DZone

Composing a Sharded MongoDB Cluster on Docker Containers

by Ayberk Cansever · Aug. 04, 17 · Database Zone · Tutorial

In this article, we will write a `docker-compose.yaml` file and a cluster initiation scripts which will deploy a sharded MongoDB cluster on Docker containers.

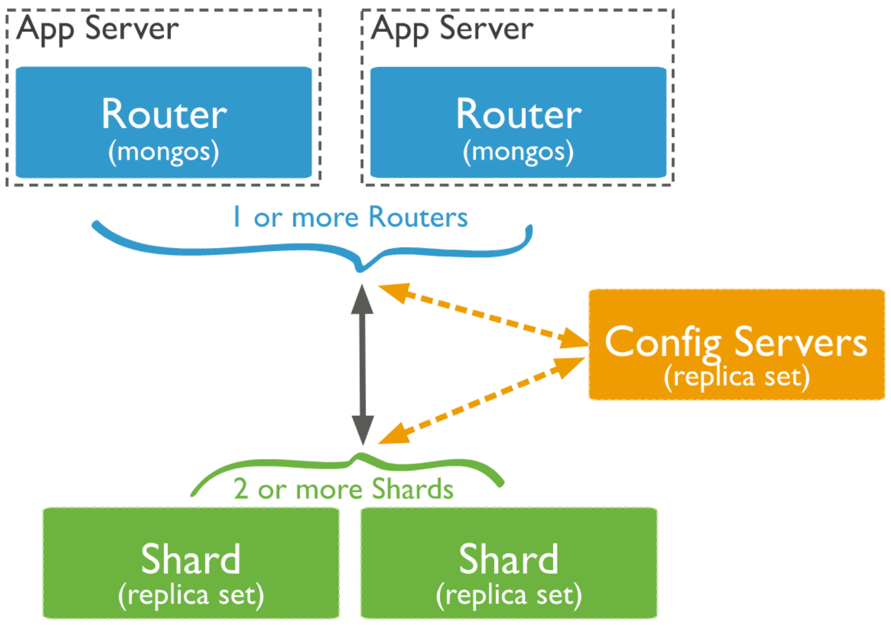
Initially, let's look what kind of components we are going to need for a sharded MongoDB cluster. If we look at the official documentation, we need three main components which are obviously defined:

1. **Shard**: Each shard contains a subset of the sharded data. Each shard can be deployed as a replica set.

2. **Mongos (router):** The mongos acts as query routers, providing an interface between client applications and the sharded cluster.

3. **Config servers:** Config servers store metadata and configuration settings for the cluster.

In the official documentation, the main architecture of a sharded cluster looks like this:



Now, we are beginning to build a cluster that consists of a shard that is a replica set (three nodes), config servers (three nodes replica set), and two router nodes. In total, we will have eight Docker containers running for our MongoDB sharded cluster. Of course, we can expand our cluster according to our needs.

Let's start to write our `docker-compose.yaml` file by defining our shard replica set:

```
version: '2'
services:
  mongorsn1:
    container_name: mongors1n1
    image: mongo
    command: mongod --shardsvr --replSet mongors1 --dbpath /data/db --port 27017
    ports:
      - 27017:27017
    expose:
      - "27017"
    environment:
      TERM: xterm
    volumes:
      - /etc/localtime:/etc/localtime:ro
      - /mongo_cluster/data1:/data/db
  mongors1n2:
    container_name: mongors1n2
    image: mongo
    command: mongod --shardsvr --replSet mongors1 --dbpath /data/db --port 27017
    ports:
      - 27027:27017
    expose:
      - "27017"
    environment:
      TERM: xterm
    volumes:
      - /etc/localtime:/etc/localtime:ro
      - /mongo_cluster/data2:/data/db
  mongors1n3:
    container_name: mongors1n3
    image: mongo
    command: mongod --shardsvr --replSet mongors1 --dbpath /data/db --port 27017
    ports:
      - 27037:27017
    expose:
      - "27017"
    environment:
      TERM: xterm
    volumes:
      - /etc/localtime:/etc/localtime:ro
      - /mongo_cluster/data3:/data/db
```

As you see, we defined our shard nodes by running them with the `shardsvr` parameter. Also, we mapped the default MongoDB data folder (*/data/db*) of the container, as you see. We will build a replica set with these three nodes when we finish writing our `docker-compose.yaml` file.

Now, let's define our three config servers:

```
mongocfg1:
    container_name: mongocfg1
    image: mongo
    command: mongod --configsvr --replSet mongors1conf --dbpath /data/db --port 27017
    environment:
      TERM: xterm
    expose:
      - "27017"
    volumes:
      - /etc/localtime:/etc/localtime:ro
      - /mongo_cluster/config1:/data/db
  mongocfg2:
    container_name: mongocfg2
    image: mongo
    command: mongod --configsvr --replSet mongors1conf --dbpath /data/db --port 27017
    environment:
      TERM: xterm
    expose:
      - "27017"
    volumes:
      - /etc/localtime:/etc/localtime:ro
      - /mongo_cluster/config2:/data/db
  mongocfg3:
    container_name: mongocfg3
    image: mongo
    command: mongod --configsvr --replSet mongors1conf --dbpath /data/db --port 27017
    environment:
      TERM: xterm
    expose:
      - "27017"
    volumes:
      - /etc/localtime:/etc/localtime:ro
      - /mongo_cluster/config3:/data/db
```

Our config servers are running with the `configsvr` parameter, as you see.

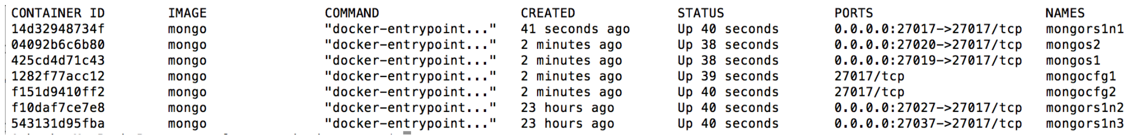
Finally, we are going to define our mongos (router) instances:

```
mongos1:
    container_name: mongos1
    image: mongo
    depends_on:
      - mongocfg1
      - mongocfg2
    command: mongos --configdb mongors1conf/mongocfg1:27017,mongocfg2:27017,mongocfg3:27017 --port 27017
    ports:
      - 27019:27017
    expose:
      - "27017"
    volumes:
      - /etc/localtime:/etc/localtime:ro
  mongos2:
    container_name: mongos2
    image: mongo
    depends_on:
      - mongocfg1
      - mongocfg2
    command: mongos --configdb mongors1conf/mongocfg1:27017,mongocfg2:27017,mongocfg3:27017 --port 27017
    ports:
      - 27020:27017
    expose:
      - "27017"
    volumes:
      - /etc/localtime:/etc/localtime:ro
```

These mongos are dependent on our config servers. They take the `configdb` parameter to obtain metadata and configuration settings.

At last, we built our `docker-compose.yaml` file. If we compose it up, we will see eight running docker containers: 3 shard date replicate set + 3 config servers + 2 mongos (routers):

```
docker-compose up
docker ps
```



```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                      NAMES
14d32948734f        mongo               "docker-entrypoint..."   41 seconds ago      Up 40 seconds       0.0.0.0:27017->27017/tcp   mongors1n1
04092b6c6b80        mongo               "docker-entrypoint..."   2 minutes ago       Up 38 seconds       0.0.0.0:27020->27017/tcp   mongos2
425cd4d71c43        mongo               "docker-entrypoint..."   2 minutes ago       Up 38 seconds       0.0.0.0:27019->27017/tcp   mongos1
1282f77acc12        mongo               "docker-entrypoint..."   2 minutes ago       Up 39 seconds       27017/tcp                  mongocfg1
f151d9410ff2        mongo               "docker-entrypoint..."   2 minutes ago       Up 40 seconds       27017/tcp                  mongocfg2
f10daf7ce7e8        mongo               "docker-entrypoint..."   23 hours ago        Up 40 seconds       0.0.0.0:27027->27017/tcp   mongors1n2
543131d95fba        mongo               "docker-entrypoint..."   23 hours ago        Up 40 seconds       0.0.0.0:27037->27017/tcp   mongors1n3
```

But we're not finished yet. Our sharding cluster needs to be configured. For this purpose, we will run some commands, which will build our cluster on related nodes.

First, we will configure our config servers replica set:

```
docker exec -it mongocfg1 bash -c "echo 'rs.initiate({_id: \"mongors1conf\",configsvr: true, members: [{ _id : 0, host
```

We can check our config server replica set status by running the below command on the first config server node:

```
docker exec -it mongocfg1 bash -c "echo 'rs.status()' | mongo"
```

We are going to see three replica set members.

Secondly, we are going to build our shard replica set:

```
docker exec -it mongors1n1 bash -c "echo 'rs.initiate({_id : \"mongors1\", members: [{ _id : 0, host : \"mongors1n1\"
```

Now, our shard nodes know each other. One of them is primary and two are secondary. We can check the replica set status by running the status check command on the first shard node:

```
docker exec -it mongors1n1 bash -c "echo 'rs.status()' | mongo"
```

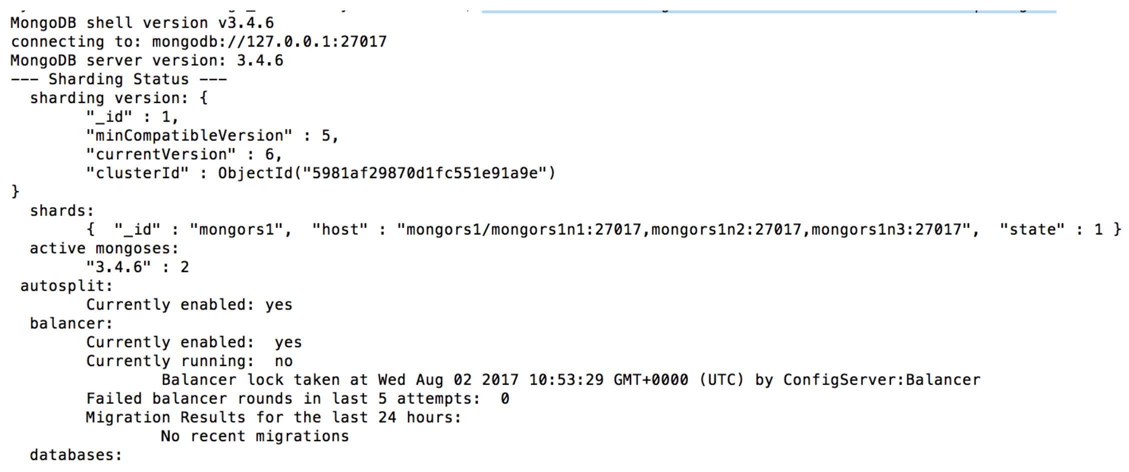
Finally, we will introduce our shard to the routers:

```
 docker exec -it mongos1 bash -c "echo 'sh.addShard(\"mongors1/mongors1n1\")' | mongo "
```

Now our routers, which are the interfaces of our cluster to the clients, have the knowledge about our shard. We can check the shard status by running the command below on the first router node:

```
 docker exec -it mongos1 bash -c "echo 'sh.status()' | mongo "
```

We see the shard status:



```
MongoDB shell version v3.4.6
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.4.6
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "minCompatibleVersion" : 5,
        "currentVersion" : 6,
        "clusterId" : ObjectId("5981af29870d1fc551e91a9e")
}
  shards:
        {  "_id" : "mongors1",  "host" : "mongors1/mongors1n1:27017,mongors1n2:27017,mongors1n3:27017",  "state" : 1 }
  active mongoses:
        "3.4.6" : 2
 autosplit:
        Currently enabled: yes
  balancer:
        Currently enabled:  yes
        Currently running:  no
                Balancer lock taken at Wed Aug 02 2017 10:53:29 GMT+0000 (UTC) by ConfigServer:Balancer
        Failed balancer rounds in last 5 attempts:  0
        Migration Results for the last 24 hours:
                No recent migrations
  databases:
```

We see that we have a single shard named `mongors1` , which has three mongod instances. But we do not have any databases yet, as you see. Let's create a database named testDb:

```
docker exec -it mongors1n1 bash -c "echo 'use testDb' | mongo"
```

This is not enough; we should enable sharding on our newly created database:

```
docker exec -it mongos1 bash -c "echo 'sh.enableSharding(\"testDb\")' | mongo "
```

Now, we have a sharding-enabled database on our sharded cluster! It's time to create a collection on our sharded database:

```
docker exec -it mongors1n1 bash -c "echo 'db.createCollection(\"testDb.testCollection\")' | mongo "
```

We created a collection named `testCollection` on our database, but it is not sharded yet again. We must shard our collection by choosing a sharding key. Let's assume that we have decided to shard our collection on a field named `shardingField` then:

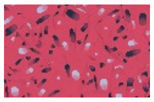
```
docker exec -it mongos1 bash -c "echo 'sh.shardCollection(\"testDb.testCollection\", {\"shardingField\" : 1})' | mongo
```

The sharding key must be chosen very carefully because it is for distributing the documents throughout the cluster. It is a *must* to read the official documentation about shard keys.

At the end, we have a sharded cluster, a sharded database, and a sharded collection. If we need to expand our cluster architecture in the future, we can add some new nodes as demanded!

Like This Article? Read More From DZone

Pitfalls and Workarounds for Tailing the Oplog on a MongoDB Sharded Cluster



When Should I Enable MongoDB Sharding?

MongoDB Replication and Sharding [Video]

Free DZone Refcard
Database Monitoring

Topics: NOSQL , MONGODB , DOCKER , SHARDED CLUSTER , TUTORIAL , DATABASE

Opinions expressed by DZone contributors are their own.

PostgreSQL Trends

by Kristi Anderson · Oct 09, 19 · Database Zone · Opinion

PostgreSQL Trends

PostgreSQL popularity is skyrocketing in the enterprise space. As this open source database continues to pull new users from expensive commercial database management systems like Oracle, DB2, and SQL Server, organizations are adopting new approaches and evolving their own to maintain the exceptional performance of their SQL deployments.

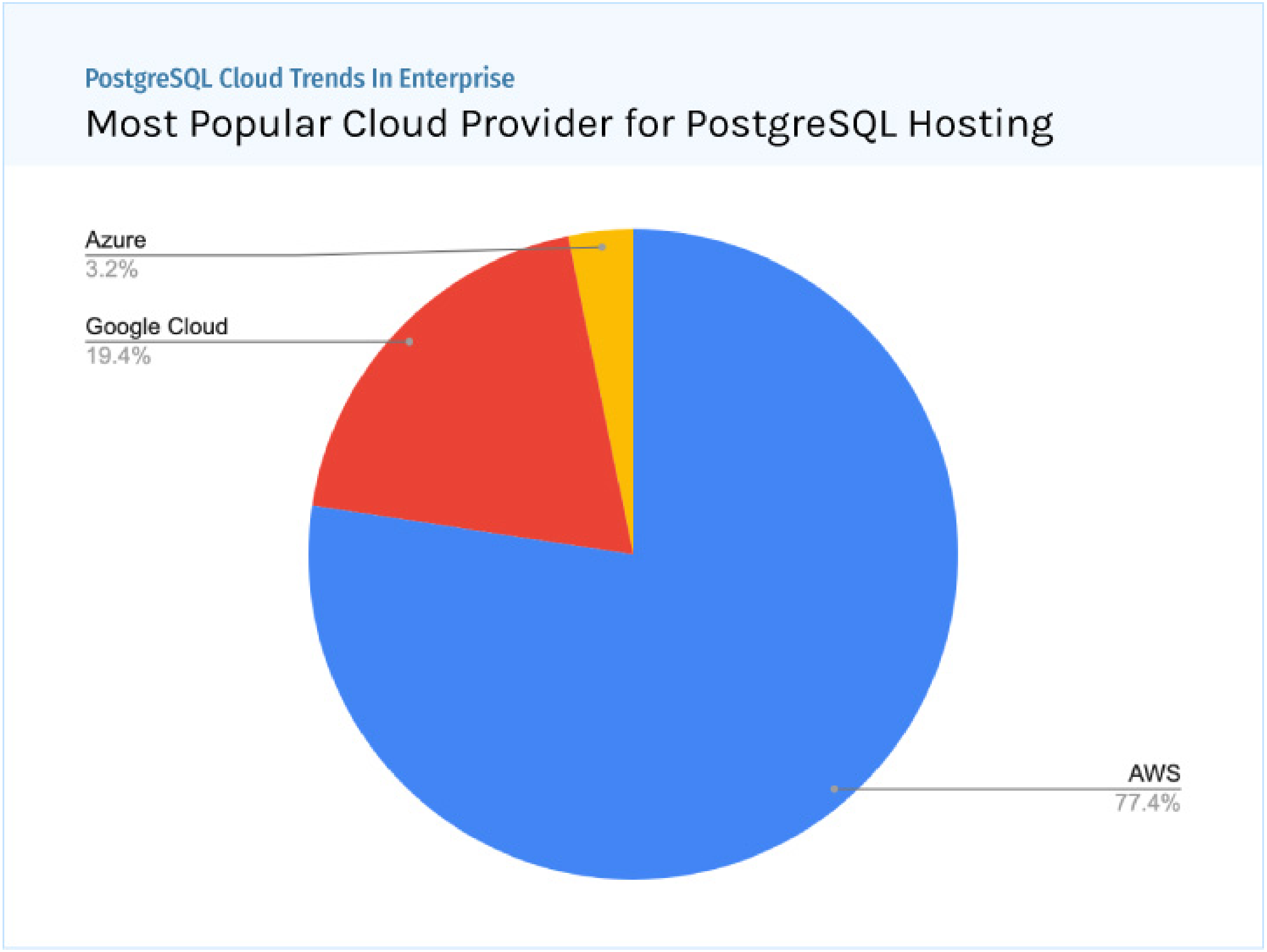
We recently attended the PostgresConf event in San Jose to hear from the most active PostgreSQL user base on their database management strategies. In this latest PostgreSQL trends report, we analyze the most popular cloud providers for PostgreSQL, VACUUM strategies, query management strategies, and on-premises vs public cloud use being leveraged by enterprise organizations.

You might also like: Essential PostgreSQL

Most Popular Cloud Providers for PostgreSQL Hosting

Let's start with the most popular cloud providers for PostgreSQL hosting. It comes as no surprise that the top three cloud providers in the world made up 100% of the PostgreSQL deployments in the crowd across this enterprise report. AWS, however, has taken a significant leap from our last report, where they now average 77.4% of PostgreSQL cloud use compared to 55.0% in April. AWS does offer a managed hosting service for PostgreSQL called Amazon RDS, but there are many other DBaaS solutions that offer PostgreSQL hosting on AWS, such as ScaleGrid, that can provide multi-cloud support so you're not locked in with a single cloud provider.

AWS was not the only cloud provider to grow – we found that 19.4% of PostgreSQL cloud deployments were hosted through Google Cloud Platform (GCP), growing 11% from April where they only averaged 17.5% of PostgreSQL hosting. This leaves our last cloud provider – Microsoft Azure, who represented 3.2% of PostgreSQL cloud deployments in this survey. This is one of the most shocking discoveries, as Azure was tied for second with GCP back in April, and is commonly a popular choice for enterprise organizations leveraging the Microsoft suite of services.



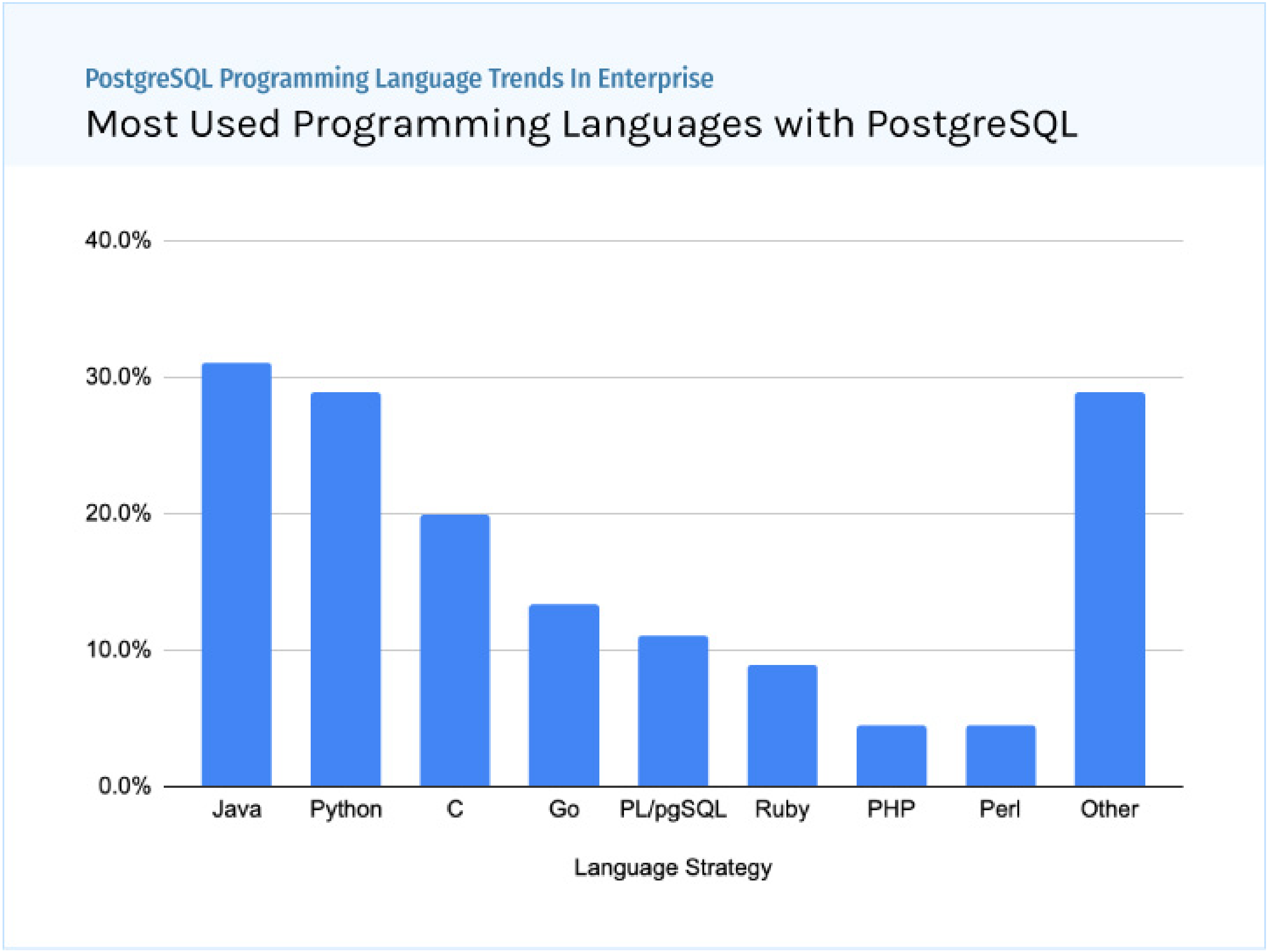
Most Used Languages With PostgreSQL

This is a new analysis we surveyed to see which languages are most popularly used with PostgreSQL. The supported programming languages for PostgreSQL include .Net, C, C++, Delphi, Java, JavaScript (Node.js), Perl, PHP, Python, and Tcl, but PostgreSQL can support many server-side procedural languages through its available extensions.

We found that Java is the most popular programming language for PostgreSQL, being leveraged by 31.1% of enterprise organizations on average. PostgreSQL can be easily connected with Java programs through the popular open source PostgreSQL Java Database Connectivity (JBDC) Driver, also known as PgJDBC.

Python was the second most popular programming language used with PostgreSQL, coming in close at an average of 28.9% use with PostgreSQL. Back in 2013, PostgreSQL surveyed their users to see which external programming languages was most often used with PostgreSQL, and found that Python only represented 10.5% of the results, showing a massive increase in popularity over the past six years.

The programming language C came in third place, averaging 20.0% use with PostgreSQL, followed by Go in fourth at 13.3%, PL/pgSQL in fifth at 11.1%, Ruby in sixth at 8.9% and both PHP and Perl in seventh at 4.4%. PHP was actually the most popular language used with PostgreSQL in 2013, representing almost half of the responses from their survey at 47.1% use. The last column, Other, was represented by C++, Node.js, Javascript, Spark, Swift, Kotlin, Typescript, C#, Scala, R, .NET, Rust and Haskell.



Most Popular PostgreSQL VACUUM Strategies

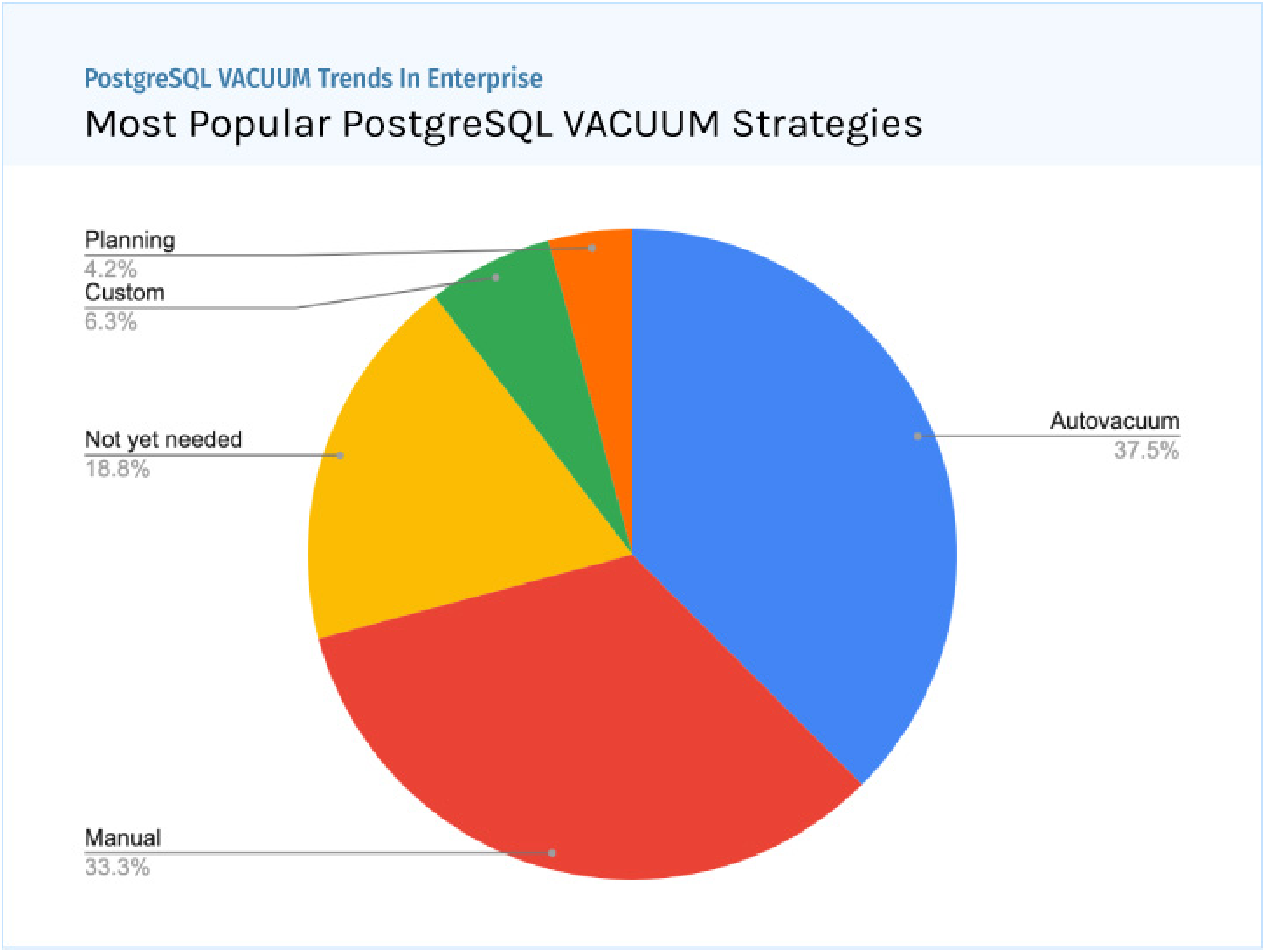
PostgreSQL VACUUM is a technique to remove tuples that have been deleted or are now obsolete from their table to reclaim storage occupied by those dead tuples, also known as Bloat. VACUUM is an important process to maintain, especially for frequently-updated tables before it starts affecting your PostgreSQL performance. In our survey, we asked enterprise PostgreSQL users how they are handling VACUUM to see what the most popular approaches are.

The most popular process for PostgreSQL VACUUM is the built-in autovacuum, being leveraged by 37.5% of enterprise organizations on average. The autovacuum daemon is optional, but highly recommended in the PostgreSQL community, at it automates both VACUUM and ANALYZE commands, continuously checking tables for deal tuples. While highly recommended, 33.3% of PostgreSQL users prefer to manually perform VACUUM in the enterprise space. Fibrevillage has a great article that outlines these common problems with autovacuum which may cause an organization to adopt a manual strategy:

- autovacuum may run even when turned off to deal with transaction ID wraparound.
- autovacuum is constantly running, which makes it start over every time it runs out of space, and start a new worker for each database in your cluster.
- autovacuum can cause out of memory errors.

- autovacuum may have trouble keeping up on a busy server.
- autovacuum can easily consume too much I/O capacity.

Another surprising discovery was that 18.8% of organizations do not use VACUUM, as it is not yet needed. This may be because they are leveraging PostgreSQL in small applications or applications that are not frequently updated. 6.6% of organizations have developed a custom solution for PostgreSQL VACUUM, and 4.2% are in the process of planning their VACUUM strategy.



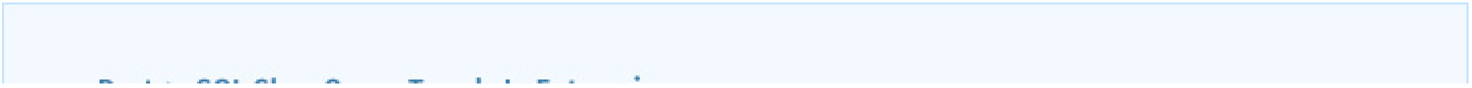
Most Popular PostgreSQL Slow Query Management Strategies

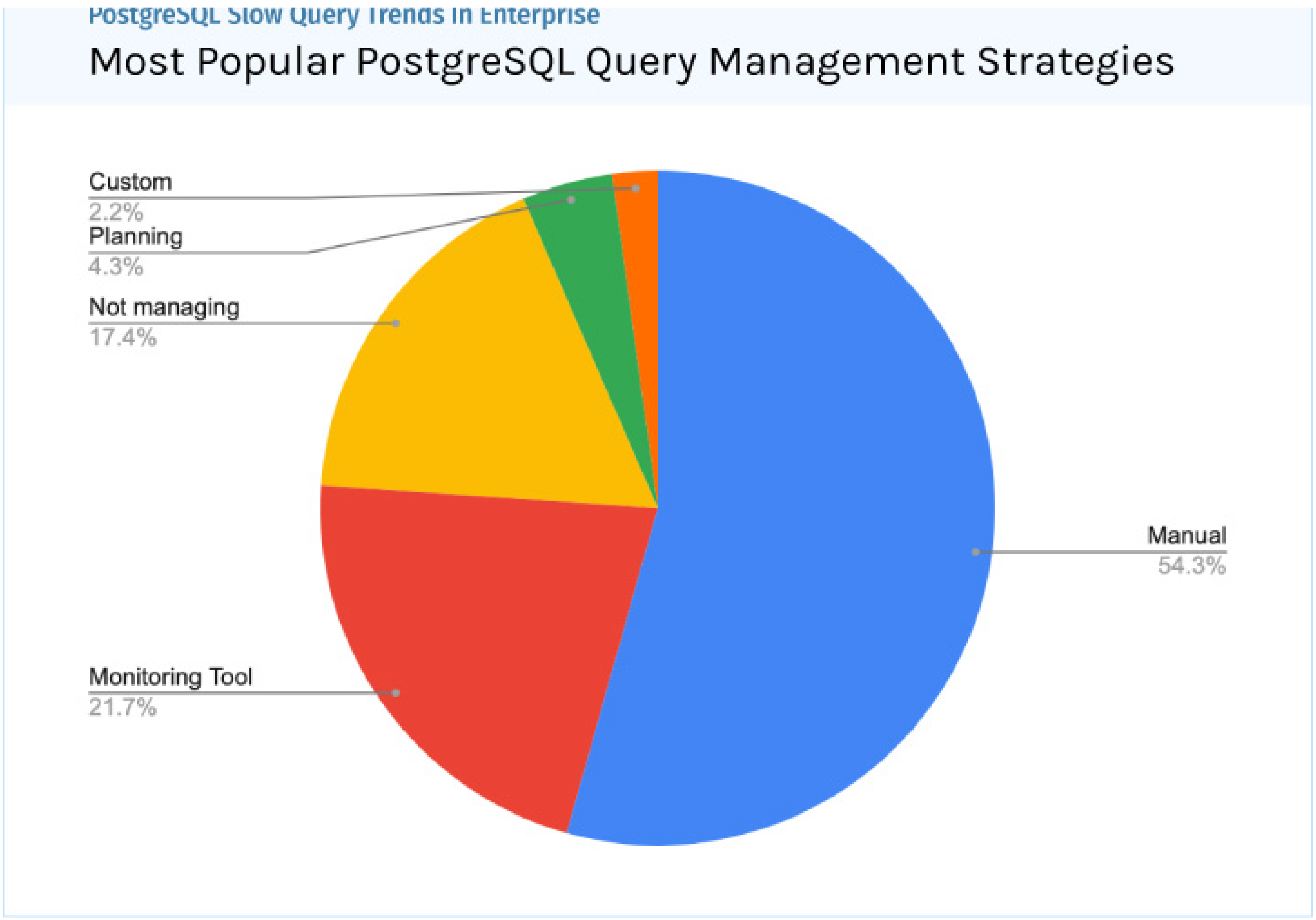
If you're working with PostgreSQL, you likely know that managing queries is the #1 most time-consuming task. It's a critical process with many aspects to consider, starting at developing a query plan to match your query structure with your data properties, to then analyzing slow-running queries, finally to optimizing those queries through performance tuning.

We found that 54.3% of PostgreSQL users are manually managing slow queries in enterprise organizations. This can be accomplished through their modules auto_explain and pg_stat_statements, checking pg_stat_activity for table and index activity on your server, analyzing the slow query log, or reviewing in your code.

On average, 21.7% of enterprise organizations are leveraging a monitoring tool to analyze and manage their PostgreSQL slow queries. This helps them significantly reduce the time it takes to identify which queries are running the slowest, most frequently, causing the most read or write load on your system, or queries missing an index by examining the rows.

17.4% of users, however, are not actively managing slow queries in the PostgreSQL deployments. We highly recommend adopting a query management strategy to ensure slow queries are not affecting the performance of your PostgreSQL deployments. 4.3% of users are currently in the process of planning their query management strategy, and 2.2% have developed a custom solution for managing their slow queries.





PostgreSQL Cloud vs On-Premises Deployments

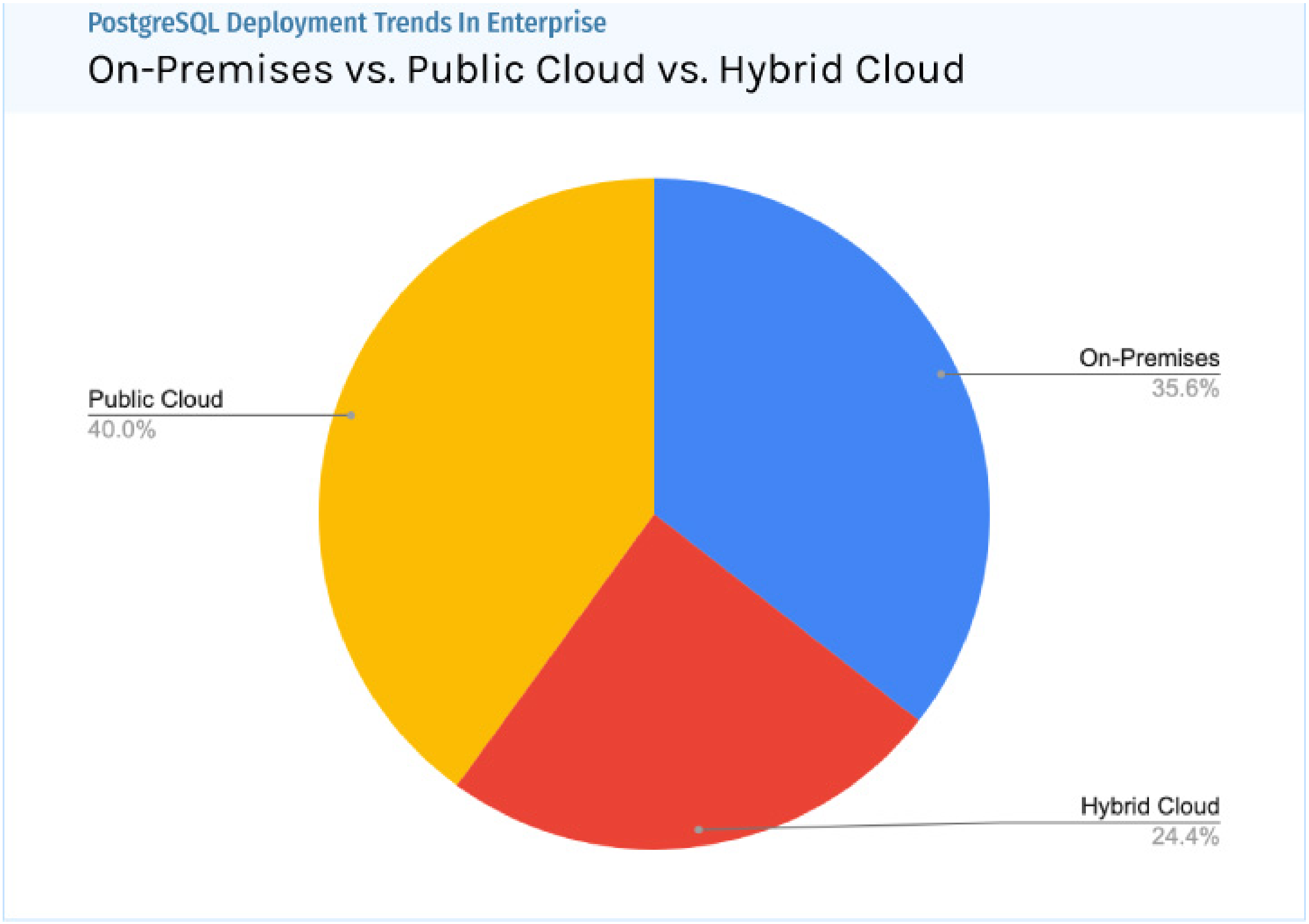
Let's end with one of the hottest topics in the PostgreSQL enterprise space – whether to deploy PostgreSQL in the cloud or on-premises. We've been actively monitoring this trend all year, and found that 59.6% of PostgreSQL deployments were strictly on-premises back in April from our 2019 PostgreSQL Trends Report and 55.8% on-premises in our 2019 Open Source Database Report just a few months ago in June.

Now, in this most recent report, we found that PostgreSQL on-premises deployments have decreased by 40% since April of 2019. On average, only 35.6% of PostgreSQL enterprise organizations are deploying exclusively on-premise. But organizations are not migrating their on-premises deployments altogether – 24.4% of PostgreSQL deployments were found to be leveraging a hybrid cloud environment. Hybrid clouds are a mix of on-premises, private cloud, and/or public cloud computing to support their applications and data. This is a significant increase from what we saw in April, jumping from 5.6% of PostgreSQL deployments up to 24.4% in September.

Hybrid cloud deployments are becoming more popular across the board — this recent report found that 57% of businesses opt for a hybrid cloud environment using both private and public clouds as places to store their data. While we see a large jump to the cloud, enterprise organizations are still leveraging on-premises environments in some capacity 60% of the time, compared to 65.2% in April. Lastly, we found that public cloud PostgreSQL deployments have grown 15% since April, now averaging 34.8% use by enterprise organizations.

It's also important to note that this survey was conducted at the PostgresConf Silicon Valley event, while our April survey was conducted in New York City. The bay area is widely known for adopting new technologies, which allows us to hypothesize that this market has a higher cloud adoption rate than the east coast.

PostgreSQL Deployment Types	**Apr**	**Jun**	**Sep**	**Apr-Sep Growth**
On-Premises	59.6%	55.8%	35.6%	-40.0%
Hybrid Cloud	5.6%	16.3%	24.4%	336%
Public Cloud	34.8%	27.9%	40.0%	15.0%



So, how do these results stack up to your PostgreSQL deployments and strategies? We'd love to hear your thoughts, leave a comment here or send us a tweet at @scalegridio.

Further Reading

Spring Boot and PostgreSQL

Set up a Spring Boot Application With PostgreSQL

Like This Article? Read More From DZone

Hybrid Cloud Versus Multi-Cloud: What's the Difference?

Functional Hybrid Cloud Alternatives for SMBs and Retail Chains

Should You Own or Rent Your Infrastructure?

Free DZone Refcard
Database Monitoring

Topics: POSTGRESQL, AWS, AZURE, GOOGLE CLOUD, QUERY PERFORMANCE, ON-PREMISE, PUBLIC CLOUD, HYBRID CLOUD, JAVA, PYTHON