Nondeterminism in MapReduce Considered Harmful?

An Empirical Study on Non-commutative Aggregators in MapReduce Programs

Tian Xiao[1,2] Jiaxing Zhang[2] Hucheng Zhou[2] Zhenyu Guo[2] Sean McDirmid[2]
Wei Lin[3] Wenguang Chen[1] Lidong Zhou[2]

[1]*Tsinghua University, China* [2]*Microsoft Research, China* [3]*Microsoft Bing, USA*

ABSTRACT

The simplicity of MapReduce introduces unique subtleties that cause hard-to-detect bugs; in particular, the unfixed order of reduce function input is a source of nondeterminism that is harmful if the reduce function is not commutative and sensitive to input order. Our extensive study of production MapReduce programs reveals interesting findings on commutativity, nondeterminism, and correctness. Although non-commutative reduce functions lead to five bugs in our sample of well-tested production programs, we surprisingly have found that many non-commutative reduce functions are mostly harmless due to, for example, implicit data properties. These findings are instrumental in advancing our understanding of MapReduce program correctness.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Languages, Reliability

Keywords

MapReduce, nondeterminism, commutativity, bug

1. INTRODUCTION

MapReduce [5] has emerged as the main programming model for data-parallel computation given its simple programming model of mappers and reducers that enable parallel failure-resilient execution on many machines. There is however a significant gulf between a static MapReduce program and its execution when we reason about correctness. For example, it is well-known that nondeterministic user-defined mappers and reducers will produce different results when re-executed in response to failures [5].

Another subtlety important to the correctness of MapReduce programs is *nondeterminism* in data shuffling that occurs between map and reduce stages. When a MapReduce program executes on a cluster of machines, mappers execute concurrently on a set of machines over their data partitions. Keyed data entries produced by mappers are exchanged via data shuffling to machines where reducers run so that data items with the same key are aggregated in a sequence to the same reducer. Due to uncertainties in the number of mappers/reducers, network latency, and scheduling decisions, the order of each sequence is nondeterministic. A reduce function that is not *commutative* might produce different results depending on different sequence orders, which can lead to correctness violations.

This problem has not gone unnoticed by the software engineering research community. For example, Csallner et al. [3] proposes that non-commutative reducers are bugs that can be detected through symbolic execution. In practice, programmers that write reducers are usually not aware of commutativity, and it remains largely unknown if non-commutative reducers are a serious issue to correctness. We have therefore conducted the first ever empirical study on the commutativity of real-world user-defined reducers in production MapReduce-style programs to answer the following questions:

- How pervasive are non-commutative reducers in real-world MapReduce programs?
- How does the output of a non-commutative reducer depend on input order? Are there any common patterns?
- Are non-commutative reducers always harmful? Is it appropriate to flag them as bugs?
- Are there real bugs caused by non-commutative reducers? If so, what do they look like and what is their impact?

Our study has collected 507 distinct custom user-defined reducers found in 13,311 real-world MapReduce-style jobs in our production cluster. We studied reducer code manually to identify those that are non-commutative with findings, summarized in Table 1, that are quite surprising. Non-commutative reducers not only exist, but are pervasive: 58% of the reducers examined are non-commutative. More importantly, our investigation indicates that most of those non-commutative reducers do not lead to correctness issues. Flagging non-commutative reducers as bugs, as proposed in [3], is then likely to create many false positives that will frustrate programmers.

Our further investigation reveals that surprisingly most (88%) of the non-commutative reducers can be categorized into five simple patterns even though they encode a wide variety of algorithms in different coding styles. For some patterns, non-commutative reducers lead to nondeterministic results, but the nondeterminism appears known and tolerated by programmers. For other patterns, non-commutative reducers are guaranteed to produce deterministic results as long as the data they operate on has certain properties.

ICSE '14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2768-8/14/05 ...$15.00.

Table 1: Major findings and their implications.

	Findings	Implications
1	Over half of user-defined reducers (58%) are non-commutative. Many of them are found in well-tested recurring jobs.	It is unnecessarily restrictive to mark all non-commutative reducers as incorrect.
2	Most non-commutative reducers (88%) can be classified into five simple patterns according to how their output depends on input order.	Most non-commutative reducers can be automatically recognized and understood well by a pattern matching approach.
3	48% of the non-commutative reducers would yield deterministic results if certain implicit data properties are satisfied. In most cases, the properties are satisfied by real data.	Non-commutative reducers are harmless if data satisfies certain implicit properties, whose violations can indicate a bug as programmer assumptions are inconsistent with data.
4	26% of the non-commutative reducers have certain patterns such that nondeterminism in their output is very likely to be tolerable.	Non-commutativity is harmless if users are aware of and can tolerate nondeterminism in the results.
5	15% of the non-commutative reducers concatenate input items in an nondeterministic order. However, their results are often treated as a set of items in an order-insensitive way.	Non-commutativity is also harmless if the nondeterministic output of a reducer is processed in such a way in succeeding stages that the final output of the whole program is deterministic.
6	We find five bugs in well-tested production programs caused by non-commutative reducers. Their root causes include wrong assumptions on implicit data properties, data corruption, and various semantic errors leading to non-commutativity.	Non-commutativity remains a source of bugs that can go undetected for a long period of time. Checking implicit properties on real data is an effective way to detect this kind of bugs.

Judging from the structure of those reducer programs, programmers probably assume such properties hold.

We further conducted experiments to study real input data of some non-commutative reducers in order to verify if assumed data properties actually held. We successfully used violation of those data properties to find real bugs in code where reducers produce nondeterministic results that are likely to be unexpected by the programmers; we suggest that this is an effective approach to detecting correctness issues. Finally, we present five real bugs found in non-commutative reducers that manifest undesirable nondeterministic results. All have run in production for over three months before we reported them, showing the subtlety of such nondeterministic bugs and their impact on real production systems.

This paper makes the following contributions:

- We present the first comprehensive study on non-commutative reducers for real-world data-parallel programs. Our study reveals surprising results that are instrumental to understanding the significance of this issue, as well as helping to develop effective detection mechanisms that avoid false positives.

- We discover that programmers often rely on certain implicit data properties to ensure determinism in the output of their non-commutative reducers. Our study on real data shows that such data properties can be wrong and lead to real bugs.

- We present five bugs caused by non-commutative reducers.

The rest part of the paper is organized as follows. Section 2 demonstrates nondeterminism in MapReduce with examples. Section 3 describes our study methodology. Reducer classification with respect to commutativity and common non-commutativity patterns are discussed in Section 4, followed by our data property verification results on real data in Section 5. Section 6 presents five real bugs found in production jobs that produce undesirable nondeterministic results. Section 8 surveys related work and Section 9 concludes.

2. BACKGROUND

A generalized MapReduce program processes data as sets of rows, each row consisting of multiple columns. A simple MapReduce program has a map stage followed by a reduce stage (Figure 1) that are automatically split into mappers and reducers, which are scheduled to run in parallel. Computation on mappers and reducers is usually given by user-defined functions `map` and `reduce`. Mappers filter and transform the input data into a row set, some columns of which are specified as the reduce key. Between the mappers and reducers is a data shuffling stage where rows with the same reduce key are grouped together into a sequence by a merge-sort. Reducers are invoked to aggregate each of these groups. An example `reduce` function written in C#, shown in Figure 2, accepts a sequence of input rows from an iterator (`IEnumerable<Row> input`) and sums up values in the second column (`row[1].Integer`) of rows that are grouped by the first column and passed through the parameter `input`. A single row is emitted for each group on line 8.

The implementation of data shuffling introduces uncertainty in group row order. In Figure 1, rows with the reduce key ‘a’ are initially generated by multiple mappers, and are grouped together

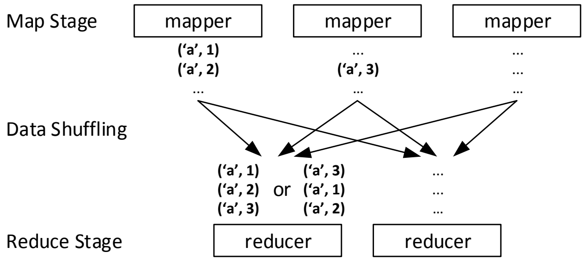

Figure 1: The nondeterminism in MapReduce. The rows with the same reduce key ‘a’ (the first column) are shown. In the data shuffling stage, these rows can be grouped and passed to a reducer in different orders.

```
1 IEnumerable<Row> reduce(IEnumerable<Row> input, Row output) {
2     int sum = 0;
3     foreach (Row row in input) {
4         output[0].Set(row[0].String);
5         sum += row[1].Integer;
6     }
7     output[1].Set(sum);
8     yield return output;
9 }
```

Figure 2: An example `reduce` function that sums up the second column in all rows in a group.

```
IEnumerable<Row> reduce(IEnumerable<Row> input, Row output) {
    int last = 0;
    foreach (Row row in input) {
        output[0].Set(row[0].String);
        last = row[1].Integer;
    }
    output[1].Set(last);
    yield return output;
}
```

Figure 3: An example `reduce` function that is sensitive to its input order.

in an arbitrary order. For reducers doing commutative aggregation, such as in Figure 2, divergence of input order does not matter. However, some reducers are sensitive to the order of their input rows. Figure 3 gives an example simplified from real reducers that returns the last value of the second column in each group so input order is critical to the final result. The author of this reducer may assume that all values in that column for each group are the same where nondeterministic output results if this assumption is wrong.

A reducer is *commutative* if its output remains the same when its input rows are reordered, which is formalized as follows.

DEFINITION 1. *A **commutative** reducer R is a pure function and, for any sequence of input rows $\overline{X}$ and any of its permutation, $\overline{X'}$, $R(\overline{X}) = R(\overline{X'})$ holds.*

A commutative reducer is a pure function [11] that does not depend on any hidden information or state, nor does have side effects. In practice, a reducer can be impure if it calls a random number generator, reads environment variables on the local machine, or maintains stateful variables in class members whose lifecycle spans multiple invocations on different input groups. Impure reducers might produce different results in different runs. We do not consider impure reducers in this study, using *non-commutative reducer* to describe a reducer that is pure but not commutative.

2.1 The SCOPE Language

SCOPE [15] is a SQL-like declarative language with C# user-defined functions, similar to Pig [10] and Hive [12], that is used by many production teams in Microsoft to write MapReduce programs. SCOPE makes it easy to build a MapReduce pipeline through a single script. We briefly describe the SCOPE language through the example in Figure 4. The `EXTRACT` statement (lines 1 to 2) reads the input file on the distributed file system into a *row set* called `R`, each row of which consists of two columns: `Url` of default type `string` and `Score` of type `int`. The `REDUCE` statement (lines 4 to 6) specifies a reduce stage that invokes a user-defined reducer `MyReducer`, like those defined in Figure 2 and Figure 3, to aggregate the row set on the reduce key column `Url`. The resulting row set `S` is then written to the output file by the `OUTPUT` statement (line 8). The reducer can access columns in a row by either indices (`row[1]`) or names (`row["SomeColumnName"]`).

3. METHODOLOGY

We take user-defined reducers in real SCOPE jobs of a production cluster as our study subjects. Many jobs are submitted to this cluster every day, most of which complete successfully. Jobs are developed and submitted by various production teams, including the Bing engineering team, the advertisement team, and several data mining teams. We randomly sampled 13,311 successful jobs from the cluster within two weeks as our sample job set and collected all user-defined reducers from these jobs. Because a large part of our sample job set are recurring jobs that run periodically (mostly daily or hourly), a job and its reducers can be sampled multiple times. To reduce manual work in studying these reducers, we approximately identify duplicates by name. We also exclude reducers that are invoked with a secondary sort. For a `REDUCE` statement with a secondary sort, the runtime engine sorts the rows in each input group on specified non-key columns before invoking the reducer. Though the secondary sort eliminates the uncertainty in input row order partially or completely (depending on whether all columns are specified for sorting), it introduces significant overhead. In our sample job set, we find that the majority of user-defined reducers are invoked without a secondary sort. Finally, we find 507 unique user-defined reducers as our study subjects.

We manually studied all 507 user-defined reducers to understand whether or not each of them was non-commutative. For reducers in pre-compiled dynamic-link libraries, we decompiled them using the open-sourced decompiler ILSpy [7] that can recover source code for most reducers. Although local variable names cannot be recovered, ILSpy has a feature to highlight all appearances of the variable under cursor, greatly helping us track reducer data dependencies.

SCOPE script

```
1 R = EXTRACT Url, Score:int FROM "path/input"
2     USING DefaultTextExtractor;
3
4 S = REDUCE Data ON Url
5     PRODUCE Url, ScoreSum:int
6     USING MyReducer;
7
8 OUTPUT S TO "path/output";
```

Figure 4: An example SCOPE program.

3.1 Threats to Validity

As an empirical study, our study is subject to several validity issues:

Representativeness of applications. The applications in our sample job set are developed by engineers from many different production teams. The reducers also implement a wide variety of algorithms, ranging from simple summation and counting to complex data mining and face recognition algorithms. However, all applications are commercial and differ from open-sourced applications. The major difference we have noticed is that data processed by our jobs usually consists of a much larger number of columns compared to that in open-sourced MapReduce programs such as Mahout [2], probably due to the complexity of large-scale production systems like Bing. As a result, the data processed by our jobs are more likely to include implicit relationships between columns that lead to various non-commutativity patterns as described in Section 4.1. Therefore, we believe that our sample job set represents MapReduce programs for large-scale commercial systems, but may not reflect the characteristics of those working on simpler data.

Threats in framework and language. We conduct our study on applications written solely in SCOPE with C# user-defined functions. However, nondeterminism is a general issue in the MapReduce programming model and also exists in other popular MapReduce implementations such as Hadoop [1], and its high-level abstractions such as Pig and Hive. We do not see any special influence of the C# language in user-defined reducers. All five non-commutativity patterns discussed in Section 4.1 are implemented in straight-forward ways without using any special C# or SCOPE features. However, in contrast to low-level MapReduce implementations such as Hadoop

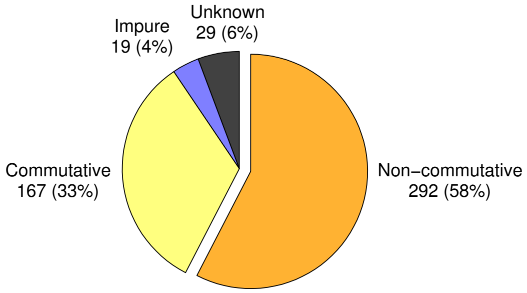

Figure 5: Reducer classification according to Definition 1.

where users must implement every reducer themselves, high-level languages, such as SCOPE, Pig and Hive, provide a number of built-in aggregating reducers, e.g. SUM, COUNT and MAX, that users often reuse in lieu of writing their own. As most built-in aggregators are commutative, the percentage of commutative user-defined reducers would increase for jobs written in low-level frameworks.

Subjectivity in manual study. Manual study is indispensable, including reducer classification with respect to commutativity and recognition of non-commutativity patterns. We minimize subjectivity by being conservative. In both classifications (Figure 5 and Figure 6), we have an "Unknown" or "Other" category for reducers that we do not fully understand or are not sure about. We also use double verification to minimize subjective errors.

4. REDUCER CODE AUDITING

This section presents our study results of 507 user-defined reducers, including reducer classification with respect to commutativity and five patterns that we recognized in non-commutative reducers. Non-commutative reducers are first identified in our sample set, classifying 507 reducers as impure, pure commutative, and pure non-commutative. Figure 5 shows the number of reducers in each category along with an "Unknown" category for reducers that we failed to understand. Each category is described as follows.

Non-commutative. Surprisingly, we find that 58% of the reducers in the sample set are non-commutative. Because many of these reducers are found in recurring jobs that have been well-tested and periodically running for a long time, it is hard to believe that they are all buggy. This finding motivates our further study to understand the intention of programmers in writing non-commutative reducers and how these reducers work with real data.

Finding 1: Over half of user-defined reducers (58%) are non-commutative. Many of them are found in well-tested recurring jobs.
Implication: It is unnecessarily restrictive to mark all non-commutative reducers as incorrect.

Commutative. We find that 167 reducers (33%) in the sample set are commutative. Over a half of these 167 reducers perform simple aggregations on their input, including 51 SUMs, 24 COUNTs, 8 MAXs, 4 logical ANDs and some custom computations such as finding top ten and computing the median. The other reducers in this category also access their input in an order-independent way. For example, 7 reducers store input into `Lists` and sort them before further processing while 22 reducers store the input into hash tables and only test for existence. There are also 19 reducers that process each input row independently just like a mapper. It is unclear why they were implemented as reducers rather than mappers.

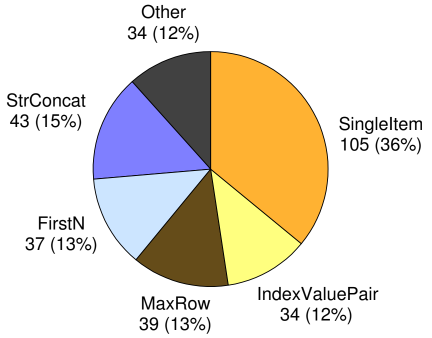

Figure 6: Patterns in 292 non-commutative reducers.

Type 1
```
int x = 0;
foreach (Row row in input) {
    x = row["x"].Integer;
    // ...
}
```

Type 2
```
bool flag = true;
int x = 0;
foreach (Row row in input) {
    if (flag) {
        flag = false;
        x = row["x"].Integer;
    }
    // ...
}
```

Figure 7: The SingleItem pattern.

Impure. 19 reducers are found to be impure and so may yield different results given the same input in the same row order. Most of them call a random number generator explicitly to sample a small portion of data in their input or to implement randomized algorithms such as *k*-means clustering [6]. Nondeterministic outcome of these reducers are very likely to be intended by their author.

Unknown. There are two reasons for 29 reducers to be put in this category. For some reducers, ILSpy fails to decompile them. The other reducers are just too complicated for us to judge whether they are commutative or not.

4.1 Non-commutativity Patterns

To better understand why so many reducers are non-commutative, we further classify them according to how their output is sensitive to input order. The second surprising finding in our study is that most 292 non-commutative reducers can be categorized into five simple patterns according to the root cause of their non-commutativity even as we see a wide variety of algorithms and coding styles among them. In the rest of this section, C_{Key} denotes the reduce key column(s) and C with another subscript to denote a non-key column. Figure 6 shows classification results that are described as follows.

Finding 2: Most non-commutative reducers (88%) can be classified into five simple patterns according to how their output depends on input order.
Implication: Most non-commutative reducers can be automatically recognized and understood well by a pattern matching approach (discussed in Section 7).

Script

```
Hashed = SELECT Url, GetHash(Url) AS UrlHash, ...
         FROM Data;

T = REDUCE Hashed ON Url
    PRODUCE ...
    USING UrlReducer;
```

UrlReducer

```
string url = null;
string urlHash = null;
foreach (Row row in input) {
    url = row["Url"].String;
    urlHash = row["UrlHash"].String;
    // aggregates the other columns
}
```

Figure 8: An example SingleItem reducer that yields deterministic results because functional dependency `Url` → `UrlHash` is guaranteed by the preceding map stage (the `SELECT` statement).

SingleItem. A reducer with this pattern processes input rows in a loop, extracting some non-key column C_{Single} in the first or last row for later use to compute the output. Since the input row order is nondeterministic, it is possible for the reducer to use C_{Single} in any row while that column in other rows is discarded. If two rows have different values in C_{Single}, either value can be selected. Figure 7 shows two examples of this pattern where reducer output depends on the value of `x`, which is sensitive to input row order. 105 out of the 292 non-commutative reducers belong to this pattern.

Although we can construct a set of input rows for each SingleItem reducer so that output depends on input row order, we find two clues in many such reducers that implies that their authors know of a certain implicit input data property that guarantees deterministic results. Such a data property is a *functional dependency* (written $C_{Key} \rightarrow C_{Single}$) in relational database theory [4] that ensures that non-key column C_{Single} always has the same value in an input group with the same C_{Key}. We express this in SCOPE's data model as:

DEFINITION 2. *In a table, a set of columns X is said to **functionally determine** another set of columns Y (written $X \rightarrow Y$) if, and only if, each X value is associated with precisely one Y value.*

For simplicity, we also use $X \rightarrow Y$ to denote functional dependency between two single columns.

For a functional dependency $C_{Key} \rightarrow C_{Single}$, it is common that the reduce key column C_{Key} is named by a unique identifier of some objects while the name of C_{Single} is an attribute of the objects. For example, C_{Key} is "`UserID`" and C_{Single} is "`UserType`". These names imply that all rows with the same `UserID` refer to the same user that has a fixed `UserType`.

For some SingleItem reducers, functional dependency is also guaranteed at some previous stage. For example, `UrlReducer` in Figure 8 is non-commutative with a SingleItem pattern on the non-key column `UrlHash`, which is calculated from the key column `Url` in the preceding map stage (the `SELECT` statement) by the `GetHash` function that returns the hash value of a given string. For all rows with the same `Url` in the `Hashed` table, the `UrlHash` column has a constant value. As a result, in each invocation of the reducer, `UrlHash` in all `input` rows are the same.

As data is sometimes processed by multiple jobs in a production cluster, such functional dependency can also be guaranteed by some stage in another job that generates the input file of the job invoking the SingleItem reducer. We even found a SingleItem reducer relying on a functional dependency that was guaranteed two jobs before in the pipeline. Our finding indicates that implicit data properties are very important in understanding user-defined functions, which requires not only whole-program analysis but also analyzing other jobs that generate target job input.

Type 1

```
Dictionary<int, int> dict = new ...;
foreach (Row row in input) {
    int x = row["x"].Integer;
    int y = row["y"].Integer;
    dict[x] = y;
    // ...
}
```

Type 2

```
Dictionary<int, int> dict = new ...;
foreach (Row row in input) {
    int x = row["x"].Integer;
    int y = row["y"].Integer;
    if (!dict.Contains(x))
        dict[x] = y;
    // ...
}
```

Figure 9: The IndexValuePair pattern.

Script

```
SegView = SELECT ClientID, SegmentID, SUM(Views) AS Views
          FROM Data GROUP BY ClientID, SegmentID;

AggView = REDUCE SegView ON ClientID
          PRODUCE ...
          USING SegmentsReducer;
```

Figure 10: SCOPE script that guarantees the functional dependency (`ClientID`, `SegmentID`) → `Views` in the first reduce stage (the `SELECT` statement). Therefore the following IndexValuePair reducer `SegmentsReducer` always yields deterministic results.

IndexValuePair. In this pattern, a reducer considers two non-key columns C_{Index} and C_{Value} as index-value pairs and stores them in an array or hash table with C_{Index} being indices/keys and C_{Value} being values. The reducer's output also depends on the final content of the array or hash table. If two rows have the same index in C_{Index} but different values in C_{Value}, the value stored in the array or hash table depends on the order of these two rows. There are also two types of this pattern, as shown in Figure 9, keeping the first or the last value in case of conflicts, respectively. 34 reducers exhibit this pattern.

Like the SingleItem pattern, a functional dependency of the form $(C_{Key}, C_{Index}) \rightarrow C_{Value}$ is required for an IndexValuePair reducer to yield deterministic results. This functional dependency can also be guaranteed by a preceding statement. Sometimes there is a stronger property on the data; e.g. `SegmentsReducer` in Figure 10 is an IndexValuePair reducer where C_{Index} is `SegmentID` and C_{Value} is `Views`. Its `SegView` input is the result of an aggregation on columns (`ClientID`, `SegmentID`) performed by the preceding `SELECT` statement. Therefore in the `SegView` table, no two rows have the same value of (`ClientID`, `SegmentID`) and so in each group of `ClientID` all values in `SegmentID` are distinct, which is stronger than the functional dependency (`ClientID`, `SegmentID`) → `Views`.

Although implicit data properties are crucial for the SingleItem and IndexValuePair reducers to yield deterministic results, they are not checked by the runtime engine of SCOPE or other popular MapReduce implementations such as Hadoop. There is also no language-level support for users to annotate desired data properties explicitly so that they are enforced. As a result, the user's assump-

```
int max = 0;
int y = 0;
foreach (Row row in input) {
    int x = row["x"].Integer;
    if (max < x) {
        max = x;
        y = row["y"].Integer;
    }
}
// emit (max, y)
```

Figure 11: The MaxRow pattern.

tion on an implicit data property is error prone. For example, the consistency between the reducer and the previous statement or job guaranteeing the data property may not be maintained properly. The assumption can be wrong because the user overlooks some corner cases, or the assumption can be correct but the input data is corrupted due to other faults. Even worse, when such a property is not satisfied, the reducer silently produces somewhat well-formed but nondeterministic results, where the error in a large volume of data can be very hard to observe. Three real bugs presented in Section 6 are caused by violations of implicit data properties; both were exercised daily but went undetected for at least three months until we reported the issues to the authors. To avoid such bugs, designers of MapReduce systems can allow users to annotate desired data properties so that they can be enforced automatically.

> **Finding 3**: 48% of the non-commutative reducers exhibit the SingleItem or IndexValuePair pattern. They would yield deterministic results if certain implicit data properties are satisfied, that, as later shown in Section 5, are in most cases satisfied by real data.
> **Implication**: Non-commutative reducers are harmless if data satisfies certain implicit properties, whose violations can indicate a bug as programmer assumptions are inconsistent with data.

MaxRow. Finding the maximum (or minimum) value in a column (C_{Max}) is a common reducer operation. Although it is commutative by itself, user-defined reducers often emit other non-key columns C_{Other} in the row where a maximum value is found. If two rows have the same maximum value but different values in C_{Other}, the values emitted for C_{Other} are nondeterministic. Figure 11 demonstrates this pattern where C_{Max} is column x and C_{Other} is column y.

A MaxRow reducer can also yield deterministic results if certain data properties are satisfied (e.g., $(C_{Key}, C_{Max}) \rightarrow C_{Other}$), but in a way that is different from SingleItem and IndexValuePair. Among the 39 MaxRow reducers, timestamp is the primary data type of the column according to which the maximum row is selected; such reducers usually want to find the latest event in their input. There are also some reducers that find the most frequent item by selecting the row with the maximum "count" column computed by a previous reduce stage. For most of these MaxRow reducers, it is highly likely that any maximum row is acceptable in case of a tie.

FirstN. 37 reducers are non-commutative because they only return the first N rows and discards the rest of their input (Figure 12). The N output rows directly depends on the input row order, unless a certain data property is satisfied, such as that the number of rows in each input group is not greater than N, or all the input rows are identical for each reduce key. Note that FirstN reducers with $N = 1$ also satisfy the criteria of the SingleItem pattern. We categorize them into the FirstN pattern because they, like other FirstN reducers, are only emitting N input rows unchanged while SingleItem reducers also aggregate other columns.

There are two completely different purposes in writing these FirstN reducers, depending on the value of N. In 23 out of the 37 FirstN reducers $N = 1$, which indicates that these reducers want to extract any one row in each group. Many of them just want to eliminate redundant items and we even find "dedup", "duplicate", "unique" or "distinct" in the name of 7 reducers. A large N tends to indicate the other purpose, which is to limit the number of rows in each group: three reducers have N greater than 1 million and all of them have "throttle" in their names. There are also 10 reducers with N between 2 and 3000 and 1 reducer for which we cannot find the value of N since it is computed through a complex logic. It is hard to determine their purpose without checking their real input data, but it is likely that the author of a FirstN reducer is aware of the uncertainty in the output.

```
int count = 0;
foreach (Row row in input) {
    count++;
    if (count >= N) break;
    // ...
}
```

Figure 12: The FirstN pattern.

```
List<string> names = new List<string>();
foreach (Row row in input) {
    names.Add(row["Name"].String);
    // ...
}
string s = String.Join("|", names);
// emit s
```

Figure 13: The StrConcat pattern.

> **Finding 4**: 26% of the non-commutative reducers exhibit the MaxRow or FirstN pattern. Nondeterminism in their output is very likely to be tolerable.
> **Implication**: Non-commutativity is harmless if users know and can tolerate nondeterminism in the results.

StrConcat. The 43 reducers with this pattern emit concatenation of some string columns from all or a part of their input rows. With different input row orders, a StrConcat reducer concatenate strings in different orders and thus generates different output strings. Figure 13 shows an example StrConcat reducer.

String concatenation in StrConcat reducers are usually used as a custom serialization of string arrays. We observe that in many cases input strings are concatenated by special delimiters in these reducers, and the result is then split into the components back for future processing in succeeding stages or another job that consumes this job's output. Impact of the uncertainty in such string concatenation depends on how future stages or jobs process the string components. Without manually tracking the usage of concatenated strings and analyzing future processing logic, we conservatively consider these reducers harmless. However, we do catch a subtle bug caused by a StrConcat reducer that is described in Section 6.

Finding 5: 15% of the non-commutative reducers exhibit the StrConcat pattern. However, their results are often treated as a set of items in an order-insensitive way.
Implication: Non-commutativity is also harmless if the nondeterministic output of a reducer is processed in such a way in succeeding stages that the final output of the whole program is deterministic.

5. CHECKING REAL DATA

The classification of user-defined reducers shows that many of them are non-commutative. However, it is not enough to claim that a non-commutative reducer is buggy by only analyzing its code. It is possible that nondeterminism in the reducer's output is known and tolerable or even intended; the author can have certain domain knowledge on the input data that guarantees that the reducer always produces deterministic results even if it is non-commutative. In Section 4, we have found some non-trivial data properties guaranteeing determinism for SingleItem and IndexValuePair reducers. But the SCOPE runtime system is not aware of these patterns and does not enforce corresponding data properties automatically. There is also no language-level support for users to annotate desired data properties manually. When some data property is violated, unexpected nondeterministic results might show up unnoticed. We therefore check these properties on real input data for some reducers to understand whether programmers rely on the properties and, if so, whether or not the properties are satisfied by the data.

We randomly sample some reducers exhibiting the SingleItem and IndexValuePair patterns. After obtaining the reducer's input, we submit a new SCOPE job to check whether or not the data property is satisfied in each input group. For 28 out of the 37 reducers, implicit data properties are satisfied entirely in the input data so their non-commutativity is harmless. For the other 9 reducers, implicit data properties are violated in at least one input group. By further studying other statements in the job of each reducer, we find that sometimes a few columns in the reducer's output are later discarded and nondeterminism does not propagate to the final output of the job. We conservatively consider a reducer to be buggy only if we are sure that the final output is also nondeterministic. In this way, we find three bugs that we present in Section 6.

6. BUG CASE STUDY

This section describes five real bugs that we found in our manual study; they manifest themselves by causing programs to produce nondeterministic results due to non-commutative reducers. All bugs were in production programs that were running periodically (daily in most cases) for at least three months. All bugs were confirmed with the authors of the programs and eventually fixed.

Bug 1: Bad Assumption on Functional Dependency

Non-commutativity pattern: SingleItem.

This program processes logs of Bing's on-line advertisement system through many pipelined map and reduce stages. The bug was found in one reduce stage whose input data records page views associated with advertisement events such as impressions and clicks. Each record consists of over 30 columns including ViewID, attributes such as State and AdId, and many numerical metrics of an advertisement event such as Clicks and Revenue. Each page view is often associated with multiple events where the buggy reducer (shown in Figure 14) aggregates them into one output record according to page view ID. The author assumed that ViewID functionally determines all page view attributes (ViewID → State, ViewID → AdId, etc.). As a result, the reducer selects attributes in the last row of each ViewID as output (lines 8 and 9).

Reducer

```
 1 string id = null;
 2 string state = null;
 3 string adid = null;
 4 int clicksTotal = 0;
 5 double revenueTotal = 0;
 6 foreach (Row row in input) {
 7     id = row["ViewID"].String;
 8     state = row["State"].String;
 9     adid = row["AdId"].String;
10     clicksTotal += row["Clicks"].Integer;
11     revenueTotal += row["Revenue"].Double;
12     // aggregate other columns
13 }
14 output["ViewID"].Set(id);
15 output["State"].Set(state);
16 output["AdId"].Set(adid);
17 output["Clicks"].Set(clicksTotal);
18 output["Revenue"].Set(revenueTotal);
19 // set other columns
20 yield return output; // emit the result
```

Figure 14: Code for Bug 1.

Table 2: An example input group and two possible output rows for Bug 1.

An example input group:

ViewID	State	AdId	···	Clicks	Revenue	···
DEADBEEF	CA	10000	···	1	1.00	···
DEADBEEF	CA	10000	···	0	0.10	···
DEADBEEF	CA	10000	···	0	0.00	···
DEADBEEF	TX	10000	···	1	0.80	···

Possible output 1:

DEADBEEF	CA	10000	···	2	1.90	···

Possible output 2:

DEADBEEF	TX	10000	···	2	1.90	···

The assumed functional dependency turns out to be incorrect: there are cases where two events with the same ViewID have two different values in the State column; for example, when a user travels across states. Table 2 shows an example input group and two possible outputs. The program might output either "CA" or "TX" as the State of this ViewID due to nondeterminism. We re-ran the whole program twice on the same real input in the production cluster, observing differences in the outputs.

The impact of this bug is non-negligible even as the assumed functional dependency held for about 99.9% of the ViewIDs. Because this program aggregates the numerical metrics of advertisement events on State, AdId and some other page view attributes in the following steps, we have found that the aggregation results are nondeterministic for 49% of the aggregation groups. We calculated the minimal and maximal values for the nondeterministic groups on the real data: in some cases, the maximal values can be an order of magnitude higher than the minimum. The writer of the program confirmed the seriousness of the issue and rewrote the program.

Bug 2: Data Corruption

Non-commutativity pattern: SingleItem.

This bug is similar to Bug 1 in that it also involves the SingleItem pattern where an assumption of functional dependency does not always hold. The violation in this case is caused by data corruption, however. The input data is another log from an advertisement system in a complex semi-structured format. The program extracts useful columns by a user-defined mapper and later aggregates the data

```
                         Script
 1 T = REDUCE Data ON AdId
 2     PRODUCE AdId, ...
 3     USING EntropyReducer;
```

```
                     EntropyReducer
 4 string key = "";
 5 foreach (Row row in input) {
 6     key = row[36].String;
 7     // ...
 8 }
 9 output[0].Set(key);
10 // ...
11 yield return output;
```

Figure 15: Code for Bug 3.

by Key in a reducer. We find that occasionally two rows with the same reduce key have different XID where one is actually a prefix of the other. Further study on the original log verifies our conjecture that a row is truncated in arbitrary row positions: either the head or the tail of a row can be missing. The functional dependency Key → XID is violated due to such data truncation. In the first user-defined mapper that extracts useful information from the log, there is already some code to filter out malformed log entries. But it only checks the existence of some useful columns and not their length. XID is usually the last portion of useful data in a log entry and so incomplete ones are undetected.

The impact of this bug is yet unclear. On the one hand, only several truncated log entries can be found in the log every day and the reducer's nondeterministic output is not amplified in succeeding stages as in Bug 1. On the other hand, this log is very important to the advertisement system and also consumed by at least ten other programs, none of which filters out truncated log entries correctly.

Bug 3: Incorrect Column Index

Non-commutativity pattern: SingleItem.

The reducer in this case computes the entropy in two non-key columns for each group and also includes the reduce key AdId in its output row, as shown in Figure 15 (line 9). The root cause is simple: the column index of AdId is 37 but the author writes 36 by mistake (line 6). This was undetected for months partly because column 36 is another ID in the same format.

Although the root cause is simple, the bug manifests nondeterministic behaviors as the reducer falls in the SingleItem pattern on the non-key column 36. Checking the real data shows that the needed functional dependency does not hold in the real data, helping us to spot the bug.

Bug 4: Counting Concatenated Strings

Non-commutativity pattern: StrConcat.

The job in this case computes statistics of online articles. Each input row of ArticleReducer (shown in Figure 16) contains some information of an article, including articleId and a profiles column that contains some associated URLs concatenated by the delimiter "|". The reducer aggregates rows for the same article and outputs distinct URLs that are also concatenated and stored in the profiles column. Subsequent statements count the number of articles with different sets of URLs.

ArticleReducer finds distinct URLs by adding each URL in each input row into a HashSet and reading all URLs from it after processing all input. In the C# implementation of HashSet, the order in which the default enumerator iterates the set depends on the order in which elements are added. As a result, distinct URLs in profiles of the output row is concatenated in a nondeterministic input-dependent order. And the subsequent statements may evaluate a larger count than the correct value, because two articles with the same set of URLs may have different strings in profiles and are therefore considered different. The author either overlooks the order requirement in the "count-distinct" statements or has wrong assumptions on the implementation details of HashSet, which leads to the bug in this StrConcat reducer.

```
                         Script
// Schema of Data: articleId, profiles, ...
T1 = REDUCE Data ON articleId
     PRODUCE articleId, profiles
     USING ArticleReducer;

T2 = SELECT DISTINCT profiles FROM T1;

T3 = SELECT COUNT() + "␣unique␣articles." AS num FROM T2;
```

```
                     ArticleReducer
string articleId = null;
HashSet<string> urls = new HashSet<string>();
foreach (Row current in input) {
    articleId = current[0].String;
    string[] array = current[1].String.Split("|");
    for (int i = 0; i < array.Length; i++)
        urls.Add(array[i]);
}
outputRow[0].Set(articleId);
outputRow[1].Set(String.Join("|", urls));
yield return outputRow;
```

Figure 16: Code for Bug 4.

```
                         Reducer
 1 Special special;
 2 List<Impression> impressions = new List<Impression>();
 3 bool specialFound = false;
 4 
 5 foreach (Row row in input) {
 6     bool isSpecial = row[6].Boolean;
 7     bool isImpressions = row[7].Boolean;
 8     if (isSpecial) {
 9         special.XXX = row[0].String;
10         special.YYY = row[1].Integer;
11         specialFound = true;
12     }
13     if (isImpressions)
14         impressions.Add(new Impression(row[3].Integer,
15             row[4].Long, row[5].Integer));
16 
17     if (specialFound) {
18         // aggregate impressions and emit the result
19     }
20 }
```

Figure 17: Code for Bug 5.

Bug 5: Misplacing of a Brace

Non-commutativity pattern: Other.

Figure 17 shows the code of this buggy reducer that aggregates advertisement impressions. In each group of input rows, there are multiple impression rows and zero or one special row. The reducer stores information of each row in local variables (lines 8 to 15). It should perform the aggregation and emit the result after the loop if a special row is found. However, the author puts the emitting code inside the loop by mistake (lines 17 to 19). As a result, the emitting code would run multiple times if the special row appears before some impression rows in a group, and the amount of excessive output is nondeterministic depending on the position of the special row. The bug is not detected by the production team partially because the excessive output rows are still well-formed and meaningful. Mixed tabs and spaces in the reducer's source code also makes the bug inconspicuous for code reviews.

After fixing the bug by moving lines 17 to 19 out of the loop, the reducer remains non-commutative because it becomes a special SingleItem one on column 1 (column 0 is the reduce key) that only stores the value of this column in the last special row. But it will never produce nondeterministic results because there is at most one special row in each group of the real data.

6.1 Summary

We have shown five bugs in non-commutative user-defined reducers from production programs that survive code reviews, testing and trial on real data for at least three months, which indicates that non-commutativity can be harmful and undetected for long. In addition, Bug 1 demonstrates that just a little uncertainty in the buggy reducer's output can be amplified later and impact correctness of the final results in a significant way. Note that the nondeterminism is only a symptom but never the root cause. All these bugs are caused by logic or data errors, including wrong assumption on implicit data properties, data corruption, or various semantic errors. Making the system running deterministically does not fix any of them.

Finding 6: We find five bugs in well-tested production programs caused by non-commutative reducers. Their root causes include wrong assumptions on implicit data properties, data corruption, and various semantic errors leading to non-commutativity.
Implication: Non-commutativity remains a source of bugs that can go undetected for a long period of time. Checking implicit properties on real data is an effective way to detect this kind of bugs.

7. DISCUSSION

Our study justifies the pervasiveness of non-commutativity among user-defined reducers and shows that flagging all non-commutative reducers as bugs by only analyzing code would create too many false positives because most of them are actually harmless. In particular, SingleItem and IndexValuePair reducers behave like commutative ones if certain data properties are satisfied by real data, which is the common case. However, such reducers do lead to bugs in case of data property violations. We therefore propose to detect this kind of bugs by both recognizing certain non-commutativity patterns in code and checking desired properties on data.

Recognizing non-commutativity patterns. Section 4.1 has shown that all the five non-commutativity patterns are very simple, making a pattern matching approach feasible to recognize them automatically. For the SingleItem and IndexValuePair patterns, most reducers exactly conform to the code structures in Figures 7 and 9 according to our experience in the manual study. A static program analysis that identifies the loop enumerating `input` and matches code structure patterns can be able to recognize a large portion of SingleItem and IndexValuePair reducers, their related columns and desired data properties. We leave implementation of this pattern matching approach as our future work.

Another option is to allow users to annotate desired data properties by themselves. In SCOPE, this can be done by extending the original annotation syntax. For example, the `REDUCE` statement in Figure 8 can be annotated as below.

```
[SINGLE_ITEM_COLUMN("UrlHash")]
T = REDUCE Hashed ON Url ...
```

With a little extra efforts by users, annotations makes desired data properties explicit and helps users to avoid property violations in future maintenance. This approach can be used in combination with program analysis to increase coverage.

Instrumented SingleItem Type 2

```
1  bool flag = true;
2  int x = 0;
3  foreach (Row row in input) {
4      if (flag) {
5          flag = false;
6          x = row["x"].Integer;
7      } else if (x != row["x"].Integer) {
8          ReportViolatedProperty(...);
9      }
10     // ...
11 }
```

Instrumented IndexValuePair Type 2

```
12 Dictionary<int, int> dict = new ...;
13 foreach (Row row in input) {
14     int x = row["x"].Integer;
15     int y = row["y"].Integer;
16     if (!dict.Contains(x))
17         dict[x] = y;
18     else if (dict[x] != y)
19         ReportViolatedProperty(...);
20     // ...
21 }
```

Figure 18: Instrumentation in a IndexValuePair reducer to check desired data properties.

Checking desired data properties. Given desired data properties, the compiler can generate extra code to check the properties on real data and report violations to users. Because the properties are functional dependencies between columns, checking them only needs one pass on data and can be done in the enumerator that is used by a reducer to scan its input. If the non-commutativity patterns are recognized by the pattern matching approach, it is also possible for the compiler to instrument original user-defined reducers like lines 7, 8, 18 and 19 in Figure 18, which would usually incur limited overhead.

8. RELATED WORK

Commutativity of reducers. Previous work considered commutativity a correctness requirement on reducers in MapReduce programs. Our study shows that such a requirement might not be appropriate because non-commutative reducers are common and mostly harmless. Csallner et al. [3] proposed a white-box symbolic execution approach to test commutativity. Xu et al. [13] studied 23 reducers in open-sourced MapReduce programs and used a black-box approach to test commutativity. Both of them check code and generate different input sequences with the same data set, on which the target reducer outputs different results. Our study shows that certain data properties (functional dependencies) in the data can render non-commutative reducers harmless, making a case for checking data properties. In addition, 11 out of 23 reducers in [13] are non-commutative and exhibit the StrConcat pattern according to their study. The distribution of non-commutativity patterns is not consistent with our results, probably because open-sourced MapReduce programs in their study process simpler data with fewer cross-column relations than our samples from the production environment as discussed in Section 3.1.

Commutativity is also studied to improve job performance. Yu et al. [14] proposed various partial aggregation mechanisms in MapReduce programs to reduce network I/O when the target reducers satisfy certain commutativity properties. While our study focuses on the correctness issue of non-commutative reducers, we also plan to leverage the patterns found in this work to improve job performance further, even when the reducers are non-commutative.

Empirical study on MapReduce programs. Some researchers have studied characteristics of MapReduce programs from various angles. Kavulya et al. [8] analyzed 10-months of trace data from a production MapReduce cluster and provided the performance and failure characteristics of the jobs in this cluster. They aimed to achieve better system performance by improving scheduling decisions and resource provisioning, while our study helps to understand subtle correctness requirements in MapReduce programs. The more related work by Li et al. [9] was a characteristics study on failures of SCOPE jobs, revealing that exceptional data and mismatched data schema are a major source of job failures. Although we both focused on correctness and our study subjects both come from the SCOPE platform in Microsoft, there were essential differences between their work and ours. First, their study focused on job failures with explicit exceptions thrown out, while we analyzed jobs that terminated successfully but could potentially produce undesirable nondeterministic results due to non-commutativity. Second, their goal was to save system resources wasted by failed jobs, while we aim to minimize harmful effects on production systems by finding subtle bugs. Interestingly, we shared the same finding that inconsistency between code and data was a major cause of errors in MapReduce programs. However, they discussed inconsistency on data format, which is usually explicit and easy to diagnose, whereas we have found violations of implicit data properties that are hard to observe and could go undetected for a long period of time.

9. CONCLUSION

The correctness of MapReduce programs is going to be increasingly important as they are becoming widely used in data-parallel computation, but the subject remains under-studied and not well understood. This paper focuses on the issue of non-commutative reducers, as the nondeterminism caused by such reducers is a problem that is unique to MapReduce programs, with significant implications on correctness. Our in-depth study on production MapReduce programs and their data reveals several surprising findings on commutativity, nondeterminism, and correctness, which we believe provide valuable guidelines for advancing our understanding on and ensuring the correctness of MapReduce programs.

10. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable comments and suggestions. We are grateful to Haixun Wang and Haoxiang Lin for their feedback on our work. We also thank the production teams in Microsoft for sharing their jobs and data, and all the developers who cooperated with us. This work is partially supported by the NSFC projects 61232008 and 61103021 and the 863 project 2012AA010901.

11. REFERENCES

[1] Apache. Hadoop. `http://lucene.apache.org/hadoop/`.

[2] Apache. Mahout. `http://mahout.apache.org/`.

[3] C. Csallner, L. Fegaras, and C. Li. New ideas track: testing MapReduce-style programs. In *ESEC/FSE*, pages 504–507, 2011.

[4] C. J. Date. *Database Design and Relational Theory: Normal Forms and All That Jazz*. O'Reilly Media, 2012.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[6] E. W. Forgy. Cluster analysis of multivariate data: efficiency versus interpretability of classifications. *Biometrics*, 21:768–769, 1965.

[7] ILSpy. The open-source .NET assembly browser and decompiler. `http://ilspy.net/`.

[8] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An analysis of traces from a production MapReduce cluster. In *CCGRID*, pages 94–103, 2010.

[9] S. Li, H. Zhou, H. Lin, T. Xiao, H. Lin, W. Lin, and T. Xie. A characteristic study on failures of production distributed data-parallel programs. In *ICSE*, pages 963–972, 2013.

[10] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[11] B. O'Sullivan, J. Goerzen, and D. B. Stewart. *Real World Haskell*. O'Reilly Media, 2008.

[12] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a Map-Reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[13] Z. Xu, M. Hirzel, and G. Rothermel. Semantic characterization of MapReduce workloads. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, pages 87–97, 2013.

[14] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP*, pages 247–260, 2009.

[15] J. Zhou, N. Bruno, M.-C. Wu, P.-A. Larson, R. Chaiken, and D. Shakib. SCOPE: parallel databases meet MapReduce. In *The VLDB Journal*, volume 21, pages 611–636, October 2012.