Ephemeral Per-query Engines for Serverless Analytics

Michael Wawrzoniak[1], Rodrigo Bruno[2], Ana Klimovic[1] and Gustavo Alonso[1]

[1]Systems Group, Computer Science Department, ETH Zürich, Switzerland
[2]INESC-ID/Técnico, U. Lisboa

Abstract

We challenge the common assumption that queries are submitted to a pre-configured, already running engine and put forward the idea of dynamically instantiating a chosen data processing engine upon query submission by leveraging Function-as-a-Service (FaaS) platforms. We demonstrate the idea by running *unmodified* data processing engines (we use Apache Drill as an initial example) on real-world serverless FaaS platforms and show that such engines can be instantiated on demand when a query arrives. We aim to eventually support a wide range of queries and workloads. Wide access to such functionality would be a game changer in data processing. First, it would enable pay-per-query models supporting sporadic, interactive data analysis on arbitrary engines. Second, it would significantly increase the flexibility for data processing by enabling the possibility of dynamically choosing the actual engine, its configuration, and the resource allocation on a per-query basis. Logically, this amounts to dynamically attaching a query engine to the query rather than sending the query to a pre-configured and already deployed engine. In this paper we elaborate on this vision, outline the design of the MetaQ prototype that we are building to explore the idea, demonstrate that it is realistic through initial experiments, and discuss its many exciting practical implications.

Keywords

Serverless, Data Analytics, Functions-as-a-Service

1. Introduction

Operating a long-running query engine has several limitations. First, it generates costs even if it is idle. Second, most distributed query engines lack elasticity, which leads to deployments being over-provisioned to cope with potential peak loads [1, 2, 3]. And third, as workload diversity increases, each query might benefit from a different configuration and/or engine deployment (e.g., involving accelerators, caches, parallelism level, etc.), resulting in the engine often running in a less than optimal setting for most queries [4, 5].

In this paper we explore an ambitious and radically new design: one in which we take advantage of serverless computing to provide *ephemeral per-query engines* (EPQE), i.e., query engines dynamically instantiated for each query and discarded upon completion. The ultimate goal is to be able to select the optimal engine and configuration on a per-query basis, to eliminate the inefficiencies of using all-purpose configurations and resource overprovisioning.

In the EPQE paradigm, given a query, a query engine is instantiated (potentially selected from a variety of engines) in the best possible configuration and deployment for the query, the query is executed by the engine, and upon completion, the engine is shut down (unless there is a reason to keep it running, like a similar query arriving while the engine is active). This eliminates the need for dynamic elasticity in the engine. Every query gets an engine deployed on just the resources it needs (e.g., nodes, memory, bandwidth, CPUs). This also simplifies engine deployment (since the engine can be instantiated specifically for the query at hand, e.g., maximizing data source locality) and removes the need for auto-tuning [5] of long-running engines (the engine settings need to be optimized only for the given query, which allows for more specialized and efficient solutions [6]). The approach also eliminates the problem of idle resources since if there is no query, there is no engine running. Finally, another crucial aspect of the idea is the possibility of selecting among different data processing engines on a per-query basis. This opens up the opportunity to use different engines depending on factors like data types (e.g., relational, semi-structured, graphs), file formats used (e.g., Arrow, Parquet, CVS, JSON, etc.), expected performance (e.g., based on previous profiling), feature set (e.g., availability of required statistical functions), or suitability to the overall task (e.g., when the query is a step in an ML pipeline). The idea resembles unikernel operating systems [7] where, for each application, a specialized operating system is constructed (e.g. from a library operating system [8]) and instantiated, already optimized for the application.

The vision of EPQE is enabled by the emergence of

Joint Workshops at 49th International Conference on Very Large Data Bases (VLDBW'23) — Workshop on Serverless Data Analytics (SDA'23), August 28 - September 1, 2023, Vancouver, Canada
michalw@inf.ethz.ch (M. Wawrzoniak); rodrigo.bruno@tecnico.ulisboa.pt (R. Bruno); aklimovic@ethz.ch (A. Klimovic); alonso@inf.ethz.ch (G. Alonso)
0000-0002-1304-8420 (M. Wawrzoniak); 0000-0003-1578-5149 (R. Bruno); 0000-0001-8559-0529 (A. Klimovic); 0000-0002-4396-6695 (G. Alonso)

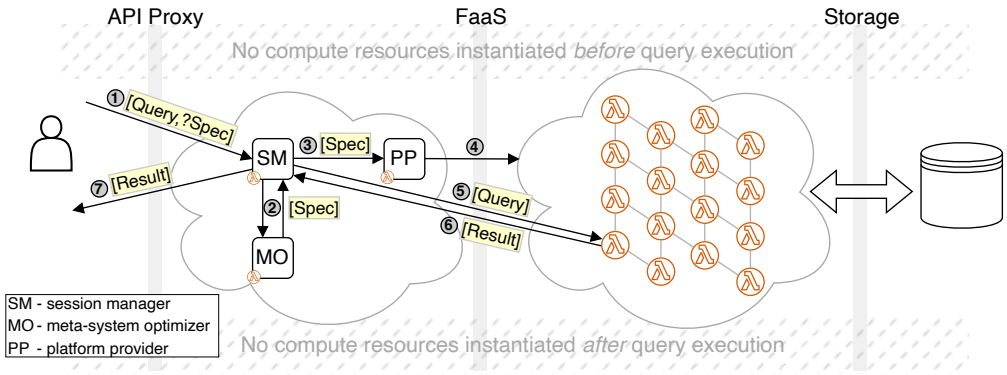

Figure 1: MetaQ system prototype mode of operation (see Section 2 for details).

Function as a Service (FaaS). In serverless computing, users deploy and invoke fine-grain functions on-demand [9, 10]. There are three main characteristics of serverless that can help in realizing the EPQE idea. First, thanks to lightweight VM system infrastructure [11, 12], functions can be instantiated quickly. For example, in AWS Lambda [13], function cold start initialization latency is ∼200ms. Such fast resource instantiation times allow starting a new engine for a query without contributing significantly to the overall execution time. Second, individual functions can be deployed with different CPU and memory configurations. Furthermore, thousands of functions can be instantiated in parallel. Such a level of resource availability and configurability allows us to right-size and right-configure engines at per-query granularity. Finally, FaaS platforms provide fine-grained resource accounting (e.g., AWS Lambda users pay at microsecond granularity), aligning the costs of the EPQEs to the work done and can play a role in deciding which engine to instantiate.

However, despite their advantages, today's FaaS serverless platforms are not adequate for general data processing [14, 15] since running queries often requires features that are missing, such as caching, support for direct communication among functions, and persistent state. This is the result of a conscious choice by providers who bundle functions with a very restricted programming model based on network-isolated, event-triggered modules composable into larger systems through workflow-based orchestration services [16, 17]. To overcome this mismatch, a significant research effort is underway. One approach involves redesigning serverless platforms from scratch and developing a completely new FaaS platform to be run on VMs (e.g., Anna key-value store [18]). Another approach relies on commercial serverless FaaS offerings and tries to overcome some of the platform shortcomings from a data processing [19, 20] or ML [21, 22] perspective. These systems propose, among other things, complex ways to reduce the overhead of communicating through cloud storage, clever optimizations to minimize the amount of data exchanged, and suggest algorithms to reduce the impact of start-up times as the number of functions needed grows. In addition, there are efforts to leverage commercial serverless FaaS offerings to provide caching and storage services to data center applications running outside of the serverless functions [23, 24].

Unlike these existing efforts that build custom experimental FaaS query engines to circumvent the limitations of serverless platforms, our approach is to leverage existing serverless infrastructure to run unmodified state-of-the-art data processing systems. By including existing unmodified engines, we will be able to take advantage of their wide variety, feature completeness, and years of effort put into their development and optimization. However, all of the real-world FaaS platforms that we are aware of do not provide execution environments that support running off-the-shelf distributed query engines. Our approach is to leverage an evolution of the Boxer [25] system, which aims to overcome FaaS limitations (e.g., by enabling inter-function networking) to provide an execution environment on top of existing commercial FaaS platforms (such as AWS Lambda) that matches the requirements of unmodified off-the-shelf query engines.

To explore the feasibility of the EPQE concept, in this paper we investigate whether (1) it is already possible to run existing query engines on a commercial serverless system (AWS Lambda); (2) whether the resulting performance is acceptable since existing distributed query engines have not been originally designed to operate on top of serverless functions; and (3) get an initial idea of whether selecting engines on a per-query basis would bring an advantage. We build a prototype system, *MetaQ*, as a way to realize the EPQE model and conduct a feasibility study.

Our focus is on distributed data processing platforms, such as Apache Spark or Apache Drill, instead of traditional database engines, such as PostgreSQL or MySQL. We do not analyze the cost tradeoffs of using AWS Lambda for data analytics, previous works [19, 20] established that serverless can reduce costs for bursty query workloads. In particular, steady, similar, high-throughput workloads are better served by long-running systems utilizing more cost-effective infrastructure than AWS Lambda (e.g., AWS EC2 virtual machines).

We report the result of using an unmodified version of Apache Drill [26] in a distributed configuration over serverless FaaS and its performance running the TPC-H benchmark. This initial experiment shows that the EPQE approach is feasible and, for all but one query, executing the query with the ephemeral approach is faster than the time it takes to simply instantiate a system with matching configuration over AWS Fargate [27]/Elastic Container Service (ECS) [28] (without even starting to run any queries). We study the start-up time of a query processing engine in this context to examine its practical feasibility. Finally, we also discuss preliminary results that indicate that some queries run faster in one engine (Apache Drill) than in others (Apache Spark [29]) and vice-versa, providing initial evidence that the per-query engine selection approach can bring important advantages.

2. MetaQ Prototype

We first outline the design of the MetaQ prototype, a proof of concept design of the EPQE paradigm.

MetaQ has three main components: *session manager* (SM), *platform provider* (PP), and *meta-system optimizer* (MO). The session manager oversees end-to-end query execution and its resources, including handling communication with the client. The platform provider orchestrates the required resources and configures the environment required for the query engine execution. The meta-system optimizer is used to determine the complete specification of the resources, the query engine to be used, its configuration, and possibly engine-specific query rewriting. In cases when users specify the complete specification, the meta-system optimizer can be bypassed since the execution is fully specified.

Figure 1 illustrates query execution in MetaQ. To execute a query, a user (step ①) starts MetaQ and specifies the query and (optionally) the specifications of the query engine and resources to use for the query execution. MetaQ launches as a serverless FaaS function that can be instantiated on demand via a request to an API proxy service of a cloud provider (such as AWS API Gateway [30]).

MetaQ begins by instantiating the session manager (SM) for the given query. If the user-supplied specifications of the query engine or resources are not complete or are left underspecified, then (step ②) the meta-system optimizer (MO) is used to choose all of the missing specifications. The specification has three elements:

(a) the initial resource allocation (e.g., where, how, and how many configured AWS Lambda functions should be started),
(b) the query engine to use (such as Apache Drill, Apache Spark, Trino [31], etc.),
(c) the configuration of the query engine instantiation, including auxiliary systems such as Zookeeper [32] (e.g., mapping of engine executors onto the resources, configuring engine settings, required storage plugins, etc.)

Once the complete specification is determined, it is used to instruct the platform provider (PP) (step ③) to instantiate and configure the specified resources and then start the configured query engine processes (and any auxiliary systems). The platform provider (step ④), using the specification of initial resource allocation (a), requests the resources from the underlying platform, such as networked FaaS functions, configures their networking, and assigns necessary names, roles, and ids to function instances. The query engine specification (b) determines which function (or container) images are instantiated from the available catalog. Finally, before the platform provider starts the query engine, the specification of the query engine configuration (c) is used to populate the necessary configuration files and environment variables for the query engine.

Once the engine is started and ready to process queries, the session manager (step ⑤) submits the user query and awaits the results from the execution engine. When the query execution completes, the session manager retrieves the results (step ⑥) and returns them to the user (step ⑦). When the query execution completes, all of the resources are released, and the system scales back to zero.

We assume that the persistent data is stored in standard formats (such as Parquet, ORC, Avro, CSV) and is available through cloud storage services compatible with the common query engines (such as S3 or EBS). We restrict the set of distributed query engines considered to ones that can be used in such networked shared-disk configurations.

Our current prototype of MetaQ uses AWS Lambda FaaS functions. To run off-the-shelf query engines despite the restricted function execution environment, we utilized Boxer to provide the required but missing functionality. Boxer is a system that runs standard datacenter applications in FaaS environments, providing the expected network-of-hosts execution model. Boxer runs in every function, alongside the application processes, and

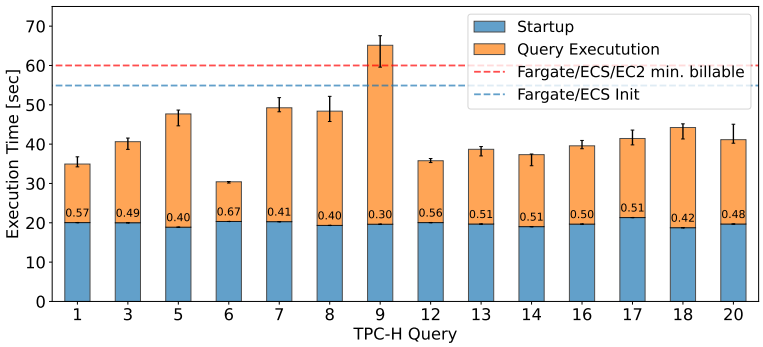

Figure 2: TPC-H (scale factor 10) query execution times of unmodified 8 worker node Apache Drill running on AWS Lambda functions. Bars are medians of 5 executions of each query. The fraction of time consumed by system startup is shown inside the bars. The error bars show the minimums and maximums for the query execution times only. The minimum billable time for EC2/ECS, and median time to only instantiate a comparable AWS Fargate/ECS container are shown for comparison.

establishes an ephemeral network between the participating functions. Boxer executes unmodified application processes (query engines and any auxiliary systems) in a FaaS environment while transparently exposing function-to-function networking via the standard POSIX interfaces (stream sockets etc.). To facilitate configuring the unmodified distributed query engines in FaaS, Boxer is used to assign roles to functions, provide name resolution, host membership, and coordinate query engine process execution. The collection of these Boxer features provides an execution environment in AWS Lambda FaaS that closely matches what is expected by distributed query engines.

Although we show how MetaQ can run in FaaS environments, its design is not tied to them. For example, MetaQ's components (SM,MO,PP) could execute locally on the user's computer, and then could provide (a subset of) standard client protocols that many distributed query engines often expose (such as PostgreSQL standard wire protocol or JDBC). Independently, there could be different platform providers (PP) giving access to different types of resources for query execution, from the user's local resources (useful for smaller workloads) to serverless container services such as AWS Fargate or future serverless platforms that may provide access to heterogeneous hardware accelerators.

3. Feasibility study

3.1. Methodology

To validate the real-world feasibility of the EPQE paradigm, we experiment with some of the basic components of the MetaQ prototype design. We focus our analysis on the technical feasibility of MetaQ rather than analyzing its cost tradeoffs, [19, 20] have shown that serverless can reduce costs for bursty query workloads. For this study, we chose to use a variant of Boxer as our MetaQ platform provider (PP) component, which allowed us to instantiate networked systems using AWS Lambda. For this initial validation, we assumed that along with every considered query, the user specifies the complete system specification (resource allocations, query engine specification, and configuration). This bypasses the meta-engine optimizer(MO), which we plan to explore in the next stages of our research.

We experiment with per-query instantiations of Apache Drill, a general-purpose distributed SQL engine inspired by Google Dremel [33]. We used the TPC-H benchmark to simulate the user queries to be evaluated using MetaQ. Using the benchmark tools, we populated S3 cloud storage with data set at scale factor 10, resulting in 12 GBytes of data and with the largest relation with almost 60 million tuples. Each TPC-H query evaluation request was accompanied by the complete query system specification specifying (a) resources for 10 AWS Lambda functions with 6 vCPUs, x86_64 architecture, and 10GB of memory each, (b) Apache Drill as the query engine (the only engine option in our experiment), and (c) stock configuration options for Apache Drill worker nodes, a head node, and a single Apache Zookeeper node (required by Apache Drill).

The experiment emulates a session manager (SM) that uses the Boxer system as the platform provided (PP) to instantiate resources on AWS Lambda and to start Apache Drill nodes (and Zookeeper). The experimental session manager then waits for the query system to be available, and then submits the query and waits for the results, and returns on completion. In this study, to factor out the effects of function caching, we ensure that only cold functions are used for each query.

3.2. End-to-end query latency

Figure 2 shows the median end-to-end query execution times. Without optimizing the Drill configuration, the observed median end-to-end query execution times were between 30.42s and 65.13s seconds. (Not all of the TPC-H queries were able to run on Drill with the current Boxer variant due to its limit of less than 1024 file descriptors available to Drill, while for some queries, Drill required more.) For comparison, if we chose an alternative platform provider (PP) based on a serverless container service such as AWS Fargate (using AWS Elastic Container Service(ECS), or AWS Elastic Kubernetes Service(EKS) [34]) we expect the execution times to be significantly higher. Such container services are not optimized for startup times, and their implementations rely on EC2 for on-demand resource allocation. We observed that the median time to just instantiate a comparable (serverless) container (8GBytes of memory, with 1024MBytes image size) using AWS Fargate/Elastic Container Service(ECS) is 54.9s (dashed line in Figure 2). This means that by the time the ECS container only begins to start the query engine, all but one of the queries executed by MetaQ are already finished, and the resources are already released. Furthermore, the minimum billable duration for AWS Fargate/ECS is 1 minute, while AWS Lambda billing is at 1ms granularity with no set minimum.

Takeaway 1: MetaQ improves performance and reduces resource usage by instantiating per-query data processing engines on FaaS infrastructure compared to containers or virtual machines.

Figure 2 shows that a significant fraction of the query execution is consumed by the startup time. The median time for the system to become ready to start executing a query is 19.67s, and (in terms of median values) that consumes between 30% (for query 9) and 67% (for query 6) of the total execution times for the queries we tested. There are many techniques that can be used to reduce this time (we have not optimized it in this experiment at all), from configuring the system to avoid starting unnecessary components to snapshotting JVM state [35, 36, 37]. Fortunately, because faster startup times are desirable for other use-cases of FaaS platforms as well, recently AWS Lambda started to offer ability to fully snapshot the initial function state to avoid this issue [38]. We have not yet explored this feature, so the current results with FaaS should be treated as a conservative upper bound since there are further optimizations that we can enable, such as restoring from snapshots.

Takeaway 2: MetaQ does not interfere with potential optimizations that cloud providers could introduce to FaaS. Its performance will only improve with these optimizations, giving it an even bigger advantage over current solutions.

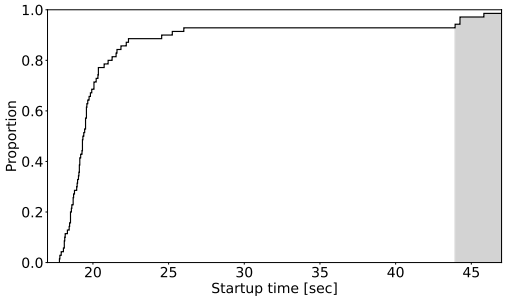

Figure 3: Empirical CDF of the observed system startup times of all instantiations in the experiment. The time from resource instantiation to the time when the 8-worker-node Apache Drill system is ready to start executing the query.

3.3. Engine startup time

We also examined the variance of the query execution times and startup times. The error bars in Figure 2 show the maximum and minimum times for each query execution time relative to the median of the startup time (the variance due to the startup time is factored out). We observe a noticeable, but acceptable, variance in the query execution time, with the majority of the queries having median execution times within 10% of the slowest and fastest executions. The highest observed dispersion was for Query 20, with the slowest observed execution being 18% slower than the median.

However, when we inspect the distribution of all of the startup times during the experiment, shown in Figure 3, we observe significantly higher variance (note that in this experiment, the startup time is independent of the executed query since the client always specifies the same complete specification with each query request, so we do not factor the startup times by query executed.) The startup times ranged widely from 17.78s to 47.02s. In particular, the measurements form two groups; of the total of 70 measurements, the top 5 times (grey area in Figure 3) were above 43s, while all the remaining runs needed less than 27s to start executing the query. Our initial investigation into the source of this variance indicates that its main contributor is the time for the Apache Drill workers to become available after their processes are started. Since the variance does not persist to the query execution times, it suggests that these stragglers are not due to their function execution context being (permanently) resource constrained. It is possible that these functions had to fetch base images from a deeper storage hierarchy as the worker processes were loading blocks of data during startup. This suggests that the meta-system optimizer (MO) strategies should consider the possibility of instantiating additional workers to compensate for

this straggler phenomenon. Once enough workers are available, MetaQ could then terminate the unnecessary stragglers. A similar technique is already performed internally by Boxer platform provider. Boxer, depending on configuration, already instantiates additional functions and proceeds only with the requested number of functions that became available first and immediately terminates the rest of the slower and unnecessary functions. Notice that these techniques that discard stragglers are feasible using FaaS because of the fine granularity of accounting and no minimum billing time.

Takeaway 3: Although limited in scope, the experiment demonstrates the real-world feasibility of the per-query engine paradigm. This very simple experiment leaves many possibilities for future improvements, but it already highlights the potential of our vision and motivates the further exploration of the design space and future work on the MetaQ prototype.

3.4. Selecting Query Engines

A key aspect of the EPQE is the possibility of choosing a different engine for each query. Although further investigation is necessary, our preliminary comparison of query execution times of Apache Drill and Apache Spark, indicates that there likely will be a performance gain from choosing different engines on per-queries granularity. We measured the query execution times of TPC-H scale factor 30 for Apache Drill and Apache Spark using 8 AWS Lambda worker nodes. Ignoring the startup times and only based on the relative query execution times, we observe that for 14 of the queries (1,4,5,6,7,9,10,11,12,13,15,16,17,22) Drill noticeably outperformed Spark, for 3 queries (14,19,20) Spark outperformed Drill, for 2 queries (8,18) perfromance was similar, while 3 queries were completed by only one of the two engines (2,21 Spark only, 3 Drill only.) These initial results suggest that, indeed, the notion of instantiating a different engine depending on the query can be beneficial. This opens up very interesting research questions in terms of how an optimizer could decide which system to use.

4. Use Cases

In this section we explore use cases that could be either implemented on top of the prototype of MetaQ or would require additional work on several aspects of the system and further research.

4.1. More Efficient Data Analytics

There is a growing amount of work exploring how to best use commercial serverless platforms for data analytics. Lambada [19] and Starling [20] both offer a data-analytics platform on top of serverless. Others have explored the benefits and pitfalls of running ML training and inference on FaaS [21, 22]. In all these cases, a major limitation is that serverless functions are stateless and exchange data through remote storage services (e.g., S3). Hence, for each query or task deployed on FaaS, a significant portion of time is spent reading/writing data from/to storage. Complex queries that require shuffling data become even more of a problem by requiring multiple rounds of access to storage servers, thereby further increasing the overhead. A lot of prior work focuses on how to mitigate the data-passing limitations of FaaS infrastructure by constructing custom experimental systems.

A first contribution and potentially the first application of the idea behind MetaQ is that it aims to run existing platforms without having to wait until a suitable new data processing or ML engine is developed matching the characteristics of serverless. Our approach enables running complex data processing tasks at a large scale using existing mature systems, using a variety of engines tailored to the query and data at hand, and deploying at the scale needed while still maintaining all the advantages of serverless.

4.2. Dynamically Extensible Engines

Data processing engines, such as traditional relational databases or many SQL-centric distributed platforms, are limited along two dimensions. One is in terms of deployment, as only one configuration is available at any time. This leads to overprovisioning to make sure the system can cope with any possible workload. The other is in terms of functionality. Very often, data is processed in these engines and then needs to be moved to other systems for further processing (e.g., ML training, statistical analysis, visualization).

MetaQ can be used as an extension of existing engines to address these two problems. In the same way we show that one can launch a complete data processing system on serverless when a query arrives, an existing engine running on a VM could do the same to trigger additional capacity when necessary. For example, the basic mechanism presented here can be used to have Apache Drill launch additional ephemeral engines when the long-running system is not able to cope with the additional load. Recently, a similar approach has been explored by modifying an existing system, Pixels-Turbo [39] is an extension of a Pixels [40] query engine that can instantiate query engines in AWS Lambda function to add elasticity to the system instantiated on long-running VMs. In the case of missing functionality for some tasks, the transition to another system can be done by triggering the corresponding system once the data processing engine finishes. This eliminates the need to have both systems

running all the time and helps to automate the process rather than copying the data and transferring it manually to the other system (and then copying results back).

Complementary to these ideas is the notion of deploying a minimalist system (i.e., requiring much fewer resources) on a permanent infrastructure using VMs and then using the mechanisms of MetaQ to launch a more complete version of the system (or one tailored exactly to the task at hand) when queries arrive that require the more advanced functionality.

4.3. User-owned Data Analytics Stack

Cloud providers offer a set of Query-as-a-Service platforms, such as AWS Athena [41], which provide a simplified interface for large-scale analytics and charge users per byte read. However, users may still prefer to run their queries on a data analytics stack that they fully control (e.g., to optimize parameters and hardware configurations for their workloads). MetaQ enables users to run their own data analytics stack while still benefiting from simple abstractions and a convenient pay-per-query cost model, as resources can be acquired and released on-demand in response to load. As Palkar and Zaharia point out, users may also prefer to run their own analytics engines and web services rather than relying on out-of-the-box cloud solutions for privacy reasons [42]. This is especially true when queries involve UDFs, as these are more difficult to securely isolate in shared infrastructure deployments. By operating their own data analytics stack, users get to control how the system is configured and monitor how they are billed for the work performed for a particular task.

4.4. Data Lakes

Data Lakes refer to collections of heterogeneous data that needs to be processed in a variety of different ways. The problem with this notion is that the processing is also highly heterogeneous, and it is the user who is responsible for handling it. Lakehouses is a new iteration of the concept that incorporates the data processing as a first-class citizen and provides support for different engines, languages, etc., while automating as much as possible the task of matching data to engines and tools [43].

MetaQ is well-suited to Lakehouses as it enables dynamically selecting the engine and processing tools on the fly, and this can be done on the basis such as data types, data sizes, type of query, user requirements, or cost, etc. Furthermore, the per-query engine vision enables an intriguing possibility: sharing of auxiliary data structures across engines (indexes, partitions, zone maps, etc.) as well as creating a general infrastructure that is engine agnostic (e.g., a main memory caching layer for data to avoid having to retrieve it from slow storage every time or a results cache). Such infrastructure exists, but it is typically system specific. MetaQ opens up the possibility of seeing these aspects as orthogonal to the actual engine. In the extreme, all common modules of query engines could become serverless components dynamically added to an engine as it is instantiated with the query-specific functionality.

5. Research Opportunities

The idea of EPQE behind MetaQ opens a number of interesting research directions which we now highlight.

5.1. The Meta-Engine

EPQE unlock a number of opportunities when it comes to selecting the most appropriate engine for each query. This can be done in a very simple manner by, for instance, asking the user to specify which engine to use. However, we are interested in automating the selection process by building an end-to-end query system that handles this. In a scenario where users write queries in an engine-agnostic syntax (for example, in a declarative language such as SQL), MetaQ's meta-system optimizer could inspect the query and determine which engine is the most efficient given the data types, its type (static or streaming), the type of operations required, etc. This leads to cross-engine optimizations, such as picking the engine that is faster to perform a given operation provided by several engines. The main research question is how to derive meta-system optimizer policies. One possible approach is to extend the domain of automatic configuration systems [5] with the additional tasks of choosing not just configuration parameters for a query engine, but also the choice of the query engine itself and resource allocation based on the query considered, eventually realizing the vision of vertically integrated per-query optimization.

5.2. Autoscaling Per-query Deployments

With a new deployment being launched and shut down per query, it is now possible to optimize the deployment where the engine will run for every query. Such deployment configuration could determine the amount of resources used, such as the CPU and/or memory budget. Such configuration could be inferred by analyzing the query and data inputs to estimate the amount of data that would be processed and, therefore, the amount of compute and memory necessary to finish the query within a particular time frame. From another perspective, it is now possible to dynamically find tradeoffs between execution time and price for each query. This tradeoff could

also be exposed to users as a way to prioritize interactive queries over batch workloads.

5.3. Query Scheduling and Caching

Beyond automatically sizing and optimizing per-query deployments, it is also possible to schedule query execution on nodes that have some locally cached data or that are close to storage nodes. For example, if a workload requires two queries to be executed, the second query could be scheduled for execution on the same physical node(s) that was used to execute the previous one. To keep data local, caching approaches such as Faa$t cache [44] can also be used to keep the output of queries.

5.4. System Infrastructure

To implement inter-function communication, MetaQ prototype uses Boxer as its platform provider. Boxer (and therefore MetaQ) do not require any cloud provider intervention and can be deployed today in AWS Lambda. However, Boxer is not yet feature complete in terms of interfaces, networking support, reliability, and integration within larger systems. That is something that we are working on at the moment so as to have a more solid basis for the system. Similarly, Boxer was initially built for AWS Lambda. We are in the process of studying how to port Boxer to other commercial serverless offerings. Doing so would open yet another wave of exciting opportunities, like triggering serverless jobs across heterogeneous clouds using the networking capabilities available in Boxer.

5.5. Generalizing to Other Engines

Our experiments are only a first step towards the per-query engine vision. We plan to test this paradigm and our MetaQ prototype on a wider range of data processing engines and platforms on top of the existing prototype to make sure it can indeed be used as a general-purpose distributed computing platform equivalent to what can be done on a VM. Systems that we are in the process of testing include Apache Spark, Trino, Databend [45], Flink [46], Clickhouse [47]. Having them running on the same serverless platform will also offer a great opportunity to study the engine designs that are most suitable for serverless, providing very valuable information on the road toward serverless native engines.

6. Conclusion

Distributed data processing engines often require to have a fixed underlying infrastructure to run in the form of pre-allocated VMs, Virtual Private Networks, and other services provided by the cloud. This results in inefficiencies that are difficult to address: over-provisioning, coarse resource allocation, generic engine configurations, low utilization, etc. In this paper, we put forward the idea of ephemeral per-query engines: selected query engines dynamically instantiated when a query arrives and removed when it terminates. In the paper, we have outlined the idea, discussed its potential to address many of the limitations of current deployments, provided a feasibility study, and demonstrated that, while there is still much work to do, it is possible to implement it in current FaaS platforms. The initial experiments are highly encouraging. They show that existing engines can be sufficiently quickly instantiated on demand to run a single query. Building on this basis, we have also discussed and presented several research directions that can be pursued based on the ideas and results presented here.

References

[1] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, T. Cruanes, Building an elastic query engine on disaggregated storage, in: Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation, NSDI'20, USENIX Association, USA, 2020, p. 449–462.

[2] Z. Tan, S. Babu, Tempo: Robust and self-tuning resource management in multi-tenant parallel databases, Proc. VLDB Endow. 9 (2016) 720–731.

[3] S. Das, F. Li, V. R. Narasayya, A. C. König, Automated demand-driven resource scaling in relational database-as-a-service, in: Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, 2016, p. 1923–1934.

[4] A. Augusta, S. Idreos, Jafar: Near-data processing for databases, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15, 2015, p. 2069–2070.

[5] D. V. Aken, D. Yang, S. Brillard, A. Fiorino, B. Zhang, C. Billian, A. Pavlo, An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems, Proc. VLDB Endow. 14 (2021) 1241–1253.

[6] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, Y. Yang, Bestconfig: Tapping the performance potential of systems via automatic configuration tuning, Association for Computing Machinery, New York, NY, USA, 2017.

[7] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, J. Crowcroft, Unikernels: Library operating systems for the cloud, Association for Computing Machinery, New York, NY, USA, 2013.

[8] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam,

A. Jung, G. Gain, C. Soldani, C. Lupu, c. Teodorescu, C. Răducanu, C. Banu, L. Mathy, R. Deaconescu, C. Raiciu, F. Huici, Unikraft: Fast, specialized unikernels the easy way, Association for Computing Machinery, New York, NY, USA, 2021.

[9] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, D. A. Patterson, What serverless computing is and should become: The next phase of cloud computing, Commun. ACM 64 (2021) 76–84.

[10] P. Castro, V. Ishakian, V. Muthusamy, A. Slominski, The rise of serverless computing, Commun. ACM 62 (2019) 44–54.

[11] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, D.-M. Popa, Firecracker: Lightweight virtualization for serverless applications, in: NSDI, 2020.

[12] L. Ao, G. Porter, G. M. Voelker, Faasnap: Faas made fast using snapshot-based vms, Association for Computing Machinery, New York, NY, USA, 2022.

[13] AWS Lambda, 2020. URL: https://aws.amazon.com/lambda, (accessed: 2020-08-17).

[14] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, C. Wu, Serverless computing: One step forward, two steps back, in: CIDR, 2019.

[15] L. Wang, M. Li, Y. Zhang, T. Ristenpart, M. Swift, Peeking behind the curtains of serverless platforms, in: Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '18, 2018.

[16] AWS Step Functions, 2023. URL: https://aws.amazon.com/step-functions/, (accessed: 2023-03-01).

[17] Azure Durable Functions, 2023. URL: https://learn.microsoft.com/en-us/azure/azure-functions/durable/, (accessed: 2023-03-01).

[18] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, A. Tumanov, Cloudburst: Stateful functions-as-a-service, Proc. VLDB Endow. 13 (2020) 2438–2452.

[19] I. Müller, R. Marroquín, G. Alonso, Lambada: Interactive data analytics on cold data using serverless cloud infrastructure, in: SIGMOD, 2020.

[20] M. Perron, R. Castro Fernandez, D. DeWitt, S. Madden, Starling: A scalable query engine on cloud functions, in: SIGMOD, 2020.

[21] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, C. Zhang, Towards demystifying serverless machine learning training, in: Proceedings of the 2021 International Conference on Management of Data, 2021, p. 857–871.

[22] Y. Wu, T. T. A. Dinh, G. Hu, M. Zhang, Y. M. Chee, B. C. Ooi, Serverless data science - are we there yet? A case study of model serving, in: SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022, 2022.

[23] A. Wang, J. Zhang, X. Ma, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan, Y. Cheng, Infinicache: Exploiting ephemeral serverless functions to build a cost-effective memory cache, in: USENIX FAST, 2020.

[24] J. Zhang, A. Wang, X. Ma, B. Carver, N. J. Newman, A. Anwar, L. Rupprecht, V. Tarasov, D. Skourtis, F. Yan, Y. Cheng, Infinistore: Elastic serverless cloud storage 16 (2023).

[25] M. Wawrzoniak, I. Müller, R. Bruno, G. Alonso, Boxer: Data analytics on network-enabled serverless platforms, in: CIDR, 2021.

[26] Apache Drill, 2022. URL: https://drill.apache.org/, (accessed: 2022-10-20).

[27] AWS Fargate – Serverless compute for containers, 2023-03-01. URL: https://aws.amazon.com/fargate/, (accessed: 2023-03-01).

[28] Amazon Elastic Container Service (Amazon ECS), 2023. URL: https://aws.amazon.com/ecs/, (accessed: 2023-03-01).

[29] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, I. Stoica, Apache spark: A unified engine for big data processing 59 (2016).

[30] Amazon API Gateway, 2023. URL: https://aws.amazon.com/api-gateway/, (accessed: 2023-03-01).

[31] Trino, 2023. URL: https://trino.io/, (accessed: 2023-06-20).

[32] P. Hunt, M. Konar, F. P. Junqueira, B. Reed, Zookeeper: Wait-free coordination for internet-scale systems, USENIX ATC'10, 2010.

[33] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, T. Vassilakis, Dremel: Interactive analysis of web-scale datasets, 2010.

[34] Amazon Elastic Kubernetes Service (EKS), 2023. URL: https://aws.amazon.com/eks/, (accessed: 2023-03-01).

[35] W. Shin, W.-H. Kim, C. Min, Fireworks: A fast, efficient, and safe serverless framework using vm-level post-jit snapshot, Association for Computing Machinery, New York, NY, USA, 2022.

[36] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, H. Chen, Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting, in: ASPLOS, 2020.

[37] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, J. Appavoo, Seuss: Skip redundant paths to make serverless fast, in: EuroSys, 2020.

[38] Improving startup performance with Lambda SnapStart, 2023. URL: https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html, (accessed: 2023-03-01).
[39] H. Bian, T. Sha, A. Ailamaki, Using cloud functions as accelerator for elastic data analytics 1 (2023).
[40] H. Bian, A. Ailamaki, Pixels: An efficient column store for cloud data lakes, in: 2022 IEEE 38th International Conference on Data Engineering (ICDE), 2022, pp. 3078–3090.
[41] Amazon Athena, 2020. URL: http://docs.aws.amazon.com/athena/, (accessed: 2020-08-17).
[42] S. Palkar, M. Zaharia, Diy hosting for online privacy, in: Proceedings of the 16th ACM Workshop on Hot Topics in Networks, HotNets-XVI, 2017.
[43] M. Zaharia, A. Ghodsi, R. Xin, M. Armbrust, Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics, in: 11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings, www.cidrdb.org, 2021.
[44] F. Romero, G. I. Chaudhry, I. n. Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, R. Bianchini, Faa$t: A transparent auto-scaling cache for serverless applications, Association for Computing Machinery, New York, NY, USA, 2021.
[45] Databend, 2023. URL: https://databend.rs/, (accessed: 2023-06-20).
[46] Apache Flink, 2023. URL: https://flink.apache.org/, (accessed: 2023-03-01).
[47] ClickHouse, 2023. URL: https://clickhouse.com/, (accessed: 2023-06-20).