Elastic Query Processing on Function as a Service Platforms

Thomas Bodner
supervised by Hasso Plattner
Enterprise Platform and Integration Concepts Group
Hasso Plattner Institute
Potsdam, Germany
thomas.bodner@hpi.de

ABSTRACT

Modern analytics workloads are not predictable anymore and require database systems to be able to adapt to their performance demands and cost constraints in an instant. Existing database architectures, however, are incapable of meeting this degree of elastic scalability. For this reason, we propose a novel architecture based on function as a service platforms. This architecture and the concepts surrounding it are being implemented in our research prototype Skyrise.

1. INTRODUCTION

Enterprises are increasingly employing modern analytics applications to gain insights from their data and to make timely and well-informed decisions about their businesses. These applications are often interactive in nature and they involve exploratory analysis on data of different sizes and shapes. The resulting query workloads for database systems are hard, if not practically impossible, to predict. Vendors of commonly used data warehouse systems report customers running queries in infrequent bursts over data with sizes varying as much as nine orders of magnitude [21]. It is difficult to provision infrastructure and to optimize query execution for such workloads in order to achieve sufficient performance and cost efficiency.

Traditional OLAP database systems are mostly based on a distributed shared-nothing architecture [13]. Persistent data is partitioned across the database nodes and stored locally. This enables scalable and efficient processing of static workloads. When system load or data relevancy, however, change over time, nodes need to be added or removed from the system and data need to be (re)partitioned or (un)loaded. All of this is costly and time consuming, rendering shared-nothing architectures inflexible and thus unfit for modern analytics workloads, even when deployed on flexible public cloud infrastructures.

Recent analytical database systems, sometimes referred to as *cloud-native*, adopt a shared-disk architecture [10]. This accounts for two aspects of modern cloud infrastructures. First, the storage is separated from the compute resources and exposed as a shared medium. Second, the bandwidth to remote storage is comparable to that of local disks [7]. Query processing can be done in a shared-nothing fashion and efficiency while all persisted data is read from remote shared storage. This removes the need for expensive data shuffles and loads on workload changes, letting shared-disk architectures scale more elastically and gracefully. They can quickly shut down compute resources entirely when idle and again ramp them up when there is load. While doing so is significantly faster than in shared-nothing systems, it still takes minutes at best [20]. The time is spent configuring and launching potentially large virtual machine instances and is still too long for interactive applications, which demand query latencies in seconds rather than minutes. So, these systems are also prone to under- or over-provisioning.

In our view, the inherent issue of the current approaches to supporting interactive ad-hoc analytics lies in them relying on coarse-grained compute resources for their workers. There will always be a set of workloads, in which these approaches will either be too slow or too expensive.

Fortunately, public cloud providers today offer means to allocate and bill finer-grained units of compute resources via function as a service (FaaS) platforms, such as AWS Lambda [4], Google Cloud Functions, [12], and Microsoft Azure Functions [17]. FaaS platforms let cloud consumers write pieces of code, so-called *cloud functions*, and run them on tiny, short-lived, and stateless workers. These platforms transparently schedule, load balance, and scale consumers' cloud functions across potentially tens of thousands of such workers with considerable combined performance. Furthermore, the small workers can be spawned in milliseconds [1] and help handling stragglers in parallel data processing [18]. FaaS platforms are economically viable for consumers, when moderately utilized. In AWS, cloud functions are currently priced at ~$0.06 per GB-hour of execution and billed per 100ms interval. This is two to eight times more expensive than conventional VMs [2, 4].

Prior work explored FaaS platforms as a foundation for general large-scale data analysis [15, 16, 19]. In this work, we study them in the context of relational query processing for interactive ad-hoc analytics, which is required by many enterprises. We are building the OLAP database system Skyrise, which exploits the elasticity of FaaS platforms to balance performance and cost efficiency, when state-of-the-art systems fail to do so.

Proceedings of the VLDB 2020 PhD Workshop, August 31st, 2020. Tokyo, Japan. Copyright (C) 2020 for this paper by its authors. Copying permitted for private and academic purposes.

CEUR-WS.org/Vol-2652/paper12.pdf

2. RESEARCH CHALLENGES

Function service platforms are a promising foundation for an elastic query processing system. They, however, do not only present opportunities, but also a variety of challenges that must be overcome in order to arrive at a system with adequate performance, reliability, and cost efficiency [14]. These platforms are subject to active research [22, 23], and while we expect some of the outcomes to work in our favor, others will not be sensible to the specific requirements of a database system.

Skyrise uses FaaS-based compute resources and shared disaggregated storage for its query execution. In its query optimization, Skyrise aims to exploit the parallelism of FaaS platforms. It further incorporates the pricing model of the cloud provider as well as user constrains to trade off cost and performance. As a result, we identify the following research challenges, which we categorize into three areas.

Cloud Functions

1. Resource limits: Cloud functions have tight resource constraints, e.g., 2 vCPUs, 3GB RAM, and a capped 15min runtime in AWS Lambda [4]. These resources must be utilized for efficiency, but are quickly exhausted. The consequences may be slow or failing executions.

2. Launch overheads: Before a cloud function is executed, it needs to be invoked and initialized. The invocation is a call to a web-based REST API. The latency of this call depends on the geographical distance between the caller and the function service. The initialization may involve provisioning a host, placing a worker on this host, and setting up the execution environment. This depends on service configuration, prior and present function concurrency, function language, instance size, and binary package size. Launching many functions to process a query in parallel may take 10s of seconds [15], thwarting interactive query latencies.

3. Indirect communication: FaaS providers disable inbound network connections to cloud functions. This requires functions to communicate via shared storage, which in turn impedes performance, particularly for data exchanging operations during query processing.

4. Observability: Function services are blackboxes to their users, making it difficult to observe and, thus, control function execution. This is problematic for distributed query processing systems, as they need to cope with operator instances straggling for various reasons, e.g., resource contention or data skew.

5. Fault tolerance: Function services guarantee to retry failed function executions. Functions, however, may have side effects. Partial or multiple executions of them may produce inadvertent results that need to be dealt with or prevented in the first place.

Disaggregated Object Storage

6. Efficiency: Current object storage services [5] show high request latencies and incur significant per-request costs. As such, they require careful handling for query processing on top of them to still be efficient [19].

7. Data consistency: These services provide only weak consistency guarantees. Individual keys may not be readable immediately after they are written. Multiple keys cannot be written together atomically. Both are guarantees commonly relied upon by database systems for query processing.

Query Optimization

8. Cost-Awareness: Trading off cost and performance of queries in a FaaS-based query processing system is non-trivial. Cloud service pricing models are complex. And, query price and performance correlate positively in some processing aspects and negatively in others.

9. Parallel Performance: To benefit from the parallelism offered by FaaS platforms, a database system's query optimizer needs to consider plans that lend themselves to parallel execution. In order to better explore the large search space of parallel plans, the optimizer itself should be parallelized. Usually, the optimizer has to compete for resources with the execution engine, and as a result, has tight computation budgets. However, in our setting, the optimizer can scale out to separate resources via cloud functions.

3. RELATED WORK

For our research, we identify two lines of related work. First, there are systems for generic data analysis that build upon FaaS platforms. They aim to provide the operational simplicity, elastic scalability, and fine-grained billing model of function services to users from a diverse set of domains. Second, relational OLAP database systems with a shared-disk architecture or extension serve our target user group and aspire to be sufficiently elastic for most workloads.

	Disaggregated Storage	FaaS-based Compute	Relational OLAP	Query Cost-Performance
PyWren	✓	✓	✗	✗
Flint	✓	✓	✗	✗
Locus	✓	✓	✗	✓
Amazon Redshift	✗	✗	✓	✗
Redshift Spectrum	✓	✗*	✓	✗
Snowflake	✓	✗	✓	✗
Skyrise	✓	✓	✓	✓

Table 1: Comparing cloud data analysis systems.

FaaS-based Data Analysis Systems

To our knowledge, PyWren [15] is the first system to provide the simple and generic data-parallel programming model MapReduce on top of a FaaS platform. Flint [16] extends on this by exposing the richer set of data-parallel programming primitives from Spark. And, finally, Locus [19] builds on PyWren and adds a distinct data exchange operator that can work with different object stores. It can further reason about the cost-performance trade-off between these stores. Compared to Skyrise, none of these systems aims to provide a complete set of relational execution operators.

Cloud-based OLAP Database Systems

Amazon Redshift [13] is a parallel data warehouse system that is based a on distributed shared-nothing architecture. It recently has been extended by an additional architectural layer named Spectrum, which operates on disaggregated storage [9]. This way, Redshift can scale elastically beyond its set of shared-nothing nodes. Interestingly, the Spectrum workers are tiny and stateless, and do not communicate. They are not explicitly said to be cloud functions, but they seem to present similar challenges. Snowflake [10] is built on a shared-disk architecture and reads persistent data from remote object storage. For intermediate data during query processing, it adds a low-latency storage layer that is hosted on its compute nodes. It can further maintain a pool of pre-launched nodes to scale faster at additional expense [21]. Both systems aim to support elastic query processing, but need to launch additional VM-based nodes at some point. Also, neither Redshift nor Snowflake seem to offer an automatic way to navigate the cost-performance space in query processing.

We see our FaaS-based OLAP database system Skyrise as a unique mix of features from above systems (cf. Table 1). As such, it presents interesting angles of research to pursue.

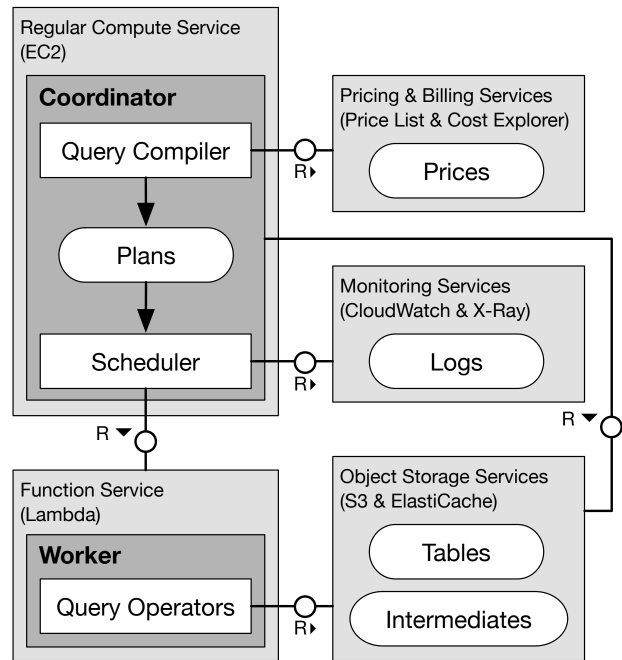

Figure 1: Skyrise Target Architecture.

4. RESEARCH STATUS

In this section, we describe our research prototype Skyrise, which we use to identify, study, and address the problems on our research agenda. We start with a description of our target architecture for Skyrise. We continue talking about our currently conducted work on its query execution engine and finish sketching our vision for its query optimizer.

4.1 Skyrise Target Architecture

Skyrise is a relational OLAP database system with a FaaS-based, shared-disk architecture. Figure 1 shows the central components in Skyrise's target architecture. We start with a brief discussion of their interactions along the lifecycle of a query. Then, we go on to highlight key design decisions for them. Note that the depicted cloud services are the ones from AWS that the current implementation is built against.

Users send SQL queries to a coordinator that runs on a regular VM [2], either embedded into an application or as a dedicated server process. The coordinator compiles the queries to optimized execution plans. For the optimization, it takes both data statistics and service prices into account. The latter are queried from the pricing and billing services of the cloud provider [3]. Then, the coordinator schedules the respective query operator functions on the function service. Once running on the function service, the query operators interact with the object storage services to consume their input data and produce their output data. They report their health and status to monitoring services as they progress. Upon eventual completion of queries, the coordinator reads their results and sends them back to the users.

4.2 Skyrise Query Engine

The Skyrise query execution engine is designed to cope with the challenges incurred by the usage of cloud functions and disaggregated object storage, as identified in Section 2.

Query Operators

The query operators are implemented in C++ to allow for efficient utilization of their limited resources. They manage their memory and they employ multi-threading to utilize the available intra-function CPU parallelism.

To keep coldstart latency low, the operator binaries are linked against the library versions of Lambda's execution environment, such that custom versions do not need to be provided in the deployment packages. Unneeded symbols are stripped from the binaries. And, we further intend to study the tradeoff between per-operator and monolithic all-operator functions on coldstart likelihood and duration.

To guarantee correct behavior under failure and retried execution, the operators are idempotent. They produce a single deterministic result object that is written atomically to object storage.

Scheduler

The scheduler employs a set of techniques to mitigate cloud function invocation and initialization latencies.

It invokes functions in parallel to hide the latencies of the REST API calls. It is also able to instruct functions to call themselves recursively, effectively distributing the task of function invocation.

To avoid function initialization entirely, the scheduler can speculatively invoke and reinvoke functions at additional costs. We plan to inform this process with knowledge from the query compiler. We also want to compare this to means by cloud providers to keep functions warm automatically [6].

Operator Communication

In FaaS-based query execution, the naïve way for operators to communicate is to materialize their outputs to shared storage, and to start all instances of an individual operator only when all instances of the preceding operator finished.

The Skyrise query engine improves on this. Along typically pipelined plan fragments, a succeeding operator's instances are started early upon partial availability of intermediate results. We plan to explore, whether some materialization points can be avoided via monolithic operator functions. We also want to study late materialization in this context.

Access to Persistent and Intermediate Data

The execution engine supports columnar and compressed file formats, such as Apache Parquet [8], to efficiently process persistent and intermediate data. We intend to improve in this area in three ways. First, we will investigate auxiliary data structures for pruning persistent data. Second, we will look into recently added capabilities of object stores to push-down query logic, such as predicates. And third, we will explore the usage of multiple storage tiers for intermediate data storage as done by Locus.

To prevent potential data consistency issues, when using current object storage services, we devise a set of measures. For the read-your-own-write issue, we let our operators only finish after they have polled the storage service long enough to verify the existence of their outputs. For the atomic multi-key write issue, we are considering the introduction of metadata files that reference multiple data files and can themselves be written atomically.

4.3 Skyrise Query Optimizer

In Skyrise, the query optimizer is the major component for balancing performance and cost, down to the granularity of individual queries. It is also key to exploiting the available parallelism of the underlying FaaS platform. For this query optimizer, we will investigate a distinguishing feature set.

It should be aware of the fact that it is targeting a FaaS-based execution environment. As such, it should know about potential degrees of freedom (e.g., fusing vs. splitting cloud functions [11], sizing function instances, or pre-allocating them). It should further know about constraints (e.g., upper function instance sizes [9]). Next to the typical statistics-based performance estimations, it also needs to conduct pricing-based cost estimations and trade them off in a meaningful way. Lastly, it should consider plans that are fit to this execution environment, e.g., ones that avoid data exchanges or introduce additional opportunities for parallelism [24].

5. CONCLUSION

We discussed our current work as well as future work plans towards our vision for elastic query processing on function as a service platforms. We presented our research prototype Skyrise and the approaches that we are pursuing with it to overcome the challenges for FaaS-based query execution. These lie in the areas of current cloud function and object storage services, and in optimizing queries for performance and cost efficiency in modern cloud infrastructures.

6. REFERENCES

[1] A. Agache, M. Brooker, A. Florescu, et al. Firecracker: Lightweight virtualization for serverless applications. In *USENIX NSDI*, pages 419–434, 2020.

[2] Amazon Inc. Amazon EC2. `https://aws.amazon.com/ec2/`, 2020.

[3] Amazon Inc. AWS Billing and Cost Management. `https://docs.aws.amazon.com/account-billing/`, 2020.

[4] Amazon Inc. AWS Lambda. `https://aws.amazon.com/lambda/`, 2020.

[5] Amazon Inc. Cloud Storage on AWS. `https://aws.amazon.com/products/storage/`, 2020.

[6] Amazon Inc. Managing concurrency for a Lambda function. `https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html`, 2020.

[7] G. Ananthanarayanan, A. Ghodsi, S. Shenker, et al. Disk-locality in datacenter computing considered irrelevant. In *USENIX HotOS*, 2011.

[8] Apache Software Foundation. Apache Parquet. `https://parquet.apache.org/`, 2020.

[9] M. Cai, M. Grund, A. Gupta, et al. Integrated querying of SQL database data and S3 data in Amazon Redshift. *IEEE Data Engineering Bulletin*, 41(2):82–90, 2018.

[10] B. Dageville, T. Cruanes, M. Zukowski, et al. The Snowflake elastic data warehouse. In *ACM SIGMOD*, pages 215–226, 2016.

[11] T. Elgamal. Costless: Optimizing cost of serverless computing through function fusion and placement. In *IEEE/ACM SEC*, pages 300–312, 2018.

[12] Google LLC. Google Cloud Functions. `https://cloud.google.com/functions/`, 2020.

[13] A. Gupta, D. Agarwal, D. Tan, et al. Amazon Redshift and the case for simpler data warehouses. In *ACM SIGMOD*, pages 1917–1923, 2015.

[14] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, et al. Serverless computing: One step forward, two steps back. In *CIDR*, 2019.

[15] E. Jonas, Q. Pu, S. Venkataraman, et al. Occupy the cloud: Distributed computing for the 99%. In *ACM SoCC*, pages 445–451, 2017.

[16] Y. Kim and J. Lin. Serverless data analytics with flint. In *IEEE CLOUD*, pages 451–455, 2018.

[17] Microsoft Corp. Azure Functions. `https://azure.microsoft.com/services/functions/`, 2020.

[18] K. Ousterhout, A. Panda, J. Rosen, et al. The case for tiny tasks in compute clusters. In *USENIX HotOS*, 2013.

[19] Q. Pu, S. Venkataraman, and I. Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *USENIX NSDI*, pages 193–206, 2019.

[20] J. Tan, T. Ghanem, M. Perron, et al. Choosing a cloud DBMS: architectures and tradeoffs. *PVLDB*, 12(12):2170–2182, 2019.

[21] M. Vuppalapati, J. Miron, R. Agarwal, et al. Building an elastic query engine on disaggregated storage. In *USENIX NSDI*, pages 449–462, 2020.

[22] C. Wu, V. Sreekanti, and J. M. Hellerstein. Autoscaling tiered cloud storage in anna. *PVLDB*, 12(6):624–638, 2019.

[23] T. Zhang, D. Xie, F. Li, et al. Narrowing the gap between serverless and its state with storage functions. In *ACM SoCC*, pages 1–12, 2019.

[24] J. Zhou, P. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *IEEE ICDE*, pages 1060–1071, 2010.