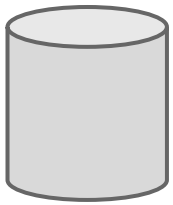
Parallelizing user-defined aggregations using symbolic execution

Veselin Raychev ETH Zurich
Madanlal Musuvathi Microsoft Research
Todd Mytkowicz Microsoft Research

Goal

Answer questions with Big Data

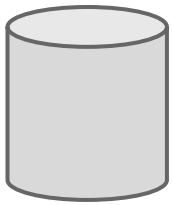
Example:
query logs



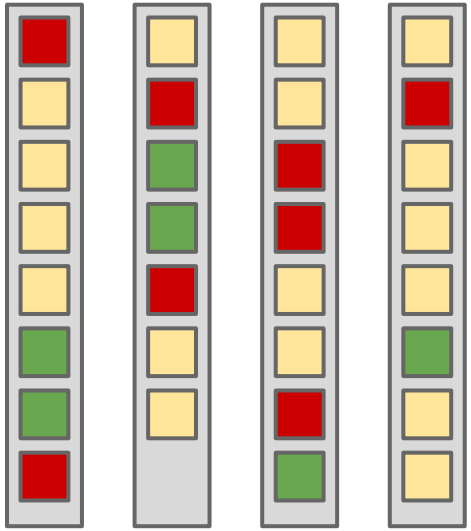
Goal

Answer questions with Big Data

Example:
query logs

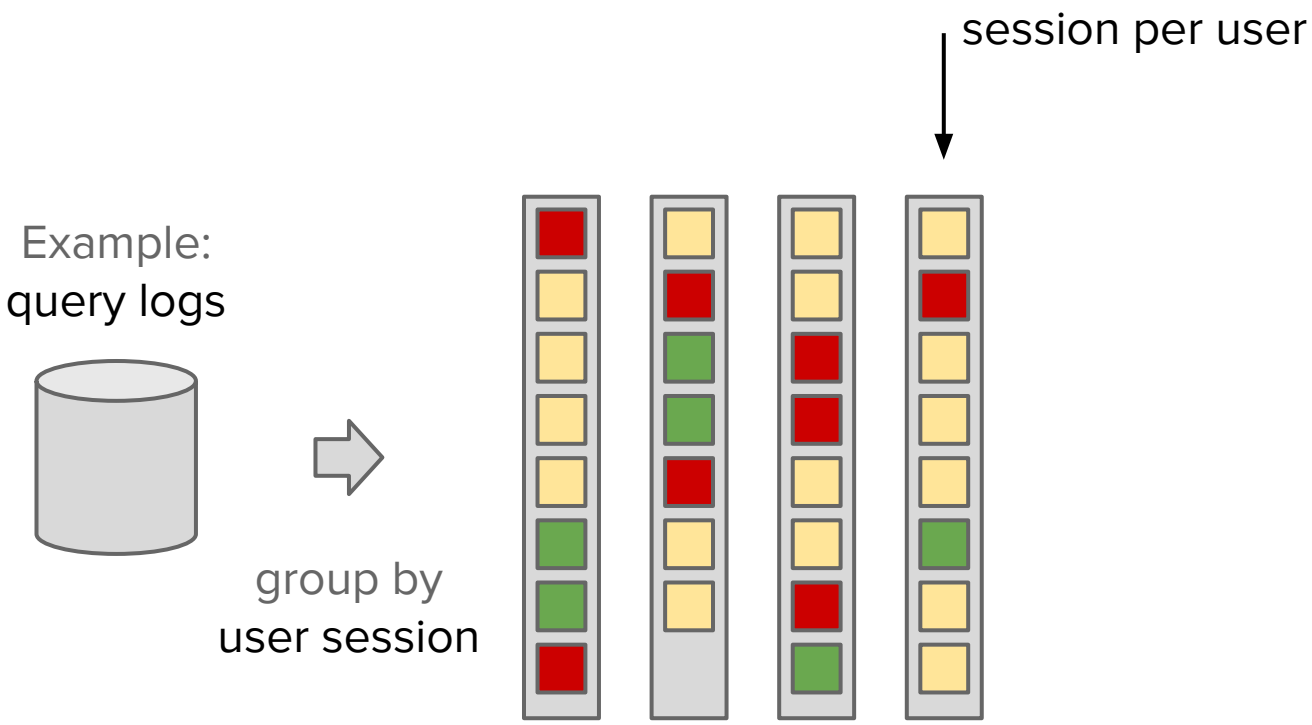


group by
user session



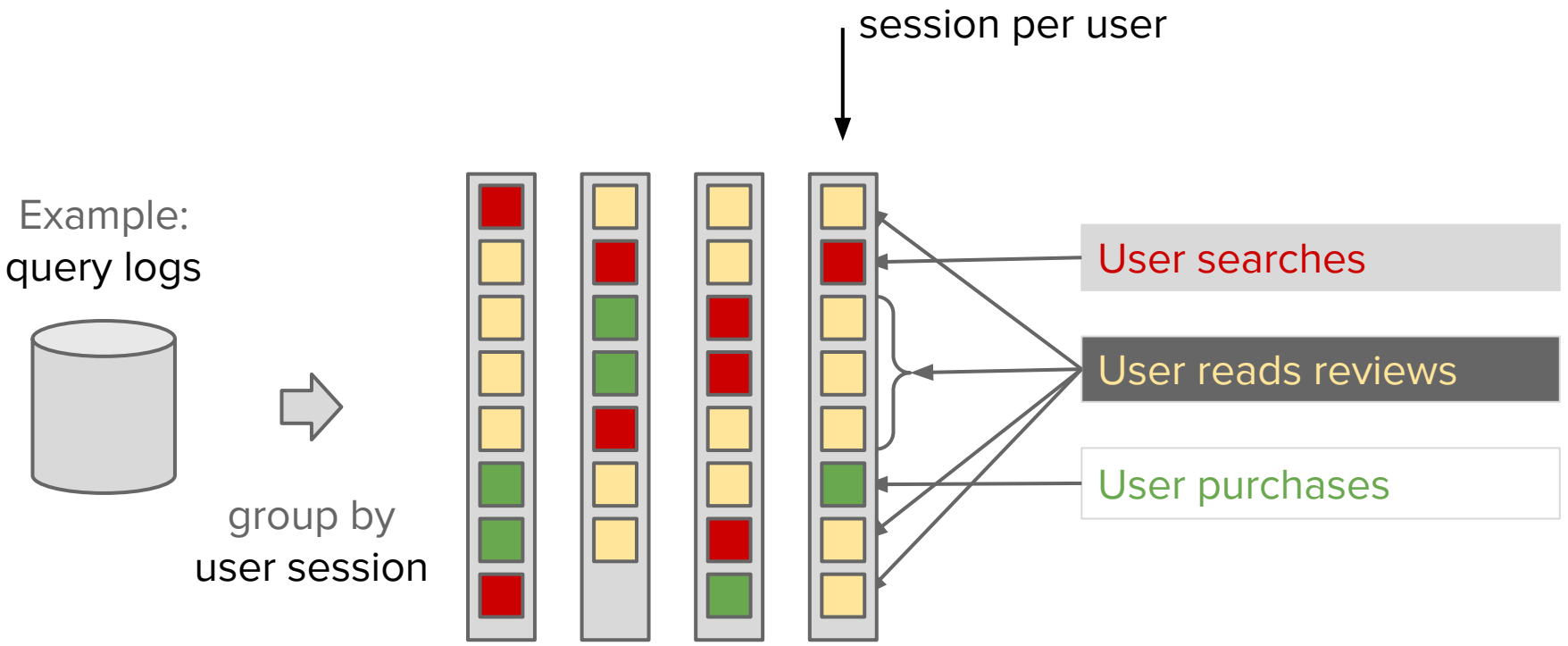
Goal

Answer questions with Big Data



Goal

Answer questions with Big Data



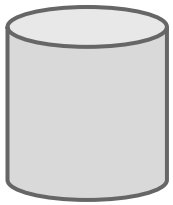
Goal

Answer questions with Big Data

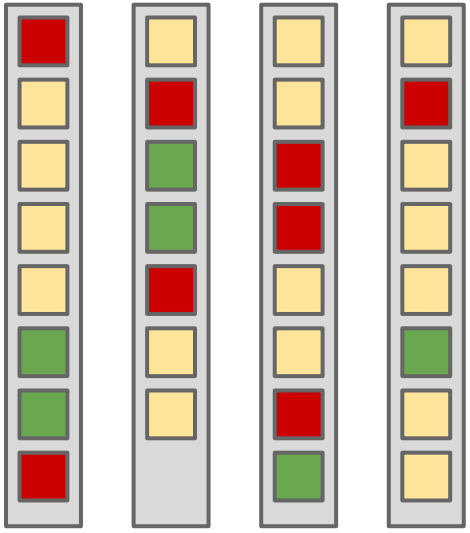
?

Queries

Example:
query logs



group by
user session



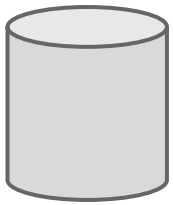
Goal

Answer questions with Big Data

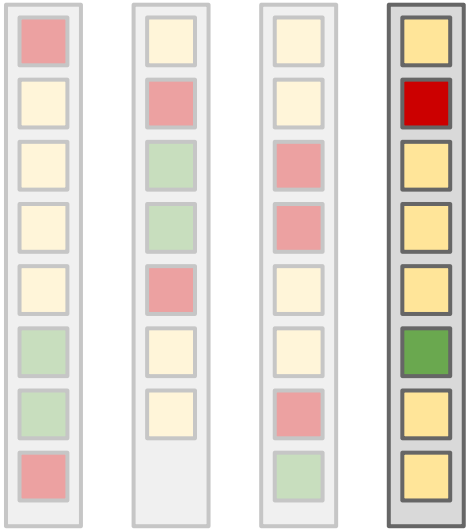
?

Queries

Example:
query logs



group by
user session



Goal

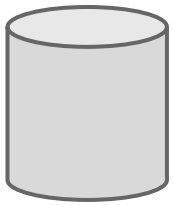
Answer questions with Big Data

?

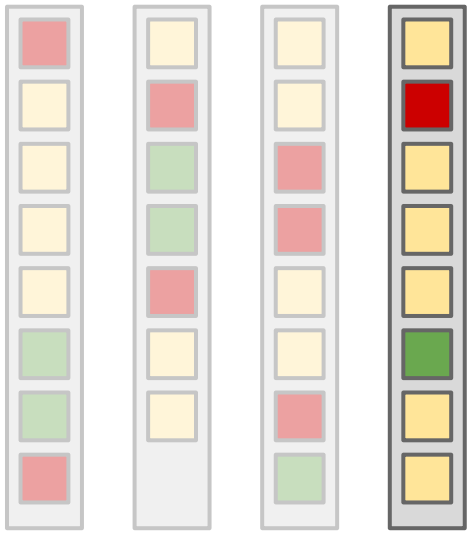
Queries

1. Number of reviews per user session

Example:
query logs



group by
user session



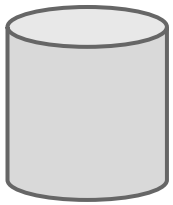
Goal

Answer questions with Big Data

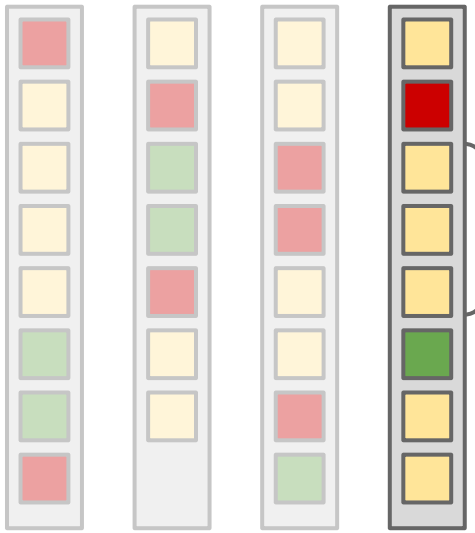
?

Queries

Example:
query logs



group by
user session



1. Number of reviews per user session

2. Number of reviews between a search and a purchase

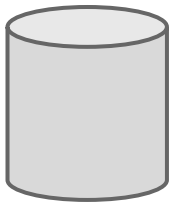
Goal

Answer questions with Big Data

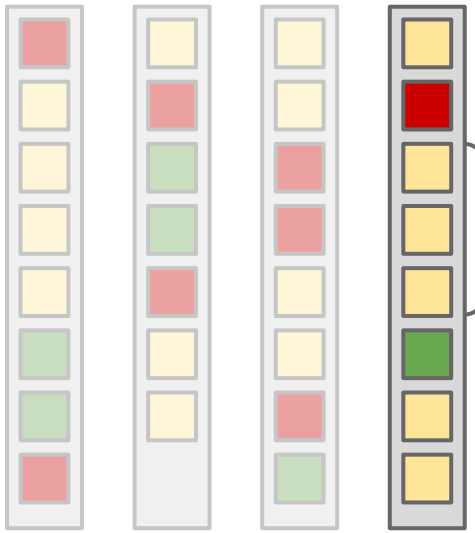
?

Queries

Example:
query logs



group by
user session



1. Number of reviews per user session
2. Number of reviews between a search and a purchase
3. Time between a search and a purchase
4. Searches not followed by a purchase

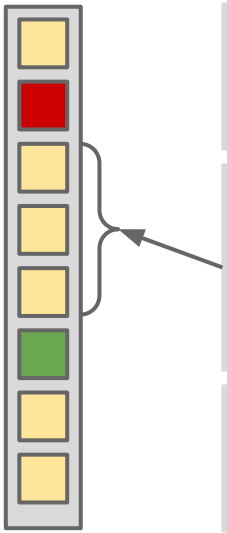
What we do?

Parallelize such queries on

a sequence of records

?

Queries



1. Number of reviews per user session
2. Number of reviews between a search and a purchase
3. Time between a search and a purchase
4. Searches not followed by a purchase

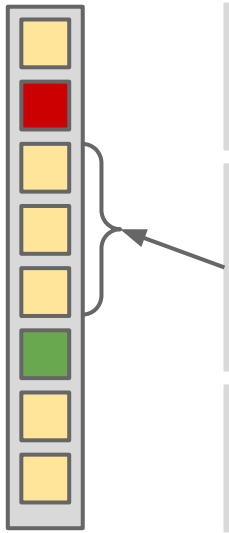
What we do?

Parallelize such queries on

a sequence of records

?

Queries



1. Number of reviews per user session
2. Number of reviews between a search and a purchase
3. Time between a search and a purchase
4. Searches not followed by a purchase

Our tool **SYMPLE**
runs such queries on
multiple CPUs
or multiple machines

What we do?

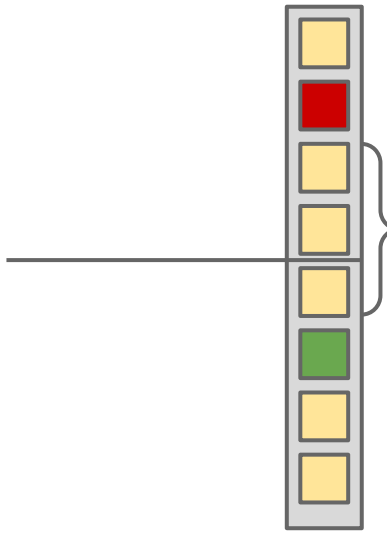
Parallelize such queries on

a sequence of records

?

Queries

Our tool **SYMPLE**
runs such queries on
multiple CPUs
or multiple machines



1. Number of reviews per user session
2. Number of reviews between a search and a purchase
3. Time between a search and a purchase
4. Searches not followed by a purchase

What we do?

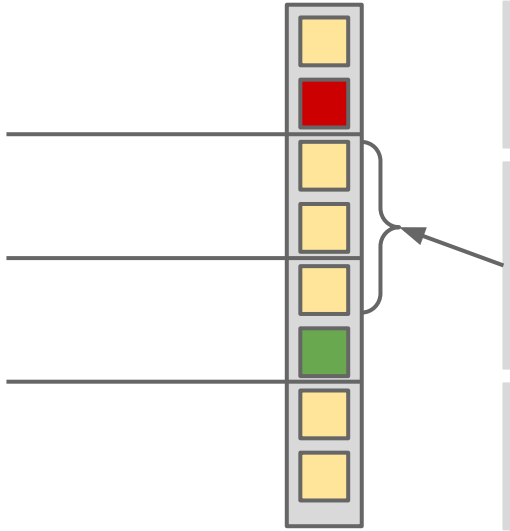
Parallelize such queries on

a sequence of records

?

Queries

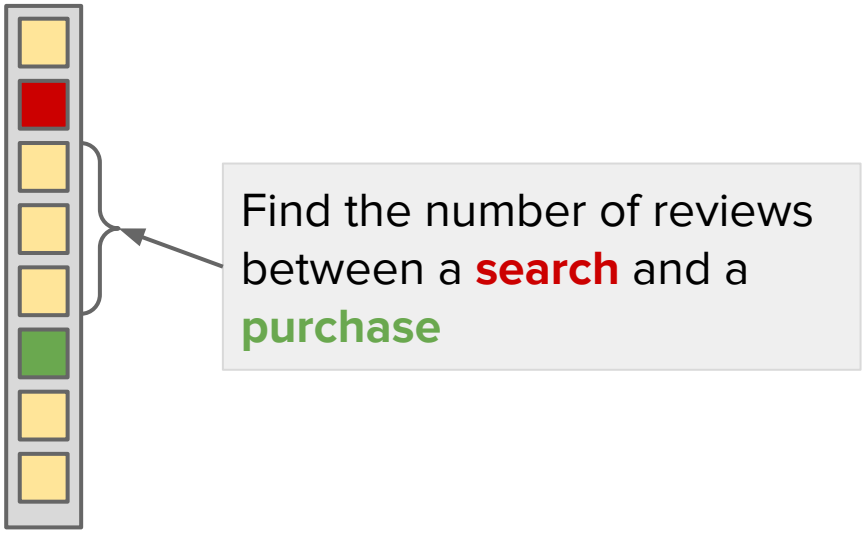
Our tool **SYMPLE**
runs such queries on
multiple CPUs
or multiple machines



1. Number of reviews per user session
2. Number of reviews between a search and a purchase
3. Time between a search and a purchase
4. Searches not followed by a purchase

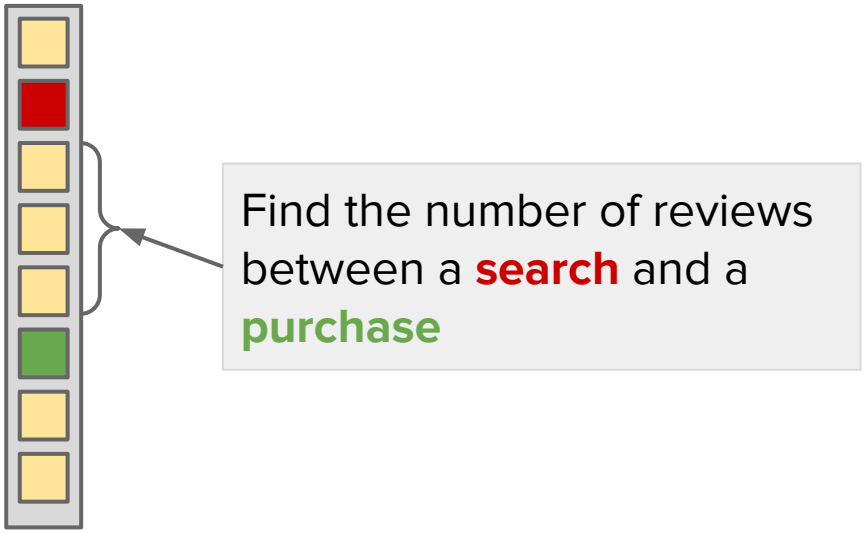
Using SYMPLE

Example query



Using SYMPLE

Example query

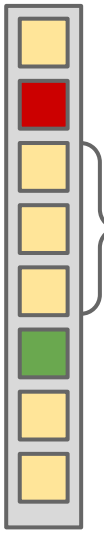



```
foreach record in user records:
    switch record.type:
        case SEARCH:
        case REVIEW:
        case PURCHASE:
```

Using SYMPLE

Example query

```
int num_reviews = 0;
```

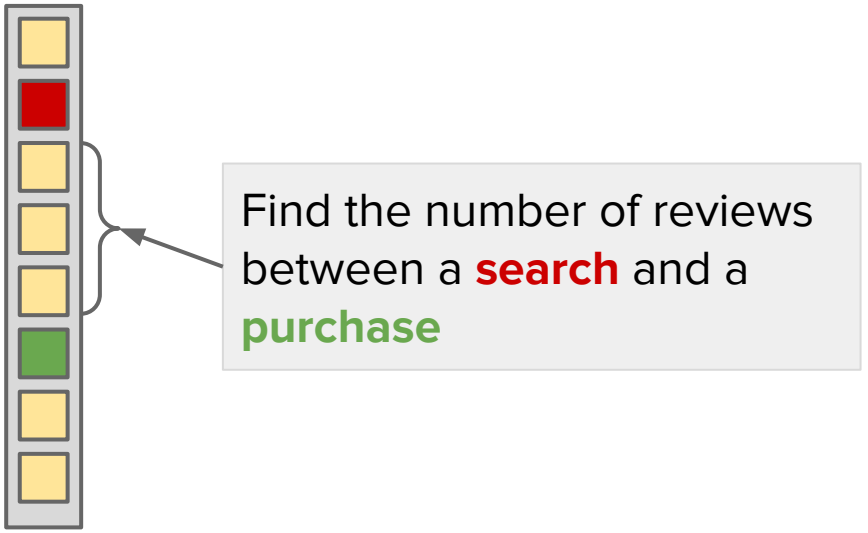


Find the number of reviews between a **search** and a **purchase**

```
foreach record in user records:
    switch record.type:
        case SEARCH:        num_reviews = 0;
        case REVIEW:        num_reviews++;
        case PURCHASE:
```

Using SYMPLE

Example query

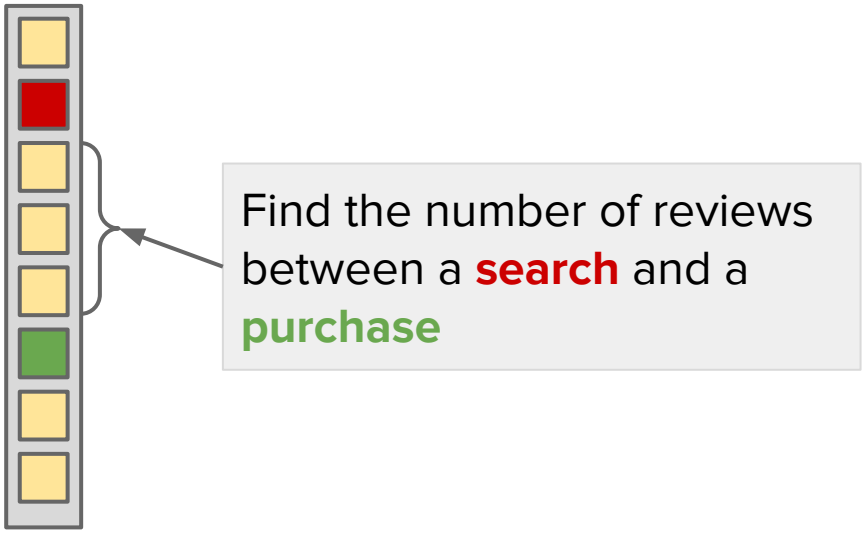



```
int num_reviews = 0;
bool search_done = false;
```

```
foreach record in user records:
    switch record.type:
        case SEARCH:        num_reviews = 0;   search_done = true;
        case REVIEW:        num_reviews++;
        case PURCHASE:      if search_done:
                                search_done = false;
```

Using SYMPLE

Example query

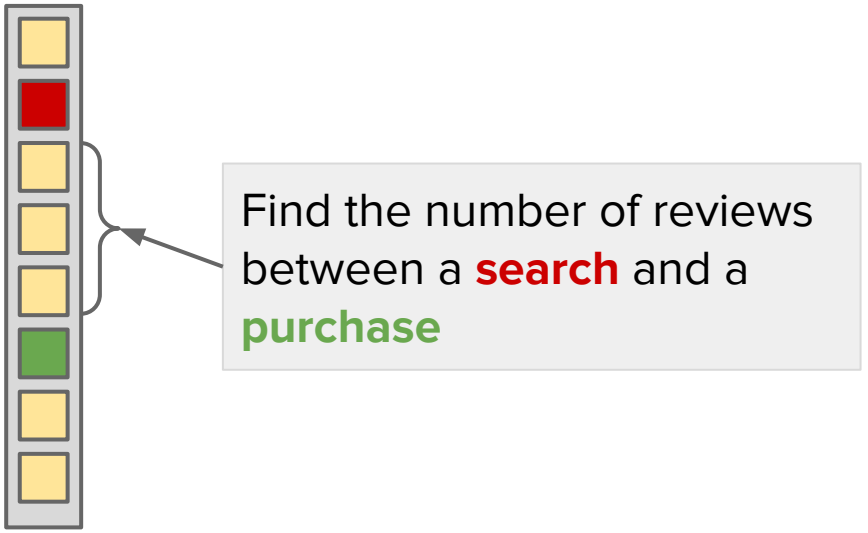



```
int num_reviews = 0;
bool search_done = false;
vector<int> result;
```

```
foreach record in user records:
    switch record.type:
        case SEARCH:        num_reviews = 0;   search_done = true;
        case REVIEW:        num_reviews++;
        case PURCHASE:      if search_done:
                                search_done = false;
                                result.push_back(num_reviews);
```

Using SYMPLE

Example query



SymInt num_reviews = 0;

SymBool search_done = false;

SymVector<**SymInt**> result;

```
foreach record in user records:
    switch record.type:
        case SEARCH:        num_reviews = 0;   search_done = true;
        case REVIEW:        num_reviews++;
        case PURCHASE:      if search_done:
                                search_done = false;
                                result.push_back(num_reviews);
```

Using SYMPLE

Example query

SymInt num_reviews = 0;

SymBool search_done = false;

SymVector<**SymInt**> result;

```
foreach record in user records:
    switch record.type:
        case SEARCH:        num_reviews = 0;   search_done = true;
        case REVIEW:        num_reviews++;
        case PURCHASE:      if search_done:
                                search_done = false;
                                result.push_back(num_reviews);
```

Benefits:

- same code for sequential and parallel computation
- not introducing bugs
- easier maintenance

Outline

1. Symbolic parallelism

2. Data types

3. Evaluation

Outline

1. Symbolic parallelism

2. Data types

3. Evaluation

Notation

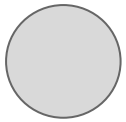
```
int num_reviews = 0;
bool search_done = false;
vector<int> result;
```

```
switch record.type:
    case SEARCH:        num_reviews = 0;   search_done = true;
    case REVIEW:        num_reviews++;
    case PURCHASE:      if search_done:
                            search_done = false;
                            result.push_back(num_reviews);
```

Notation

```
int num_reviews = 0;
bool search_done = false;
vector<int> result;
```

State s

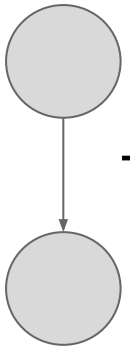


```
switch record.type:
    case SEARCH:        num_reviews = 0;   search_done = true;
    case REVIEW:        num_reviews++;
    case PURCHASE:      if search_done:
                            search_done = false;
                            result.push_back(num_reviews);
```

Notation

```
int num_reviews = 0;
bool search_done = false;
vector<int> result;
```

State s



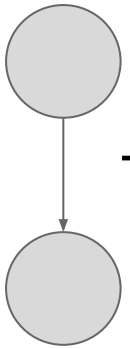
Transition for `record`

```
switch record.type:
    case SEARCH:        num_reviews = 0;   search_done = true;
    case REVIEW:        num_reviews++;
    case PURCHASE:      if search_done:
                            search_done = false;
                            result.push_back(num_reviews);
```

Notation

```
int num_reviews = 0;
bool search_done = false;
vector<int> result;
```

State s



Transition for record

Next state: s′ = t(s)

```
switch record.type:
    case SEARCH:        num_reviews = 0;   search_done = true;
    case REVIEW:        num_reviews++;
    case PURCHASE:      if search_done:
                            search_done = false;
                            result.push_back(num_reviews);
```

Why is this hard to parallelize?

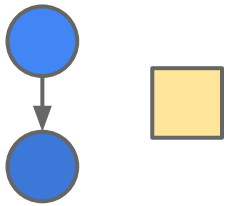
Records

Initial state 

Why is this hard to parallelize?

Records

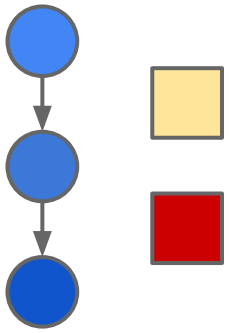
Initial state



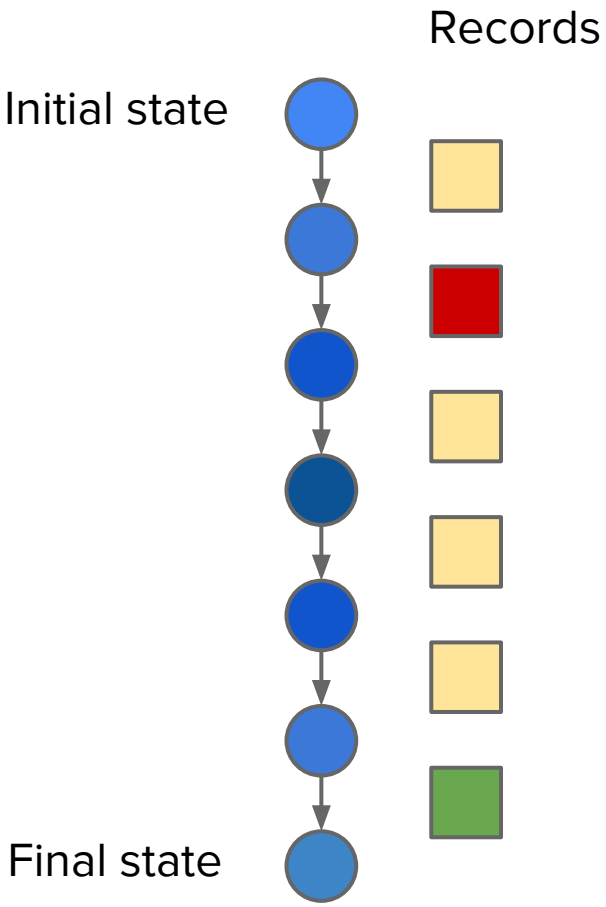
Why is this hard to parallelize?

Records

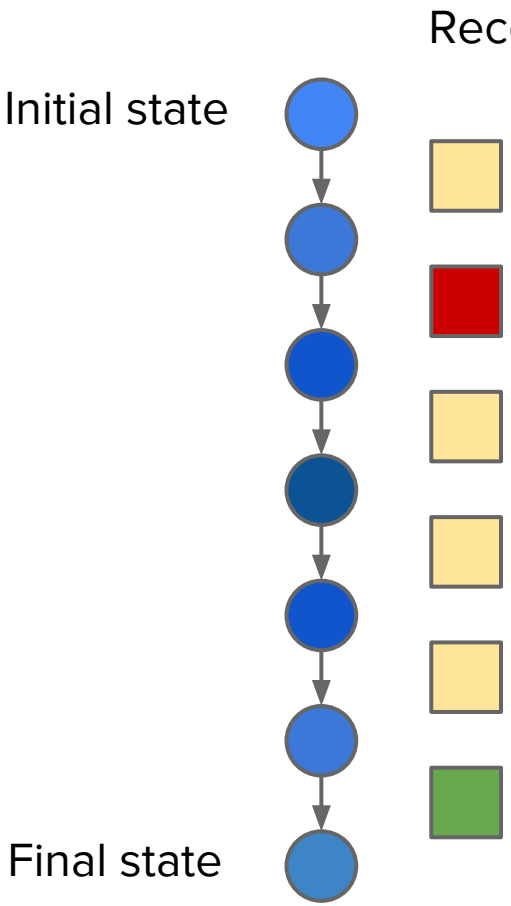
Initial state



Why is this hard to parallelize?

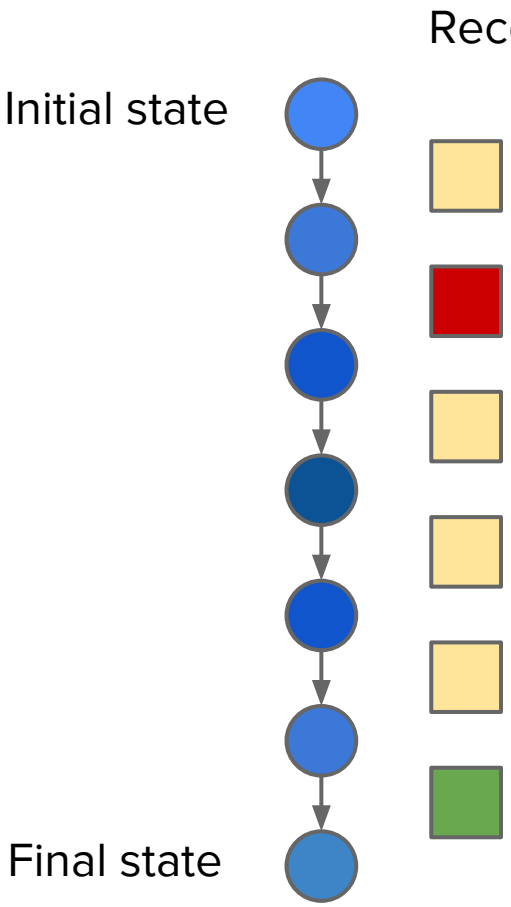


Why is this hard to parallelize?



Data dependencies

Why is this hard to parallelize?

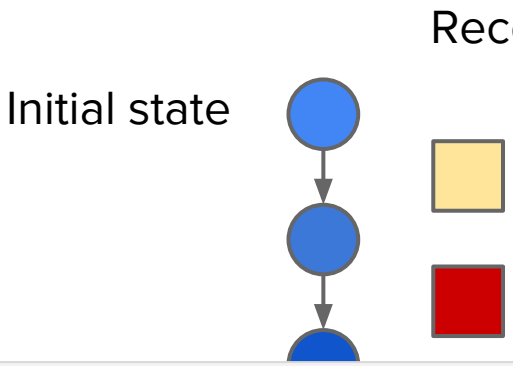


Data dependencies

Some systems include built-in aggregates that parallelize (`sum`, `max`)

Different computation based on **commutativity** and/or **associativity**

Why is this hard to parallelize?



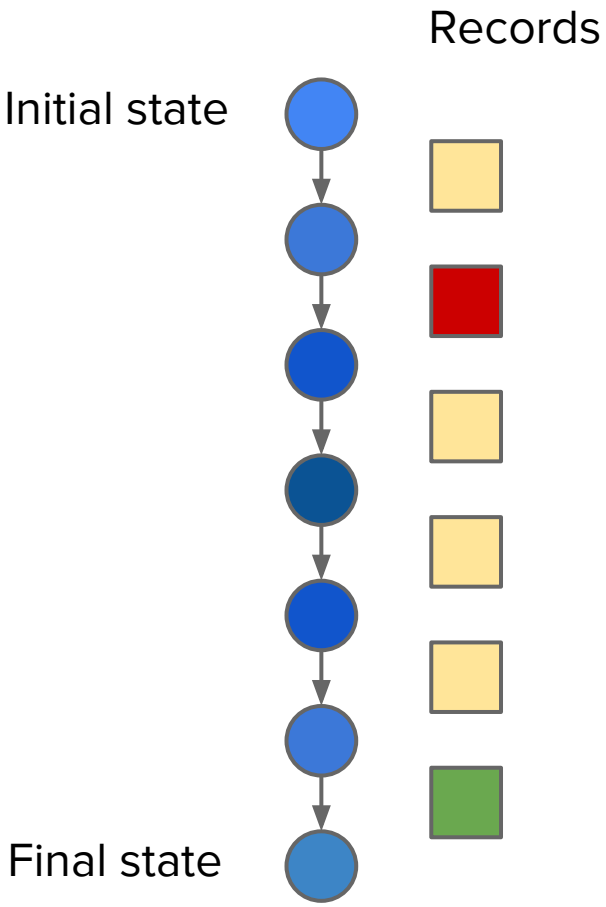
Data dependencies

But complex queries are
neither commutative, **nor associative**

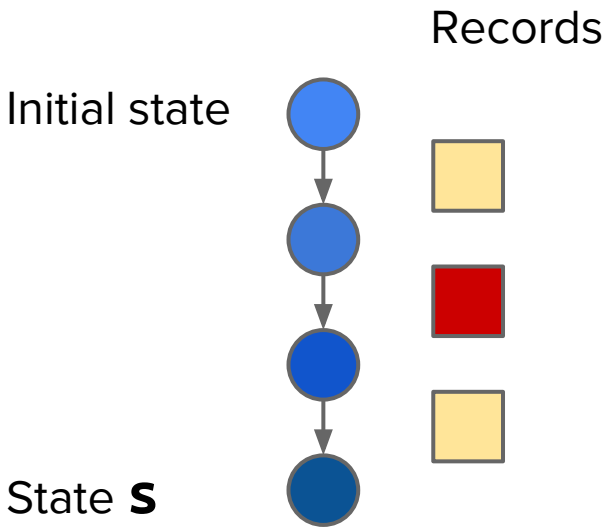
Introducing symbolic parallelism

Reorganize computation to break dependencies

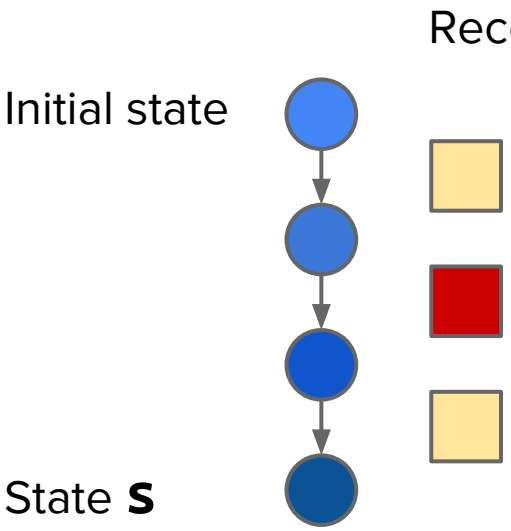
Introducing symbolic parallelism



Introducing symbolic parallelism



Introducing symbolic parallelism



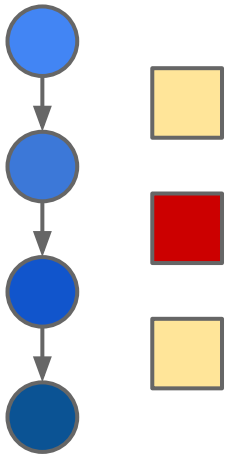
In parallel:

(Symbolic)
Unknown state **x**

Introducing symbolic parallelism

Records

Initial state

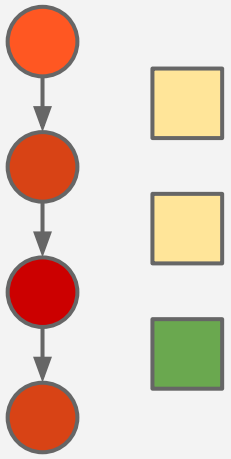


State **s**

In parallel:

Records

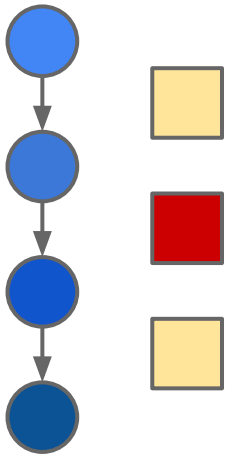
(Symbolic)
Unknown state **x**



Introducing symbolic parallelism

Records

Initial state

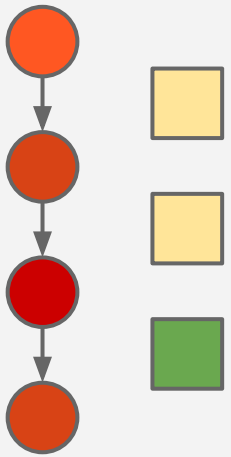


State **s**

In parallel:

Records

(Symbolic)
Unknown state **x**



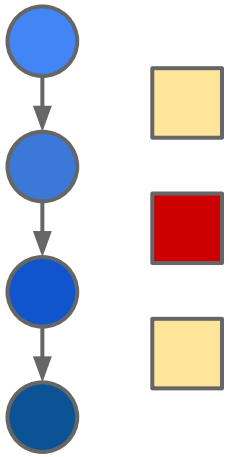
Result:

$t_{\blacksquare}(t_{\square}(t_{\square}(x)))$

Introducing symbolic parallelism

Records

Initial state

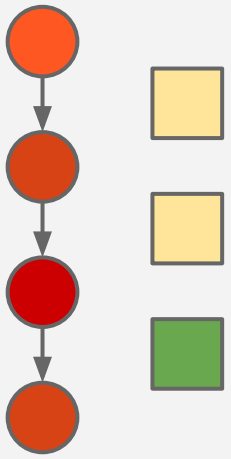


State **s**

In parallel:

Records

(Symbolic)
Unknown state x



Result:

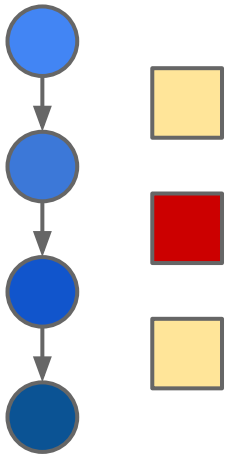
$t_{\blacksquare}(t_{\square}(t_{\square}(x)))$

Return the **function** on x

Introducing symbolic parallelism

Records

Initial state



State **s**

Combine:

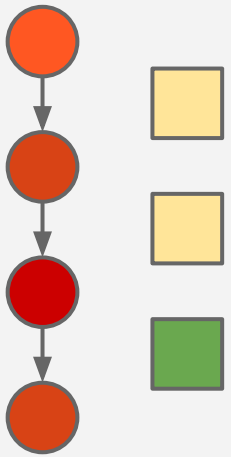
Final state:

$t_{\blacksquare}(t_{\square}(t_{\square}(s)))$

In parallel:

Records

(Symbolic)
Unknown state x



Result:

$t_{\blacksquare}(t_{\square}(t_{\square}(x)))$

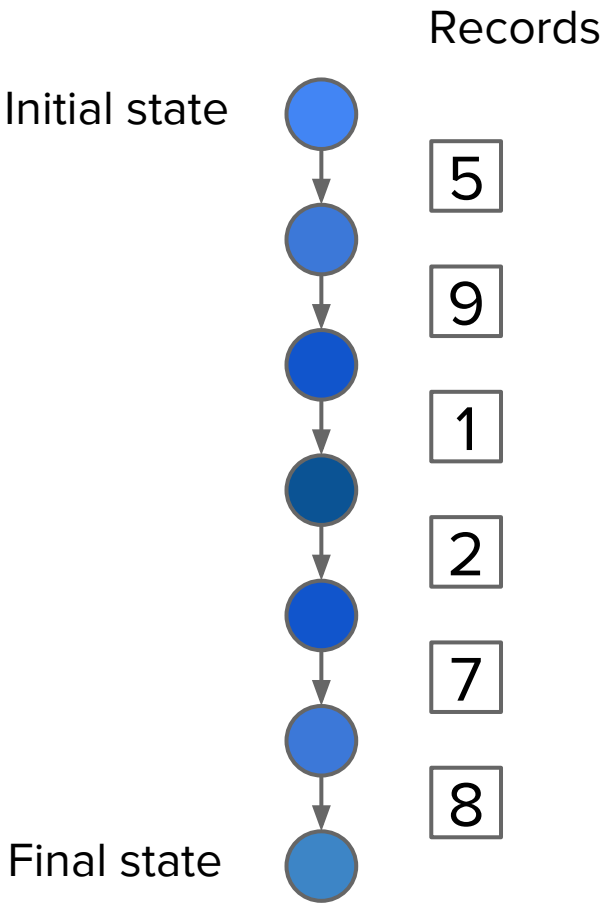
Return the **function** on x

Example aggregate: `max`

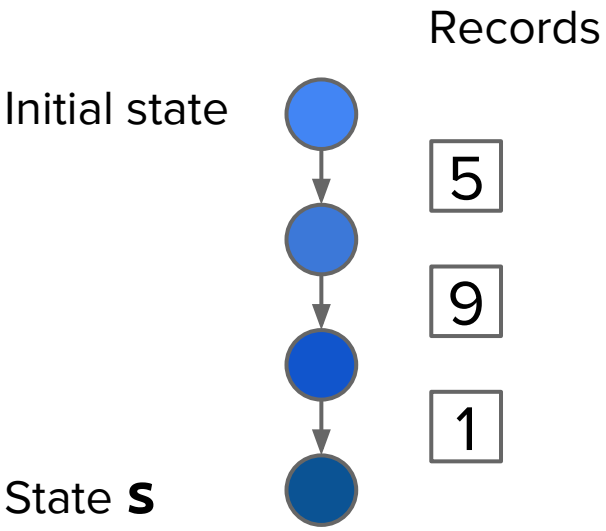
Implemented as:

```
if state < record then state = record
```

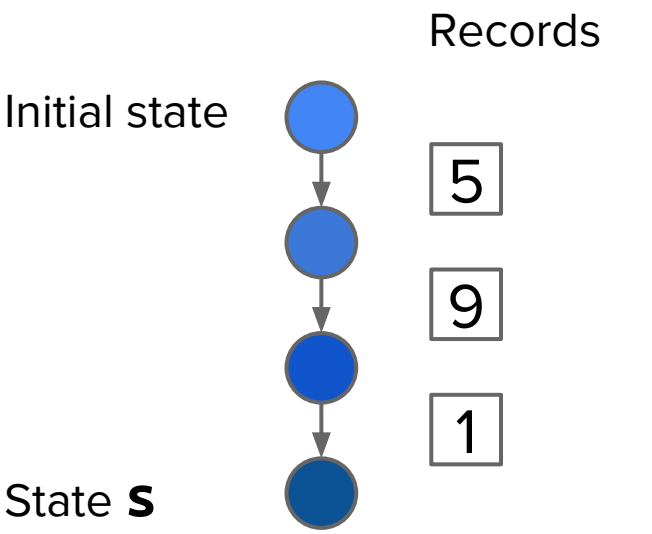
Example aggregate: `max`



Example aggregate: `max`



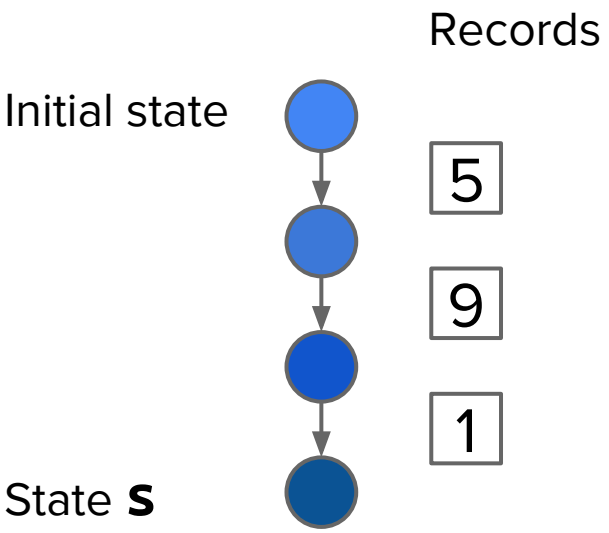
Example aggregate: `max`



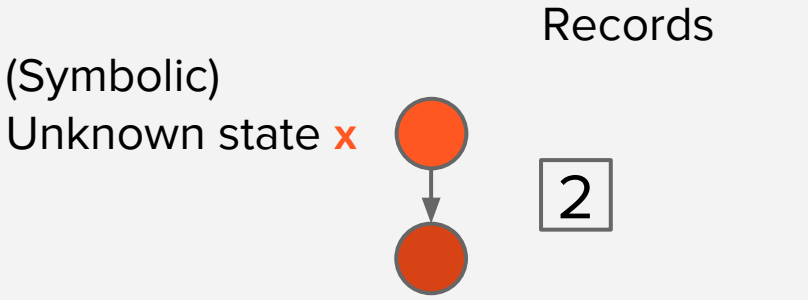
In parallel:

(Symbolic)
Unknown state **x**

Example aggregate: `max`



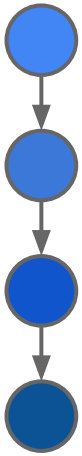
In parallel:



Example aggregate: `max`

Records

Initial state



5

9

1

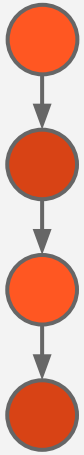
State **s**

In parallel:

Records

(Symbolic)
Unknown state x

```
if x<2
  then 2
else x
```



2

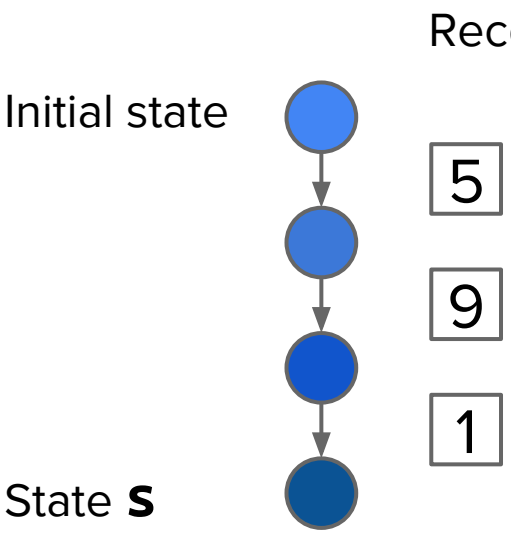
7

8

Result **f′**:

```
if x<2 then 8 else
if x<7 then 8 else
if x<8 then 8 else
else x
```

Example aggregate: `max`



Combined result

Final state:

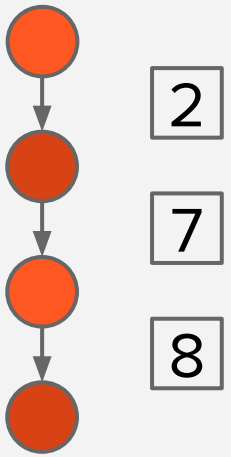
f′(s) = **f′**(9) = 9

In parallel:

Records

(Symbolic)
Unknown state x

```
if x<2
  then 2
else x
```



Result **f′**:

```
if x<2 then 8 else
if x<7 then 8 else
if x<8 then 8 else
else x
```

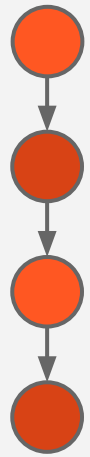
Example aggregate: `max`

In parallel:

Records

(Symbolic)
Unknown state x

```
if x<2
   then 2
else x
```



2

7

8

Result **f′**:

$x<8 \Rightarrow 8$

$x \geq 8 \Rightarrow x$

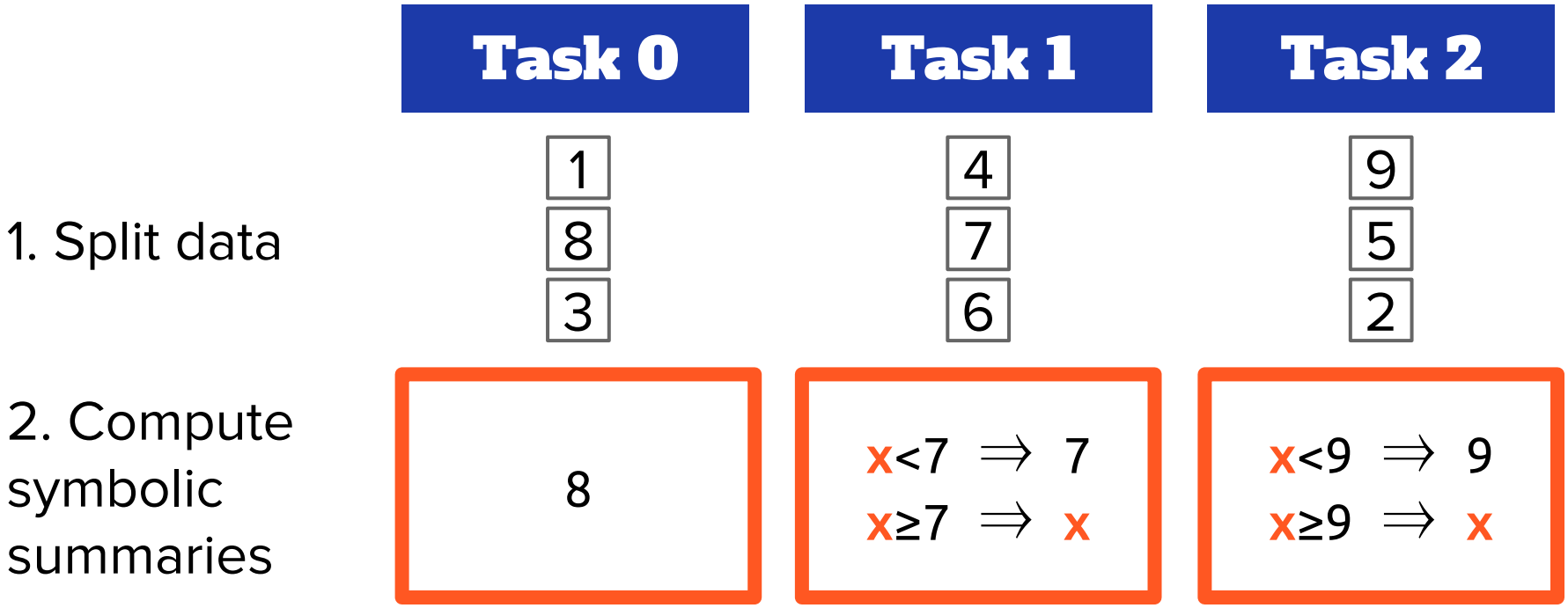
Wanted: a compact and efficient representation of a function

Full symbolic pipeline

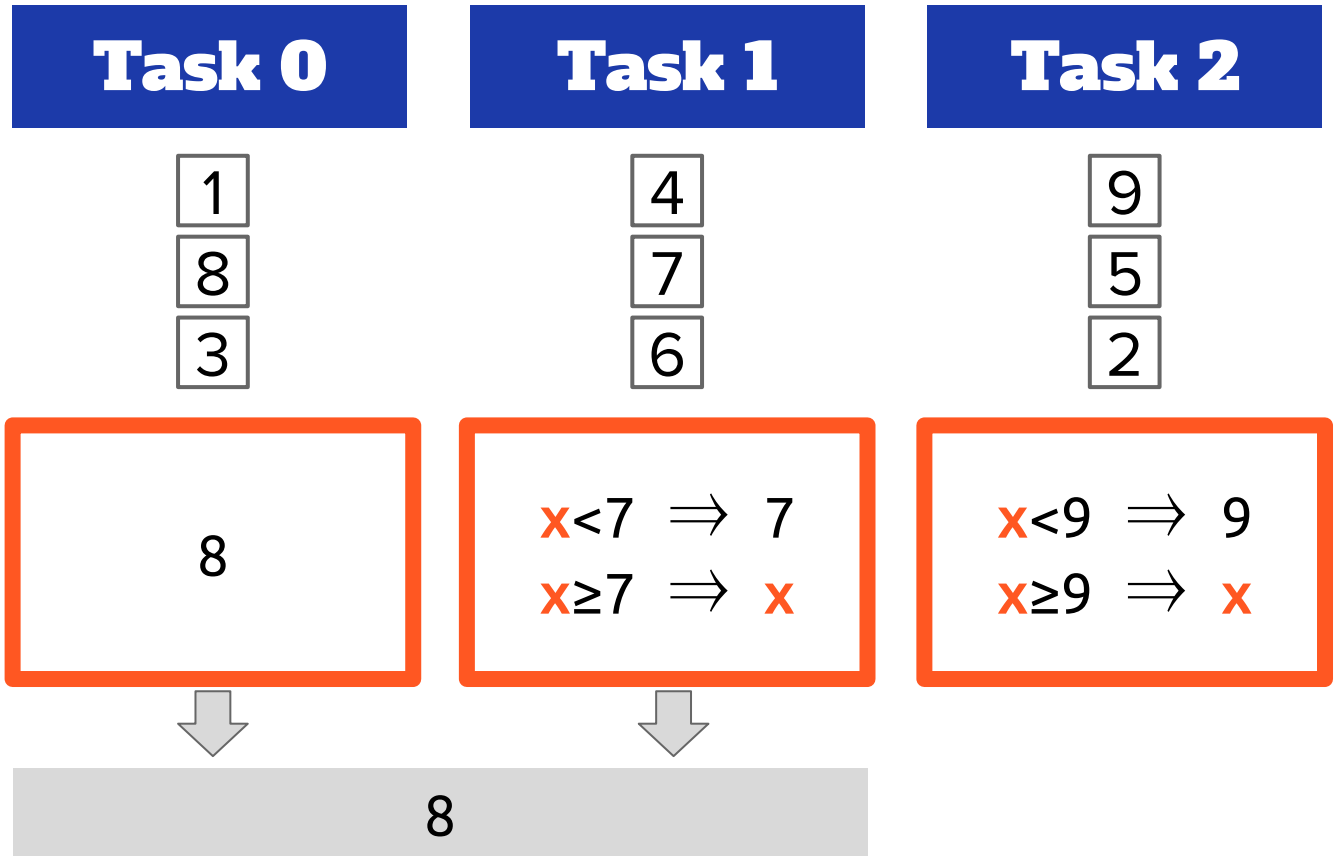
Full symbolic pipeline

	Task 0	Task 1	Task 2
1. Split data	1 8 3	4 7 6	9 5 2

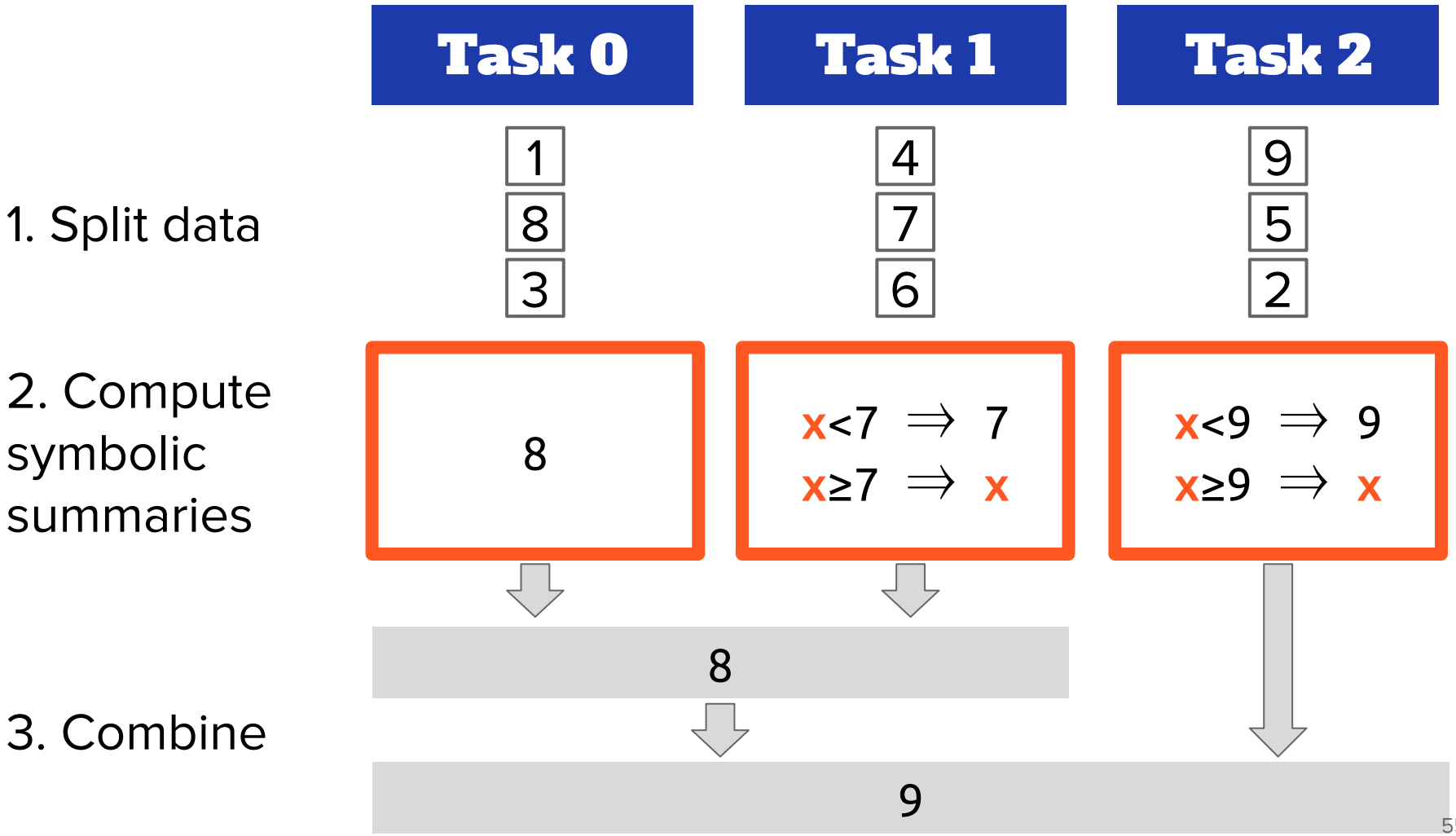
Full symbolic pipeline



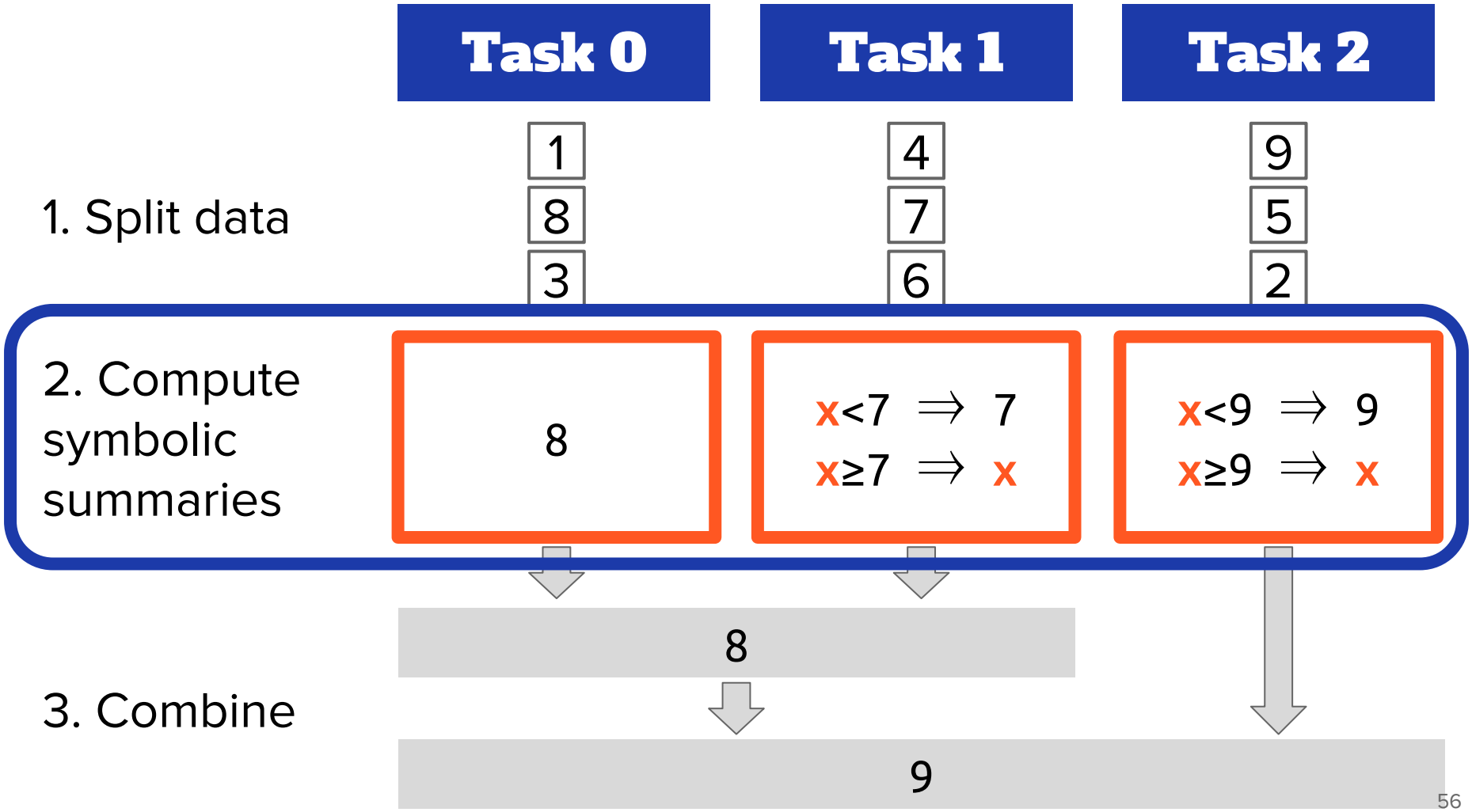
Full symbolic pipeline



Full symbolic pipeline



Full symbolic pipeline



Key insight: Symbolic data types

Representation of functions inspired by

symbolic execution and abstract interpretation

Key insight: Symbolic data types

Representation of functions inspired by

symbolic execution and abstract interpretation

$$\texttt{PathConstraint}_1 \Rightarrow \texttt{f}_1(\texttt{x})$$

$$\texttt{PathConstraint}_2 \Rightarrow \texttt{f}_2(\texttt{x})$$

$$\texttt{PathConstraint}_3 \Rightarrow \texttt{f}_3(\texttt{x})$$

Key insight: Symbolic data types

Representation of functions inspired by

symbolic execution and abstract interpretation

Explore possible program paths and accumulate path constraints

$$\texttt{PathConstraint}_1 \Rightarrow \texttt{f}_1(\texttt{x})$$

$$\texttt{PathConstraint}_2 \Rightarrow \texttt{f}_2(\texttt{x})$$

$$\texttt{PathConstraint}_3 \Rightarrow \texttt{f}_3(\texttt{x})$$

Key insight: Symbolic data types

Representation of functions inspired by

symbolic execution and abstract interpretation

Explore possible program paths and accumulate path constraints

$$\texttt{PathConstraint}_1 \Rightarrow \texttt{f}_1(\texttt{x})$$

$$\texttt{PathConstraint}_2 \Rightarrow \texttt{f}_2(\texttt{x})$$

$$\texttt{PathConstraint}_3 \Rightarrow \texttt{f}_3(\texttt{x})$$

Specific shapes of formulas similar to abstract domains: data + decision procedures

Key insight: Symbolic data types

Representation of functions inspired by

symbolic execution and abstract interpretation

Explore possible program paths and accumulate path constraints

Specific shapes of formulas similar to abstract domains: data + decision procedures

Techniques typically used for correctness

Now we use it for performance

Outline

1. Symbolic parallelism

2. Data types

3. Evaluation

Symbolic integer type: `SymInt`

Shape: $x \in [l, u] \Rightarrow ax + b$

`l`,`u`,`a`,`b` - integer constants

Symbolic integer type: SymInt

Shape: $x \in [l, u] \Rightarrow ax + b$

l,u,a,b - integer constants

State s is a concrete value b:

$$x \in [-\infty, \infty] \Rightarrow 0x + b$$

Symbolic integer type: SymInt

Shape: $x \in [l, u] \Rightarrow ax + b$

l,u,a,b - integer constants

State s is a concrete value b:

$$x \in [-\infty, \infty] \Rightarrow 0x + b$$

State s stores a symbolic value x:

$$x \in [-\infty, \infty] \Rightarrow 1x + 0$$

Symbolic integer type: `SymInt`

Shape: $x \in [l, u] \Rightarrow ax + b$

Initial state **s**	Instruction	Resulting state **s**

Symbolic integer type: SymInt

Shape: $x \in [l, u] \Rightarrow ax + b$

Initial state **s**	Instruction	Resulting state **s**
$[l, u] \Rightarrow ax + b$	`s = 7;` →	$[l, u] \Rightarrow 0x + 7$

Symbolic integer type: SymInt

Shape: $x \in [l, u] \Rightarrow ax + b$

Initial state **s**	Instruction	Resulting state **s**
$[l, u] \Rightarrow ax + b$	`s = 7;` →	$[l, u] \Rightarrow 0x + 7$
$[l, u] \Rightarrow ax + b$	`s + 3` →	$[l, u] \Rightarrow ax + (b+3)$

Symbolic integer type: SymInt

Shape: $x \in [l, u] \Rightarrow ax + b$

Initial state **s**	Instruction	Resulting state **s**
$[l, u] \Rightarrow ax + b$	`s = 7;`	$[l, u] \Rightarrow 0x + 7$
$[l, u] \Rightarrow ax + b$	`s + 3`	$[l, u] \Rightarrow ax + (b+3)$
$[l, u] \Rightarrow ax + b$	`if (s ≤ 5)`	$[l, 5] \Rightarrow ax + b$

*for u > 5

Symbolic integer type: `SymInt`

Shape: $x \in [l, u] \Rightarrow ax + b$

Initial state **s**	Instruction	Resulting state **s**

Symbolic integer type: `SymInt`

Shape: $x \in [l, u] \Rightarrow ax + b$

Initial state **s**	Instruction	Resulting state **s**
$[l, u] \Rightarrow ax + b$	`s1 + s2` →	
$[l, u] \Rightarrow ax + b$	`if (s1 ≤ s2)` →	

Symbolic integer type: `SymInt`

Shape: $x \in [l, u] \Rightarrow ax + b$

Initial state **s**	Instruction	Resulting state **s**
$[l, u] \Rightarrow ax + b$	`s1 + s2` →	**Disallowed:** static type checking error
$[l, u] \Rightarrow ax + b$	`if (s1 ≤ s2)` →	Path constraints are an **interval, not relational**

Other data types - same idea

SymBool

SymEnum

SymVector

SymPredicate

Composing symbolic variables

Multiple symbolic variables

$PathConstraint_1 \Rightarrow f_1(x_1)$ $PathConstraint_2 \Rightarrow f_2(x_2)$

means

$PathConstraint_1 \wedge PathConstraint_2 \Rightarrow (f_1(x_1), f_2(x_2))$

Overall

Overall

Predefined data types

SymInt
SymBool
SymEnum
SymVector
SymPredicate

Overall

Predefined data types

SymInt
SymBool
SymEnum
SymVector
SymPredicate

Types can
be combined

Overall

Predefined data types

SymInt
SymBool
SymEnum
SymVector
SymPredicate

Extensible with new types

Types can
be combined

Overall

Predefined data types

SymInt
SymBool
SymEnum
SymVector
SymPredicate

Extensible with new types

Types can be combined

Enable writing custom aggregates

SymInt num_reviews = 0;

SymBool search_done = false;

SymVector<**SymInt**> result;

```
foreach record in user records:
    switch record.type:
        case SEARCH:        num_reviews = 0;    search_done = true;
        case REVIEW:        num_reviews++;
        case PURCHASE:      if search_done:
                                search_done = false;
                                result.push_back(num_reviews);
```

Outline

1. Symbolic parallelism

2. Data types

3. Evaluation

Benchmarks

query logs from GitHub, RedShift, Bing, Twitter

example queries:

ID	Data	Description
1	GitHub	All operations on a repository directly preceding a delete operation
2	GitHub	Number of operations executed on a repository between pull open and close
3	RedShift	List of advertisers operating only in a single country
4	RedShift	Ads for an advertiser not showing for more than 1 hour

Implementation

User code

Symbolic data types Type checking (with C++) Execution engine

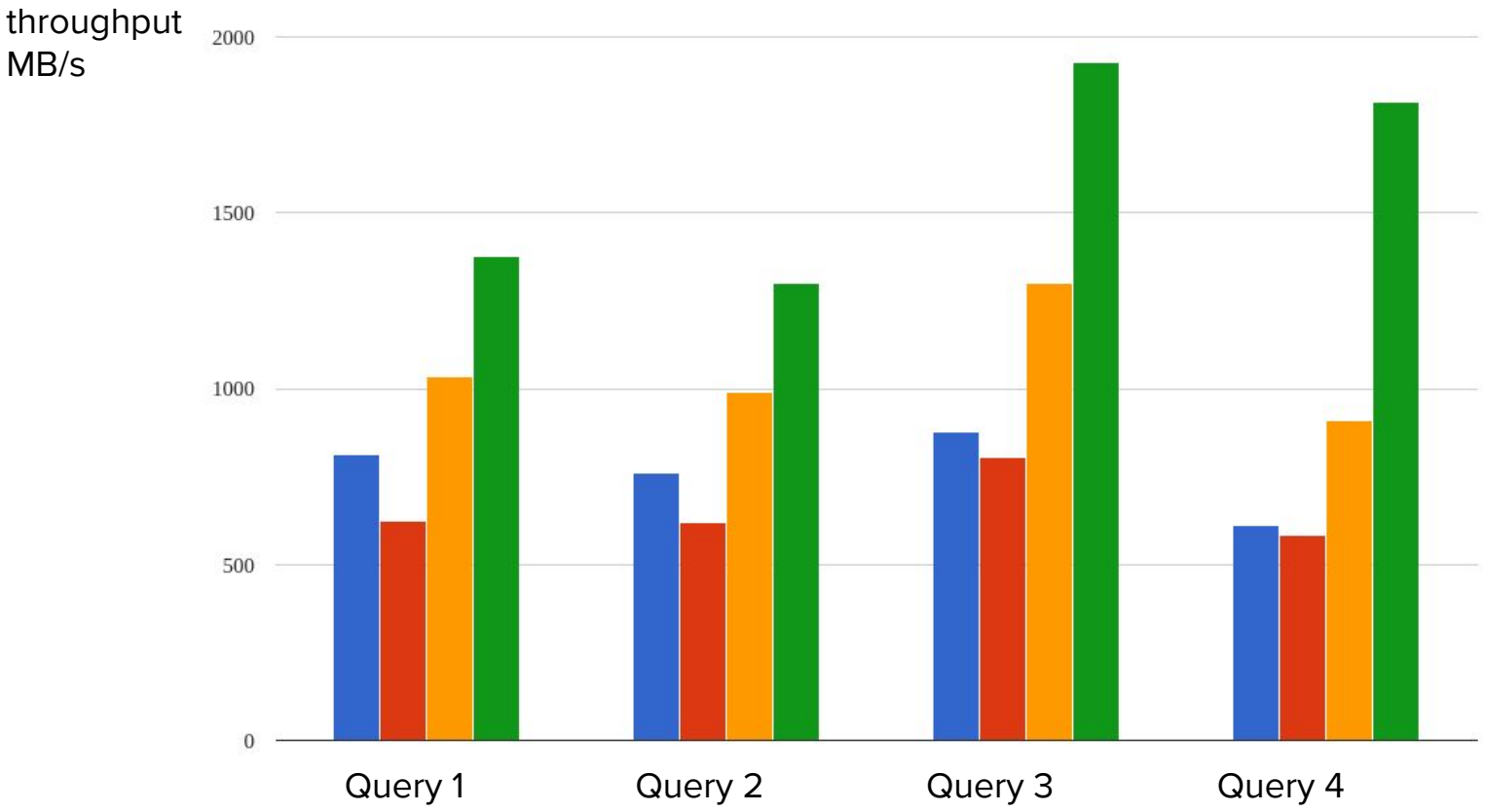
SYMPLE library

Single-machine Multi-core parallelism

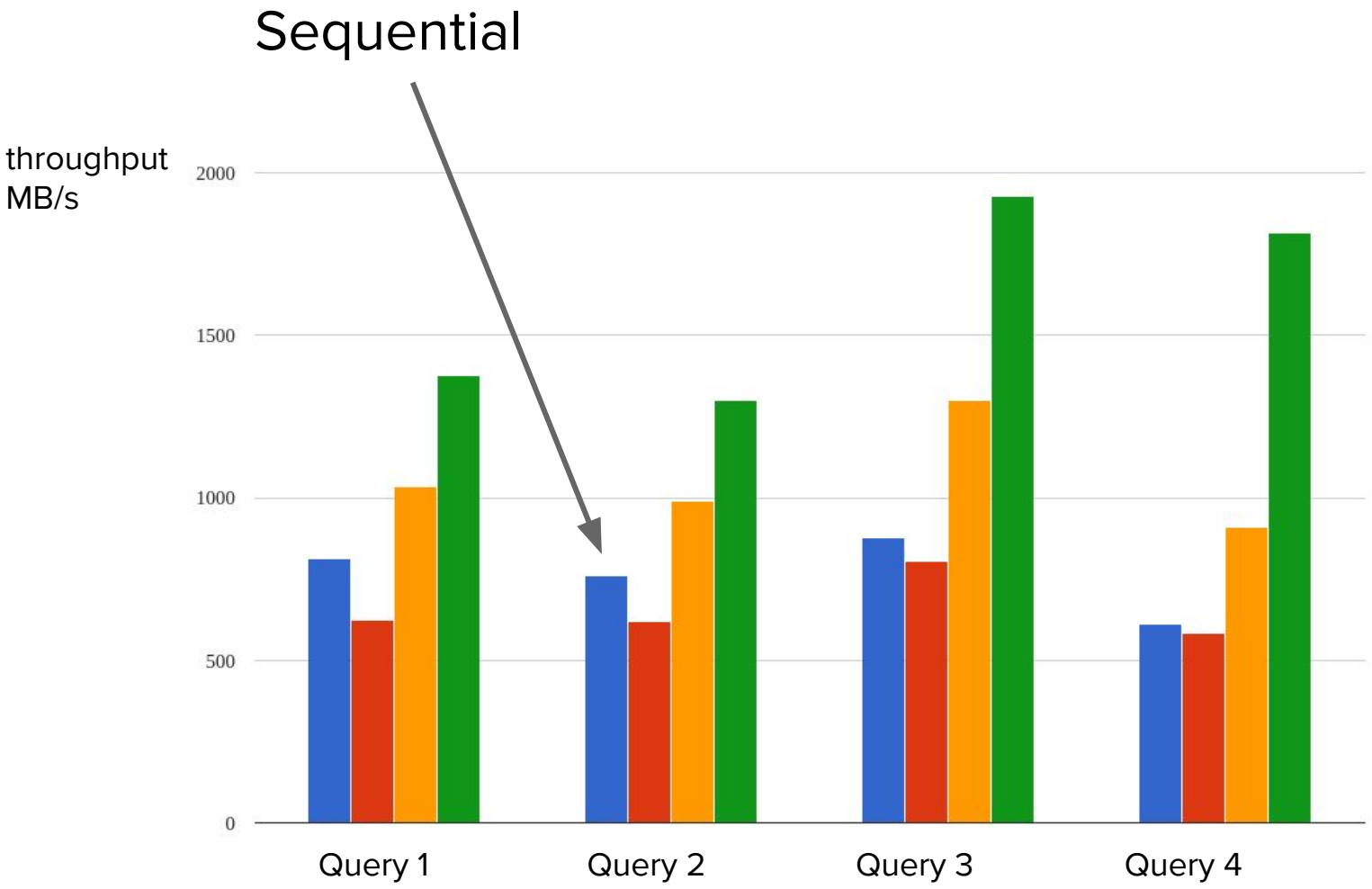
hadoop

...

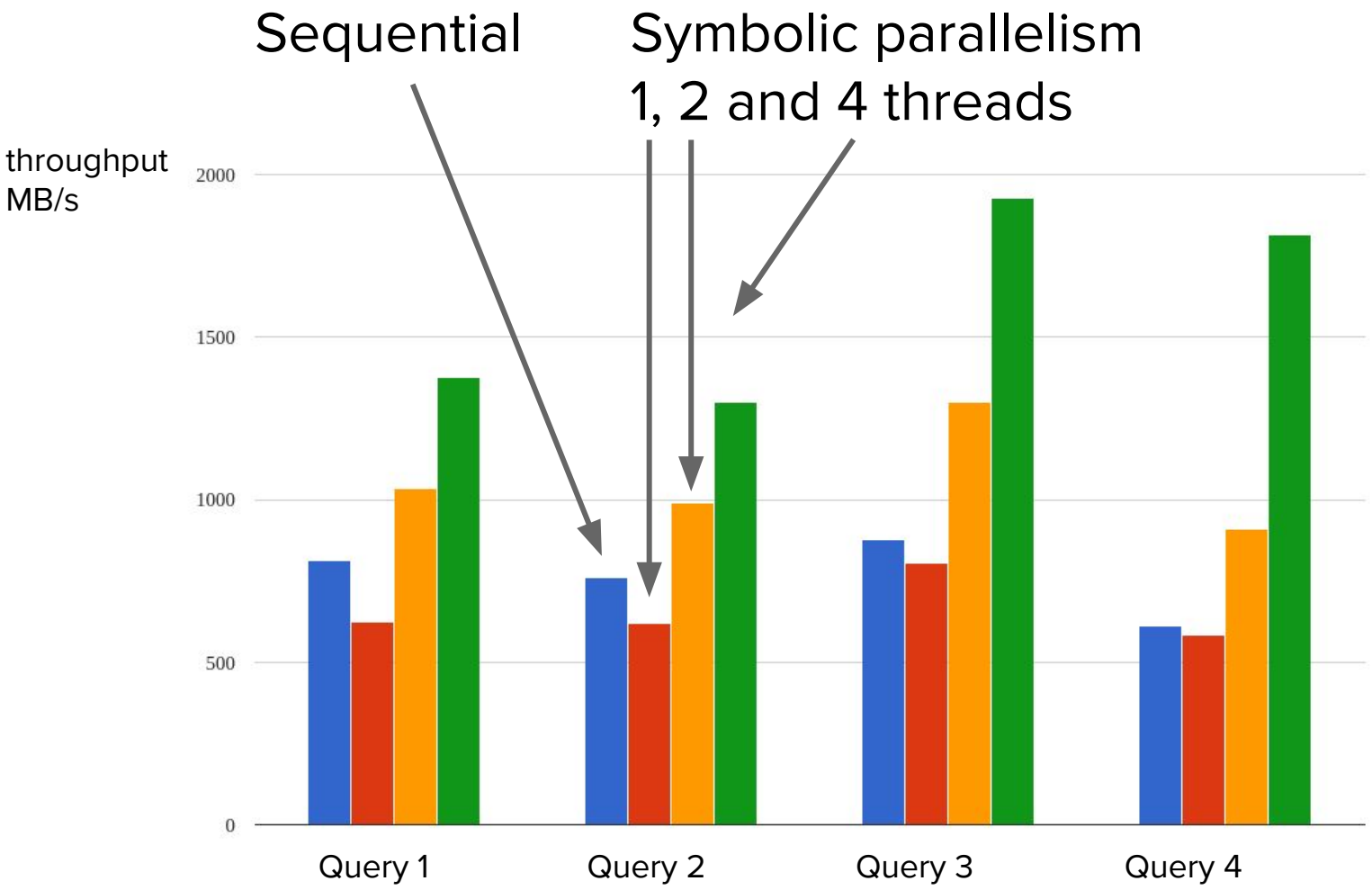
Single machine throughput



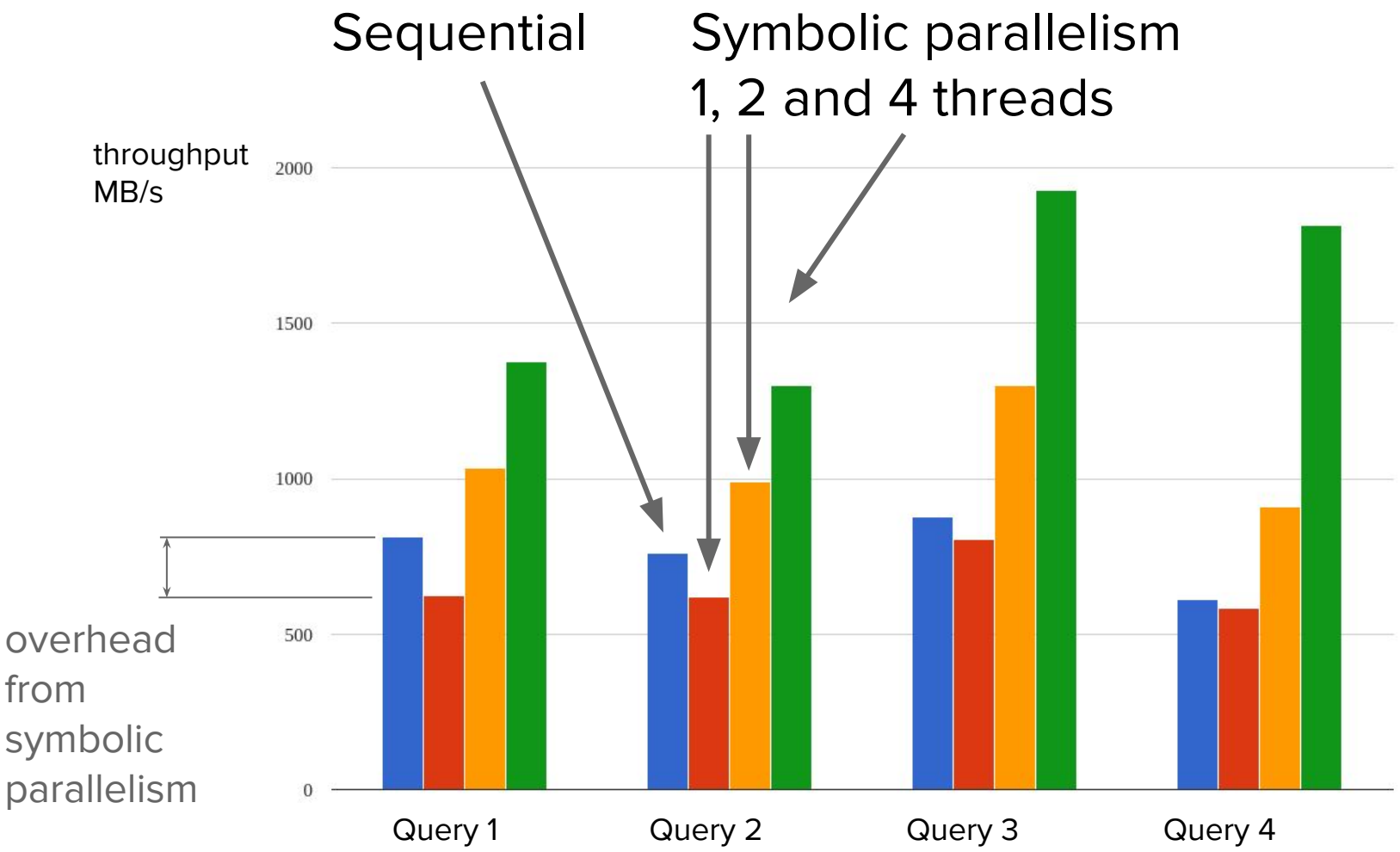
Single machine throughput



Single machine throughput



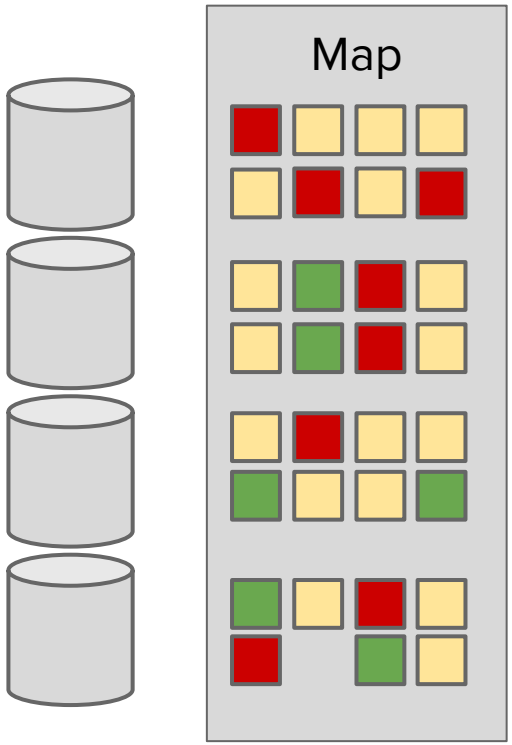
Single machine throughput



Handling large data (before)

Naive encoding in MapReduce

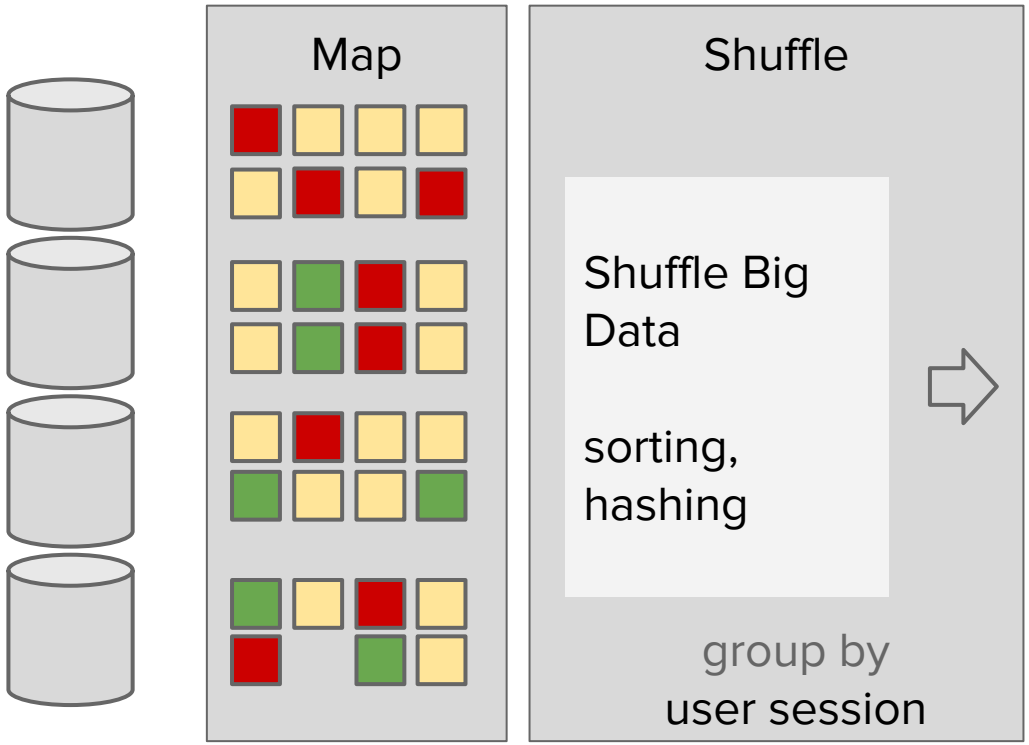
Distributed:
query logs



Handling large data (before)

Naive encoding in MapReduce

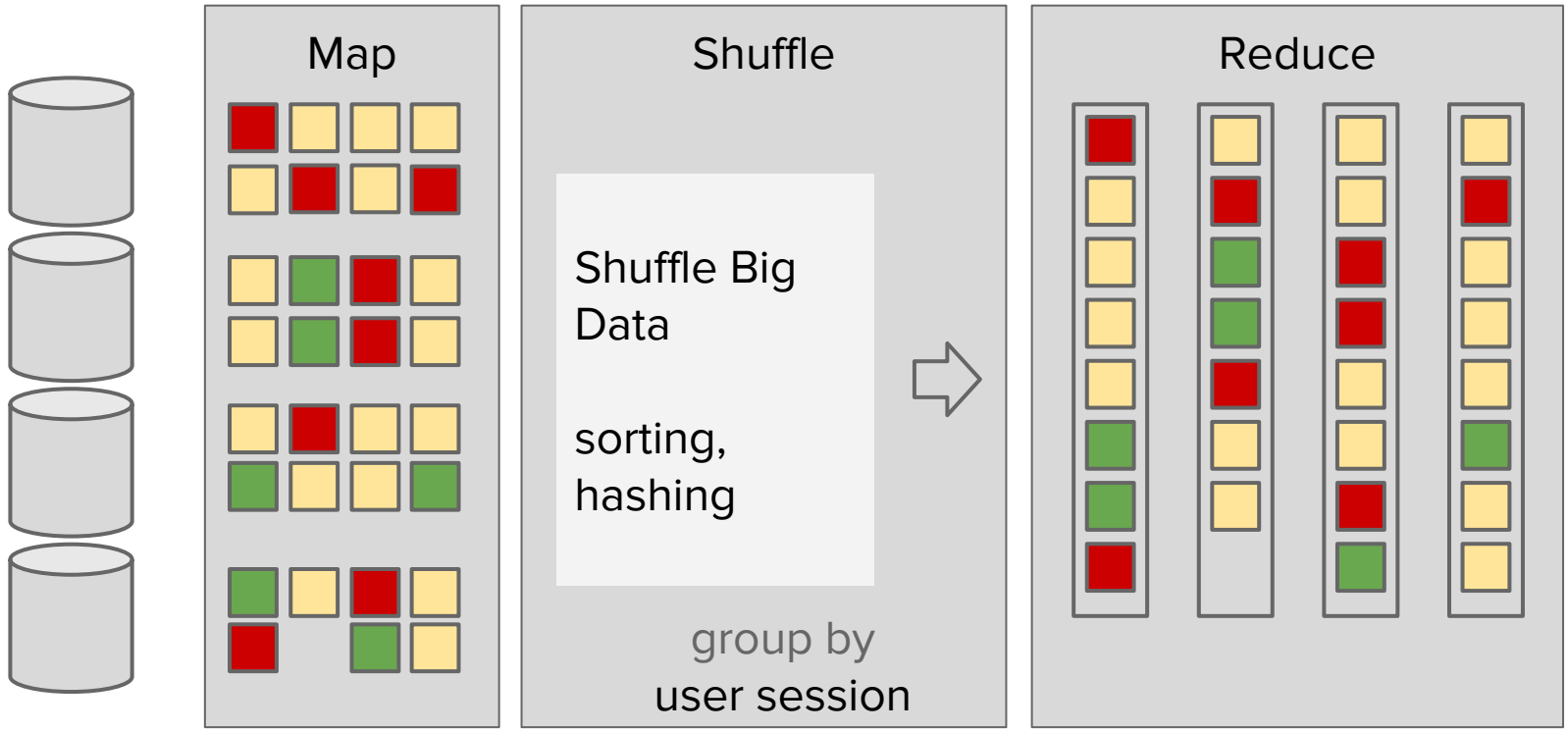
Distributed:
query logs



Handling large data (before)

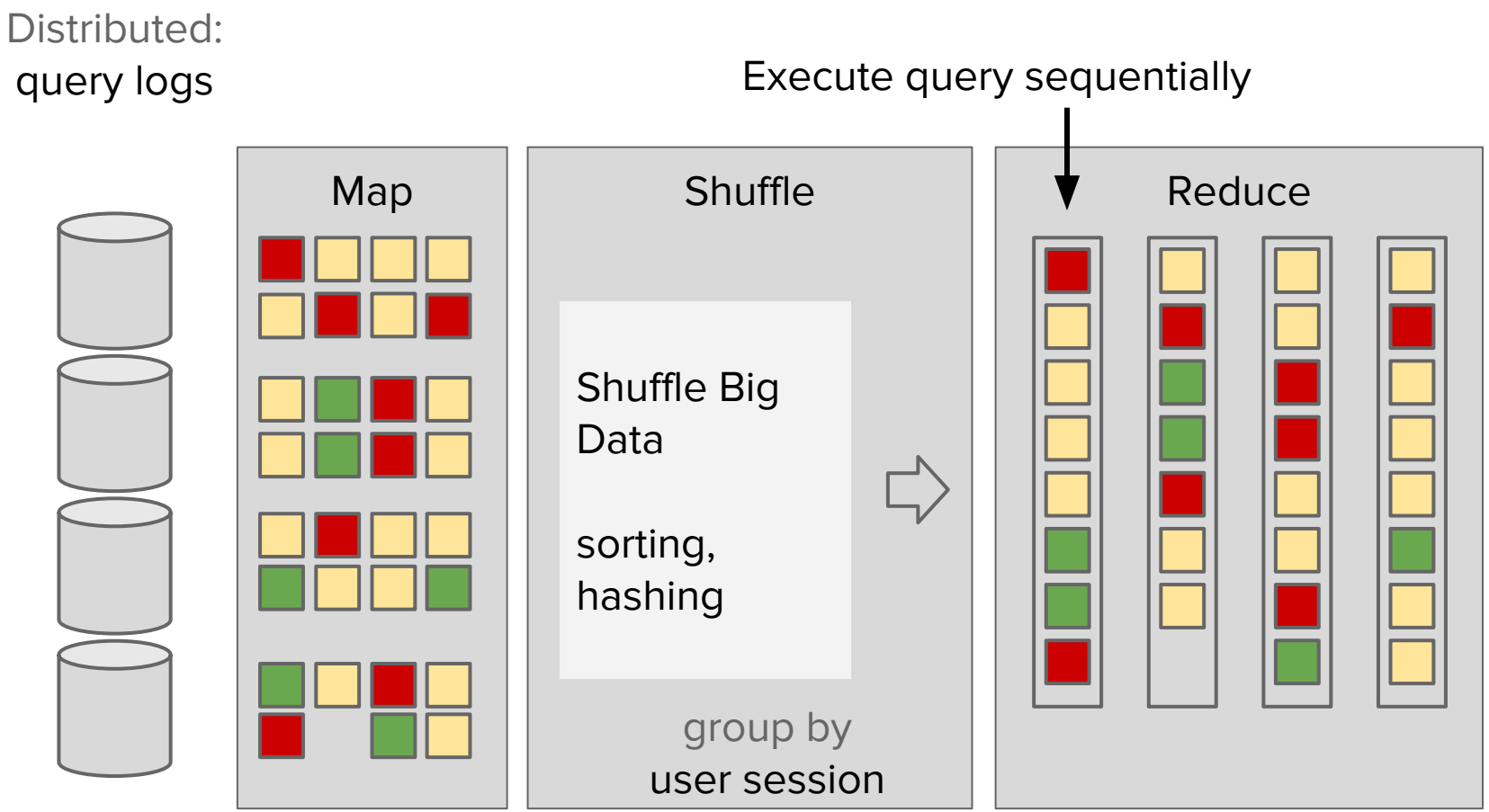
Naive encoding in MapReduce

Distributed:
query logs



Handling large data (before)

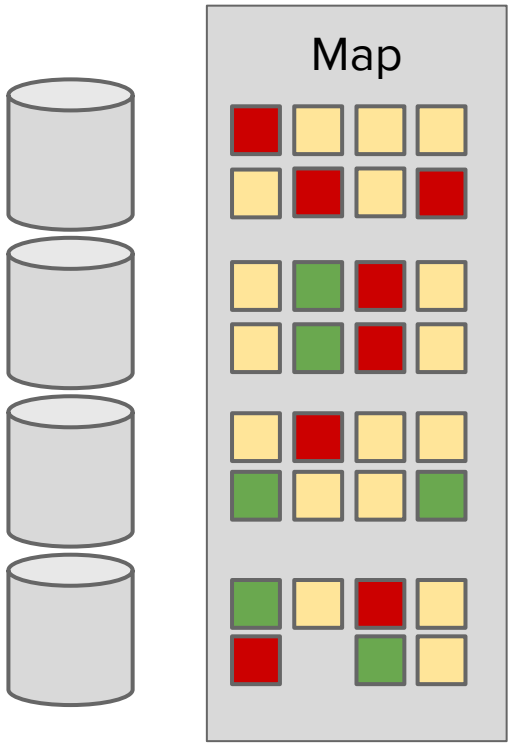
Naive encoding in MapReduce



Handling large data (now)

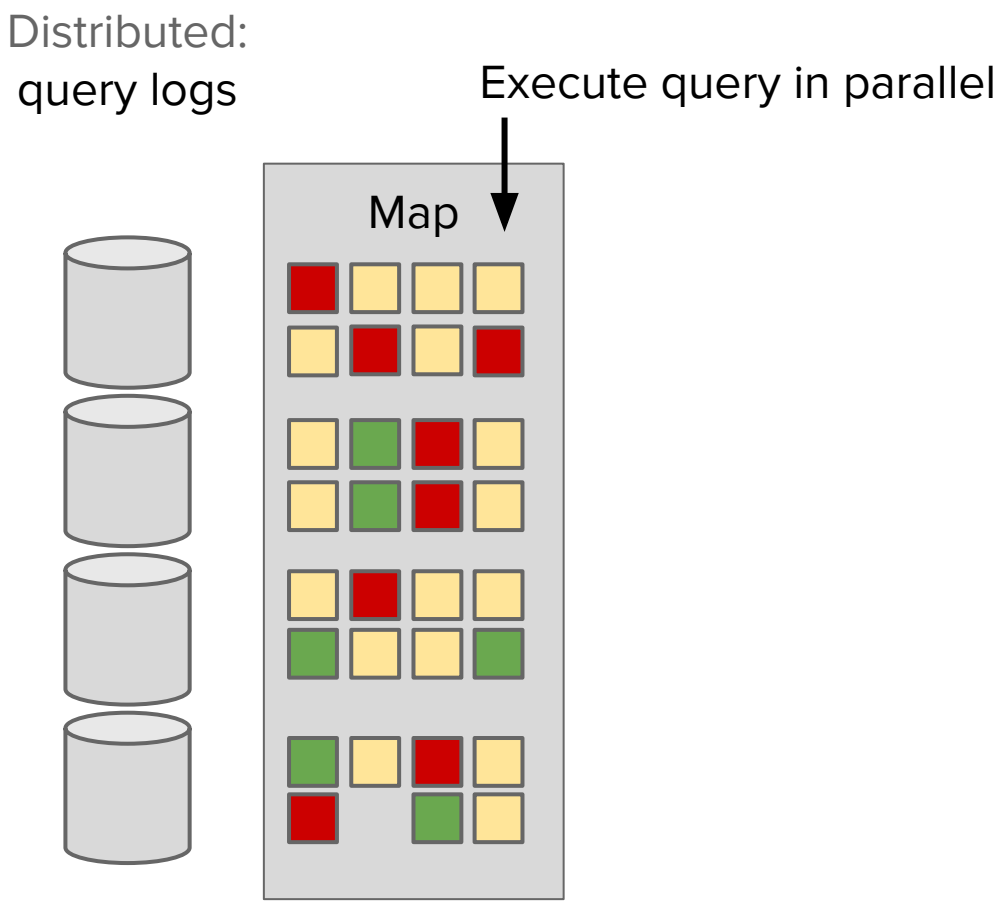
Our tool: **SYMPLE**

Distributed:
query logs



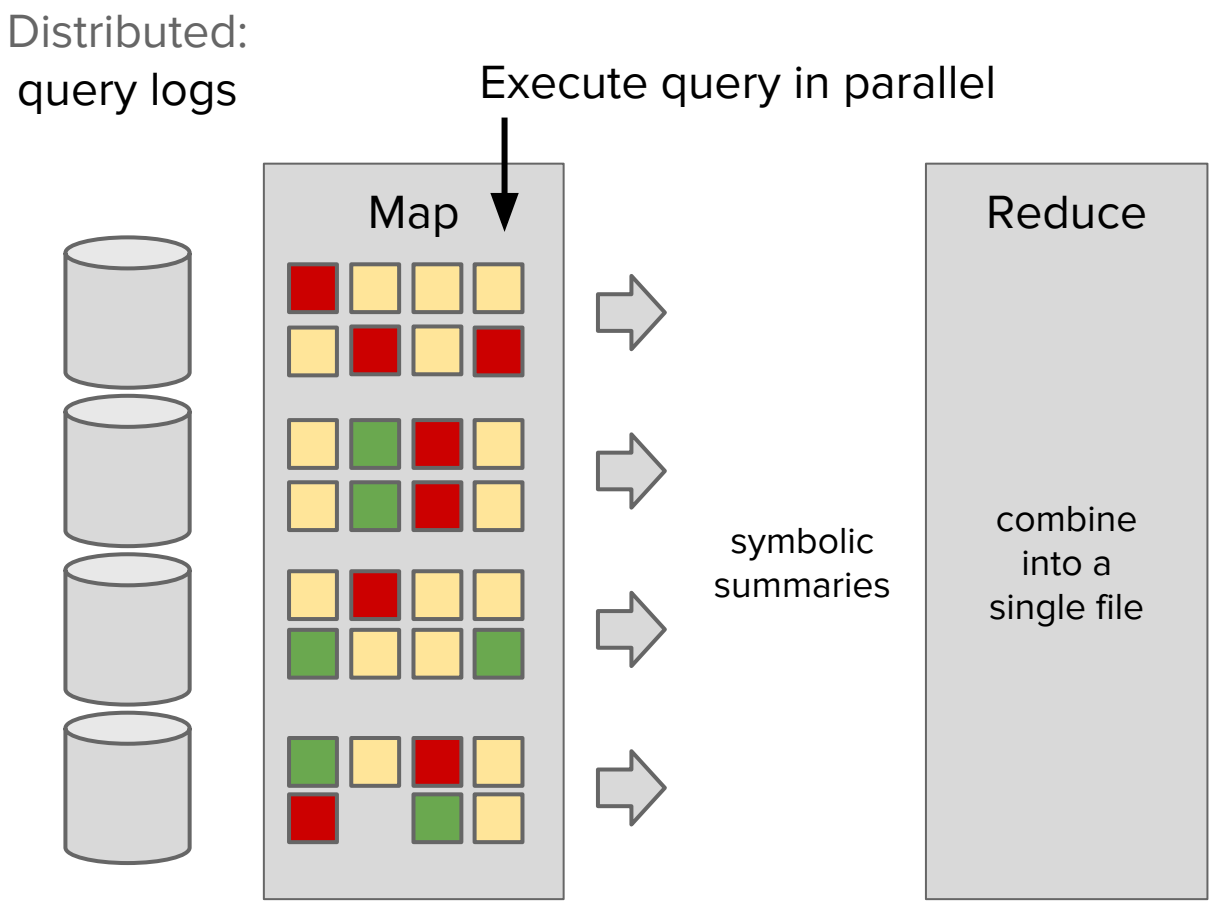
Handling large data (now)

Our tool: **SYMPLE**



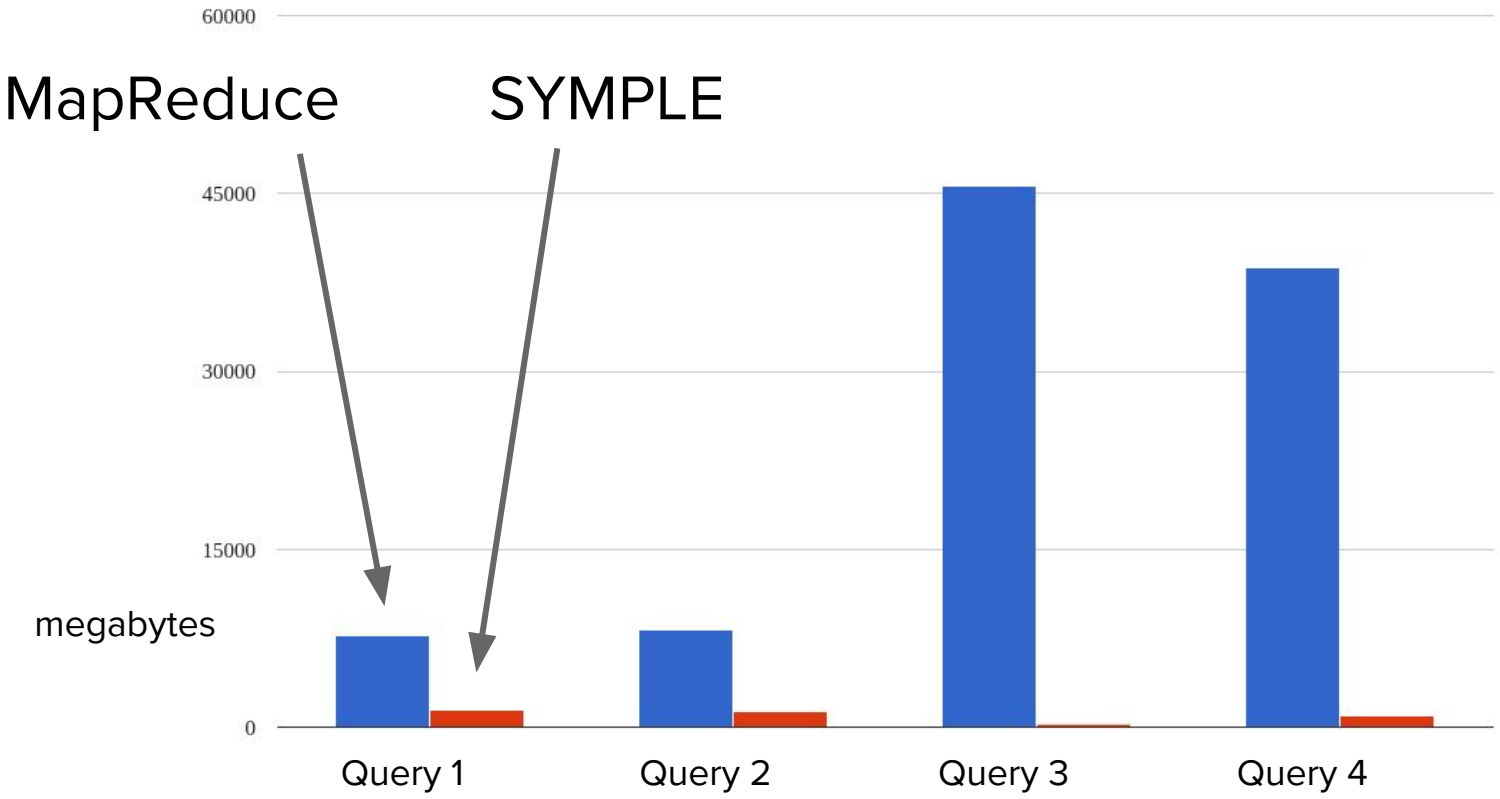
Handling large data (now)

Our tool: **SYMPLE**



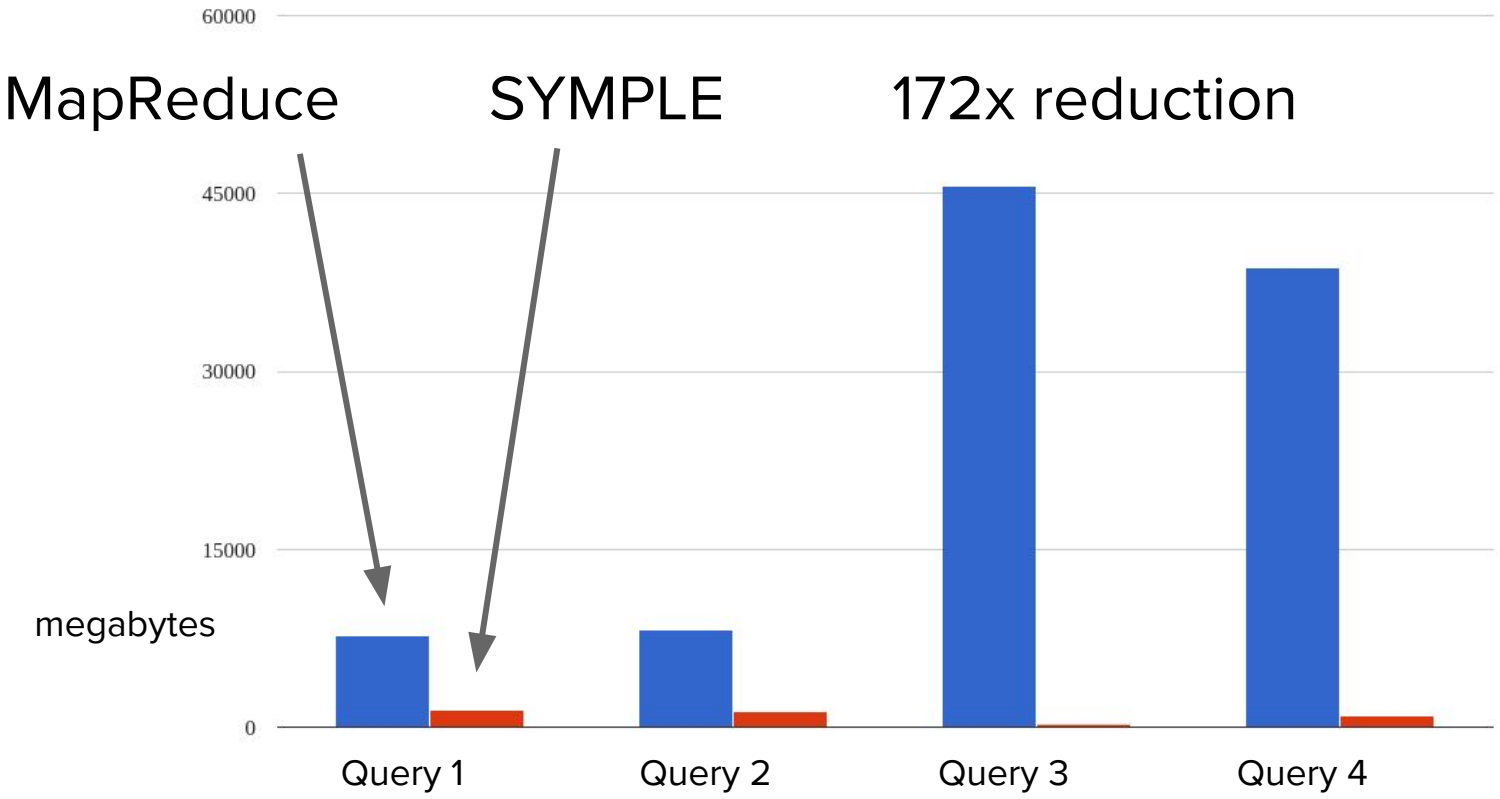
Data movement

Data shuffled from mappers to reducers



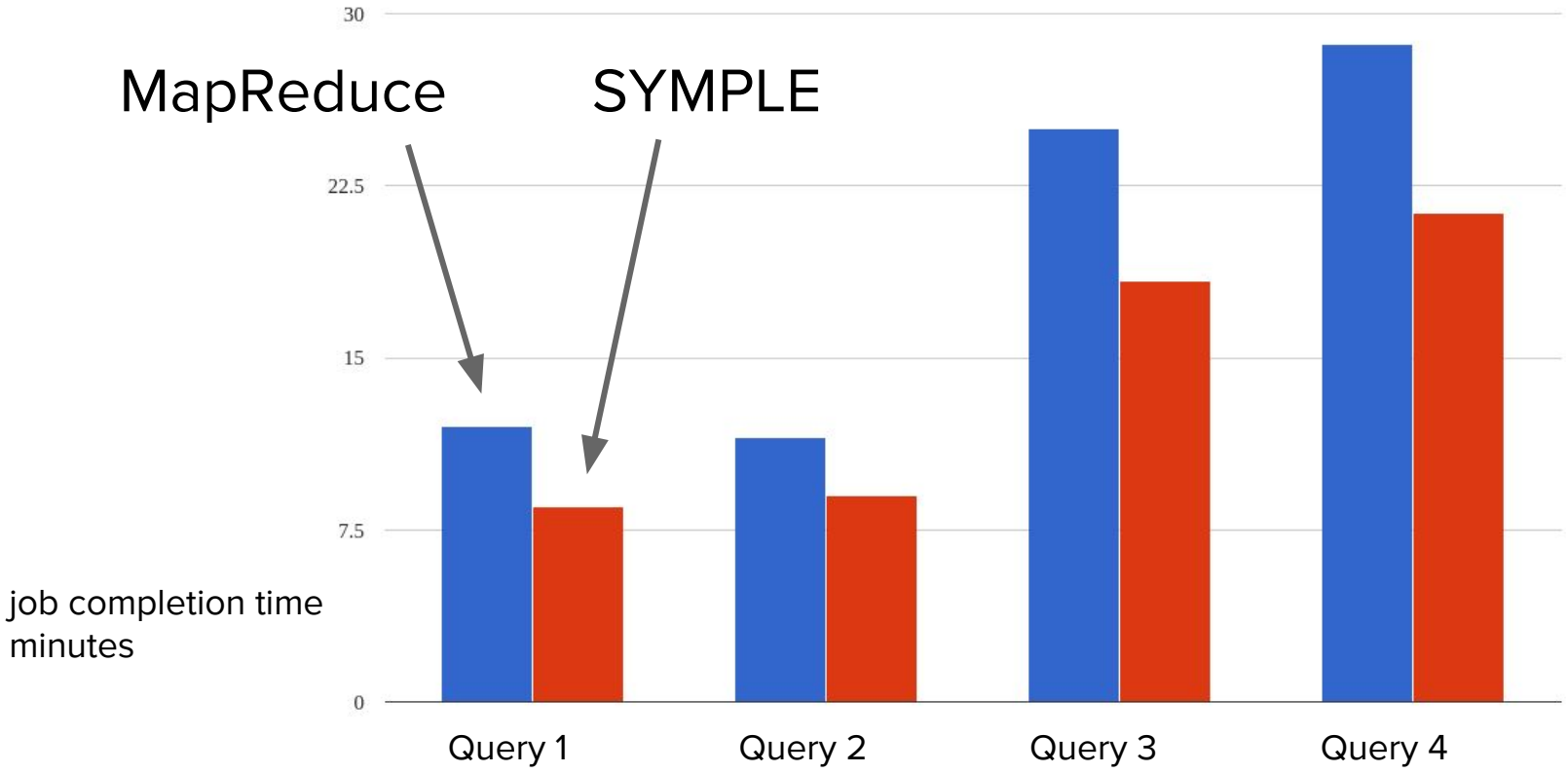
Data movement

Data shuffled from mappers to reducers



End-to-end latency

On Amazon Web Services, Hadoop and S3



End-to-end latency

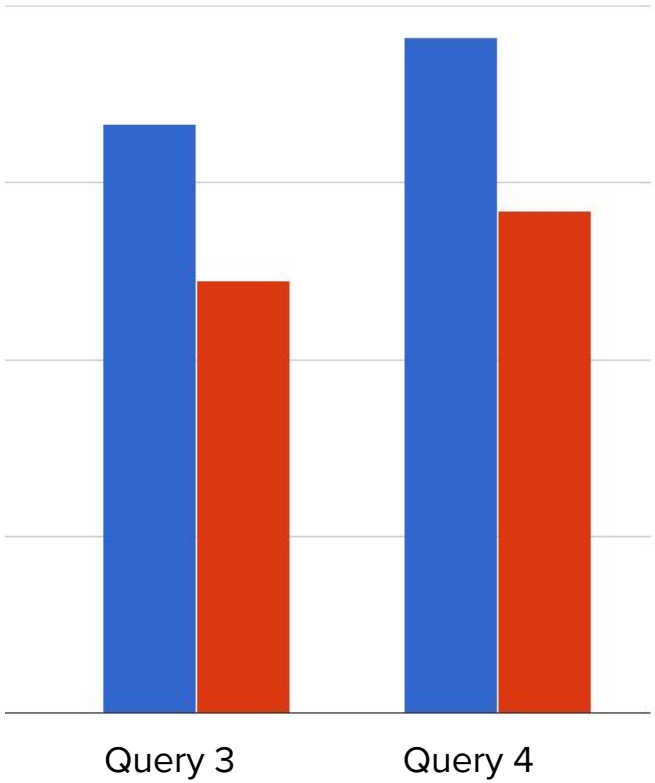
On Amazon Web Services, Hadoop and S3

Fixing shuffle inefficiency exposes us to next bottleneck:

reading input and throwing most of it away

easy solution:

compress input or use data with smaller records



End-to-end latency

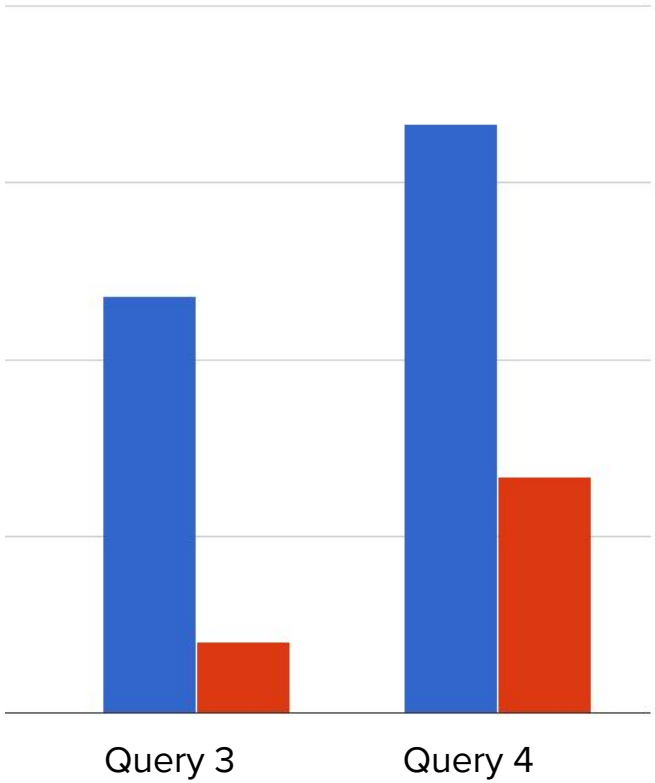
On Amazon Web Services, Hadoop and S3

Fixing shuffle inefficiency exposes us to next bottleneck:

reading input and throwing most of it away

easy solution:

compress input or use data with smaller records



Summary

Symbolic parallelism

Data types

Type checking

Efficient implementation

```
SymInt num_reviews = 0;
SymBool search_done = false;
SymVector<SymInt> result;

foreach record in user records:
    switch record.type:
        case SEARCH:        num_reviews = 0;    search_done = true;
        case REVIEW:        num_reviews++;
        case PURCHASE:      if search_done:
                                search_done = false;
                                result.push_back(num_reviews);
```