Rhea: automatic filtering for unstructured cloud storage

Christos Gkantsidis, Dimitrios Vytiniotis, Orion Hodson
Dushyanth Narayanan, Florin Dinu,[*] *Antony Rowstron*
Microsoft Research, Cambridge, UK

Abstract

Unstructured storage and data processing using platforms such as MapReduce are increasingly popular for their simplicity, scalability, and flexibility. Using elastic cloud storage and computation makes them even more attractive. However cloud providers such as Amazon and Windows Azure separate their storage and compute resources even within the same data center. Transferring data from storage to compute thus uses core data center network bandwidth, which is scarce and oversubscribed. As the data is unstructured, the infrastructure cannot automatically apply selection, projection, or other filtering predicates at the storage layer. The problem is even worse if customers want to use compute resources on one provider but use data stored with other provider(s). The bottleneck is now the WAN link which impacts performance but also incurs egress bandwidth charges.

This paper presents Rhea, a system to automatically generate and run storage-side data filters for unstructured and semi-structured data. It uses static analysis of application code to generate filters that are safe, stateless, side effect free, best effort, and transparent to both storage and compute layers. Filters never remove data that is used by the computation. Our evaluation shows that Rhea filters achieve a reduction in data transfer of 2x–20,000x, which reduces job run times by up to 5x and dollar costs for cross-cloud computations by up to 13x.

1 Introduction

The last decade has seen a huge increase in the use of "noSQL" approaches to data analytics. Whereas in the past the default data store was a relational one (e.g. SQL), today it is possible and often desirable to store the data as unstructured files (e.g. text-based logs) and to process them using general-purpose languages (Java, C#). The combination of unstructured storage and general-purpose programming languages increases flexibility: different programs can interpret the same data in different ways, and changes in format can be handled by changing the code rather than restructuring the data.

This flexibility comes at a cost. The structure of the data is now *implicit* in the program code. Most analytics jobs use a subset of the input data, i.e. only some of the data items are relevant and only some of the fields within those are relevant. Since these selection and projection operations are embedded in the application code, they cannot be applied by the storage layer; rather all the data must be read into the application code.

This is not an issue for dedicated data processing infrastructures where a single cluster provides both storage and computation, and a framework such as MapReduce, Hadoop, or Dryad co-locates computation with data. However it is a problem when running such frameworks in an elastic cloud. Cloud providers such as Amazon and Windows Azure provide both scalable unstructured storage and elastic compute resources but these are physically disjoint. There are many good reasons for this including security, performance isolation, and the need to independently scale and provision the storage and elastic compute infrastructures. Both Amazon's S3 [1] and Windows Azure Storage [4, 39] follow this model of physically separate compute and storage servers within the same data center. This means that bytes transferred from storage to compute use core data center network bandwidth, which is often scarce and oversubscribed [14] (see also Section 4.1.1).

Our aim is to retain the flexibility of unstructured storage and the elasticity of cloud storage and computation, yet reduce the bandwidth costs of transferring redundant or irrelevant data from storage to computation. Specifically, we wish to transparently run applications written for frameworks such as Hadoop in the cloud, but extract the implicit structure and use it to reduce the amount of data read over the data center network. Reducing bandwidth will improve provider utilization, by allowing more jobs to be run on the same servers, and improve performance for customers, as their jobs will run faster.

[*]Work done while on internship from Rice University

Our approach is to use static analysis on application code to automatically generate application-specific *filters* that remove data that is irrelevant to the computation. The generated filters are then run (typically, but not necessarily) on storage servers in order to reduce bandwidth. Filters need to be safe and transparent to the application code. They need to be conservative, i.e., the output of the computation must be the same whether using filters or not, and hence only data that provably cannot affect the computation can be suppressed. Since filters are using spare computational resources on the storage servers, they also need to be best-effort, i.e. they can be disabled at any time without affecting the application.

Our *Rhea* system automatically generates and executes storage-side filters for unstructured text data. Rhea extracts both *row filters* which select out irrelevant rows (lines) in the input, as well as *column filters* which project out irrelevant columns (substrings) in the surviving rows.[1] Both row and column filters are safe, transparent, conservative, and best-effort. Rhea analyzes the Java bytecode of programs written for Hadoop MapReduce, producing job-specific executable filters.

Section 2 makes the case for implicit, storage-side filtering and describes 9 analytic jobs that we use to motivate and evaluate Rhea. Section 3 describes the design and implementation of Rhea and its filter generation algorithms. Section 4 shows that storage-to-compute bandwidth is scarce in real cloud platforms; that Rhea filters achieve substantial reduction in the storage-to-compute data transfer and that this leads to performance improvements in a cloud environment. Rhea reduces storage-to-compute traffic by a factor of 2–20,000, job run times by a factor of up to 5, and dollar costs for cross-cloud computations by a factor of up to 13. Section 5 discusses related work, and Section 6 concludes.

2 Background and Motivation

In this section we first describe the design rationale for Rhea: the network bottlenecks that motivate storage-side filtering, and the case for automatically generated (implicit) filters. We finally describe the example jobs that we use to evaluate Rhea.

2.1 Storage-side filtering

The case for storage-side filtering is based on two observations. First, compute cycles on storage servers are cheap relative to core network bandwidth. Of course, since this is not an explicitly provisioned resource, use of such cycles should be opportunistic and best-effort. Second, storage-to-compute bandwidth is a scarce resource that can be a performance bottleneck. Our measurements of read bandwidth for Amazon EC2/S3 and Windows Azure confirm this (Section 4.1.1) and are consistent with earlier measurements [11, 12].

[1] For convenience we use the term "row" to refer to the input unit of a Map process, and "column" to refer to the output of the tokenization performed on the row input according to some user-specified logic.

If data must be transferred across data centers or availability zones, then this will not only use WAN bandwidth and impact performance, but also incur egress bandwidth charges for the user. This can happen if data stored in different geographical locations need to be combined, e.g., web logs from East and West Coast servers. Some jobs may need to combine public and private data, e.g. a public data set stored in Amazon S3 [31] with a private one stored on-premises, or a data set stored in Amazon S3 with one stored in Windows Azure Storage.

Our aim is to reduce network load, job run times, and egress bandwidth charges through filtering for many different scenarios. When the storage is in the cloud, the cloud provider (e.g. Amazon S3) could natively support execution of Rhea filters on or near the storage servers. In the case where the computation uses a compute cluster provided by the same provider (e.g. Amazon EC2 in the case of Amazon S3), the provider could even extract and deploy filters transparently to the customer. For on-premises ("private cloud") storage, filters could be deployed by the customer on the storage servers or near them, e.g. on the same rack. If the provider does not support filtering at the storage servers, filtering can still be used to reduce WAN data transfers by running the filters in a compute instance located in the same data center as the storage. In the latter case our evaluation shows that the savings in egress bandwidth charges outweigh the dollar cost of a filtering VM instance. Additionally, the isolation properties of Rhea filters make it possible for multiple users to safely share a single filtering VM and thus reduce this cost.

2.2 Implicit filtering

Rhea creates filters implicitly and transparently using static analysis of the programs. An alternative would be to have the programmer do this explicitly. For example a language like SQL makes the filtering predicates and columns accessed within each row explicit. E.g., the "WHERE" clause in a SQL statement identifies the filtering predicate and the "SELECT" statement for column selectivity. Several storage systems support explicit column selectivity for MapReduce jobs, e.g. "slice predicates" in Cassandra [3], "input format classes" in Zebra [41], explicit filters in Pig/Latin [13], and RC-files in Hive [34]. In such situations input data pre-filtering can be performed using standard techniques from database query optimization.

While extremely useful for this kind of query optimization and reasoning, explicit approaches often provide less flexibility, as the application is tied to a specific interface to the storage (SQL, Cassandra, etc). They are also less well-suited for free-format or semi-structured text files, which have to be parsed in an application-specific manner. This flexibility is one of the reasons that platforms such as SCOPE [5] allow a mixture of SQL-like and actual C# code. Eventually all code (including the SQL part) is compiled down to .NET and executed.

Our aim in Rhea is to handle the general case where programmers can embed application-specific column parsing logic or arbitrary code in the mapper, without imposing any additional programmer burden such as hand-annotating the code with filtering predicates. Instead, Rhea infers filters automatically using static analysis of the application byte code. Since Rhea only examines the application code, it is applicable even when *the format of the data is not known a-priori*, or *the data does not strictly conform to an input format* (for instance tabular input data with occasionally occurring comment lines starting with a special character).

2.3 Example analytics jobs

Our static analysis handles arbitrary Java byte code: we have used over 160 Hadoop mappers from various Hadoop libraries and other public and private sources to test our tool and validate the generated filters (Section 3.4). Of these, we present nine jobs for which data were also available and use them to drive our evaluation (Section 4). Here we describe these nine jobs. Note that we do not include commonly-used benchmarks such as Sort and WordCount, which are used to stress-test MapReduce infrastructures. Neither of these has any selectivity, i.e., the mapper examines all the input data, and thus Rhea would not generate any filters for them. However, we do not believe such benchmarks are representative of real-world jobs, which often do have selectivity.

GeoLocation This publicly available Hadoop example [24] groups Wikipedia articles by their geographical location. The input data is based on a publicly available data set [23]. The input format is text, with each line corresponding to a row and tab characters separating columns within the row. Each row contains a type column which determines how the rest of the row is interpreted; the example job only considers one of the two row types, and hence rows of the other type can be safely suppressed from the input.

Event log processing The next two jobs are based on processing event logs from a large compute/storage platform consisting of tens of thousands of servers. Users issue tasks to the system, which spawn processes on multiple servers. Resource usage information measured on all these servers is written to two event logs: a process log with one row per executed process, and an activity log that records fine-grained resource consumption information. We use two typical jobs that process this data. The first, *FindUserUsage*, identifies the top-k users by total process execution time. The second, *ComputeIoVolumes*, is a join: it filters out failed tasks by reading the process log and then computes storage I/O statistics for the successful tasks from the activity log.

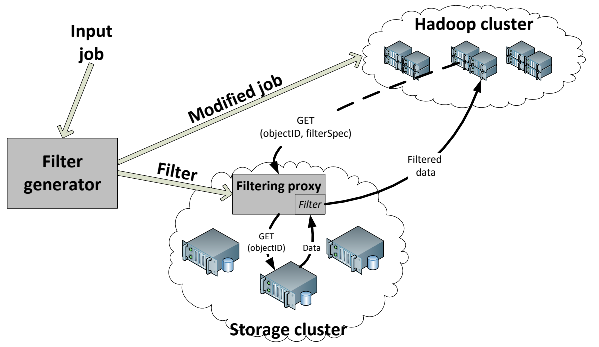

Figure 1: System architecture

IT log reporting The next job is based on enterprise IT logs across thousands of shared infrastructure machines on an enterprise network. The sample job (*IT Reporting*) queries these logs to find the aggregate CPU usage for a specific machine, grouped by the type of user generating the CPU load.

Web logs and ranking The last five jobs are from a benchmark developed by Pavlo et al. [30] to compare unstructured (MapReduce) and structured (DBMS) approaches to data analysis. The jobs all use synthetically generated data sets consisting of a set of HTML documents that link to each other, a Rankings table that maps each unique URL to its computed PageRank, and a UserVisits table that logs user visits to each unique URL as well as context information such as time-stamp, country code, ad revenue, and search context.

The first two jobs are variants of a *SelectionTask* (find all URLs with page rank higher than X). The amount of input data that is relevant to this task depends on the threshold X. Thus we use two variants with thresholds $X_{1\%}$ and $X_{10\%}$, where approximately 1% of the URLs have page rank higher than $X_{1\%}$, and 10% of the URLs have page rank higher than $X_{10\%}$. The next two jobs are based on an *AggregationTask*. They find total revenue grouped by unique source IP, and total revenue grouped by source network, respectively. Finally, the *JoinTask* finds the average PageRank of the pages visited by the source IP that generated the most revenue within a particular date range.

3 Design and Implementation

The current Rhea prototype is designed for Hadoop MapReduce jobs. It generates executable Java filters from the mapper class(es) for each job. It is important to note that although Rhea filters are executable, running a filter is different from running arbitrary application code, for example running the entire mapper task on the storage server. Filters are guaranteed to be safe and side-effect free and thus can be run with minimal sandboxing, with multiple filters from different jobs or customers co-existing in the same address space. They are transparent and best-effort, and hence can be disabled at any point to save resources without affecting the application. They are stateless and do not consume large amounts of memory to hold output, as is done by many mappers. Finally, they are guaranteed never to output more data than input. This is not true of mappers where the output data can be larger than the input data [6].

Figure 1 shows the architecture of Rhea, which consists of two components: a *filter generator* and a *filtering proxy*. The filter generator creates the filters and uploads them to the filtering proxy, and also adds a transparent, application-independent, client-side shim to the user's Hadoop job to create a Rhea-aware version of the job. The Rhea-aware version of the job intercepts cloud storage requests, and redirects them to the filtering proxy. The redirected requests include a description of the filter to be instantiated and a serialized cloud storage REST request to access the job's data. The serialized request is signed with the user's cloud storage provider credentials when it is generated on the Hadoop nodes so the filtering proxy holds no confidential user state. When the filtering proxy receives the redirected request, it instantiates the required filter, issues the signed storage request, and returns filtered data to the caller. Thus Rhea filtering is transparent to the user code, to the elastic compute infrastructure, and to the storage layer, and requires no sensitive user state. The proxy works with Amazon's S3 Storage and Windows Azure Storage, and also has a local file system back end for development and test use.

The filter generator takes the Java byte code of a Hadoop job, and generates a *row filter* and a *column filter* for each mapper class found in the program. These are encoded as methods on an extension of the corresponding mapper class. The extended classes are shipped to the filtering proxy as Java jar files and dynamically loaded into its address space. The filter generator, and the static analysis underlying it, are implemented using SAWJA [18], a tool which provides a high-level stackless representation of Java byte code. In the rest of this section we describe the static analysis used for row and column filter generation.

3.1 Row Filters

A row filter in Rhea is a method that takes a single record as input and returns `false` if that record does not affect the result of the MapReduce computation, and `true` otherwise. It can have false positives, i.e., return `true` for records that do not affect the output, but it can not have false negatives. The byte code of the filter is generated from that of the mapper. Intuitively, it is a stripped-down or "skeleton" version of the mapper, retaining only those instructions and execution paths that determine whether or not a given invocation will produce an output. Instructions that are used to compute the *value* of the output but do not affect the control flow are not present in the filter. As such, the row filter is *completely independent of the format of the input data* and only depends on the predicates that the mapper is using on the input.

Listing 1 shows a typical example: the mapper for the GeoLocation job (Section 2.3). It tokenizes the input `value` (line 7), extracts the first three tokens, (line 9–11), and then checks if the second token equals the static field `GEO_RSS_URI` (line 13). If it does, more processing follows (line 14–26) and some value is output on `outputCollector`; otherwise, no output is generated.

```
 1 ... // class and field declarations
 2 public void map(LongWritable key, Text value,
 3  OutputCollector<Text, Text> outputCollector,
 4  Reporter reporter) throws IOException {
 5
 6  String dataRow = value.toString();
 7  StringTokenizer dataTokenizer =
 8      new StringTokenizer(dataRow, "\t");
 9  String artName = dataTokenizer.nextToken();
10  String pointTyp = dataTokenizer.nextToken();
11  String geoPoint = dataTokenizer.nextToken();
12
13  if (GEO_RSS_URI.equals(pointTyp)) {
14    StringTokenizer st =
15       new StringTokenizer(geoPoint, "␣");
16    String strLat = st.nextToken();
17    String strLong = st.nextToken();
18    double lat = Double.parseDouble(strLat);
19    double lang = Double.parseDouble(strLong);
20    long roundedLat = Math.round(lat);
21    long roundedLong = Math.round(lang);
22    String locationKey = ...
23    String locationName = ...
24    locationName = ...
25    geoLocationKey.set(locationKey);
26    geoLocationName.set(locationName);
27    outputCollector.collect(geoLocationKey,
28              geoLocationName);
29  } }
```

Listing 1: GeoLocation map job

Listing 2 shows the filter generated by Rhea for this mapper. It also tokenizes the input (line 8) and performs the comparison on the second token (line 12) (`bcvar8` here corresponds to `pointTyp` in `map`). This test determines whether `map` would have produced output, and hence `filter` returns the corresponding Boolean value.

Comparison of `map` and `filter` reveals two interesting details. First, while `map` extracted three tokens from the input, `filter` only extracted two. The third token does not determine whether or not output is produced, although it does affect the value of the output. The static

```
1 public boolean filter (LongWritable bcvar1,
2    Text bcvar2,
3    OutputCollector bcvar3,
 Reporter bcvar4) {
4
5  boolean cond = false;
6  String bcvar5 = bcvar2.toString();
7  String irvar0 = "\t";
8  StringTokenizer bcvar6 =
9    new StringTokenizer(bcvar5,irvar0);
10 String bcvar7 = bcvar6.nextToken();
11 String bcvar8 = bcvar6.nextToken();
12 boolean irvar0_1=
13     GEO_RSS_URI.equals(bcvar8);
14
15 cond = ((irvar0_1?1:0)  !=  0);
16 if (!cond) return false;
17 return true;
18 }
```

Listing 2: Row filter generated for GeoLocation

analysis detects this and omits the extraction of the third token. Second, `map` does substantial processing (line 14–26) before producing the output. All these instructions are omitted from the filter: they affect the output value but not the output condition.

Row filter generation uses a variant of dependency analysis commonly found in program slicing [17, 26, 36]. Our analysis is based on the following steps:

1. It first identifies "output labels", i.e. program points at which the mapper produces output, such as calls to the Hadoop API including `OutputCollector.collect` (line 28 of Listing 1). The generated filter must return `true` for any input that causes the mapper to reach such an output label (line 17 of Listing 2). This basic definition of output label is later extended to handle the use of state in the mapper (Section 3.1.1).

2. The next step is to collect all control flow paths (including loops) of the mapper that reach an output label. Listing 1 contains a single control path that reaches an output label through line 13 of Listing 1.

3. Next, Rhea performs a label-flow analysis (as a standard forward analysis [29]) to compute a "flow map": for each program instruction, and for each variable referenced in that instruction, it computes all other labels that could affect the value of that variable.

4. For every path identified in Step 2, we keep only the instructions that, according to the flow map from Step 3, can affect any control flow decisions (line 6–13 of Listing 1, which correspond to line 6–16 of Listing 2). The result is a new set of paths which contains potentially fewer instructions per path – only the *necessary* ones for control flow to reach the path's output instruction.

5. Finally, we generate code for the *disjunction* of the paths computed in Step 4, emitting `return true` statements after the last conditional along each path. Technically, prior to this step we perform several optimizations, for instance we merge paths when both the `True` and the `False` case of a conditional statement can lead to output. We also never emit code for a loop if the continuation of a loop may reach an output instruction: in this case we simply `return true` when we reach the loop header, in order to avoid performing a potentially expensive computation if there is possibility of output after the loop.

3.1.1 Stateful mappers

This basic approach described above guarantees that the filter returns `true` for any input row for which the original mapper would produce output, but neglects the fact that `map` will be invoked on *multiple* rows, where each invocation may affect some *state* in the mapper that could affect the control flow in a subsequent invocation.

In theory this situation should not happen – in an ideal world, mappers should be stateless, to allow the MapReduce infrastructure to partition and re-order the mapper inputs without changing the result of the computation. However, in practice programmers do make use of state (such as frequency counters and temporary data structures) for efficiency or monitoring reasons, and typically via fields of the mapper class.

Consider for instance a mapper which increments a counter for each input row and produces output only on every n-th row. If we generate a filter that returns `true` for every n-th row and run the mapper on the filtered data set we will alarmingly have produced different output!

A simplistic solution to the problem would be to emit (trivial) filters that always return `true` for any `map` which depends on or modifies shared state. In practice, however, a surprising number of mappers access state and we would still like to generate non-trivial filters for these. Rhea does this by extending the definition of "output label" to include not only calls to the Hadoop API output methods but also instructions that could potentially affect shared state, such as method calls that involve mutable fields, field assignments and static methods, and also accesses of fields that are set in some part of the map method, and any methods of classes that could have some global observable effect, such as `java.lang.System` or Hadoop API methods. This ensures that the filter *approximates* the paths that could generate output in the mapper with a set of paths that (i) do not in any way depend on modifiable cross-invocation state; and (ii) do not contain any instructions that could themselves affect such shared state.

This simple approach is conservative but sound when there is use of state. More interestingly, this approach works well (i.e. generates non-trivial filters) with *common* uses of state. For example, in Listing 1, line 25 references the global field `geoLocationKey`. However, this happens in the same control flow block where the actual

```
public String select (LongWritable bcvar1,
  Text bcvar2,
  OutputCollector cvar3, Reporter bcvar4) {
 String bcvar5 = bcvar2.toString();
 String irvar0 = "\t";
 StringTokenizer bcvar6
    = StringTokenizer(bcvar5,irvar0);
 int i = 0;
 String filler = computeFiller(irvar0);
 StringBuilder out = new StringBuilder();
 String curr, aux;
 while (bcvar6.hasMoreTokens()) {
   curr = bcvar6.nextToken();
   if (i == 2 || i == 1 || i == 0) {
     aux = curr;
   } else {
     aux = filler;
   };
   if (bcvar6.hasMoreTokens()) {
     out.append(aux).append(irvar0);
   }
   else {
     out.append(aux);
   }
   i++;
 }
 return out.toString(); }
```

Listing 3: Column selector generated for GeoLocation

output instruction is located (line 28). Consequently, the generated filter is as precise as it could possibly be.

3.2 Column selection

So far we have described row filtering, where each input record is either suppressed entirely or passed unmodified to the computation. However, it is also valuable to suppress individual *columns* within rows. For example, in a top-K query, all rows must be examined to generate the output, but only a subset of the columns are relevant.

The Rhea filter generator analyzes the mapper function to produce a column selector method that transforms the input line into an output line with irrelevant column data suppressed. Column filtering may be combined with row filtering by using row filtering first and column selection on the remaining rows.

The static analysis for column selection is quite different from that used for row filtering. In Hadoop, mappers split each row (record) into columns (fields) in an application-specific manner. This is very flexible: it allows for different rows in the same file to have different numbers of columns. Mappers can also split the row into columns in different ways, e.g., using string splitting, or a tokenization library, or a regular expression matcher. This flexibility makes the problem of correctly removing irrelevant substrings challenging. Our approach is to detect and exploit common patterns of tokenization that we have encountered in many mappers. Our implementation supports tokenization based on Java's `StringTokenizer`

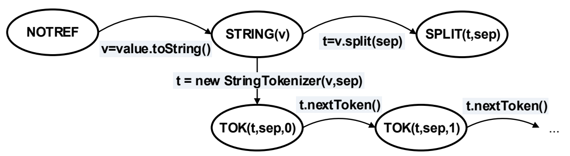

Figure 2: Transition system for column selector analysis

class and the `String.split()` API, but is easily extensible to other APIs.

For the GeoLocation map function in Listing 1, Rhea generates the column selector shown in Listing 3. The mapper only examines the first three tokens of the input (line 9–11 of Listing 1). The column selector captures this by retaining only the first three tokens. The output string is reassembled from the tokens after replacing all irrelevant tokens with a filler value, which is dynamically computed based on the separator used for tokenization.

Column filters always retain the token separators to ensure that the modified data is correctly parsed by the mapper. Dynamically computing the filler value allows us to deal with complex tokenization, e.g., using regular expressions. As a simple example, consider a comma-separated input line `"eve,usa,25"`. If the mapper splits the string at each comma, this can be transformed to `"eve,,25"`. However, if using a regular expression where multiple consecutive commas count as a single separator, `"eve,,25"` would be incorrect but `"eve,?,25"` would be correct. The `computeFiller` function correctly generates the filler according to the type of separator being used at run time.

The analysis assigns to each program point (label) in the mapper a state from a finite state machine which captures the current tokenization of the input. Figure 2 shows a simplified state machine that captures the use of the `StringTokenizer` class for tokenization. Essentially the input string can be in its initial state (`NOTREF`); it can be converted to a `String` (`STRING`); or this string can either have been split using `String.Split` (`SPLIT`) or converted to a `StringTokenizer` currently pointing to the *n*th token (`TOK(_,_,n)`).

The actual state machine used is slightly more complex. There is also an error state (not shown) that captures unexpected state transitions. The `TOK` state can also capture a non-deterministic state of the `StringTokenizer`: i.e., we can represent that at least *n* tokens have been extracted (but the exact upper bound is not known). The set of states is extended to form a lattice, which SAWJA's static analysis framework can use to map every program point to one of the states.

Assuming that no error states have been reached, we identify all program points that extract an input token that is then used elsewhere in the mapper. The tokenizer state at each of these points tells us which position(s) in the input string this token could correspond to. The union of all these positions is the set of relevant token positions, i.e. columns. The filter generator then emits code for the column selector that tokenizes the input string, retains relevant columns, and replaces the rest with the filler.

Since our typical use cases involve unstructured data represented as `Text` values, we have focused on common string tokenization input patterns. Other use models do exist – for instance substring range selection – for which a different static analysis involving numerical constraints might be required [28]. Though entirely possible to design such analysis, we have focused on a few commonly used models. Our static analysis is able to detect when our parsing model is not directly applicable to the mapper implementation, in which case we conservatively accept the whole input and we are only in a position to get optimizations from row filtering.

Unlike row filtering, the presence of state in the mappers cannot compromise the soundness of the generated column filters, since column filters that conservatively retain *all dereferenced tokens* of the input, irrespectively of whether these tokens will be used in the control flow or to produce an output value, and whether different control flow paths assume different numbers of columns present in the input row.

3.3 Filter properties

Rhea's row and filter columns guarantee correctness in the sense that the output of the mapper is always the same for both filtered and unfiltered inputs. In addition we guarantee the following properties:

Filters are fully transparent Either row or column filtering can be done on a row-by-row basis, and filtered and unfiltered data can be interleaved arbitrarily. This allows filtering to be best-effort, i.e. it can be enabled/disabled on a fine-grained basis depending on available resources. It also allows filters to be chained, i.e. inserted anywhere in the data flow regardless of existing filters without compromising correctness.

Isolation and safety Filters cannot affect other code running in the same address space or the external environment. The generated filter code never includes I/O calls, system calls, dynamic class loads, or library invocations that affect global state outside the class containing the filter method.

Termination guarantees Column filters are guaranteed to terminate as they are produced only from a short column usage specification that we extract from the mapper using static analysis. Row filters may execute an arbitrary number of instructions and contain loops. Currently we dynamically disable row filters that consume excessive CPU resources. We could also statically guarantee termination by considering loops to be "output labels" that cause an early return of `true`, or use techniques to prove termination even in the presence of loops [8, 15].

As explained previously, our guarantees for column filters come with no assumptions whatsoever. Our row filter guarantees are with respect to our "prescribed" notion of state (system calls, mutable fields of the class, static fields, dynamic class loading). A mathematical proof of correctness would have to include a formalization of the semantics of JVM, and the MapReduce programming model. In this work we focus on the design and evaluation of our proposal and so we leave the formal verification as future work.

3.4 Applicability of static analysis

We collected the bytecode and source of 160 mappers from a variety of projects available on the internet to evaluate the applicability of our analysis. We ran these mappers through our tools and manually inspected the outputs to verify correctness. Approximately 50% of the mappers resulted in non-trivial row filters; the rest are always-true, due to the nature of the job or the use of state early on in the control flow. A common case is the use of state to measure and report the progress of input processing. In this case, we have to conservatively accept all input, even though reporting does not affect the output of the job. 26% of the mappers were amenable to our column tokenization models (the rest used the whole input, which often arises in libraries that operate on preprocessed data, or use a different parsing mechanism).

In our experiments the tasks of (i) identifying the mappers in the job, (ii) performing the static analysis on the mappers, (iii) generating filter source code, (iv) compiling the filter, and (v) generating the Rhea-aware Hadoop job, take a worst case time 4.8 seconds for a single mapper job on an Intel Xeon X5650 workstation. The static analysis part takes no more than 3 seconds.

In the next section we present the benefits of filtering for several jobs for which we had input data and were able to run more extensive experiments.

4 Evaluation

We ran two groups of experiments to evaluate the performance of Rhea. One group of experiments evaluates the performance within a single cloud data center, and the other aims to evaluate Rhea when using data stored in a remote data center.

4.1 Experimental setup

We ran the experiments, unless otherwise stated, on Windows Azure. A challenge for running the experiments within the data center is that we could not modify the Windows Azure storage to support the local execution of the filters we generated. To overcome this, for the experiments run in the single cloud scenarios, we used the filters generated to pre-filter the input data and then stored it in Windows Azure storage. The bandwidth between the storage and the compute is the bottleneck resource, and this allows us to demonstrate the benefits of using Rhea. We micro-benchmark the filtering engine to demonstrate that it can sustain this throughput.

We use two metrics when measuring the performance of Rhea, selectivity and run time. Selectivity is the primary metric and captures how effective the Rhea filters are at reducing the data that needs to be transferred between the storage and compute. This is the primary metric of interest to a cloud provider, as this reduces the amount of data that is transferred across the core network between the storage clusters and compute. The second metric is run time, which is defined as the time to initialize and execute a Hadoop job on an existing cluster of compute VMs. Reducing run time is important in itself, but also because cloud computing VMs are charged per unit time, even if the VMs spend most of their time blocked on the network. Hence any reduction in execution time is important to the customer. The jobs that we run operated on a maximum input data set size of 100GB and all jobs ran in 15 minutes or less. Therefore, with per-hour VM charging, Rhea would provide little financial benefit when running a single job. However, if cloud providers move to finer grained pricing models or even per-job pricing models this will also have benefit; alternatively the customer could run more jobs within the same number of VM-hours and hence achieve cost savings per job. Unless otherwise stated, all graphs in this section show means of five identical runs, with error bars showing standard deviations.

To enable us to configure the experiments we measured the available storage-to-compute bandwidth for Windows Azure compute and scalable storage infrastructure, and for Amazon's infrastructure.

4.1.1 Storage-to-compute LAN bandwidth

The first set of experiments measured the storage-to-compute bandwidth for the Windows Azure data center by running a Hadoop MapReduce job with an empty mapper and no reducer. Running this shows the maximum rate at which input data can be ingested when there are no computational overheads at all. Each experiment read at least 60 GB to amortize any Hadoop start-up overheads. We also ran the experiment on Amazon's cloud infrastructure to see if there were significant differences in the storage-to-compute bandwidth across providers.

In the experiment we varied the number of instances used, between 4 and 16. We ran with extra large instances on both Amazon and Windows Azure, but also compared the performance with using small instances on Windows Azure. We found that bandwidth increases with the number of mappers per instance up to 16 mappers per instance, so we used 16 mappers per instance.

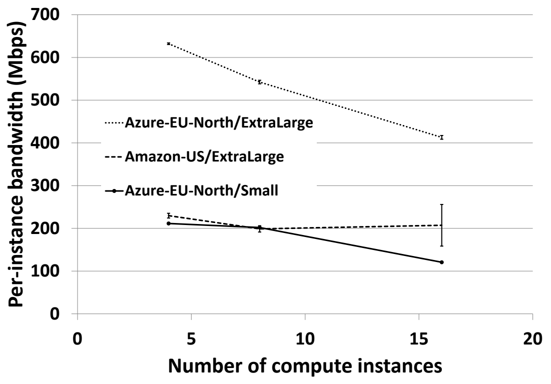

Figure 3: Storage-to-compute bandwidth in Windows Azure and Amazon cloud infrastructures. Labels show the provider, geographical location, and instance size used in each experiment.

Figure 3 shows the measured storage-to-compute transfer rate *per* compute instance. For Amazon the maximum per-instance ingress bandwidth is 230 Mbps, and the total is almost constant independent of the number of instances. For Windows Azure we observe that the peak ingress bandwidth is 631 Mbps when using 4 extra large instances. Contrary to the Amazon results, as the number of instances is increased the observed throughput per instance drops. Further, we observe that the small instance size on Windows Azure has significantly less bandwidth compared to the extra large instance.

Even in this best case (extra-large instances, no computational load, and a tuned number of mappers), the rate at which each compute instance can read data from storage is well below a single network adapter's bandwidth of 1 Gbps. More importantly it is lower than the rate at which most Hadoop computations can process data, making it the bottleneck. Hence, we would expect that reducing the amount of data transferred from storage to compute will not only provide network benefits but also, as we will show, run time performance improvements.

Based on these experiments, we run the experiments using 4 extra large compute instances on Azure-EU-North data center, each configured to run with 16 mappers per instance. This maximizes the bandwidth to the job, which is the worst case for Rhea. As the bandwidth becomes more constrained, through running on the Amazon infrastructure, by using smaller instances, or a larger number of instances the benefits of filtering will increase.

4.1.2 Job configuration

In all the experiments we use the 9 Hadoop jobs described in Section 2.3. Figure 4 shows the baseline results for input data size for each of the jobs and the run time when run in the Azure-EU-North data center with 4

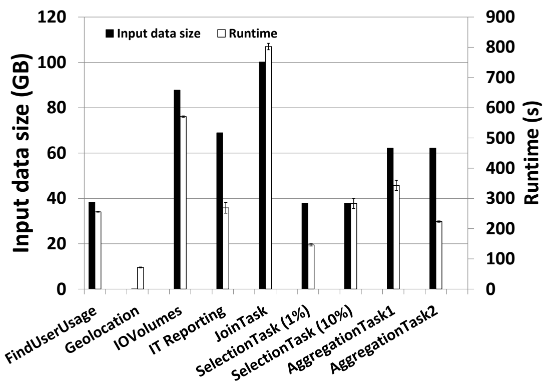

Figure 4: Input data sizes and job run times for the 9 example jobs when running on 4 extra large instances on Windows Azure without using Rhea.

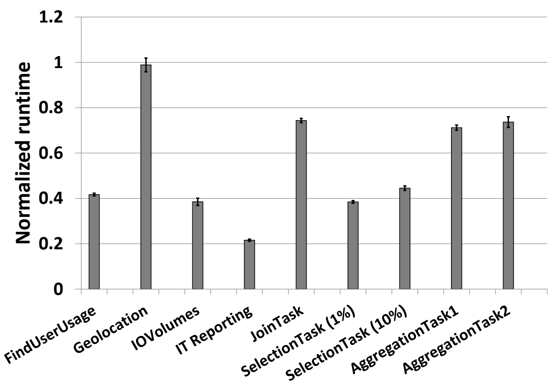

Figure 6: Job run time when using the Rhea filters normalized to the baseline execution time for the 9 example jobs when running on 4 extra large instances on Windows Azure.

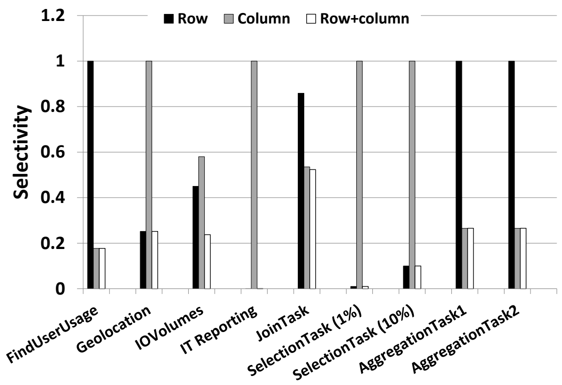

Figure 5: Selectivity for the row, column and combined filters for the 9 example jobs.

extra large compute instances without using Rhea. The input data size for the jobs varies from 90 MB–100 GB and the run times from 1–15 min. All but one job, GeoLocation, have an input data size of over 35 GB. To compare Rhea's effectiveness across this range of job sizes and run times, we show Rhea's data transfer sizes (Figure 5) and run times (Figure 6) normalized with respect to the results shown in Figure 4.

4.2 In cloud

The first set of experiments are run in a single cloud scenario: the data and compute are co-located in the same data center. The first results explore the selectivity of the filters produced by Rhea.

Selectivity For each of the nine jobs we take the input data and apply the row filter, the column filter, and the combined row and column filters and measure the selectivity. Selectivity is defined as the ratio of filtered data size to unfiltered data size; e.g. a selectivity of 1 means that no data reduction happened. Figure 5 shows the selectivity for row filters, for column filters, and the overall selectivity of row and column filters combined.

Figure 5 shows several interesting properties of Rhea. First, when using both row and column filtering across all jobs we observe a substantial reduction in the amount of input data transferred from the storage to the compute. In the worst case only 50% of the data was transferred. The majority of filters transferred only 25% of the data, and the most selective one only 0.005%, representing a reduction of 20,000 times the original data size. Therefore, in general the approach provides significant gains.

We also see that for five jobs the column selectivity is 1.0. In these cases no column filter was generated by Rhea. In three cases, the row selectivity is 1.0. In these cases, row filters were generated but did not suppress any rows. On examination, we found that the filters were essentially a check for a validly formed line of input (a common check in many mappers). Since our test inputs happened to consist only of valid lines, none of the lines were suppressed at run time. Note that a filter with poor selectivity can easily be disabled at run time without modifying or even restarting the computation.

Runtime Next we look at the impact of filtering on the execution time of jobs running in Windows Azure.

Figure 6 shows the run time for the nine jobs when using the Rhea filters normalized to the time taken to the baseline. For half the jobs we observe a speed up of over a factor of two. For four of the remaining jobs we observe that the time taken is 75% or lower compared to the baseline. The outlier in the GeoLocation example which, despite the data selectivity being high, has an identical run time. This is because the data set is small and the run time setup overheads dominate.

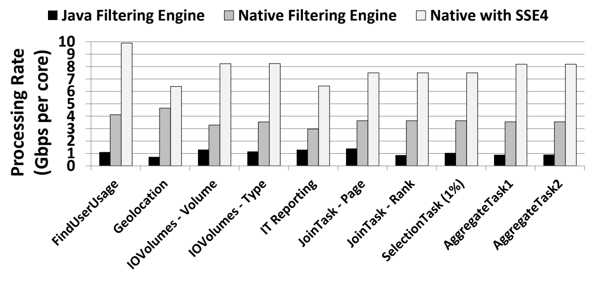

Figure 7: Input data rates achieved by filtering for row and column filters in Java and declarative column filters alone in two native filtering engines. Observe that two of the jobs contain two mappers each, for which we measure filtering performance independently.

Filtering engine These experiments run with pre-filtered data as we can not modify the Windows Azure storage layer. Separately, we micro-benchmarked the throughput of the filtering engine. Our goal is to understand if filtering can become a bottleneck for the job and hence slow down job run times. Although filtering still reduces overall network bandwidth usage, we would disable such filters to prevent individual jobs from slowing down.

Consider a modest storage server with 2 cores and a 1 Gbps network adapter. Assuming a server transmitting at full line rate, the filters should process data at an input rate of 1 Gbps or higher, to guarantee that filtering will not degrade job performance. In practice, with a large number of compute and storage servers, core network bandwidth is the bottleneck and the server is unlikely to achieve full line rate. The black bars in Figure 7 show the filtering throughput per core measured in isolation, with both input and output data stored in memory, and both row and column filters enabled for all jobs. All the filters run faster than 500 Mbps per core (on an Intel Xeon X5650 processor), showing that even with conservative assumptions filtering will not degrade job performance.

We have also experimented with *declarative* rather than executable column filters, which allows us to use a fast native filtering engine (no JVM). Recall that the static analysis for column filtering generates a description of the tokenization process (e.g. separator character, regular expression) and a list of, e.g., integers that identify the columns that are dereferenced by the mapper. Instead of converting this to Java code, we encode it as a symbolic filter which is interpreted by a fast generic engine written in C. This engine is capable of processing inputs 2.5-9x faster than the Java filtering engine (median 3.7x) (Figure 7). We have further optimized the C engine using the SSE4 instruction set. The performance increased to 5-17x faster than the Java filtering engine (median 8.6). In addition to performance, the native engine is small and self-contained, and easily isolated, but

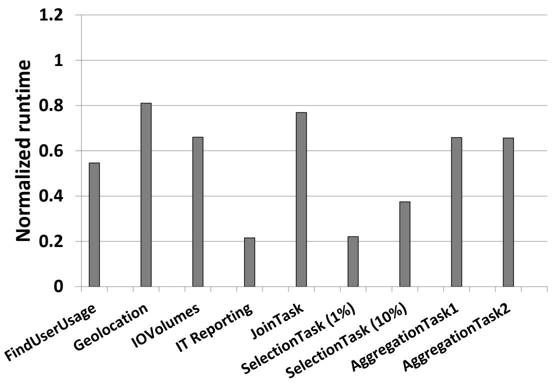

Figure 8: Job run times when using the Rhea filters normalized to the baseline execution time for the 9 example jobs fetching data across the WAN

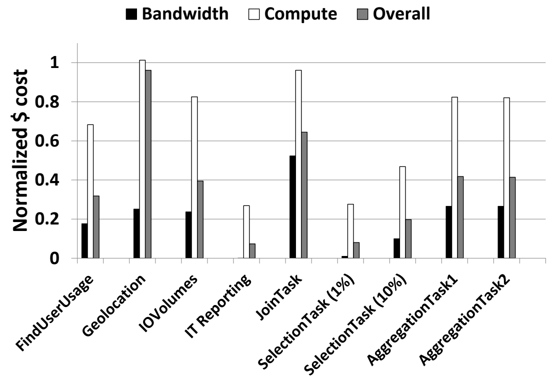

Figure 9: Dollar costs when using the Rhea filters normalized to the baseline cost for the 9 example jobs fetching data across the WAN

it does not perform row filtering. For row filtering currently we still use the slower Java filtering engine: row filters can perform arbitrary computations and we currently have no mechanism for converting them from Java to a declarative representation.

The performance numbers reported in Figure 7 are per processor core. It is straightforward to run in parallel multiple instances of the same filter, or even different filters. The system performance of filtering increases linearly with the number of cores, assuming of course enough I/O capacity for reading input data and network capacity for transmitting filtered data.

4.3 Cross cloud with online filtering

There are several scenarios where data must be read across the WAN. Data could be stored on-premise and occasionally accessed by a computation run on an elastic cloud infrastructure for cost or scalability reasons.

Alternatively, data could be in cloud storage but computation run on-premise: for computations that do not need elastic scalability and with a heavy duty cycle, on-premises computation is cheaper than renting cloud VMs. A third scenario is when the data are split across multiple providers or data centers. For example, a job might join a public data set available on one provider with a private data set stored on a different provider. The computation must run on one of the providers and access the data on the other one over the WAN.

WAN bandwidth is even scarcer than LAN bandwidth, and using cloud storage incurs egress charges. Thus using Rhea reduces both LAN and WAN traffic if the data are split across data centers. Since, we have already evaluated the LAN scenario, we will now evaluate the effects of filtering WAN traffic with Rhea in isolation. To do this we run the same nine jobs with the only difference being that the computations are run in the Azure-US-West data center and the storage is in the Azure-EU-North data center[2]. Rhea filters are deployed in a single large compute instance running as a filtering proxy in the Azure-EU-North data center's compute cluster.

Run time Figure 8 shows the run time when Rhea filtering is used, normalized to the baseline run time with no filtering. In general the results are similar to the LAN case. In all cases the CPU utilization reported on the filtering proxy was low (under 20% always). Thus the proxy is never the bottleneck. In most cases the WAN bandwidth is the bottleneck and the reduction in run time is due to the filter selectivity. However, for very selective filters (*IT Reporting*), the bottleneck is the data transfer from the storage layer to the filtering proxy over the LAN rather than the transfer from the proxy to the WAN. In this case the run time reduction reflects the ratio of the WAN egress bandwidth, to the LAN storage-to-compute bandwidth achieved by the filtering proxy.

Dollar costs In the WAN case, dollar costs reduce both for compute instances and also for egress bandwidth. While Rhea uses more compute instances (by adding a filtering instance) it significantly reduces egress bandwidth usage. Figure 9 shows the bandwidth, compute, and overall dollar costs of Rhea, each normalized to the corresponding value when not using Rhea. We use the standard Windows Azure charges of US$0.96 per hour for an extra-large instance and US$0.12 per GB of egress bandwidth. Surprisingly the compute costs also go down when using Rhea, even though it uses 5 instances per job rather than 4. This is because overall run times are reduced (again assuming per-second rather than per-hour billing, since most of our jobs take well under an hour to run). Thus compute costs are reduced in line with run time reductions and egress bandwidth charges in line with data reduction. In general, we expect the effect of egress bandwidth to dominate since computation is cheap relative to egress bandwidth: one hour of compute costs the same as only 8 GB of data egress. Of course, if filtering were offered at the storage servers then it would simply use spare computing cycles there and there would be no need to pay for a filtering VM instance.

[2]The input data sets for FindUserUsage and ComputeIOVolumes are too large to run in a reasonable time in this configuration. Hence for these two jobs we use a subset of the data, i.e. 1 hour's event logs rather than 1 day's.

5 Related work

There is a large body of work optimizing the performance of MapReduce, by better scheduling of jobs [21] and by handling of stragglers and failures [2, 40]. We are orthogonal to this work, aiming to minimize bandwidth between storage and compute.

Pyxis is a system for automatically partitioning database applications [7]. It uses profiling to identify opportunities for splitting a database application between server and application nodes, so that data transfers are minimized (like Rhea), but also *control transfers* are minimized. Unlike Rhea, Pyxis uses *state-aware* program partitioning. The evaluation has been done of Java applications running against MySQL. Compared to Rhea, the concerns are different: database applications might be more interactive (with more control transfers) than MapReduce data analytics programs; moreover in our setting we consider partitioning to be just an optimization that can opportunistically be enabled or disabled on the storage, even during the execution of a job and hence we do not modify the original job and make sure that the extracted filters are stateless. On the other hand, the optimization problem that determines the partitioning can take into account the available CPU budget at the database nodes, a desirable feature for Rhea as well.

MANIMAL is an analyzer for MapReduce jobs [25]. It uses static analysis techniques similar to Rhea's to generate an "index-generation program" which is run *off-line* to produce an indexed and column-projected version of the data. Index-generation programs must be run to completion on the entire data set to show any benefit, and must be re-run whenever additional data is appended. The entire data set must be read by Hadoop compute nodes and then the index written back to storage. This is not suitable for our scenario where there is limited bandwidth between storage and compute. By contrast, Rhea filters are *on-line* and have no additional overheads when fresh data are appended. Furthermore, MANIMAL uses logical formulas to encode the "execution descriptors" that perform row filtering by selecting appropriately indexed versions of the input data. Rhea filters can encode arbitrary Boolean functions over input rows.

Hadoop2SQL [22] allows the efficient execution of Hadoop code on a SQL database. The high-level goal is to transform a Hadoop program into a SQL query or, if the entire program cannot be transformed, parts of the program. This is achieved by using static analysis. The underlying assumption is that by pushing the Hadoop query into the SQL database it will be more efficient. In contrast, the goal of Rhea is to still enable Hadoop programs to run on a cluster against any store that can currently be used with Hadoop.

Using static analysis techniques to unravel properties of user-defined functions and exploit opportunities for optimizations is an area of active research. In the SUDO system [42], a simple static analysis of user-defined functions determines whether they preserve the input data partition properties. This information is used to optimize the shuffling stage of a distributed SCOPE job. In the context of the Stratosphere project [19], code analysis determines algebraic properties of user-defined functions and an optimizer exploits them to rewrite and further optimize the query operator graph. The NEMO system [16] also treats UDFs as open-boxes and tries to identify opportunities for applying more traditional "whole-program" optimizations, such as function and type specialization, code motion, and more. This could potentially be used to "split" mappers rather than "extract" filters, i.e. modify the mapper to avoid repeating the computation of the filter. However this is very difficult to do automatically, and indeed with NEMO manual modification is required to create such a split. Further, it means that filters can no longer be dynamically and transparently disabled since they are now an indispensable part of the application.

In the storage field the closest work is on Active Disks [20, 32]. Here compute resources are provided directly in the hard disk and a program is partitioned to run on the server and on the disks. A programmer is expected to manually partition the program, and the operations performed on the disk transform the data read from it. Rhea pushes computation into the storage layer but it does not require any explicit input from the programmer.

Inferring the schema of unstructured or semi-structure data is an interesting problem, especially for mining web pages [9, 10, 27]. Due to the difficulty of constructing hand-coded wrappers, previous work focused on automated ways to create those wrappers, often with the use of examples [27]. In Rhea, the equivalent hand-coded wrappers are actually embedded in the code of the mappers, and our challenge is to extract them in order to generate the filters. Moreover, Rhea deals with very flexible schemas (e.g. different rows may have different structure); our goal is not to interpret the data, but to extract enough information to construct the filters.

Rhea reduces the amount of data transferred by filtering the input data. Another approach to reduce the bytes transferred is with compression [33, 35]. We have found that compression complements filtering to further reduce the amount of bytes transferred in our data sets. Compression though requires changes to the user code, and increases the processing overhead at the storage nodes.

Regarding the static analysis part of this work, there is a huge volume of work on dependency analysis for slicing from the early 80's [36], to elaborate inter-procedural slicing [17]. More recently, Wiedermann *et al.* [37, 38] studied the program of extracting queries from imperative programs that work on *structured* data that adhere to a database schema. The techniques used are similar as ours here – an abstract interpretation framework keeps track of the used structure and the paths of the imperative program that perform output or update the state. A key difference is that Rhea targets unstructured text inputs, so a separate analysis is required to identify the parts of the input string that are used in a program. Moreover our tool extracts programs in a language as expressive as the original mapper – as opposed to a specialized query language. This allows us to be very flexible in the amount of computation that we can embed into the filter and push close to the data.

6 Conclusions

We have described Rhea, a system that automatically generates executable storage-side filters for unstructured data processing in the cloud. The filters encode the implicit data selectivity, in terms of row and column, for map functions in Hadoop jobs. They are created by performing static analysis on Java byte code.

We have demonstrated that Rhea filtering yields significant savings in the data transferred between storage and compute for a variety of realistic Hadoop jobs. Reduced bandwidth usage leads to faster job run times and lower dollar costs when data is transferred cross-cloud. The filters have several desirable properties: they are transparent, safe, lightweight, and best-effort. They are guaranteed to have no false negatives: all data used by a map job will be passed through the filter. Filtering is strictly an optimization. At any point in time the filter can be stopped and the remaining data returned unfiltered transparently to Hadoop.

We are currently working on generalizing Rhea to support other format such as binary formats, XML, and compressed text, as well as data processing tools and runtimes other than Hadoop and Java.

Acknowledgments

We thank the reviewers, and our shepherd Wenke Lee, who provided valuable feedback and advice. Thanks to the Microsoft Hadoop product team for valuable discussions and resources, and in particular Tim Mallalieu, Mike Flasko, Steve Maine, Alexander Stojanovic, and Dave Vronay.

References

[1] *Amazon Simple Storage Service (Amazon S3)*. http://aws.amazon.com/s3/. Accessed: 08/09/2011.

[2] G. Ananthanarayanan et al. "Reining in the Outliers in Map-Reduce Clusters using Mantri". *Operating Systems Design and Implementation (OSDI)*. USENIX, 2010.

[3] *Apache Cassandra*. http://cassandra.apache.org/. Accessed: 03/10/2011.

[4] B. Calder et al. "Windows Azure Storage: a highly available cloud storage service with strong consistency". *Proc. of 23rd Symp. on Operating Systems Principles (SOSP)*. ACM, 2011.

[5] R. Chaiken et al. "SCOPE: Easy and Effcient Parallel Processing of Massive Datasets". *VLDB*. 2008.

[6] Y. Chen et al. "The Case for Evaluating MapReduce Performance Using Workload Suites". *MASCOTS*. IEEE Computer Society, 2011.

[7] A. Cheung et al. "Automatic partitioning of database applications". *Proc. VLDB Endow.* 5.11 (2012).

[8] B. Cook, A. Podelski, and A. Rybalchenko. "Termination proofs for systems code". *Proc. of the SIGPLAN conf. on Programming Language Design and Implementation (PLDI)*. ACM, 2006.

[9] V. Crescenzi and G. Mecca. "Automatic Information Extraction from Large Websites". *J. ACM* 51.5 (2004).

[10] V. Crescenzi, G. Mecca, and P. Merialdo. "RoadRunner: Towards Automatic Data Extraction from Large Web Sites". *Proc. of 27th International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann Publishers Inc., 2001.

[11] T. von Eicken. *Network performance within Amazon EC2 and to Amazon S3*. http://blog.rightscale.com/2007/10/28/network-performance-within-amazon-ec2-and-to-amazon-s3/. Accessed: 08/09/2011. 2007.

[12] S. L. Garfinkel. *An Evaluation of Amazon's Grid Computing Services: EC2, S3 and SQS*. Tech. rep. Harvard University, 2007.

[13] A. F. Gates et al. "Building a high-level dataflow system on top of Map-Reduce: the Pig experience". *Proc. VLDB Endow.* 2.2 (2009).

[14] A. G. Greenberg et al. "VL2: a scalable and flexible data center network". *SIGCOMM*. Ed. by P. Rodriguez et al. ACM, 2009.

[15] S. Gulwani, K. K. Mehra, and T. Chilimbi. "SPEED: precise and efficient static estimation of program computational complexity". *Proc. of 36th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*. ACM, 2009.

[16] Z. Guo et al. "Nemo: Whole Program Optimization for Distributed Data-Parallel Computation." *Proc. of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2012.

[17] S. Horwitz, T. Reps, and D. Binkley. "Interprocedural slicing using dependence graphs". *SIGPLAN Not.* 39 (4 2004).

[18] L. Hubert et al. "Sawja: Static Analysis Workshop for Java". *Formal Verification of Object-Oriented Software*. Ed. by B. Beckert and C. Marché. Springer Berlin / Heidelberg, 2011.

[19] F. Hueske et al. "Opening the black boxes in data flow optimization". *Proc. VLDB Endow.* 5.11 (2012).

[20] L. Huston et al. "Diamond: A Storage Architecture for Early Discard in Interactive Search". *FAST*. USENIX, 2004.

[21] M. Isard et al. "Quincy: Fair Scheduling for Distributed Computing Clusters". *Proc. of 22nd ACM Symposium on Operating Systems Principles (SOSP)*. 2009.

[22] M.-Y. Iu and W. Zwaenepoel. "HadoopToSQL: A MapReduce query optimizer". *EuroSys'10*. 2010.

[23] S. Iyer. *Geo Location Data From DB-Pedia*. http://downloads.dbpedia.org/3.3/en/geo_en.csv.bz2. Accessed: 22/09/2011.

[24] S. Iyer. *Map Reduce Program to group articles in Wikipedia by their GEO location*. http://code.google.com/p/hadoop-map-reduce-examples/wiki/Wikipedia_GeoLocation. Accessed: 08/09/2011. 2009.

[25] E. Jahani, M. J. Cafarella, and C. Ré. "Automatic Optimization for MapReduce Programs". *PVLDB* 4.6 (2011).

[26] R. Jhala and R. Majumdar. "Path slicing". *Proc. of the 2005 ACM SIGPLAN conf. on Programming Language Design and Implementation (PLDI)*. ACM, 2005.

[27] N. Kushmerick, D. S. Weld, and R. B. Doorenbos. "Wrapper Induction for Information Extraction". *IJCAI (1)*. Morgan Kaufmann, 1997.

[28] A. Miné. "The octagon abstract domain". *Higher Order Symbol. Comput.* 19.1 (2006).

[29] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

[30] A. Pavlo et al. "A comparison of approaches to large-scale data analysis". *Proc. of 35th SIGMOD intl conf on Management of data*. ACM, 2009.

[31] *Public Data Sets on AWS*. http://aws.amazon.com/publicdatasets/. Accessed: 03/05/2012.

[32] E. Riedel et al. "Active Disks for Large-Scale Data Processing". *Computer* 34 (6 2001).

[33] *snappy: A fast compressor/decompressor*. http://code.google.com/p/snappy/. Accessed: 03/05/2012.

[34] A. Thusoo et al. "Hive - a petabyte scale data warehouse using Hadoop". *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*. IEEE, 2010.

[35] B. D. Vo and G. S. Manku. "RadixZip: linear time compression of token streams". *Proc of 33rd intl. conf. on Very Large Data Bases (VLDB)*. VLDB Endowment, 2007.

[36] M. Weiser. "Program slicing". *Proc. of 5th intl. conf. on Software Engineering (ICSE)*. IEEE Press, 1981.

[37] B. Wiedermann and W. R. Cook. "Extracting queries by static analysis of transparent persistence". *Proc. of 34th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*. ACM, 2007.

[38] B. Wiedermann, A. Ibrahim, and W. R. Cook. "Interprocedural query extraction for transparent persistence". *Proc. of 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications (OOPSLA)*. ACM, 2008.

[39] *Windows Azure Storage*. http://www.microsoft.com/windowsazure/features/storage/. Accessed: 08/09/2011.

[40] M. Zaharia et al. "Improving MapReduce Performance in Heterogeneous Environments". *Operating Systems Design and Implementation (OSDI)*. USENIX, 2008.

[41] *Zebra Reference Guide*. http://pig.apache.org/docs/r0.7.0/zebra_reference.html. Accessed: 22/09/2011. 2011.

[42] J. Zhang et al. "Optimizing data shuffling in data-parallel computation by understanding user-defined functions". *Proc. of 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.