HEXAGON COVERED AREASTRUCTURE

Triangle	Square	Hexagon
0.41 Π R2	0.64 Π R2	0.83 Π R2

TABLE V. DISTANCE COMPARISON BETWEEN CENTERS FOR THE THREE TYPES OF REGULAR GRIDS

	Square partitioning with 8 neighbours at two different distances between centres
	Triangle partitioning with 12 neighbours at three different distances

	Hexagon partitioning with 6 equidistant neighbors

Unlike square, rectangular and triangular grids, hexagonal grids are more compact, provide the highest resolution, and display uniform adjacency; each cell in a hexagon grid has six neighbors, all of which share one side with it, and all of which have centers exactly the same distance from its center as indicated in TABLE V.

B. Hexagonal Grids indexing

These different properties make hexagonal grids more optimal than square or triangular grids. For this reason, we have opted for a regular grid indexing based on hexagonal grids. This approach consists in grouping spatial data into hexagonal areas. These cells represent a collection of logical subdivisions of earth's surface in a way to cover the whole space.

In this research, we use a geospatial hexagonal grid indexing software library developed by Uber Technologies, Inc. and released for public use under open source license in 2018 [22]. We mainly use it for the linearization of vector data by converting them to hexagonal grids and each grid will have a 64-bit identifier [23]

C. Spatio-temporal data indexing on Accumulo

We propose in this paper a new RowId with a nested alphanumeric structure made up of several prefixes separated by a special character. This proposition gives every RowId portion the possibility to bring additional information and to query data in an efficient way. TABLE VI. Illustrates our proposed structure:

TABLE VI. ROWID NESTED STRUCTURE

IdGeo
#
IdAlgo
#
IDResolution
#
IdObjet
#
IdSpatial
#
IdTimeStamp

- IdGeo: prefixes the RowID with values ranging from zero to five in order to inform about geometry types. The table below gives the correspondence between the numerical value used in the index and the geometry type:

TABLE VII. ID GEO VALUES MAPPING

IdGeo	Geometry
0	Point
1	Polygon
2	Line
3	Multi-points
4	Multi-polygons
5	Multi-lines

- IDAlgo is an identifier that informs about the spatial indexing algorithm being used in order to apply its proper logic. It also gives the possibility to mix several indexing algorithms without changing the storage structure. For example, we can use GeoHash, ZorderCurve, HilbertCurve or any other linearization algorithm using the same storage structure. In the case of this paper, we assign the value 1 to IDAlgo to indicate that we are using H3 (a Uber Java implementation of hexagonal grid index).
- IDResolution is the third part of our RowID and provides information on the exact resolution used during the indexing process. In the actual case where we use Uber's H3 grid indexing, the resolution can have 16 possible values ranging from 0 to 15.
- IdObjet is the fourth part of our RowID and we use it to retrieve the spatial object business identifier. For example, it can be equal to a mobile phone number if we are in a use case of collecting and indexing Telecom subscriber's locations. In this case, IdGeo will be equal to 0 as the geometry type used is Point.
- IdSpatial is in the fifth position of our RowID and provides the spatial index value. In our case and based on the H3 library, we will have a 64-bit identifier.
- IdTimeStamp is at the sixth and last position of our RowID, it takes care of the time dimension so that we can position data over time. Depending on the use case, IdTimeStamp can be extracted from data to inform about events dates as it can obey to a specific logic (insertion dates in the Accumulo table for instance).

D. Combined index

Our first RowId design detailed in the previous section grants spatiotemporal data indexing but it does not take into account any other attribute. Which means that it is impossible for this index to respond effectively to queries that use both spatiotemporal filters in combination with other attribute filters. For example, if we need to extract the first 1000 subscribes that made at least five calls during the last two hours when they were closest to a particular point of interest. Then, Accumulo will execute a full scan to be able to retrieve data that meets spatial and non-spatial predicates. Which means that the spatial index already created will not bring any gain in terms of performance. The solution that seems obvious would be to integrate non-spatial attributes in the RowId, it will solve the performance problem. However, it has the disadvantage of

not being sustainable over time because of the possible creation of new attributes at any time and it is not convenient to act on the structure of the RowID whenever there is a need to index new attributes. To overcome this situation, we propose a new indexing method dedicated to non-spatial attributes while keeping our first RowID for spatiotemporal queries. This method uses a reverse indexing logic with a particular new structure (secondary table on Figure 6.) that gives, for a value of a given attribute, the list of records where this value appears. The notion of "reverse indexing" finds its explanation in the inversion of the roles of the two cells "RowID" and "Value" between our two tables: main and secondary on figure 5. Indeed, the RowId of the main table will be stored in the cell of the secondary table and the value corresponding to this RowId will be stored in the RowID cell of the secondary table as shown in the figure below:

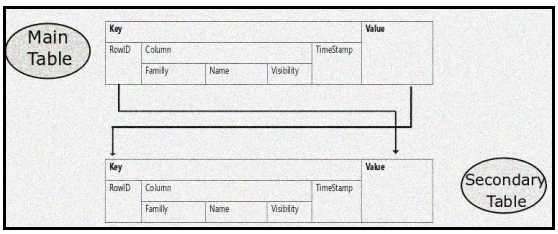


Figure 6. Accumulo inversed indexing

Data insertion and indexing occurs in a coordinated fashion so that each record successfully saved in the primary table is immediately indexed in the secondary table. In what follows, we detail the different procedures for creating records and corresponding indexes:

```
START
READ {Record R} FROM {Source S}
IF {R record is valid} THEN INITIALIZE {counter i to 0; precision P to one of the values 0..15}
WRITE {Explode fields} IN {FIELDS [0..N]}
WHILE {i <= N} DO
        WRITE {Transform FIELDS [i] as YYYYMMDDHHSS} IN {DATE}
        IF {FIELDS [i] matches the geometry of the event} THEN
        WRITE {Transform field FIELDS [i] into GeoJson} IN {Geo_C}
        WRITE {Indexing in H3 (P) of FIELD [i]} IN {IDx_H3_C}
        ELSE IF {FIELDS [i] matches a normal attribute} THEN
        WRITE {Format FIELDS [i] consistent with its type} IN {Att_C [i]}
                ELSE IF {FIELDS [i] matches an attribute with reverse indexing} THEN
WRITE {Format FIELDS [i] consistent with its type} IN {Att_Inv_C [i]}
        END IF
INCREASE {counter i by 1}
ENDWHILE
WRITE {Prepare RowID of record R}, {IDx_H3_C, DATE, Constants []} IN {RowID_R}
WRITE {Prepare the value of the R record}, {Geo_C, Att_C [0..j], Att_Inv_C [0..k]} IN {Val_R}
QUERY {Insert}, {RowID_R, Val_R} IN {the main table Accumulo P1}
END IF
INITIALIZE { counter k to 0}
WHILE {k <= M} DO
REQUEST {Insert}, {Att_Inv_C [k], RowID_R} IN {the secondary table Accumulo P2}
INCREASE {counter k by 1}
ENDWHILE
END
```

Figure 7. Data Insertion and indexing procedures

V. RANGE QUERY EXAMPLE

In this section, we detail an example of a spatial query in order to illustrate the use of our indexing and storage models. Range Query is one of the most widely used spatial queries. It helps to retrieve all the points located on a radius "r" from a focal point F (Xf, Yf) as shown on the figure below:

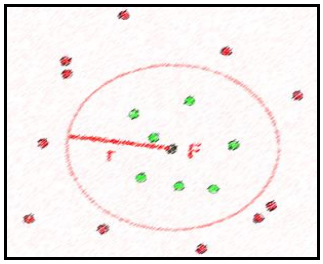


Figure 8. Range query logic

The H3 library provides a function named KRing that takes two parameters as inputs, the first contains an H3 index corresponding to the focal point F (Xf, Yf) and the second parameter contains the radius of the circle expressed in hexagonal cells. This means that KRing does not allow us to respond directly to our Range Query, there is indeed a need to add additional processing, notably to transform the Rang Query radius expressed in Euclidean metric into a Kring radius expressed in hexagonal grids. We do this calculation by simply dividing the radius R by the length of one of the sides of the hexagon corresponding to the index H3 of the focus F.

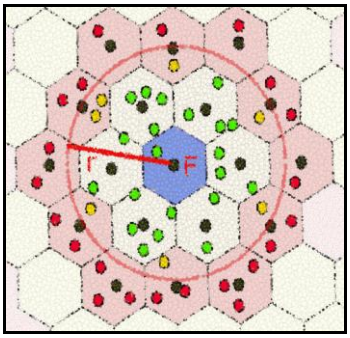


Figure 9. Range query using Hexagonal indexing

On Figure 9. we can see that there are parts of results located between the first two hexagonal cells and the third hexagonal cell. In other terms, they are located between the two following executions of the KRing function:

- Kring (H3Index (F), [r / h3.edgeLength])
- Kring (H3Index (F), [r / h3.edgeLength] +1).

The first execution does not contain all the results that satisfy RangeQuery conditions, we have highlighted these missing results in the figure (yellow points contained in the circle but are not containers on the hexagonal cells of the first execution). The second execution contains more points but some were wrongly added by the KRing function (red points in the figure; despite being out of the circle, the function returned them anyway). In order to have a more reliable result, we add

a third processing to calculate the minus between the results of both executions a) and b), once done we add a last check to eliminate all the points where the distance to the focal point is greater than the Radius. This solution makes it possible to benefit from Kring performance while maintaining a reliable result.

In what follows, we propose a second method by using the minimal circumscribed regular polygon P which best approximate the circle C(F,r). The determination of the polygon P is done by filling the circle C(F,r) with isosceles triangles and by putting them side by side, so that the sides of each triangles are equal to the radius as indicated in Figure 10. The intersection of the base of each triangle with the circle identifies two vertices of the polygon.

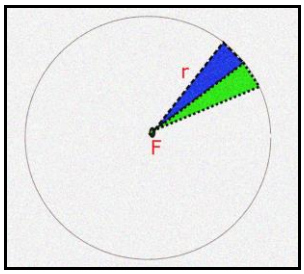


Figure 10. : Minimal circumscribed regular polygon

We now have to determine the number of sides by determining the number of triangles to use in the partitioning of circle C (F,r) . So, let N be the number of isosceles triangles Ti (Figure 12. .used in the partitioning of our circle C whose radius is r.

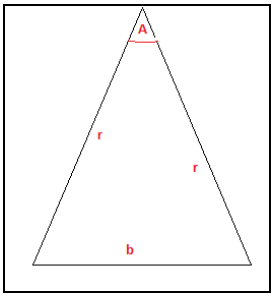


Figure 11. Isoscele triangle Ti formed from two identical sides of length r (the radius of the circle) and a base b

The triangle Ti area is equal to: $r^2 \sin(Ai) / 2$, where Ai is the angle formed by the two identical sides. Since the circle is partitioned by N identical Ti triangles then the angle Ai will be equal to 360 divided by N. The approximation therefore consists of finding N in such a way that the sum of the areas of the isosceles triangles is equal to the area of the disc formed by the circle C(F,r), which gives this equation:

$$N * r^2 * \sin(360/N) / 2 = \pi * r^2$$

By simplifying, we get the following result:

$$N * \sin(360/N) / 2 = \pi$$

Of course, the objective here is not to approximate the π number because Archimedes did it several centuries ago using this same method based on triangular partitioning. Our goal here is to determine the number of sides of our polygon P by determining the number of triangles Ti based on the real value of the π number. The table below gives the estimation error:

TABLE VIII. Π NUMBER APPROXIMATION ACCORDING TO THE NUMBER OF TRIANGLES

Number of triangles	Approximation	π - Approximation
50	3,13333084	8,26E-03
60	3,1358539	5,74E-03
70	3,13737581	4,22E-03
80	3,13836383	3,23E-03
90	3,13904132	2,55E-03
96	3,1393502	2,24E-03
100	3,13952598	2,07E-03

In our case, we opted for a polygon with 96 vertices in order to cover the circle C We have chosen to use 96 sides but for a maximum precision we can extend to higher values as displayed on table VIII. The choice depends on available hardware resources and calculation capacity.

Now that we have our polygon P with 96 vertices in a manner to cover, as much as possible, the circle C, we first fill it with all the hexagonal cells contained in it. This action is done using another H3 function named PolyFill. Second, we compare all the H3 indexes of Accumulo stored points with the polygon P contained cells H3 indexes. This method is more reliable as it uses the minimal circumscribed regular polygon P that best approximate circle C(F,r) and the more vertices he has the more optimal is the approximation as it gives the least false results compared to the precedent function KRing. Once all the hexagonal cells contained in the circle C are identified, it remains to launch an Accumulo search operation while keeping the maximum processing on the server side and not on the client application side. To achieve this result, we use three concepts. The first one is called "Scanner" and allows to connect to Accumulo tables in order to have access to all of its elements: Rowid, Columns, Column families, values. The second concept we use is the "Iterator" concept which allows to browse all the data in an Accumulo table to apply a specific filter. The third concept we use is filtering with strings regular expressions according to a previous nested RowID structure. This last concept allows us to translate a spatial query into regular expressions in line with the structure of our RowID by using quantification operators. Knowing the N+1 hexagonal cells HC[0,..N] that covers the circle C(F,r), our RangeQuery (F, r) query is equivalent to finding the list of points contained in the circle by applying the following regular expression at the scanner level for all the indexes H [0..N] corresponding to HC[0,..N] cells (Figure 12.):

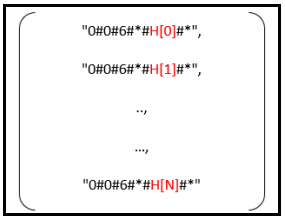


Figure 12. Regular expression spatial filter

If now we want to add a temporal filter so that the previous spatial query brings back only points contained by the circle (F, r) during a given time period for example during the first minute of the day of February 01, 2020 for example then just change the regular expression so that it looks like this:

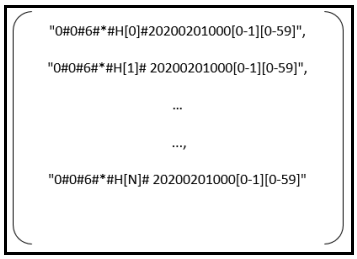


Figure 13. Spatiotemporal regular expression filter

VI. Conclusion

In this paper, we shed light on the impact of the big data phenomenon on spatial data and the challenges related to their exploitation, notably because of relational databases limitations in the presence of big data constraints. From there, we looked at the NoSQL Accumulo database to see if there was a possibility to adapt it to spatial data since it does not support them natively. In this sense, we have proposed a universal storage model specific to vector spatial data, capable of storing big spatial data independently of their source. We have also proposed a new index based on hexagonal tessellations in order to be able to execute spatial queries on stored big spatial data using existing Accumulo functions. However, not all spatial queries can be transcribed into search operations based on Accumulo index, especially complex queries. The solution offered by Accumulo consists of implementing parallel processing tasks using Hadoop MapReduce. According to literature, this processing engine is more suitable for a batch processing and not for interactive queries. Because Hadoop MapReduce has the disadvantage of reading/writing data from the disk, which generates latency, mainly when read/write operations are frequent. From this observation, one need for future work is to perform a new implementation of the MapReduce paradigm combining the advantages of parallel computing and in-memory storage while offering the possibility of interfacing it with the Accumulo database in order to reuse our actual spatial model.

References

[1] Gartner, Inc. Definition of Big Data - Gartner Information Technology Glossary. Mise à jour 17 Aout 2020. [En ligne]. Https://www.gartner.com/en/information-technology/glossary/big-data, [consulted the 01 Septembre 2020]Gartner, Inc. Definition of Big Data - Gartner Information Technology Glossary. Mise à jour 17 Aout 2020. [En ligne]. Https://www.gartner.com/en/information-technology/glossary/big-data 01 Septembre 2020]

[2] JASEENA, K. U. et DAVID, Julie M. Issues, challenges, and solutions: big data mining. CS & IT-CSCP, 2014, vol. 4, no 13, p. 131-140.

[3] Turner, Andrew. Introduction to Neogeography, O'Reilly, 1er janvier 2006, page 54.

[4] RAJ, Pethuru et RAMAN, Anupama C. The Internet of Things: Enabling technologies, platforms, and use cases. CRC Press, 2017.

[5] EF, CODD. A relational model of data for large shared data banks. Communications of the ACM, 1970, vol. 13, no 6, p. 377-387.

[6] CODD, E. F. An evaluation scheme for database management systems that are claimed to be relational. In : 1986 IEEE Second International Conference on Data Engineering. IEEE, 1986. p. 720-729.

[7] SANDBERG, Russel. The Sun network file system: Design, implementation and experience. In : in Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition. 1986.

[8] Redhat Inc. Stockage en mode fichier, bloc ou objet. Mis à jour le 01 Février 2018. [En ligne]. .https://www.redhat.com/fr/topics/data-storage/file-block-object-storage [consulted the 13 Mars 2018]

[9] GHEMAWAT, Sanjay, GOBIOFF, Howard, et LEUNG, Shun-Tak. The Google file system. In : Proceedings of the nineteenth ACM symposium on Operating systems principles. 2003. p. 29-43.

[10] GHEMAWAT, Sanjay, GOBIOFF, Howard, et LEUNG, Shun-Tak. The Google file system. In : Proceedings of the nineteenth ACM symposium on Operating systems principles. 2003. p. 29-43.

[11] BIALECKI, Andrzej. Hadoop: a framework for running applications on large clusters built of commodity hardware. http://lucene. apache. org/hadoop, 2005.

[12] CATTELL, Rick. Scalable SQL and NoSQL data stores. Acm Sigmod Record, 2011, vol. 39, no 4, p. 12-27.

[13] STONEBRAKER, Michael. SQL databases v. NoSQL databases. Communications of the ACM, 2010, vol. 53, no 4, p. 10-11.

[14] GRAY, Jim, et al. The transaction concept: Virtues and limitations. In : VLDB. 1981. p. 144-154.

[15] Pritchett, D. (2008). BASE: An acid alternative. ACM Queue, 6(3):48–55.

[16] BREWER, Eric A. Towards robust distributed systems. In : PODC. 2000.

[17] DECANDIA, Giuseppe, HASTORUN, Deniz, JAMPANI, Madan, et al. Dynamo: Amazon's highly available key-value store. ACM SIGOPS operating systems review, 2007, vol. 41, no 6, p. 205-220.

[18] CHANG, Fay, DEAN, Jeffrey, GHEMAWAT, Sanjay, et al. Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 2008, vol. 26, no 2, p. 1-26.

[19] SCOFIELD, Ben. NoSQL–Death to Relational Databases. In : Presentation at the CodeMash conference in Sandusky (Ohio). 2010. p. 01-14.

[20] S. He, L. Chu and X. Li, "Spatial query processing for location-based applications on Hbase," 2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA), 2017, pp. 110-114, doi: 10.1109/ICBDA.2017.8078787.

[21] HAN, Dan et STROULIA, Eleni. Hgrid: A data model for large geospatial data sets in hbase. In : 2013 IEEE sixth international conference on cloud computing. IEEE, 2013. p. 910-917.

[22] SHEN, Bo, LIAO, Yi-Chen, LIU, Dan, et al. A Method of HBase Multi-Conditional Query for Ubiquitous Sensing Applications. Sensors, 2018, vol. 18, no 9, p. 3064

[23] Brodsky, I. (2018). H3: Uber's Hexagonal Hierarchical Spatial Index. Uber Engineering. https://eng.uber.com/h3/.

[24] SAHR, Kevin. Hexagonal discrete global grid systems for geospatial computing. Archiwum Fotogrametrii, Kartografii i Teledetekcji, 2011, vol. 22, p. 363Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Md-hbase: a scalable multi-dimensional data infrastructure for location aware services. In Mobile Data Management (MDM), 2011 12th IEEE International Conference on, volume 1, pages 7–16. IEEE, 2011.

[25] HE, Shouwu, CHU, Longxian, et LI, Xiaoying. Spatial query processing for location based application on Hbase. In : 2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA). IEEE, 2017. p. 110-114.