

# Parallelizing User-Defined Aggregations using Symbolic Execution

Veselin Raychev, Madanlal Musuvathi, Todd Mytkowicz

Presentation by Tomasz Knopik

1. User-Defined Aggregation and parallelism

2. SYMPLE

a. idea and semantic execution

b. symbolic data types

c. implementation

3. Efficiency

# User-Defined Aggregations

The goal is to answer different user's questions on big data sets, with as much parallelism as possible.

Data set: **query logs**

It's a real life use case for real companies like:

Google, Facebook, Microsoft, Amazon, Ebay etc...

# Example questions we would like to ask

- What is an average time spent by user on our page ?
- How much time does average user spend on our page ?
- What is the most popular post on our page within last 24 hours ?
- What is the average price of products bought by people under 30 ?

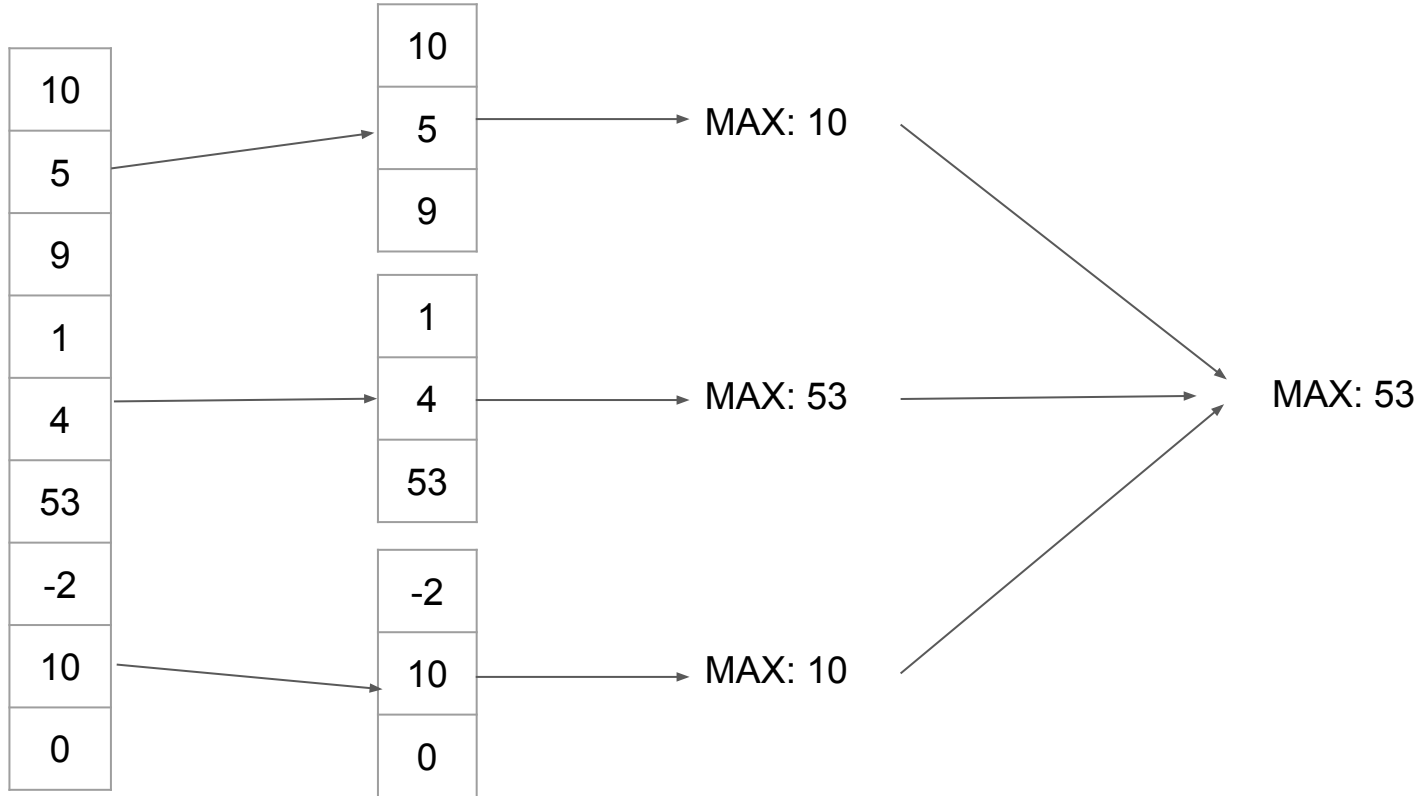
# Easy answers

Those questions are fairly easy to answer. It's only about executing filter, group by queries and SUM, MAX functions

There exist popular implementations with parallelism like:

- SQL engines
- MapReduce with fe. Hadoop

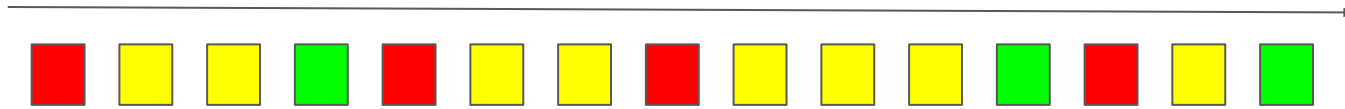
# MAX MapReduce example






# Another example questions we would like to ask

- What is an time spent by user on our page between first entry and the purchase ?
- How many times did user search for product review before the purchase ?
- How much time did user hesitate before ordering Uber with a surge ?

What is the median number of searches for review between first search for the price and the purchase ?



-  Search for price
-  Search for review
-  Purchase

The parallel answer is not obvious.

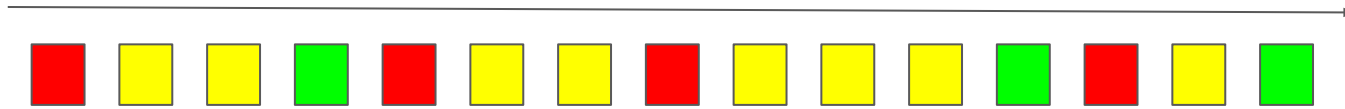
**Problem:** Data dependencies




Answer which we would give at each point depends on what has happened before!

System-supported associative aggregations, such as counting or finding the maximum, are data-parallel and thus these systems optimize their execution, leading in many cases to orders-of-magnitude performance improvements. These optimizations, however, are not possible on arbitrary UDAs



What is the median number of searches for review between first search for the price and the purchase ?



-  Search for price
-  Search for review
-  Purchase

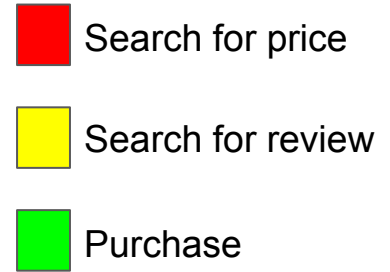
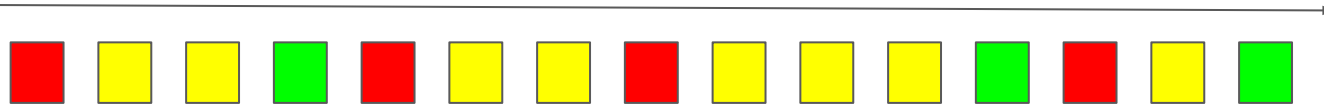
The parallel answer is not obvious.

**Problem:** Data dependencies

Answer which we would give at each point depends on what has happened before!

Complex queries are neither commutative nor associative

What is the median number of searches for review between first search for the price and the purchase ?



The parallel answer is not obvious.

**Problem:** Data dependencies

Answer which we would give at each point depends on what has happened before

To reach the answer we would have to consider a lot of corner cases, the code could become buggy and hard to maintain.

The solution is SYMPLE !

# SYMPLE

SYMPLE is system for performing MapReduce-style groupby-aggregate queries that automatically parallelizes UDAs.

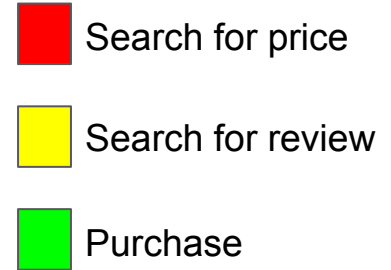
It introduces own data types, similar to standard ones, with which automatically parallelizes sequential code.

What is the median number of searches for review between first search for the price and the purchase ? SEQUENTIAL

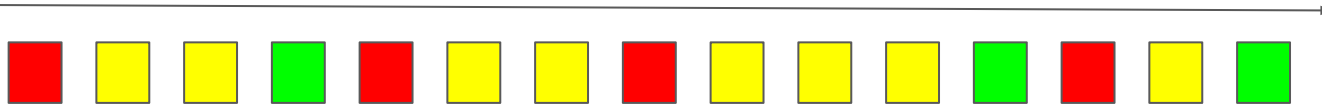


```
bool price_searched = false; int review_searches = 0;
vector<int> results;
```

```
switch record.type:
  case PRICE_SEARCH: price_searched = true; break;
  case REVIEW_SEARCH:
    if (price_searched)
      review_searches++; break;
  case PURCHASE:
    if (price_searched) {
      price_searched = false;
      results.push_back(review_searches);
      review_searches = 0;
    }
}
```



What is the median number of searches for review between first search for the price and the purchase ? SYMPLE



```
SymBool price_searched = false; SymInt review_searches = 0;  
SymVector<SymInt> results;
```

```
switch record.type:
```

```
    case PRICE_SEARCH: price_searched = true; break;
```

```
    case REVIEW_SEARCH:
```

```
        if (price_searched)
```

```
            review_searches++; break;
```

```
    case PURCHASE:
```

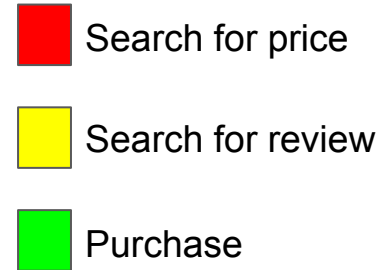
```
        if (price_searched) {
```

```
            price_searched = false;
```

```
            results.push_back(review_searches);
```

```
            review_searches = 0;
```

```
        }
```



# SYMPLE - Idea



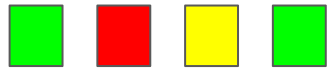
SYMPLE tries to break data dependencies and make use of symbolic execution.



initial state  $\longrightarrow$   $s'$



$U'(S)$   $\longrightarrow$   $S$



$U''(S)$   $\longrightarrow$   $S$  (final state)

Final state =  $U''(U'(s'))$

Note that now the  $U'$  and  $U''$  is created not based on the input, but on the code!

# SYMPLE

To benefit from computing functions  $U$  and  $U'$ , executing them as to be significantly less expensive than processing the records itself.  $U$  and  $U'$  behave as if they were standard UDAs, but starting from unknown state.

At the end, we call  $U$  and  $U'$  sequentially in the reducer.

SYMPLE uses **symbolic execution** to achieve simple and easy to execute (and obviously exact) for  $U$  and  $U'$

# SYMPLE: Symbolic Execution

*In computer science, symbolic execution (also symbolic evaluation) is a means of analyzing a program to determine what inputs cause each part of a program to execute*

SYMPLE with symbolic execution tries to cover all of code branches with simple canonical form of function representation

There is introduced concept of *path constraints* (PC) and *transfer functions* (TF) which are held in the SYMPLE state and represent function such as:

$$\bigwedge_i PC_i(x) \Rightarrow s = TF_i(x)$$

$$\bigvee_i PC_i(x) = \text{true} \quad \text{for all } i \neq j, PC_i \wedge PC_j = \text{false}$$



# Symbolic Execution: Example

```
SymInt Max(K key, List<int> input) {  
  SymInt max = INT_MIN;  
  foreach( e in input )  
    if( max < e )  
      max = e;  
  
  return max;  
}
```

Input: [2, 9, 1, 5, 3, 10, 8, 2, 1]

Chunks:

First = [2, 9, 1]

Second = [5, 3, 10]

Third = [8, 2, 1]

# Symbolic Execution: Example

```
SymInt Max(K key, List<int> input) {  
  SymInt max = INT_MIN;  
  foreach( e in input )  
    if( max < e )  
      max = e;  
  
  return max;  
}
```

Input: [2, 9, 1, 5, 3, 10, 8, 2, 1]

Chunks:

First = [2, 9, 1]

Second = [5, 3, 10]

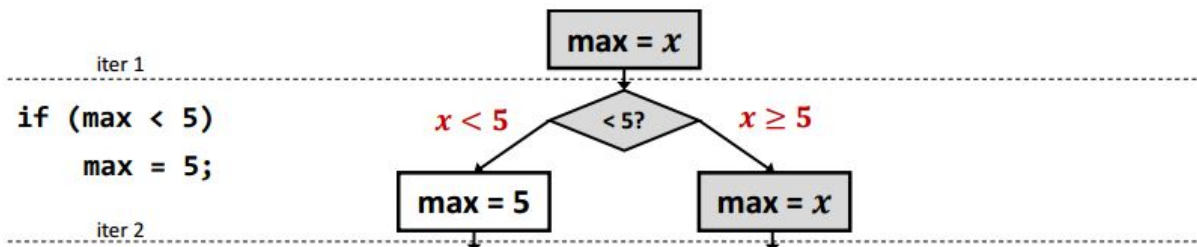
Third = [8, 2, 1]

Result for first chunk is obvious.

How does symbolic execution look like for the next chunks?

# Symbolic Execution: Example

Chunk = [5, 3, 10]



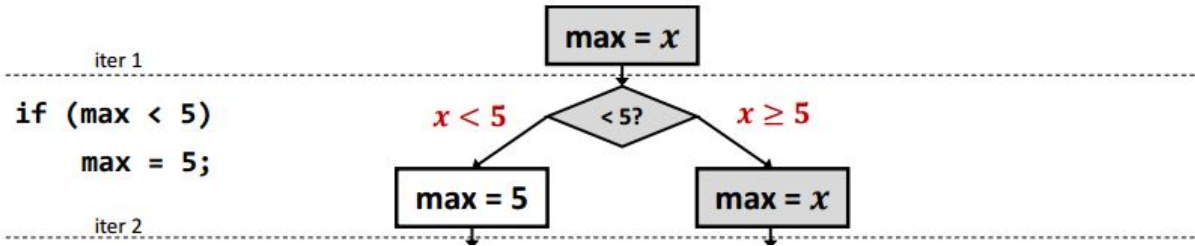
In first iteration, program splits into two branches. SYMPLE creates two path constraints with transfer functions:

$$x < 5 \Rightarrow \text{max} = 5 \wedge x \geq 5 \Rightarrow \text{max} = x$$

Note: in  $x < 5$  constraint SymInt variable `max` is set to concrete value = 5, but in the  $x \geq 5$  constraint, `max` is still symbolic, unknown value (but it has to be greater than 5)

# Symbolic Execution: Example

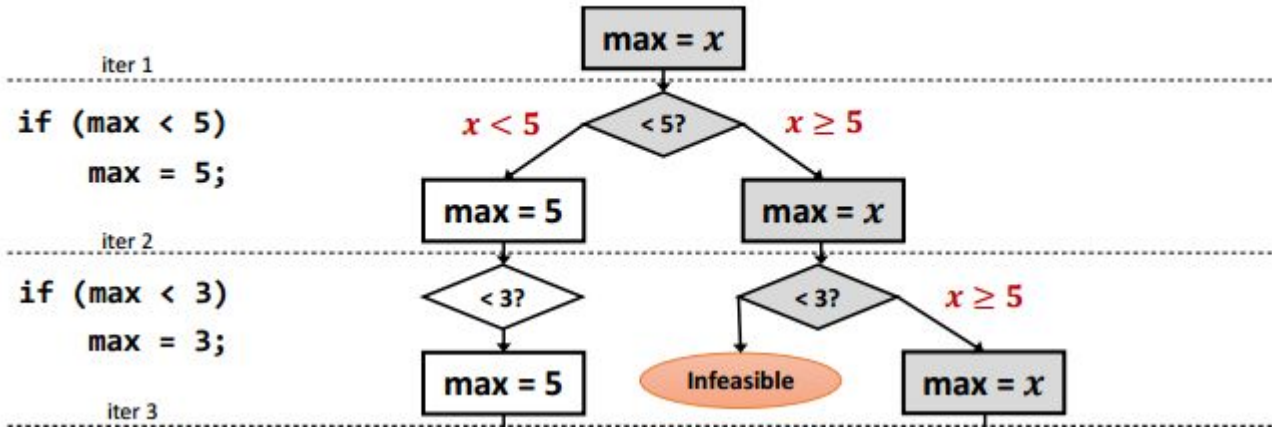
Chunk = [5, 3, 10]



$x < 5 \Rightarrow \text{max} = 5 \wedge x \geq 5 \Rightarrow \text{max} = x$

# Symbolic Execution: Example

Chunk = [5, 3, 10]

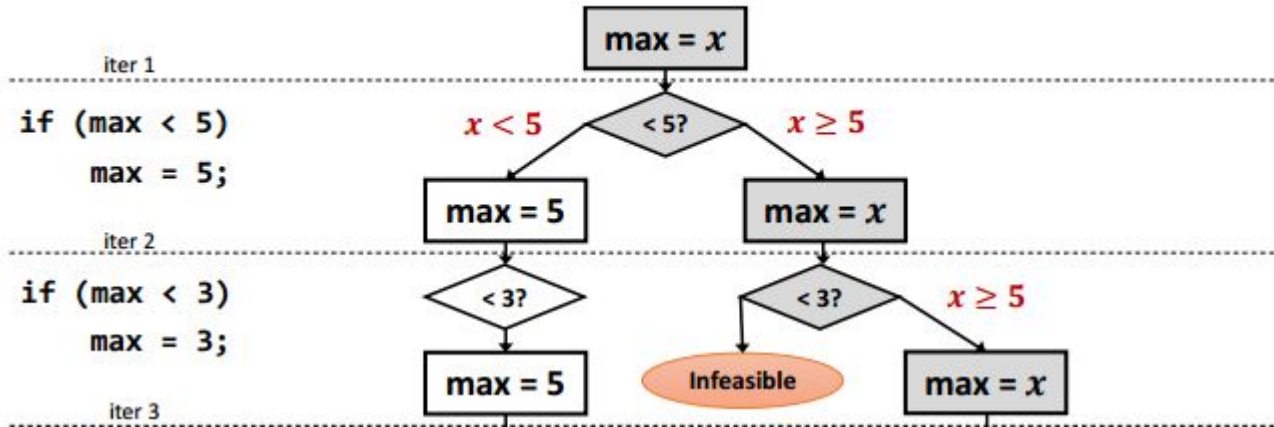


Left branch: nothing changes, no new constraints

$$x < 5 \Rightarrow \text{max} = 5 \wedge x \geq 5 \Rightarrow \text{max} = x$$

# Symbolic Execution: Example

Chunk = [5, 3, 10]

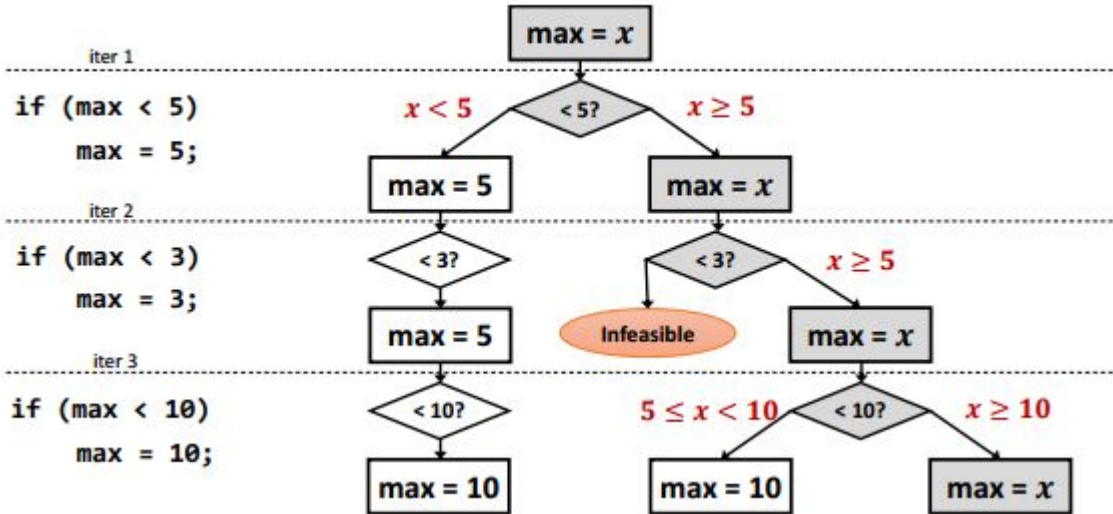


Right branch: `< 3` branch is not feasible, because of `≥ 5` constraint. SYMPLE won't explore `< 3` any more.

$$x < 5 \Rightarrow \text{max} = 5 \wedge x \geq 5 \Rightarrow \text{max} = x$$

# Symbolic Execution: Example

Chunk = [5, 3, 10]

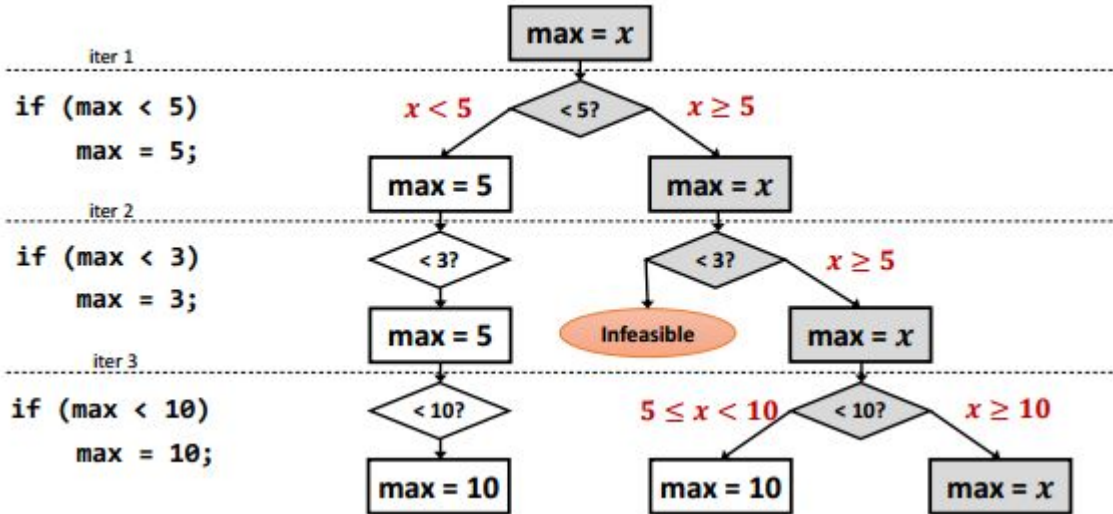


Left branch: SYMLINK recognizes that from the first condition  $x < 5$  so it sees that  $x < 10$  as well. Max gets updated to 10 in left branch constraint

$$x < 5 \Rightarrow max = 10 \wedge x \geq 5 \Rightarrow max = x$$

# Symbolic Execution: Example

Chunk = [5, 3, 10]



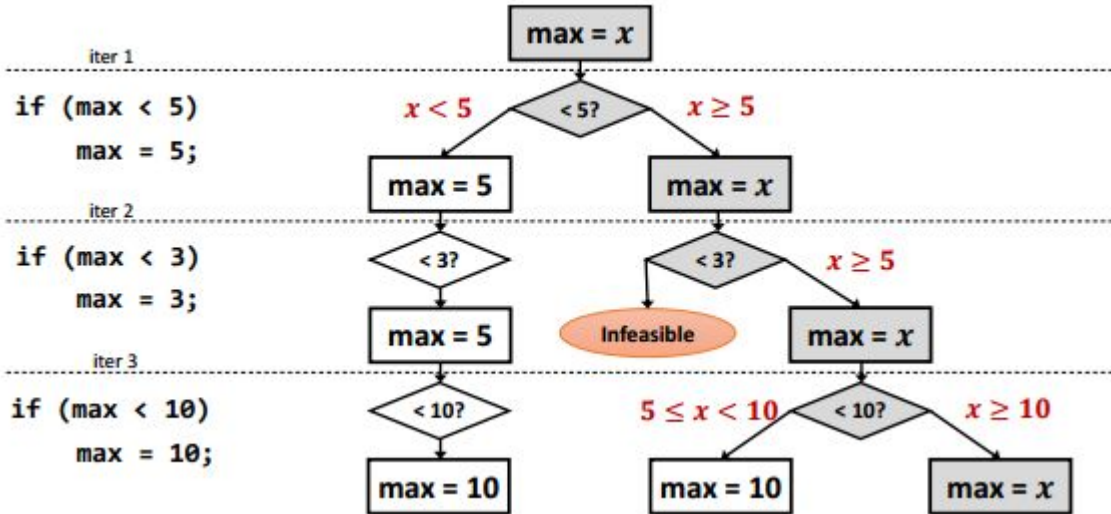
Right branch: SYMLINK brakes right branch constraint into two.

$$\begin{aligned}x < 5 &\Rightarrow \text{max} = 10 \wedge \\10 > x \geq 5 &\Rightarrow \text{max} = 10 \wedge \\x \geq 10 &\Rightarrow \text{max} = x\end{aligned}$$



# Symbolic Execution: Example

Chunk = [5, 3, 10]



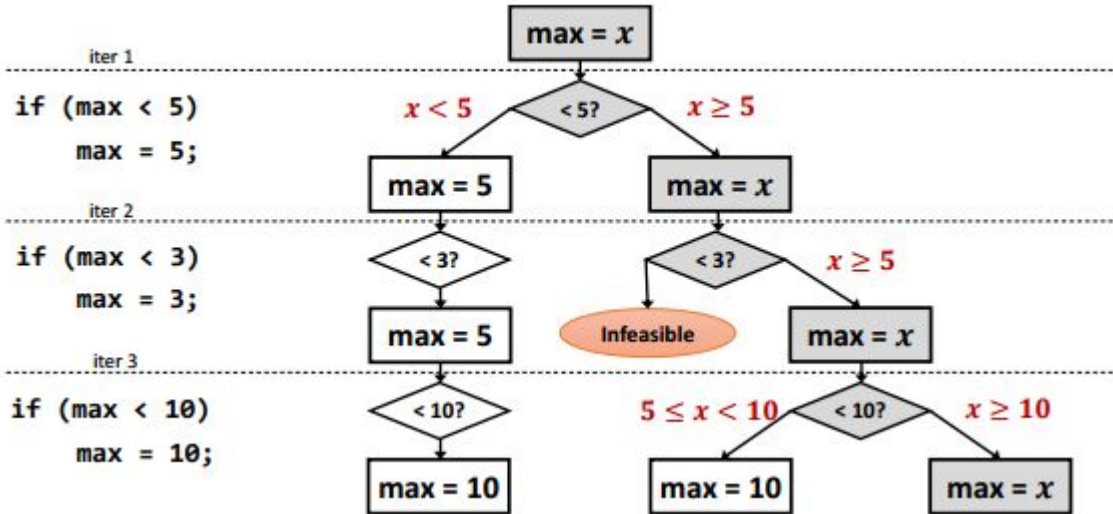
Right branch: SYMLINK brakes right branch constraint into two.

It recognizes as well that two first constraints can be merged into one

$$\begin{aligned}x < 5 &\Rightarrow \max = 10 \wedge \\10 > x \geq 5 &\Rightarrow \max = 10 \wedge \\x \geq 10 &\Rightarrow \max = x\end{aligned}$$

# Symbolic Execution: Example

Chunk = [5, 3, 10]

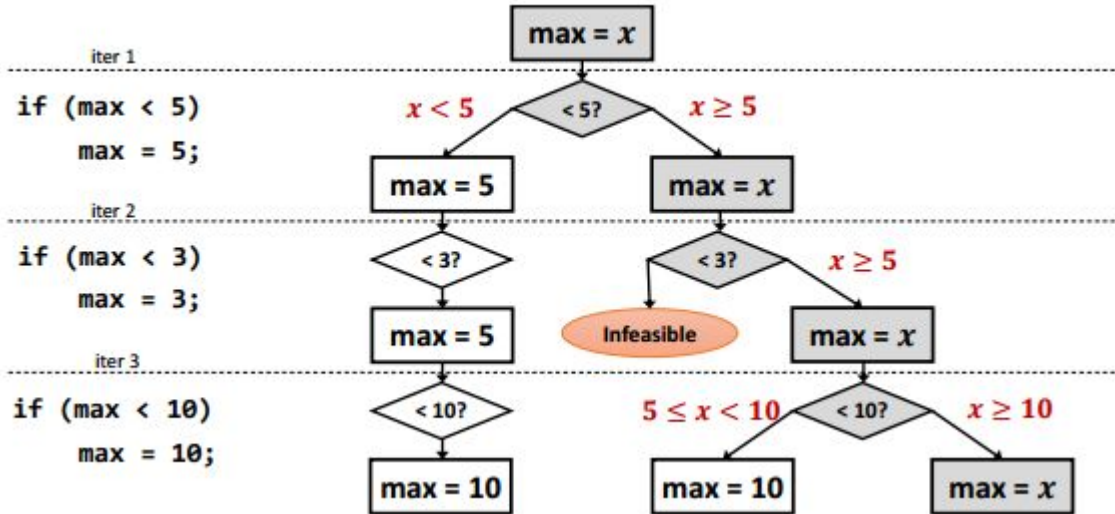


Right branch: SYMLINK brakes right branch constraint into two.  
It recognizes as well that two first constraints can be *merged* into one

$10 > x \Rightarrow \text{max} = 10 \wedge$   
 $x \geq 10 \Rightarrow \text{max} = x$

# Symbolic Execution: Example

Chunk = [5, 3, 10]



Final representation of function:

$10 > x \Rightarrow \text{max} = 10 \wedge$

$x \geq 10 \Rightarrow \text{max} = x$

The get the result:

- Repeat the action for the last chunk (parallel)
- Reduce the outcome by function composition

How does SYMPLE know whether it can merge paths or when the path is not feasible?

# Symbolic Data Types

- `SymInt`
- `SymEnum`
- `SymBool`
- `SymVect`
- `SymPred<T>`

# SymInt

- Symbolic (doesn't always hold specific value) version of C++ int
- Supports only operations with concrete values, not with another SymInt (could be possible, but would require too much computations to be fast enough)

# SymInt

**Canonical form:**  $s = (lb, ub, a, b)$

Under path constraint  $lb \leq x \leq ub$ , the value of SymInt is  $ax + b$

It allows us to operate on the variable, even if it's not defined yet (symbolic) and explains why only operations with concrete values are allowed

## Decision Procedures

When comparing a SymInt with another constant, the two outcomes split the interval  $[lb, ub]$  into two (possibly empty) intervals.

Condition like  $s \leq c$  holds when  $a*x + b \leq c$  does

This decision creates two new paths in our program:

- $[lb, (c - b) / a]$  when  $s \leq c$  holds
- $((c - b) / a, ub]$  when  $s \leq c$  doesn't hold

A path is not feasible when condition creates SymInt which an empty interval



# SymInt

## **Merging Path Constraints:**

If SYMPLE tries to merge two path constraints, it checks whether merging can take place. It's possible when intervals intersect, the new boundaries are extreme points

# SymEnum

**Canonical Form:**  $(S, \text{bound}, c): x \in S \Rightarrow v = \text{bound} ? c : x$

- $S$  - bit vector of any values (options of enum)
- $\text{bound}$  - *true* if enum has concrete value
- $c$  - concrete value of enum (gained from an assignment)

When  $\text{bound}$  is *true* SymEnum is as fast as C++ enum type.

## Decision Procedures

When a symbolic SymEnum which can take any value from a set  $S$  is compared with a constant  $c$ , there are two possible paths corresponding to the two sets  $S \cap \{c\}$  and  $S/\{c\}$ . If either of these sets is empty the corresponding path is not feasible.

# SymEnum

## Merging Path Constraints

Two path constraints  $x \in S1$  and  $x \in S2$  can be merged into  $x \in S1 \cup S2$

# SymBool

- SymEnum with true and false possibilities
- Overloaded operators

# SymVector<T>

- Similar to C++ vector
- Concretizes its elements whenever variable which those elements rely on gets concrete value

# Additional data types

Users can create structs from other Sym data types

Additionally there exists ability to create own data types, but the authors haven't found use case for using it yet

# How does SYMPLE track the paths?

It records the vector of paths taken, keeping them in lexicographic order.

For example: record 01 in vector means that path took then branch in the first if and else branch in the second conditional statement

# Paths Explosion

SYMPLE tries to cover all paths of the program and create versatile function describing all of them. However, there could exist code with number of paths grows too fast to keep the path constraints and transfer functions concise.

Solutions:

- Paths merging
- Exclusion of infeasible paths
- Detecting possible paths explosions
- Fallback to sequential computation in case of paths explosion



# SYMPLE & MapReduce

SYMPLE doesn't rely on the way in which data input is splitted, it can be easily plugged into existing implementations of MapReduce. It just has to keep track of data shuffling which takes place in mapper, as we have to keep track of the records order.

SYMPLE moves a lot of work to mappers from the reducers. It increases efficiency as sequential part in reducers is limited to simple and small functions calls

# Results

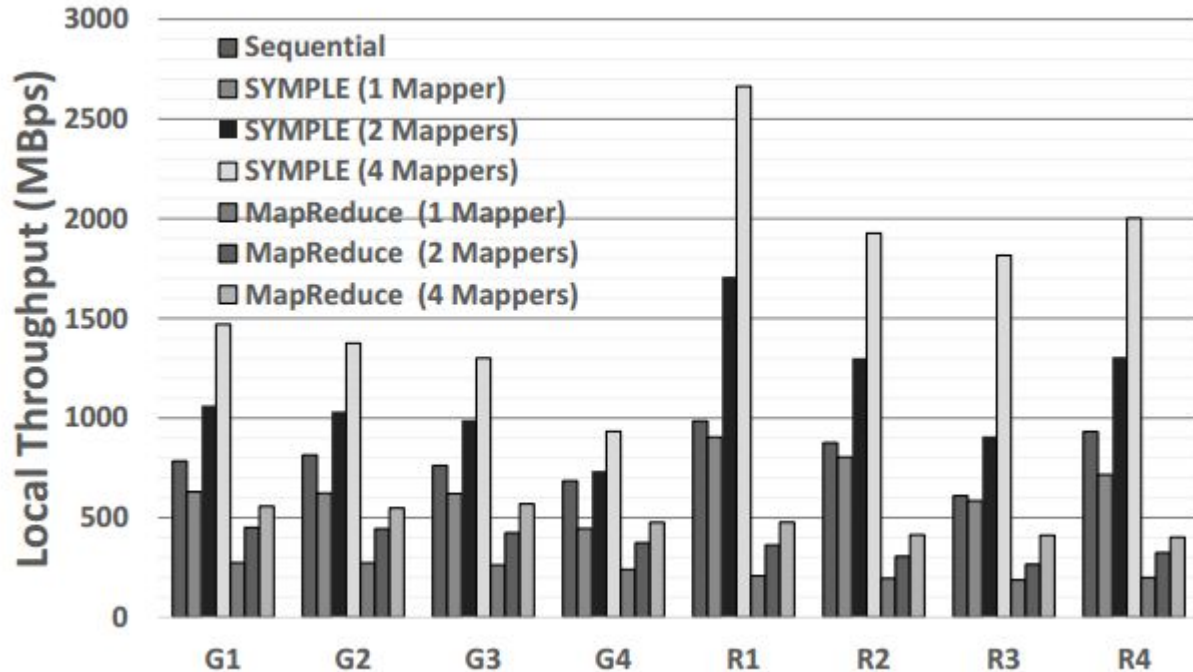
# Data sets

- github repository operations (419 GB)
- Amazon Redshift benchmark data (1.2 TB)
- Bing search engine (300 GB)
- All tweets from 24 hours period (1.23 TB)

# Queries

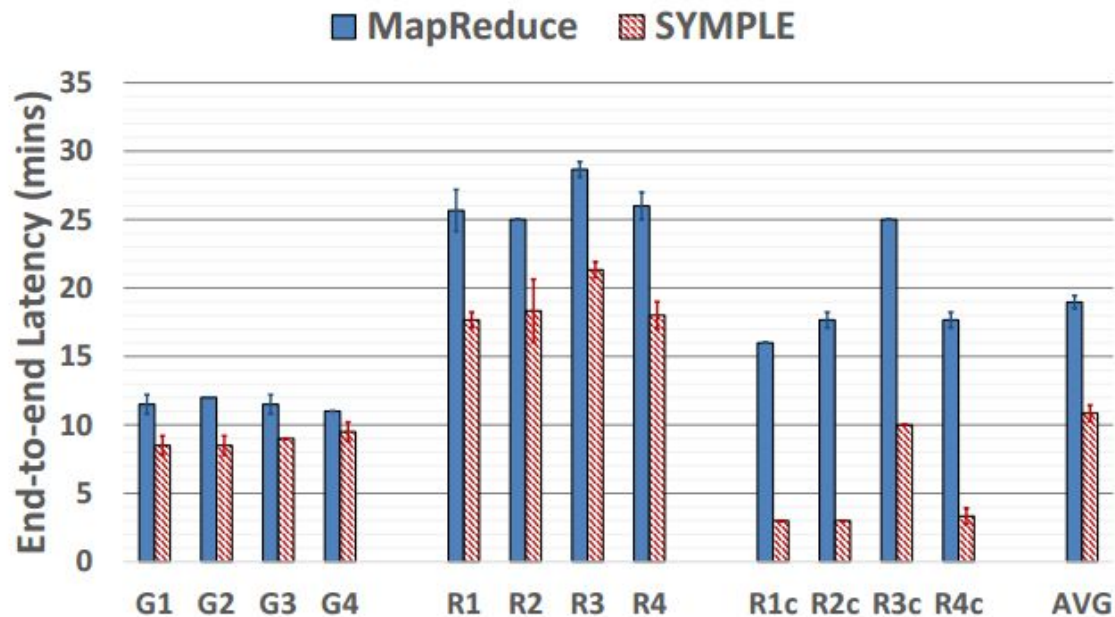
ID	Description	# Groups	Sym Types Used		
			Enum	Int	Pred
<b>419 GB List of GitHub operations on repositories from Feb 2011 to Sep 2014.</b>					
G1	Return all repositories with only push commands	12M	y		
G2	All operations on a repository directly preceding a delete operation	12M		y	
G3	Number of operations executed on a repository between pull open and close	12M	y	y	
G4	The time between branch deletion and branch creation in a repository	22M	y	y	
<b>300GB of Query Logs from the Bing search engine containing 1.9 billion queries</b>					
B1	Outages: more than 2 minutes with no successful query by any user	1		y	
B2	Outages per geographic area of the query (local outages)	*		y	
B3	Number of queries in a session per user ( $\leq 2$ minutes between queries)	*		y	y
<b>1.23TB of logs from Twitter that represent all tweets in a 24 hour interval</b>					
T1	Spam learning speed — no. queries not marked as spam, followed by at least 5 queries marked as spam per hashtag	*	y	y	
<b>1.2TB of ad impression logs from RedShift benchmark</b>					
R1	Number of impressions per advertiser	10K		y	
R2	List of advertisers operating only in a single country	10K	y		y
R3	Cases for advertiser when their ads were not showing for more than 1 hour	10K		y	
R4	Lengths of runs for which only a single campaign by an advertiser is shown	10K		y	y

# Multi-core local machine (github + redshift)



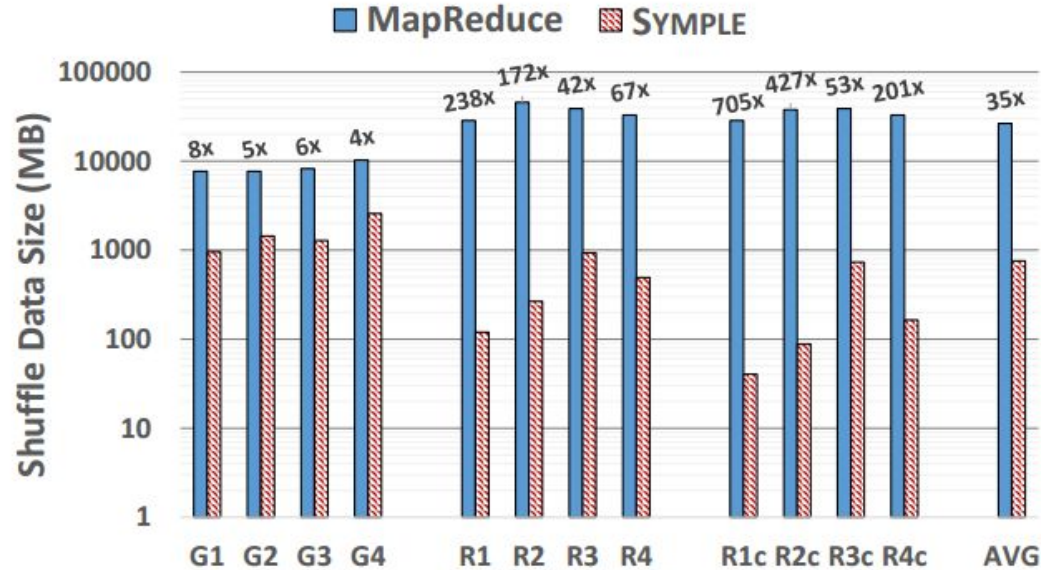
Size of data limited to 4 GB at time to avoid I/O dist limitations

# Amazon 4 CPUs machine



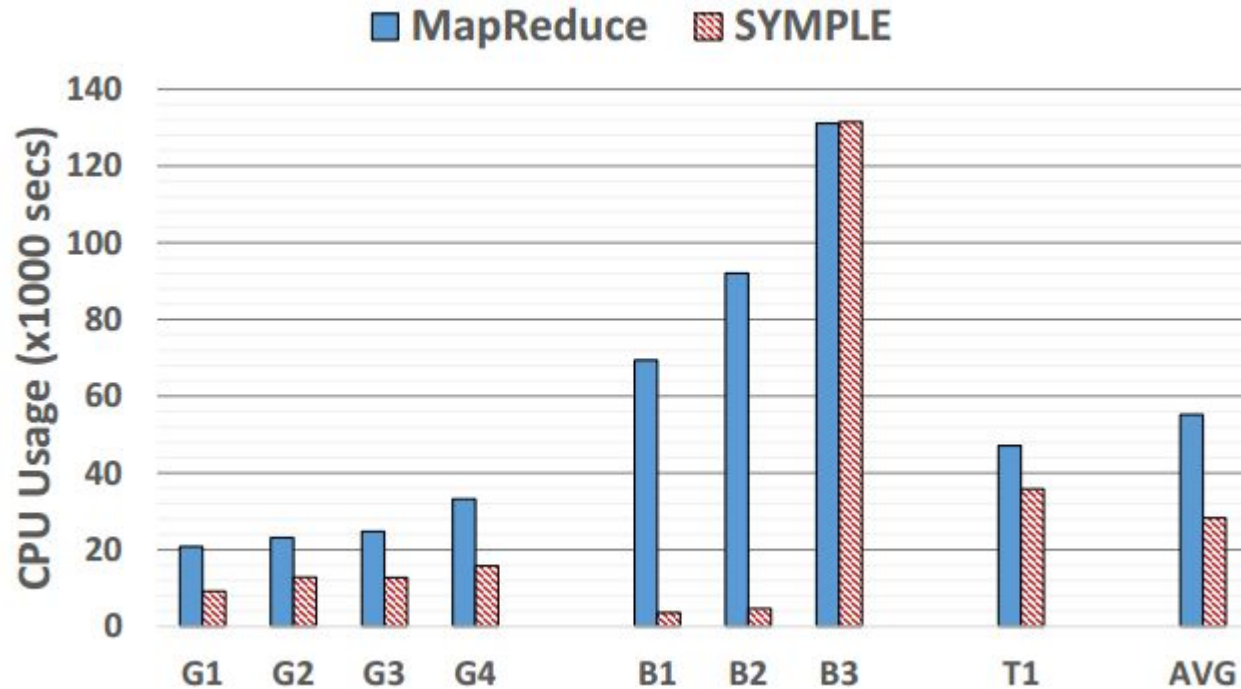
**Figure 5.** Amazon EMR end-to-end job latency.

# Amazon 4 CPUs machine (shuffle data size)



**Figure 6.** Amazon EMR shuffle data reduction. Note the log  $y$  axis.

# 380-nodes Hadoop cluster





*'In [15], a symbolic parallel engine (SYMPLE) is proposed in order to automatically parallelize User Defined Aggregations (UDAs) that are not necessarily commutative. Although interesting, the proposed framework lacks guarantees for efficiency and accuracy in the sense that it is up to users to encode a function as SYMPLE UDA. Moreover, symbolic execution may have path explosion problem.'*

Symmetric and Asymmetric Aggregate Function in Massively Parallel Computing

Thank you

Q & A

