Automatic IO Filtering for Optimizing Cloud Analytics
Microsoft Technical Report MSR-TR-2012-3

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

Abstract

Network bandwidth is often the bottleneck resource for large-scale data analytics. Cloud-based analytics platforms such as Amazon's Elastic Map Reduce provide high bandwidth within a compute cluster but limited bandwidth to storage resources such as S3 servers. If data is accessed from another public cloud or a private cloud, then the network is not only a performance bottleneck but also causes egress bandwidth charges.

This paper describes Rhea, a system to reduce traffic between storage and compute nodes for Hadoop MapReduce jobs. Rhea filters data read from storage by removing input rows, and column values within rows. Filters are job-specific and are automatically generated from static analysis of the mapper's byte code. Filters are stateless, side effect free, and do not change the output of the MapReduce computation. Filtering is transparent to both the compute and storage nodes and is best-effort, allowing it to opportunistically use spare CPU cycles when available. Our evaluation shows that Rhea filters significantly reduce the amount of bytes transferred (e.g. by a factor >5 for some of our jobs), and correspondingly reduce egress bandwidth charges and overall execution time by similar factors.

1 Introduction

Data analytics platforms like MapReduce [11], Dryad [19, 30], Hadoop [15] and SCOPE [10] enable simple scalable processing of large data sets. When combined with public cloud services that provide shared multi-tenant storage [3, 7] and compute [1, 6], they make it feasible to store and process large data sets in the cloud. For example, Amazon's Elastic MapReduce [2] allows users to upload Hadoop jobs which run on the EC2 compute service [1] and can process data stored in the S3 storage service [3].

A shared public cloud infrastructure often has limited bandwidth between the storage and the compute nodes. This is in contrast to a dedicated cluster, where computations such as MapReduce mappers can run directly on the same nodes that store the data. Cloud providers usually do not allow customer VMs to run on storage servers, for a variety of reasons including security and performance isolation. When analyzing large volumes of data, the network bandwidth between the storage and the compute servers becomes a bottleneck. This bandwidth scarcity is made worse when operating in a hybrid scenario such as a mixed public-private or public-public cloud setting. For example, a company might run a compute job in a private cluster which accesses data stored with a cloud provider. Alternatively, a compute job running on one cloud provider's infrastructure may access data stored with a different provider.

In the MapReduce programming model, used in Hadoop, the map function is invoked on every row in the input data. As a simple programming abstraction this has many advantages, and has proved to be very versatile, but it does require that every byte of the input data be transmitted to the mappers. This is a challenge when the storage and compute are not co-located. Within a single cloud this causes network congestion often putting stress on a significantly oversubscribed network. There have been several studies of different cloud providers showing the average throughput from storage to compute is low [14, 29]. When running the jobs across public clouds or between public and private clouds the situation is even worse, as there are ingress and egress bandwidth fees to pay, and the available network capacity between the clusters is even lower. Hence reducing the amount of data transferred across bottleneck or congested links should yield a reduction in execution time, and reduce costs when the data traverses cloud provider boundaries.

This paper describes Rhea, a system that automatically and transparently reduces the amount of data transferred between the storage and compute infrastructures for MapReduce-like workloads. MapReduce inputs typically consist of a large number of rows each with one

or more columns. E.g., for click-log data sets stored as text files, a row is a line of text with comma-separated columns. During the processing of the input data set the map function is applied to each row, and optionally generates some output which is then fed to the reduce phase. A key observation is that many rows result in no output at all from the map function, and the computed result would be identical if the row were not read at all. For rows that do generate output, not all columns in the row contribute to the output; many are simply ignored by the map code. Rhea exploits these two observations by filtering the data read from storage before it traverses the network. It removes superfluous rows as well as columns within rows.

Hadoop users submit MapReduce jobs to Rhea as compiled Java packages. Each job package contains the classes to be used for the map and reduce phases of the job together with job configuration state. Rhea applies static analysis techniques to the Java bytecode associated with the map operation of the user's job. The static analysis is used to create row filters and column selectors specific to that job. A row filter takes a single row as an input and returns true or false, indicating whether the row must be passed on to the mapper or not. A column selector takes a single row as input and returns a modified version of the row with one or more columns set to a null value, such as an empty string in the case of text-based rows. Thus Rhea can infer and exploit both *row selectivity* and *column selectivity* in Hadoop jobs. The filters themselves are simple Java methods. Rhea only requires the binary bytecode of the submitted job, not the source code, which means that a cloud provider can use the tool without requiring source code. Rhea filters are conservative in the sense that they never filter data that would cause a change in the mapper's output. Filters may have false positives, but never false negatives. In general, Rhea filters can be run anywhere upstream of the bottleneck link.

Rhea filters are *stateless* and free of *side effects*. This means filters are safe to run on storage servers, unlike the map code itself which can contain arbitrary operations. Filters from different users can be run simultaneously in the same address space. The filtering employed by Rhea is transparent to Hadoop. This allows filters to be inserted or removed at any time during a job. This means that filtering can be employed on a best-effort basis in response to the availability of resources. For instance, filters may be disabled when there is insufficient processing capacity on the storage server.

We have architected Rhea such that we do not need to make any modifications to Hadoop. Hadoop jobs are submitted to Rhea as a pre-processing step. The Rhea *filter generator* examines the code and identifies the conditions under which an input row (and columns within that row) will cause the mapper to generate output. Rhea also modifies the Hadoop control job object to redirect its I/O via the filtering proxy. The modified job can then be run on any unmodified Hadoop cluster, for example Amazon's Elastic MapReduce.

The rest of the paper is organized as follows. Section 2 expands on the potential benefits of Rhea in different scenarios. It also provides running examples that we use throughout the paper, to explain the operation of Rhea as well as demonstrate its benefits. Section 3 describes the design and implementation of Rhea, and details of its filter generation algorithms. Section 4 analyzes the performance improvement of our approach using realistic MapReduce jobs as well as microbenchmarks, as well as the overheads of filter generation and filter execution. Section 6 discusses related work, and Section 7 concludes with a summary of the strengths and limitations of Rhea and directions for future work.

2 Background

In this section we describe the types of cloud-based deployment where Rhea would provide benefit, and three example Hadoop applications that we use to motivate and evaluate Rhea.

2.1 Clouds

We consider three different usage scenarios, and how Rhea can be integrated to improve performance in these environments.

Public cloud This is where a single organization provides a multi-tenant cloud infrastructure with both storage and compute, e.g. Amazon EC2 (compute) and Amazon S3 (storage). Most public cloud providers, e.g. Amazon and Microsoft Azure, internally separate storage from compute for many reasons, including performance isolation and security. This separation is also driven by the fact that the storage can be accessed by services running outside the cloud as well as services running inside the cloud. When storage and compute are not co-located the bottleneck resource is the network bandwidth between them. Networks used in these cloud infrastructures are often oversubscribed, which makes the bandwidth valuable. Hence reducing storage-to-compute network traffic can improve the transfer times between storage and compute servers as well as overall utilization of the data center.

In this environment, the cloud provider would automatically generate and deploy Rhea filters when the user submits a compute job. Rhea filters would ideally run on the storage server. Alternatively, they could run on a compute server in the same rack as the storage server.

Both options require the cloud provider to explicitly support the use of Rhea filters.

Public-private cloud This refers to increasingly popular hybrid options where computation is performed on a private cluster but data is stored in a public cloud, or conversely data is stored on a private server but the elastic properties of a public compute cloud are exploited to enable compute intensive processing of the data. In these cases the bandwidth between the private and public cloud is the bottleneck resource, as providers charge ingress and egress bandwidth fees per GB transferred to and from the public cloud.

When the storage is in the private cloud, then Rhea filters can be run in that same cloud, at no additional cost. When the storage is in the public cloud, the cloud provider (e.g. S3) could natively support third-party Rhea filters running on the storage servers. However, if this is not supported, users can still run Rhea filters in a virtual machine on a compute cluster close to the storage (e.g. an EC2 instance). Such a proxy will still have better (and free) connectivity to the storage compared to accessing it over the wide area. 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.

Public-public cloud In this scenario, compute and storage are both in public clouds, but they are owned by two different operators, e.g. Amazon EC2 and Azure Storage. This could occur due to pricing or regulatory constraints about where data and compute is performed, or it could be because a job requires a public data set that is stored in a different cloud. In this case, as with the hybrid private-public cloud, the Rhea filters could be run in the storage infrastructure if supported natively, but if not the Rhea filter proxy can be run on a co-located compute infrastructure.

2.2 Example applications

We describe three Hadoop jobs that we use to motivate and evaluate Rhea. Two are based on processing large system logs; the third processes geo-location information extracted from Wikipedia articles.

2.2.1 Log processing

Hadoop is a good fit for text log processing and is frequently used for that purpose. The use of Java gives enough flexibility, for example, to parse semi-structured text rows with variable lengths and numbers of columns. At the same time, the simplicity and scalability of the MapReduce model allows large logs to be processed over many machines in parallel.

The specific example in this paper uses 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, and consume CPU and other resources on each server. The logs capture information about CPU, I/O, and other resource usage on these machines. They are periodically processed to gather statistics about utilization, identify heavy users, etc. In this system, there are two logs: the *process log* and the *activity log*. The process log has one row per process, with information about its task, user and total execution time. Each row has 18 columns. The process log accumulates 126 M rows/day, with an average row size of 325 bytes, resulting in 41 GB/day of data. The activity log records finer-grained information about the actions performed by each process, such as reading and writing files. The first column of each activity row is a type column indicating the type of activity. These rows have 10 columns, and the log accumulates at 53 GB/day.

FindUserUsage Our first example job is a *top-k query*: it identifies the top k users by total process execution time. This requires only the process log and not the activity log. It requires only 2 of the 18 columns (61 of 325 bytes on average) in each row of the process log (user ID and execution time). However, every row must be processed to correctly identify the top k users. Thus this job has column selectivity but no row selectivity.

ComputeIoVolumes Our second example job processes the log to compute a distribution across tasks of the amount of input and intermediate data read by the task from storage. This requires correlating rows representing I/O read requests from the activity log, with the task and process information in the process log. From the process log, rows corresponding to failed and killed processes are skipped, which results in only 69% of the rows being relevant. For these rows, only 4 of the 18 columns are used in the computation. From the activity log, only rows of type "I/O read" are relevant (25% of the total), and only 4 of the 10 columns are used. Thus this job has both row and column selectivity on both its inputs.

2.2.2 GeoLocation

This publicly available example application [23] groups Wikipedia articles by their geographical location. The input data is based on a publicly available data set [22]. It has 1.11 M rows and the total data size is 90 MB. 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; depending on the type, rows have either 3 or 4 columns in total. Only one of the two row types in the input data is relevant to the GeoLocation application. About 25% of the rows are of the relevant type, comprising 21% of the bytes. All the columns of the relevant rows are processed, hence there is no column selectivity.

2.3 Explicit versus 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 [8, 9] and "input format classes" in Zebra [32]. In such situations input data pre-filtering can be performed using standard ideas 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 not 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 [10] 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 from a static analysis of the application byte code.

3 Design and Implementation

The aim of Rhea is transparent filtering for MapReduce jobs running in a cloud infrastructure as well as across clouds. This leads to several design decisions. First, to use a proxy architecture: filters run in a proxy that can be placed anywhere between the compute and storage layers, and is transparent to both. With this approach, no changes are required to the compute platform (i.e Hadoop) or the storage platform (e.g. S3). It allows

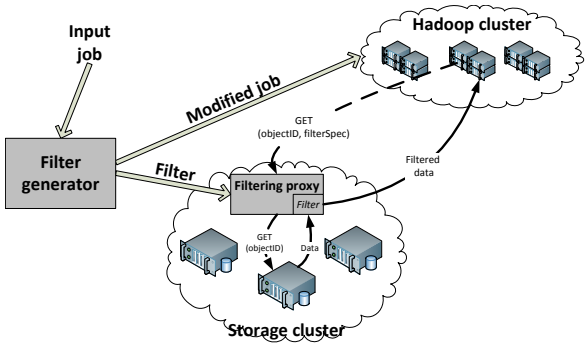


Figure 1: System architecture

flexibility in filter *placement*. For example, in the public cloud scenario the cloud provider would run filtering proxies on or near the storage servers; in the public/private and public/public cross-cloud scenarios, the user could additionally run filtering proxies in a virtual machine in a compute cluster in the same data center as the storage.

Figure 1 shows the architecture of Rhea, which consists of two components: a *filter generator* and a *filtering proxy*. The filter generator takes a Hadoop job as input. It generates a modified, Rhea-aware version of the job as well as a set of filters for it. The filters are uploaded to the filtering proxy. Hadoop jobs access storage using URIs (uniform resource identifiers); the Rhea filter generator automatically modifies these URIs to point to the filtering proxy. Additionally it embeds a "filter specification" in each URI, which is interpreted by the filtering proxy.

Our prototype filtering proxy works with Amazon's S3 storage and can also be configured to work with local storage. The proxy exposes the public S3 REST API to its client. S3 objects are named by a $< bucketID, objectID >$ tuple. The Hadoop framework wraps this in a thin "Hadoop S3 native file system" shim layer; however this layer simply uses path names as S3 object IDs. Rhea-aware jobs embed a filter specification in their path names, which is then passed unmodified to the filtering proxy inside the object ID. The filter specification is interpreted by the Rhea S3 proxy and is used to apply the appropriate filters; the proxy then removes the filter specification and passes the request on to the underlying S3 storage. This allows Rhea to be used without modifications either to Hadoop or to S3 servers. The Rhea architecture assumes that filtering proxies are run upstream of the bottleneck link, either by the provider or by the user. It also assumes that Hadoop jobs are preprocessed by the Rhea filter generator to take advantage of the proxy.

The Rhea filter generator takes a MapReduce program in Java byte code form, and generates a *row filter* and a

column selector for each mapper class found in the program. These are encoded as boolean 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 thus key components of Rhea. They are implemented on top of SAWJA [18], a tool which provides a high-level stackless representation of Java byte code as well as infrastructure for designing custom program analyses. We now describe how row filters and column selectors are generated.

3.1 Row Filters

Given a map method with signature:

```
public void map(LongWritable key,
    Text value,
    OutputCollector outputCollector,
    Reporter reporter)
```

the filter generator generates a method:

```
public boolean filter (LongWritable key,
   Text value,
   OutputCollector outputCollector,
   Reporter reporter)
```

Intuitively, `filter` is a "stripped-down" version of `map`, retaining only those instructions and execution paths from `map` that determine whether or not a given invocation will produce an output. Crucially, instructions that only determine the *content* of the output are not included in the filter.

Listing 1 shows an example: a fragment of the GeoLocation map method (Section 2.2.2). The method tokenizes the input `value` (line 7), then skips 3 tokens ahead (line 9–11), then examines if the `GEO_RSS_URI` static field is equal to the third token (line 13). If the condition is true, more processing follows (line 14–26) and some value is output on `outputCollector`.

Listing 2 shows the filter generated by Rhea for this mapper. Similarly to the `map` method, it tokenizes the input (line 8). It then compares the second token to the static field `GEO_RSS_URI` (line 12). Variable `bcvar8` here corresponds to `pointTyp` in `map`. This test exactly determines whether or not `map` would have produced output, and hence `filter` simply returns this boolean. In more complex cases, it may not be possible to determine exactly, for all execution paths, whether output would be produced. For execution paths where the analysis is not exact, the filter conservatively returns `true`. Thus the filter might have false positives but never false negatives.

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,

```
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

although it does affect the value of the output. The static analysis detects this and omits the extraction of the third token from `filter`. Second, `map` does substantial processing (line 14–26) before producing the output. All these instructions are omitted from the filter. This is again because these instructions affect the output *value* but are irrelevant to computing the output *condition*. The filter generator correctly detects this and omits these instructions.

3.1.1 Static analysis for row filter generation

Rhea filters are generated using a static analysis that is a variant of dependency analyses commonly found in the program slicing literature [17, 25]. At a high level, the approach is to consider all control flow paths that lead to an output statement, and then to find all instructions that influence conditional branch decisions on those paths.

More precisely, the analysis steps are:

- We identify a set OutputLabelSet of *output labels*. A *label* is simply a unique program point, with its associated instruction. An output label is a call to one of a small set of Hadoop methods that are provided for mappers to generate output, e.g., `OutputCollector.collect` or `Context.out`.
- Next, we collect all control flow labels (branch instructions) that form part of any execution path

```
public boolean filter (LongWritable bcvar1,
   Text bcvar2,
   OutputCollector bcvar3, Reporter bcvar4) {

 boolean cond = false;
 String bcvar5 = bcvar2.toString();
 String irvar0 = "\t";
 StringTokenizer bcvar6 =
   new StringTokenizer(bcvar5,irvar0);
 String bcvar7 = bcvar6.nextToken();
 String bcvar8 = bcvar6.nextToken();
 boolean irvar0_1=GEO_RSS_URI.equals(bcvar8);

 cond = ((irvar0_1?1:0)  !=  0);
 if (!cond) return false;
 return true;
}
```

Listing 2: Row filter generated for GeoLocation

```
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

leading to an output label. We call this the RelevantCtflSet. It contains both conditional and unconditional branches.

- The next step is a label-flow analysis: we implement a standard forward analysis [26] inside the analysis framework provided by SAWJA. We compute, for every point in the program, a map from a variable to the labels that may influence the value of the variable *at that program point*. Hence this step returns a map from labels, to a map from variables to label sets:

$$\mathsf{FlowMap} : \mathsf{Label} \mapsto (\mathsf{Var} \mapsto \mathsf{LabelSet})$$

- We compute the closure of the RelevantCtflSet: for every instruction in RelevantCtflSet, we accumulate all the labels that may affect the *value* that is used in that control flow instruction to perform a jump.

$$\begin{array}{rcl} \mathsf{RelevantSet} & = & \mathsf{RelevantCtflSet} \\ & \cup & \{\, \ell \mid \exists \ell_c, \exists x, \\ & & \quad \ell_c \in \mathsf{RelevantCtflSet} \\ & & \quad \wedge\, x \in \mathit{vars}(\mathit{instr}(\ell_c)) \\ & & \quad \wedge\, \ell \in \mathsf{FlowMap}(\ell_c)(x) \,\} \end{array}$$

  Here $\mathit{vars}(\mathit{instr}(\ell_c))$ is the set of variables referenced in the instruction at label ℓ_c. Since we only consider control flow instructions here, it is the set of variables used in the branch conditional (if any).
- We emit the code corresponding to RelevantSet, replacing output labels with `return true` statements, and inserting `return false` at all other exits from the control flow graph.

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 (Section 2.2.1) all rows must be examined to generate the output, but only a subset of the columns are relevant. Suppressing the irrelevant column values will save bandwidth without having any effect on correctness. As a simple example, consider a mapper which is given a comma-separated input row `"alice,usa,25"`. If the mapper only uses the first and last columns and ignores the second, we could safely transform the input record to the value `"alice,,25"` without changing the program's behavior. Note that we retain the column separators, which allows the mapper to parse the input row exactly as if we had not suppressed the irrelevant columns. This means that we can transparently mix filtered and unfiltered input data in a single execution. Some mappers use regular expressions to tokenize the input, e.g., multiple consecutive commas could count as a single separator. In this case, the string `"alice,,25"` would not be a correct filter output, but the string `"alice,?,25"` would be correct.

For each `map` method, the Rhea filter generator produces a `select` method:

```
public String select (LongWritable key,
    Text value,
    OutputCollector outputCollector,
    Reporter reporter)
```

This method is called by the filtering proxy to transform an input row into a shorter but equivalent (as far as pro-

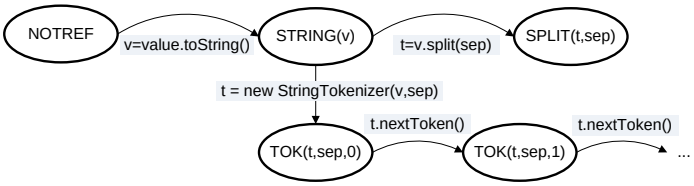


Figure 2: Simple transition system for column selector analysis

cessing is concerned) text record. Column filtering can (but need not) be combined with row filtering; typically the proxy first calls `filter` and then calls `select` for those inputs for which `filter` returns `true`.

For the GeoLocation map function in Listing 1, Rhea generates the column selector shown in Listing 3. The map function tokenizes its input and only examines the first three tokens of the input. The column selector code captures this by retaining only the first three tokens. It tokenizes the input string, and builds a new string from the generated tokens, replacing all unused tokens with a filler value, which is either an empty string or a single character. The filler value is computed dynamically based on the separator used for tokenization, to ensure that the results are always correct when the string is again tokenized by the map function. The string output from the column selector is then appended to the filter output and send back to the mapper in place of the original text.

3.2.1 Static analysis for column selection

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.split()`, or a tokenization library, or a regular expression matcher. This flexibility makes the problem of correctly removing irrelevant substrings challenging. Our solution is to design the analysis in a way that accommodates common patterns of input column usage, and conservatively performs no input column selection *at all* for usage patterns that lie outside this domain.

Our analysis first assigns to each program point one of the states from Figure 2 to capture the current representation of the input, either as a string (state `STRING`) or a sequence of tokens (state `TOK`). Calls to `String.split()` or `StringTokenizer()` result in state transitions. For the example in Listing 1 the initial state in the beginning of `map` is `NOTREF`, to mean that the input value has not been used yet. At line 6 the call to `toString()` transitions to state `STRING(dataRow)`. At line 7 the state becomes `TOK(dataTokenizer,"\t",0)`. The last parameter of `TOK` represents the current token position of `dataTokenizer`, which advances to 1, 2, and 3 in the subsequent lines. The rest of the code does not affect the state: note that `StringTokenizer` is used again, but not on the input string, and hence this does not cause a state change.

We also include an error state (not shown in the figure) to catch tokenization patterns that are not supported. Unexpected transitions, such as referencing the original input when in the `TOK` state, lead to the error state. If an error state is reached, then the analysis is abandoned, no column selector is generated, and Rhea conservatively defaults to transmitting the input row unmodified. Furthermore, to allow for a statically unknown number of calls to `nextToken()` (e.g. resulting from loops) our analysis also includes another variation of the `TOK` state, which we write `TOKAFTER(t,sep,n)` and signifies that the current position of the tokenizer `t` is at or somewhere after `n`. In our implementation we appropriately extend this set of states to form a lattice and – as in the static analysis for row filtering – instantiate SAWJA's static analysis framework, to compute the final map from program points to one of these states.

Assuming that no error states have been produced, we identify all program points that dereference an input token used in the rest of the `map` job. We then consult the computed state at each of these program points, which gives us a concrete position or a numerical constraint about the tokens used. We finally generate code that iterates over all tokens and emits those that are of interest, as Listing 3 shows. Hence our column selection is always safe, even when several control flow paths of the mapper can assume different numbers of columns present in the input row.

Finally, though we have focused on common string tokenization input patterns, the same technique extends to other simple input usage models, such as field selection from binary data or substring selection from the input row. We plan to explore these extensions in future work.

3.3 Correctness

For a generated (row or column) filter to be correct, *the output of the mapper must be the same for both the filtered and the original unfiltered input data set*. Correctness is ensured by enforcing the following properties:

Single-record correctness If the filter rejects an input row, then calling the mapper on that row will result in no output. Furthermore, if the mapper produces an output on the original row, then calling mapper on the column-filtered row will produce *the same* output.

Globally-stateless mappers Single-record correctness is not sufficient for end-to-end correctness, if state can be shared across different invocations of the mapper. Filtered and unfiltered input may result in the same output but may modify internal state (such as fields of the mapper class or static fields). If this state could then affect the output of future invocations, then the output of the mapper may be affected by filtering the input dataset. For example if `map` maintained a count of input records, and enabled output only for every 10th record, then it may not be correct to filter out any records, even those which do not directly generate output. Map methods do not usually maintain such state (state such as counters is typically handled by reducers). However, if `map` is not stateless, then no filter should be generated.

A simple way to ensure statelessness is to disallow all access to global state. However this is too restrictive: mappers often read configuration or job parameter information from fields initialized when the job is started. In some cases, mappers also need to update class fields. For example, in Listing 1, line 25–27, the variables `geoLocationKey` and `geoLocationName` are updated. However, the updated values are immediately output.

Hence we use a less restrictive condition. We allow accesses to fields of the mapper class, but ensure that if such a field is read in any execution path of the mapper then (i) that field is always set earlier in the same execution path (as in GeoLocation), or (ii) that field is never updated in *any* execution path of the mapper. To handle calls to external library methods we use *whitelisting*. Methods are only whitelisted if they are known not to mutate their arguments or receiver object, and calls to such methods are allowed. Calls to non-whitelisted methods are only allowed if they cannot be passed an argument that contains a reference to any global variable (this is ensured by a data-flow analysis).

Isolation We must finally ensure that filters for one job cannot affect other jobs or the external environment, such as performing filesystem I/O or loading classes dynamically via class loaders, or calling methods of the `OutputCollector` argument.

We have a proof sketch of the end-to-end correctness properties of Rhea based on the three aforementioned conditions. We leave to future work a formal proof based on the operational semantics of byte code and of the SAWJA high-level representation [12].

3.4 Caching and delta filters

The design described so far is aimed at reducing the amount of data fetched by each Hadoop job independent of other jobs running on the same cluster. If users repeatedly run the same or similar Hadoop job (e.g., periodically re-running a job to account for new input data), then bandwidth can be further optimized by caching input data at or near the compute nodes. E.g., a Hadoop compute cluster could cache data in the HDFS file system that is already used to stage intermediate results.

Combining caching with filtering requires two things. First, caching must be best-effort: cache resources such as local files on compute nodes are not guaranteed to be available and could be lost at any time. Second, caching must be both correct and efficient when combined with filtering. For example, when re-running a MapReduce job with the same Rhea filter but new input files, the system should only fetch the filtered data from the new files and combine them with the locally cached filtered data, to generate the same result as if the entire input data had been re-read from storage.

Rhea supports such scenarios with an optional, best-effort *caching proxy* component. The caching proxy can run anywhere between the compute nodes and the filtering proxy, and intercepts all requests to the filtering proxy. The caching proxy manages a local cache with each cache entry corresponding to a unique $< fileID, filterID >$ pair. Read requests that match both the file and the filter ID of a cache entry are served entirely locally. Requests matching only the file ID but with a different filter are split into a local cache request and a remote request. The filter specification in the remote request is replaced with a "delta" filter specification that fetches only the rows missing from the locally cached version.

If a job requests data filtered by a boolean filter $f_2(r)$, and the cache already contains the same object but filtered by a different boolean filter $f_1(r)$, then the delta filter is a function that computes $f_2(r) \wedge \neg f_1(r)$. Currently we simply encode this boolean expression into the filter specification, and the filtering proxy computes it by executing the methods that represent f_1 and f_2, sequentially. We are working on generating the delta filter as a single method that computes the entire expression, which should be more efficient. We are also working on ways to extend delta filtering to column selection: currently delta filtering must be disabled if column selection is enabled.

4 Evaluation

In this section we evaluate the performance benefits and the cost savings provided by Rhea.

4.1 Experimental setup

We use two different testbeds to evaluate the *in-cloud* and *private-public cloud* scenarios. In the in-cloud scenario, filters run directly on the storage nodes, and compute nodes are connected to storage over (oversubscribed) LAN links. Since we cannot run our code on cloud storage servers such as Amazon S3 servers, we replicate this configuration in a small private cluster testbed. The testbed has 27 compute and 1 storage node. The compute nodes each have 4 2.27 GHz Xeon cores and 12 GB of memory. The storage node has 8 2.13 GHz Xeon cores, 192 GB of memory, and an OCZ Revodrive X2 160 GB SSD with a measured sequential read performance of 783 MB/s. We use the SSD to ensure that disk I/O is never a performance bottleneck in our experiments.

All the nodes have 1 Gbps interfaces connecting via a single 1 Gbps switch. Since the benefits of Rhea improve as available storage-to-compute bandwidth decreases, experiments on our testbed will underestimate the benefits of Rhea in large data centers with oversubscribed networks between the storage and compute nodes. The aggregate storage-to-compute bandwidth in our testbed is 1 Gbps (128 MB/s), which compares favorably with measured bandwidth from S3 to EC2, reported as 20–50 MB/s [14, 29].

We also evaluate the private-public cloud scenario, where data are stored on Amazon S3 servers, filters run in an Amazon EC2 compute instance, and the Hadoop job runs in a private cloud. We use the cluster testbed described above to run the Hadoop jobs. We use a single S3 bucket to store input data and a single high-CPU extra large Windows EC2 instance to run the filtering proxy, which costs US$1.16/hour. Both the S3 bucket and the EC2 instance are in Amazon's EU "region": thus bandwidth between the data and the filters is free and relatively high. Traffic from the filtering proxy to the compute cluster must traverse WAN links with lower bandwidth, and an egress bandwidth charge of US$0.12/GB. In this scenario we run the GeoLocation job unmodified. For the log processing jobs (FindUserUsage and ComputeIoVolumes), the input data sets used for the in-cloud experiments are too large to process in a reasonable amount of time in the private-public configuration. Hence we use a subset of the input data: 1 hour's worth of system logs rather than 1 day's. The log data were also anonymized for these experiments: all values except those used in the computation were replaced by their MD5 hashes.

All graphs in this section show means of five identical runs, with error bars showing standard deviations.

4.2 Job characteristics

Table 1 shows the key characteristics of each job: the input data size, analysis time, and the selectivity achievable with row filtering, with column selection, and with both row filtering and column selection. The analysis was done on a Xeon 2.5 GHz processor and is dominated by the time to read the entire Java program including libraries into memory. We define selectivity as the ratio of output bytes to input bytes. Recall from Section 2 that FindUserUsage has only column selectivity; GeoLocation shows only row selectivity, and ComputeIOVolumes has both row and column selectivity.

4.3 Overall benefits

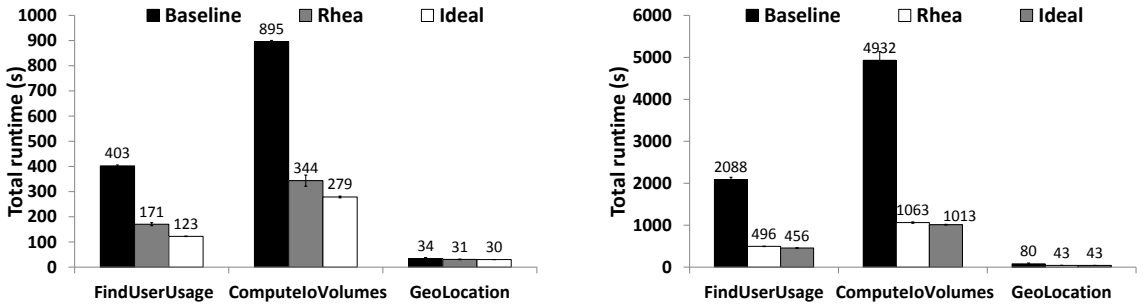
Figure 3(a) shows the performance of the three jobs for the in-cloud testbed, in three different configurations. The *baseline* configuration has no filtering enabled. The *Rhea* configuration has both row filtering and column selection enabled. Finally, the *ideal* configuration is one where the input data is pre-processed with both row filtering and column selection. The ideal configuration thus gives all the benefits of Rhea with none of the runtime overheads of online filtering.

Rhea reduces overall job runtime by 58%, 62%, and 9% respectively for FindUserUsage, ComputeIoVolumes, and GeoLocation. Thus it more than doubles the performance of the two large jobs. The GeoLocation job is very small and its runtime is dominated by constant factors such as JVM startup and Hadoop co-ordination. As the input data size increases we would expect the benefits of Rhea, which scale with the amount of data, to increase relative to these constant costs. Rhea achieves lower performance than the "ideal" configuration due to the overhead of the filtering proxy. The filtering proxy is currently an unoptimized Java implementation and we are tuning its performance. However, we note that the benefits already outweigh the costs for all the jobs.

Figure 3(b) shows the performance of the three jobs running with the private-public setup. Note that the runtimes are substantially longer, despite using reduced data sets for the two larger jobs, because of the low bandwidth across the WAN. We also note that the benefits of Rhea are much larger in this case, and very close to the ideal case. There is still a small difference between Rhea and "ideal", which is due to a network rather than a processing bottleneck. The "ideal" performance is measured by reading filtered data directly from S3, and hence the S3 bandwidth required is lower than with Rhea. In general the proxy is bottlenecked on its output connections (i.e. the WAN) and not on its input (reading from S3). However both of these bandwidths are variable, and occasionally the rate at which unfiltered data is read from S3 drops below the rate required to saturate the output

Job	Analysis time	Data	Data reduction		
			Row	Column	Total
FindUserUsage	4.5 s	38 GB	100%	18%	18%
ComputeIOVolumes	9.1 s	88 GB	45%	58%	24%
GeoLocation	4.4 s	90 MB	25%	100%	25%

Table 1: Job characteristics



(a) In-cloud

(b) Private-public cloud

Figure 3: Hadoop job runtimes

Job	Baseline cost	Rhea cost	
		Total	CPU
FindUserUsage	$0.42	$0.25	$0.16
ComputeIOVolumes	$1.00	$0.55	$0.34
GeoLocation	$0.01	$0.02	$0.01

Table 2: Dollar costs for private-public cloud

connection with filtered data. This creates a slight drop in performance for Rhea with respect to the ideal case.

In the private-public scenario, we wish not only to improve performance but also to reduce costs. Rhea reduces dollar costs by trading EC2 computation, a cheap resource, for more expensive egress bandwidth. Table 2 shows the dollar costs for the three jobs in the baseline and Rhea configurations. We see that Rhea reduces the cost of the two larger jobs by 40% and 45% respectively. The absolute dollar costs are low: this merely reflects the small size of our experimental jobs and data sets. The GeoLocation job show a very small increase in cost. Hadoop startup and co-ordination cause the filtering proxy to be mostly idle, which results in CPU charges but not bandwidth charges.

Measured CPU utilization on the filtering proxy was low (less than 9% for all jobs), with the network always being the bottleneck. The cost advantage of Rhea can be increased by improving the CPU utilization. For example, in our experiments we used a "high-cpu extra-large" instance (20 EC2 compute units, $1.16/hour). However, given the low CPU utilization, a "large" instance (4 EC2 compute units, $0.48/hour) would give the same effective performance with higher CPU utilization and reduce the CPU cost further.

Table 2 assumes per-second billing of VM instances, not the per-hour billing offered by Amazon. Since our sample data sets all take less than 2 hours to process, the effect of rounding up to the nearest hour would be significant. In practice, this round-up effect can be removed by keeping the filtering proxy utilized with large jobs, by running jobs back to back, and by sharing the proxy across multiple users.

4.4 Detailed performance results

In this section we show further experimental results to examine the benefits of Rhea in more detail. These results only use the in-cloud testbed as it lends itself more easily to repeatable controlled experiments.

Figure 4 shows the separate effects of row filtering and column selection for the three jobs. In addition to the baseline, Rhea, and ideal configurations, we show a *row-only* and a *column-only* configuration. These correspond to running the Rhea filtering proxy with only row filtering and only column selection enabled, respectively.

As expected, we note that FindUserUsage does not benefit from row filtering, whereas ComputeIoVolumes

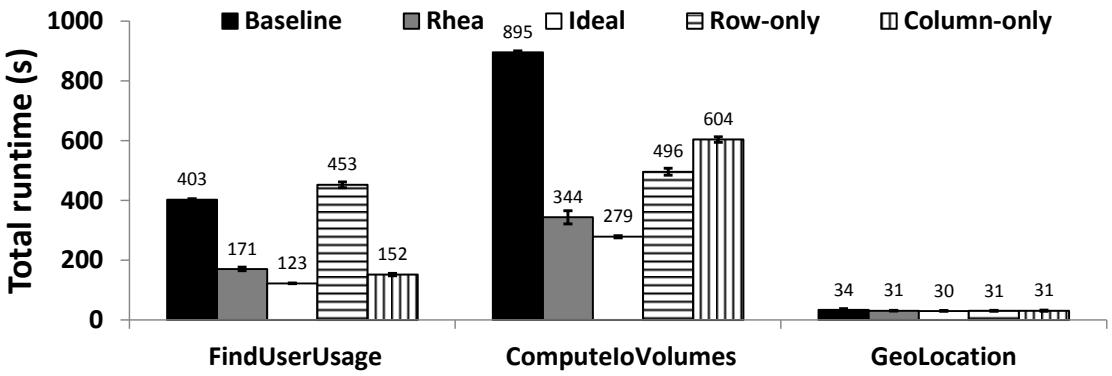


Figure 4: Run times for different configurations (in-cloud)

benefits from both row filtering and column selection. Row filtering for FindUserUsage actually reduces performance slightly due to the overhead of the filtering proxy, but this is more than compensated for by the benefits of column selection. Row filtering and column selection can be individually disabled, for example if their overheads are high: thus for FindUserUsage we would choose the "column-only" configuration rather than the default "Rhea" configuration.

The runtimes shown so far include computation time on the Hadoop cluster. Since Rhea's main benefit comes from reducing the data fetch time from storage to compute nodes, it is useful to examine this alone. We did this by re-running the same experiments as above with the same sets of filters, but with the map functions in the compute nodes set to no-ops. This meant that no map or reduce work was done in the compute cluster, and the job was considered finished when all input data had been fetched to all mapper nodes. Figure 5 shows the result of these experiments. The benefits of Rhea measured on fetch time alone are slightly higher (66%, 68% and 12% for the three jobs) than when the entire runtime is considered. However, the differences are small, and the overall results are similar. This is because the runtime is dominated by the time to fetch data, even in the in-cloud testbed: the computation time only adds a small and constant factor to the runtime.

5 Discussion

The results in the previous section have demonstrated the performance savings that we can generate for three representative Hadoop jobs. However, for jobs that use all their input data Rhea will not provide benefit. Jobs often used to benchmark MapReduce systems, such as Sort and WordCount, fall into this category. On the other hand, we do believe that there are many jobs that benefit from Rhea. We have not been able to quantify the percentage of such jobs as few organizations openly share the jobs and data sets that they are processing. However, we have observed *many* data analytics jobs in our organization that are selective and benefit from using Rhea.

Currently in Rhea we generate filters that are executable. However, the static analysis could also produce non-executable symbolic versions of the filters in addition to the executable filters. This would allow us to reason about the relationship between filters, e.g., to know that one filter subsumes another or to combine two filters statically to generated a more efficient delta filter. We have not yet explored this direction.

We have also, so far, restricted ourselves to analyzing native Hadoop jobs. Hadoop is a generic programming framework that allows the use of programs written in other languages to be called during execution of a job using the pipes interface. Rhea does not support the analysis of these programs, and simply does not produce any filters. If a filter is not generated the data is fetched directly from the storage layer. As a result all input data will be transferred from storage to compute servers, but correctness is guaranteed.

In this paper we have assumed by default that data is stored and transmitted uncompressed. Rhea filters can also be used transparently on compressed data, with a small additional CPU overhead of uncompressing and recompressing the data in the filtering proxy.

6 Related work

There is a large amount of work on improving the performance of MapReduce, usually focusing around the

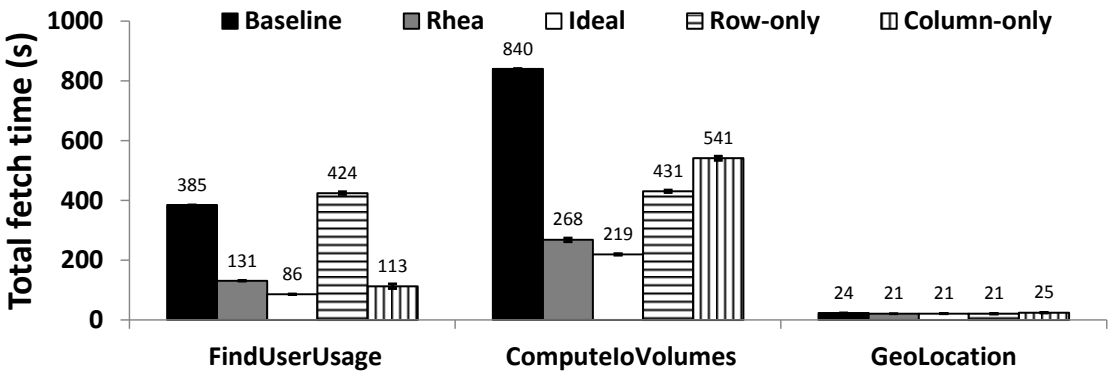


Figure 5: Data fetch times for different configurations (in-cloud)

scheduling of jobs [20] and handling of stragglers and failures [5, 31]. We are orthogonal to this work, in that we are considering how to minimize storage to compute bandwidth requirements.

Concurrently with this work, Jahani et al. [24] have developed the MANIMAL analyzer for MapReduce jobs. Given a Hadoop program, MANIMAL uses static analysis techniques similar to Rhea's to generate an "index-generation program" which is run *off-line* to generate an indexed and column-projected version of the data. Index-generation programs themselves are MapReduce programs and must be run *off-line*. This means that they 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 expensive and impractical for our scenario where there is limited bandwidth between storage and compute. By contrast, Rhea filters are *on-line*, transparent, cheap, and best-effort. They can be dynamically enabled/disabled during job execution and there is no overhead when they are disabled. Because they are online, they do not cause additional overheads when fresh data is 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 by contrast can encode arbitrary boolean functions over input rows. This allows Rhea to handle mappers which perform complex processing of input fields, e.g. string manipulation.

Hadoop2SQL [21] 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. The filters generated are interposed on the data path between the storage and compute to ensure only data that is to be processed is transferred.

In the storage field the closest work is on Active Disks [27]. 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. The computation performed is determined automatically using static analysis. Furthermore, the filters only suppress irrelevant data rather than transforming or processing the data in other ways. At any point in an execution the filter can be disabled, and the correctness of the Hadoop job will not be impacted.

As far as the static analysis part of this work is concerned, there exist effective tools for the static analysis of Java code, such as Java model checkers that can check properties of certain execution paths [4, 28]. The extraction of (symbolic) conditions for path reachability is also a common theme in the literature [13, 16] – in our case the output must be executable conditions which moreover have to preserve the end-to-end semantics of the Map/Reduce job. The Key ideas behind our analysis originate in classical work on how to identify "irrelevant" instructions for the control flow to reach certain points [17, 25].

7 Conclusions

We have described Rhea, a system that automatically generates input data filters. 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 a Hadoop job.

In jobs that use only subsets of the input data, such as most log processing jobs, we have demonstrated that this can yield significant reductions in the data transferred between storage and compute. Even in jobs that touch all rows, the column filtering can provide significant benefit. Filtering the data improves performance for both in-cloud and private-public cloud scenarios and also reduces dollar costs in the private-public cloud scenario.

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 are 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 investigating generalizing Rhea to support other data processing tools beyond Hadoop (which use different language runtimes).

References

[1] Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`. Accessed: 08/09/2011.

[2] Amazon Elastic MapReduce. `http://aws.amazon.com/elasticmapreduce/`. Accessed: 08/09/2011.

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

[4] S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In O. Grumberg and M. Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 134–138. Springer, 2007.

[5] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in Map-Reduce clusters using Mantri. In *OSDI*, 2010.

[6] Windows Azure Compute. `http://www.microsoft.com/windowsazure/features/compute/`. Accessed: 08/09/2011.

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

[8] Apache Cassandra. `http://cassandra.apache.org/`. Accessed: 03/10/2011.

[9] Apache Cassandra API. `http://wiki.apache.org/cassandra/API`. Accessed: 03/10/2011.

[10] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S.Weaver, and J. Zhou. SCOPE: Easy and effcient parallel processing of massive datasets. In *VLDB*, 2008.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 2004.

[12] D. Demange, T. Jensen, and D. Pichardie. A provably correct stackless intermediate representation for java bytecode. In *Proceedings of the 8th Asian conference on Programming languages and systems*, APLAS'10, pages 97–113, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 270–280, New York, NY, USA, 2008. ACM.

[14] S. L. Garfinkel. An Evaluation of Amazon's Grid Computing Services: EC2, S3 and SQS. Technical report, Harvard University, 2007.

[15] Apache Hadoop. `http://hadoop.apache.org/`. Accessed: 08/09/2011.

[16] C. Hammer, R. Schaade, and G. Snelting. Static path conditions for Java. In *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, PLAS '08, pages 57–66, New York, NY, USA, 2008. ACM.

[17] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Not.*, 39:229–243, April 2004.

[18] L. Hubert, N. Barré, F. Besson, D. Demange, T. Jensen, V. Monfort, D. Pichardie, and T. Turpin. Sawja: Static Analysis Workshop for Java. In B. Beckert and C. Marché, editors, *Formal Verification of Object-Oriented Software*, pages 92–106. Springer Berlin / Heidelberg, 2011.

[19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *European Conference on Computer Systems (EuroSys)*. ACM, 2007.

[20] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

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

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

[23] 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`, 2009. Accessed: 08/09/2011.

[24] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for MapReduce programs. *PVLDB*, 4(6):385–396, 2011.

[25] R. Jhala and R. Majumdar. Path slicing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 38–47, New York, NY, USA, 2005. ACM.

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

[27] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active disks for large-scale data processing. *Computer*, 34:68–74, June 2001.

[28] W. Visser and P. C. Mehlitz. Model checking programs with Java PathFinder. In *SPIN*, page 27, 2005.

[29] 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/`, 2007. Accessed: 08/09/2011.

[30] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Operating Systems Design and Implementation (OSDI)*. USENIX, 2008.

[31] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.

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