End-to-End Declarative Data Analytics: Co-designing Engines, Interfaces, and Cloud Infrastructure

Pinghe Li*
Systems Group, ETH Zurich
Switzerland

Tom Kuchler*
Systems Group, ETH Zurich
Switzerland

Marko Kabić*
Systems Group, ETH Zurich
Switzerland

Tobias Stocker*
Systems Group, ETH Zurich
Switzerland

Gustavo Alonso
Systems Group, ETH Zurich
Switzerland

Ana Klimovic
Systems Group, ETH Zurich
Switzerland

ABSTRACT

The declarative nature of the relational model and database engines shields users from system implementation complexity, system evolution, data representation details, and enables optimizations across workloads and use cases. However, recent trends in data management introduce significant challenges to declarative interfaces. For instance, these days, the engine might not own the data (e.g., data lakes), might have to deal with different data representations (e.g., CSV, Arrow, Parquet, Iceberg), and needs to support operations beyond relational SQL (e.g., vector databases, machine learning). To complicate matters, in the cloud, many layers of infrastructure — virtual machines (VMs), container schedulers, general-purpose OSes — stand between the engine and the compute/storage/network fabric. Today's cloud infrastructure exposes only low-level, VM-centric knobs and treats analytics workloads as opaque binaries, leaving the engine with little visibility into data placement, hardware availability, and network congestion. We propose to extend the declarative approach to data management end-to-end. We start from a composite database engine and extend its declarative interfaces (SQL and query plans) down to the cloud execution layer. The database engine exposes each query as a dataflow graph, composed of functions (operators) that each declare their inputs, outputs, and available parallelism to the cloud platform. The cloud platform takes this graph as input and provides a declarative execution substrate responsible for resource scaling, data and operator placement, caching, and isolation mechanism implementation decisions. Our early prototype shows how rethinking the interface between the engine and the cloud platform enables elastic data-dependent parallel execution over data lakes, automatic caching, and opens new research directions for cloud analytics.

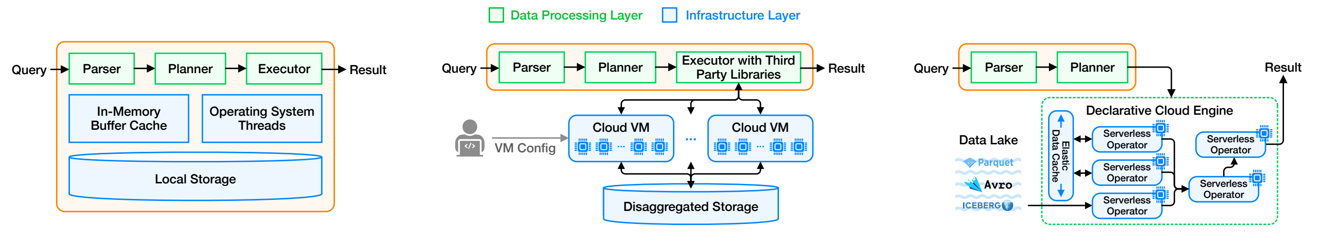
1 INTRODUCTION

Declarative data processing interfaces such as SQL enable users to express a high-level intent rather than detail its execution. In doing so, they enable the underlying execution systems to optimize performance and efficiency. However, the declarative nature of today's data analytics system stack ends for the most part at the query planning and optimization layer. Beneath the SQL engines lie many opaque infrastructure layers such as virtual machines and general-purpose operating systems, which lack declarative interfaces and are not co-designed with the query engine. This limits performance optimization opportunities and leads to inefficient utilization of the compute, memory, and network resources. The disconnect between the query engine and infrastructure layers is especially problematic in today's cloud-based deployments, where data engines ingest data from disaggregated object storage and run on top of VMs that are slow to boot and cumbersome to scale up or down. While query optimizers reason about, e.g., cardinality, filters, and join cardinality, they have limited visibility into the physical execution environment (what hardware is available, which nodes are close to the input data, or how network congestion might affect query plans). Meanwhile, the infrastructure layer treats analytics jobs as opaque operations, unaware of whether they're compute-bound, I/O-heavy, or latency-sensitive. The result is overprovisioning and underutilization of resources [8, 9, 35, 36, 53].

At the same time, the landscape of cloud data analytics is rapidly evolving [3, 34, 62]. Composite engines such as BOSS [40], Maximus [27], Azure Fabric [3], or Photon [11] as well as engine components such as Substrait [46], Calcite [10], or Velox [42] increasingly allow users to manage large data collections while blending SQL, user-defined functions (UDFs), and machine learning pipelines. Users no longer submit plain SQL queries; they write workflows that join tables, run ML inference, and apply complex transformations based on arbitrary user-defined logic. These workloads demand an infrastructure that is elastic, programmable, and co-optimized with the engine itself for efficiency. The infrastructure must also securely isolate tenants, as anyone with a credit card can submit arbitrary functions to cloud platforms. Doing this is difficult if the data processing logic is a black box to the platform and if the code is oblivious to the underlying infrastructure it runs on.

In this paper, we discuss the idea of extending the declarative nature of data analytics down to the infrastructure layer. Just as SQL enables logical-physical separation in databases, a declarative infrastructure should enable co-optimizing the execution plan with its resource allocation. Rather than provisioning static VMs, cloud data analytics systems should take a more holistic approach by planning query execution, resource allocation, hardware selection, and data movement together. This cannot be done at compile time; it must involve a runtime component that is aware of the workload's input data. We envision an end-to-end declarative analytics system in which users describe computation over data, while the system takes end-to-end responsibility for transforming that specification into an optimized, cost-efficient, and hardware-aware execution

*These authors contributed equally to this work



(a) Traditional, monolithic engine.

(b) Composite engine in the cloud today.

(c) Composite engine co-design with declarative cloud infrastructure (our proposal).

Figure 1: Evolution of data analytics engine architecture from (a) a monolithic engine that owns data on local storage and manages a buffer cache to (b) a composite engine running on cloud VMs that ingests data from disaggregated storage to (c) our proposal of co-designing the engine with declarative cloud infrastructure, enabling data-dependent parallel execution and ingestion from data lakes, elastic caching, and more.

plan for the cloud rather than for a monolithic engine. This requires rethinking the interface between data processing engines and cloud execution platforms. We propose going beyond VMs to fine-grained, serverless execution backends and beyond opaque query engine executables to richer APIs that expose a query's dataflow to the infrastructure layer. This new paradigm co-designs the declarative nature of data analytics and infrastructure management.

We explore the potential of end-to-end declarative analytics and the requirements for the two main system layers involved: the data processing and infrastructure layers. We draw inspiration by experimenting with a preliminary integration of two existing systems: a composite database engine for the data processing layer, Maximus [27], and a declarative cloud execution platform for the infrastructure layer, Dandelion [32]. The former provides the ability to run query plans over heterogeneous architectures (CPU and GPU at the time of writing) by encapsulating operators from different libraries and engines and orchestrating their execution in the form of a dataflow graph. The latter provides an elastic execution environment abstracting the underlying computing platform behind declarative interfaces. For instance, user functions only specify their inputs, outputs, and computation logic; they do not directly make I/O calls as I/O is executed and optimized by the underlying system according to the dataflow DAG specified in the user application. The system also automatically creates and tears down function execution environments as needed and scales compute resources based on the parallelism available in the dataflow program.

Combining a composite database engine with a declarative execution platform offers more than the sum of their parts. The data layer provides a query plan in the form of a DAG specifying the data being accessed, the operator order, and the preference for where to execute each part of the plan. The query plan can then be translated into a Dandelion DAG (a dataflow specification of the functions to execute). The tables or files accessed by each operator are extracted from the query statement or in the form of intermediate results between the operators of the plan. The operators are encapsulated into functions that are pre-declared to Dandelion. In turn, the infrastructure decides, e.g., how many copies of an operator to instantiate for the access methods reading data from storage and then aggregate the results for the next stages of the query plan. It can also monitor runtime conditions to place the execution in the optimal location (e.g., either in terms of availability, to minimize interference, or to take advantage of locality) and manage failures, restarts, migration, and resource allocation. Dandelion securely isolates operator execution from separate tenants with lightweight sandboxes that are fast to boot for high elasticity. While enforcing secure isolation, the platform still enables fast data passing between operators and can reuse intermediate results across queries.

The synergy between the two systems, Maximus and Dandelion, provides a platform for optimized query execution in data lakes, both from the query plan and the cloud infrastructure side. The resulting system opens up interesting opportunities, which we discuss: creating ephemeral caches for intermediate results and exchanging data; automatic elasticity at both individual query and workload levels; optimized resource allocation; gathering of statistics for informing an end-to-end optimizer; incorporating hardware heterogeneity in query plans, and more.

2 BACKGROUND

The cloud, while offering many advantages, also introduces significant complexity and inefficiencies for data analytics engines.

2.1 From Monolithic to Composite Engines

The cloud is a challenge for conventional relational engines because there is a mismatch between the control of compute, memory, and storage resources that a traditional monolithic engine assumes (Figure 1a) and the deep software and hardware stack in the cloud, which consists of heterogeneous virtual machines and large pools of disaggregated storage (Figure 1b). There are significant architectural differences in, e.g., how memory or compute is allocated, how storage is accessed, etc. As a result, conventional engines often run with special settings (e.g., Amazon Relational Database Service providing block storage and specialized configurations suitable to database servers). This is often less than optimal, and alternative designs have emerged in the form of cloud-native engines. These native engines range from completely new designs such as Snowflake [55] to adaptations of existing engines [54].

A recent further step in this evolution are data lakes, a data management architecture reflecting the heterogeneous and disaggregated nature of cloud data storage, forcing the engines to relinquish ownership of the data and operate without making assumptions about where it is located or its format [33, 62]. This is a challenge but also an opportunity as it leads to revisiting the notion of federated query processing where data resides in a variety of systems, formats, and repositories that can be combined to build composite engines.

Emerging platforms tackling such scenarios include Databrick's Photon [11], Microsoft's Azure Fabric [3], or Apache Gluten [45]. These systems are focused on the challenges of federating diverse underlying systems. What is still missing is a way to connect them to the possibilities the cloud offers in terms of elasticity and better resource provisioning. Our vision focuses on the latter.

2.2 VM Provisioning: the Traditional Interface to Cloud Infrastructure

Provisioning virtual machines (VMs) is the traditional interface to the cloud. Running a database engine on the cloud today involves provisioning fixed-size VMs to execute the database engine's code (Figure 1b). This includes explicitly allocating VM instances, attaching block storage, and configuring networking among VMs and external storage. Each VM type offers a fixed ratio of compute and memory resources. Scaling resources up or down typically requires migrating to smaller or larger VMs, respectively. Scaling resources in or out involves instantiating or tearing down VMs, respectively. Data must be serialized, encrypted, and compressed as it moves from remote storage to the VMs that execute queries, adding overhead that is often referred to as datacenter tax [29].

Today's low-level VM provisioning interface offers numerous tuning knobs (e.g., VM types and size, storage options, network bandwidth) but lacks high-level declarative abstractions that would let the system automatically optimize resource usage and performance for a given analytics workload. As a result, resource provisioning is often coarse-grained, manual, and disconnected from the user query's intent (Figure 1b). The long boot time of traditional virtual machines (which can range from seconds to hundreds of seconds [24, 58]) also makes it difficult for cloud analytics engines to respond to fine-grain variations in resource requirements within and across queries. It is common practice to provision the number and size of VMs for peak workload requirements. Such overprovisioning leads to high cost. These challenges have sparked efforts such as studying the behavior of conventional engines in the cloud [8, 64], creating special environments for them (e.g., Amazon's RDS), exploring elasticity to minimize overprovisioning [9], or adding additional operators such as compression and encryption [6].

2.3 Serverless and its Current Limitations

The low-level interface needed to provision virtual machines is a general pain point in the cloud, not only for databases [25, 64]. Hence, a lot of effort has gone into developing "serverless" cloud services, which expose a higher level of abstraction to the cloud, freeing users from the burden of managing virtual machines [44, 56]. Examples include Function as a Service (FaaS) platforms like AWS Lambda and analytic engines like Athena and BigQuery.

However, current serverless platforms are inadequate for data processing [26] due to many limitations (e.g., poor support for inter-function communication, limited execution time per function, etc.) that require rather complex mechanisms to work around them [41, 43, 60, 61]. FaaS platforms leverage MicroVMs like Firecracker [1] to reduce boot times down to hundreds of milliseconds, but even this startup time can be a high overhead for short-running queries. Furthermore, while MicroVMs like Firecracker optimize for scale in/out, they do not support migration for seamless scale up/down. Engines claiming a serverless approach to elasticity often rely on standard VM provisioning techniques and optimizations that make it easier to move instances around as needed [9]. While serverless cloud analytics platforms like Athena and BigQuery relieve end-users from provisioning VMs, they miss the opportunity to optimize scheduling, placement, and data movement below the high-level interface they expose to end-users. Ultimately these platforms execute query engine code as a black box on coarse-grain VMs under the hood, rather than co-designing the data processing and cloud runtime layers to enable workload-dependent, hardware-aware cross-stack optimization.

3 OUTLINE OF A DECLARATIVE DATA ANALYTICS CLOUD SYSTEM STACK

We propose to design a declarative, cloud-native, distributed data analytics engine by integrating a composite data engine with a declarative cloud execution system. Instead of simply running a query engine on fixed-size VMs that have no insight into application characteristics to apply optimizations (Figure 1b), in our design (Figure 1c) the data processing layer exposes a query's dataflow graph to the underlying infrastructure layer. The infrastructure layer then schedules each operator on compute units (blue squares) based on the available parallelism in the workload and the available hardware on the platform. The infrastructure layer also leverages dataflow information to manage an elastic data cache and optimize data ingestion from data lakes. We start by presenting the data processing layer (based on Maximus) in §3.1 and the declarative infrastructure layer (based on Dandelion) in §3.2. §3.3 presents how we connect the two systems and the end-to-end lifetime of a query.

3.1 Declarative Data Processing Layer

As a starting point, we assume that queries are written in standard SQL. The data processing layer translates this query into a dataflow graph, by using Maximus [27]. Maximus allows executing query plans using different backends, which can run on heterogeneous hardware like CPUs or GPUs. Maximus introduces the concept of operator-level integration, which allows using operators from different engines. This means that even within the same query plan, Maximus can combine operator implementations from different engines and execute them on different backends while taking care of data transfers (e.g. CPU-GPU data transfers), conversions (e.g. between different in-memory data formats) and resource allocation (e.g. CPU and GPU memory pool, thread pools).

At the time of writing, Maximus supports CPU and GPU backends. It integrates the CPU operators from Apache Acero [50] and the GPU operators from RAPIDS cuDF [49]. In addition to operators from third-party engines, Maximus also provides its own native operators. Currently, the framework supports GPU and CPU execution of standard benchmarks like TPC-H [52], H2O-G [23] and

ClickBench [15] with a fully-integrated benchmarking and profiling layer [28]. Its modular design allows Maximus to execute query plans using various deployments. The *deployment specification* defines the *device type* (e.g. CPU, GPU) and the *engine type* (e.g. Acero, cuDF, Native) to be used during the execution. The deployment can be specified at the operator level (for a more fine-grained control) or at the query-plan level (in case we want to execute the whole query plan using the same deployment specification.

Unlike in conventional database engines, in Maximus the operators are not necessarily local and can include calls to other systems, following the principle of emerging composite engines that combine and federate several underlying systems. These operators (or sub-systems) are the ones to be invoked as serverless functions as part of the runtime system.

Using dataflow graphs is, of course, not unique to Maximus and can be found in several other systems such as Spark [63]. This is an advantage for the design as it can be extended to such systems. The new aspect is passing the dataflow graph – which contains useful information (extracted from the query) about what data needs to be ingested, intermediate results, and potentially statistics or additional resources available like indexes — to a serverless execution layer. A declarative, serverless infrastructure layer can then leverage this information to optimize scheduling, placement, caching, and data prefetching.

3.2 Declarative Cloud Infrastructure Layer

We use Dandelion [32] as an example of an elastic cloud platform with a declarative interface, which exposes the dataflow of an application (e.g., a database query) to the underlying execution system. Dandelion executes applications expressed as direct acyclic graphs (DAGs) of *pure compute functions* and *communication functions*. Pure compute functions contain data processing layer code (e.g., query operator logic, e.g., a filter) while communication functions are infrastructure layer code that enables interactions with external storage services. Each function declares its inputs and outputs.

Figure 2 illustrates an example application that loads a set of tables from cloud storage, filters and groups the data, then performs aggregation per group. On top, we show the application implemented as a single function, as it would run on existing serverless platforms like AWS Lambda. In the middle, we show the application as a DAG of functions and on the bottom an example instantiation, where five tables are fetched and the filtering produces two groups. In the Dandelion DAG, an arrow between two functions represents a set, which contains an arbitrary number of items. Each set is associated with a parallelization keyword telling Dandelion how it should parallelize over it items. The `each` keyword specifies that each item can be processed individually, so Dandelion will spawn a separate function instance for each item. In the example, the downloading and filtering use the `each` keyword, as each file can be downloaded and filtered independently. The `key` keyword tells Dandelion to group items by keys, which are set by the previous function, and to execute one function instance per key. This is used for aggregation, as all items with the same key need to be processed by a single function. Dandelion also supports the keyword `all` (not shown in this example), which forwards all items in a set to a single function instance. If a function has multiple input sets Dandelion

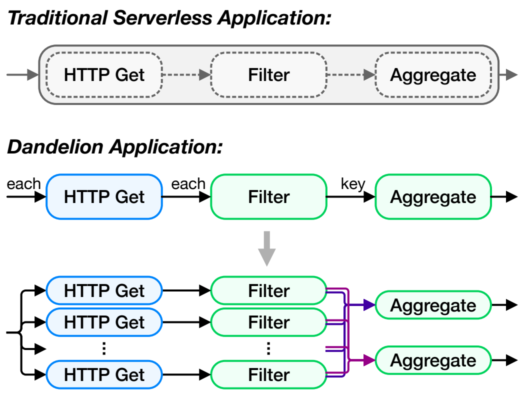


Figure 2: Decomposing a cloud application into a DAG of compute (green) and communication (blue) functions.

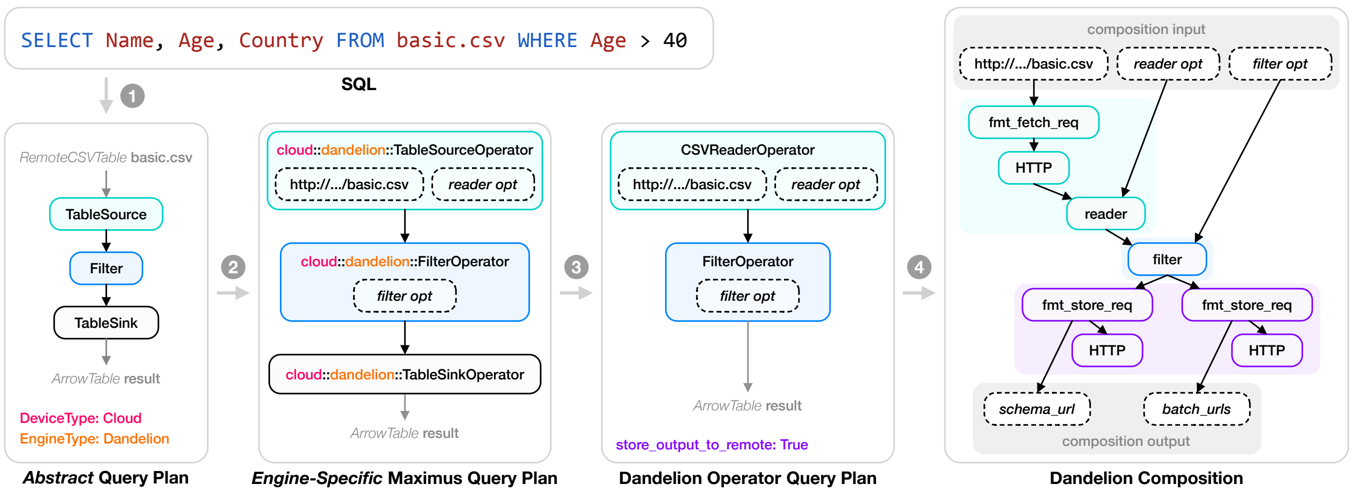
builds the cross product of the instances that should be spawned per set. For example, a function with two input sets annotated with `each` will execute once for each combination of items between the two sets, or if the second set is annotated with `key` a combination of each item in the first set and each group from the second. For the `key` keyword, the user can also specify other join strategies like left, right, inner and outer joins to replace the default cross join. The platform can directly pass items of any size between functions without relying on external services, like cloud storage. In our example, Dandelion would parallelize downloading and filtering the input tables by spawning one function instance for each input document.

Dandelion provides a simple domain-specific language to specify these DAGs. It also provides a library of implemented communication functions, such as the `HTTP Get` function used in this example.

The explicit dataflow given by the DAG representation enables the infrastructure layer to transparently optimize data movement between functions and to/from external storage. Before executing each function in the application DAG, Dandelion prepares a memory *context* for the function by pre-allocating a region of memory and copying or mapping the function's input sets and code into the context leaving the rest of the region to be used for stack and heap. Output items can be anywhere in the context, allowing the function to allocate space for them on the heap or manipulate inputs in place.

The infrastructure layer can monitor the input data that each query needs to ingest and decide which inputs to cache (see §5.2). The Dandelion infrastructure layer can also optimize placement to execute functions on hardware that is close to the function's input data. Dandelion can leverage specialized networking hardware under the hood to maximize data transfer bandwidth between nodes and/or pass data in-memory between functions that execute on the same node. Additionally, the different parallelization keywords in the DAG allow the data processing layer to communicate the intended scaling behavior to the infrastructure layer, leaving detailed decisions to the execution system for when more information about the location and size of input data, intermediate results, and hardware is available. We discuss intra-query elasticity in §5.1.

The infrastructure layer is also responsible for securely isolating code and data (i.e., each function's memory context) from different



```
SELECT Name, Age, Country FROM basic.csv WHERE Age > 40
```

Figure 3: An SQL query statement is first parsed and optimized into an *abstract* query plan (1). Next, using the deployment specification, an *engine-specific* query plan is created (2) which for the Dandelion device is then translated into a *Dandelion query* (3). Finally, a *Dandelion composition* is exported from the *Dandelion query* and sent to the runtime for execution (4).

tenants in shared cloud environments. Dandelion leverages the separation between pure compute and communication functions in the application DAG to securely isolate untrusted data processing code in lightweight sandboxes that boot quickly for high elasticity [32]. Dandelion enables sub-millisecond cold starts while maintaining strong isolation guarantees. At the time of writing, Dandelion supports four different sandboxing mechanisms, including lightweight VMs based on Linux KVM with no guest OS inside [32, 57]. These lightweight VMs can boot in hundreds of microseconds as they do not require the expensive initialization of an entire OS and network stack, since pure compute functions do not need to call into the OS during execution, as long as the cloud platform has pre-allocated a memory region for the function where the function can read/write inputs/outputs and intermediate state. Communication functions consist of code implemented by the (trusted) Dandelion platform and hence do not execute in sandboxes.

3.3 Interface Design

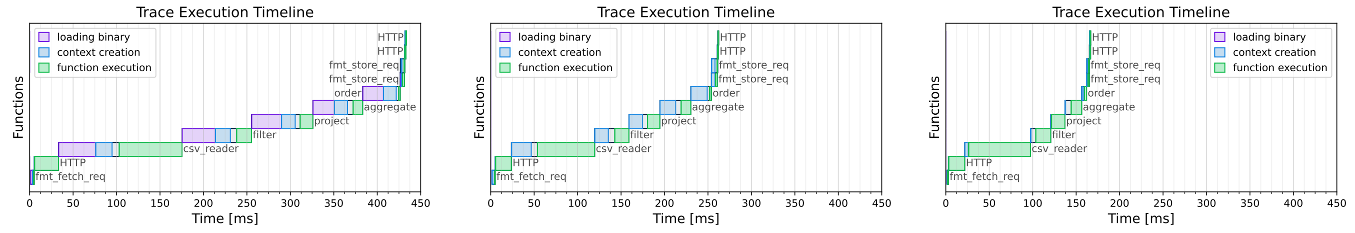
We now discuss how we integrate Dandelion as a cloud-execution backend in Maximus and show how a query (Figure 3) goes from a plan to an executable DAG of functions.

In a first step, Maximus parses and optimized the SQL query into an optimized *abstract* plan. The plan is *abstract* in that it specifies only the type of the operators (e.g. hash-join, filter, projection) but does not yet contain the physical operators used to do the actual computation. The deployment specification (see Section 3.1) further defines the device type and the engine type to be used for execution. It can be set according to the configuration of the system, the location of the data and the availability of resources. For simplicity, we assume that in our example the device in the deployment specification for all abstract operators is set to CLOUD, thus all operators are executed on Dandelion. In the more general case where the abstract query contains operators deployed on different devices (so only parts of the query are executed on Dandelion), the same translation process is applied per sub-query of connected abstract operators that all have the Dandelion deployment specification.

In a second step, Maximus instantiates the engine-specific operator for each abstract one. Compared to common instantiated operators, which are used to perform their operations on the corresponding device during query execution in Maximus, the engine-specific operators for the Dandelion engine are merely placeholders. Apart from containing the translated operator options, they have no additional functionality. The reason for this is that the Dandelion runtime schedules its operator functions by itself, as opposed to the other Maximus devices, where the Maximus executor takes care of all the scheduling. As a result, instead of executing operators individually on Dandelion, we send the entire (sub-)query to the runtime and execute it as a single task.

After instantiating the (sub-)query of Dandelion operators, a third step translates them into a Dandelion query. This step involves mapping each Maximus operator to the corresponding Dandelion one. While we map most operators one-to-one, readers are added for table sources and writers are added for table sinks only if the input and output data types of the query are different from the internal Arrow format. Furthermore, the system derives for each leaf operator whether the data is passed directly within the request or needs to be fetched from a remote storage service. Similarly, the system also determines whether the query output is stored on a remote storage service or returned directly.

The fourth and last step involves generating a composition from the Dandelion query. Each operator is implemented as a compute function in Dandelion. Thus, exactly one compute function is added to the composition DAG for each operator in general. One exception are source nodes that need to load data from remote storage. In this case, we prepend a compute function to format the request and a HTTP communication function to perform the data loading in front of the operator. Similarly, if the output needs to be persisted in a remote storage, we append the functions to do the storing. Each



(a) **Operators need to be loaded into memory (first execution).**

(b) **Operators loaded in memory, data copied between functions.**

(c) **Operators loaded in memory, zero-copy data between functions.**

Figure 4: TPC-H query 1 executed with a scale factor of 0.01. Initially Dandelion loads each operator binary into memory before execution (a). Preloading the operators on startup removes this latency (b). Using a zero-copy mechanism further reduces the function setup overhead as data passed between functions no longer needs to be copied (c).

operator function expects an input set with a single item that contains the operator options, an input set with an item that contains the data schema, and an input set with one item per data batch. The function is parallelized over the data batch items according to the operator type. To aid parallelization, we can add *splitter* operators that split batches into sub-batches. All data movement between operators including broadcasting is done by the Dandelion runtime.

4 PRELIMINARY RESULTS

We now analyze the initial prototype using TPC-H queries 1 and 2.

Hardware Setup. We run the experiments on a cluster of three Cloudlab d430 nodes. The first node hosts a Maximus instance, the second one an instance of Dandelion, and the third node acts as the remote storage service. Each node runs an Intel Xeon E5-2630 v3 CPU at 2.4 GHz clock speed with 8 physical cores over two sockets and has 64 GB DDR4 RAM and 200 GB SSD storage. All nodes are connected with 10 Gbps links and run Ubuntu 20.04.

4.1 Function Startup Optimizations

We start by evaluating our prototype system on a small scale factor of 0.01 generating roughly 10 MB of total data. With so little data to be processed, reducing the startup time of each operator function is important to achieve low latency. Figure 4 shows TPC-H query 1 execution traces as we apply various optimizations.

In the least optimized case (Figure 4a), Dandelion loads each function binary into memory before it is executed. This adds a significant amount of latency when the data being processed is minimal. We can eliminate that latency by explicitly loading every operator binary into memory on startup. In the resulting system (Figure 4b), the binary loading overhead is gone, but creating the memory context of each function now amounts to a significant portion of the total latency. The main reason for this is that the system copies the function binary and each output set from the previous function into the context of the new function. While the function binary size is constant, the overhead from copying the data sets also scales with the amount of data being processed, and thus would remain an issue for larger scale factors. We solve this problem by implementing a zero-copy mechanism in Dandelion. Using this approach, the operator binary and the output sets are copied into the consuming function's context only when it accesses the data using the page fault handler. However, due to the inner workings of the HTTP communication function, zero-copying the output sets coming from the network is not yet implemented, and instead they are still explicitly copied into the new context before the function execution. This is an aspect that can be easily optimized in teh future. The resulting system (Figure 4c) has minimal context creation overheads.

4.2 Processing More Data

We now increase the scale factor to 1, which amounts to roughly 1 GB of total data. As each operator processes more data, the startup overhead becomes less important, and the latency is dominated by the execution efficiency of the operators.

For query 1, which reads only the single largest table, the top part of Figure 5 shows that the *csv-reader* now constitutes a large portion of the overall latency. This is partly because Dandelion needs to copy the entire network context from the HTTP function into the reader context (blue part of the reader execution), as our current implementation of Dandelion cannot yet zero-copy network contexts. However, the main reason for the long execution time of the reader is that Dandelion executes each function on a single core. When the query is expressed with a single reader function, the reader parses the file using only a single thread. We can improve performance by increasing reader parallelism using multiple invocations of the reader function, each working on its own chunk of the input file. The resulting trace (bottom part of Figure 5) shows a significant improvement in latency for data reading. In addition, since every reader produces a separate output batch, Dandelion can automatically parallelize the subsequent filter and projection operators. Each processes one of the output batches, further reducing the overall latency.

We also compare the execution of query 2 with and without reader parallelization in Figure 6. In contrast to query 1, which only reads a single large table, query 2 loads and joins five smaller tables. As a result, this query contains a lot of inter-operator parallelism. Figure 6 shows how Dandelion makes use of this parallelism and already starts processing the smaller tables while continuing to load the larger ones. As in query 1, we see that parallelizing the reader that processes the largest of the five tables significantly lowers the end-to-end latency.

Currently, processing queries with large data is bottlenecked by kernel lock contention that occurs when Dandelion demand pages a

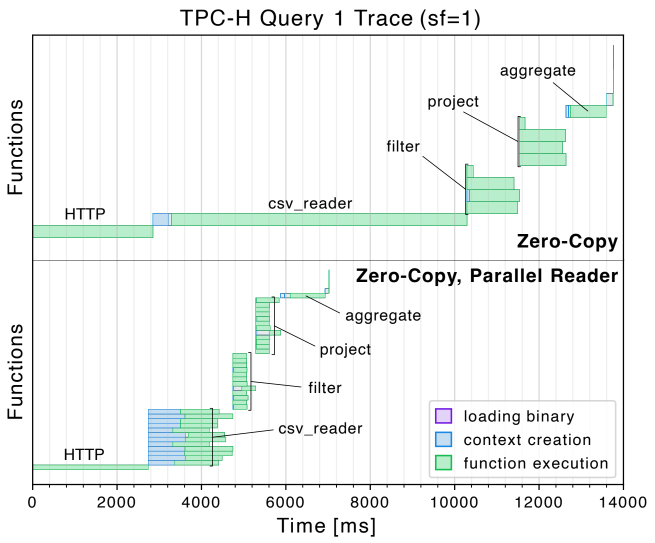


Figure 5: TPC-H query 1 with scale factor 1, with and without parallelizing the *csv_reader* operator.

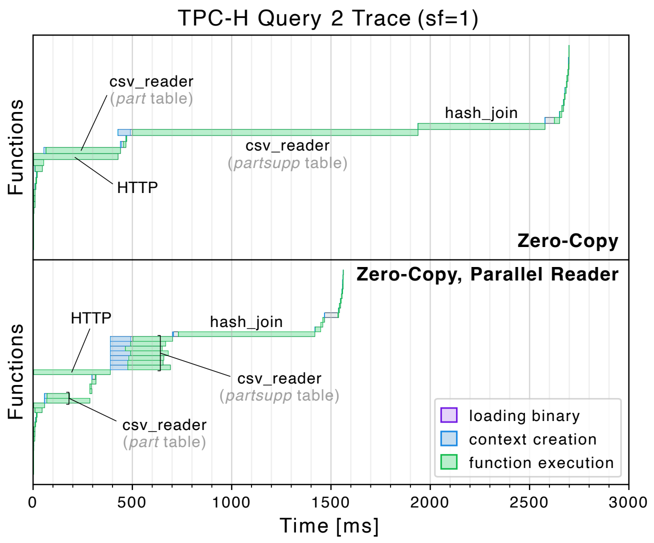


Figure 6: TPC-H query 2 with scale factor 1, with and without parallelizing the *csv_reader* operator.

function's memory context. We are implementing a custom memory management mechanism to alleviate this bottleneck, which will enable efficiently running queries with larger scale factors.

4.3 End-to-End Latency

Finally, we compare the end-to-end latencies when executing TPC-H queries 1 and 2 with scale factor 0.01 and 1. Figure 7 shows the latencies for scale factor 0.01 using the Maximus CPU backend as a baseline which under the hood uses Acero [50]. Acero is a C++ library that implements common data processing operators that take in (potentially multiple) streams of input data and output a single stream of data. It is part of the Apache Arrow project [51]. Looking at the Dandelion latencies, we can clearly see the large impact of preloading the operator binaries in memory and the zero-copy mechanism that minimizes data movement between functions. The resulting system shows ~1.9x slowdown for query 1 and ~1.5x slowdown for query 2 in its current state. The additional latency in Dandelion compared to Acero comes mainly from the need to serialize the data between operators and the startup time of each operator. In addition we send the query plan to a separate node running Dandelion and need to retrieve the output result from the remote storage.

Figure 8 shows the latencies for the same queries but processing more data, with scale factor 1. For both queries, we observe that the zero-copy optimization does not result in the same relative performance gains as for the smaller scale factor. This is because copying the data constitutes a smaller portion of the overall latency and because the HTTP input sets for the reader still need to be copied (as mentioned in Section 4.1). Parallelizing the reader significantly improves query 1 latency, in turn leading to more parallelization in the following filter and projection operators. For query 2, Dandelion achieves a lower latency than the baseline Acero backend. This is because the Maximus Acero CPU backend loads all tables fully before it starts executing the query plan. Dandelion, in comparison, starts processing the smaller tables while it is still loading the larger ones (as seen in Figure 6).

On top of the zero-copy and parallelization optimizations, Figure 8 also shows the latency when base tables are cached in memory (i.e., the outputs of the *HTTP* and *csv_reader* functions are reused) using our simple caching implementation. This optimization is helpful when queries repeatedly read the same data. We discuss more general caching support at the infrastructure layer in § 5.2.

5 CHALLENGES AND OPPORTUNITIES

Our early prototype provides a vehicle to explore opportunities in co-designing composite data processing engines with declarative cloud execution systems. We discuss ideas and research directions.

5.1 Elasticity

Data analytics workloads and their resource requirements vary widely across days [53]. The highly distributed nature of the cloud enables a wide degree of parallelism to optimize I/O in disaggregated storage environments and scale computation within and across machines based on the workload. But how should the database engine decide the right degree of parallelism and the amount of resources to allocate in the cloud?

Most data analytic services run the query engine on opaque, per-tenant VM clusters, which are too slow to resize on-demand. Adding or removing nodes incurs delays of at least seconds [2, 7, 9, 16], whereas a recent analysis of the AWS Redshift fleet showed that over 86% of queries execute for 1 second or less [53]. Some services expose the query execution graphs to a multi-tenant compute infrastructure to provide fined-grained elasticity [19]. However, there are still considerable overheads (typically 100ms to 1s) for sandboxing untrusted UDFs. Given the high load variance and short execution time in modern data analytic workloads, the current level of compute elasticity is not sufficient. As a result, cloud analytics deployments often over-provision resources, which increases cost.

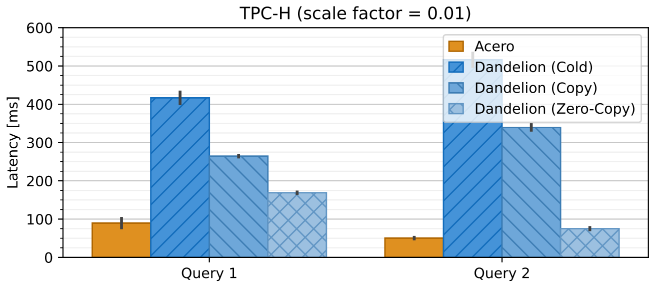


Figure 7: Latencies executing TPC-H queries 1 and 2 with scale factor 0.01 comparing native Maximus (Acero) and different Dandelion optimizations.

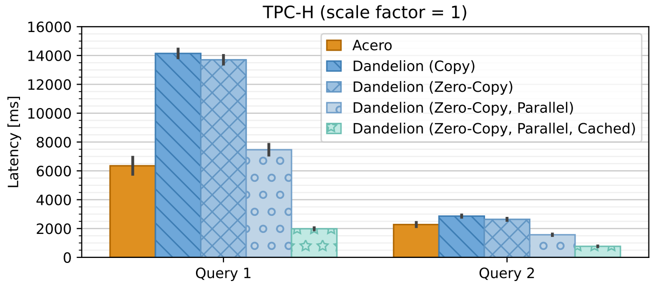


Figure 8: Latencies executing TPC-H queries 1 and 2 with scale factor 1 comparing native Maximus (Acero) and different Dandelion optimizations.

Opportunity. *Inter- and intra-query elasticity.* There is an opportunity to combine the advanced query optimization capabilities of conventional engines with the system and resource optimization capabilities of cloud infrastructure. An elastic platform like Dandelion can dynamically scale cloud resources to run an arbitrary number of concurrent queries as well as parallelize individual queries based on the number of data-independent tasks (Figure 9). For each parallelizable operator, the Maximus engine can also insert splitter functions into the DAG to partition its input into reasonable sizes, and use the parallelization keywords (`each`,`key`) to instruct Dandelion about the maximum degree of parallelism. By monitoring hardware utilization and analyzing workload size, the platform could automatically adjust the degree of parallelism with user-provided optimization targets. This fine-grain elasticity is enabled by co-designing a simple declarative interface between the query engine and the cloud infrastructure.

We ran our initial experiments in Section 4 on a single node. We are adding multi-node support in Dandelion to achieve greater elasticity for running larger scale queries in the cloud.

Challenge. *Which layer of the stack should decide the degree of parallelism?* In the cloud, parallel streams are often used to compensate for limited bandwidth. An open question is whether the query planner should decide the number of parallel data streams (and specify this as part of the dataflow graph) or whether this decision should be left to the underlying infrastructure, which can decide based on the available capacity and create as many parallel functions as needed to optimize data loading. The former approach simplifies the system. The latter opens up more opportunities for optimization. Currently, the data layer extracts the base data accessed by the query from the SQL statement and provides a mapping to files in storage that contain the data. In this way, the data flow that contains the query can state the data it needs in the format needed by the infrastructure. However, this implies that the infrastructure layer has access to the catalog information with the metadata mapping tables to files.

5.2 Caching

Relational engines heavily use caching to optimize overall performance. Materialized views and incremental view maintenance are well studied in the database literature for reusing expensive query results. Prior work has also explored automatic, fine-grained cache management for general data processing at data-center scale [22] and for web applications [18]. In a serverless setting, however, this is more difficult and has led to several designs, typically based on introducing a separate, ephemeral cache that the users explicitly interact with (e.g. Anna KVS [61]). How caching should be automatically applied for serverless data analytics is still an open question, since the query execution is usually opaque to the cloud. With our declarative engine-infrastructure interface, more sophisticated cache designs become feasible.

Opportunity. *Caching data across queries.* When the infrastructure layer has visibility into each function's declared input sets, it can collect usage statistics and temporarily keep data in the memory of a particular machine. Dandelion can schedule a subsequent function with partially cached inputs on the same machine so that the data is read directly from memory, as shown in Figure 10. This is feasible because the operator in a function no longer accesses the data directly but declares what it needs and the infrastructure fetches it. When the system recognizes that a query requests data that has already been cached, it can reuse it instead of loading it again. Since pure compute functions are deterministic with respect to their inputs, the system can also cache intermediate results and skip function execution when it detects that all inputs are the same as a previous execution of the function. For example, after filtering or projection, tables could be opportunistically kept around when there is sufficient memory or storage capacity and reuse is likely [48]. The exact cache insertion and eviction could be based on runtime statistics, such as the duration since last used, overhead to store the data, and compute resources to derive the data [22].

Challenge. *Deciding what to cache, how to reuse, and when to share?* Many classical caching questions need to be re-evaluated in this new context. When servers are processing requests from a large number of different users, it becomes much harder to decide which data is worth keeping around and how much capacity to allocate to the caches. While caching a raw table might increase the chances of another query seeing a cache hit, caching the result after the filter operation might not lead to as many hits, but could be much cheaper to keep around, offering an interesting tradeoff. Even if the cluster has cached input data for some operator in a query on a particular node, the node holding the data may not have enough resources to execute that operator immediately. The cloud infrastructure should systematically decide whether to delay the execution until there are available resources in the node with cached data or schedule the operator in a different node by

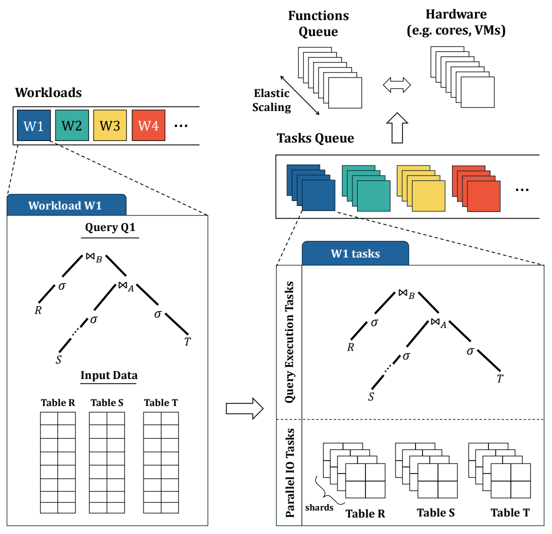


Figure 9: Elastic query processing: workloads are decomposed into compute & IO tasks, and dynamically assigned resources.

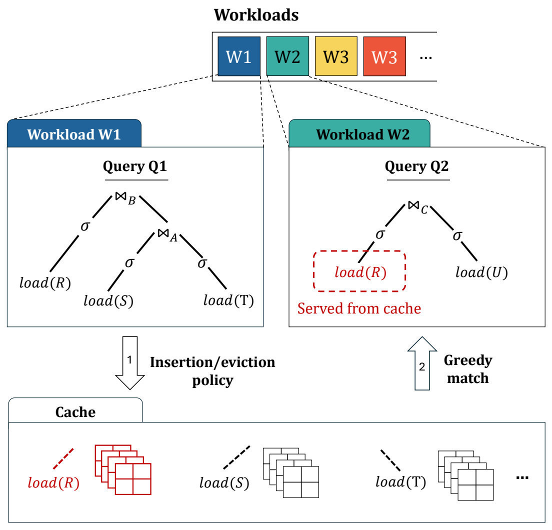


Figure 10: Elastic data caching allows reusing data from different queries across functions, optimizing data ingestion.

transferring or repopulating the cached data. Another important factor in deciding what data to cache is how many users may access it. Some data is loaded from private buckets and needs to be kept private, but functions may also access public data [21]. The benefit of caching public data might be much higher, as it can be reused across a larger number of users. Finally, since the query engine no longer owns the data in a datalake environment and therefore no longer knows when the data changes, the cloud infrastructure layer must reload cached data after a stateless/time-to-live (TTL) limit passes or compare data version metadata.

5.3 Heterogeneous Hardware

Very few data processing engines make use of the growing heterogeneity and specialization available in the cloud. For instance, GPUs are becoming a viable option for relational data processing with initial results indicating substantial performance gains [27, 40].

Opportunity. *Leverage specialized hardware.* The combination of a library of operators used in a composite engine and its encapsulation in functions gives the infrastructure layer the freedom to optimize scheduling by looking at runtime information about the current system load, potential slow machines/devices, available alternative devices for execution, etc. There is an increasing number of cloud centric tasks such as encryption and compression as well as a proliferation of different data representations and formats that often require to have filters and translation steps between the storage and the initial operators. The number of choices for how to execute these tasks on hardware platforms has also grown. TPUs, DPUs, GPUs, smart NICs, and specialized processors for encryption, compression or other tasks are becoming pervasive and there is a need to be able to take advantage of them to improve data processing especially as CPUs lose the central role they have been playing so far. Wrapping operators into functions as we envision will make it much easier to evolve engines quickly to adopt new hardware, add operators to query plans, and introduce new features by simply encapsulating specialized, new operators into functions. While many decisions can be made by the query optimizer, the exact availability of hardware may not always be known in advance and making decisions about which processing unit to use, and whether to wait for an accelerator to become available or to run on other hardware in the meantime is difficult. Fully optimizing such queries requires an additional layer for runtime optimizations, to complement regular query plan optimizations.

Challenge. *Which layer decides on what type of hardware a function will fun, e.g., CPU vs. GPU?* If the hardware type is fixed in the query plan, the runtime component cannot optimize the choice depending on runtime information like the current load on different available hardware devices. If the choice of hardware is not specified in the query plan, the data layer component needs to provide code for alternative but functionally equivalent functions that the runtime can choose between when deciding where to run an operator. Is query optimization possible when steps of the plan are open to different instantiations and might induce different data movement? This is an open question to explore.

5.4 Beyond Relational SQL

Users increasingly write data processing workflows that combine relational data operations with arbitrary user-defined logic, including machine learning and other complex transformations.

Opportunity. *Leverage platform level isolation.* Performance is often obtained through batching and efficiency through sharing resources across users. Doing so for relational operators is understood to a certain extent, but it becomes more complex with the proliferation of User Defined Functions (UDFs). A common approach used today to facilitate implementation and enforce isolation is to execute UDFs as serverless functions in isolated sandboxes like MicroVMs (e.g., as done in AWS for Athena [4]). This is expensive, as data needs to be transferred between the cluster running the query engine and the cluster running the serverless functions. Our proposed system supports isolation at the function granularity and can enforce or relax the degree of isolation as needed, since Dandelion supports different isolation mechanisms (e.g., lightweight VMs [57], processes, or rWasm modules [12]). As a result, we can allow arbitrary user-defined logic without the need for a separate system while still maintaining performance isolation and security guarantees. The function just needs to be registered before it can be added to the compositions.

Challenge. While the ability to isolate functions from each other or to enable them to share data is available, it raises the question of how should the system know when to do one or the other. An option is to require the developer to introduce hints or to derive access controls from the structure of the task at hand. This can be as simple as all functions within a DAG or all DAGs from a single user can share data. An alternative is to infer access control rules from users, extrapolate those to the executed DAGs, and then use that information to decide on sharing and placement decisions (as well as access to cache data).

6 RELATED WORK

Declarative systems. The notion of building declarative systems has been suggested in different contexts, e.g., in declarative networking [38], cluster management [47] but also for streaming and distributed systems [14], or cloud programming [13, 17, 30, 31, 37]. All these approaches advocate using higher level abstractions and specifications for defining different aspects of a system rather than requiring the developer or user to describe every step of the process in question. By doing so, the system becomes easier to use and program while also opening the door to optimization opportunities that individual users are most likely unable to implement themselves. The vision we put forward in this paper follows a similar approach for cloud analytics, but end-to-end: take advantage of the declarative nature of data processing and translate/exploit it to configure and optimize the underlying runtime infrastructure.

Data processing in the cloud. One can argue that systems such as a Spark, Presto, or Hive are already declarative. However, their declarative nature is limited to the scope of the analytics engine. The engine has no visibility into the underlying infrastructure, runtime information, or hardware availability. As a result, they can only optimize performance within the scope of their abstracted execution model. This is reflected in their limited elasticity [2, 7, 9, 16] and current extensions that cloud providers have built to improve orchestration and efficiency. Google's Lightning Engine with Intelligent Data Access [20] is designed to improve serverless Spark performance with a vectorized query engine and hot data caching. Similarly, AWS's EMR Serverless Local Storage [5] was designed to optimize the data exchanges caused by Spark's model. These attempts essentially try to work around the missing cross-layer information by extending the cloud layer with heuristics, caching, and storage mechanisms in order to adapt to the application needs. However, although such efforts improve performance, their scope remains limited. Replacing Spark with an optimized version of Apache Gluten [45] and Meta's Velox [42] does not address the fundamental limitation behind the opaque interface between the query engine and cloud infrastructure. Some of these systems also introduce additional limitations. For instance, Microsoft Azure on Serverless Spark relies on Azure Synapse, which leads to starting times of 3 to 5 minutes for the whole infrastructure [39].

Databricks Lakeguard [21] extends Spark with Spark Connect, which they leverage to implement serverless Spark with multiple users on a shared cluster. This is done by splitting queries into operators that run on trusted executors and operators that need to be isolated, like UDFs. Trusted and sandboxed executors of different users can run alongside each other in clusters managed by the Databricks cluster manager. In contrast to the system we propose, the cluster manager relies on statistics collected from previous queries and machine learning predictions to try to find the optimal number of executors a query needs to run on. However, there is no flow of information about the runtime environment and Spark Connect uses the serverless platform as a generic backend.

There are also query engines built atop unmodified cloud function services (e.g., AWS Lambda) using various techniques to work around the limitations of those serverless platforms and therefore leverage the inherent scalability of serverless. Examples include Starling [43], Lambada [41], and Boxer [59]. All these systems try to optimize execution by incorporating (static) knowledge of the platform in to the engine (such as how to optimize data exchanges, spawn many functions efficiently, or facilitate networking) but they do not incorporate any runtime information.

We argue that redesigning the interface between the query engine and the cloud infrastructure to enable simultaneous reasoning about the workload semantics and the runtime environment will make it easier to implement what cloud vendors try to provide in their platforms.

Cache management. Most distributed query engines provide caching for base tables, which are obtained from self-managed file systems (for data warehouses) or remote storage services (for data lakes). Amazon Redshift deploys a result cache to speed up highly-repetitive queries [53]. It can also push down filter and aggregation sub-queries of S3 files into multi-tenant Spectrum nodes and cache these intermediate results [7]. Some engines, like Spark, support manual caching of arbitrary intermediate results by calling library functions or creating a materialized view, but they cannot directly generalize to a serverless setting, where the memory or storage for caching should be provisioned independently of the resources for query execution. Cloud services like Google's Lightning Engine also support caching. Our proposal is to make caching more precise by enabling the data processing layer to provide information about what data is being accessed and what data can be shared, rather than inferring it from the I/O patterns.

7 CONCLUSIONS

In this paper, we propose for query optimization to go beyond query plan optimization by including optimizations at the cloud infrastructure layer. We outlined how this could be done by combining a composite data processing engine with an elastic, declarative cloud execution system. Our initial prototype provides a foundation to explore many open questions, including how to automate elastic scaling based on workload characteristics and dynamic hardware availability, which data to cache with a global view of data movement within and across queries, and how to schedule query execution across heterogeneous hardware.

REFERENCES

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *NSDI*.

[2] Josep Aguilar-Saborit, Raghu Ramakrishnan, Krish Srinivasan, Kevin Bocksrocker, Ioannis Alagiannis, Mahadevan Sankara, Moe Shafiei, Jose Blakeley, Girish Dasarathy, Sumeet Dash, Lazar Davidovic, Maja Damjanic, Slobodan Djunic, Nemanja Djurkic, Charles Feddersen, Cesar Galindo-Legaria, Alan Halverson, Milana Kovacevic, Nikola Kicovic, Goran Lukic, Djordje Maksimovic, Ana Manic, Nikola Markovic, Bosko Mihic, Ugljesa Milic, Marko Milojevic, Tapas Nayak, Milan Potocnik, Milos Radic, Bozidar Radivojevic, Srikumar Rangarajan, Milan Ruzic, Milan Simic, Marko Sosic, Igor Stanko, Maja Stikic, Sasa Stanojkov, Vukasin Stefanovic, Milos Sukovic, Aleksandar Tomic, Dragan Tomic, Steve Toscano, Djordje Trifunovic, Veljko Vasic, Tomer Verona, Aleksandar Vujic, Nikola Vujic, Marko Vukovic, and Marko Zivanovic. 2020. POLARIS: the distributed SQL engine in azure synapse. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3204–3216. https://doi.org/10.14778/3415478.3415545

[3] Rana Alotaibi, Yuanyuan Tian, Stefan Grafberger, Jesús Camacho-Rodríguez, Nicolas Bruno, Brian Kroth, Sergiy Matusevych, Ashvin Agrawal, Mahesh Behera, Ashit Gosalia, César A. Galindo-Legaria, Milind Joshi, Milan Potocnik, Beysim Sezgin, Xiaoyu Li, and Carlo Curino. 2024. Towards Query Optimizer as a Service (QOaaS) in a Unified LakeHouse Ecosystem: Can One QO Rule Them All? *CoRR* abs/2411.13704 (2024). https://doi.org/10.48550/ARXIV.2411.13704 arXiv:2411.13704

[4] Amazon Web Services. [n. d.]. Amazon Athena: Query with user defined functions. https://docs.aws.amazon.com/athena/latest/ug/querying-udf.html

[5] Amazon Web Services. [n.d.]. *Amazon EMR Serverless eliminates local storage provisioning for Apache Spark workloads.* https://aws.amazon.com/about-aws/whats-new/2025/12/amazon-emr-serverless-local-storage-provisioning-apache-spark-workloads/ Accessed: 2025-12-03.

[6] Panagiotis Antonopoulos, Arvind Arasu, Kunal D. Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, Ravi Ramamurthy, Jakub Szymaszek, Jeffrey Trimmer, Kapil Vaswani, Ramarathnam Venkatesan, and Mike Zwilling. 2020. Azure SQL Database Always Encrypted. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*.

[7] 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 *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2205–2217. https://doi.org/10.1145/3514221.3526045

[8] Pankaj Arora, Surajit Chaudhuri, Sudipto Das, Junfeng Dong, Cyril George, Ajay Kalhan, Arnd Christian König, Willis Lang, Changsong Li, Feng Li, Jiaqi Liu, Lukas M. Maas, Akshay Mata, Ishai Menache, Justin Moeller, Vivek Narasayya, Matthaios Olma, Morgan Oslake, Elnaz Rezai, Yi Shan, Manoj Syamala, Shize Xu, and Vasileios Zois. 2023. Flexible Resource Allocation for Relational Database-as-a-Service. *Proc. VLDB Endow.* 16, 13 (Sept. 2023), 4202–4215. https://doi.org/10.14778/3625054.3625058

[9] Bradley Barnhart, Marc Brooker, Daniil Chinenkov, Tony Hooper, Jihoun Im, Prakash Chandra Jha, Tim Kraska, Ashok Kurakula, Alexey Kuznetsov, Grant McAlister, Arjun Muthukrishnan, Aravinthan Narayanan, Douglas Terry, Bhuvan Urgaonkar, and Jiaming Yan. 2024. Resource Management in Aurora Serverless. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 4038–4050. https://doi.org/10.14778/3685800.3685825

[10] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 221–230. https://doi.org/10.1145/3183713.3190662

[11] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2326–2339. https://doi.org/10.1145/3514221.3526054

[12] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. 2022. Provably-Safe Multilingual Software Sandboxing using WebAssembly. In *31st USENIX Security Symposium (USENIX Security 22)*.

[13] Alvin Cheung, Natacha Crooks, Joseph M. Hellerstein, and Mae Milano. 2021. New Directions in Cloud Programming. In *11th Conference on Innovative Data Systems Research, CIDR.*

[14] David C.Y. Chu, Rithvik Panchapakesan, Shadaj Laddad, Lucky E. Katahanas, Chris Liu, Kaushik Shivakumar, Natacha Crooks, Joseph M. Hellerstein, and Heidi Howard. 2024. Optimizing Distributed Protocols with Query Rewrites. *Proc. ACM Managament of Data (SIGMOD)* (2024).

[15] ClickHouse Inc. [n.d.]. ClickBench — a Benchmark For Analytical DBMS. https://benchmark.clickhouse.com/. Accessed: 2025-02-19.

[16] Databricks. [n. d.]. SQL warehouse sizing, scaling, and queuing behavior. https://docs.databricks.com/aws/en/compute/sql-warehouse/warehouse-behavior. Accessed: 2025-11-18.

[17] Yuhan Deng, Angela Montemayor, Amit Levy, and Keith Winstein. 2022. Computation-Centric Networking. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks (HotNets '22)*.

[18] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. 2018. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 213–231. https://www.usenix.org/conference/osdi18/presentation/gjengset

[19] Google. [n. d.]. BigQuery explained: An overview of BigQuery's architecture. https://cloud.google.com/blog/products/data-analytics/new-blog-series-bigquery-explained-overview. Accessed: 2025-11-18.

[20] Google Cloud. [n.d.]. *Introducing Lightning Engine — the next generation of Apache Spark performance.* https://cloud.google.com/blog/products/data-analytics/introducing-lightning-engine-for-apache-spark Accessed: 2025-12-03.

[21] Martin Grund, Stefania Leone, Herman van Hövell, Sven Wagner-Boysen, Sebastian Hillig, Hyukjin Kwon, David Lewis, Jakob Mund, Polo-Francois Poli, Lionel Montrieux, Othon Crelier, Xiao Li, Reynold Xin, Matei Zaharia, Michalis Petropoulos, and Thanos Papathanasiou. 2025. Databricks Lakeguard: Supporting Fine-grained Access Control and Multi-user Capabilities for Apache Spark Workloads. In *Companion of the 2025 International Conference on Management of Data* (Berlin, Germany) *(SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 418–430. https://doi.org/10.1145/3722212.3724433

[22] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: Automatic Management of Data and Computation in Datacenters. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/osdi10/nectar-automatic-management-data-and-computation-datacenters

[23] H2O.ai. [n.d.]. Database-like Ops Benchmark. https://h2oai.github.io/db-benchmark/. Accessed: 2025-02-19.

[24] Jianwei Hao, Ting Jiang, Wei Wang, and In Kee Kim. 2021. An Empirical Analysis of VM Startup Times in Public IaaS Clouds. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. 398–403. https://doi.org/10.1109/CLOUD53861.2021.00053

[25] Michael Haubenschild and Viktor Leis. 2025. Oltp in the cloud: architectures, tradeoffs, and cost. *VLDB Journal* 34, 4 (2025).

[26] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *CIDR*.

[27] Marko Kabić, Shriram Chandran, and Gustavo Alonso. 2025. Maximus: A Modular Accelerated Query Engine for Data Analytics on Heterogeneous Systems. *Proc. ACM Manag. Data* 3, 3, Article 187 (June 2025), 25 pages. https://doi.org/10.1145/3725324

[28] Marko Kabić, Bowen Wu, Jonas Dann, and Gustavo Alonso. 2025. Powerful GPUs or Fast Interconnects: Analyzing Relational Workloads on Modern GPUs. *Proc. VLDB Endow.* 18, 11 (2025), 4350–4363. https://doi.org/10.14778/3749646.3749698

[29] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*.

[30] Ana Klimovic. 2025. The Case for a New Cloud-Native Programming Model with Pure Functions. *SIGMOD Record* 54, 2 (2025).

[31] Tom Kuchler, Michael Giardino, Timothy Roscoe, and Ana Klimovic. 2023. Function as a Function *(SoCC '23)*. 81–92. https://doi.org/10.1145/3620678.3624648

[32] Tom Kuchler, Pinghe Li, Yazhuo Zhang, Lazar Cvetković, Boris Goranov, Tobias Stocker, Leon Thomm, Simone Kalbermatter, Tim Notter, Andrea Lattuada, and Ana Klimovic. 2025. Unlocking True Elasticity for the Cloud-Native Era with Dandelion. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'25)*.

[33] Justin Levandoski, Garrett Casto, Mingge Deng, Rushabh Desai, Pavan Edara, Thibaud Hottelier, Amir Hormati, Anoop Johnson, Jeff Johnson, Dawid Kurzyniec, Sam McVeety, Prem Ramanathan, Gaurav Saxena, Vidya Shanmugan, and Yuri Volobuev. 2024. BigLake: BigQuery's Evolution toward a Multi-Cloud Lakehouse. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 334–346. https://doi.org/10.1145/3626246.3653388

[34] Feifei Li. 2023. Modernization of Databases in the Cloud Era: Building Databases that Run Like Legos. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 4140–4151. https://doi.org/10.14778/3611540.3611639

[35] Ji You Li, Jiachi Zhang, Yuhang Liu, Wenchao Zhou, Xin Zhou, Fangyuan Zhou, and Feifei Li. 2025. Eigen+: Memory Over-Subscription for Alibaba Cloud Databases. In *Companion of the 2025 International Conference on Management of Data* (Berlin, Germany) *(SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 525–538. https://doi.org/10.1145/3722212.3724435

[36] Ji You Li, Jiachi Zhang, Wenchao Zhou, Yuhang Liu, Shuai Zhang, Zhuoming Xue, Ding Xu, Hua Fan, Fangyuan Zhou, and Feifei Li. 2023. Eigen: End-to-End Resource Optimization for Large-Scale Databases on the Cloud. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 3795–3807. https://doi.org/10.14778/3611540.3611565

[37] Tianyu Li, Badrish Chandramouli, Sebastian Burckhardt, and Samuel Madden. 2024. Serverless State Management Systems. In *CIDR 2024, Conference on Innovative Data Systems Research*.

[38] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2009. Declarative networking. *Commun. ACM* 52, 11 (2009).

[39] Microsoft Azure. [n.d.]. *Apache Spark in Azure Machine Learning*. https://learn.microsoft.com/en-us/azure/machine-learning/apache-spark-azure-ml-concepts?view=azureml-api-2 Accessed: 2025-12-03.

[40] Hubert Mohr-Daurat, Xuan Sun, and Holger Pirk. 2023. BOSS - An Architecture for Database Kernel Composition. *Proc. VLDB Endow.* 17, 4 (Dec. 2023), 877–890. https://doi.org/10.14778/3636218.3636239

[41] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *SIGMOD*.

[42] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: meta's unified execution engine. *Proc. VLDB Endow.* 15, 12 (Aug. 2022), 3372–3384. https://doi.org/10.14778/3554821.3554829

[43] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *SIGMOD*.

[44] 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. *Commun. ACM* 64, 5 (April 2021), 76–84.

[45] Akash Shankaran, George Gu, Weiting Chen, Binwei Yang, Chidamber Kulkarni, Mark Rambacher, Nesime Tatbul, and David E. Cohen. 2023. The Gluten Open-Source Software Project: Modernizing Java-based Query Engines for the Lakehouse Era. In *Joint Proceedings of Workshops at the 49th International Conference on Very Large Data Bases (VLDB 2023), Vancouver, Canada, August 28 - September 1, 2023 (CEUR Workshop Proceedings, Vol. 3462)*. CEUR-WS.org. https://ceur-ws.org/Vol-3462/CDMS8.pdf

[46] substrait io. 2021. *Substrait: Cross-Language Serialization for Relational Algebra*. https://github.com/substrait-io/substrait

[47] Lalith Suresh, João Loff, Faria Kalim, Sangeetha Abdu Jyothi, Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, and Michael Gasch. 2020. Building scalable and flexible cluster managers using declarative programming. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*.

[48] Jacopo Tagliabue, Ryan Curtin, and Ciro Greco. 2024. FaaS and Furious: abstractions and differential caching for efficient data pre-processing . In *2024 IEEE International Conference on Big Data (BigData)*. IEEE Computer Society, Los Alamitos, CA, USA, 3562–3567. https://doi.org/10.1109/BigData62323.2024.10825377

[49] RAPIDS Development Team. 2023. *RAPIDS: Libraries for End to End GPU Data Science*. https://rapids.ai

[50] The Apache Software Foundation. [n.d.]. Acero: A C++ streaming execution engine. https://arrow.apache.org/docs/cpp/streaming_execution.html. Accessed: 2025-07-20.

[51] The Apache Software Foundation. [n.d.]. A cross-language development platform for in-memory analytics. https://arrow.apache.org/. Accessed: 2025-07-20.

[52] The Transaction Processing Council. [n.d.]. The TPC-H Benchmark. https://www.tpc.org/tpch/. Accessed: 2025-07-20.

[53] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is Not Enough: An Analysis of the Amazon Redshift Fleet. *Proc. VLDB Endow.* 17, 11 (July 2024), 3694–3706. https://doi.org/10.14778/3681954.3682031

[54] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*.

[55] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building an Elastic Query Engine on Disaggregated Storage. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'20)*. USENIX Association, USA, 449–462.

[56] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18)*.

[57] Nicholas C. Wanninger, Joshua J. Bowden, Kirtankumar Shetty, Ayush Garg, and Kyle C. Hale. 2022. Isolating functions at the hardware limit with virtines. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 644–662. https://doi.org/10.1145/3492321.3519553

[58] Michael Wawrzoniak, Rodrigo Bruno, Ana Klimovic, and Gustavo Alonso. 2024. Boxer: FaaSt Ephemeral Elasticity for Off-the-Shelf Cloud Applications. arXiv:2407.00832 [cs.DC] https://arxiv.org/abs/2407.00832

[59] Michal Wawrzoniak, Gianluca Moro, Rodrigo Bruno, Ana Klimovic, and Gustavo Alonso. 2024. Off-the-shelf Data Analytics on Serverless. In *14th Conference on Innovative Data Systems Research, CIDR*.

[60] Michal Wawrzoniak, Ingo Müller, Rodrigo Bruno, and Gustavo Alonso. 2021. Boxer: Data Analytics on Network-enabled Serverless Platforms. In *CIDR*.

[61] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. 2018. Anna: A KVS for Any Scale. In *ICDE*.

[62] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org.

[63] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (oct 2016), 56–65.

[64] Tobias Ziegler, Philip A. Bernstein, Viktor Leis, and Carsten Binnig. 2023. Is Scalable OLTP in the Cloud a Solved Problem?. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*.