Low-cost Open Data As-a-Service in the Cloud

Marin Dimitrov, Alex Simov, Yavor Petkov

Ontotext AD, Bulgaria
{first.last}@ontotext.com

Abstract. In this paper we present the architecture and prototype of a Cloud based RDF database-as-a-service (DBaaS) that enables low-cost hosting of a large number of Open Data sets as data services. The RDF DBaaS is designed for availability and reliability of the hosted data services.

Keywords: data-as-a-service, software-as-a-service, database-as-a-service, linked data, open data, cloud computing

1 Use Case

The DaPaaS project[1] has the goal to provide a Data- and Platform-as-a-Service environment where SMEs, governmental organisations and individual developers can publish and access open data services. The promise of DaPaaS is to make publishing, and reuse of open data easier and cheaper for small organisations which otherwise may lack the required expertise and resources for the job.

One of the core components of the DaPaaS platform is the data management layer for hosting a large number of open datasets, via a data-as-a-service delivery model, where each data service is a RDF database available for querying by 3rd party applications via standard OpenRDF[2] RESTful services and SPARQL endpoints. A key difference between such an open data service hosting scenario, and hosting large knowledge graphs such as DBpedia, is that the majority of the open data sets are expected to be relatively small, and not a subject to a large number of complex concurrent queries.

2 RDF Database-as-a-Service in the Cloud

The RDF database-as-a-service (DBaaS) capability of the DaPaaS platform is based on the GraphDB[3] database. The DBaaS provides a 24/7 access to open datasets, by means of RDF databases and SPARQL endpoints deployed within a multi-tenant environment. Operational aspects such as security, isolation, availability, monitoring and backups are fully managed by the platform on behalf of the users. The security

1 http://project.dapaas.eu/
2 http://openrdf.org/
3 http://www.ontotext.com/products/ontotext-graphdb/

isolation and resource utilisation control of the different database instances hosted within the same virtual machine is achieved via a container-based architecture with the Docker[4] technology. For the purpose of increased availability and reliability, the DBaaS utilises infrastructure services (storage, messaging services, and load balancers) which are designed by the cloud provider for maximum reliability and availability. The low-cost goal for the DBaaS is achieved by a employing a multi-tenant hosting model, so that the hosting infrastructure cost is shared by multiple users and economies of scale can be achieved.

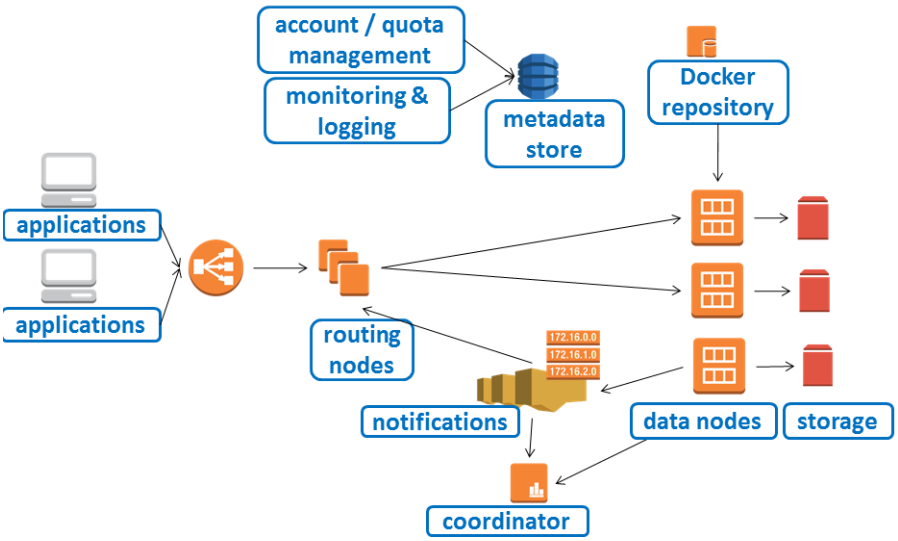
2.1 Public Cloud Platform

The RDF DBaaS is currently deployed on a public AWS[5] cloud platform and it utilizes various infrastructure services such as:

- *distributed storage* via Simple Storage Service (S3), Elastic Block Storage (EBS)
- *a key-value datastore* via DynamoDB
- *computing* and *scalability* via Elastic Compute Cloud (EC2), Auto Scaling, and Elastic Load Balancer (ELB)
- *application integration* via Simple Notification Service (SNS)

The DBaaS architecture can be ported to other public cloud platforms, such as Microsoft Azure or the Google Cloud Platform, which provide similar cloud infrastructure services for distributed storage, computing, and application integration.

Fig. 1. RDF Database-as-a-Service Architecture



[4] https://www.docker.com/
[5] http://aws.amazon.com/

2.2 Architecture

The architecture of the RDF DBaaS is based on the best practices and design patterns for scalable cloud architectures [1] and microservice architectures [2]. It is comprised of the following components (**Fig. 1**):

- *Load balancer* – the entry point to all data services is the Elastic Load Balanced (ELB) of AWS which routes incoming RESTful requests to one of the available routing nodes.
- *Routing nodes* – a routing node works as follows: first it validates the incoming request (checks API keys, access control and quota limits); then the routing node will forward the original request to the proper data node, hosting the database that the client request is trying to reach. Each routing node maintains a routing table so that it knows which data node is currently hosting a particular database. If there is a change in the data node topology (e.g. a data node crashes and another one is instantiated to replace it) then the routing tables are updated. After the routing node forwards the client request to the proper data node, it waits for the response and then forwards the response back to the client application. Routing nodes are dynamically scaled up (new nodes started) or down (an idle node shut down), based to the number of concurrent SPARQL queries within a period of time.
- *Data nodes* – a data node hosts a number of independent GraphDB database instances packaged as Docker containers. Each database container stores its data on a dedicated Network-attached storage volume (EBS), and no EBS volumes are shared between database containers, for improved performance and isolation. At present there is only one instance of each database active on the platform, though future RDF DBaaS versions will introduce container replication so that read-only requests can be distributed among multiple nodes serving the same database replica. New data nodes are dynamically added by the *Coordinator* node when needed – when a user requests the creation of a new database, and there are no available container slots on the currently active data nodes. If a data node has free container slots then it periodically contacts the coordinator for the IDs of databases which are still waiting to be deployed and initialised, so that the data node can fill up its full hosting capacity as soon as possible – by attaching the dedicated EBS volume for the database to an available container on the data node.
- *Coordinator* – there is a single coordinator responsible for distributing the database initialisation tasks among the active data nodes. The coordinator also keeps a routing table, so that it knows which databases are fully operational (already deployed on an operational data node). If a data node crashes then the coordinator marks its databases as non-operational and will re-distribute them (their IDs) to other data nodes with free database slots, or to the new data node which will be automatically instantiated by the AWS Auto Scaler to replace the crashed node.
- *Integration services* – a distributed push messaging service (Simple Notification Service, SNS) is used for loose coupling between the data nodes, routing nodes and the coordinator. Each data node sends "heartbeats" several times per minute via the notification service, so that routing nodes and the coordinator get a confirmation that the databases hosted by that node are still operational. Each data node also

sends periodic updates regarding the databases it is hosting, so that the routing nodes and the coordinator can update their routing tables if necessary.
- *Metadata store* – a key-value store (DynamoDB) stores simple metadata regarding the databases hosted on the platform (user ID, dedicated EBS volume ID, configuration parameters, etc.), and logs and usage statistics from database operations.
- *Management and monitoring services* – various microservices cover operational aspects such as logging, reporting, account and quota management.

2.3 Normal Operations

Routing Node

When a routing node is started, it immediately subscribes to the notifications service, so that it can start receiving heartbeats and routing updates from the data nodes. If a client request is forwarded to the new routing node before it has its routing table fully initialised, then it will just queue the request locally. After a short period the routing node will receive the heartbeat and routing notifications from all active data nodes, so that it can start forwarding client requests to the proper data node.

Coordinator

When the coordinator is started, it first reads the metadata about all databases on the platform from the Metadata Store, and then subscribes to the notifications service in order to receive heartbeats and routing updates from the backend nodes. If after short period there is still a database which no data node lists as operational, then the coordinator assumes this database is down and will send its ID to the next data node which contacts the coordinator requesting pending databases.

Data Node

When a data node is started, it initialises its database containers from the local Docker repository and immediately contacts the coordinator to request a list of databases (IDs) which to host on its containers. When the coordinator provides the information, the data node just attaches the dedicated Network-attached storage (EBS) volume for the database to itself and performs the final OS level configuration so that the database container can be fully initialised. At this point, a Docker container with a running GraphDB database instance and an attached data volume is fully operational on the data node. The next step is to subscribe to the notifications service and start sending regular heartbeats and routing updates for the routing nodes and the coordinator. At this point the data node is ready to serve client requests forwarded to its hosted databases by the routing nodes.

2.4 Dealing with Failure

Routing Node Failure

If a routing node crashes then the ELB will start redirecting requests to the other (healthy) routing nodes. Meanwhile the Auto Scaler will instantiate a new routing

node to replace the failed one. The new node follows the initialisation steps described in the previous section and it will be soon fully operational. Data nodes and the coordinator are not affected by a routing node failure.

Data Node Failure

If a data node crashes, it will stop sending heartbeats to the notification service and routing nodes will be aware that the databases hosted on that data node are not operational, and will start queueing the client requests for these databases locally. The coordinator will mark the databases as non-operational and be ready to send their IDs to the next data node which requests pending databases for initialisation. Meanwhile the Auto Scaler will detect the data node failure and instantiate a replacement node, which will follow the initialisation steps from the previous section: request pending databases from the coordinator, start sending heartbeats and routing updates to the routing nodes and the coordinator. Soon the routing nodes will be aware of the new location of the databases and will start forwarding client requests that were queued.

Coordinator Failure

When a coordinator crashes, the Auto Scaler will automatically instantiate a replacement coordinator node. The new coordinator will follow the standard initialisation steps, build its routing table and start distributing non-operational database IDs when requested by the data nodes.

Composite Failure

A combination of nodes may crash at any time (including *all* of the nodes on the platform) but the recovery will always follow the steps above, with the following dependencies:

- the coordinator does not depend on any active routing/data nodes to be operational
- data nodes depend on the coordinator to be operational (so that they can receive database initialisation tasks for their hosted containers)
- routing nodes depend on the data nodes to be operational (so that they can initialise their routing tables and forward client requests to the proper database)

3 Lessons Learned

The lessons learned and best practices that emerged during the design, implementation and operation of the RDF DBaaS platform can be summarised as follows:

- *Cloud native architecture is important* – designing a DBaaS that optimally utilises the elasticity and failover capabilities of the underlying Cloud platform is significantly more challenging that just deploying a database or application on a cloud platform. Nonetheless a cloud native architecture improves the reliability, availability and performance of the DBaaS layer.

- *Micro-service architectures evolve with time* – it is challenging to design the microservices at the optimum granularity from the very beginning. To avoid the risk of designing an architecture with too many interdependent services, it is better to start with a monolithic design and gradually break down the granularity and scope of the services.
- *Design for failure* – the RDF DBaaS is designed for resilience, since the complexity of a distributed Cloud infrastructure makes failures inevitable in the long term – from lower severity problems like reduced performance of a sub-system, partial or full Cloud service failure, to high severity problems like a full datacentre failure, or even failures affecting multiple datacentres at once.

4 Future Work

The work related to Linked Data Fragments [3] is of a particular interest to our RDF DBaaS implementation, since it provides a way for utilising client-side processing in order to reduce the load on the data server. Since our DBaaS architecture is agnostic to the specific type of data server used, it will be possible to deploy Triple Pattern Fragments servers on the data nodes, instead of full-featured RDF databases. This will lead to reduced resource usage on the data nodes, improved caching of identical requests, and further cost savings for the DaPaaS platform operators.

An additional improvement of the DBaaS will be a better strategy for co-locating databases on the data nodes, based on actual database utilisation statistics (instead of the current random co-location strategy), so that databases with heavy utilisation are co-located only with databases with light utilisation. This will allow for the databases with heavy utilisation to use most efficiently the shared resources (CPU, memory and bandwidth) of the data node they are deployed on.

Acknowledgements.

Some of the work presented in this paper is partially funded by the European Commission under the 7th Framework Programme, project DaPaaS[6] (No. 610988)

References

1. Amazon Web Services. AWS Reference Architectures. Available at http://aws.amazon.com/architecture/.
2. S. Newman. Building Microservices - Designing Fine-Grained Systems. O'Reilly Media. 2015
3. R. Verborgh, O. Hartig, B. De Meester, G. Haesendonck, L. De Vocht, M. Vander Sande, R. Cyganiak, P. Colpaert, E. Mannens, and R. Van de Walle. Querying Datasets on the Web with High Availability. In *Proceedings of the 13th International Semantic Web Conference*, Riva del Garda, Italy. 2014

[6] http://project.dapaas.eu/