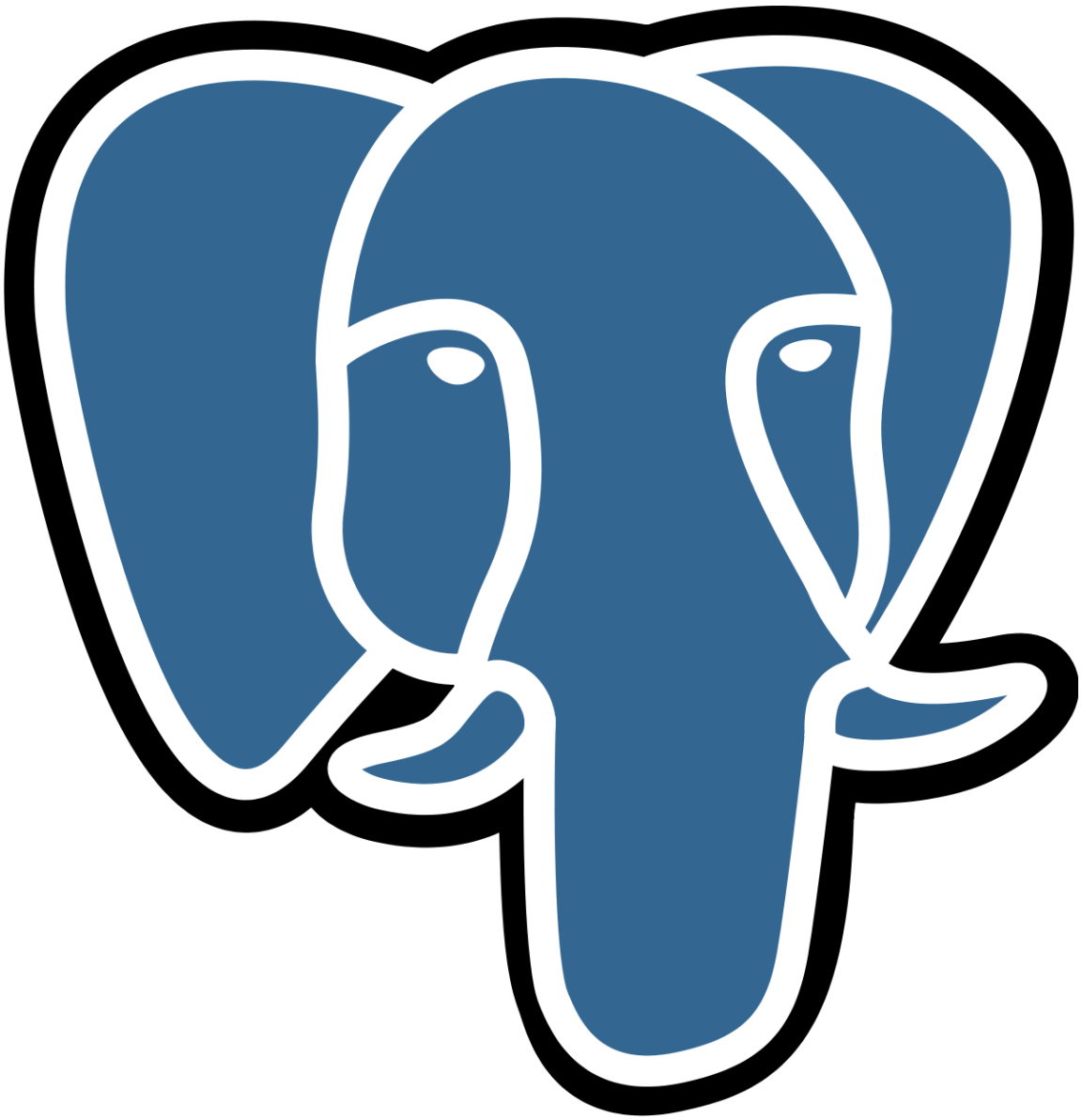
SQL DISTRIBUIDO

Juan Casado Ballesteros
Gino Cocolo Rodríguez



ÍNDICE

PROBLEMA EXISTENTE

Con el comienzo de la computación cloud comienza a haber un problema de rendimiento para manejar los datos. El número de usuarios que utilizan los sistemas crece exponencialmente y las arquitecturas tradicionales se quedan obsoletas.

Las bases de datos SQL deben actualizarse convirtiéndose en sistemas distribuidos que puedan ser capaces de soportar la demanda.

Los principios ACID por los que se rigen las bases de datos hacen que al convertirse en sistemas distribuidos solo puedan garantizar la disponibilidad y la consistencia de los datos.

- **Atomicidad**: todo o nada.
- **Consistencia**: coherencia de los datos.
- **Aislamiento**: serialización de transacciones.
- **Durabilidad**: los cambios son permanentes.

En este punto surgirán otros tipos de bases de datos NoSQL que se centrará en garantizar las otras dos combinaciones de servicios.

- Consistencia + Tolerancia a las particiones
- Disponibilidad + Tolerancia a las particiones

Dependiendo de aquello que necesitemos deberemos elegir un tipo de sistema u otro.

Pero, aun así, estas soluciones basadas en NoSQL no podían ofrecer, por cómo han sido diseñadas, esa misma consistencia que aseguraban aquellas bases de datos que seguían como filosofía los principios ACID. Eso condujo a la aparición de un nuevo diseño.

SQL DISTRIBUÍDO

SQL distribuido es el nuevo diseño que surgió en esa situación de obsolescencia del SQL tradicional en algunos de los nuevos escenarios de trabajo que habían aparecido. Es la respuesta a la necesidad de escalar las bases de datos tradicionales.

Funciona utilizando una serie de bases de datos diseminadas en distintos emplazamientos en una misma red, interrelacionadas entre sí, con un mismo gestor de base de datos que maneja todas las consultas y operaciones, dando la sensación al usuario que está tratando con una base de datos relacional tradicional.

Esto permite que, a la hora de pedir datos, aunque provengan de diferentes equipos de la red, no haya que preocuparse por eso. Ni de dónde ni cuantas replicaciones y copias hay. Ni de si los datos han sido fragmentados.

Métodos para lograr distribución

Existen dos métodos basado en principios opuestos para logarla.

Base de datos homogénea

Todas las distintas bases de datos trabajan de la misma forma. Comparten sistema operativo y estructuras de datos, así como gestor. También podemos diferenciar dentro de este tipo las que son autónomas, en las que cada base de datos es independiente y funciona por su cuenta, integrados en una aplicación que se encarga de controlarlas y compartiendo los cambios en los datos mediante el envío de mensajes; y las no autónomas, en las que la información se distribuye entre los nodos y un gestor de base de datos maestro coordina los cambios.

Base de datos heterogénea

En este tipo, cada sitio puede contar con distintos esquemas, software, gestor… Esto hace que el proceso a la hora de consultar datos sea complejo, al igual que el procesado de las transacciones. Al no tener que ser consciente un nodo del

resto de sus compañeros, esto puede ocasionar una bja cooperación cuando se intenta resolver una consulta o petición por parte del usuario. A su vez se dividen en federadas, en las que cada sistema es independiente en naturaleza e integrados juntos funcionando como uno solo; y no federadas, que incluyen un módulo central con la tarea de coordinar a través del cual se produce el acceso a las bases de datos.

Diseños de distribución

Sin replicación y sin fragmentación

Distintas tablas se guardan en distintos lugares. Generalmente, el criterio a seguir a la hora de almacenar, es almacenarlo geográficamente próximo a aquellos usuarios que más lo usen. Este diseño es utilizado cuando las consultas tienen una baja necesidad de información conjunta por varias tablas, y permite reducir costes de comunicación.

Totalmente replicada

Cada sitio guarda una copia de la base de datos en su totalidad. Esto tiene como ventaja su rapidez al no tener que comunicarse con otros nodos a la hora de servir datos de las consultas. Pero como desventaja el gran coste que conlleva realizar una actualización de los datos, al tener que realizarse en todos los nodos. Es obvio que este diseño en sistemas con un gran volumen de consultas, pero con baja necesidad de actualización es ideal.

Parcialmente replicada

Copias de las tablas o de parte de las tablas se guardan en distintos sitios. La decisión de dónde almacenar qué viene determinada por la frecuencia de acceso a los datos. Y el número de copias a guardar dependerá también de la frecuencia de consulta y el usuario que las realiza.

Fragmentado

Las tablas se dividen en varios fragmentos y son guardados en distintos sitios. Este diseño no cuenta con datos duplicados, es decir, elimina la redundancia. También procura incrementar el paralelismo de las operaciones. La

fragmentación puede ser horizontalmente, es decir, una tabla es dividida en dos o más subsets cada uno conteniendo todas las columnas, pero parte de las filas, o verticalmente, dividiendo esos subsets para que contengan parte de las columnas y todas las filas.

Independientemente de la naturaleza, todas comparten el tener un solo gestor con el que interactuar por parte del cliente. Esto ofrece la posibilidad de trabajar como hasta ahora, con todas las ventajas para la escalabilidad.

SOLUCIONES

Encontramos múltiples soluciones que implementan SQL distribuido. Exploraremos una base de datos SQL no diseñada para ser distribuida que se ha adaptado para poder serlo y otra base de datos SQL que desde un origen fue diseñada para ser distribuída.

Pgpool-II

Pgpool-II es una herramienta que nos permite centralizar el acceso a múltiples servidores que estén corriendo una misma base de datos de postgreSQL, teniendo que hacer conexión con un único cliente.

Esto conlleva que la herramienta es capaz de balancear la carga sobre los servidores, dejar abiertas conexiones para utilizarlas en múltiples consultas sin necesidad de tener que estar constantemente abriéndolas, y una alta capacidad de tolerancia ante los fallos, al ser capaz de recuperarse automáticamente al producirse un error en alguno de los servidores y seguir ofreciendo su servicio.

Su uso es tan sencillo como especificar las direcciones de las bases de datos, y poner en marcha la herramienta. A continuación se muestra un ejemplo de cómo hacer esa puesta en marcha con docker-compose.

Lo primero es crear el fichero docker-compose.yml especificando los nodos de base de datos, el nodo maestro al que se realizará la conexión por parte del cliente y el resto de configuraciones:

curl -sSL https://raw.githubusercontent.com/bitnami/bitnami-docker-pgpool/master/docker-compose.yml > docker-compose.yml

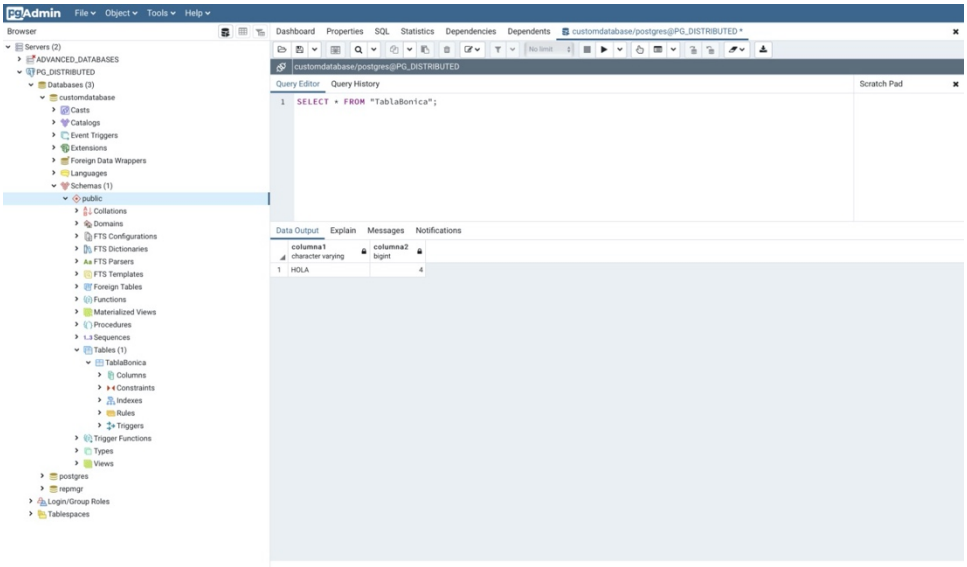
Una vez ejecutado ese comando, podemos editar el contenido para adecuarlo a nuestras necesidades, modificando usuarios y contraseñas, puertos, etc.

Solamente quedaría ponerlo en marcha con:

docker-compose up –d

```
~/Documents » docker ps                                                                    mrblissfulgrin@Juans-MacBook-Pro
CONTAINER ID        IMAGE                          COMMAND                  CREATED             STATUS                    PORTS                                                NAMES
6738391066d0        bitnami/postgresql-repmgr:11   "/entrypoint.sh /run…"   53 seconds ago      Up 50 seconds             0.0.0.0:32768->5432/tcp                              presentation_pg-0_1
3b12c60e4428        bitnami/pgpool:4               "/entrypoint.sh /run…"   53 seconds ago      Up 50 seconds (healthy)   0.0.0.0:5432->5432/tcp                               presentation_pgpool_1
3c4644d16931        bitnami/postgresql-repmgr:11   "/entrypoint.sh /run…"   53 seconds ago      Up 49 seconds             0.0.0.0:32769->5432/tcp                              presentation_pg-1_1
698b046e3e50        neo4j:4.0                      "/sbin/tini -g -- /d…"   About an hour ago   Up About an hour          0.0.0.0:7474->7474/tcp, 7473/tcp, 0.0.0.0:7687->7687/tcp   ecstatic_dhawan
```

Podemos después conectarnos desde pgAdmin4 al maestro, definido en el fichero como pgpool, con las credenciales correspondientes. Sin nosotros tratar con los demás nodos, podemos ejecutar consultas como se puede observar en la siguiente imagen.



CockroachDB

Permite tanto su despliegue en la nube (AWS, Axure...) o en los propios sistemas que uno posea.

Es un sistema distribuido con una manera particular de funcionar. Todos los datos, como tablas o índices, se encuentran almacenados en un mapa ordenado de clave-valor, a su vez dividido en rangos, segmentos contiguos de claves, para que cada clave se encuentre en un solo rango. Es decir, una tabla y sus índices están mapeados a un solo rango, donde cada clave-valor representaría una fila de la tabla, por ejemplo.

Con esa división, cuando recibe una petición SQL, lo primero que hace es convertirla a operaciones que puedan trabajar con ese almacén de clave-valor. Según se vayan almacenando los datos, CockroachDB empieza a distribuirlos

por todos los nodos en rangos de 64MiB, replicándolo en al menos tres nodos para asegurar que siempre vas a poder acceder a los datos para leer y escribir y replicar a otros nodos.

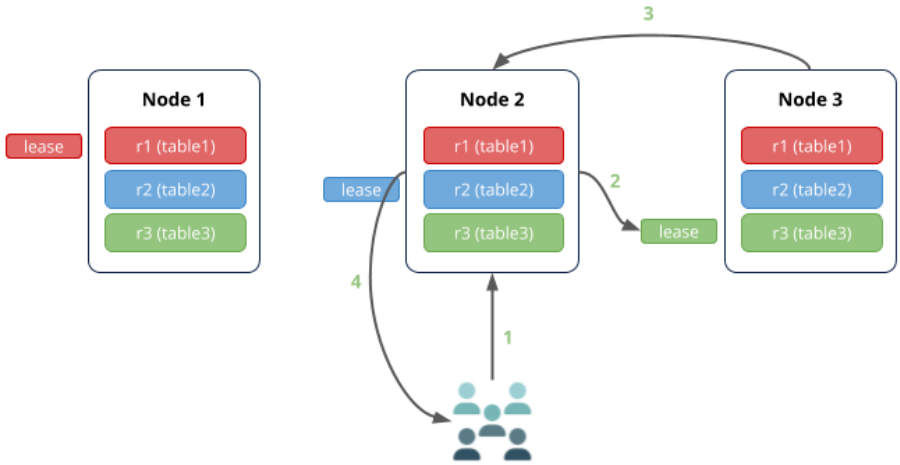
En cuanto un nodo recibe una petición de datos que no posea, busca un nodo que pueda servir esos datos y le manda una petición. Y cuando se va a realizar un cambio sobre los mismos, se aplica un algoritmo de consenso para asegurar que las réplicas ejecutan también los cambios, ofreciendo así aislamiento de los datos.

Cada operación que se ejecuta en CockroachDB pasa por sus cinco capas, en este orden:

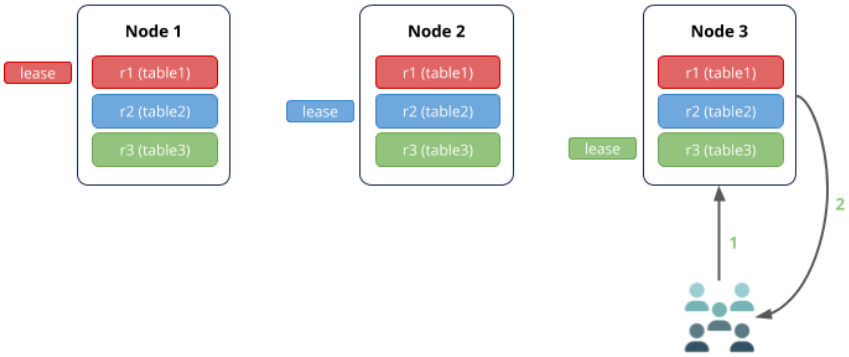
1. Capa SQL: se encarga de hacer la traducción de las consultas SQL a operaciones clave-valor.
2. Capa transaccional: permite cambios atómicos a múltiples entradas clave-valor.
3. Capa de distribución: presenta rangos de clave-valor replicados como una sola entidad.
4. Capa de replicación: se encarga de consistente y síncronamente replicar los rangos de clave-valor en varios nodos, permitiendo lecturas consistentes.
5. Capa de almacenamiento: lee y escribe claves-valores en disco.

Para entrar un poco más en detalle en las operaciones de lectura y escritura y cómo mantiene los datos, si tuviéramos tres nodos, tres pequeñas tablas que entran cada una en un rango, las tres tablas replicadas tres veces. Cada nodo posee el control, denominado por Cockroach como *lease*, de una tabla determinada.

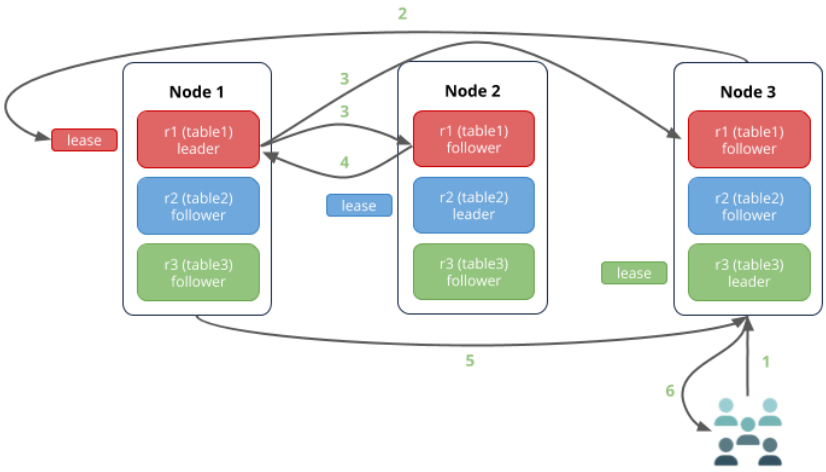
Sobre este escenario, al nodo 2 le llega una petición de lectura de la tabla 3 (1). Quien tiene el control sobre la tabla 3 es el nodo 3, por lo que le pasará el mensaje a ese nodo (2). Cuando el nodo 3 reciba el mensaje, mandará los datos al nodo 2 (3) y éste a su vez al cliente que ha hecho la petición (4).



Si la petición hubiera llegado directamente al nodo 3, no habría tantos saltos, si no que directamente el nodo 3 mandaría respuesta con los datos.



Sobre este mismo escenario, llega una petición de escritura al nodo 3 de la tabla 1 (1). Quien posee la *lease de* la tabla 1 es el nodo 1, por lo que redirige la petición ahí (2). Una vez recibe la petición, como líder, escribe en lo que se denomina *Raft log*, un log en el que se registran las escrituras en un rango que las réplicas han acordado para una replicación consistente, y tras escribir en ese log, notifica a los seguidores, las réplicas (3). Los seguidores escriben en su *Raft log* y notifica al líder (4). En cuanto el líder recibe respuesta y la mayoría de los nodos, en este caso dos, la operación de escritura se lleva a cabo. Tras realizar la operación, el nodo 1 manda confirmación de la operación al nodo 3 (5), y éste la envía al cliente (6).



Si quisiéramos realizar la instalación en una máquina Linux, es tan sencillo como primero descargar el binario y extraerlo:

wget -qO- https://binaries.cockroachdb.com/cockroach-v19.2.4.linux-amd64.tgz | tar xvz

Y una vez extraído añadirlo al PATH para poder ejecutar fácilmente cualquier comando de CockroachDB:

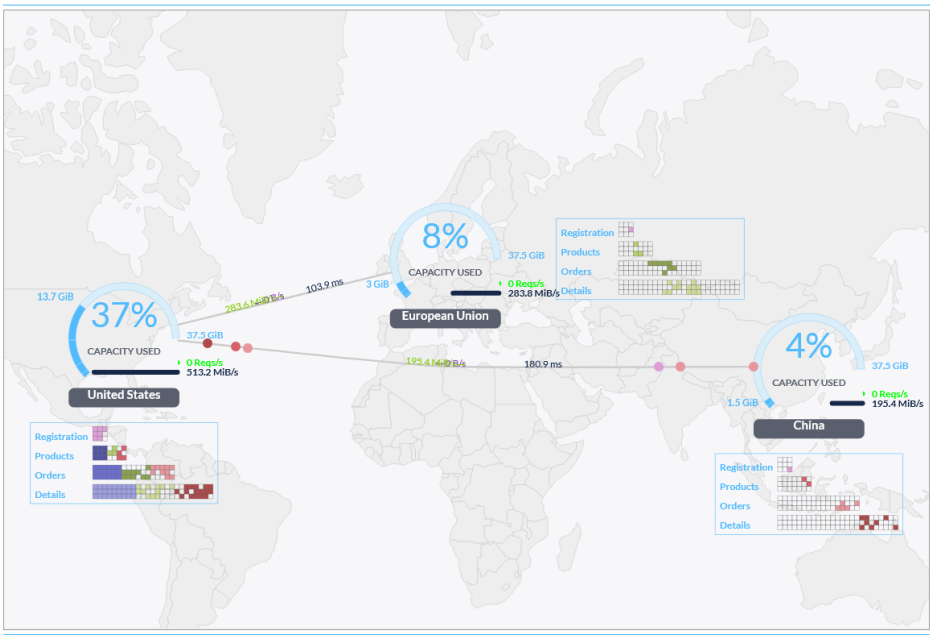
cp -i cockroach-v19.2.4.linux-amd64/cockroach /usr/local/bin/

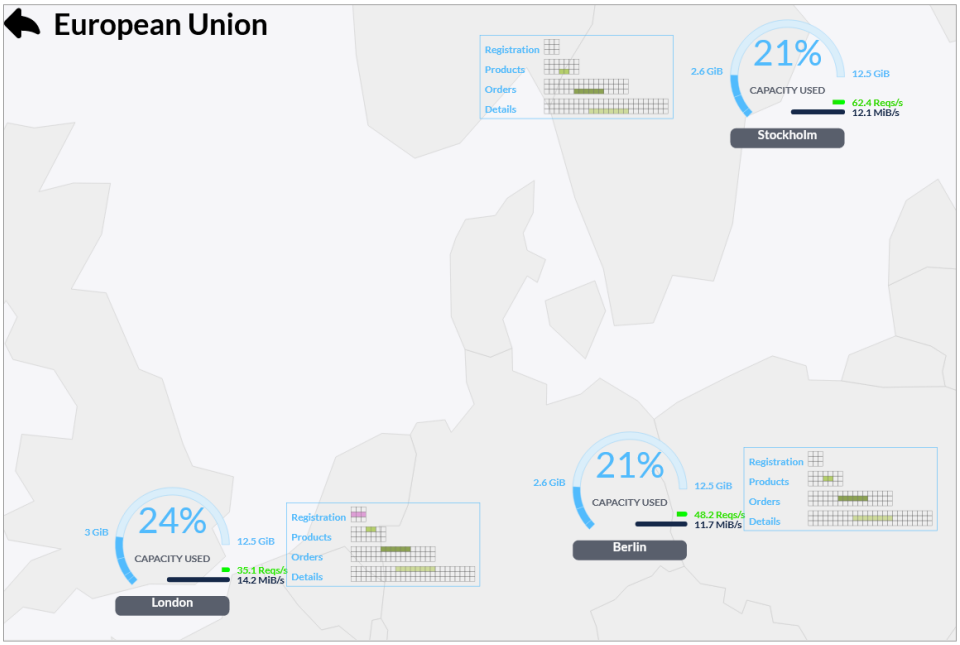
A la hora de hacer despliegue, ofrece una extensa documentación sobre todos los pasos a seguir para las plataformas en la nube más conocidas y en servidores propios, tanto los pasos a tener en cuenta antes de pasar a la fase de producción.

https://www.cockroachlabs.com/docs/stable/manual-deployment.html

https://www.cockroachlabs.com/docs/stable/recommended-production-settings.html

Una vez lo tuviéramos en marcha, podemos ver un panel de administración como el de la siguiente imagen, donde se ven los nodos geolocalizados, incluso la partición de los datos:





Por terminar, un ejemplo de empresa que haya implementado este servicio en sus operaciones es Justuno, una empresa que ofrece servicios de marketing. Migraron de Microsoft SQL Server a CockroachDB debido a su necesidad de realizar transacciones distribuidas que siguieran el dogma ACID y le permitieran tener redundancia en varias regiones distintas.

Tras probar el servicio, vieron que cumplía todas las características que necesitaban para su actividad, siendo un servicio resilente y consistente, sin

picos de latencia, llegando a ofrecer cero RPO (Recovery Point Objective, el volumen de datos en riesgo de pérdida) y cero tiempos en fuera de servicio.

BIBLIOGRAFÍA

https://www.cockroachlabs.com/blog/what-is-distributed-sql/

https://blog.yugabyte.com/what-is-distributed-sql/

https://www.geeksforgeeks.org/advantages-of-distributed-database/

https://www.geeksforgeeks.org/distributed-database-system/

https://www.geeksforgeeks.org/functions-of-distributed-database-system/

https://www.tutorialspoint.com/distributed_dbms/distributed_dbms_database_environments.htm

https://github.com/bitnami/bitnami-docker-pgpool

https://www.cockroachlabs.com/docs/stable/architecture/reads-and-writes-overview.html

https://www.cockroachlabs.com/case-studies/justuno/