Thriving in the No Man's Land between Compilers and Databases

Holger Pirk
Imperial College London
hlgr@imperial.ac.uk

Jana Giceva
Imperial College London
j.giceva@imperial.ac.uk

Peter Pietzuch
Imperial College London
prp@imperial.ac.uk

ABSTRACT

When developing new data-intensive applications, one faces a build-or-buy decision: use an existing off-the-shelf data management system (DMS) or implement a custom solution. While off-the-shelf systems offer quick results, they lack the flexibility to accommodate the changing requirements of long-term projects. Building a solution from scratch in a general-purpose programming language, however, comes with long-term development costs that may not be justified. What is lacking is a middle ground or, more precisely, a clear migration path from off-the-shelf Data Management Systems to customized applications in general-purpose programming languages. There is, in effect, a no man's land that neither compiler nor database researchers have claimed.

We believe that this problem is an opportunity for the database community to claim a stake. We need to invest effort to transfer the outcomes of data management research into fields of programming languages and compilers. The common complaint that other fields are re-inventing database techniques bears witness to the need for that knowledge transfer. In this paper, we motivate the necessity for data management techniques in general-purpose programming languages and outline a number of specific opportunities for knowledge transfer. This effort will not only cover the no man's land but also broaden the impact of data management research.

1. INTRODUCTION

The data management landscape is shifting. On the one hand, the underlying components on which data management systems (DMSs) are built are becoming more heterogeneous: programmable devices such as co-processors, smart network interfaces or memory with compute capabilities need to be exploited to maximize efficiency. On the other hand, applications are becoming ever more demanding: correctness and low response times are no longer enough. In cloud environments, users require features such as security, elasticity and cost efficiency.

Unfortunately, DMSs are increasingly falling behind in the race to provide users with the features that they demand. For example, no major database vendor is currently supporting the secure execution of arbitrary queries in protected hardware enclaves using Intel's Software Guard Extensions (SGX) despite the technology being commercially available for three years and there being massive demand.[1] Even if users would like to extend the DMS with custom features such as hardware enclaves, the system lacks the flexibility to accommodate that. The alternative is building a system from scratch, which comes at enormous ramp-up costs that may never pay off. For example, we are familiar with a major cloud music provider migrating their data management solution away from Apache Spark (a flexible framework incorporating multiple storage formats and hand-crafted query plans) to Google's BigQuery (an off-the-shelf solution with opaque storage and an SQL interface) after a number of years because there was little need for custom features and the development costs were, thus, not justified.

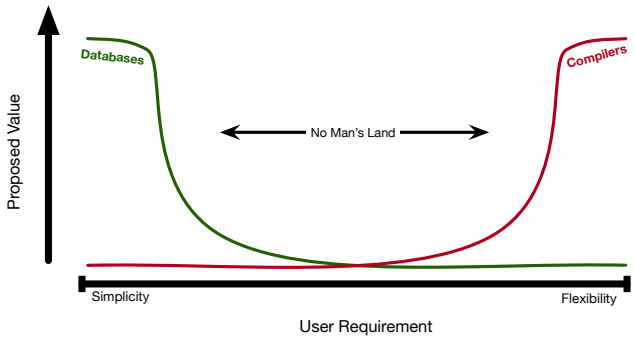

Figure 1: No Man's Land between Compilers and Databases

Figure 1 illustrates this fundamental problem. While the value proposition of DMSs is based on simplicity and a (largely) self-managing system, compilers offer practically unlimited flexibility and control. If application requirements fall onto one end of this spectrum, selecting the appropriate tool is easy; if an application falls somewhere in between, breaking the application into components with unambiguous requirements is hard—when dealing with shifting requirements (as most long-term projects do), this decision is largely based on guesswork.

To demonstrate this problem, consider an application that manages patient data for a hospital chain. It may start out as a classic OLTP database application, i.e., storing and retrieving patient records. At some point, hospitals are equipped with smart medical devices that measure patients vitals. The acquired data points are added to the database as an event stream. At this point, the company's software architects may consider a more specialized data management solution such as a stream store [10]. Later, a predic-

[1]We are only aware of SQL Server's Enclave support, which has been in preview for almost two years.

9th Biennial Conference on Innovative Data Systems Research (CIDR '19) January 13-16, 2019 , Asilomar, California, USA.

tive model management system is added to forecast the need for vaccinations and other treatments based on real-time patient admission data. When the hospital decides to cross-correlate patient's vitals with real-time data about room temperatures to create optimal recovery conditions, the time for a custom-made solution has come. Unfortunately, the hospital is unlikely to adopt such an approach because it means crossing the architectural no man's land: there is no migration path[2] from a DMSs to custom-built solutions.

To address this challenge, we need to claim the space between compilers and databases: the two fields need to move closer, with knowledge transfer in both directions. While we find that the database community is adopting techniques from compiler research [22, 33, 35], rarely, however, does knowledge flow the other way.[3].

We argue that this is less a technological issue but one of mindset: compiler researchers usually focus on small, incremental gains with broad applicability and that all but guarantee the absence of performance regression; most research on DMS performance focuses on carving out regions of the problem space that are amenable to a particular optimization technique. Stateless intermediate algebras simplify the definition of rules that prescribe the applicability of a given optimization.

Technologically, however, these two fields are less divided than they seem: both use dataflow representations internally to simplify optimization, both have notions of cost-based optimization, of transient vs. persistent state and of resource pressure—most concepts in one can be found in the other. This leads to the question:

Can DMS optimizations be applied directly to compiler's intermediate representation and do they yield similar benefits?

We believe they can, if one relaxes the expectation of universal applicability, and we give a specific example of this in Section 4. We also claim that one can go even further than that. We outline a research agenda to gradually merge the fields of databases and compilers by transplanting concepts and ideas from the database research into existing compilers. We motivate the necessity of this agenda in the next section by discussing the growing diversity in application needs as well as the increasing heterogeneity in hardware platforms. We follow with a description of the state-of-the-art in the respective fields (compilers and databases) and a discussion the potential for knowledge transfer in Section 3. After a detailed example in Section 4, we conclude in Section 5.

2. CHANGES IN APPLICATION NEEDS

Application needs are changing. Even when limiting the scope to data management requirements, we find that often they are no longer adequately served by relational DMSs. While SQL is technically Turing-complete, expressing complex data management tasks in user-defined functions (UDFs) is cumbersome and inefficient.

To assess the magnitude of this problem, we implement a linear least-squares classifier with online retraining as a single-threaded C++ program and compare it to one implemented as a PostgreSQL UDF.[4] We insert a stream of 2 million `(int,int)`-tuples generated from a realistic input distribution (fitted to a country's area and population). The model labels the countries as low, medium or high population and is updated with each inserted tuple (note that a linear least-squares model has only four parameters). Figure 2 shows the model management overhead, i.e., execution time with model update minus time with a static model, in seconds. We observe that PostgreSQL adds more than 5000× overhead. While this may technically be justified because PostgreSQL provides features such as recoverability, consistency and data independence, none of these are necessary for the application. However, they come with the package.

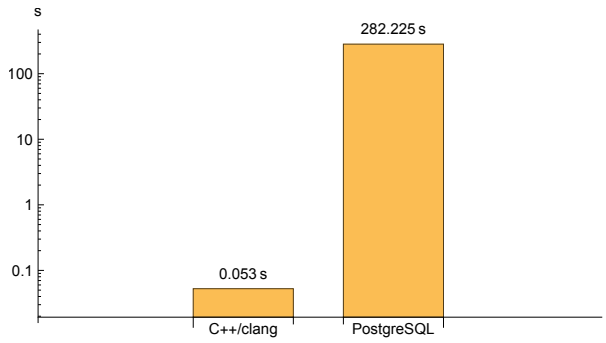

Figure 2: Overhead for Model Management in Postgres

Given the rigid nature of DMSs, it is not surprising that dataflow frameworks such as Spark [44] and Flink [4] use general-purpose programming languages as their frontends as well as to implement their engines. Their reliance on general-purpose languages, though, exposes them to their advantages and disadvantages.

In the following, we discuss these in decreasing order of importance (as perceived by us). Note that these are not necessarily independent dimensions of the problem.

Correctness. Naturally, correctness is the primary concern of application developers. The separation of the declarative frontend from the execution engine is a key advantage of DMSs when reasoning about the correctness of programs. By mixing the two, the aforementioned frameworks sacrifice guaranteed correctness. In Spark Streaming [45], e.g., the result of a window query may be affected by the speed of processing in a cluster. This leads to different "correct" outputs, which is clearly problematic: correctness should be "semantic", i.e., defined independently of the implementation. DMSs have long fulfilled this desiderata.

Performance. Good out-of-the-box performance is often the main reason for developers to opt for a DMS. However, they sacrifice tuning potential for edge-cases that compilers provide. While expecting both in all cases is unreasonable, one would assume that a compiler should, at least, be able to match the performance of a DMS for the data management aspects of an application.

One particularly important aspect of this is the exploitation of modern hardware: as the last decade has brought many-core machines with multi-tier cache hierarchies, DMSs were forced to restructure their architectures to better match the properties of the underlying hardware [2, 3, 8, 29]. This led to the development of *hardware conscious* data processing algorithms and implementations that run efficiently on modern machines by exploiting the new features introduced by hardware vendors.

Compared to the low-level intermediate representation (IR) that a compiler typically operates on, a DMS's query plan is a higher-level representation of the program, describing algorithms as well as access patterns. This knowledge can be exploited to, e.g., achieve better cache and bandwidth usage by re-structuring the data layout (data-partitioning, column vs. row storage, etc.) or hiding cache-miss latencies by software prefetching using access pattern knowl-

[2]A strategy to gradually replace one with the other at reasonable cost and without significant downtime

[3]In fact, database researchers often lament when other fields adopt individual database techniques rather than adopting the package that is a DBMS

[4]Naturally, the PostgreSQL implementation must store the model's state in a table.

edge. Such optimizations have been shown to yield significant benefit compared to compiler-generated code [7, 26, 36, 39, 43].

Flexibility. As illustrated by the online learning example, data management applications are no longer merely OLTP or OLAP. Instead users expect a platform for building generic stateful applications with the flexibility to integrate and run any code. That code could implement a classifier, a specific encryption algorithm or a data cleaning tool—its state could be held in a relation, an array, a neural network or a binary object. Applications need to satisfy requirements as diverse as latency sensitivity, throughout orientation, security sensitivity, scalability, etc. In fact, the configuration may change over the lifetime of the system. Given this list of requirements, it is hard to see how any system short of a compiled general-purpose programming language could satisfy these.

Security. Security is an important application requirement right now and will likely only increase in relevance.[5] How to accommodate security concerns in DMSs is, arguably, one of the most actively studied research questions right now. In this respect, compiler-based approaches are more advanced than their DMS counterparts: taint-tracking and information flow control [47] as security techniques for ensuring data confidentiality are well studied and can be enforced by a compiler across whole applications.

Augmenting such software solutions, hardware can provide secure and trusted execution environments (TEE), which are becoming wide-spread (e.g., Intel's SGX [11], ARM TrustZone [5]). However, efficiently leveraging such hardware extensions requires end-to-end system support and holistic decisions on when and how to use them. For example, in a cloud environment, a DMS can leverage the available TEEs when operating on sensitive data and queries for clients, which are willing to trade-off high performance for high security guarantees. In addition, they can make use of the cache allocation technology (CAT) in new Intel processors to better isolate the effects of concurrent queries, mitigating against side-channel attacks.

Heterogeneous hardware and portability. With the end of Dennard scaling, the hardware landscape is only going to become more heterogeneous with specialized, active components. Hardware extensions no longer only come within the CPU itself as new instructions (e.g., hashing, encryption, vectorization, etc.) but also off-chip e.g., in the form of co-processors such as GPUs, FPGAs, XeonPhi, ASICs, etc. Such a major shift to an *active everything* architecture represents both a challenge and an opportunity. Systems will have to be redesigned to fully exploit the potential of heterogeneous hardware components: they need to generate code that runs efficiently on the target platforms, construct and optimize dataflows to find a deployment that maps operations to devices and manage the necessary data transfers.

One major challenge for software development on such a wide variety of platforms is that they differ in various aspects such as their memory organization and the computational units. While the memory sub-system on a CPU is implicitly managed by the hardware, on a GPU this management is mixed, while on many ASICs, it becomes explicitly managed by the programmer, with caches replaced by scratchpad SRAM memory and memory fetched with DMA engines (e.g., Google's TPU [20], Oracle's DPU [1]). Similar differences can be found in computational units: while CPUs operate primarily on scalar values, GPUs operate on vectors, FPGAs on streams of data items, a domain-specific processor such as Google's TPU operates on domain objects (tensors for deep learning). This diversity in hardware properties yields challenges to the compiler infrastructure [13]. In such a setup, a compiler not only must generate code that adheres to the ISA but also explicitly manage the data layout to match the one required by the memory hierarchy of the accelerator. In addition, it must control the pipeline dependencies in order to hide memory access latencies for accelerators that do in-order processing and configure the runtime with the appropriate threading model.

This is where DMSs can exploit the independence of logical and physical data model: (1) working with query execution plans in a physical data and hardware-independent way enables optimizations that would otherwise be impossible to do in a compiler as they lack the high-level overview of the data-dependent control flow among the different operations; and (2) it enables DMSs to apply transformations that simplify the use of heterogeneous hardware platforms such as co-processors. When targeting specific hardware components, the only scalable way to manage the combinatorial explosion of high-level operations, data types, hardware intrinsics and data layouts is to generate executable code.

Ease of Development. Before concluding this section, let us briefly discuss an important area in which general-purpose programming languages have a significant lead: the ease of development and the availability of abstractions and tools. Consider the availability of unit-testing or refactoring tools for typical DMS query languages in contrast to mainstream programming languages. Given the prevalence of large applications in SQL, the lack of development support is worrying. A related concern is the availability of packaging systems as well as the support for closed-source libraries, which are crucial to many commercial applications. While the lack of tools can, at least in part, be explained by the view of SQL as an domain-specific language, having stronger tool support would help the adoption of SQL.

3. UNIFYING COMPILER AND DMS

As discussed in the previous section, the unification of compilers and DMSs is driven by changes in applications requirements and the need to exploit new hardware features. Historically, compilers and DMSs have targeted different problems. In recent years, however, they have converged slightly: on the DMS side, this was triggered by the move to memory-resident data, which forced query processors to boost their efficiency. Generating executable code (just like compilers do) was a natural means of achieving that [22, 33, 35]. On the compiler side, data-intensive (also known as "big data") applications have created an incentive to study, e.g., domain-specific languages for data processing tasks [30, 38]. There are still fundamental differences in the approaches and architectures of the respective systems, which complicate knowledge transfer.

In this section, we describe a path towards a unified system. We start by presenting the state-of-the-art in the integration and adoption of techniques from one system to the other. We follow this with a description of the challenges that any unification effort faces and conclude with the opportunities of such a unification.

3.1 State-of-the-Art

We begin with a comparison of the respective features of DMSs and compilers (see Table 1). The matrix is, naturally, non-exhaustive but indicative of a general trend. In addition to DMSs and compilers, we added two columns summarising the features that can be provided by either libraries (Lib) or the underlying hardware (HW), but are neither integrated in nor supported by the compiler itself. In addition to present (✓) and missing (✗) features, we mark those that are either a subject of active research or experimental with ⊙. One notable observation from the table is that the feature-sets of

[5] Some may argue that security should even come before correctness.

	DMS	Compiler	Lib	HW
Transactional Isolation	✓	✗	✓	✓
Transactional Atomicity	✓	✗	✓	✓
Transactional Consistency	✓	✗	✓	✗
Transactional Durability	✓	✗	✓	✗
Binary ABI	✗	✓	✗	✗
Cost-based Optimization	✓	✗	✗	✗
Indexing	✓	✗	✓	✗
Adaptive Indexing	✓	✗	✗	✗
Runtime Re-Optimization	✓	⊙	✗	✗
Defined memory layout	✗	✓	✗	✗
Data Independence	✓	✗	⊙	✗
Persistence	✓	✗	✓	✓
Declarative Interface	✓	✓	✓	✗
Access Control	✓	✗	✗	✓
Crash Recovery	✓	✗	✗	✗
Explicit State Management	✗	✓	✗	✗
Intermediate Operator State	✗	✓	✓	✗
Fallback Language	✓	✓	✗	✗
Implicit Resource Mgmt.	✓	✗	✓	✗
Explicit Resource Mgmt.	✗	✓	✗	✗
Unit Testing	✗	✗	✓	✗
Pay-as-you-go cloud pricing	✓	✗	✓	✗

Table 1: Comparing (some) Features of Execution Platforms

compilers and DMSs are largely disjoint. We believe that this merits a deeper discussion.

Knowledge Transfer to DMSs. DMSs have adopted compilers (or, more specifically, virtual machines) as backends because they fulfil the need for CPU-efficient binary code. Projects such as HyPeR [33], Legobase [22] and Voodoo [35] have demonstrated how query processors can be built *on top of compiler frameworks* and the performance benefits that ensue. What they lack is, e.g., a system that allows applications to manage their own state, such as a machine learning model, inside these query compiler systems.

Systems such as TupleWare [15] and Weld [34], which build on top of compiler frameworks, can merge the data managing code of applications with other parts of the program and optimize them jointly. The optimizations are, however, performed using classic compiler techniques with their known limitations—they fail to leverage techniques such as cardinality estimation, data compression or indexing.

Knowledge Transfer to Programming Languages. On the compiler side, one common approach is to re-create the functionality of classical query processing engines within a general-purpose programming language in the form of dataflow libraries. Spark [44], Naiad [32] and ArrayFire [27], implement dataflow paradigms reminiscent of relational algebra.

Like relational databases, they are built *on top of* rather than *integrated with* general-purpose compilers, placing them in the library category in Table 1. On the one hand, this leaves them with limited control over the low-level aspects of execution such as memory allocation, scheduling and vectorization; on the other hand, this also limits their scope: users need to use the provided abstractions (RDDs, Naiad Collections, etc.) and the provided primitives to benefit from them, which is not a fundamental improvement over DMSs. Similarly, approaches such as Exodus [9] or, more conceptually, RISC-style database systems [12] provide no compiler integration at all, although they recognize the problem posed by monolithic DMSs.

There are some early efforts to extend the applicability of data-dependent optimizations, which were previously almost exclusively used in DMSs, to a broader scope of programs by giving hints to the compiler. An example is Milk [21], which allows a compiler to apply optimization to indirect memory loads similar to those that query processors apply to page loads in foreign-key joins. Since this optimization is not universally beneficial (or correct), developers have to enable it using OpenMP pragmas.

While such changes to the internal *workings* of compilers or DMSs positively impact their respective performance, they do not fundamentally alter their *capabilities*. On the DMS side, the interface to query compilation systems is still SQL with ACID guarantees on all tables, which comes with the overhead that we shown in our experiment in Section 2. On the compiler side, Spark RDD query plans still need to be optimized by hand.

Reducing Friction between the two. There is, however, some work that accepts the strengths and shortcomings of each of the systems and attempts to use knowledge about one to optimize the other. An interesting example of that is Sloth [14], an approach to holistic optimization that defers the issuing of queries in an application as much possible. This enables optimizations such as fusion and even complete elimination of calls to the database.

3.2 Challenges

Despite some early successes, traditional DMSs and compilers are, at their core, still quite different, and their unification is not a trivial task. Let us next walk through the challenges that we see in this endeavor.

We see three aspects of these systems that need unification: their model of intermediate state, their model of computation and their model of persistence. These aspects correspond directly to the three components of a computer system: RAM, CPU and disk. Both compilers and DMSs have these aspects, but they implement fundamentally different approaches. We discuss them below and outline approaches to their unification.

Unifying the Model of Intermediate State. The model of intermediate state describes the format of intermediate (i.e., volatile) data structures that are passed between operators. For relational DMSs, the state model is a relation. This model is, arguably, the defining criterion of relational DBMSs. On the compiler side, there is no consensus on an intermediate representation. The three most prominent approaches are: registers, an operand stack and a heap.

Most compilers have a notion of a heap, i.e., an unstructured memory region that allows random access. It is shared among all threads and is usually the target for pointer arithmetics. These two aspects make it hard to reason about the state of the heap and prevent many program optimizations such as common subexpression elimination or write-combining. Therefore, most compilers use one of the other two means of maintaining state: registers, as in the case of LLVM or V8, and a stack, as in the case of .Net CLR and WebAssembly Bytecode. The Hotspot JVM implements both.

While LLVM has a lot of momentum right now, it is unclear if pure register machines are a good fit for the requirements of DMSs. Register machines limit the size of the state to a compile-time constant (LLVM provides an arbitrarily large but finite set of registers). Any state that grows to a size that is unknown (at compile-time) has to be maintained in the heap. As we mentioned earlier though, heap-allocated state prevents certain key optimizations. In addition,

stack machine instruction sets are known to yield faster interpreters, which makes them attractive for "small" queries that usually do not benefit from the effort of compiling all the way down to executable code. This aspect is of particular importance to DMSs that need to support interactive queries.

Given these constraints, the question of the right model for state merits closer investigation than it has received from either the DMS or the compiler communities. In fact, it seems likely that a unified state model for compilers and databases would be a stack rather than a register machine.

Unifying the Computational Model. An area in which DMSs are severely restricted is their support for efficient iterative processing. On the one hand, an advantage when optimizing relational algebra plans is the lack of dynamically bounded iteration and recursion. As iteration creates work at runtime, it complicates cost-estimation and, thus, increases the complexity and cost for query optimization; on the other hand, iterative processing enables many real-world use-cases such as the computation of pagerank, gradient descent and other randomized algorithms.

While most major DMSs support the definition of recursive SQL queries, database optimizers merely optimize their non-recursive parts. Such optimization closely resembles the one that compilers perform on the Single Static Assignment (SSA) form[6] of programs. Both share the same limitations, i.e., that no optimizations are performed across loop iterations.

To perform cross-loop optimizations, compilers usually represent code in higher-level forms such as the polyhedral model [24], which is a nested-loop representation with formalized constraints on the loops. While extracting a polyhedral representation from SSA form is possible, it is non-trivial and computationally expensive [17]. This is not aligned with DMS requirements for low compilation times.

It is, thus, worthwhile to look beyond the current trend into other kinds of recursion models. One likely candidate is the Continuation Passing Style (CPS) that is commonly implemented as an IR in functional language compilers. Conceptually, CPS is more expressive than SSA and closer to the recursive (functional or declarative) programming model that dominates in data processing systems. Therefore, we plan to study the use of such alternative computational models in DMS backends.

Unifying the Model of Persistence. Persistence (also known as durability) describes how the system manages data that needs to be available even after a transient failure. It is the core domain of DMS but only auxiliary to compilers: where (relational) DMS provide a single, portable persistance model with semantic guarantees (e.g., for updates), general-purpose programming languages rely on system-specific APIs (e.g., for disk access) or (more recently) portable abstractions such as Protocol Buffers [42] or Thrift [37]. While these offer portable durability, they do not offer any kind of guarantees in terms of integrity, consistency or isolation. These are offloaded to developers to implement on top of a persistence framework. Support for these properties could be generated by the compiler: it could enforce consistency by instrumenting the generated code to check for violations. In fact, the work on object-oriented databases and persistent object-stores [6] demonstrated that this is possible in standalone systems.

Most of the related research must be revisited in the context of modern hardware such as multicore CPUs, GPUs or non-volatile memory. Currently it does not address the need for highly-performant code with efficiency techniques such as pointer arithmetics, vectorization or hardware acceleration.

[6]basically a dataflow representation with goto

3.3 Opportunities

Having outlined the challenges in the unification of compilers and DMSs, let us discuss some of the created opportunities.

Recovering state of general programs. State recovery is a key reason why users opt to use DMSs, yet not all data needs to pay the price/overhead for recoverability. In databases, data recovery is typically achieved through write-ahead logging [31] or shadow paging. It is also, one of the hardest problems that DMSs solve and, thus, non-trivial to implement by hand.

As we illustrated by the online learning example in Section 2, not all parts of an application's state need to be "recoverable", as they can be easily recomputed by the application. In this case, the overhead added by the recovery protocol is likely the main source of inefficiency of the application's implementation on top of PostgreSQL. To achieve better performance and the desired recoverability, the application should be allowed to specify that the inserted base data must be recoverable, while the model does not. For convenience, developers would likely specify a customized recovery handler function (in this case, retraining the model of the base data) rather than manually running the recovery strategy from outside the DMS. One could even imagine a situation in which some attributes or tuples of a table would be recovered using a custom handler while others are recovered by the system.

Cost as a first class citizen. Cost estimation was until recently a mere supporting function in compilers as well as databases. It plays a much bigger role in databases now because it steers the exploration of the plan space during query optimization [28,40]. This is fundamentally different in compilers because the set of correct programs is theoretically infinite and practically very large, which makes it infeasible to explore exhaustively. Cost models are usually used to make localized decisions such as loop unrolling or parallelization [16]. A notion of the overall cost of a program is, if at all, acquired by running it on sample input.

With the advent of *platform-as-a-service* and *serverless* computing, cost-estimation of generic programs is becoming increasingly important: in a serverless execution setup, a cloud provider agrees to execute a generic function at an agreed-upon price. Estimating the cost of the function is, thus, important to the cloud provider. Database-inspired cost models can help fulfil that requirement.

Unifying the Model of Adaptivity. Adaptivity, i.e., learning from previous executions of a program to improve the efficiency the current one, is an area in which both DMSs and compilers have notable features. Both, however, play to their respective strengths: DMSs either gather knowledge about the data (see, e.g., the learning optimizer [41]) or adaptively build indices (see, e.g., database cracking [19]). Compilers implement adaptivity in one of two ways: for statically compiled languages, they gather execution profiles when running an unoptimized version of the program and feed the profile to a second, optimized compilation run. This process is commonly referred to as Profile-Guided Optimization [18]; in just-in-time compiled languages, the compilation/optimization process itself is subject to adaptivity: only parts of the code that are executed frequently are compiled/optimized. Most compilers represent code differently depending on the level of optimization. This way, the compiler spends less time on code that contributes little to overall runtime.

For adaptivity, we see potential for knowledge transfer in both directions. Some DMSs have already started adopting the idea of adaptive compilation [23]. They have yet to embrace the idea of changing the representation (it is currently always LLVM bytecode). This forces them to compile and optimize at a very coarse granularity (query plan fragments that can be as large as the en-

```
int process(int* input, unsigned long size) {
  int result = 0;
  for(auto i = 0; i < size; i++)
    result += input[i];
  return result;
};
```

Listing 1: Simple Aggregation in C

```
extern int* runs;
extern int* lengths;

int process(int* input, unsigned long size) {
  int result = 0;
  for(auto i = 0ul; i < size; i++)
    result += runs[i] * lengths[i];
  return result;
};
```

Listing 2: Aggregation of Run-length-encoded data in C

tire plan). If the system explores different query plans (see [46]), this quickly becomes expensive. There are, thus, gains to be made by integrating adaptive compilation into DMSs. On the other side, compilers could be more aggressive in building (and even persisting) auxiliary data structures such as indices. We envision executable binaries contain a "scratchpad-memory" that can be modified by the application to pass data to its next invocation. For example, a UNIX `grep` could store the probability of finding matching lines in a text file over the last n runs and use this information to allocate buffers.

4. EXAMPLE: COMPRESSION

Let us illustrate our approach using a textbook data management technique that has, to the best of our knowledge, never been applied in the context of general-purpose compilers: data compression (specifically run-length encoding) and operating directly on that compressed data. In the context of DMSs, it requires a developer to implement operators that work on compressed data; in the context of general-purpose programming languages, it could be implemented as a library that implements compressed data structures as well as primitives over them. In contrast, we envision a deeper integration with the host language to enable the transparent use of compression. To illustrate this, consider the code in Listing 1: it calculates a simple (ungrouped) sum over an input array. The same operation over run-length-encoded data is shown in Listing 2. Assuming a good compression ratio, we expect this implementation to be significantly faster. However, rewriting a complex program on uncompressed data into one of compressed is non-trivial. We therefore aim to apply this (admittedly complex) optimization automatically. For our example, we implement it in LLVM [25].

Following common practice, we do not apply this optimization at the source code level but at the level of the dataflow IR. In the case of LLVM, this IR follows the *static single assignment* (SSA) paradigm. The SSA representation of the aggregation program is displayed in Listing 3. Note that while there is a fair amount of boilerplate, the crucial piece of code is the loading of the input values in lines 5 and 6, and their accumulation in line 7. Transforming this program into one that operates on run-length-encoded data (the one displayed in Listing 4) is not hard: we merely need to replace the accumulation with one that multiplies each value with the length of its run first (line 12). This naturally requires the loading of the current `run` and its `length` in lines 8 and 10, respectively. This, in turn, requires the loading of the compressed `runs` and `lengths`

```
1  define i32 @process(i32*, i64) {
2  loop:
3    %i = phi i64 [ 0, %2 ], [ %nextI, %loop ]
4    %resultBefore = phi i32 [ 0, %2 ], [ %result, %loop ]
5    %inputValuePtr = getelementptr inbounds i32, i32* %0, i64 %i
6    %inputValue = load i32, i32* %inputValuePtr
7    %result = add nsw i32 %inputValue, %resultBefore
8    %nextI = add nuw i64 %i, 1
9    %end = icmp eq i64 %nextI, %1
10   br i1 %end, label %exit, label %loop
11 exit:
12   ret i32 %result
13 }
```

Listing 3: Simple Aggregation in LLVM IR

```
1  define i32 @process(i32*, i64) {
2    %runs = load i32*, i32** @runs
3    %lengths = load i32*, i32** @lengths
4    br label %loop
5  loop:
6    %i = phi i64 [ 0, %2 ], [ %nextI, %loop ]
7    %resultBefore = phi i32 [ 0, %2 ], [ %result, %loop ]
8    %runValuePtr = getelementptr inbounds i32, i32* %runs, i64 %i
9    %runValue = load i32, i32* %runValuePtr
10   %lengthValuePtr = getelementptr inbounds i32, i32* %lengths, i64 %i
11   %lengthValue = load i32, i32* %lengthValuePtr
12   %inputValue = mul nsw i32 %lengthValue, %runValue
13   %result = add nsw i32 %inputValue, %resultBefore
14   %nextI = add nuw i64 %i, 1
15   %end = icmp eq i64 %nextI, %1
16   br i1 %end, label %exit, label %loop
17 exit:
18   ret i32 %result
19 }
```

Listing 4: Aggregation of Run-length-encoded Data in LLVM IR

arrays in lines 2 and 3 (we omitted the code for compressing the input for brevity).

While this example is simplified and, thus, not particularly challenging, it illustrates the direction of our work: we apply a technique that is well-established in the realm of data management and integrate it into an existing compilation framework. The result is a compiler that automatically applies the optimization when applicable. This kind of optimizations is fundamentally different from those traditionally applied in general-purpose compilers in that it exposes the risk of hurting performance (for example, if the compression rate is poor). Estimating and bounding costs for data-intensive algorithms, however, has been one of the traditional strengths of DMSs, which will allow us to apply this optimization only when beneficial.

5. CONCLUSION

Data management systems and compilers must merge. It will free developers from the architectural challenge of selecting the right platform for their needs. It will allow infrastructure and cloud providers to maximize utilization and charge appropriate prices. Finally, it will allow database researches to work on "the hollow middle" again—core database topics, albeit applied in a more general context. In this paper, we argued the need for such a merge using experimental as well as architectural evidence. However, the road to a unified data management system/compiler system holds a number of challenges. We discussed the three core unification challenges (computation, state and persistence) and how they can be addressed. We also showed a number of opportunities that arise from a unified approach.

In conclusion, we argue that the time has come to claim the no

man's land between compilers and databases and leverage decades of database research in the context of general-purpose programming language compilers.

6. REFERENCES

[1] S. R. Agrawal et al. A Many-core Architecture for In-memory Data Processing. MICRO-50, pages 245–258, 2017.

[2] A. Ailamaki et al. DBMSs on a Modern Processor: Where Does Time Go? VLDB, pages 266–277, 1999.

[3] A. Ailamaki et al. Weaving Relations for Cache Performance. In *PVLDB*, pages 169–180, 2001.

[4] A. Alexandrov et al. The stratosphere platform for big data analytics. *VLDB*, 2014.

[5] ARM. ARM Security Technology: Building a Secure System using TrustZone Technology. Technical report, 2009.

[6] M. Atkinson and R. Morrison. Orthogonally persistent object systems. *VLDB*, 1995.

[7] C. Balkesen et al. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1), 2013.

[8] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the Memory Wall in MonetDB. *Commun. ACM*, 51(12):77–85.

[9] M. J. Carey et al. The exodus extensible dbms project: An overview. *Readings in object-oriented database systems*, 1990.

[10] U. Cetintemel et al. S-store: a streaming newsql system for big velocity applications. *VLDB*, 2014.

[11] S. Chakrabarti et al. Intel&Reg; Software Guard Extensions (Intel&Reg; SGX) Architecture for Oversubscription of Secure Memory in a Virtualized Environment. HASP, pages 7:1–7:8, 2017.

[12] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *VLDB*, 2000.

[13] T. Chen et al. TVM: an automated end-to-end optimizing compiler for deep learning. In *OSDI*, pages 578–594, 2018.

[14] A. Cheung, S. Madden, and A. Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). *TODS*, 2016.

[15] A. Crotty et al. Tupleware:" big" data, big analytics, small clusters. In *CIDR*, 2015.

[16] Z.-H. Du et al. A cost-driven compilation framework for speculative parallelization of sequential programs. *ACM SIGPLAN Notices*, 2004.

[17] T. Grosser et al. Polly-polyhedral optimization in llvm. In *IMPACT*, 2011.

[18] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *TOPLAS*, 1996.

[19] S. Idreos et al. Database cracking. In *CIDR*, 2007.

[20] N. P. Jouppi et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, June 2017.

[21] V. Kiriansky, Y. Zhang, and S. Amarasinghe. Optimizing indirect memory references with milk. In *PACT*. IEEE, 2016.

[22] Y. Klonatos et al. Building efficient query engines in a high-level language. *VLDB*, 2014.

[23] A. Kohn, V. Leis, and T. Neumann. Adaptive execution of compiled queries. ICDE, 2018.

[24] L. Lamport. The parallel execution of do loops. *CACM*, 1974.

[25] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[26] Y. Li, I. Pandis, R. Müller, V. Raman, and G. M. Lohman. NUMA-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[27] J. Malcolm et al. Arrayfire: a gpu acceleration platform. In *MSDSA*, 2012.

[28] S. Manegold, P. Boncz, and M. L. Kersten. Generic database cost models for hierarchical memory systems. In *VLDB*. VLDB Endowment, 2002.

[29] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Database Architecture for the New Bottleneck: Memory Access. *The VLDB Journal*, 9(3):231–246, 2000.

[30] E. Meijer, B. Beckman, and G. Bierman. Linq: reconciling object, relations and xml in the. net framework. In *SIGMOD*. ACM, 2006.

[31] C. Mohan et al. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *TODS*, 1992.

[32] D. G. Murray et al. Naiad: a timely dataflow system. In *SOSP*. ACM, 2013.

[33] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *VLDB*, 2011.

[34] S. Palkar et al. Weld: A common runtime for high performance data analytics. In *CIDR*, 2017.

[35] H. Pirk et al. Voodoo-a vector algebra for portable database performance on modern hardware. *VLDB*, 2016.

[36] O. Polychroniou and K. A. Ross. A Comprehensive Study of Main-memory Partitioning and Its Application to Large-scale Comparison- and Radix-sort. In *SIGMOD*, pages 755–766, 2014.

[37] A. Prunicki. Apache thrift. Technical report, Technical report, Object Computing, Inc, 2009.

[38] J. Ragan-Kelley et al. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Notices*, 2013.

[39] N. Satish et al. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *SIGMOD*, pages 351–362, 2010.

[40] P. G. Selinger et al. Access path selection in a relational database management system. In *SIGMOD*. ACM, 1979.

[41] M. Stillger et al. Leo-db2's learning optimizer. In *VLDB*, 2001.

[42] K. Varda. Protocol buffers: Google's data interchange format. *Google Open Source Blog, Available from Jul*, 2008.

[43] J. Wassenberg and P. Sanders. Engineering a multi-core radix sort. In *Euro-Par*, pages 160–169. Springer, 2011.

[44] M. Zaharia et al. Spark: Cluster computing with working sets. *HotCloud*, 2010.

[45] M. Zaharia et al. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP*, pages 423–438, 2013.

[46] S. Zeuch, H. Pirk, and J.-C. Freytag. Non-invasive progressive optimization for in-memory databases. *VLDB*, 2016.

[47] D. Y. Zhu et al. Tainteraser: Protecting sensitive data leaks using application-level taint tracking. *SIGOPS*, 2011.