INTERFACES FOR EFFICIENT SOFTWARE COMPOSITION ON MODERN HARDWARE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Shoumik Palkar
June 2020

This dissertation is online at: http://purl.stanford.edu/pn706fw7190

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Matei Zaharia, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christos Kozyrakis

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Keith Winstein

Approved for the Stanford University Committee on Graduate Studies.

Stacey F. Bent, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

For decades, developers have been productive writing software by composing optimized libraries and functions written by other developers. Though hardware trends have evolved significantly over this time—with the ending of Moore's law, the increasing ubiquity of parallelism, and the emergence of new accelerators—many of the common *interfaces* for composing software have nevertheless remained unchanged since their original design.

This lack of evolution is causing serious performance consequences in modern applications. For example, the growing gap between memory and processing speeds means that applications that compose even hand-tuned libraries can spend more time transferring data through main memory between individual function calls than they do performing computations. This problem is even worse for applications that interface with new hardware accelerators such as GPUs. Though application writers can circumvent these bottlenecks manually, these optimizations come at the expense of programmability. In short, the interfaces for composing even optimized software modules are no longer sufficient to best use the resources of modern hardware.

This dissertation proposes designing new interfaces for efficient software composition on modern hardware by leveraging *algebraic properties* intrinsic to software APIs to unlock new optimizations. We demonstrate this idea with three new composition interfaces. The first interface, *Weld*, uses a functional intermediate representation (IR) to capture the parallel structure of data analytics workloads underneath existing APIs, and enables powerful data movement optimizations over this IR to optimize applications end-to-end. The second, called *split annotations (SAs)*, also focuses on data movement optimization and parallelization, but uses annotations on top of existing functions to define an algebra for specifying how data passed between functions can be partitioned and recombined to enable cross-function

pipelining. The third, called *raw filtering*, optimizes data loading in data-intensive systems by redefining the interface between data parsers and query engines to improve CPU efficiency.

Our implementations of these interfaces have shown substantial performance benefits in rethinking the interface between software modules. More importantly, they have also shown the limitations of existing established interfaces. Weld and SAs show that a new interface can accelerate data science pipelines by over 100× in some cases in multicore environments, by enabling data movement optimizations such as pipelining on top of existing libraries such as NumPy and Pandas. We also show that Weld can be used to target new parallel accelerators, such as vector processors and GPUs, and that SAs can enable these speedups even on black-box libraries without any library code modification. Finally, the I/O optimizations in raw filtering show over 9× improvements in end-to-end query execution time in distributed systems such as Spark SQL when processing semi-structured data such as JSON.

Acknowledgements

I owe a debt of gratitude to a number of people for making this dissertation possible, and for making graduate school an exhilarating and enduring time in my life.

First and foremost, my advisor Matei Zaharia has been influential not only on this dissertation, but also on my time in graduate school overall. I look back fondly on the early days of my Ph.D. where we would spend hours whiteboarding various interfaces and designs for Weld. Over the years, he continued to provide astute advice and clarity on everything from how to structure a piece of code to how a particular argument would be best presented in a research paper. Though rather obvious in hindsight, I was also thrilled to discover he was a gifted programmer, and I would like to think that his insights and mentorship have helped me hone that passion of mine. Beyond his academic mentorship, Matei has also been a role model in humility, humor, and kindness. He has never hesitated to take my labmates and I out for a coffee or to discuss any issues or research problems over a weekend at his favorite local boba tea shop. His propensity for puns has become infamous in our lab. He would often send me Slack messages about news related to Nutella, because he knows that both he and my wife Paroma share an understandable affection for it. I hope to be able to turn to Matei for his mentorship for many years to come.

In the FutureData lab group, I had the pleasure of being in one of the most amicable and supportive social environments I could hope for throughout my Ph.D. My labmates are not only brilliant and talented researchers, but also, I hope, lifelong friends. Deepak Narayanan and I would gossip about everything from the abstractions in Weld to Formula One. Some of my most memorable moments in graduate school have been discussing existential topics such as climate change with him when sharing a hotel room at a conference. My office mates in Gates 432—Deepak, Firas Abuzaid, James Thomas, and by courtesy, Pratiksha

Thaker—have been invaluable not only for bouncing around research ideas but also for the occasional gossip session to escape the stresses of grad school. Sahaana Suri and Geet Sethi have always been willing to grab ramen with me, despite the perpetually long waits at Ramen Nagi. I also want to thank Peter Bailis and my other labmates for their continual feedback and support: Cody Coleman, Daniel Kang, Deepti Raghavan, Edward Gan, Kaisheng Tai, Keshav Santhanam, Kexin Rong, Peter Kraft.

At various points in my Ph.D., I collaborated with folks at Databricks to gain a better understand of how some of the work in this thesis could be applied in an industry setting. Alex Behm, Herman van Hovell, and Reynold Xin provided me with countless insights and mentorship on database and systems design, and gave me a unique perspective on what it takes to make research software usable at scale. Through a Databricks internship, I also took the opportunity to visit the Databricks office in Amsterdam. I met Prof. Peter Boncz and the database group at CWI while there as well. Prof. Boncz gave invaluable (and often critical!) feedback on the Weld work, for which I am grateful.

My friends Akshay Narayan, Deepti Raghavan, Sagar Karandikar, and Sheevangi Pathak have been with me throughout grad school, providing a much needed respite from research when I needed it. Our daily conversations, which over the last five years have occurred over Hangouts, Messenger, Slack, Signal, Keybase, and Messages, have been a consistent source of joy. Friendships like these are what make topics as mundane as the merits of checking in bags on an airplane as hilarious as they are. More than anything, I will always remember the tacit traditions we established. Two that I look back on fondly are the weekly Bollywood movie nights Paroma and I would have with Akshay and Deepti as we sat through the COVID-19 quarantine, and the pilgrimages to our favorite spots from undergrad (Asha Tea House, Cheeseboard, Cupcakin' Bakeshop) when we visited Sagar in Berkeley.

Yuval Gannot has been one of my closest friends since we became roommates freshman year at Berkeley. Luckily, we both ended up at Stanford and I was fortunate enough to have him as my roommate for an additional three years. The time we spent outside of the lab together at Seven Oaks will be some of my most memorable moments of graduate school, from watching strange YouTube videos at 2AM to observing Yuval cook everything from eight-hour meat sauces to popcorn in a pot.

I always felt I was lucky to attend graduate school a mere 20 minutes away from where I

grew up. Visits to my parents—both for delicious home-cooked meals and laundry—were frequent. In all seriousness, there is no chance I would have gotten through grad school without their unwavering support. I am also fortunate to have the support of my sister Ishani, my aunt, Trupti, my uncle, Sourja, and my cousins Shreya and Tvisha. I not only have the comfort of being able to treat my aunt and uncle as both elders and friends, but also the privilege of calling them my inspiration and my role models.

Finally, graduate school is as memorable as it was because of my best friend and wife, Paroma Varma. In many ways, graduate school began with Paroma, since we first met during MIT's grad school visit days in 2015. Though we spent one year apart, fate brought us back together and we ended up working on the same floor and in the same lab (DAWN), only four offices away from each other. This was all despite the two of us being in completely different fields to begin with (Paroma started her Ph.D. as an electrical engineering student working on computational photography). Paroma is primarily responsible for giving me a life outside of the lab. She has been with me, by lending her unwavering support, through the highs and the lows and the stresses and successes that come with this experience. I certainly could not have done it without her.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

For decades, developers have written software by composing functionality written by other developers. Even the earliest pioneers of computing recognized that software composition was a key enabler of programmer productivity, with figures such as Dijkstra [62] and others [127, 159, 187] advocating for software modularization and reuse.

It has long been known that the interfaces between software modules and the features of a programming language can wildly influence the performance of an application. As a simple example, the `std::sort()` function in C++ is known to outperform the C `qsort()` function because language features such as templates allow a compiler to better inline comparisons between elements in C++, as opposed to making a function call to compare elements in C [44]. This is despite the significant optimizations that have been made to `qsort()` for modern systems. Nevertheless, most popular programming environments—ranging from C to Java to Python—rely chiefly on the optimization of individual modules to achieve good performance. As a result, years of work have gone into developing fast libraries targeting numerous domains—ranging from relational algebra [157, 197] to image processing [87, 151, 176] to machine learning [2, 173, 188]—while thousands of complex applications have been built using APIs to these existing libraries. Indeed, this has led to a certain dichotomy among programmers: expert *library developers* take charge of writing software packages with high-level APIs, while *end-users* use these APIs to compose these APIs into complex applications (e.g., a machine learning pipeline written in Python that mixes linear algebra, relational algebra, and statistical modeling code).

Unfortunately, trends in hardware are eroding the performance of applications that compose even optimized modules. For example, even modern applications that compose hand-tuned libraries such as BLAS [116] can spend more time exchanging data between functions through main memory than executing function code. This is not an artifact of the libraries themselves, but rather of the *function call interface* used to compose software in many languages. With this interface, each function independently loads values from main memory, computes results, and writes these results back to main memory in order to exchange data with other functions.

As an example of the resulting bottleneck, the code from Listing 1.1 marked *Version 1* shows a snippet from a program that uses Intel MKL [132]—a highly optimized linear algebra and numeric computing library—to evaluate the Black Scholes options pricing model [22]. Each function in MKL is meticulously optimized and exploits parallelism via SIMD vectorization [192] and multi-threading. However, because the functions exchange data via pointers to large in-memory buffers, this application is bottlenecked by memory bandwidth to the CPU rather than the main computations. As a result, the MKL implementation executes up to 3× *slower* than using an unoptimized and unvectorized implementation with a single loop, as illustrated by the code shown under *Version 2* in Listing 1.1.

For this application, we can achieve the best performance when using the optimized MKL functions but circumventing the limitations of the function call interface. The code under *Version 3* in Listing 1.1 illustrates this. This version tiles the first loop by only calling MKL functions on chunks of the inputs that collectively fit in the CPU caches, rather than on buffers that will spill to main memory after each function call. By keeping values in the CPU caches and leveraging the optimized functions, this implementation outperforms the naive loop by 1.5× and the original MKL implementation by 4.5× on 16 threads. The tiling circumvents the main bottleneck of the function call interface: repeated loads and stores from main memory that cause stalls in the CPU[1]. Unfortunately, this performance comes at the cost of mixing application logic with manually constructed optimizaton code: more complex applications require optimizations that are proportionally more complex as well.

Applications that use emerging hardware accelerators have even worse performance

[1]Because this is a widely understood phenomenon, even libraries built by Intel (e.g., MKL-DNN, Intel's deep learning library [133]) use this blocking technique when composing MKL operators rather than invoking them directly.

Listing 1.1 Snippet from the Black Scholes options pricing model implemented using Intel MKL (Version 1) a naive loop (Version 2), and MKL with cache blocking (Version 3).

Version 1: Implementation with optimized MKL operators. The inputs are `double` arrays with `len` elements. Each operator runs on multiple threads.

```
vdLog1p(len, d1, d1);          // d1 = log(d1)
vdAdd(len, d1, tmp, d1);       // d1 = d1 + tmp
vdDiv(len, d1, vol_sqrt, d1);  // d1 = d1 / vol_sqrt
vdSub(len, d1, vol_sqrt, d2);  // d2 = d1 - vol_sqrt
vdMuli(len, d1, invsqrt2, d1); // d1 = d1 * invsqrt2
vdErf(len, d1, d1);            // d1 = erf(d1)
...
```

Version 2: Naive loop parallelized with OpenMP. The loop is not vectorized by the compiler due to unsupported operations. Compared to Version 1, this performs up to 3x better on 16 threads on a modern Intel processor.

```
#pragma omp parallel for
for (int i = 0; i < len; ++i) {
  d1[i] = (log1p(d1) + tmp) / vol_sqrt;
  d2[i] = d1 - vol_sqrt;
  d1[i] = erf(d1 * invsqrt2);
  ...
}
```

Version 3: MKL operators with "blocked" calls. Functions only exchange `BATCH_SIZE` elements, which collectively fit in the CPU caches. This reduces loads from main memory by making better use of the memory hierarchy. The outer loop is parallelized over batches.

```
#pragma omp parallel for
for (int i = 0; i < len - BATCH_SIZE; i += BATCH_SIZE) {
  vdLog1p(BATCH_SIZE, d1, d1);          // d1 = log(d1)
  vdAdd(BATCH_SIZE, d1, tmp, d1);       // d1 = d1 + tmp
  vdDiv(BATCH_SIZE, d1, vol_sqrt, d1);  // d1 = d1 / vol_sqrt
  vdSub(BATCH_SIZE, d1, vol_sqrt, d2);  // d2 = d1 - vol_sqrt
  vdMuli(BATCH_SIZE, d1, invsqrt2, d1); // d1 = d1 * invsqrt2
  vdErf(BATCH_SIZE, d1, d1);            // d1 = erf(d1)
  ...
}
```

implications when using existing interfaces for software composition. For example, mixing functions from a contemporary accelerator library such as PyTorch [173] with a traditional CPU-based library such as NumPy will repeatedly move data across an expensive off-chip link and can cause out-of-memory issues on the accelerator. In short, the existing *interfaces*

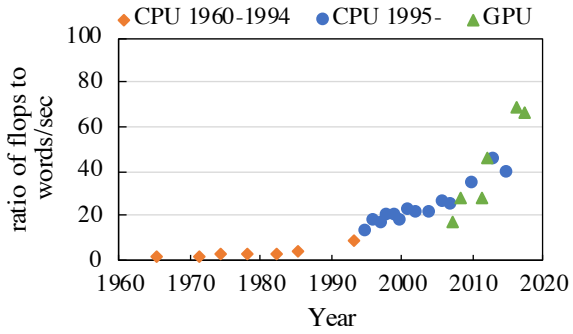


Figure 1.1: Ratio of FLOPS to memory bandwidth using data from online datasheets [91–95, 163, 164, 166] (CPUs, 1965-1994), the STREAM benchmark [129] on commercial server configurations (CPUs, 1995-present), and NVIDIA GPGPU specs [148]. Computational throughput has outpaced memory bandwidth across the board.

used to compose software functionality pose serious performance challenges for modern applications. Two hardware trends exacerbate this observation over time.

First, *moving data* between individual software components has grown increasingly expensive, in part due to the growing gap between *memory access speeds* and *computational throughput* in modern hardware. To illustrate this, Figure 1.1 shows the ratio of FLOPS to memory bandwidth of commercial computer systems over the last several decades. The plot shows a tenfold increase in this ratio over time, indicating that modern applications are increasingly sensitive to data movement between main memory and the CPU. This is especially true for *data-intensive* domains such as big data analytics and machine learning, where applications spend few cycles of processing time per-byte of data. Several recent systems (including ones presented in this dissertation) [55, 110, 145, 154, 220] have identified that data science pipelines that use contemporary libraries such as Pandas [157], NumPy [146], and TensorFlow [2] can spend 90% of their execution loading and moving data.

Second, the rapid introduction of new *hardware accelerators* makes efficient composition even more difficult, since applications that wish to combine traditional CPU-only code with accelerator code must manage new concerns such as cross-device data transfer and limited device memory. With the emergence of new libraries such as PyTorch [173] and RAPIDS [179]—which aim to democratize programming these new accelerators by providing APIs in high level languages such as Python—the line between accelerator and CPU code is blurred, and interfaces that allow efficient composition of existing CPU code

and new accelerator code becomes even more important. Together, these trends erode a fundamental assumption of software composition: that composing optimized components will produce an optimized application.

One strawman solution to these trends is to ask all developers to adopt monolothic systems that can offer end-to-end optimizations under a single framework, such as Spark [224], Dryad [222], or others [38, 109]. While such systems provide significant performance benefits by taking a unified view of an application, they leave behind years of effort spent in developing robust optimized software libraries, as well as thousands of legacy applications that rely on these libraries already. In addition, these systems are complex to develop and maintain compared to domain-specific libraries. Another possible solution is to circumvent the limitations of existing interfaces by foregoing code reuse or manually implementing optimizations such as pipelining chunks of data between functions. For example, the optimized *Version 3* implementation from Listing 1.1 applies pipelining over already-optimized library functions to reduce data movement. However, these techniques come at the cost of *programmability*. This is especially true as developers shift toward developing even complex distributed applications in high-level languages such as Python, R, and Scala. Having to reason about performance under composition obviates many of the benefits of using these abstractions at all. To summarize, existing software composition interfaces face the following challenges:

1. **The increasing costs of moving data:** Data movement between optimized software components has become relatively more expensive as the gap between memory access speeds and computation throughput has increased for new hardware.

2. **The advent of hardware accelerators:** Accelerators pose new challenges under composition, such as managing limited memory and moving data between devices.

3. **The need to support existing libraries and applications:** Programmers have spent decades developing libraries, frameworks, and applications using existing infrastructure, and would like to continue using these components in the future.

4. **The sacrifice of programmability:** Developers today must sacrifice programmability in the name of extracting the best performance under composition.

As a step toward addressing these challenges, this dissertation argues that we need a new set of *interfaces* for efficient software composition on modern hardware. The goal of a new set of interfaces is to enable developers to once again write optimized software applications by combining optimized software components, while still using existing high-level abstractions (e.g., an existing library API). The main idea of this dissertation is that we can expose certain *algebraic properties* in new composition interfaces to enable new optimizations, sometimes in *existing software*. For example, an interface can expose the fact that computations in a library (e.g., relational algebra functions in a DataFrame library such as Pandas) can be represented as a simple set of data-parallel primitives around black-box functions: an underlying system can then implement optimizations such as loop fusion over this representation to improve memory locality. As another example, an interface over existing library code such as BLAS can mark certain functions as commutative and their inputs as amenable to partitioning, enabling automatic parallelization and cross-function pipelining in some cases. In both cases, the interface enables a new optimization under composition, but maintains the APIs that thousands developers are already familiar with. In some cases, we show that optimizations such as parallelization or data pipelining are possible even without any modifications to existing libraries.

Thesis statement: *Interfaces that expose and leverage algebraic properties of existing software APIs can enable optimizations that better utilize the most precious resources of modern hardware under composition.*

In the next section, we discuss the contributions of this dissertation and provide a summary of results. We conclude with an outline of the remainder of the dissertation.

1.1 Contributions and Results Summary

We argue that it is possible to build new interfaces for efficient software composition on modern hardware by leveraging algebraic properties intrinsic to software APIs to unlock new optimizations. We demonstrate this with three new interfaces and systems, which are summarized in Table 1.1. The first, *Weld*, uses a functional intermediate representation

System	Interface	Optimizations Enabled
Weld (Chapter 2)	Functional intermediate representation with lazy runtime API	Pipelining via loop fusion, parallelization, other compiler optimizations
Split annotations (Chapter 3)	Annotations over unmodified library functions with API to split and merge data	Cache-level pipelining and parallelization
Sparser (Chapter 4)	Filtering functions that discard bytes with a false positive rate	Discarding data before parsing it in a query engine

Table 1.1: Interfaces described in this dissertation.

to capture the parallel structure of data analytics workloads under existing APIs, and applies algebraic compiler-style rules over this representation to enable cross-function data movement optimizations and parallelization. The second, called *split annotations*, also focuses on data movement optimization and parallelization, but takes a different design point where both end users and library developers do not need to modify existing code. This interface uses annotations on top of existing functions to define an algebra for specifying how data passed between functions can be partitioned and recombined in order to enable cross-function pipelining. The third looks at a different problem to demonstrate our thesis's generality: the technique, called *raw filtering*, optimizes data loading in data analytics systems by redefining the interface between *data parsers* and *query engines*. Our systems implementing these techniques all show that redefining the interface between different software components can lead to significant performance gains in real applications.

1.1.1 Weld: A Common Runtime for Data Analytics

Our first system focuses on the limitations of a foundational software composition interface: *the function call*, in which control jumps to a new program point, and functions exchange data via references to in-memory values. The underlying assumption behind this interface has always been that *exchanging data* between functions through memory is faster than *processing the data*, as was the case when subroutines first appeared for the ENIAC in 1945 [130]. For decades, function calls have provided a familiar, general, and efficient means of combining software. Today, 75 years after its introduction, hardware trends in

memory access speeds are challenging the basic assumptions of the function call interface.

We present a new system called *Weld*, an intermediate representation (IR) and runtime that captures the parallel structure of existing data-parallel library functions to enable compiler-style optimizations such as loop fusion, common subexpression elimination, and code generation in programs that combine many such functions. Unlike prior DSLs and IRs [32, 176, 201], Weld's IR is designed to apply the optimizations most important when composing computations from different domains: its referentially transparent, functional design prioritizes data movement optimization and parallelization, and the use of concepts such as linear types [216] allow library functions that have no knowledge of each other to construct fragments of IR that can be efficiently concatenated into a single program while efficiently enforcing cross-function memory safety.

Data-parallel libraries interface with Weld by using a *lazy runtime API* to submit fragments of IR code to the Weld runtime. When an IR program must be executed (e.g., when a lazy result needs to be sent over the network), an *optimizer* optimizes the IR program, generates parallel machine code, and executes the generated program. The optimizer leverages the algebraic properties of the IR (which in turn models the properties of the library API) to enable each optimization. The IR is structured in a way that makes it amenable for code generation targeting parallel architectures such as GPUs or vector processors. Interestingly, many data analytics libraries already implement their functions in a performance-oriented language such as C, so Weld provides an alternative way to implement these functions that enables efficient composition.

Libraries use Weld by replacing existing function implementations with ones that use the runtime API to generate Weld IR. Nevertheless, we show that common libraries can shift to using Weld *incrementally*, while inter-operating with functions unsupported by Weld. This allows library developers to support the most important functions in their libraries with Weld first, while maintaining compatibility with legacy code.

Overall, Weld's new lazy interface, combined with its new IR, has several advantages over traditional function calls that eagerly execute:

- **Efficient composition of independently-written library functions:** The properties of Weld's IR (e.g,. referential transparency, statically known call tree, parallel primitives for looping) prioritize data movement optimization and automatic parallelization

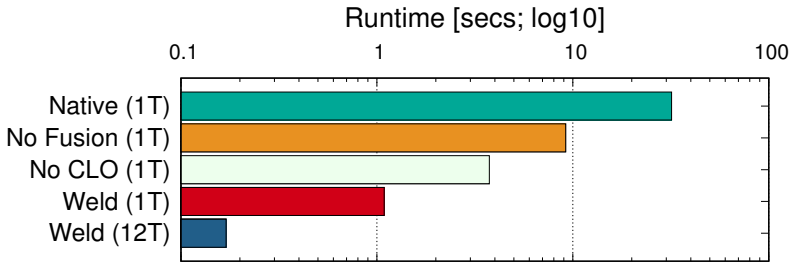


Figure 1.2: Performance of a city crime analysis data science workflow (log scale) in unmodified Pandas and NumPy (where only individual operators are written in C), Weld without loop fusion, Weld without cross-library optimization (CLO), Weld with all optimizations enabled, and Weld with 12 threads.

and allow treating the specific computation done on each value as a black-box. This enables generality across domains, as well as cross function optimizations such as loop fusion (which reduces data movement by keeping values in cache) via efficient pattern matching rules, even for parallel programs.

- **Incremental integration:** Weld is designed to be integrated into existing libraries incrementally, allowing Weld-enabled functions to inter-operate with legacy code without modification.

- **Native support for hardware accelerators:** Weld can target new hardware accelerators by generating accelerator-specific code (e.g., CUDA [47]) from its natively parallel IR.

In real workloads using popular libraries such as NumPy, Pandas, and TensorFlow, we show that Weld can improve end-to-end performance by up to two orders of magnitude by enabling optimizations for reducing data movement, eliminating common sub-expressions, and performing code generation. Figure 1.2 illustrates one such result from a data science pipeline using NumPy and Pandas, with and without Weld. Weld can optimize function calls both within each library, but also across libraries (cross-library-optimization, or "CLO"): together with automatic parallelization, Weld can speed up this workload by 180× on 12 threads. We also find that Weld's compiler can often generate code whose performance is competitive with more specialized systems (e.g., TensorFlow's XLA linear algebra compiler and HyPer's [141] SQL code generation).

Listing 1.2 SAs for three functions in Intel MKL.

```
@splittable(size: SizeSplit(size), a: ArraySplit(size),
  mut out: ArraySplit(size))
void vdLog1p(long size, double *a, double *out);

@splittable(size: SizeSplit(size), a: ArraySplit(size),
  b: ArraySplit(size), mut out: ArraySplit(size))
void vdAdd(long size, double *a, double *b, double *out);
void vdDiv(long size, double *a, double *b, double *out);
```

In this dissertation, we also provide a detailed analysis of *which* optimizations have the biggest impact on ad hoc data science workloads—the first study of its kind for this class of workloads—and find that cross-library optimization and data movement optimization are both critically important to achieve the best performance, especially in a parallel setting.

In many ways, Weld is a "clean-slate" approach that requires some redevelopment of existing libraries to achieve the best benefits: developers express the whole computation in the library using a constrained algebra (i.e., the IR). At the same time, this work shows that a significant number of workloads are most impacted by only a small number of optimizations: namely, parallelization and the ones that reduce *data movement*. Is there another design point in the space of interfaces that will allow us to achieve these benefits on existing, legacy APIs without rewriting library code?

1.1.2 Split Annotations: Optimizing Existing Data-Parallel Libraries

The second contribution of this dissertation is an abstraction called split annotations (SAs), which proposes an alternate interface to Weld for optimizing the function call interface for data-parallel libraries. SAs allow an *annotator*, which can be either the library developer or the end user of a library, to annotate functions in an existing library with information that specifies how to *split and merge* data passed into the function. This property allows a simple underlying runtime to pipeline cache-sized split pieces among functions in an application, and also allows the runtime to execute multiple copies of the function in parallel, each executing on split pieces. Unlike Weld, SAs do not require changes to existing library code.

Listing 1.2 shows examples of split annotations for functions in Intel's MKL [132] linear

algebra library. At a high level, annotators use the SA and a new abstraction called *split types* to define how to split each function argument into partitions that fit in the CPU's caches. For example, the annotator can define a split type `ArraySplit` to indicate that the array arguments be split into smaller, regularly-sized arrays. The split type `SizeSplit` then indicates that the size argument be split to represent the lengths of these arrays. The split types define a small algebra that define *compatibility* of the split pieces between two functions, and also implicitly guarantee properties such as commutativity. Annotators bridge the abstraction of the split type with code that performs the splitting (e.g., by offsetting into the array pointer) by implementing a *splitting API* for each split type.

The split annotations interface provides several advantages:

- **Automatic data movement optimization and parallelization:** Like Weld, SAs use a runtime and lazy evaluation to enable data movement optimization and parallelization. However, the annotations allow *automatically* capturing a lazy task graph rather than requiring the use of a lazy API, and use a simpler algebra to capture only the properties needed to enable these optimizations.

- **Ease of integration**: Because using SAs only requires writing annotations and not re-implementing library functions, they require little developer effort (usually less than one line of code on average per function in a library).

- **Ability to leverage already-optimized code:** Since SAs rely on existing library functions for execution, they leverage code that developers have already optimized.

We integrated SAs with a subset of functions in several existing libraries: NumPy [146], Pandas [157], spaCy [194], MKL [132], and ImageMagick [87]. Perhaps surprisingly, SAs can sometimes match or outperform domain-specific JIT compilers *just by parallelizing and reducing data movement* between functions and reusing optimized functions expert developers have already written. Figure 1.3 compares our system implementing SAs, *Mozart*, against Intel's MKL library and MKL integrated with Weld for the Black Scholes options pricing benchmark. SAs outperform MKL by 4×, and in this case, even outperform Weld's compiler by leveraging MKL's highly optimized compute kernels directly rather than compiling code from scratch.

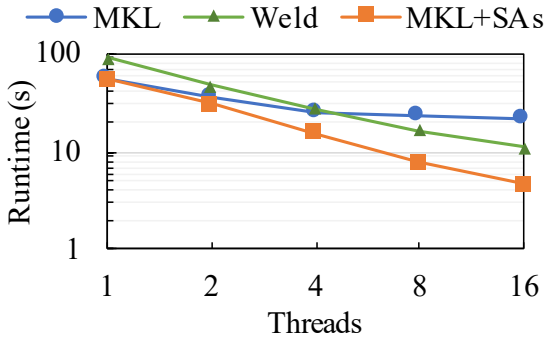


Figure 1.3: Performance of the Black Scholes benchmark [22] on 1–16 threads with MKL, Weld, and MKL with SAs (using our system Mozart).

1.1.3 Raw Filtering: Optimizing Data Loading for Unstructured Data

The third contribution of this dissertation looks at a more specific form of composition that appears in query processing engines. Systems with query engines such as Spark and others [38, 109, 222] decompose functionality into individual operators and pass data from one operator to the next. This can lead to significant performance inefficiencies. We looked at a specific form of inefficiency as a result of this pattern: parsing serialized data (e.g., at rest on a disk or read from a network) that will eventually be discarded by upstream *filters*. Data parsing is especially expensive for unstructured or semi-structured data formats such as JSON or CSV, where traditional parsers execute a read-compute-jump loop *per byte of data*. In fact, for queries over these formats, over 99% of the execution time can go into just parsing records into in-memory objects.

The interface to most data parsing libraries takes as input a raw byte string, and outputs some structured object representing the parsed record (e.g., a C-style struct with each field at a fixed offset). This is wasteful if a significant fraction of records will be discarded by an upstream query operator, or if every field in the record is not accessed by the query.

We introduce a new interface and technique called *raw filtering* that allows an upstream query filter to opportunistically discard data (with a false positive rate) *before parsing it*. By extracting the properties of each filtering function into the parsing interface interface, this system can reject raw bytestreams before parsing them, resulting in significant performance improvements especially for queries over semi-structured data formats. Raw filtering decomposes query filters into simple operators called *raw filters*, which have the property of

only producing false positives relative to the filter from which they are derived. As a simple example, a query filter that looks for JSON records with `name="Athena"` can use a raw filter to search for the byte string `"Athe"`: this can be done efficiently using modern CPU SIMD instructions. We can combine many such raw filters to filter out records that are guaranteed to fail the upstream query filter: any false positives will be handled by the upstream filter.

Our system implementing raw filtering, Sparser, composes these operators using an optimizer that balances the false positive rate with the runtime of the full data parser and the execution time of the filters themselves to achieve the minimum expected execution time. We show that this system can improve *end-to-end* query execution times on Spark workloads over JSON data by over $4\times$, and can even accelerate some queries over optimized binary formats such as Avro [14] or Parquet [160].

1.1.4 Discussion

Each interface leverages algebraic properties of existing functions or software components to enable well-studied optimizations underneath existing library APIs. For example, Weld provides well-known compiler and database optimizations by lifting the structure of each function into a parallel IR, and then applying optimizations over the IR directly. Split annotations similarly use a *type system* to obtain algebraic properties about how functions interact: in particular, each annotation defines how data can be split and how split data can be passed between other annotated functions. This interface allows a small runtime to enable *vectorization* [26], another well-known optimization from the database literature. Finally, raw filtering uses small composable functions—the raw filters—and uses properties of the queries and the filters together to construct a cascade to discard unneeded data. In each case, the interface communicates the properties required to achieve a performance improvement, and a system backing it implements the runtime for realizing those improvements.

We have implemented each system as an open source project to demonstrate each of their benefits. Our implementations can accelerate existing data science workloads by up to two orders of magnitude and come close to hand-optimized performance. We show that, unlike with existing interfaces for composing software, users can take full advantage of modern parallel hardware with little to no effort.

From a longer-term perspective, we also discuss techniques for extending the work presented in this thesis for emerging hardware accelerators (e.g., vector processors), new kinds of libraries (e.g., high-level Python libraries that target GPUs), and new methodologies for programming (e.g., building fast kernels that can be easily composed into efficient applications). We believe that these new techniques will continue to impact the design of new software stacks as modern hardware becomes increasingly parallel and continues to bare the limitations of interfaces that were conceived decades ago.

1.2 Outline of Dissertation

The rest of this dissertation is organized as follows. Chapter 2 discusses Weld, our new intermediate representation and runtime that enables cross-function optimizations on top of existing library APIs. Chapter 3 then discusses split annotations, which adopts the optimizations from Weld to legacy APIs, enabling data movement optimizations without library code changes. Chapter 4 discusses raw filtering, a new interface for optimizing filters in query processing workloads by considering their composition with data parsers. Finally, Chapter 5 discusses some lessons learned, proposes potential future work, and concludes.

Chapter 2

Weld: A Common Runtime for Data Analytics

2.1 Introduction

This chapter introduces Weld [155], a novel interface and runtime that can optimize *across* data-intensive libraries and functions while preserving their user-facing APIs. As discussed in Chapter 1, the traditional interface for composing libraries has been function calls that exchange data via pointers to in-memory values. Unfortunately, the gap between memory bandwidth and processing speeds has grown over time [99], so that, on modern hardware, many applications spend most of their time on *data movement* between functions. Weld addresses this by using its interface to expose the data-parallel structure of each library function (e.g., a map operation or an aggregation), and using algebraic rewrite rules over this structure to perform data movement optimizations on top of existing APIs.

Weld targets data analytics applications that utilize libraries and functions from multiple domains, such as Pandas [157], NumPy [146], and TensorFlow [2]. This spans ad hoc applications used for cursory data visualization to applications for drug discovery [71]. Even though the functions from these libraries are usually individually well-optimized (e.g., NumPy and Pandas are largely implemented in C or Cython), we find that cross-function data movement and lack of end-to-end optimizations still lead to significant slowdowns.

Weld consists of three main components (Figure 2.1). First, Weld asks libraries to

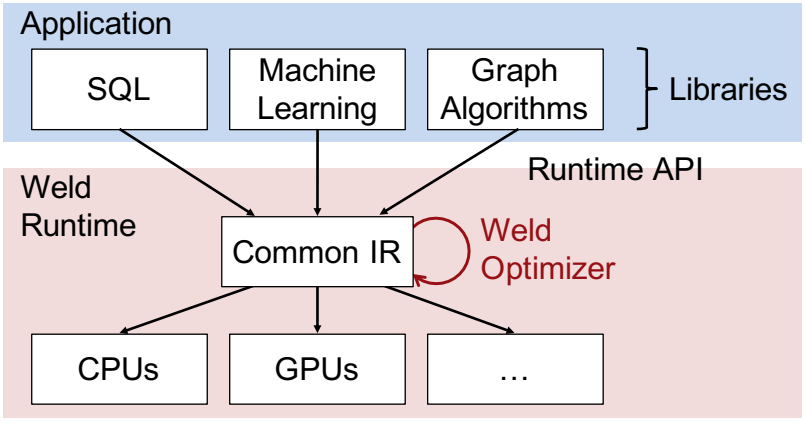


Figure 2.1: Weld captures diverse data-parallel workloads using a common intermediate representation (IR) to emit efficient code for the end-to-end application on diverse hardware.

represent their computations using a functional *intermediate representation (IR)*. This IR captures the data-parallel structure of each function, and closed transformations over the IR enable data movement optimizations such as loop fusion or loop tiling [121]. The computations encapsulated within the parallel structure (e.g., the function applied to each element in a map operation) can itself be represented within the IR, but Weld also allows this to be a black-box (e.g., by calling external C functions, such as kernel functions that may already be available in a data analytics library). Libraries then submit their computations to Weld through a lazily-evaluated *runtime API* that can collect IR code from multiple functions before executing it. Finally, Weld's *optimizer* combines these IR fragments into efficient machine code for diverse parallel hardware. Weld's approach, which combines a unified IR with lazy evaluation, enables complex optimizations *across* independently written libraries for the first time.

Weld's first component is its IR. We sought to design an IR that is both highly general (to capture a wide range of data analytics computations) and amenable to complex optimizations (e.g., loop fusion, loop tiling, vectorization, and execution on diverse hardware). To this end, Weld uses a small functional IR based on two concepts: nested parallel loops and an abstraction called *builders* for composing results in parallel. Builders are a hardware-independent abstraction that specify *what* result to compute (e.g., a sum or a list) without giving a low-level implementation (e.g., atomic instructions), allowing for different implementations

on different hardware. We show that Weld's IR is general enough to capture relational, graph and machine learning workloads, and to generate code with state-of-the-art performance for these workloads.

Weld's second component is a runtime API that uses lazy evaluation to capture work across function call and library boundaries. Unlike interfaces such as OpenCL and CUDA [47, 198], which execute work eagerly, Weld registers the work from multiple functions (even in different languages) and optimizes across them only when the program forces an evaluation (e.g., before writing to disk). The API supports accessing data in the application's memory without changing its format, allowing Weld to work against the native in-memory formats of common libraries such as Spark SQL and NumPy.

Weld's third component is an *automatic optimizer* that converts the fragments of IR code submitted to the runtime into an optimized end-to-end program. Although the problem of optimizing applications in Weld is similar to database and compiler optimization at a high level, we address two challenges specific to the common runtime setting:

1. Unlike many human-authored SQL queries or programs, the Weld IR code generated by calling multiple libraries is often *highly redundant*, because imperative libraries like NumPy and Pandas materialize results after most operations. It is thus crucial to eliminate redundancies that arise from library composition, e.g., intermediate results that can be pipelined.

2. Weld needs to optimize ad-hoc analytics workloads with *no pre-computed data statistics* (e.g., interactive data analysis in Python), without adding significant runtime overhead. This is in contrast to traditional database systems, which often have catalogs with statistics to guide optimizer decisions.

To address these problems, we designed a fast adaptive optimizer for Weld based on (*1*) a rich set of pipelining, common subexpression elimination and loop fusion [105] rules to eliminate redundancy and (*2*) adaptive optimizations for predicating branches [104] and choosing efficient data structures based on data sampled at runtime. We show that our optimization, compilation and sampling steps take sub-second time on realistic workloads and produce high performance code. The optimizer treats each optimization as an algebraic rewrite of the IR.

We show that Weld can unlock order-of-magnitude speedups in data analytics applications, even when they use well-optimized libraries. We implemented Weld with APIs in C, Java and Python, and Rust, a full backend for multicore x86 CPUs, and a partial backend for GPUs and the SX-Aurora vector processor [13]. We then integrated Weld into four popular libraries: Spark SQL, NumPy, Pandas, and TensorFlow. In total, on a single thread Weld can offer speedups of 3–29× in applications that use multiple libraries, and 2.5–6.5× even in applications that use multiple functions from the same library, by minimizing data movement and generating efficient machine code. Moreover, because Weld's IR is data-parallel, it can also parallelize the computations of single-threaded libraries such as NumPy and Pandas, yielding speedups of up to 180× when allowed to use more cores than the original computation with no additional programmer effort. Weld's compiler is also competitive with code generators for narrower domains, such as HyPer [141] for SQL and XLA [220] for linear algebra. We also found that porting each library to integrate with Weld only required a few days of effort and could be done incrementally, with noticeable benefits even when just a few common operators were ported. On average, our integration required around 1000 lines of code up front, and fewer than 20 lines of code per operator thereafter.

We also study the impact of each of our optimizations (e.g., pipelining, vectorization and adaptive predication) on our suite of workloads to identify which are critical to support in a common runtime. We find that the data movement optimizations are, in most cases, the most impactful ones when optimizing cross-library workloads. Other optimizations have moderate to high impact on specific classes of workloads.

Finally, we evaluate Weld on two hardware backends other than the CPU: a prototype GPU backend, and a backend developed by NEC for the SX-Aurora vector processor [13]. We show that Weld can automatically generate GPU code that comes within 1.4× a hand-optimized CUDA kernel, and comes within 2× on a subset of the TPC-H benchmark compared to an optimized GPU database. We also show that the SX-Aurora backend can provide over 3× speedups compared to the same workload using vectorized code on a CPU, with no code modification.

Overall, Weld highlights the idea of using algebraic properties in the *interface for composing software libraries* as a key enabler of high performance and high developer productivity at the same time. Today, most systems either feature a domain-specific, high-level

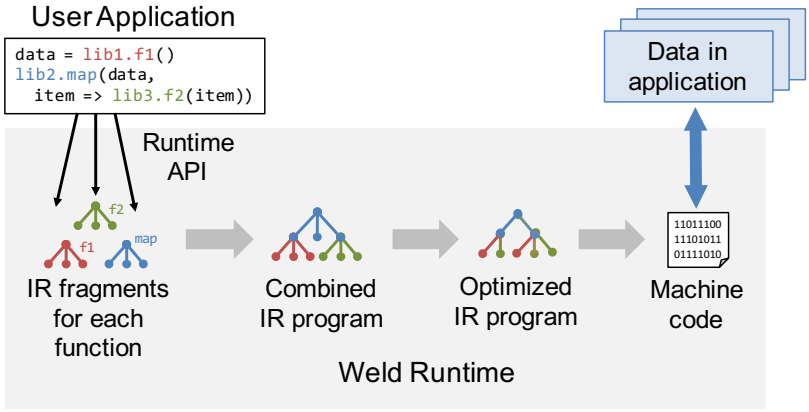


Figure 2.2: Overview of Weld. Weld collects fragments of IR code for each Weld-enabled library call and combines them into a single IR program. It then compiles this program to optimized code that runs on the application's in-memory data.

interface (such as SQL) that enables rich optimization but requires all of the computation to be understood by a single engine, or low-level interfaces (such as function calls) that can be invoked from anywhere but do not offer end-to-end optimization. Weld's IR aims to give library developers enough freedom to implement their own computations, while still supporting the most performance-critical optimizations, such as pipelining. This lets users have the benefits of a wide library ecosystem and database-like end-to-end optimization at the same time. Weld is open source at https://www.github.com/weld-project/weld.

2.2 System Overview

Figure 2.2 shows an overview of Weld. As described earlier, Weld has three components: a data-parallel IR that libraries use to express computations, a lazy runtime API for submitting IR fragments to Weld's runtime, and an optimizer that transforms these fragments into a compiled end-to-end application.

Weld aims to accelerate in-memory applications that compose data-parallel computations on a single machine, taking advantage of multicore and SIMD processing while minimizing memory movement. There are many possible designs for such a system, ranging from one with built-in primitives for every domain (e.g., machine learning, graph algorithms or relational algebra) to low-level languages such as OpenCL that give library developers full

control over how to implement their computations. With Weld, we designed a minimal IR and runtime that enables the most impactful cross-library optimizations while being expressive enough to allow libraries to represent their algorithms. Specifically, Weld was designed to focus primarily on *data movement* optimizations for *data-parallel* operators from domains such as relational and linear algebra. These operators consume the bulk of time in many applications by causing memory traffic, and benefit from co-optimization. Domain-specific optimizations such as database index maintenance, reordering linear algebra expressions or reordering SQL joins still need to be implemented within each library outside of Weld, but many systems already perform these. For example, a relational library could first optimize a logical plan by ordering joins, pushing down predicates, etc. and then use Weld for physical plan optimization. For already-optimized code such as BLAS or for domain-specific tasks such as using an index to load on-disk data, Weld supports calling existing C functions.

We expect libraries to integrate Weld in two ways. First, many libraries, such as Pandas and NumPy, already implement performance-sensitive functions in low-level languages such as C. Developers can port individual functions in these libraries to use Weld's functional IR instead, thus automatically benefiting from cross-function optimizations. These libraries often already have compact in-memory data representations (e.g., NumPy arrays [146]), so Weld can work directly against their in-memory data at no extra cost. Second, some libraries, such as Spark SQL and TensorFlow, already perform code generation beneath a lazily evaluated API. For these libraries, Weld offers both the ability to interface efficiently with other libraries and a systematic way to JIT code. For example, much of the complexity in code generators for databases comes from operator fusion logic that requires transforming a tree of operators into a single loop [5, 141]. With Weld, each operator can emit a separate loop over its inputs, and our optimizer will automatically fuse them.

2.2.1 A Motivating Example

We illustrate the benefit of Weld in a data science workflow adapted from a tutorial for Pandas and NumPy [158], shown in Listing 2.1. Pandas and NumPy are two popular Python data science libraries: Pandas provides a "DataFrame" API for manipulating data in a tabular

Listing 2.1 Example that uses Pandas and NumPy to compute a per-city "crime index."

```
def crime_index(data):
  # Get all city information with total population greater than 500,000
  df = data[data["TotalPopulation"] > 500000]
  stats = df["TotalPopulation", "AdultPopulation", "NumRobberies"].values
  vals = np.array([1.0, 2.0, -2000.0]
  predictions = np.exp(np.dot(stats, vals)) / 100000.0)
  predictions = predictions / predictions.sum()
  df = data_big_cities[["State short"]]
  df["Crime index"] = predictions
  df = df.groupby("State short").sum()
  return df["Crime index"]
```

format, while NumPy provides fast linear algebra operators over vectors and matrices. Both Pandas and NumPy offer optimized operators, such as data filtering and vector addition, written in C or Cython. However, workloads that combine these operators still experience substantial overhead from materializing intermediate results.

Our workload consists of filtering large cities out of a population-by-cities dataset, evaluating a linear model using features in the DataFrame to compute a crime index, and then aggregating these crime indices into a total crime index. It combines relational operators from Pandas with vector arithmetic operators from NumPy. Figure 2.3 shows its performance on a 6 GB dataset. Porting each operator to run on Weld yields a 3× speedup (shown in the No Fusion bar) due to Weld's more efficient, vectorizing code generator. Enabling Weld's loop fusion optimization then leads to a further 2.8× speedup *within* each library, and an additional 3.5× speedup when enabled *across* libraries. This gives Weld a total 29× speedup on a single thread, largely due to this data movement optimization (Weld 1T bar). Finally, Pandas and NumPy are single-threaded, but Weld can automatically parallelize its generated code without any change to the user application. Enabling multithreading gives a further 6.3× speedup on 12 cores, at which point Weld saturates the machine's memory bandwidth, for a total of 187× speedup versus single-core NumPy and Pandas (with each library backed by C-based operators).

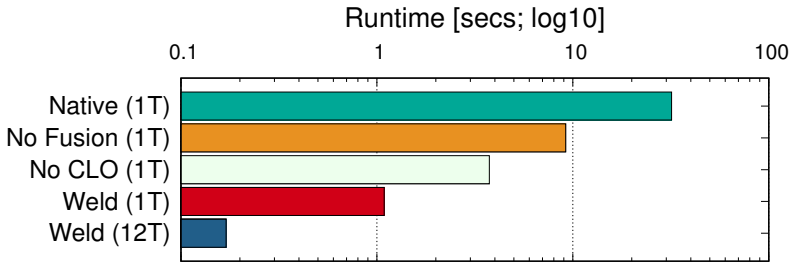


Figure 2.3: Performance of a data science workflow (log scale) in unmodified Pandas and NumPy (where only individual operators are written in C), Weld without loop fusion, Weld without cross-library optimization (CLO), Weld with all optimizations enabled, and Weld with 12 threads.

2.3 Weld's Intermediate Representation

Libraries communicate the computations they perform to Weld using a data-parallel intermediate representation (IR). This component of the Weld interface determines both the workloads that Weld supports and the optimizations that the optimizer can apply. To support a wide range of data-intensive workloads, we designed Weld's IR to meet three goals:

1. **Generality:** we wanted an IR that could express diverse data analytics tasks (e.g., relational and linear algebra), as well as *composition* of these tasks into larger programs (e.g., calling one library inside another).

2. **Ability to support data movement optimizations:** we wanted an IR that could support data movement optimizations such as loop fusion and loop tiling.

3. **Parallelism:** we wanted the IR to be explicitly parallel so that Weld can automatically generate code for modern parallel hardware, e.g., multicores and GPUs.

To meet these goals, we designed a small IR inspired by monad comprehensions [75], similar to functional languages but operating at a lower level that makes it easier to express the fusion rules that are critical for data movement optimization.

Builder Types	
`vecbuilder[T]`	Builds `vec[T]` by appending merged values of type `T`.
`merger[T,op]`	Builds `T` by merging values with associative operation `op`.
`dictmerger[K,V,op]`	Builds `dict[K,V]` by merging `{K,V}` with associative operation.
`vecmerger[T,op]`	Builds `vec[T]` by merging `{index,T}` pairs into specific cells in the vector using an associative operation.
`groupbuilder[K,V]`	Builds `dict[K,vec[V]]` by merging `{K,V}` and grouping by key.

Table 2.1: Builder types in Weld.

2.3.1 Data Model

Weld's basic data types are scalars (e.g., `int` and `float`), variable-length vectors (denoted `vec[T]` for a type `T`), fixed-length arrays (denoted `simd[T;LENGTH]` for their application in enabling SIMD-vectorization), structures (denoted `{T1,T2,...}`), and dictionaries (`dict[K,V]`). All data types are nestable. We chose these types because they appear commonly in data-intensive applications and in low-level data processing code (e.g., dictionaries are useful to implement database joins).

2.3.2 Expressing Computations

Weld's IR is a functional, expression-oriented language. It contains basic operators for arithmetic, assigning names to values, sequential `while` loops, sorting, and collection lookups. It also supports calling external functions in C. The Weld documentation online [218] describes the full set of operators supported by Weld.

Weld's IR has two main *parallel* constructs: a parallel `for` loop and a hardware-independent abstraction for constructing results in parallel called a *builder*. Parallel loops can be nested arbitrarily, which allows complex function composition. Each loop can merge (i.e., write) values into multiple builders; for example, a single loop can merge values into one builder to produce a sum and another to produce a list.

Weld includes multiple types of builders, shown in Table 2.1. For example, a `vecbuilder[T]` takes values of type `T` and builds a vector of merged values. A `merger[func,U]` takes an

associative function `(T, U)=> U` and an identity value of type `U` and combines values into a single result of type `U`. Builders can be used to implement higher level functional primitives, e.g., a `vecbuilder` can be used to implement a map operation while a `merger` can be used to compute reductions.

The IR supports interaction with builder types with three operations. `merge(b, v)` adds a new value `v` into the builder `b` and returns a new builder to represent the result.[1] Merges into builders are associative, enabling them to run in parallel. `result(builder)` destroys the builder and returns its final result: no further operations are allowed on it after this. Finally, Weld's parallel `for` loop is also an operator that consumes and returns builders. `for(vectors , builders, func)` applies a function of type `(builders, index, elem)=> builders` to each element of one or more vectors in parallel, then returns the updated builders. Each call to `func` receives the index of the corresponding element and a struct with the values from each vector. The loop can also optionally take a start index, end index and stride for each input vector to express more complex strided access patterns over multidimensional arrays (e.g., matrices). Weld's `for` loops can be nested arbitrarily, enabling Weld to express irregular parallelism (e.g., graph algorithms) where different iterations of the inner loop do different amounts of work.

Finally, Weld places two restrictions on the use of builders for efficiency. First, each builder must be consumed (passed to an operator) exactly once per control path to prevent having multiple values derive from the same builder, which would require copying its state. Formally, a builder is a linear type [216]. Second, functions passed to `for` must return builders derived from their arguments. These restrictions allow Weld's compiler backends to safely implement builders using mutable state.

2.3.3 UDFs and Macros

Weld supports calling user-defined C functions by name as part of its IR. Users pass values from the Weld program to the C function and specify the Weld return type of the function. Each Weld type has a standard C-compatible data layout (§2.4) with which the UDF can interface. UDFs can be used to call compute-optimized kernels such as Level-3 BLAS

[1] In practice, some mutable state will be updated with the merged value, but Weld's functional IR treats all values as immutable, so we represent the result in the IR as a new builder object.

Listing 2.2 Examples of using builders.

```
// Merge two values into a builder
b1 := vecbuilder[int];
b2 := merge(b1, 5);
b3 := merge(b2, 6);
result(b3)   // returns [5, 6]

// Use a for loop to merge multiple values
b1 := vecbuilder[int];
b2 := for([1,2,3], b1, (b,i,x) => merge(b, x+1));
result(b2)   // returns [2, 3, 4]

// Loop over two vectors and merge results only
// on some iterations.
v0 := [1, 2, 3];
v1 := [4, 5, 6];
result(
  for({v0, v1},
     vecbuilder[int],
     (b,i,x) => if(x.0 > 1) merge(b, x.0+x.1) else b
)) // returns [7, 9]
```

routines. Although not enforced in our implementation, Weld UDFs are expected to be side-effect-free and to not hold locks.

To aid developers, Weld also contains macros that implement common functional operators such as `map`, `filter` and `reduce` using builders, so that library developers familiar with functional APIs can concisely express their computations. These operators all map into loops and builders. For example, the second snippet in Listing 2.2 effectively implements a `map` over a vector.

Macros and UDFs in Weld can also be used together to implement common functionality required across libraries. For example, Weld does not have a native string type, but can represent byte buffers using vectors of `uint8`. We provide a small string "library" by using macros wrapped around UDFs to call an existing UTF-8 C library to process these vectors.

Listing 2.3 The first program computes a `map` and `reduce` over the same input vector. The second uses the builder abstraction to fuse these into a single pass over the data.

```
// Program 1, with map and reduce
data := [1,2,3];
r1 := map(data, x => x+1);
r2 := reduce(data, 0, (x, y) => x+y)

// Program 2, fused using builders
data := [1,2,3];
result(
  for(data, { vecbuilder[int], merger[+,0] },
     (bs, i, x) => { merge(bs.0, x+1), merge(bs.1, x) }
)) // returns {[2,3,4], 6}
```

2.3.4 Why Loops and Builders?

Given the broad use of functional APIs such as MapReduce and Spark, a strawman design for an IR might have used these functional operators as the core constructs rather than loops and builders. Unfortunately, this design prevents expressing many common optimizations in the IR. For example, consider the first program in Listing 2.3, which runs two operations on the same input vector. Even though the `map` and `reduce` operations can be computed in a shared pass over the data, no operator akin to `mapAndReduce` exists that computes both values in one pass. Richer optimizations such as loop tiling are even harder to express in a high-level functional IR. By exposing a loop construct that can update multiple builders, patterns like this can easily be fused into ones such as the second program in Listing 2.3.

2.3.5 Generality and Limitations of the IR

Weld's parallel loop and builders can express all of the functional operators in systems such as MapReduce, LINQ and Spark, as well as relational algebra, linear algebra and other data-parallel operations. Given the wide range of algorithms implemented over these APIs [222, 225], we believe Weld can benefit many important workloads. Our evaluation shows workloads from each of these domains. §2.10 discusses some limitations of the IR and of our implementation of Weld.

2.4 Runtime API

The second component of Weld is its runtime API. Unlike interfaces like OpenCL and CUDA, Weld uses a *lazy* API to construct a computation graph. Library functions use the API to compose fragments of IR code (perhaps across libraries) and provide access to data in the application. The API uses the IR fragments and data to build a DAG of computations. When libraries evaluate a computation, Weld fuses the graph into a single IR program, optimizes it, and executes it. We show examples of the API in Python here, though Weld also API bindings for Java, Scala, C, and Rust.

Consider the program in Listing 2.4 as a motivating example, which uses the `itertools` library's `map` function to iterate over a set of vectors and apply `numpy.dot` to each one:

Listing 2.4 A Python program that combines library calls.

```
scores = itertools.map(vecs, lambda v: numpy.dot(v, x))
print(scores)
```

A standard call to `itertools.map` would treat `numpy.dot` as a black box and call it on each row. If both libraries use Weld, however, the result `scores` is instead an object encapsulating a Weld program capturing *both* functions. Weld evaluates this object only when the user calls `print`. Before evaluation, Weld will optimize the IR code for the entire workflow, enabling optimizations that would not make sense in either function on its own. For example, Weld can tile the loop to reuse blocks of the `x` vector across multiple rows of `v` for cache efficiency.

2.4.1 API Overview

Developers integrate Weld into their libraries using an interface called `WeldObject`, which represents either a lazily evaluated sub-computation or an external value in the application. A `WeldObject` may depend on other `WeldObjects` (possibly from other libraries), forming a DAG for the whole program where leaves of the DAG represent external data. Table 2.2 summarizes Weld's API.

Developers create `WeldObjects` using the `NewWeldObject` call. This call has two variants:

API	Summary
`NewWeldObject(value, type, encoder)`	Creates a new `WeldObject` that wraps an in-memory `value` with a given Weld `type` and `encoder`. The encoder implements marshaling (§2.4.2).
`NewWeldObject(deps, expr, encoder)`	Creates a new `WeldObject` with other lazy computations as dependencies. `expr` is a Weld IR fragment that uses the dependencies.
`Evaluate(obj, ctx: WeldContext)`	Evaluate a `WeldObject`. The `ctx` argument is used for memory management (§2.4.3).
`NewWeldContext(ctx)`	Creates a new `WeldContext`, which represents a buffer pool holding allocations made by Weld.
`FreeWeldContext(ctx)`	Frees a `WeldContext`, which frees all memory allocated by a Weld call.

Table 2.2: Weld's runtime API.

one to encapsulate external data dependencies in the application and one to encapsulate sub-computations and dependencies *among* `WeldObjects`. To encapsulate an external data dependency, developers pass as arguments a pointer to the data dependency, the Weld type of the dependency (e.g., `vec[int]`), and an *encoder* for marshaling between native library formats and Weld-compatible objects. §2.4.2 discusses encoders in detail.

To encapsulate sub-computations and dependencies with other `WeldObjects`, developers pass a list of dependent `WeldObjects` (`deps`), a Weld IR expression representing the computation, and an encoder for the expected type of the `WeldObject`. If the library evaluates the `WeldObject`, this encoder is used to marshal data returned by Weld to a native-library format. The provided IR expression must depend only on the dependencies declared in `deps`. The example below shows a function for computing the square of a number using Weld's API.

```
def square(arg):
  # Programatically construct an IR expression.
  expr = weld.Multiply(arg, arg)
  return NewWeldObject([arg], expr)
```

The `Evaluate` API call evaluates a `WeldObject` instance and returns a result. Libraries can choose when to evaluate an object in several ways. In our integrations with Python libraries, we used methods that save or print the object (e.g., the `__str__` method to convert it

to a string) as evaluation points to introduce lazy evaluation behind the library's existing API. Systems like Spark and TensorFlow already have lazy APIs with well-defined evaluation points. `Evaluate` returns a special handle called `WeldResult` that can be checked for failure or queried for a pointer to the returned data. `Evaluate` also takes an additional `WeldContext` argument, which is used to manage memory allocated by Weld, as described in §2.4.3. Developers can allocate a new context for each call to evaluate or reuse contexts in cases where state (e.g., builders that have not been finalized) is to be reused across calls to Weld.

The `Evaluate` call also allows evaluating multiple Weld objects in one computation, which is useful in applications that produce multiple results. Our optimizer automatically identifies shared sub-computations across these results. While conceptually simple, this design is different from lazy evaluation APIs such as Spark [224], which only allow submitting one computation at a time.

2.4.2 Marshalling Data

Weld specifies a standard binary format for its basic data types that allows it to operate over existing in-memory data. Specifically, scalar types (`int`, `float`, etc.) and structs follow C packed structure layout, and vectors (`vec[T]`) are represented as an `{int64,T*}` structure, with a length and data pointer field. Dictionaries are represented as single `int64` pointers and have an external API for iteration and external lookup: Weld also has an operator to convert dictionaries to vectors of key/value pairs for greater flexibility. Since builders are abstract types whose internal implementation is hidden and may change dynamically, they are always passed out of Weld as opaque `int64` handles. Users should only pass builders out of Weld to pass them into another program: data inside a builder should be accessed only after finalizing the builder with `result`.

Library developers provide *encoders* to map types in their native programming languages (e.g., Python) to Weld-usable data and vice-versa. The encoder interface is a single function `encode`. When used with `NewWeldObject` to declare a data dependency, this function maps an object in the library's native language and returns data understood by Weld's runtime. For example, an encoder for NumPy arrays would take in a NumPy object and extract a pointer to its internal data array, which already happens to be a packed array of primitive

types in NumPy [146]. When used with `NewWeldObject` to declare a sub-computation, the function takes a pointer to data in Weld's format and returns an object in a library format.

2.4.3 Memory Management

Values in Weld are considered either *owned by library* or *owned by Weld*. Input data is always owned by libraries. These data are neither mutated nor freed by Weld. Values that Weld allocates during execution are owned by Weld, as are results of `Evaluate` (i.e., values encapsulated by `WeldResults`). Decoders can either point to these values directly when wrapping them into library-specific objects (as long as the library does not free the corresponding `WeldResult`), or they can copy out data and free the Weld-owned memory.

The `WeldContext` argument of `Evaluate` is the main interface used for Weld-owned memory management, and acts as a stateful buffer pool for memory allocated by Weld. The context represents at memory allocated by Weld during execution that *could be* externally accessible. Internally, Weld's runtime can choose how to free memory guaranteed to be inaccessible (reference counting, garbage collection, etc.). Freeing a `WeldContext` frees all of its managed memory, including memory returned by a call to `Evaluate`, so developers should copy out values they need before doing this.

The `WeldContext` also allows developers to share state between individual Weld programs or repeatedly call a Weld program to mutate builders. For example, in our integrations with database-like libraries, we use Weld by repeatedly calling `Evaluate` on batches of data. Each call merges values into a builder, and then returns the result of the merge (which is itself a builder). The builder is an integer representing a pointer pointing to memory allocated by Weld and managed by the `WeldContext` (§2.4.2). After consuming all batches, we use another Weld program *with the same context* to finalize the builder with `result` and retrieve the final result. Without the context, it would not be possible to pass the pointer to the builder state out of Weld *and* guarantee the liveness of the builder state.

Finally, the `WeldContext` can be used to set resource limits on Weld. For example, a developer can set the context to allocate at most 1GB of memory; if Weld tries to allocate more, it will fail with an error code. This is useful when integrated Weld into a larger system with its own memory management, e.g., Spark [224].

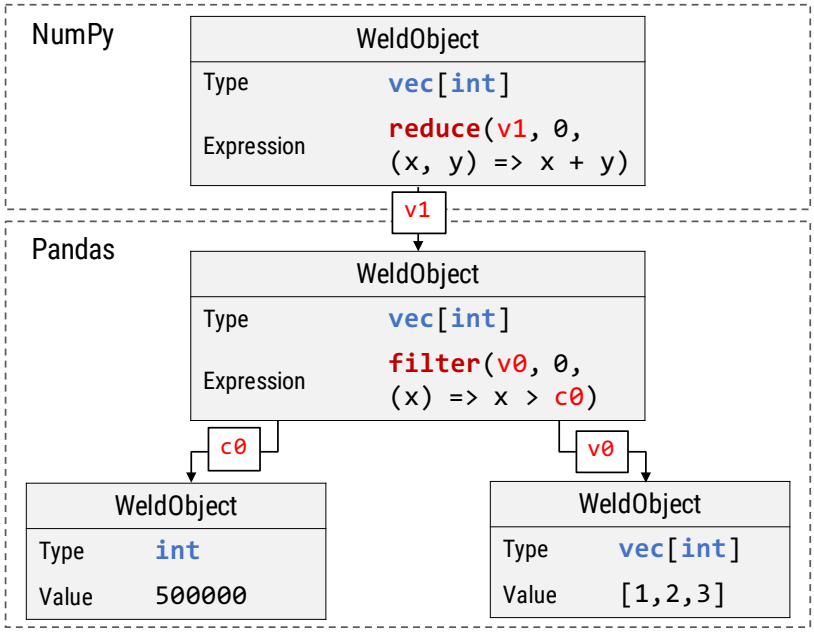


Figure 2.4: The example from Listing 2.6 as a computation graph.

2.4.4 User Defined Functions (UDFs)

Libraries such as Spark and `itertools` can take functions as arguments. To support these with Weld, a `WeldObject`'s IR expression can represent a function, with dependencies pointing to variables in its closure. In order to facilitate passing functions into Weld, we implemented a UDF translator for Python that walks Python abstract syntax trees (ASTs) to convert them to Weld IR, based on techniques in existing systems [45, 138, 183]. Listing 2.5 shows an example; the `@weld` annotation provides a type signature for the function in Weld.

Listing 2.5 Python UDF with a Weld type annotation.

```
# Produces (x: int) => x + 1 in Weld IR
@weld("(int) => int")
def increment(x): return x + 1
```

2.4.5 Example: Combining NumPy and Pandas

Weld's API enables optimizations even across independent libraries. We illustrate this with the function in Listing 2.6, which uses the Python Pandas library and NumPy to compute the total population of all cities with over 500,000 residents. NumPy represents data in C arrays wrapped in Python objects. Pandas uses data frames [157], which are tables with named columns that are also encoded as NumPy arrays.

Listing 2.6 A sample Python program using Pandas and NumPy.

```
# Original Python Function
def large_cities_population(data):
  filtered = data[data["population"] > 500000]
  sum = numpy.sum(filtered)
  print sum
```

In the native Pandas and NumPy libraries, this code causes two data scans: one to filter out values greater than 500,000 and one to sum the values. Using Weld, these scans can be fused into a single loop and the sum can be computed "on the fly." The loop also benefits from optimizations such as vectorization.

To enable using Weld for this program, we must extend the `DataFrame` object in Pandas to return lazily evaluated `WeldObjects`. We must also provide a IR fragment for the `>` operator on `DataFrame` columns and the `numpy.sum` function.

Listing 2.7 shows the Weld expressions for implementations of each of these functions. Each implementation takes either a data dependency (e.g., a NumPy array) or `WeldObject` as input, and incrementally builds a Weld computation graph by returning another composable `WeldObject` instance.

After porting these operators to Weld, the libraries can use the API to lazily compose a computation graph for the full workload without any modifications to the user's code. Listing 2.8 shows the final fused Weld expression of the variable `sum` (now a `WeldObject`) before it is printed. Calling `print` on this instance invokes the `evaluate` method to compute a result. Figure 2.4 shows the computation graph for the workload.

After optimization, this function becomes a single parallel loop using builders, as shown in the second half of Listing 2.8. The optimizer can also apply optimizations such as

Listing 2.7 Pandas and NumPy functions using Weld.

```
# DataFrame > filter
# Input: Vector v0, constant c0
filter(v0, (x) => x > c0)

# numpy.sum
# Input: Vector v0
reduce(v0, 0, (x, y) => x + y)
```

Listing 2.8 The combined Weld program.

```
// Combined Weld program
reduce(filter(v0, (x) => x>500000), 0, (x,y) => x+y)

// Combined Weld program after converting macros to builders and fusing loops
    .
// Further optimization is possible (e.g,. SIMD), but omitted for clarity.
result(for(v0, merger[+,0], (b, i, x) => if (x > 500000) merge(b, x) else b))
```

predication and vectorization here but we omit these for clarity in this example.

2.5 Weld Automatic Optimizer

The third main component of Weld is its automatic optimizer. Optimizing Weld computations is conceptually similar to relational query optimization, but two aspects of Weld's "common runtime" setting pose unique challenges:

1. IR code passed to Weld is generated out of fragments from different library functions that have no knowledge of each other, even though they may depend on the same data or sub-computation. This contrasts with many queries to a DBMS, where the full SQL query is generated as one program or even written by one user. Weld thus needs to identify and eliminate many forms of *redundancy* that arise from imperative data analytics libraries. We addressed this challenge by performing a wide range of redundancy elimination transformations from the database and compiler literature (e.g., pipelining [72] and fusion transformations across multiple loops [105], which are similar

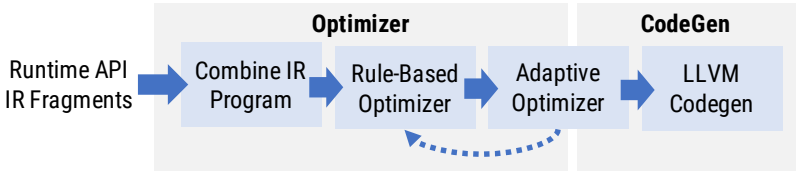


Figure 2.5: Architecture of the Weld optimizer. IR code submitted to Weld is combined, optimized via rules, and transformed to generate adaptive optimization code for decisions based on runtime statistics. The optimized IR code is compiled to assembly using LLVM.

to scan sharing [174] and some types of multi-query optimization [189]). Our loop-and-builder IR with loops with multiple inputs also help with this task.

2. In the most general setting for Weld (ad-hoc data analysis in a language such as Python), there are no pre-computed data statistics, so Weld needs to make data-dependent decisions adaptively at runtime. Moreover, each query might only run once, so any data sampling to compute statistics must be fast. Here, we focused on designing an optimizer for this setting to show that effective optimization is still possible. The optimizer can also use application-provided statistics instead of measuring them at runtime.

We designed an optimizer using a combination of transformation rules and adaptive decisions that combines ideas from both the database and compilers communities (Figure 2.5). The optimizer begins with rule-based optimizations from the database and compiler literature, then moves on to an adaptive optimization phase where key data-dependent decisions are considered (such as whether to use branching or predication based on a filter's selectivity [104]). For these decisions, our optimizer generates code to sample the relevant property of the data at runtime and switch between two possible execution plans. Both plans are represented in Weld IR and are passed through the rule-based optimizer again. Finally, given a Weld IR program that contains both the sampling code and the possible execution branches, we apply a code generator based on LLVM to JIT-compile the code. This code then runs on a multithreaded, memory-managed runtime. §2.5.1–2.6 describe the rule-based, adaptive, and code generation optimizations in turn.

Rule-based Optimization Passes	
Fusion	Pipelines loops to avoid materializing intermediate results when the output of one loop is used as the input of another. Also fuses multiple passes over the same vector.
Size-inference	Infer the size of output vectors based on the size of inputs.
Vectorization	Convert loops to use SIMD vectors if available on the CPU.
Common sub-expression elimination	Cache or eliminate repeated computations.
Algebraic	Simplify algebraic expressions and fold constants.
Inliner	Inline redundant expressions (e.g., creation of struct immediately followed by a struct field access).

Table 2.3: Some of the rule-based optimizations in Weld. The optimizer is extensible, and new rules can be added externally, similar to other rule-based systems [11].

2.5.1 Rule-Based Optimizations

The first step of the optimizer is to apply rule-based optimizations on the combined IR program. Like other database optimizers and compilers [115, 190], we group our rules into *phases* and run each phase to a fixpoint. Moreover, each rule produces a new abstract syntax tree (AST) in the Weld IR, making it easy to combine rules in an arbitrary order. After each pass, Weld also applies a number of *simplification transformations* to remove redundant computation, such as inlining variables that are only used in one expression, common subexpression elimination, constant propagation, and standard algebraic simplifications. These rule-based optimizations help eliminate redundancies caused by composing independently-written functions and libraries. We elaborate on key optimization passes. Table 2.3 lists our optimization passes in the order they run.

Fusion

The fusion pass performs the main data movement optimizations in Weld, merging data-parallel operations from different libraries and functions into a single parallel `for` loop and removing redundant iteration over the same data or materialization of results. "Fusion" is our general term for several transformations including pipelining `for` loops whose output is directly consumed by another loop (as in many database engines [72, 141]) and fusing independent loops that read the same input data (similar to some forms of multi-query optimization [189] and scan sharing [174] in the database literature, or loop fusion [105] in

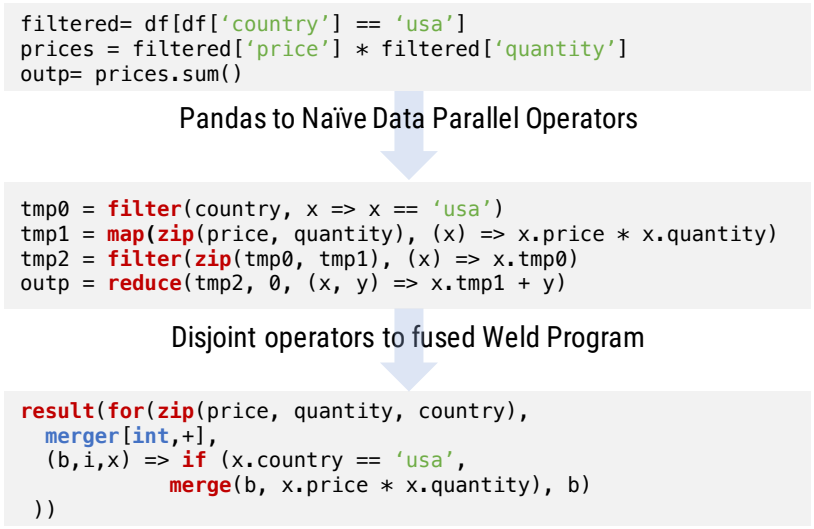


Figure 2.6: Optimizing a simple Pandas program that filters and aggregates data. While Pandas repeatedly loops over data, Weld integration enables pipelining.

the compilers literature). This pass is critical because independently called libraries will produce separate loops, even if these loops are running over the same data. For example, in Pandas and NumPy, every operator used to build an expression (e.g., `vec1+vec2+vec3`) produces a new array or data column for its intermediate result. Figure 2.6 shows an example of a simple Pandas program that benefits from fusion.

We perform two types of fusion: *pipelining* and *horizontal fusion* of loops over the same data. Listing 2.9 gives an example of both. In the pipelining transformation, Weld fuses the body of loops whose result is consumed in just one other loop. Rather than traversing (and materializing) vectors after each operation, pipelining enhances memory locality by loading data into the cache or registers and applying all operations at once. While DBMS engines usually pipeline operators in the Volcano query processing model, without Weld, independently written libraries have no common substrate to allow for such optimizations. Compared to code-generating databases like HyPer [141], which pipeline operators via a produce/consume API and a complex translation to imperative code to fuse individual operators, Weld's approach is to use a higher level IR that facilitates identification and fusion of pipelinable loops while simultaneously capturing parallelism. For example, in Listing 2.9, Weld identifies that `v1` is only used in one downstream expression, and fuses it into that loop. The optimized loop still executes in parallel and is represented in the same IR, allowing for

Listing 2.9 Pipelining and horizontal loop fusion optimizations expressed using Weld.

```
// -------- Pipelining Fusion --------

// Before pipelining.
v1 := result(for(
    v0, vecbuilder[int], (b,i,x) => merge(b, x+1)))
v2 := result(for(
    v1, vecbuilder[int], (b,i,x) => merge(b, x*5)))

// After pipelining.
v2 := result(for(
    v0, vecbuilder[int], (b,i,x) => merge(b, (x+1)*5)))

// -------- Horizontal Fusion --------

// Before horizontal fusion
v1 := result(for(
    v0, vecbuilder[int], (b,i,x) => merge(b,x+1)))
v2 := result(for(
    v0, merger[int,+], (b,i,x) => merge(b,x)))
{v1, v2}

// After horizontal fusion
tmp := for(v0, {vecbuilder[int], merger[int,+]},
    (bs,i,x) => {merge(bs.0, x+1), merge(bs.1, x)}
)
{result(tmp.0), result(tmp.1)}
```

further optimizations.

In horizontal fusion, loops over the same input data that produce different results are fused to loop over data just once, and return a tuple of results. Like pipelining, horizontal fusion enhances memory locality. The transform helps when repeatedly computing multiple results from the same input data: for example, the Pandas API creates a new vector from each operator on a DataFrame column (e.g., `col+1`), as shown in Figure 2.6.

Size Analysis Optimizations

Weld contains two optimizations based on knowledge of the size of a vector (`vec[T]`) at optimization time. *Loop unrolling* [122] allows replacing small loops with a sequence of statements to reduce control flow, while *preallocation* allows allocating memory for data structures in advance instead of growing them dynamically.

When Weld's optimizer is invoked, it knows the size of the input vectors to the computation. Moreover, libraries can mark the size of intermediate results in IR code with an annotation (e.g., if a library knows it works with 3-element vectors). The optimizer propagates this information to other intermediate results when possible (e.g., a loop that merges exactly one value per iteration into a `vecbuilder` will produce a result of the same size as the input).

Using this information, Weld first unrolls `for` loops with a simple body to run sequentially if they are sufficiently small. This can yield better pipelining within the CPU by eliminating control flow and branching logic, and also reduce overheads from Weld's multi-threaded runtime by running small loops sequentially.

Second, Weld attempts to preallocates memory for data structures that might otherwise be resized. For example, the `vecbuilder` supports merging a variable number of records, and is implemented using a dynamically-growing array by default. However, if the loop merging values into a `vecbuilder` adds in exactly one value per iteration (i.e., each control path in its body has just one `merge`), the optimizer can pre-allocate a vector of the same size as the input.

Unrolling and preallocation have a large impact in numerical code especially, such as NumPy code working with arrays of fixed dimensions. While these transformations also exist in traditional compilers, they are more powerful in Weld because the optimizer has access to size information that is only available at *runtime*.

Vectorization

The vectorization pass changes `for` loops to use SIMD instructions when possible to take advantage of modern CPU execution capabilities. Specifically, the optimizer change loops over elements of type `T` to loops over `simd[T]`, where `simd[T]` is a fixed-length SIMD

data type. This pass only vectorizes loops without branches; our adaptive predication optimization (§2.5.2) determines whether to vectorize branches because the benefit depends on selectivity.

Although LLVM also provides a vectorizer, we found that it was easier to get consistent results by vectorizing at the Weld IR level. This is because Weld's IR is purely functional, with no side-effects, and is thus simpler to analyze than lower-level IRs. These optimizations are harder to apply in a lower-level IR such as LLVM due to the need to analyze pointer aliasing (whether two addresses in mutable memory might point to the same data) [125]. Even in cases where the aliasing analysis is successful, applying these passes on Weld IR is beneficial because it reduces compile time—an important factor because Weld needs to JIT-compile code.

2.5.2 Adaptive Optimizations

Certain optimization decisions are data-dependent. In a DBMS, statistics catalogs and cost-based optimizers guide whether such optimizations should be applied to a plan, but the libraries Weld integrates with generally do not have these tools a priori. We therefore designed a set of *adaptive optimizations* that can occur at runtime based on observed properties of the data. In particular, we consider two adaptive optimizations that we found have the greatest impact: *adaptive predication* and *adaptive data structures*.

Adaptive Predication

Given a Weld program with a branch (i.e., an `if` statement), *predicating* the branch evaluates the condition, on-true expression, and on-false expression unconditionally, and then uses a hardware `select` instruction to choose between the true or false expression depending on whether the condition is `true` or `false`. The key advantage of predication is that it enables vectorization: modern CPUs contain vectorized `select` instructions, whereas loops with branches or other control flow are difficult to vectorize. The tradeoff is that both the true and false expression are always evaluated (whereas in a branched expression, only one of the two is evaluated). If the branch is highly predictable and computing the true and false expressions is expensive, predication may hurt performance. The choice of whether to

predicate thus depends on (*1*) the selectivity of the branch (a data-dependent factor) and (*2*) the branch target costs [104].

Adaptive predication proceeds as follows. Given a loop with a branch that can be vectorized, the optimizer replaces the loop with three loops. The first loop samples the input vector and evaluates the condition of the branch on each sample to estimate the branch selectivity. The second loop is a copy of the original branched loop with scalar instructions, and the third loop is a vectorized and predicated version of the original loop. We generate code that uses the measured selectivity to choose between these two loops using a cost model. The cost model determines the costs of the true and false expressions to determine whether, at the given selectivity, evaluating *both* expressions unconditionally with SIMD will provide a speedup.

We use a simplified version of existing database cost models [104, 126] to make this decision. These cost models take as input a conditional of the form `expr = if(cond, merge(b, body), b)` and the measured selectivity of `cond`. The memory accesses in branched expressions can be separated into two sets: the set of accesses in the *condition*, C, and accesses in the *body*, B. In branched code, we assume that memory accesses in the condition are *sequential*, and accesses in the body are *random* (i.e., they will not be prefetched). This is because accesses in the condition are performed on every loop iteration. We assume sequential accesses are prefetched by the CPU, and the access time is thus limited by the memory bandwidth. Columns accessed only in the body will not always be prefetched; therefore, the memory latency (computed as in [126]) dominates the access time. Therefore, we compute the cost for branched code as:

$$\sum_{e \in C} \frac{\texttt{sizeof}(e)}{T} + s \times \sum_{e \in B} L, \tag{2.1}$$

where T is the memory throughput in bytes per second, L the memory latency, and s the measured selectivity.

In predicated code, we treat accesses in both the condition and body are sequential, since both execute on every input row. For simplicity, we captured the expected speedup from vectorization in predicated code using a constant multiplier v to reflect the reduced number of instructions, but we validated that the results were similar to modeling instruction

costs and memory access time separately. This gives us the following expression for the predicated cost:

$$v \times \sum_{e \in C \cup B} \frac{\texttt{sizeof}(e)}{T} \tag{2.2}$$

The generated code evaluates both of these expressions based on the measured selectivity and chooses the plan with the lower estimated cost (i.e., lower expected runtime). We show in §2.9 that predication can either increase or decrease performance depending on the data, so choosing whether to apply it *adaptively* is important.

Adaptive Data Structures

Weld's builders are declarative data types that can choose an implementation without specifying one in the IR. An important data-dependent decision Weld makes for builders using dictionary-like data structures (e.g., hash tables for `groupBy` operations) is determining when to use thread-local dictionaries that are merged when `result` is called, and when to use a single global dictionary shared by all threads. The local strategy performs better at lower cardinalities because it avoids the synchronization overhead of the global dictionary, but uses more memory due to keys appearing in multiple dictionaries. The global strategy naturally performs better at large key cardinalities due to its more efficient use of memory.

Our builder implementations for dictionaries thus use an adaptive design that comes close to the best of both worlds: for each thread, we use a local dictionary until the dictionary reaches a certain size threshold, then switch to using a global dictionary for all new keys that are not already in the local one. Thus, hot keys can be accessed in the local dictionary without synchronization overhead, while keeping memory usage and the risk of page faults and TLB misses in check. This design adapts to both small and large cardinalities in the event that Weld does not have access to statistical information (e.g., cardinality estimates) at optimization time.

Listing 2.10 Simple example of the internal Weld SIR.

```
// Weld IR
|x: i32|
  let b1 = vecbuilder[i32];
  let b2 = merge(b1, x);
  result(b2)

// SIR
F0 -> vec[i32]: // an SIR function.
Params: // Parameters to the SIR function.
  x: i32
Locals: // Local variables defined in the function
  fn0_tmp: vecbuilder[i32]
  fn0_tmp__1: vecbuilder[i32]
  fn0_tmp__2: vec[i32]
B0:
  // Single static assignment: each variable is assigned to exactly once.
  fn0_tmp = new vecbuilder[i32]()
  fn1_tmp__1 = merge(fn0_tmp, x)
  fn0_tmp__2 = result(fn0_tmp__1)
  return fn0_tmp__2
```

2.6 Code Generation and Runtime

In this section, we sketch the main challenges in implementing a compiler and parallel backend for CPUs and GPGPUs.

2.6.1 Weld's Sequential IR

Internally, Weld's compiler uses a "two-level-IR" approach. The core Weld IR as described in this chapter is represented as a tree, and most optimizations described occur as pattern matching rules over this tree structure. The tree is akin to an operator tree in a database. After running these optimizations, the tree is transformed into a second "sequential" IR (SIR), which represents IRs in more traditional compilers such as LLVM. The IR is a sequence of functions, each with a sequence of statements. Each statement's right-hand-side assigns a value to a variable with single static assignment (i.e., variable names are unique, and variables are immutable). The RHS expression resembles operators from the Weld IR,

and are much higher level than expressions in LLVM (e.g., a single for loop is an expression in the SIR). The purpose of the SIR is to simplify compilation from Weld IR to lower level, machine specific code (e.g., LLVM) in the backend. Rather than forcing each backend to compute the scope of variables, deduce control flow, etc., the SIR provides a common interface that allows backends to (in most cases) construct a simple mapping from SIR expressions to low-level code. Listing 2.10 gives an example of an SIR snippet for a simple Weld program.

2.6.2 Multi-Threaded CPU Backend

For the CPU backend, Weld JIT-compiles its optimized SIR code into multithreaded assembly code using LLVM, and then executes it on a custom runtime. Our runtime supports dynamic load balancing between threads using a work-stealing mechanism inspired by Cilk [25], allowing Weld to support workloads with irregular parallelism.

The execution engine represents a Weld program as a linear task graph with each outer `for` loop represented as a single task that is a dependency of all loops and statements that come after it. A loop can only execute after its dependencies. The engine creates a worker thread for each core and a task queue for each worker, where a task consists of a range of loop iterations. Initially, all iterations of the first loop are given to the first worker. Loops are split into multiple tasks to load-balance work; a worker steals tasks from other workers' queues when idle. An executing worker creates a new task with half its remaining iterations for the current loop when it observes that its own task queue is empty. Tasks for nested loops are created by splitting the outermost loop with more than one iteration remaining; this policy ensures that expensive inner loops will be split across cores, but smaller inner loops will not incur task creation overhead.

Much of our flexibility in the parallel builder design, in particular our ability to maintain per-thread data structures until a `result` call, is due to the *write-only, build-once* specification of the builder contract. Builders support merging values from multiple tasks. Internally, each builder type follows an API that includes an initialization step, a function to "promote" a local builder (running in just one task) to a "global" builder (when its task gets split), and a function for merging values that also receives the iteration index of the value being merged.

For example, the `merger` type will store one value per thread and combine them in its `result` operation, while the `vecbuilder` will keep inserted items in loop iterator order.

2.6.3 GPGPU Backend

GPUs excel at problems with significant data parallelism such as relational queries or computation over collections of vectors. Our GPU backend is a restricted prototype for the subset of the Weld IR that is likely to yield performance benefits compared to CPUs (avoiding tricky cases such as the construction of dictionaries using `dictmerger`). This backend translates Weld programs into Voodoo [168], a virtualization layer over OpenCL. Even in the limited subset of the language we ported, there are two performance challenges with respect to efficient computing results: flattening nested data structures and massively parallel (hierarchical) merging of values into builders.

Unnesting. Since GPUs do not support dynamic memory allocation or pointers, nested vectors must be flattened. We apply a similar unnesting process to the one described in [24]: we concatenate the nested vectors and keep a secondary vector containing the offsets of the beginnings of each of the nested vectors (similar to the compressed sparse row representation of a matrix). At runtime, iterations over the nested vectors are translated into iterations over the subranges of the flattened vector – each subrange is mapped to an OpenCL thread. SIMT GPUs experience performance penalties when vector sizes are skewed. We thus partition vectors into batches of no more than 128 elements and maintain a third vector that maps the original vector to its partition.

Massively Parallel Merging. Since builders form the core of Weld, their efficient implementation is imperative. As with the multicore backend, there are two design options: a shared data structure protected by atomic instructions or a partitioned per-core data structure that is merged when a result is produced. We select the strategy based on the (estimated) cardinality of the output: shared data structures for high cardinalities, hierarchical merges for low cardinalities. While the partition granularity of the merge is a (statically) tunable parameter, we found that processing 4096 scalar values per work item (and parallelizing the work items) consistently yields good performance.

2.6.4 NEC SX-Aurora Backend

To demonstrate a third implementation, we describe a backend designed for the NEC SX-Aurora [13]. This backend was developed by researchers at NEC, and is open source [217]. The SX-Aurora is a vector processor with wide SIMD-type registers and a multi-way arithmetic unit, making it well-suited for Weld's data-parallel execution model. It contains a high performance memory (around 1.2 TB/sec) and achieves roughly 307 Gigaflops/core.

The goal of the SX-Aurora backend was to allow the vector processor to interface with high-level data science libraries that Weld already integrates with (§2.8) that can benefit from the vector processing model. For example, applications that use Weld-enabled libraries such as NumPy and Pandas can leverage the vector processor with no code modification. Weld thus acts as a bridge between these libraries and new hardware platforms.

The SX-Aurora toolchain contains a custom C compiler that compiles C code to vector machine code. To integrate Weld, NEC researchers developed a new C-based backend that compiles the Weld SIR code into a series of C statements and loops. The C code is structured in a way that makes it amenable to auto-vectorization via the SX-Aurora compiler. Currently, the Aurora backend only supports execution on a single thread. The backend contains support for a subset of the builders, can call existing C code, and automatically manages data transfer between host memory and device memory.

2.7 Implementation

Weld's core interface and compiler are implemented in roughly 30000 lines of Rust, with parts of the parallel runtime implemented in C/C++. The core Weld API is in Rust; we generate C bindings to the Rust API, and use the C bindings to build bindings to other languages (e.g., Python and Java). We chose Rust as the implementation language because its strong pattern matching support and algebraic data types made writing optimizer passes easy. Unlike other languages with these features, Rust's minimal runtime (no garbage collection or JVM, etc.) also made the Weld library straightforward to embed into other environments. The source code is available at https://www.github.com/weld-project/weld.

Library	Glue LoC	Per-Operator LoC
NumPy	Python: 84, C++: 201	avg: 16, max: 50
Pandas	Python: 416, C++: 284	avg: 22, max: 64
Spark SQL	Python: 5, Scala: 300	avg: 23, max: 63
TensorFlow	Python: 175, C++: 652	avg: 22, max: 85

Table 2.4: Number of lines of code in our library integrations.

2.8 Library Integrations

We integrated Weld into four popular libraries: Spark SQL, TensorFlow, NumPy, and Pandas. Each integration was performed *incrementally* and supports porting only a subset of the operators to Weld while interoperating with non-ported ones. Each integration required some up front "glue code" for marshalling data and enabling lazy evaluation (if the library was eagerly evaluated), as well as code for each ported operator. Overall, we found the integration effort to be modest across the board, as shown in Table 2.4.

Spark SQL. We integrated Weld with Spark SQL [11] to accelerate its local computations on each node, which can be a bottleneck on modern hardware [153]. Spark SQL already has a lazy API to build an operator graph, and already performs Java code generation using a similar technique to HyPer [141], so porting it to use Weld was straightforward: we only needed to replace the generated Java code with Weld IR via Weld's API. Spark SQL's existing Java code generator uses complex logic [5] to directly generate imperative loops for multiple chained operators because the Java compiler cannot perform these optimizations automatically. In contrast, our Weld port emits a separate IR fragment for each operator without considering its context, and Weld automatically fuses these loops.

TensorFlow. Like Spark SQL, TensorFlow [2] also has a lazily evaluated API that generates a data flow graph composed of modular operators. Our integration with TensorFlow required two components: *(i)* a user-defined `WeldOp` operator that runs Weld programs, and *(ii)* a graph transformer that replaces a subgraph of the TensorFlow data flow graph with an equivalent `WeldOp` node. Before execution, the transformer searches the original data flow graph for subgraphs containing only operators that are understood by our port, and replaces

each such subgraph with a `WeldOp` node for their combined expression, relying on Weld to fuse these expressions. Our integration leverages TensorFlow's support for user-defined operators and graph rewriting and makes no changes to the core TensorFlow engine.

NumPy and Pandas. Our integrations with NumPy and Pandas required more effort because these libraries' APIs are *eagerly* evaluated. To enable lazy evaluation in NumPy, we created a `WeldObject`-based subclass of its array type called `weldarray`. This routes the NumPy *ufuncs* (C-based implementations of low-level operators such as addition, dot product, or element-wise logarithms) through the `weldarray` class and allows us to easily offload unsupported operations to NumPy (thus enabling incremental integration). Our integration supports most NumPy *ufuncs* and partially supports other NumPy features such as reductions and broadcasting. It also accesses existing NumPy arrays directly without copying memory, because they already stored as packed arrays of primitive types that we can pass to `NewWeldObject`.

To work around NumPy's eagerly evaluated interface, each operation adds a dependency or computation to the `weldarray` `WeldObject`, and evaluation occurs only when necessary. Our evaluation points are similar to Bohrium's [110], another effort at building a lazily evaluated NumPy. Specifically, we evaluate a `weldarray` when the data is accessed (e.g., by `print`) or when the values are needed for operators we have not ported to Weld. Like Bohrium, our port requires minimal changes to NumPy applications (just importing a different package). `weldarray` is less than 1000 lines of Python code, and did not require extensive familiarity with internals.

NumPy also supports indexed access into vectors and matrices to access specific elements in an array or to view "slices" of the array (i.e., a range of values across a set of axes). When a user performs indexed access into a `weldarray`, our port evaluates the array and defers indexing to NumPy. This also allows Weld to support NumPy's advanced indexing features [147]. Notably, indexed access removes opportunities for lazy evaluation, and as in native NumPy and Bohrium, they can be slow due to the overhead of running Python code. In general, in NumPy it is considered best practice to avoid indexed access in loops [79].

We ported Pandas in a similar way, by creating wrapper objects around the Pandas `DataFrame` and `Series` classes. We ported Pandas' filtering, sorting, predicate masking,

aggregation, groupby, merge, per-element string slicing, `getUniqueElements`, and pivot table operations to Weld. Pandas represents DataFrame columns as NumPy arrays, so we reuse code from our NumPy port to pass pointers to this data to Weld. We additionally added custom encoding functions for string data, because Pandas stores them as an array of pointers to Python string objects. We copy these strings to arrays of characters managed by our code when we need to access them from Weld.

2.9 Evaluation

Our evaluation seeks to answer the following questions:

1. Does Weld speed up realistic data science workloads?
2. Which optimizations have the greatest impact on performance?
3. Can Weld provide benefits when integrated incrementally?
4. How does Weld compare to specialized systems for domains such as relational algebra, linear algebra, and graph analytics?

Unless otherwise noted, we ran experiments on an Amazon EC2 `r4.8xlarge` instance, with 16 Intel Xeon E5-2686v4 cores (32 hyperthreads) and 244GB of memory. We used LLVM 3.8 for compilation. Each result is an average of five runs, and all end-to-end Weld runtimes include Weld optimization and LLVM compilation times, sampling time, and data encoding and decoding. We present results on one and eight threads. We compare against NumPy v1.13.1, Pandas v0.19.2, TensorFlow v1.2, and Spark v2.2.

2.9.1 Workloads and Datasets

We evaluate Weld primarily on ten real data science workloads we found from various online sources such as tutorials or cookbooks for specific libraries, popular GitHub repositories, and Kaggle competitions (Table 2.5). Each workload uses one or more of our ported libraries (§2.8). We ran the workloads with no code changes (modulo importing our versions of the libraries) when possible, except for two workloads where we also evaluated adding a grouped evaluation call across multiple results: Black Scholes and Flight Delays.

Workload	Libraries	Description (# Operators)
Data Cleaning [57]	Pandas	Cleans a DataFrame of 311 requests [1] by replacing `NULL`, broken, or missing values with `NaN`. (8)
Crime Index	Pandas NumPy	Computes an average "crime index" score, given per-city population and crime information. (16)
Black Scholes	NumPy	Computes the Black Scholes [22] formula over a set of vectors. (19)
Haversine	NumPy	Computes Haversine Dist. [81] from a set of GPS coordinates to a fixed point. (18)
N-Body	NumPy	An n-body simulation that uses Newtonian force equations to determine the position/velocity of stars over time. (38)
Birth Analysis [120]	Pandas NumPy	Given a dataset of number of births by name and year, computes the proportion of names starting with "Lesl" grouped by gender and year-of-birth. (12)
MovieLens	Pandas NumPy	Given the MovieLens dataset [80], finds the movies that are most divisive between male and female viewers. (13)
Log. Reg.	NumPy TensorFlow	Whitens and normalizes MNIST [134] images in NumPy, and then evaluates a logistic regression classifer on the whitened images in TensorFlow. (14)
NYC Filter	Pandas	Counts the number of taxi rides [149] which occur outside of Manhattan, have zero cost, and zero distance. (12)
Flight Delays	Pandas NumPy	Computes the mean delay, the unique tail numbers, the unique carriers, and the total number of flights [65] from any of four airports in NYC to Seattle. (13)

Table 2.5: Workloads used in our evaluation. An operator represents a single library API call (e.g., a pivot table construction or groupby in Pandas or an element-wise sum or logarithm in NumPy).

2.9.2 End-to-End Performance

In this section, we study the end-to-end performance of our ten workloads using Weld compared to the native versions of the libraries. Because NumPy and Pandas are single-threaded, we report results on both one and eight threads, to show that Weld improves single-thread efficiency too. §2.9.3 details the optimizations that impact performance in each workload.

Single Library Workloads

NumPy Workloads: Black Scholes, Haversine, and N-Body. These workloads perform numerical vector computations using NumPy. Figure 2.7 shows the results. Overall, Weld's optimizer improves performance over native NumPy by 2.5–5× across the three workloads, even though individual operators in NumPy are implemented in C and BLAS [116]. In Black Scholes and Haversine, Weld fuses every vector math operation into a single loop to reduce memory allocation and data movement. On a single thread, we found this improves performance by reducing page faults caused by demand paging [58] by removing allocation of intermediate results. N-Body uses NumPy's indexing features to set the diagonal of a matrix to 0; this forces Weld to evaluate a partial computation and prevents fusion of loops that occur before and after the index operation. In all workloads, NumPy implements vector operations using L1 and L2 BLAS calls, but Weld still accelerates them by reducing data movement. Weld automatically parallelizes each workload as well, increasing the speedup over the single-threaded NumPy versions to 4–30×. Data encoding time in these workloads is negligible, since NumPy internally represents vectors as C arrays and encoding only involves a pointer copy.

Pandas Workloads: Data Cleaning and NYC Filtering. These workloads use Pandas to filter, normalize and clean a DataFrame by dropping `NULL` values, selecting values that pass predicates, and replacing broken values (e.g., short zip codes) with placeholders. These workloads represent common preliminary tasks seen in data science and are often bottlenecked by memory movement.

Although most individual Pandas operators are implemented in C, NumPy or Cython [53],

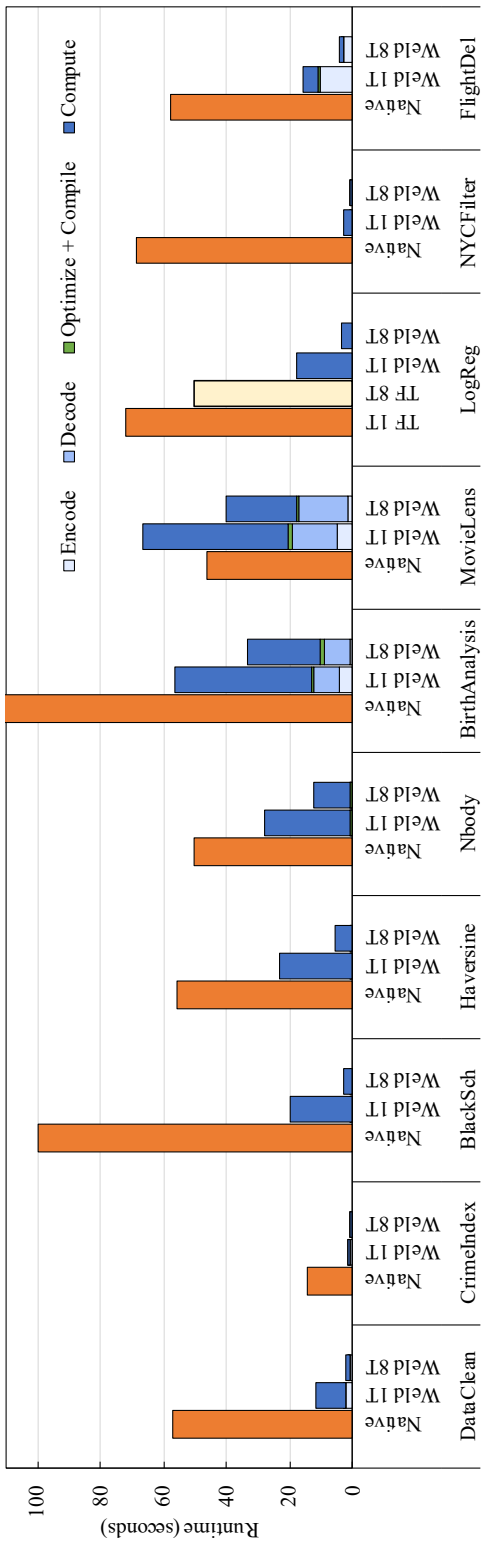


Figure 2.7: Performance of each workload on 1 and 8 threads, compared vs. native libraries. Weld's compile time includes data sampling.

Weld still provides speedups as shown in Figure 2.7. The speedups come from fusing every loop into a single one: the NYC workload in particular benefits greatly from lazy evaluation and fusion because the final output is a scalar, so after loop fusion Weld does not need to allocate any extra memory. The NYC workload shows a 23× speedup on one core, while the data cleaning workload shows a 5× speedup. Weld also provides further transparent multicore speedups over Pandas on eight threads.

Multi-Library Workloads

The remaining data science workloads from §2.9.1 combine operators from multiple libraries. Figure 2.7 shows the results for each workload, compared against its native implementation. We describe each result in more detail below.

Birth Analysis and MovieLens. These workloads construct a pivot table and filter columns using Pandas in a preprocessing stage, and then compute aggregate statistics using NumPy in an analysis phase. Weld accelerates the birth analysis workload by fusing loops in the analysis portion. The MovieLens workload is dominated by hash table performance, which is similar in both Pandas and Weld (both use an optimized C hash table). Weld does not speed up the pivot table construction substantially for this reason. Weld matches the *runtime* performance of Pandas here on one thread, showing that its general IR can capture complex workloads effectively. However, Weld performs worse than Pandas *end-to-end* on the MovieLens workload due to decoding time, since C-strings returned by Weld must be marshalled into Python strings. Furthermore, since the Python C API is not thread-safe, the decoding step cannot be parallelized on eight threads. In addition, Weld's merging of thread-local dictionaries into a final result is also serial. This is an area for future improvement and can improve scalability. Overall, Weld accelerates the end-to-end birth analysis workload by 3.5× and is within 25% on MovieLens on one thread. Weld speeds up MovieLens by 1.25× on eight threads, with 40% of the end-to-end time spent in the *serial* decode step. Efforts toward common data layouts such as Apache Arrow [12] can prevent expensive data conversion steps.

Flight Delay and Crime Index: The flight delay workload produces multiple results, each of which are computationally inexpensive to compute (e.g., a filtered column using Pandas or a mean using NumPy). Weld again fuses every loop in the workloads here, producing a single pass over the input to compute multiple results. Weld accelerates this workload by 3.7× on one thread and 14× on eight threads. Weld improves performance of the crime index workload for similar reasons by fusing the dot products used to compute the crime index into a single loop and using vectorization, by 9× and 32× on one and 8 threads.

Logistic Regression: This workload, which combines NumPy and TensorFlow to normalize (whiten) images and evaluate a logistic regression model over them, shows a 4× performance improvement via Weld over NumPy and TensorFlow *with* XLA, on a single thread. This performance improvement increases to 13× over the native library implementations with eight threads. Weld provides a speedup despite TensorFlow's specialized XLA compiler optimizing the scoring computation because it co-optimizes the image whitening task with the model scoring task. With eight cores, the speedup is due to parallelizing the whitening computation; in the native library implementation of the workload with eight cores, TensorFlow parallelizes the model scoring but NumPy continues to run on a single thread. Performance improvements over NumPy and TensorFlow without XLA were slightly better; we omit them for brevity.

Optimization and Sampling Times

Weld's optimization times (including IR optimization and LLVM code generation) ranged from 62ms to 257ms (mean 126ms) across all workloads. The overhead of adding sampling for adaptive predication similarly ranged from 100–250ms. Since we expect real analytics workloads to run for many seconds to minutes (as in our experiments), we believe that these times are acceptable. Our optimizer is thus able to produce high-quality optimization decisions quickly in ad-hoc workloads with no statistics.

Experiment	All	-Fuse	-Unrl	-Pre	-Vec	-Pred	-Grp	-ADS	-CLO
DataClean	1.00	1.82	1.05	1.06	1.05	1.05			
CrimeIndex	1.00	15.86	3.17	1.02	1.01	1.01			2.95
BlackSch	1.00	2.72		1.00	1.85		1.51		
Haversine	1.00	2.19		1.06	1.01	1.01			
Nbody	1.00	1.69		1.50	1.12	1.02			
BirthAn	1.00	1.07		1.05	0.98				1.00
MovieLens	1.00	1.07		1.00	0.98				1.01
LogReg	1.00	2.51		0.99					1.25
NYCFilter	1.00	11.21		0.99	1.40	4.45			
FlightDel	1.00	2.02		1.01	1.00	1.00	4.59		2.16
NYC-Sel	1.00	62.68	0.99		1.03	0.99			
NYC-NoSel	1.00	10.27	0.99		1.60	1.49			
Q1-Few	1.00	1.38							
Q1-Many	1.00	1.08							
Q3-Few	1.00	1.23							
Q3-Many	1.00	1.10							
Q6-Sel	1.00	1.45	0.97	0.95	1.00	1.05			
Q6-NoSel	1.00	10.00	1.01	1.02	2.69	2.49			

Figure 2.8: Slowdown from removing each optimization (while applying all others) on one thread. Fuse = Fusion, Unrl = Loop Unrolling, Pre = Preallocation, Vec = Vectorization, Pred = Adaptive Predication, Grp = Grouped Eval, ADS = Adaptive Data Structures, CLO = Cross-Library Optimization.

2.9.3 Effects of Individual Optimizations

In this section, we study the effects that *individual* optimizations have on the runtime of the ten workloads. We specifically consider the optimizations discussed in §2.5.1–2.6: fusion, vectorization, loop unrolling, buffer preallocation, the runtime API's grouped evaluation feature, adaptive predication, adaptive data structures, and cross-library optimization. For each workload, we turn each optimization off *one at a time* and measure its impact.

Figure 2.8 shows the results of this study on a single thread. Each box shows the relative slowdown of turning the optimization off, compared to having all optimizations on (i.e., numbers close to 1.0 mean the optimization had little effect, while larger numbers mean a large effect). Blank entries mean the optimization did not apply, and colors show the scale of the effect. Figure 2.9 shows the same experiment with eight threads. To show the

effects of adaptive predication and data structures, we also ran additional versions of some workloads with different selectivity shown below the line (the Sel and NoSel versions), as well as several TPC-H queries where we varied selectivity and number of keys; we discuss these below.

Fusion. The fusion transformations have the most impact across our workloads. This shows that optimizing memory allocation and data movement has a substantial cost in these workloads. Fusion optimizations affect every real workload except MovieLens (which is dominated by one operation) by up to 15× on one thread. Optimizing memory movement is especially important on on eight threads, where there is less memory bandwidth per thread. Data science libraries will thus need some form of cross-operator fusion to achieve optimal performance, and would benefit greatly from lazy runtime APIs like Weld's (indeed, many newer libraries use lazy APIs and optimizers on top of them to tackle this problem [2, 11]).

Vectorization. Vectorization also shows significant impact in several compute-heavy workloads. In some cases, vectorization can only be applied if predication is used, which requires adaptivity.

Grouped Evaluation. This API feature takes lazy evaluation allows users to submit multiple results to evaluate at once at the cost of requiring a small code change. It helped in two workloads where, without grouping, Weld had to recompute a common subexpression that would otherwise be shared across the evaluation of two results.

Preallocation, Loop Unrolling: These optimizations impact CPU efficiency by reducing memory allocations or other overheads. They help most in NumPy workloads with fixed-size arrays.

Adaptive Predication: To show the effects of adaptive predication on different selectivities, we ran the NYC Taxi workload on synthetic data with both high and low selectivity. Turning adaptive predication off always runs the branched version of the code. The `NYC-Sel` and `NYC-NoSel` entries in Figures 2.8 and 2.9 show the results. In `NYC-Sel`, most data is

Experiment	All	-Fuse	-Unrl	-Pre	-Vec	-Pred	-Grp	-ADS	-CLO
DataClean	1.00	2.44	0.97	0.99	0.98	0.95			
CrimeIndex	1.00	195	2.04	1.00	1.02	0.96			3.23
BlackSch	1.00	6.68		1.44	1.95		1.64		
Haversine	1.00	3.97		1.20	1.02				
Nbody	1.00	1.78		2.22	1.01				
BirthAn	1.00	1.02		0.97	0.98				1.00
MovieLens	1.00	1.07		1.02	0.98				1.09
LogReg	1.00	20.18		1.00					2.20
NYCFilter	1.00	9.99		1.20	1.23	2.79			
FlightDel	1.00	1.27		1.01	0.96	0.96	5.50		1.47
NYC-Sel	1.00	32.43	1.29		0.96	0.93			
NYC-NoSel	1.00	6.16	1.02		1.26	1.17			
Q1-Few	1.00	2.60						3.75	
Q1-Many	1.00	1.13						1.12	
Q3-Few	1.00	1.86						2.56	
Q3-Many	1.00	1.10						0.97	
Q6-Sel	1.00	1.45	1.00	1.00	0.99	0.98			
Q6-NoSel	1.00	10.04	0.99	0.99	2.44	2.66			

Figure 2.9: Slowdown from removing each optimization on 8 threads.

filtered by the first predicate in the workload, so adaptive predication chooses not to predicate the code and removing adaptive predication does nothing. `NYC-NoSel` shows the opposite effect: the filter always passes, so removing adaptive predication (and thus vectorization) results in a slowdown over the adaptively chosen plan, which would be to apply predication. We also show the same experiment on TPC-H Q6, which performs a number of filters and performs an aggregation, to the same effect.

Adaptive Data Structures: We ran a similar benchmark to show the effect of a non-adaptive dictionary on modified versions of TPC-H Q1 (an aggregation) and TPC-H Q3 (which contains a hash join). We again ran two versions of each experiment, varying the number of unique keys in the hash table. Figure 2.9 shows the results (`Q1-Few/Many` and `Q3-Few/Many`, referring to the number of distinct keys). The `Few` setting used 1024 distinct keys, and the `Many` setting used 2^{28} keys. We used 2^{31} total records in all cases. The comparison in Figure 2.9 is between the adaptive dictionary and the worse of the two simple

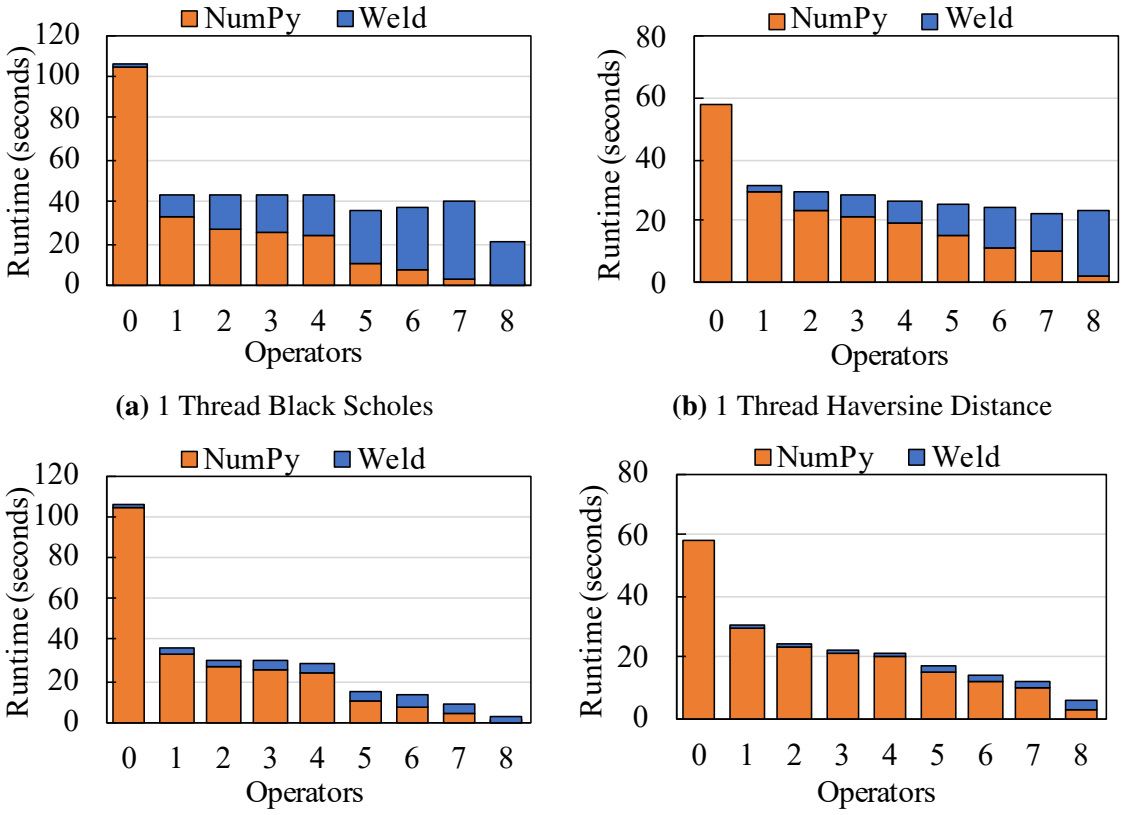
dictionary implementation strategies: global and thread-local. We observe up to a 3.75× performance difference between the adaptive dictionary and the worst alternative.

Cross-Library Optimizations: We evaluate the effect of cross-library optimizations (CLOs) by forcing Weld to evaluate computations at library boundaries. Disabling CLO for Crime Index, Flight Delays, and Logistic Regression both prevents loop fusion and causes Weld to allocate temporary buffers for intermediate values, resulting in slowdowns. Birth Analysis and MovieLens predominately use Pandas operators, and only use NumPy for a small subset of the data after calling a top-K operator in Pandas over the pivot table columns. These workloads are also primarily bottlenecked by hash table operations in Pandas, so disabling CLO has a relatively small effect. Overall, the results show that CLO provides up to 3× *further* speedups even after optimizing computations in individual libraries.

2.9.4 Incremental Integration

To show that Weld can be integrated into libraries incrementally, we ran the Black Scholes and Haversine workloads and incrementally integrated one operator at a time to use Weld. Operators that were not Weld-enabled used native NumPy. The workloads both use eight unique operators; we ported these one by one to Weld in order of which operator took the most CPU cycles in each workload.

Figure 2.10 shows the results. On one thread, implementing the first operator in Weld gives a 1.5× speedup, and implementing half the operators gives a 2.5× speedup over native NumPy, by providing vectorized implementations for functions that NumPy runs sequentially. Interestingly, adding operators in Black Scholes slightly regressed performance in two cases, due to an extra `Evaluate` call, which causes some recomputation. On eight threads, the speedups at each step are significantly higher because Weld accelerates even the operators that can't yet be fused with other nearby ones by multithreading them. These results indicate that library developers can add Weld incrementally into the most widely used operators to start enjoying speedups without fully porting their libraries.



(a) 1 Thread Black Scholes

(b) 1 Thread Haversine Distance

(c) 8 Thread Black Scholes

(d) 8 Thread Haversine Distance

Figure 2.10: Incremental Integration on two of the NumPy workloads.

2.9.5 Comparison to Specialized Systems

We also evaluated our optimizer's code generation on x86 CPUs by comparing performance on several workloads against several state-of-the-art, domain-specific compilers and systems.

TPC-H Queries. Figure 2.11 show the results for a subset of the TPC-H queries (scale factor 10) on eight threads, compared against the HyPer [141] database and a hand-optimized C baseline using Intel's AVX2 intrinsics for vectorization and OpenMP [152] for parallelization. We chose these queries because they cover the major join types, scans, predicates, and aggregations and do not include complex string operations. HyPer generates LLVM IR for SQL queries, which is then compiled to machine code before execution. For Weld, we used the same physical plan as HyPer from its web interface [85] but wrote each operator in Weld.

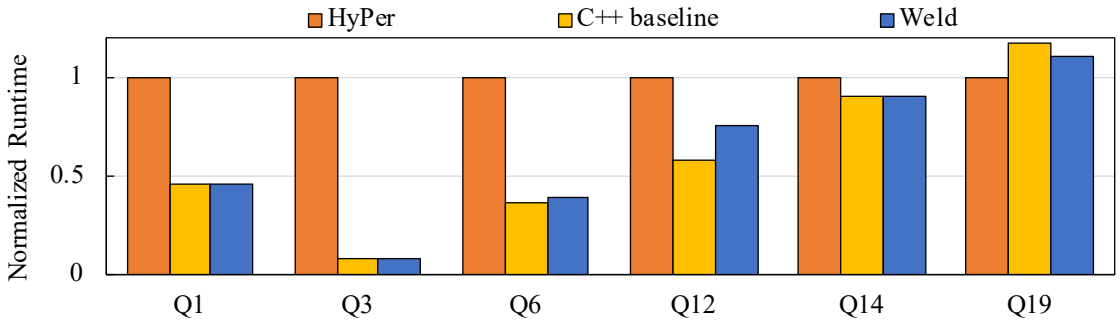
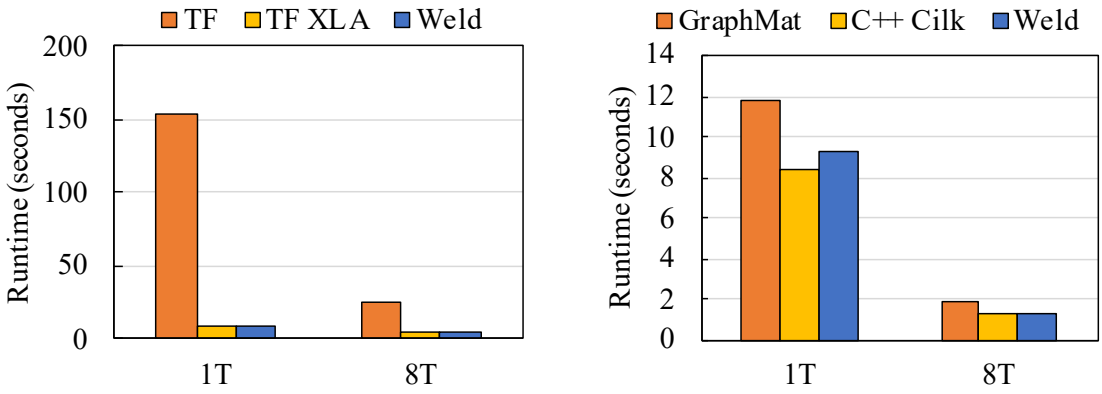


Figure 2.11: TPC-H microbenchmarks on 8 threads, compared against Hyper and hand-written C++ code using OpenMP for threading. Each result is normalized to HyPer's performance.

Execution time was competitive with HyPer across the board. Weld outpeformed HyPer on Q6 and Q12 because it applied predication and generated vectorized LLVM code. Results on one thread were similar.

Linear Algebra. Figure 2.12a shows Weld's performance on training a binary logistic regression classifier on the MNIST [134] dataset, classifying each digit as either zero or non-zero. We compare against TensorFlow with and without its specialized XLA compiler [220]. Unlike the logistic regression example in §2.9.2, the computation here is fully exposed to XLA for optimization, allowing it to achieve its best performance. Weld and XLA outperform standard TensorFlow by fusing operators, but perhaps surprisingly, Weld also matches XLA's performance even though XLA is built specifically for TensorFlow's linear algebra operators. We also compared Weld against Bohrium, an open-source lazy NumPy. Figure 2.13 shows the results; Weld outperforms Bohrium by vectorizing more operators and using grouped evaluation to eliminate redundant computation.

Finally, we benchmarked Weld's performance on dense matrix multiplication (DMM). This workload is both compute-bound and heavily optimized by existing linear algebra libraries. We compare Weld's generated code against a C baseline and MKL [132], the fastest DMM implementation we are aware of on Intel CPUs. We implemented the C and Weld kernels using (*1*) naive, triply nested loops and (*2*) blocked loops for improved cache performance. The MKL implementation is proprietary but highly optimized and written in assembly to control aspects such as instruction scheduling. On a 8192×8192 matrix,



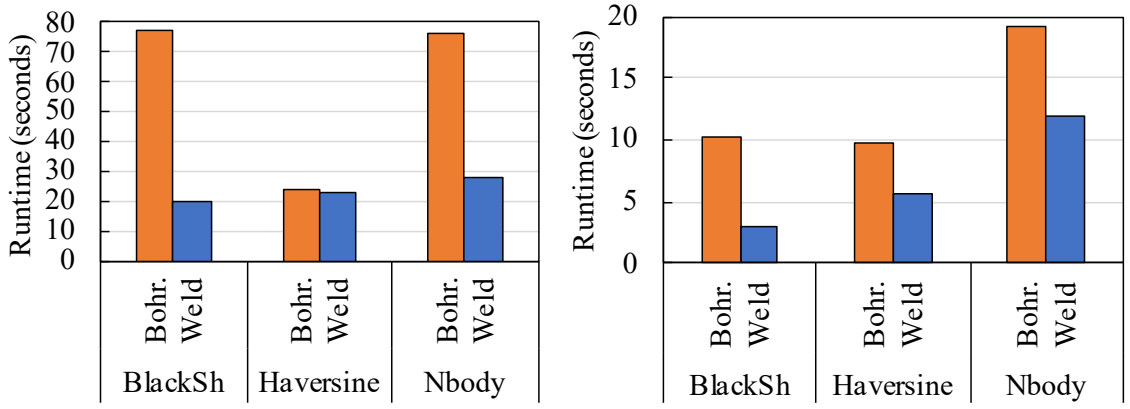
(a) Logistic Regression **(b)** PageRank

Figure 2.12: Specialized systems for linear algebra and graphs.

we found that Weld's generated code matches C, but MKL outperforms both even against the blocked algorithm by 10×. Weld imposes no overhead when calling MKL as a UDF compared to a C program calling MKL. In short, like in systems designed for linear algebra [2, 49], users should call existing DMM kernels by using UDFs rather than relying on code generation to achieve the best performance.

Graph Analytics. Figure 2.12b shows results for a PageRank implementation in Weld, compared against the GraphMat [202] graph processing framework. GraphMat had the fastest multicore PageRank implementation we found. Weld's per-iteration runtime for both serial and parallel code outperforms GraphMat, and is competitive with a hand-optimized Cilk [25] based C++ baseline.

Spark SQL. To illustrate Weld's benefits in a distributed framework, we evaluate a partial integration of Weld in Spark SQL's execution engine. Spark SQL natively generates Java bytecode, and uses a HyPer-like code generation process to pipeline code from different operators. In our Weld integration, we updated each Spark SQL operator to emit single Weld IR fragment encapsulating its computation, and relied on Weld's optimizer to fuse and co-optimize these fragments (§2.8). We tested the Weld integration on TPC-H queries 1 and 6 with 20 Amazon EC2 `r3.xlarge` worker instances (2 cores, 30 GB memory) and



(a) 1 Thread **(b)** 8 Threads

Figure 2.13: Performance of Weld vs. Bohrium.

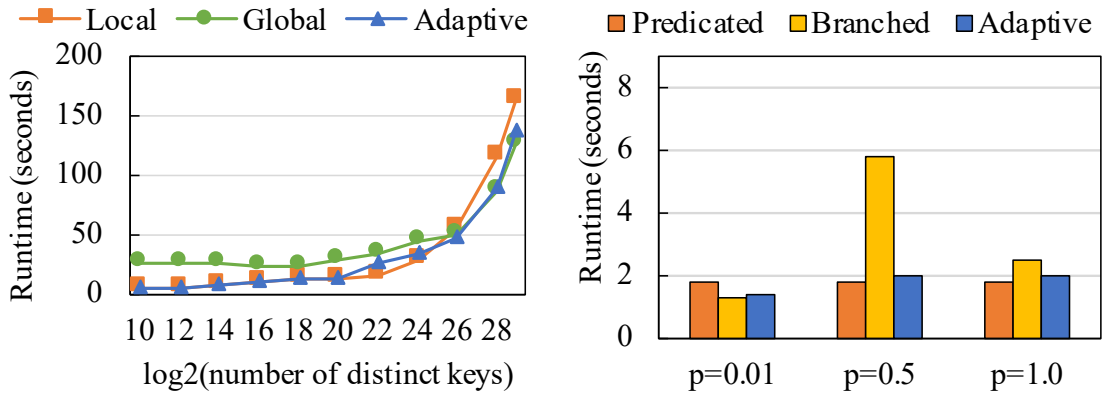
800GB of TPC-H data (scale factor 800). We chose these two queries because they only contain scans, filters, and aggregations that our current Weld integration supports. Data was read from Spark's in-memory cache. We observed that Weld provides a 6.2× speedup for TPC-H Q1 and 6.5× for Q6, with Weld's speedup coming largely from its ability to generate low-level, vectorized x86 code, since the JVM did not vectorize Spark's code.

Memory Usage. In each experiment, Weld's runtime memory usage matched the memory usage of the optimized C baseline implementations. In summary, Weld produces machine code competitive with existing systems on a variety of workloads, demonstrating both the expressiveness of its IR and the effectiveness of our optimizer.

2.9.6 Adaptive Optimization Microbenchmarks

In this section, we evaluate the quality of the adaptive transformations across a variety of inputs. For the dictionary, we plotted the runtime of a global-only, local-only, and adaptive dictionary implementation while varying the number of distinct keys in TPC-H Q1. We used 2^{31} total records in all cases. Figure 2.14a shows our results. In all cases, the adaptive dictionary achieves the performance of the better of the two dictionary strategies.

Figure 2.14b plots the runtime of a predicated, branched, and adaptively predicated version of TPC-H Q6 at various selectivities. Once again, adaptive predication matches the



(a) Adaptive Dictionaries **(b)** Adaptive Predication

Figure 2.14: 2.14a. Performance of TPC-H Q1 with three different dictionary implementation strategies. The experiments all used 16 threads. **2.14b.** adaptively predicating the branch in TPC-H Q6.

better of the branched and predicated strategies in the three cases. As the selectivity of the branch increases, the adaptive optimizer determines that evaluating a conditional branch has a lower cost than unconditionally evaluating both branch targets, and disables predication.

2.9.7 Results on Heterogeneous Hardware

Weld's IR is designed to support diverse hardware platforms. Weld currently supports two hardware backends apart from the CPU: a GPU backend built on top of the Voodoo vector algebra [168], and a backend for the NEC SX-Aurora [13] HPC vector processor, developed and maintained by NEC. We show results for both below.

GPU Results

We ran a set of Weld programs on a prototype GPU backend built on top of the Voodoo engine (§2.6) to illustrate that the abstractions in the IR are sufficient to enable fast code generation across platforms. We compare our GPU results to other existing systems that also target GPU execution using an NVIDIA GeForce GTX Titan X with 12 GB of internal DDR5 memory running NVIDIA's OpenCL implementation (version 367.57, CUDA 7.5). The results are summarized in Figure 2.15.

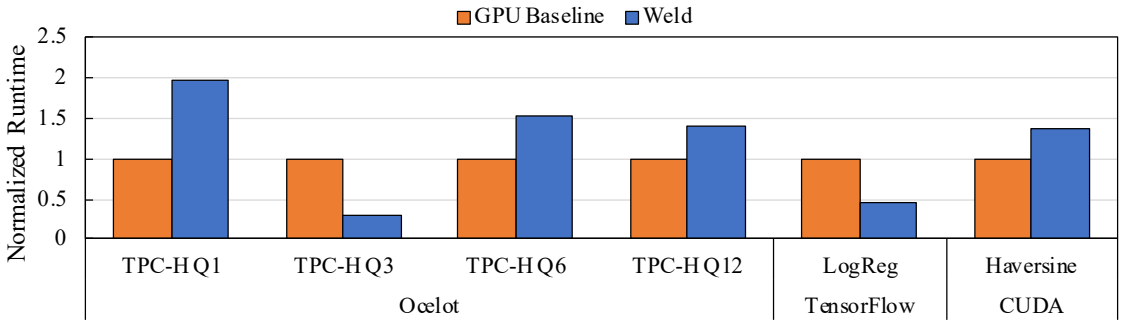


Figure 2.15: Results of Weld vs. state-of-the-art systems optimized for GPUs for relational algebra (Ocelot) and linear algebra (TensorFlow, custom CUDA kernel). Weld's generated code is competitive with each baseline. Results are normalized to the performance of the baseline.

TPC-H. Figure 2.15 shows the results of running four of the TPC-H queries from Figure 2.11 against Ocelot [83], a database optimized for GPUs. We find that for most queries, Weld generates OpenCL code that generally outperforms the CPU backend and is on par with Ocelot, apart from Q1 where it is within a factor of 2× of Ocelot.

Logistic Regression. Figure 2.15 also shows the results of running the logistic regression classifier as before, using TensorFlow (with its GPU backend enabled) with and without Weld integration. Weld's generated code outperforms TensorFlow's GPU-enabled operator graph by 2.2×.

Haversine. Finally, Figure 2.15 shows the performance of the Haversine distance benchmark on a GPU, compared against a hand-optimized CUDA implementation that computes the result for each data point with a single dense kernel. Weld performs within 1.4× of the hand-optimized implementation.

Overall, Weld produces competitive performance on GPUs across a variety of different workloads. Our Weld backend can also be further optimized, e.g., by optimizing data transfers by using features such as CUDA streams [48], which our current implementation does not do.

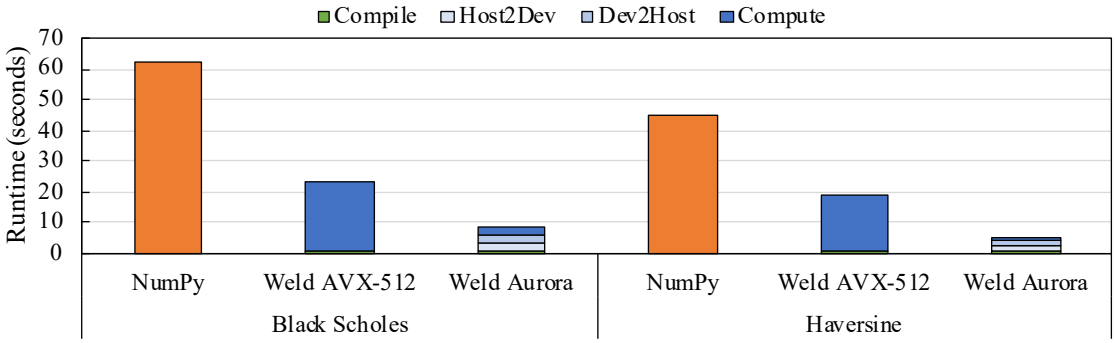


Figure 2.16: Results of Weld with AVX-512 vs. the NEC SX-Aurora backend.

NEC SX-Aurora Results

Researchers at NEC [2] have developed a Weld backend for the SX-Aurora vector processor [217], as described in §2.6.4. The backend allows Weld-enabled libraries to run on the vector processor without modification. Figure 2.16 shows two results, comparing the CPU implementation of Black Scholes and Haversine on a modern Intel CPU with AVX-512 [17] (Intel's latest and widest vector register set), and the SX-Aurora. We also show the breakdown in runtime. All results are on a single thread.

End-to-end, the Aurora backend outperforms the AVX-512 backend by 2.8× for Black Scholes and 3.4× for Haversine, and outperforms native NumPy by over 10×. The breakdown shows that most of the time is spent in data transfer between host memory and accelerator memory: the accelerator reduces the computation time by 15–30× compared to the AVX-512 code. We note that the Weld integration does not overlap processing time and data transfer time: this optimization can further improve performance by masking the cost of moving data to the accelerator. Workloads that are more computationally expensive can also reduce relative data transfer overhead.

[2]These results are taken from a supercomputing presentation by NEC, and is work done by researchers at NEC, not the author of the thesis. We present it here to demonstrate Weld's applicability to heterogeneous hardware. Special thanks to Harumichi Yokoyoma for making these results available for this dissertation.

2.10 Limitations

Weld's interface and implementation have several limitations. First, Weld currently only aims to accelerate single-machine code in a shared-memory environment (e.g., multicore CPU or GPU). Its IR includes shared-memory operations such as random lookups into an array, which are difficult to implement efficiently in a distributed setting. Nonetheless, Weld can also be integrated into distributed systems such as Spark to accelerate each node's local computations. These distributed frameworks are often CPU- or memory-bound [11, 45, 153]. We show in § 2.9 that by optimizing computation on each node, Weld can accelerate Spark applications by 5-10×.

Second, Weld's IR cannot express asynchronous computations where threads race to update a result [180]; all Weld programs are race-free. It also lacks constructs to let programmers control locality (e.g., pinning data to a CPU socket).

Third, Weld executes computations lazily, which makes programs harder to debug. Fortunately, lazy APIs are becoming very common in performance-sensitive systems such as LINQ [131], Spark and TensorFlow, and we believe that programmers can use similar debugging techniques (e.g., printing an intermediate result).

Finally, Weld requires integration into libraries in order to speed up applications. As discussed in §2.2, we believe that this is worthwhile for several reasons. First, many libraries already write their performance-sensitive operators in C or OpenCL; Weld offers a higher-level (functional) interface to write code that is more hardware-independent. Second, Weld enables significant speedups *even within a single library*, creating incentives for individual libraries to use it. Finally, Weld can be integrated incrementally and still offer substantial speedups. Nevertheless, in this dissertation, we also explore another design point in Chapter 3 that provide some of the benefits of Weld with no code modification by both the end-user and library developer.

2.11 Related Work

Weld builds on ideas in multiple fields, including compilers, parallel programming models and database engines. Unlike existing systems, however, Weld aims to design an interface

that can be used to optimize across *diverse existing libraries* instead of creating a monolithic programming model in which all applications should be built. We show that such cross-library optimization is crucial for performance in workloads that combine multiple libraries. In addition, unlike systems that do runtime code generation for more specific workloads (e.g., databases [141] or linear algebra [220]), Weld offers a small and general IR that can achieve state-of-the-art performance *across* these workloads.

Multiple prior systems aim to simplify programming parallel hardware. Hardware-independent intermediate languages such as OpenCL [198], LLVM [115], and SPIR [106] are based on low-level sequential instruction sets where threads communicate via shared memory, which makes it hard to perform complex optimizations for *parallel* code, such as loop fusion or loop tiling across parallel functions. Moreover, the APIs to these systems are evaluated eagerly, resulting in multiple independent invocations when applications combine multiple libraries. In .NET languages, LINQ [131] offers a lazily evaluated API that has been used to target clusters [222] and heterogeneous hardware [183] using relational algebra as an IR. However, LINQ is designed for the "closed" world of .NET programs and does not provide a means of interfacing with code outside the .NET VM. Moreover, although its relational IR can support various fusion optimizations [139], it is difficult to express other optimizations such as loop tiling. Spark [225], FlumeJava [38] and other systems [9, 70] also perform optimizations using a relational or functional IR, while Pydron [138] uses annotations to parallelize Python code.

Runtime code generation has been used in systems including databases [11, 45, 141] and TensorFlow's XLA compiler [220]. However, most of these systems are limited to a narrow workload domain (e.g., linear algebra or SQL with sequential user-defined functions [45]). Our work shows that Weld's more general IR, together with its optimizer, enables code generation on par with these systems for multiple key workloads and simultaneously allows optimizing across them.

The compilers literature is also rich with parallel languages, IRs and domain-specific languages (DSLs) [23, 25, 37, 185, 201]. However, most of this work focuses on static compilation of an entire program, whereas Weld allows dynamic compilation at runtime, even when programs load libraries dynamically or compose them in a dynamically typed language like Python. Weld's IR is closest to monad comprehensions [75] and to Delite's

DMLL [32], both of which support nested parallel loops that can update multiple results per iteration.[3] Builders are also similar to Cilk reducers [67] and to LVars [112], although these systems to not implement them in different ways on different hardware platforms. Finally, Weld's optimization techniques (e.g., loop fusion) are standard, but our contribution is to demonstrate that they can be applied easily to Weld's IR and yield high-quality code even when libraries are combined using the narrow interface Weld provides.

2.12 Summary

This chapter presented Weld, a novel interface and runtime for accelerating data-intensive workloads using disjoint functions and libraries. Today, these workloads often perform an order of magnitude below hardware limits due to extensive data movement. Weld addresses this problem through an intermediate representation (IR) that can capture data-parallel applications and a runtime API that allows IR code from different libraries to be combined. Together, these ideas allow Weld to bring powerful optimizations to multi-library workflows without changing their user-facing APIs. We showed that Weld is easy to integrate into Spark SQL, TensorFlow, Pandas and NumPy and provides speedups of up to 6.5× in workflows using a single library and 180× across libraries. Importantly, Weld also shows that changing the interface for software composition is a fruitful direction for extracting the best performance from modern hardware. In the next chapter, this dissertation looks at a different approach for achieving some of the same optimizations as Weld, *without having to change library code*.

[3] Weld has some differences from these IRs, however, most notably that it represents builders as a first-class type. For example, in Delite, the result of each loop is a collection, not a builder, meaning that nested loops cannot efficiently update the same builder unless the compiler treats this case specially.

Chapter 3

Split Annotations

3.1 Introduction

In the previous chapter, we showed that *data movement* between memory and the CPU is one of the largest bottlenecks in data-intensive applications [154]. Hardware parallelism such as multicore and SIMD has caused computational throughput to outpace memory bandwidth by an order of magnitude over several decades [99, 128, 219]. As a result of this trend, optimizing each library function in isolation is no longer enough to achieve the best performance in data-intensive applications that compose many such functions.

In response to this problem, Weld proposed rewriting library functions using an intermediate representation (IR) to enable cross-function data movement optimization, parallelization, and JIT-compilation. Data movement optimizations such as loop fusion alone showed improvements of two orders of magnitude in real data analytics pipelines [154, 221]. Other compiler-based systems such as TensorFlow XLA [2, 220] and others [110, 117, 176, 182, 200, 201] have also shown promising results by using an IR and compiler to optimize data movement. Finally, compiler-based approaches also have opportunities to perform other optimizations (e.g., common subexpression elimination) and can even target other platforms via code generation (e.g., both Weld and XLA can generate code for GPUs).

Despite these advantages, a significant drawback of using compilers to enable the most impactful data movement optimizations specifically is that they are highly complex to implement. This manifests in two major disadvantages. First, leveraging these compilers

requires intrusive changes to the library itself. Many of these systems [39, 52, 110, 155, 220] require reimplementing each operator in entirety to obtain any benefits. In addition, the library must often be redesigned to "thread a compiler" through each function by using an API to construct a dataflow graph (e.g., TensorFlow Ops [206] or Weld's Runtime API [155]). These restrictions impose a large burden of effort on the library developer and hinder adoption–for example, if one library developer decides that the compiler integration is too expensive to maintain, that library cannot be part of optimizations. Second, the code generated by compilers might not match the performance of code written by human experts. For example, even state-of-the-art compilers tailored for linear algebra [39,176,220] generate convolutions and matrix multiplies that are up to 2× slower than hand-optimized implementations [78,207,210]. Developers thus face a tough choice among expanding these compilers, dropping their own optimizations, or forgoing optimization across functions.

In this chapter, we propose a new technique called *split annotations*, which provides the data movement and parallelization optimizations of existing compilers and runtimes *without* requiring modifications to existing code. Unlike prior work that requires reimplementing libraries, our technique only requires an *annotator* (e.g., a library developer or third-party programmer) to annotate functions with a split annotation (SA) and to implement a *splitting API* that specifies how to split and merge data types that appear in the library. Together, the SAs and splitting API define algebraic rules for how data passed into an unmodified function can be partitioned into cache-sized chunks, pipelined with other functions to reduce data movement, and parallelized automatically by a runtime transparent to the library. We show that SAs yield similar performance gains with up to 17× less code compared to prior systems that require function rewrites, and can even outperform them by as much as 2× by leveraging existing hand-optimized functions.

There are several challenges in enabling data movement optimization and automatic parallelization across black-box library functions. First, a runtime cannot naïvely pipeline data among all the annotated functions: it must determine whether function calls operating over split data are compatible. For example, a function that operates on rows of pixels can be split and pipelined with other row-based functions, but not with an image processing algorithm that operates over patches (similar to determining valid operator fusion rules in compilers [37, 110, 155, 220]). This decision depends not on the data itself, but on the shape

of the data (e.g., image dimensions) at runtime. To address this challenge, we designed the SAs around a type system with two goals: (*1*) determining how data is split and merged, and (*2*) determining which functions can be scheduled together for pipelining. Annotators use SAs to specify a *split type* for each argument and return value in a function. These types capture properties such as data dimensionality and enable the runtime to reason about function compatibility. For each split type, annotators implement a *splitting API* to define how to split and merge data types.

Second, in order to pipeline data across functions, the runtime requires a lazily evaluated dataflow graph of the annotated functions in an application. While existing systems [110, 133, 154, 201] require library developers to change their functions to explicitly construct and execute such a graph using an API, our goal is to enable these optimizations without library modifications. This is challenging since most applications will not only make calls to annotated functions, but also execute arbitrary un-annotated code that cannot be pipelined. To address this challenge, we designed a client library called `libmozart` that captures a dataflow graph from an existing program at runtime and determines when to execute it without library modification, and minor to no changes to the application. We present designs for `libmozart` for both C++ and Python. Our C++ design generates transparent wrapper functions to capture function calls lazily, and uses memory protection to determine when to execute lazy values. Our Python design uses decorators and value substitution to achieve the same result. In both designs, `libmozart` requires no library modifications.

Finally, once the client library captures a dataflow graph of annotated functions, a runtime must determine how to execute it efficiently. We designed a runtime called *Mozart* that uses the split types in the SAs and the dependency information in the dataflow graph to split, pipeline, and parallelize functions while respecting correctness constraints.

We evaluate SAs by integrating them with several data processing libraries: NumPy [146], Pandas [157], spaCy [194], Intel MKL [132], and ImageMagick [87]. Our integrations require up to 17× less code than an equivalent integration with an optimizing compiler. We evaluate SAs' performance benefits on the data science benchmarks from the Weld evaluation [154], as well as additional image processing and numerical simulation benchmarks for MKL and ImageMagick. Our benchmarks demonstrate the generality of SAs and include options pricing using vector math, simulating differential equations with matrices

and tensors, and aggregating and joining SQL tables using DataFrames. End-to-end, on multiple threads, we show that SAs can accelerate workloads by up to 15× over single-threaded libraries, up to 5× compared to already-parallelized libraries, and can *outperform* compiler-based approaches by up to 2× by leveraging existing hand-tuned code. Our source code is available at https://www.github.com/weld-project/split-annotations.

Overall, we make the following contributions:

1. We introduce split annotations, a new technique for enabling data movement optimization and automatic parallelization with no library modifications.

2. We describe `libmozart` and Mozart, a client library and runtime that use annotations to capture a dataflow graph in Python and C code, and schedule parallel pipelines of black-box functions safely.

3. We integrate SAs into five libraries, and show that they can accelerate applications by up to 15× over the single-threaded version of the library. We also show that SAs provide performance competitive with existing compilers.

3.2 Motivation and Overview

Split annotations (SAs) define how to split and pipeline data among a set of functions to enable data movement optimization and automatic parallelization. With SAs, an *annotator*—who could be the library developer, but also a third-party developer—annotates functions and implements a splitting API that defines how to partition data types in a library. Unlike prior approaches for enabling these optimizations under existing APIs, SAs require no modification to the code library developers have already optimized. SAs also allow developers building new libraries to focus on their algorithms and domain-specific optimizations rather than on implementing a compiler for enabling cross-function optimizations. Our annotation-based approach is inspired by the popularity of systems such as TypeScript [212], which have demonstrated that third-party developers can annotate existing libraries (in TypeScript's case, by adding type annotations [213]) and allow other developers to reap their advantages.

Listing 3.1 Snippet from the Black Scholes options pricing model implemented using Intel MKL.

```
// inputs are 'double' arrays with 'len' elements
vdLog1p(len, d1, d1);          // d1 = log(d1)
vdAdd(len, d1, tmp, d1);       // d1 = d1 + tmp
vdDiv(len, d1, vol_sqrt, d1); // d1 = d1 / vol_sqrt
```

3.2.1 Motivating Example: Black Scholes with MKL

To demonstrate the impact of SAs, consider the code snippet in Listing 3.1, taken from an Intel MKL implementation of the Black Scholes options pricing benchmark. This implementation uses MKL's parallel vector math functions to carry out the main computations. Each function parallelizes work using a framework such as Intel TBB [181] and is hand-optimized using SIMD instructions. Unfortunately, when combining several of these functions on a multicore CPU, the workload is bottlenecked on *data movement*.

The data movement bottleneck arises in this workload because each function completes a full scan of every input array, each of which contains `len` elements. For example, the call to `vdLog1p`, which computes the element-wise logarithm of the `double` array `d1` in-place, will scan `sizeof(double)*len` bytes: this will often be much larger than the CPU caches. The following call to `vdAdd` thus cannot exploit any locality of data access, and must *reload values for `d1` from main memory*. Since individual MKL functions only accept lengths and pointers as inputs, their internal implementation has no way to prevent these loads from main memory *across* different functions. On modern hardware, where computational throughput has outpaced memory bandwidth between main memory and the CPU by over an order of magnitude [99, 128, 219], this significantly impacts multicore scalability and performance, even in applications that already use optimized libraries.

SAs and their underlying runtime, Mozart, address this data movement bottleneck by splitting function arguments into smaller pieces and pipelining them across function calls. Listing 3.2 shows the SAs an annotator could provide for the three functions in Listing 3.1. We describe the SAs fully in §3.3, but at a high level, annotators use the SA and a new abstraction called split types to define how to split each function argument into small pieces. For example, the annotator can define a split type `ArraySplit` to indicate that the

Listing 3.2 SAs for three functions in Intel MKL.

```
@splittable(size: SizeSplit(size), a: ArraySplit(size),
  mut out: ArraySplit(size))
void vdLog1p(long size, double *a, double *out);

@splittable(size: SizeSplit(size), a: ArraySplit(size),
  b: ArraySplit(size), mut out: ArraySplit(size))
void vdAdd(long size, double *a, double *b, double *out);
void vdDiv(long size, double *a, double *b, double *out);
```

array arguments be split into smaller, regularly-sized arrays. The split type `SizeSplit` then indicates that the size argument be split to represent the lengths of these arrays. Annotators then bridge the abstraction of the split type with code that performs the splitting (e.g., by offsetting into the original array pointer) by implementing a *splitting API* for each split type.

Given these SAs, Mozart executes the code in Listing 3.1 in a markedly different way. Instead of calling each MKL function on the full input arrays at once, Mozart splits each array into small, equally-sized chunks that collectively fit in the CPU cache (e.g., chunks of 4096 elements per array). Mozart then assigns these chunks across threads, and has each thread call all the functions in the pipeline in sequence (`vdLog1p`, `vdAdd`, etc.) on one chunk at a time. Data for each chunk resides in each CPU's local cache across all functions. Although the total number of loads and stores remains unchanged (i.e., across all chunks, each function still loads and processes all elements of `d1`, `tmp`, and `vol_sqrt`), each array element is loaded from main memory *only once* and served from cache for all subsequent accesses. This is a stark reduction compared to the original execution, which loads each array from main memory for each function call.

Figure 3.1 shows the impact of SAs on the full Black Scholes benchmark, which contains 32 vector operations, on a modern Intel Xeon CPU. The benchmark runs on 11GB of data. While un-annotated MKL bottlenecks on memory at around four threads, Mozart scales to all the cores on the machine. In this benchmark, Mozart also *outperforms* the optimizing Weld compiler, which applies optimizations such as loop fusion to reduce data movement by keeping data in CPU registers. We found this was because Weld does not generate vectorized code for several operators that MKL does vectorize. Although it is feasible to

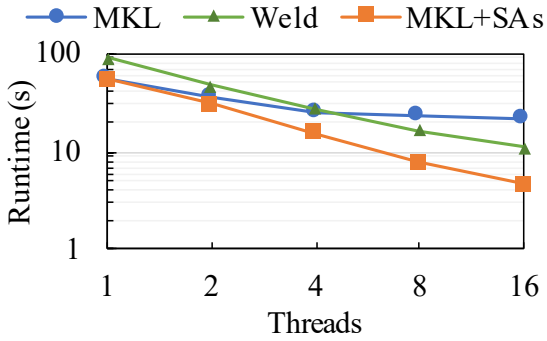


Figure 3.1: Performance of the Black Scholes benchmark on 1–16 threads with MKL, Weld, and MKL with Mozart.

extend Weld to include SIMD versions of these operators, this benchmark is one example of the advantages of leveraging code that developers have already hand-optimized.

To enable these performance improvements, Mozart first captures a dataflow graph of annotated functions called in the application (Figure 3.2). Since our goal is to leave libraries unmodified, we wish to capture such a graph without an explicit API that libraries implement. The `libmozart` *client library* (§3.4) transparently handles constructing such a graph and determining when to execute it, using a combination of auto-generated wrapper functions and memory protection in C/C++ and function decorators in Python. Mozart then runs a planning step to determine which functions to pipeline, and then executes tasks using the SAs and dataflow graph (§3.5).

3.2.2 Limitations and Non-Goals

SAs have a number of restrictions and non-goals. First, since SAs make repeated calls to black-box functions to achieve pipelining and parallelism, they are limited to functions that do not cause side effects or hold locks. Second, SAs do not apply the compute-based optimizations (e.g., common subexpression elimination or SIMD vectorization) that systems like Weld and TensorFlow XLA can provide by re-implementing functions in an IR. Nevertheless, because data movement has been shown to be one of the major bottlenecks in modern applications, we show that SAs can provide speedups competitive with these compilers without rewriting functions.

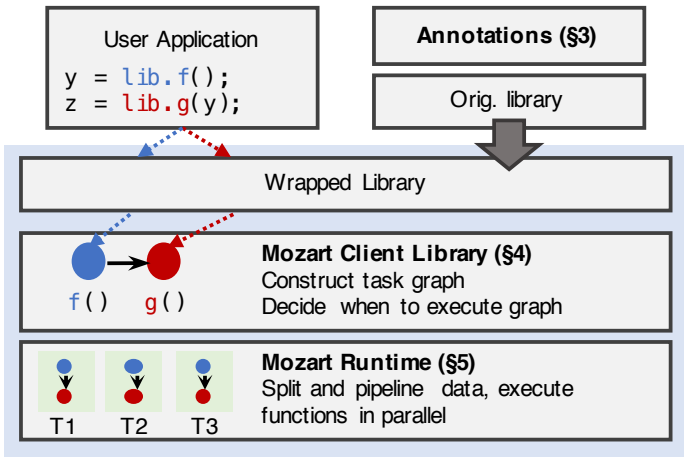


Figure 3.2: Overview of Mozart.

3.3 Split Annotation Interface

In this section, we present the split annotation (SA) interface. We first discuss the challenges in pipelining arbitrary functions to motivate our design. We then introduce *split types* and *split annotations*, the core abstractions used to address these challenges. We conclude this section by describing the *splitting API* that annotators implement to bridge these abstractions with code to split and merge data.

3.3.1 Why Split Types?

Split types are necessary because some data types in the user program can be split in multiple ways. As an example, matrix data can be split into collections of either rows or columns, or can even be partitioned into blocks. However, some functions require that data be split in a specific way to ensure correctness. As an example of this, consider the program below, which uses a function to first normalize the rows of a matrix (indicated via `axis=0`), followed by the columns of a matrix.

```
normalizeMatrixAxis(matrix, axis=0);
normalizeMatrixAxis(matrix, axis=1);
```

In the first call, the matrix must be split by rows, since the function requires access to

Listing 3.3 Full syntax of a split annotation.

```
@splittable([mut] <arg1-name>: [<arg1-split-type>|_], ...)
  [-> <ret-split-type>]
/* one or more functions */
```

all the data in a row when it is called. Similarly, the second call requires that the matrix be split into columns. Split types allow distinguishing these cases, even when they depend on runtime values (e.g., `axis`). Beyond ensuring correctness, split types also enable pipelining data split in the same way across functions.

3.3.2 Split Types and Split Annotations

A *split type* is a parameterized type $N\langle V_0 \ldots V_n \rangle$ defined by its name N and a set of parameter values $V_0 \ldots V_n$. A split type defines how a function argument is split. Two split types are equal if their names and parameters are equal. If the split types for two arguments are equal, it means that they are split in the same way, and their corresponding pieces can be passed into a function together. [1] Each split type is associated with a single concrete data type (e.g., an integer, array, etc.).

The library annotator decides what a "split" means for the data types in her library. As an example, an annotator for MKL's vector math library, which operates over C arrays, can choose to split arrays into multiple regularly-sized pieces. The annotator can then define a split type $ArraySplit\langle int, int \rangle$ that uniquely specifies how an array is split with two integer parameters: the length of the full array and the number of pieces the array is split into. An array with 10 elements can be split into two length-5 pieces with a split type $ArraySplit\langle 10, 2 \rangle$, or into five length-2 pieces with a split type $ArraySplit\langle 10, 5 \rangle$: these splits have *different* split types since their parameters are different, even if the underlying data refers to the same array. In the remainder of this section, we omit the parameter representing the number of pieces, because (*1*) every split type depends on it, and (*2*) Mozart sets it automatically (§3.5) and guarantees its equality for split types it compares.

A *split annotation (SA)* is then an annotation over a side-effect-free function that assigns

[1] Since type equality depends not only on the type name but also on the parameter values, split types are formally *dependent types* [59].

a name and a split type to each of the function's arguments and its return value. Listing 3.3 shows the full syntax of a single SA. For arguments that should not be split (and thus copied to each pipeline), annotators can give them a "missing" split type, denoted with "_" (e.g., "`arg: _`"). In addition to providing split types, the SA specifies which of the function arguments are *mutable* using the `mut` tag, which Mozart uses to detect data dependencies between functions when building a dataflow graph (§3.4). Listing 3.2 shows SAs for MKL array functions. Taking the `vdAdd` function as an example, the SA assigns the names `size`, `a`, `b`, and `out` to the arguments and assigns each a split type. The SA marks `out` as `mut` to indicate that the function mutates this argument.

Split Type Constructors. One subtlety in writing SAs arises due to the split types' parameters. Even though a split type's name is known when the annotator writes an SA, its parameters will generally not be known until *runtime*. For example, in the *ArraySplit* split type defined above, the length of an array will not be known until the program executes. This creates a challenge because Mozart needs to know the full split type including the concrete values of its parameters.

To address this, a split type can use function arguments to compute its parameters at runtime. Specifically, for an SA over a function F, each split type in the SA uses a *constructor* $A_0 \ldots A_n \Rightarrow V_0 \ldots V_n$ to construct its parameters, where $A_0 \ldots A_n$ are zero or more arguments of F. Within an SA, we use the syntax `Name(A0...An)` to refer to a constructor for a split type with a name `Name` that constructs its parameters with function arguments `A0...An`, where `A0...An` are names assigned to arguments in the SA. Note that the split type constructor is a part of the *splitting API* that annotators implement for each split type—we discuss this API further in §3.3.3. Unless otherwise noted, we assume here that split types use the identity function $A_0 \ldots A_n \Rightarrow A_0 \ldots A_n$ as their constructor.

As an example, consider Ex. 1 in Listing 3.4, which takes a matrix argument and an axis that determines whether the function operates over rows or columns (similar to the function in §3.3.1). We can represent splitting this matrix by either rows or columns by using a split type with three integer parameters called $MatrixSplit\langle int, int, int\rangle$. The parameters represent the matrix dimensions and the axis to iterate over. Within an SA, an annotator can write `MatrixSplit(m, axis)` to represent this split type: Listing 3.4 shows the constructor

definition for this split type, which maps the matrix and axis into the split type's three parameters. The split type for matrices does not depend on the matrix data itself, since the underlying data does not affect how the matrix is split. The SAs in Listing 3.2 similarly use the `size` argument (but not the array itself) to construct the *ArraySplit* split type.

With split types, Mozart can determine whether two functions can be pipelined safely. For each annotated function that Mozart captures in a dataflow graph, it initializes the parameters of the split types using the function's arguments. If all data passed between two functions have matching corresponding split types, they are pipelined. Otherwise, split data must be *merged* and re-split before passing it to the next function to prevent errors. Returning to the example from §3.3.1, we can use the split type from Ex. 1 in Listing 3.4 to assign the matrix arguments the split types $MatrixSplit\langle rows, cols, 0\rangle$ and $MatrixSplit\langle rows, cols, 1\rangle$: since these split types do not match, Mozart will not pipeline them.

Generics. SAs also support assigning generics to an argument. Generics in an SA are similar to generic types in languages such as Java or Rust: if two generics within an SA have the same name (e.g., `S`), the runtime ensures that the split types they are assigned are equal. Names for generics are local to an SA. Generics across SAs are propagated via *type inference* (§3.5), another common feature of existing type systems [6, 167].

Ex. 2 and 3 in Listing 3.4 show generics. Ex. 2 shows a function that adds two matrices element-wise: if `left` and `right` are split in the same way (indicated by their matching generic `S`), the function can process them together.

Unknown Split Type. Some functions will *change* the split type of a value in an unknown way upon execution. Ex. 4 in Listing 3.4 shows an example. The `filterZeroedRows` function changes the dimensions of its input, so its output split type is unknown after the call. We represent this in an SA using a special split type **`unknown`**, which represents a *unique* split type. Uniqueness prevents pipelining **`unknown`** values with any other split value: for example, if we tried to pass two **`unknown`** values to `add`, the types would not match, thus preventing pipelining. However, generic functions such as `scaleMatrix` (Ex. 3), which take a *single* argument split in any way, can still accept **`unknown`** values. This allows SAs to support operators such as filters in libraries such as Pandas.

Listing 3.4 Examples of SAs over matrices. Ex. 1 shows concrete split types, Ex. 2-3 show generics, Ex. 4 shows unknown split types, and Ex. 5 shows a reduction function.

```
// Parameters are (rows, cols, axis)
splittype MatrixSplit(int, int, int)
// Constructor for MatrixSplit
MatrixSplit(m, axis) => (m.rows, m.cols, axis)

// Ex. 1: Normalize along an axis in a matrix.
@splittable(mut m: MatrixSplit(m, axis), axis: _)
void normalizeMatrixAxis(matrix m, int axis);

// Ex. 2: Add two matrices element-wise.
@splittable(left: S, right: S) -> S
matrix add(matrix left, matrix right);

// Ex. 3: Scale a matrix element-wise.
@splittable(mut m: S, val: _)
void scaleMatrix(matrix m, double val);

// Ex. 4: Remove zero-valued rows from a matrix.
@splittable(m: S) -> unknown
matrix filterZeroedRows(matrix m);

// Ex. 5: Reduce a matrix to a vector by summing.
@splittable(m: MatrixSplit(m, axis), axis: _) -> ReduceSplit(axis)
vector sumReduceToVector(matrix m, int axis);
```

3.3.3 Splitting and Merging with the Splitting API

Annotators bridge the split type abstraction with an implementation using the splitting API. This API has several roles: it provides the constructor for constructing parameters, defines how to split data, and defines how to merge split pieces back into a full result. Table 3.1 summarizes these functions.

Constructor. The constructor maps values that will appear as function arguments to the split type's parameters. In our *ArraySplit* example, the constructor takes a `long` value representing an array's size and returns that size as its parameter. The constructor should not modify its arguments.

Splitting API Summary (§3.3.3)
`NameConstructor(A0,...An)=> Parameters`
`Split(D arg, int start, int end, Parameters)=> D`
`Merge(Vector<D>, Parameters)=> D`
`Info(D arg, Parameters)=> RuntimeInfo`

Table 3.1: The API annotators implement for a split type $Name\langle Parameters\rangle$. The argument has a data type `D`.

***Split* Function.** The split function performs the splitting operation using the original function argument and the split type parameters. It returns a split value representing the range $[start, end)$ in the function argument. Returning to the MKL *ArraySplit* example, the split function would return pointers offset from the base array pointer. Mozart dynamically selects the number of elements in $[start, end)$ and ensures that *end* does not surpass the total number of items in the argument. The `Info` function relays information to Mozart for this (§3.5). In our implementation, the split function also takes additional parameters such as a thread ID and the number of threads. This allows splits that are not based on integer ranges.

***Merge* Function.** The associative merge function coalesces split pieces into a single value once Mozart finishes processing them. In our MKL SAs, updates occur in-place, so no merge operation is needed, but if each function returned a new array instead, the merge function could concatenate the split arrays into a final result. This function can also be used to perform operations such as reductions in parallel using SAs.

Ex. 5 in Listing 3.4 shows an example of a reduction operator that collapses a matrix into a vector by summing either its rows or columns. The input matrix is split using *MatrixSplit* defined earlier, but the result is a new split type $ReduceSplit\langle axis\rangle$ that represents partial results. In particular, *ReduceSplit* represents either reduced row values or column values, depending on `axis`: its merge function uses `axis` to reconstruct either a row-vector or a column-vector.

3.3.4 Conditions to Use SAs

To summarize, a side-effect-free function $F(a,b,\ldots) \rightarrow c$ can be annotated with an SA with split types $(A,B,\ldots) \rightarrow C$ if:

$$F(a,b,\ldots) = Merge_C(F(a_1,b_1,\ldots),F(a_2,b_2,\ldots),\ldots)$$

where $Split_A(a) \rightarrow [a_1,a_2,\ldots]$ is the split function for split type A and $Merge_C(c_1,c_2,\ldots)$ is the associative merge function for split type C. There are no constraints on the number of splits each split function produces, as long as all split functions produce the same number of splits for a given function.

3.3.5 Summary: How Annotators Use SAs

To annotate a library, an annotator first decides how to split the library's core data types. She then defines split types and implements their splitting APIs. The annotator then writes SAs for side-effect-free functions using the defined split types. For functions that perform reductions or require custom merge operations, the annotator implements per-function split types that only implement the merge function. In our integrations, we required up to three split types for the core data types per library (e.g., DataFrames in Pandas), and one split type per reduction function. We generated most SAs using a script, since functions with matching function signatures can share the same SA. We discuss effort of integration in detail in §3.8.

3.3.6 Generality of SAs

A split type can capture a variety of data formats by splitting inputs and enabling pipelining and parallelization because annotators define their splitting behavior. We show in our library integrations in §3.7 and in our evaluation in §3.8 that SAs can split and optimize numerical and scientific workloads that use arrays, matrices, and tensors, SQL-like workloads that use DataFrames for operations ranging from projections and selections to groupBys and joins, image processing workloads, and natural language processing workloads. We note

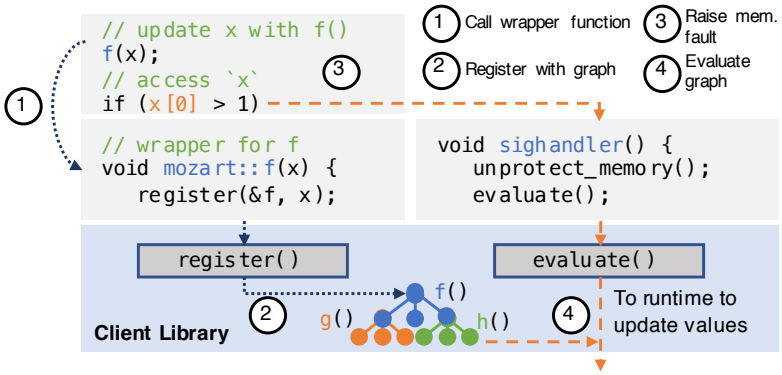


Figure 3.3: Overview of the C++ client library.

that, because SAs primarily aim to accelerate data parallel workloads that can be pipelined (i.e., cases where data movement optimizations in IRs are most applicable), they will be most impactful over collection-like data. For example, sparse data structures such as CSR matrices would only see a benefit from SAs if splitting their underlying dense representation allowed for greater temporal cache locality across operators.

3.4 The Mozart Client Libraries

Mozart relies on a lazily evaluated dataflow graph to enable cross-function data movement optimizations. Nodes in the dataflow graph represent calls to annotated functions and their arguments, and edges represent data passed between functions. Constructing such a graph without library modifications (e.g., by using an API as in prior work [156, 206]) is challenging because applications will contain a mix of annotated function calls and arbitrary code that may access lazy values. The `libmozart` client library is responsible for capturing a graph and determining when to evaluate it. The library has a small interface: *register(function, args)* registers a function and its arguments with the dataflow graph, and *evaluate()* evaluates the dataflow graph (§3.5 describes the runtime) when arbitrary code accesses lazy values. Since the `libmozart` design is coupled with the annotated library's language, we discuss its design in two languages: C++ and Python.

3.4.1 C++ Client Library

Our C++ client library uses code generation and OS-level memory protection to build a dataflow graph and to determine when to evaluate it. Figure 3.3 outlines its design.

Writing Annotations. An annotator registers split types, the splitting API, and SAs over C++ functions by using a command line tool we have built called `annotate`. This tool takes these definitions as input and generates namespaced C++ *wrapper functions* around each annotated library function. These wrapper functions are packaged with the splitting API and a lookup table that maps functions to their SAs in a shared library. The application writer links this wrapped library and calls the wrapper functions instead of the original library functions as always. This generally requires a namespace import and no other code changes–we note one exception below.

Capturing a Graph. The wrapper functions are responsible for registering tasks in the dataflow graph. When an application calls a wrapper, its function arguments are copied into a buffer, and the `libmozart` *register* API adds the function and its argument buffer as a node in the dataflow graph. The wrapper knows which function arguments are mutable, based on which arguments in the SA were marked `mut` (the SA is retrieved using the lookup table). This allows `libmozart` to add the correct data-dependency edges between calls.

Determining Evaluation Points. We evaluate the dataflow graph upon access to lazy values. There are two cases: (*1*) the accessed value was returned by an annotated function, and (*2*) the accessed value was allocated outside of the dataflow graph but mutated by an annotated function.

To handle the first case, if the library function returns a value, its wrapper instead returns a type called `Future<T>`. For types where `T` is a pointer, `Future<T>` is a pointer with an overloaded dereference operator that first calls *evaluate* to evaluate the dataflow graph. For non-pointer values, the value can be accessed explicitly with a `get()` method, which forces the execution. We also override the copy constructor of this type to track aliases, so copies of a lazy value can be updated upon evaluation. Wrapper functions can accept both `Future<T>`

values and `T` values, so `Future` values may be pipelined. Usage of `Future<T>` is the main code change applications must make when using SAs in C++.

We handle the second case (shown in Figure 3.3 in the access to `x[0]`) by using memory protection to intercept reads of lazy values. Applications must use our drop-in `malloc` and `free` functions for memory accessed in the dataflow graph (e.g., via `LD_PRELOAD`). Our version of `malloc` uses the `mmap` call to allocate memory with `PROT_NONE` permissions, which raises a protection violation when read or written. `libmozart` registers a signal handler to catch the violation, unprotects memory, and evaluates the dataflow graph registered so far. When calling a wrapper function for the first time after evaluation, `libmozart` re-protects all allocated memory to re-enable capturing memory accesses. This technique has been used in other systems successfully [107, 110] to inject laziness.

3.4.2 Python Client Library

Writing Annotations. Developers provide SAs by using Python function decorators. In Python, split types for positional arguments are required, and split types for keyword arguments default to "_" (but can be overridden).

Capturing a Graph. `libmozart` constructs the dataflow graph using the same function decorator used to provide the SA. The decorator wraps the original Python function into one that records the function with the graph using *register()*. The wrapper function then returns a placeholder `Future` object.

Determining Evaluation Points. Upon accessing a `Future` object, `libmozart` evaluates the task graph. In Python, we can detect when an object is accessed by overriding its builtin methods (e.g., `__getattribute__` to get object attributes). After executing the task graph, the `Future` object forwards calls to these methods to the evaluated cached value. To intercept accesses to variables that are mutated by annotated functions (based on **`mut`**), when `libmozart` registers a mutable object, it overrides the object's `__getattribute__` to again force evaluation when the object's fields are accessed. The original `__getattribute__` is reinstated upon evaluation.

3.5 The Mozart Runtime

Mozart is a parallel runtime that executes functions annotated with SAs. Mozart takes the dataflow graph generated by the client library and converts it into an *execution plan*. Specifically, Mozart converts the dataflow graph into a series of *stages*, where each stage splits its inputs, pipelines the split inputs through library functions, and then merges the outputs. The SAs dictate the stage boundaries. Mozart then executes each stage over *batches* in parallel, where each batch represent one set of split inputs.

3.5.1 Converting a Dataflow Graph to Stages

Recall that each node in the dataflow graph is an annotated function call, and each edge from function f_1 to f_2 represents a data dependency, i.e., a value mutated or returned by f_1 and read by f_2. Mozart converts this graph into stages. The functions f_1 and f_2 are in the same stage if, for every edge between them, the source value and destination value have the same split type. If *any* split types between f_1 and f_2 do not match, split data returned by f_1 must be merged, and a new stage starts with f_2. Mozart traverses the graph and checks types to construct stages.

To check split types between a source and destination argument, Mozart first checks that the split types have the same name. If the names are equal, Mozart uses the function arguments captured as part of the dataflow graph to initialize the split types' parameters. If the parameters also match, the source and destination have the same split type. If either split type is a generic, Mozart uses type inference [6, 167] to determine its type by pushing known types along the edges of the graph to set generics. If a split type cannot be inferred (e.g., because all functions use generics), Mozart falls back to a default for the data type: in our implementation, annotators provide a default split type constructor per data type.

This step produces an execution plan, where each stage contains an ordered list of functions to pipeline. The inputs to each stage are split based on the inputs' split types, and the outputs are merged before being passed to the next stage.

3.5.2 Execution Engine

After constructing stages, Mozart executes each stage in sequence by (*1*) choosing a batch size, (*2*) splitting and executing each function, and (*3*) merging partial results.

Step 1: Discovering Runtime Parameters. Mozart sets the number of elements in each batch and the number of elements processed per thread as runtime parameters. Since the goal of pipelining is to reduce data movement, we use a simple heuristic for the batch size: each batch should contain roughly *sizeof(L2 cache)* bytes. To determine the batch size, Mozart calls each input's `Info` function (§3.3), which fills a struct called `RuntimeInfo`. This struct specifies the number of total elements that will be produced for the input (e.g., number of elements in an array or number of rows in a matrix), and the size of each element in bytes. The batch size is then set to $\frac{C \times L2CacheSize}{\sum sizeof(element)}$ (where C is a fixed constant). We found that this value works well empirically (see §3.8) when pipelines allocate intermediate split values too, since these values are still small enough to fit in the larger shared last-level-cache.

Workers partition elements equally among themselves. The user configures the number of workers. Mozart checks to ensure that each split produces the same total number of elements. We opted for static parallelism rather than dynamic parallelism (e.g., via work-stealing) because it is simpler to schedule and we found that it leads to similar results for most workloads: however, dynamic work-stealing schedulers such as Cilk [25] are also compatible with Mozart.

Step 2: Executing Functions. After setting runtime parameters, Mozart spawns worker threads and communicates to each thread the range of elements it processes. Each worker allocates thread-local temporary buffers for the split values and enters the main driver loop. The worker's driver loop calls the *Split* function for each input and writes the result into the temporary buffers. If *Split* returns `NULL`, the driver loop exits. For arguments with the missing "_" split type, the original input value is copied (usually, this is just a pointer-copy) rather than split. The *start* and *end* arguments of the *Split* function are set based on the batch size and the thread's range.

To execute the function pipeline per thread, Mozart tracks which temporary buffers should be fed to each function as arguments by using a mapping from unique argument IDs

to buffers. The execution plan represents function calls using these argument IDs (e.g., a call $f_1(a0, a1, a2) \rightarrow a3$ will pass the buffers for $a0$, $a1$, and $a2$ as arguments and store the result in the buffer for $a3$). After each batch, these buffers are moved to a list of partial results, and Mozart starts the next batch.

Step 3: Merging Values. Once the driving loop exits, each worker merges each list of temporary buffers via the merge function (the stage tracks the split type of each result), and then returns the merged result. Once all workers return their partial results, Mozart calls the merge function again on the main thread to compute the final merged results.

3.6 Implementation

Our C++ version of `libmozart` and Mozart is implemented in roughly 3000 lines of Rust. This includes the parser for the SAs, the `annotate` tool for generating header files containing the wrapper functions, the client library (including memory protection), planner, and parallel runtime. We use Rust's threading library for parallelism. We make heavy use of the `unsafe` features of Rust to call into C/C++ functions. Memory allocated for splits is freed by the corresponding mergers. Mozart manages and frees temporary memory.

The Python implementation of these components is in around 1500 lines of Python. The SAs themselves use Python's function decorators, and split types are implemented as abstract classes with splitter and merger methods. We use process-based parallelism to circumvent Python's global interpreter lock. For native Python data, we only need to serialize data when communicating results back to the main thread. We leverage copy-on-write *fork()* semantics when starting workers, which also means that "_" values need not be cloned.

3.7 Library Integrations

We evaluate SAs by integrating them with five popular data processing libraries: NumPy [146], Pandas [157] spaCy [194], Intel MKL [132] and ImageMagick [87]. Table 3.3 in §3.8 summarizes effort, and we discuss integration details below.

NumPy. NumPy is a popular Python numerical processing library, with core operators implemented in C. The core data type in the library is the `ndarray`, which represents an N-dimensional tensor. We implemented a single split type for `ndarray`, whose splitting behavior depends on its shape and the axis a function iterates over (the split type's constructor maps `ndarray` arguments to its shape). We added SAs over all tensor unary, binary, and associative reduction operators. We implemented split types for each reduction operator to merge the partial results: these only required merge functions.

Pandas. Our Pandas integration implements split types over DataFrames and Series by splitting by row. We also added a *GroupSplit* split type for `GroupedDataFrame`, which is used for groupBy operations. Aggregation functions that accept this split type group chunks of a DataFrame, create partial aggregations, and then re-group and re-aggregate the partial aggregations in the merger. We only support commutative aggregation functions. We support most unary and binary Series operators, filters, predicate masks, and joins: joins split one table and broadcast the other. Filters and joins return the `unknown` split type, and most functions accept generics.

spaCy. SpaCy is a Python natural language processing library with operators in Cython. We added a split type that uses spaCy's builtin minibatch tokenizer to split a corpus of text. This allows any function (including user-defined ones) that accepts text and internally uses spaCy functions to be parallelized and pipelined via a Python function decorator.

Intel MKL. Intel MKL [132] is an optimized closed-source numerical computation library used as the basis for other computational frameworks [2, 76, 110, 144, 146, 173]. To integrate SAs, we defined three split types: one for matrices (with rows, columns, and order as parameters), one for arrays (with length as a parameter), and one for the size argument. Since MKL operates over inputs in place, we did not need to implement merger functions. We annotated all functions in the vector math header, the saxpy (L1 BLAS) header, and the matrix-vector (L2 BLAS) headers.

ImageMagick. ImageMagick [87] is a C image processing library that contains an API where images are loaded and processed using an opaque handle called `MagickWand`. We implemented a split type for the `MagickWand` type, where the split function uses a crop function to clone and return a subset of the original image. ImageMagick also contains an API for appending several images together by stacking them—our split function thus returns entire rows of an image, and this API is used in the merger to reconstruct the final result.

3.7.1 Experiences with Integration

Unsupported Functions. We found that there were a handful of functions in each library that we could not annotate. For example, in the ImageMagick library, the `Blur` function contains a boundary condition where the edges of an image are processed differently from the rest of the image. SAs' split/merge paradigm would produce incorrect results here, because this special handling would occur on each split rather than on just the edges of the full image. Currently, annotators must manually determine whether each function is safe to annotate (similar to other annotation-based systems such as OpenMP). We found that this was straightforward in most cases by reading the function documentation, but tools that could formally prove an SA's compatibility with a function would be helpful. We leave this to future work. We did not find any functions that internally held locks or were not callable from multiple threads in the libraries we annotated.

Debugging and Testing. Since annotators must manually enforce the soundness of an SA, we built some mechanisms to aid in debugging and testing them. The `annotate` tool, for example, will ensure that a split type is always associated with the same concrete type. The runtimes for both C and Python include a "pedantic mode" for debugging that can be configured to panic if a function receives splits with differing numbers of elements, receives no elements, or receives `NULL` data. The runtime can also be configured to log each function call on each split piece, and standard tools such as Valgrind or GDB are still available. Anecdotally, the logs and pedantic mode made debugging invalid split/merge functions and errors in SAs unchallenging. We also fuzz tested our annotated functions to stress their correctness.

3.8 Evaluation

In evaluating split annotations, we seek to answer several questions: (*1*) Can SAs accelerate end-to-end workloads that use existing unmodified libraries? (*2*) Can SAs match or outperform compiler-based techniques that optimize and JIT machine code? (*3*) Where do performance benefits come from, and which classes of workloads do SAs help the most?

Workload	Libraries	Description (# Operators)
Black Scholes	NumPy, MKL	Computes the Black Scholes [22] formula over a set of vectors. (32)
Haversine	NumPy, MKL	Computes Haversine Dist. [81] from a set of GPS coordinates to a fixed point. (18)
nBody	NumPy, MKL	Uses Newtonian force equations to determine the position/velocity of stars over time. (38)
Shallow Water	NumPy, MKL	Estimates the partial differential equations used to model the flow of a disturbed fluid [191]. (32)
Data Cleaning [57]	Pandas	Cleans a DataFrame of 311 requests [1] by replacing `NULL`, broken, or missing values with `NaN`. (8)
Crime Index	Pandas, NumPy	Computes an average "crime index" score, given per-city population and crime information. (16)
Birth Analysis [120]	Pandas, NumPy	Given a dataset of number of births by name/year, computes fraction of names starting with "Lesl" grouped by gender and year-of-birth. (12)
MovieLens	Pandas, NumPy	Joins tables from the MovieLens dataset [80] to find movies that are most divisive by gender. (18)
Speech Tag	spaCy	Tags parts of speech and extracts features from a corpus of text. (8)
Nashville	ImageMagick	Image pipeline [90] that applies color masks, gamma correction, and HSV modulation. (31)
Gotham	ImageMagick	Image pipeline [89] that applies color masks, saturation/contrast adjustment, and modulation. (15)

Table 3.2: Workloads used in our evaluation. Descriptions for workloads from Weld are taken from [154]. The number in parentheses shows the number of library API calls.

Experimental Setup. We evaluate workloads that use the five libraries in §3.7: NumPy v1.16.2, Pandas v0.22.0, spaCy v2.1.3, Intel MKL 2018 and ImageMagick v7.0.8. We ran all experiments on an Amazon EC2 `m4.10xlarge` instance with Intel Xeon E5-2676 v3 CPUs (40 logical cores) and 160GB of RAM, running Ubuntu 18.04 (Linux 4.4.0). Unless otherwise noted, results average over five runs.

3.8.1 Workloads

We evaluate the end-to-end performance benefits of SAs using Mozart on a suite of 15 data analytics workloads (of which four are repeated in NumPy and Intel MKL). Eight of the benchmarks are repeated from the Weld evaluation (§2.9), which obtained them from popular GitHub repositories, Kaggle competitions, and online tutorials. We also evaluate an additional numerical analysis workload (Shallow Water) over matrix operations, taken from the Bohrium paper [110] (Bohrium is an optimizing NumPy compiler that we compare against here). Finally, we evaluate two open-source image processing workloads [88] that use ImageMagick, and a part-of-speech tagging workload [194] that uses spaCy.

3.8.2 End-to-end Performance Results

Figure 3.4 shows Mozart's end-to-end performance on our 15 benchmarks on 1–16 threads. Each benchmark compares Mozart vs. a base system without SAs (e.g., NumPy or MKL without SAs). We also compare against optimizing compiler-based approaches that enable parallelism and data movement optimization without changing the library's API, but require *re-implementing functions* under-the-hood.

Summary of all Results. On 16 threads, Mozart provides speedups of up to 14.9× over libraries that are single-threaded, and speedups of up to 4.7× over libraries that are already parallelized due to its data movement optimizations. Across the 15 workloads, Mozart is within 1.2× of all compilers we tested in six workloads, and outperforms all the compilers we tested in four workloads. Compilers outperform Mozart by over 1.5× in two workloads. Compilers have the greatest edge over Mozart in workloads that contain many operators

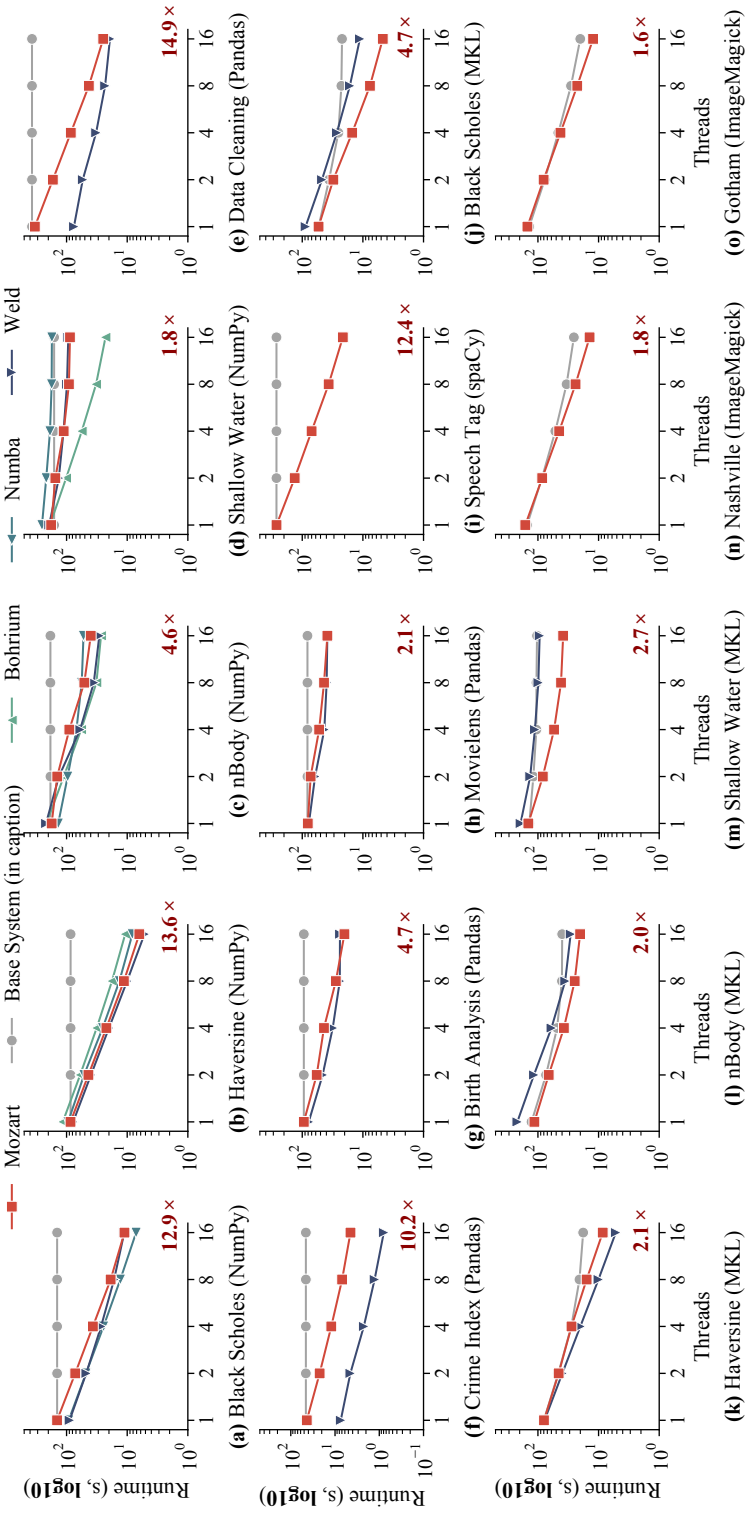


Figure 3.4: End-to-end performance on 15 benchmarks compared against a base system (in caption, e.g., NumPy) and several optimizing compilers that require rewriting libraries. We show results on 1–16 threads. Each plot displays the speedup (in red) that Mozart enables on 16 threads over the base system.

implemented in interpreted Python, since they naturally benefit more from compilation to native code than they do from data movement optimizations. We discuss workloads below.

NumPy Numerical Analysis (Figures 3.4a-d). We evaluate the NumPy workloads against un-annotated NumPy and three Python JIT compilers: Bohrium [110], Numba [144], and Weld. Each compiler requires function rewrites under-the-hood. Figures 3.4a-b show the performance of Black Scholes and Haversine, which apply vector math operators on NumPy arrays. All systems enable near-linear scalability since all operators can be pipelined and parallelized. Overall, Mozart enables speedups of up to 13.6×.

The nBody and Shallow Water workloads operate over tensors and matrices, and contain operators that cannot be pipelined. For example, Shallow Water performs several row-wise matrix operations and then aggregates along columns to compute partial derivatives. Mozart captures these boundaries using split types and still pipelines the other operators in these workloads. Figures 3.4c-d show that Mozart enables up to 4.6× speedups on 16 threads. Bohrium outperforms other systems in the Shallow Water benchmark because it captures indexing operations that Mozart cannot split and that the other compilers could not parallelize (Bohrium converts the indexing operation into its IR, whereas Mozart treats it as a function call over a single element that cannot be split).

Data Science with Pandas and NumPy (Figures 3.4e-h). We compare the Pandas workloads against Weld: the other compilers did not accelerate these workloads. The Data Cleaning and Crime Index workloads use Pandas and NumPy to filter and aggregate values in a Pandas DataFrame. Figures 3.4e-f show the results. Mozart parallelizes and pipelines both of these workloads and achieves an up to 14.9× speedup over the native libraries. However, Weld outperforms Mozart by up to 5.85× even on 16 threads, because both contain operators that use interpreted Python code which Weld compiles.

Figures 3.4g-h shows the results for the Birth Analysis and MovieLens workloads. These workloads are primarily bottlenecked on grouping and joining operations implemented in C. In Birth Analysis, Mozart accelerates groupBy aggregations by splitting grouped DataFrames and parallelizing (there are no pipelined operators), leading to a 4.7× speedup. In MovieLens, we pipeline and parallelize two joins and parallelize a grouping

aggregation, leading to a 2.1× speedup. In both, Mozart outperforms Weld. Weld's parallel grouping implementation bottlenecked on memory allocations around 8 threads in Birth Analysis. In MovieLens, speedups were hindered due to serialization overhead (Weld marshals strings before processing them, and Mozart sends large join results via IPC), but the Weld serialization could not be parallelized.

Speech Tagging with spaCy (Figures 3.4i). Figure 3.4i shows the performance of the speech tagging workload with and without Mozart. This workload operates over a corpus of text from the IMDb sentiment dataset [124]. It tags each word with a part of speech and normalizes sentences using a preloaded model. Mozart enables 12× speedups via parallelization. Unfortunately, no compilers supported spaCy.

Numerical Workloads with MKL (Figures 3.4j-m). We evaluate Mozart with MKL using the same numerical workloads from NumPy. Unlike NumPy, MKL already parallelizes its operators, so the speedups over it come from optimizing data movement. Figures 3.4j-m show that Mozart improves performance by up to 4.7× on 16 threads, even though MKL also parallelizes functions. Mozart outperforms Weld on three workloads here, because MKL vectorizes and loop-blocks matrix operators in cases where Weld's compiler does not.

Image workloads in ImageMagick (Figures 3.4n-o). Figure 3.4 shows the results on our two ImageMagick workloads. Like MKL, ImageMagick also already parallelizes functions, but Mozart accelerates them by pipelining across operators. Mozart outperforms base ImageMagick by up to 1.8×. End-to-end speedups were limited despite pipelining because splits and merges allocate and copy memory excessively. Mozart sped up just the computation by up to 3.4× on 16 threads.

3.8.3 Effort of Integration

In contrast to compilers, Mozart only requires annotators to add SAs and implement the splitting API to achieve performance gains. To quantify this effort, we compared the lines of code required to support our benchmarks with Mozart vs. Weld. Table 3.3 shows the results. Our Weld results only count code for operators that we also support, and only counts

		LoC for SAs			LoC for Weld		
Library	#Funcs	SAs	Split. API	Total	Weld IR	Glue	Total
NumPy	84	47	37	**84**	321	73	**394**
Pandas	15	72	49	**121**	1663	413	**2076**
spaCy	3	8	12	**20**			
MKL	81	74	90	**155**			
ImageMagick	15	49	63	**112**			

Table 3.3: Integration effort for using Mozart. Numbers show the total lines of code per library. Mozart requires up to 15× fewer LoC to support the same operators as Weld.

integration code. We do not count code for the Weld compiler itself (which itself is over 25K LoC and implements a full compilation backend based on LLVM). Similarly, we only count code that an annotator adds for Mozart, and do not count the runtime code.

Overall, for our benchmarks, SAs required up to 17× less code to enable similar performance to Weld in many cases. Weld required at least tens of lines of IR code per operator, a C++ extension to marshal Python data, and usage of its runtime API to build a dataflow graph. Anecdotally, with Weld our experience was that just supporting the Birth Analysis workload—even after having implementing the core integration—was a multi-week effort that required several extensions to the Weld compiler itself and also required extensive low-level performance tuning. In contrast, we integrated SAs with the same Pandas functions in roughly half a day, and the splitting API was implemented using existing Pandas functions with fewer than 20 LoC each.

3.8.4 Importance of Pipelining

The main optimization Mozart applies beyond parallelizing split data is pipelining it across functions to reduce data movement. To show its importance, Table 3.4 compares three versions of the Black Scholes and Haversine workloads on 16 threads: un-annotated MKL, Mozart with pipelining, and Mozart without pipelining (i.e. Mozart parallelizes functions on behalf of MKL). We also used the Linux `perf` utility to sample the hardware performance counters for the three variations of the workload. Mozart without pipelining does not result in a speedup over parallel MKL. In addition, the most notable difference with pipelining is

	Black Scholes			Haversine		
	MKL	Mozart (-pipe)	**Mozart**	MKL	Mozart (-pipe)	**Mozart**
Normalized Runtime	1.00	1.01	**0.21**	1.00	0.97	**0.48**
LLC Miss (avg/stddev)	39.86% (2.26%)	42.96% (1.56%)	**22.12% (1.99%)**	50.44% (7.99%)	57.47% (0.40%)	**41.18% (3.67%)**
Inst/Cycle (avg/stddev)	0.536 (0.06)	0.511 (0.04)	**1.221 (0.10)**	0.892 (0.08)	1.362 (0.22)	**1.65 (0.33)**

Table 3.4: Hardware counters show that pipelining reduces cache misses, which translates to higher performance.

in the last level cache (LLC) miss rate: the miss rate decreases by a factor of 2×, confirming that pipelining does indeed generate less traffic to main memory and reduce data movement. This in turn leads to better overall performance on multiple threads. We found no other notable differences in the other reported counters. We saw similar results for the other MKL workloads as well.

3.8.5 System Overheads

To measure the system overheads that Mozart imposes, Figure 3.5 shows the breakdown in running time for the Black Scholes and Nashville workloads on 16 threads. We report the client library time for registering tasks, the memory protection time for unprotecting pages during execution, planning, splitting, task execution, and merging. The Nashville workload had the highest relative split and merge times, since both the splitters and mergers allocate memory and copy data. Across all workloads, the execution time dominates the total running time, and the client library, memory protection, and planner account for less than 0.5% of the running time.

Most of the overhead is attributed to handling memory protection. In our setup, unprotecting each gigabyte of data took roughly 3.5ms, indicating that this overhead could be significant on task graphs that perform little computation. One mechanism for reducing the overhead of memory protection is the recent `pkeys` [169] set of system calls, which allows

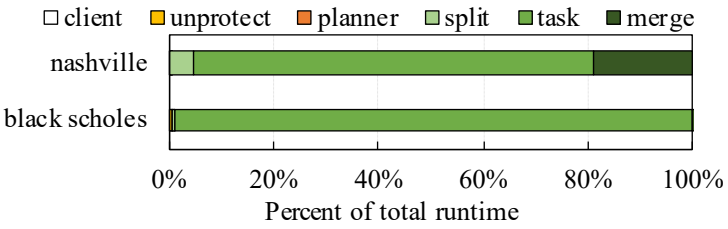
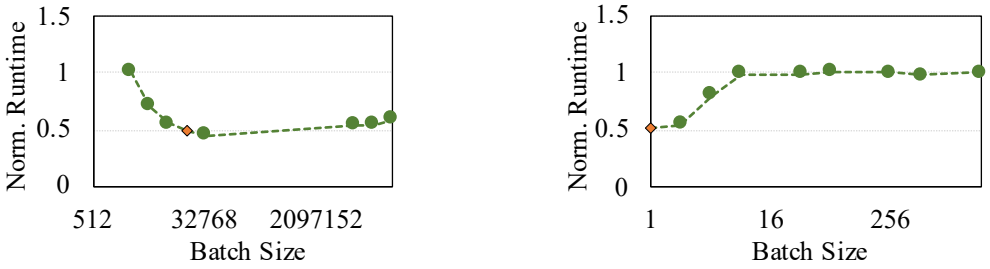


Figure 3.5: Breakdown of total running time in the Nashville and Black Scholes workloads. Across all workloads, we observed 0.5% overhead from other components.



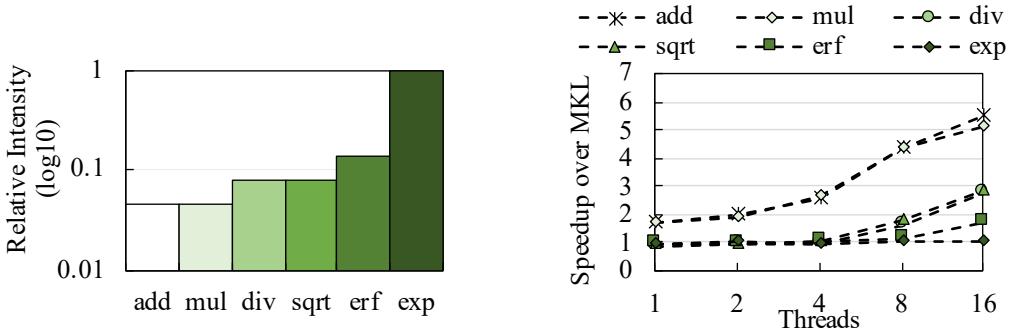
(a) Black Scholes. **(b)** nBody.

Figure 3.6: Effect of batch size on two workloads. Mozart selects a batch size near the optimal using L2 cache size.

for O(1) memory permission changes by associating pages with a registered protection key. After tagging memory pages with a key, the cost of changing their permissions is a register write, so the time to unprotect or protect *all* memory allocated with Mozart becomes negligible (tens of microseconds to unprotect 1GB in a microbenchmark we ran).

3.8.6 Effect of Batch Size

We evaluate the effect of batch size by varying it on the Black Scholes and nBody workloads and measuring end-to-end performance for each parameter. We benchmarked these two workloads because they contain "elements" of different sizes: Black Scholes treats each `double` as a single element, while the matrices' split types in nBody treat rows of a matrix (256KB in size each) as a single element. Figure 3.6 shows the results. The marked point shows the batch size selected by Mozart using the strategy described in §3.5. The plots show that batch size can have a significant impact on overall running time (too low imposes too



(a) Relative Intensity **(b)** Speedup over no SAs

Figure 3.7: Impact of compute-intensiveness in Mozart.

much overhead, and too high obviates the benefits of pipelining), and that Mozart's heuristic scheme selects a reasonable batch size. Across all the workloads we benchmarked, Mozart chooses a batch size within 10% of the best batch size we observed in a parameter sweep.

3.8.7 Compute- vs. Memory-Boundedness

To study when Mozart's data movement optimizations are most impactful, we measured the intensity (defined as *cycles spent per byte of data*) of several MKL vector math functions by calling them in a tight loop on an array that fits entirely in the L2 cache. We benchmarked the following operations, in order of increasing intensity: add, mul, sqrt, div, erf, and exp. Figure 3.7a shows the relative intensities of each function (i.e., `vdExp` spends roughly 7× more cycles per byte of data than `vdErf`). We then ran each math function 10 times on a large 8GB array, with and without Mozart. Figure 3.7b shows the *speedup* of Mozart over un-annotated MKL on 1–16 threads. Mozart has the largest impact on memory-intensive workloads that spend few cycles per byte, and shows increasing speedups as increasing amounts of parallelism starve the available memory bandwidth.

3.9 Extension: Offload Annotations for Accelerators

The main idea in split annotations is to expose properties of existing functions to enable pipelining and data movement optimizations. However, an annotation-based system could

Listing 3.5 Example of offload annotations for two NumPy functions. PyTorch is used as the offload library. We show syntax for Python here.

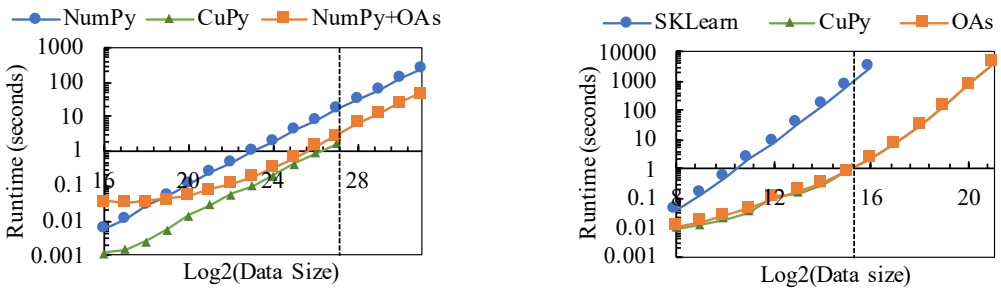
```
# Split types contain information for how to map np.ndarray to pytorch.tensor
@offload(left: ArraySplit(size), right: ArraySplit(size),
  return: ArraySplit(size), gpufunc=pytorch.add)
np.add(left: np.ndarray, right: np.ndarray) -> np.ndarray

@offload_alloc(size: SizeSplit(size), return=ArraySplit(size),
  gpufunc=pytorch.empty)
np.empty(size: int) -> np.ndarray
```

also be used to optimize existing applications in other ways. We propose one extension to split annotations, called *offload annotations* [223], that allow inter-operability of existing CPU libraries and emerging GPU libraries.

Accelerator libraries targeting platforms such as GPUs have grown increasingly popular. These libraries also aim to provide APIs that mirror those of existing CPU libraries (e.g., NumPy and PyTorch [173] have very similar interfaces). However, these libraries often support only a subset of the functionality of their CPU counterparts, sometimes for fundamental reasons. For example, a particular function in a CPU library may not be amenable to parallelization on a GPU. This requires mixing CPU and GPU libraries, but causes expensive data movement between the CPU and the accelerator. In addition, the relatively small amount of memory available GPU means that memory must be carefully managed when processing CPU-resident datasets on the GPU.

Offload annotations (OAs), are an extension of split annotations for heterogeneous computing. They allow third-party developers to incrementally add hardware-accelerated versions of library functions to existing libraries, and an underlying runtime (and extension of Mozart) manages the offload and execution of these functions across devices. OAs extend SAs with support for representing data in different formats on different devices as part of the split type interface, and deciding when to offload a computation based on its estimated transfer size and computation cost. OAs also distinguish functions that allocate memory and perform computation, to further optimization device placement (e.g., so the allocation functions can execute directly on the device where computation will occur).



(a) Black Scholes (NumPy + CuPy) **(b)** DBSCAN (Scikit-Learn + CuPy)

Figure 3.8: Offload annotations allow applications expressed with CPU libraries to offload onto the GPU with no code changes. The dotted line shows where the GPU library implementation runs out of memory: in contrast, OAs split and pipeline data automatically through the GPU and can scale to datasets larger than the GPU memory.

Listing 3.5 shows an example of two offload annotations for NumPy, annotated to automatically offload to PyTorch. The annotation for `np.add` resembles an SA, but additionally specifies an equivalent GPU function (in this case, `pytorch.add`. In addition, the split type API for `ArraySplit` contains two additional functions `from_device()` and `to_device()` to transfer data between the CPU and GPU. The annotation for `np.empty` (which allocates an empty array) is marked with a special `oa_alloc` annotation that indicates allocation: this allows the OA runtime to co-locate allocations on the device where the allocated data will be processed.

By extending split types with device placement information and heuristics on cross-device data transfer costs, OAs can enable developers that use familiar CPU libraries such as NumPy to leverage GPUs without any application code changes. In addition, because OAs still split data before sending it to an accelerator, OAs can enable workloads whose working sets are larger than the total available GPU memory to execute on the GPU without code changes as well: without OAs, developers would manually need to manage low level concepts such as GPU streams.

We show two results from our OA work [223] in Figure 3.8. Figure 3.8a shows an implementation of the Black Scholes workload using NumPy, with NumPy functions annotated with OAs using CuPy [50] for the GPU. CuPy is a Python wrapper around optimized CUDA kernels. Figure 3.8b shows an implementation of the DBSCAN [51]

clustering algorithm implemented using scikit-learn [193], also annotated using CuPy.

The results have two main takeaways. First, OAs allow existing applications built on top of CPU libraries to target GPUs with no code changes, and thus can enable these applications to leverage the performance improvements of using the GPU. The DBSCAN workload is 1200× faster on the GPU, while Black Scholes is 5.5× faster. OAs match the performance of a pure CuPy implementation in the DBSCAN workload and come within 2× in the Black Scholes workload.

The second takeaway is that OAs enable existing applications whose working sets exceed the GPU memory (which can often be more than an order of magnitude smaller than CPU memory) to leverage the GPU by splitting and pipelining computations. Both the Black Scholes and DBSCAN results exhibit this: the CuPy implementations run out of memory when the input dataset size reaches the dotted line, while the OA implementation can continue scaling by partitioning data and only processing chunks on the GPU. We thus believe that OAs are a promising technique to bridge existing CPU libraries and emerging GPU libraries.

3.10 Related Work

SAs are influenced by work on building new common runtimes or IRs for data analytics [117, 155,201,222] and machine learning [2,39,200]. Weld [155] and Delite [201] are two specific examples of systems that use a common IR to detect *parallel patterns* and automatically generate parallel code. Although Mozart does not generate code, we show in §3.8 that in a parallel setting, the most impactful optimizations are the data movement ones, so SAs can achieve competitive performance without requiring developers to replace code. API-compatible replacements for existing libraries such as Bohrium [110] also have completely re-engineered backends.

Several existing works provide black-box optimizations and automatic parallelization of functions. Numba [144] JITs code using a single decorator, while Pydron [138], Dask [55] and Ray [142] automatically parallelize Python code for multi-cores and clusters. In C, frameworks such as Cilk [25] and OpenMP [54] parallelize loops using an annotation-style interface: many such systems use compiler `#pragma` directives to parallelize code or provide

hints to allow a compiler to generate faster code. Unlike these systems, in addition to parallelization, SAs enable data movement optimizations across functions and reason about pipelining safety. Other `pragma` directives can also reason about loop fission or fusion [170]. However, these directives require the loops to be in a single compilation unit, while SAs work on existing arbitrary pre-compiled libraries under composition.

The optimizations that SAs enable have been studied before: Vectorwise [226] and other vectorized databases [26, 46, 113] apply the same pipelining and parallelization techniques as SAs for improved cache locality. Unlike these databases, Mozart applies these techniques on a diverse set of black-box libraries and also reason about the *safety* of pipelining different functions using split types. SAs are also influenced by prior work in the programming languages community on automatic loop tiling [74], pipelining [37, 73], and link-time optimization [20, 64], though we found these optimizations most effective over nested C loops in user code, and not over compositions of complex arbitrary functions.

Finally, split types are conceptually related to Spark's partitioners [195] and Scala's parallel collections API [171]. Scala's parallel collections API in particular features a `Splitter` and `Combiner` that partition and aggregate a data type, respectively. Unlike this API, SAs enable pipelining and also reason about the safety of pipelining black-box functions: Scala's collections API still requires introspecting collection implementations. Spark's Partitioners similarly do not enable pipelining.

3.11 Summary

Data movement is a significant bottleneck for data-intensive applications that compose functions from existing libraries. Although researchers have developed compilers and runtimes that apply data movement optimizations on existing workflows, they often require intrusive changes to the libraries themselves. We introduced a new black-box approach called split annotations (SAs), which specify how to safely split data and *pipeline* it through a parallel computation to reduce data movement. We showed that SAs require no changes to existing functions, are easy to integrate, and provide performance competitive with clean-slate approaches in many cases.

Chapter 4

Raw Filtering

4.1 Introduction

Thus far, this dissertation has studied new software composition interfaces in the context of combining data-intensive libraries and applying optimizations for reducing *data movement* between them. However, the claim that we can optimize programs by exposing algebraic properties in composition interfaces has applications across the software stack.

In this chapter, we turn our attention to a new domain: big data frameworks that process serialized data at rest (e.g., on a disk). Many analytics workloads process data stored in unstructured or semi-structured formats, including JSON, XML, or binary formats such as Avro and Parquet [14, 160]. Rather than loading datasets in these formats into a DBMS, researchers have proposed techniques [7, 101–103, 118, 150] for executing queries in situ over the raw data directly.

Unfortunately, a key bottleneck in querying raw data is parsing the data itself. Parsers—especially for human-readable formats such as JSON—are typically expensive because they rely on state-machine-based algorithms that execute a series of instructions per byte of input [36, 178]. In contrast, modern CPUs are optimized for operations on multiple bytes in parallel (e.g., SIMD). In response, researchers have recently developed new parsing methods that utilize modern hardware more effectively [118, 137]. One such example is the Mison JSON parser [118], which uses SIMD instructions to find special characters such as brackets and colons to build a *structural index* over a raw JSON string, enabling efficient field

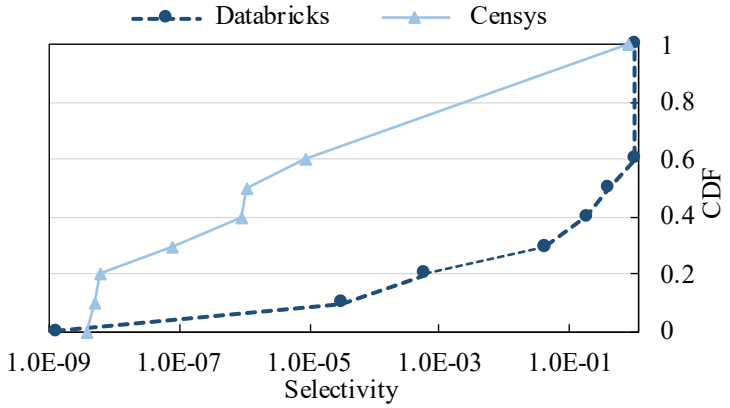


Figure 4.1: CDF of selectivities from (*1*) Spark SQL queries on Databricks that read JSON or CSV data, and (*2*) researchers' queries over JSON data on the Censys search engine [63]. Both sets of queries are highly selective.

projection without deserializing the record completely. This approach delivers substantial speedups: we found that Mison can parse highly nested in-memory data at over 2GB/s per core, over 5× faster than RapidJSON [178], the fastest traditional state-machine-based parser available [98]. Even with these new techniques, however, we still observe a large memory-compute performance gap: a single core can scan a raw bytestream of JSON data 10× faster than Mison parses it. Perhaps surprisingly, similar gaps can even occur when parsing binary formats that require byte-level processing, such as Avro and Parquet.

One fundamental reason for why querying raw data formats is slow is the *interface* that parsing libraries expose to applications. All modern parsers take as input a raw bytestring and output a structured object (e.g., a DOM tree or a C-style struct with fields at fixed-length offsets). Downstream query processing occurs over these objects rather than on the bytestring directly. However, in many cases, parsing every field in each serialized record can be wasteful, because the query may not access each field or may discard the record entirely due to a filter.

In this work, we accelerate raw data processing by exploiting a key property of many exploratory analytics workloads: high selectivity. Figure 4.1 illustrates this, showing query selectivity data collected in 2018 from two cloud services: profiling metadata about Spark SQL queries on Databricks' cloud service [56] that read JSON or CSV data, and researchers' queries over Censys [63], a public search engine of Internet port scan data broadly used in

the security community. 40% of the Spark queries select less than 20% of records, while the median Censys query selects only 0.001% of the records for further processing.

We propose *raw filtering*, a new approach that leverages modern hardware and high query selectivity to filter data *before* parsing it. Raw filtering uses a set of filtering primitives called *raw filters* (RFs), which are operators derived from a query predicate that filter records by inspecting a raw bytestream of data, such as UTF-8 strings for JSON or encoded binary buffers for Avro or Parquet. Rather than parsing records and evaluating query predicates exactly, RFs filter records by evaluating a format-agnostic filtering function over raw bytes with some false positives, but no false negatives. Raw filtering thus proposes using a new interface between data parsers and query engines: rather than treating the two as separate components, raw filtering allows a query specification to be pushed down into the parser to enable new optimizations.

To decrease the false positive rate, we can use an optimizer to compose multiple RFs into an *RF cascade* that incrementally filters data. A full format-specific parser (e.g., Mison) then parses and verifies any remaining records. Raw filtering is thus complementary to existing work on fast projection [33, 60, 61, 82, 118]. We show that raw filtering can accelerate state-of-the-art parsers by up to 22× on selective queries.

Because RFs can produce false positives and still require running a full parser on the records that pass them, two key challenges arise in utilizing raw filtering efficiently. First, the RFs themselves have to be highly efficient, allowing us to run them without impacting overall parsing time. Second, the RF cascade optimizer must quickly find efficient cascades, which is challenging because the space of possible cascades is combinatorial, the passthrough rates of different RFs are not independent, and the optimizer itself must not add high overhead. We discuss how we tackle these two challenges in turn.

Challenge 1: Designing Efficient Raw Filters. The first challenge is ensuring that RFs are hardware-efficient. Since RFs produce false positives, an inefficient design for these operators could increase total query execution time by adding the overhead of applying RFs without discarding many records. To address this challenge, we propose a set of SIMD-enabled RFs that process multiple bytes of input data per instruction. For example, the *substring search* RF searches for a byte sequence in raw data that indicates whether a record

could pass a predicate. Consider evaluating the predicate `name = "Albert Einstein"` over JSON data. The substring RF could search over the raw data for the substring `"Albe"`, which fits in an AVX2 SIMD vector lane and allows searching 32 bytes in parallel. This is a valid RF that only produces false positives, because the string `"Albe"` must appear in any JSON record that satisfies the predicate. Without fully parsing the record, however, the RF may find cases where the substring comes from a different string (e.g., `"Albert Camus"`) or from the wrong field (e.g., `friend = "Albert Einstein"`). Likewise, a *key-value search* RF extends substring search to look for key-value pairs in the raw data (e.g., a JSON key and its value). While designing for hardware efficiency imposes some limitations on the predicates we can convert to RFs, our RFs can be applied to many queries in diverse workloads, and can stream through data 100× faster than existing parsers.

Challenge 2: Choosing an RF Cascade. The second challenge is selecting the RF cascade that produces the highest expected parsing throughput, i.e., determining which RFs to include, how many to include, and what order to apply them in. The performance of each RF cascade depends on its execution cost, its false positive rate, and the execution cost of running the full parser. Unfortunately, determining these metrics is difficult because they are data-dependent and the passthrough rates of individual RFs in the cascade can be highly correlated with one another. For example, a cascade with a single substring RF `"Albe"` may not benefit from an additional substring RF `"Eins"`: whatever records match `"Albe"` are likely to match `"Eins"` as well. In our evaluation, we show that modeling this interdependence between RFs is critical for performance, and can make a 2.5× difference compared to classical methods for predicate ordering that assume independence [18].

To address this challenge, we propose a fast optimizer that uses SIMD to efficiently select a cascade while also accounting for RF interdependence. Our optimizer periodically takes a sample of the data stream and estimates the individual passthrough rates and execution costs of the valid RFs for the query on it. It stores the result of each RF on the sample records in a bitmap that allows us to rapidly compute the passthrough rate for any cascade of RFs using SIMD bitwise operators. With this approach, the optimizer can efficiently search through a large space of cascades and pick the one with the best expected throughput. We show that choosing the right cascade can make a 10× difference in performance, and

that our optimizer only adds 1.2% overhead. We also show that updating the RF cascade periodically while processing the input data can make a 25× difference in execution time due to changing data properties.

Summary of Results. We evaluate raw filtering in a system called Sparser, which implements the SIMD RFs and optimizer described above. Sparser takes a user query predicate and a raw bytestream as input and returns a filtered bytestream to a downstream query engine. Our evaluation shows that Sparser outperforms standalone state-of-the-art parsers and accelerates real workloads when integrated in existing query processing engines such as Spark SQL [11]. On exploratory analytics queries over Twitter data from [118, 203], Sparser improves Mison's JSON parsing throughput up to 22×. When integrated into Spark SQL, Sparser accelerates distributed queries on a cluster by up to 9× end-to-end, including the time to load data from disk. Perhaps surprisingly, Sparser can even accelerate queries over binary formats such as Avro and Parquet by 1.3–5×. Finally, we show that raw filtering accelerates analytics workloads in other domains as well, such as filtering binary network packets and analyzing text logs from the Bro [28] intrusion detection system.
To summarize, our contributions are:

1. We introduce raw filtering, an approach that leverages the high query selectivity of many exploratory workloads to filter data before parsing it for improved performance. We also present a set of SIMD-based RFs optimized for modern hardware.

2. We present a fast optimizer that selects an efficient RF cascade in a data- and query-dependent manner while accounting for the potential interdependence between RFs.

3. We evaluate RFs in Sparser, and show that it complements existing parsers and accelerates realistic workloads by up to 9×.

4.2 Problem Statement and Goals

Raw filtering is motivated by the rising volumes of unstructured and semi-structured data, such as text logs, JSON, Avro [14] and Parquet [160]. With organizations embracing a "store everything" philosophy [100, 184] and an ever-increasing volume of machine generated data

ranging from server diagnostic logs to network telemetry [19, 97, 165], accessing the records of relevance for a query is often expensive. Faster I/O devices such as SSDs, 100Gbps Ethernet, and RDMA have also shifted bottlenecks to the CPU [153]. As a result of these shifts, in cases where queries extract a small subset of the data, systems waste cycles parsing records that will eventually be discarded. This scenario—in which only a fraction of the data needs to be parsed—is common for many real-world exploratory workloads, as shown in our Databricks and Censys traces (Figure 4.1).

The goal of raw filtering is to maximize the throughput of parsing and filtering raw, serialized data, by applying query predicates before parsing data. A serialized record could be newline-separated UTF-8 JSON objects or a single line from a text log. Raw filtering primarily accelerates exploratory workloads over unstructured and semi-structured textual formats, but we show in § 4.7 that it is also applicable to queries over binary formats such as Avro and Parquet. Raw filtering is most impactful for selective queries.

In settings where the same data is accessed repeatedly, users can load the data into a DBMS, build an index over it [7, 103]), or convert it into an efficient file format such as Parquet. Raw filtering thus only targets workloads where the data is not yet indexed, perhaps because it is accessed too infrequently to justify continuously building an index (e.g., for high-volume machine telemetry), or because the workloads themselves are performing data ingest. For example, at Databricks, customers commonly use Spark to convert ingested JSON and CSV data to Parquet, as shown by the large fraction of JSON/CSV jobs with selectivity 1 in Figure 4.1. Despite this, there are still a large number of selective queries on CSV and JSON directly (60% of the queries in Figure 4.1). In Censys, a purely exploratory workload, most queries are highly selective.

4.3 Overview

This section gives an overview of raw filtering and introduces Sparser, a system that addresses its challenges.

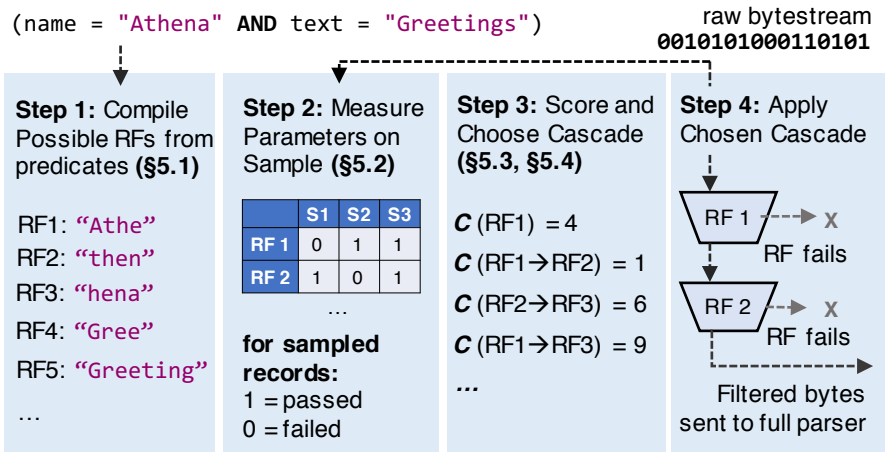


Figure 4.2: An overview of Sparser. Sparser builds a cascade of raw filters to filter data before parsing it.

4.3.1 Raw Filtering

Raw filtering uses a primitive called a *raw filter* (RF) to discard data. Formally, an RF has the following two properties:

1. **An RF operates over a raw bytestream.** Raw filters do not require fully parsing records and operate on an opaque sequence of bytes. For example, the bytestream could be UTF-8 strings encoded in JSON or CSV, or packed binary data encoded in Avro or Parquet.

2. **An RF may produce false positives, but no false negatives.** A raw filter searches for a sequence of bytes that originates from a predicate, but it offers no guarantee that finding a match will produce the same result as the original predicate. Therefore, for supported predicate types (which we discuss in § 4.3.3), raw filters can produce false positives, but no false negatives. Composing multiple RFs into an *RF cascade* [215] can further reduce—but not eliminate—false positives. Thus, an RF (or RF cascade) must be paired with a full, format-specific parser (e.g., RapidJSON or Mison) that can parse the records from the filtered bytestream and apply the query predicates.

We implement raw filtering in a system called Sparser, which addresses two main challenges with using RFs: designing an efficient set of filters that can leverage modern

hardware, and selecting a composition of RFs—an RF cascade—that maximizes throughput.

4.3.2 System Architecture

Figure 4.2 summarizes Sparser's architecture. First, Sparser decomposes the input query predicate into a set of RFs. RFs in Sparser filter data by searching for byte sequences in the bytestream using SIMD instructions. Each RF has a false positive rate for passing records (§4.3.1) as well as an execution cost. The RFs' false positive rates and execution times are not known *a priori* since both are data-dependent. Sparser thus regularly estimates these quantities for the individual RFs by evaluating the individual RFs on a periodic sample of records from the input. These data-dependent factors guide Sparser in choosing which RFs to apply on the full bytestream.

Sparser chooses an RF cascade to execute over the full bytestream using an optimizer to search through the possible combinations of RFs and balance the runtime overhead of applying the RFs with their combined false positive rate. The main challenge in the optimizer is to accurately and efficiently model the joint false positive rates of the RFs, since the individual false positive rates are not independent and the system only measures individual RF rates. Directly measuring the execution overheads and passthrough rates of a combinatorial number of cascades would be prohibitively expensive; instead, the optimizer uses a SIMD-accelerated bitmap data structure to obtain joint probabilities using only the estimates of the *individual* RFs. The optimizer then uses a cost function to compute the expected parsing throughput of the cascade, using the execution costs of the individual RFs, the false positive rates of the RFs, and the execution cost of the full parser. The chosen RF cascade incrementally filters the bytestream using SIMD-based implementations of the RFs. The downstream query engine parses, filters, and processes records that pass the cascade. The records parsed while measuring probabilities are re-parsed in case the parser supports features such as projection.

4.3.3 Interface to Sparser

Sparser takes as input a pointer to a bytestream, a format specifier (used only to determine how records are delimited within the bytestream, e.g., by newline or by a fixed offset) and a

Filter	Example
Exact String Match	`WHERE user.name = 'Athena'`, `WHERE retweet_count = 5000`
Contains String	`WHERE text LIKE '\%VLDB\%'`
Key-Value Match	`WHERE user.verified = true`
Contains Key	`WHERE user.url != NULL`
Conjunctions	`WHERE user.name = 'Athena'` `AND user.verified = true`
Disjunctions	`WHERE user.name = 'Athena'` `OR user.verified = true`

Table 4.1: Filters supported in Sparser. Only equality-based filters are supported—numerical range filters (e.g., `WHERE num_retweets > 10`) are not supported.

list of predicate expressions from the query engine. Sparser outputs a filtered bytestream that can be passed to a full data format parser and for further downstream query processing.

Sparser supports several types of predicate expressions, summarized in Table 4.1. The system supports exact equality matches, substring matches, key-value matches, and key-presence matches. Predicates that check for the presence of a key are only valid for data formats such as JSON, where keys are explicitly present in the record. These supported predicates are also valid over binary data formats (e.g., Avro and Parquet). Furthermore, key names in the predicate can be nested (e.g., `user.name`): nested keys are a shorthand for checking whether each non-leaf key exists (a non-leaf key is any key that has a nested object as a value), and whether the value at the leaf matches the provided filter. For example, in the predicate `user.name = "Athena"`, Sparser checks for the existence of the field `"user"` and `"name"` (where `"name"` is a sub-field of `"user"`), and then checks if the value of the key `"name"` matches `"Athena"`. Applications may also provide a value without an associated key. This is useful for binary formats, where key names often do not appear explicitly. Finally, users can specify arbitrary conjunctions (**AND** queries) and disjunctions (**OR** queries) of individual predicates.

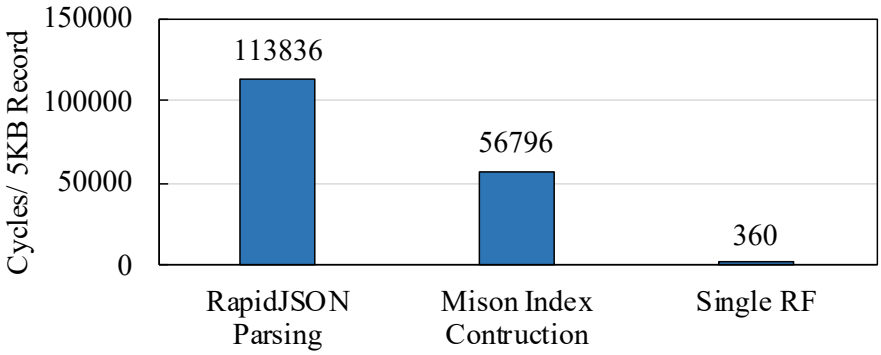


Figure 4.3: Comparison of CPU cycles spent parsing a single 5KB JSON record in L1 cache, using state-of-the-art parsers vs. applying a single RF that searches for a one-byte value with SIMD in the same buffer. The difference in performance is over 100×.

4.3.4 System Limitations

Sparser has a number of limitations and non-goals. First, Sparser's RFs do not support every type of predicate expression found in SQL. In particular, Sparser does not support range-based predicates over numerical values (e.g., `retweet_count > 15`) and inequality predicates for string values (e.g., `name != "Athena"`). Second, Sparser does not support equality in cases where the underlying bytestream could represent equal values in different ways. For example, Sparser does not support integer equality in JSON if the integers are encoded with different representations (e.g., `"3.4"` vs. `"34e-1"`). Third, because the search space of cascades is combinatorial, Sparser bounds the maximum depth of the cascade to at most four RFs. We show in §4.7 that, despite the bounded search space, Sparser's SIMD-accelerated optimizer still produces cascades that accelerate parsing by over 20× in real workloads. Additionally, we show that, on these workloads, Sparser's parsing throughput is only up to 1.2% slower than searching the unbounded set of possible cascades. Finally, Sparser's speedups depend on high query selectivity. Sparser exhibits diminishing speedups on queries with low selectivity, but critically imposes almost no overhead (§4.7).

4.4 Sparser's Raw Filters

Recall that Sparser's objective is to discard records that fail query predicates as fast as possible without parsing. A key challenge in Sparser is thus designing a set of efficient filtering operators that inspect only a raw bytestream: an operator over a bytestream necessarily lacks access to format-specific structural information built via parsing (e.g., the offsets of specific fields [118]).

To address this challenge, Sparser utilizes a set of SIMD-accelerated raw filters (RFs) to discard data by searching for format-agnostic byte sequences in the input. The RFs use this design to take advantage of the throughput gap between parsing a record to extract structural information and scanning it to search for a byte sequence that fits in a single SIMD vector lane: Figure 4.3 illustrates this by comparing cycles spent parsing a 5KB record with two best-of-breed JSON parsers and a SIMD search for a one-byte value in the record.

An RF returns a boolean signal specifying whether it passed or failed a record depending on whether it found its byte sequence. Sparser produces several possible RFs for a given query and chooses a cascade using these RFs (§4.5).Sparser contains two RF primitives: *substring search* and *key-value search*.

4.4.1 Substring Search

The substring search RF is the main filtering primitive in Sparser. This RF searches for a byte sequence (i.e., a binary "substring") that indicates a record could pass a query predicate. Consider the predicate in Listing 4.1, which selects records where `text` and `name` match the specified value. Assuming the underlying bytestream encodes data as UTF-8, the substring search RF can look for several number of bytes sequence that indicate a record could pass this filter: a few examples are `"VLDB"`, `"submissi"`, `"subm"`, and `"Athe"`. If the RF does not find any of these strings, it can discard the record with no false negatives since the predicate cannot be satisfied.

Listing 4.1 A sample query predicate that Sparser takes as input.

```
name = "Athena" AND text = "My submission to VLDB"
```

Figure 4.4: An example of a substring search. The length of the substring denotes the number of shifts required. In this example, the fourth shift produces a match since one of the repeated 4-byte sequences in the vector register matches the input text.

The substring search RF leverages SIMD instructions in the CPU by packing multiple copies of the byte sequence into a single vector register. This allows the RF to search over multiple bytes in parallel with a small sliding window. To illustrate this, consider the predicate in Listing 4.1 and the input string in Figure 4.4. The figure shows an example of an RF with the 4-byte UTF-8 sequence `"VLDB"`. This string is repeated eight times in a 32-byte vector register. To find all occurrences of the sequence, the operator needs to look at a 4-byte-wide sliding window. In the example, the fourth shift finds an occurrence of the sequence in the input, meaning that the RF passes the record containing the input string. Sparser considers 2-, 4- and 8-byte substrings with this RF.

Note that the value of the `text` field in Figure 4.4 does not match the predicate on the `text` field in Listing 4.1, indicating that the RF in the example passed as a false positive. The RF does not return false negatives since it finds all occurrences of the sequence. The main advantage of the substring search RF is that, for a well-chosen sequence, the operator can reject inputs at nearly the speed of streaming data through a CPU core.

4.4.2 Key-Value Search

The key-value search RF searches for all co-occurrences of a key and a corresponding value within a record in the input data; this operator is useful for finding associations between two sequences. For example, if we search for `name = "Athena"`, we want to increase the probability that the `"Athena"` we find is associated with the field `"name"`. This operator is especially useful in cases where the key and value appear frequently independently, but rarely together. One example is searching for the value `true` in a JSON record, which likely appears somewhere in the record but not always with the key specified by the predicate.

```
Key: name Value: Athena Delimiter: ,
Ex 1: “name”: “Athena”,                (PASS)
Ex 2: “name”: “Actually, Athena”       (FAIL)
Ex 3: “text”: “My name is Athena”      (PASS, False positive)
```

Figure 4.5: Example of using the key-value search RF for the predicate `name = "Athena"`. The key is `name`, the value is `Athena`, and the delimiter is `','`. The underlined regions are the parts of the input where the filter looks for the value after finding the key.

This RF takes three parameters: a key, a value, and a delimiter set. Keys and values are byte sequences, as in the substring search operator. The delimiter set is a set of one-byte characters that signals the stopping point for a search; the value search term must appear after the key and before any of the delimiters. Thus, after finding an occurrence of the key, the operator searches for the value and terminates its search at the first occurring delimiter character found after the key. Sparser can search for the key, value, and stopping point using the packed-vector technique from the substring search RF for hardware efficiency, by either checking for a 2-, 4-, or 8-byte substring of both the key and value.

Unlike the substring search RF (which searches for arbitrary sequences and can be used for both **`LIKE`** and `=` predicates), the key-value search operator is only applicable for equality predicates; **`LIKE`** predicates are not supported. To understand why this constraint is necessary, Figure 4.5 provides an example of a key-value search on a JSON record that could yield false negatives if a **`LIKE`** predicate was specified instead of an equality predicate (i.e., `name` **`LIKE`** `Athena` instead of `name = "Athena"`). In this example, the operator first looks for the key `"name"`. After finding it, the operator looks to see if the key's corresponding value (the bytes before the delimiter `','`) is exactly equal to `"Athena"`.

If the system permitted **`LIKE`** queries with the key-value search RF, the second example in the figure, `"name": "Actually, Athena"`, would return a false negative, since the value search term `"Athena"` associated with the key `"name"` would never be found. Concretely, the problem is that the delimiter `','` can also appear within the value in this record, and it is impossible to distinguish between these two cases without a full parse of the entire key-value pair (and by extension, the full record)[1]. By only allowing equality predicates,

[1] Extending the set of delimiters to include `'"'` would also yield false negatives for scenarios in which an escaped double-quote occurs within the entire value string.

false negatives cannot occur: if the search finds one of the delimiters but not the value search term, then either the delimiter appears after the entire value, or the value search term is not present. Sparser also disallows RFs where the value search term and the delimiter set have overlapping bytes for the same reason.

4.5 Sparser's Optimizer

Sparser's RFs provide an efficient but inexact mechanism for discarding records before parsing: these operators have high throughput but also produce false positives. To decrease the overall false positive rate while processing data, Sparser combines individual RFs into an RF cascade to maximize the overall filtering and parsing throughput. Finding the best cascade is challenging because a cascade's performance is both data- and query-dependent. Therefore, we present an optimizer that employs a cost model to score and select the best RF cascade. The optimizer takes as inputs a query predicate, a bytestream from the input file, and a full parser, and outputs an RF cascade to maximize the expected parsing throughput. Overall, the optimizer proceeds as follows:

1. Compile a set of possible RFs based on the clauses in the query predicate (§4.5.1).
2. Draw a sample of records from the input and measure data-dependent parameters such as the execution cost of the full parser, the execution costs of each RF, and the passthrough rates of each RF on the sample (§4.5.2).
3. Generate valid cascades to evaluate using the possible RFs (§4.5.3). A valid cascade does not produce false negatives.
4. Enumerate possible valid RF cascades and select the best one using the estimated costs and passthrough rates (§4.5.4).

4.5.1 Compiling Predicates into Possible RFs

The first task in the optimizer is to convert the user-specified query predicate into a set of possible RFs. The query predicate is a boolean expression evaluated on each record: if a

record causes the expression to evaluate to true, the record passes, and if the expression evaluates to false, the record may be discarded. By definition, the RFs generated by the optimizer for a given query must produce only false positives with respect to this boolean expression, but no false negatives (i.e., an RF may occasionally return true when the predicate evaluates to false, but never vice versa). The query predicate may also contain conjunctions and disjunctions that the optimizer must consider when generating RFs. Sparser thus takes following steps to produce a set of possible RFs:

1. Convert the boolean query predicate to disjunctive normal form (i.e., of the form $(a \wedge b \ldots) \vee (c \wedge \ldots) \vee \ldots$). DNF allows Sparser's optimizer to systematically generate RF cascades that never produce false negatives. We refer to an expression with only conjunctions (e.g., $a \wedge b \wedge \ldots$) as a *conjunctive clause*.

2. Convert each *simple predicate* (i.e., predicates without conjunctions or disjunctions, such as equality or `LIKE` predicates) in the conjunctive clauses into one or more RFs. We elaborate on this procedure below using Listing 4.2 as an example.

Listing 4.2 An example predicate in DNF with two conjunctive clauses and three simple predicates.

```
(name = "Athena" AND text = "Greetings") OR name = "Jupiter"
```

Since each RF represents a search for a raw byte sequence, the conversion from a simple predicate to a set of RFs is format-dependent. For example, when parsing JSON, a predicate such as `name = "Athena"` in Listing 4.2 will produce both substring and key-value search RFs. However, for binary formats such as Avro and Parquet, field names (e.g., `text`) are typically not present in the data explicitly, which means that key-value search RFs would not be effective. Therefore, the optimizer only produces substring search RFs for these binary formats. For the sake of brevity, this section discusses only the JSON format (and assumes queries are over raw textual JSON data), which supports all RFs available in Sparser.

For each simple predicate, Sparser produces a substring search RF for each 4- and 8-byte substring of each *token* in the predicate expression. A token is a single contiguous value in the underlying bytestream. Sparser generates 2-byte substring search RFs only if

a token is less than 4-bytes long. As an example, the simple predicate `name = "Athena"` in Listing 4.2 contains two tokens: `"name"` and `"Athena"`. For this predicate, the optimizer would generate the following substring RFs: `"name"`, `"Athe"`, `"then"`, and `"hena"`. The optimizer additionally produces an RF that searches for each token in its entirety: `"name"` and `"Athena"` in this instance. Lastly, because `name = "Athena"` is an equality predicate, the optimizer generates key-value search RFs with the key `"name"` and the value set to each of the 4-byte substrings of `"Athena"`.

Each simple predicate is now associated with a set of RFs where each RF only produces false positives. If any RF in the set fails, the simple predicate also fails. By extension, each conjunctive clause is also associated with a set of RFs with the same property: for a conjunctive clause with n simple predicates, this set is $\bigcup_{i=1}^{n} r_i$, where r_i is the RF set of the ith simple predicate in the clause. This follows from the fact that RFs cannot produce false negatives: if any one RF in a conjunctive clause fails, some simple predicate failed, and so the full conjunctive clause must fail. To safely discard a record when processing a query with disjunctions, the optimizer must follow one rule when generating RF cascades: an RF from *each* conjunctive clause must fail to prevent false negatives. Returning to the example in Listing 4.2, the optimizer must ensure that an RF from *both* conjunctive clauses fails before discarding a record.

4.5.2 Estimating Parameters by Sampling

The next step in Sparser's optimizer is to estimate data-dependent parameters by drawing a sample of records from the input and executing the possible RFs from §4.5.1 and the full parser on the sample. Specifically, the optimizer requires the passthrough rates of the individual RFs, the runtime costs of executing the individual RFs, and the runtime cost of the full parser. This sampling technique is necessary because these parameters can vary significantly based on the format and dataset. For example, parsing a binary format such as Parquet requires fewer cycles than parsing a textual format such as JSON. Thus, for Parquet data, Sparser should choose a computationally inexpensive cascade to minimize runtime overhead, and the optimizer should capture that tradeoff.

To store the passthrough rates of the individual RFs, the optimizer uses a compact

Algorithm 1 Estimating Data-Dependent Parameters by Sampling

```
1:  procedure ESTIMATE(records, candidateRFs)
2:      C ← len(candidateRFs)
3:      R ← len(records)
4:      ParserRuntime ← 0                          ▷ Average parser runtime
5:      RFRuntimes[C] ← 0                          ▷ Average RF runtimes
6:      B[C,R] ← 0_{C,R}                           ▷ C × R matrix of bits
7:      for (j, record) ∈ records do
8:          update running avg. ParserRuntime with parser(record)
9:          for (i, RF) ∈ candidateRF do
10:             update running avg. RFRuntimes[i] with RF on record
11:             if RF ∈ record then
12:                 B[i, j] ← 1
        return B, ParserRuntime, RFRuntimes
```

bit-matrix representation. This matrix stores a 1 at position i, j if the ith RF passes the jth record in the sample, and a 0 otherwise. Rather than storing the passthrough rate as a single numerical value per RF, each row in the bit-matrix compactly represents precisely which records in the sample passed for each RF as a bitmap. The optimizer leverages this data structure when scoring cascades with its cost model (§4.5.4) to compute the joint passthrough rates of multiple RFs efficiently.

Algorithm 1 summarizes the full parameter estimation procedure. The optimizer first initializes a $C \times R$ bit-matrix, where C is the number of possible RFs and R is the number of sampled records. For each sampled record, the optimizer updates an average of the full parser's running time in CPU cycles (e.g., using the x86 `rdtsc` instruction). Sparser can use any full parser, such as Mison. Then, for each RF, the optimizer applies the RF to the sampled record and measures the running time in CPU cycles. If RF i passes the record j, bit i, j is set to 1 in the matrix. After sampling, the optimizer has a populated matrix representing the records in the sample that passed for each RF, the average running time of each RF, and the average running time of the full parser.

4.5.3 Cascade Generation and Search Space

The third step in the optimizer is to generate valid RF cascades from the query predicate. Recall that, for a cascade to be valid in Sparser's optimizer, *at least one RF from each*

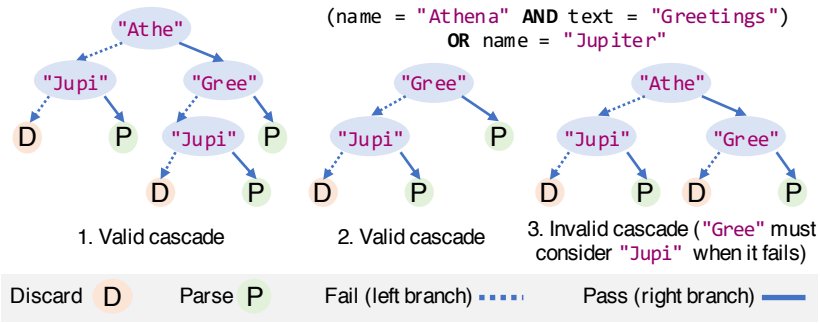


Figure 4.6: A set of RF cascades for the predicate in Listing 4.2. The third cascade does not check an RF from both conjunctive clauses on some paths and is thus invalid. The second cascade does not check all RFs in a conjunction but is still valid, since it checks one RF from each conjunctive clause.

conjunctive clause in the query predicate must fail before discarding a record. RF cascades are thus binary trees, where non-leaf nodes are RFs, leaf nodes are decisions (parse or discard), and edges represent whether the RF passes or fails a record. Figure 4.6 shows an example query predicate with examples of generated valid and invalid cascades. By considering at least one RF from every conjunctive clause, the optimizer only generates valid cascades, which may have false positives—but no false negatives—when evaluated on a given record.

The optimizer enumerates all cascades up to depth D that meet the above constraint. Our optimizer uses a pruning rule to prune the search space further by skipping cascades where two RFs from the same conjunction have overlapping substrings (e.g., a cascade which searches for `"Athena"` and `"Athe"`). For completeness, the optimizer also considers the empty cascade (i.e., always parsing each record) to allow efficient formats such as Parquet to skip raw filtering altogether for queries that will exhibit no speedup. In our implementation, we set $D = max(\#\text{ Conjunctive Clauses}, 4)$ RFs, and generate up to 32 possible candidate RFs. If there are more than 32 possible RFs, we select 32 by picking a random RF generated from each token in a round-robin fashion. For the queries in §4.7, we show that these choices still generate cascades with overall parsing time within 10% of the globally optimal cascade.

Given a $C \times R$ bit-matrix of estimates B:	
$\Pr[a]$	$\texttt{popcnt}(B[a,\ldots])/R$
$\Pr[\neg a]$	$\texttt{popcnt}(\neg B[a,\ldots])/R$
$\Pr[a,..,z]$	$\texttt{popcnt}(B[a,\ldots] \wedge \ldots \wedge B[z,\ldots])/R$

Table 4.2: Estimating joint probabilities using the bit-matrix. $B[i,...]$ indicates accessing the sampled bits for RF i. Bit i,j is set if RF i passed sampled record j. Bitwise operators $(\neg, \wedge)$ use SIMD.

4.5.4 Choosing the Best Cascade

Given a set of candidate cascades, the optimizer's final task is to choose the best cascade. To make this choice, the optimizer evaluates the expected per-record CPU time of each cascade using a cost model, and selects the one with the lowest expected cost.

The cost of an RF cascade depends on c_i, the cost of executing the ith RF in given cascade, $\Pr[execute_i]$, the probability of executing the ith RF, as well as c_{parse} and $\Pr[execute_{parse}]$, which represent the respective cost and probability of executing the full parser. The optimizer measures the passthrough rates of the individual RFs in the previous step, as well as the execution times of the RFs and the full parser (c_i and c_{parse} respectively). However, for any RF i that relies on other RFs to pass or fail, $\Pr[execute_i]$ will be a joint probability. For example, in the example cascade $x \rightarrow y \rightarrow z$, the RF z will only execute after the first two RFs passed the record; therefore, $\Pr[execute_z] = \Pr[x,y]$, where $\Pr[x]$ and $\Pr[y]$ are the passthrough rates of x and y the optimizer previously measured.

The challenge is that these joint probabilities are not necessarily independent (i.e., $\Pr[x,y] \neq \Pr[x]\Pr[y]$). For example, an RF that searches for the substring `"Gree"` may be highly correlated with an RF that searches for the substring `"ting"`, because both may indicate the presence of the string `"Greetings"`. Our evaluation shows that a strawman optimizer that does not consider these correlations achieves parsing throughputs 2.5× lower than Sparser, because the strawman chooses an inferior cascade.

Another strawman solution is to estimate the joint passthrough rates of multiple RFs directly by executing RF cascades on the sample of records described in §4.5.2. However, executing each combination of RFs on the sample is inefficient, since this requires executing a combinatorial number of cascades.

Instead, Sparser's optimizer uses the bit-matrix representation (§4.5.2) to quickly estimate the joint passthrough rates using only sample-based measurements of the individual RFs. Recall that the matrix stores as a single bit whether an RF passes or fails each record in the sample (a 1 if the record passed the RF, and 0 otherwise). The passthrough rate of RF i is thus the number of 1s (i.e., the `popcnt`) of the ith row, or bitmap, in the matrix. Conversely, the probability of any RF i not passing a record is the number of 0s in row i. The joint passthrough rate of two RFs i and k is the number of 1s in the bitmap after taking the bitwise-and of the ith and kth bitmaps.

The key advantage to this approach is that these bitwise operations have SIMD support in modern hardware and complete in 1-3 cycles on 256-bit values on modern CPUs (roughly 1ns on a 3GHz processor). The matrix thus allows the optimizer to quickly estimate joint passthrough probabilities of RFs. This optimization allows Sparser to scale efficiently and accurately to handle complex user-specified query predicates that combine multiple predicate expressions. Table 4.2 summarizes the matrix operations.

With an efficient methodology to accurately compute the joint probabilities, the optimizer scores each cascade and chooses the one with the lowest cost. Let $R = \{r_1, \ldots, r_n\}$ be the set of RFs in the RF cascade. To evaluate C_R, the expected cost of the cascade on a single record, Sparser's optimizer computes the following:

$$C_R = \left(\sum_{i \in R} \Pr[execute_i] \cdot c_i \right) + \Pr[execute_{parse}] \cdot c_{parse}.$$

As an example, consider the first cascade in Figure 4.6. The probabilities of executing each RF in the cascade are:

$$
\begin{aligned}
\Pr[execute_{\text{Athe}}] &= 1, \\
\Pr[execute_{\text{Gree}}] &= \Pr[\text{Athe}], \\
\Pr[execute_{\text{Jupi}}] &= \Pr[\neg\text{Athe}] + \Pr[\text{Athe}, \neg\text{Gree}], \\
\Pr[execute_{parse}] &= \Pr[\neg\text{Athe}, \text{Jupi}] + \\
&\qquad \Pr[\text{Athe}, \text{Gree}] + \Pr[\text{Athe}, \neg\text{Gree}, \text{Jupi}].
\end{aligned}
$$

The cost of the full cascade is therefore:

$$\sum_{i \in \{\text{Athe},\text{Gree},\text{Jupi},parse\}} \Pr[execute_i] \times c_i.$$

§4.7.4 shows that, with the bit-matrix technique to compute joint probabilities, the optimizer adds at most 1.2% overhead in our benchmark queries, including sampling and scoring time.

4.5.5 Periodic Resampling

Sparser occasionally recalibrates its cascade to account for data skew or sorting in the underlying input file. §4.7 shows that recalibration is important for minimizing parsing runtime over the *entire* input, because a cascade chosen at the beginning of the dataset may not be effective at the end. For instance, consider an RF that filters on a particular date, and the underlying input records are also sorted by date. The RF may be highly ineffective for one range of the file (e.g., the range of records that all match the given date in the filter) and very effective for other ranges. To address this issue, Sparser maintains an exponentially weighted moving average of its own parsing throughput. In our implementation, we update this average on every 100MB block of input data. If the average throughput deviates significantly (e.g., 20% in our implementation), Sparser reruns its optimizer algorithm to select a new RF cascade.

4.6 Implementation

We implemented Sparser's optimizer and RFs in roughly 4000 lines of C. Our implementation supports mapping query predicates to RFs for text logs, JSON, Avro, Parquet, and PCAP, the standard binary packet capture format [162]. RFs leverage Intel's AVX2 [16] vector extensions. Other architectures feature similar operators [140].

JSON. Our JSON implementation uses two state-of-the-art JSON parsers: Mison [118] and RapidJSON [178]. Sparser assumes that the input bytestream contains textual JSON records (e.g., a Tweet from the Twitter Stream API) terminated by a newline character (similar to other systems such as Spark [66, 224]). Sparser uses SIMD to find the start of each record by searching for the newline, and applies the RF cascade on the raw byte buffer, where each RF searches until the following newline. If the record passes the full cascade, Sparser passes a pointer to the beginning of the record to the full parser. Otherwise, Sparser skips it and continues filtering the remaining bytestream. Our implementation also supports case-insensitive search for ASCII (i.e., letters A–Z). These characters have upper and lowercase values that differ by 32 (e.g., `'a' - 'A' = 32`), so Sparser can use SIMD to convert a search query and the target text to all lowercase to perform a case-insensitive search.

Our implementation has a few limitations. First, the JSON standard allows floating-point values to be formatted using scientific notation (e.g., `3.4` vs. `34E-1`). Sparser does not support searches for data represented in this way. Second, Sparser does not support values that have different string representations encoding the same numerical value (e.g., due to loss in precision, such as `0.99..` vs. `1.0`). For both of these cases, users can set a flag to specify that numerically-valued fields may be encoded in this way, and Sparser will treat predicates over them as requiring a full parse. We found that both cases did not appear in our machine-generated real-world datasets. Sparser can handle integer equality queries (e.g., searches for user IDs) by searching for substrings of the integer.

Finally, the RFC 8259 JSON standard [27] allows any character to be Unicode-escaped (e.g., the character `"A"` and its escaped Unicode representation, `"\u41"`, should be considered equal). To handle Unicode escapes, Sparser additionally searches each record for the `"\u"`

escape and falls back to a slow path if this sequence is found. This is the only valid alternate representation of a character permitted by the JSON standard[2]: the standard does not allow unescaped whitespace (except space) in string literals [27] (e.g., a tab literal in a string is disallowed and must be represented using the Unicode escape or `"\t"`), so the Unicode is the only special case. Other characters such as `"\"` must also be escaped in JSON, but similarly only have a single possible non-Unicode-escaped representation.

Binary Formats. For binary data, records are not explicitly delimited (e.g., by newlines), so Sparser does not know where to start or stop a search for a given RF. Rather than search line by line, Sparser treats the full input buffer as a single record and begins searching from the very beginning of the buffer. When an RF finds a match, Sparser uses a format-specific function for navigating and locating different records in the file. In our implementation, this function moves a pointer from the last processed record by the full parser to the record containing the match. The function also computes the end of the matched record (in most binary formats, this is the start of the record plus the record length, stored as part of the data) and returns both the pointer to the matching record and the length back to Sparser's search function to check the remaining RFs within the byte range. If all RF matches pass, Sparser calls the callback again and the record is processed just as before. Otherwise, Sparser resets its record-level state and continues. Sparser speeds up parsing binary formats by skipping over large blocks of data, processing only regions that contain RF matches (§4.7).

Integration with Spark. We also integrated Sparser with Spark SQL [11] using Spark's Data Sources API. The Data Sources API enables column pruning and filtering to be pushed down to the parser itself, in line with the core tenets of Sparser. To date, however, these features have primarily been used only in optimized columnar formats, such as Parquet, or for index construction to access individual rows in the underlying data source efficiently. The API passes individual file partitions (which map to a filename, byte offset, and length) to a callback function; these arguments are then passed via the Java Native Interface (JNI) to call into Sparser's C library. This means that Sparser runs its calibration, raw filtering, and parsing steps on a per file-partition basis, rather than on a single file. Sparser reads, filters,

[2]The `"/"` is the only exception and has three valid representations.

Name	Query	Selectivity (%)
Twitter 1	`COUNT(*)WHERE text LIKE '%Donald Trump%'AND date LIKE '%Sep 13%'`	0.1324
Twitter 2	`user.id, SUM(retweet_count)WHERE text LIKE '%Obama%'GROUP BY user.id`	0.2855
Twitter 3	`id WHERE user.lang == 'msa'`	0.0020
Twitter 4	`distinct user.id WHERE text LIKE '%@realDonaldTrump%'`	0.3313
Censys 1	`COUNT(*)WHERE p23.telnet.banner.banner != null AND autonomous_system.asn = 9318`	0.0058
Censys 2	`COUNT(*)WHERE p80.http.get.body LIKE '%content=wordpress 3.5.1%'`	0.0032
Censys 3	`COUNT(*)WHERE autonomous_system.asn=2516`	0.0757
Censys 4	`COUNT(*)WHERE location.country = 'Chile'AND p80.http.get.status_code != null`	0.1884
Censys 5	`COUNT(*)WHERE p80.http.get.servers.server LIKE '%DIR-300%'`	0.1884
Censys 6	`COUNT(*)WHERE p110.pop3.starttls.banner != null OR p995.pop3s.tls.banner != null`	0.0001
Censys 7	`COUNT(*)WHERE p21.ftp.banner.banner LIKE '%Seagate Central Shared%'`	2.8862
Censys 8	`COUNT(*)WHERE p20000.dnp3.status.support=true`	0.0002
Censys 9	`asn, COUNT(ipnt)WHERE autonomous_system.name LIKE '%Verizon%'GROUP BY asn`	0.0002
Bro 1	`COUNT(*)WHERE record LIKE '%HTTP%'AND record LIKE '%Application%'`	15.324
Bro 2	`COUNT(*)WHERE record LIKE '%HTTP%'AND (record LIKE '%Java*dosexec%'OR record LIKE '%dosexec*Java%')`	1.1100
Bro 3	`COUNT(*)WHERE record LIKE '%HTTP%'AND record LIKE '%http*dosexec%'AND record LIKE '%GET%'`	0.5450
Bro 4	`COUNT(*)WHERE record LIKE '%HTTP%'AND (record LIKE '%80%'OR record LIKE '%6666%' OR record LIKE '%8888%'OR record LIKE '%8080%')`	12.294
PCAP 1	`* WHERE http.request.header LIKE '%GET%'`	81
PCAP 2	`* WHERE http.response AND http.content_type LIKE '%image/gif%'`	1.13
PCAP 3	`Flows WHERE tcp.port=110 AND pop.request.parameter LIKE '%user%'`	0.001
PCAP 4	`Flows WHERE http.header LIKE '%POST%'AND http.body LIKE '%password%'`	0.0095

Table 4.3: Queries used in the evaluation. §4.7.1 elaborates on the datasets and sources of the queries.

and parses data, writing the extracted fields directly to an off-heap buffer allocated in Spark to store the parsed records. Spark treats the off-heap buffer as a standard DataFrame for the remainder of the query.

4.7 Evaluation

We evaluate Sparser and the raw filtering approach across a variety of workloads, datasets, and data formats. We find that:

- With raw filtering, Sparser accelerates diverse analytics workloads by filtering out records that do not need to be parsed. Sparser can improve the parsing throughput of state-of-the-art JSON parsers up to 22×. For distributed workloads, Sparser can improve the end-to-end runtime of Spark SQL queries up to 9×.
- Sparser can accelerate parsing throughput of binary formats such as Avro and Parquet by up to 5×. For queries over unstructured text logs, Sparser can reduce the runtime by up to 4×.
- Sparser's optimizer improves parsing performance compared to strawman approaches, selecting RF cascades that are within 10% of the global optimum while only incurring a 1.2% runtime overhead during parsing.

4.7.1 Experimental Setup

We ran distributed Spark experiments on a 10-node Google Cloud Engine cluster using the `n1-highmem-4` instance type, where each worker had 4 vCPUs from an 2.2GHz Intel E5 v4 (Broadwell), 26GB of memory, and locally attached SSDs. We used Spark v2.2 for our cluster experiments. Single-node benchmarks ran on an Intel Xeon E5-2690 v4 CPU with 512GB of memory. All single-node experiments were single-threaded—we found that Sparser scales linearly with the number of cores for each workload, and omit these results for brevity.

Our experiments ran over the following real-world datasets and queries, with some experiments running over a subset of the data. Table 4.3 summarizes the queries and their selectivities.

Twitter Tweets. We used the Twitter Streaming API [211] to collect 68GB of JSON tweets. We benchmarked against 23GB of the data for our single-node experiments, and the entire dataset for our distributed experiments. We obtained queries from [118, 203].

Censys Scan. We obtained a 652GB JSON dataset from Censys [63], a search engine broadly used in the Internet security community. We benchmarked against 16GB of the data for our single-node experiments, and the entire dataset for our distributed experiments. Each record in the dataset represents an open port on the wide-area Internet. Censys data is highly nested: each data point is over 5KB in size. We obtained the queries over Censys data by sampling randomly from the 50,000 most popular queries to the engine. Raw data is available at [35].

Bro IDS Logs. Bro [28] is a widely deployed network intrusion detection system that generates ASCII logs while monitoring networks. Network security analysts perform post-hoc data analyses on these logs to find anomalies. We obtained a 10GB dataset of logs and a set of queries over them from security forensics exercises [29–31].

Packet Captures. To evaluate Sparser's applicability in other domains, we obtained a 5GB trace of network traffic from a university network. Traffic is stored in standard binary file format called PCAP [162], which stores the binary representation of individual network packets. We selected queries for this trace from [43, 77, 84], which represent real workloads over captured network traffic, such as searching for insecure network connections.

4.7.2 End-to-End Workloads

Spark Queries. To benchmark Sparser's effectiveness parsing JSON in a production-quality query engine, we executed the four Twitter queries and nine Censys queries (all of

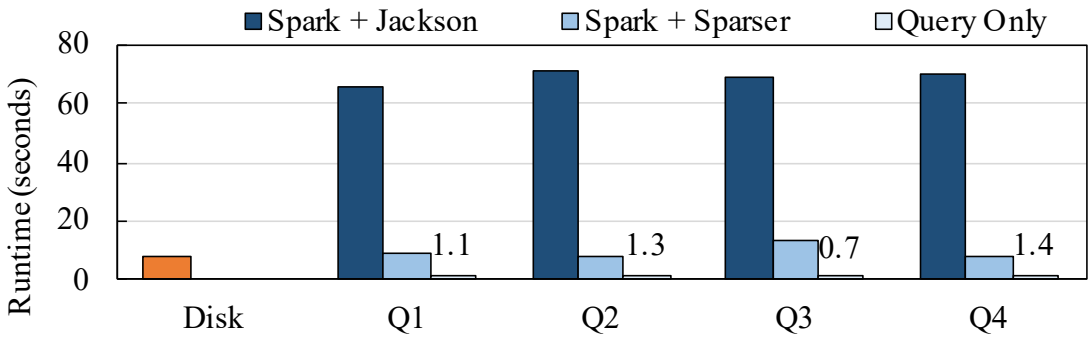


Figure 4.7: Twitter queries on Spark over JSON data end-to-end. The time to load the data from disk is shown on the far left.

which are over JSON data) from Table 4.3 on our 10-node Spark cluster and measured the end-to-end execution time.

Figures 4.7 and 4.8 show the end-to-end execution time of native Spark (which uses the Jackson JSON parser [96]) vs. Sparser integrated with Spark via the Data Source API [196]. Data is read from disk and passed to Sparser as chunks of raw bytes. Sparser runs its optimizer, chooses an RF cascade, and filters the batches of data, returning a filtered bytestream to Spark. Spark then parses the filtered bytestream into a Spark SQL DataFrame and processes the query. The presented execution time includes disk load, parsing (in Sparser, this includes both the optimizer's runtime and filtering), and querying. In each query, Sparser outperforms Spark without Sparser's raw filtering by at least 3×, and up to 9×.

Packet Filtering. To illustrate that raw filtering can accelerate a diverse set of analytics workloads, we apply it to filtering captured network traffic and evaluate its performance. Using Sparser, we implemented a simple packet-analysis library and benchmarked its throughput against `tshark` [208], a standard tool in the networking community for analyzing packet traces. We also compared against `tcpdump` [204] (a lightweight version of `tshark`) and a simple `libpcap`-based C program that hard-codes the four queries. The `libpcap` [119] library is the standard C library for parsing network packets: this baseline represents the packet parsing procedure without any overheads imposed by other systems.

Figure 4.9 shows the results. In the first query, each system (except `tshark`) performs

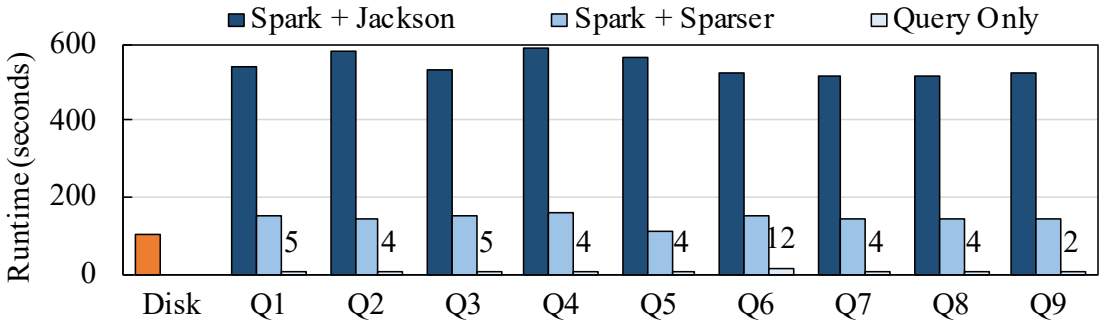


Figure 4.8: Censys queries on Spark over JSON data.

similarly because the query's relatively low selectivity prevents Sparser from delivering large speedups. In the second query, Sparser performs roughly 3× faster than `libpcap`, which parses and searches each packet for a string. The `tcpdump` tool does not support searches in a packet's payload. In the third and fourth queries, Sparser outperforms `libpcap` by 2.5×. Overall, Sparser accelerates these packet filtering workloads using its search technique, even compared to libraries with little overhead.

Bro Log File Analysis. Ad-hoc log analysis is a common task, especially for IT administrators maintaining servers or security experts analyzing logs from systems such as Bro [28]. Tools such as `awk` and `grep` aid in these analyses; generally, the first step is to use these tools to filter for events of interest (e.g., errors, specific protocols, etc.). We collected a set of network forensic analysis workloads [29–31] and replaced the log filtering stage with Sparser's optimizer and raw filtering technique. Each query looks for a set of threat signatures and then performs a count with `wc`.

Figure 4.10 shows that Sparser shows speedups of up to 4× on these queries. For the Q1, Q2, and Q4, the performance of all three systems is similar since both GNU `grep` and `ripgrep` (the fastest `grep` implementation we found [68]) are optimized and use vectorization. Sparser marginally outperforms both by searching for the most uncommon substring to discard rows faster.

In Q3 we search for one of three terms in each line, but some terms are much more selective than others. Sparser's optimizer identifies the most selective term and searches for it, while the other two systems naively search for the first term (the default policy with

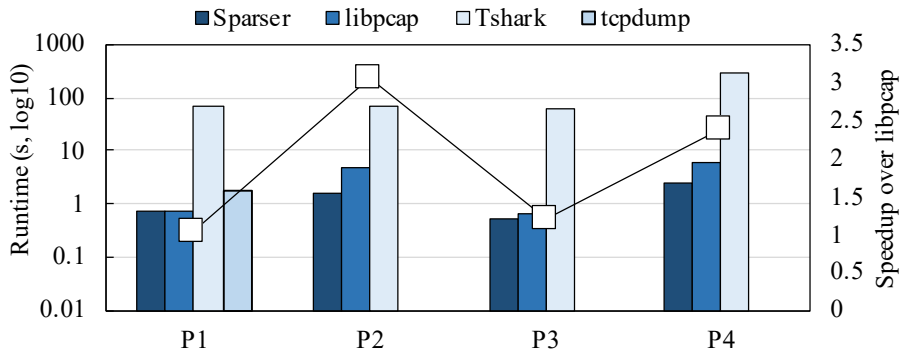


Figure 4.9: Packet filtering on binary `tcpdump` PCAP files. The line shows the speedup of Sparser over `libpcap`.

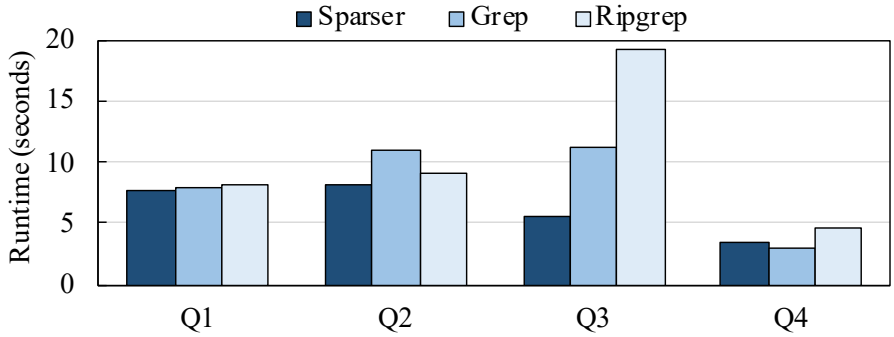


Figure 4.10: Performance of log analysis tasks on Bro IDS logs, where Sparser is used as a `grep`-like search tool.

multiple search strings). Overall, Sparser is competitive with and sometimes faster than command-line string search tools such as `grep` and `ripgrep`, despite the fact that these tools are also designed for hardware efficiency and do not perform any parsing of the inputs. Performance depends only on the *skew* in selectivity of individual search tokens in these queries, since both the `grep` tools and Sparser search for values on each record. Sparser shows the greatest speedups when leveraging high selectivity to *avoid parsing*.

TPC-H Queries. We evaluated Sparser on TPC-H Q3, Q12, and Q19 using a flat JSON file to represent each table and ran the queries on Spark with scale factor 1 (3GB of JSON data). We chose these queries because they contain tables with varying selectivities for Sparser-supported predicates (28%, 0.5%, and 2.1% respectively [118]). Figure 4.11 shows

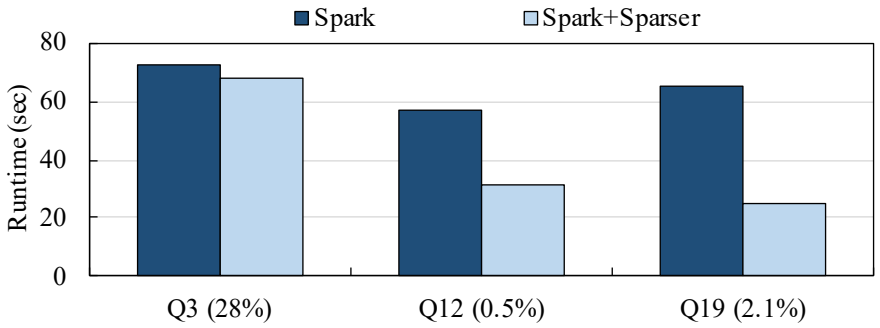


Figure 4.11: Performance on a subset of TPC-H queries with varying Sparser-supported predicate selectivities (in parentheses). Sparser exhibits the largest speedups when *each* table is filtered.

the results. Q3 exhibits the smallest speedup because the Sparser-supported predicate applies to the smaller Orders table, but processing time is dominated by query evaluation and parsing the larger LineItem table, which Sparser cannot filter. As demonstrated by [118], a faster parser can improve performance by reducing loading times here. Q12 exhibits a higher speedup because most records in the LineItem table are not parsed. Finally, Q19 exhibits the largest speedup, even though Q3 has a higher selectivity on the filtered table, because *both* tables in Q19 are filtered. Overall, Sparser will only exhibit substantial speedups in SQL queries when it can apply RFs to *each* table.

4.7.3 Comparison with Other Parsers

JSON Parsing Performance. To evaluate Sparser's ability to accelerate JSON parsing, we integrated Sparser with RapidJSON [178] and Mison [118], two of the fastest C/C++-based parsers available. Because the Mison implementation is not open source, we implemented its algorithm as described in the paper using Intel AVX2, and benchmark only against the time to create its per-record index (i.e., we assume that using the index to extract the fields and evaluating the predicate is free). We believe this is a fair baseline.

Figure 4.12 shows the parsing runtime of RapidJSON and Mison both with and without Sparser on the nine Censys queries; we benchmarked the queries on a 15GB sample of the full dataset on a single node. Because these queries have low selectivity, Sparser can accelerate these optimized parsers up to 22×, due to its ability to efficiently filter records

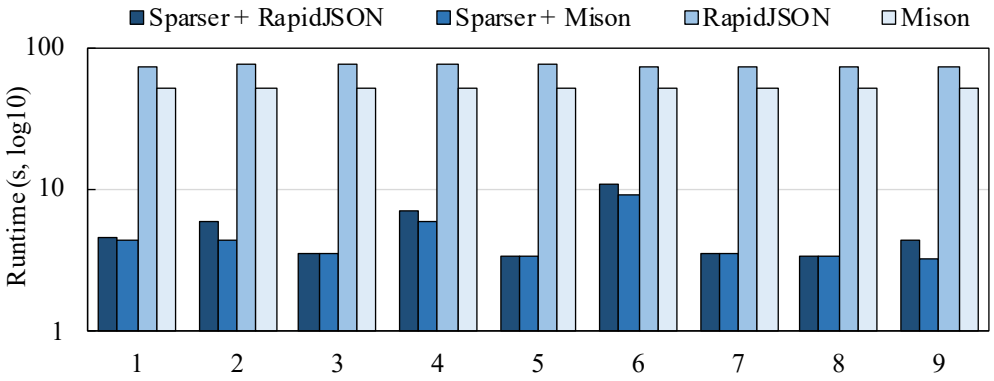


Figure 4.12: Parsing time for the nine Censys JSON queries compared against Mison and RapidJSON.

that do not need to be parsed. Raw filtering is thus complementary to Mison's projection optimizations.

JSON Parsing Sensitivity to Selectivity. An underlying assumption of raw filtering is that many queries in exploratory workloads exhibit high selectivity. To study Sparser's sensitivity to selectivity, we benchmarked the parsing runtime of Sparser + {RapidJSON, Mison} on a synthetic query from the Twitter dataset, and varied the selectivity of the query from 0.01% to 100%.

Figure 4.13 shows the results, comparing Sparser's parsing time against RapidJSON and Mison at various filter selectivities. As the selectivity increases, the benefits of Sparser diminish. However, Sparser still outperforms both parsers by rejecting some records and adaptively tuning its cascade to choose fewer filters as the selectivity increases. In the worst case, when all the data is selected, Sparser always calls the downstream parser, resulting in no speedup.

Binary Formats: Avro and Parquet. In addition to human-readable formats such as JSON, many big data workloads operate over record-oriented binary formats such as Avro [14], or columnar binary formats such as Parquet [160]. Both of these formats are optimized to reduce storage and minimize the overhead of parsing data when loading it into memory. To evaluate Sparser's effectiveness on these formats, we converted the Twitter

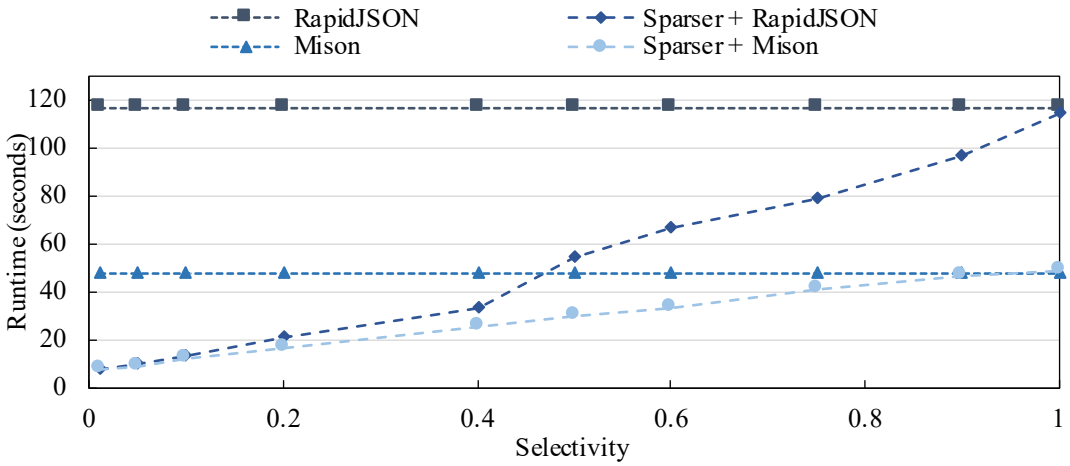
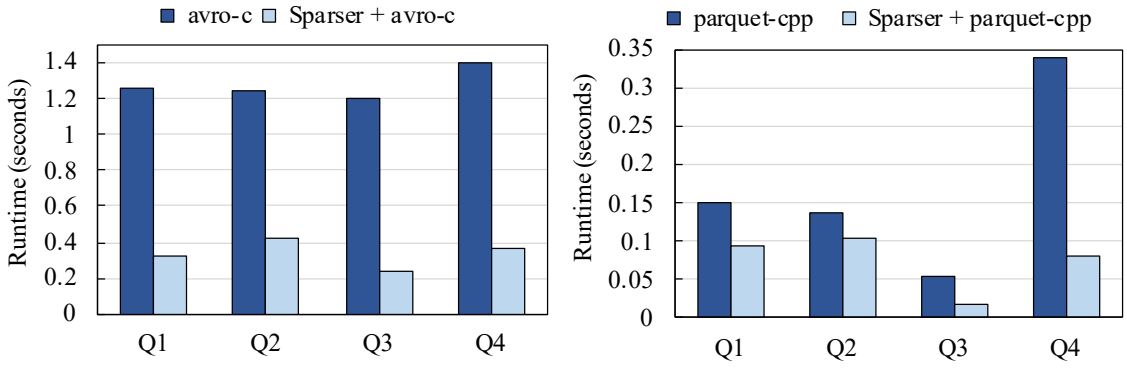


Figure 4.13: Selectivity vs. parsing time for parsing Twitter data.

dataset (originally in JSON) to both Avro and Parquet files using Spark, and benchmarked the four Twitter queries on each format on a single node. Figure 4.14a summarizes the results for parsing Avro: compared to `avro-c`, an optimized C parser, Sparser's raw filtering reduces the end-to-end query time by up to 5×. For Parquet (Figure 4.14b), Sparser improves the parsing time of the `parquet-cpp` library by up to 4.3× across the four queries.

In Avro, data is organized into blocks that contain records, where each record is stored as a sequence of fields without delimiters, and each variable-length field is prefixed with its length encoded as a variable-length integer [15]. Avro also prefixes each *block* of records with its corresponding byte length. Therefore, to parse and filter Avro data, a parser must find each field one at a time to traverse through each record, and check the relevant fields in each record. However, with Sparser, we can skip entire blocks of records if the RFs do not match anywhere in the block. If the RFs do match, Sparser can skip checking the fields of every record in the block, and only check the fields in records where the RFs matched.

In Parquet, Sparser uses a similar strategy, but adapts it to Parquet's columnar format: we seek to the portions of the file that contain the columns of interest, search over those columns, and then seek to the matches within each column. Columns in Parquet are split across Row Groups [161], and we can seek to the Row Group that contains our potential match. Within each Row Group of a given column, the column data is stored across pages, which are typically 8KB each. Within each data page, the column values are stored using



(a) Twitter queries on Avro. **(b)** Twitter queries on Parquet.

Figure 4.14: Parsing time for Avro and Parquet data on the four Twitter queries with and without Sparser.

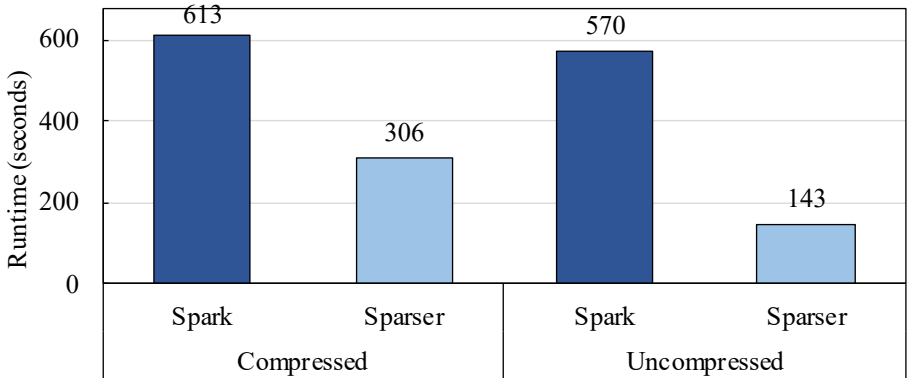


Figure 4.15: Sparser's runtime on Twitter Q1 on an uncompressed vs. `gzip`'d file, benchmarked in Spark on a single node.

the same format found in Avro (i.e., length in bytes, followed by value), which therefore requires the same sort of incremental traversal over the values. Sparser's speedups for binary formats thus come from avoiding full record-by-record value comparisons and by avoiding recursive traversals of nested data.

Speedups on Compressed Data. Data on disk is often compressed and requires running a computationally expensive decompression algorithm, such as `gzip`. While there are some proposed solutions for directly querying compressed data [4], this step is unavoidable for general query processing. To show that parsing is an important factor on compressed data,

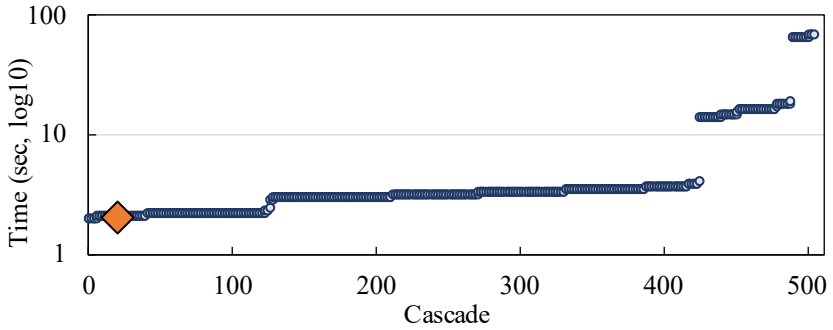


Figure 4.16: The performance of each cascade Sparser considers for Censys Q1, sorted by runtime from left to right. The difference between the best and worst of the 506 cascades is 35×. Sparser (the marked point) does not pick the globally best cascade, since it relies on sampling but is within 10% of the best cascade.

Runtime (ms)	**Average**	**Min**	**Max**
Query Time	5213	3339	11002
Optimizer	3.01	1.85	4.11
Measuring RFs	2.10	1.18	3.09
Measuring Parser	0.63	0.44	0.81
Scoring Cascades	0.28	0.05	0.77
Percent Overhead	0.67%	0.19%	1.18%

Table 4.4: Measurements from the optimizer on Censys queries.

we benchmarked Twitter Q1 on both an uncompressed and `gzip`-compressed version of the JSON data. Figure 4.15 shows the results of the end-to-end runtimes in Spark on a single node: even on compressed data, Sparser improves the end-to-end query runtime by 4× by minimizing the time spent parsing.

4.7.4 Evaluating Sparser's Optimizer

We now examine the optimizer's impact on end-to-end query performance and measure its ability to select RF cascades. We also evaluate the optimizer on multiple microbenchmarks to examine its key design decisions, such as the optimizer's modeling of joint passthrough rates and its periodic resampling strategy for handling skew in the input data.

Impact of the Optimizer. Sparser's optimizer uses a cost function to calculate the expected parsing time a given RF cascade; amongst many candidate cascades, the optimizer will select the one with the lowest expected cost. To show the overall impact of the optimizer, we ran the first Censys query on each cascade considered by the optimizer to study the difference in performance across the different candidates. Figure 4.16 shows the result, where each point represents a cascade, and cascades are sorted by runtime from left to right. Many of the cascades contain RFs that discard few records, which produces little improvement in end-to-end parsing time compared to a standard parser. However, the best cascades substantially reduce overall parsing times. Sparser selects one of the best cascades, showing that the optimizer effectively filters out poor RF combinations, and that Sparser's sampling-based measurement is sufficient to produce a cascade within 10% of the best-performing one.

To evaluate the effect of the bounded cascade depth and possible RFs (§4.5.3), we also ran all nine Censys queries without bounding the combinatorial search. For these queries, we found that only Q6 chose a cascade with depth greater than $D = 4$ (our maximum default depth), and the difference in runtime performance was 1.2%. Across all queries in Table 4.3, Sparser's chosen cascade exhibited performance within 10% of the best cascade. The Bro queries, which did not perform any parsing, were least affected by the choice of cascade (mean performance difference between the best and worse cascade was only 2×), and the Censys queries were the most affected since parsing each record was expensive (mean 33× difference between the best and worst cascade).

Optimizer Overhead. Table 4.4 summarizes the optimizer's average runtime across all nine Censys queries and includes a breakdown of the runtime across each of the optimizer's stages. On average, the optimizer spends 3ms end-to-end, including measuring the parser, measuring the RFs, and searching through cascades. We also measured the effects of the pruning rule that skips cascades with overlapping substrings and found that, on average, 86% of the cascades were not scored. Despite the combinatorial search space, the optimizer's pruning rule and use of bit-parallel operations enable it to account for only up to 1.5% of the total running time on Censys.

	Cascade	Est. Sel.	Real Sel.	Runtime
Sparser	`"teln"` → `"30722"`	0.010%	0.031%	2.221s
Naive	`"teln"` → `"p23"`	0.090%	2.997%	4.454s

Table 4.5: Sparser's optimizer vs. a naive optimizer that assumes RFs are independent of one another.

Interdependence of RFs. The optimizer uses a bitmap-based data structure to store the passthrough rates of each individual RF, allowing it to quickly compute the joint passthrough rates of RF combinations. However, a strawman optimizer could also assume that RFs are independent of one another, and use only the individual passthrough rates to evaluate candidate cascades. To show the impact of capturing the correlations between RFs, we compared the performance of this strawman against Sparser's optimizer on Censys Q1. (For demonstration purposes in this experiment, we substitute `asn = 9318` with `asn = 30722`, a common value in the data.)

Table 4.5 summarizes the results of this experiment. Because the RF that searches for `"30722"` has a high passthrough rate (9.83%), the strawman optimizer instead searches for two tokens with smaller passthrough rates (3% each): `"teln"` (a substring of `"telnet"`) and `"p23"`. However, searching for both tokens adds marginal benefit compared to searching for only one of them—if the token `"teln"` is present, it is almost always accompanied by `"p23"` as well. Sparser's optimizer captures the co-occurrence rate of the two terms and instead searches for `"teln"` and `"30722"`. Although `"30722"` does appear frequently throughout the input on its own, it occurs much less frequently with `"teln"`. As a result, the cascade chosen by Sparser's optimizer is 2× faster than the naive optimizer's cascade.

Key-Value RF. To measure the utility of the key-value RF, we benchmarked Sparser both with and without the key-value RF on a synthetic query over the Twitter dataset. The query finds all tweets with `favorited = true` and has a small selectivity—only 0.002%. Our results showed that without the key-value RF, Sparser fails to outperform the standard parsers, since almost every record contains the terms `"favorited"` and `"true"`. The key-value RF associates both terms together when searching through the raw bytestream, thus enabling a 22× speedup.

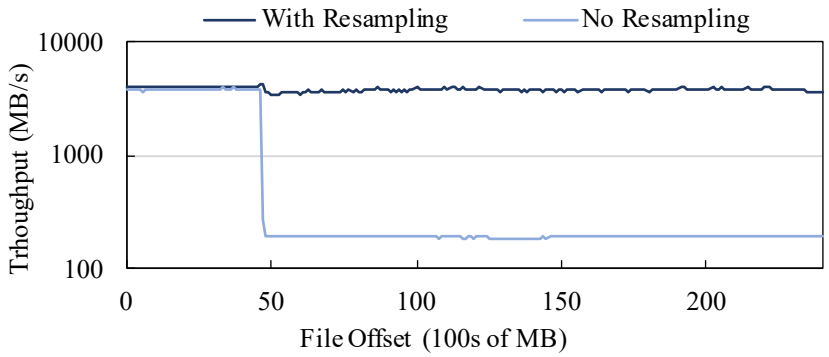


Figure 4.17: Resampling allows Sparser to adapt to changing distributions in the data.

Periodic Resampling. To study the impact of periodic resampling in Sparser's optimizer, we examine the first Twitter query from Table 4.3, which searches for tweets mentioning `"Donald Trump"` on a particular date. Because the tweets were collected as a stream, the `date` field has high temporal locality in the input file—the `date LIKE '%Sep 13%'` predicate selects all the data in some range, but none in the rest. We benchmarked Sparser both with and without its resampling step on this query, and Figure 4.17 shows the result. During the initial sampling, Sparser finds that the `date` predicate is highly selective and includes a substring RF based on `"Sep 13"`. However, in the range where the date does match, the RF no longer remains selective. With periodic resampling, Sparser detects this change and recalibrates its RF cascade to search for a substring of `"Donald Trump"`, rather than a substring of the date. By including this step in the optimizer, Sparser's parsing throughput over the entire input file is 25× faster than it would be otherwise.

4.8 Related Work

Processing Raw Data. Many researchers have proposed query engines over raw data formats. NoDB [7] proposes building indices incrementally over raw data to accelerate access to specific fields. Alagiannis et al. [8] and others [86] consider storage layouts and access patterns for query processing over raw data, and examine how to adapt to workloads online. ViDa introduces JIT-compiled access paths for adapting queries to underlying raw data formats [101–103]. Slalom [150] monitors access patterns to build indices for fast

in-situ data access. SCANRAW uses parallelism to mask in-situ data access times via pipelining [40, 41], while Abouzied et al. [3] propose masking load times using MapReduce jobs. While these approaches propose full query engines over raw data, raw filtering focuses on the problem of filtering and loading it as quickly as possible using format-agnostic RFs and an optimizer. Existing raw processing systems can thus use raw filtering in a complementary manner to filter before downstream processing.

Parsers for Semi-Structured Data. For JSON parsing, the Mison JSON parser [118] is the closest to Sparser in that it takes both filtering expressions to apply to the data and a set of output fields to project as part of its API. Mison always begins by building a structural index using SIMD and bit-parallel operators. The index finds special JSON characters such as colons and brackets to create a mapping from byte offset to field offset. Mison then builds another data structure called a pattern tree to speculatively jump to the desired field position using this structural index, and then applies predicates to the retrieved fields. We showed in §4.7 that just building the structural index in Mison is slower than rejecting RFs with Sparser on selective workloads. In addition, because Mison searches for format-specific delimiters to construct its index, its techniques are not applicable to binary formats that eschew delimiters, such as Avro and Parquet. Sparser is designed to work across both textual and binary formats, and speeds up queries across both. Nevertheless, since Sparser only filters data and optimized parsers [96, 178] extract values from data quickly, the approaches are complementary.

For XML, many approaches used optimized automata to parse and filter XML efficiently [34, 60, 61, 82]. In contrast, this work relies on SIMD instructions rather than automata to leverage data-parallelism in modern hardware. Similar to Mison, Parabix [33] uses SIMD instructions to parse XML, and Teubner et al. [205] and Moussalli et al. [135,136] devise algorithms to leverage data-parallelism on GPUs and FPGAs to accelerate XML filtering and parsing. These systems still extract structural information about the format and, like with optimized JSON parsers, necessarily spend more time than an RF-based search for filtering data. Existing work on fast XML parsing is again complementary with raw filtering, because these systems can use raw filtering to filter data efficiently before parsing.

Predicate Ordering. Sparser's optimizer reorders predicates to optimize overall runtime and is inspired by a long lineage of work on predicate ordering in database systems. Babu et al. [18] propose a way to order conjunctive commutative filters to minimize runtime overhead by adaptively measuring selectivities and considering correlations across filters. The algorithms incur runtime overhead while filtering when accounting for correlations and explores the tradeoffs among ordering quality, decreased overhead, and algorithm convergence. Raw filtering instead uses a new SIMD-enabled optimizer to find an optimal depth-D ordering based on sampled selectivity estimates while always considering filter correlations, and also supports disjunctions of predicates. Scheufele et al. [186] propose an algorithm for optimal selection and join orderings but only consider the cost of individual predicates. Ma et al. [123] similarly order predicates using only their individual costs and selectivities, while Sparser considers correlations among the predicates. Vectorwise [175] uses *micro-adaptivitiy* to dynamically tune query plans: Sparser uses a similar resampling-based approach to tune its cascade dynamically in order to avoid a computationally expensive parse. Lastly, Sparser's approach of combining multiple RFs into an RF cascade is inspired by previous work in computer vision, most notably the Viola-Jones object detector [215]. In Viola-Jones, a cascade is a sequence of increasingly accurate but expensive classifiers; if any classifier is confident about the output, the cascade short-circuits evaluation, improving execution speed. In Sparser, an RF cascade is a binary tree, and the ordering of RFs in the tree is determined by their execution costs and joint passthrough rates.

Fast Substring Search. String search algorithms are used in network and security applications such as intrusion detection. DFC [42] is a recent algorithm for accelerating multi-pattern string search using small, cache-friendly data structures. Other work [199] accelerates multi-pattern string search using vector instructions or other optimizations [143, 209]. These search algorithms, however, are primarily designed for settings with thousands of signatures, while Sparser focuses on quickly rejecting records that do not match a small number of filters, allowing it to work effectively with a sequence of simple tests. Sparser also uses an optimizer to choose an RF cascade based on the input data.

4.9 Summary

Raw filtering that accelerates one of the most expensive steps in data analytics applications—parsing unstructured or semi-structured data—by rejecting records that do not match a query without parsing them. Although this work targets a different domain from our Weld and split annotation interfaces, it still leverages the idea of *algebraic properties* to enable efficient composition. With raw filters (RFs), we exploit the property that "summing" multiple raw filters leads to a more aggressive filter that still only yields false positives. We implement raw filtering in Sparser, which has two key components: a set of fast, SIMD-based RFs, and an optimizer to efficiently select an RF cascade at runtime to balance the false positive rate with the runtime overhead of applying RFs. Sparser accelerates existing high-performance parsers for semi-structured formats by 22× and provides up to an order-of-magnitude speedup on real-world analytics tasks, including Spark analytics queries and log mining.

Chapter 5

Conclusion

This dissertation has argued that rethinking the interfaces for software composition can better utilize the most precious resources of modern hardware. By exposing new *algebraic properties* of software APIs in an interface, underlying systems can leverage these properties to accelerate applications that compose software by an order of magnitude.

The raw filtering interface shows that redefining interfaces between specific software components—in this case, a query engine and a data parser—can lead to substantial speedups. By exposing data filters in an upstream query as a composable "algebra", raw filtering generates approximations of filters from an upstream query to discard raw bytes before parsing them with a full data parser. This improves performance by reducing cycles spent parsing data that would later be discarded by the query.

Weld and split annotations (SAs) both propose systems that use a new interface—a functional intermediate representation (IR) and black-box function annotations respectively—to optimize *data movement* and enable parallelization underneath even existing library APIs. Both systems use their interface to enable algebraic transformations of their input programs: Weld did this via source-to-source IR transformations, while SAs defined a set of rules over its split types to determine pipeline compatibility.

Despite their similar goals, Weld and SAs represent two extreme design points. With Weld, developers must re-implement libraries from scratch to enable cross-function optimizations beyond just data pipelining. Weld can JIT machine code, apply compiler optimizations such as common subexpression elimination, and even target new kinds of hardware, granted

a compiler backend exists for it. In theory, this allows Weld to achieve performance closer to the "roofline" of a particular application and hardware platform, at the cost being a nearly clean-slate solution. In contrast, the split annotations (SAs) interface takes a more pragmatic approach at the expense of foregoing compiler-style optimization, focusing only on automatic parallelization and data pipelining to optimize data movement. A study of several data science workloads in Weld showed that these optimizations are the most important ones in applications that compose independently written libraries. SAs can in many cases match compilers by applying just these optimizations, with a fraction of the effort on the part of the library developer. Nevertheless, SAs rely on existing library code to be already-optimized, and do not support targeting hardware accelerators.

Is there a middle-ground when designing a new interface for software composition on modern hardware? In the remainder of this chapter, we summarize some lessons that we learned from the work in this dissertation, discuss impact, and discuss future directions.

5.1 Lessons Learned

Optimizing individual software components is no longer enough. In many of the applications and workloads we benchmarked, each individual component (e.g., each function call to a library) was already highly-optimized. For example, NumPy and Pandas, two libraries that we used heavily in our benchmarks, each implement core operators in C or compiled Cython. The performance of any individual function is very close to an equivalent implementation in a low-level programming language such as C. Nevertheless, under composition, the end-to-end program sometimes performed an order-of-magnitude worse than an end-to-end program that was hand-optimized in a low-level programming language (§1).

This observation parallels one that was made for distributed systems. Many such systems are bottlenecked by network or disk I/O [153], while our work shows that applications running on parallel hardware are similarly bottlenecked by data movement between the CPU and memory. Just as researchers have spent effort in mitigating the I/O bottleneck (e.g., as Spark did for MapReduce with better abstractions for data sharing [224]), we must design new abstractions to better use the resources of parallel hardware.

The value of exposing existing properties of software to enable optimizations. A major theme of this dissertation is exposing algebraic properties to enable systems to create new optimization opportunities. We found this idea to be powerful not only when building new software libraries, but also when trying to optimize *existing* software. For example, split annotations describe properties intrinsic to existing functions, and then leverage these properties in a runtime to orchestrate *how* functions are called. Weld uses its loops and builders to define how different data-parallel operators can be fused. Finally, raw filtering exploit the property that filters can be composed to decrease the false positive rate to construct an efficient filtering cascade. We believe that the idea of exposing properties of existing APIs has use cases in other areas of computing as well, e.g., by by introducing new heuristics for scheduling decisions or adding constraints that can enable new forms of parallelism.

Careful measurement is the best way to unveil new bottlenecks. An interesting takeaway we had during this dissertation work was that careful performance analysis will reveal new, important bottlenecks that should be optimized first. For example, when we started the Weld project, the vision was to allow data-parallel workloads to achieve roofline performance via a JIT compiler and runtime that treated data-parallelism as a first-class principle in its IR. The runtime supported features such as work-stealing to handle load imbalance and irregular parallelism. In addition, we thought that computational optimizations at the expression level (e.g., CSE and efficient per-expression code generation) also had a large role to play, so we designed an IR that can represent the expressions being computed directly. While many of these themes still exist in Weld today, measurements showed that the main bottleneck in these workloads was neither the computation itself nor artifacts of the design of the parallel runtime, but rather data movement between individual functions.

Similarly, the raw filtering project started because we noticed that much of the execution time in workloads in Pandas or other data science libraries went into data loading. While we initially thought this was due to disk I/O, it turned out that *parsing* unstructured formats such as CSV or JSON (common formats for ad hoc workloads despite their inefficiencies) was the main cause of the slowdown. Further measurement showed similar slowdowns in systems such as Spark: we thus decided to design raw filtering as a way to avoid doing this work when it was not necessary.

5.2 Impact

Since the Weld project was open sourced in 2016, it has received contributions from over 35 developers around the world. Research groups in several universities have adopted Weld as a research platform for building database engines [214], new libraries [21], and exploring new data processing models such as streaming [111] or adaptive query execution [69]. Research with Weld continues at Stanford as well, for building new distributed runtimes to optimizing machine learning serving [108]. A hardware company has also used Weld to target their new vector accelerator [217]. Finally, Alibaba has used Weld [10] for expression code generation in their MapReduce-style data processing framework to achieve state-of-the-art results on the TPC-DS benchmark.

Several recent works have built on top of the split annotations concept, and in particular on the idea of using a type system to express algebraic properties via annotations. Posh [177] uses annotations to understand semantics of shell commands, such as which command line arguments represent files and how the command could be split into parallel invocations, to optimally schedule commands closer to the data they access. Offload annotations (OAs) similarly extend the split types and annotations of SAs to enable bridging libraries targeting new accelerators (e.g., PyTorch [173]) with libraries designed for the CPU (e.g., NumPy [146]). The split types define how values can be marshalled and transferred to an accelerator, and also enable mechanisms such as streaming and paging by splitting data.

Finally, several large technology companies have explored using raw filtering in their data processing stacks. Teradata experimented with it for loading JSON data, and Google employed similar techniques for accelerating processing over protocol buffers [172].

All of the work in this dissertation is open source:

1. **Weld:** https://www.github.com/weld-project/weld
2. **SAs:** https://www.github.com/weld-project/split-annotations
3. **Raw filtering:** https://www.github.com/stanford-futuredata/sparser

5.3 Future Directions

In this section, we propose a few ideas that continue work in redefining composition interfaces for new hardware.

Integrating SAs with a compiler IR. SAs and Weld represent two extreme design points in the space of possible interface designs. One possible way to bridge the two is to allow SAs to optionally specify IR snippets that can be compiled. The SA runtime can then *optionally* choose to compile specific functions, e.g., if they are measured to be CPU-bound at runtime or if adjacent functions both support compilation and loop fusion. This allows libraries to benefit from the data movement optimizations of SAs without code changes, but also enables the benefits of a compiler and IR (optimized execution via code generation, ability to target new hardware, etc.).

One challenge with this approach is managing data layout. Generally, a system based around a compiler IR will require input data to be in a specific format. If data format changes require expensive copies, this can quickly obviate the benefits of an improved composition interface. A possible solution to this is to incorporate data transfer costs into a cost model that decides whether to perform compilation. Yet another option is to design an IR over "abstract data types" that can directly operate over library data formats over a well-defined API, perhaps at the expensive of some performance. These are both interesting directions for future research.

Enabling optimizations beyond data movement on top of existing APIs. Many existing libraries already contain functions that are functionally equivalent to other programs that can be expressed using the library's API. As a simple example, a user of a linear algebra library could equivalently express a series of dot products as a matrix multiply (this can be several orders of magnitude faster due to optimizations such as loop tiling). As another example, a library such as Pandas, which contains many relational-style operators over its DataFrames, could benefit from optimizations that would be available in database systems. We found that reordering filtering operations over DataFrames based on selectivity (a well-known database optimization) can improve performance in some of the Weld benchmarks (§2.9) by up to 2×: this just requires re-ordering existing function calls

in the library. These simple function-to-function optimizations can thus have significant performance implications.

One possible future direction for split annotations is to design an optimizer that is agnostic to any one library, but enables developers to write algebraic rules that implement closed transformations over a library's own interface. This could enable developers to focus on writing small, highly optimized "kernels" that can be transformed into efficiently composed programs via an automatic optimizer.

Interfaces that extend beyond userspace. Is it possible to build a software composition interface that involves the operating system? In fact, some examples of this already exist. In the early days of UNIX, developers realized that many applications required feeding data produced by one process to another process as input. To achieve this, a developer would write intermediate data to a temporary file and then feed this file to the next process. Compositional optimizations such as I/O pipelining, if applied at all, would have to be manually scripted, resulting in code that was cumbersome to read, write, and maintain. The interface designed in response to this pattern—the UNIX pipe—is still ubiquitous today and enables efficient, concise composition of I/O streams between processes.

New operating system abstractions can perhaps recycle these old ideas for new domains. For example, data center workloads strive for low latency across the network stack. A declarative "IR" can be used to specify components that a particular network connection needs, and with some OS support, the IR can be compiled into a specialized, application-specific network path.

As a concrete example, say a particular application needs an authenticated, encrypted, persistent connection to a key-value store. A declarative IR could specify these properties to the network subsystem: the IR can then "compile" a protocol for communication between the client and the key-value store, with optimizations such as, e.g., combining the connection handshake and key exchange for authentication and encryption (similar to QUIC [114]). The OS can provide a `send` and `recv` interface to the application, but use the specialized network stack for communication under the hood. These mechanisms come with their own challenges, (e.g., resource management), but are interesting directions of future work.

Bibliography

[1] 311 Service Requests Dataset. https://github.com/jvns/pandas-cookbook/blob/master/data/311-service-requests.csv, 2019.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, volume 16, pages 265–283, 2016.

[3] Azza Abouzied, Daniel J. Abadi, and Avi Silberschatz. Invisible Loading: Access-driven Data Transfer from Raw Files into Database Systems. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 1–10. ACM, 2013.

[4] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling Queries on Compressed Data. In *NSDI*, pages 337–350, 2015.

[5] Sameer Agarwal, Davies Liu, and Reynold Xin. Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop, 2016.

[6] Alexander Aiken and Edward L. Wimmers. Type Inclusion Constraints and Type Inference. In *FPCA*, volume 93, pages 31–41. Citeseer, 1993.

[7] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 241–252. ACM, 2012.

[8] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2O: A Hands-free Adaptive Store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1103–1114, New York, NY, USA, 2014. ACM.

[9] Alexander Alexandrov, Andreas Kunft, Asterios Katsifodimos, Felix Schüler, Lauritz Thamsen, Odej Kao, Tobias Herb, and Volker Markl. Implicit Parallelism Through Deep Language Embedding. In *SIGMOD '15*, 2015.

[10] (Translated) Alibaba Cloud EMR calculation speed increased by 2.2 times, breaking the world record of the most difficult competition in the field of big data for two consecutive years! https://mp.weixin.qq.com/s/pvbrWOp8rm0hQkddwHpcRA, 2020.

[11] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proc. ACM SIGMOD*, pages 1383–1394, 2015.

[12] Apache Arrow. https://arrow.apache.org/, 2018.

[13] NEC SX-Aurora. https://www.hpc.nec/index.en.

[14] Apache Avro. https://avro.apache.org, 2018.

[15] Apache Avro 1.8.1 Specification. https://avro.apache.org/docs/1.8.1/spec.html, 2018.

[16] Intel AVX2. https://software.intel.com/en-us/node/523876, 2015.

[17] Intel AVX-512. https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html, 2020.

[18] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive Ordering of Pipelined Stream Filters. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 407–418. ACM, 2004.

[19] Peter Bailis, Edward Gan, Samuel Madden, Deepak Narayanan, Kexin Rong, and Sahaana Suri. MacroBase: Prioritizing Attention in Fast Data. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 541–556. ACM, 2017.

[20] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 46(4):41–52, 2011.

[21] Baloo: The Bare-Necessities of Pandas. https://github.com/radujica/baloo, 2019.

[22] Black Scholes Formula. http://gosmej1977.blogspot.com/2013/02/black-and-scholes-formula.html, 2013.

[23] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. Implementation of a Portable Nested Data-parallel Language. *SIGPLAN Not.*, 28(7):102–111, 1993.

[24] Guy E Blelloch and Gary W Sabot. Compiling Collection-oriented Languages onto Massively Parallel Computers. In *Frontiers of Massively Parallel Computation, 1988. Proceedings., 2nd Symposium on the Frontiers of*, pages 575–585. IEEE, 1988.

[25] Robert D. Blumenofe, Christopher F. Joerg, Bradley C. Kurzmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[26] Peter A Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, volume 5, pages 225–237, 2005.

[27] Tim Bray. RFC 8259: The Javascript Object Notation (JSON) Data Interchange Format, 2017.

[28] Bro. https://www.bro.org/, 2017.

[29] Bro Exchange 2013 Malware Analysis. https://github.com/LiamRandall/BroMalware-Exercise, 2017.

[30] Network Forensics with Bro. http://matthias.vallentin.net/slides/bro-nf.pdf, 2011.

[31] Understanding and Examining Bro Logs. https://www.bro.org/bro-workshop-2011/solutions/logs/index.html, 2017.

[32] Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Arvind K. Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. Have Abstraction and Eat Performance, Too: Optimized Heterogeneous Computing with Parallel Patterns. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO 2016, pages 194–205. ACM, 2016.

[33] Robert D. Cameron, Kenneth S. Herdy, and Dan Lin. High Performance XML Parsing Using Parallel Bit Stream Technology. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08, pages 17:222–17:235, New York, NY, USA, 2008. ACM.

[34] K Selçuk Candan, Wang-Pin Hsiung, Songting Chen, Junichi Tatemura, and Divyakant Agrawal. AFilter: Adaptable XML Filtering with Prefix-caching Suffix-clustering. In *Proceedings of the 32nd VLDB*, pages 559–570. VLDB Endowment, 2006.

[35] Censys. Research Access to Censys Data. https://support.censys.io/getting-started/research-access-to-censys-data, 2017.

[36] Writing a Really, Really Fast JSON Parser. https://chadaustin.me/2017/05/writing-a-really-really-fast-json-parser/, 2017.

[37] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: A status report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 10–18, New York, NY, USA, 2007. ACM.

[38] Chambers, Craig and Raniwala, Ashish and Perry, Frances and Adams, Stephen and Henry, Robert R. and Bradshaw, Robert and Weizenbaum, Nathan. FlumeJava: Easy, Efficient Data-Parallel Pipelines. *SIGPLAN Not.*, 45(6):363–375, June 2010.

[39] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An Automated End-to-end Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, pages 578–594, 2018.

[40] Yu Cheng and Florin Rusu. Parallel In-situ Data Processing with Speculative Loading. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 1287–1298. ACM, 2014.

[41] Yu Cheng and Florin Rusu. SCANRAW: A Database Meta-Operator for Parallel In-Situ Processing and Loading. *ACM Trans. Database Syst.*, 40(3):19:1–19:45, October 2015.

[42] Byungkwon Choi, Jongwook Chae, Muhammad Jamshed, Kyoungsoo Park, and Dongsu Han. DFC: Accelerating String Pattern Matching for Network Applications. In *NSDI*, pages 551–565, 2016.

[43] Wireshark Filters. http://www.lovemytool.com/blog/2010/04/top-10-wireshark-filters-by-chris-greer.html, 2017.

[44] Bjarne Stroustrup's C++ Style and Technique FAQ. http://www.stroustrup.com/bs_faq2.html, 2019.

[45] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. An Architecture for Compiling UDF-centric Workflows. *Proc. VLDB Endow.*, 8(12):1466–1477, August 2015.

[46] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B Zdonik. Tupleware: Big data, big analytics, small clusters. In *CIDR*, 2015.

[47] CUDA. http://www.nvidia.com/object/cuda_home_new.html, 2020.

[48] GPU Pro Tip: CUDA 7 Streams Simplify Concurrency. https://devblogs.nvidia.com/gpu-pro-tip-cuda-7-streams-simplify-concurrency/, 2020.

[49] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, and S. Zdonik. A Demonstration of SciDB: A Science-oriented DBMS. *PVLDB*, 2(2):1534–1537, 2009.

[50] CuPy: A Numpy-compatible matrix library accelerated by CUDA. https://cupy.chainer.org/, 2020.

[51] Demo of DBSCAN clustering algorithm. https://scikit-learn.org/stable/auto_examples/cluster/plot_dbscan.html#sphx-glr-auto-examples-cluster-plot-dbscan-py, 2020.

[52] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, and Omar Kanawi. Intel nGraph. https://ai.intel.com/intel-ngraph/, 2018.

[53] Cython. http://cython.org, 2018.

[54] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[55] Dask. https://dask.pydata.org, 2018.

[56] Databricks. https://databricks.com/, 2017.

[57] Pandas Cookbook chapter 7: cleaning up messy data. https://github.com/jvns/pandas-cookbook/, 2019.

[58] Demand paging. https://en.wikipedia.org/wiki/Demand_paging, 2019.

[59] Dependent Type. https://en.wikipedia.org/wiki/Dependent_type, 2019.

[60] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *ACM Transactions on Database Systems (TODS)*, 28(4):467–516, 2003.

[61] Yanlei Diao and Michael J. Franklin. High-performance XML Filtering: An Overview of YFilter. *IEEE Data Eng. Bull.*, 26(1):41–48, 2003.

[62] Edsger W Dijkstra. Notes on Structured Programming. *Structured programming*, pages 1–82, 1969.

[63] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. A Search Engine Backed by Internet-wide Scanning. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 542–553. ACM, 2015.

[64] Mary F Fernandez. Simple and effective link-time optimization of modula-3 programs, 1995.

[65] Flight Delays and Cancellations Dataset. https://www.kaggle.com/usdot/flight-delays/data.

[66] The Apache Foundation. JSON Datasets. https://spark.apache.org/docs/latest/sql-programming-guide.html#json-datasets, 2015.

[67] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and Other Cilk++ Hyperobjects. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 79–90, New York, NY, USA, 2009. ACM.

[68] Andrew Gallant. ripgrep is faster than grep, ag, git grep, ucg, pt, sift. https://blog.burntsushi.net/ripgrep.

[69] Richard Gankema. Loop-Adaptive Execution in Weld. Master's thesis, Universiteit van Amsterdam, 2018.

[70] 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 *Proc. ACM EuroSys*, pages 2:1–2:16, 2015.

[71] Large-Scale Machine Learning for Drug Discovery . https://ai.googleblog.com/2015/03/large-scale-machine-learning-for-drug.html.

[72] Goetz Graefe. *Encapsulation of Parallelism in the Volcano Query Processing System*, volume 19. ACM, 1990.

[73] Clemens Grelck, Karsten Hinckfuß, and Sven-Bodo Scholz. With-Loop fusion for data locality and parallelism. In *Symposium on Implementation and Application of Functional Languages*, pages 178–195. Springer, 2005.

[74] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in llvm. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, page 1, 2011.

[75] Torsten Grust. *Monad Comprehensions: A Versatile Representation for Queries*, pages 288–311. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[76] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[77] TShark Tutorial and Filter Examples. https://hackertarget.com/tshark-tutorial-and-filter-examples/, 2017.

[78] Matrix Multplication is 3X Slower than OpenBLAS. https://github.com/halide/Halide/issues/3499, 2019.

[79] Jess Hamrick. The Demise of for Loops. https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.indexing.html.

[80] F Maxwell Harper and Joseph A Konstan. The Movielens Datasets: History and context. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 5(4):19, 2016.

[81] A beginner's guide to optimizing pandas code for speed. goo.gl/dqwmrG, 2019.

[82] Bingsheng He, Qiong Luo, and Byron Choi. Cache-conscious Automata for XML Filtering. *IEEE Transactions on Knowledge and Data Engineering*, 18(12):1629–1644, 2006.

[83] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proceedings of the VLDB Endowment*, 6(9):709–720, 2013.

[84] Analyze HTTP Requests with TShark. http://kvz.io/blog/2010/05/15/analyze-http-requests-with-tshark/, 2017.

[85] HyPer Web Interface. http://hyper-db.de/interface.html, 2013.

[86] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. Here are my data files. Here are my queries. Where are my results? In *Proceedings of 5th Biennial Conference on Innovative Data Systems Research*, 2011.

[87] ImageMagick. https://imagemagick.org/, 2019.

[88] instagram-filters. https://github.com/acoomans/instagram-filters/tree/master/instagram_filters/filters, 2019.

[89] Gotham. https://github.com/acoomans/instagram-filters/tree/master/instagram_filters/filters/gotham.py, 2019.

[90] Nashville. https://github.com/acoomans/instagram-filters/tree/master/instagram_filters/filters/nashville.py, 2019.

[91] Intel 4004 Datasheet. http://www.applelogic.org/files/4004Data.pdf.

[92] Intel 4040 Datasheet. http://datasheets.chipdb.org/Intel/MCS-40/4040.pdf.

[93] Intel 80C826 Datasheet. https://www.renesas.com/us/en/www/doc/datasheet/80c286.pdf.

[94] Intel 80836 Datasheet. http://pdf.datasheetcatalog.com/datasheet/Intel/mXtuvqv.pdf.

[95] Intel 8086 Datasheet. https://datasheet.octopart.com/QD8087-Intel-datasheet-38976620.pdf.

[96] Jackson. https://github.com/FasterXML/jackson, 2017.

[97] Anthony Jasta. Observability at Twitter: Technical Overview, Part I. https://blog.twitter.com/engineering/en_us/a/2016/observability-at-twitter-technical-overview-part-i.html.

[98] nativejson-benchmark. https://github.com/miloyip/nativejson-benchmark.

[99] A Kagi, James R Goodman, and Doug Burger. Memory Bandwidth Limitations of Future Microprocessors. In *Computer Architecture, 1996 23rd Annual International Symposium on*, pages 78–78. IEEE, 1996.

[100] Karthik Kambatla, Giorgos Kollias, Vipin Kumar, and Ananth Grama. Trends in Big Data Analytics. *Journal of Parallel and Distributed Computing*, 74(7):2561–2573, 2014.

[101] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. Fast Queries over Heterogeneous Data through Engine Customization. *PVLDB*, 9(12):972–983, 2016.

[102] Manos Karpathiotakis, Ioannis Alagiannis, Thomas Heinis, Miguel Branco, and Anastasia Ailamaki. Just-in-time Data Virtualization: Lightweight Data Management with ViDa. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2015.

[103] Manos Karpathiotakis, Miguel Branco, Ioannis Alagiannis, and Anastasia Ailamaki. Adaptive Query Processing on RAW Data. *PVLDB*, 7(12):1119–1130, 2014.

[104] Alfons Kemper, Florian Funke, Holger Pirk, Stefan Manegold, Ulf Leser, Martin Grund, Thomas Neumann, and Martin Kersten. CPU and Cache Efficient Management of Memory-resident Databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 14–25, Washington, DC, USA, 2013. IEEE Computer Society.

[105] Ken Kennedy and Kathryn S McKinley. Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 301–320. Springer, 1993.

[106] John Kessenich. An introduction to SPIR-V. https://www.khronos.org/registry/spir-v/papers/WhitePaper.pdf, 2015.

[107] Helena Kotthaus, Ingo Korb, Michael Engel, and Peter Marwedel. Dynamic page sharing optimization for the R language. *ACM SIGPLAN Notices*, 50(2):79–90, 2015.

[108] Peter Kraft, Daniel Kang, Deepak Narayanan, Shoumik Palkar, Peter Bailis, and Matei Zaharia. Willump: A Statistically-Aware End-to-end Optimizer for Machine Learning Inference. *MLSys 2020*, 2019.

[109] SPT Krishnan and Jose L Ugia Gonzalez. Google BigQuery. In *Building Your Next Big Thing with Google Cloud Platform*, pages 235–253. Springer, 2015.

[110] Kristensen, Mads RB and Lund, Simon AF and Blum, Troels and Skovhede, Kenneth and Vinter, Brian. Bohrium: a Virtual Machine Approach to Portable Parallelism. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 312–321. IEEE, 2014.

[111] Lars Kroll, Klas Segeljakt, Paris Carbone, Christian Schulte, and Seif Haridi. Arc: an ir for batch and stream programming. In *Proceedings of the 17th ACM SIGPLAN International Symposium on Database Programming Languages*, pages 53–58, 2019.

[112] Lindsey Kuper and Ryan R. Newton. LVars: Lattice-based Data Structures for Deterministic Parallelism. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 71–84, New York, NY, USA, 2013. ACM.

[113] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage Using Both Vectorization and Compilation. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 311–326, New York, NY, USA, 2016. ACM.

[114] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The

QUIC Transport Protocol: Design and Internet-scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 183–196, 2017.

[115] Christian Lattner and Vikram Adve. LLVM: a Compilation Framework for Lifelong Program Analysis Transformation. In *International Symposium on Code Generation and Optimization, 2004*, pages 75–86, 2004.

[116] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.

[117] HyoukJoong Lee, Kevin Brown, Arvind Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Implementing Domain-specific Languages for Heterogeneous Parallel Computing. *IEEE Micro*, 31(5):42–53, 2011.

[118] Yinan Li, Nikos R Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. Mison: A Fast JSON Parser for Data Analytics. *PVLDB*, 10(10):1118–1129, 2017.

[119] libpcap. http://www.tcpdump.org, 2017.

[120] Wayne Liu. Python and Pandas Part 4: More Baby Names. http://beyondvalence.blogspot.com/2014/09/python-and-pandas-part-4-more-baby-names.html, 2014.

[121] Loop Tiling. https://en.wikipedia.org/wiki/Loop_tiling, 2020.

[122] Loop Unrolling. https://www.cs.umd.edu/class/fall2001/cmsc411/proj01/proja/loop.html, 2001.

[123] Lu Ma and Grace Kwan-On Au. Techniques for Ordering Predicates in Column Partitioned Databases for Query Optimization, July 3 2014. US Patent App. 13/728,345.

[124] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning Word Vectors for Sentiment Analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

[125] Saeed Maleki, Yaoqing Gao, Maria J. Garzar, Tommy Wong, David A Padua, et al. An evaluation of vectorizing compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 372–382. IEEE, 2011.

[126] Stefan Manegold, Peter Boncz, and Martin L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 191–202. VLDB Endowment, 2002.

[127] John W Mauchly. Preparation of problems for EDVAC-type machines. In *The Origins of Digital Computers*, pages 393–397. Springer, 1982.

[128] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[129] John D McCalpin et al. Memory bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA)*, 1995:19–25, 1995.

[130] Scott McCartney. *ENIAC: The triumphs and tragedies of the world's first computer*. Walker & Company, 1999.

[131] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *SIGMOD*, pages 706–706. ACM, 2006.

[132] Intel Math Kernel Library. https://software.intel.com/en-us/mkl, 2018.

[133] Intel MKL-DNN. https://intel.github.io/mkl-dnn/, 2019.

[134] MNIST. http://yann.lecun.com/exdb/mnist/.

[135] Roger Moussalli, Robert J. Halstead, Mariam Salloum, Walid A Najjar, and Vassilis J Tsotras. Efficient XML Path Filtering Using GPUs. In *ADMS@ VLDB*, pages 9–18. Citeseer, 2011.

[136] Roger Moussalli, Mariam Salloum, Walid Najjar, and Vassilis J Tsotras. Massively Parallel XML Twig Filtering using Dynamic Programming on FPGAs. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 948–959. IEEE, 2011.

[137] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Instant Loading for Main Memory Databases. *PVLDB*, 6(14):1702–1713, 2013.

[138] Stefan C Müller, Gustavo Alonso, Adam Amara, and André Csillaghy. Pydron: Semi-Automatic Parallelization for Multi-Core and the Cloud. In *OSDI*, pages 645–659, 2014.

[139] Derek Gordon Murray, Michael Isard, and Yuan Yu. Steno: Automatic Optimization of Declarative Queries. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 121–131, 2011.

[140] ARM NEON. https://developer.arm.com/technologies/neon.

[141] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB*, 4(9):539–550, 2011.

[142] Robert Nishihara, Philipp Moritz, Stephanie Wang, Alexey Tumanov, William Paul, Johann Schleier-Smith, Richard Liaw, Michael I. Jordan, and Ion Stoica. Real-time machine learning: The missing pieces. *CoRR*, abs/1703.03924, 2017.

[143] Marc Norton. Optimizing Pattern Matching for Intrusion Detection. *Sourcefire, Inc., Columbia, MD*, 2004.

[144] Numba. https://numba.pydata.org, 2018.

[145] Numexpr. https://github.com/pydata/numexpr.

[146] NumPy. http://www.numpy.org/, 2019.

[147] NumPy Array Indexing. https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.indexing.html, 2009.

[148] List of NVIDIA GPUs. https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units.

[149] NYC Taxi Dataset. https://cloud.google.com/bigquery/public-data/nyc-tlc-trips.

[150] Matthaios Olma, Manos Karpathiotakis, Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. Slalom: Coasting through Raw Data via Adaptive Partitioning and Indexing. *PVLDB*, 10(10):1106–1117, 2017.

[151] OpenCV. https://opencv.org/, 2020.

[152] OpenMP. http://openmp.org/wp/.

[153] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, Byung-Gon Chun, and V ICSI. Making Sense of Performance in Data Analytics Frameworks. In *NSDI*, volume 15, pages 293–307, 2015.

[154] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, et al. Evaluating End-to-end Optimization for Data Analytics Applications in Weld. *Proceedings of the VLDB Endowment*, 11(9):1002–1015, 2018.

[155] Shoumik Palkar, James Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, and Matei Zaharia. Weld: A Common Runtime for High Performance Data Analytics. In *Conference on Innovative Data Systems Research (CIDR)*, 2017.

[156] Shoumik Palkar, James J. Thomas, Deepak Narayanan, Anil Shanbhag, Rahul Palamuttam, Holger Pirk, Malte Schwarzkopf, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. Weld: Rethinking the interface between data-intensive applications. *CoRR*, abs/1709.06416, 2017.

[157] Pandas. https://pandas.pydata.org, 2015.

[158] Pandas Cookbook example. http://nbviewer.jupyter.org/github/jvns/pandas-cookbook/blob/v0.1/cookbook/Chapter%207%20-%20Cleaning%20up%20messy%20data.ipynb.

[159] David Lorge Parnas. On the Criteria to be used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[160] Apache Parquet. https://parquet.apache.org.

[161] apache/parquet-format. https://github.com/apache/parquet-format, 2017.

[162] Libpcap File Format. https://wiki.wireshark.org/Development/LibpcapFileFormat, 2017.

[163] PDP-8 Floating-Point System Programmer's Reference Manual. http://bitsavers.informatik.uni-stuttgart.de/pdf/dec/pdp8/software/DEC-08-YQYB-D_PDP-8_Floating-Point_System_Programmers_Reference_Manual_Sep69.pdf.

[164] Programmed Data Processor-1 Manual. https://www.computerhistory.org/pdp-1/_media/pdf/dec.pdp-1.pdp-1_programmed_data_processor-1.1961.102664957.pdf.

[165] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A Fast, Scalable, In-memory Time Series Database. *PVLDB*, 8(12):1816–1827, 2015.

[166] Intel Pentium P5 Processor. http://datasheets.chipdb.org/Intel/x86/Pentium/24199710.PDF.

[167] Benjamin C Pierce and David N. Turner. Local Type Inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

[168] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. Voodoo - a Vector Algebra for Portable Database Performance on Modern Hardware. *Proc. VLDB Endow.*, 9(14):1707–1718, October 2016.

[169] pkeys. http://man7.org/linux/man-pages/man7/pkeys.7.html, 2019.

[170] Chapter 8: Loop Nest Pragma Directives. http://csweb.cs.wfu.edu/~torgerse/Kokua/More_SGI/007-3587-004/sgi_html/ch08.html, 2020.

[171] Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. A Generic Parallel Collection Framework. In *European Conference on Parallel Processing*, pages 136–147. Springer, 2011.

[172] Google Protocol Buffers. https://developers.google.com/protocol-buffers/.

[173] Pytorch. http://pytorch.org, 2019.

[174] Lin Qiao, Vijayshankar Raman, Frederick Reiss, Peter J. Haas, and Guy M Lohman. Main-memory Scan Sharing for Multi-core CPUs. *PVLDB*, 1(1):610–621, 2008.

[175] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. Micro Adaptivity in Vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1231–1242. ACM, 2013.

[176] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a Language and Compiler for Optimizing Parallelism, Locality, and Cecomputation in Image Processing Pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[177] Deepti Raghavan, Sadjad Fouladi, Phil Levis, and Matei Zaharia. POSH: Process-Offload Shell. In *To appear as part of the 2020 USENIX Annual Technical Conference*, page (to appear), 2020.

[178] RapidJSON. https://rapidjson.org, 2017.

[179] NVIDIA RAPIDS. https://developer.nvidia.com/rapids, 2020.

[180] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 693–701. Curran Associates, Inc., 2011.

[181] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[182] Matthew Rocklin. Dask: Parallel computation with Blocked Algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference*. Citeseer, 2015.

[183] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *Proc. ACM SOSP*, pages 49–68. ACM, 2013.

[184] Seref Sagiroglu and Duygu Sinanc. Big Data: A Review. In *Collaboration Technologies and Systems (CTS), 2013 International Conference on*, pages 42–47. IEEE, 2013.

[185] Tao B Schardl, William S Moses, and Charles E Leiserson. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. In *PPoPP*, pages 249–265. ACM, 2017.

[186] Wolfgang Scheufele and Guido Moerkotte. *Optimal Ordering of Selections and Joins in Acyclic Queries with Expensive Predicates*. RWTH, Fachgruppe Informatik, 1996.

[187] Oliver Schoett. *Data abstraction and the Correctness of Modular Programming*. The University of Edinburgh, 1986.

[188] SciPy. https://www.scipy.org/, 2020.

[189] Timos K Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.

[190] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. How to Architect a Query Compiler. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1907–1922, New York, NY, USA, 2016. ACM.

[191] ShallowWater. https://github.com/mrocklin/ShallowWater/, 2019.

[192] SIMD. https://en.wikipedia.org/wiki/SIMD.

[193] scikit-learn. https://scikit-learn.org/stable/index.html, 2020.

[194] spaCy. https://spacy.io/, 2019.

[195] Spark Partitioner. https://spark.apache.org/docs/2.2.0/api/java/org/apache/spark/Partitioner.html, 2014.

[196] Spark SQL Data Sources API: Unified Data Access for the Apache Spark Platform. https://databricks.com/blog/2015/01/09/, 2017.

[197] SQLite. https://sqlite.org/index.html, 2020.

[198] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering*, 12(3):66–73, 2010.

[199] Charalampos Stylianopoulos, Magnus Almgren, Olaf Landsiedel, and Marina Papatriantafilou. Multiple Pattern Matching for Network Security Applications: Acceleration through Vectorization. In *Parallel Processing (ICPP), 2017 46th International Conference on*, pages 472–482. IEEE, 2017.

[200] Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. OptiML: an Implicitly Parallel Domain-specific Language for Machine Learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 609–616, 2011.

[201] Arvind K Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A Compiler Architecture for Performance-oriented Embedded Domain-specific Languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):134, 2014.

[202] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. GraphMat: High Performance Graph Analytics Made Productive. *PVLDB*, 8(11):1214–1225, 2015.

[203] Daniel Tahara, Thaddeus Diamond, and Daniel J Abadi. Sinew: a SQL system for Multi-structured Data. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 815–826. ACM, 2014.

[204] tcpdump. http://www.tcpdump.org, 2017.

[205] Jens Teubner, Louis Woods, and Chongling Nie. XLynx: an FPGA-based XML Filter for Hybrid XQuery Processing. *ACM Transactions on Database Systems (TODS)*, 38(4):23, 2013.

[206] Adding a New Op. https://www.tensorflow.org/guide/extend/op, 2019.

[207] XLA: TensorFlow, Compiled! (TensorFlow Dev Summit 2017). https://www.youtube.com/watch?v=kAOanJczHA0&feature=youtu.be&t=41m46s, 2017.

[208] TShark. https://www.wireshark.org/docs/man-pages/tshark.html, 2017.

[209] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic Memory-efficient String Matching Algorithms for Intrusion Detection. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 4, pages 2628–2639. IEEE, 2004.

[210] How to optimize GEMM on CPU. https://docs.tvm.ai/tutorials/optimize/opt_gemm.html, 2019.

[211] Introduction to Twitter JSON. https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/intro-to-tweet-json, 2017.

[212] Typescript. https://www.typescriptlang.org/, 2019.

[213] Typescript Decorators. https://www.typescriptlang.org/docs/handbook/decorators.html/, 2019.

[214] Mihai Varga, Peter Boncz, and Hannes Mühleisen. Just-in-time compilation in MonetDB with Weld. Master's thesis, Universiteit van Amsterdam, 2018.

[215] Paul Viola and Michael Jones. Rapid Object Detection using a Boosted Cascade of Simple Features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–I. IEEE, 2001.

[216] David Walker. Substructural Type Systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 1. MIT Press, 2005.

[217] Weld for NEC SX-Aurora. https://github.com/sx-aurora-dev/weld, 2020.

[218] Weld Language Overview. https://github.com/weld-project/weld/blob/master/docs/language.md, 2019.

[219] Wm. A. Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[220] TensorFlow XLA. https://www.tensorflow.org/performance/xla/, 2018.

[221] TensorFlow XLA JIT. https://www.tensorflow.org/performance/xla/jit, 2018.

[222] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI*, volume 8, pages 1–14, 2008.

[223] Gina Yuan, Shoumik Palkar, Deepak Narayanan, and Matei Zaharia. Offload Annotations: Bringing Heterogeneous Computing to Existing Libraries and Workloads. In *To appear as part of the 2020 USENIX Annual Technical Conference*, page (to appear), 2020.

[224] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

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

[226] Marcin Zukowski, Mark van de Wiel, and Peter Boncz. Vectorwise: A Vectorized Analytical DBMS. In *2012 IEEE 28th International Conference on Data Engineering (ICDE)*, pages 1349–1350. IEEE, 2012.