Using Cloud Functions as Accelerator for Elastic Data Analytics

HAOQIONG BIAN, EPFL, Switzerland
TIANNAN SHA, EPFL, Switzerland
ANASTASIA AILAMAKI, EPFL, Switzerland

Cloud function (CF) services, such as AWS Lambda, have been applied as the new computing infrastructure in implementing analytical query engines. For bursty and sparse workloads, CF-based query engine is more elastic than the traditional query engines running in servers, i.e., virtual machines (VMs), and might provide a higher performance/price ratio. However, it is still controversial whether CF services are good suites for general analytical workloads, in respect of the limitations of CFs in storage, network, and lifetime, as well as the much higher resource unit prices than VMs.

In this paper, we first present micro-benchmark evaluations of the features of CF and VM. We reveal that for query processing, though CF is more elastic than VM, it is less scalable and is more expensive for continuous workloads. Then, to get the best of both worlds, we propose Pixels-Turbo - a hybrid query engine that processes queries in a scalable VM cluster by default and invokes CFs to accelerate the processing of unpredictable workload spikes. In the query engine, we propose several optimizations to improve the performance and scalability of the CF-based operators and a cost-based optimizer to select the appropriate algorithm and parallelism for the physical query plan. Evaluations on TPC-H and real-world workload show that our query engine has a 1-2 orders of magnitude higher performance/price ratio than state-of-the-art serverless query engines for sustained workloads while not compromising the elasticity for workload spikes.

CCS Concepts: • **Information systems** → **Data warehouses**; **Online analytical processing engines**; **DBMS engine architectures**; **Relational parallel and distributed DBMSs**; *MapReduce-based systems*; **Data analytics**; **Database query processing**; **Cloud based storage**; **Column based storage**; *Record and block layout*; *Computing platforms*; • **Computer systems organization** → **Cloud computing**.

Additional Key Words and Phrases: OLAP, QaaS, serverless, query processing, query optimization, cloud databases, data lake, data warehouse, FaaS, cloud function, column store, cloud storage, elasticity, cost efficiency

ACM Reference Format:
Haoqiong Bian, Tiannan Sha, and Anastasia Ailamaki. 2023. Using Cloud Functions as Accelerator for Elastic Data Analytics. *Proc. ACM Manag. Data* 1, 2, Article 161 (June 2023), 27 pages. https://doi.org/10.1145/3589306

1 INTRODUCTION

Cloud function (CF) service, or function-as-a-service (FaaS), is a category of cloud computing services that allows the developers to run their program as auto-scaling and short-lived function instances in the cloud. It has a very short startup time (e.g., 1-2 seconds to start an instance) and very high elasticity (e.g., spawns hundreds of parallel instances in a second). Therefore, it is now widely used in microservices and data processing. Recent systems, such as Starling [66] and Lambada [65], proposed solutions to implement an analytical query engine purely based on CFs. Benefiting from

Authors' addresses: Haoqiong Bian, EPFL, Lausanne, Switzerland, haoqiong.bian@epfl.ch; Tiannan Sha, EPFL, Lausanne, Switzerland, tiannan.sha@epfl.ch; Anastasia Ailamaki, EPFL, Lausanne, Switzerland, anastasia.ailamaki@epfl.ch.

2836-6573/2023/6-ART161 $15.00
https://doi.org/10.1145/3589306

Table 1. Resource unit prices in AWS (as of September 2022).

Resource type	Lambda	EC2 on-demand	EC2 spot
CPU (¢/core-h)	10	4.8	1.1
RAM (¢/GB-h)	6	1.2	0.27
Network (¢/Gbps-h)	85.71	15.36	3.53

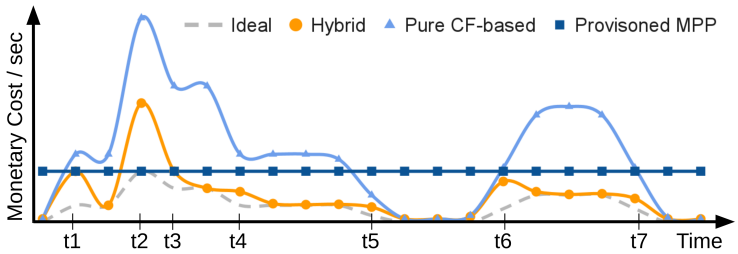


Fig. 1. Illustration of the monetary cost of different query engines under the same workload and performance.

the elasticity and simplicity of CF, a pure CF-based query engine can scale to zero (except for a small coordinator) and greatly reduce the maintenance cost of the system.

However, elasticity and simplicity come at the price of ceding the control of the hardware and runtime to the cloud providers. In order to efficiently automate the resource scheduling, cloud providers have added some restrictions to CF, including limited size (e.g., up to 10GB RAM and 6 vCPUs per instance), no persistent local disk, and no inter-instance communications [69]. Therefore, cloud storage such as S3 is used to store the tables and intermediate results for query processing. For example, in Starling [66] and Lambada [65], to execute a hash-partitioned join, the tables have to be partially partitioned and written into S3 by the first stage of CF invocations, and then the intermediate partitioned files in S3 are read by the next stage to complete the join. This introduces significant I/O overhead and hinders the parallelism of CPU and network.

Moreover, the unit prices of the resources in CFs are higher since the cloud provider has to pay more scheduling and provisioning costs on their side. As shown in Table 1, in AWS, the resource unit prices of CF (i.e., Lambda [15]) are 9.09-24.28x higher than those of VM (i.e., EC2 spot instances [5] [1]). In this example, we use the instance type *m5.8xlarge* for EC2 on-demand [4] and EC2 spot, and 10GB RAM size configuration for Lambda (which has ≈0.7Gbps network bandwidth [64]). CF and VM services prices in other major public clouds, such as Azure and Google Cloud, are very close to those in AWS.

For analytical processing, the key advantage of CF is better elasticity. In our evaluations, it takes only 1-2 seconds to startup hundreds of AWS Lambda instances, whereas launching a new EC2 VM takes about one minute. Therefore CF is appropriate for processing bursty and sparse workloads [65, 66], whereas VM is more appropriate for stable and continuous workloads. However, continuous workloads with peak hours and spikes are general in today's analytical applications [46, 50, 70, 71]. Neither CF nor VM is the clear winner for such type of workload.

In this paper, to achieve a better performance/price ratio in all conditions, we propose Pixels-Turbo - a hybrid query engine that uses CFs as the auxiliary computing resource to process unpredictable

[1]Spot VM (e.g., EC2 spot) is a kind of VM available at a 70-90% discount compared to normal on-demand VMs. It could be reclaimed by the cloud provider given a grace period, which is tolerable to modern query engines such as Spark and Trino [38, 74]. The reclaim rate of spot VMs is very low. In major cloud platforms, the expectation of a spot VM being reclaimed within a month is usually below 10% [12, 16].

workload spikes. Pixels-Turbo executes the queries in an MPP (massively parallel processing) cluster by default. The cluster is automatically scaled according to workload changes. However, scaling out the cluster usually takes 1-2 minutes. If the workload bursts sharply, the cluster can not scale out in time to process the spike. In this case, Pixels-Turbo invokes CFs to accelerate query processing before the cluster scaling is done. We implement Pixels-Turbo as the query engine in Pixels [45] and open-sourced it on GitHub (https://github.com/pixelsdb/pixels/tree/master/pixels-turbo).

The benefit of our query processing solution in Pixels-Turbo is illustrated in Figure 1. Given a query workload that changes over time: an ideal (but impractical) query engine would use the cheapest VMs and scale perfectly for the workload changes; a provisioned MPP query engine has to provision the VMs for the maximum requirement; a pure CF-based query engine executes queries in CFs, which can scale perfectly for workload changes but may be much more expensive than VMs. Our hybrid solution combines the advantages of both CF and VM. When the workload suddenly bursts, CFs get involved in processing some of the queries (at $t1$, $t2$, and $t6$). After a couple of minutes, the VM cluster scales to the capable size and reduces the cost. In our solution, we release the VMs lazily, thus the cost reduction lags a bit behind the workload reduction (at $t3$, $t4$, $t5$, and $t7$). We also use spot VMs in the cluster to improve cost efficiency further.

This solution has to automatically scale the MPP cluster and tolerate spot VM reclaiming without interrupting queries. To address this problem, we propose a framework that monitors the workload and triggers actions to add or remove VMs gracefully. When a new VM is created, a script in the OS image automatically initializes the software stack and adds the VM into the cluster. When a VM is to be terminated or reclaimed, a daemon process in the VM calls the graceful-shutdown API [38] to wait for the running query tasks and then remove the VM from the cluster.

In addition to auto-scaling, we have to divert the workload spikes to CFs transparently. In Pixels-Turbo, the queries to be executed in the VMs or the CFs have separate query queues. A query enters the CF query queue only when the VM query queue is full. We implement a secondary query executor that can fully execute operations such as select, projection, join, and aggregation in the CFs. Thus, for the query in the CF query queue, expensive operations in the query plan will be pushed down into a sub-plan that is executed by the CF-based executor.

Meanwhile, improving the CF-based query performance is crucial as CF has more restrictions and overheads. Otherwise, users may suffer from poor performance when the CF executor is in use. To address this problem, we propose a set of optimizations for the CF-based query executor: (1) direct-write-back that directly returns the sub-plan result to the cluster; (2) chain-join that merges sequential broadcast joins into fewer stages to reduce the amount of intermediate I/Os in joins; (3) early-projection that purges redundant columns in data shuffling; (4) late-shuffling that postpones shuffling to reduce the amount of data to shuffle in aggregations. We also propose a cost-based optimizer to select the best algorithm and parallelism for each query operator in the sub-plan.

In summary, the main contributions of this paper are:

- We analyze the pros and cons of CF and VM by micro-benchmarks. Based on the insights, we propose the query engine Pixels-Turbo that uses CF as the accelerator for processing unpredictable workload spikes, thus providing a high performance/price ratio without compromising elasticity (Section 3).
- To address the intermediate I/O bottleneck when CFs get involved in query processing, we propose optimizations to reduce the amount of intermediate I/Os in joins, aggregations, and result returning (Section 4).
- We propose a cost-based optimizer to select the appropriate algorithm and parallelism for the CF-based query plan. To the best of our knowledge, this is the first work that provides a cost-based optimizer for SQL-style jobs in CFs (Section 5).

- Evaluations on TPC-H and real-world workload show that Pixels-Turbo can provide 1-2 orders of magnitude higher performance/price ratio than state-of-the-art serverless query engines from industry (Athena [47] and Redshift-serverless [7, 43]) and academia (Starling [66]) for sustained workloads, while not compromising the elasticity for unpredictable workload spikes (Section 6).

2 PRELIMINARIES

In this section, we briefly introduce how data is managed and queried in today's cloud data lakes and serverless query services and how cloud functions are used in serverless query processing.

2.1 Cloud Data Lakes

In today's cloud data lakes, storage and query execution are decoupled [45]. The query engines, such as Trino/Presto [36, 70] and Spark SQL [42], are responsible for executing queries. Whereas the tables are stored as a set of raw files with a specific format in cloud storage services such as AWS S3 [8]. Each raw file contains a horizontal partition, a.k.a., *row group*, of the table.

In a real-world deployment, the catalog of the raw files and the metadata of the tables (e.g., schema and statistics) are often maintained by a table management system, such as Delta [63] and AWS Glue [14]. Some table management systems also support caching, DML, and schema evolution [9, 10, 63]. They actually act as a kind of lightweight storage engines based on open file formats.

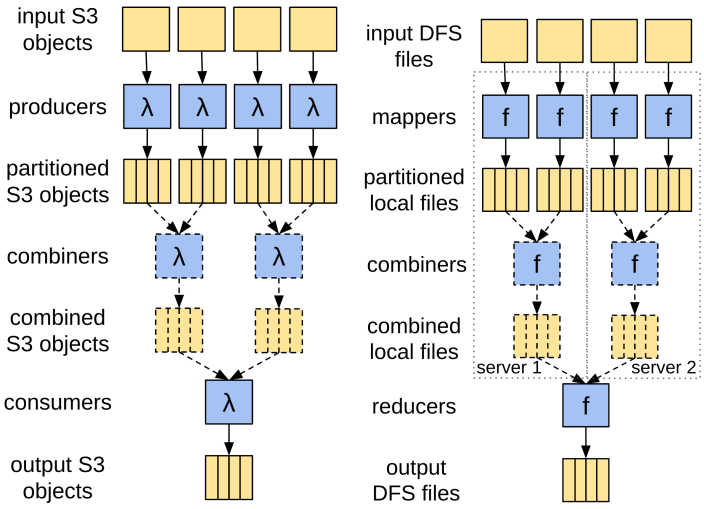
The query engines do not persist any data or metadata, thus they are highly scalable. They provide similar APIs named *connector* or *data source* for external storage [36, 42, 45]. Third-party table management systems or storage engines can be integrated into the query engine by implementing their own connectors. The connector provides metadata to the query engine for query parsing and optimization. During query execution, each query task in the query engine calls the connector to scan a split (i.e., a set of row groups) from the table. The other operations (e.g., join, aggregation, and ordering) in the query plan are executed on the scan result.

To reduce the amount of data fetched from the connector, most query engines support operation pushdown [26, 32, 39]. Projection and selection can be easily pushed down into the connector with the other table scan parameters such as schema and table names. However, pushing down other operations such as join and aggregation is more complicated. In Trino, this is done by creating a logical temporary table for the operation [39]. Then the connector, instead of the query engine, is responsible for executing the operation and returning the result to the query engine. In Spark, aggregation and limit operation pushdown are supported in a similar way [31]. Besides operation pushdown, a very promising technology called compressed data direct computing [76] has been proposed to reduce both time and space costs, and has been proven to be successfully applied in distributed and heterogeneous environments [75].

2.2 Serverless Query Services

Traditionally, cloud users have to purchase and maintain a provisioned VM cluster to run queries. The maintenance cost may take a large portion of the total cost of ownership (TCO) of a data lake. In recent years, many cloud providers developed serverless query services to address this problem. Examples are AWS Athena [47], Google BigQuery [22], and Azure Analysis Services [17]. Using these services, users only need to pay for the executed queries.

Under the hood, the queries are executed in a shared cluster, where the serverful query engine still exists. For example, BigQuery executes queries in a Dremel cluster [19, 62], whereas Athena executes queries in a customized Presto cluster [47]. The cloud provider maintains the cluster on



(a) Starling shuffle. (b) MapReduce shuffle.

Fig. 2. Shuffle operator in Starling and MapReduce.

behalf of the users and only exposes the query interface. Some traditional query engines, such as Redshift, are also evolving their architectures following the serverless paradigm [43]. Redshift-serverless [7] is the new serverless architecture of Redshift. It automatically scales the query to a cluster of small servers named Redshift Processing Units (RPUs) and charges the customer by the number of RPU-seconds used on the queries. One RPU provides 16 GB of memory [27].

2.3 Cloud Functions

Cloud function (CF) is a category of serverless computing service in the cloud. Developers implement a function in a programming language of choice and upload it to the cloud. The function can be invoked by programs or events and used as an alternative to virtual machines (VMs) in data processing applications. The main difference between CF and VM is: (1) CF has a limited size, e.g., up to 10GB RAM and 6 vCPUs; (2) CF is short-lived, e.g., up to 15 minutes; (3) CF does not have persistent local disks or network connections between instances; (4) CF has a very short startup time, e.g., 1-2 seconds, and very high elasticity, e.g., spawning hundreds of instances per second. (5) CF is billed on a per invocation basis according to the size (RAM or CPU) and execution duration, e.g., $\$1.667 \times 10^{-5}$ per GB-RAM per second.

2.4 Query Processing in Cloud Functions

As discussed in Section 2.3, the computing infrastructures of commercial serverless query services lack generality and are not open to the research community. Therefore, cloud function (CF) attracts more attention in the research on serverless query processing [55, 61, 65–67].

Query processing in CFs is similar to that in MapReduce [54, 55]. For example, Figure 2 shows the shuffle operator in Starling [66] (a state-of-the-art serverless query engine based on AWS Lambda) [2] and MapReduce [51]. In either system, shuffle consists of a producer (mapper) stage to produce a partitioned file for each input split, an optional combiner stage to partially merge the partitioned files, and a consumer (reducer) stage to generate the final output. The processing unit (blue squares

[2]The shuffle implementation in Starling is a typical example of CF-based shuffle. The multi-level exchange (shuffle) operator in another recent CF-based query engine Lambada [65] is equivalent to Starling's shuffle with combiners.

in Figure 2) in either shuffle operator is a function instance executed in a container or process. The main difference is in data locality. In MapReduce, the mapper can be scheduled to where the data is located; and mappers and combiners can be executed in the same server without remote I/O. Whereas in CFs, all stages must be synchronized through the cloud storage (e.g., S3) as there is no direct connection between CF instances.

In the past decade, MapReduce has been proved inefficient for query processing [42, 49, 70, 72]. Therefore, memory-optimized MapReduce variants (e.g., Spark [74] and Tez [68]) and lightweight MPP engines (e.g., Presto/Trino [70], Hawq [48], and Impala [58]) are designed to replace the ordinary MapReduce in data lakes. Intuitively, as an analogue of MapReduce with even more limitations, CF is neither an efficient option for query processing. In Section 3.1, we use experimental results to support this intuition.

Some existing works tried to improve the query performance in CFs. Sonic [61] reduces the cloud storage I/O overhead by shuffling data through VMs, whereas Boxer [64] uses NAT hole-punching to enable network connection between CF instances. However, these solutions either sacrifice the elasticity of CFs or are not officially permitted by the cloud providers and therefore have stability issues. To improve the I/O efficiency, Starling [66] employs parallel reads to improve the read throughput and a retry policy to mitigate the I/O stragglers. However, these methods are also proved effective in VMs [45]. Therefore, it does not help bridge the performance gap between CF-based and VM-based query processing.

For query processing, the key advantage of CFs is their higher elasticity. We can start hundreds or even thousands of CF instances in a couple of seconds. These instances will be automatically released after invocation. For bursty and sparse workloads, CFs may be more cost-efficient than VMs [65, 66].

3 MOTIVATION AND DESIGN

The scalability and cost-efficiency of a cloud-native query engine highly depend on the underlying computing services. In this section, we first present micro-benchmark evaluations of the scalability and cost-efficiency of cloud computing services. Then we introduce the overview and design choices in Pixels-Turbo that are motivated by the insights from the experimental study.

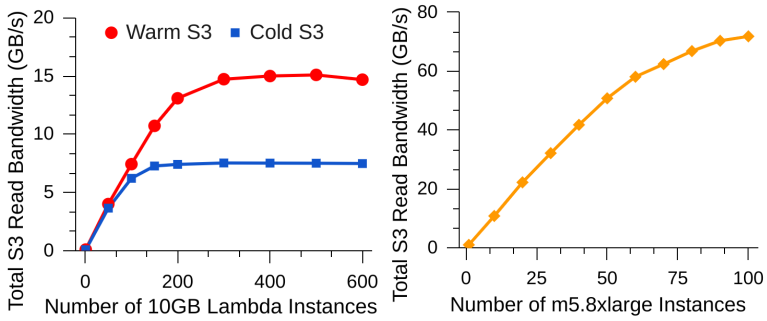
We consider cloud function (CF) and virtual machine (VM) as the two major types of cloud computing services in this paper and use AWS Lambda and AWS EC2 as the representatives of CF services and VM services, respectively. There are also some container services in the cloud, such as AWS ECS and Azure Containers. However, for query processing, there is no significant difference between containers and VMs.

3.1 Scalability of Computing Services

Scalability is important for query processing. As the data volume increases, a scalable query engine should be able to keep the query performance by adding additional computing resources near-linearly. The scalability of a query engine is highly related to the underlying computing resources. Here, we analyze the scalability of CFs and VMs from the two crucial aspects: storage and computation.

I/O Scalability. Queries have to read the tables from cloud storage. Whereas CFs also shuffle data through cloud storage (see Section 2.4). Since table scan and shuffle are usually the most expensive operations, the scalability of cloud storage I/O in computing services will greatly affect the scalability of query execution.

Figure 3 shows the total S3 read bandwidth in Lambda and EC2. In this micro-benchmark, we store 600 files (objects) of size 512MB in S3 and read them for a fixed period of time in Lambda



(a) AWS Lambda. (b) AWS EC2.

Fig. 3. Total S3 read bandwidth in AWS Lambda and EC2.

or EC2. For Lambda, we use the configuration of 10GB RAM and 6 vCPUs [3]. For EC2, we use the instance type *m5.8xlarge* (32 vCPUs, 128GB RAM, 10Gbps network). Each Lambda or EC2 instance keeps reading 1MB chunks from a subset of the files with as many threads as the vCPUs. In this way, we get reasonably close to the maximum bandwidth in the instance (70-80MB/s for Lambda; 1-1.2GB/s for EC2). The write bandwidth in Lambda is very close to the read bandwidth, whereas queries in EC2 do not write intermediate results to S3. Therefore we only show the read bandwidth.

In Figure 3, we can see the total read bandwidth of Lambda and EC2 scale with the number of instances. However, the read bandwidth is capped to ≈7.5GB/s when Lambda scales to more than 150 instances (*Cold S3*). Whereas for EC2, the read bandwidth can scale to at least 72GB/s. Due to the quota limit of our account, we can not create more than 100 m5.8xlarge instances. If we repeat the evaluation multiple times (>10), it seems that S3 will dynamically increase the cap for Lambda, say to ≈15GB/s (*Warm S3*). Sometimes the bandwidth can burst to over 20GB/s for a few minutes, but it is hard to reproduce. We believe this is not due to caching, as it also increases the cap for the other files that were never read. The bandwidth will drop if there are no massive reads for a period of time (≈10 minutes). Such short-time warm bandwidth may be ineffective for the bursty or sparse workloads studied in [65, 66].

Moreover, we find the bandwidth of the instances is not balanced after reaching the maximum total bandwidth, which leads to some stragglers that further reduce the I/O scalability of CFs. We do not know the exact reason for this scalability gap. But it is reasonable as the provisioning algorithm behind CF services has to serve general applications that are not IO-intensive, such as microservices. Aggressive storage bandwidth provisioning will lead to a waste of resources. Whereas adaptively adjusting the storage bandwidth for different workloads is more cost-efficient but challenging.

Computational Scalability. We use the same instance configurations as above to evaluate the computational scalability of Lambda and EC2. Instead of file reading, we test the memory allocation and column encoding performance, which are memory and CPU-intensive, respectively. The result shows that both Lambda and EC2 scale linearly on such workloads.

We did not evaluate other cloud platforms as they have more restrictions on I/O throughput and CF parallelism [18, 23, 24, 28, 30].

Takeaway: In terms of cloud storage I/O, VMs are at least 5-10x more scalable than CFs. This significantly affects the scalability of I/O intensive workloads. The storage-based shuffle in CFs further widens this gap for analytical queries. For CPU and memory-intensive workloads, both CFs and VMs scale linearly.

[3] AWS Lambda does not allow users to select the number of vCPUs. However, the number of vCPUs in a Lambda instance is proportional to the RAM size. More precisely, every 1769MB RAM gets at least one vCPU [20].

3.2 Cost-efficiency of Computing Services

The cost-efficiency of a query engine in the cloud is highly related to the unit prices and the elasticity of the computing resources.

Unit Price. As shown in Table 1, the unit prices of the resources in Lambda are 2.1-5.6x higher than those in EC2 on-demand VMs, and 9.1-24.3x higher than those in EC2 spot VMs. The prices of VMs and CFs in other major cloud platforms, such as Azure and Google Cloud, are very similar to those in AWS. Furthermore, the vCPUs in CFs are also slower. In the micro-benchmark evaluation, we find that the per vCPU column encoding throughput in Lambda is only half of that in EC2. Therefore, considering the bills incurred during query execution, CF is much more expensive than VM, even if Cf is not bounded by the I/O bandwidth.

Elasticity. However, CF is billed on a per-invocation basis and has a very short startup time. In our micro-benchmark evaluations, the cold startup time of Lambda is 1-2 seconds. AWS does not immediately terminate the Lambda instance after the first invocation. The subsequent invocations within 5 minutes (to our experience) can benefit from a warm startup which is much faster. Unlike Lambda, launching an EC2 on-demand or spot instance usually takes 1-2 minutes, which is also billed. Therefore, if VM is idle for most of its lifetime (e.g., $\geq$96%), it could be more expensive than CF. This is why CF-based applications could result in lower prices [66, 69].

Takeaway: The unit price of CF is more than one order of magnitude higher than that of VM. However, CF is more elastic and is not billed for startup and idle time; thus, it may result in a lower price for sparse workloads. More importantly, only CF can provide interactive response latency for unpredictable workload spikes.

3.3 Architecture Overview

As discussed in Section 3.1 and Section 3.2, for dynamic workloads, there is no clear winner between VM and CF. To provide the optimal performance/price ratio when workload pattern changes, we propose our hybrid query engine Pixels-Turbo that uses CFs as the auxiliary computing resources to process workload spikes.

The architecture of Pixels-Turbo is illustrated in Figure 4. The blue blocks are the components of Pixels-Turbo. The solid and dotted arrows are the required and optional steps to execute a query. Like other cloud data lakes, we store the base tables in cloud object storage such as S3. The *Coordinator* can run in a small VM and is the only long-running component in the system. It is responsible for managing metadata (which is stored in key-value stores or relational databases), query parsing and scheduling, metrics collection, and returning the query result to users.

The *VM Workers* and the *CF Workers* are two groups of workers that execute queries collaboratively. The former is managed by an auto-scaling framework (see Section 3.4); The latter is managed by the CF service in the cloud. Both of them can scale to zero.

Query Execution Overview. As discussed in Section 3.2, compared to CFs, VMs requires a longer time to startup and scale out. Thus, CF Worker only participates in query execution when the VM cluster has no additional resources to process new coming queries while the new VMs have not been added to the cluster. This is controlled by the query queues in the coordinator (see Section 3.5).

The coordinator selects one of the two execution methods for each query: (1) If CF Workers are not used, the whole query plan is executed by the VM cluster; (2) If CF Workers are used, the coordinator pushes down the expensive operations, including table scans, joins, and aggregations, into a sub-plan. The top-level plan will consume the result of the sub-plan as a logical temporary table. Then the coordinator invokes the CF Workers to execute the sub-plan and send the top-level plan to the VM cluster to assemble the final query result. The sub-plan result is directly fetched

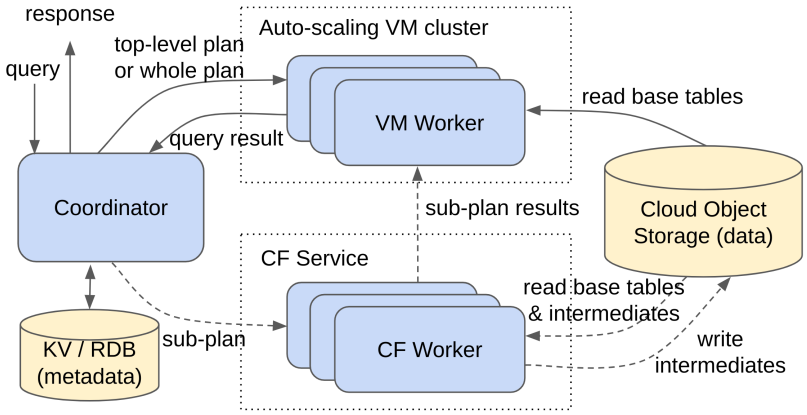


Fig. 4. Architecture of Pixels-Turbo.

by the VM Workers without transferring through the cloud storage. More details about query execution are discussed in Section 4.

Implementation. We use Trino [36] for VM-based query processing and use Pixels [45] for storage management in our implementation. Trino is a lightweight MPP query engine without its own storage. Whereas Pixels is an efficient columnar storage engine that can connect to Trino. In our query engine, Pixels provides metadata management and columnar storage layout for base tables and intermediates. The storage layer of Pixels has similar I/O optimizations as our baseline Starling. We use the operation pushdown API in Trino to implement hybrid query execution. We implement the query executors in CFs from scratch. Note that our solution can also be implemented on other computation frameworks (e.g., Spark) and storage engines (e.g., Iceberg [10] and Hudi [9]).

3.4 Cluster Auto-scaling

The auto-scaling framework consists of three parts: (1) a metrics collector that collects the workload and the VM status in the cluster; (2) a scaling manager that tracks the metrics and makes the scaling decisions; (3) the scaling utils that perform the scaling actions.

Metrics Collector. The metrics collector is running in the coordinator. It collects the metrics of the workload (e.g., the number of concurrent queries) and the VMs (e.g., CPU and memory usage). These metrics are consumed by the *Scaling Manager*.

Scaling Manager. In the scaling manager, users can define scaling policies, each triggering a specific scaling action when the metric value goes below or above the threshold. In this paper, we use the number of concurrent queries as the primary metric. The reason is that the MPP engines usually try to distribute a query to every core in every node to fully utilize all the computation and I/O resources. A few concurrent queries are enough to exhaust all the resources in the cluster if there is no resource isolation. To our observation, for TPC-H in Trino, more than two concurrent queries will not significantly reduce the overall execution time further. However, this is the case for decision support queries that have enough parallel tasks to occupy the whole cluster. If the query engine has to support multi-tenant resource isolation or run small operational queries, it could be more effective to monitor the CPU, memory, or I/O metrics of the cluster. Choosing the best metrics and thresholds are out of the scope of this paper. Users can choose suitable metrics and thresholds for their application scenario.

Active scale-out. We collect query concurrency value every 5 seconds. Scale-out action is triggered when a number of (by default 2) consecutive values are above the scale-out threshold.

Although this number is configurable, we set it small to actively scale out the VM cluster, thus reducing CF usage.

Lazy scale-in. We trigger scale-in action when the average value of the metric in a delayed period (by default 5 min) is below the scale-in threshold. The period is configurable. It avoids repeated and ineffective scale-in and out in a short period of time. This is reasonable because VMs are very cheap but slow to startup.

Scaling Utils. The scaling utils are composed of: (1) the OS image and VM template used by the scaling manager to create new VMs with the software stack preinstalled; (2) the *launch script* that is auto-started by the OS to launch the software stack and add the VM to the cluster; and (3) the *termination script* to gracefully shut down the software stack when the VM is about to terminate. The launch script also starts a daemon process to detect the VM termination notification signaled by the scaling manager and the spot VM reclaim notifications signaled by the cloud provider. The termination script will be executed by the daemon if the termination or reclaim notifications for the VM is detected. The scaling manager will create a new spot VM to replace the reclaimed one.

Users can use on-demand VMs or spot VMs (or a mix of them if the spot VM quota is not enough). There is little difference between them as the spot VM reclaim can be simply handled as a scale-in action triggered by the cloud provider. When AWS decides to reclaim a spot VM, it first signals a *rebalance notification* to CloudWatch [3] (a monitoring service for DevOps) that indicates the spot VM is at elevated risk of reclaim [21]. Then, two minutes before the actual reclaim, AWS signals another *reclaim notification* to CloudWatch [33]. We can proactively and gracefully shut down the software stack in the VM by detecting these notifications.

The auto-scaling framework is robust and easy to maintain. We directly use *Auto Scaling Group (ASG)* [11] in AWS as the scaling manager. It supports a variety of scaling policies. We use the step scaling policy in ASG [34] for auto-scaling. *CloudWatch* in AWS automatically collects the CPU, memory, and network metrics of the VMs. We can easily implement the metrics collector by reporting custom metrics (e.g., query concurrency) to CloudWatch. Other major cloud platforms also provide similar services as Auto Scaling Group and CloudWatch. These services ensure the robustness and availability of our cluster auto-scaling framework.

For sustained workloads, the VM cluster will scale to a stable capacity that can process all the queries without invoking CF workers. For bursty workloads, if the interval between two adjacent spikes is shorter than the delayed period of scale-in (e.g., 5 min), the VM cluster will maintain its capacity to process the spikes efficiently. This is cost-efficient as (spot) VMs are 1-2 orders of magnitude cheaper than CFs but take longer (1-2 min vs. ≈1 sec) to launch and release. When the interval between two adjacent bursts is always equal or slightly longer than the delayed period of scale-in, however, the VM cluster will be scaled in and out ineffectively. To solve the problem, we can either disable the VM cluster by setting the VM query queue length to zero or extending the delayed period of scale-in. We can make the decision according to cost analysis.

3.5 Query Queues

There are two query queues in the coordinator: (1) the *VM query queue* for the queries to be executed in the VM workers; (2) the *CF query queue* for the queries to be executed in the CF workers. The VM query queue has a higher priority than the CF query queue. A new coming query only enters the CF query queue when the VM query queue is full. Queries are dequeued when finished or aborted. Users can set the length of each queue. If the length of the VM query queue is the same as the query concurrency threshold that triggers scale-out actions, CF workers will immediately get involved in query execution when the cluster is about to scale out.

4 QUERY EXECUTION

As discussed in Section 3.5, for the query in the CF query queue, the top-level plan and the sub-plan are executed in the VM cluster (backed by Trino) and the CF workers, respectively. In this section, we introduce how the sub-plan is built and executed in CFs, and how the sub-plan result is consumed by the top-level plan.

4.1 Table Scan

To execute a table scan, the coordinator first reads the table's metadata and builds a set of *splits*, each containing the table scan parameters (e.g., schema and table names, projection, and predicates) and the locations of a set of row groups. The splits are distributed to the VM workers. Each VM worker launches a set of query tasks (threads) to process the splits. For a query in the VM query queue, the query tasks directly call the file format library to read the splits. For a query in the CF query queue, each query task packs a number of splits into a CF request and invokes a CF worker to process the selections and projections on the splits.

Direct Write-back. In the CF worker, the scan result on the splits, which is likely much smaller than the original data, is directly returned to the VM. This is done by attaching the VM's address to the CF request. Although CF does not have network ingress, it can access EC2 endpoints in a similar bandwidth of accessing S3. Then, the result is processed locally by the query task. Compared to passing results through the cloud storage, we reduce one network round trip in this way. We call this optimization *direct write-back*.

4.2 Join

When a query arrives, the coordinator parses the query and reorders the joins based on the statistics. For the queries to be executed in CFs, after pushing down table scans into the connector, the coordinator continues iterating the plan tree in a depth-first manner and tries to push down every iterated join into the connector. Following the order of join pushdown, we can reconstruct a sub-plan in the connector with the same join order as the original query plan. The result of the sub-plan can be seen as a temporary table, which is then consumed by the top-level plan like consuming scan results.

To execute the joins in CFs, same as in Starling [66] and Lambada [65], we implement two standard join algorithms: broadcast join and hash partitioned join. Broadcast join only requires one stage of CF workers, in which each worker builds a hash table in memory for the small table; and probes the hash table using the tuples from the splits of the large table. It is only used if the small table can be loaded into the memory of the CF worker. Otherwise, we use the hash partitioned join, in which the two tables are shuffled using the CF-based shuffle algorithm introduced in Section 2.4. We discuss the details of join algorithm selection in Section 5.3.

In the shuffle, say there are m producers and n consumers, then the producers write m partial partitioned files to S3; after that, the consumers issue $m \times n$ read requests to read the partitions from the partial partitioned files. It results in significant I/O overhead and hinders the parallelism of CPU and I/O. The combiners can reduce the number of read requests issued by the consumers, thus preventing them from being throttled by S3. But this introduces 2x intermediate I/Os (in bytes). To reduce the intermediate I/Os, we merge the producers into the last stage of the previous operator (if any). This optimization is also applied in Starling [66].

Chain Join. To further reduce the I/O overhead in the CF-based joins, we propose *chain join*. The idea is to merge the sequential broadcast joins into the same stage of CF workers until reaching the shuffle boundary. Thus, the result of a broadcast join in the sequence is directly used to build the hash table of the subsequent join without writing intermediates to S3. In our join-algorithm selection strategy (Section 5.3), we estimate the size of each side in the join recursively. Thus, we

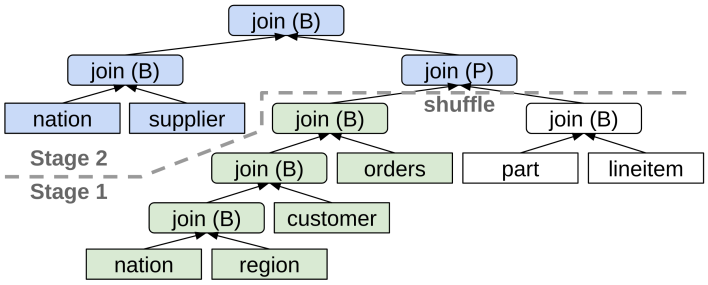


Fig. 5. Illustration of the chain joins. Blocks in the same color are merged into the same chain join.

ensure all the tables in the sequential broadcast joins, except the large table in the last join, can be broadcast.

An example of the join plan with chain joins is shown in Figure 5, where *join (B)* and *join (P)* denote the broadcast and hash partitioned join, respectively. Small tables are on the left side. In this example, there are one chain-join under the shuffle boundary (in green) and one chain-join above the boundary (in blue). In the green chain join, the last (top) broadcast join is the core join; all the other joins are executed in the same group of CF workers of the core join. We call this kind of chain joins *broadcast chain join*. Whereas in the blue chain join, the core join is the hash partitioned join; all the broadcast joins are executed in the consumers of the core join. We call this kind of chain joins *partitioned chain join*. The producers of the hash partitioned join are merged into the CF workers of the green chain join and the broadcast join of *part* and *lineitem*. Therefore, we only need two stages to execute this 8-table join.

Pipelining consecutive broadcast joins is straightforward in MPP databases that execute queries using memory-buffered pipelines. To the best of our knowledge, however, merging consecutive broadcast joins into a single stage of mappers (producers) or reducers (consumers) is not proposed in MapReduce-like query engines. This is challenging as the memory-buffered pipeline no longer exists. It requires a major rewrite of the query plan and the join algorithms.

Early Projection. Many DBMSs, such as PostgreSQL and Trino, prune the unnecessary columns (e.g., the join keys that are not needed by other operators) as early as possible. This is also supported in our CF-based joins. Besides pruning unnecessary join keys in the join result, for hash partitioned join, we generate a projection operation for the producer stage to early prune the columns that are only used in the selection predicates. Thus, we reduce the I/O and encoding/decoding cost of the intermediate results. We call this optimization *early projection*. It is applied in traditional DBMSs but neglected in CF-based query engines [65, 66].

4.3 Aggregation

The coordinator tries to push down aggregations after pushing down table scans and joins. We implement the same CF-based aggregation algorithm in Starling [66] that executes aggregation in two stages: (1) a set of CF workers each writes a file containing the partial aggregates to S3; (2) a final CF worker reduces the partial aggregates into a final aggregate. The first stage is merged into the last stage of the previous operator (if any). We call this algorithm *direct aggregation*, which is feasible if the partial and final aggregates are small. For large aggregates, Starling partitions the input data (i.e., table scan or join result) on the group-by keys before generating partial aggregates, so that the second stage can run in parallel workers. We call this *shuffled aggregation*.

Late Shuffling. Shuffled aggregation generates a large amount of intermediate data. To reduce the intermediate I/Os, we postpone the shuffle to the partial aggregates, i.e., we shuffle the partial

aggregates instead of the input data. As the partial aggregates are generally much smaller than the input data, *late shuffling* greatly reduces the amount of intermediate I/Os in shuffled aggregation.

4.4 Implementation

In addition to the optimized query operators in CFs, at the implementation level, we have the following optimizations that are critical for providing a reasonable performance, although they may have been applied in other data analytic systems.

Zero Materialization. The file format library reads columnar row batches from the files. In joins and aggregations, instead of reconstructing row-wise tuples, we directly use the row batches to build/probe hash tables or compute aggregates by storing an intra-row-batch offset for each tuple. This dramatically reduces the memory copy and garbage collection overhead.

Vectorized Filtering and Hashing. The selection predicates on each table are transformed into a conjunctive normal form (CNF). All the predicates on the same column form a clause in the CNF. For each row batch, we produce a bitmap for the clause on each column using vectorization. Then, we use bitwise operations on the bitmaps to evaluate the CNF and get the filtering result on the row batch. In joins and aggregations, we compute the hash value of the keys for each row batch also using vectorization.

Asynchronous Invocation. We invoke CFs using the asynchronous API of AWS Lambda. In this way, we can easily perform thousands of invocations per second, thus avoiding the complexity and overhead of the multi-level invocation in Lambada [65].

5 SUB-PLAN OPTIMIZATION

In Pixels-Turbo, we use the query optimizer of Trino to optimize the original query plan. We store the statistics of the tables and columns, including row count, column size, average column-chunk size, cardinality, null fraction, and min/max. These statistics are exposed to the Trino optimizer for query optimization. In addition to the Trino optimizer, we have an optimizer responsible for the physical optimization (planning) of the sub-plan based on the statistics. We introduce the sub-plan optimization in this section.

5.1 Instance Size of Cloud Functions

AWS Lambda allows users to set the memory size of the CF instance (ranges from 128MB to 10GB) and allocates a proportional number of vCPUs to the instance. We use the largest instance size (i.e., 10GB) for the sub-plan because: (1) larger instance requires fewer CF workers, and therefore fewer intermediate files and less S3 request cost for shuffles (see Section 4.2); (2) I/O performance of CF can only scale out to 150-200 instances (see Section 3.1); therefore a large number of small instances with the same total amount of memory will not necessarily improve the performance; (3) the executor runtime has a constant memory overhead and a background thread to retry the I/O stragglers, therefore using larger instance is more memory-and-CPU efficient. In our evaluation, we find that using the largest instance usually delivers the best performance and price, even for small queries with fewer than 150 instances.

5.2 Split Size of Base Tables

As discussed in Section 4.1, the coordinator builds a set of *splits* for each base table. The split size, i.e., the number of row groups in each split, determines the parallelism of the operators that read base tables. Each split is read, decoded, and filtered by a thread in the worker. For joins and aggregations, the filtered result is continuously consumed by partitioning, hash table probing, or aggregates computing. Vectorized filter and partitioning are very efficient (see Section 4.4). Their cost is negligible compared to the cost of read-and-decode (which is I/O and memory intensive),

hash table probing, and aggregates computing (which is CPU intensive). Therefore, we calculate the split size for CF workers by Equation 1:

$$S_s = \min(\frac{cap_{sb}}{\sum_{c_i \in C} chunk\_size(c_i)}, \frac{g * cap_{sr}}{S_r}) \tag{1}$$

where cap_{sb} is the cap of the number of bytes read from a split, cap_{rs} is the cap of the number of filtered rows per split, g is the total number of row groups in the table, C is the set of columns to read, $chunk\_size$ refers to the average size of the column chunks of the column in all row groups. and Sr is the filtered rows on the tables (Equation 3). The default values of cap_{sb} and cap_{sr} are 128MB and 5M rows, respectively. The $chunk\_size$ of each column is read from the statistics. We have 8 threads in each CF worker to process splits, as there are 6 vCPUs in each worker and it is commonly recommended to use slightly more threads than CPUs for parallel query processing [37, 46]. By setting cap_{sb} and cap_{sr}, we can ensure the CF workers will not block on I/O or CPU or fail on out-of-memory (OOM) errors.

5.3 Join Algorithm Selection

We set two *size caps* for broadcast tables, one in bytes (cap_{bb}), the other in the number of rows (cap_{br}). cap_{bb} is to control the reading and decoding cost of the broadcast table; cap_{br} is to control the cost of building the hash table (too many entries will increase the collision rate as our hash value is a 32-bit integer; using 64-bit hash value and the larger hash table will increase memory consumption and may lead to OOM). Their default values are 256MB and 20M rows, respectively. To decide whether use broadcast join or hash partitioned join, we estimate the table size in bytes and rows by Equation 2 and Equation 3, respectively:

$$S_b = \sum_{c_i \in C} size(c_i) \tag{2}$$

$$S_r = n * \min_{c_i \in C_p} selectivity(c_i) \tag{3}$$

where C is the set of columns to read, C_p is the set of columns in the predicates, and n is the row count of the table. As we currently do not have histograms on the table, we estimate the joint selectivity on the table conservatively using the minimum selectivity on the columns (as the predicates on the table are represented as a CNF on the columns, see Section 4.4) [4]. The joint selectivity might be overestimated in this way, but it avoids broadcasting oversized tables. The selectivity of a column is estimated with the assumption of a uniform distribution between min and max. With S_b and S_r, we select broadcast join if the sizes of the smaller side of the join are under the caps. If a side of the join is a child join, we estimate the result size of the child join recursively by adding up the sizes of its children, as there are no histograms.

5.4 Aggregation Algorithm Selection

After applying *late shuffling* (see Section 4.3), *direct aggregation* can be seen as a *shuffled aggregation* with only one partition; thus, the aggregation algorithm selection problem becomes the partition number selection problem. To decide the number of partitions, we need the size of the shuffle intermediates. For aggregation, that is the size of the partial and final aggregates, which are determined by the joint cardinality of the group-by keys that is estimated as:

$$D_g = \max_{c_i \in C_g} cardinality(c_i) \tag{4}$$

[4]Although various types of histograms can be used to improve the estimation accuracy, maintaining these histograms in an open data lake is expensive and challenging, as the data is ingested from different data sources and manipulated by different applications. We plan to address this problem in the next step.

Table 2. Representative queries in the log analysis workload.

	q1	q2	q3	q4	q5
# group keys	2	1	1	1	2
types of keys	date,timestamp	string	string	date	date,string
card. of keys	13K	53.7M	2984	234	53.7M
selectivity	0.19%	14.6%	14.6%	14.6%	0.14%
# result rows	1440	17.7M	2513	1	233K

where C_g is the set of group-by key columns. We do not use the production of each column's cardinality because it often leads to an extreme overestimation of the joint cardinality. Calculating the partition number from the cardinality is discussed in Section 5.5.

5.5 Number of Partitions

It is important to decide the number of partitions in shuffle-based operators. For hash partitioned join, similar to join algorithm selection (see Section 5.3), we have two caps for the partition size, one in bytes (cap_{pb}) and the other in the number of rows (cap_{pr}). Their default values are 1024MB and 10M rows, respectively. Then, we calculate the number of partitions by Equation 5:

$$P_j = \max(\frac{S_{b1} + S_{b2}}{cap_{pb}}, \frac{S_{r1} + \alpha * S_{r2}}{cap_{pr}}) \tag{5}$$

where S_{b1} and S_{b2} are the sizes in bytes of the two tables, S_{r1} and S_{r2} are the sizes in rows of the two tables, and α is the speed ratio of hash building and hash probing. In our implementation, hash building is about one order of magnitude slower than hash probing, thus we set α to 0.1. In the equation, $S_{b1} + S_{b2}$ is to control the read-and-decode cost on each partition, whereas $S_{r1} + \alpha * S_{r2}$ is to control the join cost on each partition. For shuffled aggregation, we calculate the number of partitions by dividing the joint cardinality of the group-by keys by cap_{ar} which is the cap of the number of rows per partition in shuffled aggregation. As grouping tuples plus computing aggregates is more expensive than building a hash table, we set a smaller default value of 2.5M rows for cap_{ar}.

6 EVALUATION

6.1 Experimental Setup

We first evaluate the query latency, monetary cost, and scalability of the pure VM and CF modes of Pixels-Turbo and the other baselines. Thus we confirm whether VM-based query execution is more cost-efficient and scalable than CF-based query execution, which is the motivation for designing a hybrid query engine. Then, we evaluate the elasticity of Pixels-Turbo and the baselines. After that, we evaluate the effects of the optimizations in this paper and explain how these optimizations contribute to the end-to-end results. We use two workloads in our evaluations:

TPC-H. We perform most of the evaluations related to table-scan and joins on the TPC-H dataset of scale factor 1000 (1TB). In the scalability experiment, we also use the dataset of scale factor 100 (100GB). Within the 22 TPC-H queries, we select q_1 and q_6 as the representatives of table-scan queries, and q_3, q_5, q_7, q_8, q_{10}, q_{11}, q_{12}, and q_{14} as the representatives of join queries. The selected table-scan queries cover the cases of low and high selectivity. The selected join queries cover the cases of 2, 3, 4, 6, and 8-table joins with different input sizes and join selectivity.

Real-world Log Analysis. Trino currently does not support pushing down most of the aggregations with expressions in TPC-H; hence, we evaluate the aggregations on a real-world log analysis

workload from a well-known Internet company. The workload has a single table with 105 columns, in which 52 are string columns, 25 are boolean columns, 21 are bigint columns, and the rest are date/timestamp (ts)/int columns. The size of the dataset is 3.51TB, with 3.94 billion rows. There are 145 queries of the following form:

```
SELECT ... FROM t WHERE ...
GROUP BY ... [ORDER BY ...] LIMIT n;
```

Each query accesses 10.3 columns on average. As shown in Table 2, we select 5 representative queries for our evaluations, where *card. of keys* is the joint cardinality of the group-by keys, *selectivity* is the joint selectivity of the selection predicates, and *# result rows* is the number of rows in the aggregation result (not limited).

We evaluate 4 systems in our experiments:

Pixels-Turbo. In terms of storage, we would like to reuse the I/O optimizations in Starling for the fairness of comparison. However, Starling is not open-source. Therefore, we use Pixels format [45] as an alternative for storing base tables and intermediate results. Pixels and Starling use similar file layouts and have similar I/O optimizations, including (1) They both hide I/O latency by reading data items asynchronously or in parallel; (2) They both have I/O straggler mitigation policies to reduce I/O tail latency. Pixels currently only supports columnar encoding, such as run-length and dictionary encoding. Heavy compression algorithms such as Snappy and ZSTD are not supported. For TPC-H 1TB and 100GB, the Pixels data sizes after encoding are 710GB and 69.6GB, respectively. For the log analysis dataset, the Pixels data size after encoding is 1.35TB.

In terms of computation, we use AWS Lambda for the CF engine and EC2 for the VM engine (i.e., Trino), respectively. For Lambda, we enable all the optimizations in Section 4 and use the default optimizer parameters in Section 5. For EC2, we use the instance type *m5.8xlarge* (32 vCPUs, 128GB RAM, 10Gbps network). We use spot VM in all the evaluations in this section. We requested AWS to increase our Lambda parallelism from 1000 to 10000 and increase our vCPUs quota for both spot and on-demand VMs from 640 to 1600, thus we can create enough CF and VM instances.

To know the query latency, monetary cost, and scalability of the queries on different computing services, we force Pixels-Turbo to execute queries in either VMs (namely *Pure-VM*) or CFs (namely *Pure-CF*) in the corresponding evaluations. Pure-VM relies on Trino to run queries. We only provide statistics to Trino through our customized connector for query optimization. In the elasticity evaluation, we let Pixels-Turbo work in the *Hybrid* mode.

Starling (simulated). We re-implemented the scan, join, and aggregation algorithms in Starling [66] using Pixels as the storage format. Thus, we simulate Starling by forcing Pixels-Turbo to execute the queries using these algorithms. We call this *Starling-S*. The data used in Starling-S is the same as in Pixels-Turbo.

Redshift-serverless. We denote it as *Redshift-SL* in the evaluations. As introduced in Section 2.2, Redshift-SL and Athena are the two serverless query services in AWS. They can query the open-format (e.g., Parquet) data in S3. We would like to use the same file format implementation in all the systems. However, Redshift-SL and Athena do not support custom file formats (e.g., Pixels). Therefore, we use Parquet in these two systems. To be fair, we do not use heavy compression in Parquet and ensure that the average number of rows in each row group is about the same in both Parquet and Pixels. In this way, the encoded data size of each Parquet file is around 128MB, which is recommended by Redshift and Athena [6, 35]. For TPC-H 1TB and 100GB, the Parquet data sizes after encoding are 769GB and 75.2GB, respectively. For the log analysis dataset, the Parquet data size after encoding is 1.5TB.

In Redshift-SL, users can set the parallelism of RPUs (Redshift processing units, each has 16GB memory [27]) to 32 or above, thus controlling the performance and monetary cost. In the evaluations, to compare Redshift-SL, Starling, and Pixels-Turbo, we tune the parallelism of RPUs to ensure

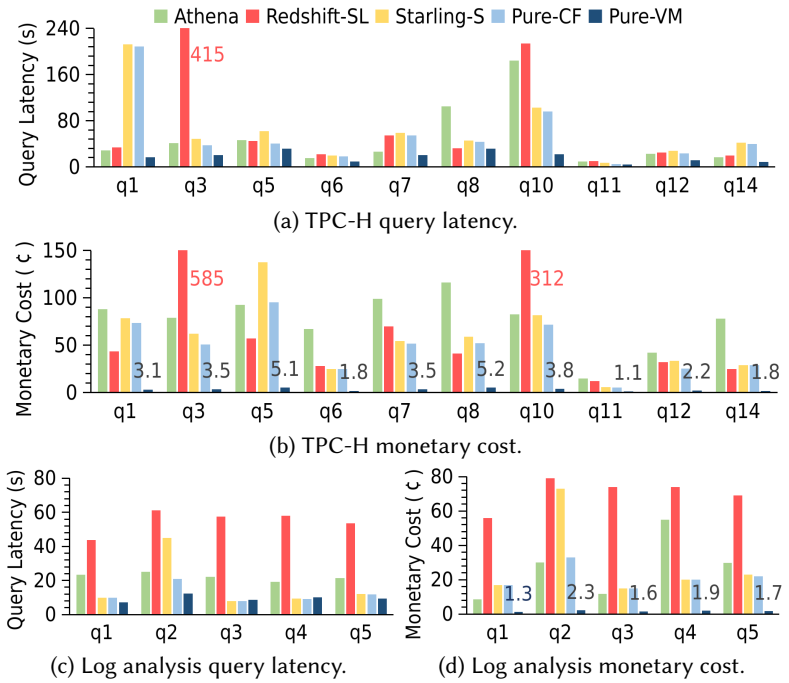


Fig. 6. Query latency and monetary cost on TPC-H 1TB and log analysis datasets.

Redshift-SL has the same total memory capacity as other systems. This is fair because memory is essential for query processing, and the other resources in the cloud, such as vCPUs and network bandwidth, are often allocated proportionally to the memory capacity.

Athena. As discussed above, we use the same Parquet datasets in Athena and Redshift-SL. Athena does not allow users to tune the underlying cluster or query engine. Hence, we can not control the amount of resources used by Athena. In the experiments, we use Athena engine version 3, which has integrated some improvements from the Trino and Presto open-source communities [2].

6.2 Query Latency

In this evaluation, to compare the systems fairly, we ensure each system (except Athena) has 2TB of total memory. With this configuration, Redshift-serverless (Redshift-SL) uses 128 RPUs; Starling-S and Pure-CF use 200 parallel Lambda instances; Pure-VM has 16 m5.8xlarge VMs. Athena is automatically scaled by the backend platform. For query optimization, we provide the row count of each table to Athena and Redshift-SL as this is the only statistic they support; we provide the statistics discussed in Section 5 to the Trino [36] optimizer to generate the optimized query plan for Starling-S, Pure-CF and Pure-VM.

The result is shown in Figure 6a and Figure 6c. Pure-VM is the apparent winner in this evaluation, thanks to higher I/O bandwidth and the efficient MPP engine. Currently, Athena can not effectively reorder the complex joins in TPC-H q_8 and q_{10}. It generates a similar join order as the one generated by the rule-based query optimizer in Trino or Presto. This is why Athena performs poorly on these queries. Redshift-SL has a similar problem. It can reorder the joins based on the row count of the tables and generate a good join order for TPC-H q_8. However, it generates sub-optimal join order for q_3 and q_{10}. In addition, Athena and Redshift-SL perform poorly on the aggregation queries (Figure 6c). This might be an implementation issue of the aggregation operator.

Starling-S and Pure-CF only differ in the scan, join, and aggregation algorithms. Both of them return the sub-plan result to the VM worker. In this and the following evaluations, we run a single VM worker located on the same VM of the Coordinator for Starling-S and Pure-CF. TPC-H q_1 has high selectivity, thus both Starling-S and Pure-CF perform poorly as they have to return 183GB scan results to the VM worker. Their performance is good on the other scan query q_6 with low selectivity. It can be seen that Starling-S and Pure-CF are slower than the other non-CF-based systems (especially Pure-VM) on the join queries. This is due to the S3-based shuffle discussed in Section 2.4 and the S3 bandwidth limitation discussed in Section 3.1. This performance gap is more evident on TPC-H q_5, which has high selectivity on the base tables. However, Pure-CF is faster than Starling-S on TPC-H q_3, q_5, and log analysis q_2. This is due to the *early projection*, *chain join*, and *late shuffling* optimizations discussed in Section 4.

Takeaway: MPP cluster (Pure-VM) is more efficient than CF-based query engines, although our CF-based optimizations can improve the performance of specific queries. Serverless query services based on clusters can be performant if they are not suffering from lousy query optimization or operator implementation. In addition, simply pushing down table-scan to CFs like what Redshift Spectrum does [43] is only efficient for low-selectivity queries.

6.3 Monetary Cost

The configuration of this evaluation is the same as in Section 6.2. For Pure-VM, we calculate the cost of a query by the cumulative VMs cost during query execution. For Pure-CF and the other systems, the cost of a query is calculated by the amount of resources consumed. There is an additional cost on the S3 requests for all these systems.

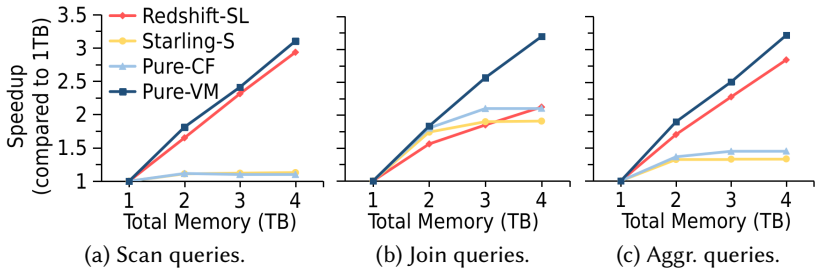
The result is shown in Figure 6b and Figure 6d. We can see that Pure-VM has 1-2 order of magnitude lower monetary cost than the other query engines on both TPC-H and log analysis workloads. For TPC-H queries, the average query cost of Pure-CF, Starling-S, Redshift-SL, and Athena are 47.86, 56.57, 120.51, and 75.87 US cents, respectively. For the log analysis queries, the average query cost of Pure-CF, Starling-S, Redshift-SL, and Athena are 21.4, 29.6, 70.4, and 27.04 US cents, respectively. As discussed in Section 3.4, Pixels-Turbo will work in pure VM mode for sustained workloads. Considering the higher performance of Pure-VM shown in Section 6.2, we claim that for sustained workloads, our query engine provides 1-2 orders of magnitude higher performance/price ratio than the serverless query engines.

The cost of Athena is proportional to the amount of data scanned ($5/TB), hence for the scan-heavy queries (e.g., TPC-H q_1, q_6, q_{14}, and log analysis q_4), it tends to be more expensive than the other systems. Redshift-SL charges on the cumulative RPU-seconds consumed by the query, hence the cost is proportional to the RPU parallelism and the query latency. The sub-optimal join order for TPC-H q_3 and q_{10} and the inefficient aggregation implementation dramatically increase its average cost. Starling-S and Pure-CF are more expensive than Pure-VM due to the higher query latency and higher price of AWS Lambda. However, we can see that the monetary cost gap between Starling-S and Pure-CF is larger than their performance gap. This is because the optimizations in Section 4 further reduces the amount of S3 requests, which is charged in addition to the Lambda cost.

Takeaway: For analytical query processing in the cloud, MPP cluster in spot VMs (Pure-VM) is 1-2 orders of magnitude more cost-efficient than other solutions. Our execution optimizations can generally reduce the monetary cost of queries in CFs.

6.4 Scalability

We evaluate the scalability of the systems from two dimensions: (1) increasing both data size and hardware scale to see whether the system can scale out to support larger datasets; (2) only increasing



(a) Scan queries. (b) Join queries. (c) Aggr. queries.

Fig. 7. The speedup of scan, join, and aggregation queries in different systems under different hardware scales (i.e., total memory capacities).

the hardware scale to see whether the system can scale out to provide lower query latency (i.e., higher query throughput). The latter is more significant for interactive analytical workloads, as the query throughput may change faster than the data volume. We control the hardware scale by tuning the total memory capacity of the system, except for Athena. For Pure-CF and Starling-S, we tune the total memory capacity by proportionally tuning the caps cap_{sb}, cap_{sr}, cap_{pb}, and cap_{pr}, introduced in Section 5.

In the first dimension, we use 512GB memory for TPC-H 100GB because of the minimum RPUs (i.e., 32) that Redshift-SL supports. The result is compared with the query latency in Section 6.2 (TPC-H 1TB, 2TB memory). That is, the data size is increased by 10x while the amount of resources (total memory) is increased by 4x. If the system scales linearly (ideal), the query latency will increase by only 2.5x on TPC-H 1TB. The result shows that in Athena, Redshift-SL, Starling-S, Pure-CF, and Pure-VM, the average query latency on TPC-H 1TB increased by 5.48x, 4.43x, 4.55x, 4.14x, and 2.69x, respectively. It can be seen that Pure-VM has the best scalability, whereas Pure-CF is more scalable than Starling-S. Athena and Redshift-SL do not suffer from the sub-optimal join orders on TPC-H 100GB as the dataset is small enough.

In the second dimension, we use the TPC-H 1TB and the log analysis datasets, but tune the memory capacity from 1TB to 4TB. Athena is not evaluated. The result is shown in Figure 7. On scan queries, both Pure-VM and Redshift-SL scale linearly, whereas both Pure-CF and Starling-S only scale to ≈1.1x because of the bandwidth bottleneck of reading data and returning results. On join queries, Pure-VM scales near-linearly. Redshift-SL suffers from the sub-optimal join orders on TPC-H q_3 and q_{10}. If we exclude these two queries, its speedup on 4TB memory increases from 2.12x to 2.52x. Pure-CF has better scalability than Starling-S, although they are both bounded by the I/O bandwidth at a scale larger than 2TB. On aggregation queries, Pure-VM and Redshift-SL scale linearly, whereas Pure-CF and Starling-S are in a similar condition as on join queries (but they have lower speedups on aggregation queries as the aggregation queries are more affected by the I/O latency on small partial aggregates).

Takeaway: The systems designed for clusters with inter-node communication (Pure-VM and Redshift-SL) have better scalability if not affected by the sub-optimal query plan. Our execution optimizations can improve the scalability of the queries in CFs.

6.5 Elasticity

In the aforementioned evaluations, we demonstrate that for sustained workloads, Pixels-Turbo is significantly more cost-efficient and scalable than the serverless query engines. In this evaluation, we further confirm that for bursty workloads, Pixels-Turbo achieves a high performance/price ratio without compromising the elasticity of allocating resources.

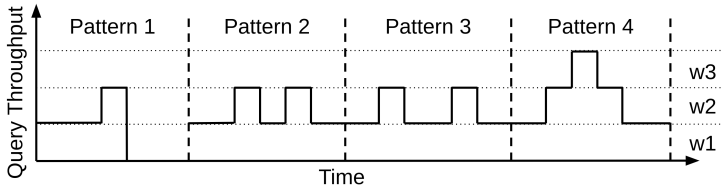


Fig. 8. Illustration of the representative workload patterns.

We evaluate the elasticity of Hybrid (the hybrid mode of Pixels-Turbo), Pure-CF, Pure-VM, Athena, and Redshift-SL on four representative workload patterns shown in Figure 8. The first pattern demonstrates a single workload spike followed by a cliff-like workload drop; The second pattern demonstrates two consecutive workload spikes with an interval shorter than the delayed period of scale-in (see Figure 9c); The third pattern demonstrates two consecutive workload spikes with an interval slightly longer than the delayed period of scale-in; The last pattern demonstrates two overlapping workload spikes. Each workload pattern lasts for one hour. To generate the workload patterns, we use three workloads: w_1 is a simple-join workload running TPC-H q_{12} at the rate of 3 queries per minute (QPM); w_2 is a scan-and-aggregate workload running log analysis q_2 at 3 QPM; And w_3 is a complex-join workload running TPC-H q_5 at 2 QPM. Each workload is submitted by a JDBC client. In workload pattern 1, we run w_1 in the first 40 minutes and run w_2 between minutes 31-40 (both inclusive, the same below). In workload patterns 2, 3, and 4, we run w_1 as the background workload. To simulate the workload spikes, in pattern 2, we run w_2 between minutes 31-34 and minutes 39-42; In pattern 3, we run w_2 between minutes 31-34 and minutes 41-44; In pattern 4, we run w_2 between minutes 31-40 and run w_3 between minutes 35-36.

We use m5.8xlarge spot VMs in Hybrid and Pure-VM. For Hybrid, we launch 8 VMs as the initial capacity, and use the following scaling policies: (1) Add 100% instances when two consecutive query concurrency values (see Section 3.4) are larger than 2 (same as the VM query queue length, see Section 3.5); (2) Release 50% instances when the average query concurrency within 5 minutes is in the range of [0.25, 0.5); (3) Release 75% instances when the average query concurrency within 5 minutes is in the range of [0, 0.25). When the current scaling action is not completed, other actions cannot be triggered. For Pure-VM, we provision the VMs to ensure that workload spikes can be processed smoothly. In our experiments, Redshift-SL did not show any auto-scaling behavior, and we did not find any related settings in the documents. Therefore, we set the base RPUs (default capacity) of Redshift-SL to ensure that workload spikes can be processed without congestion.

The monetary costs of the systems under the workload patterns are shown in Figure 9. We measure the cost at the minute granularity, and the cost of each query is attributed to the minute it is submitted. Following the officially recommended method to check the usage [40], we found that Redshift-SL is actually billed on a per-minute basis. Therefore its monetary cost is consistent as long as every minute has queries. Athena is charged for the amount of data scanned, thus its lines are flat when the query rate is stable. Pure-CF is charged for the running time of Lambda, thus its lines float with the latency of the queries.

In Figure 9, we can see that: (1) For all the workload patterns, Hybrid provides the lowest cumulative monetary cost; (2) Same as the serverless query engines, Hybrid has the ability of scale-to-zero when the workload is stopped (see Figure 9a); (3) Upon workload spikes, Hybrid is as elastic as the serverless query engines in rapidly allocating more resources; (4) Lazy scale-in can effectively handle frequent workload spikes and provide a high cost-efficiency (see Figure 9b), but it is ineffective when the spikes interval is slightly longer than the delayed period of scale-in; (5) Active scale-out can effectively reduce CF usage (i.e., costs) for long spikes (e.g., 10 minutes in

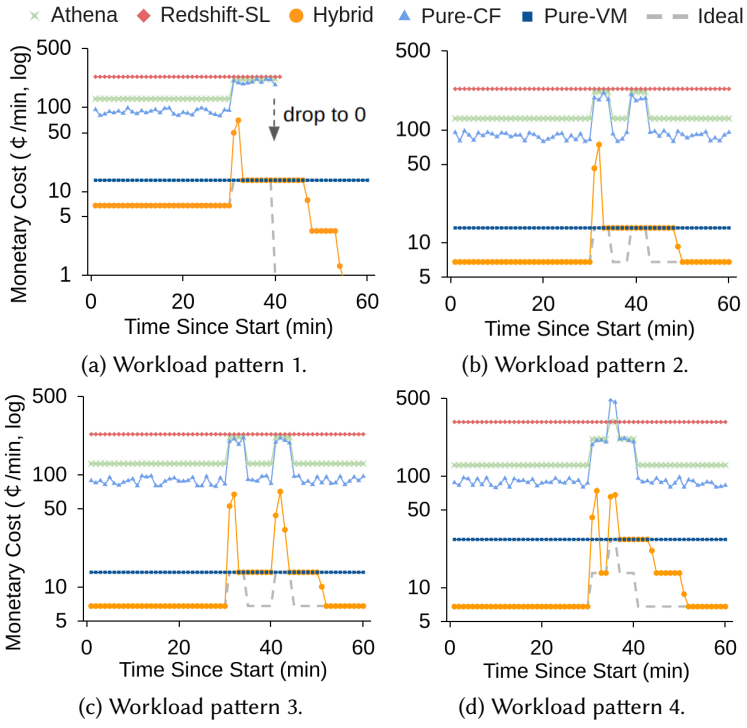


Fig. 9. Monetary cost per minute of the workload patterns.

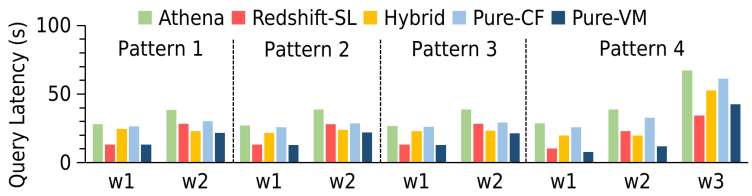


Fig. 10. Average query latency of the workload patterns.

Figure 9a) and frequent spikes (e.g., in Figure 9b), but it may lead to additional costs (e.g., during minutes 37-44 in Figure 9d) for a single short spike of 1-2 minutes. As discussed in Section 3.4, we might achieve a better performance/price ratio for our workloads by tuning the auto-scaling parameters.

The average query latency of the workload patterns is shown in Figure 10. We can see that Hybrid provides lower query latency than Athena and Pure-CF in all cases, as most of the queries are efficiently processed in the VM cluster. Redshift-SL is faster than Hybrid on w_1 (i.e., TPC-H q_{12}) and w_3 (i.e., TPC-H q_5), whereas Pure-VM is faster than Hybrid in all cases. The reason is that Redshift-SL and Pure-VM are provisioned for the workload spikes. However, provisioning brings higher monetary and operation costs.

Takeaway: For the evaluated bursty workload patterns, Hybrid provides the lowest monetary cost and scales rapidly when the workload grows. Further, Hybrid provides lower query latency than the auto-scalable baselines.

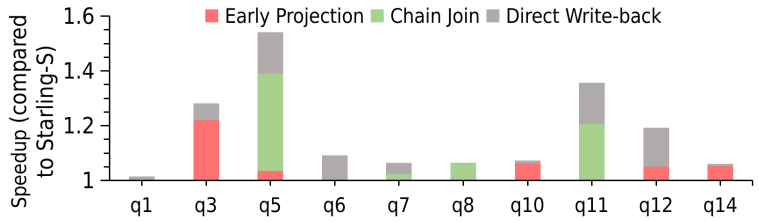


Fig. 11. Speedup of the optimizations on TPC-H 1TB.

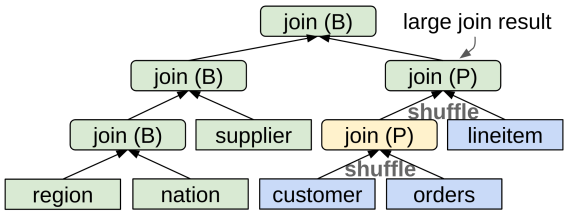


Fig. 12. The join plan of TPC-H q_5 in the hybrid mode of Pixels-Turbo.

6.6 Effects of Early Projection

We evaluate the performance gains of our query execution optimizations using the same configuration in Section 6.2. The result on TPC-H queries is shown in Figure 11.

As discussed in Section 4.2, in hash partitioned joins, early projection avoids shuffling the columns only used in selections. Figure 11 shows that early projection improves the performance of TPC-H q_3, q_5, q_{10}, q_{12}, and q_{14}. Especially for TPC-H q_3, early projection reduces the intermediate results in shuffling from 65.7GB to 59.65GB, thus speeding up the query by 1.22x. Note that early projection not only reduces the amount of I/Os but also reduces the CPU time on encoding and decoding the redundant columns.

6.7 Effects of Chain Join

As discussed in Section 4.2, chain join reduces the amount of intermediate I/Os in executing the query plan that has sequential broadcast joins. In Figure 11, we can see that chain join is effective on TPC-H q_5, q_7, q_8, and q_{11}. They have complex joins of 3-8 tables. We find that chain join is more effective in two cases: (1) the query plan only contains one sequence of broadcast joins (e.g., TPC-H q_{11}); (2) the query plan has a sequence of broadcast joins on top of a high-selectivity join (e.g., TPC-H q_5). In the first case, all the joins are merged into a single stage without any intermediate I/Os. The second case is illustrated by the join plan of TPC-H q_5 in Figure 12, where blocks in the same color denote the same stage; *join (B)* and *join (P)* denote the broadcast and hash partitioned joins, respectively. In this query, the two hash-partitioned joins have a relatively high selection and join selectivity. Therefore the intermediate join result is large. With the partitioned chain join (in green), the large intermediate join result is directly consumed as the probe side by the root of the join plan without being materialized.

6.8 Effects of Direct Write-back

As discussed in Section 4.1, direct write-back returns the result of the sub-plan directly to the VM workers. In Figure 11, we can see that direct write-back has performance gain on most of the queries. However, the performance gain is insignificant in the cases: (1) the sub-plan result is very

large (e.g., TPC-H q_1, q_{10}), therefore the result transfer is bounded by the network inbound of the VM worker no matter where the result comes from (cloud storage or CF worker); (2) the sub-plan result only contains a few small files (e.g., TPC-H q_8, q_{14}, and all the log analysis queries), therefore the transfer cost of the result is negligible. On the queries that return a large number of result files from the sub-plan, such as TPC-H q_5, the performance gain is more significant because in this case, direct write-back reduces many network round-trips in fetching the result. However, it has a risk of causing disk/memory-full errors if the result in the VM is not cleaned properly.

6.9 Effects of Late Shuffling

As discussed in Section 4.3, late shuffling avoids shuffling the large base table when the joint cardinality of the group-by keys is large. Among the selected queries of the log analysis workload (see Table 2), q_2 and q_5 have large cardinality (53.7M) on the group-by keys. For Pure-CF, the sub-plan optimizer chooses shuffled aggregation for these two queries and sets the number of partitions to 21 (see Section 5.4 and Section 5.5). For Starling-S, late shuffling is disabled, we hard-code the number of partitions for the base table partitioning to the same number 21. In Figure 6c, we can see that late shuffling has a 2.15x speedup on log analysis q_2 but no gain on q_5. The reason is that log analysis q_5 has a very low selectivity (0.14%), hence the table scan result is already small, and early or late shuffling does not make a significant difference in this case.

6.10 Effects of Query Optimization

Trino Optimizer. In Pixels-Turbo, the Trino optimizer is responsible for join reordering. To do this, we provide the table and column statistics to Trino through the connector API. However, we can turn off the cost-based join reordering in Trino using configuration properties or by returning empty statistics. We compared the performance of the join queries on TPC-H 1TB (in a cluster of 16 m5.8xlarge VMs) with and without cost-based join reordering. The result shows that cost-based join reordering provides an average speedup of 1.48x. In Presto (where Trino forked from), the cost-based join reordering is disabled by default [25]. This might be a minor reason why Starling outperformed Presto in [66], in addition to the proposed I/O optimizations such as parallel reads and straggler mitigation. In this paper, we enable both I/O optimizations and cost-based query optimization for the VM workers, thus Pure-VM can outperform Pure-CF and Starling-S.

Tunable Sub-plan Optimizer. As discussed in Section 5, there are 7 tunable caps in our sub-plan optimizer. In Section 6.4, we tune 4 of them, i.e., cap_{sb}, cap_{sr}, cap_{pb}, and cap_{pr}, to scale Pure-CF and Starling-S to different number of parallel Lambda workers. The result shows the effectiveness of these caps. The cap for shuffled aggregation, i.e., cap_{ar}, is not tuned as the cardinality of the data is not changed. The other two caps, i.e., cap_{bb} and cap_{br}, are used as the threshold of broadcasting. They are set according to the capacity of the CF instance. We also tried to manually tune the sub-plan for TPC-H queries. The result shows that the manually tuned sub-plans only outperform the sub-plans generated through our sub-plan optimizer by 9.1% under the same monetary cost. However, there is a precondition that the data distribution in TPC-H is approximately uniform. For non-uniform distributions, we need to further improve our sub-plan optimizer, for example, by adding histograms.

7 RELATED WORK

Serverless Data Processing. In addition to the serverless query services discussed in Section 2.2, there are some general-purpose data processing services in the cloud, such as elastic MapReduce [1, 13]. These services help customers reduce the maintenance cost of big data frameworks. In the research community, many works focus on cloud functions (CFs). [54] points out the limitations of CFs for data processing. Whereas Boxer [64] and Sonic [61] propose solutions to address some of

these limitations. PyWren [56], Locus [67], Flint [57], and Astrea [55] propose solutions to execute and optimize the MapReduce-style jobs in CFs. Starling [66] and Lambada [65] propose frameworks to execute SQL queries in CFs. Our paper proposes a solution to execute queries adaptively in CFs and virtual machines (VMs), thus achieving a higher cost-efficiency without compromising elasticity. It can be used as the backend of serverless query services.

Data Processing using Spot VMs. Spot VM is promising for cost-efficient data processing in the cloud. Redy [78] and CompuCache [77] use Spot VMs to reduce the monetary cost of a distributed memory cache. MHVMs [52] studies the memory allocation strategies for spot VMs and the strategies' effects on the data processing frameworks running in spot VMs. Our paper uses spot VMs to improve the cost-efficiency of query processing.

Hybrid Query Execution. HadoopDB [41] and Hadapt [44] integrate relational databases as a kind of co-processor into the MapReduce framework. PushdownDB [73] pushes down the scan and partial join/aggregation into the smart storage such as *S3-Select* [29]. Redshift Spectrum [43] pushes down selection and projection into a function service that is similar to Lambda. Modularis [59] improves the query engine's portability on different infrastructures by dividing relational operators into sub-operators. We relate to these works in the sense of pushing down query operators into a different runtime. But instead of finding the best runtime for each operator, our purpose of pushing down is to temporarily divert the workload spikes into the auxiliary executor for better elasticity. Therefore, our pushdown is more aggressive, and our auxiliary executor has the full functionality for executing a query.

Query Optimization. Traditional DBMSs (e.g., PostgreSQL) and MPP databases (e.g., Greenplum [60] and Redshift (cluster) [53]) have powerful cost-based query optimizers. Early Hadoop systems dropped the query optimizer. Later, SQL-on-Hadoop systems, such as Hive, Presto, and Spark, bring the optimizers back into their kernels [42, 70]. However, maintaining complex statistics in an open data lake is challenging; hence, most of the optimizers are either rule-based or based on simple statistics. And they are designed for clusters. Our paper proposes a cost-based optimizer for SQL queries in CFs based on the limited statistics in data lakes.

8 CONCLUSION

This paper reveals that cloud functions are more elastic than virtual machines but are less cost-efficient and scalable. To combine the elasticity of cloud functions and the cost efficiency and scalability of virtual machines, we propose Pixels-Turbo - a hybrid query engine that uses cloud functions as the auxiliary computing resource for query processing. The query engine executes queries in an auto-scaling spot virtual machine cluster by default and invokes cloud functions to temporarily process the queries when the cluster has not scaled out to the necessary scale. To further improve the performance of the workload spikes, we propose execution optimizations and the cost-based optimizer for query execution in cloud functions. Compared to existing serverless query engines, Pixels-Turbo is 1-2 orders of magnitude more cost-efficient for sustained workloads while not compromising elasticity for unpredictable workload spikes.

ACKNOWLEDGMENTS

We would like to thank Dr. Panagiotis Sioulas for his suggestion to use spot instances to improve the cost efficiency of our query engine further. We thank Dimitra Tsaoussis Melissargos for her help in managing the AWS account for our experiments. We also thank the anonymous reviewers and the other members of the Data Intensive Applications & Systems Lab at EPFL, for their constructive feedback that greatly improves the quality of this paper.

REFERENCES

[1] 2022. *Alibaba Cloud E-MapReduce.* https://www.alibabacloud.com/product/emapreduce
[2] 2022. *Amazon Athena Engine Version 3.* https://docs.aws.amazon.com/athena/latest/ug/engine-versions-reference-0003.html
[3] 2022. *Amazon CloudWatch.* https://aws.amazon.com/cloudwatch/
[4] 2022. *Amazon EC2 On-demand Instances.* https://aws.amazon.com/ec2/spot/
[5] 2022. *Amazon EC2 Spot Instances.* https://aws.amazon.com/ec2/pricing/on-demand/
[6] 2022. *Amazon Redshift Advisor recommendations.* https://docs.aws.amazon.com/redshift/latest/dg/advisor-recommendations.html
[7] 2022. *Amazon Redshift Serverless.* https://aws.amazon.com/redshift/redshift-serverless/
[8] 2022. *Amazon S3.* https://aws.amazon.com/s3/
[9] 2022. *Apache Hudi.* https://hudi.apache.org/
[10] 2022. *Apache Iceberg.* https://iceberg.apache.org/
[11] 2022. *Auto Scaling Groups.* https://docs.aws.amazon.com/autoscaling/ec2/userguide/auto-scaling-groups.html
[12] 2022. *AWS - Spot Instance advisor.* https://aws.amazon.com/ec2/spot/instance-advisor/
[13] 2022. *AWS EMR.* https://aws.amazon.com/emr/
[14] 2022. *AWS Glue.* https://aws.amazon.com/glue/
[15] 2022. *AWS Lambda.* https://aws.amazon.com/lambda/
[16] 2022. *Azure - Use Azure Spot Virtual Machines - Pricing and eviction history.* https://learn.microsoft.com/en-us/azure/virtual-machines/spot-vms#pricing-and-eviction-history
[17] 2022. *Azure Analysis Services.* https://azure.microsoft.com/en-us/services/analysis-services/#overview
[18] 2022. *Azure Functions Hosting Options - Scale.* https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale#scale
[19] 2022. *BigQuery under the Hood.* https://cloud.google.com/blog/products/bigquery/bigquery-under-the-hood
[20] 2022. *Configuring Lambda Function Options.* https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html
[21] 2022. *EC2 Instance Rebalance Recommendations.* https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/rebalance-recommendations.html
[22] 2022. *Google BigQuery.* https://cloud.google.com/bigquery
[23] 2022. *Google Cloud Storage - Quotas and Limits - Bandwidth.* https://cloud.google.com/storage/quotas#bandwidth
[24] 2022. *Google Cloud Storage - Request rate and access distribution guidelines.* https://cloud.google.com/storage/docs/request-rate
[25] 2022. *Presto Docs - Join Reordering Strategy.* https://prestodb.io/docs/current/admin/properties.html#optimizer-join-reordering-strategy
[26] 2022. *Presto. https://prestodb.io/.* https://prestodb.io/
[27] 2022. *Redshift Serverless Considerations.* https://docs.aws.amazon.com/redshift/latest/mgmt/serverless-considerations.html
[28] 2022. *Resource Quota of Google Cloud Functions 2nd Gen.* https://cloud.google.com/functions/docs/concepts/version-comparison#new-in-2nd-gen
[29] 2022. *S3 Select and Glacier Select – Retrieving Subsets of Objects.* https://aws.amazon.com/blogs/aws/s3-glacier-select/
[30] 2022. *Scalability and performance targets for Blob storage.* https://learn.microsoft.com/en-us/azure/storage/blobs/scalability-targets
[31] 2022. *Spark Operator Pushdown.* https://www.databricks.com/dataaisummit/session/spark-data-source-v2-performance-improvement-aggregate-push-down
[32] 2022. *Spark SQL.* http://spark.apache.org/sql/
[33] 2022. *Spot Instance Interruption Notices.* https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-instance-termination-notices.html
[34] 2022. *Step and simple scaling policies for Amazon EC2 Auto Scaling.* https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scaling-simple-step.html
[35] 2022. *Top 10 Performance Tuning Tips for Amazon Athena.* https://aws.amazon.com/blogs/big-data/top-10-performance-tuning-tips-for-amazon-athena/
[36] 2022. *Trino.* https://trino.io/
[37] 2022. *Trino - Task Properties.* https://trino.io/docs/current/admin/properties-task.html
[38] 2022. *Trino Graceful-shutdown.* https://trino.io/docs/current/admin/graceful-shutdown.html
[39] 2022. *Trino Operator Pushdown.* https://trino.io/docs/current/optimizer/pushdown.html
[40] 2023. *Billing for Amazon Redshift Serverless.* https://docs.aws.amazon.com/redshift/latest/mgmt/serverless-billing.html

[41] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. 2009. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB* 2, 1 (2009).
[42] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*.
[43] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-Invented. In *SIGMOD*.
[44] Kamil Bajda-Pawlikowski, Daniel J. Abadi, Avi Silberschatz, and Erik Paulson. 2011. Efficient Processing of Data Warehousing Queries in a Split Execution Environment. In *SIGMOD*.
[45] Haoqiong Bian and Anastasia Ailamaki. 2022. Pixels: An Efficient Column Store for Cloud Data Lakes. In *ICDE*.
[46] Haoqiong Bian, Ying Yan, Wenbo Tao, Liang Jeff Chen, Yueguo Chen, Xiaoyong Du, and Thomas Moscibroda. 2017. Wide Table Layout Optimization Based on Column Ordering and Duplication. In *SIGMOD*.
[47] Nicolas Bruno, Johnny Debrodt, Chujun Song, and Wei Zheng. 2022. Computation Reuse via Fusion in Amazon Athena. In *ICDE*.
[48] Lei Chang, Zhanwei Wang, Tao Ma, Lirong Jian, Lili Ma, Alon Goldshuv, Luke Lonergan, Jeffrey Cohen, Caleb Welton, Gavin Sherry, and Milind Bhandarkar. 2014. HAWQ: A Massively Parallel Processing SQL Engine in Hadoop. In *SIGMOD*.
[49] Yueguo Chen, Xiongpai Qin, Haoqiong Bian, Jun Chen, Zhaoan Dong, Xiaoyong Du, Yanjie Gao, Dehai Liu, Jiaheng Lu, and Huijie Zhang. 2014. A Study of SQL-on-Hadoop Systems. In *BPOE@ASPLOS/VLDB*.
[50] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *SIGMOD*.
[51] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*.
[52] Alex Fuerst, Stanko Novakovic, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. 2022. Memory-Harvesting VMs in Cloud Platforms. In *ASPLOS*.
[53] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *SIGMOD*.
[54] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *CIDR*.
[55] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. 2022. Astrea: Auto-Serverless Analytics Towards Cost-Efficiency and QoS-Awareness. *TPDS* 33, 12 (2022), 3833–3849.
[56] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *SoCC (SoCC '17)*.
[57] Y. Kim and J. Lin. 2018. Serverless Data Analytics with Flint. In *CLOUD*.
[58] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*.
[59] Dimitrios Koutsoukos, Ingo Müller, Renato Marroquín, Ana Klimovic, and Gustavo Alonso. 2021. Modularis: Modular Relational Analytics over Heterogeneous Distributed Platforms. *PVLDB* 14, 13 (2021).
[60] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. 2021. Greenplum: A Hybrid Database for Transactional and Analytical Workloads. In *SIGMOD*.
[61] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *ATC*.
[62] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *PVLDB* 3, 1 (2010).
[63] Armbrust Michael, Das Tathagata, Sun Liwen, Yavuz Burak, Zhu Shixiong, Murthy Mukul, Torres Joseph, Hovell Herman, van, Ionescu Adrian, Luszczak Alicja, Switakowski Michal, Michalm Szafranski, Li Xiao, Ueshin Takuya, Mokhtar Mostafa, Boncz Peter, Ghodsi Ali, Paranjpye Sameer, Senster Pieter, Xin Reynold, and Zaharia Matei. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *PVLDB* 13, 12 (2020).
[64] Wawrzoniak Michal, Müller Ingo, Bruno Rodrigo, and Alonso Gustavo. 2021. Boxer: Data Analytics on Network-enabled Serverless Platforms. In *CIDR*.
[65] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *SIGMOD*.

[66] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *SIGMOD*.
[67] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *NSDI*.
[68] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. 2015. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. In *SIGMOD*.
[69] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. 2021. What Serverless Computing is and Should Become: The next Phase of Cloud Computing. *CACM* 64, 5 (apr 2021), 76–84.
[70] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *ICDE*.
[71] Panagiotis Sioulas and Anastasia Ailamaki. 2021. Scalable Multi-Query Execution Using Reinforcement Learning. In *SIGMOD*.
[72] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. 2010. MapReduce and Parallel DBMSs: Friends or Foes? *CACM* 53, 1 (2010), 64–71.
[73] Xiangyao Yu, Matt Youill, Matthew E. Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2020. PushdownDB: Accelerating a DBMS Using S3 Computation. In *ICDE*.
[74] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets.. In *HotCloud*.
[75] Feng Zhang, Zaifeng Pan, Yanliang Zhou, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2021. G-TADOC: Enabling efficient GPU-based text analytics without decompression. In *ICDE*.
[76] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Wenguang Chen. 2018. Efficient document analytics on compressed data: Method, challenges, algorithms, insights. *PVLDB* 11, 11 (2018), 1522–1535.
[77] Qizhen Zhang, Phil Bernstein, Daniel S. Berger, Badrish Chandramouli, Boon Thao Loo, and Vincent Liu. 2022. CompuCache: Remote Computable Caching using Spot VMs. In *CIDR*.
[78] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, and Badrish Chandramouli. 2021. Redy: Remote Dynamic Memory Cache. *PVLDB* 15, 4 (2021).

Received October 2022; revised January 2023; accepted February 2023