View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ZHAW digitalcollection

Cloud-Native Databases: An Application Perspective

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. As cloud computing technologies evolve to better support hosted software applications, software development businesses are faced with a multitude of options to migrate to the cloud. A key concern is the management of data. Research on *cloud-native applications* has guided the construction of highly elastically scalable and resilient stateless applications, while there is no corresponding concept for *cloud-native databases* yet. In particular, it is not clear what the trade-offs between using self-managed database services as part of the application and provider-managed database services are. We contribute an overview about the available options, a testbed to compare the options in a systematic way, and an analysis of selected benchmark results produced during the cloud migration of a commercial document management application.

1 State Management in Cloud-Native Applications

Cloud-native applications (CNA) are software applications which pass down beneficial cloud computing characteristics. They use cloud platform and infrastructure services to become executable, offer their own functionality as software service interfaces, are resilient against dependency service unavailability and other incidents, scale elastically with user requests, are always available on demand and are billed with a pay-per-use utility scheme without upfront cost [2]. The inherent service orientation required for CNA favours a microservices model with explicitly stateful and stateless services. The handling of data is confined to the stateful services. These must in turn be highly available and resilient to prevent loss, corruption or delay of data operations. Databases, message queues, key-value stores, filesystems and other data access models have been analysed in prior works concerning these requirements [7, 13, 14]. The desired characteristics depend on near-instant service replication [10] which implies consistent data replication and sharding mechanisms.

Fig. 1 shows a typical topology of stateful and stateless microservices orchestrated to offer a single application as a service in a highly available and resilient manner on top of plain cloud infrastructure services. Almost all approaches rely on coordinated replication which brings self-awareness about its role (e.g., master or slave) to each microservice. Furthermore, they rely on fast-spawning service

implementations (e.g., containers or light-weight hypervisors) to achieve rapid elasticity upon request spikes and instance recovery after crashes.

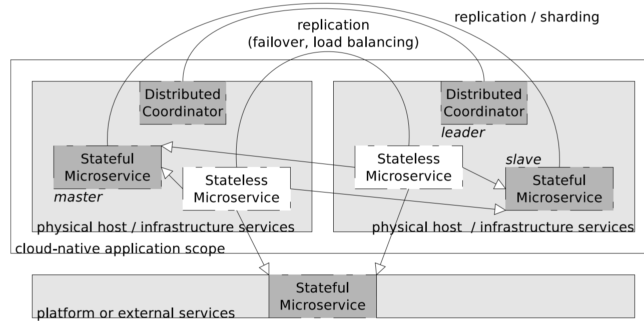


Fig. 1: Cloud-native application with internal and external data management

It is however not clear under which circumstances applications should manage data by themselves. The range of commercially offered platform-integrated stateful services is increasing. Their common value proposition can be expressed in a simplified way by *paying more to manage less*. But for a business decision, the value needs to be quantified. Due to the multitude of possible options, businesses need to obtain metrics on which such decisions can be performed. Apart from the pricing, such decisions need to account for risks and for end-to-end service provisioning quality and effort which from a software engineering perspective always includes the consideration of client-side bindings to the services.

To reduce the problem scope, we limit the research to applications which handle large structured documents in database systems. Hence, through this paper, empirical studies of how databases operated in a cloud-native context behave in commercial cloud environments are made possible. The main contribution is a testbed to measure and compare different database options from a vendor-neutral perspective. The resulting distinction between self-managed and provider-managed databases covered by the testbed is expressed in Fig. 2.

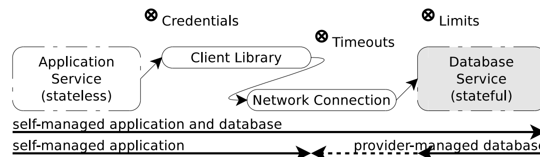


Fig. 2: Scopes of cloud-native database management

The paper is structured as follows: First, it presents the possible options for cloud-native databases, differentiating between fully managed and application-

controlled offers. Then, it defines a method to compare and rate cloud database services both on the technical and on the pricing level. The method translates into a design of a testbed whose architecture and implementation we present. Experiments we have conducted based on this method are then explained together with the obtained results. The findings from the experiments need to be interpreted in alignment with business strategies. For this reason the paper concludes with an open discussion about the strategic impact of using the testbed in a systematic way during the application engineering process.

2 Cloud-Native Database Options

Apart from conventional relational or document-centric databases, the migration of applications into the cloud and the associated new operational requirements have led to novel design choices and along with them new research challenges. Recent database models thus include encrypted, privacy-preserving and stealth databases, energy-efficient database operators and adaptive query systems over dynamically provisioned resources [8, 4]. Few of these designs have progressed beyond prototypical systems, but from an applied science perspective, we are interested in what recommendations can be given to application developers today. Hence, only conventional (relational and document-centric) database systems and services which are widely available on cloud platforms are considered along with systems commonly described as cloud-enabled or cloud-ready.

We distinguish between the choices of database hosting primarily by the responsibility. Database management systems can be managed by the application provider as part of the application (e.g. as application-controlled container), outside of the application scope itself (e.g. as a separate virtual machine whose autoscaling is determined by cloud facilities), and as fully cloud-managed service, typically named Database-as-a-Service (DBaaS). Our focus is on *cloud-native database* (CNDB) options which adhere to expectations from cloud application developers such as elastic scaling, resilience against unexpected issues, flexible multi-tenancy isolation and high performance at low price.

2.1 Self-Managed Database Systems and Microservices

The widespread proliferation of open source database management systems has led to the inclusion of these systems into software applications. The application logic then controls the lifecycle of the database, launches and terminates it as needed, and directly accesses it, often without authentication or through a single user account. Tenants in the application are in this case mapped to the database through identifiers or unprivileged separation such as tables or columns.

Cloud-native applications are often decomposed into horizontally scalable microservices where all instances are of equal importance in a peer structure. Only few database systems are currently mirroring this ability. Many still require a master-slave setup where the master instance needs to be launched before the

slave instances and must never fail, or variants thereof with multi-master replication. We analyse selected database systems concerning their use as disposable microservices in Table 1. Of these, only Crate fully conforms to this model, although a technology preview also exists for MongoDB (for master-slave replication).

Table 1: Available self-managed database microservices

Name	Relation of instances
CouchDB	master-slave and master-master replication, manual sharding
MongoDB	replica sets with master-slave replication, keyed sharding
Crate	set of peers with automated sharding upon scaling
PostgreSQL	master-slave replication, sharding through Citus
MySQL	master-slave replication, sharding through Fabric

2.2 Provider-Managed Database Services

From a cloud application perspective, it is desirable to maximise the flexibility by freely choosing among application-controllable software and managed services for the assumed database interface. Despite efforts to standardise the interfaces for database-as-a-service (DBaaS), the implementation differences are significant enough to warrant the propagation of information about the underlying database system. For instance, a developer may know how to write SQL statements but can optimise them and avoid pitfalls when knowing that the engine behind the SQL interface is in fact a MariaDB 10.2 with the XtraDB storage engine. This knowledge should be conveyed and flexibly interpreted using discoverable service descriptions, but in practice, it is often tightly coupled to the application. Furthermore, database interface and implementation options provided in the commercial cloud space vary significantly. Table 2 compares the availability of database interfaces at six public cloud (platform) providers from two countries, USA and Switzerland. Implementations marked with asterisk are available as open source and thus allocatable for local testing by application developers prior to paying for the cloud deployment.

Despite multi-database service offers by most providers, the table is sparse. This means that vendor lock-in risks need to be assessed. Furthermore, the pricing of DBaaS differs for offers with the same interface. For instance, MongoDB services are offered by Microsoft (as interface adapter to CosmosDB) and by the Swisscom Application Cloud (AC). The Swisscom offer excluding high availability starts at CHF 12 per month including 1 GB storage and 256 MB RAM. The equivalent Azure offer (hosted in Europe-West) starts at CHF 134 but includes 10 GB storage and 5 DTUs, a custom unit expressing the processing power. For an application engineer who wants to process a data volume of 1 GB, it is not clear if the cheaper offer would be performance-wise on par without further ex-

Table 2: Available provider-managed database services

Amazon Web Services	Google Cloud	Microsoft Azure	IBM Bluemix	APPUiO	Swisscom AC	Application Interface	Implementation
X[1]	X[2]					SQL	MySQL *
				X	X		MariaDB *
			X	X			PostgreSQL *
							Aurora
							Oracle DB
		X[3]					SQL Server
			X				DB2
		(X)[4]		X	X	JSON QL or similar (Mango etc.)	MongoDB *
			X[5]				CouchDB *
X							DynamoDB
		X					CosmosDB
		X				other	TableStorage
	X						Datastore
	X						BigTable (*)
X[6]	X[7]	X			X		Redis *

Notes: 1: RDS, 2: Cloud SQL, 3: Database Service, 4: CosmosDB adapter, 5: as Cloudant NoSQL, 6: as ElastiCache; 7: via external RedisLabs service

periments. There are detailed studies on cloud database services in general [11]. In contrast, our focus is on their suitability for cloud-native applications.

3 Comparison Method and Testbed

The automatable comparison of databases is rooted in two main characteristics: performance and resilience. Other metrics such as price and isolation can be derived from trace data in conjunction with external information. Several queries and transactions are run to measure the performance through an application-specific benchmark. It includes the preparation of structures (tables, collections), individual inserts, bulk inserts, queries and deletions. Furthermore, the availability is measured and in the case of self-managed database services actively impeded by controlled interference and termination, leading to data about the resilience.

As our chosen approach is to provide a testbed to compare database options, its functional and non-functional requirements need to be defined first. The functional requirements are:

1. The testbed must run itself in the target environment of the cloud-native application to yield realistic metrics with simple queries and complex transactions.
2. Both self-managed and provider-managed database services need to be supported.
3. The testbed operator must be able to choose the dataset under test, either an existing one or a synthetic one which is generated as part of the operation.

The non-functional requirements are:

1. The scale of testing needs to be configurable to balance representative and timely results. Therefore, the runtime needs to be chosen to range from mere minutes to multi-day sampling.
2. All tests need to be idempotent to allow for repetitions and statistical detection of anomalies.

3.1 Testbed Architecture and Implementation

The testbed architecture is derived from the requirements. To correlate with cloud-native applications, a containerised approach is taken. Both the testbed itself, with its performance benchmark and resilience calculation parts, and all self-managed database services are launched as container compositions. Fig. 3 visualises the technique of how the experiments are conducted by using Docker Compose as orchestrator of containers. One container contains a performance benchmark application, another one a fault provocation application, two stateful containers serve as persistent input and output volumes for the reference dataset and the results respectively, and additional containers spawn the database systems. The testbed containers allow for parameterisation through environment variables to override any values in the internal configuration file. The most important properties include binding metadata and credentials. Furthermore, the testbed supports five configurable multi-tenancy isolation levels.

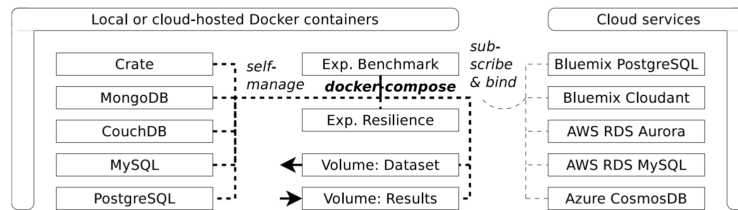


Fig. 3: Orchestrated containers and services as part of the experiment setup

Our implementation of this architecture is called CNDBbench, focusing on the benchmarking part while also containing the resilience part. It is consisting of Python classes for all supported database interfaces and the Docker image generation scripts, and is made available as open source software for use in other migration cases (see Repeatability).

3.2 Testbed Preparation: Document Management Scenario

Each instance of the testbed needs to be prepared according to application-specific needs. The guiding objective of our research has been to analyse database options for the class of cloud-native document management applications. Their requirement is storing millions of documents (e.g. scanned PDFs of dozens of MB in size) along with document metadata such as ownership, permissions, audit trails and searchable full text determined by OCR prior to insertion. From the application perspective, the design then involves stateful (database) components which are realised as bindings to database services or instances of application-controlled database microservices. Fig. 4 demonstrates a document management scenario and the possible realisation options.

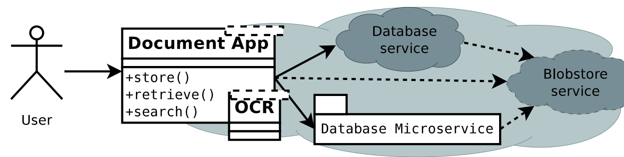


Fig. 4: Document management scenario

The reference dataset to evaluate the database choices consists of 100,000 generated entries which correspond to an actual domain-specific dataset with scanned newspaper articles. With associated metadata such as origin and access control lists, there are 1.4 million entries in total. The medium-sized data with large blob documents and structured metadata is representative for the domain of document management in the cloud through databases; alternative hybrid designs using blob storage are not considered in the present scenario. The following operations are performed to get both performance and deviation metrics: insertion of data, search and retrieval of partial data. This selection matches transactions in typical document management applications where updates and deletions happen rather sporadically.

3.3 Testbed Operation

Once the testbed is prepared, it needs to be operated in a way which most closely corresponds to the eventual operation of the application. Specifically, network delays and latencies as well as microservice execution technologies need to be properly reflected. Fig. 5 shows seven testbed configurations which correspond to all possible combinations of how to manage application data in the cloud. More variability is added by defining for the cases of application-managed databases where to physically store the data. Our research assumes attaching volume containers whereas provider-managed storage areas would be another option.

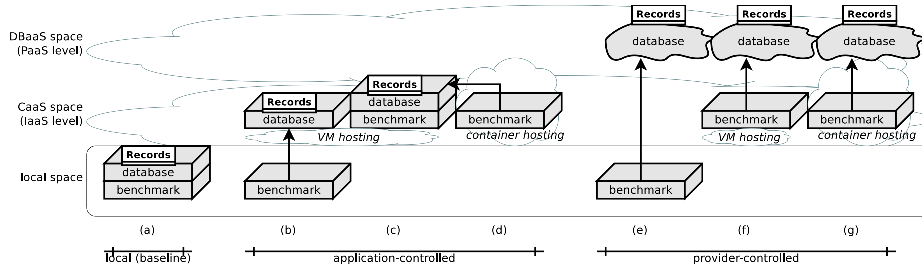


Fig. 5: Combinations of local, application-managed and provider-managed containers with application-managed and provider-managed databases

4 Selected Results

This section reports on results we have obtained from running the testbed in some of the explained operational combinations using the document management dataset. The research on the figurative *cloud-nativeness* of databases have been conducted with experiments targeting the desired technical properties of the specific application domain of document management. In total, 28 experiments have been performed and recorded, showing the versatility of CNDBbench. Selected results concerning performance, multi-tenancy flexibility and pricing will be reported. Apart from the results described here due to interesting observations, all experiments and results are analysed and described in a technical appendix to this paper (see Repeatability).

Five relational and document database systems from Table 2 have been selected for the study of the first group. They are briefly summarised in Table 3. Among those, PostgreSQL and MySQL are relational database systems (albeit with recently added JSON document processing capabilities) and have been available in early versions since the mid-1990s. CouchDB and MongoDB are often-cited representatives for document-centric systems which appeared in the late 2000s. Crate is the most recent system, created in 2014, whose focus on cloud deployments is stressed by masterless distributed operation and automatic node recovery in combination with a standard SQL-over-HTTP interface. It offers a mixed document/column store. All five systems have subtle differences in how they shard (and replicate) data.

For the second group, summarised in the bottom half of the table, three database service providers have been chosen: Amazon Web Service's Relational Database Service (RDS) with the Aurora implementation, which is a custom storage engine, in addition to the stock MySQL with its InnoDB, MyISAM and other default engines, IBM's Cloudant NoSQL and PostgreSQL service on its Bluemix platforms, which as the name suggests are a document store and a relational database, respectively, and Azure's CosmosDB née DocumentDB. An interesting observation is that even more sharding options are present which affect how well data can be managed by cloud-native applications. Interestingly, Aurora despite being a cloud service does not offer sharding for horizontal scal-

Table 3: Evaluated database system software and cloud services

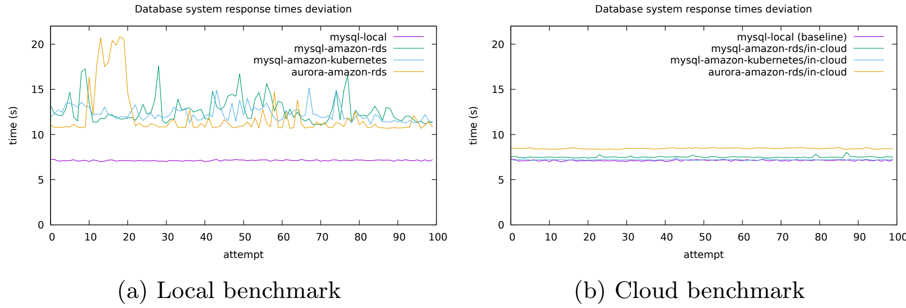
Software/Service	Data model	Runtime	Distribution
CouchDB	document	Erlang	create-sharding
MongoDB	document	C++	config-sharding
Crate	mixed-model	Java	auto-sharding
PostgreSQL	relational	C	master-sharding
MySQL	relational	C, C++	fabric-sharding
AWS RDS Aurora	relational	MySQL	read-replicas
AWS RDS MySQL	relational	MySQL	read-replicas
Azure CosmosDB	document	DocumentDB	key-sharding
Bluemix PostgreSQL	relational	PostgreSQL	failover-replicas
Bluemix Cloudant	document	CouchDB	none

ability. More variety is available at other providers, for instance Azure offering key-sharded data in CosmosDB which would otherwise resemble Cloudant.

4.1 Database Performance

The first experiment compares the deviation of response times as measure of instability between a local database system and a database system or service in the cloud, represented by AWS. A complex document management transaction consisting of six individual queries was performed with MySQL first as this system is reflected in the largest variety of cloud hosting options. The benchmark itself ran both on the local machine and as close as possible to the database, i.e. with high affinity in the cloud. Fig. 6a shows that the local queries are much faster and their response time more predictable than those of the cloud counterpart when the benchmark runs locally and thus all queries need to traverse the wide-area network. Fig. 6b contrasts the results with the affine benchmark. All such measurements are suffixed with */in-cloud*. The trivial comparison shows that a local benchmark with a local MySQL system performs equal to a Kubernetes-hosted benchmark and MySQL container pair, as both communicate via local link. As soon as the provider's services are involved, this translates into a local-area network transmission within one hosting region.

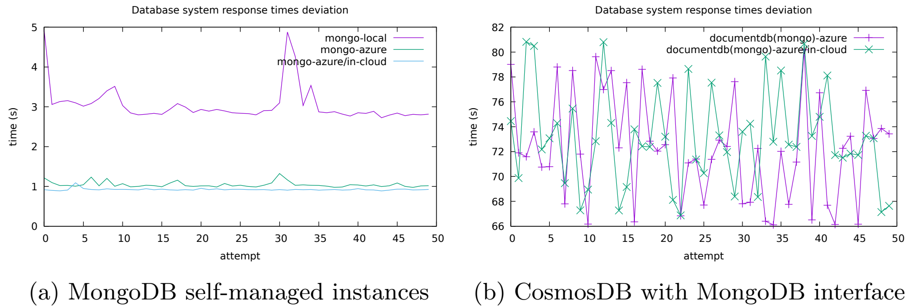
In Fig. 7a, a different set of queries was tested with MongoDB, hence the different absolute times and network delay effects. Nevertheless, the cloud-hosted database container shows a higher stability in response times with both local and cloud benchmark, while the latter also has a lower response time as expected from the observation of MySQL. The interesting difference is that the response time deviations are high for local MongoDB queries but low for local MySQL queries which suggests that not only the network influences the variation in response times. In contrast, Fig. 7b reports on the same experiments using the MongoDB adapter for CosmosDB which was conducted over two non-consecutive days. In both the local and cloud-hosted benchmark cases, the latter using an Azure VM, the performance is relatively stable within one day, vary-



(a) Local benchmark (b) Cloud benchmark

Fig. 6: Query times for MySQL

ing a lot between the days (about 33%), and extremely low compared to the native MongoDB counterparts. Additionally, Fig. 8 compares two database services from Bluemix to complete the variations in engines, providers, services and benchmark locations. The interesting observation is that not only are the absolute response times of PostgreSQL strictly below the ones of MySQL ($\bar{rt} = 0.92$ vs. 6.23), their deviation is also a lot smaller ($\sigma = 2.60$ vs. 28.32).



(a) MongoDB self-managed instances (b) CosmosDB with MongoDB interface

Fig. 7: Query times for MongoDB/CosmosDB, both local and cloud benchmarks

4.2 Database Multi-Tenancy

Data management is affected by the level of isolation between the tenants in a multi-tenant database service setup. Fig. 9 represents the model of matching isolation level to estimated performance and cost. For three out of the five different levels, we have measured the actual behaviour with three different implementations each.

Fig. 10 contains the corresponding results. The multi-threaded implementation (MT) takes longer per thread to return the results but all threads return close to each other, leading to a speedup of 22.5%, 41.9% and 59.6% over the

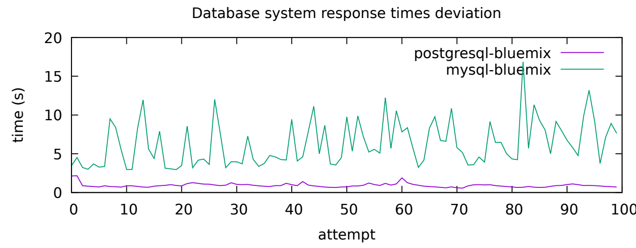


Fig. 8: Query times for MySQL and PostgreSQL, local benchmark

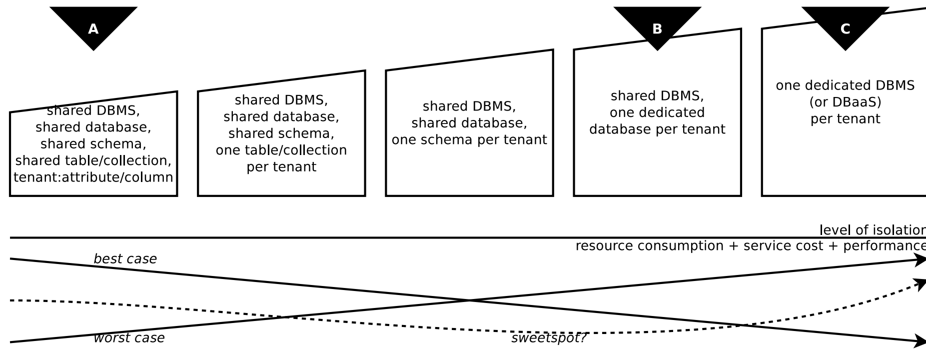


Fig. 9: Model of flexible multi-tenancy configurations for database services, with MongoDB

single-threaded implementation of A, B and C, respectively. Option C is the fastest and most isolated option, but does not represent an unconditional overall sweetspot due to also being the most expensive one.

4.3 Database Pricing

Of interest to the application provider is the total cost of provisioning in relation to a quality of experience which allows for a surplus-generating revenue. Our findings indicate that there is no clear price advantage of self-managed containers on the SaaS level versus a comparable DBaaS option, or vice-versa, when taking replicated containers for higher resilience into account. From a methodic point of view, we derive an unquantified graphical representation of pricing in relation to performance, availability/resilience, reliability, multi-tenancy and scalability as shown in Fig. 11 and propose to derive a comparison tool for application engineers.

5 Findings and Recommendations

As the selected results have shown, a general statement about a single best database option will not be possible, and a sharp definition of CNDB remains im-

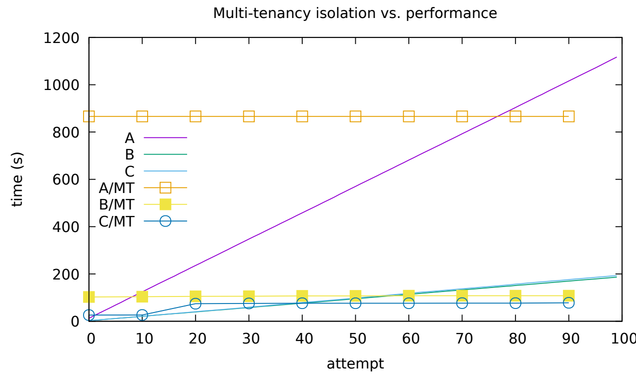


Fig. 10: Results for multi-tenancy options A, B and C with and without multi-threading

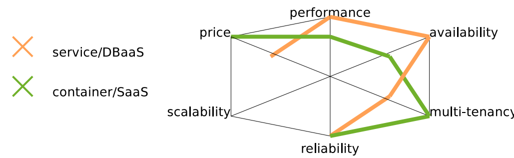


Fig. 11: Spider graph for pricing trade-offs, sampled for MySQL at Google; outside = best

possible. Our general recommendation is therefore that tools such as CNDBbench should be used in cloud application migration projects to produce metrics upon which selection decisions can be based.

Several systems and services have undocumented or undiscoverable limitations which can be revealed by systematic testing as is the case with CNDBbench. For instance, Crate only returns up to 10,000 rows by default and requires a `LIMIT` clause to return more. Azure CosmosDB limits the maximum requests to 1000 per second, which can be increased to 10,000, and requires the activation of further instances to grow beyond, despite low load on the database. Several protocols and client-side libraries are setting up timeouts. Some are merely difficult to deactivate, others even impossible, like the 20 second query timeout when inserting many records through PyMongo.

For the mentioned limitations, we recommend a discoverable description of these properties in addition to more complete documentation [6]. For the construction of future applications, assuming more maturity and choice in containerised database systems, we recommend auto-clustering microservices as currently implemented for Crate. In any case, the economics of self-managed instances depends to a large degree on the business background, including the skills and qualifications of the application engineers. In tech-savvy companies, self-managed

database containers running on top of virtual machines using container management frameworks are recommended.

6 Discussion and Conclusion

We discuss our findings in the context of recent publications about both cloud-native databases and database characteristics in the cloud in general.

Szczyrbowski and Myszor present a behaviour comparison between the Oracle Database Schema Service which offers an HTTP interface [13] and the local 11g equivalent. Their main focus is on performance stability, minimising deviations in query times for three operations: INSERT, UPDATE and SELECT. Their approach is comparable to ours apart from updates and technological choices. The findings suggest that the cloud service has a much lower deviation apart from also being (presumably due to opaque hardware differences) faster in the worst, average and best case. We were able to reproduce this for MongoDB but not for MySQL, and therefore assume that their findings cannot be generalised.

Another performance comparison is authored by Seriatos et al. [12]. The focus is on three database systems – MongoDB, Cassandra and HBase – in the BONFIRE cloud testbed. Cost and scaling are not discussed. The YCSB benchmark is used. The findings tell that each of the system performs differently depending on the workload which implies two future work directions: The first, mentioned by the authors, is the tuning of parameters; the second, added by us, is the design of adaptive multi-database connectivity as the next evolutionary step for CNDBs.

The focus on cost is set by Mian et al. in an analysis of resource configuration using the TPC-C/E/H benchmarks in three application scenarios [9]. While the authors focus on AWS EC2, the DBaaS services of the same provider are not considered. A similar aim is conveyed in the work by Floratou et al. albeit with a critical look at unpleasant surprises in terms of financial risks when using DBaaS [5]. The findings are that more expensive hourly services may turn out more cost-effective overall, which is substantiated with observations of MySQL and SQL Server running on local hardware. The authors propose a benchmark-as-a-service for application developers (as database users). To cover the scaling and resilience characteristics which are important in a cloud setting, Bagui et al. look at sharding techniques and propose an implementation [1]. The work is demonstrated with MySQL and extends to other engines. Costa et al. examined partial database migration to the cloud [3]. The migration path in this work is from local PostgreSQL to AWS DynamoDB without giving up the former by adding a transparent adapter to the application. The finding is that scalability bottlenecks can be circumvented by offloading data to DynamoDB. While we have not analysed the same system, our results with non-ACID confirm this observation.

Table 4 summarises which of the cloud database properties were covered by related works and whether our findings agree (☑) or disagree(⊗) with them.

When the results are not clear, the need for future experimental research (⚙) is shown instead. The lack of a reusable testbed from the related work is evident.

Table 4: Related work comparison

Study	Performance	Scalability	Resilience	Tenancy	Price	Testbed
Szczyrbowski et al. [13]	✖					
Seriatos et al. [12]	⚙					
Mian et al. [9]					⚙	
Floratou et al. [5]					⚙	
Bagui et al. [1]		⚙	⚙			
Costa et al. [3]		☑				

We conclude that cloud-native databases are a challenging topic in need of more formal expressions concerning their configuration and characteristics and of more experiments. We suggest that future research should be directed towards a holistic approach of assessing flexible database options in the cloud which involve self-hosted data containers, blob storage services and DBaaS.

Repeatability

Our benchmark implementation, CNDBbench, is publicly available to repeat our experiments. For reference and reproducibility of the results, the experiment setup including hardware specifications and instructions is given in detail in a raw open science notebook which is made available together with a technical appendix due to the page number limitation. The notebook also contains reference results, additional experiments and findings concerning resilience, scalability and pricing[1,2]. We encourage the critical examination and re-use of the datasets.

Acknowledgements

This research has been funded by the Swiss Commission for Technology and Innovation (CTI) in project ARKIS/18992.1. It has also been supported by an AWS in Education Research Grant, an IBM Academic Initiative for Cloud offer, a Microsoft Azure Research Award and a Google Cloud credit, all of which helped us to conduct our experiments on public commercial cloud environments.

[1] CNDBbench: https://github.com/serviceprototypinglab/cndbbench
[2] CNDBresults: https://github.com/serviceprototypinglab/cndbresults

References

1. Bagui, S., Nguyen, L.T.: Database Sharding: To Provide Fault Tolerance and Scalability of Big Data on the Cloud. International Journal of Cloud Applications and Computing (IJCAC) 5(2), 36–52 (2015)
2. Brunner, S., Blöchlinger, M., Toffetti, G., Spillner, J., Bohnert, T.M.: Experimental Evaluation of the Cloud-Native Application Design. In: 4th International Workshop on Clouds and (eScience) Applications Management (CloudAM). Limassol, Cyprus (December 2015)
3. Costa, C.H., Maia, P.H., Mendonça, N.C., Rocha, L.S.: Supporting Partial Database Migration to the Cloud Using Non-intrusive Software Adaptations: An Experience Report. In: 4th ESOCC. CCIS, vol. 567. Taormina, Italy (September 2015)
4. Costa, C.M., Leite, C.R.M., Sousa, A.L.: Efficient SQL adaptive query processing in cloud databases systems. In: IEEE EAIS. pp. 114–121. Natal, Brazil (May 2016)
5. Floratou, A., Patel, J.M., Lang, W., Halverson, A.: When Free Is Not Really Free: What Does It Cost to Run a Database Workload in the Cloud? In: Topics in Performance Evaluation, Measurement and Characterization – Third TPC Technology Conference (TPCTC). LNCS, vol. 7144. Seattle, Washington, USA (August 2011)
6. Frey, S., Hasselbring, W., Schnoor, B.: Automatic Conformance Checking for Migrating Software Systems to Cloud Infrastructures and Platforms. J. Softw. Evol. and Proc. 25(10), 1089–1115 (October 2013)
7. Goldschmidt, T., Jansen, A., Koziolek, H., Doppelhamer, J., Breivold, H.P.: Scalability and Robustness of Time-Series Databases for Cloud-Native Monitoring of Industrial Processes. In: 7th IEEE International Conference on Cloud Computing (CLOUD). pp. 602–609. Anchorage, Alaska, USA (July 2014)
8. Götz, S., Ilsche, T., Cardoso, J., Spillner, J., Kissinger, T., Aßmann, U., Lehner, W., Nagel, W.E., Schill, A.: Energy-Efficient Databases using Sweet Spot Frequencies. In: 1st International Workshop on Green Cloud Computing (GCC). pp. 871–876. London, UK (December 2014)
9. Mian, R., Martin, P., Zulkernine, F.H., Vázquez-Poletti, J.L.: Cost-Effective Resource Configurations for Multi-Tenant Database Systems in Public Clouds. International Journal of Cloud Applications and Computing (IJCAC) 5(2), 1–22 (2015)
10. Nguyen, H., Shen, Z., Gu, X., Subbiah, S., Wilkes, J.: AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service. In: 10th International Conference on Autonomic Computing (ICAC). pp. 69–82. San Jose, California, USA (June 2013)
11. Sakr, S.: Cloud-hosted databases: technologies, challenges and opportunities. Cluster Computing 17(2), 487–502 (2014)
12. Seriatos, G., Kousiouris, G., Menychtas, A., Kyriazis, D., Varvarigou, T.A.: Comparison of Database and Workload Types Performance in Cloud Environments. In: First International Workshop on Algorithmic Aspects of Cloud Computing (ALGOCLOUD). LNCS, vol. 9511, pp. 138–150. Patras, Greece (September 2015)
13. Szczyrbowski, M., Myszor, D.: Comparison of the Behaviour of Local Databases and Databases Located in the Cloud. In: 12th International Conference on Beyond Databases, Architectures and Structures. Advanced Technologies for Data Mining and Knowledge Discovery. CCIS, vol. 613, pp. 253–261. Ustroń, Poland (May 2016)
14. Wiese, L.: Advanced Data Management for SQL, NoSQL, Cloud and Distributed Databases. DeGruyter/Oldenbourg (2015)