The Collection Virtual Machine: An Abstraction for Multi-Frontend Multi-Backend Data Analysis

Ingo Müller[1], Renato Marroquín[2], Dimitrios Koutsoukos[1], Mike Wawrzoniak[1], Sabir Akhadov[3], Gustavo Alonso[1]

[1]{ingo.mueller,dkoutsou,michal.wawrzoniak,alonso}@inf.ethz.ch
Systems Group, Department of Computer Science, ETH Zurich

[2]renato.marroquin@oracle.com
Oracle Labs

[3]sabir.akhadov@databricks.com
Databricks

ABSTRACT

Getting the best performance from the ever-increasing number of hardware platforms has been a recurring challenge for data processing systems. In recent years, the advent of data science with its increasingly numerous and complex types of analytics has made this challenge even more difficult. In practice, system designers are overwhelmed by the number of combinations and typically implement a single analytics type on one platform, leading to repeated implementation effort—and a plethora of semi-compatible tools for data scientists.

In this paper, we propose the "Collection Virtual Machine" (or CVM)—an extensible compiler framework designed to keep the specialization process of data analytics systems tractable. It can capture at the same time the essence of a large span of low-level, hardware-specific implementation techniques as well as high-level operations of different types of analyses. At its core lies a language for defining nested, collection-oriented intermediate representations (IRs). Frontends produce programs in their IR flavors defined in that language, which get optimized through a series of rewritings (possibly changing the IR flavor multiple times) until the program is finally expressed in an IR of platform-specific operators. While reducing the overall implementation effort, this also improves the interoperability of both analyses and hardware platforms. We have used CVM successfully to build specialized backends for platforms as diverse as multi-core CPUs, RDMA clusters, and serverless computing infrastructure in the cloud and expect similar results for many more frontends and hardware platforms in the near future.

ACM Reference Format:
Ingo Müller, Renato Marroquín, Dimitrios Koutsoukos, Mike Wawrzoniak, Sabir Akhadov, and Gustavo Alonso. 2020. The Collection Virtual Machine: An Abstraction for Multi-Frontend Multi-Backend Data Analysis. In *International Workshop on Data Management on New Hardware (DAMON'20), June 15, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3399666.3399911

DAMON'20, June 15, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-8024-9/20/06…$15.00
https://doi.org/10.1145/3399666.3399911

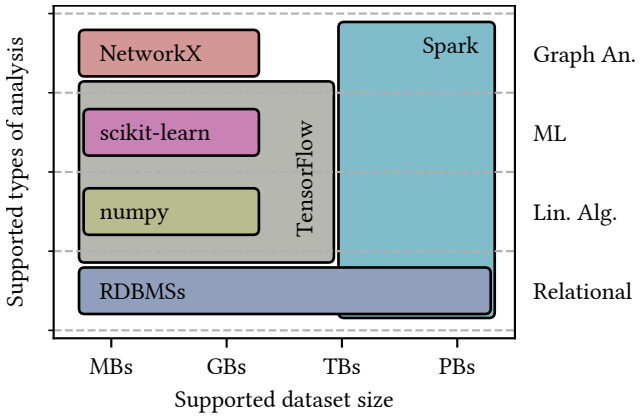

Figure 1: Illustration of today's data science tools.

1 INTRODUCTION

A major goal of systems design has always been to translate increased hardware performance into higher application performance. This consists more and more of exploiting specialized hardware across the entire stack—be it parallelization on the level of SIMD [36, 50, 61], multi-cores [6, 12, 16, 44], NUMA [5, 35, 38], and machines [30, 39, 40, 51, 53, 60], or support for accelerators such as GPUs [1, 20, 23, 26, 47], FPGAs [18, 28, 41, 42], or specialized IO devices including NVMe-based storage [25, 59] or RDMA-capable networking [7, 8, 19, 37, 54]. With the advent of data science and its more diverse and more complex types of analytics, the challenge for system designers has been extended to yet another dimension.

While there is ample research on how to exploit each hardware platform in isolation, practitioners are struggling to build systems that support more than one or a few of them at the same time. Figure 1 illustrates this problem. It shows the most popular Python packages for numerous types of analytics with the size of datasets they support as an indication of what platforms they run on. It is clear no single system currently covers both the breadth of analytics and all dataset sizes. Using RDBMSs and SQL can support datasets of virtually any size, but the packages for linear algebra, graph analytics, and machine learning are mainly built for running on a single machine, thus supporting datasets of at most some tens of gigabytes. TensorFlow [1] has arguably the largest coverage, but, to the best of our knowledge, is not designed for petabyte-scale

[2]Most contributions of this author took place while affiliated with ETH Zurich.
[3]The contributions of this author took place while affiliated with ETH Zurich.

analytics and has only limited support for graph analytics and relational algebra. Finally, systems scaling to racks or clusters, such as Spark [60], do support larger datasets; however, if (mis)used in a single-machine setup, they are typically one or several orders of magnitude less efficient. Overall, tools tend to specialize in a relatively narrow type of analysis/platform combination. As a consequence, individual users are forced to switch tools constantly as their datasets, focus of investigation, or hardware change. At the same time, many basic system components are reimplemented in each of the specialized systems leading to both higher implementation effort and less efficient implementations.

To overcome that situation, this project aims to provide system designers with a unified framework across both hardware platforms and target domains. We hypothesize that all (or at least most) modern hardware platforms and types of data analysis used by data scientists today are similar enough to be expressed in intermediate representations (IRs) based on the common abstraction of *(nested) transformations of (nested) collections*. As we explain in more detail below, analytics in relational algebra, graph analysis, linear algebra, and machine learning work on relations of records, set of vertices and edges, vectors and matrices of numbers, and bags of samples, respectively, all of which are some sort of "collection" of "tuples" of "atoms." Also implementations, including the most optimized forms, can be described naturally in a nested way: inner loops can be seen as the transformation of individual atoms of one collection into another and the orchestration code around them as the nested compositions of these transformations.

Based on this hypothesis, we are building the "Collection Virtual Machine" (CVM), a compiler framework for multi-frontend multi-backend data analysis. Its core is a language for defining collection-oriented intermediate representations (IRs) that consists of arbitrary collection-based "instructions" that we call "operators."[1] Frontends languages then map to a program in an IR defined in this language, typically using high-level operators that may be in part specific to that frontend. Similarly, the backend of a particular hardware platform can define its instructions expressing the low-level implementation techniques required to maximize performance. Since programs at all levels of abstraction are expressed in the same IR language, rewritings between them can be implemented in a common optimizer framework to bring the input program into an optimized, platform-specific form.

We have used CVM for the IRs of three different systems: *JITQ* [4], specialized for multi-core CPUs, *Modularis* [31], specialized for RDMA clusters, and *Lambada* [43], specialized for serverless cloud functions. While the three platforms are diverse and require different, specific implementation techniques, they not only share CVMs compiler infrastructure but also the overwhelming majority of their IRs and the rewritings through which they are compiled. Furthermore, they share a generic Python frontend allowing data scientists to change platforms seamlessly. In experiments, we show that the multi-core and RDMA-based systems are roughly on par with mature systems specialized for these platforms while the cloud backend is up to an order of magnitude faster and up to two orders of magnitude cheaper than two commercial RDBMSs optimized for the same use case.

[1] We use the terms interchangeably in this document.

2 RELATED WORK

Our work draws heavy inspiration from relational database systems. Consequently, all work on query optimization and execution techniques are relevant because we are designing CVM such that all of them could be implemented in its IR language. This includes work from the 90s on relational algebra on nested relations [55] and sequences [52], as well as more recent efforts on array database systems [11]. Similarly, there are large bodies of research on domain-specific implementation techniques for linear algebra [22], machine learning algorithms [58], and graph analysis algorithms [15] and we design CVM such that it can also incorporate these techniques.

There have been numerous projects increasing the breadth of systems using a mix of compiler and query optimization techniques. For example, TensorFlow XLA [34] is a high-level compiler built to support different computing platforms including accelerators. To combine different types of analysis in one system, systems like LaraDB [27], LevelHeaded [2], and AIDA [14] integrate relational algebra with linear algebra in a single abstraction. Systems like LB2 [57] and EmptyHeaded [3] do the same with relational algebra and graph analytics. Raven [29] uses an IR that enables cross-optimization and integrated execution of ML inference and relational queries. To target even more domains, Tupleware [13] and Weld [46] use query optimization and just-in-time compilation to run algorithms from different domains efficiently but the first is restricted to optimizations possible on UDFs and the IR of the latter is fundamentally limited to shared-memory systems. SystemDS [9] builds on SystemML's [10] compilation toolchain and can run a wide range of data science processes on multiple backends, including local CPU/GPU and Spark. Similar systems include Musketeer [21] and Vodoo [49], as well as the works of Kunft et al. [32], Gubner [24], and Pirk and Giceva [48]. While the above are built on some kind of IR, they all have in common that their IR consists of a fixed set of instructions, making it difficult to extend with further frontends and backends and thus support a more narrow analysis/platform combination than CVM targets. Other systems, like Naiad [45], achieve an impressive breadth of applications and performance, but by means of a fixed, low-level programming model rather than a compiler framework. More recently, the MLIR [33] compiler framework aims to provide tools and abstractions for expressing, transforming, and composing of a wide range of intermediate representations and compilation to a broad range of hardware targets, including ML accelerators, but with a focus on deep learning on GPUs and TPUs.

Researchers have also advocated for the opposite direction, most notably Stonebraker and Çetintemel [56], who argued that the "one-size-fits-all" paradigm is not feasible anymore. We believe that the work referenced above shows that covering several types of workloads in one system is, in fact, already possible and hope to extend the coverage even further with CVM.

3 THE COLLECTION VIRTUAL MACHINE

3.1 Architecture Overview

CVM purposefully defines a *language* of intermediate representations instead of a concrete IR with a fixed set of instructions. It fixes *how* instructions and collection types look like—not *which* of them

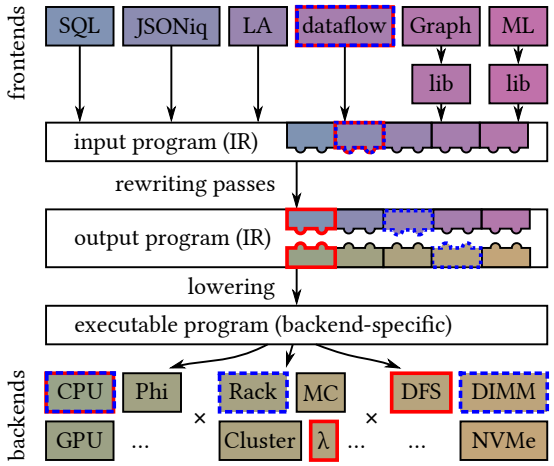

Figure 2: Architecture of the Collection Virtual Machine. Elements composing Lambada [43] and Modularis [31] are outlined in red and dashed blue lines, respectively.

exist. This allows both frontends and hardware-specific backends to define the precise building blocks they need and still evolve as hardware, applications, and experience in IR design make progress.

Figure 2 illustrates an overview of the different components of the Collection Virtual Machine (CVM) and the workflow of transforming a frontend program into an executable form. The figure shows several frontend languages and interfaces that deal with collections but is not meant to be exhaustive. Analyses in the frontends are expressed as or translated into an intermediate representation (IR) defined in the CVM IR language. This initial translation should be as thin as possible. Frontends may define their own IR flavor including high-level operators, collection types, or data types, for example to perform domain-specific optimizations in a frontend-specific rewriting pass. The frontend and backends we implemented so far are highlighted in the figure.

Once in a CVM IR, the program undergoes a succession of rewritings that bring it into an optimized, executable form. Which rewritings are applied and in which order depends on the frontend and target backend(s) of the system. During the rewriting, the program may change the IR flavor several times, typically (but not necessarily) going from more high-level IRs to more low-level ones and intermediate programs may contain a mix of different IR flavors. Since all programs use IRs defined in the same IR language, mixing both IRs and rewritings is seamless such that system builders can share their implementation effort. For example, the three systems we have implemented so far share a common set of rewritings that produce generic data-parallel programs in a common IR and then rewrite some of the instructions as different instructions or sequences thereof in their respective target-specific IR flavors.

Finally, the program is in a form where its instructions correspond directly to the executable building blocks of the target backend. Like in compilers, we call the translation process of the final IR flavor into that executable form *lowering*. For example, a traditional query compiler would lower the IR of physical operators into an execution plan. In JITQ, Modularis, and Lambada, we use a combination of two lowerings: we lower pipelines representing the data paths into native machine code using just-in-time compilation and the surrounding orchestration logic into a dataflow-based execution layer.

3.2 IR Language

All IRs in CVM are built on the mental model of an abstract virtual computer that we call the *Collection Virtual Machine*. The virtual machine has an unlimited number of registers that store *collections* and executes linear sequences of *instructions* called *programs*. Any transformation or execution of its IRs must preserve the behavior as if it was executed on that machine.

The IR language allows to define IR flavors consisting of a set of instructions and collection types. All collection types are generic with the following recursive structure:

$$item := \{\, atom \mid tuple \text{ of } items \mid collection \text{ of } items \,\}, \qquad (1)$$

where an *atom* is an undividable value of a particular domain, a *tuple* is a mapping from a domain of names to items, and a *collection* is the generalization of any (abstract or physical) data type holding a finite, homogeneous multiset. We denote tuple types by $\langle fieldName0 : ItemType0, \ldots, fieldNameK : ItemTypeK \rangle$ and collection types by $CollectionType\langle ItemType \rangle$.

Instructions (or operators) defined by any IR flavor have the following structure: They read the collections from zero or more previously assigned registers and assign results to zero or more previously unassigned registers; registers are hence immutable and programs always in static single assignment (SSA) form. Instructions may be parameterized with (constant) items and programs. If an instruction takes a program as parameter, we call it a *higher-order* instruction. Any instruction is thus of the following form[2]:

$$Out_1, ..., Out_m \leftarrow \textsc{Instruction}(Para_1, ..., Para_k)(In_1, ..., In_n)$$

where In_i and Out_i are the input and output registers, respectively, and $Para_i$ the parameters (i.e., constant items and programs).

3.3 Collection Types

We now show how to define several collection types in CVM's IR language to express both abstract and physical data structures from various domains. These examples are meant to show the expressiveness of the IR language rather than a final set of types and we expect to add more frontend and backend-specific types as we implement other IRs in the future. Table 1 shows the corresponding data types.

Abstract collection types. First, collections can represent abstract data types, like the ones shown in the upper half of the table. To that aim, we define the generic collection types *Set*, *Bag*, *Seq* (for sequences), *kDSeq* (for k-dimensional sequences), and Seq^{∞} (for unbounded sequences). We use them to compose high-level data types of various domains. For example, relations from the original set-based relational algebra (RA) are simply *Set*s of tuples of atoms. For this domain, the fact that items may be tuples is essential as their field names and types represent the schema of the relation. To express the more practical bag-based relations (Bag RA, which is used in our current frontend), sorted relations (from relational algebra on sequences, Seq. RA), and relations in non-first

[2]For brevity, we omit empty components.

Domain	Data structure	CVM data type
RA	$R(A_1 : D_1, \ldots A_k : D_k)$	$Set\langle A_1 : D_1, \ldots A_k : D_k\rangle$
Bag RA	$R(A_1 : D_1, \ldots A_k : D_k)$	$Bag\langle A_1 : D_1, \ldots A_k : D_k\rangle$
Seq. RA	$R(A_1 : D_1, \ldots A_k : D_k)$	$Seq\langle A_1 : D_1, \ldots A_k : D_k\rangle$
RA (NF^2)	$R_1(A : R_2(\ldots))$	$Bag\langle A : Bag\langle\ldots\rangle\rangle$
	$R(A_1 : (A_2 : D))$	$Bag\langle A_1 : \langle A_2 : D\rangle\rangle$
Streaming	$S_T(T, D)$	$Seq^{\infty}\langle\langle t : Num, event : D\rangle\rangle$
LA	$v \in \mathbb{R}$	$Seq\langle Num\rangle$
	$M \in \mathbb{R}^2$	$2DSeq\langle Num\rangle$ or $Seq\langle Seq\langle Num\rangle\rangle$
	$M \in \mathbb{R}^k$	$kDSeq\langle Num\rangle$ or $Seq\langle\ldots Seq\langle Num\rangle\ldots\rangle$
Graph	$G = (V, E)$	$Set\langle ID\rangle$ and $Set\langle\langle src : ID, dst : ID\rangle\rangle$
row-store	`struct{ D1 field1; …}`	$\langle v1 : D_1, v2 : \ldots\rangle$
	`struct{ D1 field1; …}*`	$Vec\langle\langle v1 : D_1, v2 : \ldots\rangle\rangle$
col-store	`struct{ D1* col1; …}`	$Single\langle\langle v1 : Vec\langle D_1\rangle, \ldots\rangle\rangle$
dense LA	`float* A`	$Vec\langle float\rangle$
	`struct{ int size[2]; float* A; }`	$Single\langle\langle d1 : int, d2 : int, A : Vec\langle float\rangle\rangle\rangle$
sparse LA	`struct{ int nnz; float* A; int* I; int* O; }`	$Single\langle\langle A : Vec\langle float\rangle, I : Vec\langle int\rangle, O : Vec\langle int\rangle\rangle\rangle$
SIMD	`__m256 v`	$Array8\langle float\rangle$

R, R_i: relation; A, A_i: attribute/field name; D, D_i: atomic domain

Table 1: Abstract (top) and physical (bottom) collection types.

normal-form (NF^2), i.e., nested relations, we simply use *Bag* or *Seq* instead of *Set* and allow non-atomic fields, respectively. Streams of timestamped events are Seq^{∞} of two-dimensional tuples of timestamp and event domain. Similarly, we define vectors, matrices, and higher-dimensional tensors from linear algebra (LA), as well as the vertices and edges of graphs. In those domains, the items in the collections have no further structure but are just some type of number (*Num*) or vertex identifier (*ID*).

Physical collection types. Second, collections can express physical data layouts as well. As a basic building block, we define the generic type *Vec* (for vector) to represent an array of items in a single contiguous block of memory. Furthermore, we express fixed-width records with ordered fields (like `structs` in C) as tuples where the lexicographical order of the field names defines the physical order in the layout.

This allows us again to compose many common physical data layouts, such as those shown in the lower half of the table. Both row-store and column-store layout of relations are typically implemented as *array of structs* and *struct of arrays*, which we can express with tuples and *Vec*. The three systems we have built so far use both relation types in their IRs. Notice that we define the generic type *Single* as a singleton collection holding just one tuple as a helper to store a group of collections in a single register. Similarly, the data structures used typically for linear algebra (both dense and sparse) are composed of arrays and structs, so we can express them with the same data types as shown in the table. We only show the sparse matrix format CSR (for "compressed sparse row"), which consists of an integer for the number of non-zero elements and three arrays (the non-zero elements, their column indices, and the offsets of each row into the first two), but the other common formats can be defined analogously.

Finally, as shown in the table, we define *ArrayN* as sequence with compile-time size N to express vectors of machine words for SIMD-style processing. The same collection type can also be used to model a row of a narrow dense matrix to enable compile-time optimizations for that special case.

The actual physical representation is decided by the lowering. For example, our three systems have an execution layer that stores tuples of fixed-width fields in the memory layout of a C-arrays of C-structs and thus require the final IR to contain only *Sequences* of anonymous tuples, which are then lowered accordingly. For that to work, we activate a sequence of rewriting passes that bring the programs into the expected form. We discuss these rewritings in more detail later in this section.

Custom collection types. Finally, we can define new collection types to support arbitrary physical formats and data structures. For example, we have defined collection types for Apache Arrow and Parquet, and other formats such as Protocol Buffers or Avro could be supported with the same approach. This allows frontends to support existing data formats and backends to use specialized, highly-tuned data structures as data types in their respective IR flavor.

3.4 Instructions

As described above, instructions defined in the CVM IR language transform collections into other collections. Instructions may have restrictions on the item types of their input collections and the types of their outputs may depend on their input types. Broadly speaking, the level of abstraction of instructions corresponds to the level of abstraction of the collections they work on. Table 2 shows instructions and their input and output types of various levels of abstractions.

High-level instructions. The upper part of the table shows high-level, domain-specific instructions that typically constitute the IRs used for the initial translation of user-facing programs. For example, an IR for a relational query processor could define the usual relational operators on this level. The table shows the definition of projection (Proj), which is only defined on collections of tuples.[3] If the collection is a sequence (*Seq*) or set (*Set*), then so is the output. While the projection only restricts the field names of the tuples, the extended projection (ExProj) also allows us to compute new fields. We also define a Map instruction, which we use in our generic dataflow frontend and which, in contrast to the projections, can work on arbitrary item types. We define instructions for other relational or generic dataflow operators in much the same way as Proj and Map.

As an example of an IR of a different application domain, the table shows an instruction for matrix-matrix multiplication (MMMult). We can define instructions for other basic operations of linear algebra including multiplications of tensors of different dimensions, inversion, transposition, etc. analogously. This allows

[3] However, the fields of the tuples may consist of arbitrary items.

Instruction	Input type(s)	Output type(s)
Proj$(A_1, \ldots A_k)(C)$	$C : Coll\langle A_1, \ldots A_k \ldots\rangle$	$Bag\langle A_1, \ldots A_k\rangle$
	$C : Set\langle A_1, \ldots A_k \ldots\rangle$	$Set\langle A_1, \ldots A_k\rangle$
	$C : Seq\langle A_1, \ldots A_k \ldots\rangle$	$Seq\langle A_1, \ldots A_k\rangle$
	$C : Seq^{\infty}\langle A_1, \ldots A_k \ldots\rangle$	$Seq^{\infty}\langle A_1, \ldots A_k\rangle$
ExProj$(\{A'_i, f_i\}_l)(C)$, $f_i : \{A_j\}_k \rightarrow I_i$	$C : Coll\langle A_1, \ldots A_k\rangle$	$Bag\langle\{A'_i : I_i\}_l\rangle$
Map$(f : I_1 \rightarrow I_2)(C)$	$C : Coll\langle I_1\rangle$	$Bag\langle I_2\rangle$
	$C : Seq\langle I_1\rangle$	$Seq\langle I_2\rangle$
	$C : Seq^{\infty}\langle I_1\rangle$	$Seq^{\infty}\langle I_2\rangle$
MMMult(C_1, C_2)	$C_i : 2DSeq\langle Num\rangle$	$2DSeq\langle Num\rangle$
Loop$(n, P)(C_1, \ldots C_k)$ $P : \{C_i\}_k \rightarrow \{C_i\}_k$	$C_i : Coll_i\langle I_i\rangle$	$C_i : Coll_i\langle I_i\rangle$
While$(P)(C_1, \ldots, C_k)$ $P : \{C_i\}_k \rightarrow \mathbb{B}, \{C_i\}_k$	$C_i : Coll_i\langle I_i\rangle$	$C_i : Coll_i\langle I_i\rangle$
Cond$(P)(C_1, \ldots, C_k)$ $P : \{C_i\}_k \rightarrow \mathbb{B}, \{C'_j\}^2_l$	$C_i : Coll_i\langle I_i\rangle$	$C'_j : Coll'_j\langle I'_j\rangle$
Call$(P)(C_1, \ldots, C_k)$ $P : \{C_i\}_k \rightarrow \{C'_j\}^2_l$	$C_i : Coll_i\langle I_i\rangle$	$C'_j : Coll'_j\langle I'_j\rangle$
ConcurExecute$(P)(C)$ $P : Single\langle I_1\rangle \rightarrow Single\langle I_2\rangle$	$C : Coll\langle I_1\rangle$	$Bag\langle I_2\rangle$
	$C : Seq\langle I_1\rangle$	$Seq\langle I_2\rangle$
ScanVec(C)	$C : Coll\langle Vec\langle I\rangle\rangle$	$Seq\langle I\rangle$
MatVec(C)	$C : Coll\langle I\rangle$	$Single\langle Vec\langle I\rangle\rangle$
SplitVec(n)(C)	$C : Coll\langle Vec\langle I\rangle\rangle$	$Bag\langle Vec\langle I\rangle\rangle$
	$C : Seq\langle Vec\langle I\rangle\rangle$	$Seq\langle Vec\langle I\rangle\rangle$
BuildHTable(C)	$C : Coll\langle T\rangle$, $T : \langle key : I_1, val : I_2\rangle$	$Single\langle HTab\langle T\rangle\rangle$
ProbeHTable(C, H)	$C : Coll\langle T_1\rangle$, $H : Single\langle HTab\langle T_2\rangle\rangle$, with $T_1.key = T_2.key$	$Bag\langle T_3\rangle$

A_i: attribute/field name; C, C_i: collection type; I, I_i: item type; $n \in \mathbb{N}$; f, f_i: function type; P: nested program; $\mathbb{B} = \{\top, \bot\}$; T, T_i: tuple type

Table 2: Domain-specific, control-flow-like, and low-level instructions.

high-level optimizations based on mathematical and other domain-specific equivalences.

Notice how our definition of collections on different types of items allows expressing linear algebra and relational algebra in the same framework. We can convert collections of one domain to the other by packing (or unpacking) each item into (from) a tuple with a single field, so our IR allows combining programs of various front-ends and doing optimizations across interface barriers.

Control flow. The middle part of the table shows instructions we use to express control-flow-like behavior. Notice that the CVM IR language does not allow for traditional control flow such as jumps. This is done by design as jumps make it hard to understand the semantics of a program, which makes many optimizations difficult or impossible to achieve. However, we can use the capability of defining higher-order instructions to achieve similar effects: The table gives the example of a Loop instruction that is parameterized with a nested program and a constant number n and executes the program n times. It reads its input through input registers as any other instruction, forwards them as initial input of the inner program, and then uses the result registers of the previous run as new input. The final result of the Loop instruction corresponds to what the Return instruction of the last run of the inner program returns. While and Cond (for conditional expression) can be defined in a similar way.[4]

Parallel execution may also be counted as control flow. The table shows the ConcurrentExecute instruction, which we use to represent parallelism in the three systems we have implemented so far. It has similar semantics as the higher-order instruction Map, i.e., it executes a program on each input item to compute an output item, but guarantees that these programs are executed concurrently such that the different executions can exchange data among them. Furthermore, each system has its own, platform-specific version of ConcurrentExecute, which implements the concurrent execution of threads, MPI workers, and serverless cloud functions, respectively.

Low-level instructions. Finally, low-level instructions represent specific building blocks of different backends. On this level, we follow the philosophy to make these operators as small as possible to make them more generic and hence reusable. Our goal is to express cleverness as a sophisticated combination of simple operators instead of a simple combination of sophisticated operators. We refer to our work on Modularis [31] for details. For example, we have a scan operator, a materialize operator, and potentially a split operator (for parallelization) for each of the backend-level collection types mentioned above (the table shows those of *Vec*). Similarly, we define a build and a probe operator for each hash table type that we implement (some of which are tuned for a very specific case).

Furthermore, many if not all low-level tuning techniques developed by the database community in the last years can be encapsulated as operators:

- hardware-conscious algorithms and data structures,
- light-weight compression schemes,
- build and probe of spatial indices or other domains, and
- predicated or vectorized scans, to name just a few.

Notice that all of them fall into our structure of collection-based instructions. Our IR language thus makes it possible to use very specialized implementation techniques and still represent them in a common abstraction.

3.5 Lowerings to Execution Layers

The CVM compilation toolchain can be used to target any execution layer. As mentioned before, a traditional relational query engine could define its physical query plans as an IR and lower programs in that IR into an execution plan composed of its executable operators. For example, we could use MonetDB's execution layer by translating programs through a series of rewritings into an IR that replicates the MonetDB Assembly Language (MAL) and then lowering them into actual MAL for execution.

[4] The first register returned by the nested program contains a Boolean value indicating whether or not to stop the loop or which branch to return, respectively.

Algorithm 1 Initial CVM program of TPC-H Query 6.

```
p: predicate on "l_shipdate", "l_discount", and "l_quantity"
T_lineitem: tuple corresponding to lineitem schema
program TpchQ6Seq(lineitem : Coll⟨T_lineitem⟩)
    filtered ← Select(p)(lineitem)
    projected ←
        ExProj("l_eprice" · "l_disc" → "x")(filtered)
    result ← Aggr(("x", sum) → "revenue")(projected)
    Return(result)
```

Algorithm 2 Parallelized CVM program TPC-H Query 6.

```
program TpchQ6Par(lineitem : Coll⟨T_lineitem⟩)
    parts ← Split(p)(lineitem)
    part_results ← ConcurExecute(TpchQ6Seq)(parts)
    unnested ← Scan(part_results)
    result ←
        Aggr(("revenue", sum) → "revenue")(unnested)
    Return(result)
```

For JITQ, Modularis, and Lambada, we use a common execution layer for the data paths and specialized components for the execution of and communication among parallel workers. The common part deals with the most fine-grained level, where operators pass individual tuples between each other. In a rewriting pass, we extract tree-shaped parts of a program and translate them into pipelines of Volcano-style iterators. To eliminate the overhead of this operator interface and to allow low-level optimizations across operator boundaries, we just-in-time-compile each pipeline to native machine code. The inputs and outputs of each pipeline constitute necessary materialization points of the original program.

In order to lower instructions on unbounded streams, the execution layer needs to support streaming, of course. However, we believe that implementation techniques for streaming are in large parts suitable for finite collections as well and thus represent an interesting lower target for a wide variety of frontends.

3.6 Rewritings

The rewriting mechanism of CVM is highly flexible and configurable, such that every frontend/backend combination can do the rewritings that are best suited for that combination. For the different IR flavors to co-exist, at least during compilation, rewritings must work in the presence of collection types and instructions of any IR. Optimizations (or lowerings) that require a particular property (such as tree-shaped data dependencies) thus either have to rewrite the program to establish that property first or work only on those parts of a program where the property holds.

Algorithms 1 and 2 illustrate how the rewriting for generic parallelization works taking Query 6 of the TPC-H benchmark as an example. The initial program (Tpch6Seq) consists of a selection, a computation, and a scalar aggregation. Our rewriting rule first replaces the usage of the input relation (*lineitem*) with a Split followed by an empty ConcurrentExecute and a Scan.[5] Notice that the sequence of these three operators is a logical no-op. Then it applies rules that expand the ConcurrentExecute in a way that preserves the semantics: It moves Select and ExProj inside, while it copies Aggr as pre-aggregation. If an unknown instruction had been encountered, then the rule would leave it as is. The resulting parallelized program is shown by Algorithm 2.[6] As mentioned earlier, our three systems each continue with a target-specific rewriting pass that rewrites the program in Algorithm 2 into an IR for thread-parallelism, RDMA clusters, or cloud functions, respectively.

[5] This intermediate program is not shown.

[6] The inner program of the ConcurrentExecute happens to be the same as the original program, which is why we refer to Tpch6Seq instead of spelling it out.

In the future, we plan to extend the rewritings considerably. We think that all traditional query optimization techniques from database systems can be done on CVM IRs, including join reordering, index selection, etc.

4 EXPERIMENTS

In this section, we show the experimental results of CVM on three different hardware platforms: in-memory, distributed, and serverless. Although CVM is not always the fastest solution in the comparison, the goal of the experimental study is not to focus on raw performance but rather to show the flexibility of our frontends and backends through the use of platform-specific operators and rewrite rules. In general, CVM has at worst a reasonable performance overhead compared to state-of-the-art data processing systems, while it is often on par or even faster. For all experiments described below, unless otherwise stated, we run each query four times, use the first run as a warm-up and then report the average of the other runs.

In-memory. For the in-memory experiments, we use two workloads: (1) TPC-H queries with scale factor 10, and (2) the k-means clustering algorithm with a synthetic dataset comprising of 2^{24} 5-dimensional points. The numbers for HyPer and Flare are taken from [17]. For k-means, we choose the most popular ML Python package, `scikit-learn` ("sklearn"), as a competitor and we report the time of a single iteration. Both experiments were run on an Intel Xeon E5-2630 v3 CPU running at 2.4 GHz. We report the execution times for TPC-H queries (left) and the k-means algorithm (right) in Figure 3.

We observe that the column-wise operations performed by MonetDB for Q1 have a negative impact on the running time. JITQ lowers the same query into a single pipeline, which leads to producing the result in a single pass. We also observe that when the input data are largely reduced due to very selective filters, such as in Q19, JITQ outperforms the competitors. We believe that implementing other missing optimizations like support for narrow data types, a more sophisticated optimizer, and index-based grouping will make our performance on par for other queries as well.

For k-means, we achieve the performance of the hand-written C++ library used under the hood in `scikit-learn`, mainly due to a plan analysis that enables run-based aggregation. The experiments show that the combination of high-level analysis and just-in-time compilation achieves in-memory processing speed that matches hand-written code performance.

Distributed RDMA cluster. For the distributed experiments, we use eight machines, each with two CPUs Intel Xeon E5–2609

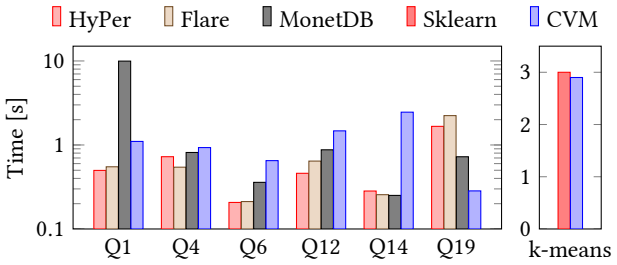

Figure 3: TPC-H (SF 10) and k-means on a single machine.

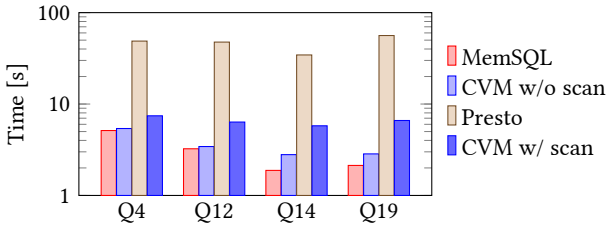

Figure 4: TPC-H (SF 500) on an RDMA cluster.

running at 2.40 GHz and 128 GiB of RAM. The machines are connected through an InfiniBand network with a Mellanox QDR HCA network card. We use TPC-H queries with scale factor 500 and compare against two popular distributed systems, MemSQL and Presto which were configured to use the entire cluster. For Presto, we use HDFS nodes with default configurations to store TPC-H data.

Figure 4 shows the running times for executing the TPC-H queries across the three systems. To have a fair comparison with Presto, we also include the time that Modularis needs to read the input data. We observe that for Q4 and Q12, Modularis is on par with MemSQL while MemSQL is 33 % and 25 % faster than CVM on Q14 and Q19, respectively. Our system is 6× to 9× faster than Presto, depending on the query. Modularis' performance is thus close to a highly optimized in-memory distributed database system and almost an order of magnitude better than that of a popular big-data SQL query engine.

Additionally, in contrast to the other systems, CVM supports this platform only by implementing a few hardware-conscious operators (i.e., `MPIExecutor`, `MPIExchange`, `MPIHistogram`) and by adding additional rewrite rules for incorporating such operators. For instance, we wrote a specialized version of `ConcurrentExecute` called `MPIExecutor` that uses OpenMPI to distribute processes among the machines in the cluster.

Serverless functions. Finally, we show how Lambada executes analytical workloads on a serverless platform. Figure 5 shows the running time and monetary cost of TPC-H queries on serverless cloud services. To showcase the elasticity of serverless computing, we use TPC-H data at scale factor 1000 and decide to use as many serverless workers needed to enable running queries with *interactive latencies*. Compared to other serverless solutions, Google BigQuery and Amazon Athena, Lambada is up to an order of magnitude faster and up to two orders of magnitude cheaper. This

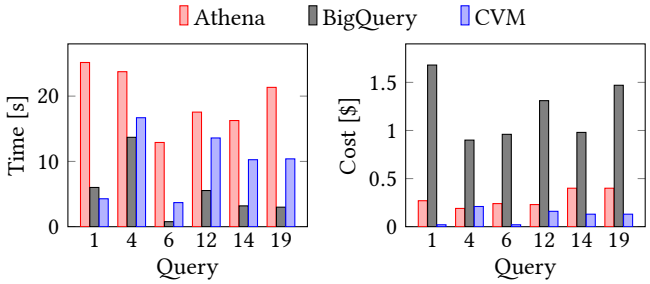

Figure 5: TPC-H (SF 1k) on serverless platforms.

shows that the addition of new lowerings for this platform is orthogonal to existing optimizations.

It is worth restating that the frontend programs implementing the queries are the same in all CVM-based platforms; only the compilation target is changed through a configuration switch. This triggers the use of different rewrite rules, which in turn make use of different, platform-specific operators. For instance, Lambada lowers `ConcurrentExecute` into `ConcurrentLambdaExecute`, an operator that invokes AWS Lambda workers. Similarly, it transforms other operators into Amazon S3-specific operators. Other optimizations such as selections and projections can be pushed into the operator that reads from Amazon S3. Adding such functionality in the competing systems would possibly imply major code rewrites.

5 DISCUSSION AND FUTURE WORK

While the results shown in this paper are very promising and suggest that CVM can indeed solve the problem it was designed for, a large number of questions is still open. In our opinion, some of the most interesting include:

Modular execution layer. What should be the executable building blocks of an ideal execution layer for CVM? As motivated through-out the paper, each building block should be simple and there should be few of them in order to keep their implementation effort low. However, they should still deliver high performance and be able to express arbitrary collection-based computations. In our on-going effort on Modularis [31], we are pursuing this question in the context of clusters and main-stream CPUs, but as we extend CVM to other frontends and backends, we expect new requirements to arise.

Intermediate IRs. What are good IR flavors for intermediate compilation stages? While we believe that frontend and backend-specific IR flavors are necessary for optimizations on these levels, we also believe that common intermediate IR flavors would add significant value to CVM. The more common aspects of frontend and backends they can capture, the more optimization rules can be shared across systems, which would, hence, all benefit from them.

Tractable optimization space. Related to the previous point is the question of how to keep query optimization tractable. Many traditional query optimization techniques for relational database systems rely on the fact that there is a small number of operators such that a large number of query plans can be enumerated and compared. In contrast, the number of IR instructions in CVM is—by design—much larger (and not even defined in advance). We believe that this is not a contradiction: traditional query optimization

should take place on a high-level IR flavor where the number of instructions is still small. But even with this in mind, there are still many possibilities to design the exact layering of optimizations at different levels of abstractions. Exploring them represents an exciting direction for future work.

Streaming. How to implement stream processing within CVM? As we sketch with the examples on Seq^{∞} above, the CVM IR language is able to express unbounded collections just like any other collection type. However, in order to tap into the full potential of CVM, high-level instructions on such unbounded collections should be translated at least in large parts into the same lower-level instructions as those on finite ones, which in turn should be executed by the same execution layer. We have started to work in this direction, but many challenges are yet to be overcome.

We believe that many of these questions transcend CVM and are of general interest to the field. At the same time, we think that CVM represents an excellent basis to answer them: it is designed to be as extensible as possible, making it easy to add and evolve IRs, while still allowing to build on common concepts and tools.

6 CONCLUSIONS

In this paper, we proposed the Collection Virtual Machine, an abstraction for system designers that keeps supporting the growing number of combinations of domain-specific frontends and hardware backends tractable. We have used CVM for the IRs of three different systems: JITQ [4], Modularis [31], and Lambada [43]. While their target platforms are diverse, we have shown how CVM allows the three systems to share large parts of their IRs and rewritings in a common framework and still get comparable performance with systems designed from scratch for the respective hardware platforms. In the near future, we plan to add other frontends, a streaming execution model, and more hardware platforms, where we expect similar results.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, Xiaoqiang Zheng, and Google Brain. "TensorFlow: A System for Large-Scale Machine Learning". In: *OSDI*. 2016.

[2] Christopher R Aberger, Andrew Lamb, Kunle Olukotun, and R Christopher. "LevelHeaded : A Unified Engine for Business Intelligence and Linear Algebra Querying". In: *ICDE*. 2018. DOI: 10.1109/ICDE.2018.00048.

[3] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. "EmptyHeaded: A Relational Engine for Graph Processing". In: *SIGMOD*. 2016. DOI: 10.1145/2882903.2915213.

[4] Sabir Akhadov. "PySpark at Bare-Metal Speed". MA thesis. ETH Zürich, 2017. DOI: 10.3929/ethz-b-000263341.

[5] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. "Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems." In: *PVLDB*. Vol. 5. 10. 2012.

[6] Cagri Balkesen, Jens Teubner, and Gustavo Alonso. "Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware". In: *ICDE*. 2013. DOI: 10.1109/ICDE.2013.6544839.

[7] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. "Rack-Scale In-Memory Join Processing using RDMA". In: *SIGMOD*. 2015. DOI: 10.1145/2723372.2750547.

[8] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. "Distributed Join Algorithms on Thousands of Cores". In: *PVLDB*. 2017. DOI: 10.14778/3055540.3055545.

[9] Matthias Boehm, Iulian Antonov, Mark Dokter, Robert Ginthoer, Kevin Innerebner, Florijan Klezin, Stefanie Lindstaedt, Arnab Phani, and Benjamin Rath. "SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle". In: *CIDR*. 2020.

[10] Matthias Boehm, Michael W Dusenberry, Deron Eriksson, Alexandre V Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R Reiss, Prithviraj Sen, Arvind C Surve, and Shirish Tatikonda. "SystemML: Declarative Machine Learning on Spark". In: *PVDLB* 9.13 (2016). DOI: 10.14778/3007263.3007279.

[11] Paul G. Brown. "Overview of SciDB: Large Scale Array Storage, Processing and Analysis". In: *SIGMOD*. 2010. DOI: 10.1145/1807167.1807271.

[12] John Cieslewicz and K.A. Ross. "Adaptive Aggregation on Chip Multiprocessors". In: *PVLDB*. 2007.

[13] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. "An Architecture for Compiling UDF-centric Workflows". In: *VLDB* 8.12 (2015). DOI: 10.14778/2824032.2824045.

[14] Joseph Vinish D'silva, Florestan De Moor, and Bettina Kemme. "AIDA - Abstraction for Advanced In-Database Analytics". In: *VLDB* 11.11 (2018). DOI: 10.14778/3236187.323619.

[15] Niels Doekemeijer and Ana Lucia Varbanescu. *A Survey of Parallel Graph Processing Frameworks*. Tech. rep. DS-2014-003. TU Delft, 2014.

[16] Kayhan Dursun, Carsten Binnig, Ugur Cetintemel, Garret Swart, and Weiwei Gong. "A Morsel-Driven Query Execution Engine for Heterogeneous Multi-cores". In: *PVLDB*. Vol. 12. 12. 2018. DOI: 10.14778/3352063.3352137.

[17] Gregory Essertel, Ruby Tahboub, James Decker, Kevin Brown, Kunle Olukotun, and Tiark Rompf. "Flare: Optimizing Apache Spark for Scale-Up Architectures and Medium-Size Data". In: *OSDI*. 2018.

[18] Yuanwei Fang, Chen Zou, and Andrew A. Chien. "Accelerating Raw Data Analysis with the ACCORDA Software and Hardware Architecture". In: *PVLDB* 12 (2018). DOI: 10.14778/3342263.3342634.

[19] Philip W. Frey, Romulo Goncalves, Martin Kersten, and Jens Teubner. "A Spinning Join That Does Not Get Dizzy". In: *ICDCS*. 2010. DOI: 10.1109/ICDCS.2010.23.

[20] Henning Funke and Jens Teubner. "Data-Parallel Query Processing on Non-Uniform Data". In: 13 (2020). DOI: 10.14778/3380750.3380758.

[21] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P Grosvenor, Allen Clement, and Steven Hand. "Musketeer:

all for one, one for all in data processing systems". In: *EuroSys*. 2015. DOI: 10.1145/10.1145/2741948.2741968.

[22] Kazushige Goto and Robert A. van de Geijn. "Anatomy of High-Performance Matrix Multiplication". In: *TOMS* 34.3 (2008). DOI: 10.1145/1356052.1356053.

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

[24] Tim Gubner. "Designing an adaptive VM that combines vectorized and JIT execution on heterogeneous hardware". In: *ICDE*. 2018. DOI: 10.1109/ICDE.2018.00215.

[25] Gabriel Haas, Michael Haubenschild, Viktor Leis, Friedrich-Schiller-Universität Jena, and Tableau Software. "Exploiting Directly-Attached NVMe Arrays in DBMS". In: *CIDR*. 2020.

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

[27] Dylan Hutchison, Bill Howe, and Dan Suciu. "LaraDB: A Minimalist Kernel for Linear and Relational Algebra Computation". In: *BeyondMR*. 2017. DOI: 10.1145/3070607.3070608.

[28] Kaan Kara, Jana Giceva, and Gustavo Alonso. "FPGA-based Data Partitioning". In: *SIGMOD*. 2017. DOI: 10.1145/3035918.3035946.

[29] Konstantinos Karanasos, Matteo Interlandi, Doris Xin, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Supun Nakandal, Subru Krishnan, Markus Weimer, et al. "Extending Relational Query Processing with ML Inference". In: *CIDR*. 2020.

[30] Donald Kossmann and Donald. "The state of the art in distributed query processing". In: *ACM Computing Surveys* 32.4 (2000). DOI: 10.1145/371578.371598.

[31] Dimitrios Koutsoukos, Ingo Müller, Renato Marroquín, and Gustavo Alonso. *Modularis: Modular Data Analytics for Hardware, Software, and Platform Heterogeneity*. 2020. arXiv: 2004.03488 [cs.DB].

[32] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, and Volker Markl. "An intermediate representation for optimizing machine learning pipelines". In: *PVLDB*. 2019. DOI: 10.14778/3342263.3342633.

[33] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. "MLIR: A Compiler Infrastructure for the End of Moore's Law". In: (2020). arXiv: 2002.11054.

[34] Chris Leary and Todd Wang. "TensorFlow, Compiled". In: *TensorFlow Dev Summit*. 2017.

[35] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. "Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age". In: *SIGMOD*. 2014. DOI: 10.1145/2588555.2610507.

[36] D. Lemire and L. Boytsov. "Decoding billions of integers per second through vectorization". In: *Softw. Pract. Exper.* 45.1 (2015), pp. 1–29. DOI: 10.1002/spe.2203.

[37] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. "Accelerating Relational Databases by Leveraging Remote Memory and RDMA". In: *SIGMOD*. 2016. DOI: 10.1145/2882903.2882949.

[38] Yinan Li, Ippokratis Pandis, Rene Mueller, Vijayshankar Raman, and Guy Lohman. "NUMA-aware algorithms: the case of data shuffling". In: *CIDR*. 2013.

[39] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. "On the Design and Scalability of Distributed Shared-Data Databases". In: *SIGMOD*. 2015. DOI: 10.1145/2723372.2751519.

[40] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, and Ion Stoica. "Ray: A Distributed Framework for Emerging {AI} Applications". In: *OSDI*. 2018.

[41] Rene Mueller and Jens Teubner. "FPGAs: A New Point in the Database Design Space". In: *EDBT*. 2010.

[42] Rene Mueller, Jens Teubner, and Gustavo Alonso. "Data processing on FPGAs". In: *PVLDB*. 2009. DOI: 10.14778/1687627.1687730.

[43] Ingo Müller, Renato Marroquín, and Gustavo Alonso. "Lambada: Interactive Data Analytics on Cold Data using Serverless Cloud Infrastructure". In: *SIGMOD*. 2020. DOI: 10.1145/3318464.3389758.

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

[45] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. "Naiad: A Timely Dataflow Systems". In: *SOSP*. 2013. DOI: 10.1145/2517349.2522738.

[46] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, Samuel Madden, Matei Zaharia, S Palkar, J Thomas, D Narayanan, P Thaker, R Palamuttam, and P Negi. "Evaluating End-to-End Optimization for Data Analytics Applications in Weld". In: *PVLDB* 11.9 (2018). DOI: 10.14778/3213880.3213890.

[47] Johns Paul, Jiong He, and Bingsheng He. "GPL: A GPU-based Pipelined Query Processing Engine". In: *SIGMOD*. 2016. DOI: 10.1145/2882903.2915224.

[48] Holger Pirk and Peter Giceva Jana Pietzuch. "Thriving in the No Man's Land between compilers and databases". In: *CIDR*. 2019.

[49] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. "Voodoo - A Vector Algebra for Portable Database Performance on Modern Hardware". In: *VLDB*. 2016. DOI: 10.14778/3007328.3007336.

[50] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. "Rethinking SIMD Vectorization for In-Memory Databases". In: *SIGMOD*. 2015. DOI: 10.1145/2723372.2747645.

[51] Orestis Polychroniou, Wangda Zhang, and Kenneth A. Ross. "Distributed Joins and Data Placement for Minimal Network Traffic". In: *TODS* 43 (2018). DOI: 10.1145/3241039.

[52] R. Ramakrsihnan, D. Donjerkovic, A. Ranganathan, K.S. Beyer, and M. Krishnaprasad. "SRQL: Sorted Relational Query Language". In: *SSDBM*. 1998. DOI: 10.1109/SSDM.1998.688114.

[53] Matthew Rocklin. "Dask: Parallel Computation with Blocked algorithms and Task Scheduling". In: *SciPy*. 2015.

[54] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. "High-Speed Query Processing over High-Speed Networks". In: *PVLDB* 9.4 (2015). DOI: 10.14778/2856318.2856319.

[55] Mark A. Roth, Herry F. Korth, and Abraham Silberschatz. "Extended Algebra and Calculus for Nested Relational Databases". In: *TODS* 13.4 (1988). DOI: 10.1145/49346.49347.

[56] Michael Stonebraker and Uğur Çetintemel. ""One size fits all": An idea whose time has come and gone". In: *ICDE*. 2005. DOI: 10.1109/ICDE.2005.1.

[57] Ruby Y. Tahboub, Xilun Wu, Grégory M Essertel, and Tiark Rompf. "Towards Compiling Graph Queries in Relational Engines". In: *SIGPLAN*. 2019. DOI: https://doi.org/10.1145/3315507.3330200.

[58] Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. *Data Mining: Practical Machine Learning Tools and Techniques*. 4th ed. 2017. ISBN: 9780128043578.

[59] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. "Performance analysis of NVMe SSDs and their implication on real world databases". In: *SYSTOR*. 2015. DOI: 10.1145/2757667.2757684.

[60] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster Computing with Working Sets". In: *HotCloud*. 2010.

[61] Jingren Zhou and Kenneth A. Ross. "Implementing database operations using SIMD instructions". In: *SIGMOD*. 2002. DOI: 10.1145/564691.564709.