Journal of Big Data

RESEARCH Open Access

Efficient storage of heterogeneous geospatial data in spatial databases

Atle Frenvik Sveen*

*Correspondence:
atle.f.sveen@ntnu.no
Norwegian University of Science and Technology, Trondheim, Norway

Abstract

The no-schema approach of NoSQL document stores is a tempting solution for importing heterogenous geospatial data to a spatial database. However, this approach means sacrificing the benefits of RDBMSes, such as existing integrations and the ACID principle. Previous comparisons of the document-store and table-based layout for storing geospatial data favours the document-store approach but does not consider importing data that can be segmented into homogenous datasets. In this paper we propose "The Heterogeneous Open Geodata Storage (HOGS)" system. HOGS is a command line utility that automates the process of importing geospatial data to a PostgreSQL/PostGIS database. It is developed in order to compare the performance of a traditional storage layout adhering to the ACID principle, and a NoSQL-inspired document store. A collection of eight open geospatial datasets comprising 15 million features was imported and queried in order to compare the differences between the two storage layouts. The results from a quantitative experiment are presented and shows that large amounts of open geospatial data can be stored using traditional RDBMSes using a table-based layout without any performance penalties.

Keywords: NoSQL, Document-store, Geospatial data, Spatial database, Relational database, Database benchmark

Introduction

New sources of geospatial data, such as the Internet of Things (IoT), Volunteered Geographic Information (VGI), and Open Geospatial Data, are becoming increasingly popular. This shift creates a demand for new ways to collect, manage, store, and analyse geospatial data. These challenges are mirrored in the general computer science concept of *big data*, a term describing datasets that are too large to be managed and processed by traditional technologies [1].

Laney [2] characterizes *big data* using the 3 Vs; Volume, Velocity, and Variety. These properties relate to geospatial data as well. Massive geospatial datasets originating from sensors are characterized by both high Volume and high Velocity, and open geospatial datasets from disparate sources comes with a high degree of Variety. This means that *geospatial big data* can be treated as a subset of *big data*, and opens up the possibility of using *big data* techniques to handle geospatial data [3, 4]. NoSQL (or Not Only SQL) data stores is one proposed solution to some of the challenges posed by *big data*. These data stores offer ways to handle the 3 Vs utilizing new techniques and architectures.

However, most new technology is no silver bullet. The promises of NoSQL may seem tempting, but there are several negative consequences of this approach as well. Chandra [5] uses the acronym Basically Available, Soft state, Eventual consistency (BASE) to describe NoSQL databases and contrast them with the ACID principle of relational databases. BASE also points to some of the drawbacks of NoSQL databases, such as the possibility of temporary inconsistencies. Another aspect is the lack of a universal query language. In light of this, we want to heed the advice from Stonebraker and Hellerstein [6] and examine if we really need to abandon the principles of Relational Database Management Systems (RDBMSes). In particular, we want to investigate if a combination of automated import routines and RDBMSes can offer the same advantages as NoSQL solutions when it comes to management and storage of heterogenous geospatial data.

In order to achieve this, we have implemented the Heterogeneous Open Geodata Storage (HOGS) system. This is a command line utility, written in Python, that leverages the open source GDAL/OGR geospatial library to automate imports of heterogenous geospatial data to a PostgreSQL/PostGIS database. By using both a traditional relational database layout and a NoSQL document-store layout we are able to benchmark both the import and query performance of the two storage layouts.

Background

RDBMSes dates back to the 1970′s [6], and Spatial database systems has been a term for about 30 years [7]. Today several of the best-known RDBMSes offer spatial capabilities according to the OGC Simple Feature Access specification. These spatial capabilities are often provided through an extension, such as PostGIS for PostgreSQL or Oracle Spatial for Oracle. In this paradigm, data types for spatial geometries are available alongside traditional data types and special SQL operators are available for spatial queries and operations. This means that a geometry can be treated as a normal column in a relational database table [8].

NoSQL data stores emerged in the late 2000 along with the "Web 2.0" movement [9]. The rise of these "not only SQL" systems was triggered by the need to handle "big data", or datasets that are too large to be managed and processed by traditional technologies [1]. This typically involves sacrificing or weakening the Atomicity, Consistency, Isolation, and Durability (ACID) principle underlining traditional RDBMSes [10].

There is no entirely agreed upon definition of NoSQL, but Cattell [9] offers six key features of such systems:

- Horizontal scaling.
- Replication and distribution over many servers.
- Simple call interface.
- Weakening of the ACID principle.
- Distributed indexes and RAM.
- The ability to add new attributes to records dynamically.

NoSQL data stores can also be categorized by capabilities and intended uses. Ameya et al. [11] presents five different types of NoSQL data stores; Key-value stores, column-oriented databases, document-stores, graph databases, and object-oriented databases.

The most interesting NoSQL data store type in the context of collections of open geospatial data is document-stores, with two well well-known examples being MongoDB and CouchDB. Document-stores store data as documents, reminiscent of records in a relational database, but without a pre-defined schema. Each document in the store has its own structure, and can include nested structures. A unique key is used for indexing the documents, which are usually stored using standard formats such as JSON (JavaScript Object Notation) or Extensible Markup Language (XML). The "no schema" approach of document-stores makes them popular to web developers. Partly due to their facilitation of quick integration of data from different sources, but also because they reduce the need for up-front database schema design [12].

These properties also make document-stores interesting for working with collections of open geospatial data. Such datasets originates from disparate sources and uses different file formats, coordinate systems, and attribute schemas [13]. Collecting open geospatial datasets in a traditional RDBMS requires a lot of work related to schema design and data import, where both attributes and geometries potentially have to be mapped, translated, and converted. The prospect of a "no schema"-solution that enable easy import of heterogenous datasets from a wide array of sources is intriguing. Maintaining an up-to-date collection of open geospatial data carries a lot of potential for developing value-added services and analyses, and the premise of NoSQL document-stores is that this can be achieved with less overhead. Both MongoDB and CouchDB offer spatial capabilities, using the JSON-based GeoJSON standard [14].

Another approach to tap into the benefits of a document-store is using an RDBMS that implements a document-store datatype. In these systems, a JSON or XML datatype with support for indexing and querying is made available to the RDBMS user. A document-based JSON storage type is implemented by several well-known RDBMSes, such as MySQL, Oracle, and PostgreSQL [15, 16]. These solutions have proved comparable to the NoSQL data-stores. For instance, Linster [17] reports a benchmark where the PostgreSQL document-store outperformed MongoDB on selecting, loading, and inserting a complex document dataset consisting of 50 million records.

Related work

Examples and benchmarks of NoSQL document-store datatypes for storing geospatial data are scarce in the existing literature. In the following we review the studies that most closely resembles the work we present.

A preliminary study by Navarro-Carrión et al. [18] examined the feasibility of using a NoSQL document-store to store EU land cover and land use data. In their experimental set-up, they used two PostgreSQL/PostGIS instances. One implemented a relational model, while the other implemented a NoSQL document-store model. Using these instances, they evaluated the query times of a bounding box search clause iteratively run using varying cell sizes. Using a dataset of more than 10.4 million soil occupation observations for roughly 2.5 million polygon geometries, they found that the document-oriented model was about 19% faster than the relational model. The authors point out that for several workflows a document-oriented model should be considered, and specifically points to massive polygon retrievals. An issue worth

```sql
SELECT
     column->>'key' as key
FROM
    tablename
WHERE
    column->>'key' = 'value';
```

Fig. 1 PostgreSQL json query example

noting is that they found the query syntax for JSON queries "somewhat convoluted" (see Fig. 1 for an example of the syntax).

Amirian et al. [19] performed a benchmark of three different storage strategies for "geospatial big data" using Microsoft SQL Server 2012. Four geospatial datasets containing 100,000, 1 million, 10 million, and 100 million polygons, was stored using a relational, a spatial, and an XML-based layout. Performance of these strategies where evaluated based on single feature and range query retrieval, as well as a scalability test. In their setup the XML document (NoSQL document-store) layout provided the best performance and scalability, but the authors recommend a polyglot geospatial data persistence approach for geospatial big data handling.

Maia et al. [20] evaluated the performance of storing VGI in the document-based NoSQL data store MongoDB. Their system stored geographic locations as points in MongoDB using the GeoJSON format. An important takeaway from their work is the fact that document-based NoSQL databases provide greater flexibility when storing heterogenous data and does not require any previous knowledge of the data schema. Their study also compared the performance of the NoSQL setup with a relational setup using PostgreSQL. While their results are considered preliminary, they "favoured the use of NoSQL in the persistence layer of a VGIS, especially when dealing with large amounts of data". It should however be noted that the read-time benchmarks performed did not include any spatial filters.

Bartoszewski et al. [21] compared the spatial query performance of MongoDB and PostgreSQL/PostGIS. Using point and polygon data, they performed point-in-polygon-, radius-, and composite nearest neighbour and intersection queries. Their results show that MongoDB outperforms PostGIS in the point-in-point (3× faster) and compound (6× faster) queries. However, with increasing radii, PostGIS outperforms MongoDB by a factor of about 3× in the radius queries. The authors also note that NoSQL databases are lacking in terms of available geospatial operations compared to RDBMSes, but postulate that this will change in the future.

Santos et al. [22] evaluated relational (PostGIS), document-based (MongoDB), and graph-based (Neo4J) databases with a focus on the needs of mobile users that involve constant spatial data traffic. Their goal is to "highlight aspects in which different spatial DBMS architectures behave differently", rather than provide a benchmark. They defined four query sets, based on operations typically performed in mobile spatial applications: Nearby Points of Interest, Map View, Urban Routing, and Position Tracking. For each set they defined a set of database queries. Their results show that PostGIS in general provides the best performance, and "provides the most spatial features". However, they note that MongoDB outperformed PostGIS in radius and k-NN queries. In addition, MongoDB is easy to scale horizontally.

Methodology

This section covers the implementation of the HOGS system and the experimental setup for the benchmarks performed on the system. First some common terminology is presented, then the architecture and implementation of the HOGS system is presented, before the experimental setup is described.

In this context we consider geospatial data to be described by the atomic unit of a *Feature*. A feature is a geographic *shape* (e.g. point, linestring, or polygon) as well as a list of accompanying key-value *attributes*. An example of a feature is a building footprint represented by a vector geometry describing a polygon, accompanied by attributes such as address, name of the owner, the year it was built, etc. A collection of features of the same type is a *Dataset* (or Feature collection). To continue the example, all building footprints in a city, municipality, or country makes up a specific building footprint feature collection. All features in a dataset shares the same attribute schema. Features belonging to a dataset are distributed as one or more files in one of several file formats and coordinate systems.

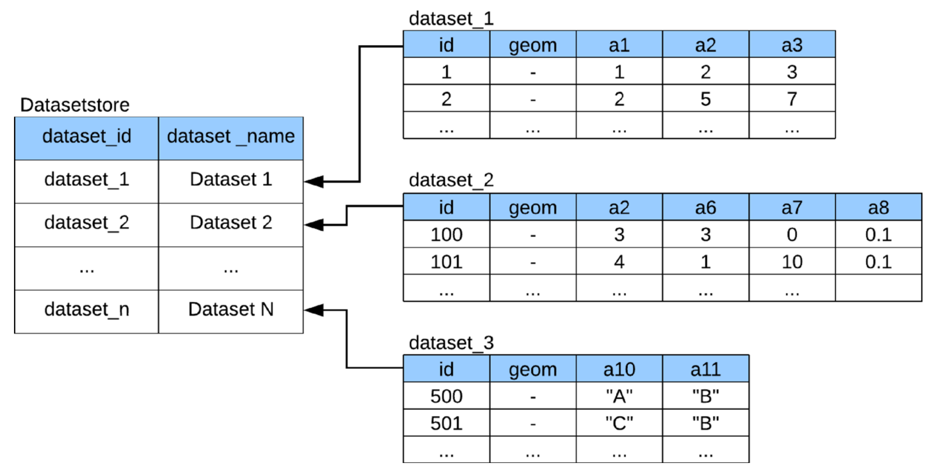
The HOGS system should be able to import multiple feature collections without any prior knowledge about the schema apart from what can be inferred from the data itself. The user supplies a list of files and what target dataset they belong to, as well as information about the database they are to be imported to. The Python programming language was chosen to implement the system, due to its multi-platform availability and the integration with the open source geospatial libraries GDAL/OGR and GEOS. The use of existing tools for common operations ensures a reduction of complexity and allows the system to support a wide range of geospatial file formats.[1]

Three overarching guidelines was followed when designing the system. First, the system should be *simple*. This is achieved by limiting the scope of the system, confining it to importing data. Second, the system should be *fast*. This is achieved by means of parallelization, exploiting the data structure to split the import into smaller tasks. Third, the system should offer *reproducibility*. This means that there should be no manual steps in the update procedure, so subsequent imports will behave the same way. This is ensured by the use of a configuration file.

Storage layouts

The two different *storage layouts* offered by HOGS determine how features and datasets are stored in the database. In the traditional *table-based* layout we create one database table per dataset. Each feature is a row in this table, with a column for each attribute, a geometry column, and a feature id column. While this approach could allow us to specify the geometry type as well, we opted for the generic *Geometry* data type, as some of our datasets contains mixed-type geometries. An example of the table-based layout is provided in Fig. 2.

[1] The list of supported vector formats in GDA/OGR at http://gdal.org/1.11/ogr/ogr_formats.html currently lists 78 formats.



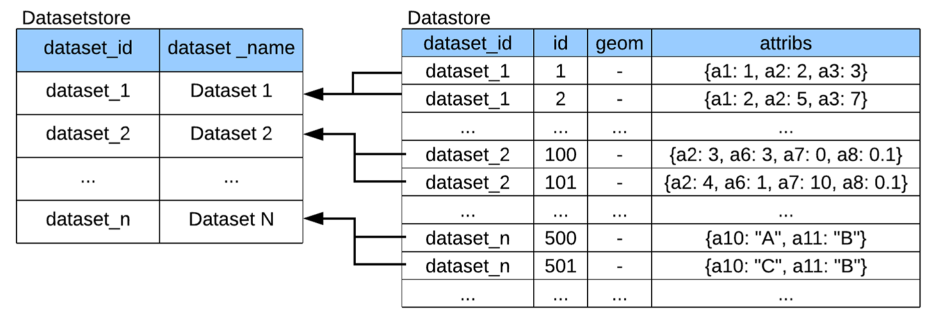
dataset_id	dataset _name
dataset_1	Dataset 1
dataset_2	Dataset 2
...	...
dataset_n	Dataset N

id	geom	a1	a2	a3
1	-	1	2	3
2	-	2	5	7
...	...	...	...	...

id	geom	a2	a6	a7	a8
100	-	3	3	0	0.1
101	-	4	1	10	0.1
...	...	...	...	...	

id	geom	a10	a11
500	-	"A"	"B"
501	-	"C"	"B"
...	...	...	...

Fig. 2 Illustration of the table-based layout. Each dataset has its own table, with each attribute as a separate column. The table name is referenced in the metadata-table



dataset_id	dataset _name
dataset_1	Dataset 1
dataset_2	Dataset 2
...	...
dataset_n	Dataset N

dataset_id	id	geom	attribs
dataset_1	1	-	{a1: 1, a2: 2, a3: 3}
dataset_1	2	-	{a1: 2, a2: 5, a3: 7}
...	...	...	...
dataset_2	100	-	{a2: 3, a6: 3, a7: 0, a8: 0.1}
dataset_2	101	-	{a2: 4, a6: 1, a7: 10, a8: 0.1}
...	...	...	...
dataset_n	500	-	{a10: "A", a11: "B"}
dataset_n	501	-	{a10: "C", a11: "B"}
...	...	...	...

Fig. 3 Illustration of the jsonb layout. All datasets are stored in the common feature table, with their attributes in a jsonb-column. The *dataset_sid*-columns links features to a dataset in the metadata-table

In the NoSQL document-store inspired *jsonb* layout, we create a single database table that holds features for all datasets. Each row in this table is a feature, with the dataset id stored in a column. Geometry and feature id are also separate columns, similar to the table-based layout, as shown in Fig. 3. The main difference is that all the attributes are stored in a column of the jsonb type. The layout of the jsonb layout feature table is shown in Fig. 3. Another aspect of the jsonb layout is how it uses database views to emulate the table-based layout. For each dataset stored in the feature table a database view that expose the attributes as individual columns is created. This is done since most GIS tools are designed to work with the traditional table-based layout. By hiding the underlying structure from these tools, we ensure that they still work as expected.

HOGS support dataset versioning by using incremental version numbers with associated timestamps. When importing a dataset with an existing dataset id, this is considered a new version of the same dataset and the version number is increased. This means that an import can be run several times on the same database, but the storage layout of a previously initialized database cannot be changed.

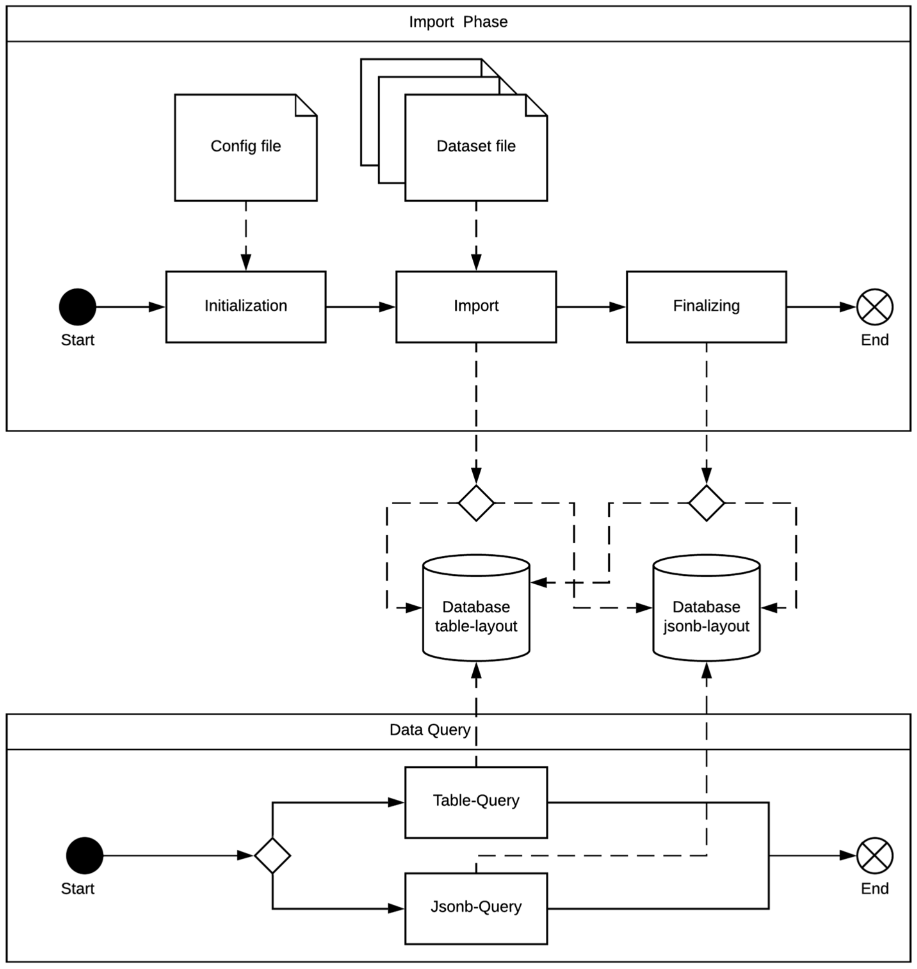


Fig. 4 An overview of the experimental setup. The import phase (top section) consists of three phases, and results in a database populated with the import dataset, using either the table-layout or the jsonb-layout, depending on the chosen storage strategy indicated in the config-file. The query phase is a separate process, which issues different queries depending on the storage layout

Import workflow

The actual import process consists of three phases: the initialization phase, the import phase, and the finalizing phase, as shown in the upper portion of Fig. 4. The content of these phases depends on the chosen storage layout and the previous state of the database.

In the *initialization phase* the configuration file is read and HOGS connects to the provided database. The first time HOGS connect to a database two metadata tables are created. These holds information about the stored datasets and determines the storage layout. If the jsonb layout is chosen, the aforementioned feature table is also created. The next initialization step is to parse the list of files associated with each dataset to be imported. The first file in each dataset is read using GDAL/OGR, to determine the attribute schema of the dataset. This information is stored in a metadata table. For the table-based layout the schema is used to create a temporary import table for each dataset. For the jsonb layout this information is used to create or update the database views.

In the *import phase* the files for each dataset is read and parsed, geometries are checked for errors and optionally transformed to geographic coordinates in the WGS84 datum (EPSG:4326), and then data is written to the database using a COPY statement. This was found to be the fastest operation when writing a large number of rows to the database. The geometry is copied using the EWKB format, the jsonb-attributes as a json encoded string, and the other columns are copied as native data types. Since the COPY statement bypasses table constraints on the database the geometries are validated using the GEOS library before they are written to the database.

Since each file in an import is independent of the other files, this phase can be executed in parallel. This can reduce the import time drastically and is an optimization worth implementing. HOGS achieve parallelization by creating a pool of import workers using the Python multiprocessing module. The size of this pool is set by the user and should correspond to the number of available CPU cores. In principle this pool could be distributed on individual machines as well, with one machine acting as a master node, coordinating the work.

When all files belonging to a dataset is imported, the dataset import proceeds to the *finalizing stage*. The contents of this stage depend on the chosen storage layout. For the jsonb layout this phase consists of creating, or updating, the aforementioned views and updating the metadata table to point to the correct version. For the table-based layout the finalizing stage creates an index on the geometry column, swaps the current version of the table with the temporary table, and stores the previous table with an identifier including its version number. When all datasets have finished the finalizing phase the import is completed.

Experimental setup

The HOGS system implements both a NoSQL storage approach and a traditional table-based storage layout. Therefore, we utilize HOGS in our laboratory setup to examine if there are any differences in import speed and query performance between the two layouts. We performed a quantitative analysis consisting of a series of imports and database queries. Using the same collection of datasets, we measured three features of each data storage layout: *import speed*, *query speed*, and *database size*.

All benchmarks where performed using an open geospatial dataset from the Norwegian Mapping authority known as *N50*. This is a 1:50,000 scale topological dataset of the Norwegian mainland, containing eight sub-datasets (feature collections), covering features such as area cover, transportation networks, place names, and height contours. Each of these sub-datasets have different attribute schemas and use different geometry types. The dataset is delivered in the Norwegian text-based geospatial file format SOSI, divided by dataset type and municipality. In total, the complete dataset contains 3415 files, totalling 7.9 GB on disk after extraction. This corresponds to approximately 15 million features, more specifically 2 million point features, 10 million linestring features, and 3 million polygon features. An overview of the N50-dataset is provided in Fig. 5.

The experimental setup consisted of a standard enterprise hardware setup, equipped with an Intel Core i7-4710MQ Processor, 32 GB RAM, and a 300 GB HDD, running Windows 10. PostgreSQL 9.6.3 with PostGIS 2.3 was installed using a Docker-image. The installation used the default configuration and was wiped between each run. HOGS

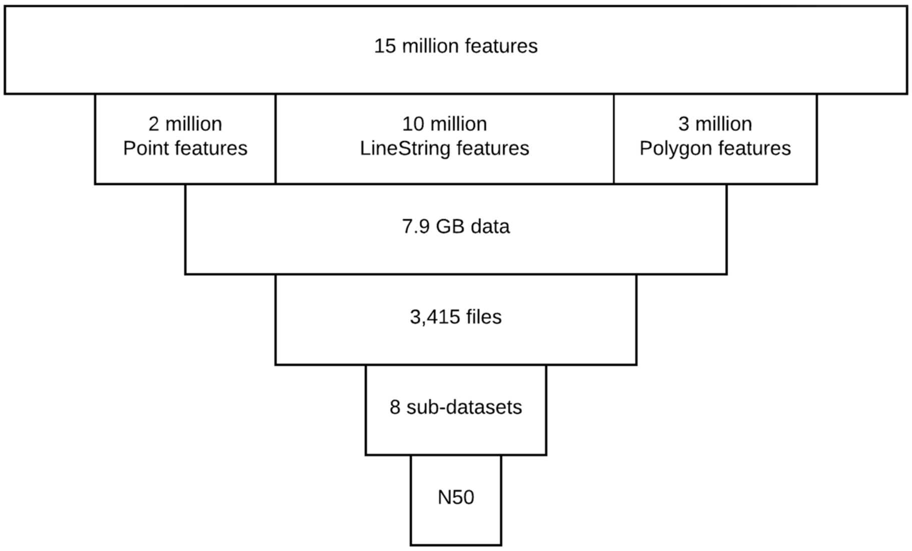


Fig. 5 Overview of the N50 dataset

Table 1 Benchmark results for the two examined storage layouts

	Import		**Query—intersect**		**Query—intersect/attribute**		**Disk size**
	Speed (m)	**SD**	**Speed (s)**	**SD**	**Speed (s)**	**SD**	**(GB)**
Table-based	*79*	3.57	*19*	0.54	*99*	3.24	*12.29*
Jsonb	179	4.50	25	1.00	162	3.30	17.50
Difference	100		6		63		5.21

The better results for each metric are emphasized

itself was run using Python 2.7 on the Windows Subsystem for Linux. This means that the complete experiment was run on a single machine, with no network speed and latency to consider.

Results

Timing from the experiments are affected by several factors. We have chosen to focus on the relative difference between the two storage layouts, not the elapsed time on its own. The results of the actual benchmarks are summarized in Table 1 and presented in detail in the following sections.

Import benchmark

Import speed is calculated as the time it takes from HOGS is provided with a configuration file containing a list of datasets and associated files until the data in these files are available in the provided database in the specified layout. This is the upper portion of Fig. 4. In our case this means the time it takes to read the 3415 SOSI files from disk and store their contents in the database.

```python
def get_intersects(self, table_name, geom):
    with self.conn.cursor() as cur:
        query = sql.SQL('''
            SELECT * FROM {}
            WHERE ST_Intersects(geom, ST_GeomFromWKB(%s, 4326))
        ''').format(
            sql.SQL(table_name)
        )

        cur.execute(query, (psycopg2.Binary(geom),))
        res = []
        for record in cur:
            res.append(record)
        return res
```

Fig. 6 Intersect query used for benchmarking

We relied on the built-in logging capabilities of HOGS, noting the time an import started and finished. The import was run five times for each storage layout, with HOGS presented with a new database instance on each run.

Our benchmarks show that the average import speed for the table-layout is 1 h and 19 min, while the jsonb layout on average took 3 h. The results indicate that the table-based layout is 56% faster than the jsonb layout with regard to the import phase.

Query benchmark

Database query optimization is complex and it is impossible to provide a benchmark that covers all usage patterns of a generalized geospatial data storage such as the one described here. However, we chose to base our query benchmarks on the usage pattern of a known system and use data gathered from the logs of this system.

The system in question performs a series of intersection queries using a query polygon against a series of datasets in order to find areas of interest. From the query logs of this system we extracted 840 query geometries. These polygons cover areas in the range 1–100 m^2 on the Norwegian mainland and are distributed according to the needs of the users of the system. The query benchmark is depicted in the lower portion of Fig. 4, and is independent of the design of the import phase, as it only relates to the resulting database contents and layout.

Two queries were designed to be run against each of the eight datasets in the n50 dataset. One plain intersection query using the PostGIS *ST_Intersect* and one query consisting of an intersection as well as an attribute query (see Figs. 6 and 7). For the attribute query we chose the attribute "objekttypenavn", which is present for all our datasets, and for each dataset we used all the distinct values of this attribute. This means that both queries were executed about 7000 times.

Each of these series of queries were run five times for each database layout, and the total query time for each layout was averaged. For the plain intersection queries the average time was 19 s for the table-based layout, and 25 s for the jsonb layout. This

```python
def get_intersects_with_objtype(self, table_name, geom, objtype):
    with self.conn.cursor() as cur:
        query = sql.SQL('''
            SELECT * FROM {}
            WHERE ST_Intersects(geom, ST_GeomFromWKB(%s, 4326))
            AND objekttypenavn = %s
        ''').format(
            sql.SQL(table_name)
        )

        cur.execute(query, (psycopg2.Binary(geom), objtype, ))
        res = []
        for record in cur:
            res.append(record)
        return res
```

Fig. 7 Intersect and attribute query used for benchmarking

means that the plain intersection queries are 25% faster using the table-based layout. For the intersection queries with an additional attribute query the average query time in the table-based layout was 99 s, while the jsonb-layout took 162 s on average. This means that intersection queries with attribute queries are 39% faster using the table-based layout.

Database size

This benchmark measures the actual size on disk used to store the datasets using the two different storage layouts. The size of the databases was measured using the PostgreSQL system table *pg_database*, and the operator *pg_database_size*. These numbers show that the table-layout database uses 12.29 GB on disk, while the jsonb-layout use 17.5 GB. This means that the table-based layout takes up 30% less space than the jsonb-layout.

Discussion

The performed benchmarks show that the table-based layout performs better than the NoSQL-inspired jsonb-layout on all metrics. Insertion speed is the metric with the largest difference. Here, the table-based layout is able to insert the test-data more than twice as fast as the jsonb-layout. These findings contradicts similar studies found in literature [18, 23], which report that NoSQL document stores or data types outperform relational layouts.

However, many factors influence benchmark results, and while the setups in the related studies are similar there are several differences in design that may explain the difference in results. We suspect that the most important factor in our setup is table size. Since our two layouts are both implemented in PostgreSQL/PostGIS, and both layouts use the PostGIS geometry types, the main difference between them is the way attributes are stored, and how many tables are used. This difference holds the explanation to why the table-based layout performs better.

Both layouts use a spatial index for the geometry column. In the case of insertion, the way this is handled differs. For the table-based layout the index is created after data is inserted into the table. In the jsonb-layout this is not an option, as we are inserting data into a common table. This means that for the jsonb-layout, the spatial index has to be created in the initialization phase and updated in-place during the import, which is more time-consuming than creating an index after all data is added. In the case of data queries, the main difference is table size, with the common table in the jsonb-layout being larger. While this table is indexed on dataset id, it is still faster to directly query a table with just the relevant features than to select these using an index.

None of the examined related studies used data that could be logically segmented into sub-datasets, and thus the table sizes would have been similar in both cases. This may explain why our findings differ. However, many geospatial datasets can be segmented into separate datasets by partitioning on what types of features are being mapped. If this is the case, our results show that a table-based layout is favourable. A counterpoint is that the "one table per dataset" approach can be combined with the jsonb-layout as well. While this is technically true, a key feature of NoSQL data stores is that there is no need to logically separate data in tables. In order to keep with this philosophy, we chose to implement one common table for the jsonb-layout.

Another important aspect of a database used for managing open geospatial data is usability. Navarro-Carrión et al. [18] noted that the query syntax used for the PostgreSQL JSON data type is "somewhat convoluted", an assessment we find to hold true (see Fig. 1 for an example). In addition, we found that widely used desktop GIS packages such as QGIS are unable to read attributes stored as jsonb with the same ease as they read traditional tables. This was mitigated by creating database views that maps the jsonb-syntax to a traditional relational table-layout, with one dataset per table and one attribute per column.

We used the HOGS system to perform benchmarks on the Norwegian n50-dataset, delivered as files in the SOSI format. This does not imply that the system is limited to one file format. Due to the use of the GDAL/OGR library, a plethora of geospatial vector formats (78 at the time of writing) can be imported using HOGS. For example, we have successfully imported data downloaded from OpenStreetMap using HOGS.

Conclusions and further work

We have found that, for homogenous collections of spatial datasets, a traditional one-table per-dataset layout outperforms a NoSQL document-store combined-table layout. The traditional layout performs better on both insertion and query speed, and it uses less storage space. We expected that the NoSQL approach would enable an easier insertion routine, but with the HOGS system leveraging GDAL/OGR we found that the overhead of creating individual tables for each dataset can be automated and introduces no extra complexity.

We also found that while a single table containing a heterogenous mix of features from different datasets intuitively sounds easier to work with, this kind of layout is not compatible with a range of off-the-shelf WMS-servers and desktop GIS packages. In practice this means that a NoSQL layout must emulate a traditional table-based layout using views in order to work with such applications.

These findings differ from the other studies examined. While one explanation for this discrepancy may be the fact that we used data that could be segmented into sub-datasets, this shows that further examination is required. A more thorough benchmark-setup, including a larger pool of datasets is a natural next step. Leveraging other sources of open geospatial data, such as OpenStreetMap, and European INSPIRE-data, would enable us to verify our results on a wide selection of geometry types and attribute schemas. Another possible route is to enable a cloud-based lab-setup, where automation in used to create, run, and tear down the database and import environments between each test-run. This would enable us to exclude all possible side-effects of running the benchmarks on a single hardware setup, which would also allow for adjustment of additional parameters, such as processor speeds and available memory.

In terms of further work, a third storage layout worth examining is the so-called Data Lake [24]. In this concept, the data is stored "as is" in raw format, and only processed when needed [25]. This allows for easy storage of vast amounts of data, but we envision this would present its own performance issues related to queries, where both geometries and attributes will have to be parsed and transformed. However, we find this concept interesting, and would like to investigate how it can be applied to geospatial data.

In conclusion, the results presented in this paper indicate that the NoSQL layout is slower, both in terms of import and query speed, when considering heterogenous geospatial data. In addition, the NoSQL layout does not offer any additional simplification of the import process. Based on these conclusions, we cannot recommend the use of the jsonb-datatype in PostgreSQL for storing geospatial data that can be segmented into homogenous datasets. This statement holds as long as the storage-space requirements does not exceed the capabilities of a single database instance. This in turn means that relatively large amounts of open geospatial data can be efficiently stored and queried using traditional RDBMS technologies. This approach is beneficial, as it enables the use of existing software integrations and does not require a weakening of the ACID-principle.

Abbreviations

ACID: Atomicity, Consistency, Isolation, and Durability; BASE: Basically Available, Soft state, Eventual consistency; EWKB: Extended Well-Known Binary; HOGS: Heterogeneous Open Geodata Storage; IoT: Internet of Things; JSON: JavaScript Object Notation; NoSQL: "Not Only SQL"; RDBMS: Relational Database Management System; SOSI: Samordnet Opplegg for Stedfestet Informasjon—Systematic Organization of Spatial Information; VGI: Volunteered Geographic Information; XM: Extensible Markup Language.

Acknowledgements

The author would like to thank Dr. Alexander Salveson Nossum for providing valuable feedback on the contents of the paper, and Dr. Birgitte Hjelmeland McDonagh for valuable help with grammar, style, and proofreading.

Authors' contributions

AFS is the sole author of this manuscript, as well as the developer of the HOGS system presented herein. The author read and approved the final manuscript.

Funding

This work was supported by Norkart AS and The Research Council of Norway [Grant Number: 261304I].

Availability of data and materials

The datasets used for then benchmarking in this manuscript are available from the corresponding author on reasonable request.

Competing interests

The author declares no competing interests.

Received: 25 June 2019 Accepted: 4 November 2019
Published online: 18 November 2019

References

1. Chen M, Mao S, Liu Y. Big data: a survey. Mobile networks and applications. US: Springer; 2014. p. 171–209.
2. Laney D. 3D data management: controlling data volume, velocity and variety. META Group Research Note 6; 2001.
3. Lee J-G, Kang M. Geospatial Big Data: challenges and opportunities. Big Data Res. 2015;2:74–81. https://doi.org/10.1016/j.bdr.2015.01.003.
4. Li S, Dragicevic S, Castro FA, et al. Geospatial big data handling theory and methods: a review and research challenges. ISPRS J Photogramm Remote Sens. 2016;115:119–33.
5. Chandra DG. BASE analysis of NoSQL database. Fut Gen Comput Syst. 2015;52:13–21. https://doi.org/10.1016/j.future.2015.05.003.
6. Stonebraker M, Hellerstein J. What goes around comes around. Readings in database systems. 2005;4:1724–35.
7. Güting RH. An introduction to spatial database systems. VLDB J. 1994;3:357–99. https://doi.org/10.1007/BF01231602.
8. OGC. OpenGIS® implementation standard for geographic information—simple feature access—part 1: common architecture, vol. 93. Wayland: Open Geospatial Consortium Inc; 2010.
9. Cattell R. Scalable SQL and NoSQL data stores. ACM SIGMOD Rec. 2011;39:12. https://doi.org/10.1145/1978915.1978919.
10. Leavitt N. Will NoSQL databases live up to their promise? Computer. 2010;43:12–4. https://doi.org/10.1109/MC.2010.58.
11. Ameya N, Anil P, Dikshay P. Type of NOSQL databases and its comparison with relational databases. Int J Appl Inf Syst. 2013;5:16–9.
12. Chasseur C, Li Y, Patel JM. Enabling JSON document stores in relational systems. In: WebDB; 2013. p. 14–15.
13. Sveen AF. The open geospatial data ecosystem. Kart og plan. 2017;77:108–20.
14. MongoDB. MongoDB Manual; 2018. https://docs.mongodb.com; https://docs.mongodb.com/manual/reference/geojson/index.html. Accessed 11 Apr 2018.
15. Del Alba L. Faster Operations with the JSONB Data Type in PostgreSQL; 2017. https://www.compose.com/articles/faster-operations-with-the-jsonb-data-type-in-postgresql/. Accessed 11 Apr 2018.
16. Petković D. JSON integration in relational database systems. Int J Comput Appl. 2017;168:14–9. https://doi.org/10.5120/ijca2017914389.
17. Linster M. Postgres outperforms MongoDB and ushers in new developer reality. In: The EDB Blog; 2014. https://www.enterprisedb.com/node/3441. Accessed 10 Apr 2018.
18. Navarro-Carrión JT, Zaragozí B, Ramón-Morte A, Valcárcel-Sanz N. Should eu land use and land cover data be managed with a Nosql document store? Int J Des Nat Ecodyn. 2016;11:438–46. https://doi.org/10.2495/DNE-V11-N3-438-446.
19. Amirian P, Basiri A, Winstanley A. Evaluation of data management systems for geospatial big data. Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics). Cham: Springer; 2014. p. 678–90.
20. Maia DCM, Camargos BDC, Holanda M. Performance analysis on voluntary geographic information systems with document-based NoSQL database. Stud Comput Intell. 2018;718:181–97. https://doi.org/10.1007/978-3-319-58965-7_13.
21. Bartoszewski D, Piorkowski A, Lupa M. The comparison of processing efficiency of spatial data for PostGIS and MongoDB databases. In: Kozielski S, Mrozek D, Kasprowski P, et al., editors. Beyond databases, architectures and structures. Paving the road to smart data processing and analysis. New York: Springer International Publishing; 2019. p. 291–302.
22. Santos PO, Moro MM, Davis CA. Comparative performance evaluation of relational and NoSQL databases for spatial and mobile applications. In: Chen Q, Hameurlain A, Toumani F, editors. Database and expert systems applications. New York: Springer International Publishing; 2015. p. 186–200.
23. Amirian P, Basiri A, Winstanley A. Efficient online sharing of geospatial big data using NoSQL XML databases. In: 2013 fourth international conference on computing for geospatial research and application. New York: IEEE; 2013. p. 152–152.
24. Dixon J. Pentaho, Hadoop, and Data Lakes. In: James Dixon's Blog; 2010. https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/. Accessed 9 Sept 2019.
25. Miloslavskaya N, Tolstoy A. Big Data, Fast Data and Data Lake Concepts. Procedia Comput Sci. 2016;88:300–5. https://doi.org/10.1016/j.procs.2016.07.439.

Publisher's Note