

Interpreting the data: Parallel analysis with Sawzall

Rob Pike, Sean Dorward, Robert Griesemer and Sean Quinlan
Google, Inc. CA, USA

Abstract Very large data sets often have a flat but regular structure and span multiple disks and machines. Examples include telephone call records, network logs, and web document repositories. These large data sets are not amenable to study using traditional database techniques, if only because they can be too large to fit in a single relational database. On the other hand, many of the analyses done on them can be expressed using simple, easily distributed computations: filtering, aggregation, extraction of statistics, and so on.

We present a system for automating such analyses. A filtering phase, in which a query is expressed using a new procedural programming language, emits data to an aggregation phase. Both phases are distributed over hundreds or even thousands of computers. The results are then collated and saved to a file. The design – including the separation into two phases, the form of the programming language, and the properties of the aggregators – exploits the parallelism inherent in having data and computation distributed across many machines.

1. Introduction

Many data sets are too large, too dynamic, or just too unwieldy to be housed productively in a relational database. One common scenario is a set of many plain files – sometimes amounting to petabytes of data – distributed across many disks on many computers (Fig. 1). The files in turn comprise many records, organized along some obvious axes such as time or geography. Examples might include a web page repository used to construct the index for an internet search engine, the system health records from thousands of on-line server machines, telephone call records or other business transaction logs, network packet traces, web server query logs, or even higher-level data such as satellite imagery.

Quite often the analyses applied to these data sets can be expressed simply, using operations much less sophisticated than a general SQL query. For instance, we might wish to count records that satisfy a certain property, or extract them, or search for anomalous records, or construct frequency histograms of the values of certain fields in the records. In other cases the analyses might be more intricate but still be expressible as a series of simpler steps, operations that can be mapped easily onto a set of records in files.

Not all problems are like this, of course. Some benefit from the tools of traditional programming – arithmetic, patterns, storage of intermediate values, functions, and so on – provided by procedural languages such as Python [1] or Awk [12]. If the data is unstructured or textual, or if the query requires extensive calculation to arrive at the relevant data for each record, or if many distinct but related calculations must be applied during a single scan of the data set, a procedural language is often the right tool. Awk was specifically designed for making data mining easy. Although it is used for many other things today, it was inspired by a tool written by Marc Rochkind that executed procedural code when a regular expression matched a record in telephone system log data [13]. Even today, there are large Awk programs in use for mining telephone logs. Languages such as C or C++, while capable of handling

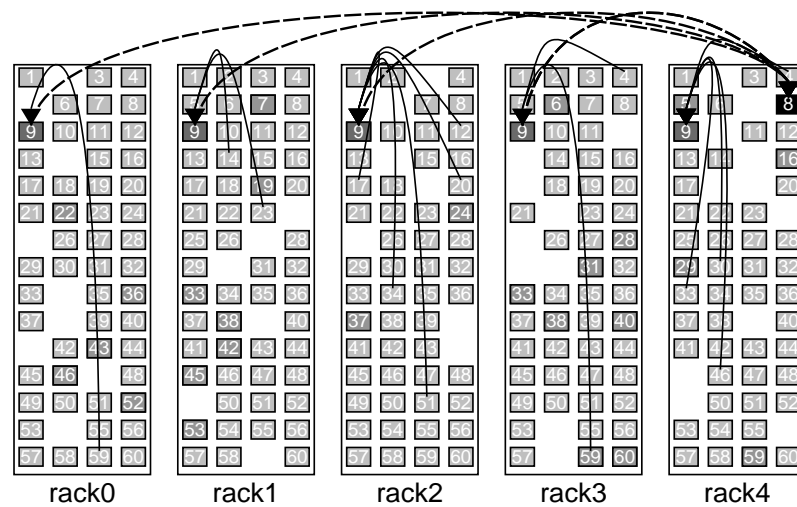


Fig. 1. Five racks of 50–55 working computers each, with four disks per machine. Such a configuration might have a hundred terabytes of data to be processed, distributed across some or all of the machines. Tremendous parallelism can be achieved by running a filtering phase independently on all 250+ machines and aggregating their emitted results over a network between them (the arcs). Solid arcs represent data flowing from the analysis machines to the aggregators; dashed arcs represent the aggregated data being merged, first into one file per aggregation machine and then to a single final, collated output file.

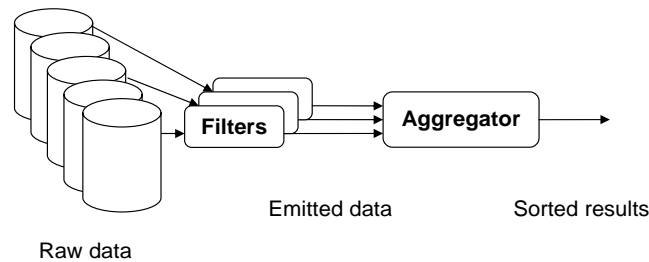


Fig. 2. The overall flow of filtering, aggregating, and collating. Each stage typically involves less data than the previous.

such tasks, are more awkward to use and require more effort on the part of the programmer. Still, Awk and Python are not panaceas; for instance, they have no inherent facilities for processing data on multiple machines.

Since the data records we wish to process do live on many machines, it would be fruitful to exploit the combined computing power to perform these analyses. In particular, if the individual steps can be expressed as query operations that can be evaluated one record at a time, we can distribute the calculation across all the machines and achieve very high throughput. The results of these operations will then require an aggregation phase. For example, if we are counting records, we need to gather the counts from the individual machines before we can report the total count.

We therefore break our calculations into two phases. The first phase evaluates the analysis on each record individually, while the second phase aggregates the results (Fig. 2). The system described in this paper goes even further, however. The analysis in the first phase is expressed in a new procedural programming language that executes one record at a time, in isolation, to calculate query results for each record. The second phase is restricted to a set of predefined aggregators that process the intermediate results generated by the first phase. By restricting the calculations to this model, we can achieve very high throughput. Although not all calculations fit this model well, the ability to harness a thousand or more machines with a few lines of code provides some compensation.

Of course, there are still many subproblems that remain to be solved. The calculation must be divided into pieces and distributed across the machines holding the data, keeping the computation as near the data as possible to avoid network bottlenecks. And when there are many machines there is a high probability of some of them failing during the analysis, so the system must be fault tolerant. These are difficult and interesting problems, but they should be handled without the involvement of the user of the system. Google has several pieces of infrastructure, including

GFS [9] and MapReduce [8], that cope with fault tolerance and reliability and provide a powerful framework upon which to implement a large, parallel system for distributed analysis. We therefore hide the details from the user and concentrate on the job at hand: expressing the analysis cleanly and executing it quickly.

2. Overview

A typical job will process anywhere from a gigabyte to many terabytes of data on hundreds or even thousands of machines in parallel, some executing the query while others aggregate the results. An analysis may consume months of CPU time, but with a thousand machines that will only take a few hours of real time.

Our system's design is influenced by two observations.

First, if the querying operations are commutative across records, the order in which the records are processed is unimportant. We can therefore work through the input in arbitrary order.

Second, if the aggregation operations are commutative, the order in which the intermediate values are processed is unimportant. Moreover, if they are also associative, the intermediate values can be grouped arbitrarily or even aggregated in stages. As an example, counting involves addition, which is guaranteed to be the same independent of the order in which the values are added and independent of any intermediate subtotals that are formed to coalesce intermediate values.

The constraints of commutativity and associativity are not too restrictive; they permit a wide range of valuable analyses, including: counting, filtering, sampling, generating histograms, finding the most frequent items, and many more.

The set of aggregations is limited but the query phase can involve more general computations, which we express in a new interpreted, procedural programming language called Sawzall.¹ (An interpreted language is fast enough: most of the programs are small and on large data sets the calculation tends to be I/O bound, as is discussed in the section on performance.)

An analysis operates as follows. First the input is divided into pieces to be processed separately, perhaps individual files or groups of records, which are located on multiple storage nodes.

Next, a Sawzall interpreter is instantiated for each piece of data. This will involve many machines, perhaps the same machines that store the data or perhaps a different, nearby set. There will often be more data pieces than computers, in which case scheduling and load balancing software will dispatch the pieces to machines as they complete the processing of prior pieces.

The Sawzall program operates on each input record individually. The output of the program is, for each record, zero or more intermediate values – integers, strings, key-value pairs, tuples, etc. – to be combined with values from other records.

These intermediate values are sent to further computation nodes running the aggregators, which collate and reduce the intermediate values to create the final results. In a typical run, the majority of machines will run Sawzall and a smaller fraction will run the aggregators, reflecting not only computational overhead but also network load balancing issues; at each stage, the amount of data flowing is less than at the stage before (see Fig. 2).

Once all the processing is complete, the results will be spread across multiple files, one per aggregation machine. A final step collects, collates, and formats the final results, to be viewed or stored in a single file.

3. A simple example

These ideas may become clearer in the context of a simple example. Let's say our input is a set of files comprising records that each contain one floating-point number. This complete Sawzall program will read the input and produce three results: the number of records, the sum of the values, and the sum of the squares of the values.

¹Not related to the portable reciprocating power saw trademark of the Milwaukee Electric Tool Corporation.

```

count: table sum of int;
total: table sum of float;
sum_of_squares: table sum of float;
x: float = input;
emit count <- 1;
emit total <- x;
emit sum_of_squares <- x * x;

```

The first three lines declare the aggregators `count`, `total`, and `sum_of_squares`. The keyword `table` introduces an aggregator type; aggregators are called tables in Sawzall even though they may be singletons. These particular tables are sum tables; they add up the values emitted to them, `ints` or `floats` as appropriate.

For each input record, Sawzall initializes the pre-defined variable `input` to the uninterpreted byte string of the input record. Therefore, the line

```
x: float = input;
```

converts the input record from its external representation into a native floating-point number, which is stored in a local variable `x`. Finally, the three `emit` statements send intermediate values to the aggregators.

When run, the program is instantiated once for each input record. Local variables declared by the program are created anew for each instance, while tables declared in the program are shared by all instances. Emitted values are summed up by the global tables. When all records have been processed, the values in the tables are saved to one or more files.

The next few sections outline some of the Google infrastructure this system is built upon: protocol buffers, the Google File System, the Workqueue, and MapReduce. Subsequent sections describe the language and other novel components of the system in more detail.

4. Protocol buffers

Although originally conceived for defining the messages communicated between servers, Google's *protocol buffers* are also used to describe the format of permanent records stored on disk. They serve a purpose similar to XML, but have a much denser, binary representation, which is in turn often wrapped by an additional compression layer for even greater density.

Protocol buffers are described by a data description language (DDL) that defines the content of the messages. A *protocol compiler* takes this language and generates executable code to manipulate the protocol buffers. A flag to the protocol compiler specifies the output language: C++, Java, Python, and so on. For each protocol buffer type in the DDL file, the output of the protocol compiler includes the native data structure definition, code to access the fields, marshaling and unmarshaling code to translate between the native representation and the external binary format, and debugging and pretty-printing interfaces. The generated code is compiled and linked with the application to provide efficient, clean access to the records. There is also a tool suite for examining and debugging stored protocol buffers.

The DDL forms a clear, compact, extensible notation describing the layout of the binary records and naming the fields. Protocol buffer types are roughly analogous to C `structs`: they comprise named, typed fields. However, the DDL includes two additional properties for each field: a distinguishing integral tag, used to identify the field in the binary representation, and an indication of whether the field is required or optional. These additional properties allow for backward-compatible extension of a protocol buffer, by marking all new fields as optional and assigning them an unused tag.

For example, the following describes a protocol buffer with two required fields. Field `x` has tag 1, and field `y` has tag 2.

```

parsed message Point {
    required int32 x = 1;
    required int32 y = 2;
};

```

To extend this two-dimensional point, one can add a new, optional field with a new tag. All existing in courier stored on disk remain readable; they are compatible with the new definition since the new field is optional.

```
parsed message Point {
    required int32 x = 1;
    required int32 y = 2;
    optional string label = 3;
};
```

Most of the data sets our system operates on are stored as records in protocol buffer format. The protocol compiler was extended by adding support for Sawzall to enable convenient, efficient I/O of protocol buffers in the new language.

5. Google File System (GFS)

The data sets are often stored in GFS, the Google File System [9]. GFS provides a reliable distributed storage system that can grow to petabyte scale by keeping data in 64-megabyte “chunks” stored on disks spread across thousands of machines. Each chunk is replicated, usually 3 times, on different machines so GFS can recover seamlessly from disk or machine failure.

GFS runs as an application-level file system with a traditional hierarchical naming scheme. The data sets themselves have regular structure, represented by a large set of individual GFS files each around a gigabyte in size. For example, a document repository (the result of a web crawl) holding a few billion HTML pages might be stored as several thousand files each storing a million or so documents of a few kilobytes each, compressed.

6. Workqueue and MapReduce

The business of scheduling a job to run on a cluster of machines is handled by software called (somewhat misleadingly) the Workqueue. In effect, the Workqueue creates a large-scale time sharing system out of an array of computers and their disks. It schedules jobs, allocates resources, reports status, and collects the results.

The Workqueue is similar to several other systems such as Condor [16]. We often overlay a Workqueue cluster and a GFS cluster on the same set of machines. Since GFS is a storage system, its CPUs are often lightly loaded, and the free computing cycles can be used to run Workqueue jobs.

MapReduce [8] is a software library for applications that run on the Workqueue. It performs three primary services. First, it provides an execution model for programs that operate on many data items in parallel. Second, it isolates the application from the details of running a distributed program, including issues such as data distribution, scheduling, and fault tolerance. Finally, when possible it schedules the computations so each unit runs on the machine or rack that holds its GFS data, reducing the load on the network.

As the name implies, the execution model consists of two phases: a first phase that *maps* an execution across all the items in the data set; and a second phase that *reduces* the results of the first phase to arrive at a final answer. For example, a sort program using MapReduce would map a standard sort algorithm upon each of the files in the data set, then reduce the individual results by running a merge sort to produce the final output. On a thousand-machine cluster, a MapReduce implementation can sort a terabyte of data at an aggregate rate of a gigabyte per second [9].

Our data processing system is built on top of MapReduce. The Sawzall interpreter runs in the map phase. It is instantiated in parallel on many machines, with each instantiation processing one file or perhaps GFS chunk. The Sawzall program executes once for each record of the data set. The output of this map phase is a set of data items to be accumulated in the aggregators. The aggregators run in the reduce phase to condense the results to the final output.

The following sections describe these pieces in more detail.

7. Sawzall language overview

The query language, Sawzall, operates at about the level of a type-safe scripting language. For problems that can be solved in Sawzall, the resulting code is much simpler and shorter – by a factor of ten or more – than the corresponding C++ code in MapReduce.

The syntax of statements and expressions is borrowed largely from C; `for` loops, `while` loops, `if` statements and so on take their familiar form. Declarations borrow from the Pascal tradition:

```
i: int;          # a simple integer declaration
i: int = 0;      # a declaration with an initial value
```

The basic types include integer (`int`), a 64-bit signed value; floating point (`float`), a double-precision IEEE value; and special integer-like types called `time` and `fingerprint`, also 64 bits long. The `time` type has microsecond resolution and the libraries include convenient functions for decomposing and manipulating these values. The `fingerprint` type represents an internally-computed hash of another value, which makes it easy to construct data structures such as aggregators indexed by the fingerprint of a datum. As another example, one might use the fingerprint of a URL or of its HTML contents to do efficient comparison or fair selection among a set of documents.

There are also two primitive array-like types: `bytes`, similar to a C array of unsigned `char`; and `string`, which is defined to hold characters from the Unicode character set. There is no “character” type; the elements of byte arrays and strings are `int` values, although an `int` is capable of holding a much larger value than will fit in a byte or string element.

Compound types include arrays, maps (an overloaded term in this paper), and tuples. Arrays are indexed by integer values, while maps are like associative arrays or Python dictionaries and may be indexed by any type, with the indices unordered and the storage for the elements created on demand. Finally, tuples represent arbitrary groupings of data, like a C struct or Pascal record. A `typedef`-like mechanism allows any type to be given a shorthand name.

Conversion operators translate values from one type to another, and encapsulate a wide range of conversions. For instance, to extract the floating point number represented by a string, one simply converts the value:

```
f: float;
s: string = "1.234";
f = float(s);
```

Some conversions take parameters; for example

```
string(1234, 16)
```

generates the hexadecimal representation of the integer, while

```
string(utf8_bytes, "UTF-8")
```

will interpret the byte array in UTF-8 representation and return a string of the resulting Unicode character values.

For convenience, and to avoid the stutter endemic in some languages’ declaration syntaxes, in initializations the appropriate conversion operation (with default parameter values) is supplied implicitly by the compiler. Thus

```
b: bytes = "Hello, world!\n";
```

is equivalent to the more verbose

```
b: bytes = bytes("Hello, world!\n", "UTF-8");
```

Values of any type can be converted into strings to aid debugging.

One of the most important conversion operations is associated with protocol buffers. A compile-time directive in Sawzall, `proto`, somewhat analogous to C’s `#include` directive, imports the DDL for a protocol buffer from a file and defines the Sawzall tuple type that describes the layout. Given that tuple description, one can convert the input protocol buffer to a Sawzall value.

For each input record, the special variable `input` is initialized by the interpreter to the uninterpreted byte array holding the data, typically a protocol buffer. In effect, the execution of the Sawzall program for each record begins with the implicit statement:

```
input: bytes = next_record_from_input();
```

Therefore, if the file `some_record.proto` includes the definition of a protocol buffer of type `Record`, the following code will parse each input record and store it in the variable `r`:

```
proto "some_record.proto" # define 'Record'
r: Record = input;        # convert input to Record
```

The language has a number of other traditional features such as functions and a wide selection of intrinsic library calls. Among the intrinsics are functions that connect to existing code for internationalization, document parsing, and so on.

7.1. Input and aggregation

Although at the statement level Sawzall is a fairly ordinary language, it has two very unusual features, both in some sense outside the language itself:

1. A Sawzall program defines the operations to be performed on a single record of the data. There is nothing in the language to enable examining multiple input records simultaneously, or even to have the contents of one input record influence the processing of another.
2. The only output primitive in the language is the `emit` statement, which sends data to an external aggregator that gathers the results from each record and correlates and processes the result.

Thus the usual behavior of a Sawzall program is to take the input variable, parse it into a data structure by a conversion operation, examine the data, and emit some values. We saw this pattern in the simple example of Section 3.

Here is a more representative Sawzall program. Given a set of logs of the submissions to our source code management system, this program will show how the rate of submission varies through the week, at one-minute resolution:

```
proto "p4stat.proto"
submitsthroughweek: table sum[minute: int] of count: int;

log: P4ChangelistStats = input;
t: time = log.time; # microseconds

minute: int = minuteof(t) + 60*(hourof(t) + 24*(dayofweek(t) - 1));

emit submitsthroughweek[minute] <- 1;
```

The program begins by importing the protocol buffer description from the file `p4stat.proto`. In particular, it declares the type `P4ChangelistStats`. (The programmer must know the type that will arise from the `proto` directive, but this is an accepted property of the protocol buffer DDL.)

The declaration of `submitsthroughweek` appears next. It defines a table of sum values, indexed by the integer `minute`. Note that the index value in the declaration of the table is given an optional name (`minute`). This name serves no semantic purpose, but makes the declaration easier to understand and provides a label for the field in the aggregated output.

The declaration of `log` converts the input byte array (using code generated by the `proto` directive) into the Sawzall type `P4ChangelistStats`, a tuple, and stores the result in the variable `log`. Then we pull out the time value and, for brevity, store it in the variable `t`.

The next declaration has a more complicated initialization expression that uses some built-in functions to extract the cardinal number of the minute of the week from the time value.

Finally, the `emit` statement counts the submission represented in this record by adding its contribution to the particular minute of the week in which it appeared.

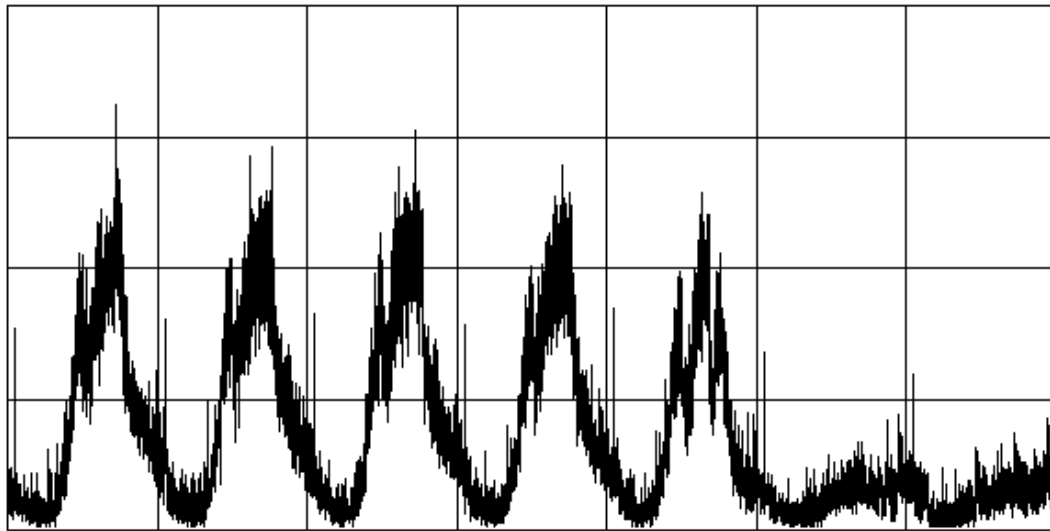


Fig. 3. Frequency of submits to the source code repository through the week. The graph starts at midnight Monday morning.

To summarize, this program, for each record, extracts the time stamp, bins the time to a minute within the week, and adds one to the count for that minute. Then, implicitly, it starts again on the next record.

When we run this program over all the submission logs – which span many months – and plot the result, we see an aggregated view of how activity tends to vary through the week, minute by minute. The output looks like this:

```
submitsthroughweek[0] = 27
submitsthroughweek[1] = 31
submitsthroughweek[2] = 52
submitsthroughweek[3] = 41
...
submitsthroughweek[10079] = 34
```

When plotted, the graph looks like Fig. 3.

The salient point is not the data itself, of course, but the simplicity of the program that extracts it.

7.2. More about aggregators

Aggregation is done outside the language for a couple of reasons. A more traditional language would use the language itself to correlate results, but some of the aggregation algorithms are sophisticated and best implemented in a native language and packaged in some form. More important, drawing an explicit line between filtering and aggregation enables a high degree of parallelism, even though it hides the parallelism from the language itself. Nowhere in Sawzall is the multiplicity of records apparent, yet a typical Sawzall job operates on billions of records, often on hundreds or thousands of machines simultaneously.

Focusing attention on the aggregators encourages the creation of unusual statistical collectors. There are a number of aggregators available; here is an incomplete list, with examples:

Collection:

```
c: table collection of string;
```

A simple list of all the emitted values, including all duplicates, in arbitrary order.

Sample:

```
s: table sample(100) of string;
```


Like collection, but chooses an unbiased sample of the emitted values. The size of the desired sample is provided as a parameter.

Sum:

```
s: table sum of { count: int, revenue: float };
```

The summation of all the emitted values. The values must be arithmetic or composed of arithmetic values, as is the tuple in the example. For compound values, the components are summed elementwise. In the example shown, if count were always 1 when emitted, the average revenue could be computed after the run by dividing revenue by count.

Maximum:

```
m: table maximum(10) of string weight length: int;
```

The highest-weighted values. The values are tagged with the weight, and the value with the highest weight is chosen. The parameter (10 in the example) specifies the number of values to keep. The weight, introduced by the obvious keyword, has its type (here int) provided in the declaration and its value in the emit statement. For the example given,

```
emit m <- s weight len(s);
```

will report the ten longest strings in the data.

Quantile:

```
q: table quantile(101) of response_in_ms: int;
```

Use the set of emitted values to construct a cumulative probability distribution represented by the quantile values for each increment of probability. (The algorithm is a distributed variant of that of Greenwald and Khanna [10].) The example could be used to see how system response varies. With parameter 101, it computes percentiles; table indices in that case range from 0 to 100, so the element at index 50 records the median time, while the element at index 99 is the time value for which 99% of the response times were equal or lower.

Top:

```
t: table top(10) of language: string;
```

Estimate which values are the most popular. (By contrast, the maximum table finds the items with the highest weight, not the highest frequency of occurrence.)

For our example,

```
emit t <- language_of_document(input);
```

will estimate the ten most frequently occurring languages found in a document repository.

For large data sets, it can be prohibitively expensive to find the precise ordering of frequency of occurrence, but there are efficient estimators. The top table uses a distributed variant of the algorithm by Charikar, Chen, and Farach-Colton [5]. The algorithm is approximate: with high probability it returns approximately the correct top elements. Its commutativity and associativity are also approximate: changing the order that the input is processed can change the final results. To compensate, in addition to computing the counts of the elements, we compute the estimated error of those counts. If the error is small compared to the counts, the quality of the results is high, while if the error is relatively large, chances are the results are bad. The algorithm used in the top tables works well for skewed distributions but produces unreliable results when all the values occur with similar frequency. For our analyses this is rarely an issue.

Unique:

```
u: table unique(10000) of string;
```

The unique table is unusual. It reports the estimated size of the population of unique items emitted to it. A sum table could be used to count the total number of elements, but a unique table will ignore duplicates; in effect it computes the size of the set of input values. The unique table is also unusual in that its output is always a count, regardless of the type of the values emitted to it. The parameter specifies the size of the internal table used to do the estimation; a value of 10000 generates a final value within $\pm 2\%$ of the correct answer with 95% probability. (For set size N , the standard deviation is about $N \times param^{-1/2}$.)

7.3. Implementing an aggregator

Occasionally, ideas arise for new aggregators to be supported by Sawzall. Adding new aggregators is fairly easy, although not trivial. Their implementations are connected to the Sawzall runtime and interact with system internals, managing low-level data formats and distributed computations. Moreover, most aggregators handle many different Sawzall data types, further complicating their implementations. There is support in the Sawzall runtime to ease these tasks, but we have not lavished these libraries with the same care given the Sawzall language.

The Sawzall runtime itself manages the plumbing of emitting values, passing data between machines, and so on, while a set of five functions specific to each aggregator class handles type checking, allocation of aggregation state, merging emitted values, packaging the data into an intermediate form suitable for further merging, and producing the final output data. To add a new aggregator, the programmer implements this five-function interface and links it with the Sawzall runtime. For simple aggregators, such as `sum`, the implementation is straightforward. For more sophisticated aggregators, such as `quantile` and `top`, care must be taken to choose an algorithm that is commutative, associative and reasonably efficient for distributed processing. Our smallest aggregators are implemented in about 200 lines of C++, while the largest require about 1000.

Some aggregators can process data as part of the mapping phase to reduce the network bandwidth to the aggregators. For instance, a `sum` table can add the individual elements locally, emitting only an occasional subtotal value to the remote aggregator. In MapReduce terminology, this is the *combining* phase of the MapReduce run, a sort of middle optimization step between map and reduce.

It would be possible to create a new aggregator-specification language, or perhaps extend Sawzall to handle the job. However, we have found that on the rare occasion that a new aggregator is needed, it's been easy to implement. Moreover, an aggregator's performance is crucial for the performance of the system, and since it needs to interact with the Mapreduce framework, it is best expressed in C++, the language of the Sawzall runtime and Mapreduce.

7.4. Indexed aggregators

An aggregator can be indexed, which in effect creates a distinct individual aggregator for each unique value of the index. The index can be of any Sawzall type and can be compounded to construct multidimensional aggregators.

For example, if we are examining web server logs, the table

```
table top(1000) [country: string] [hour: int] of request: string;
```

could be used to find the 1000 most popular request strings for each country, for each hour.

The Sawzall runtime automatically creates individual aggregators as new index values arise, similarly to maps adding entries as new key values occur. The aggregation phase will collate values according to index and generate the appropriate aggregated values for each distinct index value.

As part of the collation, the values are sorted in index order, to facilitate merging values from different machines. When the job completes, the values are therefore arranged in index order, which means the output from the aggregators is in index order.

The indices themselves form a useful source of information. To return to the web server example above, after the run the set of values recorded in the `country` index form a record of the set of countries from which requests were received. In effect, the properties of index generation turn the indices into a way to recover sets of values. The indices resulting from

```
t1: table sum[country: string] of int
```

would be equivalent to the values collected by

```
t2: table collection of country: string
```

with duplicates removed. Its output would be of the form

```
t1["china"] = 123456
t1["japan"] = 142367
...
```

which the user would need to post-process with a separate tool to extract the set of countries.

8. System model

Now that the basic features of the language have been presented, we can give an overview of the high-level system model by which data analyses are conducted.

The system operates in a batch execution style: the user submits a job, which runs on a fixed set of files, and collects the output at the end of the run. The input format and location (typically a set of files in GFS) and the output destination are specified outside the language, as arguments to the command that submits the job to the system.

The command is called `saw` and its basic parameters define the name of the Sawzall program to be run, a pattern that matches the names of the files that hold the data records to be processed, and a set of files to receive the output. A typical job might be instantiated like this:

```
saw --program code.sz1 \
    --workqueue testing \
    --input_files /gfs/cluster1/2005-02-0[1-7]/submits.* \
    --destination /gfs/cluster2/$USER/output@100
```

The `program` flag names the file containing the Sawzall source of the program to be run, while the `workqueue` flag names the Workqueue cluster on which to run it. The `input_files` argument accepts standard Unix shell file-name-matching metacharacters to identify the files to be processed. The `destination` argument names the output files to be generated, using the notation that an at sign (@) introduces the number of files across which to distribute the results.

After the run, the user collects the data using a tool that accepts a `source` argument with the same notation as the `destination` argument to `saw`:

```
dump --source /gfs/cluster2/$USER/output@100 --format csv
```

This program merges the output data distributed across the named files and prints the final results in the specified format.

When a `saw` job request is received by the system, a Sawzall processor is invoked to verify that the program is syntactically valid. If it is, the source code is sent to the Workqueue machines for execution. The number of such machines is determined by looking at the size of the input and the number of input files, with a default upper limit proportional to the size of the Workqueue. A flag to `saw` can override this behavior and set the number explicitly.

These machines each compile the Sawzall program and run it on their portion of the input data, one record at a time. The emitted values are passed to the Sawzall runtime system, which collects them and performs an initial stage of aggregation locally, to reduce the amount of intermediate data sent to the aggregation machines. The runtime sends the data when the memory used for intermediate values reaches a threshold, or when input processing is complete.

The aggregation machines, whose number is determined by the count in the `destination` flag to `saw`, collect the intermediate outputs from the execution machines and merge them to each produce a sorted subset of the output. To ensure the merging process sees all the necessary data, the intermediate outputs are assigned to aggregation machines deterministically according to the particular table and any index values it may have. Each aggregation machine's output is written to a file in GFS; the output of the entire job is a set of files, one per aggregation machine. Aggregation is therefore a parallel operation that can take full advantage of the cluster's resources.

In general, the generation of the output is fairly evenly spread across the aggregation machines. Each element of a table has relatively modest size and the elements are spread across the different outputs. Collection tables are an exception; such tables can be of arbitrary size. To avoid overloading a single aggregation machine, the values inside a collection table are spread across the aggregation machines in a round-robin fashion. No aggregation machine sees all the intermediate values, but collection tables by definition do not combine duplicate results so this is not a problem.

Values stored by the aggregation machines are not in final form, but instead in an intermediate form carrying information to facilitate merging it with other values. In fact, this same intermediate form is used to pass the intermediate values from the execution machines to the aggregation machines. By leaving the values in an intermediate form, they can be merged with the values from another job. Large Sawzall jobs can be broken into pieces to be run

separately, with final results constructed by merging the elements in a final phase. (This is one advantage of this system over plain MapReduce; even MapReduce can have problems with jobs that run for days or weeks of real time.)

This multi-stage merge – at execution machines, aggregations machines, and perhaps across jobs – is the reason the aggregators must be associative to work well. They must also be commutative, since the order in which the input records are processed is undefined, as is the order of merging of the intermediate values.

Typically, the data resulting from an analysis is much smaller than the input, but there are important examples where this is not the case. For instance, a program could use an indexed collection table to organize the input along some relevant axis, in which case the output would be as large as the input. Such transformations are handled efficiently even for large data sets. The input and the output are evenly partitioned across the execution and aggregation engines respectively. Each aggregation engine sorts its portion of the output data using a disk-based sort. The resulting data can then be combined efficiently to produce the final output using a merge sort.

8.1. Implementation

The Sawzall language is implemented as a conventional compiler, written in C++, whose target language is an interpreted instruction set, or byte-code. The compiler and the byte-code interpreter are part of the same binary, so the user presents source code to Sawzall and the system runs it directly. (This is a common but not universal design; Java for instance splits its compiler and interpreter into separate programs.) Actually, the Sawzall language system is structured as a library with an external interface that accepts source code and compiles and runs it, along with bindings to connect to externally-provided aggregators.

Those aggregators are implemented by *saw*, which is a program that links to the Sawzall compiler and interpreter library and is in turn implemented above MapReduce, running Sawzall in the map phase and the aggregators in the reduce phase. MapReduce manages the distribution of execution and aggregation machines, locates the computation near the GFS machines holding the data to minimize network traffic and thereby improve throughput, and handles machine failure and other faults. The system is therefore a “MapReduction” [8], but an unusual one. It actually incorporates two instances of MapReduce.

The first, quick job runs in parallel on the Workqueue to evaluate the size of the input and set up the main job. (There may be many thousands of input files and it’s worth checking their size in parallel.) The second job then runs Sawzall. *Saw* thus acts in part as a convenient wrapper for MapReduce, setting up the job automatically based on the input rather than the usual method involving a number of user-specified flags. More important is the convenience of Sawzall and its aggregators. Leaving aside the comfort afforded by its simple programming model, Sawzall is implemented within a single binary that runs whatever code is presented to it; by contrast, other MapReductions must be written in C++ and compiled separately. Moreover, the various aggregators in Sawzall provide functionality that significantly extends the power of MapReduce. It would be possible to create a MapReduce wrapper library that contained the implementation of *top* tables and others, but even then they would not be nearly as simple or convenient to use as they are from Sawzall. Moreover, the ideas for some of those aggregators grew out of the system model provided by Sawzall. Although restrictive, the focus required by the Sawzall execution model can lead to unusual and creative programming ideas.

8.2. Chaining

One common use of the output of a Sawzall job is to inject the resulting data into a traditional relational database for subsequent analysis. This is usually done by a separate custom program, perhaps written in Python, that transforms the data into the SQL code to create the table. We might one day provide a more direct way to achieve this injection.

Sometimes the result of a Sawzall job is provided as input to another, by a process called *chaining*. A simple example is to calculate the exact “top 10” list for some input. The Sawzall *top* table is efficient but approximate. If the exact results are important, they can be derived by a two-step process. The first step creates an indexed sum table to count the frequency of input values; the second uses a *maximum* table to select the most popular. The steps are separate Sawzall jobs and the first step must run to completion before the second step can commence. Although a little clumsy, chaining is an effective way to extend the data analysis tasks that can be expressed in Sawzall.

9. More examples

Here is another complete example that illustrates how Sawzall is used in practice. It processes a web document repository to answer the question: for each web domain, which page has the highest PageRank [14]? Roughly speaking, which is the most linked-to page?

```
proto "document.proto"

max_pagerank_url:
  table maximum(1) [domain: string] of url: string
    weight pagerank: int;

doc: Document = input;

emit max_pagerank_url[domain(doc.url)] <- doc.url
  weight doc.pagerank;
```

The protocol buffer format is defined in the file "document.proto". The table is called `max_pagerank_url` and will record the highest-weighted value emitted for each index. The index is the domain, the value is the URL, and the weight is the document's Page Rank. The program parses the input record and then does a relatively sophisticated emit statement. It calls the library function `domain(doc.url)` to extract the domain of the URL to use as the index, emits the URL itself as the value, and uses the Page Rank for the document as the weight.

When this program is run over the repository, it shows the expected result that for most sites, the most-linked page is `www.site.com` – but there are surprises. The Acrobat download site is the top page for `adobe.com`, while those who link to `banknotes.com` go right to the image gallery and `bangkok-th.com` pulls right to the `Night_Life` page.

Because Sawzall makes it easy to express calculations like this, the program is nice and short. Even using MapReduce, the equivalent straight C++ program is about a hundred lines long.

Here is an example with a multiply-indexed aggregator. We wish to look at a set of search query logs and construct a map showing how the queries are distributed around the globe.

```
proto "querylog.proto"

queries_per_degree: table sum[lat: int][lon: int] of int;

log_record: QueryLogProto = input;

loc: Location = locationinfo(log_record.ip);
emit queries_per_degree[int(loc.lat)][int(loc.lon)] <- 1;
```

The program is straightforward. We import the DDL for the query logs, declare a table indexed by integer latitude and longitude and extract the query from the log. Then we use a built-in function that looks up the incoming IP address in a database to recover the location of the requesting machine (probably its ISP), and then count 1 for the appropriate latitude/longitude bucket. The expression `int(loc.lat)` converts `loc.lat`, a `float`, to an integer, thereby truncating it to the degree and making it suitable as an index value. For a higher-resolution map, a more sophisticated calculation would be required.

The output of this program is an array of values suitable for creating a map, as in Fig. 4.

10. Execution model

At the statement level, Sawzall is an ordinary-looking language, but from a higher perspective it has several unusual properties, all serving the goal of enabling parallelism.



Fig. 4. Query distribution.

The most important, of course, is that it runs on one record at a time. This means it has no memory of other records it has processed (except through values emitted to the aggregators, outside the language). It is routine for a Sawzall job to be executing on a thousand machines simultaneously, yet the system requires no explicit communication between those machines. The only communication is from the Sawzall executions to the downstream aggregators.

To reinforce this point, consider the problem of counting input records. As we saw before, this program,

```
count: table sum of int;
emit count <- 1;
```

will achieve the job. By contrast, consider this erroneous program, which superficially looks like it should work:

```
count: int = 0;
count++;
```

This program fails to count the records because, *for each record*, it sets `count` to zero, increments it, and throws the result away. Running on many machines in parallel, it will throw away all the information with great efficiency.

The Sawzall program is reset to the initial state at the beginning of processing for each record. Correspondingly, after processing a record and emitting all relevant data, any resources consumed during execution – variables, temporaries, etc. – can be discarded. Sawzall therefore uses an arena allocator [11], resetting the arena to the initial state after each record is processed.

Sometimes it is useful to do non-trivial initialization before beginning execution. For instance, one might want to create a large array or map to be queried when analyzing each record. To avoid doing such initialization for every record, Sawzall has a declaration keyword `static` that asserts that the variable is to be initialized once (per runtime instance) and considered part of the initial runtime state for processing each record. Here is a trivial example:

```
static CJK: map[string] of string = {
    "zh": "Chinese",
    "ja": "Japanese",
    "ko": "Korean",
};
```

The CJK variable will be created during initialization and its value stored permanently as part of the initial state for processing each record.

The language has no reference types; it has pure value semantics. Even arrays and maps behave as values. (The implementation uses copy-on-write using reference counting to make this efficient in most cases.) Although in general this can lead to some awkwardness – to modify an array in a function, the function must return the array – for the typical Sawzall program the issue doesn't arise. In return, there is the possibility to parallelize the processing within a given record without the need for fine-grained synchronization or the fear of aliasing, an opportunity the implementation does not yet exploit.

11. Domain-specific properties of the language

Sawzall has a number of properties selected specifically for the problem domain in which it operates. Some have already been discussed; this section examines a few others.

First, atypical of most such “little languages” [2], Sawzall is statically typed. The main reason is dependability. Sawzall programs can consume hours, even months, of CPU time in a single run, and a late-arising dynamic type error can be expensive. There are other, less obvious reasons too. It helps the implementation of the aggregators to have their type fixed when they are created. A similar argument applies to the parsing of the input protocol buffers; it helps to know the expected input type precisely. Also, it is likely that overall performance of the interpreter is improved by avoiding dynamic type-checking at run time, although we have not attempted to verify this claim. Finally, compile-time type checking and the absence of automatic conversions require the programmer to be explicit about type conversions. The only exception is variable initialization, but in that case the type is still explicit and the program remains type-safe.

Static typing guarantees that the types of variables are always known, while permitting convenience at initialization time. It is helpful that initializations like

```
t: time = "Apr 1 12:00:00 PST 2005";
```

are easy to understand, yet also type-safe.

The set of basic types and their properties are also somewhat domain-dependent. The presence of time as a basic type is motivated by the handling of time stamps from log records; it is luxurious to have a language that handles Daylight Savings Time correctly. More important (but less unusual these days), the language defines a Unicode representation for strings, yet handles a variety of external character encodings.

Two novel features of the language grew out of the demands of analyzing large data sets: the handling of undefined values, and logical quantifiers. The next two sections describe these in detail.

11.1. Undefined values

Sawzall does not provide any form of exception processing. Instead, it has the notion of *undefined values*, a representation of erroneous or indeterminate results including division by zero, conversion errors, I/O problems, and so on. If the program attempts to read an undefined value outside of an initialization, it will crash and provide a report of the failure.

A predicate, `def()`, can be used to test if a value is defined; it returns `true` for a defined value and `false` for an undefined value. The idiom for its use is the following:

```
v: Value = maybe_undefined();
if (def(v)) {
    calculation_using(v);
}
```

Here's an example that must handle undefined values. Let's extend the query-mapping program by adding a time axis to the data. The original program used the function `locationinfo()` to discover the location of an IP address using an external database. That program was unsafe because it would fail if the IP address could not be found in the database. In that case, `locationinfo()` would return an undefined value, but we can protect against that using a `def()` predicate.

Here is the expanded, robust, program:

```
proto "querylog.proto"

static RESOLUTION: int = 5; # minutes; must be divisor of 60

log_record: QueryLogProto = input;

queries_per_degree: table sum[t: time][lat: int][lon: int] of int;

loc: Location = locationinfo(log_record.ip);
if (def(loc)) {
    t: time = log_record.time_usec;
    m: int = minuteof(t); # within the hour
    m = m - m % RESOLUTION;
    t = truncctohour(t) + time(m * int(MINUTE));
    emit queries_per_degree[t][int(loc.lat)][int(loc.lon)] <- 1;
}
```

(Notice that we just disregard the record if it has no known location; this seems a fine way to handle it.) The calculation in the `if` statement uses some built-in functions (and the built-in constant `MINUTE`) to truncate the microsecond-resolution time stamp of the record to the boundary of a 5-minute time bucket.

Given query log records spanning an extended period, this program will extract data suitable for constructing a movie showing how activity changes over time.²

A careful programmer will use `def()` to guard against the usual errors that can arise, but sometimes the error syndrome can be so bizarre that a programmer would never think of it. If a program is processing a terabyte of data, there may be some records that contain surprises; often the quality of the data set may be outside the control of the person doing the analysis, or contain occasional unusual values beyond the scope of the current analysis. Particularly for the sort of processing for which Sawzall is used, it will often be perfectly safe just to disregard such erroneous values.

Sawzall therefore provides a mode, set by a run-time flag, that changes the default behavior of undefined values. Normally, using an undefined value (other than in an initialization or `def()` test) will terminate the program with an error report. When the run-time flag is set, however, Sawzall simply elides all statements that depend on the undefined value. To be precise, if a statement's computation references the undefined value in any way, the statement is skipped and execution is resumed after that statement. Simple statements are just dropped. An `if` statement with an undefined condition will be dropped completely. If a loop's condition becomes undefined while the loop is active, the loop is terminated but the results of previous iterations are unaffected.

For the corrupted record, it's as though the elements of the calculation that depended on the bad value were temporarily removed from the program. Any time such an elision occurs, the run-time will record the circumstances in a special pre-defined `collection` table that serves as a log of these errors. When the run completes the user may check whether the error rate was low enough to accept the results that remain.

One way to use the flag is to debug with it off – otherwise bugs will vanish – but when it comes time to run in production on a large data set, turn the flag on so that the program won't be broken by crazy data.

This is an unusual way to treat errors, but it is very convenient in practice. The idea is related to some independent work by Rinard et al. [15] in which the `gcc` C compiler was modified to generate code that protected against certain

²The movie can be seen at <http://labs.google.com/papers/sawzall.html>.

errors. For example, if a program indexes off the end of an array, the generated code will make up values and the program can continue obliviously. This peculiar behavior has the empirical property of making programs like web servers much more robust against failure, even in the face of malicious attacks. The handling of undefined values in Sawzall adds a similar level of robustness.

11.2. Quantifiers

Although Sawzall operates on individual records, those records sometimes contain arrays or other structures that must be analyzed as a unit to discover some property. Is there an array element with this value? Do all the values satisfy some condition? To make these ideas easy to express, Sawzall provides *logical quantifiers*, a special notation analogous to the “for each”, “for any”, and “for all” quantifiers of mathematics.

Within a special construct, called a *when* statement, one defines a quantifier, a variable, and a boolean condition using the variable. If the condition is satisfied, the associated statement is executed.

Quantifier variables are declared like regular variables, but the base type (usually `int`) is prefixed by a keyword specifying the form of quantifier. For example, given an array `a`, the statement

```
when (i: some int; B(a[i]))
    F(i);
```

executes `F(i)` if and only if, for *some* value of `i`, the boolean condition `B(a[i])` is true. When `F(i)` is invoked, `i` will be bound to the value that satisfies the condition.

There are three quantifier types: *some*, which executes the statement if the condition is true for any value (if more than one satisfies, an arbitrary choice is made); *each*, which executes the statement for all the values that satisfy the condition; and *all*, which executes the statement if the condition is true for all valid values of the quantifier (and does not bind the variable in the statement).

Sawzall analyzes the conditional expression to discover how to limit the range of evaluation of a *when* statement. It is a compile-time error if the condition does not provide enough information to construct a reasonable evaluator for the quantifier. In the example above, the quantifier variable `i` is used as an index in an array expression `a[i]`, which is sufficient to define the range. For more complex conditions, the system uses the `def()` operator internally to explore the bounds safely if necessary. (Indexing beyond the bounds of an array results in an undefined value.) Consider for example scanning a sparse multidimensional array `a[i][j]`. In such an expression, the two quantifier variables `i` and `j` must pairwise define a valid entry and the implementation can use `def` to explore the matrix safely.

When statements may contain multiple quantifier variables, which in general can introduce ambiguities of logic programming [6]. Sawzall defines that if multiple variables exist, they will be bound and evaluated in the order declared. This ordering, combined with the restricted set of quantifiers, is sufficient to eliminate the ambiguities.

Here are a couple of examples. The first tests whether two arrays share a common element:

```
when (i, j: some int; a1[i] == a2[j]) {
    ...
}
```

The second expands on this. Using array slicing, indicated by `:` notation in array indices, it tests whether two arrays share a common subarray of three or more elements:

```
when (i0, i1, j0, j1: some int; a[i0:i1] == b[j0:j1] && i1 >= i0+3) {
    ...
}
```

It is convenient not to write all the code to handle the edge conditions in such a test. Even if the arrays are shorter than three elements, this statement will function correctly; the evaluation of *when* statements guarantees safe execution.

In principle, the evaluation of a *when* statement is parallelizable, but we have not explored this option yet.

Table 1

Microbenchmarks. The first is a Mandelbrot set calculation: 500×500 pixel image with 500 iterations per pixel maximum. The second uses a recursive function to calculate the first 35 Fibonacci numbers

	Sawzall	Python	Ruby	Perl
Mandelbrot run time	12.09 s	45.42 s	73.59 s	38.68 s
factor	1.00	3.75	6.09	3.20
Fibonacci run time	11.73 s	38.12 s	47.18 s	75.73 s
factor	1.00	3.24	4.02	6.46

12. Performance

Although Sawzall is interpreted, that is rarely the limiting factor in its performance. Most Sawzall jobs do very little processing per record and are therefore I/O bound; for most of the rest, the CPU spends the majority of its time in various run-time operations such as parsing protocol buffers.

Nevertheless, to compare the single-CPU speed of the Sawzall interpreter with that of other interpreted languages, we wrote a couple of microbenchmarks. The first computes pixel values for displaying the Mandelbrot set, to measure basic arithmetic and loop performance. The second measures function invocation using a recursive function to calculate the first 35 Fibonacci numbers. We ran the tests on a 2.8 GHz x86 desktop machine. The results, shown in Table 1, demonstrate Sawzall is significantly faster than Python, Ruby, or Perl, at least for these microbenchmarks. (We are somewhat surprised by this; the other languages are older and we would expect them to have finely tuned implementations.) On the other hand, for these microbenchmarks Sawzall is about 1.6 times slower than interpreted Java, 21 times slower than compiled Java, and 51 times slower than compiled C++.³

The key performance metric for the system is not the single-machine speed but how the speed scales as we add machines when processing a large data set. We took a 450 GB sample of compressed query log data and ran a Sawzall program over it to count the occurrences of certain words. The core of the program looked schematically like this:

```
result: table sum[key: string][month: int][day: int] of int;
static keywords: array of string =
  { "hitchhiker", "benedict", "vytorin",
    "itanium", "aardvark" };

querywords: array of string = words_from_query();
month: int = month_of_query();
day: int = day_of_query();

when (i: each int; j: some int; querywords[i] == keywords[j])
  emit result[keywords[j]][month][day] <- 1;
```

We ran the test on sets of machines varying from 50 2.4 GHz Xeon computers to 600. The timing numbers are graphed in Fig. 5. At 600 machines, the aggregate throughput was 1.06 GB/s of compressed data or about 3.2 GB/s of raw input. If scaling were perfect, performance would be proportional to the number of machines, that is, adding one machine would contribute one machine's worth of throughput. In our test, the effect is to contribute 0.98 machines.

13. Why a new language?

Why put a language above MapReduce? MapReduce is very effective; what's missing? And why a *new* language? Why not just attach an existing language such as Python to MapReduce?

³The Java implementation was Sun's client Java™ 1.3.1_02 with HotSpot™; C++ was gcc 3.2.2.

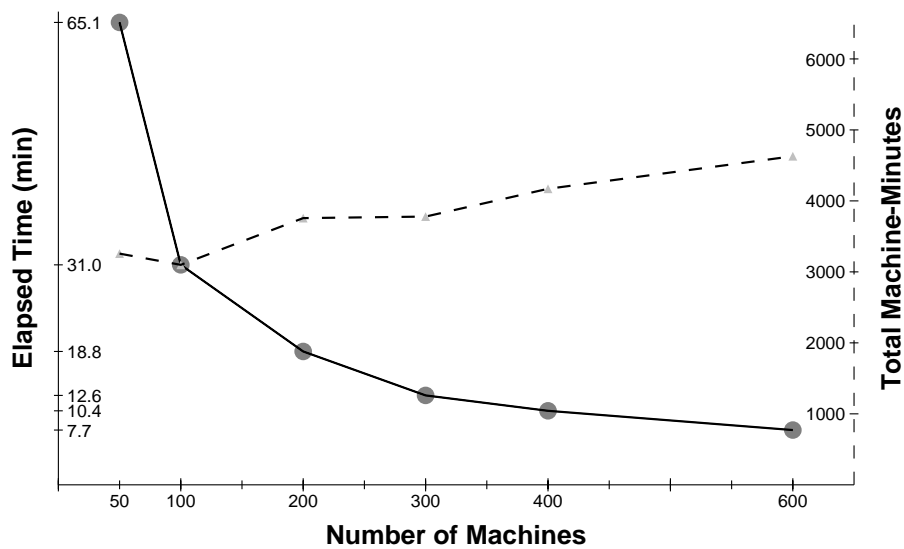


Fig. 5. Performance scales well as we add machines. The solid line is elapsed time; the dashed line is the product of machines and elapsed time. Over the range of 50 to 600 machines, the machine-minutes product degrades only 30%.

The usual reasons for creating a special-purpose language apply here. Notation customized to a particular problem domain can make programs clearer, more compact, and more expressive. Capturing the aggregators in the language (and its environment) means that the programmer never has to provide one, unlike when using MapReduce. It also has led to some elegant programs as well as a comfortable way to think about data processing problems in large distributed data sets. Also, support for protocol buffers and other domain-specific types simplifies programming at a lower level. Overall, Sawzall programs tend to be around 10 to 20 times shorter than the equivalent MapReduce programs in C++ and significantly easier to write.

Other advantages of a custom language include the ability to add domain-specific features, custom debugging and profiling interfaces, and so on.

The original motivation, however, was completely different: parallelism. Separating out the aggregators and providing no other inter-record analysis maximizes the opportunity to distribute processing across records. It also provides a model for distributed processing, which in turn encourages users to think about the problem in a different light. In an existing language such as Awk [12] or Python [1], users would be tempted to write the aggregators in that language, which would be difficult to parallelize. Even if one provided a clean interface and library for aggregators in these languages, seasoned users would want to roll their own sometimes, which could introduce dramatic performance problems.

The model that Sawzall provides has proven valuable. Although some problems, such as database joins, are poor fits to the model, most of the processing we do on large data sets fits well and the benefit in notation, convenience, and expressiveness has made Sawzall a popular language at Google.

One unexpected benefit of the system arose from the constraints the programming model places on the user. Since the data flow from the input records to the Sawzall program is so well structured, it was easy to adapt it to provide fine-grained access control to individual fields within records. The system can automatically and securely wrap the user's program with a layer, itself implemented in Sawzall, that elides any sensitive fields. For instance, production engineers can be granted access to performance and monitoring information without exposing any traffic data. The details are outside the scope of this paper.

14. Utility

Although it has been deployed for only about 18 months, Sawzall has become one of the most widely used programming languages at Google. There are over two thousand Sawzall source files registered in our source

code control system and a growing collection of libraries to aid the processing of various special-purpose data sets. Most Sawzall programs are of modest size, but the largest are several thousand lines long and generate dozens of multi-dimensional tables in a single run.

One measure of Sawzall's utility is how much data processing it does. We monitored its use during the month of March 2005. During that time, on one dedicated Workqueue cluster with 1500 Xeon CPUs, there were 32,580 Sawzall jobs launched, using an average of 220 machines each. While running those jobs, 18,636 failures occurred (application failure, network outage, system crash, etc.) that triggered rerunning some portion of the job. The jobs read a total of 3.2×10^{15} bytes of data (2.8 PB) and wrote 9.9×10^{12} bytes (9.3 TB) (demonstrating that the term "data reduction" has some resonance). The average job therefore processed about 100 GB. The jobs collectively consumed almost exactly one machine-century.

15. Related work

Traditional data processing is done by storing the information in a relational database and processing it with SQL queries. Our system has many differences. First, the data sets are usually too large to fit in a relational database; files are processed *in situ* rather than being imported into a database server. Also, there are no pre-computed tables or indices; instead the purpose of the system is to construct *ad hoc* tables and indices appropriate to the computation. Although the aggregators are fixed, the presence of a procedural language to process the data enables complex, sophisticated analyses.

Sawzall is very different from SQL, combining a fairly traditional procedural language with an interface to efficient aggregators that process the results of the analysis applied to each record. SQL is excellent at database join operations, while Sawzall is not. On the other hand, Sawzall can be run on a thousand or more machines in parallel to process enormous data sets.

Brook [3] is another language for data processing, specifically graphical image processing. Although different in application area, like Sawzall it enables parallelism through a one-element-at-a-time computation model and associative reduction kernels.

A different approach to the processing of large data stores is to analyze them with a data stream model. Such systems process the data as it flows in, and their operators are dependent on the order of the input records. For example, Aurora [4] is a stream processing system that supports a (potentially large) set of standing queries on streams of data. Analogous to Sawzall's predefinition of its aggregators, Aurora provides a small, fixed set of operators, although two of them are escapes to user-defined functions. These operators can be composed to create more interesting queries. Unlike Sawzall, some Aurora operators work on a contiguous sequence, or window, of input values. Aurora only keeps a limited amount of data on hand, and is not designed for querying large archival stores. There is a facility to add new queries to the system, but they only operate on the recent past. Aurora's efficiency comes from a carefully designed run-time system and a query optimizer, rather than Sawzall's brute force parallel style.

Another stream processing system, Hancock [7], goes further and provides extensive support for storing per-query intermediate state. This is quite a contrast to Sawzall, which deliberately reverts to its initialization state after each input record. Like Aurora, Hancock concentrates on efficient operation of a single thread instead of massive parallelism.

16. Future work

The throughput of the system is very high when used as intended, with thousands of machines in parallel. Because the language is modest in scope, the programs written in it tend to be small and are usually I/O bound. Therefore, although it is an interpreted language, the implementation is fast enough most of the time. Still, some of the larger or more complex analyses would be helped by more aggressive compilation, perhaps to native machine code. The compiler would run once per machine and then apply the accelerated binary to each input record. Exploratory work on such a compiler is now underway.

Occasionally a program needs to query external databases when processing a record. While support is already provided for a common set of small databases, such as the IP location information, our system could profit from an interface to query an external database. Since Sawzall processes each record independently, the system could suspend processing of one record when the program makes such a query, and continue when the query is completed. There are obvious opportunities for parallelism in this design.

Our analyses sometimes require multiple passes over the data, either multiple Sawzall passes or Sawzall passes followed by processing by another system, such as a traditional database or a program written in a general-purpose programming language. Since the language does not directly support such “chaining”, these multi-pass programs can be awkward to express in Sawzall. Language extensions are being developed that will make it easy to express the chaining of multiple passes, as are aggregator extensions allowing output to be delivered directly to external systems.

Some analyses join data from multiple input sources, often after a Sawzall preprocessing step or two. Joining is supported, but in general requires extra chaining steps. More direct support of join operations would simplify these programs.

A more radical system model would eliminate the batch-processing mode entirely. It would be convenient for tasks such as performance monitoring to have the input be fed continuously to the Sawzall program, with the aggregators keeping up with the data flow. The aggregators would be maintained in some on-line server that could be queried at any time to discover the current value of any table or table entry. This model is similar to the streaming database work [4,7] and was in fact the original thinking behind the system. However, before much implementation had been completed, the MapReduce library was created by Dean and Ghemawat and it proved so effective that the continuous system was never built. We hope to return to it one day.

17. Conclusions

When the problem is large enough, a new approach may be necessary. To make effective use of large computing clusters in the analysis of large data sets, it is helpful to restrict the programming model to guarantee high parallelism. The trick is to do so without unduly limiting the expressiveness of the model.

Our approach includes a new programming language called Sawzall. The language helps capture the programming model by forcing the programmer to think one record at a time, while providing an expressive interface to a novel set of aggregators that capture many common data processing and data reduction problems. In return for learning a new language, the user is rewarded by the ability to write short, clear programs that are guaranteed to work well on thousands of machines in parallel. Ironically – but vitally – the user needs to know nothing about parallel programming; the language and the underlying system take care of all the details.

It may seem paradoxical to use an interpreted language in a high-throughput environment, but we have found that the CPU time is rarely the limiting factor; the expressibility of the language means that most programs are small and spend most of their time in I/O and native run-time code. Moreover, the flexibility of an interpreted implementation has been helpful, both in ease of experimentation at the linguistic level and in allowing us to explore ways to distribute the calculation across many machines.

Perhaps the ultimate test of our system is scalability. We find linear growth in performance as we add machines, with a slope near unity. Big data sets need lots of machines; it’s gratifying that lots of machines can translate into big throughput.

Acknowledgements

Geeta Chaudhry wrote the first significant Sawzall programs and gave invaluable feedback. Amit Patel, Paul Haahr and Greg Rae were enthusiastic and helpful early users. Paul Haahr created the Page Rank example. Dick Sites and Renée French helped with the figures. We received helpful comments about the paper from Dan Bentley, Dave Hanson, Patrick Jordan, John Lamping, Dick Sites, Tom Szymanski, Deborah A. Wallach, and Lawrence You.

References

- [1] D.M. Beazley, *Python Essential Reference*, New Riders, Indianapolis, 2000.
- [2] J. Bentley, Programming pearls, *CACM August* **29**(8) (1986), 711–721.
- [3] I. Buck et al., *Brook for GPUs: Stream Computing on Graphics Hardware*, Proc. SIGGRAPH, Los Angeles, 2004.
- [4] D. Carney et al., Monitoring Streams – A New Class of Data Management Applications, Brown Computer Science Technical Report TR-CS-02-04. At http://www.cs.brown.edu/research/aurora/aurora_tr.pdf.
- [5] M. Charikar, K. Chen and M. Farach-Colton, *Finding frequent items in data streams*, Proc 29th Intl. Colloq. on Automata, Languages and Programming, 2002.
- [6] W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, Springer, 1994.
- [7] Cortes et al., *Hancock: A Language for Extracting Signatures from Data Streams*, Proc. Sixth International Conference on Knowledge Discovery and Data Mining, Boston, 2000, 9–17.
- [8] J. Dean and S. Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, Proc 6th Symposium on Operating Systems Design and Implementation, San Francisco, 2004, 137–149.
- [9] S. Ghemawat, H. Gobioff and S.-T. Leung, *The Google File System*, Proc. 19th Symposium on Operating System Principles, Lake George, New York, 2003, 29–43.
- [10] M. Greenwald and S. Khanna, *Space-efficient online computation of quantile summaries*, Proc. SIGMOD, Santa Barbara, CA, May 2001, 58–66.
- [11] D.R. Hanson, Fast allocation and deallocation of memory based on object lifetimes, *Software – Practice and Experience* **20**(1) (January 1990), 512.
- [12] B. Kernighan, P. Weinberger and A. Aho, *The AWK Programming Language*, Addison-Wesley, Massachusetts, 1988.
- [13] B. Kernighan, Personal communication.
- [14] L. Page, S. Brin, R. Motwani and T. Winograd, *The pagerank citation algorithm: bringing order to the web*, Proc. of the Seventh conference on the World Wide Web, Brisbane, Australia, April 1998.
- [15] M. Rinard et al., *Enhancing Server Reliability Through Failure-Oblivious Computing*, Proc. Sixth Symposium on Operating Systems Design and Implementation, San Francisco, 2004, 303–316.
- [16] D. Thain, T. Tannenbaum and M. Livny, *Distributed computing in practice: The Condor experience*, Concurrency and Computation: Practice and Experience, 2004.

Copyright of Scientific Programming is the property of IOS Press and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.