Adaptive data transformations for QaaS

Dimitrios Koutsoukos*
ETH Zurich
Switzerland
koutsoukos.dimitr@gmail.com

Renato Marroquín
Oracle Inc.
Switzerland
renato.marroquin@oracle.com

Ingo Müller
Google
Switzerland
ingomueller@google.com

Ana Klimovic
ETH Zurich
Switzerland
aklimovic@ethz.ch

ABSTRACT

In today's data-driven landscape, organizations have large amounts of semi-structured data that they want to get quick insights from. Query-as-a-Service (QaaS) systems are ideal for this use case, as the user can query the data *in situ* without spinning up or maintaining any dedicated infrastructure. However, these systems lack traditional database structures (e.g. indexes) making their cost and latency suboptimal in many cases. As a remedy, we show how QaaS can utilize simple data transformations (transcoding, chunking, sorting) using serverless functions (FaaS) to automatically modify input data based on data characteristics and the incoming queries to improve latency and cost. We envision that a cloud service can act as a middleware between the user and QaaS to adopt such transformations, where customers can opt in and specify their latency/cost budgets. Finally, we evaluate the various trade-offs of these transformations.

1 INTRODUCTION

Data management and analysis has changed rapidly over the last few years, influenced by both the increasing amount of data produced every day and the influence of Machine and Deep Learning (ML/DL) on data processing. This evolution is further pushed by the prevalent use of semi-structured data (e.g. CSV, JSON, Parquet), which are very popular both in the scientific domain and for ML datasets [6, 7, 21, 26]. Especially textual formats (e.g. CSV, JSON) offer simplicity and align well with the flexible nature of ML/DL algorithms but on the other hand lack complex schemas and they are optimized to be human-readable. Organizations, influenced by the evolving landscape, find themselves still storing an increasing portion of their data in textual format, which they need to analyze, get insights from, and use to their benefit. To do so, they can use cloud data warehouses or analytics platforms (e.g. Databricks, Snowflake)—or QaaS systems (e.g. Amazon Athena, Google Big Query). Data warehouses are a very good fit for data that need to be continuously processed and maintained because they optimize data layouts and materialize views of transformations to optimize performance. However, there are many use cases that produce data that will only be used and processed a handful of times or even just once. Take, for example, a particle physicist producing insights on a single experiment for High-Energy Physics [22]. For these sparse, infrequent workloads, maintaining an always-on infrastructure is not efficient and QaaS are a great solution. By declaring semi-structured data as external tables, users can query them and extract the necessary information while only paying for the amount of data they scanned and without the need to worry about scaling the infrastructure or optimizing the input data layouts.

*Currently at Apple

	CSV		Parquet	
	Runtime[s]	Cost[$]	Runtime[s]	Cost[$]
BigQuery	227	9.05	71.5	2.11
Athena	273.3	9.09	132.3	0.76

Table 1: Runtime and cost of running TPC-H SF-100 on Parquet and CSV for Amazon Athena and Google BigQuery

Between the two extremes of data warehousing and QaaS, there are many opportunities for improvement in between. For example, converting the input data to a columnar format (e.g. Apache Parquet) can have a huge impact on the execution time and cost of QaaS. Consider the cost and runtime for running TPC-H SF-100 on two popular QaaS in Table 1. We observe that the runtime is reduced more than 3x and the cost by an order of magnitude just with a simple transformation. Nevertheless, for sporadic execution, converting the whole dataset to Parquet might be ineffective in terms of cost and/or latency. The ultimate decision depends on the size of the dataset and the nature and arrival pattern of the queries.

Hence, in this paper we present our vision for adaptive data transformations for QaaS, which adjust based on the characteristics of incoming queries over time and a cost/latency budget provided by the client. We show how simple transformations (e.g. transcoding, sorting, chunking) can have a large impact on the cost and/or execution time of QaaS, in contrast to a non-future proof one-time transformation. To convert the data, we use FaaS because serverless functions are a good fit for occasional, bursty workloads due to their high elasticity, fine-grain resource billing, and the ability to scale to zero at low load [29]. In this way, QaaS can invoke FaaS only when additional budget is given by the customer and without introducing query rewriting. To understand the various trade-offs with adaptive transformations, we experiment with Amazon Athena and AWS

Lambda . Based on our observations, we provide actionable advice on where and which transformation pays off and how users can reduce their cost and latency for their sporadic workloads. Our insights are transferable between different cloud providers and also pave the way to practical storage optimization in the cloud, an area that is becoming increasingly important through data lakehouses [14, 33, 34]. Finally, we sketch future directions on how adaptive data transformations can be integrated in the cloud as part of QaaS as a cost/runtime advisor that can help clients to perform incremental transformations based on their own needs and budgets.

2 BACKGROUND

In this section we go over the necessary background needed for our paper. We start covering Apache Parquet, a popular columnar storage format and briefly discuss QaaS and FaaS, the technologies we built our transformations around.

2.1 Apache Parquet

Apachet Parquet [3] is an open-source columnar storage file format, designed to improve storage and access efficiency of tabular data compared to row-based files like CSV. Internally, Parquet files are divided into row groups. Each row group contains part of the data and within a row group, data are organized into column chunks. Each column chunk is divided into pages. Finally, each file has footer metadata about row groups and columns chunks (e.g. data types, total row group size, encodings, min-max indexes). The columnar storage, together with the footer metadata allow for compression techniques such as dictionary or run-length encoding. Additionally, using metadata, query engines can apply various optimizations (e.g. predicate pushdown, bloom filters, column statistics) to skip reading irrelevant row groups based on query predicates. All these features, together with the fact that Parquet is language-agnostic, have made this format the backbone of lakehouse formats, such as Deltalake [8] and Apache Iceberg [2].

2.2 Query-as-a-Service (QaaS)

Query-as-a-Service (QaaS) systems are cloud computing systems that allow users to execute SQL (or SQL-like) queries directly on data stored in the object storage. They offer significant advantages compared to traditional databases, as they can query datasets on demand through external tables without the need for upfront loading. Additionally, there is no need to manage the underlying infrastructure. Popular examples of such systems are Amazon Athena [1], and Google BigQuery [9]. QaaS support a variety of data formats (e.g CSV, JSON, Avro, Parquet). They only charge users based on the amount of data scanned by individual queries. However, when working on external data, they are lacking other traditional optimizations (e.g. join reordering) because most of these optimizations are based on statistics calculated upon table ingestion. This can lead to an increase in cost or execution time since, for example, they might read tables more than once or pick a non-optimal path to execute a query.

2.3 Serverless functions

Serverless functions or Function-as-a-Service (FaaS) are a cloud execution model where cloud providers dynamically manage the allocation and provisioning of servers. They are called serverless as managing instrastructure is abstracted away from developers, who only need to focus on writing the application code. The functions are executed in response to events (e.g. HTTP requests). Popular examples of serverless functions are AWS Lambda [4], Azure Functions [5], and Google Cloud Functions [10]. FaaS are a good fit for sparse, bursty workloads. First of all, users only pay for the compute time they consume when needed and they do not need to keep the infrastructure on the whole time. They are also highly scalable, reaching up to a few thousand function invocations per second, while keeping a simple and quick deployment. Additionally, they are especially useful for data-independent tasks that start up and shut down quickly and run on an infrequent basis [29].

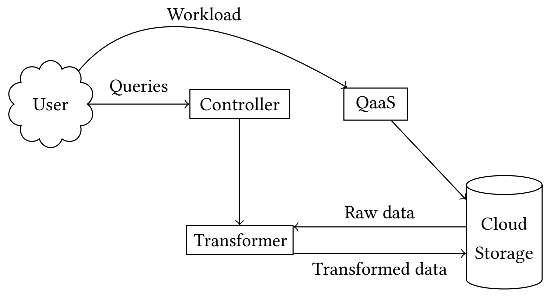

Figure 1: System architecture

3 DATA TRANSFORMATIONS IN PRACTICE

We first describe how we imagine that our adaptive data transformations will be integrated as a cloud service (Figure 1). The user submits a workload consisting of semi-structured, uncompressed data, queries, and the frequency of the workload to QaaS. Our system has visibility on the queries, table metadata (e.g. types), and the frequency at which the workload will execute. The service then analyzes the queries to check the size of the tables, their predicates, how often this table is queried, and the number of partitions each table has. The user optionally gives a monetary budget to drive more targeted optimizations. For our initial implementation, our module will only have three transformations (transcoding, chunking, and sorting) since these are the most influential and well-known in terms of latency/cost. The controller first checks if the budget is enough to convert a few or all of the largest tables of the dataset to a compressed format. It then checks for additional optimizations. For example, if the number of partitions on large tables is very low or there are many queries that have the same selection predicates, the controller checks if it can combine more than one optimizations. For a transformation to be valid, the execution cost should be lower than the budget that the user has given but the execution cost is directly proportional to the execution time since FaaS has a cost on a per-use basis. After the necessary analysis has been performed and the transformations are decided, the controller launches serverless functions through the transformer, which has access to the data through the cloud storage, to perform the required transformations. The transformed data are then available to the QaaS for running the initial workload. Subsequent executions of the workload follow again the same execution flow. For every set of transformations performed, we keep additional copies of the data in S3. We do not

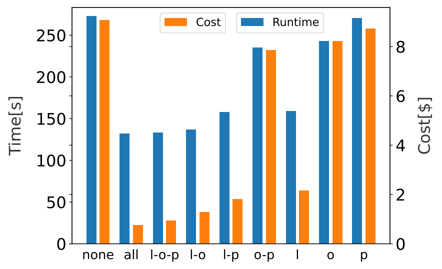

Figure 2: Query runtime (left axis) and query cost (right axis) of running TPC-H on Amazon Athena with different tables on Parquet

have the cost of keeping additional data as part of our budget because this depends on the amount of time the user keeps the data on S3. For sparse-running workloads, the cost pattern might be very irregular and there is a possibility to even delete the data right after the workload is completed.

4 EXPLORING THE TRADE-OFFS

In this section, we analyze the runtime and cost of performing the three conversions (transcoding, chucking, sorting). We use AWS Athena as an example QaaS and AWS Lambda to run the data transformations. In both cases we read/write the data from/to S3. To reduce the variability of serverless function latency, in particular, due to cold starts [24, 27, 35], we pre-warm the functions and take the median of 10 runs for each experiment.

4.1 Transcoding to columnar format

Conversion to a columnar format (or transcoding), such as Apache Parquet, is one of the most effective transformations for users to reduce cost and latency for QaaS because Parquet has a number of optimizations (Section 2.1) compared to CSV/JSON. Although transcoding is generally beneficial, for sparse-running workloads, it is not obvious which tables to convert to Parquet to maximize the potential benefit, because there are many factors to consider, such as table sizes, the frequency each table appears in the query set, and the projection and selection predicates of each query.

To understand the trade-offs, we run TPC-H SF-100 with 300 files per table on Amazon Athena. We transform combinations of tables in Parquet and show the execution time and cost in Figure 2. We transform either all tables or a combination of the three bigger tables of the workload (`lineitem`, `orders`, and `partsupp`), which constitute 97% of the total dataset size. Transforming all data to Parquet reduces the overall runtime in half and the total cost around 10x. Since the first three tables make up almost the entire dataset, transforming them brings the cost and the runtime very close to having all data on Parquet.

We now experiment with transcoding combinations of two tables from `lineitem`, `orders`, and `partsupp`. Transforming `lineitem` and `orders` is very similar to transforming all three tables, whereas converting `lineitem` and `partsupp` has both higher runtime and cost, respectively 20s and 0.5$. This increase is much larger proportionally than the 5GB difference between `orders` (17GB) and

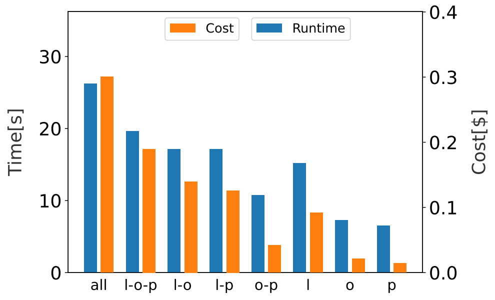

Figure 3: Transformation runtime (left axis) and transformation cost (right axis) of transcoding different tables on Amazon Lambda

`partsupp` (12GB). We make similar observations when we convert only `orders` and `partsupp` or individual tables in the right side of the figure. More specifically, if we run the benchmark with `orders` and `partsupp` on Parquet, the runtime reduces 12% and the cost 13%, whereas both these tables are around 25% of the overall workload size. The trend becomes even worse if we alter `orders` or `partsupp` individually. For `orders`, the runtime is 11% lower and the cost 10% lower. For `partsupp`, the runtime is identical to not having converted the table and the cost is 5% lower. Finally, when having `lineitem` in Parquet, the runtime reduces almost 40% and the cost more than 4x, besides the table being only 70% of the workload. Therefore, it is not always straightforward or obvious which tables we should transcode. That becomes even more complicated if we take into account the latency and cost of converting our tables from one format to another.

To understand how expensive and time-consuming transcoding of the tables is, we show the runtime and cost of converting the same table combinations as in the previous experiment to Parquet in Figure 3. Every table is split across 300 files of equal size, and we invoke one function per file for the conversion using a large degree of parallelism for the function invocation.

The experiment mirrors partially what we noticed previously. First, there is a sub-linear behavior depending on the table size, both for latency and cost. That comes from the fixed "setup" cost (e.g. invocation of functions, time until a function starts etc.), which is independent of the input size. Additionally, converting `lineitem` with other table combinations takes roughly the same time as converting `lineitem` alone, although the cost increases as we convert more tables. That shows that transforming a large table dominates the running time when the overall number of function invocations is not very large (up to a 1000). On the other hand, if we convert all tables to Parquet, the overall runtime and cost increase significantly, in a disproportionate way compared to the additional data transformed. In particular, the rest of the tables are only 5GB but the overall increase both in runtime and cost is more than 30%. That shows the influence of the setup cost in the overall runtime as well as the impracticality of always transforming everything into a more suitable format as it may not be cost effective, especially for large data amounts.

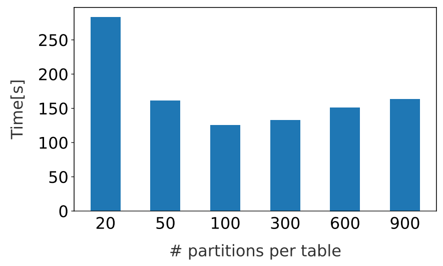

Figure 4: Runtime of running TPC-H on Amazon Athena with different number of parquet partitions per table

4.2 Splitting by size (chunking)

Changing the number of files a table consists of is another effective transformation that influences the latency but not the cost of a workload. However, it is not obvious what partition file size to choose because cloud providers do not provide explicit instructions. For example, Athena's documentation says that "datasets that consist of many small files result in poor overall query performance" [11] but does not give indications on what the file size should be.

The performance drop that Athena mentions happens because, first, QaaS systems need to keep a list of all the partition locations and, therefor, with an increasing number of partitions, the listing time increases and, second, because they usually schedule the number of tasks based on the number of partitions. If there are too many small files, each task has negligible work and most of the time is spent communicating across different tasks in exchange phases. However, a few large files can also be detrimental because not enough tasks are scheduled and the overall CPU power is not sufficient.

To understand the file size effect, we alter the number of table CSV partitions from 20 to 900. In a plot not shown, we see that the number of CSV partitions has no influence on the execution time. Because the actual implementation of Athena is not open source, we can only speculate about why. When observing the query plans, we notice that the number of tasks stays the same independently of the number of partitions and is also quite high. Therefore, a plausible explanation is that Athena sets a high degree of parallelism directly when reading CSV and splits the data across the workers with a mechanism that is independent of the number of files.

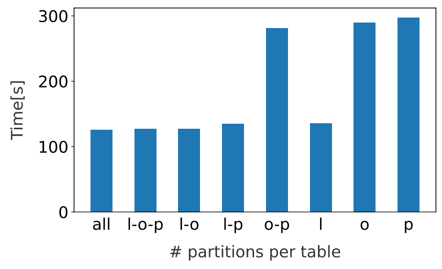

Figure 5: Runtime of running TPC-H on Amazon Athena when having different number of parquet partitions in a subset of tables

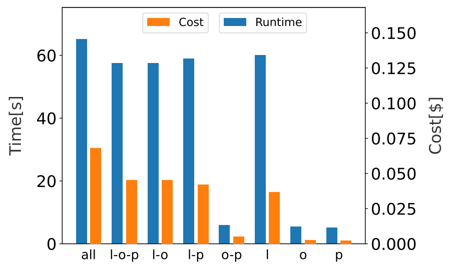

Figure 6: Transformation runtime (left axis) and transformation cost (right axis) of splitting tables into smaller files on Amazon Lambda

In contrast, the number of Parquet files has a big influence on the query runtime. We observe this in Figure 4, where we run TPC-H SF 100 with an increasing number of Parquet files per table. The plot has a U-shape: If we have a very small or very large number of files, the performance is not optimal. Especially, for small numbers of files (e.g., 20 files per table), the overall runtime is even larger than running the same workload on CSV. The optimal performance occurs with 100 partitions per table. After that, increasing the number of partitions increases gradually the execution time. Based on the query plans, we observe that Athena sublinearly increases the number of tasks as the number of partitions increases. For a small number of partitions, based on the CPU time, there is not enough parallelism to process data as fast as possible, whereas for large number of partitions, the CPUs are underutilized. The parallelism does not only have to do with the number of partitions but also the size of each partition. From our experiments, we observed that Athena can handle I/O better with Parquet file size less than 300MB.

We repeat a similar experiment to Section 4.1. We run TPC-H using 20 partitions per table while using 100 chunks for the tables denoted in the x-axis (Figure 5). We pick either all tables or combinations of the three largest tables. We notice that chunking `orders`, `partsupp`, or both at the same time gives a marginal or no performance advantage, especially when considering their size in the overall benchmark (around 25%). On the other hand, transforming only `lineitem` is 8% slower than splitting all the tables into 100 partitions and dividing more tables other than that gives marginal increases, if any. The rationale for that is that the number of tasks that can be scheduled for a particular table increases for tables like `lineitem` but not for small tables like `partsupp`.

To gain insights into how time and cost-effective chunking is, we use serverless workers to divide an individual Parquet file into smaller ones for various TPC-H tables. We use one function per input file, load the file gradually from S3, and transform it in batches to be as memory efficient as possible (Figure 6). We see that the overall cost is much lower than if we convert files from CSV to Parquet because we use a smaller number of functions compared to transcoding and transfer less data from S3 to Lambda due to the compression that Parquet has.

Table 2: TPC-H SF-100 queries with sorted and unsorted selection predicate

Query	Selectivity	No sorting		l_shipdate sorting	
		Runtime[s]	Cost[$]	Runtime[s]	Cost[$]
Query 1	1%	23.21	0.026	21.75	0.021
Query 3	46%	12.33	0.033	11.58	0.040
Query 6	85%	9.53	0.023	5.31	0.013
Query 12	51%	14.44	0.021	11.55	0.020
Query 14	99%	11.18	0.037	7.07	0.020
Query 15	96%	12.52	0.070	8.21	0.046
Query 20	84 %	13.37	0.039	7.95	0.022
TPC-H total (common data)	-	283.19	0.718	272.6	0.737
TPC-H total (separate data)	-	283.19	0.718	260.36	0.650

A second key observation is that any transformation involving `lineitem` has roughly the same execution time, except for partitioning all the tables. The difference between splitting `lineitem` alone or together with other tables comes from the natural variation of cloud environments. Using this, we can partition a couple of additional smaller tables together with a very large table without increasing the overall execution time and with minimal additional cost. Finally, transforming `orders` and `partsupp` separately or together has exactly the same runtime and a negligible difference in cost. This occurs because, for transformation of small tables and for a small number of files, the setup cost has minor duration compared to the actual work the function does.

4.3 Sorting a selection column (sorting)

As a final transformation, we explore sorting Parquet files based on a column: each FaaS invocation reads the Parquet file it gets assigned, sorts the input by the sorting column, and writes the result into a new Parquet file. This improves compression and makes min/max indexing more effective. Sorting does not offer any performance advantage on semi-structured file formats such as CSV.

We first experiment with isolating `l_shipdate`, the column in `lineitem` that has the most selection predicates. This will give an indication of the largest reduction that we can get since (1) `lineitem` is the largest table and (2) one third of the benchmark discards large amounts of data based on `l_shipdate`. We run these queries on tables split over 20 files with and without the sorted column and present the results in Table 2. The queries have a ranging degree of selectivity from 1% to 99%. However, the decrease in runtime and cost is also affected by other factors such as the query plan and the other tables the query scans from. The biggest decrease both for latency and cost is for Queries 6 and 20. Both of these have a 43% cost and latency improvement due to less data scanned. In contrast, for Query 3, we notice an interesting trend: although the latency is reduced, the cost increases, meaning that Athena scans more data. This happens because by sorting on a column, we are disrupting the order of other columns. That may lead to unwanted results, e.g. for tables that are sorted on the primary key, which is the case for TPC-H by default. As a consequence, we cannot predict the effect of altering the primary key order on the runtime and cost. To verify this assumption, we run the whole benchmark with the data sorted on `l_shipdate` (Table 2). The total runtime is reduced only by 4% and the cost is increased by 3%. To avoid this and to make sure that sorting will have a positive effect, we keep two separate copies of the data: (1) the original data for queries that do not include a selection predicate on the columns that is sorted and (2) the transformed data for the rest of the queries. With this approach, we maximize the expected return. We verify this by running TPC-H with two data copies (Table 2). The execution time is reduced by 8% and the total cost is decreased by 10%.

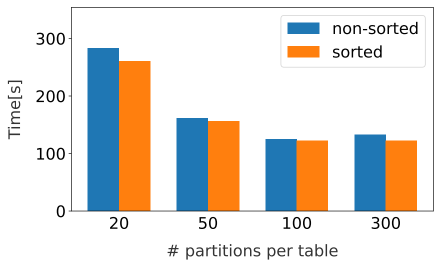

Figure 7: Runtime of running TPC-H on unsorted data vs sorted on `l_shipdate` data for various partitions per table on Amazon Athena

Additionally, sorting also interacts with chunking. When we have small files sorted by a previously unsorted column, the value range of each fixed-size row group after sorting is larger. That effectively means larger distance between the min and max values per row group and, hence, less effective min/max filters.

We verify this hypothesis by increasing the number of files and running TPC-H SF 100 with sorted and unsorted data on `l_shipdate` (Figures 7 and 8). If running with sorted data, we have two separate data copies. We observe similar trends for both plots: As we increase the number of partitions, sorting provides a smaller latency/cost reduction. Especially for 100 partitions, the time difference is less than 2%. We conclude that that sorting and chunking are negating each other and that we have to pick only one of them if the input file sizes are large. Finally, we discuss the cost and runtime of sorting the `l_shipdate` column of `lineitem` using AWS Lambda. While we used serverless functions with 2GB of main memory for the previous transformations, for sorting, this is not sufficient. As every file is around 1.1GB and sorting is memory-intensive, we need to use the lambda functions with the highest memory (10GB). Thus, sorting might not be feasible for every workload, table, and partition size. The total execution time is 63s and the cost is 0.25$. From a first look, sorting does not provide any advantage, given

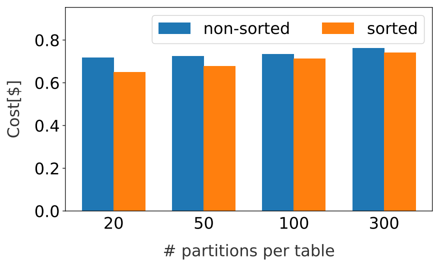

Figure 8: Cost of running TPC-H on unsorted data vs sorted on `l_shipdate` data for various partitions per table on Amazon Athena

the runtime and cost of transforming the files. While this is the case for TPC-H, the actual advantage might be different for other workloads or queries and depending on how many times we run a workload.

5 RELATED WORK

Our vision is inspired by database cracking [23] but has important differences. First of all, our approach examines data copying and not only reorganization through different formats and other transformations and it is not necessarily done on a per-query basis but can be also done ahead of time. Additionally, it is tailored for QaaS that use external data and not for in-memory databases that ingest data and create tables. Our adaptive data transformations are also similar in spirit with many in-situ and caching efforts [12, 15, 30, 31]. However, all these approaches are based on on-premise, disk-based systems, which consequently do not consider data copies but substituting data, or use different data layouts in caches. Our approach is cloud-based and does not use caching but different data copies. Additionally, data formats like the one proposed by Snowflake [19] or in self-organizing data containers [28] are orthogonal to our work since we do not explore new data structures but instead reorganize data on a per-need basis to take advantage of various QaaS characteristics for querying external data. Systems that use FaaS to do query processing [17, 29, 32] are independent from our approach as we use FaaS not to perform query processing but to lower the runtime and cost for QaaS. Finally, in parallel to our strategy, Boncz et al [18] propose adaptive indexing for query optimization in the cloud given a particular budget and Bang et al [16] suggest to avoid using ML techniques to optimize cloud systems but instead to use systems as oracles and perform convex optimization. We do not necessarily build indexes but we instead perform data reorganization tailored for QaaS infrastructures and could also build on linear/convex optimization to investigate which transformations are feasible and worth given a particular cost/latency budget.

6 DISCUSSION AND FUTURE WORK

In this paper, we introduce the notion of adaptive data transformations for QaaS. We show how the end user can leverage simple transformations to reduce the cost and latency of infrequent workloads that run on QaaS, leveraging FaaS infrastructure. This opens up a few exciting avenues for future work.

Workload-instance advisor. Based on our study, we can develop an advisor to minimize workload latency given a particular cost budget. The advisor will be part of the controller in the system architecture of Figure 1. In addition to queries, it can take as input queries from the workload and metadata about the data set (e.g., data types) and use optimization algorithms or ML techniques to propose transformations. This advisor would not need access to the actual data to calculate the transformations and is easily extensible to other cloud deployments/transformations/data formats with additional recalibration. However, there are also challenges in developing this type of solution due to the variability in the cloud and because systems like Athena or BigQuery are not open-source. In case we also have access to the internals of the QaaS, we could tailor the budget to include the total budget of running the queries and transformations. That would need knowledge on how QaaS perform I/O operations since these transformations are effectively reducing I/O time and they are not influencing query plans. Such a scheme would also allow for more optimizations in case the users allow transformations as long as the total budget decreases.

Transformation optimizations and data maintenance. Compared to traditional database deployments, the cloud has "unlimited" resources in terms of compute and storage. In our work, we did not explore optimizations [25, 36] for the transformations done in FaaS. We also did not experiment with more advanced optimizations like joins or QD-trees [20, 37], which can be considered as more mature versions of chunking and sorting. Additionally, we did not investigate partial transformations that only touch a part of a table. A partial transformation introduces additional complexity as the set of queries has to potentially be executed against many different versions of the data. It is also interesting only for sorting. Transcoding has to read the whole table, because there is not a way of only partially reading a textual format on the cloud. Furthermore, chunking is more effective for whole tables. In terms of storage, databases are usually limited by the amount of system memory, which is not the case for S3. Theoretically, we could keep a large number of copies tailored to the minimal runtime and cost of specific queries within a workload. However, this would become infeasible in terms of cost and version management. We imagine that the five-minute rule [13] would be also applicable here, where periodically, we will need to clean up our storage by deleting unnecessary dataset versions. We could also utilize Deltalake and/or Iceberg to introduce a view maintenance of the dataset.

7 ACKNOWLEDGEMENTS

We thank Gustavo Alonso and Theo Rekatsinas for discussions and suggestions on an earlier draft of this work.

REFERENCES

[1] Accessed 2024-08-01. Amazon Athena. https://aws.amazon.com/athena/.
[2] Accessed 2024-08-01. Apache Iceberg. https://iceberg.apache.org/.
[3] Accessed 2024-08-01. Apache Parquet. https://parquet.apache.org/.
[4] Accessed 2024-08-01. AWS Lambda. https://aws.amazon.com/lambda/.
[5] Accessed 2024-08-01. Azure Functions. https://learn.microsoft.com/en-us/azure/azure-functions.
[6] Accessed 2024-08-01. CSV explained. https://ai-jobs.net/insights/csv-explained/.
[7] Accessed 2024-08-01. CSV Files: Use cases, Benefits, and Limitations. https://www.oneschema.co/blog/csv-files.
[8] Accessed 2024-08-01. Deltalake. https://delta.io/.
[9] Accessed 2024-08-01. Google BigQuery. https://cloud.google.com/bigquery/.
[10] Accessed 2024-08-01. Google Cloud Functions. https://cloud.google.com/functions/.
[11] Accessed 2024-08-01. Performance tuning in Athena. https://docs.aws.amazon.com/athena/latest/ug/performance-tuning.html.
[12] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. 2012. NoDB in action: adaptive query processing on raw data. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1942–1945.
[13] Raja Appuswamy, Renata Borovica, Goetz Graefe, and Anastasia Ailamaki. 2017. The five minute rule thirty years later and its impact on the storage hierarchy. In *Proceedings of the 7th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures.*
[14] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. 2021. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of CIDR*, Vol. 8.
[15] Tahir Azim, Manos Karpathiotakis, and Anastasia Ailamaki. 2017. Recache: Reactive caching for fast analytics over heterogeneous data. *Proceedings of the VLDB Endowment* 11, ARTICLE (2017), 324–337.
[16] Tiemo Bang, Conor Power, Siavash Ameli, Natacha Crooks, and Joseph M. Hellerstein. 2023. Optimizing the cloud? Don't train models. Build oracles! arXiv:2308.06815 [cs.DB]

[17] 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 (jun 2023), 27 pages. https://doi.org/10.1145/3589306

[18] Peter Boncz, Yannis Chronis, Jan Finis, Stefan Halfpap, Viktor Leis, Thomas Neumann, Anisoara Nica, Caetano Sauer, Knut Stolze, and Marcin Zukowski. 2023. SPA: Economical and Workload-Driven Indexing for Data Analytics in the Cloud. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 3740–3746.

[19] 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 *Proceedings of the 2016 International Conference on Management of Data*. 215–226.

[20] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. Instance-optimized data layouts for cloud analytics workloads. In *Proceedings of the 2021 International Conference on Management of Data*. 418–431.

[21] Chang Ge, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. 2019. Speculative distributed CSV data parsing for big data analytics. In *Proceedings of the 2019 International Conference on Management of Data*. 883–899.

[22] Dan Graur, Ingo Müller, Mason Proffitt, Ghislain Fourny, Gordon T Watts, and Gustavo Alonso. 2023. Evaluating query languages and systems for high-energy physics data. In *Journal of Physics: Conference Series*, Vol. 2438. IOP Publishing, 012034.

[23] Stratos Idreos, Martin L Kersten, Stefan Manegold, et al. 2007. Database Cracking.. In *CIDR*, Vol. 7. 68–78.

[24] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[25] Simon Kassing, Ingo Müller, and Gustavo Alonso. 2022. Resource allocation in serverless query processing. *arXiv preprint arXiv:2208.09519* (2022).

[26] Jan Kossmann, Daniel Lindner, Felix Naumann, and Thorsten Papenbrock. 2022. Workload-driven, lazy discovery of data dependencies for query optimization. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[27] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. 2022. Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through {Inter-Function} Container Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 69–84.

[28] Samuel Madden, Jialin Ding, Tim Kraska, Sivaprasad Sudhir, David Cohen, Timothy Mattson, and Nesime Tatbul. 2022. Self-organizing data containers. *Memory* 1 (2022), 2.

[29] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 115–130.

[30] Matthaios Olma, Manos Karpathiotakis, Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. 2017. Slalom: Coasting through raw data via adaptive partitioning and indexing. *Proceedings of the VLDB Endowment* 10, 10 (2017), 1106–1117.

[31] Matthaios Olma, Manos Karpathiotakis, Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. 2020. Adaptive partitioning and indexing for in situ query processing. *The VLDB Journal* 29 (2020), 569–591.

[32] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A scalable query engine on cloud functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 131–141.

[33] Sivaprasad Sudhir, Wenbo Tao, Nikolay Laptev, Cyrille Habis, Michael Cafarella, and Samuel Madden. 2023. Pando: Enhanced Data Skipping with Logical Data Partitioning. *Proc. VLDB Endow.* 16, 9 (may 2023), 2316–2329. https://doi.org/10.14778/3598581.3598601

[34] Suketu Vakharia, Peng Li, Weiran Liu, and Sundaram Narayanan. 2023. Shared Foundations: Modernizing Meta's Data Lakehouse. In *The Conference on Innovative Data Systems Research, CIDR*.

[35] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 133–146.

[36] Mike Wawrzoniak, Ingo Müller, Rodrigo Fraga Barcelos Paulus Bruno, and Gustavo Alonso. 2021. Boxer: Data analytics on network-enabled serverless platforms. In *11th Annual Conference on Innovative Data Systems Research (CIDR 2021)*.

[37] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning data layouts for big data analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 193–208.