Parallelizing User-Defined Aggregations using Symbolic Execution

Veselin Raychev
ETH Zurich
veselin.raychev@inf.ethz.ch

Madanlal Musuvathi
Microsoft Research
madanm@microsoft.com

Todd Mytkowicz
Microsoft Research
toddm@microsoft.com

Abstract

User-defined aggregations (UDAs) are integral to large-scale data-processing systems, such as MapReduce and Hadoop, because they let programmers express application-specific aggregation logic. 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.

This paper presents SYMPLE, a system for performing MapReduce-style groupby-aggregate queries that automatically parallelizes UDAs. Users specify UDAs using stylized C++ code *with* possible loop-carried dependences. SYMPLE parallelizes these UDAs by breaking dependences using symbolic execution, where unresolved dependences are treated as symbolic values and the SYMPLE runtime partially evaluates the resulting symbolic expressions on concrete input. Programmers write UDAs using SYMPLE's symbolic data types, which look and behave like standard C++ types. These data types (i) encode specialized decision procedures for efficient symbolic execution and (ii) generate compact symbolic expressions for efficient network transfers. Evaluation on both Amazon's Elastic cloud and a private 380-node Hadoop cluster housing terabytes of data demonstrates that SYMPLE reduces network communication up to several orders of magnitude and job latency by as much as 5.9x for a representative set of queries.

1. Introduction

The explosive growth of data and the need to efficiently mine them has led to several large-scale data-processing systems [1, 2, 8, 9, 12, 14, 16, 22, 38, 39]. These systems are highly optimized to exploit the massive parallelism available in data-centers while minimizing the disk and network I/O required to perform a given computation.

The programming interface to these systems support data-parallel operators, such as map, filter, and groupby, which lets programmers easily expose the parallelism inherent in their computation. In addition, these systems allow associative aggregation functions, such as `Sum` or `Max`, that are easy to parallelize and thus can be heavily optimized. But, these optimizations are not usually available for application-specific user-defined-aggregations (UDAs). Thus, while UDAs increase the expressivity of a system, they do so by sacrificing its efficiency. For example, when a user specifies counting as a UDA, rather than using a built-in aggregation primitive, the resulting runtime can be orders-of-magnitude slower (e.g. see [3]).

This paper proposes SYMPLE (symbolic parallel engine), a system for performing MapReduce-style groupby-aggregate queries that automatically parallelizes UDAs. In particular, given an aggregation function that iterates over a list of records *with* loop-carried dependences, SYMPLE provides a general mechanism to process chunks of the input list in parallel despite these dependences. The key idea is to treat unresolved dependences as "unknown" symbolic values and execute the UDA symbolically. The resulting symbolic execution manipulates both the symbolic expressions arising from these dependences as well as the concrete values from the input. After processing its input chunk, the execution returns a *symbolic summary* that represents its output as a function of its input dependences. SYMPLE computes these summaries in parallel for all input chunks and composes them at a final reduction step to produce an output that matches a sequential execution of the UDA. We call this notion of breaking dependences using symbolic execution *symbolic parallelism*.

Figure 1 shows an example UDA that a SYMPLE user can write (barring few syntactic differences — see Section 2.1) and which the SYMPLE runtime can automatically parallelize. This UDA processes an input list of activity events belonging to a single user from a timestamp-ordered web log. The UDA detects those items that a user (i) searched for, (ii) subsequently read at least ten reviews, and (iii) eventually purchased. Because of the data dependences in such a

SOSP'15, October 4–7, 2015, Monterey, CA.

http://dx.doi.org/10.1145/2815400.2815418

```
1  Aggregate (Key user, vector<Event>& events) {
2    // state read/updated in the loop below
3    SymBool srch_found = false; // is a bool
4    SymInt count = 0;           // is an int
5    SymVector<ItemId> ret;      // is a vector
6
7    foreach (e in events) {
8      // look for a search event
9      if (!srch_found && is_search(e)) {
10       // start counting reviews
11       srch_found = true;
12       count = 0;                              }
13
14     // count reviews
15     if (srch_found && is_review(e))
16       count++;
17
18     // on a purchase event
19     if (srch_found && is_purchase(e)) {
20       // report if count > 10
21       if (count > 10)
22         ret.push_back(e.item);
23
24       // look for the next search
25       srch_found = false;                   }}
26
27   return ret;                                }
```

Figure 1. A SYMPLE UDA that reports the items that a user purchased after searching for the item and reading at least 10 reviews.

UDA, Hadoop (and similar systems) will sequentially execute this UDA in a reducer, after a map phase that groups activity-events per user (or per user-item pair, if a user can simultaneously interact with multiple items).

In SYMPLE, programmers specify their UDAs in imperative C++ code as a loop over a sequence of records with the key requirement that the code uses *symbolic data types*, such as `SymInt` and `SymVector`, for all variables with loop-carried dependences. These data types behave like their corresponding standard C++ data types, such as integer and vector, but encode specialized decisions procedures necessary for efficient symbolic execution. In addition, these data types represent symbolic expressions in a compact canonical form that can be efficiently serialized and transferred across the network. Finally, the symbolic data types restrict the operations to only those that are amenable to efficient symbolic execution. This provides a useful guide for programmers to write efficiently-parallelizable UDAs.

1.1 Capabilities

By parallelizing such UDAs using symbolic parallelism, SYMPLE is able to *lift* the UDA computation from reducers to the mappers, just like built-in associative aggregations like `Sum` or `Max`. This lifting optimization is implemented by most data-processing systems [9, 12, 14, 39] for the built-in aggregations, and provides two important benefits. First, the sequential computation at a reducer is now performed in parallel by multiple mappers. Exposing additional parallelism is paramount for groupby-aggregate queries with little or no group-level parallelism (e.g. grouping activity per U.S. state). Second, the lifting optimization greatly reduces the amount of data exchanged between mappers and reducers. This *data-shuffle* is a critical bottleneck in data-processing systems [40], making this optimization crucial for good system performance.

Note that UDAs given to SYMPLE exploit *symbolic parallelism*, and may not be readily parallelizable with other techniques. For the example in Figure 1, using data-flow analysis to discover independent loop iterations will not lead to parallelism in the UDA computation. Further, the UDAs that SYMPLE can parallelize are not necessarily commutative as they process an ordered sequence of records. As a result, the SYMPLE runtime has to maintain the temporal order when symbolically executing UDAs and when composing symbolic summaries.

1.2 System

We implement and evaluate SYMPLE in a Hadoop system. Users of SYMPLE provide a groupby function (similar to the map function in MapReduce) that parses and groups the input based on a key, and a UDA specified using the SYMPLE library. The SYMPLE runtime translates these functions into appropriate Hadoop map and reduce functions.

To demonstrate the efficacy of SYMPLE, we evaluate groupby-aggregate queries on temporal data such as click-logs, process-logs, and telemetry data. We evaluate SYMPLE on a private 380-node Hadoop cluster that hosts Bing query logs (300GB) and Twitter tweets (1.23TB), and on a 10-instance Amazon EC2 cluster housing a 1.2TB dataset of ad impressions. Our evaluation uses queries that we believe are representative of what users might ask on these datasets. For these queries, SYMPLE reduces network communication up to several orders of magnitude and job latency by as much as 5.9x.

The ideas behind SYMPLE are independent of the underlying data-processing system and thus, are applicable to parallel databases, Dryad [16, 38], and Spark [39]. Our goal in this paper is to evaluate SYMPLE on the important application of mining patterns from temporal data with a query plan that consists of a groupby followed by an aggregate. Using symbolic parallelism to optimize more sophisticated query plans is an interesting area of future work that we do not address in this paper.

2. Symbolic Parallelism

This section describes symbolic parallelism, a general method for parallelizing user-defined aggregation functions and its instantiation in the SYMPLE system. We first describe the SYMPLE programming model.

2.1 SYMPLE Programming model

The input to SYMPLE is a groupby-aggregate query. This programming model is similar to the MapReduce programming

model [12] except that the input data is a sequence of records sorted by some field in the record, such as a timestamp. We will assume that the input data is distributed across several machines allowing us to exploit I/O parallelism. In addition, we assume that each distributed chunk has an identifier that allows the system to reconstitute the input data in the correct order.

A groupby-aggregate query consists of the following two functions

- `GroupBy`: `List<R>` $\rightarrow$ `Set<(K, List<E>)>` takes an ordered sequence of input records, parses each record to extract a key of type `K`, emits an output record of possibly different type `E` per input record, and groups these output records into per-key lists that retain the sorted order of the input list. This is executed by a MapReduce mapper.
- `Aggregate`: `(K, List<E>)` $\rightarrow$ `V` takes a key and a sequence of records and returns an aggregated result of type `V` by iterating the input list once. This function is normally executed by a MapReduce reducer, but the goal of this paper is to lift most of this computation to the mappers.

SYMPLE implements the `Aggregate` function with the following template:

```
V Aggregate (K key, List<E> input) {
  State s;  //init aggregation state
  foreach ( e in input )
    // update state based on e
    Update(s, e);
  return Result(s);                }
```

and requires the user to provide an initial aggregation state, an `Update` function that updates the state for each record in the input list, and a `Result` function that extracts the result from the aggregation state. The programmer is responsible for capturing all the side-effects of the `Update` function in the aggregation state and for ensuring that `Result` is a pure function. Figure 1 shows an example of an aggregate function that fits the template above. For ease of exposition, we inline the `Update` and the `Result` functions as in Figure 1, while in practice, users have to use appropriate C++ lambda functions to achieve the same effect. Similarly, we inline the `State` structure in our examples.

2.2 Breaking dependences

The UDA above has an obvious loop-carried dependence — every iteration reads and modifies the aggregation state. Symbolic parallelism breaks this dependence using symbolic execution [17]. Figure 2 describes the instantiation of symbolic parallelism in SYMPLE. Each mapper processes a file segment and groups the records into a per-key list. These chunks for a given key, stitched in the right order, form the input to a call to the UDA. In the standard MapReduce, a reducer collects all these chunks and calls the UDA sequentially.

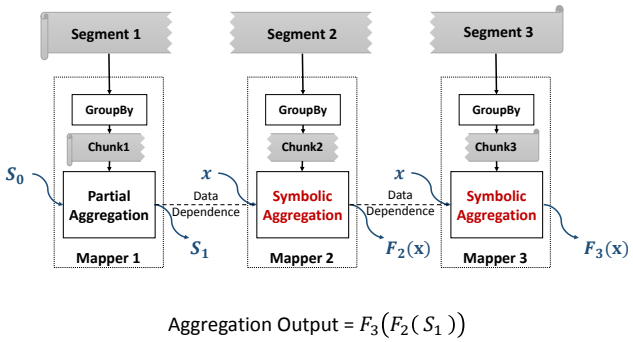

Figure 2. Using symbolic execution to break dependence in aggregation. The latter two mappers symbolically execute the UDA to compute symbolic summaries F_2 and F_3. The aggregation output is composition of these summaries with the output S_1 from the first mapper.

In SYMPLE, however, each mapper executes the UDA on its chunk (for every key). While the first mapper can start from the initial aggregation state, all other mappers depend on the output from the previous mapper, as shown by the dashed-lines in Figure 2. To break this dependence, these mappers execute the UDA symbolically from an "unknown" symbolic state x. In a symbolic execution, variables contain symbolic expressions and programs manipulate these expressions rather than concrete values. For example, a variable containing the expression x becomes $x + 1$ on an increment.

The goal of each mapper is to create a *symbolic summary* of the output aggregation state as a function of its input x. In the figure, the latter two mappers produce summaries $F_2(x)$ and $F_3(x)$. These summaries represent a partial-evaluation [21] of the UDA on the respective chunks. As a result, evaluating these summaries for a given x does not require rereading the chunk data. The mappers send the summaries to a reducer which composes them in the right order to produce the output of the aggregation. In the figure, $F_3(F_2(S_1))$ is the desired output, where S_1 is the output of the first mapper.

2.3 Challenges

There are several challenges in making symbolic parallelism feasible, in practice. First, symbolic execution must be acceptably fast when compared to the concrete execution. At the minimum, mappers should be able to process data at disk speeds *despite* the overheads of manipulating symbolic expressions. Second, the symbolic expression should be represented in a compact form for efficient serialization and transfer across the network.

The third challenge, which has direct impact on the two challenges above, is the *path explosion* problem. When a symbolic execution incurs a branch that depends on the symbolic input, execution has to proceed for both possible outcomes. Thus, while a concrete execution executes one

path through the UDA, a symbolic execution might explore a number of paths exponential in the size of the input. Even worse, in the presence of loops, the number of paths could be unbounded. One syntactic way to mitigate this latter issue is to require programmers to avoid loops that depend on the aggregation state. Nevertheless, the potential of exponential blowups needs to be controlled.

Finally, in contrast to its applications in verification and testing, the symbolic execution in SYMPLE has to be both sound *and* precise — leaving no room for under- or over-approximations. Any approximation can cause the output to deviate from the sequential semantics. While it is interesting to explore applications that can accept approximate outputs, we do not consider those design points in this paper.

3. Efficient symbolic execution

This section describes how SYMPLE implements efficient symbolic execution and the design decisions that let us handle the challenges described in the previous section.

3.1 Running example

We will use a simple aggregate function below that computes the maximum of a list of integers as our running example.

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

Obviously, `Max` is an associative operation and is thus readily parallelizable. However, this is not apparent when the computation is presented imperatively as shown above. SYMPLE can automatically parallelize this function. An interesting side-effect of this is that SYMPLE programmers no longer have to distinguish between in-built aggregation functions, such as `Sum` or `Max`, from not-so-readily-parallelizable aggregations, such as the one in Figure 1.

The symbolic execution in SYMPLE is encapsulated in symbolic data types, like `SymInt`, used above. With some restrictions, programmers can use these data types exactly like the corresponding standard C++ types, such as integers. Section 4 describes the implementation of these data types. Programmers are responsible for encoding the aggregation state in its entirety with symbolic data types.

Consider a run of the `Max` function on an input list of nine elements split into three chunks:

```
first = [2,9,1]  second = [5,3,10]  third = [8,2,1]
```

The first chunk executes concretely as usual and produces a result of 9. The second and the third chunk, executed in parallel with the first, are unaware of this outcome and thus start from a symbolic value for the `max` variable, say x and y respectively.

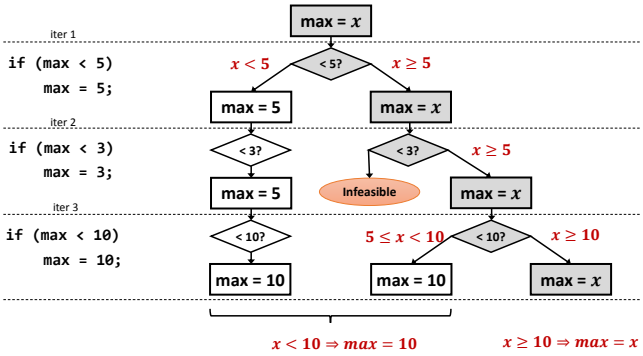

Figure 3. Symbolic execution of the `Max` function on an input list `[5,3,10]` generating a symbolic summary shown above.

3.2 Symbolic Summaries

Figure 3 shows the symbolic execution for the second chunk. In the first iteration, the execution compares `max`, which currently holds the symbolic value x, with the first input 5. Since both outcomes of the branch are feasible, symbolic execution explores both branches. When exploring the *then* branch, we know that $x < 5$. Symbolic execution adds this predicate to the current *path constraint*. A path constraint determines the conditions on the initial state that forces the execution along a given path. When $x < 5$, the `Max` function updates the `max` variable to 5. In the *else* branch, `max` remains as x. These two outcomes are shown in Figure 3 as the children of the first decision point. The state on the left is unshaded as `max` now contains a concrete value 5.

After the first iteration, the two outcomes represent the following symbolic summary.

$$\begin{array}{lcll} x < 5 & \Rightarrow & max = 5 & \wedge \\ x \geq 5 & \Rightarrow & max = x & \end{array}$$

In general, a symbolic summary is of the form

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

Here, PC_i represents the path constraint and TF_i represents the *transfer function* that determines the current state s as a function of the initial input x. This essentially means that if the UDA is executed with an initial aggregation state that satisfies $PC_i(x)$, the aggregation state after executing the UDA to the current point will be $TF_i(x)$.

A valid symbolic summary has $\bigvee_i PC_i(x) = \texttt{true}$ and for all $i \neq j$, $PC_i \wedge PC_j = \texttt{false}$. This means that a summary covers all possible execution paths of the UDA.

3.3 Decision Procedures

The second iteration compares `max` with 3. The symbolic execution considers each path constraint in the summary individually. Under the path constraint $x < 5$, `max` has a concrete value of 5 and we can right away conclude that the

branch is not taken without any symbolic reasoning. Figure 3 emphasizes this by showing only one outcome for branches on concrete paths.

The more interesting case is the path constraint $x \geq 5$, under which $max = x$ is symbolic. On the branch $x < 3$, the symbolic execution should check the feasibility of the two branch outcomes: $x < 3$ and $x \geq 3$ for the current path constraint. The *then* branch is not feasible while the *else* branch is. Thus, the symbolic execution has to only execute the latter path.

In general, when exploring the path constraint $PC(x)$ in which the current aggregation state is $s = TF(x)$, on a branch with a predicate $P(s)$, the symbolic execution needs to check the feasibility of $PC(x) \wedge P(TF(x))$ and $PC(x) \wedge \neg P(TF(x))$. Doing so and pruning infeasible paths from execution is crucial in limiting the path explosion problem. To make such feasibility decisions, we need *decision procedures* that can reason about symbolic constraints.

One could potentially use decision procedures implemented in general purpose theorem provers [11] here. However, calling into an external prover several times for every input record will be unacceptably slow even with recent advances in these provers. Instead, SYMPLE encodes special-purpose efficient decision procedures in symbolic data types.

3.4 Canonical form

Each data type maintains path constraints in an easy-to-reason canonical form. For instance, the `SymInt`s only maintain constraints of the form $lb \leq x \leq ub$ for some constants lb, ub. This canonical form enables deciding the feasibility of branches in (small) constant times. In addition, they recursively enable efficient simplification of path constraints into canonical forms. For instance, in our running example, the path constraint on the *else* branch in second iteration is $x \geq 5 \wedge x \geq 3$. This can be simplified into $x \geq 5$. Finally, the canonical forms enable efficient serialization of symbolic constraints.

To maintain the constraints in canonical form, each symbolic data type restricts the operations allowed on variables of that type. For instance, `SymInt`s do not allow comparison with another `SymInt`. This is a careful trade-off between expressivity of SYMPLE programs and the efficiency of decision procedures.

Going back to our running example, the path constraint simplification, along with the infeasibility of the *then* branch results in the following summary after the second iteration.

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

which is the same as the summary after the first iteration.

3.5 Path merging

The third iteration compares `max` with 10. As before, the concrete case of $x < 5$ is straightforward — `max` with a current value of 5 is updated to 10. For the second path constraint of $x \geq 5$, the execution needs to explore both branch outcomes, resulting in the following summary.

$$\begin{array}{rcll} x < 5 & \Rightarrow & max = 10 & \wedge \\ 5 \leq x < 10 & \Rightarrow & max = 10 & \wedge \\ x \geq 10 & \Rightarrow & max = x & \end{array}$$

as shown in Figure 3.

The first two path constraints produce the same transfer function $max = 10$. This means that the fate of symbolic execution from this point will be the same for input states that satisfy these two path constraints. Thus, we can avoid redundant exploration by *merging* these two paths. Doing so is important to alleviate the path explosion problem. In general, if the summary contains $PC_1(x) \Rightarrow s = F(x)$ and $PC_2(x) \Rightarrow s = F(x)$, the path constraints can be merged into

$$(PC_1(x) \vee PC_2(x)) \Rightarrow s = F(x)$$

Such a merging is useful only if the disjunction of path constraints can be converted into the canonical form. In the running example, the decision procedure in `SymInt` is able to represent $x < 5 \vee 5 \leq x < 10$ simply as $x < 10$. This results in the following summary.

$$\begin{array}{rcll} x < 10 & \Rightarrow & max = 10 & \wedge \\ x \geq 10 & \Rightarrow & max = x & \end{array}$$

The final summary tells us that if the input to the second chunk x is less than 10, then the output of `Max` is 10, else it is x. Of course, we would have come to the same conclusion *if* we had known that maximum is an associative function and that the maximum of a list is the maximum of the partial-maximum of its chunks.

3.6 Summary composition

Using arguments similar to the one above, we can see that symbolically executing the third chunk with an initial value of y will produce the summary

$$\begin{array}{rcll} y < 8 & \Rightarrow & max = 8 & \wedge \\ y \geq 8 & \Rightarrow & max = y & \end{array}$$

Let $S_2(x)$ and $S_3(y)$ respectively refer to the summary of the second and third chunks. The first chunk runs concretely and produces a value of 9. In order to produce the outcome of the entire computation, SYMPLE applies the summaries in the order of the input chunks $S_3(S_2(9)) = 10$. Note, if the summaries are sound and precise, this results in exactly the same output as running the computation sequentially.

In general, if C_1 is the concrete output of the first chunk, and $S_2, S_3, ...S_n$ are the symbolic summaries of the remaining $n - 1$ chunks, the ouptut of the computation is given by

$$S_n(\ldots(S_3(S_2(C_1))))$$

This evaluation can be done sequentially at the reducer.

Alternately, one can further parallelize this computation as function composition is associative. That is, rather than compute $S_3(S_2(9))$, one can first compose the two summaries $S_3 \circ S_2$ and apply the composed function to the output of the first chunk. To do this composition, we start by renaming max in the second-chunk summary to y, as the output of the second chunk is now the input to the third chunk. Now, we perform a cross-product of the conjuncts in the two summaries and eliminate infeasible paths. For example, the first conjunct in the second-chunk summary has an outcome $y = 10$, which when composed the second conjunct in the summary above produces the constraint $x < 10 \Rightarrow max = 10$. In effect, $S_3 \circ S_2$ results in the following summary

$$\begin{array}{lll} x < 10 & \Rightarrow & max = 10 \quad \wedge \\ x \geq 10 & \Rightarrow & max = x \end{array}$$

4. Symbolic data types

Symbolic data types maintain path constraints in a canonical form which in turn enable efficient decision procedures. Programs manipulate these data types almost exactly like they manipulate standard types such as integers. For instance, the `Update` function in Figure 1 works as stated with `SymBool`, `SymInt`, and `SymVector` data types and the appropriate operator overloading. These operators provide a restricted set of operations over their corresponding standard data types in order to allow efficient symbolic execution. For instance, the `SymInt` data type does not allow division operators.

4.1 Symbolic enumerations

The `SymEnum` is a symbolic version of C++ `enum` class that supports checking equality or inequality with and assignment to integral constants. Users can use `SymEnum`s to represent any state with bounded values.

Canonical Form A `SymEnum` variable v maintains its current symbolic summary in the form:

$$x \in S \Rightarrow v = (bound\ ?\ c : x)$$

That is, an instance of `SymEnum` contains a bit-vector S, a Boolean variable $bound$, and an integer constant c. When $bound$ is true, the variable has the concrete value c. Otherwise, v is symbolic and can contain any value in S. `SymEnum` overloads the equality and inequality operators that take an integer as an argument. In particular, to maintain the canonical form above, two `SymEnum`s cannot be compared. The value of a `SymEnum` is bound on an assignment to a constant. Once bound, `SymEnum`s are as fast as a C++ enum but for the $bound$ check.

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. This can be determined efficiently from the bit-vector.

Merging Path Constraints Two path constraints $x \in S_1$ and $x \in S_2$ can be merged into $x \in S_1 \cup S_2$.

4.2 Symbolic Booleans

Note that `SymBool` is an instance of `SymEnum` over the bounded set `true`, `false` with the appropriate operator overloading with boolean constants.

4.3 Symbolic integers

The `SymInt` is a symbolic version of C++ integer data type (parametrized with the desired bit length) that supports addition, subtraction, multiplication, and standard comparison operations. A conscious design decision is to only allow operations between a `SymInt` and a concrete integer. In particular, the type system prevents adding two `SymInt`s or comparing them. This ensures that all the constraints generated during symbolic execution contain a single symbolic variable. This greatly simplifies the decision procedure as otherwise SYMPLE would have to call into a general-purpose integer-linear solver with worst-case exponential time complexity. In contrast, `SymInt` constraints can be decided in (a small) constant time.

Canonical Form Each `SymInt` variable contains four values (lb, ub, a, b) interpreted as follows. If x is the initial symbolic value of this variable, under the path constraint $lb \leq x \leq ub$, the current value of the variable is $a * x + b$. Operations that update variable update a and b appropriately. For example, incrementing a `SymInt` amounts to incrementing b.

Decision Procedures When comparing a `SymInt` with another constant, the two outcomes split the interval $[lb, ub]$ into two (possibly empty) intervals. For instance, we know that `SymInt` $s = (lb, ub, a, b) \leq c$ holds whenever $a.x + b \leq c$. Assuming $a > 0$ (the other case is similar), this implies a constraint $x \leq \lfloor (c - b)/a \rfloor$. Calling this new bound nb, we have two possible intervals — $[lb, nb]$ under which $s \leq c$ holds and $(nb, ub]$ under which $s \leq c$ does not hold. The symbolic execution has to explore those outcomes in which the corresponding intervals are not empty.

Merging Path Constraints If the symbolic summary contains two `SymInt` symbolic data types which have overlapping bounds and the same transfer function, SYMPLE merges those two `SymInts` into a one such that the lower bound is the min of the two and the upper bound is the max of the two.

4.4 Black-box predicates

There are instances where programmers need the flexibility to evaluate complex predicates on the aggregation state and for which symbolic reasoning is impossible or not practical. SYMPLE provides a `SymPred` data type for this purpose. A `SymPred<T>` is essentially a place holder for a, possibly symbolic, value of type `T` with two operations: (a) assigning a value of type `T` and (b) evaluating a pre-specified but arbitrary "black-box" predicate between a `SymPred<T>` and `T`.

Consider the following example, where the programmer wants to split a sequence of GPS events into sessions, where each session is defined as a contiguous sequence in which every event is within some bounded distance from the one prior. The function `CountEventsInSessions` below breaks the input into sessions by remembering the previously seen event and computing its distance from the current event. The `evalPred` function performs this check by calling the `distanceLessThanBound` function, provided as an argument to the constructor of `prev`.

```
bool distanceLessThanBound(GPSCoord sym, GPSCoord
     val) {
 // check if distance between sym and val
 // is less than a given bound
}

CountEventsInSessions(K users, List<Event> events) {
  SymInt count = 0;
  SymVector<SymInt> counts;
  SymPred<GPSCoord> prev( distanceLessThanBound );

  foreach( event in events ) {
    if (prev.evalPred(event.gpsCoord))
      // same session
      count++;
    else {
      // reset;
      counts.push_back(count);
      count = 0;                                    }
    prev.setValue(event.gpsCoord);                  }
  return (users, counts);                           }
```

`SymPred`s allow the parallelization of this code by breaking the dependence on `prev`. Initially, `prev` represents an unknown symbolic `GPSCorrd` value. On the first event, this code requires computing the distance between a symbolic value and a concrete value, a nonlinear computation that is not amenable to symbolic reasoning.

Instead, SYMPLE blindly explores both outcomes of the branch. Unfettered use of such `SymPred`s can obviously lead to exponential path explosion. But an important point to note here is that, in the program above, `prev` is assigned to a concrete value in both branches when processing the first event. Subsequent events can simply be processed concretely. As a result, there can at most be a path blowup of two.

This pattern of *windowed* dependence, where the UDA only depends on a small number of previous events, is fairly common. As such, for these queries, `SymPred`s provide powerful expressiveness without exponential path blowups. All the queries we evaluate in this paper use a window of size one.

4.5 Symbolic Aggregates

Symbolic Vector Inspired by reducer HyperObjects [13] in Cilk, SYMPLE implements a `SymVector<T>`, a vector of a possibly symbolic type T that can only be appended to. `SymVector`s are useful for capturing the output of a UDA computation. Like reducer HyperObjects, each symbolic UDA computation appends to a local vector which is then stitched in the right order at summary composition. Additionally, a `SymVector` symbolically evaluates its elements on a composition, converting them to concrete values when appropriate. For instance, if a UDA appends a symbolic count variable in the vector, say $x + 5$, then later on a composition that resolves x to a concrete value, the vector concretizes all elements that depend on x.

Symbolic Struct Programmers can create symbolic structs that contain fields of other symbolic data types. These structs can themselves be used as symbolic data types. The one implementation complexity here is that with lack of reflection support in C++, it is not possible to cleanly enumerate all fields, say for serialization, without programmer support. In SYMPLE, the programmer provides a `list_fields` function that returns a tuple of all fields with symbolic data types.

Other data types In addition to these data types, SYMPLE exposes a C++ interface for specifying new data types. This provides a modular way to increase the expressivity of SYMPLE. Of course, these user-provided data types should (i) have a canonical form, (ii) implement efficient decision procedures, (iii) implement a merge function so as to enable efficient representation of symbolic summaries, and (iv) serialization functions to enable network transfers.

5. Implementation

This section describes how SYMPLE implements the ideas discussed above.

5.1 Systematically exploring paths

Given an aggregation state specified as a symbolic data type, the current concrete input, an update function for updating this state on each input, and a result function for extracting the return value of the UDA, SYMPLE repeatedly executes the update function per input record with the intention that each execution takes a different feasible path.

SYMPLE implements this systematic exploration as follows. In each run, SYMPLE maintains a *choice* vector of branch outcomes that represents a feasible path through the program. Considering the running example of `Max` from Section 3.1, and assuming that the update function is unrolled three times, then Figure 3 shows the three feasible paths possible for a single invocation of the update function. We can encode the three paths as a sequence of binary outcomes: 0, 10, 11, where each bit represents a branch at which both outcomes are feasible, a 0 represents the execution on a *then* branch, 1 represents the execution on a *else* branch.

The goal of the symbolic execution is to explore this space lexicographically. Starting from the first run with an empty choice vector, each symbolic data type that incurs a choice at a branch uses the next bit in the choice vector to determine the path to follow. After exhausting all bits in the vector, the data types append a 0 to the vector and take the *then* branch. At the end of the run, SYMPLE "increments" the choice vector

by popping 1s at the end of the vector and converting the last 0 to a 1. This is lexicographically the next path to explore. This process repeats till the choice vector has no 0s.

SYMPLE does this exploration by judicious use of operator overloading and copy constructors. In particular, SYMPLE requires all symbolic data types overload comparison operators (i.e., those which can incur a choice) such that they are able to lexicographically explore all paths through a UDA. As a consequence, SYMPLE's symbolic exploration is just a library and does not require any compiler support.

5.2 Dealing with path explosion

Controlling the number of paths explored during symbolic execution is crucial to the efficacy of SYMPLE. As discussed above, each symbolic data type alleviates path explosion by pruning out paths that are provably infeasible and by merging paths that have the same transfer function. In addition, `SymEnum`s and windowed `SymPred`s bound the explosion possible on variables of those types. Nevertheless, path explosion is possible with `SymInt`s.

SYMPLE attempts to perform merging as soon as the number of paths exceeds a previously reached maximum for the number of paths. For example, in the max function as in our example in Figure 3, every time the number of paths reaches three, SYMPLE looks for merge candidates and reduces the paths down to two.

SYMPLE contains two other mechanisms for handling path explosion. First, if the number of paths explored on a *single* input exceeds a maximum bound, then SYMPLE halts with a warning that the UDA potentially has a loop that depends on the aggregation state. Second, as the UDA is processing inputs, if the total number of symbolic paths exceeds a bound (currently set to 8), then SYMPLE uses this to trade parallelism for sequential efficiency. Rather than proceed with the current symbolic execution, SYMPLE stores the current summary and restarts the symbolic execution from a fresh "unknown" symbolic variable. This way, each mapper, rather than producing a single summary, produces multiple summaries that have to composed at the reducer. This behavior serves two purposes. First, SYMPLE dynamically identifies chunks of the input where achieving symbolic parallelism is easy. Second, it provides a fallback to no parallelization in the worst case when the UDA has no symbolic parallelism.

5.3 SYMPLE C++ library and verification of user code

The current version of SYMPLE relies on the C++ type checker to detect a large number of possible errors, however it relies on the user to provide code in the following pattern:

```
struct State {
  // SymTypes
  ...
  tuple<...> list_fields() ...
} state;

int main(int argc, char** argv) {
  MapReduceMain(
      ... // input, output, groupby and
      ,   // configuration parameters
      [](const GroupByKey& key,
         const InputRecord& record,
         State* state) {
        ... // code for UDA:
      });
}
```

The user-provided lambda function is passed to a function pointer and thus cannot capture local variables. Instead, SYMPLE assumes that all the state is stored in the `State` structure and the user does not modify any global variables in the lambda function. Additionally, SYMPLE relies on a user-provided `list_fields` method for the `State` structure that produces a tuple of references to all the fields in the structure. This enables SYMPLE to serialize the `State` structure without compiler support. If a future versions of C++ supports static reflection, this function would no longer be necessary.

Then, using the C++ type system and the `list_fields` method, SYMPLE statically checks that only `Sym` types are used in the `State` structure and that all used operations on symbolic types are valid methods.

5.4 Mapping SYMPLE to MapReduce

SYMPLE does not include its own framework for distributed processing, but relies on an existing MapReduce implementation such as Apache Hadoop. The MapReduce framework takes care of groupby parallelism and SYMPLE parallelizes within one group using symbolic parallelism. SYMPLE treats the records within a group as one single sequence coming in the order as in the input data.

Since MapReduce treats groupby records as a set instead of a sequence, SYMPLE alters every input record R to be a triple $(\texttt{mapper\_id}, \texttt{record\_id}, R)$ to keep track of the order of the input records. Here $\texttt{mapper\_id} \in \mathbb{N}$ is a sequential index of the map task, $\texttt{record\_id} \in \mathbb{N}$ is a sequential index of the record within the file processed by a map task. Then, for each group in the groupby, a SYMPLE map task sends a set of records of the type $(\texttt{mapper\_id}, \texttt{record\_id}, S)$ to the MapReduce shuffle phase. Here `mapper_id` is the map task id, `record_id` is the index of the last record processed by the mapper for the group and S is the symbolic summary. In the shuffle phase, the records of each group are sorted lexicographically by `mapper_id` and `record_id` which orders the symbolic summaries according to their order in the input data. Finally, SYMPLE uses the reduce phase to combine the symbolic summaries for each group.

6. Evaluation

This section evaluates the efficacy of SYMPLE on three scenarios: (a) on a multi-core machine to study the CPU overheads of symbolic execution and ensuring that SYMPLE can scale on multiple cores without being constrained by the memory bandwidth, (b) on a 5-to-10-node-instance private Hadoop cluster on Amazon Elastic MapReduce (EMR) to

study the reduction in data shuffled across machines and the reduction in end-to-end job latency, and (c) on a 380-node shared Hadoop cluster to study the scalability of SYMPLE on a large cluster.

6.1 Data and queries

We used several datasets for evaluation: `github`, which contains repository operations between February 2011 and September 2014 (419GB) [1], an Amazon Redshift benchmark data [2] - a 1.2TB dataset representing 4 months of ad impressions, `Bing`, which contains 1.9 billion queries from the `Bing` search engine (300GB), and `Twitter`, which contains all Tweets in a 24 hour period (1.23TB). The `Bing` and `Twitter` data sets are hosted in the 380-node cluster and are not allowed to leave this cluster for privacy and legal reasons. Thus, we only use the `github` and Redshift data sets for the multi-core and EMR scenarios.

Table 1 lists 12 queries with UDAs that mine temporal patterns in the data sets above. These queries are inspired by real-world scenarios, such as finding partial service outages, usage patterns, and spam detection. We took best efforts to replicate the complexity of UDAs that arise in practice. For instance, we are aware of specialized systems that are designed to pre-process and index query logs for efficiently processing queries similar to the `Bing` queries in Table 1. By making such queries efficient in Hadoop, our hope is that SYMPLE would obviate the need for such specialized solutions in the future. Note for privacy reasons, we do not show the number of groups for the proprietary `Bing` and `Twitter` queries (beyond B1, which has just 1 group).

The symbolic data types presented in this paper are sufficient to express all the queries in Table 1. This should not be surprising as the design of these data types coincided with our evaluation of SYMPLE on these queries. However, we do believe that current set of symbolic data types are expressive enough for useful UDAs outside of our evaluation set. Our queries were relatively easy to write and range anywhere from 40 to 100 lines of C++ code.

We also made an explicit choice to pick queries and datasets with different number of groups in the groupBy. The amount of groupby parallelism affects the efficacy of symbolic execution. At one extreme, when the query has a single group (B1), symbolic parallelism is the only source of parallelism. At the other extreme, for queries with large number of groups (B3, T1), it is likely that each mapper processes at most one event per group and SYMPLE has little opportunity for improvement.

6.2 Multi-core evaluation

The primary motivation for evaluating the multi-core scenario is to study the performance implications of the SYMPLE symbolic execution in a setting that excludes the overheads

[1] http://githubarchive.org

[2] https://github.com/hapyrus/redshift-benchmark

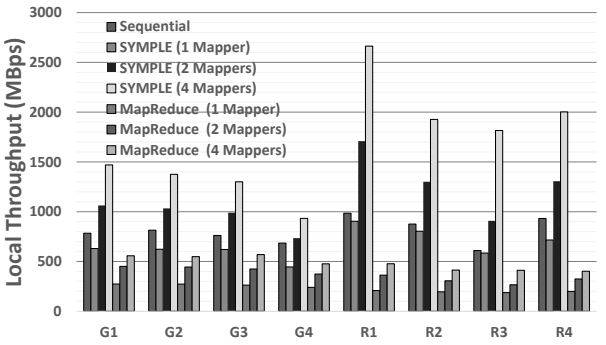

Figure 4. Throughput on a multi-core machine.

of the Hadoop framework. We also used the single-machine version of SYMPLE to identify and fix performance problems during development.

Our evaluation seeks the answer to the following questions.

- How much CPU overhead does symbolic execution add over concrete execution?
- Can SYMPLE process data at the speeds of a commodity hard disk?
- Do the memory overheads of symbolic execution cause SYMPLE to hit the memory-bandwidth bottleneck?

To this end, we study the following single-machine implementations:

Sequential: Given input data and a query, this baseline reads data sequentially and executes the UDA concretely.

Local MapReduce: This simulates a single-machine MapReduce with multiple processes and pipes. Each mapper process reads a part of the input, performs groupby, and pipes the result to reducer processes. We use Unix `sort` to sort mapper results by groupby key and merge (`sort -m`) to per-key lists. Each mapper is optimized to only send input record fields that are used by the UDAs.

Local* SYMPLE*: implements SYMPLE on the single-machine simulation of MapReduce discussed above. Each mapper groups the input as per the groupby key and additionally executes the UDA symbolically on the output. The resulting symbolic summaries are sorted, merged (via `sort`) and sent to a single reducer process.

For this evaluation, we used a single 4-core Core i7 4770K processor workstation, with 16GB RAM, 256GB of SSD disk and running 64-bit Ubuntu 14.04. As a primarily goal of this experiment is to determine if SYMPLE can symbolically execute faster than the disk, we restrict queries to 4.45GB of `github`'s data and 3.19GB of the RedShift Benchmark data that fit in memory. Before reporting numbers, we read the data to keep it in cache and remove the effect of disk I/O.

ID	Description	# Groups	Sym Types Used		
			Enum	Int	Pred
419 GB List of GitHub operations on repositories from Feb 2011 to Sep 2014.					
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	
300GB of Query Logs from the Bing search engine containing 1.9 billion queries					
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
1.23TB of logs from Twitter that represent all tweets in a 24 hour interval					
T1	Spam learning speed — no. queries not marked as spam, followed by at least 5 queries marked as spam per hashtag	*	y	y	
1.2TB of ad impression logs from RedShift benchmark					
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

Table 1. A summary of the datasets and queries performed on each dataset

Figure 4 summarizes these in-memory experiments. Each bar gives the throughput (MB/s) of a particular configuration on eight queries. The sequential baseline is fast: running at least $6\times$ faster than a commodity disk (100 MB/s). The SYMPLE (1 mapper) bars represent two overheads: the symbolic execution overhead (as the only mapper computes symbolic summaries) and the overhead of inter-process communication through pipes. Both together result in an overhead of 4% to 35% for the eight queries, with an average of 22%. This is a conservative estimate of the cost of symbolic execution. Moreover, for all of the queries, Local SYMPLE scales with the number of mappers, suggesting that the computation is not memory bound.

Finally, the Local MapReduce implementation demonstrates poor throughput ($3.6\times$ lower throughput than Local SYMPLE with 4 mappers, on average). The reason is that Local MapReduce has significant overhead in shuffling large amounts of data before sending it to reducers, which dominates the runtime; the SYMPLE runtime lifts the UDA into mappers and thus significantly reduces this overhead. These numbers suggest that symbolic parallelism could potentially benefit UDA optimizations for in-memory databases and streaming engines.

To summarize, Local SYMPLE is significantly faster than a commodity hard disk implying that the cost of computing symbolic summaries will not be the bottleneck in a larger Hadoop cluster. Further, SYMPLE demonstrates good scaling (i.e., by adding more mappers), which implies the added cost (both in CPU utilization and memory consumption) of symbolic execution pays for itself when considering the significant savings it incurs by lifting the UDA to mappers, and thus reducing the overheads of shuffling.

6.3 Elastic MapReduce evaluation

When dealing with large amounts of data, computation often has to go to where the data are. This section evaluates the efficacy of SYMPLE on Amazon Elastic Mapreduce [3]—a managed Hadoop cluster that runs on top of dynamically allocated virtual machine instances. We evaluate on data stored in Amazon S3 [4]—a scalable storage system that holds (potentially large) data objects also accessible via Hadoop file system bindings. Elastic MapReduce lets a user allocate virtual machines only for the duration of the computation and thus shorter computation directly reduces the monetary cost of the job. As a consequence, end-to-end latency is a critical metric and we show that SYMPLE reduces job latency.

Data We report numbers for two of our datasets that we could use in the Amazon Elastic MapReduce setting – `github` and RedShift Benchmark data. Both datasets consist of a sequence of relatively large (around 1KB) records with various fields, which means that most queries will read through the datasets and discard most of their fields.

To also include testing results on data that includes fewer fields or is stored in a fashion similar to a columnar storage, we included two variants of the RedShift data. The first (*complete*) variant includes all fields of all records, while the second (*condensed*) variant only keeps the four columns we use – advertiser, campaign, timestamp and country. This condensed variant reduces the dataset size to 50GB and avoids scanning through columns that are then discarded. We perform queries R1–R4 on the variant with the complete RedShift data, and queries R1c–R4c on the condensed variant

[3] http://aws.amazon.com/elasticmapreduce/

[4] http://aws.amazon.com/s3/

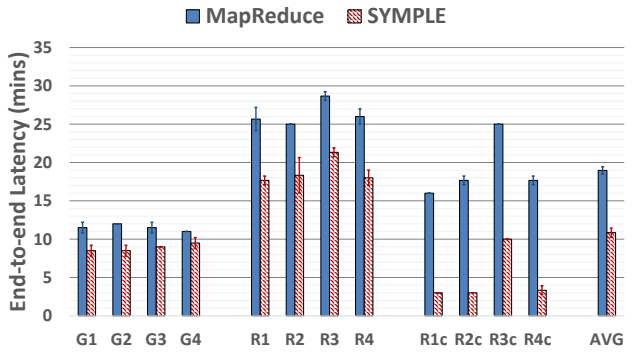

Figure 5. Amazon EMR end-to-end job latency.

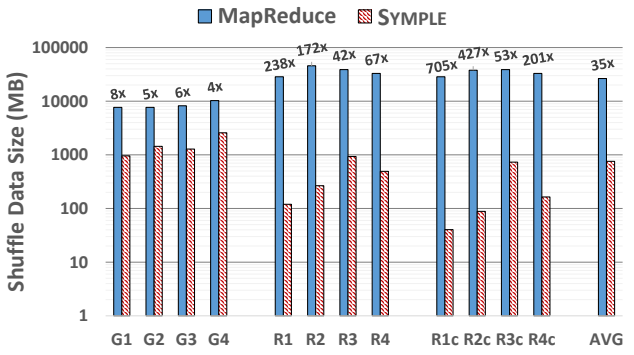

Figure 6. Amazon EMR shuffle data reduction. Note the log y axis.

of the data. All the input data is stored in Amazon S3 in a gzipped format.

Configuration For our experiments, we used m3.xlarge instances, each with 4 virtual CPUs, 15GB RAM, and 2x40GB of SSD storage. A virtual CPU of these instances corresponds to a thread of a 2.5GHz hyper-threaded Xeon CPU. We used 10 instances for the complete version of the RedShift dataset, and 5 instances for the condensed version of RedShift dataset and for the `github` experiments.

For our experiments, we used Amazon Hadoop 2.4.0 and we set the number of reducers to the number of machines. To speed up data processing, we implemented a pipeline that reads the data directly from Amazon S3 via http, decompresses it and streams it to the C++ map and reduce tasks, thus avoiding double-parsing of the data both by Hadoop and our pipeline. Apache Hadoop then takes care only of the data shuffling and sorting and then streams it into our reducer tasks. Our whole pipeline is very efficient and we checked that it manages to saturate the inbound network of the machine instances. These improvements are effective for our baseline as well as SYMPLE.

Baseline MapReduce: is a hand-optimized Hadoop baseline. The groupby executes in the mapper while the UDA executes in the reducer. The groupby only emits fields of the input record that are used in the UDA.

SYMPLE implements the SYMPLE algorithm wherein both the groupby and UDA execute in the mapper (the UDA symbolically) and a reducer composes those symbolic summaries to produce the jobs output.

Results We summarize the results of our experiments in Figure 5 and Figure 6. A bar in Figure 5 gives the runtime (in minutes) for our baseline and SYMPLE on the three datasets. For the G1–G4 and R1–R4 queries, the baseline MapReduce takes between 15% and 45% longer to execute than the SYMPLE MapReduce. On the other hand, for the R1c–R4c queries, the speed-up from using SYMPLE MapReduce is between 2.5× and 5.9× compared to the baseline.

Note that when running on the RedShift Benchmark *condensed* data, the speed of both our baseline and SYMPLE are higher that on the *complete* RedShift data. This is because each mapper reads significantly less from Amazon S3. As a consequence, the runtime is dominated by reading data from S3 (i.e., both the baseline and SYMPLE saturate the connection between S3 and the compute cluster), which dampens SYMPLE's improvement over the baseline. However, on the *condensed* variant of the data, SYMPLE provides a significant runtime savings as the connection to S3 is no longer a bottleneck: SYMPLE is over 5× faster for the R1c, R2c and R4c queries and 2.5× for R3c. We profiled R3c and found its runtime is dominated by C standard lib datetime parsing, which slows all versions of the query. In other words, symbolically executing a UDA was not the bottleneck (parsing the data was).

The overall reason for such large speedups is because SYMPLE significantly reduces the amount of data shuffled from our map tasks to the reduce tasks. We summarize these results in Figure 6. The baseline MapReduce shuffles between 7.7GB and 10.3GB for the `github` dataset and between 28.5GB and 45.6GB of data for the RedShift Benchmark dataset. The SYMPLE MapReduce reduces that data transfer significantly. For the `github` dataset, where the queries also have a lot of groupby parallelism, the savings are between 4 and 8 times. In the RedShift data queries, there are much fewer groups the bandwidth savings are around two orders of magnitude. These large communication efficiency improvements do not necessarily translate to speed-ups due to other overheads such as data reading, decompressing, parsing, etc.

6.4 Large cluster evaluation

The prior two sections demonstrate (i) SYMPLE runs faster than a commodity disk and (ii) SYMPLE significantly reduces network bandwidth by lifting a UDA into a mapper, and thus saving on an expensive shuffle. This section demonstrates SYMPLE generates code which scales on a large and shared Hadoop cluster which houses interesting commercial data. We evaluated SYMPLE on a data-center with 380 machines, each with 16 Intel Xeon CPU E5-2450L cores running at

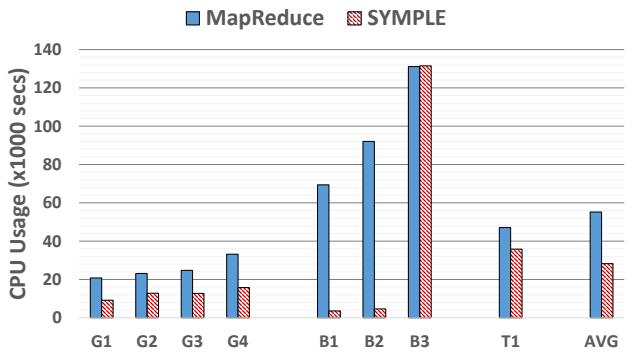

Figure 7. CPU usage for running 8 queries on a 380-node Apache Hadoop cluster.

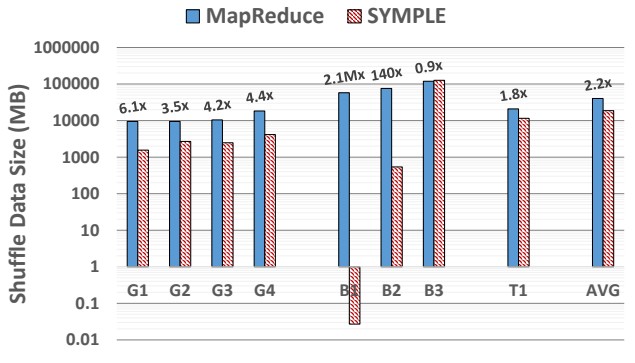

Figure 8. Amount of shuffled data for executing 8 queries on a 380-node Apache Hadoop cluster.

1.80GHz and 192GB of RAM, running Windows Server 2012 Data-center and HortonWorks Hadoop 2.4. We determine the number of mappers by the number of files containing the input (`github` uses 405 mappers, `Bing` uses 199, and `Twitter` uses 501). We set the number of reducers to 50 to ensure jobs are not limited by the latency of any one reducer.

In this setting, we are using a shared, batching, cluster and the majority of the time is taken in scheduling our job in the cluster and performing tasks that are uncontrollable by us. Once the job is scheduled, it often (except for some baselines) completes in a few minutes. We expect such clusters to be heavily used by multiple concurrent jobs using the resources of the machines. Thus we use the overall CPU usage and the amount of data used in a shuffle as our key metrics, as reducing both helps maintain the health of the overall cluster.

CPU Resources A bar in Figure 7 (a) provides the CPU resources used for each of our 8 queries across our two different implementations. For the `github` queries, we observed around 2× savings in CPU usage by switching from standard MapReduce to SYMPLE. There is very high variance in the other queries: B1 and B2 get significant speed-up from using SYMPLE, T3 gains 30% and there is no improvement from using SYMPLE for the B3 query.

In our experiments, we also observed latency benefits in using SYMPLE. For example, in the most extreme case of query B1, all input sessions are sent to only one reducer since there is no groupby parallelism for this query. In this case, the baseline MapReduce computation requires 4.5 hours. In contrast, SYMPLE completed only in 5 minutes and 30 seconds.

Efficient Hadoop by minimizing data movement Like in other scenarios, the main savings for SYMPLE over baseline MapReduce come from reduced traffic. A bar in Figure 8 gives the total bandwidth consumed between the map and reduce phase for each of the 8 queries. In the corner case of B1 with no groupby parallelism, the bandwidth savings are extreme – instead of sending all records parsed by each mapper, the SYMPLE mappers send to the reducers one single record. Similarly high savings are observed in the B2 query, where each mapper sends only one symbolic summary per geographic area. The least savings are observed for the B3 and T1 queries where the records are grouped by user or hashtag and the mappers must still send a massive number of records to the reducers.

6.5 Scalability

For almost all queries we tried, SYMPLE provided speed and bandwidth saving during data processing. The only query with no improvement in our evaluation was B3. Upon deeper inspection, however, we have found that there was no inefficiency in the symbolic computation of UDA, but the problem was the groupby function. In the B3 query, we group the records first by user, which leads to a high number of groups and leads to little opportunity for the SYMPLE symbolic summaries to save bandwidth. On the other hand, all other queries in our evaluation have a groupby function that contains a sufficiently high number of records per group and thus parallelizing the UDA computation leads to performance improvements.

7. Related work

This section discusses works most related to this paper.

Parallel execution frameworks This paper is directly motivated by the success of parallel execution frameworks [1, 2, 8–10, 12, 14, 16, 22, 26, 38, 39] for processing big data. These frameworks require the user to specify data-parallel computation but handle parallelization, locality, fault tolerance, and load distribution automatically. The goal of this paper is to increase the expressivity of these frameworks by allowing computation, such as groupby-aggregate with UDA queries that are not efficiently parallelizable. For example, Dremel [26] and Trill [10] both provide fast execution of SQL like queries wherein fields are stored in an efficient columnar store. To enable parallelism, these systems only parallelize a class of known SQL aggregators. The insights behind SYMPLE, namely breaking dependencies through symbolic summaries,

could enrich both Dremel and Trill's expressivity by letting them parallelize across arbitrary SQL aggregators. Likewise, Pig [14] and Pig Latin [28], Spark [39], FlumeJava [9] all provide a lifting operation that pushes aggregations into maps, however, *only* for associative functions (most often only for built-in functions). Symbolic parallelism increases the class of UDAs that can use the lifting optimization.

Analysis of UDAs Yu et al. [38] apply commutativity analysis to systems like MapReduce which offer grouped aggregation on large data as a primitive and show how and when a UDA can be lifted to earlier mappers [37]. They demonstrate how DryadLINQ can lift UDAs composed of multiple built-in associative and commutative functions (i.e., `sum` and `max`). Liu et al. [20] build on this work and demonstrate how to synthesize partial aggregations in SCOPE scripts from UDAs which contain finite-state machines or simple counting. While their results are promising, their analysis relies on the UDAs being commutative, a property not satisfied by the UDAs that mine temporal patterns studied here. In addition, SYMPLE can parallelize a much larger class of UDAs than these prior works, such as UDAs that make conditional checks on a counter.

On the other hand, some database systems allow users to specify a richer class of UDAs using SQL extensions for pattern matching [24, 29], however these systems do not discuss if they can extract any parallelism from such queries.

Parallelizing compilers and runtimes Compiler techniques for parallelizing sequential programs is an old but an active field of study [4, 6, 7, 30, 36]. Most of these techniques focus on transforming a program while *honoring* static program dependences to expose parallelism. Commutativity analysis [35] identifies operations that can be logically reordered despite dependences.

Reduction and parallel-prefix computations [5, 15, 19] involving associative operations can be parallelized despite dependences. Similarly, hyperobjects [13] in Cilk provide a programmatic way for specifying reduction of associative operations. In contrast, the focus on this paper is on breaking dependences that are not readily apparent as associative. One can consider the symbolic approach of SYMPLE as transforming a computation as a sequence of function compositions and using the associativity of composition [18] to perform the computation in parallel. However, automating this approach requires symbolic data types that are amenable to efficient symbolic manipulation.

The recent work on Mold [33] is closely related to SYMPLE. Mold automatically parallelizes sequential code and scales it out to a MapReduce backend. It operates by transforming loops into functional programming style `fold` operations and then applying transformation rules. While this is a promising approach, its parallelization rules apply only when the performed operations are associative.

Parallel runtimes such as Galois [31] exploit the observation that most static dependences vanish for irregular programs and dynamically schedule independent computations in parallel. The techniques discussed in this paper can be used to break these dynamic dependences as well.

Speculation An alternate approach to break dependences is to use speculation [25, 32, 34]. Speculation, unfortunately, does not scale to massively parallel back-ends as the probability of mis-speculation dramatically increases with the number of guesses causing a sequential bottleneck. Moreover, speculation with unbounded data types (such as counters) is impossible except in the presence of well-behaved distributions of their values. Finally, speculation based parallelization does not fit a MapReduce model due to the need to re-execute failed speculation attempts and iterative calls to MapReduce are expensive.

Data-parallel FSMs Recent work on parallelizing finite-state machines (FSM) [23, 27] use an approach that enumerates a dependent FSM computation for every FSM state. This enumeration can be considered as a special instance of symbolic parallelism advocated in this paper. For instance, imperative code that uses a `SymEnum` type can be used to encode a FSM. Even here, the canonical form used in `SymEnum` types do not eagerly enumerate all states and are, thus, likely to be compact and more efficient. On the other hand, the static structure of eager enumeration presents with opportunities for exploiting SIMD capabilities of modern hardware [27]. The primary focus of this paper is on exploiting large-scale compute and I/O parallelism available in data centers today.

8. Conclusion and future work

Parallelizing computation has, to date, largely been about finding independent computation. This paper introduces symbolic parallelism a new method for breaking dependences and thus exposing parallelism from seemingly sequential computation. We believe this is an interesting direction of research, where ideas from program verification, software testing, and partial evaluation, along with some help from the programmer, can mechanize the process of parallelizing computation.

To demonstrate that symbolic parallelism is useful in practice, we introduce SYMPLE, a system for automatically parallelizing MapReduce-style groupby-aggregate queries. Our experiments on Amazon's Elastic cloud and our own private 380 node Hadoop cluster can process terabytes of data in minutes. Our experiments demonstrate significant savings in datacenter resources such as CPU and bandwidth, and end-user latency. With more hardware parallelism, we believe that SYMPLE provides a platform for interactive ad-hoc querying of complex patterns in big data.

References

[1] Apache hadoop. http://hadoop.apache.org/.

[2] ALEXANDROV, A., BERGMANN, R., EWEN, S., FREYTAG, J.-C., HUESKE, F., HEISE, A., KAO, O., LEICH, M., LESER,

U., MARKL, V., NAUMANN, F., PETERS, M., RHEINLÄNDER, A., SAX, M. J., SCHELTER, S., HÖGER, M., TZOUMAS, K., AND WARNEKE, D. The stratosphere platform for big data analytics. *The VLDB Journal 23*, 6 (Dec. 2014), 939–964.

[3] Avoid groupbykey. http://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best_practices/prefer_reducebykey_over_groupbykey.html.

[4] BACON, D. F., GRAHAM, S. L., AND SHARP, O. J. Compiler transformations for high-performance computing. *ACM Comput. Surv. 26*, 4 (Dec. 1994), 345–420.

[5] BLELLOCH, G. E. Prefix sums and their applications. Tech. rep., Synthesis of Parallel Algorithms, 1990.

[6] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 1995), PPOPP '95, ACM, pp. 207–216.

[7] BOCCHINO, JR., R. L., ADVE, V. S., DIG, D., ADVE, S. V., HEUMANN, S., KOMURAVELLI, R., OVERBEY, J., SIMMONS, P., SUNG, H., AND VAKILIAN, M. A type and effect system for deterministic parallel java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (New York, NY, USA, 2009), OOPSLA '09, ACM, pp. 97–116.

[8] CHAIKEN, R., JENKINS, B., LARSON, P.-A., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. Scope: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow. 1*, 2 (Aug. 2008), 1265–1276.

[9] CHAMBERS, C., RANIWALA, A., PERRY, F., ADAMS, S., HENRY, R. R., BRADSHAW, R., AND WEIZENBAUM, N. Flumejava: Easy, efficient data-parallel pipelines. PLDI '10, ACM, pp. 363–375.

[10] CHANDRAMOULI, B., GOLDSTEIN, J., BARNETT, M., DELINE, R., FISHER, D., PLATT, J. C., TERWILLIGER, J. F., AND WERNSING, J. Trill: A high-performance incremental query processor for diverse analytics. VLDB 2015.

[11] DE MOURA, L., AND BJØRNER, N. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2008), TACAS'08/ETAPS'08, Springer-Verlag, pp. 337–340.

[12] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[13] FRIGO, M., HALPERN, P., LEISERSON, C. E., AND LEWIN-BERLIN, S. Reducers and other cilk++ hyperobjects. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2009), SPAA '09, ACM, pp. 79–90.

[14] GATES, A. F., NATKOVICH, O., CHOPRA, S., KAMATH, P., NARAYANAMURTHY, S. M., OLSTON, C., REED, B., SRINIVASAN, S., AND SRIVASTAVA, U. Building a high-level dataflow system on top of map-reduce: The pig experience. *Proc. VLDB Endow. 2*, 2 (Aug. 2009), 1414–1425.

[15] HILLIS, W. D., AND STEELE, G. L. Data parallel algorithms. In *Commun. ACM* (Dec 1986), vol. 29, pp. 1170–1183.

[16] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev. 41*, 3 (Mar. 2007), 59–72.

[17] KING, J. C. Symbolic execution and program testing. *Commun. of ACM 19*, 7 (July 1976), 385–394.

[18] KOGGE, P. M., AND STONE, H. S. A parallel algorithm for the efficient solution of a general class of recurrence equations. *Computers, IEEE Transactions on C-22*, 8 (Aug 1973), 786–793.

[19] LADNER, R. E., AND FISCHER, M. J. Parallel prefix computation. *Journal of the ACM 27*, 4 (1980), 831–838.

[20] LIU, C., ZHANG, J., ZHOU, H., MCDIRMID, S., GUO, Z., AND MOSCIBRODA, T. Automating distributed partial aggregation. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, ACM, pp. 1:1–1:12.

[21] LOMBARDI, L. A., AND RAPHAEL, B. Lisp as the language for an incremental computer. *In The Programming Language Lisp: Its Operation and Applications* (1964).

[22] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 135–146.

[23] MARGUS VEANES, TODD MYTKOWICZ, D. M., AND LIVSHITS, B. Data-parallel string-manipulating programs. *Symposium on Principles of Programming Languages (POPL)* (2015).

[24] Match clause in HP Vertica. http://my.vertica.com/docs/7.1.x/HTML/Content/Authoring/SQLReferenceManual/Statements/SELECT/MATCHClause.htm.

[25] MEHRARA, M., HAO, J., HSU, P.-C., AND MAHLKE, S. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2009), PLDI '09, ACM, pp. 166–176.

[26] MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., AND VASSILAKIS, T. Dremel: Interactive analysis of web-scale datasets. In *Proc. of the 36th Int'l Conf on Very Large Data Bases* (2010), pp. 330–339.

[27] MYTKOWICZ, T., MUSUVATHI, M., AND SCHULTE, W. Data-parallel finite-state machines. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2014), ASPLOS '14, ACM, pp. 529–542.

[28] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2008), SIGMOD '08, ACM, pp. 1099–1110.

[29] Sql for pattern matching in oracle 12c. http://docs.oracle.com/database/121/DWHSG/pattern.htm.

[30] PADUA, D. A., AND WOLFE, M. J. Advanced compiler optimizations for supercomputers. *Commun. ACM 29*, 12 (Dec. 1986), 1184–1201.

[31] PINGALI, K., NGUYEN, D., KULKARNI, M., BURTSCHER, M., HASSAAN, M. A., KALEEM, R., LEE, T.-H., LENHARTH, A., MANEVICH, R., MÉNDEZ-LOJO, M., PROUNTZOS, D., AND SUI, X. The tao of parallelism in algorithms. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 12–25.

[32] PRABHU, P., RAMALINGAM, G., AND VASWANI, K. Safe programmable speculative parallelism. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2010), PLDI '10, ACM, pp. 50–61.

[33] RADOI, C., FINK, S. J., RABBAH, R., AND SRIDHARAN, M. Translating imperative code to mapreduce. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (New York, NY, USA, 2014), OOPSLA '14, ACM, pp. 909–927.

[34] RAMAN, A., KIM, H., MASON, T. R., JABLIN, T. B., AND AUGUST, D. I. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2010), ASPLOS XV, ACM, pp. 65–76.

[35] RINARD, M. C., AND DINIZ, P. C. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Trans. Program. Lang. Syst. 19*, 6 (Nov. 1997), 942–991.

[36] RINARD, M. C., AND LAM, M. S. The design, implementation, and evaluation of jade. *ACM Trans. Program. Lang. Syst. 20*, 3 (May 1998), 483–545.

[37] YU, Y., GUNDA, P. K., AND ISARD, M. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *ACM Symposium on Operating Systems Principles (SOSP)* (October 2009), pp. 247–260.

[38] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, U., GUNDA, P. K., AND CURREY, J. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. OSDI'08, USENIX Association, pp. 1–14.

[39] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (Berkeley, CA, USA, 2010), HotCloud'10, USENIX Association, pp. 10–10.

[40] ZHANG, J., ZHOU, H., CHEN, R., FAN, X., GUO, Z., LIN, H., LI, J. Y., LIN, W., ZHOU, J., AND ZHOU, L. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 22–22.