Generating Fast Data Planes for Data-Intensive Systems

Shoumik Palkar and Matei Zaharia

Stanford DAWN

Many New Data Systems

We are building many specialized high-performance data systems.

- **Graph Engines**
  PowerGraph, Ligra, Gemini, Galois
- **Serving**
  Parameter Server, TensorFlow Serving, Clipper
- **SQL, Time Series Databases, Stream Engines, *etc*...**

Building these systems w/high performance is a ***huge*** effort

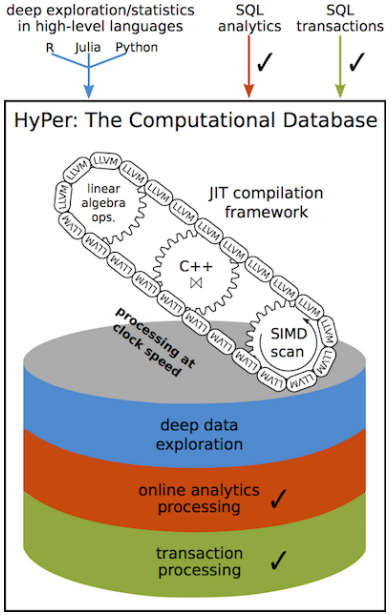
Code Generation for High Performance

Increasingly popular technique for high performance: **code generation**

End-to-end high performance mandates **monolithic** design

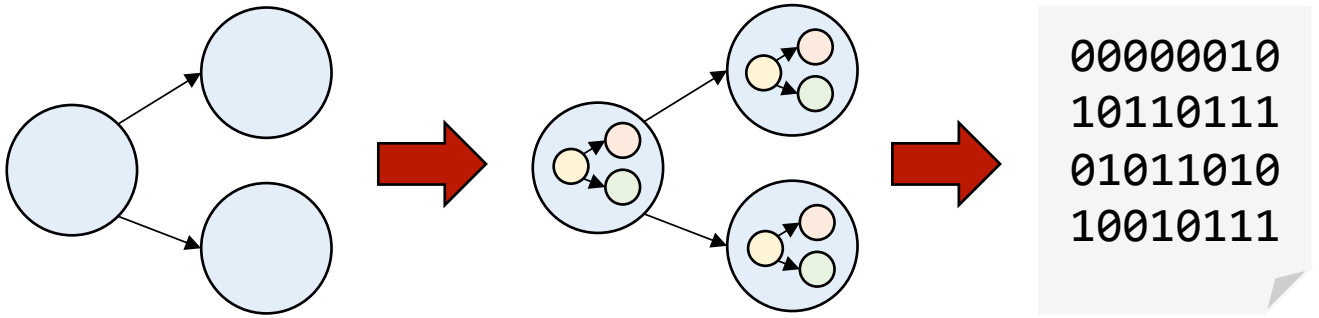
Example: Achieving high performance in HyPer, Spark SQL

Need to thread a compiler through the whole system!



Vision: Interfaces for Code Gen with Guaranteed Optimizations

- Use **composable interfaces** with *guaranteed optimizations* to implement application operators
- generate code at runtime



Decompose Application Operators

Implement using Interfaces which guarantee certain optimizations

Execute compiled, optimized code

CORGIs: Interfaces for Data Systems

Composable, **O**ptimizable **R**untime-**G**enerated **I**nterfac

Runtime-Generated

Optimizations applied at runtime after operator tree is built, and executed via code generation

Building Systems with CORGIs

1. **Decompose system into set of operators**

   Many systems already do this!

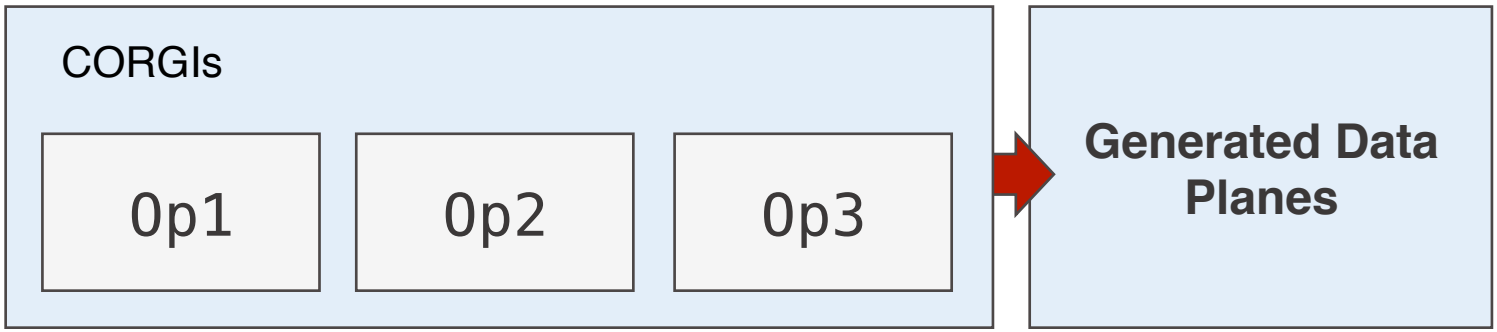
2. **Implement operators using CORGIs**

   Offload performance optimizations to the guarantees of the interface

3. **Code generation via runtime**

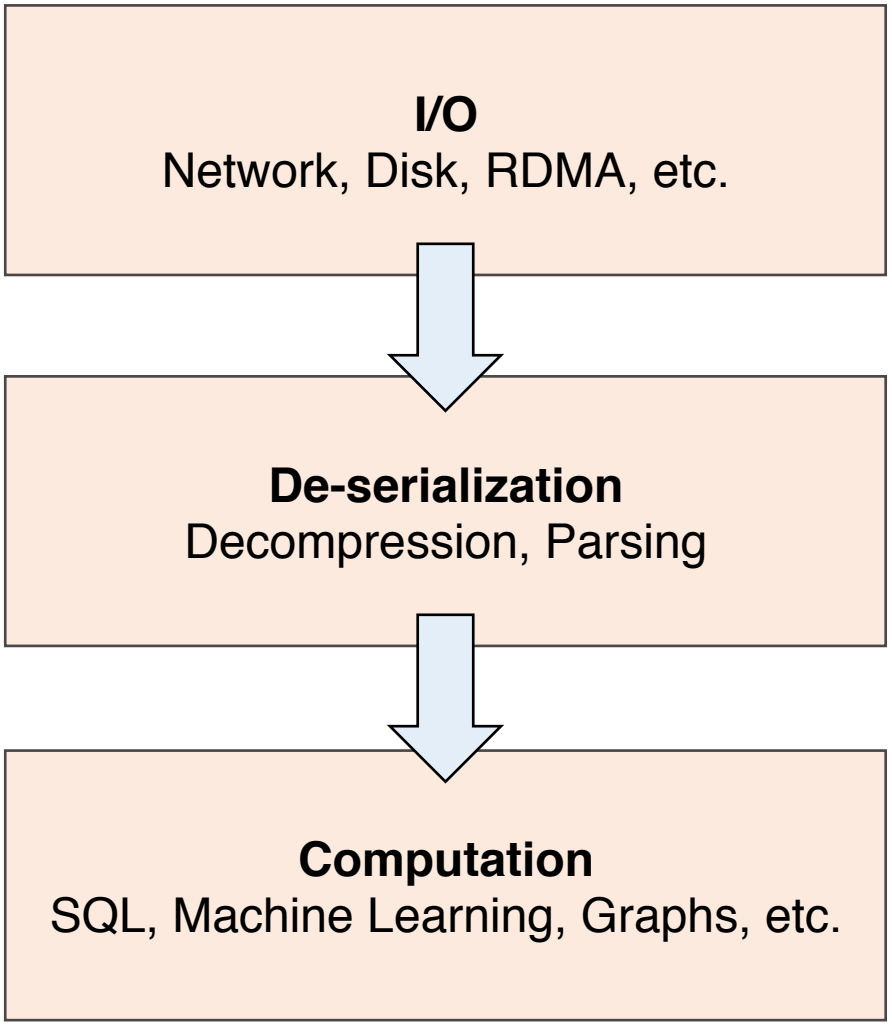
   interface optimizations provide performance

CORGIs



The Key Question:
what exactly *are* these interfaces, and what optimizations can they guarantee?

CORGIs for Data Systems

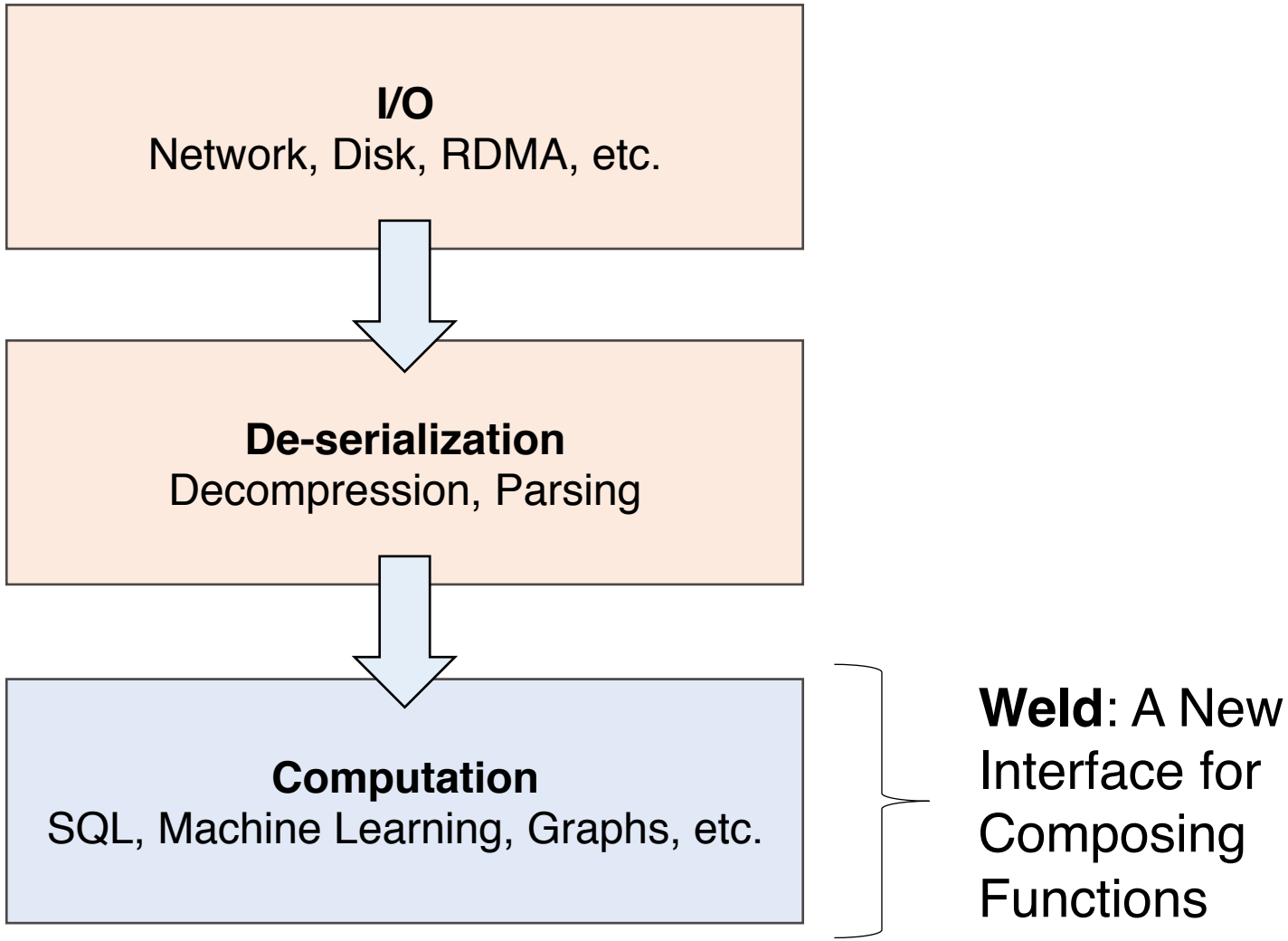


Interfaces for operators which load data from hardware

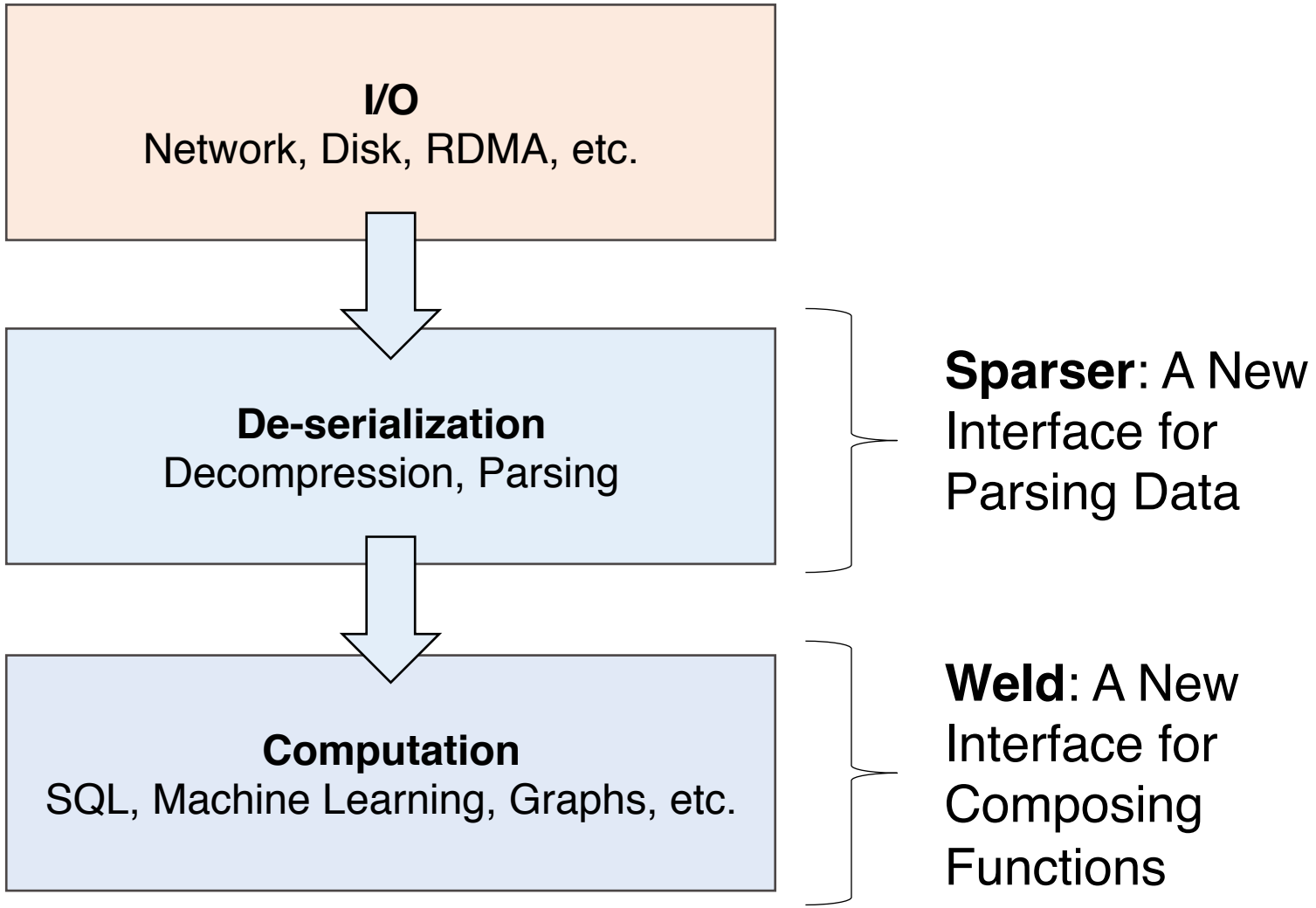
Interfaces for operators which “prepare” data for computation

Interfaces for operators which express general computation

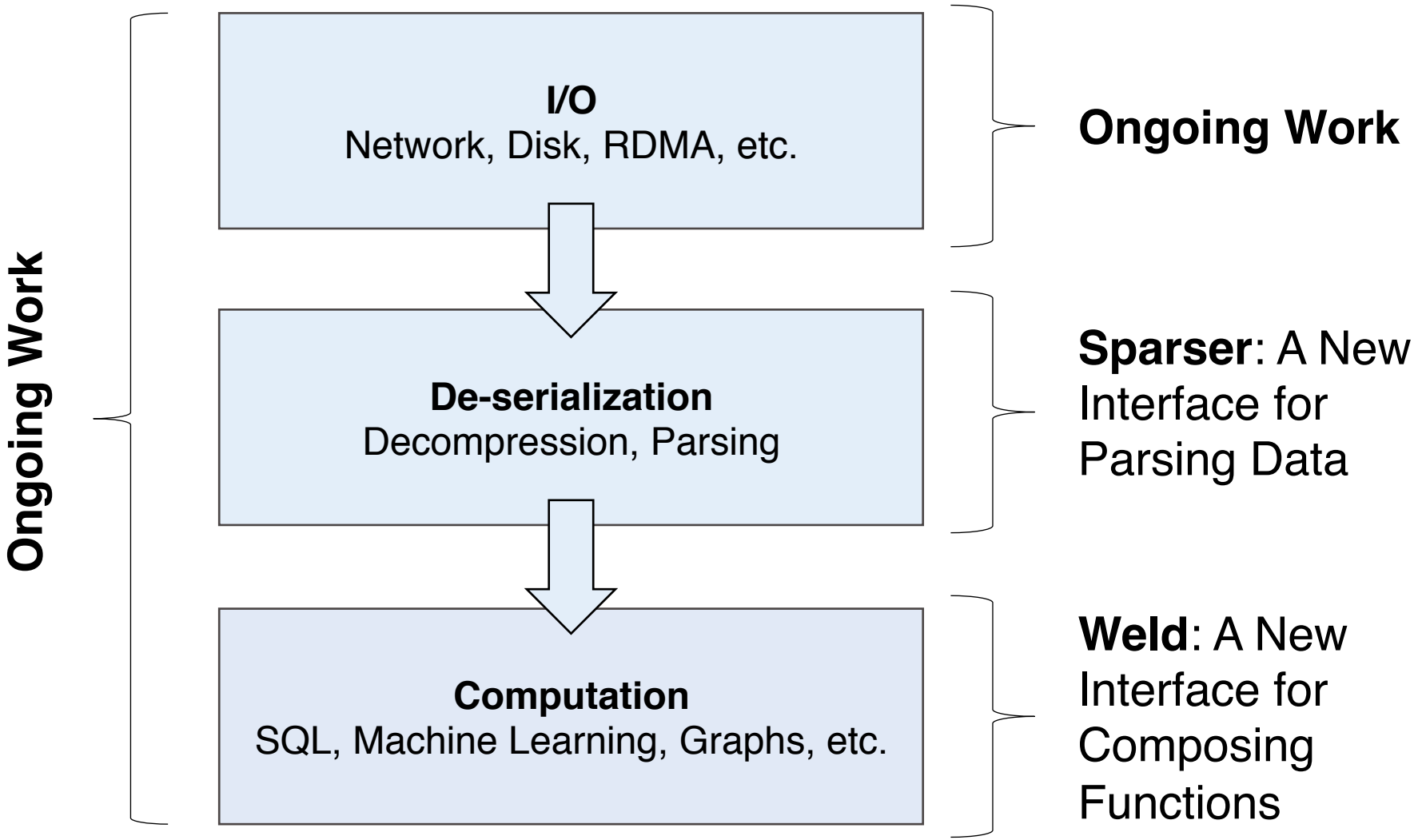
CORGIs for Data Systems



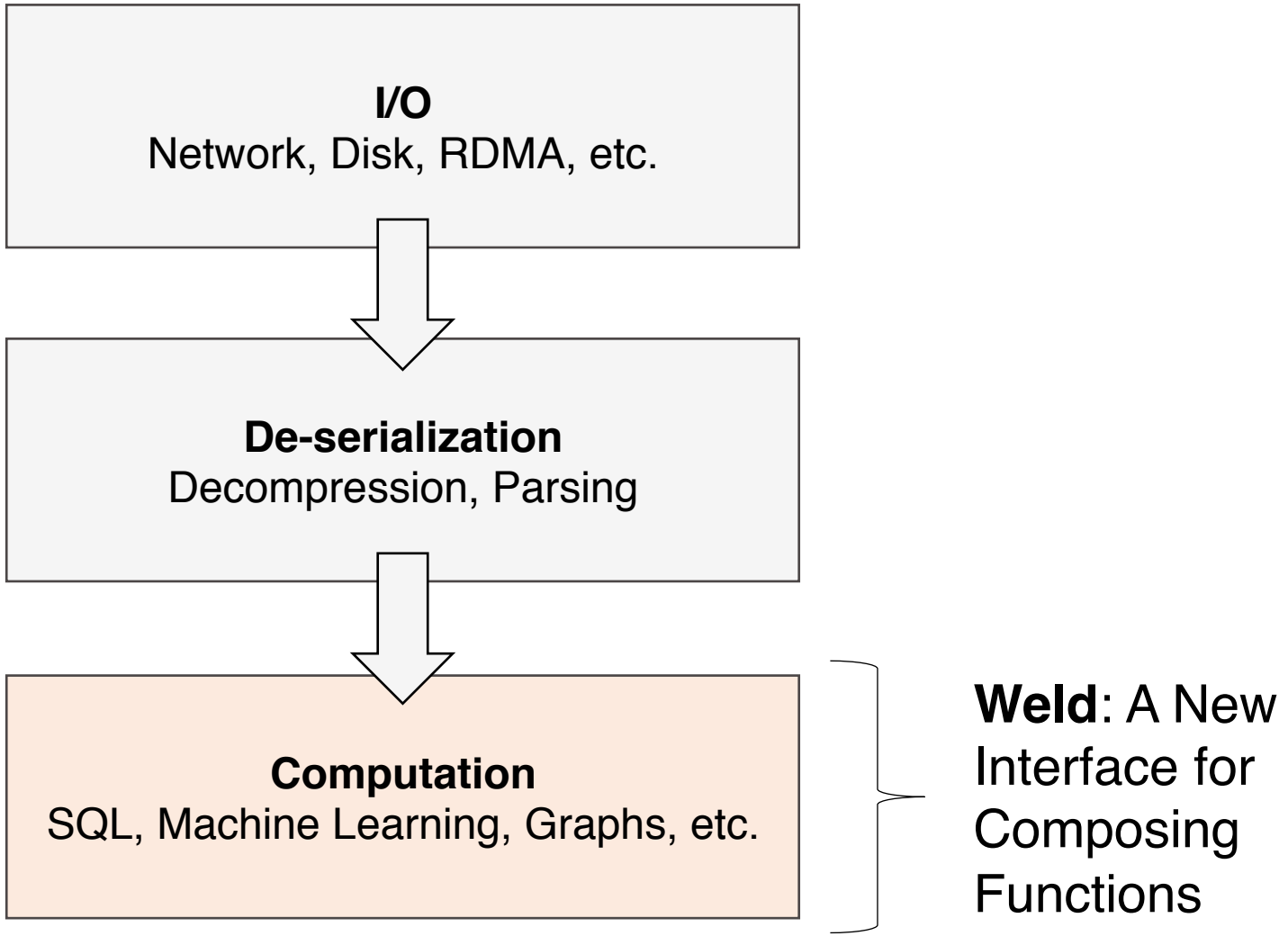
CORGIs for Data Systems



CORGIs for Data Systems



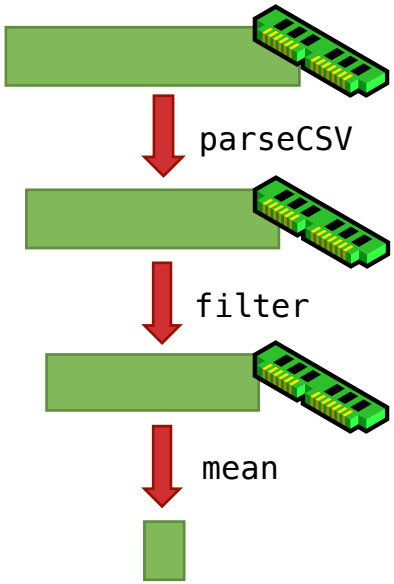
Weld: Optimizing Compute



Composing Functions is Slow

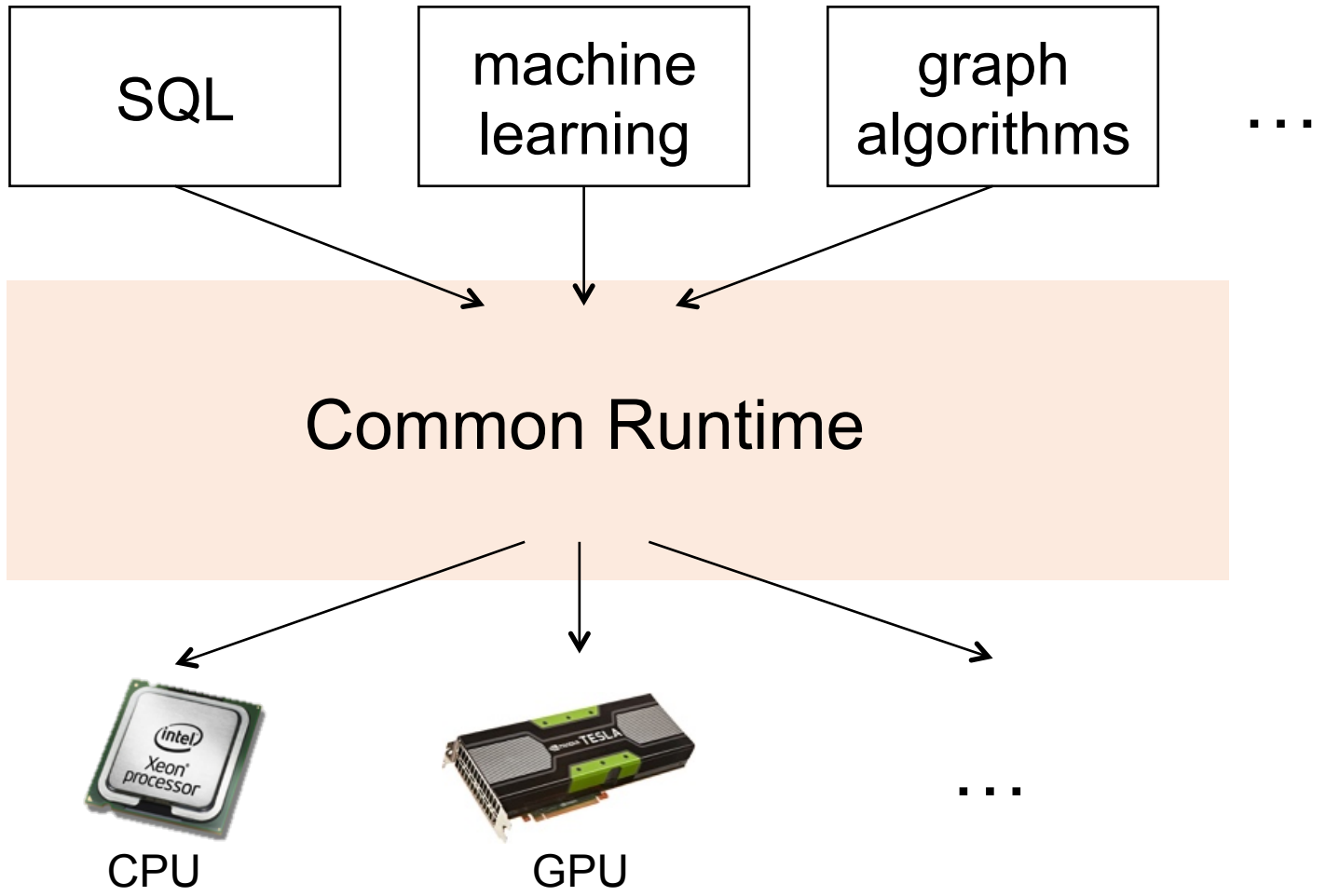
The traditional way of composing functionality in a system, the *function call*, is no longer sufficient.

```
data = parseCSV(string)

filtered = filter(NULL, data)

avg = Mean(filtered)
```

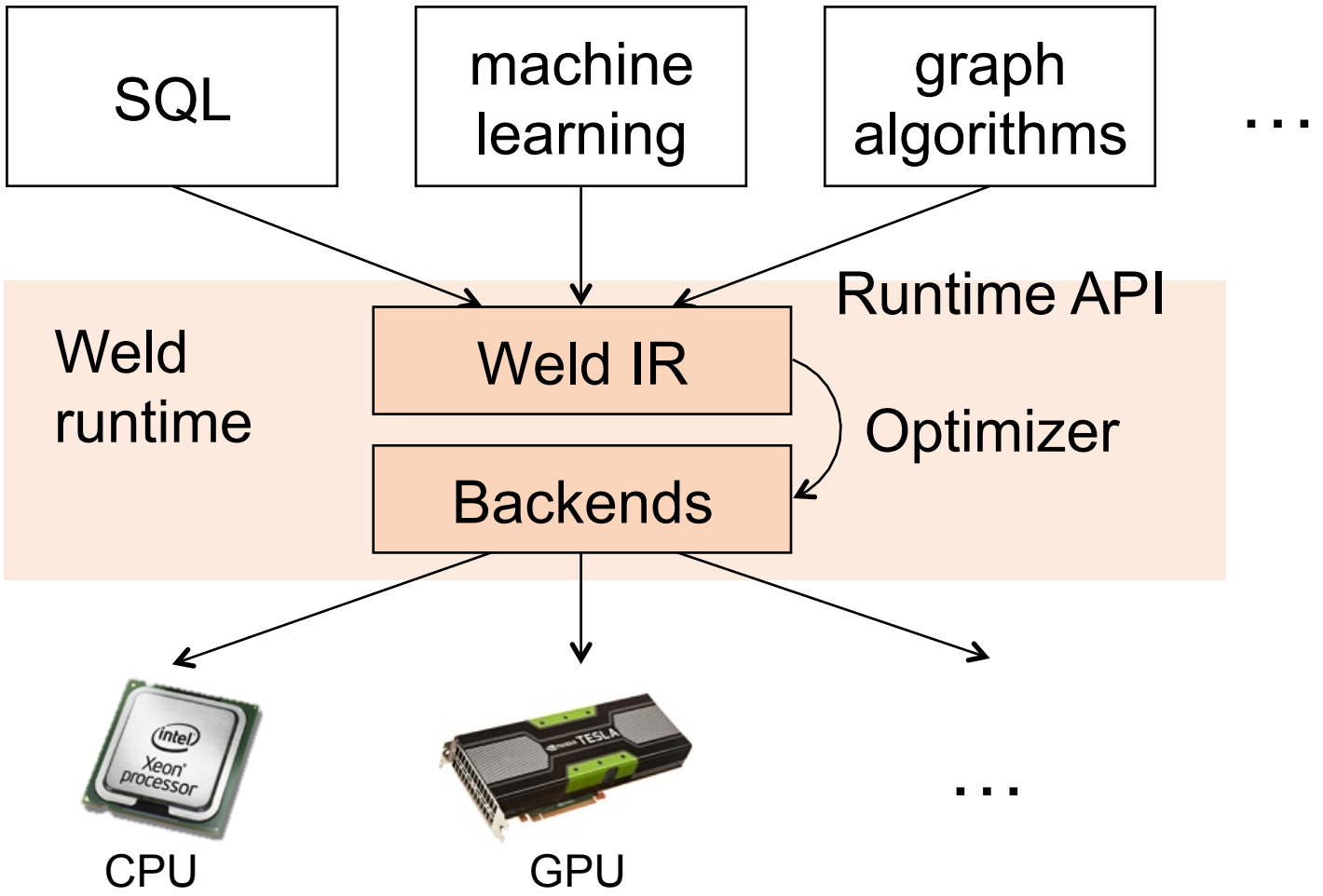


Up to 30x slowdowns in NumPy, Pandas, TensorFlow, etc. compared to an optimized C implementation

Weld Architecture

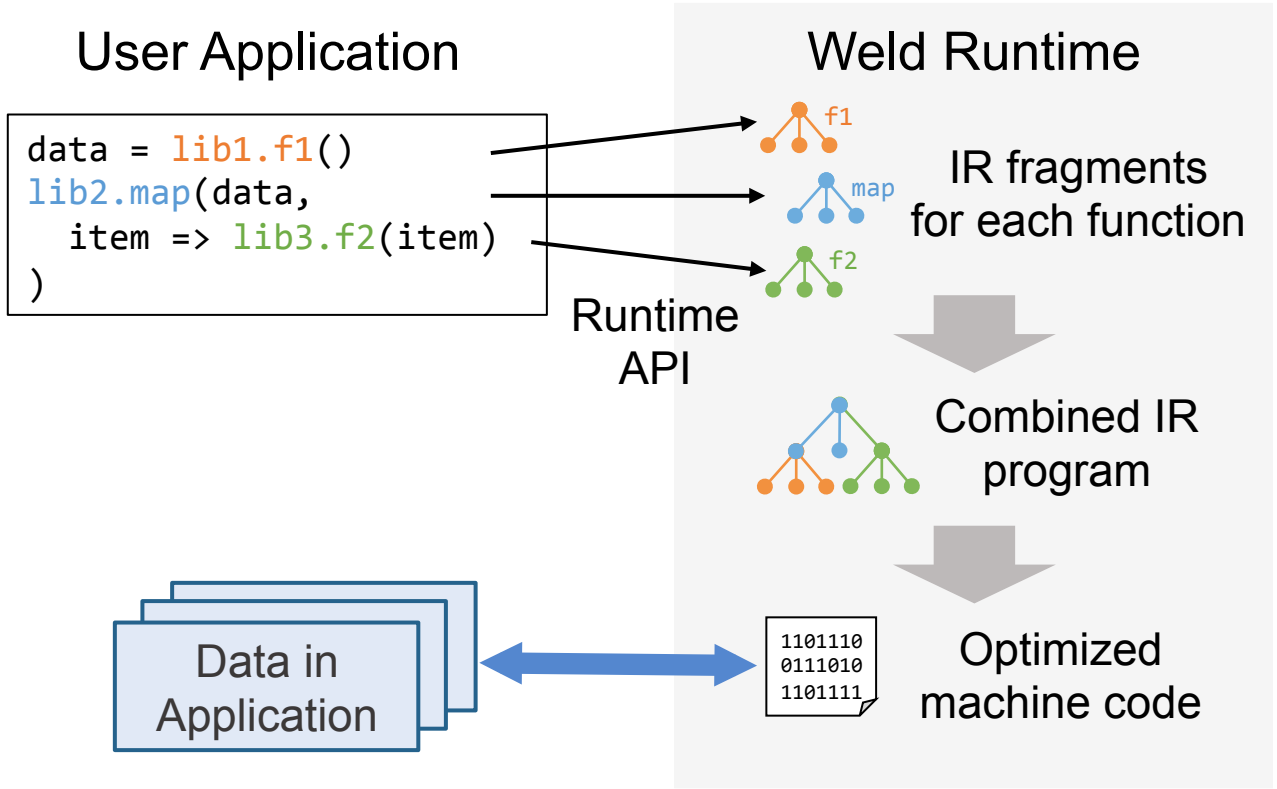


Weld Architecture



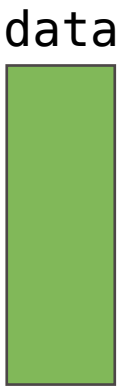
Weld's Runtime API

Uses lazy evaluation to collect work across libraries



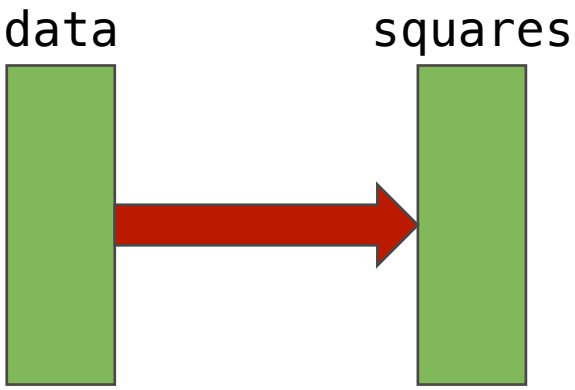
Without Weld

```
import itertools as it
squares = it.map(data, |x| x * x)
sum = sqrt(it.reduce(squares, 0, +))
```



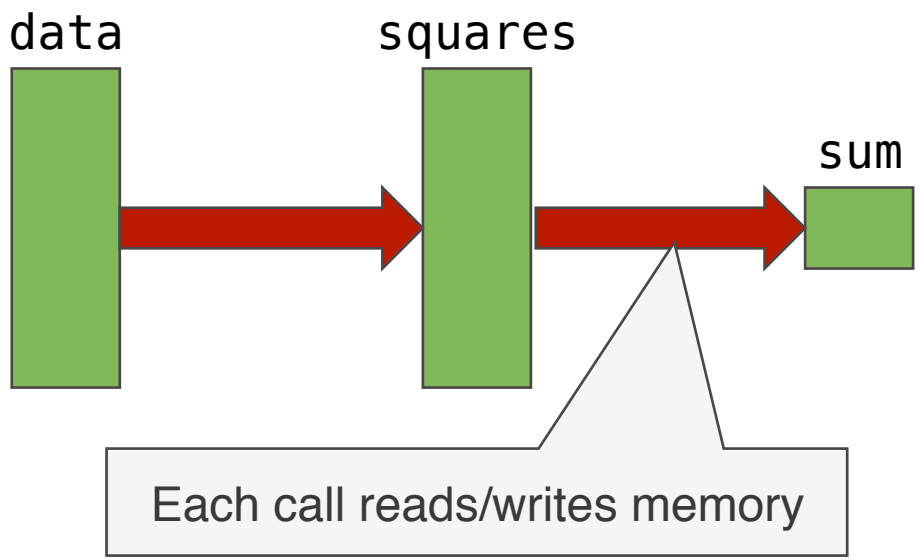
Without Weld

```
import itertools as it
squares = it.map(data, |x| x * x)
sum = sqrt(it.reduce(squares, 0, +))
```



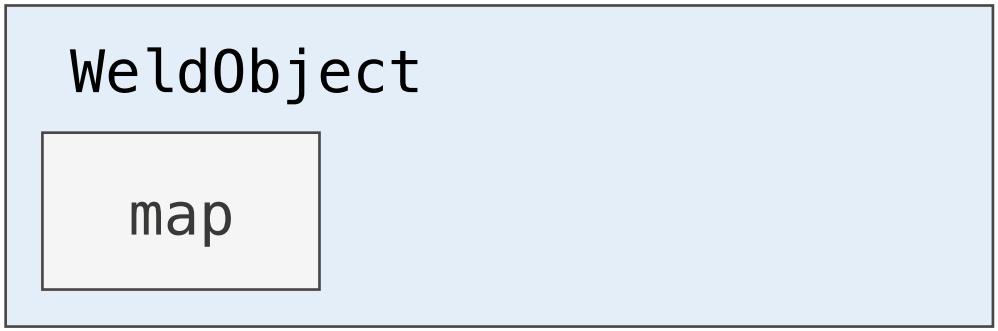
Without Weld

```
import itertools as it
squares = it.map(data, |x| x * x)
sum = sqrt(it.reduce(squares, 0, +))
```



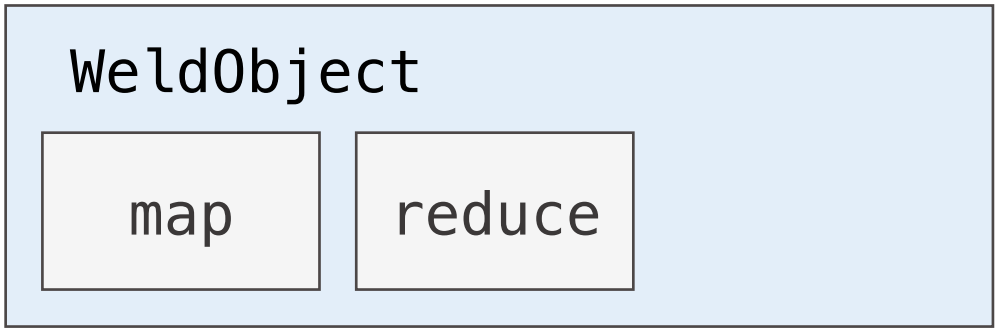
With Weld

```
import itertools as it
squares = it.map(data, |x| x * x)
sum = sqrt(it.reduce(squares, 0, +))
```



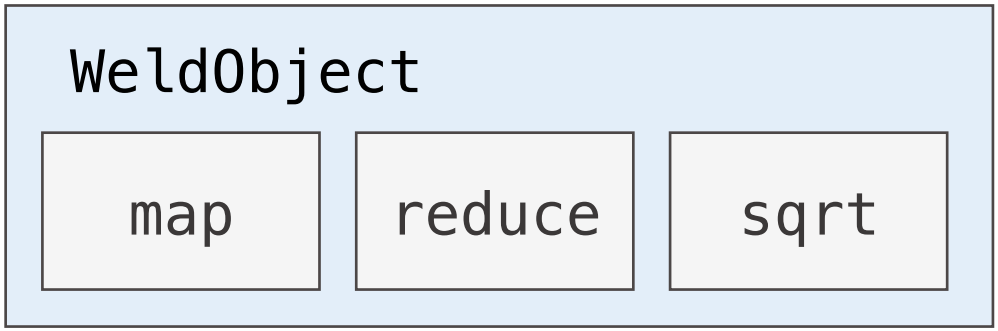
With Weld

```
import itertools as it
squares = it.map(data, |x| x * x)
sum = sqrt(it.reduce(squares, 0, +))
```



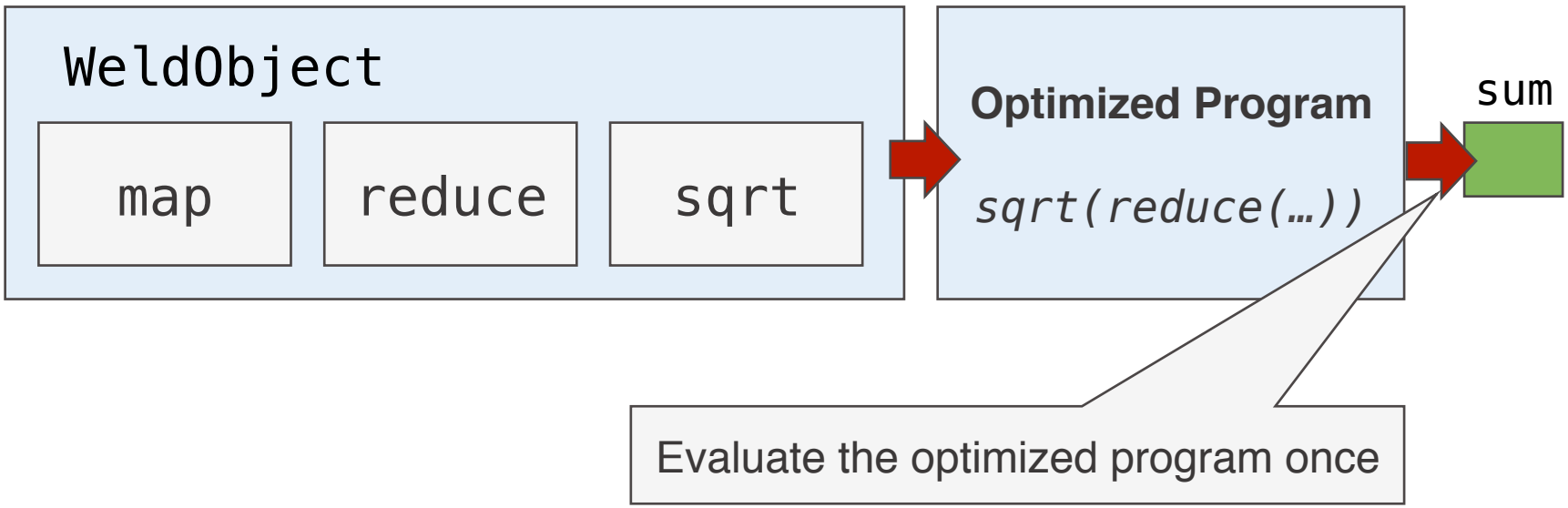
With Weld

```
import itertools as it
squares = it.map(data, |x| x * x)
sum = sqrt(it.reduce(squares, 0, +))
```



With Weld

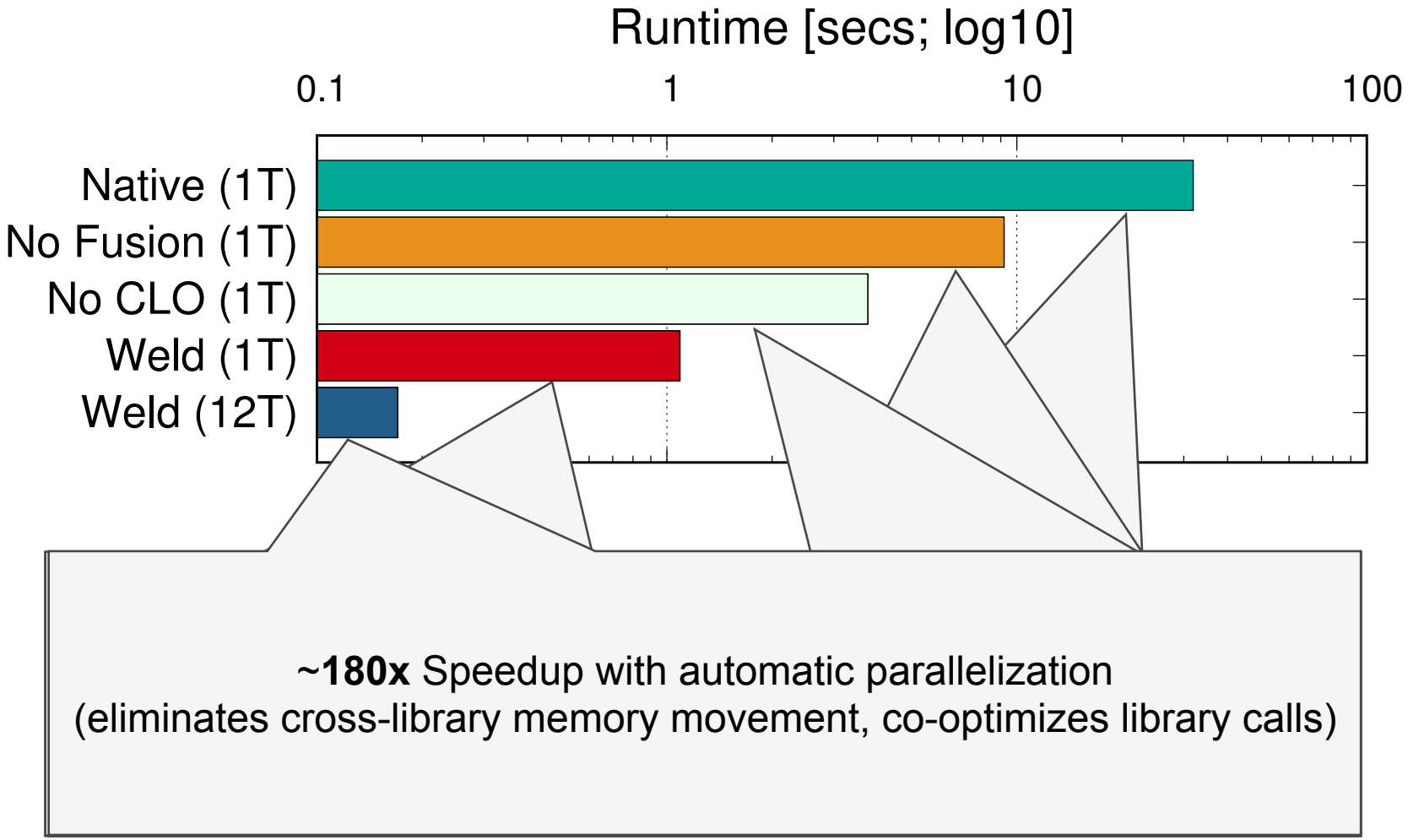
```
import itertools as it
squares = it.map(data, |x| x * x)
sum = sqrt(it.reduce(squares, 0, +))
```



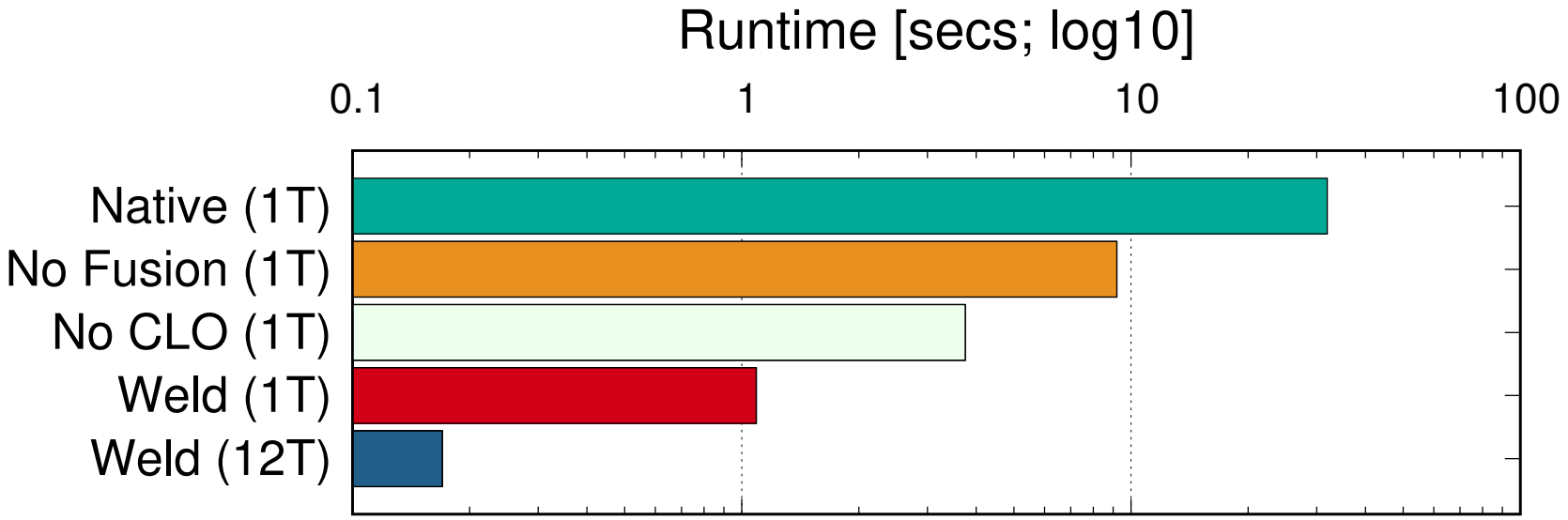
Optimizations in Weld

- **Data movement optimizations**
  - e.g., loop fusion
- **Hardware optimizations**
  - SIMD, predication, automatic parallelization
- **Data Structure optimizations**
  - Hash table implementations, reordering operators

Optimizing Compute with Weld



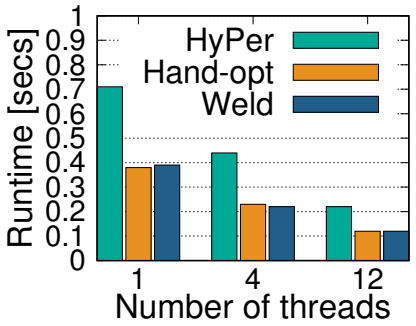
Optimizing Compute with Weld



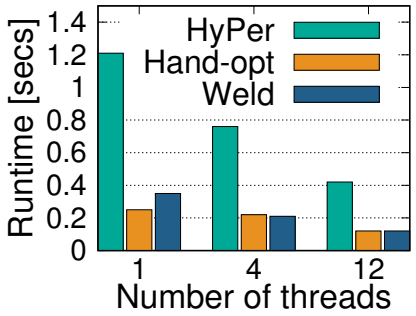
Bare Metal Performance *without* changing user applications

Modularity without sacrificing Performance

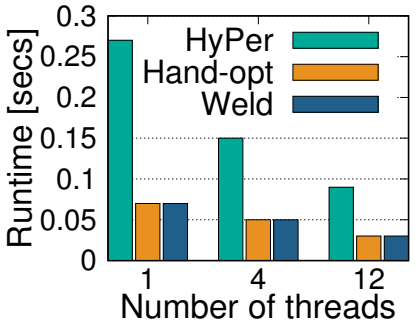
Competitive with High-Performance Databases



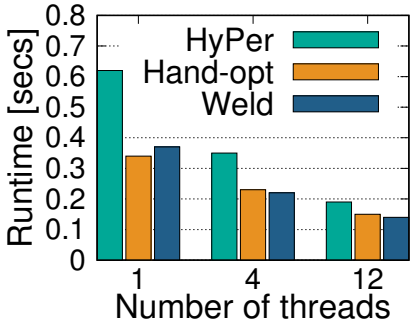
Q1



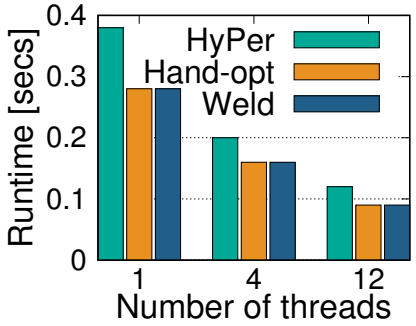
Q3



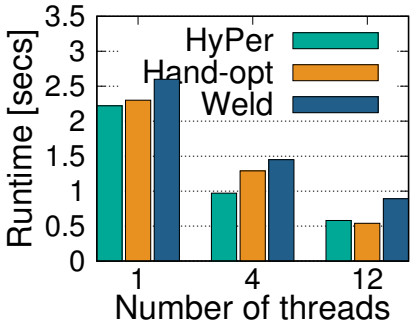
Q6



Q12



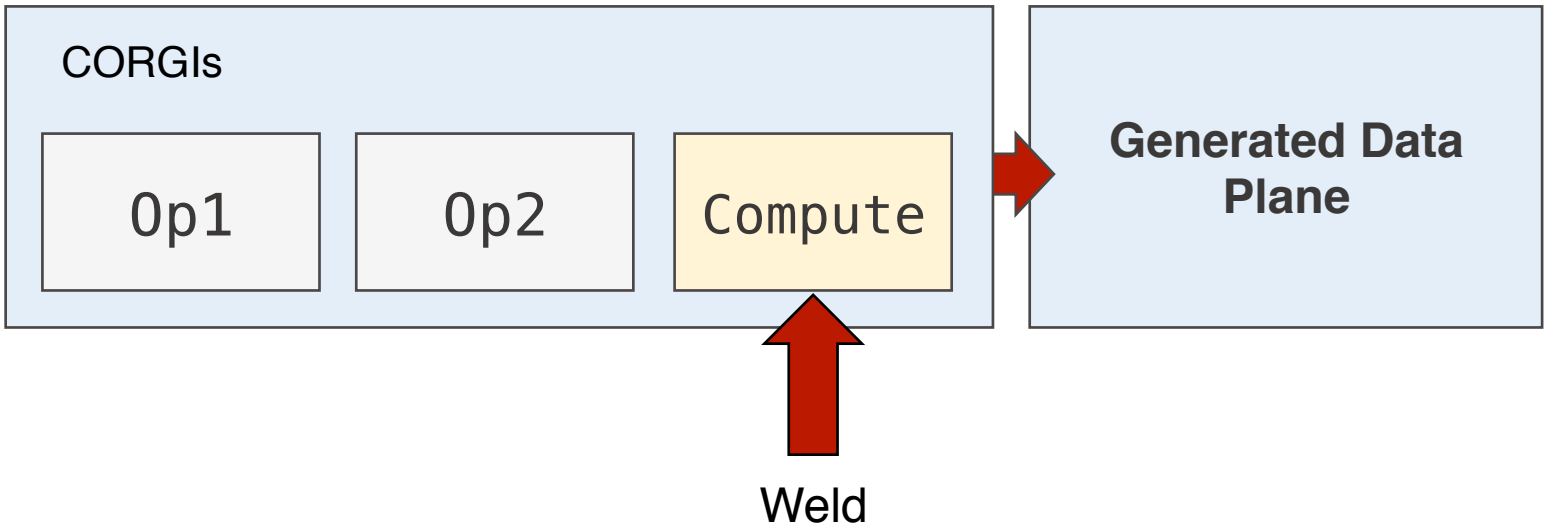
Q14



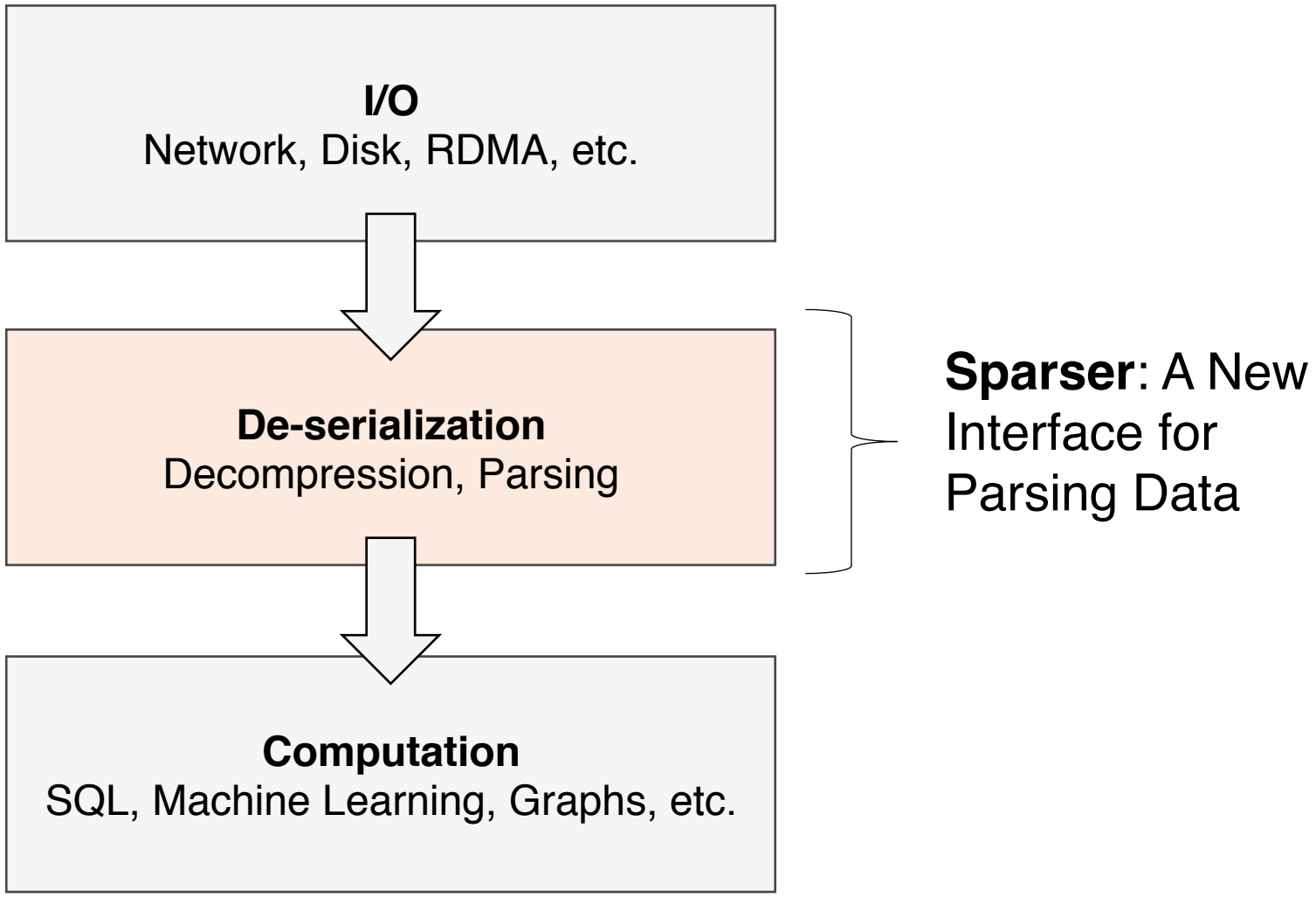
Q19

Competitive with HyPer (code generating, main-memory DB)

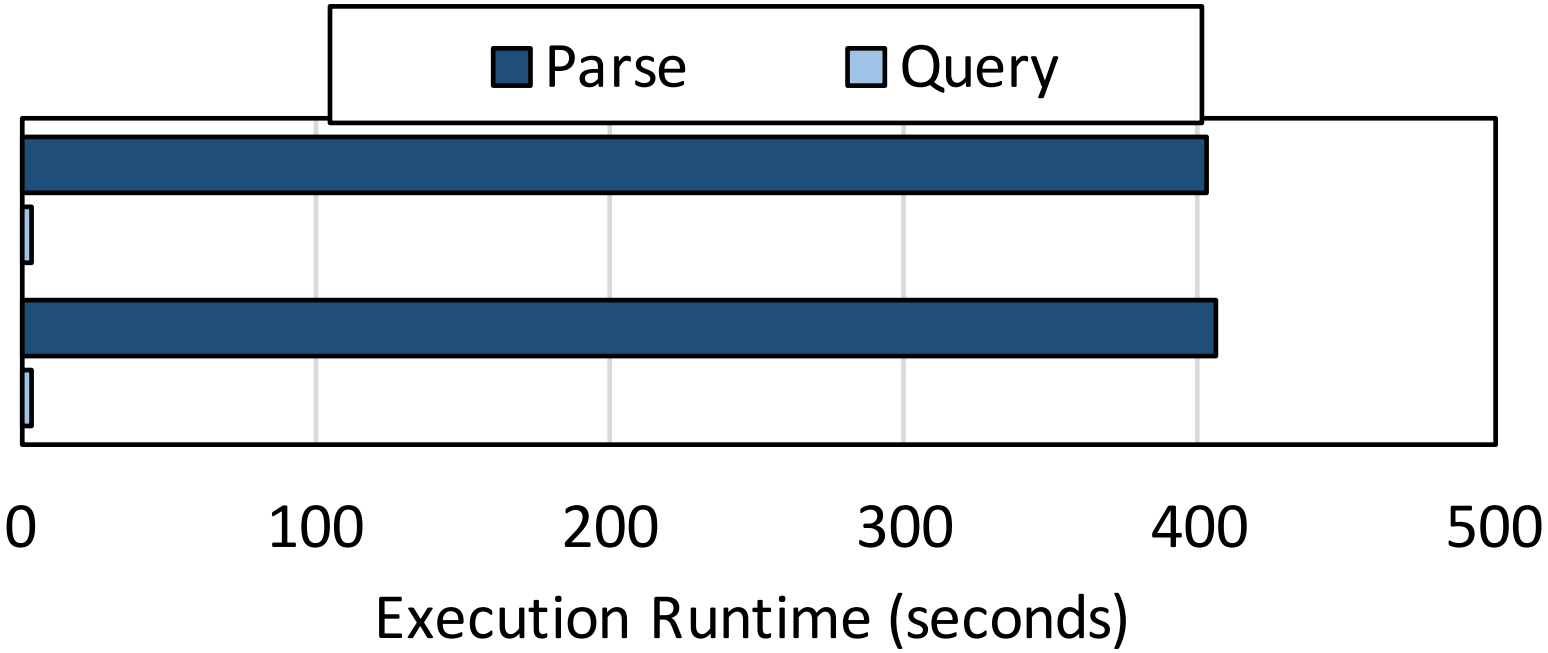
CORGIs



Sparser: Optimizing Parsing

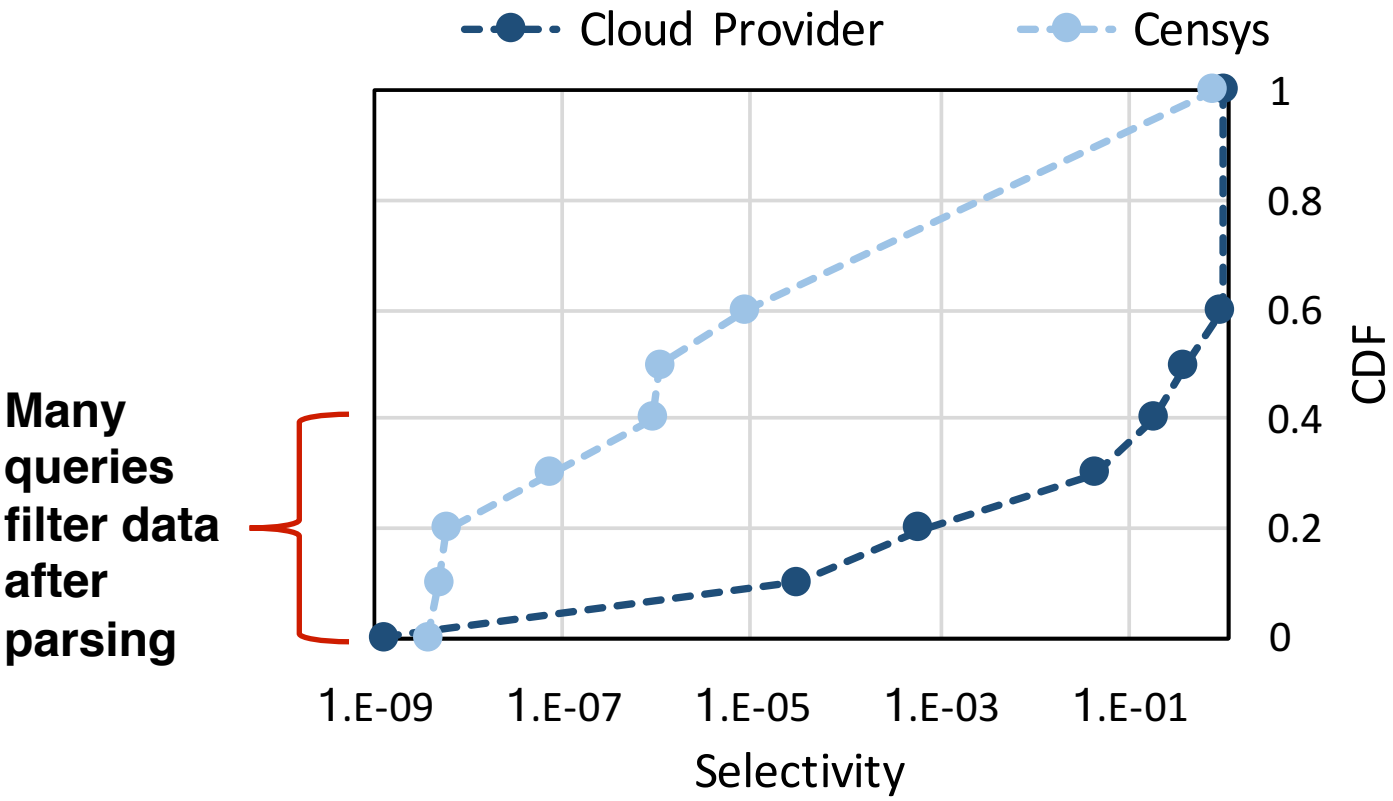


Parsing Time



Parsing as a bottleneck – common in many workloads!

Big Opportunity: Selectivity



Sparser: Accelerating Loading

Two main ideas:

1. **Provide API to tell system what to load**

   "Predicate pushdown" for arbitrary data formats

2. **Reject data which will be filtered out very quickly (*i.e.*, do not parse it)**

   Accelerates loading for high selectivity

Sparser API

Specify *filtering expressions*.

```
“tweet.text contains ‘I love HPTS’ &&
tweet.date contains Oct 11”
```

Reject Data with Pre-Filters

Approximate pre-filters with **false positives**

Example: SIMD to search for substrings.

```
Input   : I'd think I love HPTS 2017. Cool!
Shift 1: HPTSHPTSHPTSHPTSHPTSHPTSHPTS-------
Shift 2: -HPTSHPTSHPTSHPTSHPTSHPTSHPTS------
Shift 3: --HPTSHPTSHPTSHPTSHPTSHPTSHPTS-----
Shift 4: ---HPTSHPTSHPTSHPTSHPTSHPTSHPTS----
```

Full Search Query

Approximate Pre-Filter with SIMD vectorization

How to Pick Pre-Filters?

“tweet.text contains ‘I love HPTS’ && tweet.date contains Oct 11”

We could pick many potential search strings. Some examples:

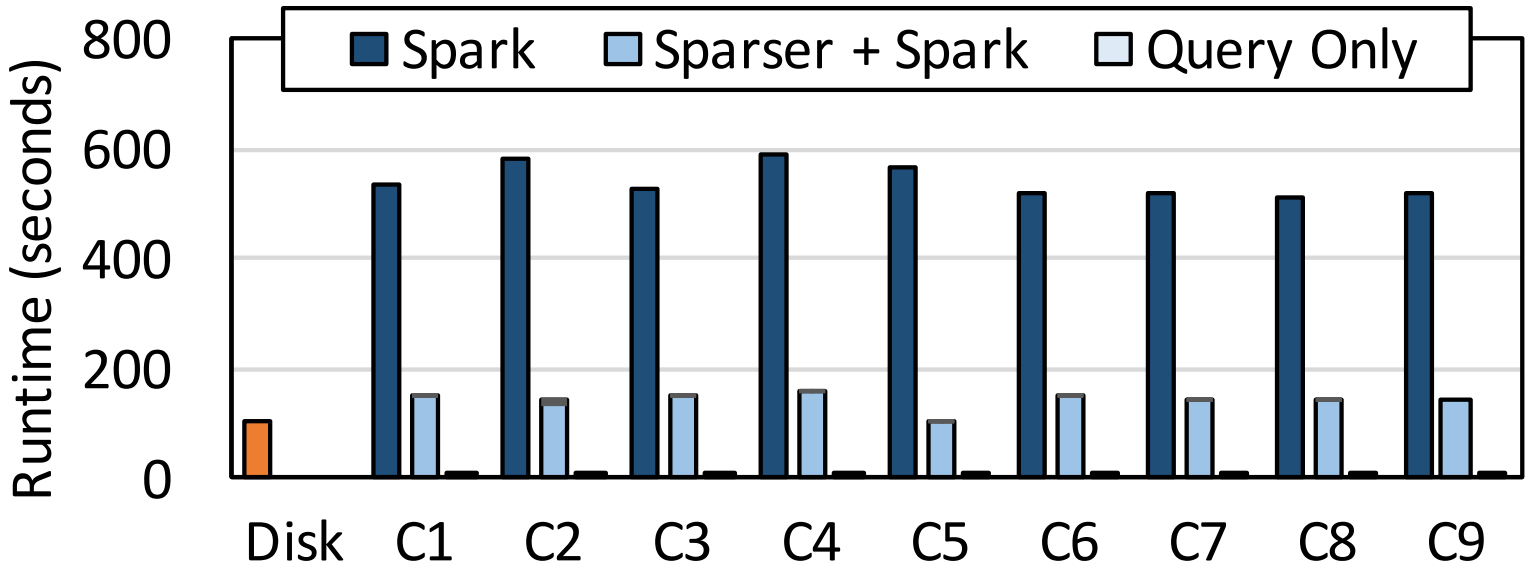
- Oct
- ct 1
- HP
- HPTS

Cheaper to search for since smaller substring

Use a *scheduler* to choose the best pre-filter cascade

Minimize Expected Parsing Time

Avoid Parsing → 10x Speedups

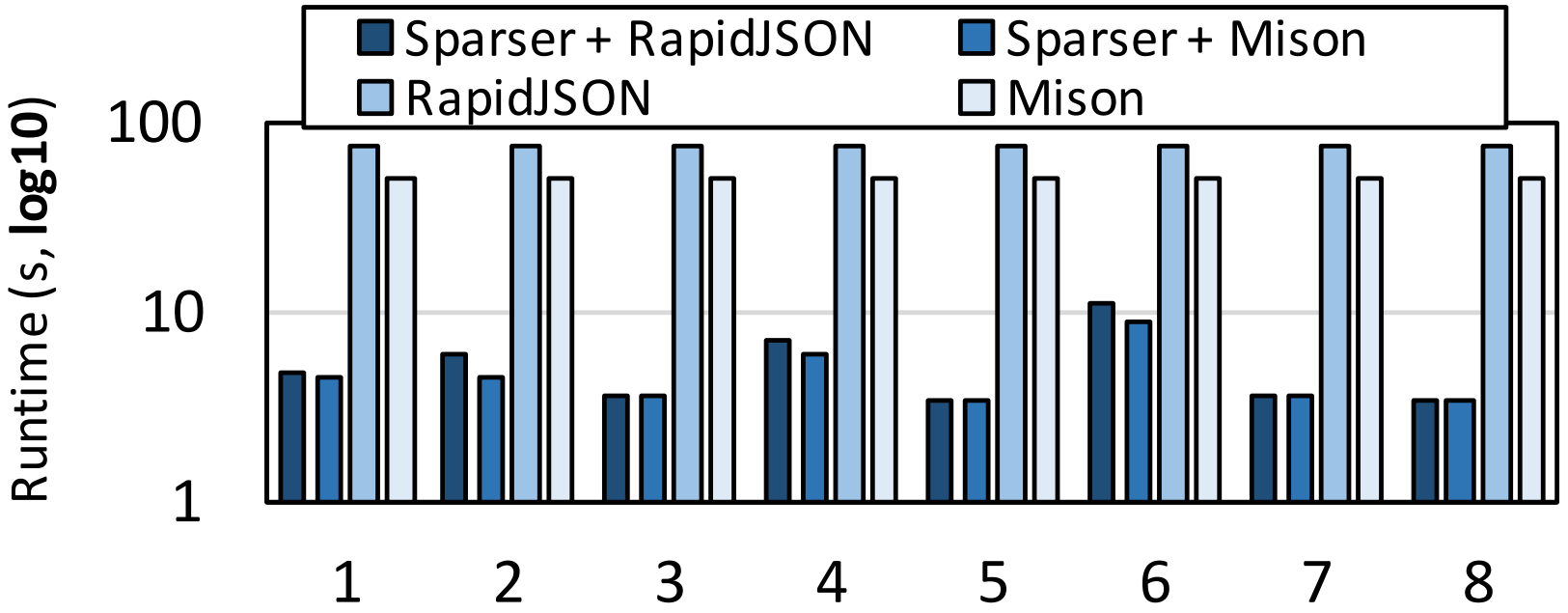


Popular queries* from Censys, a search engine of the IPv4 address space

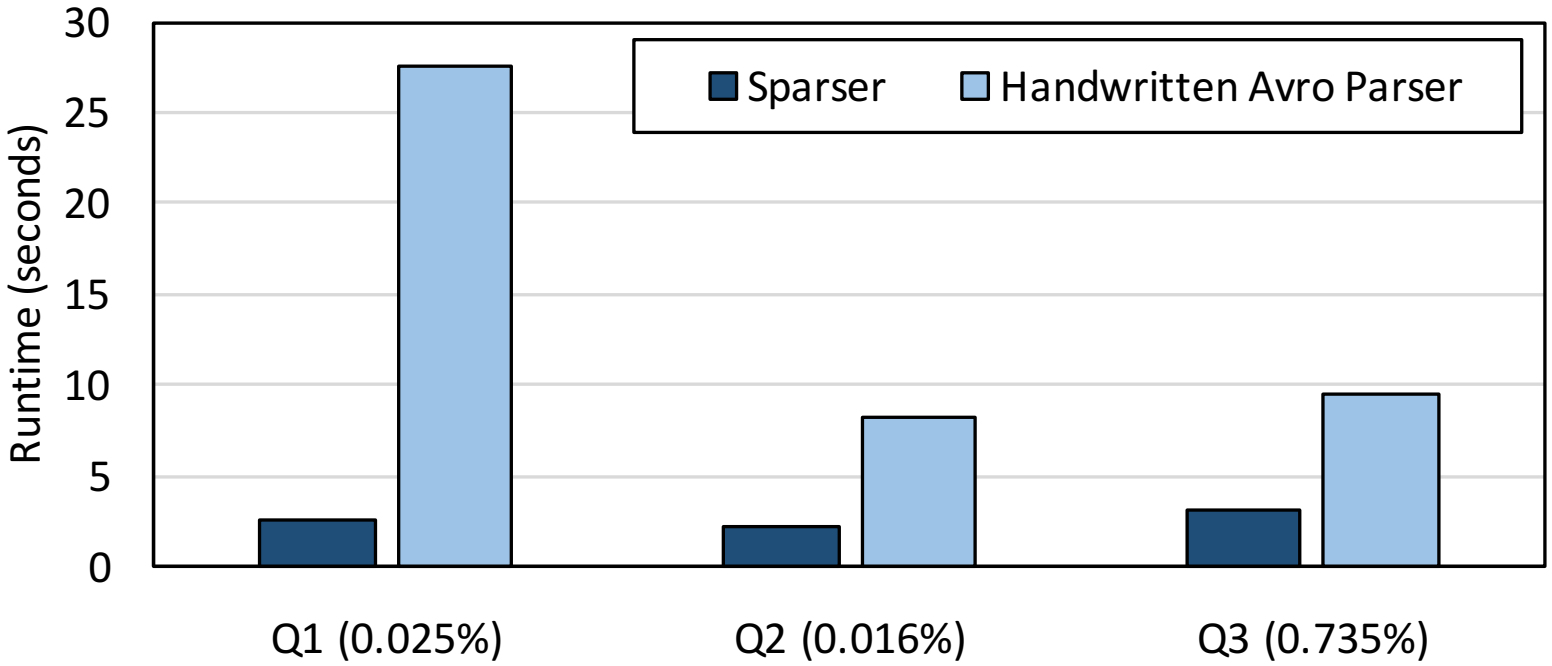
*queries have low selectivity! < 0.0001%

10x Speedup in end-to-end Spark application

10x over State-of-the-art Parsers

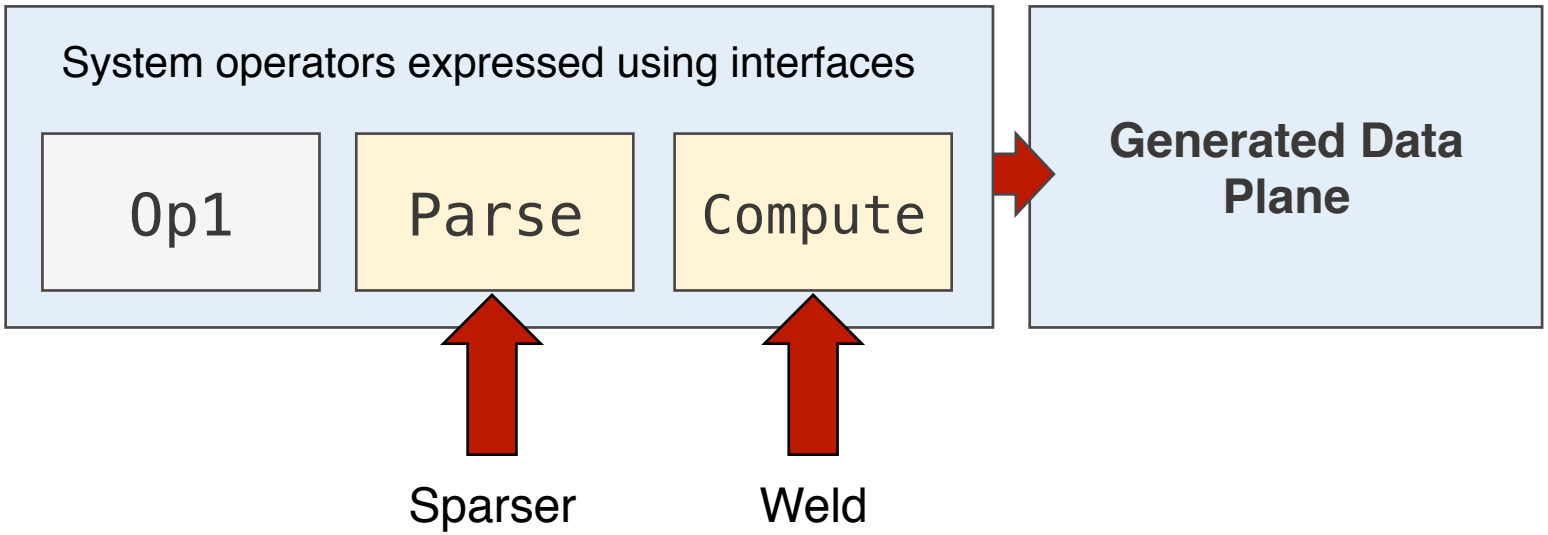


Works for Binary, Too!

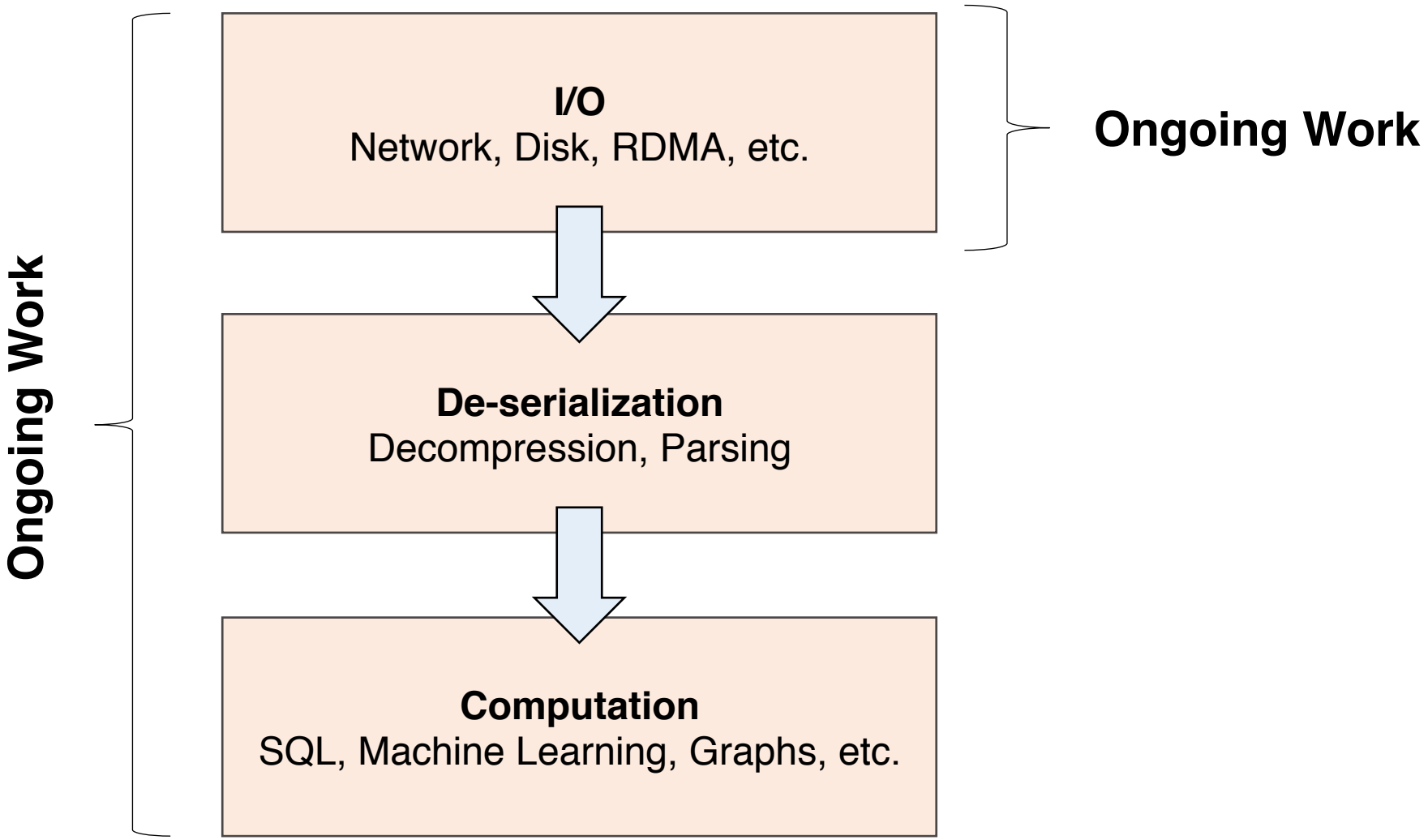


7x Speedup compared with C-based Avro Parser

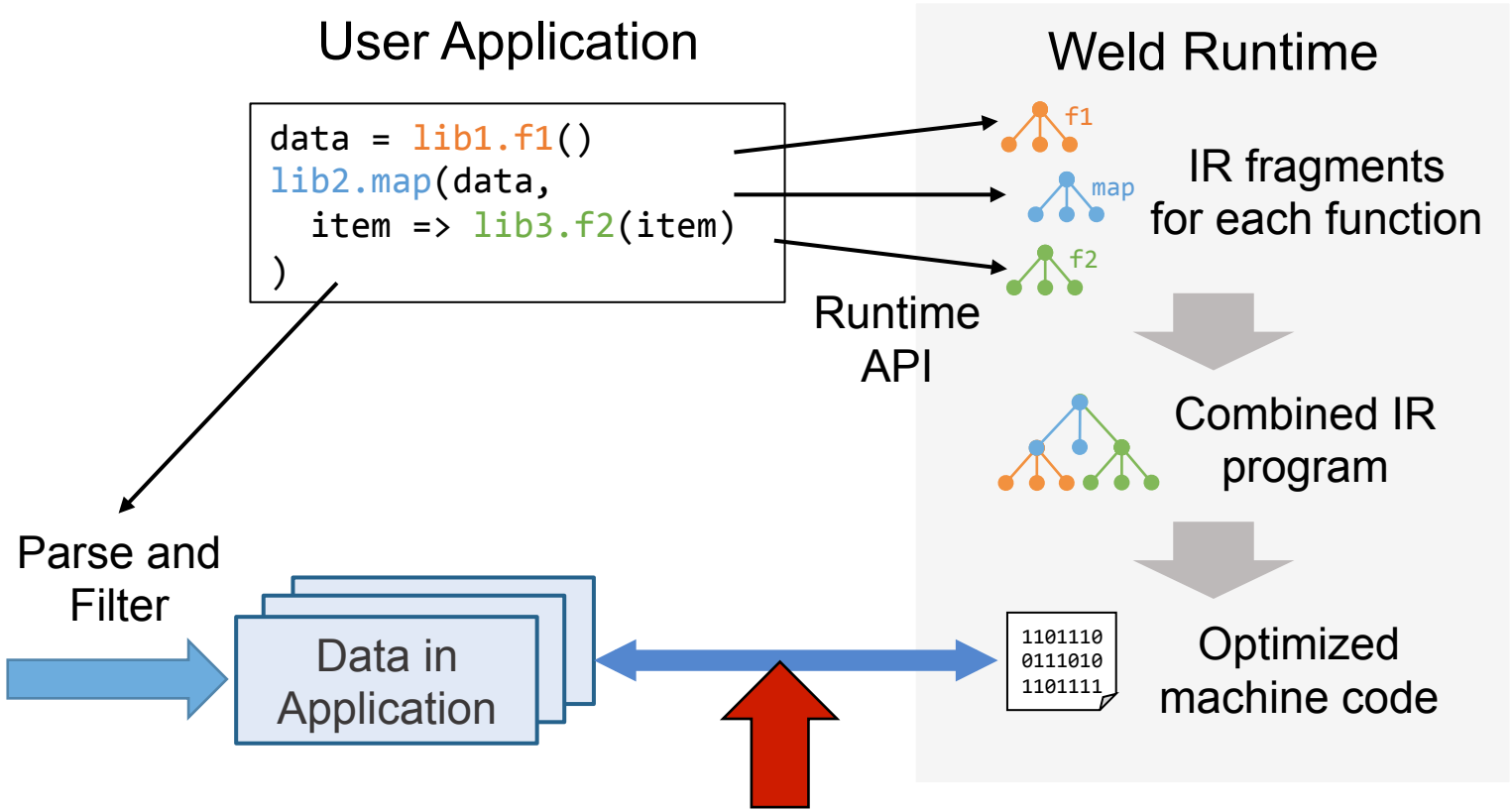
CORGIs


System operators expressed using interfaces
Op1
Parse
Compute
Generated Data Plane
Sparser
Weld

Toward an Analytics Data Plane



Unifying Interfaces



This arrow represents many things!

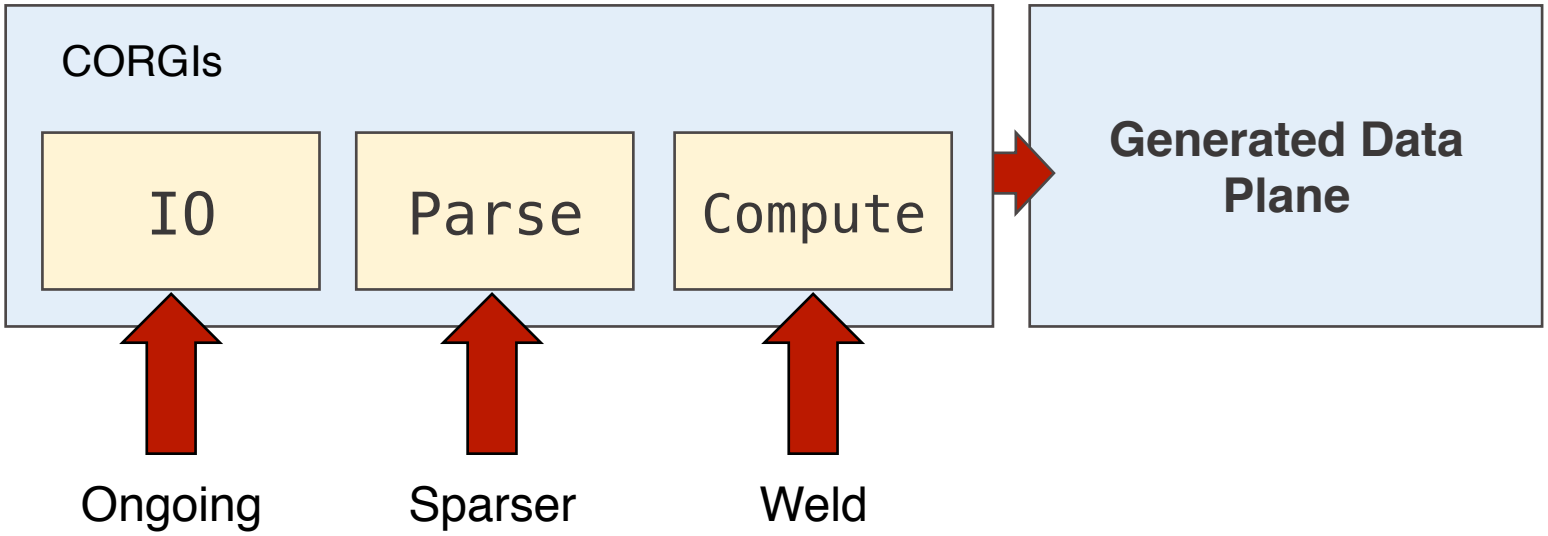
Example: Generating Format-Specific Code

Provide a **data format specification** and a computation graph

Generate code for the format **directly** instead of marshalling

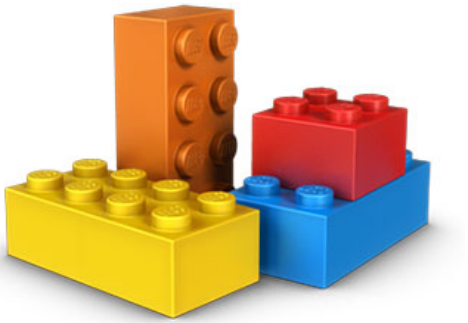
- RPCs
- Network code

CORGIs


CORGIs
IO
Parse
Compute
Generated Data Plane
Ongoing
Sparser
Weld

Small Pieces, Fast Systems

Powerful, composable interfaces = Fast, optimized systems



Conclusion

Try out Weld for yourself, or contribute!

https://www.github.com/weld-project

http://weld.stanford.edu

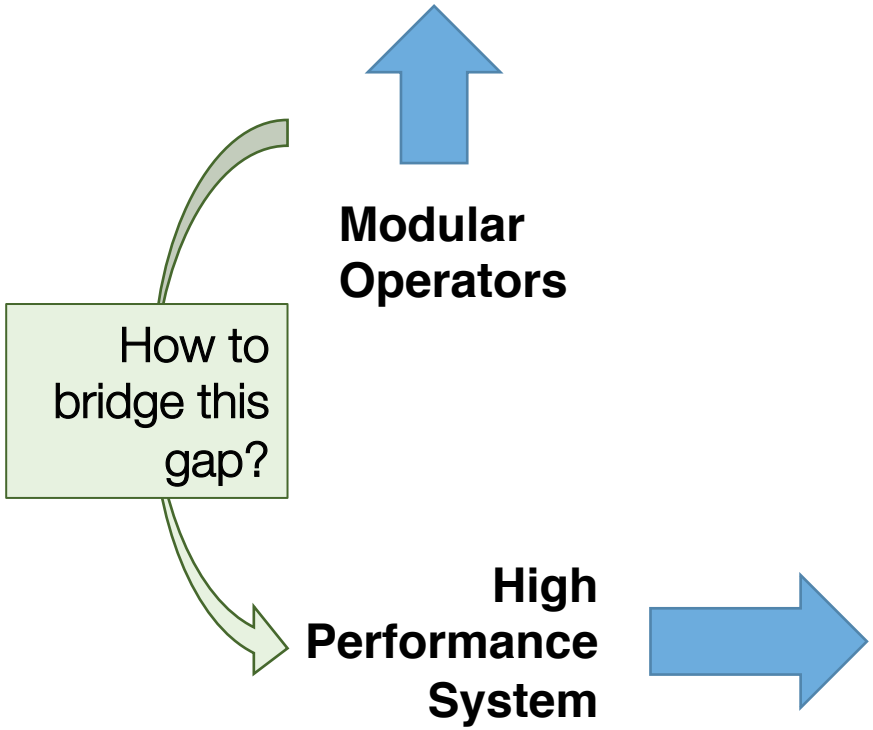
Sparser paper and code coming soon!

shoumik@cs.stanford.edu

https://shoumik.xyz

Modularity vs. Performance

```
data = loadCSV(input)
filtered = Filter(data, NULL)
avg = Mean(filtered)
```




```
#ifdef MEASURE_CYCLES
    long start = rdtsc();
#endif

    if (matchmask != allset) {
      const char *base = input + i;
      __m256i val = _mm256_loadu_si256((__m256i const *)(base));
      unsigned mask = _mm256_movemask_epi8(_mm256_cmpeq_epi32(val, q1));
      //__m256 vmask =_mm256_cmpeq_epi32(val, q1);

      __m256i val2 = _mm256_loadu_si256((__m256i const *)(base + 1));
      mask |= _mm256_movemask_epi8(_mm256_cmpeq_epi32(val2, q1));
      //vmask = _mm256_or_si256(vmask, _mm256_cmpeq_epi32(val2, q1));

      __m256i val3 = _mm256_loadu_si256((__m256i const *)(base + 2));
      mask |= _mm256_movemask_epi8(_mm256_cmpeq_epi32(val3, q1));
      //vmask = _mm256_or_si256(vmask, _mm256_cmpeq_epi32(val3, q1));

      __m256i val4 = _mm256_loadu_si256((__m256i const *)(base + 3));
      mask |= _mm256_movemask_epi8(_mm256_cmpeq_epi32(val4, q1));
      //vmask = _mm256_or_si256(vmask, _mm256_cmpeq_epi32(val4, q1));

      //unsigned mask = _mm256_movemask_epi8(vmask);
      mask &= 0x11111111;

      if (mask > 0) {
        unsigned matched = _mm_popcnt_u32(mask);
        stats.total_matches += matched;
        matchmask |= 0x1;
      }

      if (!IS_SET(matchmask, 1)) {


        mask = _mm256_movemask_epi8(_mm256_cmpeq_epi32(val, q2));
        mask |= _mm256_movemask_epi8(_mm256_cmpeq_epi32(val2, q2));
        mask |= _mm256_movemask_epi8(_mm256_cmpeq_epi32(val3, q2));
        mask |= _mm256_movemask_epi8(_mm256_cmpeq_epi32(val4, q2));
        /*
        vmask = _mm256_cmpeq_epi32(val, q2);
        vmask = _mm256_or_si256(vmask, _mm256_cmpeq_epi32(val2, q2));
        vmask = _mm256_or_si256(vmask, _mm256_cmpeq_epi32(val3, q2));
        vmask = _mm256_or_si256(vmask, _mm256_cmpeq_epi32(val4, q2));
        mask = _mm256_movemask_epi8(vmask);
        */
        mask &= 0x11111111;

        if (mask) {
          unsigned matched = _mm_popcnt_u32(mask);
          stats.total_matches += matched;
          matchmask |= (0x1 << 1);
        }
      }
    }

    // check if all the filters matched by checking if all the bits
    // necessary were set in matchmask.
    if (matchmask == allset) {
      stats.sparser_passed++;

      // update start. Using vectors here seems to only make a marginal performance difference.
      long start = i;
      __m256i nl = _mm256_set1_epi8('\n');
      for (; start >= 32; start -= 32) {
        __m256i tmp = _mm256_loadu_si256((__m256i *)(input + start - 32));
        if (_mm256_movemask_epi8(_mm256_cmpeq_epi8(tmp, nl))) {
          start -= 32;
          break;
        }
      }}
      input[end] = a;

      // Reset record level state.
      matchmask = 0;

      // Done with this record - move on to the next one.
      i = end + 1 - VECSZ;
    }
```