CWI
Database Architectures

Reconciling progress in statistical computation and data management

Hannes Mühleisen

CWI Scientific Meeting, 2018-01-26

Starting Point

- (Interpreted) Scalar Code
- DB: Tuple-at-a-Time, PostgreSQL, MySQL, …
- Problem:
  Massive overheads problematic if data gets big™

Memory Hierarchy

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns

New: https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html

[Jeff Dean]

Bulk Processing

- Need to **avoid** cache misses
  - Process large chunks of data in one go using efficient code (C)
- S/R (1984), DSM (1985), NumPy (1995), MonetDB (~1995)
  - “Column-at-a-time”
- Example: Python vs. NumPy
  - `statistics.mean(range(1, pow(10, 7)))` 7.52s
  - `numpy.arange(1, pow(10, 7)).mean()` 0.03s

MonetDB Query Plan

```
create table fuu (id integer, type integer);
explain select id from fuu where type=1;
```

```
X_19:bat[:int] := sql.bind(X_5, "sys", "fuu", "type", 0:int);
C_6:bat[:oid] := sql.tid(X_5, "sys", "fuu");
C_28 := algebra.thetaselect(X_19, C_6, 1:int, "==");
X_9:bat[:int] := sql.bind(X_5, "sys", "fuu", "id", 0:int);
X_30 := algebra.projection(C_28, X_9);
```

What if almost all `type` values are 1?

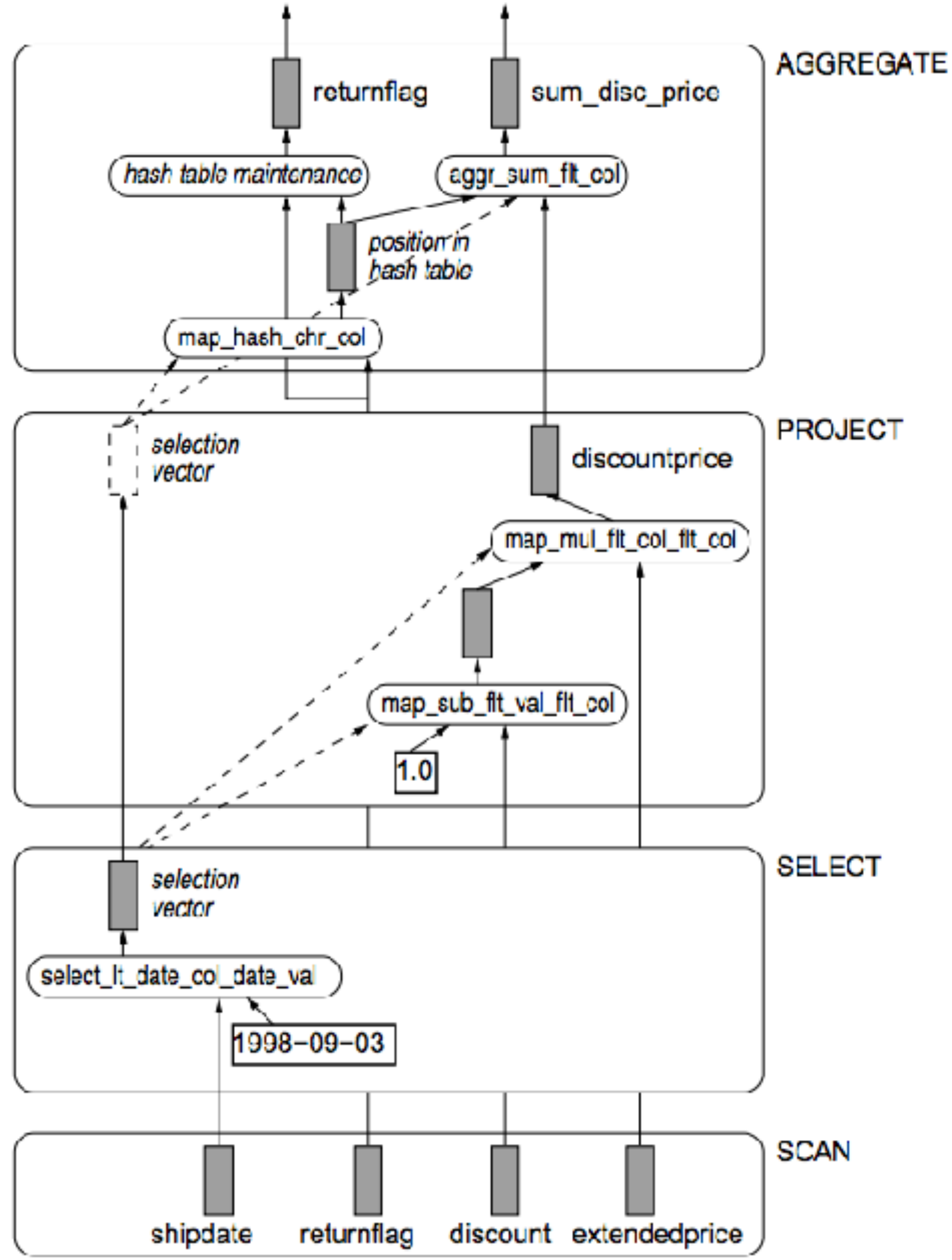
Large Intermediates

- Intermediate results can get large, too
- Problematic, since they have to go to memory
  - or worse, to disk

Vectorized

- Innovation 1: *Vectorized* Processing

- Best of both worlds, small intermediate results, interpreted batch processing.

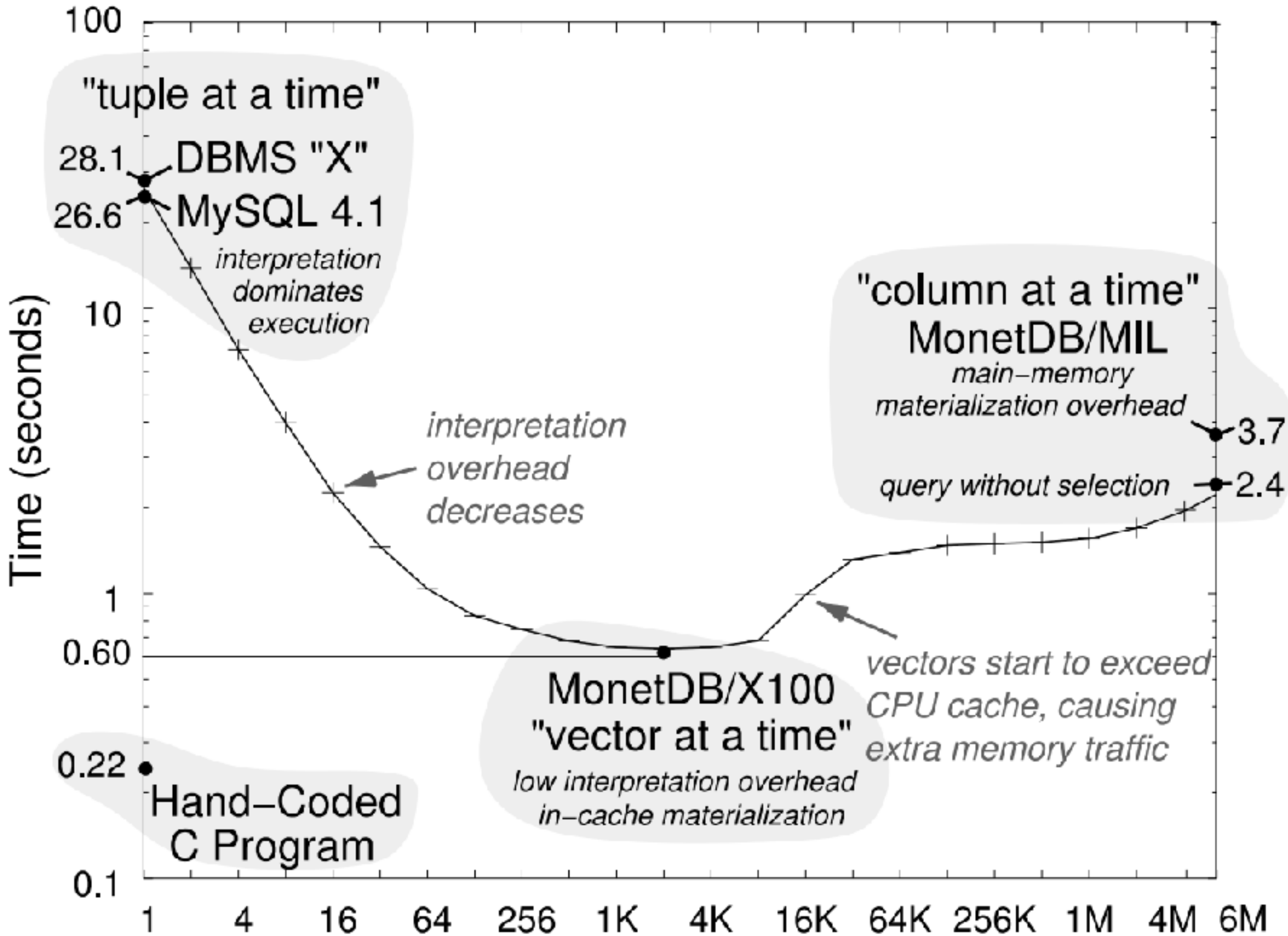


[Boncz et al., MonetDB/X100…]

JIT Compilation

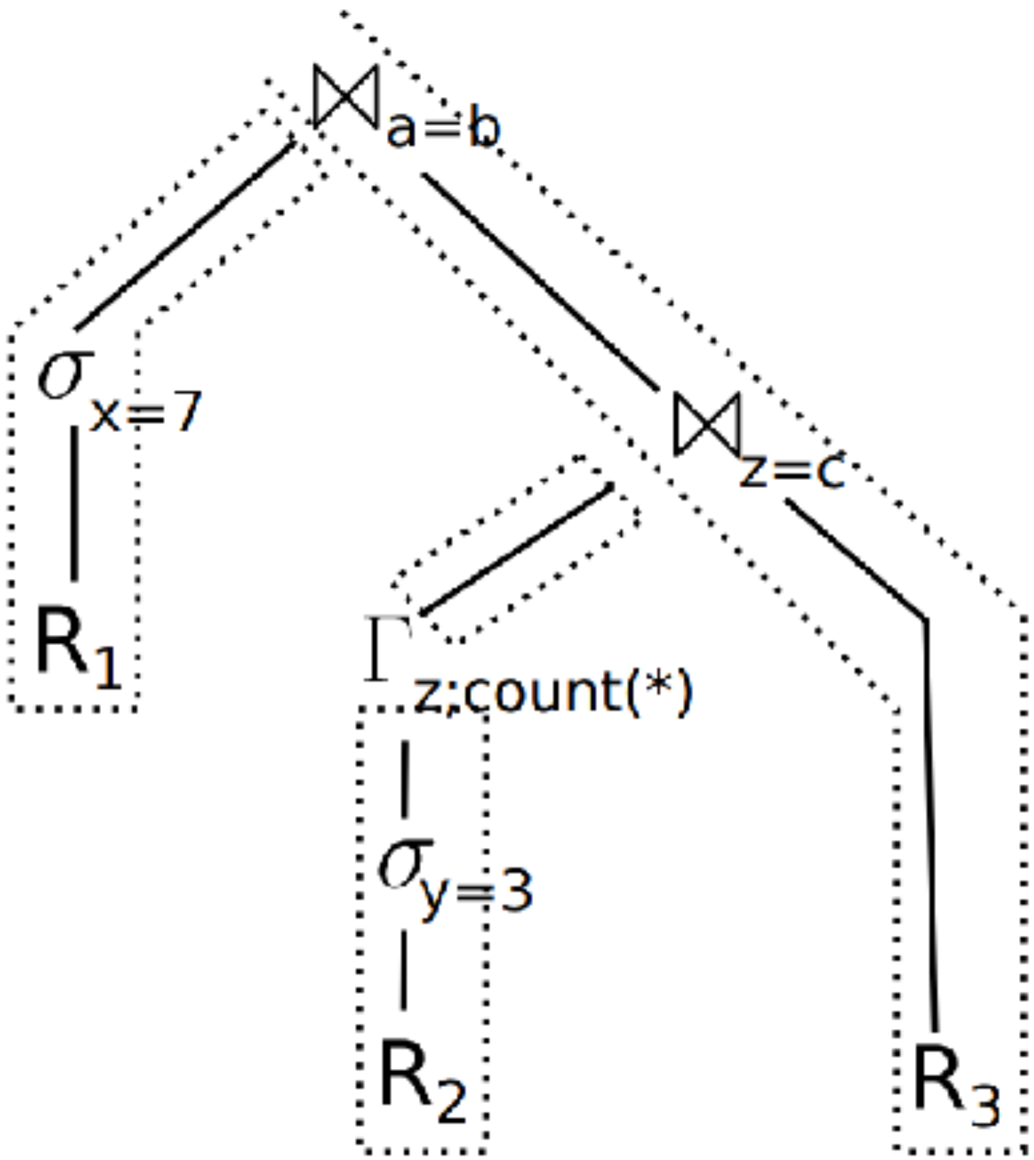
- Innovation 2: Ad-hoc generate scalar code that computes results, *compile* & run
- Avoids interpretation overheads & allows free pipelining
- Keep declarative user interface

How does this help avoiding
branch mis-prediction & cache misses?



[Neumann, Efficiently Compiling Efficient Query Plans for Modern Hardware]

Query Compilation



initialize memory of $\bowtie_{a=b}$, $\bowtie_{c=z}$, and Γ_z

for each tuple t in R_1
    if $t.x = 7$
        materialize t in hash table of $\bowtie_{a=b}$

for each tuple t in R_2
    if $t.y = 3$
        aggregate t in hash table of Γ_z

for each tuple t in Γ_z
    materialize t in hash table of $\bowtie_{z=c}$

for each tuple t_3 in R_3
    for each match t_2 in $\bowtie_{z=c}[t_3.c]$
        for each match t_1 in $\bowtie_{a=b}[t_3.b]$
            output $t_1 \circ t_2 \circ t_3$

[Neumann, Efficiently Compiling Efficient Query Plans for Modern Hardware]

- Code generation for numerical computation
- Tradeoff between performance & productivity
    - Immutable types, fixed types for variables, immutable, but extensible code, `const`
- Type inference, code generation & optimization
- LLVM

[Bezanson et al., Julia: A Fresh Approach to Numerical Computing]

Down the rabbit hole

```
julia> sum(1:100)
```

Step 1: Type Inference

```
Any[CodeInfo(:(begin
        return $(Expr(:invoke, MethodInstance for
sum(::UnitRange{Int64}), :(Main.sum), :($(Expr(:new,
UnitRange{Int64}, 1, :((Base.select_value)((Base.sle_int)(1,
100)::Bool, 100, (Base.sub_int)(1, 1)::Int64)::Int64)))))
    end))=>Int64]
```

http://blog.leahhanson.us/post/julia/julia-introspects.html

Down the rabbit hole

```
julia> sum(1:100)
```

Step 2: LLVM

```
%0 = alloca %UnitRange, align 8
%1 = getelementptr inbounds %UnitRange, %UnitRange* %0, i64 0, i32 0
store i64 1, i64* %1, align 8
%2 = getelementptr inbounds %UnitRange, %UnitRange* %0, i64 0, i32 1
store i64 100, i64* %2, align 8
%3 = call i64 @julia_sum_62577(%UnitRange* nocapture nonnull readonly %0)
ret i64 %3
```

Down the rabbit hole

```
julia> sum(1:100)
```

Step 3: Native Code

```
subq      $16, %rsp
movq      $1, -16(%rbp)
movq      $100, -8(%rbp)
movabsq   $sum, %rax
leaq      -16(%rbp), %rdi
callq     *%rax
addq      $16, %rsp
popq      %rbp
retq
```

What's missing here?

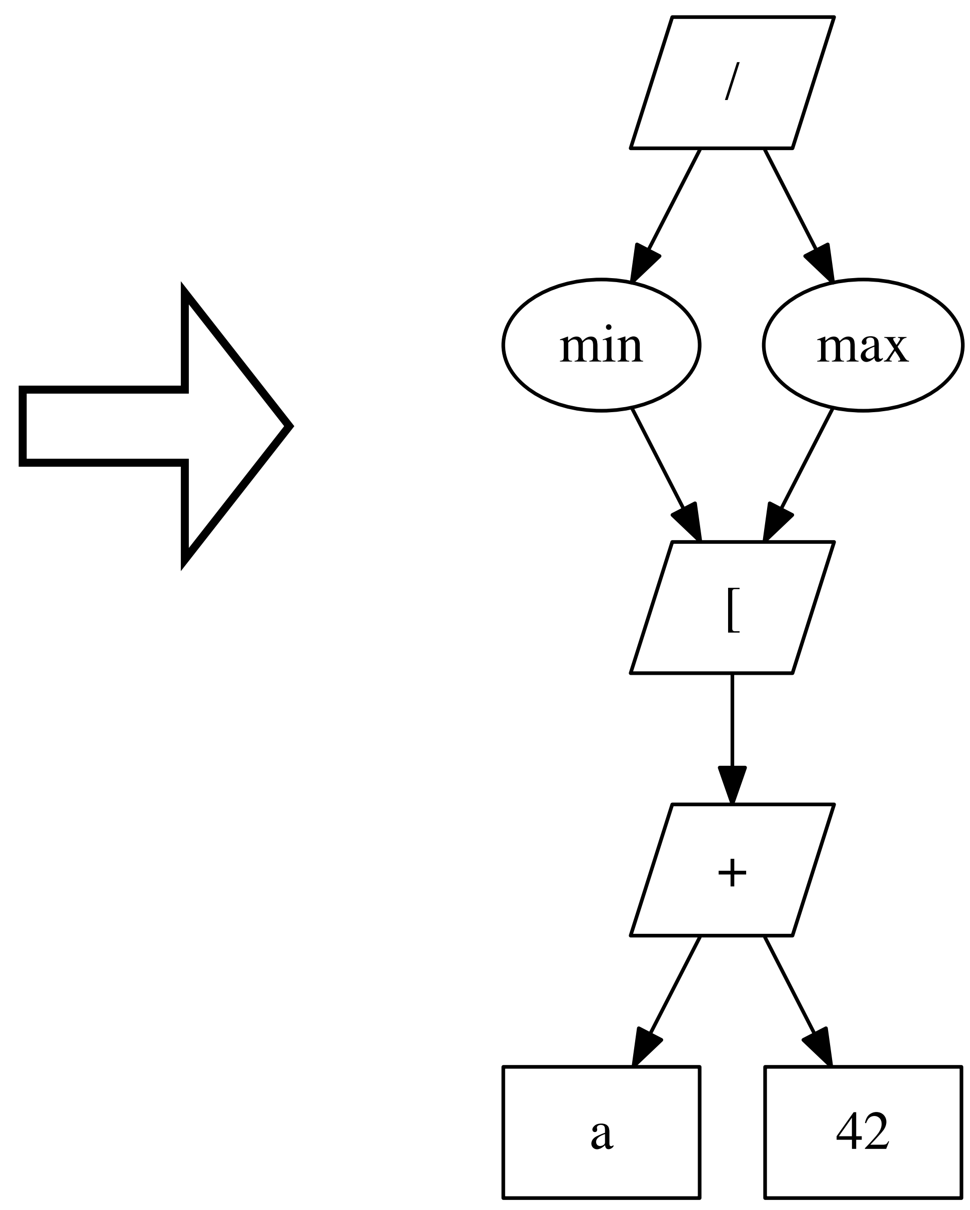
Non-Standard Evaluation

- Bringing DB-style high-level optimisations to Statistical Programming
- Lazy evaluation of R scripts to build tree of deferred ops, then optimize

[Mühleisen et al., Relational Optimizations for Statistical Analysis]

Deferred Evaluation

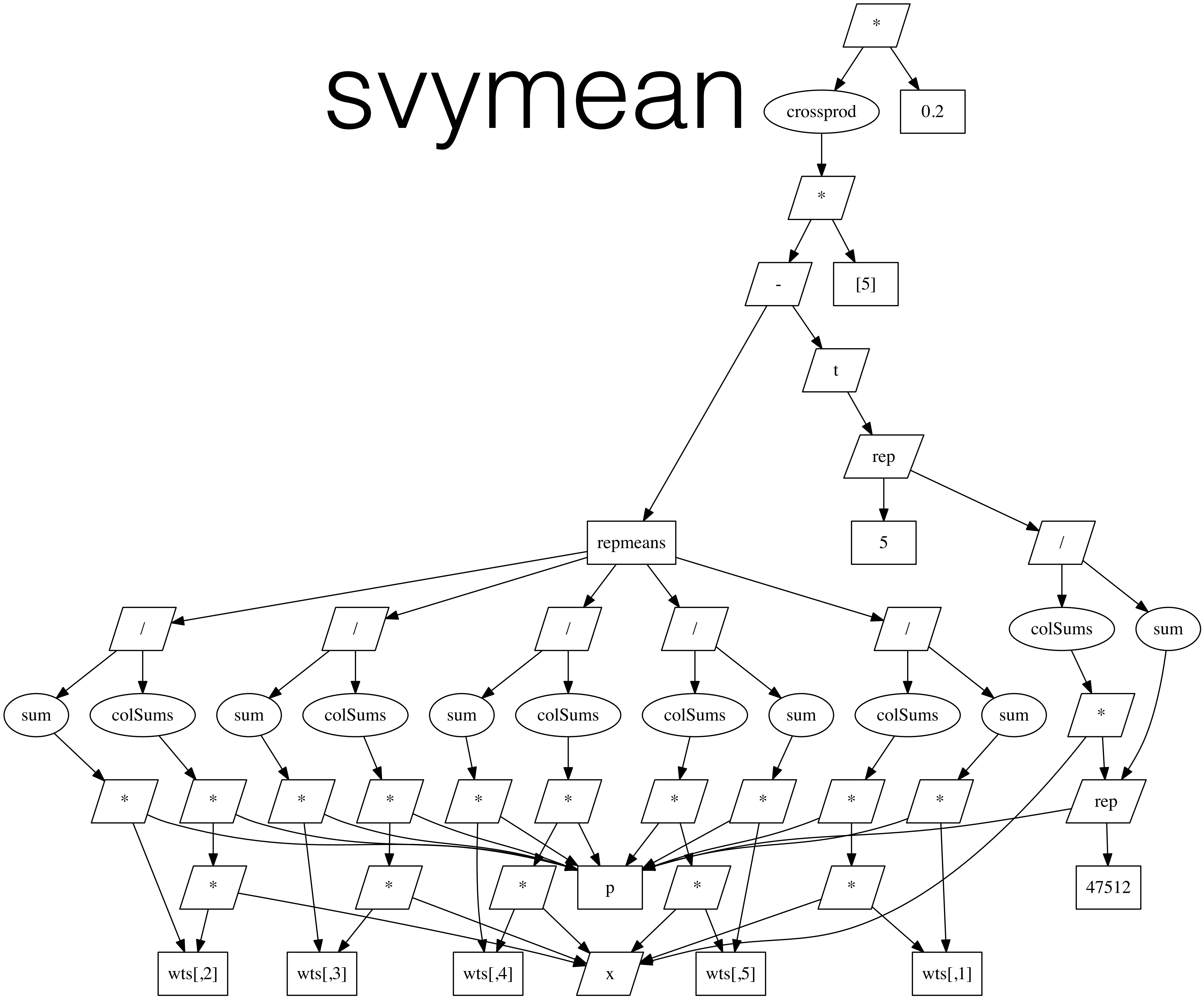
```
a <- 1:1000
b <- a + 42
c <- b[1:10]
d <- min(c) / max(c)
print(d)
```



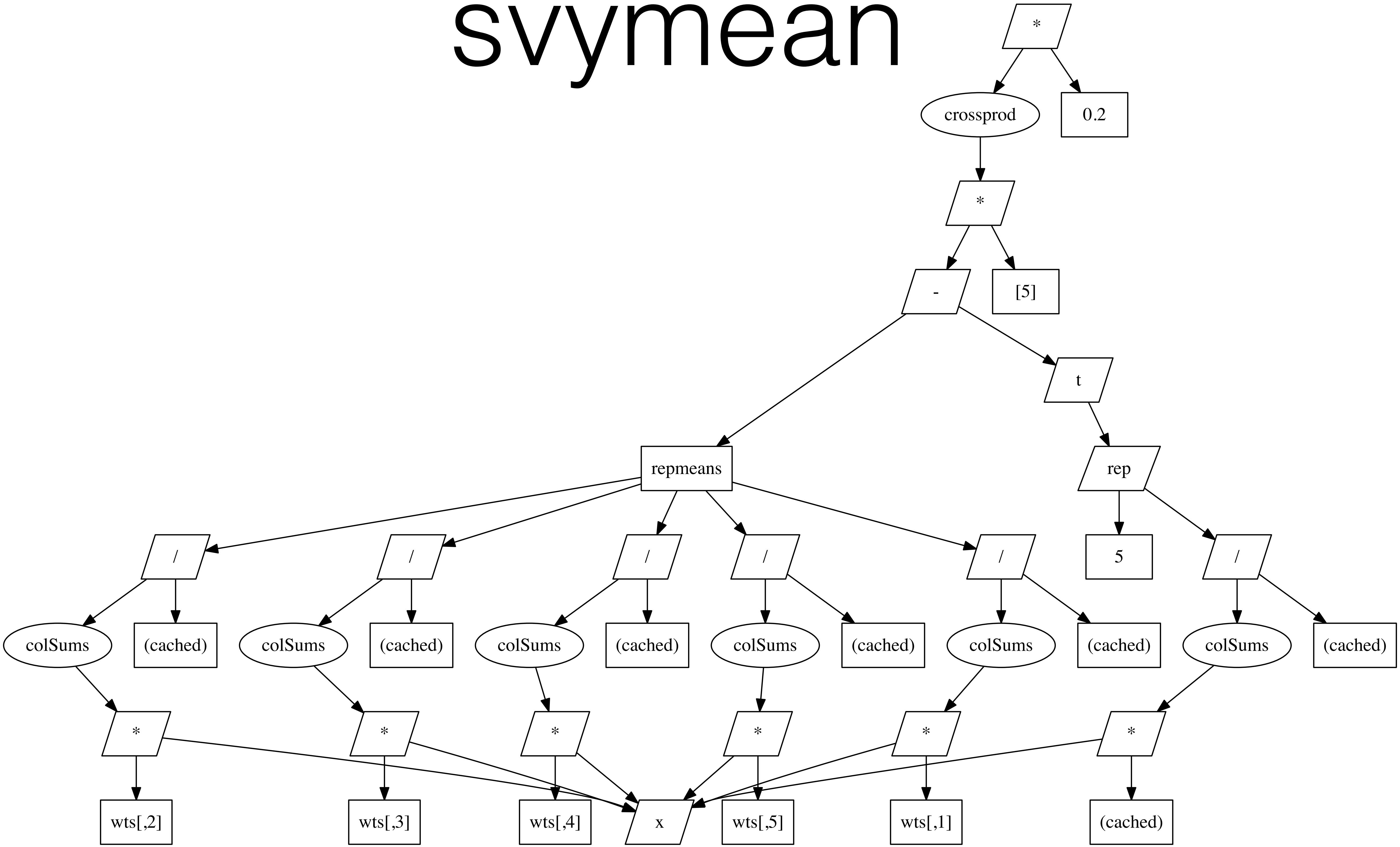
svymean

```
agep <- svymean(~agep, svydsgn, se=TRUE)


for(i in 1:ncol(wts)) {
  repmeans[i,]<-t(colSums(wts[,i]*x*pw)/
      sum(pw*wts[,i]))
}
[…]
v<-crossprod(sweep(thetas,2,
meantheta,"-")*sqrt(rscales))*scale
```

svymean

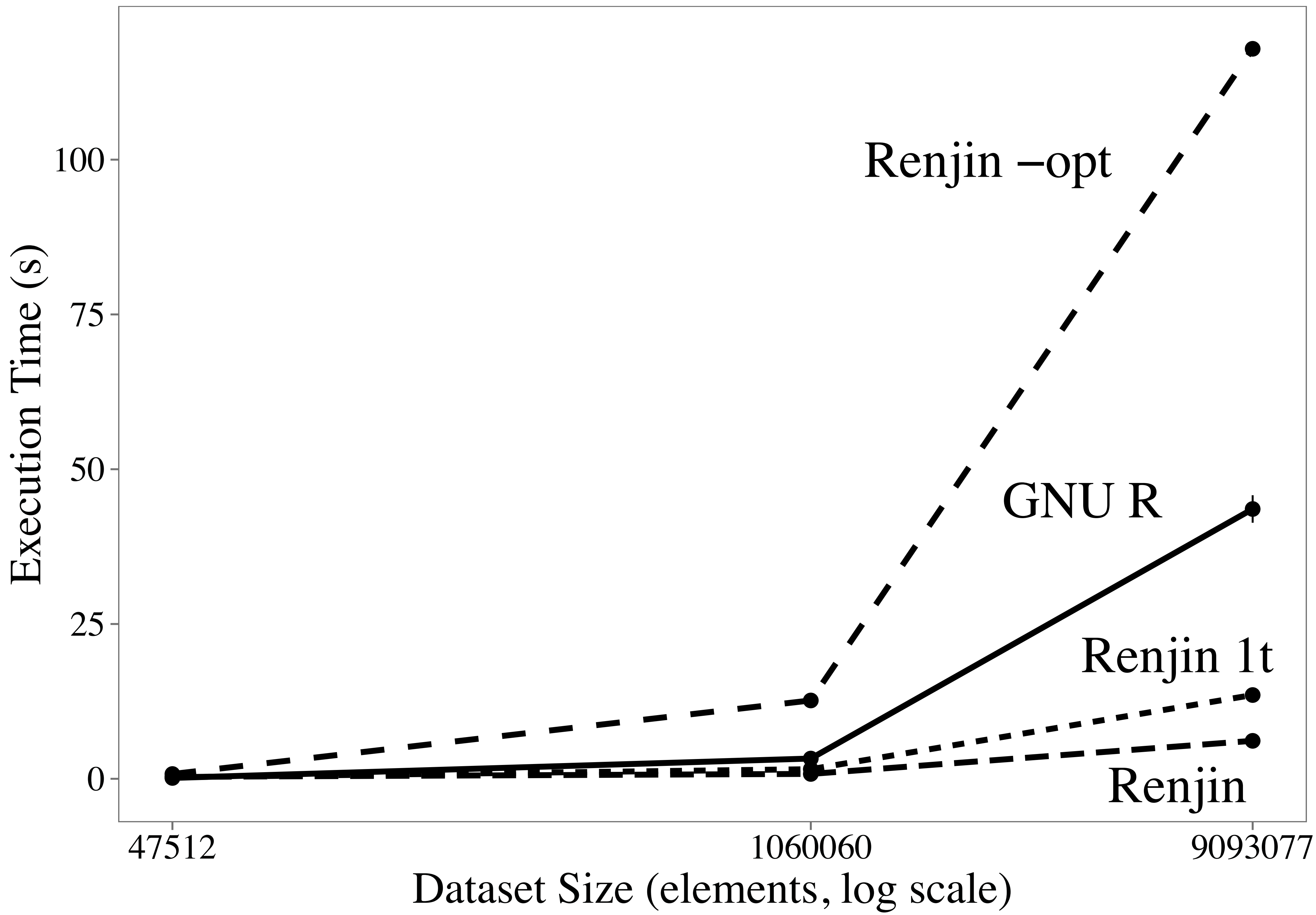

*
crossprod
0.2
*
-
[5]
t
rep
repmeans
5
/
/
/
/
/
/
/
sum
colSums
sum
colSums
sum
colSums
colSums
sum
colSums
sum
colSums
sum
*
*
*
*
*
*
*
*
*
*
*
*
rep
*
*
*
*
*
p
47512
wts[,2]
wts[,3]
wts[,4]
x
wts[,5]
wts[,1]

svymean



*
crossprod
0.2
*
-
[5]
t
repmeans
rep
/
/
/
/
/
5
/
colSums
(cached)
colSums
(cached)
colSums
(cached)
colSums
(cached)
colSums
(cached)
colSums
(cached)
*
*
*
*
*
*
wts[,2]
wts[,3]
wts[,4]
x
wts[,5]
wts[,1]
(cached)

svymean


Renjin –opt
GNU R
Renjin 1t
Renjin
Execution Time (s)
0
25
50
75
100
47512
1060060
9093077
Dataset Size (elements, log scale)

5.5. Performance Recap. In the early days of high-level numerical computing languages, the thinking was that the performance of the high-level language did not matter so long as most of the time was spent inside the numerical libraries. These libraries consisted of blockbuster algorithms that would be highly tuned, making efficient use of computer memory, cache, and low-level instructions.

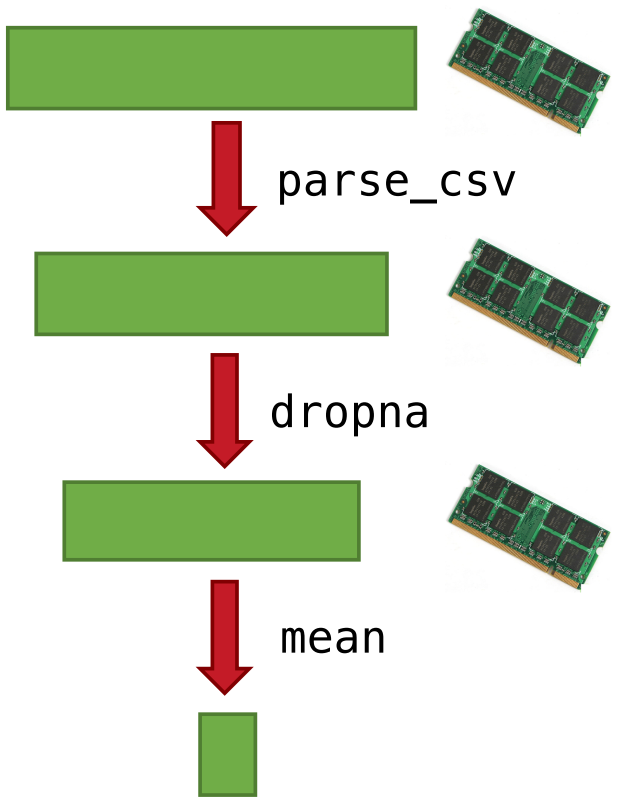
What the world learned was that only a few codes spent a majority of their time in the blockbusters. Most codes were being hampered by interpreter overheads, stemming from processing more aspects of a program at run time than are strictly necessary.

[Bezanson et al., Julia: A Fresh Approach to Numerical Computing]

Weld

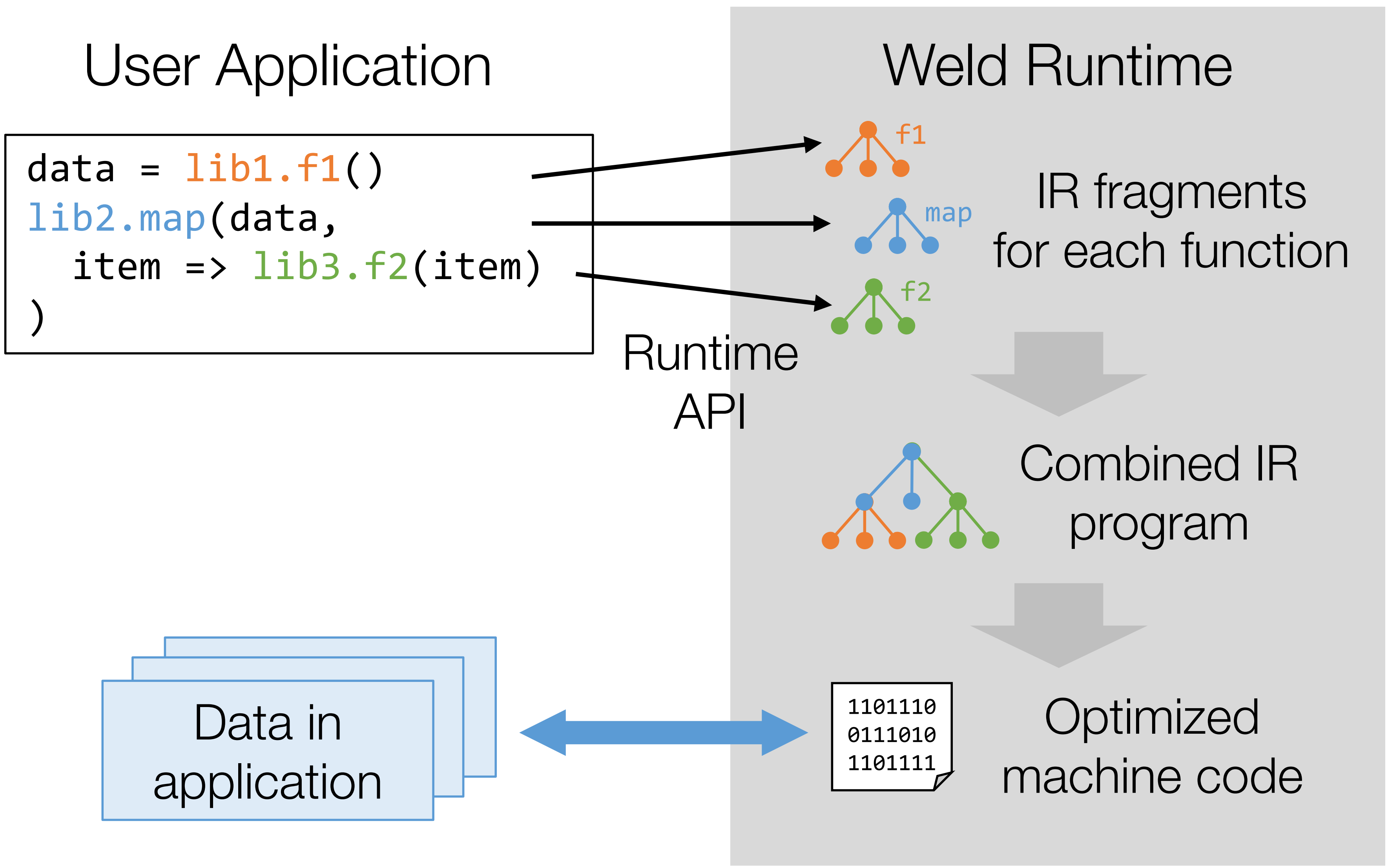
Growing gap between memory/processing makes traditional way of combining functions worse

```
data = pandas.parse_csv(string)

filtered = pandas.dropna(data)

avg = numpy.mean(filtered)
```



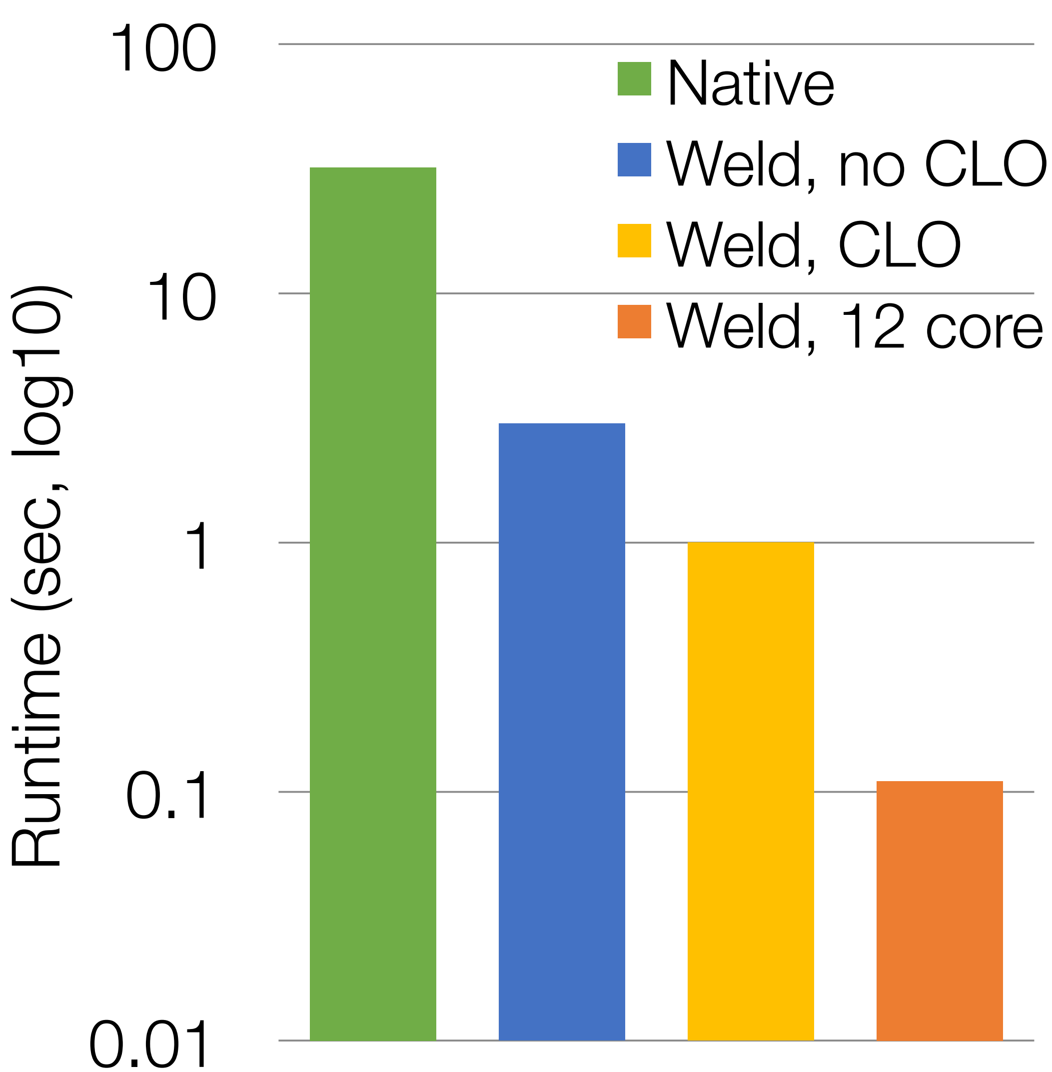
[Slides: Palkar et al., Weld: A Common Runtime for Data Analytics]

Weld Architecture



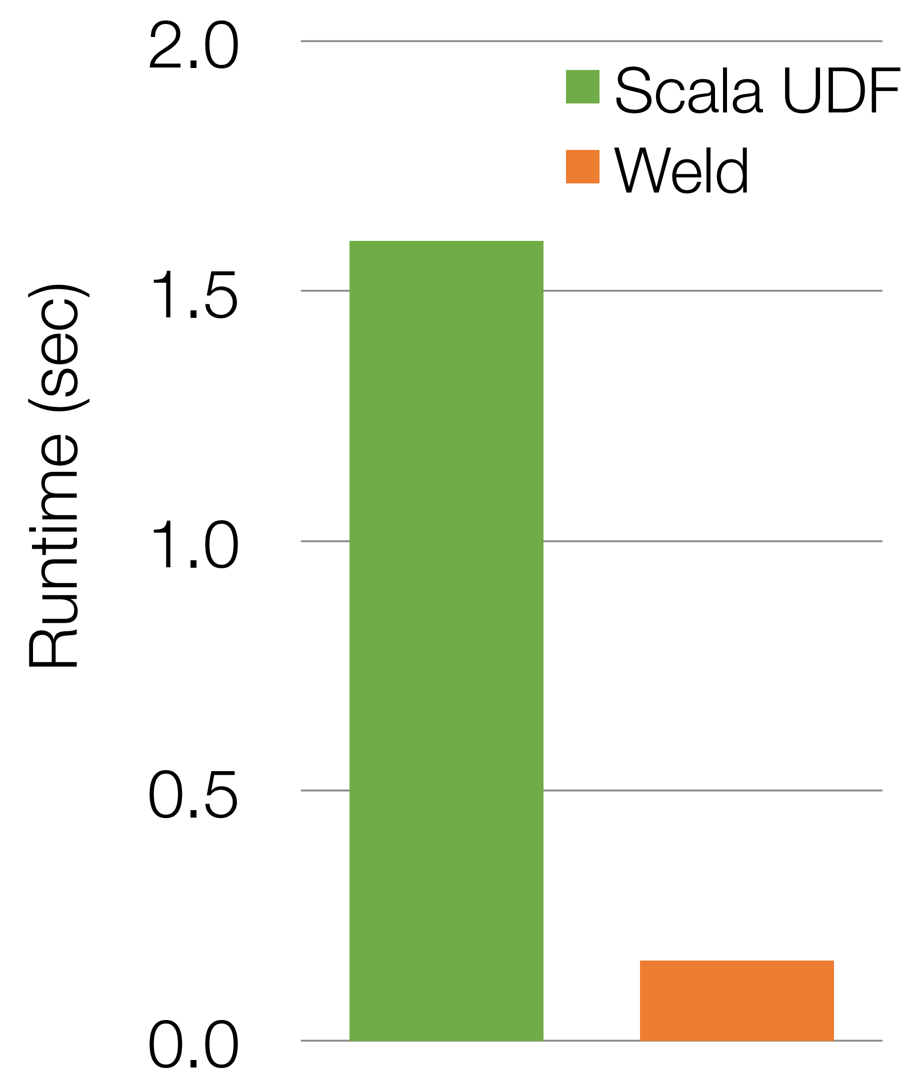
Weld Cross-Library

Pandas + NumPy



CLO = with cross library optimization

Spark SQL UDF



Weld Example

```
squares = map(data, x => x * x)
sum = reduce(data, 0, +)
```

↓

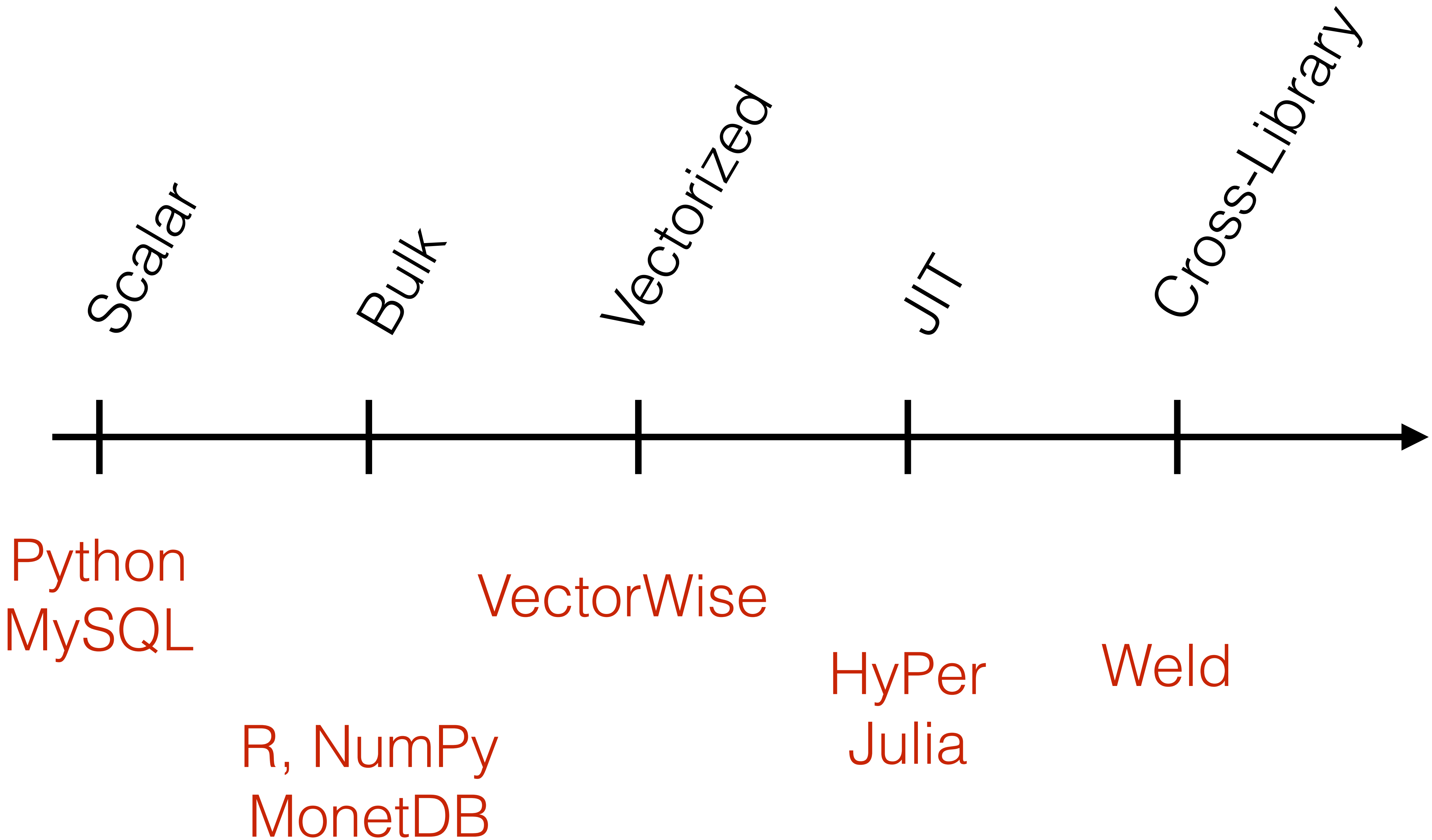
```
bld1 = new vecbuilder[int]
bld2 = new merger[0, +]
for x in data:
    merge(bld1, x * x)
    merge(bld2, x)
```

Differences to Julia?

Optimisation Challenge

- Optimize this:
  - `PLOT(SQL(R(C(BLAS(data)))))`
  - `SQL(Python(TensorFlow(Pandas(NumPy(C(data))))))`

Summary


Scalar
Bulk
Vectorized
JIT
Cross-Library
Python
MySQL
R, NumPy
MonetDB
VectorWise
HyPer
Julia
Weld