


RESEARCH

Open Access

Benchmarking geospatial database on Kubernetes cluster



Bharti Sharma¹, Poonam Bansal¹, Mohak Chugh¹, Adisakshya Chauhan¹, Prateek Anand², Qiaozhi Hua^{3*}  and Achin Jain⁴

* Correspondence: 11722@hbuas.edu.cn

³Computer School, Hubei University of Arts and Science, Xiangyang 441000, China

Full list of author information is available at the end of the article

Abstract

Kubernetes is an open-source container orchestration system for automating container application operations and has been considered to deploy various kinds of container workloads. Traditional geo-databases face frequent scalability issues while dealing with dense and complex spatial data. Despite plenty of research work in the comparison of relational and NoSQL databases in handling geospatial data, there is a shortage of existing knowledge about the performance of geo-database in a clustered environment like Kubernetes. This paper presents benchmarking of PostgreSQL/PostGIS geospatial databases operating on a clustered environment against non-clustered environments. The benchmarking process considers the average execution times of geospatial structured query language (SQL) queries on multiple hardware configurations to compare the environments based on handling computationally expensive queries involving SQL operations and PostGIS functions. The geospatial queries operate on data imported from OpenStreetMap into PostgreSQL/PostGIS. The clustered environment powered by Kubernetes demonstrated promising improvements in the average execution times of computationally expensive geospatial SQL queries on all considered hardware configurations compared to their average execution times in non-clustered environments.

Keywords: Distributed data processing, Geospatial databases, Cluster computing, Geospatial-data, Geospatial-databases, Benchmarking, Database-as-a-service (DBaaS)

1 Introduction

The use of geospatial data has been increased in many applications, including traffic management, ride-hailing services, and food sector, etc. The volume of geospatial data is predicted to increase by 20% every year. The increase of geospatial information requires new architectures or systems to handle data thus creating new challenges. At present, mainly two types of databases store geospatial information: relational databases and NoSQL databases. Relational databases are the most universally used and the most developed database information systems used in industries for decades. Due to a lack of native support for geospatial data in relational databases, some modern databases have updated and changed their database design specifically for spatial data to

extend support for various operations on geo-data [1]. Examples of relational databases for geographic information include PostGIS, WebGIS, Oracle 19c, the Microsoft Azure SQL Database, and a few more. The relational databases can define spatial entities, extend their support for the different spatial data entities (polygon relationships), and acquire various optimizations for refining the query operations execution time. With the arrival of new cloud technologies, spatial information application systems are also incurring updations and changes rapidly to handle different operations on complex and colossal data efficiently [2]. Storing, managing, querying data, and managing geospatial databases in the environment effectively are the problems that are being tried to be solved for many years.

Running multiple machines as a cluster is a method of managing multiple containers. Docker is one of the technology solutions that are compatible with any computer to run containerized applications. Containerization is a process of isolating applications from the host machine. It creates an environment similar to having a separate operating system, even though there might be other containers running on the host machine. Containerization helps the host machine to run, create, and manage multiple containers on a single host machine. Kubernetes is an open-source technology that serves as a container orchestration tool that automates installing and managing a cluster of Docker containers. Docker images contain the desired application and service elements, and Kubernetes can be used to deploy and manage these components. Kubernetes allows us to automate the provisioning of containers, networking, load-balancing, security, and scaling across all its nodes [3].

Clustering and orchestration of containers automatically allocate the client to the machine with the least resource usage. Database clustering and containerization take a different approach in order to maintain atomicity, consistency, isolation, and durability (ACID) properties. In the database cluster mode, every single node is fully isolated and has its own methods of managing the data and ACID properties. Since there is more than one server instance, consistency is difficult to operate, and the concept of eventual consistency is used. But the result of this offers an alternative in the event of a crash or a failure.

The traditional data management technologies face frequent read-write problems and scalability problems while dealing with such dense and complex spatial data. Using Geographic Information System (GIS) in a cluster environment can be an effective way to solve spatial data problems by having the benefits of horizontal extension on low-cost computers, which can provide large and scalable storage, computing power, load balancing, high availability, and monitoring and automation. The structure and principle of containers in the cluster environment make the technology very prominent and efficient for database workflows. One of the reasons being that once a container has been built, it will run on any platform. The cluster environment can ensure availability, management of resources which simplifies reproducibility and deployment. Performing different kinds of operations on geospatial data is compute intensive, i.e., it needs high computational resources to run. Therefore, there is a need for evaluating the performance of compute intensive operations on geospatial data running in a cluster environment of Kubernetes.

2 Related work

There have been many studies on spatial data storage due to the increasing spatial data and processing scale. This prompts the development of spatial data technologies, including aspects: the data model for storage, spatial indexes, and various types of query operations processing. Management and processing of spatial vector data is complex and needs unique storage models, mechanisms for processing, scanning, and specific usage systems for its use in various applications. A geographic information system (GIS) is used for geospatial vector data gathering, accumulation, and processing to assist the general or specific types of applications [1]. The fast-paced development of data systems, space technology, and sensor technology has led to an increase in the huge volume of geospatial data in several subjects. Hence, spatial data services are often used with cloud technologies to respond faster [2]. Geospatial data representing information is confined to the location object, structure, and characteristics of entities and entity dependency on each other [3]. New geospatial applications need versatile schematics, reasonably faster execution of query operations, and more scalability than the existing conventional geospatial relational databases [4]. In fact, the bottlenecks observed in the management and processing of spatial vector data have been continuously the driving force for the development of system designs due to the limitations which reside in the current systems which are used for handling the specific type of huge information and its manipulations and computations [5]. In spatial information systems, support for various spatial data services as used in any information system is required. Several experiments and studies have realized that conventional relational databases are not efficient for big data storage and queries for industrial purposes operating at large-scale accessing millions of data points at enormous speed in various geospatial applications [6, 7]. NoSQL databases are widely considered for storing big data due to the capability to accumulate, manage, and support the creation of various types of indexes on data fields while horizontally scalable providing the ability to serve the huge number of retrieval operations [8].

Subjective comparison of experiments has also shown that no fixed schema-based databases have faster execution or query processing times than flexible schema databases when operating on a huge volume of data [9, 10]. Creating spatial indexes is crucial to validate spatial databases to access and view data efficiently; thus, affecting the overall performance of the spatial databases compared to using non-indexed spatial databases directly [10, 11]. All these operations are not only confined to huge storage space but also need comparatively more computation power. In the subjective comparison of the widely used query operations in various database systems, NoSQL databases have outperformed relational databases. Current NoSQL database designs used for industrial purposes cannot serve as a fully viable option for geospatial data. NoSQL databases have some advantages than traditional relational databases that can be easily operated as a distributed system and do not have fixed structured data, which eases its capability to be scaled horizontally [12, 13].

Collecting open geospatial datasets in a traditional relational database management system (RDBMS) requires a lot of work related to schema design and data import, where both attributes and geometries have to be mapped, translated, and converted [14, 15]. Relational databases have also some advantages compared to NoSQL databases that provide standard ACID properties (atomicity, consistency, isolation, and durability)

that maintain the integrity of the database system when performing concurrent operations on it [16, 17]. Conventional data exploration analysis and methods using a specific software to find crucial information for use in various geospatial applications can be computationally expensive [18–20]. They cannot be possible in every case without having special methods to support the processing of big geospatial data [6, 21].

NoSQL or document databases provide much more flexibility in retrieving and inserting geospatial data than key-value databases. Using the geoJSON format, many document databases easily support geo-data management. Due to the flexible nature of NoSQL databases, they can be more efficient in performing geospatial data queries. One of NoSQL databases' shortcomings is that they do not provide any functions other than the basic spatial functions, lesser than relational databases. However, this approach leads to the usage of benefits of RDBMS, such as strong relational mappings, ACID properties, and strong foreign key constraints [22, 23]. Distinct characteristics of spatial data such as high dimensionality, several complex dependencies between entities on each other (e.g., distance entity, the dimension of direction, and geometrical relationships) leads to the requirement for time-consuming operations, and computationally exhausting algorithms for performing operations. The geospatial data in the cloud can provide a suitable efficient computation architecture that can support the processing of such huge data [24, 25].

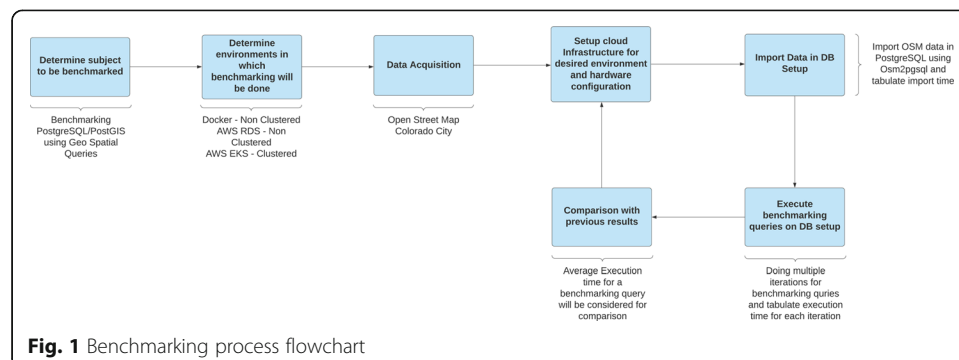
3 Methodology

This section defines the experimental setup and execution of the benchmarking process for GeoDatabase deployed in a clustered and non-clustered environment. The steps are shown in Fig. 1.

3.1 Subject to be benchmarked

PostgreSQL has been chosen as the subject for the benchmarking process. It is an open-source software program that adds support for geographic objects to its object-relational database using a PostGIS spatial database extender. This allows location queries to be run in SQL. The easy installation process across platforms turns out to be a good fit for a GeoDatabase that we can use as a subject in clustered and non-clustered environments.

A spatial query is a special type of database query that is supported by spatial geodatabases. These queries allow the use of geometry data types such as points, lines, and



polygons. They also consider the spatial relationship between these geometries. The spatial queries' execution time is used as a parameter for comparing performance in our benchmarking process.

3.2 Execution environments

The execution environments chosen for the selected geo-database are powered by Amazon Web Services (AWS) and are as follows:

1. PostgreSQL on Amazon Elastic Compute Cloud (AWS EC2)
2. PostgreSQL on Amazon Relational Database Service (AWS RDS)
3. PostgreSQL on Amazon Elastic Kubernetes Service (AWS EKS)

All the execution environments operate on two hardware configurations:

1. Hardware Configuration 1 (HC-1)
2. Hardware Configuration 2 (HC-2)

These hardware configurations differ in the allocated random-access memory (RAM), virtual central processing units (vCPUs).

Uniform hardware configuration is the key ingredient to make a benchmarking process fair for all the execution environments. PgAdmin is used as the monitoring tool to get all the benchmarking results for the spatial queries.

The first execution environment (AWS EC2) depicts how a student or researcher would set up a project database. Setting up a virtual machine on-premise or on-cloud and running the database on it is the simplest of all the available options. However, there is an overhead of manually scaling the database according to the incoming requests.

The second execution environment (AWS RDS) depicts the scenario of how a startup or any organization in the software industry would like to set up and manage their databases for all their projects. Relying on third-party services such as RDS or any other database-service provider takes off the load of managing and maintaining the setup. As these options provide less flexibility in scaling options and limited architectural control, they do not prove to be cost-effective.

The third execution environment (AWS EKS) depicts the scenario of a database running in a clustered environment that provides flexible scaling options, full architectural control, and good fail-over support.

3.2.1 PostgreSQL on Amazon Elastic Compute Cloud (AWS EC2)

The base of the environment is Amazon EC2 instance. This environment is used with two hardware configurations:

1. HC-1
 - a. Instance type–t2.Medium
 - b. RAM–4 GB
 - c. vCPUs–2

- d. Storage–8 GB general-purpose solid-state drive (SSD)
- 2. HC-2
 - a. Instance type–t3.Large
 - b. RAM–8 GB
 - c. vCPUs–2
 - d. Storage–8 GB general-purpose SSD

On the EC2 instance, docker and docker-compose are installed. The docker images of “mdillon/postgis:9.5-alpine” to setup PostgreSQL with PostGIS and “dpage/pgadmin4:latest” to setup PgAdmin using docker-compose on the Amazon EC2 instance are utilized.

3.2.2 PostgreSQL on Amazon Relational Database Service (AWS RDS)

The base of the environment is Amazon EC2 instance. This environment is used with two hardware configurations:

- 3. HC-1
 - a. EC2 instance type–t2.Medium
 - b. EC2 vCPUs–2
 - c. EC2 storage–8 GB general-purpose SSD
 - d. Database instance type–db.m3.medium
 - e. Database RAM–4 GB
 - f. Database vCPUs–1
 - g. Database capacity–20 GB SSD
- 4. HC-2
 - a. EC2 instance type–t3.Large
 - b. EC2 vCPUs–2
 - c. EC2 storage–8 GB general-purpose SSD
 - d. Database instance type–db.m5.large
 - e. Database RAM–8 GB
 - f. Database vCPUs–2
 - g. Database capacity–20 GB SSD

Docker and docker-compose are installed on the instance the docker image “dpage/pgadmin4:latest” is used to set up PgAdmin using docker-compose on the Amazon EC2 instance. PgAdmin is connected with the Amazon RDS instance.

3.2.3 PostgreSQL on Amazon Elastic Kubernetes Service (AWS EKS)

The base of the environment is Amazon EKS cluster with a node group attached with two hardware configurations:

- 5. HC-1
 - a. Node group instance type–t2.Medium
 - b. RAM–4 GB
 - c. vCPUs–2

- d. Storage—20 GB SSD
- 6. HC-2
 - a. Instance type—t3.Large
 - b. RAM—8 GB
 - c. vCPUs—2
 - d. Storage—20 GB SSD

On the EKS cluster, PostgreSQL and PGAdmin are deployed for benchmarking purposes using docker images “mdillon/postgis:9.5-alpine” and “dpage/pgadmin4:latest”.

3.3 Custom Kubernetes setup

There are cases where spinning up an AWS EKS instance can be very costly and may not be useful for research or for testing purposes. In that case, it is recommended building your own Kubernetes cluster, which can be done either on the cloud, or on the local machine(s). This kind of setup can greatly reduce the cost and enable researchers and students to set up their own distributed environment quickly and easily. The authors aim to build an easy to set up a heterogeneous clustered environment that can connect to different types of machines in different environments. Providing a methodology to set up a production-like environment quickly and easily can help greatly in validating conceptual architectures. This architecture needs to be cost-effective, flexible, and scalable at the same time. Kubernetes clusters can also provide certain benefits such as the following:

- (i) Load balancing—a methodical and efficient distribution of network or application traffic across multiple servers in a server farm. Each load balancer handles between client devices and backend servers, receiving and then distributing incoming requests to any available server capable of fulfilling them.
- (ii) Failover support—it ensures that a business intelligence system remains available for use if an application or hardware failure occurs. Clustering provides failover support in two ways: load redistribution and request recovery. The purpose of developing high-performance database clusters is to produce high performing computer systems. They operate co-extending programs that are needed for time-exhaustive computations. The scientific industries commonly prefer such a variety of clusters. The basic aim is intelligently sharing the workload.
- (iii) Monitoring and automation—clustering allows automating a lot of the processes of the database while it permits to set up rules to warn potential issues.

This installation process of the custom Kubernetes cluster has a lot of management overhead from the user’s perspective but provides desirable performance especially on lower configuration systems. This setup enables small-scale use cases to deploy and validate conceptual architectures for much less costs as compared to AWS EKS with slightly comparable performance. This setup can be created either by using KIND (Kubernetes in a docker) for test use cases or using Kubeadm to setup a master-agent configuration. One important point to note while setting up is that all the machines

should be on the same network or should be able to discover each other in order to connect and operate as a cluster.

3.4 Data acquisition

Geospatial data is used for benchmarking, since retrieving and fetching data can be a very resource intensive task and may provide us better and more accurate results since such resource-intensive tasks portray a more accurate description of deploying of databases in the real world. The choice of database for benchmarking is PostgreSQL since one of the biggest benefits of running PostgreSQL is running the cluster in primary-replica setup for the purposes of high-availability or load balancing the read-only queries. It is not necessarily simple to deploy a primary-replica setup out of the box, but the process can be simplified by using modern containerization technology. PostgreSQL provides the flexibility and the granular control to deploy the database in the desired and most effective configuration while having great tooling and support.

In this context, the geospatial data can be described by the atomic unit of a *feature*. A feature is a geographic *shape* (e.g., point, line string, 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, and the year it was built. Considering the map data of Colorado and Washington states of USA, provided by OpenStreetMap (OSM). The data when downloaded initially is in the file format **.osm.pbf*, which is a few hundred megabytes. This file format cannot be directly imported in PostgreSQL and hence it needs to be transformed first.

Osm2pgsql package available as a cli on ubuntu repository is an open-source tool to import the **.osm.pbf* file into the PostgreSQL database. *Osm2pgsql* is software to import OpenStreetMap data into a PostgreSQL database that has PostGIS extension installed already before import. It is an essential part of many rendering tool chains. The following are the stages of the process of importing OSM data into PostgreSQL:

1. Reading **.osm.pbf* file using PBF parser
2. Sorting of data and creation of index

The time taken to import OSM data depends on the following:

1. Hardware specifications of the machine where *Osm2pgsql* is running
2. The network bandwidth—to be able to share transformed data with PostgreSQL
3. The target database specifications—to sort data and create indices

Thus, using a separate Amazon Elastic Compute Cloud (EC2) instance within the same Virtual Private Cloud (VPC) as that of the desired execution environment was considered. The instance for a given hardware configuration has *Osm2pgsql* installed for importing OSM data in PostgreSQL for the 3 execution

environment. With this, the import time depends solely on the database running in the desired execution environment.

Based upon the OSM data, 8 geospatial queries listed in Table 1 were used for the benchmarking process. All spatial-queries are SELECT operations on the geo-database, and every query represents a real-world use-case where clients want to perform read queries from a geospatial web service. As updates to geo-data are less frequent than read operations, so considering read-queries enables benchmarking of the geo-database deployment performance in a real-world scenario.

3.5 Iterative benchmarking

The prerequisite for beginning benchmarking for the desired state is to have the infrastructure setup as described by the corresponding architecture diagrams (refer to Figs. 2, 3, and 4).

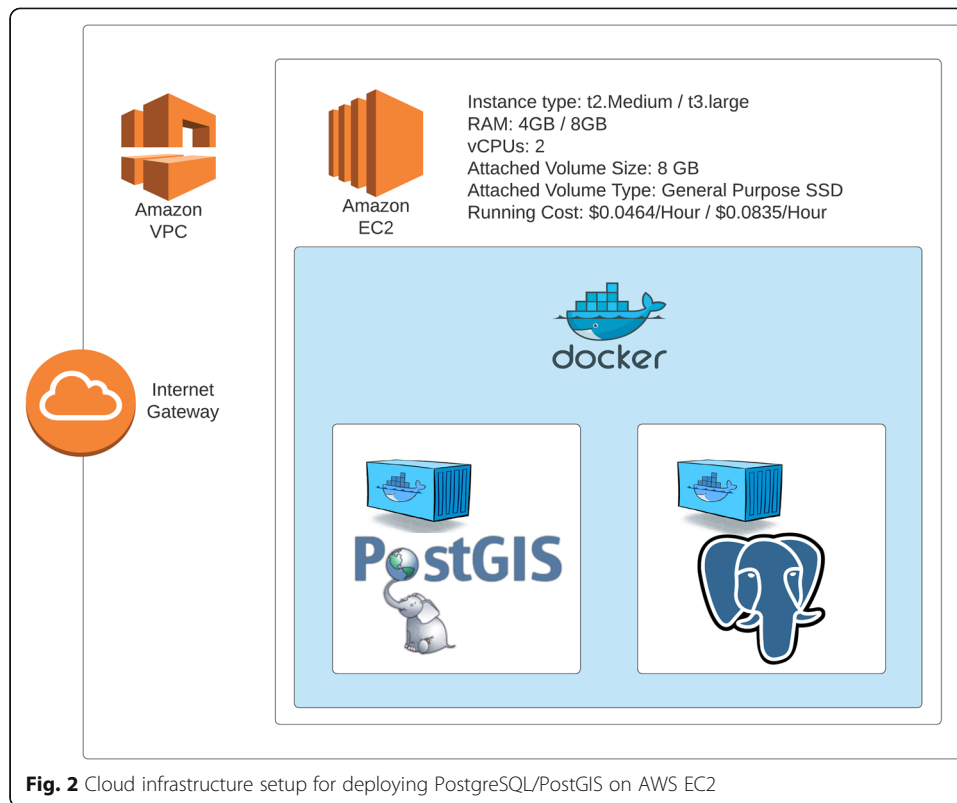
After infrastructure is set up, there will be a PostgreSQL database with PostGIS extension installed. Then, OSM data is imported in PostgreSQL running in the desired state. After the import is complete, *Osm2pgsql* gives the total time taken to import OSM data in PostgreSQL.

When the import process is complete, the next step is to execute the benchmarking queries. The benchmarking queries described in Table 1 are run using PgAdmin Query Tool, a robust, feature-rich environment that allows the execution of arbitrary SQL commands and retrieves the result set along with the execution time for each SQL query. Every benchmarking query is run in 10 iterations and execution time for each iteration is tabulated. After all iterations for all the benchmarking queries are done, the average execution time for every benchmarking query is calculated using the execution time for 10 iterations obtained. This average execution time obtained at the end of the process is considered the parameter of comparison of performance between the AWS EC2, AWS RDS, and AWS EKS.

The average execution time (AET) for each benchmarking query is calculated by taking the average of all its iterations in a particular execution environment. This average execution time (AET) of all the benchmarking queries is then used to compare the execution environments' performance under consideration. In the experiment, the AET in one environment is considered to be comparable with

Table 1 Benchmarking queries

Query ID	Description
Q1	Get count of nodes, ways, roads
Q2	Get count of points, lines, polygons
Q3	Top amenities
Q4	Get names of all restaurants and number of branches
Q5	Get all info about all the restaurants
Q6	Restaurants with 2 or more branches
Q7	Length of all roads (in km)
Q8	Generated objects and cardinalities



the AET in other environment only if the percentage difference in AET (positive or negative) between the two is significant, i.e., greater than 10%, because of multiple reasons which are not directly linked with the database like network latency, bandwidth available to transfer data which can affect the statistics. If the percentage improvement in AET is less than 10% then the AET of both the environments are considered to be similar.

We have represented the percentage improvement in AET in environment A with respect to AET in environment B as PI_{AB} which is calculated using Eqs. 1, 2, and 3:

$$AET_A = \text{Average Execution Time in environment A} \quad (1)$$

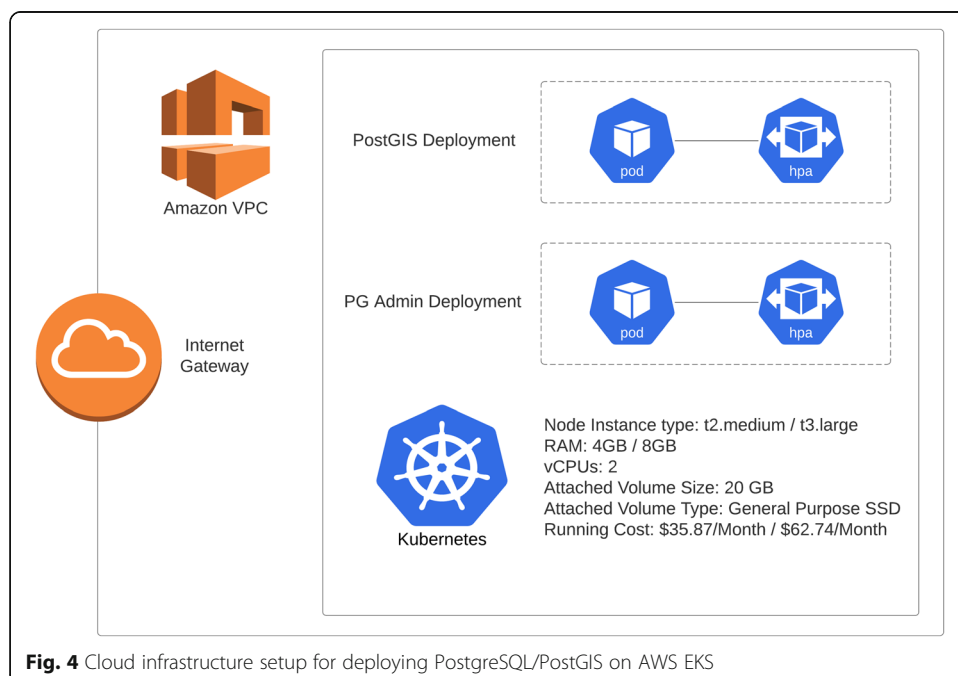
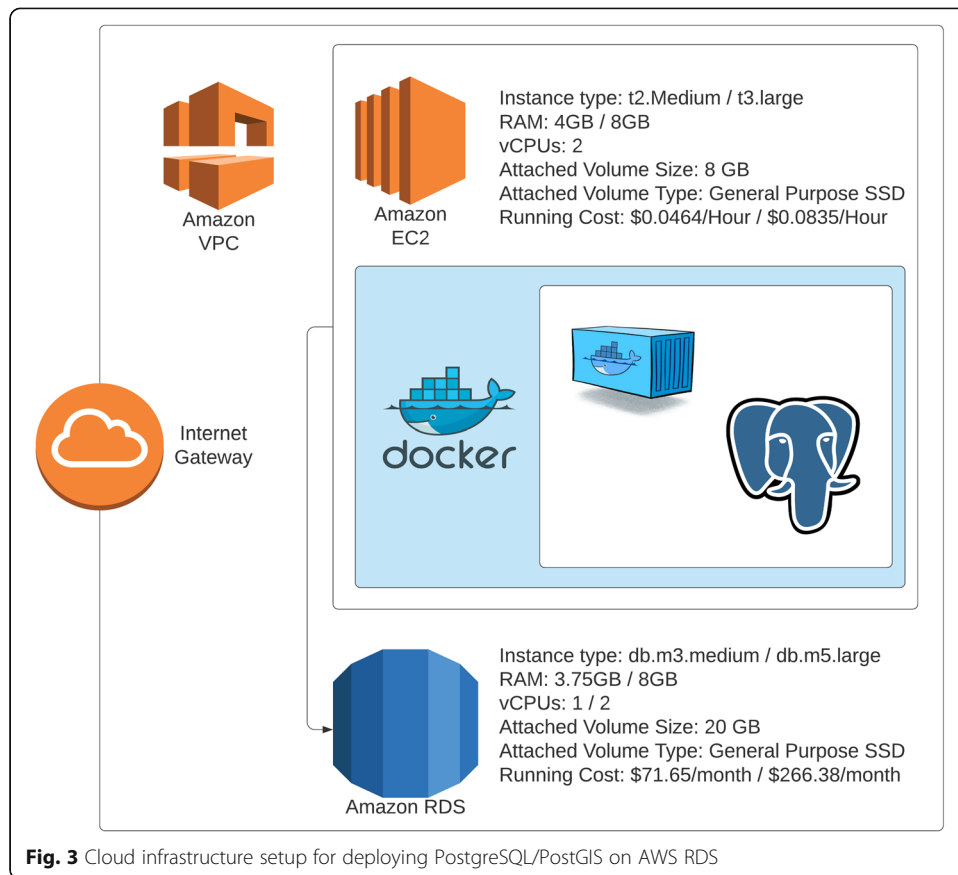
$$AET_B = \text{Average Execution Time in environment B} \quad (2)$$

$$\text{Percentage improvement in AET} = PI_{AB} = \frac{AET_B - AET_A}{AET_B} \times 100 \quad (3)$$

If PI_{AB} is positive, then the AET in environment A has improved by PI_{AB} percent compared with AET in environment B for a given benchmarking query. If PI_{AB} is negative, then the AET in environment A has degraded by PI_{AB} percent compared with AET in environment B for a given benchmarking query.

4 Experiments and results analysis

Following the methodology, we will compare our considered execution environments based on total time taken to import the data into PostgreSQL/PostGIS and the average execution times of benchmarking queries in the given environment. While several



factors might lead to these results, we chose to focus on the hardware configuration, resource usage, and the deployment architecture as the root cause of the results.

4.1 Benchmarking on import time

When downloaded in the file format *.osm.pbf, the OpenStreetMap data of the Colorado and Washington states cannot be directly imported in PostgreSQL and hence is transformed using *Osm2pgsql* CLI. *Osm2pgsql* also logs the timestamp corresponding to each stage of the import process described in Section 3.4. Import time is the summation of the time taken by each step in the import process.

The hardware specifications for the virtual machine where *Osm2pgsql* operates are the same for all the execution environments. The import time depends solely on the database's ability to run in the desired execution environment to create relations, insert, and index data.

Creation of tables, sorting, and indexing of geospatial data are computationally expensive operations. Table 2 shows the tabulated import times for the OSM data corresponding to Colorado and Washington states in all execution environments for both the considered hardware configurations. We observed that the import was quickest for databases operating in AWS EKS, because of its ability to scale up or down based on resource usage while the database operating in AWS EC2 took longer time to complete the import process as there was no scaling ability. Import time for AWS RDS varied greatly compared with the other two execution environments, because AWS RDS is not optimized to work with geospatial data. Certain extensions for PostGIS support are not compatible with it. The custom Kubernetes setup took similar time for the import process as AWS EKS, as it is also a clustered environment with an ability to scale on demand.

Table 2 Import time for considered execution environments

Execution environment	Architecture type	OSM data	Hardware configuration	Import time (s)
Amazon EC2	Non-clustered	Colorado	HC-1	796
			HC-2	651
		Washington	HC-1	765
			HC-2	707
Amazon RDS	Non-clustered	Colorado	HC-1	4956
			HC-2	1295
		Washington	HC-1	4914
			HC-2	1248
Amazon EKS	Clustered	Colorado	HC-1	720
			HC-2	570
		Washington	HC-1	698
			HC-2	608
Custom Kubernetes	Clustered	Colorado	HC-2	595
		Washington	HC-2	655

Table 3 Benchmarking queries

Query identity	Description	Number of rows fetched for Colorado	Number of rows fetched for Washington	SQL syntax
Q1	Get count of nodes, ways, roads, and rels.	4	4	SELECT 'nodes' AS tbl, COUNT(*) AS cnt FROM public.planet_osm_point UNION SELECT 'roads', COUNT(*) FROM public.planet_osm_line UNION SELECT 'ways', COUNT(*) FROM public.planet_osm_polygon UNION SELECT 'rels', COUNT(*) FROM public.planet_osm_polygon;
Q2	Get count of points, lines, and polygons	3	3	SELECT 'point' AS tbl, COUNT(*) AS cnt FROM public.planet_osm_point UNION SELECT 'line', COUNT(*) FROM public.planet_osm_line UNION SELECT 'polygon', COUNT(*) FROM public.planet_osm_polygon ;
Q3	Top amenities	168	210	SELECT amenity, COUNT(amenity) as num FROM planet_osm_point GROUP BY amenity ORDER by num DESC;
Q4	Get names of all restaurants and number of branches	2501	3794	SELECT name, count(name) FROM planet_osm_point WHERE amenity = 'restaurant' GROUP BY name ORDER BY count DESC;
Q5	Get all info about all the restaurants	3005	4321	SELECT * FROM planet_osm_point WHERE amenity = 'restaurant';
Q6	Get restaurants with 2 or more branches	178	84	SELECT name, count(name) as number FROM planet_osm_point WHERE amenity = 'restaurant' GROUP BY name HAVING count(name) >= 3 ORDER BY name ASC;
Q7	Get length of all roads (in km)	386224	494877	SELECT highway, name, way, st_length(way)/1000 AS length FROM planet_osm_line WHERE highway NOT IN ('construction', 'footway', 'path', 'steps', 'track', 'cycleway', 'pedestrian', 'abandoned', 'disused') AND (service NOT IN ('parking_aisle', 'driveway') OR service is null) AND (access NOT IN ('no', 'private') or access is null) ORDER BY name;
Q8	Get generated objects and cardinalities	8	8	SELECT *, pg_size_pretty(total_bytes) AS total , pg_size_pretty(index_bytes) AS INDEX , pg_size_pretty(table_bytes) AS TABLE FROM (SELECT *, total_bytes-index_bytes-COALESCE(toast_bytes,0) AS table_bytes FROM (SELECT c.oid, nspname AS table_schema, relname AS TABLE_NAME , c.reltuples AS row_estimate , pg_total_relation_size(c.oid) AS total_bytes , pg_indexes_size(c.oid) AS index_bytes , pg_total_relation_size(reltoastrelid) AS toast_bytes FROM pg_class c LEFT JOIN pg_namespace n ON n.oid = c.relnamespace WHERE relkind = 'r') a) a where a.table_schema = 'public';

4.2 Benchmarking on queries

Table 3 describes the geospatial queries that we have considered for benchmarking. These queries are executed on the imported OpenStreetMap data in each execution environment running on both the considered hardware configurations using PgAdmin.

In above-mentioned queries, the attributes on which the geo-data is indexed in PostgreSQL are *nodes*, *roads*, *ways*, *rels*, *point*, *line*, and *polygon*.

Tables 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, and 15 describe the execution time (in seconds) for 10 iterations (ET-1 to ET-10) of each query (Q1–Q8) in each of our

Table 4 Query execution time for Amazon EC2 with Colorado state data in HC-1

	ET-1	ET-2	ET-3	ET-4	ET-5	ET-6	ET-7	ET-8	ET-9	ET-10	AET
Q1	1.542	1.744	1.505	1.365	1.423	1.51	1.485	1.819	1.395	1.428	1.5216
Q2	1.698	1.81	1.184	0.995	1.06	1.17	1.63	0.852	1.718	0.949	1.3066
Q3	1.225	1.25	1.552	0.945	0.886	0.84	1.45	1.39	0.979	1.455	1.1972
Q4	0.89	0.852	0.709	0.777	0.799	0.813	0.76	0.86	0.745	0.95	0.8148
Q5	2.354	2.8	2.89	2.43	2.612	2.543	2.44	3.12	2.6	2.74	2.6529
Q6	1.11	1.52	0.984	1.16	1.2	1.41	1.12	0.789	1.26	1.372	1.1925
Q7	6.61	6.725	7.62	6.74	7.66	6.11	5.724	6.9	5.98	7.98	6.8049
Q8	0.929	1.23	1.122	0.929	0.883	1.128	0.998	0.985	1.124	0.976	1.0304

execution environments. Here, “ET” refers to “execution time,” “AET” refers to “average execution time” which is the average of ET-1 to ET-10 and “ET-*i*” refers to the “execution time for the *i*-th iteration for a given query.”

4.2.1 Queries operating on indexed attributes

Q1 and Q2 are geospatial queries which operate on indexed attributes and retrieve less than 5 rows. From Tables 4, 5, 6, 7, 8, 9, 10, and 11, we observe that AWS EC2 and AWS EKS gave similar AET for them as these attributes were indexed during the import process so there is less processing overhead because of efficient retrieval of data from the geo-database based on these attributes. But AWS RDS proved to be slower in operating on indices because it is not fully compatible to operate with PostGIS. AWS RDS gave slower AET for HC-1 for both Colorado and Washington states, but when compute resources were upgraded to HC-2, AWS RDS gave comparable AET to AWS EKS for Q1 and Q2.

4.2.2 Queries operating on non-indexed attributes

Tables 4 and 5 describe the AETs that are observed by running the queries on in AWS EC2 for HC-1 as shown in Fig. 2, for Colorado State OSM data and Washington State OSM data respectively.

Computation overhead involved in Q4 is less as compared to Q3 where aggregate functions are being used. Moderate computational overhead is introduced in Q6 using

Table 5 Query execution time for Amazon EC2 with Washington State data in HC-1

	ET-1	ET-2	ET-3	ET-4	ET-5	ET-6	ET-7	ET-8	ET-9	ET-10	AET
Q1	1.173	1.205	1.169	1.288	1.175	1.162	1.181	1.2	1.186	1.183	1.1922
Q2	0.821	0.814	0.817	0.825	0.82	0.81	0.829	0.818	0.822	0.815	0.8191
Q3	0.652	0.68	0.665	0.654	0.658	0.648	0.662	0.661	0.658	0.649	0.6587
Q4	0.599	0.621	0.607	0.613	0.6	0.618	0.617	0.627	0.623	0.618	0.6143
Q5	1.559	1.532	1.562	1.574	1.601	1.499	1.498	1.522	1.702	1.561	1.561
Q6	0.582	0.569	0.521	0.511	0.519	0.544	0.531	0.501	0.517	0.574	0.5369
Q7	7.821	8.236	7.012	7.899	7.211	7.134	8.798	8.125	7.184	7.234	7.6654
Q8	1.014	1.122	0.899	0.971	0.952	1.101	1.007	1.231	0.856	1.12	1.0273

Table 6 Query execution time for Amazon EC2 with Colorado State data in HC-2

	ET-1	ET-2	ET-3	ET-4	ET-5	ET-6	ET-7	ET-8	ET-9	ET-10	AET
Q1	1.176	1.173	1.182	1.161	1.153	1.176	1.185	1.186	1.181	1.203	1.1776
Q2	0.808	0.801	0.811	0.81	0.831	0.809	0.813	0.814	0.803	0.805	0.8105
Q3	0.772	0.696	0.685	0.684	0.694	0.685	0.719	0.703	0.695	0.699	0.7032
Q4	0.625	0.62	0.617	0.609	0.61	0.67	0.614	0.605	0.609	0.606	0.6185
Q5	1.738	1.75	1.552	1.754	1.546	1.553	1.31	1.359	1.342	1.392	1.5296
Q6	0.607	0.662	0.594	0.599	0.598	0.609	0.606	0.597	0.655	0.596	0.6123
Q7	4.365	5.07	4.826	4.279	4.334	4.579	4.75	4.186	4.721	4.303	4.5413
Q8	0.56	0.522	0.523	0.527	0.533	0.538	0.531	0.503	0.527	0.505	0.5269

HAVING clause along with count and grouping operation on an indexed attribute. Q4 and Q6 look similar but in Tables 4 and 5, their average execution times differ by a significant margin of 377.7 ms and Q4 being the quickest to execute, this is because in Q6 additional computations are being performed to get result corresponding to the HAVING clause which requires an additional traversal of the resultant rows after grouping operation. Q7 is a computationally expensive query calculating the length of all roads in the city using ST_LENGTH PostGIS function and retrieves 386224 rows and we see that it took maximum time to execute.

From Table 5, similar observations can be seen for Washington State OSM data. Q7 being the slowest query to execute and even slower than Q7 for Colorado State OSM data from Table 4, this is because Washington State has more roads compared to Colorado State, and this can also be seen from Table 3, where Q7 fetch 494k rows for Washington while 386k rows for Colorado. Queries with low computational overhead like Q1, Q2, Q3, and Q4 gave less AET for Washington compared to Colorado from Tables 4 and 5, because Washington State OSM data is smaller in size compared to Colorado State OSM data making it easier to operate on.

Tables 6 and 7 describe the AET's that are observed by running the queries on in AWS EC2 for HC-2 as shown in Fig. 2, for Colorado State OSM data and Washington State OSM data respectively.

In Table 6, on upgrading the hardware configuration of AWS EC2 from HC-1 to HC-2, all the benchmarking queries saw improvement in AET compared to Table 4.

Table 7 Query execution time for Amazon EC2 with Washington State data in HC-2

	ET-1	ET-2	ET-3	ET-4	ET-5	ET-6	ET-7	ET-8	ET-9	ET-10	AET
Q1	1.191	1.217	1.172	1.17	1.166	1.167	1.17	1.163	1.117	1.154	1.1687
Q2	0.791	0.787	0.794	0.789	0.806	0.788	0.793	0.787	0.802	0.798	0.7935
Q3	0.657	0.648	0.645	0.647	0.652	0.655	0.653	0.692	0.649	0.65	0.6548
Q4	0.596	0.598	0.601	0.602	0.596	0.669	0.594	0.602	0.596	0.597	0.6051
Q5	1.534	1.505	1.497	1.582	1.545	1.527	1.511	1.538	1.507	1.518	1.5264
Q6	0.581	0.574	0.586	0.585	0.584	0.587	0.58	0.585	0.572	0.582	0.5816
Q7	5.486	5.459	5.401	5.342	5.023	5.301	5.003	5.388	5.083	5.187	5.2673
Q8	0.517	0.52	0.522	0.523	0.511	0.532	0.512	0.545	0.514	0.526	0.5222

Table 8 Query execution time for Amazon RDS with Colorado State data for HC-1

	ET-1	ET-2	ET-3	ET-4	ET-5	ET-6	ET-7	ET-8	ET-9	ET-10	AET
Q1	2.162	2.163	2.142	2.1	2.185	2.113	2.11	2.472	2.404	2.266	2.2117
Q2	1.731	1.723	1.6	1.879	1.697	1.865	1.629	1.795	1.66	1.894	1.7473
Q3	1.135	0.875	1.21	1.215	0.994	0.993	1.1	0.879	0.953	0.87	1.0224
Q4	0.956	0.85	0.868	0.832	0.981	0.829	0.908	0.95	0.891	0.865	0.893
Q5	1.754	1.836	1.675	1.78	1.654	1.913	1.83	1.546	1.824	1.52	1.7332
Q6	0.881	1.203	1.11	1.103	0.904	0.929	0.804	1.02	0.862	1.136	0.9952
Q7	11.1	13.085	13.603	13.458	11.377	13.264	13.574	12.587	11.98	13.413	12.7441
Q8	0.865	0.919	0.982	0.973	0.896	0.844	0.851	0.891	0.904	0.886	0.9011

The query with highest computational overhead Q7 saw an improvement of 2.2636 s in AET. While the queries involving moderate computation overhead like Q5, Q6 saw an improvement of 1.1233 s and 580 ms in AET respectively. Queries with low computational overhead like Q3, Q4, and Q8 saw an improvement of 200 ms, 494 ms, and 500 ms in AET respectively.

Table 7 shows that we recorded similar observations for Washington State OSM data. Q7 saw an improvement of 3.1241 s in AET, while queries with low computational overhead like Q3 and Q4 saw marginal improvements less than 10 ms in AET.

Tables 8 and 9 describe the AET's that are observed by running the queries on in AWS RDS for HC-1 as shown in Fig. 3, for Colorado State OSM data and Washington State OSM data respectively.

Q7 again was observed as the query requiring the maximum execution time. But in this case, we observe that the AET for all queries increased when compared with corresponding AET in AWS EC2 and AWS EKS from Tables 4 and 12. There can be multiple reasons for this behavior; certain PostGIS extensions required for installation and operation in PostgreSQL are not compatible with AWS RDS and are not optimized to deal with geospatial data.

From Table 9, it can be seen that similar observations were made for Washington State OSM data. Q7 being the slowest to execute again. All other queries involving moderate and low computational overhead gave better AET than Colorado OSM data from Table 8 because Washington State OSM data is smaller in size compared to Colorado State OSM data.

Table 9 Query execution time for Amazon RDS with Washington State data for HC-1

	ET-1	ET-2	ET-3	ET-4	ET-5	ET-6	ET-7	ET-8	ET-9	ET-10	AET
Q1	2.144	2.207	2.597	2.51	2.197	2.6	2.402	2.176	2.204	2.182	2.3219
Q2	1.667	1.642	1.646	1.694	1.654	1.609	1.641	1.78	1.643	1.645	1.6621
Q3	0.751	0.723	0.731	0.723	0.729	0.738	0.735	0.727	0.737	0.734	0.7328
Q4	0.728	0.663	0.671	0.687	0.665	0.684	0.672	0.675	0.661	0.673	0.6779
Q5	1.394	1.371	1.603	1.611	1.368	1.375	1.493	1.39	1.401	1.487	1.4493
Q6	0.764	0.651	0.672	0.665	0.663	0.643	0.709	0.646	0.655	0.65	0.6718
Q7	15.565	15.983	14.819	15.973	15.095	15.152	15.79	16.07	15.583	15.186	15.5216
Q8	0.535	0.521	0.519	0.529	0.53	0.54	0.526	0.531	0.523	0.537	0.5291

Table 10 Query execution time for RDS with Colorado State data for HC-2

	ET-1	ET-2	ET-3	ET-4	ET-5	ET-6	ET-7	ET-8	ET-9	ET-10	AET
Q1	0.85	0.787	0.796	0.798	0.797	0.795	0.799	0.801	0.806	0.796	0.8025
Q2	0.72	0.718	0.716	0.717	0.724	0.755	0.729	0.719	0.721	0.725	0.7244
Q3	0.739	0.653	0.656	0.659	0.702	0.66	0.662	0.712	0.658	0.684	0.6785
Q4	0.599	0.593	0.602	0.594	0.59	0.608	0.595	0.594	0.592	0.597	0.5964
Q5	1.524	1.496	1.512	1.527	1.535	1.53	1.488	1.522	1.496	1.497	1.5127
Q6	0.594	0.593	0.588	0.579	0.578	0.604	0.591	0.587	0.589	0.61	0.5913
Q7	5.437	5.196	5.18	5.281	5.438	5.692	5.191	5.842	5.236	5.484	5.3977
Q8	0.578	0.534	0.515	0.519	0.513	0.579	0.527	0.539	0.535	0.58	0.5419

Tables 10 and 11 describe the AET's that are observed by running the queries on in AWS RDS for HC-2 as shown in Fig. 3, for Colorado State OSM data and Washington State OSM data respectively.

In Table 10, on upgrading the hardware configuration of AWS RDS from HC-1 to HC-2, all the benchmarking queries saw improvement in AET compared to Table 8 similar to what we saw in case of AWS EC2 from Tables 4 and 6. The query with highest computational overhead Q7 saw an improvement of 57.64%, i.e., 7.3464 s in AET. While the queries involving moderate computation overhead like Q5, Q6 saw an improvement of 220 ms and 403.9 ms in AET respectively. Queries with low computational overhead like Q3, Q4, and Q8 saw an improvement of 343.9 ms, 296.6 ms, and 359.2 ms in AET. Significant improvements were observed for RDS on upgrading the hardware configuration.

AETs for queries operating on Washington State OSM data shown in Table 11, we recorded similar observations. Q7 saw an improvement of 57.18%, i.e., 8.88 s in AET, queries with moderate and low computational overhead like Q3, Q4, Q5, Q6, and Q8 saw marginal improvements, i.e., less than 10%.

Tables 12 and 13 describe the AETs that are observed by running the queries on in AWS EKS for HC-1 as shown in Fig. 4, for Colorado State OSM data and Washington State OSM data respectively.

Here also Q7 took maximum time to execute, but this time it was quicker to execute when compared with corresponding AET for AWS EC2 and AWS RDS in Tables 4 and 8. We also observed that the absolute difference between average execution time of

Table 11 Query execution time for RDS with Washington State data for HC-2

	ET-1	ET-2	ET-3	ET-4	ET-5	ET-6	ET-7	ET-8	ET-9	ET-10	AET
Q1	0.773	0.902	0.7	0.717	0.803	0.716	0.791	0.718	0.927	0.723	0.777
Q2	0.733	0.705	0.719	0.715	0.705	0.702	0.708	0.716	0.702	0.708	0.7113
Q3	0.747	0.643	0.626	0.644	0.618	0.621	0.672	0.623	0.64	0.63	0.6464
Q4	0.808	0.58	0.625	0.59	0.585	0.574	0.576	0.602	0.578	0.745	0.6263
Q5	1.468	1.697	1.477	1.479	1.493	1.48	1.63	1.684	1.678	1.486	1.5572
Q6	0.558	0.556	0.561	0.555	0.59	0.563	0.574	0.565	0.568	0.564	0.5654
Q7	7.161	6.548	6.342	6.508	6.805	6.787	6.198	6.449	6.86	6.801	6.6459
Q8	0.521	0.509	0.511	0.495	0.501	0.553	0.497	0.51	0.509	0.52	0.5126

Table 12 Query execution time for Amazon EKS with Colorado State data for HC-1

	ET-1	ET-2	ET-3	ET-4	ET-5	ET-6	ET-7	ET-8	ET-9	ET-10	AET
Q1	1.511	1.55	1.17	1.548	1.251	1.23	1.165	1.79	1.72	1.5	1.4435
Q2	0.983	1.135	0.923	1.133	1.422	1.61	0.908	1.76	1.157	1.215	1.2246
Q3	1.229	1.35	1.208	0.965	1.117	0.867	0.806	0.905	0.898	0.995	1.034
Q4	0.809	0.874	0.817	0.954	0.916	0.844	0.988	0.848	0.986	0.85	0.8886
Q5	2.22	1.987	1.669	1.55	1.174	1.826	1.951	1.272	1.282	2.183	1.7114
Q6	0.736	0.937	1.145	0.931	0.976	0.841	0.789	0.841	0.773	0.806	0.8775
Q7	5.481	5.987	4.935	4.915	5.235	5.8	5.71	5.56	4.413	5.13	5.3166
Q8	0.626	0.731	0.992	0.72	0.917	0.911	0.787	0.884	0.928	0.88	0.8376

Q4 and Q6 reduced to 11.1 ms when compared with 377.7 ms in Table 4 and 102.2 ms in Table 8, also their AET was comparable keeping the margin of error in mind. This improvement in performance results from AWS EKS to scale up and down on-demand based on resource usage.

For Washington State, OSM data similar records were observed as shown in Table 13. AET for all benchmarking queries were observed to have improved when compared to other execution environments like AWS EC2 from Table 5 and AWS RDS from Table 7.

In Table 14, on upgrading the hardware configuration of AWS EKS from HC-1 to HC-2, all the benchmarking queries saw improvement in AET compared to Table 12. The query with highest computational overhead Q7 saw an improvement of 18.87%, i.e., 1.0037 s in AET. Queries with low computational overhead like Q3, Q4, and Q8 saw an improvement of 411.1 ms, 327.2 ms, and 427.1 ms in AET. Significant improvements in AET for query involving high computational overhead were observed for EKS on upgrading the hardware configuration.

For Washington State OSM data from Table 15, we recorded similar observations. Q7 saw an improvement of 15.39%, i.e., 8.88 s in AET, queries with moderate computational overhead like Q5 and Q6 saw an improvement of 189.7 ms and 136.8 ms respectively.

Q3 operates on non-indexed attributes and uses aggregate functions on them to retrieve 168 rows, and it can also be considered to be a standard SQL text query; the retrieval has moderate computational overhead. From Tables 4, 5, 6, 7, 8, and 9, it can be

Table 13 Query execution time for Amazon EKS with Washington State data for HC-1

	ET-1	ET-2	ET-3	ET-4	ET-5	ET-6	ET-7	ET-8	ET-9	ET-10	AET
Q1	0.911	0.935	1.01	1.021	1.202	1.029	1.157	1.89	1.249	1.21	1.1614
Q2	0.767	0.809	0.731	0.727	0.798	0.748	0.807	0.768	0.813	0.815	0.7783
Q3	0.623	0.523	0.623	0.535	0.621	0.653	0.528	0.598	0.592	0.582	0.5878
Q4	0.809	0.524	0.871	0.609	0.621	0.814	0.621	0.521	0.536	0.801	0.6727
Q5	1.28	1.01	0.966	1.05	1.1	0.976	0.991	0.995	1.08	0.987	1.0435
Q6	0.463	0.473	0.415	0.413	0.467	0.414	0.456	0.481	0.472	0.468	0.4522
Q7	6.374	6.106	6.281	6.138	6.348	6.296	6.194	6.248	6.326	5.484	6.1795
Q8	0.621	0.701	0.921	0.718	0.92	0.909	0.698	0.852	0.919	0.821	0.808

Table 14 Query execution time for Amazon EKS with Colorado State data for HC-2

	ET-1	ET-2	ET-3	ET-4	ET-5	ET-6	ET-7	ET-8	ET-9	ET-10	AET
Q1	0.887	0.851	0.721	0.813	0.802	0.909	0.801	0.819	0.749	0.721	0.8073
Q2	0.669	0.612	0.603	0.727	0.739	0.642	0.601	0.768	0.603	0.81	0.6774
Q3	0.613	0.623	0.708	0.735	0.621	0.553	0.581	0.521	0.692	0.582	0.6229
Q4	0.509	0.574	0.619	0.512	0.607	0.611	0.618	0.509	0.506	0.549	0.5614
Q5	0.428	0.401	0.466	0.405	0.41	0.406	0.414	0.418	0.408	0.411	0.4167
Q6	0.346	0.347	0.316	0.314	0.346	0.314	0.346	0.318	0.318	0.312	0.3277
Q7	4.473	4.61	4.261	4.135	4.408	4.226	4.191	4.148	4.263	4.414	4.3129
Q8	0.412	0.41	0.415	0.418	0.42	0.409	0.406	0.414	0.401	0.4	0.4105

observed that AWS EKS because of its scaling ability and AWS RDS which is designed to work with text queries gave better AET than standard PostgreSQL on AWS EC2. For queries with high computational overhead like Q5 which traverses all the data in the planet_osm_point table based on non-indexed attribute and Q8 which traverses complete dataset to retrieve the information about the generated objects and cardinalities after the import process is completed, AWS EKS gave the best AET because of its ability to scale based on resource usage.

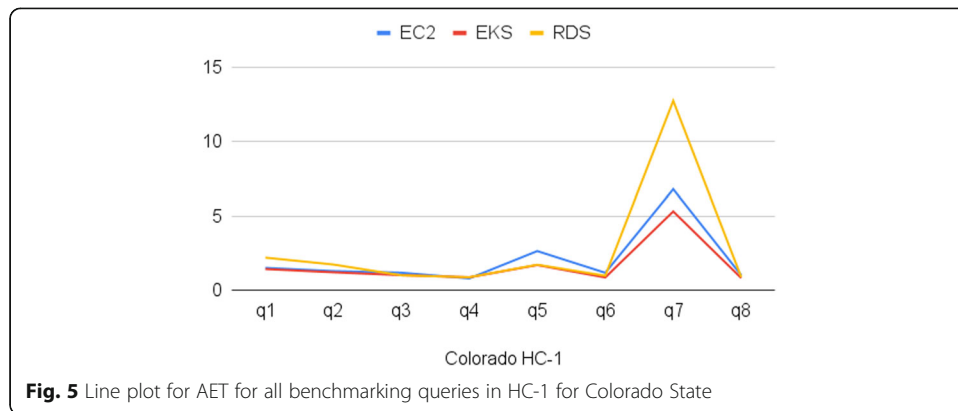
4.3 Benchmarking on AET

Figure 5 is a line plot showing the AET (y-axis) in seconds for all benchmarking queries Q1–Q8 (x-axis) operated for Colorado OSM data in AWS EC2 for HC-1. The plot is made using the AET values from Tables 4, 5, 6, 7, 8, and 9 to visualize the variation of AET for benchmarking queries in all execution environments.

The plot shows that for queries on indexed attributes Q1 and Q2, AWS EKS and AWS EC2 gave similar AET but AWS RDS deviated and gave slower AET. For a standard SQL text query Q3, AWS RDS performed similarly to AWS EKS and both outperformed standard PostgreSQL in AWS EC2. The absolute difference between the AET of query with lowest computational overhead Q4 and the query involving moderate overhead Q6 was observed to be minimum in case of AWS EKS, thus where AWS EC2 and AWS RDS deviated for moderate load, AWS EKS performed similarly

Table 15 Query execution time for Amazon EKS with Washington State data for HC-2

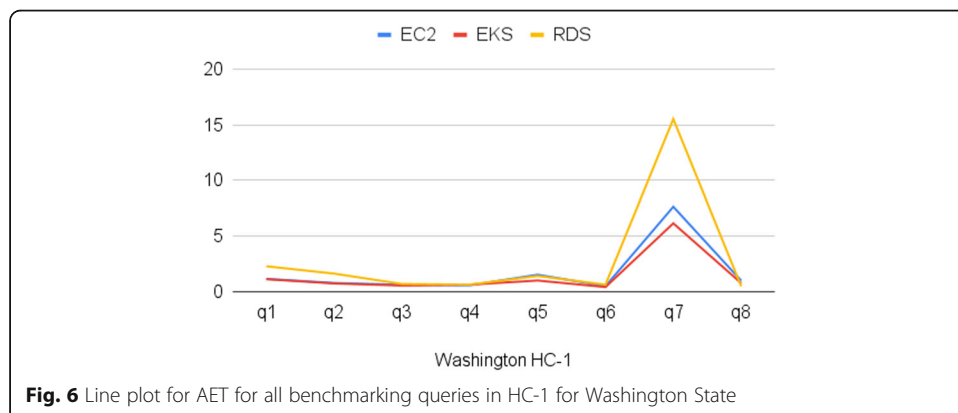
	ET-1	ET-2	ET-3	ET-4	ET-5	ET-6	ET-7	ET-8	ET-9	ET-10	AET
Q1	0.901	0.802	0.821	0.821	0.891	0.929	0.851	0.821	0.831	0.819	0.8487
Q2	0.727	0.729	0.781	0.721	0.728	0.781	0.606	0.781	0.713	0.715	0.7282
Q3	0.621	0.723	0.723	0.635	0.621	0.753	0.821	0.698	0.692	0.601	0.6888
Q4	0.721	0.724	0.628	0.609	0.621	0.714	0.621	0.723	0.676	0.621	0.6658
Q5	0.86	0.864	0.805	0.81	0.87	0.876	0.881	0.885	0.8	0.887	0.8538
Q6	0.325	0.317	0.315	0.317	0.314	0.316	0.312	0.319	0.311	0.308	0.3154
Q7	5.184	5.187	5.29	5.214	5.235	5.3	5.25	5.2	5.213	5.21	5.2283
Q8	0.412	0.41	0.42	0.418	0.419	0.415	0.412	0.408	0.414	0.417	0.4145

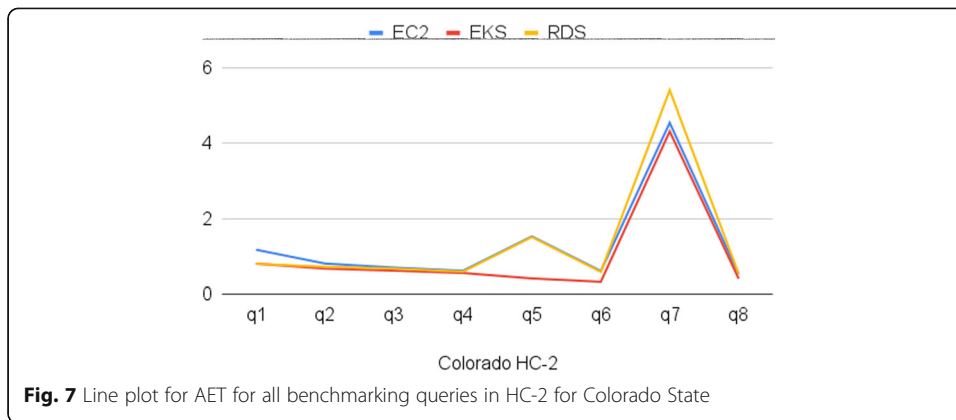


for low to moderate load. For Q7, which is using PostGIS function AWS RDS and AWS EC2 deviated by a significant margin from AWS EKS. For queries involving traversal of complete dataset like Q5 and Q8, AWS EKS and AWS RDS gave better AET when compared to AWS EC2. These observations are further explored in Section 4.4 using PI_{AB} .

Figure 6 is a line plot showing the AET (y-axis) in seconds for all benchmarking queries Q1–Q8 (x-axis) operated for Washington OSM data in all execution environments for HC-1. The plot is made using the AET values from Tables 5, 9, and 13. It can be observed from the plot that similar to the case of Colorado State, AWS EKS and AWS EC2 gave similar AET for Q1 and Q2 while AWS RDS deviated to give a slower AET. Here also AWS EKS gave similar AET for low to moderate load in Q4 and Q6. AWS EKS again gave the best AET for Q7, which was the slowest query to execute because of the highest computational overhead.

Figure 7 is a line plot showing the AET (y-axis) in seconds for all benchmarking queries Q1–Q8 (x-axis) operated for Colorado OSM data in all execution environments for HC-2. The plot is made using the AET values from Tables 6, 10, and 14. On upgrading the hardware configuration, we observed that the difference in AETs was within the margin of error when compared with AETs for Colorado State in HC-1 as depicted in Fig. 6, for queries involving low computational overhead. For such queries, AWS EKS outperformed AWS EC2 and AWS RDS but now by significant



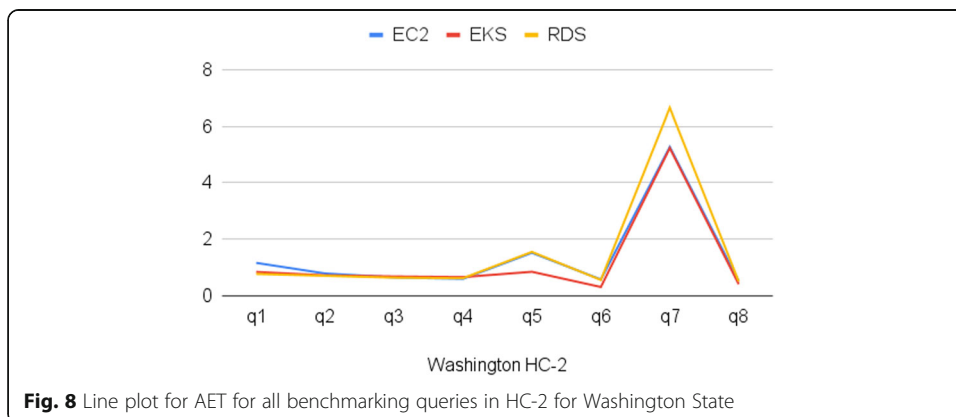


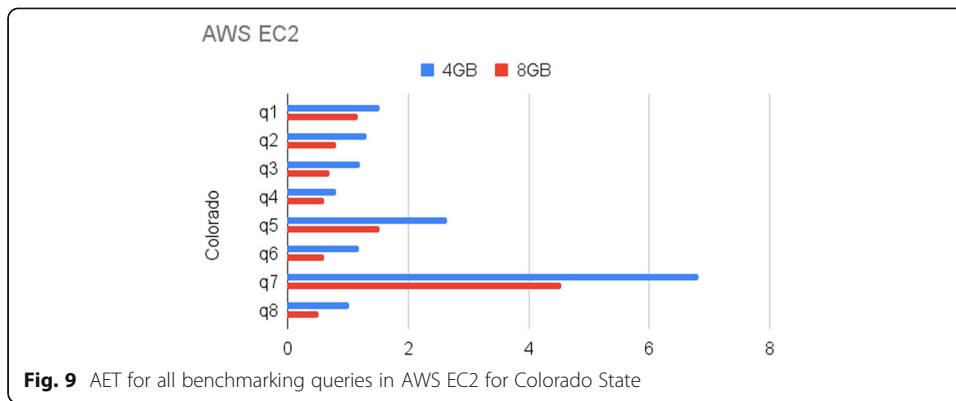
margin. Greater improvement in AET can be observed for queries involving moderate and high computational overhead, for which AWS EKS outperformed AWS EC2 and AWS RDS by significant margin, which points us to the fact that on increasing the compute resources, the performance or AET will improve for the queries which actually need that much resources.

Figure 8 is a line plot showing the AET (y-axis) in seconds for all benchmarking queries Q1–Q8 (x-axis) operated for Washington OSM data in all execution environments for HC-2. The plot is made using the AET values from Tables 7, 11, and 15. Similar to our observation from Fig. 7, the difference in AET between all execution environments became marginal for queries involving low computational overhead, but AWS EKS continued to yield better AET for queries involving moderate to high computational overhead.

Figs. 9, 10, 11, 12, 13, and 14 depict the improvements in AET for all benchmarking queries operating for both hardware configuration for a given execution environment. These plots can enable us to understand the effect of hardware configurations on AET for benchmarking queries in a given environment. Line plot for percentage improvement (PIAB) in AET, is shown in Fig. 15.

Figure 9 is a double-bar plot showing the AET (x-axis) in seconds for all benchmarking queries Q1–Q8 (y-axis) operated for Colorado State OSM data in AWS EC2 for both HC-1 and HC-2. The plot is made using the AET values from Tables 4 and 6. It can be observed that on increasing the resources the AET of the benchmarking queries



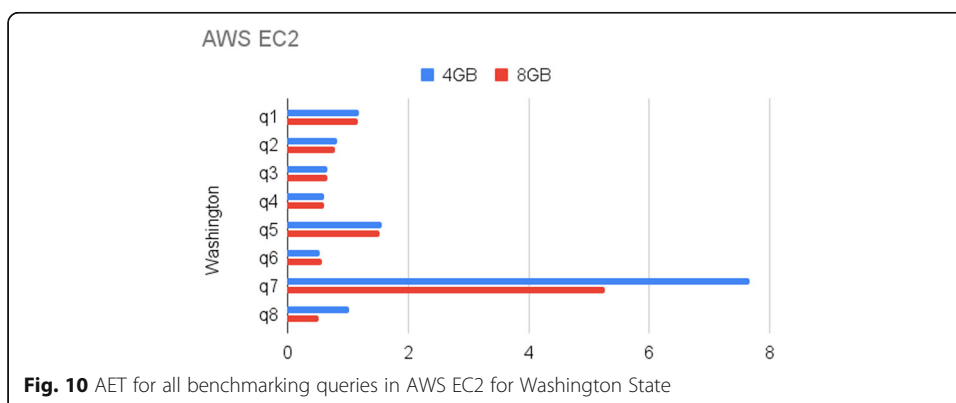


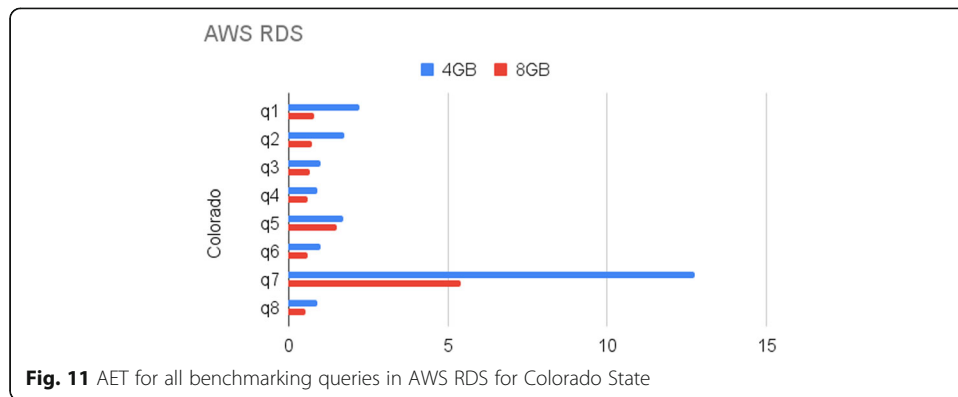
decreased. Queries such as Q5 and Q7 showed more improvement since they are moderate and high computationally intensive as compared to other queries and benefited from upgrading the compute resources.

Figure 10 is a double-bar plot showing the AET (x-axis) in seconds for all benchmarking queries Q1–Q8 (y-axis) operated for Washington OSM data in AWS EC2 for both HC-1 and HC-2. The plot is made using the AET values from Tables 5 and 7. The improvement in AETs for Q1, Q2, Q3, and Q4 is marginal because these queries are less resource intensive for Washington State. This trend is due to the fact that Washington State OSM data is smaller than Colorado State OSM data. Because of which the improvements were better in Colorado State than in Washington State.

Figure 11 is a double-bar plot showing the AET (x-axis) in seconds for all benchmarking queries Q1–Q8 (y-axis) operated for Colorado OSM data in AWS RDS for both HC-1 and HC-2. The plot is made using the AET values from Tables 8 and 10. We observed great improvements in AET for the benchmarking queries on upgrading the hardware configuration for AWS RDS. Hence, it can be said that AWS RDS requires more compute resources than other execution environments to deliver comparable results.

Figure 12 is a double-bar plot showing the AET (x-axis) in seconds for all benchmarking queries Q1–Q8 (y-axis) operated for Washington OSM data in AWS RDS for both HC-1 and HC-2. The plot is made using the AET values from Tables 9 and 11.





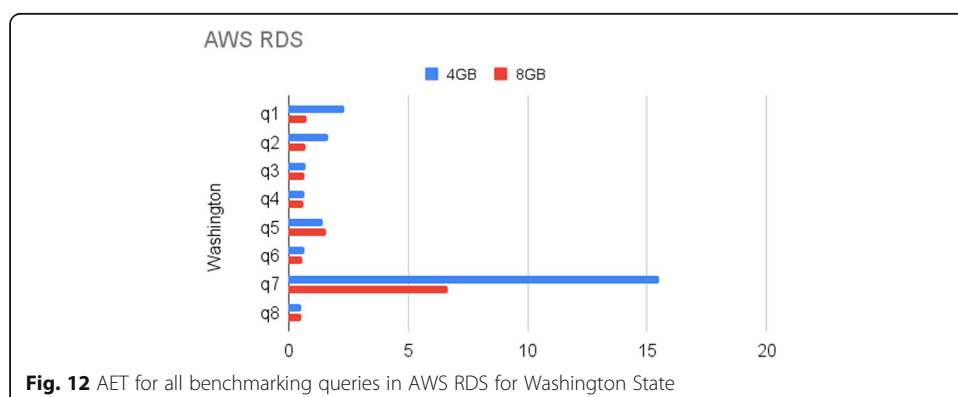
Similar to the observations from Fig. 11, the AET for benchmarking queries is improved significantly for Q7, but for queries involving low to moderate computational overhead the improvement was marginal.

Figure 13 is a double-bar plot showing the AET (x-axis) in seconds for all benchmarking queries Q1–Q8 (y-axis) operated for Colorado OSM data in AWS EKS for both HC-1 and HC-2. The plot is made using the AET values from Tables 12 and 14. AWS EKS when upgraded to HC-2 shows good improvement as compared to AWS EKS in HC-1. The AET improved for all the benchmarking queries.

Figure 14 is a double-bar plot showing the AET (x-axis) in seconds for all benchmarking queries Q1–Q8 (y-axis) operated for Washington OSM data in AWS EKS for both HC-1 and HC-2. The plot is made using the AET values from Tables 13 and 15. The exact same observation can be seen in Fig. 13, but Q1, Q2, Q3, and Q4 are queries involving low computational overhead for Washington State; hence, the improvement was not significant.

4.4 Comparison of execution environments based on PI_{AB}

We know that the Colorado State OSM data is larger in size than the Washington State OSM data; therefore, we considered comparing the performance of the benchmarking queries for Colorado State OSM data in all execution environments to



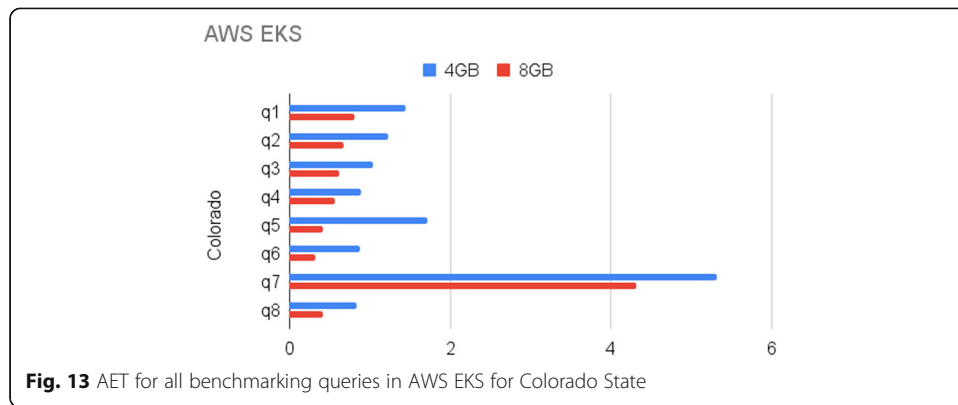


Fig. 13 AET for all benchmarking queries in AWS EKS for Colorado State

find out which one of them yields the best AETs for a lower hardware configuration (HC-1).

Table 16 represents percentage improvement in AET (PI_{AB}) among all the execution environments which is calculated using Eq. 3. AET of a benchmarking query in environment A is said to have improved relative to its AET in environment B if AET_A is less than AET_B or PI_{AB} is positive.

From Tables 10 and 11, we observed that AWS EKS when compared to AWS EC2 gave similar average execution times for Q1 and Q2 which operated on indexed attributes. Figure 6 shows the line plot for percentage improvement (PI_{AB}) in AET. They again were not very dissimilar for Q4, because of less computational overhead involved. But AWS EKS outperformed AWS EC2 when moderate computational overhead was introduced in Q6 and Q3, because of its ability to scale up and down based upon resource usage. Q5, Q7, and Q8 involved high computational overhead and AWS EKS continued to outperform AWS EC2 because of its ability to scale on demand flexibly. Certain PostgreSQL/PostGIS extensions are not compatible with AWS RDS, as a result of which AWS EKS outperformed it for Q1 and Q2 which operate on indexed attributes and Q7 which makes use of PostGIS function. They both yielded similar AET for Q3 and Q4, Q3 is similar to a standard SQL query and AWS RDS is designed to handle such text queries and Q4 involved low computational overhead. In the case of Q6, the scaling ability of AWS EKS made it outperform AWS

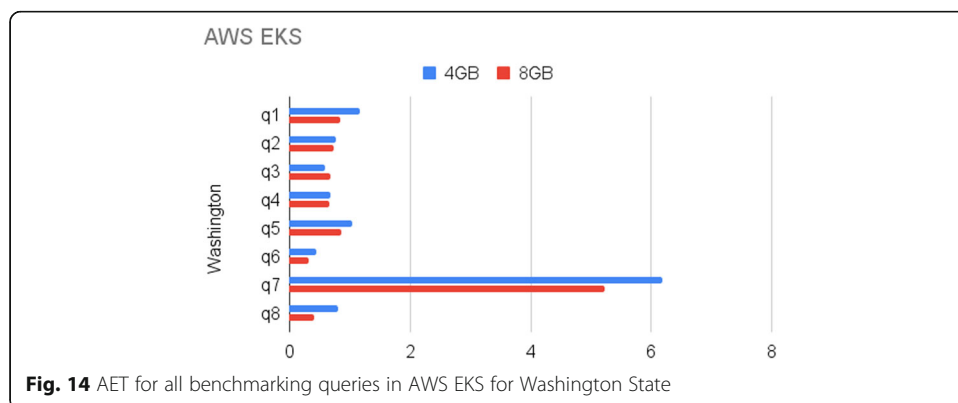
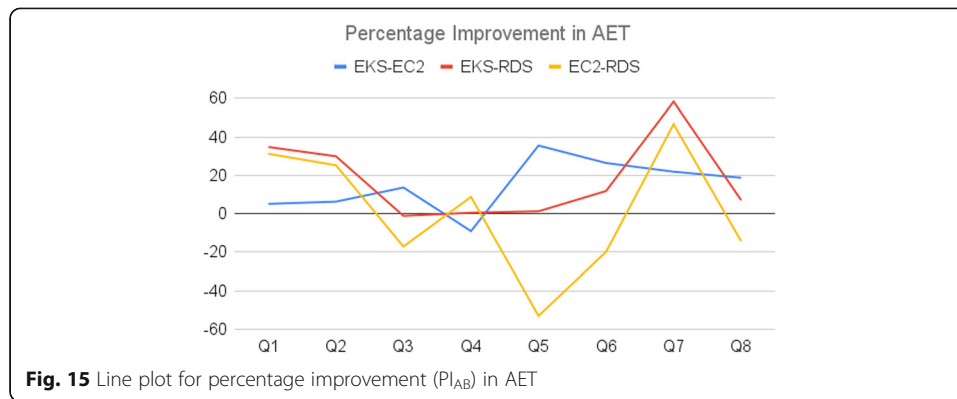


Fig. 14 AET for all benchmarking queries in AWS EKS for Washington State



RDS for moderate load. The deployment of AWS RDS shown in Fig. 3 does not support scaling as a result of which for queries Q5 and Q8 which involve high computational overhead because of complete traversal of tables and dataset, AWS EKS outperformed AWS RDS by fine margins. If AWS RDS would have been allotted a similar amount of RAM allocated to standard PostgreSQL in AWS EKS, these margins would increase.

AWS EC2 outperformed AWS RDS for Q1, Q2, and Q7, because of AWS RDS's non-compatibility for certain PostGIS extensions. AWS RDS is optimized to work with standard text queries like Q3 and thus it outperformed AWS EC2 running standard PostgreSQL. They both gave similar AET for Q4 because of low computational overhead involved. From Figs 2 and 3, we can see that AWS RDS has more available RAM to operate when compared to standard PostgreSQL on AWS EC2, because of which it outperformed AWS EC2 for Q5, Q6, and Q8 which involved moderate to high computational overhead.

4.5 Comparing AWS EKS with custom Kubernetes cluster

From Table 17, it can be seen that the custom Kubernetes cluster performed similar to AWS EKS from Table 15, since it is a similar clustered environment. AET for all the benchmarking queries are not significantly different to AWS EKS (slightly higher than EKS). This slightly increased time can arise due to the network latencies in load balancers created in the custom Kubernetes cluster.

Table 16 Comparing percentage improvement (PI_{AB}) in AET

Query identity	$PI_{EKS-EC2}$	$PI_{EKS-RDS}$	$PI_{EC2-RDS}$
Q1	5.132754995	34.733462947	31.202242618
Q2	6.275830395	29.914725576	25.221770731
Q3	13.63180755	-1.134585289	-17.097026604
Q4	-9.057437407	0.492721164	8.7569988801
Q5	35.489464359	1.2577890606	-53.063697207
Q6	26.415094339	11.826768488	-19.82516077
Q7	21.871004717	58.281871611	46.603526337
Q8	18.71118012	7.046942625	-14.349128842

Table 17 Query execution time for custom Kubernetes cluster on Washington State for HC-2

	ET-1	ET-2	ET-3	ET-4	ET-5	ET-6	ET-7	ET-8	ET-9	ET-10	AET
Q1	1.021	1.224	1.287	1.221	1.291	1.029	1.351	1.921	1.831	1.819	1.3995
Q2	1.2927	0.929	1.081	1.021	1.328	1.381	0.902	1.881	1.013	1.15	1.19787
Q3	1.421	1.223	1.223	0.935	1.121	0.823	0.921	0.898	1.092	0.991	1.0648
Q4	0.921	0.924	0.728	0.929	0.821	0.924	0.924	0.923	0.876	0.921	0.8891
Q5	0.986	0.924	0.905	0.921	0.987	0.926	0.984	0.985	0.832	0.887	0.9337
Q6	0.625	0.417	0.515	0.417	0.414	0.426	0.452	0.519	0.511	0.408	0.4704
Q7	5.984	5.187	5.929	5.314	5.235	5.531	5.259	5.234	5.813	5.516	5.5002
Q8	0.512	0.51	0.542	0.518	0.619	0.615	0.512	0.508	0.514	0.517	0.5367

For Table 18 similar to Table 17, we observed that the AETs for the custom Kubernetes cluster is comparable to AWS EKS from Table 14, but slightly higher due to the network latencies in the setup. Since the data of Washington is smaller than Colorado, the AET for Washington State data is less than Colorado State data.

From Fig. 16, it can be observed that the AETs of custom Kubernetes cluster are comparable to EKS, but take slightly more time; this can be due to the hidden latencies in the local load balancer.

From Fig. 17, it can be seen that for the Washington State as well the AETs for custom Kubernetes cluster is slightly higher than AWS EKS, because of the similar reason as seen in the Colorado dataset from Fig. 16.

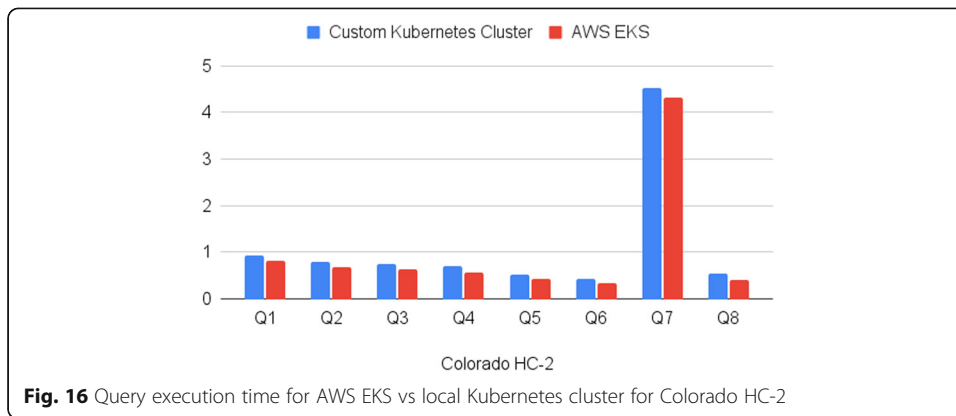
5 Discussion

Deploying and managing software applications and databases in a clustered environment is not an easy task, although the containerized applications can meet the requirements of ease of migration, portability, scalability, and flexibility when operating in a clustered environment.

Plenty of studies have been carried out comparing relational (SQL) and NoSQL databases in handling geospatial data. But these traditional database management technologies face frequent scalability problems when dealing with geospatial data. This paper demonstrates benchmarking the performance of operations on geospatial database by comparing the execution times of geospatial queries in clustered

Table 18 Query execution time for local Kubernetes cluster with Colorado State for HC-2

	ET-1	ET-2	ET-3	ET-4	ET-5	ET-6	ET-7	ET-8	ET-9	ET-10	AET
Q1	0.987	0.951	0.921	0.813	0.902	1.009	0.981	0.929	0.849	0.921	0.9263
Q2	0.869	0.712	0.703	0.827	0.859	0.842	0.781	0.838	0.753	0.831	0.8015
Q3	0.813	0.723	0.908	0.865	0.763	0.653	0.651	0.621	0.732	0.682	0.7411
Q4	0.709	0.774	0.719	0.762	0.763	0.611	0.768	0.609	0.656	0.599	0.697
Q5	0.528	0.501	0.566	0.535	0.457	0.536	0.524	0.518	0.538	0.491	0.5194
Q6	0.446	0.447	0.416	0.414	0.456	0.424	0.456	0.418	0.418	0.392	0.4287
Q7	4.673	4.861	4.361	4.335	4.608	4.526	4.491	4.248	4.563	4.494	4.516
Q8	0.712	0.513	0.515	0.518	0.542	0.499	0.496	0.514	0.531	0.499	0.5339

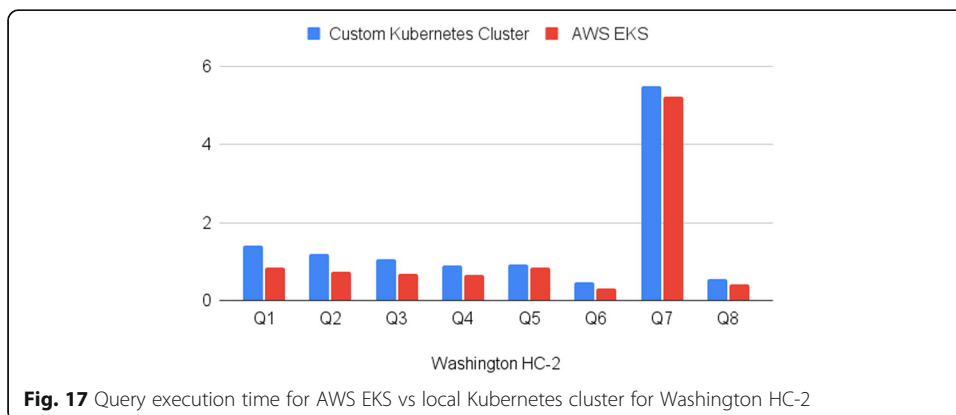


environment like Kubernetes and non-clustered environments. Kubernetes demonstrated its advantages by scaling on demand based on resource usage and performing better when compared to non-clustered environments for computationally expensive operations; this ability is particularly important for mission-critical applications and geospatial databases as they tend to be compute intensive thereby can be benefited immensely from operating in clustered environment. Setting up a custom local Kubernetes cluster proved to be a viable option for testing and validating conceptual architectures if we want the benefits of a clustered environment without incurring high costs.

A disadvantage of using a clustered environment with PostgreSQL compared to a managed non-clustered environment like AWS RDS is that we lose the ability to use a fully managed database. We have to set up, operate, manage, and maintain the database ourselves, which might not be very cost-efficient. We have to manage our own backups, survive downtime in the case of a crash, and increase our deployment cost, and in case of local Kubernetes clusters, we have to manage availability of clusters on top of all this.

6 Conclusions

The work aimed to benchmark geospatial databases in clustered and non-clustered environments. It was found that on processing geospatial queries operating upon indexed attributes and involving low computational overhead, both clustered and non-clustered environments offered similar performance, keeping the margin of error in mind. The



clustered environments performed better than non-clustered environments in scenarios where a computationally expensive geospatial query is involved or the query operated on non-indexed attributes and large data was retrieved from the geo-database. A clustered environment like AWS EKS could do this because of its ability to scale flexibly. Also, operating geo-databases in a clustered environment like AWS EKS (Kubernetes) can drastically improve its performance and scale on demand and automate administration or routine tasks, a good improvement, especially when computationally expensive operations are to be performed efficiently.

Abbreviations

SQL: Structured query language; ACID: Atomicity, consistency, isolation, and durability GIS: Geographic Information System; vCPU: Virtual central processing unit; RDBMS: Relational Database Management System; OSM: OpenStreetMap; AWS: Amazon Web Services; EC2: Amazon Elastic Compute Cloud; RDS: Amazon Relational Database Service; EKS: Amazon Elastic Kubernetes Service; HC-1: Hardware configuration 1; HC-2: Hardware configuration 2; RAM: Random-access memory; SSD: Solid-state drive; VPC: Virtual Private Cloud; AET: Average execution time

Acknowledgements

The authors would like to acknowledge all the participants for their contributions to this research study.

Authors' contributions

Methodology and writing-original draft preparation: Adisakshya Chauhan and Mohak Chugh. Experiments were led by Adisakshya Chauhan. Supervision and validation: Poonam Bansal and Bharti Sharma. Formal analysis and investigation: Prateek Anand, Qiaozhi Hua, and Achin Jain. The authors read and approved the final manuscript.

Funding

Not applicable.

Availability of data and materials

All data generated or analyzed during this study are included in this published article.

Declarations

Ethics approval and consent to participate

Not applicable.

Consent for publication

Not applicable.

Competing interests

The authors declare that they have no competing interests.

Author details

¹MSIT, GGSIPU, Delhi, India. ²NSUT, East Campus, Delhi, India. ³Computer School, Hubei University of Arts and Science, Xiangyang 441000, China. ⁴Bharati Vidyapeeth's College of Engineering, Delhi, India.

Received: 29 March 2021 Accepted: 23 June 2021

Published online: 19 July 2021

References

1. D. Guo, E. Onstein, State-of-the-art geospatial information processing in NoSQL databases in *ISPRS International Journal of Geo-Information*, 9(5), 331, (2020).
2. N. Zhang, G. Zheng, H. Chen et al., Hbasespatial: a scalable spatial data storage based on Hbase in proceedings of the 2014 IEEE 13th International Conference on trust, security and privacy in computing and communications, (2014), pp. 644-651.
3. S. Nishimura, S. Das, D. Agrawal, A. El Abbadi, MD-Hbase: design and implementation of an elastic data infrastructure for cloud-scale location services. *Distributed Parallel Databases* 31(2), 289-319 (2013)
4. D. Bartoszewski, A. Piorkowski, M. Lupa. The comparison of processing efficiency of spatial data for PostGIS and MongoDB databases. In *International Conference: Beyond Databases, Architectures and Structures* (pp. 291-302). Springer, Cham. (2019)
5. D. Han, E. Stroulia, Hgrid: a data model for large geospatial data sets In *Hbase in proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing, Santa Clara, CA, USA, (2013)*, pp. 910-917.
6. P. Yue, L. Jiang, BigGIS: how big data can shape next-generation GIS in proceedings of the Third International Conference on Agro-Geoinformatics, Beijing, China, (2014), pp. 413-418.
7. E. Baralis, A. Dalla Valle, et al., SQL versus NoSQL databases for geospatial applications in proceedings of the 2017 IEEE International Conference on Big Data, IEEE, Boston, MA, USA, (2017), pp. 3388-3397.
8. S. Schmid, E. Galicz, W. Reinhardt, Performance investigation of selected SQL and NoSQL databases in proceedings of the AGILE 2015, Lisbon, Portugal, (2015).

9. S. Agarwal, K.S. Rajan, Performance analysis of MongoDB versus PostGIS/PostgreSQL databases for line intersection and point containment spatial queries. *Spatial Inf. Res.* **24**(6), 671–677 (2016)
10. G. P. O. Reddy, Spatial data management analysis and modeling in GIS: principles and applications in Geospatial Technologies in Land Resources Mapping Monitoring and Management, Cham, Switzerland, (2018), pp. 127–142.
11. S. Ramzan, IS Bajwa, R Kazmi, Challenges in NoSQL-based distributed data storage: a systematic literature review. *Electronics*, **8**(5), 488 (2019).
12. E. Tang, Y. Fan, Performance comparison between five NoSQL databases in proceedings of the 2016 7th International Conference on Cloud Computing and Big Data, Macau, China, (2016), pp. 105–109.
13. J. Zhang, K. Yu, et al., 3D reconstruction for motion blurred images using deep learning-based intelligent systems. *CMC-Comput. Mater. Continua* **66**(2), 2087–2104 (2021)
14. R. Simmonds, P. Watson, J. Halliday, Antares: a scalable, real-time, fault-tolerant data store for spatial analysis in 2015 IEEE World Congress on Services, (2015), pp. 105–112.
15. K. Yu, L. Tan, et al., Blockchain-enhanced data sharing with traceable and direct revocation in IIoT. *IEEE Transact. Indust. Inf.* (2021)
16. L.I.U. Zhen, G.U.O. Huadong, W.A.N.G. Changlin, *Considerations on geospatial big data in IOP Conference Series* (2016), p. 46
17. K. Yu, L. Lin, et al., Deep learning-based traffic safety solution for a mixture of autonomous and manual vehicles in a 5G-enabled intelligent transportation system. *IEEE Transact Intell Transport Syst* (2020)
18. J. Bhimani, Z. Yang, M. Leaser, N. Mi, *Accelerating big data applications using lightweight virtualization framework on the enterprise cloud in High-Performance Extreme Computing Conference (HPEC)* (2017), pp. 1–7
19. N. Shi, L. Tan, et al., A blockchain-empowered AAA scheme in the large-scale HetNet, digital communications and networks, (2020).
20. V. Srivastava, S. Srivastava, G. Chaudhary, et al., A systematic approach for COVID-19 predictions and parameter estimation. *Pers Ubiquit. Comput.* (2020). <https://doi.org/10.1007/s00779-020-01462-8>
21. K. Yu, L. Tan, et al., Efficient and privacy-preserving medical research support platform against COVID-19: a blockchain-based approach. *IEEE Consumer Electron. Mag.* **10**(2), 111–120 (2021)
22. Y. Zhong, J. Han, T. Zhang, J. Fang, A distributed geospatial data storage and processing framework for large-scale WebGIS in Proceedings of the 2012 20th International Conference on Geoinformatics, Hongkong, China, (2012), pp. 1–7.
23. Tan L, Xiao H, Yu K, Aloqaily M, Jararweh Y. (2021). A blockchain-empowered crowdsourcing system for 5g-enabled smart cities. *Computer Standards & Interfaces*, 76. p. 103517.
24. C. Feng, K. Yu, et al., Efficient and secure data sharing for 5G flying drones: a blockchain-enabled approach. *IEEE Netw.* **35**(1), 130–137 (2021)
25. Zhen L, Bashir AK, Yu K, Al-Otaibi YD, Foh CH, Xiao P Energy-efficient random access for LEO satellite-assisted 6G internet of remote things. *IEEE Internet of Things Journal*, **8**(7), 5114–5128 (2020)

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Submit your manuscript to a SpringerOpen[®] journal and benefit from:

- Convenient online submission
- Rigorous peer review
- Open access: articles freely available online
- High visibility within the field
- Retaining the copyright to your article

Submit your next manuscript at ► [springeropen.com](https://www.springeropen.com)
