Cloud-Native Databases: Experimental Results (Appendix)

Josef Spillner, Giovanni Toffetti, Manuel Ramírez López

Zurich University of Applied Sciences, School of Engineering
Service Prototyping Lab (blog.zhaw.ch/icclab/)
8401 Winterthur, Switzerland
{josef.spillner,toff,ramz}@zhaw.ch

Abstract. This document is an appendix to our paper 'Cloud-Native Databases: An Application Perspective' published at the 3rd International Workshop on Cloud Adoption and Migration (CloudWays) at ESOCC 2017, September 27, Oslo, Norway [2]. It contains additional results to express the versatility of the CNDBbench testbed to assess elastic scalability, resilience, performance, multi-tenancy isolation and pricing of database services and self-managed database systems. The raw results including an open science notebook are available from `https://github.com/service prototypinglab/cndbresults`.

Note: The local experiments were conducted on an Apple MacBook Pro 12 with Intel Core i5 processor clocked at 2.7 GHz and 16 GB RAM. Running Mac OS X El Capitan 10.11.5 and Docker 1.12, the experiment stack consists additionally of the most recent Docker images at experiment time (starting August 2016), with a single configuration change to increase `max_allowed_packet` in the MySQL container as the default is not suitable for high insertion frequencies. The Docker images in use were `{mongo,crate,postgres,mysql,couchdb}:latest` which at the time of writing corresponded to `mysql:5.7.15`, `postgres:9.5.4`, `couchdb:1.6.1`, `crate:0.55.4` and `mongo:2.6.12`. On the client side, the connections were performed with the Python packages `pymongo` (with TLS for DocumentDB), `couchdb`, `crate`, `mysql.connector` and `psycopg2`.

The reference comparison queries on the dataset with 100000 entries are: equal-to with few (19985) results: `number=1`, called eq/few, and with many (100000) results: `tenant_option='F'`, called eq/many; not-equal-to with many (80015) results: `number!=1`, called neq/many, and contains with many (99952) results: `blob contains 'quam'`, called contains/many. They select based on equality, unequality, a contains relation and a full text search returning many references to files or blobs (dubbed manyfiles). The six domain-specific queries for any further measurement are composed as follows: database creation, table or collection creation, record-for-record insertion, deletion, bulk insertion, statistics retrieval. All query experiments were conducted 100 times over the course of several hours of independent runs on several days.

The experiment numbers in the sections correspond to the individually recorded experiments in the open science notebook.

0.1 Experiment 7: Database Resilience

While measuring the cost, performance and scalability requires proper use of the database, the resilience is evaluated forcing improper use and provocation of faults. Resilience as a general term refers to continued level of service in the presence of faults, slowness, interferences and other undesired influence factors. We focus on fault tolerance as the most critical factor in cloud resilience. Metrics collection on the resilience thus requires active fault provocation in order to not having to wait for actual faults, failures and incidents which, while not uncommon, are relatively rare in cloud environments [1]. Given that both local and remote database services are used, the experimental induction of faults is derived from a fault catalogue consisting of appropriate system-level and service-level faults. An excerpt of possible faults along with tools to generate them is shown in Fig. 1.

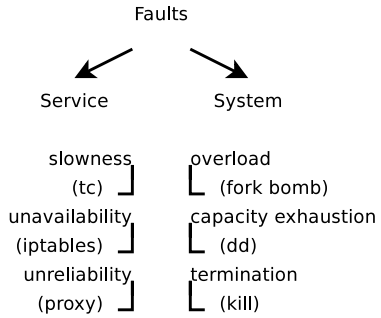

Fig. 1. Catalogue of generated faults

A major interest of application providers is how well the overall functionality and quality of experience can be maintained in the presence of faults. Fig. 2 shows what happens when in a three-node Crate database setup the two nodes which assumed the slave roles are terminated. As the node tries to re-establish connections, the response times increase and the availability of data decreases.

0.2 Experiment 8: Blob Storage Services

An important aspect when handling large documents is the targeted use of both database and blob storage services. While database systems are also capable of storing larger blobs of data, they are typically not optimised for delta transfers, caching and other storage features, and are more heavily priced. Blob storage is in particular offered commercially as a service in two flavours: real-time and archive storage. The possible combinations of what to store where in which scenarios encourages the research with a given scenario.

A comparison between managed blob storage and managed databases is given in Table 1 for the cloud provider Amazon Web Services (AWS). The monthly

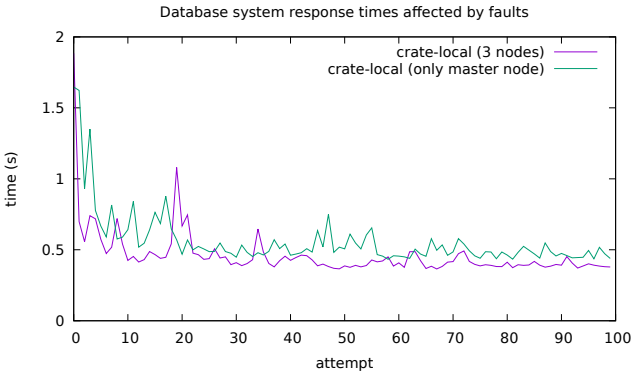

Fig. 2. Fault resistence query times comparison for Crate

price has been determined with the AWS price calculator for the Ireland (EU-West) region for a data volume of 5 GB and a small RDS instance class (t1.micro) without consideration of backup, replication or transfer cost. The processing is assumed to be 100% or 5 million mixed requests per month which corresponds to 2 requests/s on average. The results clearly show that for data which is merely stored and retrieved, blob storage, which can be referenced from table storage for large unstructured but associated data blobs, is significantly cheaper. The definitions of AWS standard and infrequent storage tariffs are provider-specific with additional constraints such as minimum billed object sizes of 128 kB for infrequent objects.[1]

Table 1. Blob storage versus database services; all prices in CHF

Service	Processing cost	+Storage cost	=Total cost
RDS MySQL	25.62	0.55	26.17
RDS PostgreSQL	27.82	0.55	28.37
S3 Standard	13.50	0.15	13.65
S3 Infrequent	27.50	0.07	27.57

For standard storage, the request cost is 0.5 µ¢ per complex request (e.g. POST) and 0.04 µ¢ per simple request (GET), and the storage cost is 3 ¢ per GB. For infrequent storage, it is more costly with 1 µ¢ per complex request and 0.1 µ¢ per simple request, although the storage cost is lower with 1.3 ¢ per GB. There is not a single minimum-cost sweet-spot but rather a continuous line of sweet spots as function of two cost parameters. The sweet spot line between standard storage and infrequent storage can thus be determined depending on those two input variables request and storage cost. Fig. 3 shows a suitable 3D

[1] AWS S3 pricing: https://aws.amazon.com/de/s3/pricing/

diagram from which the more economic choice can be derived depending on the known or predicted application behaviour.

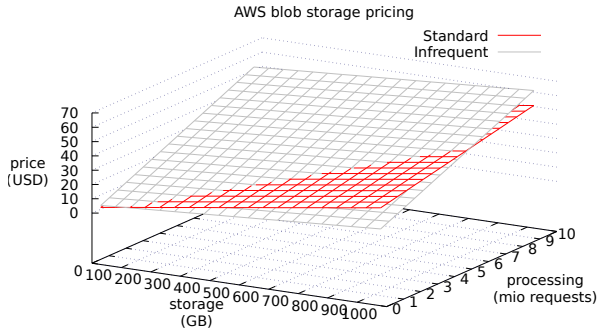

Fig. 3. Blob storage price sweetspot line as function of two variables storage size and processing load

0.3 Experiments 9–18: Database Scalability

To gain insight into the scalability behaviour, MongoDB was chosen again for clustering and compared with Crate as both are designed to be used in cloud applications. Both a single instance and a cluster were set up. The data was inserted record-for-record or all at once.

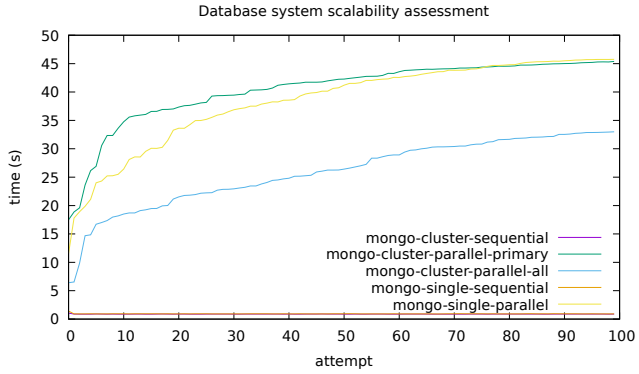

Fig. 4. Scalability comparison for MongoDB

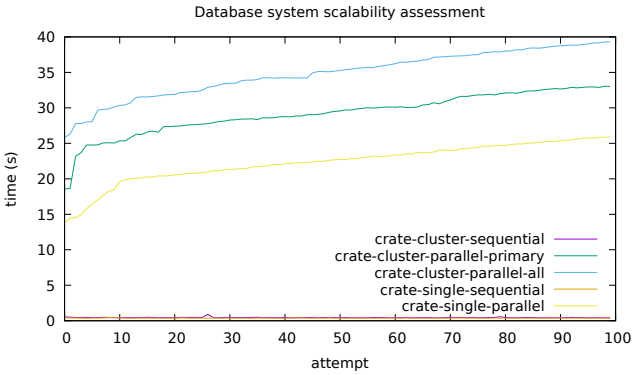

Fig. 5. Scalability comparison for Crate

In the case of a cluster with two instances, the parallel insertion point was determined to be either just the primary node or all nodes. Fig. 4 shows the results of the scaling and associated insertion combinations for MongoDB, and 5 shows the corresponding results for Crate. Several differences can be observed. With MongoDB, inserting data into all nodes of the cluster speeds up the operation, but with Crate, it slows it down. Furthermore, the growth rate is less steep with Crate.

0.4 Experiment 28: Database Pricing

Table 2 compares the pricing of self-managed databases (as part of the application, i.e. SaaS) compared to one consumed as platform service (i.e. DBaaS as part of a PaaS). The combinations are MySQL on the Google Cloud and MongoDB (DocumentDB) on Azure, and the experiment consists of the beforementioned complex query. The Google configuration is: 1 vCPU, 3.75 GB RAM, US central zone. The Azure configuration is: 1 vCPU, 3.5 GB RAM, 10000 RUs and 399000 RUs (interpolated), western Europe zone. The interpolation has been applied to achieve a comparable performance to the MongoDB container.

Table 2. Database pricing on clouds: SaaS vs. PaaS

Cloud	Setup	Specification	Performance	Price
Google	MySQL service	n1-standard	16.64 s	0.097 CHF/h
Google	MySQL container (Kubernetes)	db-n1-standard	19.29 s	0.050 CHF/h
AWS	MySQL or Aurora service (RDS)	(smallest)	(unknown)	0.178 CHF/h
AWS	MySQL container (Kubernetes)	(smallest)	(unknown)	0.294 CHF/h
Azure	DocumentDB service	10 kRU	72.30 s	0.830 CHF/h
Azure	DocumentDB service	400 kRU/i	1.05 s	32.984 CHF/h
Azure	MongoDB container (in a VM)	D1	1.05 s	0.087 CHF/h

On AWS, the price for the DBaaS is lower although the performance has not been determined. We have used the AWS billing management console which gives daily or hourly instance usage reports, but surprisingly, despite hourly billing, only monthly or daily cost exploration which exclude the free tier hidden credits. The majority of EC2 cost contributors are instance runtime (82%) and load balancing (15%), while the data transfer even with inserts caused negligible cost. On Google, self-managing the container achieves a comparable performance for a much lower price which means that the DBaaS option needs to be evaluated in a trade-off between performance, price and (not considered in the container case) availability. On Azure, the container performance is even higher despite an even lower price which makes the DBaaS option less compelling.

0.5 Experiments 20–22, 24, 26–27: Database Reliability

According to the documentation of DocumentDB, one can *turn on protocol support for MongoDB and use DocumentDB as a fully managed database service for your MongoDB app without any code changes.* Our experiments show that this statement is not true because the same queries, once executed in MongoDB and once executed in DocumentDB with MongoDB adapter, return a different number of results. Fig. 6 shows the deviations. With 100000 iterations, only one out of four queries, eq (many), returns the same number of results consistently. With 10000 iterations, two more do but one, contains (many), remains volatile in this regard.

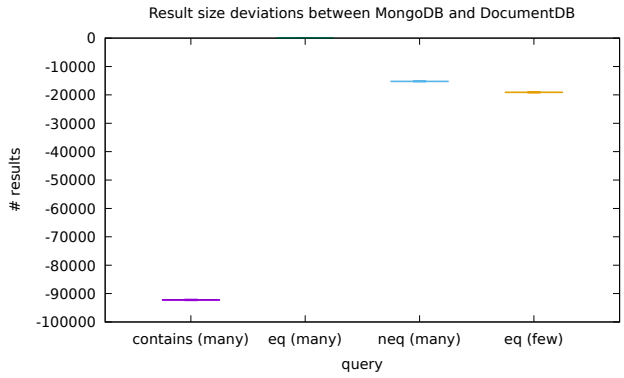

Fig. 6. Results deviations between MongoDB and DocumentDB with MongoDB interface adapter

References

1. Cérin, C., Coti, C., Delort, P., Diaz, F., Gagnaire, M., Mijic, M., Gaumer, Q., Guillaume, N., Lous, J.L., Lubiarz, S., Raffaelli, J.L., Shiozaki, K., Schauer, H.,

Smets, J.P., Séguin, L., Ville, A.: Downtime Statistics of Current Cloud Solutions – Update version. International Working Group on Cloud Computing Resiliency (March 2014)
2. Spillner, J., Toffetti, G., López, M.R.: Cloud-Native Databases: An Application Perspective. In: 3rd International Workshop on Cloud Adoption and Migration (CloudWays) (September 2017), to appear.