Modularis: Modular Relational Analytics over Heterogeneous Distributed Platforms

Dimitrios Koutsoukos
ETH Zurich, Switzerland
dkoutsou@inf.ethz.ch

Ingo Müller
ETH Zurich, Switzerland
ingo.mueller@inf.ethz.ch

Renato Marroquín*
Oracle Inc., Zurich, Switzerland
renato.marroquin@oracle.com

Ana Klimovic
ETH Zurich, Switzerland
aklimovic@ethz.ch

Gustavo Alonso
ETH Zurich, Switzerland
alonso@inf.ethz.ch

ABSTRACT

The enormous quantity of data produced every day together with advances in data analytics has led to a proliferation of data management and analysis systems. Typically, these systems are built around highly specialized monolithic operators optimized for the underlying hardware. While effective in the short term, such an approach makes the operators cumbersome to port and adapt, which is increasingly required due to the speed at which algorithms and hardware evolve. To address this limitation, we present *Modularis*, an execution layer for data analytics based on *sub-operators*, i.e., composable building blocks resembling traditional database operators but at a finer granularity. To demonstrate the feasibility and advantages of our approach, we use Modularis to build a distributed query processing system supporting relational queries running on an RDMA cluster, a serverless cloud platform, and a smart storage engine. Modularis requires minimal code changes to execute queries across these three diverse hardware platforms, showing that the sub-operator approach reduces the amount and complexity of the code to maintain. In fact, changes in the platform affect only those sub-operators that depend on the underlying hardware (in our use cases, mainly the sub-operators related to network communication). We show the end-to-end performance of Modularis by comparing it with a framework for SQL processing (Presto), a commercial cluster database (SingleStore), as well as Query-as-a-Service systems (Athena, BigQuery). Modularis outperforms all these systems, proving that the design and architectural advantages of a modular design can be achieved without degrading performance. We also compare Modularis with a hand-optimized implementation of a join for RDMA clusters. We show that Modularis has the advantage of being easily extensible to a wider range of join variants and *group by* queries, all of which are not supported in the hand-tuned join.

PVLDB Reference Format:
Dimitrios Koutsoukos, Ingo Müller, Renato Marroquín, Ana Klimovic, Gustavo Alonso. Modularis: Modular Relational Analytics over Heterogeneous Distributed Platforms. PVLDB, 14(13): 3308 - 3321, 2021.
doi:10.14778/3484224.3484229

*The work of this author was done while employed at ETH Zurich.

Proceedings of the VLDB Endowment, Vol. 14, No. 13 ISSN 2150-8097.
doi:10.14778/3484224.3484229

1 INTRODUCTION

The growing popularity of machine learning applications and the increasing amount of data that analytics applications must process have had a substantial influence on the way systems are designed and optimized. There is a constant stream of specialized accelerators (TPUs, GPUs, FPGAs, smart NICs, smart storage, near memory processing) and platforms (large appliances, InfiniBand clusters, serverless, cloud instances, data centers) that forces a continuous redesign of data processing engines—often leading to new engines—simply to exploit the capabilities of new hardware [16].

Often, to gain performance, developers design monolithic operators that are highly tailored to the underlying hardware [11, 13, 14, 52, 56]. However, as the algorithms and platforms evolve quickly, it becomes very difficult to reuse these operators in newer versions of the system. Examples abound: For instance, a join optimized for multi-core machines [11] requires fundamental changes to run on Remote Direct Memory Access (RDMA) [13, 14] due to the different communication schemes between NUMA nodes and the network. As another example, although FPGAs are not competitive with multi-core machines for full joins, they can significantly accelerate the partitioning phase [40]. Supporting distributed query processing on serverless computing has received a lot of recent attention and requires a specialized exchange operator to allow communication through storage [52, 56]. With the current approach of highly engineered, monolithic operators, it is difficult to exploit the potential of new architectures and platforms without major redesigns.

We argue that to cope with the fast changes in the hardware and platform landscape, query processing needs to become more modular and composable at a finer granularity than conventional relational operators. Having many versions of highly optimized, monolithic operators is not a design approach compatible with the high degree of specialization we observe. In almost all cases where hardware or platform advances offer new opportunities, the potential advantages affect only part of an operator (e.g., only one of partitioning, build, or phase of a join) and often require to change other significant parts (e.g., intermediate data placement in an exchange operator, partitioning strategies, etc.). It is very rare that the whole operator can become faster or that all operators benefit. This effect is notable in accelerators (FPGA, GPU) [22, 26, 29, 32, 34–36, 57, 65, 66], specialized processor components (AVX, SGX), [9, 11, 20, 42, 53, 63] evolving networks in distributed systems (Infiniband, RDMA, smart NICs, etc.) [13, 17, 50, 51, 60, 70, 71], and even platforms (Infiniband clusters [13, 50, 51], cloud/serverless computing [10, 19, 30, 39, 43–45, 52, 56, 59, 62, 64]). Yet, the current design approaches often

require a complete redesign because they are based on careful tailoring to the underlying platform and hardware.

In this paper, we focus on the modular design of query processing. We show how to design a modular distributed query processing engine with performance comparable to its monolithic counterparts. The topic of modularity in database operators has been visited many times in the past [25, 37] and recently [24, 47]. However, to our knowledge, we are the first to implement modularity at the hardware platform level, while at the same time formulating concrete design principles. We argue that modularity at this level is a necessity rather than a nice-to-have feature. To this end, we have built *Modularis*—an execution engine aiming to maximize performance without specializing neither the engine nor the bulk of the operators to the target platform. Modularis is based on a collection of composable sub-operators that are both as small and simple as possible, as well as reusable, while retaining the ability to execute entire SQL queries. Modularis' sub-operators share the same goal of composability of operators present in traditional database engines: sub-operators can be freely and easily combined, have a well-defined interface, and adhere to a common execution model. Our sub-operators are similar to the microcode used in processor design to implement more complex instructions. We use them to build up complex plans like TPC-H queries. This allows Modularis to run seamlessly on three different platforms: an InfiniBand RDMA cluster, a serverless cloud service, and a smart storage engine, by simply replacing in the query plan only those operators actually affected by the change in the platform (e.g., the exchange operator), leaving everything else unaffected.

We have evaluated Modularis' performance and behavior extensively. First, to explore the end-to-end performance of Modularis, we compare it to mature systems using the TPC-H benchmark. When compared to Presto, a system that is general enough to utilize various storage layers and distributed set-ups, Modularis is an order of magnitude faster. When compared to SingleStore (previously called MemSQL), a system that specializes in in-memory analytics using SQL, Modularis is faster for the majority of the queries. The speed-up compared to both of these systems comes from the ability to optimize at the sub-operator level and the usage of RDMA for fast data transfer. Next, we show how Modularis adapts to heterogeneous environments by discussing the minimal changes necessary to go from running on an RDMA cluster to a serverless platform using AWS Lambda or a smart storage engine (S3Select) —a very significant architectural change in the underlying platform. For TPC-H queries, Modularis-on-serverless is competitive against commercial Query-as-a-Service systems (Athena, BigQuery). This last experiment shows that modularity does not need to result in end-to-end performance losses while it enables the execution of workloads on two fundamentally different platforms with minimal development effort, which mainly involves the design of new exchange and executor operators. That way, we get the best of both worlds: maximum performance across platforms, without redesigning the whole system from scratch.

Second, we quantify the potential performance overhead of the modular design by comparing a join composed from several Modularis' sub-operators with a hand-tuned join. For the latter, we use the best monolithic implementations available for RDMA clusters [13, 14]. Modularis is always within 30 % of the performance of the specialized implementation (and often closer). Nevertheless, our system uses 3.8 × fewer lines of code than the monolithic operator and all its sub-operators are not specific to the join but can be reused in other query plans.

Third, we demonstrate the advantages of sub-operators over monolithic approaches when it comes to extending existing operators. We show how to use the same sub-operators that we used for the join for optimizations for sequences of joins and a distributed `GROUP BY` (with one additional sub-operator). In contrast, extending existing manually handcrafted joins [13, 14, 51] to support, e.g., inner, outer, semi, and anti joins plus grouping for partitioned, sorted, and general inputs would be very difficult (and has not been attempted, to our knowledge). In Modularis, once we had the sub-operators for the initial distributed radix hash join, we needed only a small effort to develop the rest of the operators. These results put into perspective the potential performance loss when comparing Modularis, which can run code over different platforms, with handcrafted algorithms tailored to run on a single target system.

2 RELATED WORK

Database operators design. Operator modularity is a crucial design decision as it significantly affects performance. Determining the right granularity of operators has been a reoccurring topic of research: from the bracket model [23] for parallelization in the early days of databases to objected-oriented modular designs [25], record-oriented components adapted at runtime [37], morsel-driven parallelism [49], and "deep" query optimization [24] more recently. In Modularis, we use the Volcano model [33] as the basis for the interfaces between operators, but we also support collections instead of just flat records. Modularis shares a similar vision to that of Dittrich and Nix [4, 24], Bandle and Giceva [12], and Kohn et al. [47] in having operators defined at a finer granularity. Dittrich and Nix argue that this enables a deeper level of query optimization and easy implementation of research ideas without sacrificing performance. Bandle and Giceva analyze how sub-operators can be used for general data analytics. Both these works sketch a vision for modular operator design, and they focus on a few variations of aggregation using different indexes (sorting/hashing) or on how to build more complex algorithms (e.g hash joins, k-means) out of sub-operators. In contrast, Modularis is a full system comprising a variety of operators and runs full TPC-H queries using plans tailored to three very different platforms. Kohn et al. [47] develop a framework that uses modularity to speed up query execution when the query contains multiple aggregates. Modularis follows a similar spirit but builds a more generic execution layer that achieves a similar goal for query execution in general. In contrast to these approaches, we tackle the problem at the platform level and formulate design principles for these operators. We also do not focus on the optimization aspect from the view of query rewriting. Our goal is to reduce the implementation effort and keep up with the fast pace with which the hardware evolves, without having to rewrite systems completely. Finally, efforts seeking to optimize sets of operators are orthogonal to Modularis as they could be applied to the query plan generated by Modularis as additional optimization passes. For instance, Leeka et al. [48] fuse similar operators into a single *super-operator* by using a streaming interface.

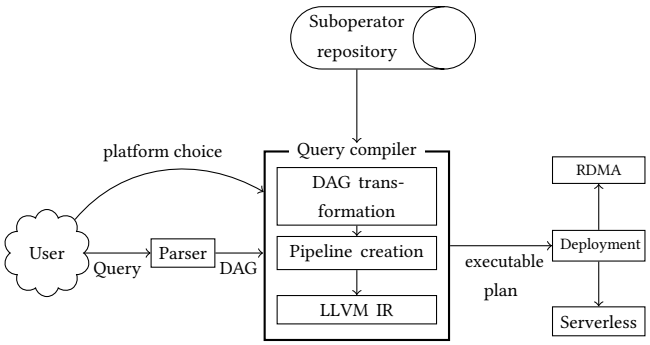


Figure 1: System architecture

Emerging technologies in data processing. Two of the most recent emerging technologies in distributed data processing are RDMA and serverless computing. On the one hand, RDMA has been used to implement highly distributed relational operators [13, 14, 51] and several projects have explored the network bottleneck and the use of RDMA for query processing and database design in a variety of contexts [15, 16, 50, 60, 61]. On the other hand, serverless computing has been extensively studied over the last few years for a variety of applications [10, 19, 30, 39, 43–45, 56, 59, 62, 64]. In both cases, research focused on solving platform-specific challenges and was often carried out in built-from-scratch prototypes that were limited in functionality. In Modularis, by leveraging the granularity of the sub-operators, we develop specialized platform-specific operators that leverage the latest technologies of both platforms but keep most of the other operators and the rest of the system hardware-agnostic. This shows that we can achieve competitive performance without having to redesign the entire system.

Query compilation for data processing. Modularis is also related in part to systems that perform Just-in-Time (JiT) query compilation as pioneered in HyPer [54]. Similar techniques were later used and extended in several other systems: Tupleware [21], which targets machine learning workloads; Weld [55], which incorporates a large class of data analytics algorithms by translating frontend languages into an intermediate representation (IR) that is later Just-in-Time compiled; LegoBase [46], which builds a database using a high-level language; and Flare [27, 28], which combines data processing tasks with machine learning. Modularis uses JiT compilation techniques similar to those in such systems to eliminate the potential performance overhead of a modular design. The systems above either have an IR tailored towards single-machine execution and rely on systems such as Spark [69] for their distributed setup (e.g. Weld) or, if they have a distributed setting (e.g. Tupleware), they use a very generic model that is very hard to be adapted in case of platform changes (e.g. from VMs to serverless functions). Modularis on the other hand, targets directly distributed analytics and focuses on tailoring execution to the underlying platforms by only using minimal changes to the plans.

3 MODULAR OPERATORS

3.1 Modularis architecture

In this section, we describe the architecture of Modularis and how it executes a query. We show the system architecture in Figure 1. The user writes queries in a UDF-based library interface written in Python (similar to PySpark). When the user submits the query, she specifies the target execution platform using a flag (e.g. ,`--rdma`, `--lambda`). The query is then parsed and translated into an IR, representing a DAG of operators. The DAG is serialized and passed to Modularis' backend, written in C++, which applies a series of both plan-specific and platform-specific transformations.

The plan-specific transformations involve projection and selection push-downs, operations typical in DBMSs to reduce data movement and I/O. Next, Modularis cuts the DAG into tree-shaped sub-plans, each of which represents a pipeline with a materialization point at its end. Then, we transform the query into its distributed equivalent. This step involves wrapping the initial plan into a distributed executor and adding exchange operators in the plan inputs. Depending on the target platform, we specialize the generic operators with hardware-specific ones. In the case of serverless, we use the exchange operator of Lambada [52] and an executor that spawns the workers in a tree-plan fashion. In the case of smart storage, we use a specialized operator to get data from S3Select. In the case of RDMA, we use an MPI executor and an RDMA-based exchange operator based on the one of Barthels et. al [14] (with modifications to avoid unnecessary data movement). We give concrete examples of the final plan in Sections 4.4 and 4.5. Finally, the plan is lowered into LLVM IR and Just-in-Time compiled to native machine code. To translate UDFs into LLVM IR, Modularis uses Numba [5] and inlines the generated LLVM code into the remainder of the plan to eliminate any function calls or interpretation in inner loops. The query is then executed on the target platform, based on the specialized operator that it has been wrapped around. When the results are produced, they are returned back to the user.

3.2 Design principles

Modularis builds on the observation that the commonly used operators, e.g., for high-speed networks [13] or multi-core CPUs [11], are built around the same *conceptual* building blocks: when the authors describe their algorithms, they use some visual or textual representation of "reading data", "partition by key", "for each partition", etc. To readers, these terms imply the same operation is being done during different phases of query execution. However, there is no code reuse—the implementations differ in intricacies related to how and where data is stored, how it is passed from one phase to the next, how they depend on the state of some enclosing scope, etc.

The goal of Modularis is to identify pieces of code that reoccur in operators in slight variations, factor out their common logic, and package them behind a well-defined interface such that they can be reused and recomposed. In other words, our goal is to derive *actual* building blocks from the conceptual ones. To derive sub-operators systematically, we follow these design principles:

(1) *Each sub-operator consists of or is a part of at most one inner loop.* If a high-level algorithm consists of several phases, each of these phases is expressed by at least one sub-operator. Phases often reoccur across and within monolithic operators, sometimes in slight variations. For example, join operators typically use partitioning to improve cache locality. In Section 4.3, we show that by factoring out the partitioning logic into a dedicated sub-operator, we can reuse it to improve cache locality in grouping operators as well.

(2) *Use dedicated sub-operators for each physical (in-memory) data materialization format.* This decouples the processing of data

from where and how it is stored. Consequently, other sub-operators become independent of the physical formats of their inputs and outputs and are more generic. One high-level example is to have different scan sub-operators for reading base tables or intermediate materializations in RDMA buffers. That way, a single partitioning sub-operator implementation can consume inputs of two different scan operators (or any other operator) instead of having two specialized partitioning operators (see Section 4.1.2).

(3) *Express high-level control flow as (nested) operators.* This allows connecting plan fragments of sub-operators that express the heavy-lifting data processing through the same operator interface. In monolithic operators, such orchestration logic is usually implemented as imperative code specific to that operator and, thus, it makes it necessary to reimplement the data path as imperative code as well. One high-level example is to express the in-memory join of two partition pairs occurring in a classical partitioned hash join as a nested query plan that is executed for each pair of matching partitions. That allows the use of partition-unaware sub-operators in the inner plan. We introduce the `NestedMap` sub-operator for this purpose below (Section 4.1.2).

Generality of design principles. The above somewhat different implementations do not only apply to the context of CPUs, but also to other hardware architectures (e.g. FPGAs, GPUs). Therefore, the goal of the design principles is still applicable, as we can use them to derive the basic building blocks that the underlying architectures use. By following our design principles, we do not expect to design operators that work solely in different architectures without any modifications. We rather want to avoid reimplementing the same conceptual building blocks with small modifications. Our goal is to end up with a very similar set of operators for different architectures. For example, an operator that reads data in columnar format is relevant to both CPUs and FPGAs. By having a similar set of building blocks for different architectures, we can offload parts of a query to different hardware configurations seamlessly, just by swapping out the hardware-specific part of the plan. We give a concrete example of how this can be done in the context of smart storage in Section 4.5.

3.3 The sub-operator interface

We base the interface of sub-operators on one of the most known models in the database community, the Volcano model [33]. The model is based on iterators that pass records along the data path of a tree of `Next()` function calls. Like in a traditional execution engine, the iterator interface allows us to combine operators in almost arbitrary ways, limited only by the schema or types that operators may require.

The main distinctive feature of the sub-operator interface compared to traditional Volcano-style operators is the type system of the records (or "tuples") passed between them. While records in relations (in the First Normal Form) consist of atomic fields, we need a more expressive type system for a generic physical execution layer. For example, to split the materialize and scan operators of a given physical data format into two distinct sub-operators, these operators need to pass "records" containing the materialization from one to the other. Similarly, if we want to express operators that work on individual records as well as those that work on batches (or morsels [49]) in the same interface, we need to be able to represent the concept of a "batch". We thus extend the concept of tuples with that of "collections", which is the generalization of any physical data format of tuples of a particular type. This allows expressing physical execution properties into the query plan rather than hard-coding them for the entire execution engine.

More formally, sub-operators are iterators over *tuples* and the tuples are of a statically known type from the following recursive type structure:

$$tuple := \langle item, \ldots, item \rangle$$
$$item := \{\, atom \mid collection \text{ of } tuples \,\},$$

where a *tuple* is a mapping from a domain of (static) field identifiers to item types, an *item* is a (statically known) atomic or collection-based type, an *atom* is a particular domain of undividable values, and a *collection* is the generalization of any physical data format one might want to use in the execution layer. We denote tuple types by $\langle fieldName0 : ItemType0, \ldots, fieldNameK : ItemTypeK \rangle$ and collection types by $CollectionType\langle TupleType \rangle$. Most operators are *generic* in the sense that they require their upstream operator(s) to produce tuples of a type with a particular structure but accept any type of that structure. Their output type usually depends on the type(s) of their upstream(s). For example, the scan operator for a C-array of C-structs (which we call *RowVector*) requires from their upstreams to produce tuples of type $RowVector\langle TupleType \rangle$ and returns tuples of *TupleType*, where *TupleType* is allowed to be any tuple type. Similarly, operators consuming or producing batches can do that by consuming or producing tuples with *RowVector* fields.

We also extend the Volcano-style execution model to DAGs, whereas operators in the original Volcano-style execution model could only have one consumer (i.e., plans had to be trees). Before execution, we cut a DAG of operators into pipelines, where pipelines start with either the original plan inputs or the result of any operator with several consumers. In each pipeline, that result will be read only once, so the sub-plan of the pipeline is a tree and can thus be executed with the iterator model. Pipelines materialize their results, such that multiple downstream pipelines can read them. For simplicity, we present the plans as DAGs in the remainder of the paper and omit the pipelines and materialization points.

3.4 Initial set of sub-operators

In this section, we introduce the initial set of sub-operators that we use in Modularis. We choose the sub-operators such that they are expressive enough to support all the major relational operations, e.g. selections, projections, aggregations, and joins. At the same time, we follow our design principles. While the design principle of making operators simple aims indeed at increasing code re-usage and reducing implementation effort, this does not necessarily mean that there is no redundancy. In fact, we do add new operators if they provide a sufficiently large and broad performance benefit. Such an example is the different join implementations (inner, semi, anti). They could all be based on the same hash-build and a separate probe operator. However, having a special operator for each of them provides better performance. Similarly, having a second version for each of them with flipped build and probe sides increases performance further. Finally, while the initial set of sub-operators is enough for relational analytics, we expect that, as we expand

Modularis to more types of analytics (e.g. linear algebra, ML), we need to expand our sub-operator set.

We present our initial list of sub-operators in Table 1. The sub-operators fall into six categories: orchestration operators, data processing operators, MPI-specific operators (to support RDMA clusters), Lambda-specific operators (to support serverless), Smart storage-specific operators, and generic materialize and scan operators. Orchestration operators enable the execution of nested computations. Data processing operators express the computations carried out on the data inside the inner loops. MPI- and Lambda-specific operators are the ones that are aware of the distributed nature of query execution. Smart storage-specific operators use pushdown computations to smart storage. Finally, materialize and scan operators read and write *tuples* from and to nested *collections*. We give an overview of our operators in Table 1. We believe that the semantics of most operators are clear given their names. However, the two orchestration operators merit an explanation, as they differ substantially from other systems.

The `ParameterLookup` operator encapsulates plan inputs in the operator interface, such that other operators can consume them. This operator is the only operator aware of plan inputs. The `NestedMap` operator executes a nested plan independently on each input tuple, which typically contains a nested collection. Each invocation of the nested plan produces an output tuple that may contain nested collections as well. This allows us to process nested collections using the same building blocks regardless of the nesting level. This operator consumes tuples of any type, and the `ParameterLookup` operator(s) in the nested plan return a tuple of that type.

4 FROM OPERATORS TO COMPLEX QUERY PLANS

4.1 High-performance distributed join

We illustrate how Modularis' sub-operators can express optimized monolithic operators with a case study of a state-of-the-art distributed join algorithm proposed by Barthels et al. [14].

Category	Operators
Orchestration operators	Parameter Lookup, NestedMap
Data processing operators	(Parametrized) Map, Projection, Cartesian Product, Filter, Reduce (By Key), GroupBy, Zip, Local Histogram, Build and Probe, Partition, Semi-join, Sort, Top-K
MPI-specific operators	MPI Executor, MPI Histogram, MPI Exchange
Lambda-specific operators	Lambda Executor, Lambda Exchange
Smart storage-specific operators	S3Select Scan
Materialize and scan operators	Local Partitioning (AVX-based), Partition, Row Scan, Column Scan, Parquet Scan, Materialize Row Vector, Arrow table to collection

Table 1: Initial set of sub-operators

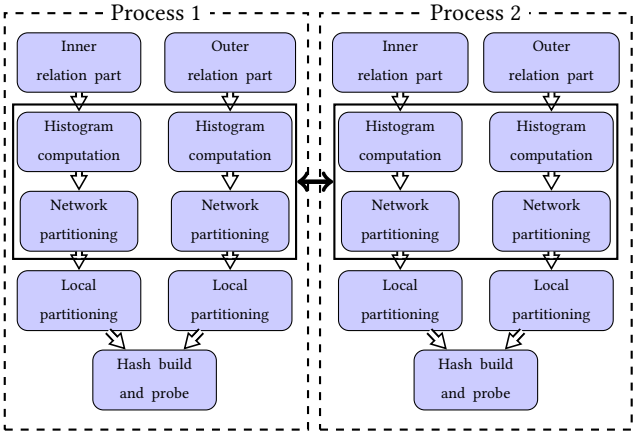


Figure 2: RDMA-aware hash join algorithm for two processes proposed by [14]

4.1.1 State-of-the-art distributed join. We start with a very brief summary of the algorithm as it was originally proposed (Figure 2). The algorithm consists of three phases: (1) histogram computation, (2) multi-pass partitioning including network transfer, and (3) hash table build and probe. The two phases where communication happens amongst processes, namely the histogram calculation and the network partitioning phase, are depicted using black boxes around them. The original algorithm is optimized for a workload involving two relations where both relations consist of 16-byte tuples (8 bytes for the key and another 8 for the payload). For more details, we refer the readers to the original paper [14].

4.1.2 Query plan in Modularis. We now show how the same join algorithm can be expressed using sub-operators in our RDMA backend. The resulting query plan is shown in Figure 3. To keep the graphical representation concise, we abbreviate the operator names as shown in Table 2. Furthermore, we omit materialization points and, instead, express the plan as a DAG as discussed previously. Finally, most of the operators in the Figure are part of a nested plan inside a `MpiExecutor` operator, which is executed concurrently by all MPI processes (which we call *ranks* in the rest of the section) in the cluster in a data-parallel way, illustrated with a stacked frame.

The join starts by computing the histogram of each of the two inputs using the `LocalHistogram` operator. The inputs can be produced by any operator producing tuples with key fields, e.g., a scan operator reading from a base table stored in main memory. On each of the two sides, a `MpiHistogram` computes the global histogram from the local ones until the `MpiExchange` consumes the local histograms, the global histogram, and the original input. With this information, each rank allocates a contiguous memory area in the main memory of the host (called *RMA window*) that will hold all tuples it will receive in this phase and computes the offsets of its exclusive regions inside the windows of the target ranks. Then, each rank reads the input again, computes the target partition for each input tuple the same way as it did for computing the histogram, and writes the tuple into a buffer corresponding to that partition. This partitioning routine is based on well-known techniques using streaming stores and software write-combining [53, 58, 63]

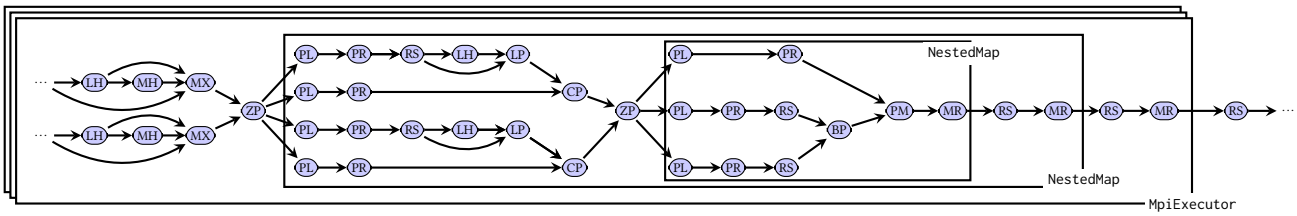


Figure 3: Plan that runs the distributed hash join with modular operators across many nodes

to achieve the full memory bandwidth. When the buffer of a partition is full, it is sent to the target rank using an asynchronous RDMA write operation, it replaces the buffer with an empty one, and continues partitioning the input immediately. This overlaps computation with communication and increases performance.

In the network partitioning phase, as in the original algorithm, each rank compresses the 16-byte workload of the algorithm into 8 bytes to reduce the data transmitted by a factor of two. This optimization comes as an additional pass to our query compiler, and although it is very specific, it is useful for dictionary-encoded data. The compression uses the fact that some bits of the key are common for each partition. Specifically, if we use the identity hash function and radix partitioning with a fan-out of 2^F, the first F bits of each partition are identical. Furthermore, we assume that keys and values come from a dense domain and can be represented with P bits each. Thus, key and value can be stored in a single 64-bit word if $2 \cdot P - F \leq 64$. After the partitioning and compression, the operator returns the partitions as ⟨*networkPartitionID*, *partitionData*⟩ pairs such that all tuples of each partition end up on only one rank. Because we extend the original algorithm with materialization of the input tuples, we recover the missing bits by forwarding the *networkPartitionID* further downstream. As we can observe, the `MPIExchange` operator batches the tuples in order to avoid the overhead of sending a tuple-at-a-time over the network.

The subsequent plan joins the tuples inside two corresponding partitions of the two sides. An imperative implementation would express this as a loop over matching partition pairs. In Modularis,

Abbreviation	Operator name	SLOC
PL	Parameter lookup	28
NM	Nested map	49
PR	Projection	27
BP	Hash build and probe	103
LH	Local histogram	77
ZP	Zip	44
CP	Cartesian product	54
PM	Parametrized map	51
RK	Reduce by key	75
RS	Row Scan	59
LP	Local partitioning	143
MR	Materialize row vector	56
ME	MPI Executor	140
MX	MPI Exchange	269
MH	MPI Histogram	52

Table 2: Source line of code per operator

we use the `NestedMap` operator for the same purpose: We take the corresponding ⟨*networkPartitionID*, *partitionData*⟩ pairs and pass them through a `Zip` operator, which produces ⟨*networkPartitionID*, *partitionData*, *networkPartitionID*, *partitionData*⟩ tuples (note that they are produced in dense, ordered sequence). This way, all data belonging to one partition pair is represented in a single tuple, and we can express the remaining logic as a nested plan transforming each such tuple. The nested plan starts by dissecting the input tuple. The tuple has four fields: the partition ID and data of the two sides, respectively. A sequence of `ParameterLookup` (which returns the entire tuple) and `Projection` operators (which retains one of the fields) extracts one of the fields, each. The partition data is partitioned further on both sides by a sequence of `RowScan` (which extracts individual tuples from the nested collection inside the *partitionData* fields), `LocalHistogram`, and `LocalPartitioning` operators. Note that each of these sequences returns several ⟨*localPartitionID*, *partitionData*⟩ pairs. To be able to recover the dropped bits further downstream, we augment each of these pairs with the *networkPartitionID* by using the `CartesianProduct` operator. Its left side only consists of a single tuple (containing the network partition ID), so it does not increase the number of tuples.

The hash and probe phase happens inside another nested plan, which is executed for each pair of sub-partitions. As before, we use a `Zip` operator to combine all information of each pair of partitions into a single tuple, on which we call a nested plan using `NestedMap`. We use sequences of `ParameterLookup` and `Projection` operators to extract the partitions of the two sides. Each partition is read by a `RowScan` operator and individual tuples produced by these two are finally fed into the `BuildProbe` operator, which produces the matching pairs. To recover the dropped bits from the network phase, we use a `ParametrizedMap` operator: It contains a function that, given a parameter from upstream (the network partition ID), shifts that parameter by a certain amount and adds the result to the key field of each input tuple from the other upstream.

The remainder of the plan depends on what happens with the join output. The Figure shows a plan that materializes that result. Since each `NestedMap` needs to return a single tuple, the result of each nested plan needs to be materialized using a `MaterializeRowVector` operator. This operator produces a single tuple containing its input tuples as a nested *RowVector*. Each `NestedMap` thus returns several such tuples (one for each input tuples) and the inner tuples can be recovered as a flat stream using `RowScan` operators.

In the above description, we reuse many of our building blocks to construct the high-performance distributed join. Also, we showcase many of our design principles in action, such as the dedicated read/write operators to read data either from RMA windows or local

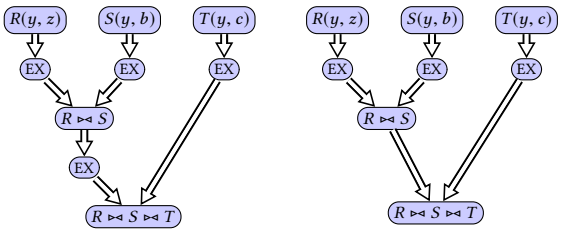


Figure 4: Naive (left) and optimized (right) versions for a sequence of two joins on the same attribute

partitions, how the high-level control functions work, and finally how we reuse operators across different phases of the algorithm.

4.2 Sequences of joins

A key advantage of Modularis is that, once we have the original join algorithm, it is straightforward to extend the plan to run sequences of joins. None of the work to date on high-performance joins on multi-core CPUs or over RDMA has ever addressed this design due to the complexity of modifying the highly tuned operators. For a cascade of N joins, the output of the $n-1$-th join is joined with the n-th relation, where $n = 1, \ldots, N$. Therefore, in the original plan of Figure 3, after the `RowScan` operator, we return the new data to the `LocalHistogram` and `MpiExchange` operators. On the other side, another upstream operator returns tuples that go through the network partitioning phase. This pattern is repeated on one side of the corresponding join for each output and on the other side for the corresponding new relation, until all of the N joins are performed.

However, if all the joins are on the same attribute (i.e., the attribute y in Figure 4) and the relations fit in main memory, we can apply the following optimization: We network-partition all relations at the beginning instead of reshuffling the output of every join through the network. This is possible since we execute the output of the first join again on the attribute y, and therefore we can pre-partition all the relations from the beginning of the query instead of waiting for the result of each join . This way, for a cascade of N joins, we shuffle through the network $N+1$ instead of $2N$ relations. We show our optimization for a sequence of two joins in Figure 4, where instead of shuffling four relations through the network, namely R, S, T, and the output of $R \bowtie S$, we shuffle only R, S, and T. For conciseness, we depict with the operator `EX` the chain of a LocalHistogram, MPI Histogram, and MPI Exchange operators.

This optimization is easily applied because of the operator modularity. In the case of monolithic operators, a system engineer would have to take special care of this case by adapting a large part of the system and possibly by reimplementing parts of the algorithm. In contrast, Modularis restructures the sub-operators inside the query plans and takes advantage of the common attribute in a sequence of joins. After it performs all the network partitioning phases at the beginning of the inner plan of Figure 3, it carries out all the local partitioning phases in the first nested map. Finally, it forms a sequence of `BuildProbe` operators where the output of the $n-1$-th `BuildProbe` is the input of the n-th `BuildProbe` and the final build probe output is the input of the `ParametrizedMap` operator.

4.3 Distributed `GROUP BY`

To illustrate how Modularis simplifies operator development and provides extensibility, we implement a distributed `GROUP BY` operator by re-using components from the previous use cases. We show the corresponding plan in Figure 5. The algorithm workload is a 16-byte tuple (8 bytes for the key and 8 bytes for the value).

The plan starts with any upstream operator that returns tuples to the `LocalHistogram` and `MpiExchange` operators. Since we have multiple consumers from one operator, these tuples have to be materialized and put into a separate pipeline (the materialization is not shown in the Figure as discussed earlier). After the local and global histogram calculations, the tuples are partitioned and distributed through the network. The operator performs a similar compression scheme as in Section 4.1.2. As before, this compression allows us to reduce the network traffic in half, which is crucial for performance. Every output tuple (which consists of $\langle$*networkPartitionID*, *partitionData*$\rangle$ pairs) coming from the `MpiExchange` operator is the input of a `NestedMap`, which executes its nested plan for every input partition.

The execution of the nested plan starts with the `ParameterLookup` operators. The tuples returned by these operators are passed to `Projection` operators, which ensure that each downstream operator gets the correct input. Specifically, the network partitioning data are passed to a `RowScan` operator that returns a tuple at a time to the `LocalHistogram` and `LocalPartitioning` operators. After the histogram calculation, the `LocalPartitioning` consumes both the input data and the calculated histogram to calculate the necessary prefixes inside a partition. It then performs the data partitioning. The corresponding partitioned data is concatenated with the network bits that the `MpiExchange` removed, using a `CartesianProduct`. Lastly, the $\langle$*networkPartitionID*, *localPartitionID*, *partitionData*$\rangle$ triples are the input to `NestedMap` operator, which executes the final aggregation for each input partition.

To perform the final aggregation, we first have to restore the original keys as we did in the join algorithm after the hash build and probe phase. The difference now is that we have to restore the full keys using the `ParametrizedMap` operator before we forward each tuple to a `ReduceByKey` operator, which aggregates the data per local partition. Afterward, we materialize the output tuples of the `ReduceByKey` operator with a `MaterializeRowVector` operator. Like in the distributed hash join case, we finish the plan with

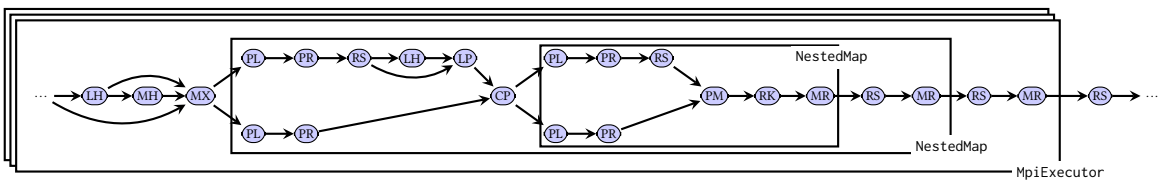


Figure 5: Plan that runs the distributed `GROUP BY` with modular operators across many nodes

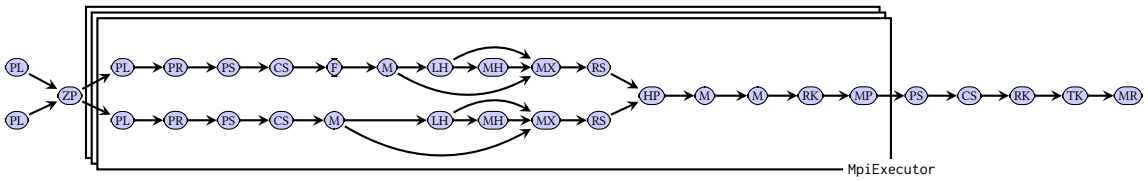


Figure 6: Modularis plan for TPC-H Q12 on RDMA

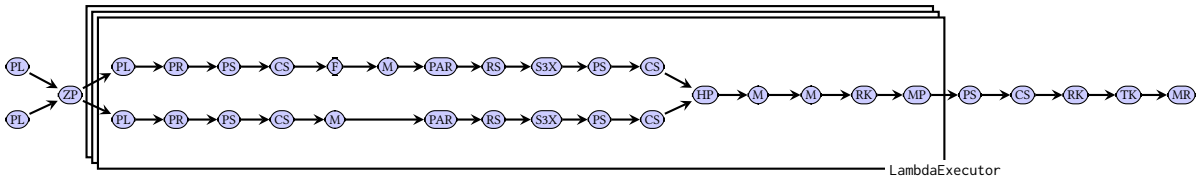


Figure 7: Modularis plan for TPC-H Q12 on Serverless

the `RowScan` operators that remove nesting levels of the `MaterializeRowVector` operators that should end every nested plan. Finally, the individual results from the workers return to the driver.

Based on the previous description, it is evident that the distributed `GROUP BY` plan is very similar to the distributed hash join plan. The main differences are 1) the total number of input relations and 2) that for distributed `GROUP BY` operator we do not perform a hash build and probe phase in the end but an aggregation using a `ReduceByKey` operator. The large overlap between the two plans shows how Modularis uses a similar set of sub-operators to implement different relational database operators in contrast to using monolithic operators, where the different operators would probably have to be reimplemented almost from scratch, although the logic behind shares many similarities (e.g., the partitioning phases).

4.4 TPC-H queries

So far we have used Modularis to implement key components of a relational query processor. We can use the same sub-operators to implement TPC-H queries. In fact, by altering only our MPI-specific sub-operators to Lambda-specific we can execute the same set of TPC-H queries on a different hardware platform. The latter is a strong argument towards modularity, as systems that operate on different hardware platforms have fundamentally different execution layers making it almost impossible to apply the techniques used in one for the other without major reimplementation.

We take as an example TPC-H Q12 and show (simplified) query plans of Modularis in Figures 6 and 7 for RDMA and serverless respectively. Although the plans are simplified, they still preserve the semantics of the system. Since the network bandwidth in serverless is rather slow (around 80 Mbit s^{-1}, see [52]), we use the partitioning only as a pre-processing step required by the exchange operator and do not partition the exchanged data further. This means that each worker processes only one data partition after the exchange, so we do not have any `NestedMap` operators in that platform. Both plans take Parquet file paths to the base tables as input, either in S3 or NFS depending on the platform. The paths of the left and right input are zipped and each resulting pair is used as the input for a nested plan instance by the executor of the respective platform. The execution of a nested plan starts again with `ParameterLookup` operators, from which we project the paths for the left and right relations, pass them to `ParquetScan` operators, and finally extract individual tuples from the column chunks produced by that operator using the `ColumnScan` operator. The next operators express the corresponding operations of Query 12.

At this point, we differentiate the plans with the operators that constitute the exchange routine for each platform. Note that we do not perform any compression of the tuples, such as in the case of the distributed join and the `GROUP BY`. In the case of RDMA, the plan looks similar to the ones that we have presented before. In the case of serverless, we use the exchange algorithm of Lambada [52]: First, we partition the data into a sequence of partitions using a `Partition` operator. Subsequently, the `GroupBy` operator takes the `<pid, data>` partitions and groups them by `pid`. Then the `RowScan` operator reads each partition and forwards it to the `S3Exchange` operator, which writes the data into a file on S3 whose file name is based on the ID of the sender containing one row group per receiver. The `S3Exchange` then returns triples of worker-specific S3 paths and the first and last row group to read from that file, which is then read by the subsequent `ParquetScan` operator. This pattern implements the "write combining" optimization of Lambada, which significantly reduces the number of write requests to S3. Finally, the column chunks produced by the `ParquetScan` operator are consumed by a `ColumnScan`, which extracts individual tuples.

After the exchange phase finishes, the plans converge again: We perform the join of the two relations using the `HashProbe` operator. The matched tuples are transformed with a series of `Map` and `ReduceByKey` operators to get the final result, and we use a `MaterializeParquet` operator to return the individual results of the workers to the driver process. We read the individual results using a sequence of `ParquetScan` and `ColumnScan` operators. Finally, we merge the results of the workers using a `ReduceByKey` operator, we select the first tuple using a `TopK` operator and we return the results to the user with a `MaterializeRowVector` operator.

We therefore show how altering only a handful of operators that are hardware-specific, allows us to run the same TPC-H query on two very different hardware platforms. That way, implementation effort is reduced vastly. Using the same ideas, we could extend the TPC-H implementation to use an exchange operator based on TCP. The addition of more backends only requires changing the executor and the operators that comprise the network exchange phase.

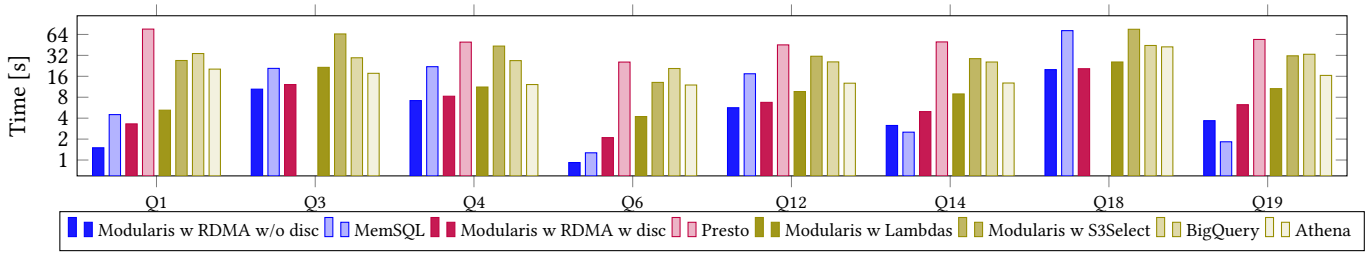


Figure 8: TPC-H queries runtime using SF-500

4.5 Integration with smart Storage

To show how Modularis can use a smart storage component offered by a major cloud provider, we integrate S3Select [6] into our system. S3Select is a smart storage engine offered by Amazon that follows the trend of pushing computation into storage [38, 67] to overcome the I/O bottleneck. S3Select pushes computations directly into S3 and thus it pulls only the data that the user needs from these objects.

S3Select takes as parameters an SQL query, the S3 input path, and an output serialization format (either JSON or CSV). In our case, the user writes an SQL expression in the frontend, which pushes selections and projections to S3Select. We have created an additional sub-operator called `S3SelectScan` that our query compiler decomposes into three simpler, more re-usable separate operators. The first sub-operator performs an API call to S3Select and requests the data, which S3Select returns in CSV format. The sub-operator then uses Apache Arrow to convert the CSV to an Arrow Table and forwards the table downstream. The next operator converts the Arrow Table to a *collection of tuples* as this is defined in Section 3.3. Finally, the third operator is a `ColumnScan` that gets this collection and returns the individual tuples. Then, we continue with the execution of the rest of the plan in our serverless backend.

Thus, by following the same design principles, we only develop the sub-operators needed to integrate S3Select into Modularis. We did not have to redesign the system from scratch to adapt to this system component. We only had to make small adjustments, whereas other systems would have to make a major redesign to incorporate such a change. Finally, this integration shows the generality of our design principles, because the implementation done is not limited to S3Select. We could use the same architecture to request data from a different smart storage engine that is based on an accelerator (like Amazon AQUA [1]). As databases push more computation to storage or to accelerators, we expect that systems like Modularis will be able to use these components with only minimal changes.

5 MODULARIS EVALUATION

In this section, we first compare Modularis to commercial systems on TPC-H queries on two hardware platforms: RDMA and serverless. We then analyze the performance of the system against a distributed hash join. Finally, we show the runtime of a distributed `GROUP BY` operator and variations of plans for sequences of joins. We run all of the RDMA experiments using all available cores from a cluster of 8 machines (specifications in Table 3). Regarding the MPI implementation, we use OpenMPI 3.1.4 as opposed to foMPI [31] used by [14] because foMPI is specific to Cray machines. Finally, unless otherwise mentioned, we run each experiment five times and report the average among such runs.

5.1 TPC-H queries

We start our evaluation by comparing Modularis to commercial systems, both for RDMA and serverless using TPC-H queries. We present the results for scale factor 500 in Figure 8. We pick TPC-H Queries 1, 3, 4, 6, 12, 14, 18, and 19, which is more than a third of the benchmark and a representative subset of all the challenges that the benchmark poses. More specifically, Q1 has a large aggregation, Q3 has a large join, Q4 involves an inclusion test, Q6, Q14, and Q19 have selective filtering that largely reduces the processed tuples of the input tables, Q12 involves almost all of the major operators a relational engine should implement (join, selection, projection, aggregation, sorting), and Q18 has a high-cardinality aggregation. As mentioned, the challenges involved in these queries together make up for the most important challenges of the benchmark itself and an execution layer that has a good performance on these queries has a high probability to perform well on the whole benchmark [18, 41]. The inclusion of more queries is not a limitation of the system but involves the implementation of a more sophisticated optimizer, which is part of future work and out of the scope of this paper; for now, we concentrate on the execution layer alone.

5.1.1 RDMA. For the RDMA backend, we compare Modularis against two different systems, Presto and SingleStore (previously MemSQL). Both of these systems do not use RDMA for the network exchange but we configure them to use the InfiniBand network for data transfer at higher rates. We verify this by monitoring the network traffic. For all the RDMA experiments, we run Modularis using 64 workers. For SingleStore, we use version 7.0.12 and deploy a master aggregator node and 7 leaf nodes. We do a warm run for each of the queries and report averages of 5 runs. We deploy the TPC-H database using both a row-store [8] and a column-store [7] format. The row-store format is completely in-memory and has very good random seek performance. The disk-backed column-store format has optimizations such as indices, compression, fast

Component	Specs
CPUs	2 × Intel Xeon E5–2609 2.40 GHz
Cores/Threads	2 × 4/4
RAM	128 GB
L1 Cache	2 × 4 × 64 KB
L2 Cache	2 × 4 × 256 KB
L3 Cache	2 × 10 MB
InfiniBand	Mellanox QDR HCA

Table 3: RDMA cluster specification

aggregations, and table scans, most of which are not supported by Modularis. We use the column-store format, as it is the best for all the queries. Although the column-store format is disk-backed, SingleStore serves all queries from an in-memory cache; for a fair comparison, we thus exclude the time Modularis needs to read the data from disk as well. We observe that Modularis is between 30% up to more than 3x faster for all the queries except Q14 and Q19.

For Q19, Modularis is slower because the histogram-based RDMA network exchange is slower than a broadcast operator for small joins. Modularis is 40% slower in Q14, because the `Map` operator does not perform the selective `LIKE` as fast as SingleStore. For the queries where Modularis is faster, all except Queries 1 and 6 have large joins, where the RDMA network exchange has better performance. Another operator responsible for a large part of the overall runtime is `ReduceByKey`, which uses a highly optimized version of a parallel hash map in Modularis and thus executes very fast large aggregations. The difference is obvious in Queries 1 and 18 where Modularis is 3 and 3.5 times faster, respectively.

For Presto, we deploy Presto SQL (now Trino) version 327 along with HDFS (Hadoop version 2.6.0) in the eight-machine cluster using one node exclusively as coordinator and NameNode. We configure HDFS to use replication factor 3 and Presto to use as much memory as possible. We run each query four times, use the first run as a warm-up and then report the average of the other runs. To have a fair comparison, we include also the time that Modularis needs to read the input data from disk. For Queries 3 and 18, Presto cannot execute the queries and fails because of insufficient resources. Our system is 6-9x faster than Presto, depending on the query, partly because of the optimized RDMA network exchange and partly because of the highly-performance sub-operators.

5.1.2 Serverless. For the serverless backend, we use the two Query-as-a-Service systems Athena and BigQuery as baselines. These systems run queries over cloud storage without the need to start or maintain any infrastructure. For BigQuery, we use external tables [3] that point to 512 Parquet files per relation stored on a single-region bucket in Google Cloud Storage in the standard storage tier. Creating such a table is just a metadata operation and takes <1s, i.e., no data is loaded. Instead, the query runs against the original files on cloud storage. We use the same files for Athena, for which we created an external table [2] as well. The remaining configuration is done by the cloud provider. For Modularis we use the same Parquet files. For the integration with S3Select, we use 512 workers and for the experiments with the `ParquetScanOperator` we use 256 workers. The workers for both these configurations have 2 GiB of main memory each. For all three systems we run each query 6 times and report averages. As we observe, Modularis with S3Select is comparable to BigQuery for the majority of the queries but slower than Athena. To investigate the reason, we isolate the calls to S3Select and time them independently of Modularis. We find out that they make up for the majority of the runtime (e.g. for Q1, 24 out of 27 seconds are spent getting data from S3Select). The problem is enlarged when we read many relations (e.g. Q3, Q18) and is almost negligible for high selective queries that read only one relation (e.g. Q6). In fact, for Q6 Modularis with S3Select is very close to Athena and faster than BigQuery. The larger running times of calls to S3Select are because the service returns chunks of uncompressed CSV data, whereas our `ParquetScan` reads data in compressed format and also pushes down projections. Our observations agree with the ones made in [68]. This problem is ameliorated if we read only one Parquet file per worker, which is why we use 512 workers in this experiment.

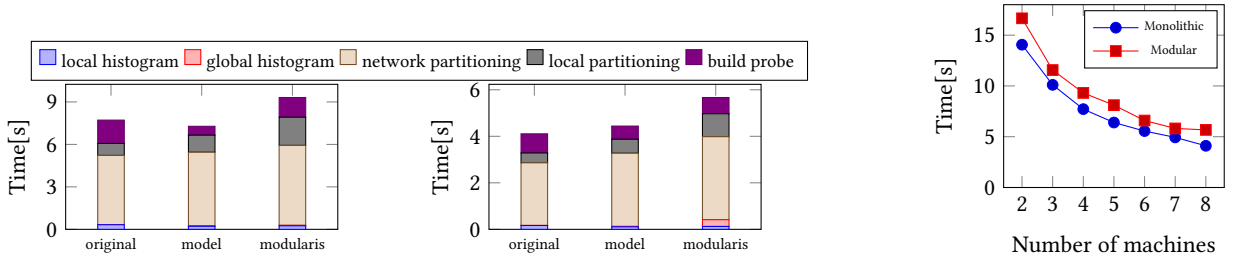
We can improve the performance significantly by using the specialized `ParquetScan` operator. In that case, Modularis outperforms both baselines in all queries except for Query 3, where Athena is marginally faster thanks to a better query plan. For Queries 1 and 6, the running time difference is due to the `ParquetScan` operator, which is optimized to push down projections while reading data in compressed format. For the other queries, the specialized `LambdaExchange` operator can run workloads involving data exchange between workers faster than the commercial baselines. This is evident because the workers are mostly bounded by network bandwidth and latency. The current performance of S3Select illustrates the advantage of modularity: today, we can use the much faster alternative of our optimized `ParquetScan`, but using `S3SelectScan` can be enabled easily if AWS improves the performance of their smart storage engine in the future.

5.2 State-of-the-art distributed join

5.2.1 Implementation effort comparison. Before we delve into a performance comparison of the execution of the distributed hash join algorithm between Modularis and the original codebase, we measure the implementation effort between the two approaches by using the number of lines of code of each of them. Although this metric is not always reliable, most of the time it gives a good indication of the implementation effort. The operators that are used in the plan according to Table 2 sum up to 1152 lines of code while the original implementation adds to 1754 lines of code, leading to a 35% reduction. One can argue that this can be attributed to coding style but the main take-away from this comparison is not only the size reduction but the extensibility of our sub-operators. While to support other join types (e.g. semi-joins, anti-joins) we only need to modify the `HashProbe` operator that consists of 103 lines, the original codebase has to be replicated for every join variant. Furthermore, the only operators that are platform-specific used are: `MpiExecutor`, `MpiHistogram`, and `MpiExchange`, which sum up to 461 lines of code. In contrast, the original monolithic code would have to be rewritten from scratch if we wanted to change the target platform, involving 3.8 × more code.

5.2.2 Performance comparison. We compare the distributed hash join code by Barthels et al. [13], which consists of a monolithic operator, against our equivalent Modularis plan. For a fair comparison, we extend the original code base with a similar materialization operation to our `MaterializeRowVector` operator. The workload consists of two relations with 2048 million tuples each. Unless otherwise mentioned, we use a 1-to-1 correspondence between the keys in the inner and outer relation. This setup is consistent with the workload used in the original paper [13] in the scale-out experiment (shown in Figure 7(a) in [13]).

We present our results in Figure 9. To understand how our system performs, we also microbenchmark our sub-operators. These microbenchmarks show the model performance that Modularis' components can achieve. We compare the total runtime of these



(a) Breakdown analysis for 4 (left) and 8 (right) machines **(b) Comparison across machines**

Figure 9: Modularis distributed join execution time per phase and compared to Monolithic design

microbenchmarks (referred to as *model*) against the entire Modularis query plan and the original code base. We present our results for two machine configurations in Figure 9a. We start our analysis by comparing the three execution times phase-by-phase.

Starting with the *local histogram phase*, we observe that, compared to the original code, both the model and the whole query plan have a small speedup. We attribute this speedup to the fact that, as we mention in Section 4.1.2, the local histogram calculation is isolated in a small pipeline because its input has to be consumed by multiple readers. This allows for compiler optimizations (e.g. automatic usage of SIMD instructions, function inlining) that remove our sub-operators abstractions. These optimizations are not possible in larger pipelines, because in larger pipelines the compiler cannot inline all the `next()` functions effectively.

The *global histogram phase* has almost the same execution time in our model and the original code. However, in the full query plan and especially when the join is executed on more machines, the total time is significantly larger. This is associated with the `MPI_Allreduce` function that calculates the histogram, which is a collective operation that requires data from all the processes. In case a process is stalled in a previous phase of the algorithm, then every other process must wait until it has the required data from it. In the original algorithm, this phenomenon is not present because the histograms are calculated sequentially for both relations. This also holds for the model. On the other hand, during the execution of the join in Modularis, the global histogram calculation happens in two distinct phases, one for each upstream path and a network partitioning phase is between the two global histogram phases. Because the network partitioning phase has a slight variation in its execution time, it causes tail latencies for some processes during the calculation of the global histogram of the second relation.

The *network partitioning phase* is slower in the model and the query plan than the original code base for two reasons. First, this operator is part of a large pipeline in our generated code and therefore, as mentioned before, the compiler cannot perform all the possible optimizations and remove all of our abstractions. To validate this assumption and find the cause of the slowdown, we run the following benchmark: We generate 1 billion integers and record the time that `RowScan` needs to read them and compute their sum, compared to a simple C++ program that does the same. `RowScan` needs about 1 second, whereas the C++ program needs around 0.8 seconds. The second reason for the slowdown is due to tail latencies because, as before, the window allocation/synchronization function calls are collective operations. In the original code base, they are called almost at the same time for both relations but, in Modularis, they happen at different times, one for each upstream path.

The *local partitioning phase* is faster in the original code base than in the model and the query plan. Part of the slowdown can be again explained due to the complex pipelines that this phase belongs to, which causes a comparative slowdown in the `RowScan` operator. This effect is more eminent in the query plan because there the pipeline is even bigger than the one in our microbenchmarks. Another cause of the slowdown is that in the query plan, we cannot exactly isolate the local partitioning phase, but we instead measure the whole sub-plan present in the first `NestedMap` of Figure 3 and subsequently subtract the one present in the second `NestedMap`. This nested plan includes an extra materialization of the output partitions, and the processing of metadata necessary for later phases of the algorithm. Although the latter is not significantly compute-heavy, it attributes to a small part of the slowdown present.

The *build probe phase* is faster in the model, compared to the original code base and the query plan. The slowdown in the first case is explained by the fact that the `MaterializeRowVector` uses the `realloc` function to request more memory, compared to the allocator interface of the original code base. In the second case, build probe is again part of a large pipeline. Therefore we lack some compiler optimizations. On 8 machines, each process materializes fewer tuples and requests less memory, and these effects are ameliorated.

Finally, in Figure 9b, we depict the total runtime of the monolithic operator compared to the Modularis plan for the distributed join algorithm. Our modular integration is from 12 to 28% slower, depending on the number of machines used, due to the reasons explained before. However, the operators present in this plan exhibit the advantage that they can be reused in a variety of queries.

5.3 Distributed `GROUP BY`

In this section, we run the distributed `GROUP BY` plan presented in Section 4.3. We show our results in Figure 10. On the left side of the Figure, we run the plan across different machine configurations for a workload of 2048 unique million keys. As expected, the total runtime decreases as the algorithm load is distributed across more nodes. On the right side of the Figure, we increase the number of distinct keys in the input (and hence the number of groups in the result) and execute the plan for three different machine configurations. Because the total execution time is dominated by the network time and the materialization of the tuples, the time is almost steady for each machine configuration. Overall, we observe that Modularis performs grouping of billion of elements in a few

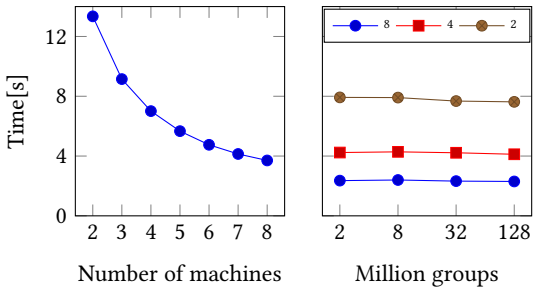


Figure 10: Distributed `GROUP BY` runtime: varying cluster size with fixed key cardinality (left); varying key cardinality for different cluster sizes (right)

seconds without having to design a specialized monolithic operator for this cause but mainly by reusing existing system components.

5.4 Sequences of joins

Finally, we present the results of executing sequences of joins. As a baseline, we use the naive version of the plan that shuffles four relations through the network. We compare the naive plan against an optimized one that shuffles only three. Both of these versions are implemented in our system. For this experiment, we use multiple relations with 2048 million tuples each, similar to the relations used in [13]. Additionally, we use a 1-to-1 correspondence between the keys in the inner and outer relation unless otherwise mentioned.

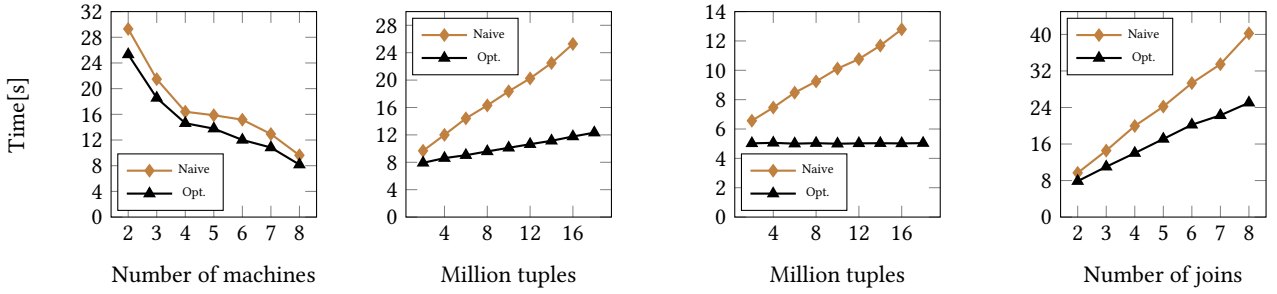
Figure 11a shows the execution time when performing a sequence of two joins across several machines with the two variants of the algorithm, *naive* and *optimized*. We observe that there is a constant speedup in the optimized version compared to the baseline, which is partly due to the network shuffling of one less relation and partly due to the materialization of only the final result instead of materializing both the intermediate and the final join output. The execution time has a sublinear speedup as the number of machines increases because the tail latencies mentioned in the previous section in the network phases are even more eminent. Because the network phases constitute a larger part of the execution time of the optimized version, these tail latencies are the main reason that the speedup between the baseline and the optimized version is decreased as the number of machines increases.

However, the advantage of the optimized version is more conspicuous when the first join has an increasing join output. We show the total runtime of such an experiment across 8 machines in Figure 11b and the time spent on partitioning data through the network in Figure 11c. While the naive version increases linearly with a high rate as the algorithm materializes and shuffles through the network an extra relation that has an increasing size, the optimized version has a sublinear increase in its total execution time. To analyze further the cause of this time difference, we show in Figure 11c that for the optimized version the time spent on shuffling data through the network is constant, as all three relations are pre-partitioned at the beginning of the plan execution while the network time is increasing linearly as more data are shuffled through the network. In these two plots, we cannot execute the baseline algorithm for more than 18 million tuples due to memory constraints.

Lastly, we compare the two versions while increasing the number of joins performed. We show the results in Figure 11d. The difference in the total runtime between the two versions is proportional to the number of joins because for N joins, the optimized plan performs $N-1$ materializations and $N+1$ network shuffling phases less than the naive plan. This shows, as in the `GROUP BY` case, that we can apply optimizations that have a significant performance impact. This comes without rewriting the whole system from scratch, as in the case of monolithic operators, but mainly by restructuring operators across plans.

6 CONCLUSION

In this paper, we have proposed Modularis—an execution engine based on sub-operators. After sketching the design principles that the sub-operators should follow, we propose an initial set of sub-operators and demonstrate how they can be combined to build traditional database operators such as a distributed hash join or a `GROUP BY` query as well as more complex query plans like sequences of joins and TPC-H queries. We show that by changing a small subset of our sub-operators we can execute the same TPC-H query plans on diverse hardware platforms (RDMA, serverless clusters, smart storage). Through extensive experiments, we show that modularity reduces implementation effort without requiring users to sacrifice performance. Modularis is an order of magnitude faster than Presto, a data warehouse supporting various storage layers and distributed setups, and more performant in the majority of cases than SingleStore, an in-memory analytics SQL engine. Modularis also outperforms Query-as-a-Service systems such as BigQuery and Athena by simply changing the RDMA-specific operators with dedicated serverless-based ones so that queries run on the cloud.



(a) Across machines **(b) Increasing join output** **(c) Network time** **(d) Across number of joins**

Figure 11: Sequences of joins in Modularis

REFERENCES

[1] Accessed 2021. Amazon AQUA. https://aws.amazon.com/redshift/features/aqua/.

[2] Accessed 2021. Athena external tables. https://docs.aws.amazon.com/athena/latest/ug/create-table.html.

[3] Accessed 2021. BigQuery external tables. https://cloud.google.com/bigquery/external-data-sources.

[4] Accessed 2021. Mutable project. https://bigdata.uni-saarland.de/projects/mutable/.

[5] Accessed 2021. Numba python package. http://numba.pydata.org/.

[6] Accessed 2021. S3Select. https://aws.amazon.com/blogs/aws/s3-glacier-select/.

[7] Accessed 2021. SingleStore column-store format. https://docs.memsql.com/v7.1/concepts/columnstore/.

[8] Accessed 2021. SingleStore row-store format. https://docs.memsql.com/v7.1/concepts/rowstore/.

[9] Sandeep R. Agrawal, Sam Idicula, Arun Raghavan, Evangelos Vlachos, Venkatraman Govindaraju, Venka tanathan Varadarajan, Cagri Balkesen, Georgios Giannikis, Charlie Roth, Nipun Agarwal, and Eric Sedlar. 2017. A Many-core Architecture for In-Memory Data Processing. In *MICRO*. https://doi.org/10.1145/3123939.3123985

[10] Lixiang Ao, Liz Izhikevich, Geoffrey M Voelker, and George Porter. 2018. Sprocket: A serverless video processing framework. In *SoCC*. 263–274. https://doi.org/10.1145/3267809.3267815

[11] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *ICDE*. https://doi.org/10.1109/ICDE.2013.6544839

[12] Maximilian Bandle and Jana Giceva. 2021. Database Technology for the Masses: Sub-Operators as First-Class Entities. *PVLDB* (2021).

[13] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-Scale In-Memory Join Processing Using RDMA. In *SIGMOD*. https://doi.org/10.1145/2723372.2750547

[14] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. 2017. Distributed Join Algorithms on Thousands of Cores. *PVLDB* 10, 5 (2017). https://doi.org/10.14778/3055540.3055545

[15] Claude Barthels, Ingo Müller, Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. 2019. Strong consistency is not hard to get: Two-Phase Locking and Two-Phase Commit on Thousands of Cores. *PVLDB* 12, 13 (2019). https://doi.org/10.14778/3358701.3358702

[16] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The end of slow networks: it's time for a redesign. *PVLDB* 9, 7 (2016). https://doi.org/10.14778/2904483.2904485

[17] Spyros Blanas, Paraschos Koutris, and Anastasios Sidiropoulos. 2020. Topology-aware Parallel Data Processing: Models, Algorithms and Systems at Scale. In *CIDR*.

[18] Peter Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*. https://doi.org/10.1007/978-3-319-04936-6_5

[19] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: A serverless framework for end-to-end ml workflows. In *SoCC*. 13–24. https://doi.org/10.1145/3357223.3362711

[20] John Cieslewicz and K.A. Ross. 2007. Adaptive Aggregation on Chip Multiprocessors. In *VLDB*.

[21] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. 2015. An Architecture for Compiling UDF-centric Workflows. *PVLDB* 8, 12 (2015). https://doi.org/10.14778/2824032.2824045

[22] Christopher Dennl, Daniel Ziener, and Jurgen Teich. 2012. On-the-fly Composition of FPGA-Based SQL Query Accelerators Using a Partially Reconfigurable Module Library. In *FCCM*. https://doi.org/10.1109/FCCM.2012.18

[23] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. 1990. The Gamma Database Machine Project. *TKDE* 2, 1 (1990). https://doi.org/10.1109/69.50905

[24] Jens Dittrich and Joris Nix. 2020. The Case for Deep Query Optimisation. In *CIDR*.

[25] Klaus R Dittrich and Andreas Geppert. 2000. *Component database systems*. Elsevier.

[26] Kayhan Dursun, Carsten Binnig, Ugur Cetintemel, Garret Swart, and Weiwei Gong. 2019. A Morsel-Driven Query Execution Engine for Heterogeneous Multi-Cores. *PVLDB* 12, 12 (2019). https://doi.org/10.14778/3352063.3352137

[27] Grégory Essertel, Ruby Y. Tahboub, Fei Wang, James Decker, and Tiark Rompf. 2019. Flare & Lantern: Efficiently Swapping Horses Midstream. *PVLDB* 12, 12 (2019). https://doi.org/10.14778/3352063.3352097

[28] Grégory M. Essertel, Ruby Y. Tahboub, James M. Decker, Kevin J. Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing Apache Spark with Native Compilation for Scale-up Architectures and Medium-size Data. In *OSDI*.

[29] Yuanwei Fang, Chen Zou, and Andrew A Chien. 2019. Accelerating raw data analysis with the ACCORDA software and hardware architecture. *PVLDB* 12, 11 (2019). https://doi.org/10.14778/3342263.3342634

[30] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *USENIX ATC*. 475–488.

[31] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. 2018. Enabling Highly Scalable Remote Memory Access Programming with MPI-3 One Sided. *CACM* 61, 10 (2018). https://doi.org/10.1145/3264413

[32] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. 2006. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management Naga. In *SIGMOD*. https://doi.org/10.1145/1142473.1142511

[33] Goetz Graefe. 1990. Encapsulation of Parallelism in the Volcano Query Processmg System. In *SIGMOD*. https://doi.org/10.1145/93597.98720

[34] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2009. Relational Query Coprocessing on Graphics Processors. *TDS* 34, 4 (2009). https://doi.org/10.1145/1620585.1620588

[35] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2008. Relational Joins on Graphics Processors. In *SIGMOD*. https://doi.org/10.1145/1376616.1376670

[36] Zhenhao He, David Sidler, Zsolt István, and Gustavo Alonso. 2018. A Flexible K-Means Operator for Hybrid Databases. In *FPL*. https://doi.org/10.1109/FPL.2018.00069

[37] Florian Irmert, Michael Daum, and Klaus Meyer-Wegener. 2008. A new approach to modular database systems. In *Proceedings of the 2008 EDBT workshop on Software engineering for tailor-made data management*. 40–44.

[38] Insoon Jo, Duck-Ho Bae, Andre S Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. 2016. YourSQL: a high-performance database system leveraging in-storage computing. *PVLDB* (2016), 924–935.

[39] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: Distributed computing for the 99%. In *SoCC*. 445–451. https://doi.org/10.1145/3127479.3128601

[40] Kaan Kara, Jana Giceva, and Gustavo Alonso. 2017. FPGA-Based Data Partitioning. In *SIGMOD*. https://doi.org/10.1145/3035918.3035946

[41] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *PVLDB* 11, 13 (2018). https://doi.org/10.14778/3275366.3284966

[42] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *PVLDB* 2, 2 (2009). https://doi.org/10.14778/1687553.1687564

[43] Youngbin Kim and Jimmy Lin. 2018. Serverless data analytics with flint. In *IEEE CLOUD*. 451–455. https://doi.org/10.1109/CLOUD.2018.00063

[44] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding ephemeral storage for serverless analytics. In *ATC*. 789–794.

[45] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *OSDI*. 427–444.

[46] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building Efficient Query Engines in a High-level Language. *PVLDB* 7, 10 (2014). https://doi.org/10.14778/2732951.2732959

[47] André Kohn, Viktor Leis, and Thomas Neumann. 2021. Building Advanced SQL Analytics From Low-Level Plan Operators. In *SIGMOD*. 1001–1013.

[48] Jyoti Leeka and Kaushik Rajan. 2019. Incorporating Super-Operators in Big-Data Query Optimizers. *PVLDB* 13, 3 (2019). https://doi.org/10.14778/3368289.3368299

[49] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. In *SIGMOD*. https://doi.org/10.1145/2588555.2610507

[50] Feng Li, Sudipto Das, Manoj Syamala, and Vivek Narasayya. 2016. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *SIGMOD*. https://doi.org/10.1145/2882903.2882949

[51] Feilong Liu, Lingyan Yin, and Spyros Blanas. 2017. Design and Evaluation of an RDMA-aware Data Shuffling Operator for Parallel Database Systems. In *EuroSys*. https://doi.org/10.1145/3064176.3064202

[52] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *SIGMOD*. https://doi.org/10.1145/3318464.3389758

[53] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. 2015. Cache-Efficient Aggregation: Hashing Is Sorting. In *SIGMOD*. https://doi.org/10.1145/2723372.2747644

[54] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011). https://doi.org/10.14778/2002938.2002940

[55] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-end Optimization for Data Analytics Applications in Weld. *PVLDB* 11, 9 (2018). https://doi.org/10.14778/3213880.3213890

[56] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *SIGMOD*. 131–141. https://doi.org/10.1145/3318464.3380609

[57] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *SIGMOD*. https://doi.org/10.1145/2723372.2747645

[58] Orestis Polychroniou and Kenneth A. Ross. 2014. A Comprehensive Study of Main-Memory Partitioning and its Application to Large-Scale Comparison- and Radix-Sort. In *SIGMOD*. https://doi.org/10.1145/2588555.2610522

[59] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *NSDI*. 193–206.

[60] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. 2015. High-Speed Query Processing over High-Speed Networks. *PVLDB* 9, 4 (2015). https://doi.org/10.14778/2856318.2856319

[61] Abdallah Salama, Carsten Binnig, Tim Kraska, Ansgar Scherp, and Tobias Ziegler. 2017. Rethinking Distributed Query Execution on High-Speed Networks. *IEEE Data Eng. Bull.* 40, 1 (2017). https://doi.org/10.14778/2904483.2904485

[62] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. 2018. Serverless data analytics in the IBM cloud. In *Proceedings of the 19th International Middleware Conference Industry*. 1–8. https://doi.org/10.1145/3284028.3284029

[63] Felix Martin Schuhknecht, Pankaj Khanchandani, and Jens Dittrich. 2015. On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning. *PVLDB* 8, 9 (2015). https://doi.org/10.14778/2777598.2777602

[64] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. 2018. Numpywren: Serverless linear algebra. *arXiv preprint arXiv:1810.09679* (2018).

[65] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures. In *SIGMOD*. https://doi.org/10.1145/3035918.3035954

[66] Jens Teubner and Rene Mueller. 2011. How Soccer Players Would do Stream Joins. In *SIGMOD*. https://doi.org/10.1145/1989323.1989389

[67] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: An intelligent storage engine with support for advanced sql offloading. *PVLDB* (2014), 963–974.

[68] Xiangyao Yu, Matt Youill, Matthew Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2020. PushdownDB: Accelerating a DBMS using S3 computation. In *2020 IEEE ICDE*.

[69] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[70] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. 2019. Rethinking Database High Availability with RDMA Networks. *PVLDB* 12, 11 (2019). https://doi.org/10.14778/3342263.3342639

[71] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks. In *SIGMOD*. https://doi.org/10.1145/3299869.3300081