International Conference on Cloud Computing and Big Data 2015

Beyond Big Data—Rethinking Programming Languages for Non-Persistent Data

Milind Kulkarni and Yung-Hsiang Lu
School of Electrical and Computer Engineering, Purdue University
West Lafayette, Indiana 47907-2035, USA
milind@purdue.edu and yunglu@purdue.edu

***Abstract*—In "Big Data" research, the data acquired from many sources are fused and analyzed to obtain valuable and sometimes unexpected information. Even though the volumes of data are unprecedented, the data are usually stored for post-experiment analysis and for sharing among scientists. Typical scenarios implicitly assume that the data are stored and can be re-analyzed later. The reality is, unfortunately, not so ideal because the data may be "non-persistent" and allow only one-time use. We propose to reformulate how big data programs are developed, and introduce the notion of data-carrying programs that are, in a sense, self-validating. By writing these programs in a specially-defined language, and transforming them to store sample data, programs can save enough data to provide high-confidence validation of their results.**

Keywords–big data, sampling, non-persistent data, programming language.

I. Non-Persistent Data (NPD)

A spy movie usually starts with a high-tech gadget giving an agent a secret mission. The gadget self-destructs after one-time use, leaving no evidence connecting the mission to any high-level government official. While this movie plot may seem far-fetched, one-time use data are already around us.

Big data is one of the hottest topics in recent years. The data acquired from many sources are fused and analyzed to obtain valuable and sometimes unexpected information, from customer preference to healthcare, from physics to biology. Plenty of stories are available about the rapidly growing data: petabytes of data are generated from experiments in physics, biology, medicine, etc. Even though the volumes of data are unprecedented, the data are usually stored for post-experiment analysis and for sharing among scientists. Typical scenarios in big data analytics implicitly assume that the data are stored and can be re-analyzed later if desired. Cloud computing, with the seemingly unlimited computing and storage resources, is considered the solution for many problems related to big data. The reality is, unfortunately, not so ideal because the data may be "non-persistent" (i.e., transient) for various reasons. Analyzing non-persistent data (NPD) requires rethinking about how analysis programs are written. Why would data be non-persistent? What are the examples of non-persistent data?

- The data volumes are too high and the data archives have no obvious value. Many countries deploy traffic cameras to visually monitor congestion. The information is valuable for travellers when they are on the roads but the values from obsolete data are unclear. Most travellers would like to know whether the roads they are about to travel are congested. Few travellers would be interested knowing whether the same roads were congested one week earlier. As a result, the data from traffic cameras are usually not archived.
- Another reason is the cost. Storing large amount of data would require significant amounts of resources. For surveillance cameras, the common practice is to store the data for a short duration (say two weeks). If no event (such as a crime) is reported, the archived data are erased and the storage capacity is reused.
- Yet another reason of non-persistent data is that analyzing large amounts data is computationally intense. Even if the data are archived, the high computation demand makes analyzing the entire archive, or a significant portion, prohibitively expensive (in terms of time as well as money). For pay-per-use cloud computing, the cost to rerun analysis programs can be a significant limit. Thus, the data are practically non-persistent. To put this in another way, non-persistent data can be processed only once and then they are lost forever because reprocessing the data is infeasible or uneconomic.
- Internet of Things (IoT) may collect and process large amounts of data. These things do not have enough storage space to archive all the data. Transmitting data require resources: energy, radio spectrum, etc. Thus, these things may never transmit all the sensed data.

- Re-analyzing the data may take too long and decisions must be made quickly. Imagine that the data from sensors suggest imminent natural disasters (such as earthquakes or volcanic eruptions). Government officials must decide whether to evacuate residents. Even though the data are archived and re-analysis is feasible, the decisions must be made before re-analysis can complete.
- Even if all the data are save and there is plenty of time analyzing the data, the data may still be "lost" because the data or the analysis programs cannot be easily located.

Due to the reasons mentioned above, many sources of data can be considered as non-persistent. The question is whether it would be possible to analyze non-persistent data and find valuable information with some degrees of confidence.

NPD may be streaming data (generated continuously) or archived data. If the streaming data are not saved, analysis must occur while the data are produced. For archived data, since the data can be processed only once, the data are equivalent to a stream. Existing programming languages' internal data represent programs' states and are inappropriate for processing non-persistent data. Let us consider the common approach for analyzing data. A team of researchers develops a program and tests the program on a small sample of the data. When they are confident that the program "works", the program is used to analyze the much larger volume and possibly non-persistent data. This is not a satisfactory solution for "big data" because the potential of big data analytics is the possibility of discovering something unexpected, something unavailable on a smaller set of sample data. If all the relevant information were already in the smaller dataset, it would no longer be necessary analyzing the larger dataset. If the information is absent in the smaller dataset, how could anyone write a program and confidently claim that the program could find anything new in the larger set of data? We believe that the problem lies in the current design of programming languages. *Existing languages are not designed to deal with non-persistent data. In other words, new programming languages are needed.*

II. Existing Work

Two well-known large-scale data processing languages are MapReduce [8] and Dryad [10]. Both allow programmers to develop analysis pipelines consisting of multiple stages, each of which applies a different type of processing to data. Both require programmers to write operations for each stage using low-level languages such as Java. To ease the challenge of writing programs in systems like MapReduce and Dryad, higher level languages are introduced to allow applications to express using SQL-like constructs compiled down into MapReduce or Dryad pipelines. Examples of such languages include Pig Latin [12], DryadLINQ [15], Sawzall [13] and GLADE [6]. Most of them focus on *batch processing* programs, where all of the data is available at the beginning of computation. This makes them ill-suited for online analysis problems, where data sources produce *streams* of data.

Languages like Hadoop-online [7], SPC [4] and Apache Storm [1] used MapReduce-like strategies to tackle streaming data. While streaming data has some similarity to NPD (in both cases, the data is not stored), we note that in general none of these languages tackles the specific problems of NPD: how can one gain any confidence that the results of a program are valid?

There has been substantial interest in using sampling techniques to more efficiently perform computations. In the databases community, Aqua [2], STRAT [5], and BlinkDB [3] support *approximate queries*, where an SQL-like query can be answered approximately, with some confidence. These languages target a restricted class of problems (database queries), and are not suited for more complex data analysis tasks. Moreover, they often rely on preprocessing a large, existing database; they are not designed to handle NPD or provide support for streaming or validation. SciBORQ [14] and Olston et al. [11] describe support for sample-based data analysis, while ApproxHadoop [9] is a system for performing approximate map-reduce queries. These schemes do not provide support for NPD, nor do they support operations on streaming data.

III. Programming Languages for Processing Non-Persistent Data

A. External Data, Internal Data, and Sampled Data

Before explaining the reason and a conceptual solution, let us examine the current design of programming languages. The relationships between processing and data have always been the center of programming languages. For procedural languages, the relationships are usually implicit: programs have functions that process the data. The code is usually stateless responding to the contents of the data from the inputs. The relationship between processing and data is implicitly enforced by the data types. Object-oriented languages internalize some data, called objects' *attributes*, so that the behavior of the code depends on the attributes' values. Even though in most cases the attributes' values are directly affected by the data to be processed, the attributes are not the data to be processed. Instead, the attributes are essentially the programs' states. The attributes may not directly represent the data. We call the attributes this program's *internal* data. Figure 1 illustrates how

today's programs are written: the data are *external* to the programs.

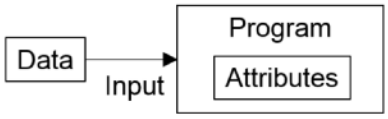

Fig. 1. A typical object-oriented program reads *external data* from an Input interfaces, processes the data, and stores the information as internal data in the form of attributes.

To explain this further, consider a program whose sole purpose is to determine the content in an input device that contains infinite numbers of x or y. This program can read one letter at a time and reports the occurrences of x and y. A simple program would have the following structure:

```
initialize xcount to zero;
initialize ycount to zero;
while True do
    read oneletter from Input;
    if oneletter is x then
        increment xcount;
    else
        increment ycount;
    end
end
```

This program has two counters to store the numbers of occurrences of x and y so far. For simplicity, we do not consider overflow of the counters. The counters are the program's internal data. The letter from Input is the external data. The program's internal data (*oneletter* and the two counters) are directly affected by the data to be processed (reading from Input) but there is no guarantee that the counters are updated correctly. In particular, *oneletter* is reused and the previously read letter (the external data) is lost in the next iteration.

If the line between **else** and **end** increments xcount (should be ycount), the program is wrong. The question is how to discover this mistake. The common approach is to test this program using a finite set of input that contains known numbers of x and y. If the program's output is unexpected, the program is inspected carefully with the hope to identify the mistake. The difficulty is that the test input must contain x and y (and nothing else). The program is incorrect if the real, non-persistent, data contains x, y, and z. The program will increment ycount when the input is z. Obviously, this is an oversimplified example. A real program would be much more complex and finding the mistake would not be so easy. When processing non-persistent data for discovering unexpected information, unfortunately, there may be no luxury of using smaller sets of data for testing.

For non-persistent data, one approach to validating programs' correctness is to rely on samples created while processing the data. What does this mean? It means that the same program should create the sample data while processing the data. In addition to maintaining the program's states, the program needs to store the samples. Consider the following conceptual design. Instead of discarding the non-persistent external data, this program takes samples while processing the data. To validate this program's correctness, it uses the same procedure to process the sampled data and the complete data. Since the input data can be infinitely long, the sample data will also grow indefinitely. Thus, the program must discard the sample data from time to time. The following code snippet describes this concept. The gray-background code is generated by the compiler. Please notice that the **if** - **else** - **end** section in the gray-background has the identical structure to the original program, except the internal variables (xsamplecount and ysamplecount) are different.

```
initialize xcount to zero;
initialize ycount to zero;
initialize xsamplecount to zero;
initialize ysamplecount to zero;
while True do
    read oneletter from Input;
    // save sample data and the results
    if takesample is true then
        store oneletter as samples;
        if oneletter is x then
            increment xsamplecount;
        else
            increment ysamplecount;
        end
    end
    // original program
    if oneletter is x then
        increment xcount;
    else
        increment ycount;
    end
end
```

We call this program "data-carrying" because it carries sample data during execution. To illustrate how this program works, consider a section of the input:

x	y	y	x	x	y	x	y	y	y
x	y	x	x	y	x	y	x	y	y

This section of input data contains 20 letters: 9 x and 11 y. Suppose the program samples 7 letters

(gray background). The samples can be used to check whether the program is correct. The sampling code is added by the compiler, not hand-written. This provides a consistent solution for saving samples, transforming *some* of the non-persistent data to persistent.

Many important questions arise for this approach. First, which data should be sampled? This depends on the applications and the data contents. For some applications, random samples may suffice. Periodic sampling may be appropriate for some applications but this is prone to aliasing problems. For some other applications, sampling may be triggered by events detected in the data. Second, how often should the samples be taken? Sampling slows down the program because the sampled data are processed twice. If the sampling rate is too high, the overhead can be unacceptable and the required storage can be costly. If the sampling rate is too low, the samples may not be representative to validate the program.

B. Utility of sampling

One obvious question is, why is sampling data better than using a smaller input? The answer is that the sampled data is constructed *online* from the real NPD. As a result, it will better reflect the characteristics of the NPD, due to the properties of random sampling. If a programmer were to rely on hand-generated smaller input sets, the programmer might mistakenly generate only inputs that contain x and y, and miss the error that arises when the NPD contains z. Please note that we cannot *guarantee* that the sampled data will display the same properties as the original data, sampling from the original data may improve confidence confidence. Also, inadequately covering possible inputs is a common error when building test suites.

Another question is, how does the sampled data help the programmer catch errors? In particular, if the sampled program is automatically generated from the original program, as in our example, then any errors in the original program might be reflected in the sampled program—the sampled program does not correctly account for zs, either. Here, however, we note that the sampled data is much smaller than the NPD—and can be stored. As a result, the sampled program has output that is easier to digest, and, more importantly, the sampled *input* data is easier to analyze. For example, manual inspection of the sampled data could reveal the existence of a z in the input.

Manual inspection is an instance of a more general tactic: because the sampled data is substantially smaller than the original NPD, programmers can use *alternate analyses* to compare the output of the sampled program to the expected output. Consider a complex, optimized, data-mining program that operates over NPD. That same program can be run over sampled NPD and then, because the sampled data is smaller, a simpler, but slower, program that solves the same problem can *also* be run over the sampled data, providing validation of the result, and some confidence that the complex program is doing the right thing when run over NPD.

C. A (Somewhat) Formal Model

To understand the difficulties in generating a sampled version of the program for validation, let us consider the problem a little bit more formally. We can think of the original program as a function f that takes an input data stream I to produce output O:

$$f : I \rightarrow O \tag{1}$$

To validate the result of this program, a validation function v_f considers both the input and the output of f to decide whether the result is correct or not.

$$v_f : I \times O \rightarrow \{\text{true, false}\} \tag{2}$$

The essential problem with the validation of big-data programs is that the input data (and possibly the output data as well) are too large to efficiently validate. When processing non-persistent data, the input data can be seen only once, making the validation problem much more difficult.

Framing the problem in this manner suggests several possible solutions. First, one could imagine online validation within a window of input data: If each output corresponds to a small window of the input stream, then the amount of data that needs to be saved to perform validation is bounded by the window. For example, if, rather than counting the number of x in the entire stream, the analysis program counted the number of x in a 5-item window, validation can be performed by saving some windows of the input, and discarding them once validation is complete.

For many analysis programs, unfortunately, the amount of data required to produce enough output for validation is too large to save. In such cases, we turn to the sampling approach, described above. We may design a sampling function s that transforms the input stream into a much smaller amount of data:

$$s : I \rightarrow I' \text{ and } |I| \gg |I'| \tag{3}$$

This sampled input stream is passed to a transformed program that operates over the sampled data and produces some output:

$$f' : I' \rightarrow O' \tag{4}$$

A new validation function is produced: v'_f validates the transformed program over the sampled data:

$$v'_f : I' \times O' \rightarrow \{\text{true, false}\} \tag{5}$$

The problem of testing validity of a data analysis program can thus be cast as the problem of finding s, f', and v'_f such that:

$$v'_f(s(I), f'(s(I))) \Rightarrow v_f(I, f(I)) \tag{6}$$

In other words, if the transformed program validates, the original program is believed to produce a valid output. Crucially, because the transformed program operates on a smaller (sampled) input and produces a smaller output, the modified validation function can be evaluated even in scenarios where it may be impractical to run the original validation function.

The next question, naturally, is whether f', s and v'_f exist for every program? If we adopt the natural requirement that reduces the size of the input, we can see that even for very simple models of computation, there exist analysis programs that cannot be effectively run in a sampled manner.

It is well-known that proving the equivalence of two Turing machines is unsolvable. Thus, we intend to ask, and informally answer, a simpler question by considering that the function f is a finite automaton that detects the appearance of an event called 'x'. This finite automaton starts at the initial state '1'. If the input stream contains one 'x', the finite automata enters a different state '2'. If the input stream contains no 'x', the finite automata stays at the initial state.

Figure 2 depicts this finite automata. Here, A is the set of input alphabets.

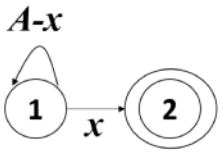

Fig. 2. A state machine detects the presence of x.

This finite automata can be implemented as follows.

```
start at state 1;
while True do
    read oneletter from Input;
    if oneletter is x then
        move to state 2;
    end
end
```

It is conceivable that this program may be incorrectly written as

```
start at state 1;
while True do
    read oneletter from Input;
    if oneletter is x then
        move to state 1;
    end
end
```

If x is a rare event, it is possible that the sample contains no x. The finite automata does not enter state '2' and this is the correct behavior. This correct result cannot be distinguished from the result of the incorrect program.

This example explains that even for a two-state finite automata, sampling cannot help validate the correctness of the program. The implication of this example is profound: it is not possible to debug a finite automaton by testing it using samples when processing non-persistent data. Nevertheless, a more relaxed correctness criterion, which states that the validation function should be correct with high confidence may allow for validation even of this type of program. The above discussion suggests the need to restrict the space of possible programs f, sampling functions s, and input stream characteristics to make progress in this problem. For example, if the analysis program is computing some aggregate measure of the input stream (e.g. the proportion of 1's in a binary input stream), then sampling of the input stream should produce a somewhat similar result, *with higher confidence*.

D. Why Language Support?

Another important question is whether this should be supported by a programming language. We think so. The language support can appropriately restrict the types of operations performed during an analysis program so that (i) the resulting program can be validated using a sampling approach (e.g. it does not allow writing the finite automaton program described above); (ii) the transformed, sampled program can be automatically generated; and (iii) programmers can convey semantic information that informs how the input data should

be sampled and how the analysis program should be transformed. These sorts of restrictions are common in many languages used for analyzing data and these languages allows only operations with certain behaviors, or statistical languages like R, which use knowledge of data types such as vectors to implement analysis operations. The new languages should explicitly support samples from the data, as illustrated in Figure 3.

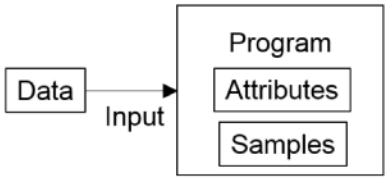

Fig. 3. Programs written in the proposed new language automatically store data samples. Compared with Figure 1, some of the input data are *automatically* sampled by the language and stored with the program's internal data.

IV. Proposed language features

Writing a validation function that satisfies the restrictions described above may be complicated. Given the wide variety of possible analysis programs and validation strategies, we believe a much more promising approach is to rely on code transformations to automatically generate the sampled, validated variant of the code from the original program. To perform these transformations, it is necessary to identify operations on non-persistent data, to identify sources of non-persistent data, and to analyze the program code to determine which computations should be performed in the sampled variant. All of these tasks are eased through the use of programming language features such as types and special control constructs. Through the use of special types, the programmer can identify non-persistent data. As this data is processed by a program, a type system can identify which computations are being performed over non-persistent data, feeding that information to a compiler that can generate the transformed program. While we do not propose a specific new language in this article, we believe that any language that can tackle the problems identified above must contain several features:

- The language must be able to operate on persistent data, non-persistent data, and sampled data, and be able to distinguish them. The intention is that the original computation operates on original data but it is also possible that the programmer might already perform some sampling as part of the analysis. As a result, later validation may want to take advantage of the existing sampled data. This may require a type system that can track whether data are "original" or "sampled". For sampled data, it needs to provide information about the sampling that was done (e.g. the sampling rate).
- To facilitate the generation of validation functions, the language must have some way of specifying which data is too big and should be subject to sampling. Not all data in the original program needs to be sampled. For example, much of the data might be scalar configuration parameters. Large-scale input data needs to be identified and sampled.
- Validating functions should be automatically generated and the language must have some ways of defining validation functions. Moreover, the validation function can be analyzed to ensure that its computations can use sampled data.

One alternative to introducing language support is to incorporate the necessary functionality for analyzing non-persistent data in a library. However, this approach has some drawbacks. As our example shows, the code to perform the sampled version of the program is intimately tied to the structure of the original code: a different program would lead to a different "sampled version." Moreover, the clear correspondence between the sampled version of the code and the original program means that there is redundancy between the two versions.

How is a data-carrying program different from program annotations that check the program's correctness? They are fundamentally different because annotations are based on the program's internal data. In contrast, the carried data are samples from the external (input) data. Can existing programming languages accomplish the same things? Is a new language really necessary? Can the samples simply be attributes in the programs? To answer these questions, let's review the primary reason of creating a programming language: to unify commonly used features so that programmers can write correct programs more efficiently. An assembly language is Turing-complete, so is Java. However, most people would agree that Java is better for writing complex programs that involve networking and web applications. If data-carrying programs are necessary for many applications, language support would be preferred.

V. Conclusion

In conclusion, we believe that the time is ripe to re-think how big data programs are written. With evermore critical decisions are made on the basis of data analysis programs, it is becoming crucial to *validate* those decisions. The fundamental properties of non-persistent big-data make validation difficult: It is impossible to store enough data to re-run an analysis. We

thus propose to reformulate how big data programs are developed, and introduce the notion of data-carrying programs that are, in a sense, self-validating. By writing these programs in a specially-defined language, and transforming them with compilers that can generate sampling code, programs can automatically save enough data to provide high-confidence validation of their results.

Let us go back to the spy movie mentioned at the beginning of this article. If an agent uses the programming language with automatic sampling to process the data from the self-destructing gadget, the agent may keep enough evidence linking the mission to the high-level government official. The official, with the fear of being identified, orders another agent to destroy the runtime system of the language we propose in this article. A new movie plot has just been created.

ACKNOWLEDGMENT

We want to thank our colleague Edward J Delp for the valuable discussion about non-persistent data.

REFERENCES

[1] Apache storm project. http://storm-project.net.

[2] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The aqua approximate query answering system. In *ACM SIGMOD International Conference on Management of Data*, pages 574–576, 1999.

[3] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *ACM European Conference on Computer Systems*, pages 29–42, 2013.

[4] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani. Spc: A distributed, scalable platform for data mining. In *International Workshop on Data Mining Standards, Services and Platforms*, pages 27–37, 2006.

[5] S. Chaudhuri, G. Das, and V. Narasayya. Optimized stratified sampling for approximate query processing. *ACM Trans. Database Syst.*, 32(2), June 2007.

[6] Y. Cheng, C. Qin, and F. Rusu. Glade: Big data analytics made easy. In *ACM SIGMOD International Conference on Management of Data*, pages 697–700, 2012.

[7] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, and R. Sears. Online aggregation and continuous query support in mapreduce. In *ACM SIGMOD International Conference on Management of Data*, pages 1115–1118, 2010.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, 2004.

[9] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 383–397, 2015.

[10] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, 2007.

[11] C. Olston, E. Bortnikov, K. Elmeleegy, F. Junqueira, and B. Reed. Interactive analysis of web-scale data. In *Conference on Innovative Data Systems Research*, 2009.

[12] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *ACM SIGMOD International Conference on Management of Data*, pages 1099–1110, 2008.

[13] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, 13(4):277–298, Oct. 2005.

[14] L. Sidirourgos, M. Kersten, and P. Boncz. Sciborq: Scientific data management with bounds on runtime and quality. In *Conference on Innovative Data Systems Research*, 2011.

[15] 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 *USENIX Conference on Operating Systems Design and Implementation*, pages 1–14, 2008.