A Characteristic Study on Failures of Production Distributed Data-Parallel Programs

Sihan Li[1,2], Hucheng Zhou[2], Haoxiang Lin[2], Tian Xiao[2,3], Haibo Lin[4], Wei Lin[5], Tao Xie[1]
[1]North Carolina State University, USA, [2]Microsoft Research Asia, China, [3]Tsinghua University, China,
[4]Microsoft Bing, China, [5]Microsoft Bing, USA
sli20@ncsu.edu, {huzho, haoxlin, v-tixiao, haibolin, weilin}@microsoft.com, xie@csc.ncsu.edu

***Abstract*—SCOPE is adopted by thousands of developers from tens of different product teams in Microsoft Bing for daily web-scale data processing, including index building, search ranking, and advertisement display. A SCOPE job is composed of declarative SQL-like queries and imperative C# user-defined functions (UDFs), which are executed in pipeline by thousands of machines. There are tens of thousands of SCOPE jobs executed on Microsoft clusters per day, while some of them fail after a long execution time and thus waste tremendous resources. Reducing SCOPE failures would save significant resources.**

This paper presents a comprehensive characteristic study on 200 SCOPE failures/fixes and 50 SCOPE failures with debugging statistics from Microsoft Bing, investigating not only major failure types, failure sources, and fixes, but also current debugging practice. Our major findings include (1) most of the failures (84.5%) are caused by defects in data processing rather than defects in code logic; (2) *table-level* failures (22.5%) are mainly caused by programmers' mistakes and frequent data-schema changes while *row-level* failures (62%) are mainly caused by exceptional data; (3) 93% fixes do not change data processing logic; (4) there are 8% failures with root cause not at the failure-exposing stage, making current debugging practice insufficient in this case. Our study results provide valuable guidelines for future development of data-parallel programs. We believe that these guidelines are not limited to SCOPE, but can also be generalized to other similar data-parallel platforms.

I. Introduction

The volume of data in real-world applications has been exploding, making traditional solutions with a central database impractical for storing and processing massive data. To solve this problem, previous work proposed a distributed storage system with a data-parallel programming paradigm on top, such as MapReduce [7]. In industry, state-of-the-art data-parallel programming languages are hybrid languages that consist of declarative SQL-like queries and imperative user-defined functions (UDFs), including Pig Latin [22] at Yahoo!, Hive [26] at Facebook, FlumeJava [4] at Google.

Inside Microsoft Bing, we have SCOPE [3] built atop Dryad [11], which is adopted by thousands of developers from tens of different product teams for index building, web-scale data mining, search ranking, advertisement display, etc. Currently, there are thousands of SCOPE jobs per day being executed on Microsoft clusters with tens of thousands of machines. A notable percentage of these jobs threw runtime exceptions and were aborted as failures. Among all failures from Microsoft clusters within two weeks, 6.5% of them had execution time longer than 1 hour; the longest one executed for 13.6 hours and then failed due to un-handled null values. A similar situation was reported by Kavulya *et al.* [14]. They analyzed 10 months of Hadoop [2] logs from the M45 supercomputing cluster [28], which Yahoo! made freely available to selected universities. They indicated that about 2.4% of Hadoop jobs failed, and 90.0% of failed jobs were aborted within 35 minutes while there was a job with a maximum failure latency of 4.3 days due to a copy failure in a single reduce task. Such failures, especially those with long execution time, resulted in a tremendous waste of shared resources on clusters, including storage, CPU, and network I/O. Thus, reducing failures would save significant resources.

Unfortunately, there is little previous work that studies failures of data-parallel programs. The earlier-mentioned work by Kavulya *et al.* [14] studies failures in Hadoop, an implementation of MapReduce. Their subjects are Hadoop jobs created by university research groups, being different from production jobs. Besides, state-of-the-art data-parallel programs in industry are written in hybrid languages with a different and more advanced programming model than MapReduce. Thus, their results may not generalize to these industry programs. Moreover, they focus on studying the workloads of running jobs for achieving better performance by job scheduling, rather than failure reduction in development.

To fill such a significant gap in the literature and the academia/industry, we conduct the first comprehensive characteristic study on failures and fixes of state-of-the-art production data-parallel programs for the purpose of failure reduction and fixing in future development. Our study includes 200 SCOPE failures/fixes and 50 SCOPE failures with debugging statistics randomly sampled from Microsoft Bing. We investigate not only major failure types, failure sources, and fixes, but also current debugging practice. Note that our study focuses on only failures caused by defects in data-parallel programs, and **excludes** the underlying system or hardware failures. Particularly, the failures in our study are runtime exceptions that terminate the job execution. We do **not** study failures where the execution is successfully finished but the produced results are wrong, since we do not have test oracles for result validation. Moreover, all SCOPE jobs in our study are obtained from Microsoft clusters. Programmers may conduct local testing before they submit a job to clusters. Hence, some failures that are addressed by local testing may not be present in our study.

TABLE I
OUR MAJOR FINDINGS ON FAILURE CHARACTERISTICS OF REAL-WORLD DATA-PARALLEL PROGRAMS AND THEIR IMPLICATIONS

Findings on Failures	**Implications**
(1) Most of the failures (84.5%) are caused by defects in data processing rather than defects in code logic. The tremendous data volume and various dynamic data sources make data processing error-prone.	Documents on data and domain knowledge from data sources could help improve the code reliability. Programmers are encouraged to browse the content of data before coding, if possible.
(2) 22.5% failures are *table-level*; the major reasons for *table-level* failures are programmers' mistakes and frequent changes of data schema.	Local testing with a small portion of real input data could effectively detect *table-level* failures.
(3) *Row-level* failures are prevalent (62.0%). Most of them are caused by exceptional data. Programmers cannot know all of exceptional data in advance.	Proactively writing exceptional-data-handling code with domain knowledge could help reduce *row-level* failures.
Findings on Fixes	**Implications**
(4) There exist some fix patterns for many failures. Fixes are typically very small in size, and most of them (93.0%) do not change data processing logic.	It is possible to automatically generate fix suggestions to programmers.
Findings on Debugging	**Implications**
(5) There are cases (8.0%) where the current debugging tool in SCOPE may not work well because the root cause of the failure is not at the failure-exposing stage.	Automatically generating program inputs to reproduce the entire failure execution could be a complementary approach to current debugging practices.

Specifically, our study intends to address the following research questions:

RQ1: *What are common failures of production distributed data-parallel programs? What are the root causes?* Knowing common failures and their root causes would help programmers to avoid such failures in future development.

RQ2: *How do programmers fix these failures*[1]*? Are there any fix patterns?* Fix patterns could provide suggestions to programmers for failure fixing.

RQ3: *What is the current debugging practice? Is it efficient?* Efficient debugging support would be beneficial for failure fixing.

Our major findings and implications are summarized in Table I. The most significant characteristic of failures in data-parallel programs is that most failures (84.5%) are caused by defects in data processing rather than defects in code logic. The tremendous data volume and various dynamic data sources make data processing error-prone. More knowledge on data properties, such as nullable for a certain column, could help improve code reliability. We also find that most failures can be put into a few categories, and there are limited corresponding failure sources (e.g., exceptional data) and fix patterns (e.g., setting default values for null columns). Such knowledge on failure types, failure sources, and fix patterns would bring substantial benefits to failure reduction and fixing. Moreover, to balance the cost of data storage and shifting, the SCOPE debugging tool used in practice enables locally debugging only the computation stage where the failure is exposed (i.e., failure-exposing stage). It does not work well if the root cause of the failure is not at the failure-exposing stage. This finding implies that whole-program debugging with low cost is needed in some cases. Although our study is conducted on only the SCOPE platform, we believe that most of our findings and implications can also be generalized to other similar data-parallel systems. We will discuss in detail the generality of our study in Section VII.

[1]Programmers resolve failures by fixing code defects, system, or hardware configurations, providing workaround, etc. Without causing confusion, in this paper, we refer to such failure resolution as failure fixing.

Besides our study results, we also share our current practices and ongoing efforts on failure reduction and fixing. There is a series of tools that could be used to improve reliability of SCOPE jobs, including compile-time checking, local testing, failure reproduction, and fix suggestion.

In summary, our paper makes the following contributions:

- We present the first comprehensive characteristic study on failures and fixes of production data-parallel programs and provide valuable findings and implications for future development and research.
- We find the current debugging practice in SCOPE to be efficient in most cases in terms of fast failure reproduction, yet revealing some interesting cases that call for improvement on the current practice.
- We present a series of our existing and under-development tools for failure reduction and fixing.

The rest of the paper is organized as follows. Section II provides a brief primer on SCOPE. Section III describes our study methodology. Section IV presents our study results on failures/fixes, which answer our research questions RQ1 and RQ2. Section V presents our study results on current debugging practice in SCOPE, which answer our research question RQ3. Section VI presents our current and future work. Section VII discusses the generality of our study and our suggestions for failure resilience. Section VIII lists related work and Section IX concludes this paper.

```
public class CopyProcessor : Processor {
  public Schema Produces(string[] columns,
      string[] args, Schema input_schema) {
    return input_schema.Clone();
  }
  public IEnumerable<Row> Process(RowSet input,
            Row output_row, string[] args) {
    foreach (Row input_row in input.Rows) {
      input_row.CopyTo(output_row);
      yield return output_row;
    }
  }
}
```

Fig. 1. The simplest user-defined processor `CopyProcessor`, which returns the copy of input. Any UDF processor inherits from `ScopeRuntime.Processor`, and implements two methods: (1) `Produce`, which defines the output schema, (2) `Process`, which implements the processing logic and generates the output.

II. SCOPE Background

SCOPE is the production data-parallel computation platform for Microsoft Bing services. The SCOPE language is a hybrid of declarative SQL language for expressing high-level data flow and imperative C# language for implementing user-defined functions as local computation extension, similar to Pig Latin [22], Hive [26], FlumeJava [4] and Microsoft DryadLINQ [11]. The rest of this section describes the SCOPE data model and programming model, as well as the execution and life cycle of a SCOPE job.

A. Relational Data Model

SCOPE provides a relational data model like SQL, which encapsulates data sets with *column*, *row*, and *table*. A table consists of a set of rows; a row consists of a set of columns with primitive or complex user-defined types. Each table is associated with a well-defined schema represented as $(columnName_1 : Type_1, ..., columnName_n : Type_n)$. Columns are accessed by either name or index in the form of $row[columnName]$ or $row[columnIndex]$.

B. UDF Centric Programming Model

The programming model of SCOPE provides three elementary operators: *processor*, *reducer*, and *combiner*, as the base classes for all user-defined functions (UDFs); while *extractor* and *outputter* derived from *processor* are dedicated to read from and write to underlying data streams. *Processor* and *reducer* are similar to *mapper* and *reducer* in MapReduce, respectively. SCOPE extends MapReduce with *combiner*, which generalizes *join* on heterogenous data sets. SCOPE offers built-in implementations of many common relational operations for programmers' convenience, and also allows programmers to implement customized operators as C# UDFs. Relational operations like *filter*, *selection*, and *projection* are achieved by *processors*, while *distinct* can be implemented as a *reducer*. Figure 1 illustrates a simple *processor*, which sequentially copies every input row. From a structural perspective, SCOPE models the application workflow as a

```
1  REFERENCE "/my/PScoreReducer.dll";
2  t1 = EXTRACT query:string, clicks:long,
              market:int, ...
4        FROM     "/my/click_1029"
5        USING    DefaultTextExtractor()
6        HAVING   IsValidUrl(url);
7  t2 = REDUCE  t1 ON query
8        PRODUCE query, score, mvalue, cvalue
9        USING    PScoreReducer("clicks")
10 t3 = PROCESS t2 PRODUCE query, cscore
11       USING    SigReportProcessor("cvalue")
12       OUTPUT   t3 TO "/my/click/1029";
```

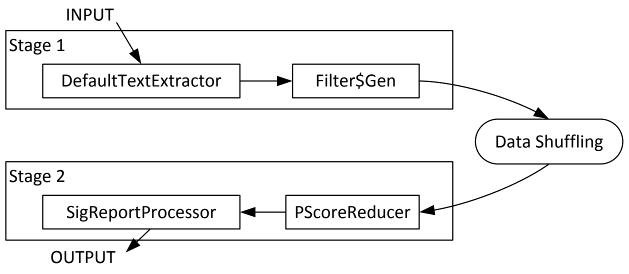


Fig. 2. A SCOPE job with its execution graph.

direct acyclic graph (DAG), different from MapReduce, which strictly follows a two-phase workflow with *mapper-reducer*.

C. Job Execution

A SCOPE job consists of input data, compiled binaries, and a DAG execution plan describing *computation* and *data-shuffling* stages. A computation stage includes one or more chained operators, starts after all its predecessors have finished, and independently runs on a group of machines with partitioned data. A data-shuffling stage then connects two consecutive computation stages by transmitting requisite data among machines. A typical SCOPE job has three phases: *data extraction*, extracting raw data into a structured table format; *data manipulation*, manipulating and analyzing data; and *output*, exporting results to an external storage.

Figure 2 shows a sample SCOPE job with its execution plan. An external DLL file is first explicitly referenced (Line 1). Next, rows of typed columns (Line 2) are extracted from a raw log file (Line 4) as the initial input using a default text extractor (Line 5) and filtered by certain conditions (Line 6). Then input rows are fed to a user-defined operator `PScoreReducer` (Line 9) to produce a new table with four columns (Line 8). Finally, the user-defined operator `SigReportProcessor` (Line 11) is applied and the ultimate result is exported (Line 12). In the execution plan graph, the `Filter$Gen` operator is generated from the `HAVING` clause at Line 6; other operators correspond to keywords `EXTRACT`, `REDUCE`, `PROCESS`, respectively. Each directed edge represents the data flow between operators.

D. Life Cycle

The typical life cycle of a SCOPE job starts from the development phase. After being tested locally, source scripts are submitted to clusters and the job is executed in parallel. If the job fails or ends with an unexpected result, the

TABLE II
CLASSIFICATION OF SCOPE FAILURES

Dimension	Category	Failure Description	No.	Ratio
Table Level	Undefined Column	Accessing column with incorrect name or index	24	12.0%
	Wrong Schema	Mismatch between data schema and table schema	16	8.0%
	Others	Other table-level failures	5	2.5%
	Subtotal	Total table-level failures	45	22.5%
Row Level	Incorrect Row Format	Corrupt rows with exceptional data	45	22.5%
	Illegal Argument	Argument not satisfying method requirement	34	17.0%
	Null Reference	Dereference on null column values	21	10.5%
	User-defined	Customized exceptions defined in UDFs	14	7.0%
	Out of Memory	Memory used up by accumulating data under processing	3	1.5%
	Others	Other row-level failures	7	3.5%
	Subtotal	Total row-level failures	124	62%
Data Unrelated	Resource Missing	Cannot find DLLs or scripts for execution	10	5.0%
	Index Out of Range	Accessing array element with out-ranged index	9	4.5%
	Key Not Found	Accessing dictionary item with non-existing key	5	2.5%
	Others	Other data-unrelated failures	7	3.5%
	Subtotal	Total data-unrelated failures	31	15.5%

programmer downloads relevant data to a local computer and starts debugging in the SCOPE IDE, the environment for developing SCOPE jobs. After defects are fixed, the patched job is re-executed until the correct result is returned.

III. METHODOLOGY

A. Subjects

We took real SCOPE jobs submitted by Microsoft production teams as our study subjects, and collected all job information including initial input data, source scripts, compiled binaries, execution plan, and runtime statistics.

Failures in our study were runtime exceptions that terminated job executions. We did not study failures where the execution was successfully finished but the produced results were wrong since we did not have test oracles for result validation. Hence, the job that ended without exceptions was regarded as a successful one.

Sample Set A. To study failures and fixes in production jobs, we collected all Failed/Successful (F/S) job pairs within two weeks by matching both job names and submitter names. 200 F/S job pairs were randomly sampled out as Sample Set A.

Sample Set B. To study the current debugging practice in SCOPE, we collected all failed jobs that were debugged using the local debugging tool, along with their debugging statistics. We randomly sampled 50 of them as Sample Set B.

B. Classification and Metrics

Failure classification for Sample Set A was done manually. We first carefully went through all 200 F/S job pairs and understood why the failures happened and how they were fixed. Then we classified these failures from the data point of view: whether the failure was related to input data, and which data level (table or row) triggered the failure. Furthermore, we classified these failures based on their exception types obtained from the error messages.

We next describe the metrics used in our study. First, the number of lines of source code (LOC) was used to measure the size of fixes. We relied on the *WinDiff* tool to find differences between the failed and successful scripts, and then manually counted the LOC of changes belonging to the fixes since there might be fix-irrelevant code modifications. We also measured the execution time of SCOPE jobs and the size of downloaded data for debugging, which were directly obtained from runtime statistics and SCOPE IDE logs.

C. Threats To Validity

Threats To Internal Validity. Subjectiveness in the failure classification was inevitable due to the large manual effort involved. Besides, there also might be human mistakes in counting LOC and filtering fix-irrelevant code changes. These threats were mitigated by double-checking all manual work. If there were different opinions, a discussion was brought up to reach an agreement.

Threats To External Validity. We conducted our study within only Microsoft, making it possible that some of our findings might be specific to SCOPE and would not hold in other systems. Hence, we do not intend to draw general conclusions for all distributed data-parallel programs. In Section VII, we discuss in detail which findings could be generalized to other systems similar to SCOPE.

IV. FAILURES AND FIXES IN DISTRIBUTED DATA-PARALLEL PROGRAMS

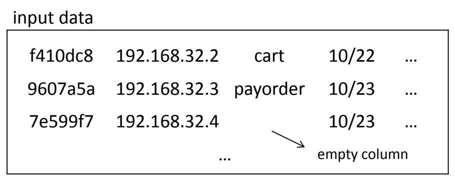
In this section, we first present the failure classification in Sample Set A, and then describe root causes and fixes for each category. Finally, we summarize what we learned from real-world cases.

A. Failure Classification

Compared to traditional counterparts, data-parallel programs are more data centric and their code logic generally focuses on

```
  QueryData = SELECT RawQuery, FormCode,
-             ~~Frequency, ...~~
+             Market, Frequency, ...
              FROM RawData;
  FilteredData = PROCESS QueryData
                 USING FormCodeFilter();
  ...
  public class FormCodeFilter: Processor {
      ...
      if(allowedMarkets.Contains(
         row["Market"].String))
      ...
  }
```

Fig. 3. A real-world example of an undefined-column failure. The `Market` column is not selected into `QueryData` but accessed through expression `row["Market"]`, causing the failure. The fix is adding the `Market` column to `QueryData`.


input data
empty column

f410dc8	192.168.32.2	cart	10/22	...
9607a5a	192.168.32.3	payorder	10/23	...
7e599f7	192.168.32.4		10/23	...
		...		

```
  ResultSet = SELECT ClientId, UserIp,
                ScenarioName, Date, ...
                FROM @InputData
-               ~~USING DefaultTextExtractor();~~
+               USING DefaultTextExtractor("-silent");
  ...
```

Fig. 4. A real-world example of a *row-level* column-number mismatch due to a null column value. The `DefaultTextExtractor` extracts one fewer column for the third row because of the null value for column `ScenarioName`. The failure is fixed by adding the silent option to filter out rows with incorrect format.

data analytics and processing. Table II depicts the classification of 200 failures in Sample Set A. We classify these failures into two big categories: data-related failures and data-unrelated failures. Most failures (169/200) belong to the former as being caused by data-processing defects, while only 31 failures belong to the latter due to defects in code logic, or other reasons. We further divide data-related failures into *table-level* and *row-level*, and build subcategories by failure exception types. We next go through each failure category in detail: how the failure happens, what is the root cause, and how to fix it.

1) Table-Level Failures: A failure is regarded as *table-level* if every row in the table could trigger it. 22.5% failures in Sample Set A are *table-level* failures, classified into the following major subcategories.

Undefined Column. This subcategory is the most frequent *table-level* failures (24/45). SCOPE enables column access by either name or index. Such failures occur when an invalid column is referenced: its column name cannot be found or its index is not within the range. Figure 3 shows an undefined-column example. In the C# code, the expression `row["Market"]` gets the value of column named *Market* in the row. This operation is similar to accessing items in a dictionary. However, there is no `Market` column in the `QueryData` table because it is not produced by the `SELECT` statement. The fix, the code in red, is straightforward by adding the `Market` column in selection. An immediate impression from this example is that an undefined column can be detected at compile-time. It is true in this example because the `Market` column is accessed through a constant string name. However, the column name or index could be variables whose values are determined at runtime. In this case, the compiler can never decide whether the column access is valid or not. Hence, column-access validity is always checked at runtime.

Wrong Schema. A wrong-schema failure (16/45) usually occurs in the data-extraction phase, where raw input data are extracted into a structured table for later manipulation. A wrong schema is caused by mismatch of either the column number or column type. For example, there are 10 columns in the input while only 9 columns are defined in the schema, or the first column contains float values while it is declared as integer.

There were two major reasons that led to *table-level* failures. One reason (13/45) was programmers' mistakes. It was common to see misspells in column names and miscounts in column indices. At first, we were a little surprised at the high mistake ratio. However, after further investigation, we found that real-world SCOPE scripts may involve tables with hundreds or even thousands of columns. In this case, manually writing schemas or accessing columns could be error-prone. Another major reason was the frequent changes of input-data schema without updating processing programs. Since the data were usually from multiple dynamic sources such as web contents or program outputs, the programmers might be unaware of the changes of data sources. We found that the input of quite some failed jobs changed its schema frequently. Even worse, it was often the case that the data producer was not the SCOPE job programmer. Fortunately, for most *table-level* failures, our study indicated that programmers could easily locate the defects based on the error message, and fixes were usually straightforward, such as modifying the schema or correcting the corresponding column name or index.

2) Row-Level Failures: A failure is said to be *row-level* when only a portion of rows in the table could cause the failure while the other rows are processed successfully. In Sample Set A, there are 124 (62.0%) *row-level* failures including the following major subcategories.

Incorrect Row Format. Incorrect row format is the most frequent subcategory (22.5%) among all failures. Similar to the wrong-schema failure, it also happens in the data extraction phase due to column-number mismatch or column-type mismatch. The difference is that the incorrect-row-format failure is caused by a few rows with exceptional data rather than schema mismatch. Here, by exceptional data, we mean special cases in data under processing. Thus, only those exceptional rows could trigger the failure while the other rows are well processed. Figure 4 shows a real-world example of a *row-level* column-number mismatch due to a null column

```
  ...
  SELECT AVG(a.StrToInt(Duration)) AS AvgD,
         MAX(Helper.StrToInt(Duration)) AS MaxD
         FROM Rowset;
         OUTPUT TO @PARAM_OUTPUT_SET;
  public class Helper {
      public int StrToInt {
          ...
-         ~~if(String.IsNullOrEmpty(val))~~
+         if(!Int32.TryParse(val, out ivalue))
              ivalue = 0;
-         ~~ivalue = Int32.Parse(val);~~
          ...
      }
  }
```

Fig. 5. A real-world example of an illegal-argument failure. Any integer value that exceeds the range of `Int32` will trigger the failure. The fix is adding a safer method `TryParse` to guard against all exceptional data.

```
t2 = REDUCE t1 ON name
     PRODUCE tag, name, seconds, count
     USING MyReducer();
...
public class MyReducer: Reducer {
  List<Row> list = new List<Row>();
  int last = -1; int i;
  foreach (Row row in input.Rows {
    list.Add(row);
    if (row[3].String == "pattern")
      last = list.Count - 1;
  }
  for (i = last + 1; i < list.Count; i++) {
    ...
  }
}
```

Fig. 6. A simplified real-world example of the out of memory failure. All input rows are accumulated in memory for later global processing. The fix requires a more memory efficient algorithm.

value, from Sample Set A. The input file in the figure is synthesized by us to simulate real input characters. When processing the third row with the empty column `ScenarioName`, the `DefaultTextExtractor` would not know there is a null value for `ScenarioName` column, and thus extracts one fewer column for this row. Hence, an incorrect-row-format exception is thrown. The failure is fixed by adding the silent option, which tells the `DefaultTextExtractor` to discard rows that cannot be extracted into the correct format.

Illegal Argument. The illegal argument is the second most frequent (17.0%) subcategory. Such failures happen when the argument value does not satisfy the requirement of the invoked method (e.g., requiring none-empty/null value or positive integer). Figure 5 shows an example of an illegal-argument failure from Sample Set A. Although the programmer already considers exceptional values like null and empty, and replaces them with default value 0. However, there are some other integer values that exceed the range of `Int32` and thus violate the argument requirement of `Int32.Parse`. The fix is using a safer method `TryParse`, which returns true and assigns the parsed result to `ivalue` if the parsing succeeds; returns false if the parsing fails.

Null Reference. A null-reference failure happens when a null value is dereferenced. The null value usually comes from a null column in data under processing rather than an uninitiated object declared in C# code. A typical example is that a string column contains a null string value, and the programmer performs string operations on this column (e.g., `row["StringColumn"].IndexOf("-")`) without nullity checking so that the null value is dereferenced.

Out of Memory. An out-of-memory failure occurs when the programmer attempts to load extremely huge data into memory all at once. Although we find only 3 such failures, this subcategory is very interesting and important. It reveals an essential difference between distributed data-parallel programming and traditional small-scale counterpart: more memory-efficient algorithms should be devised facing unpredictably massive data. Figure 6 shows a typical example that the programmer accumulates all input rows in memory so as to conveniently process them after the last occurrence of a certain pattern. The `input` to `MyReducer` is a group of rows with the same key to be reduced. As the row number of the group could be very large, such an attempt, adding all rows of a group into a list, results in memory exhaustion quickly. Fixing an out-of-memory failure is not straightforward since the programmer has to come up with a more memory-efficient implementation for the same code logic.

As we can see from the preceded typical examples, most (99/124) of the *row-level* failures are due to exceptional data. However, we should not blame programmers for these failures because the data volume is so large that it is impossible for programmers to know about all exceptional data in advance.

We find two patterns for fixing the failures caused by exceptional data. One is the row-filtering pattern (43/99), which discards exceptional rows. Since there are millions of rows in datasets, a few exceptional rows could be treated as noises and filtered out without really affecting the job results. The other is the default-value pattern (31/99), which replaces the exceptional values with the default value of its type.

3) Data-unrelated failures: We find 31 failures not to be closely related to data in Sample Set A. Some of them involve language features while the others are due to semantic errors.

Resource Missing. A resource-missing failure occurs when the job cannot find the needed resources (e.g., referenced scripts or external DLLs) for execution. In Sample Set A, the main reason (8/10) for resource-missing failure is the programmer's neglect of an important SCOPE language feature. That is, if one needs to reference an external resource, she should explicitly import the referenced resource in the script by using SCOPE keyword `RESOURCE` for script files or `REFENRENCE` for DLL files. Only with these keywords, the SCOPE engine would know that the referenced files should be copied to each distributed machine because these distributed machines are independent and they all need a copy of referenced files for execution. Hence, even if one uploads the referenced resources to the cluster without these keywords, the referenced resources would not be copied to each machine. The tricky part is that in the local development environment, these keywords are not required when the referenced resources are in the default project workspace, making such failures not revealed in local testing. All such failures are fixed by adding keywords `RESOURCE`/`REFENRENCE` to import the missing resources.

Other Data-unrelated Failures. The index-out-of-range and key-not-found exceptions are just like those in ordinary C# programs. The index-out-of-range exception is thrown when accessing an element of an array with the index outside the array bound, and the key-not-found exception is thrown when retrieving an element from a collection (e.g., dictionary) with a key that does not exist in the collection. In Sample Set A, there are various reasons for these failures including the programmer's mistakes and defects in algorithms. Due to these various reasons, the fixes are diverse and we do not find any fix pattern for these failures.

B. Learning From Practice

The major characteristic of SCOPE failures is that most of them are caused by defects in data processing rather than defects in code logic. Essentially, such characteristic is due to the tremendous volume and dynamism of input data. Since the data are extremely large and come from various domains, it is common that there are missing data or certain special case data. It is impossible for programmers to know all these exceptional data before they really run programs against them. Moreover, the data are usually obtained from multiple dynamic sources, such as web contents and program outputs, which may change frequently. It is difficult and undesirable to keep programmers updated with these changes all the time. These challenges are not unique for SCOPE but also exist in other similar platforms for big-data processing.

However, we can somehow alleviate such problems by providing more information on data. For example, for those data with relatively stable schema, the data producer, such as log-file designers, could provide detailed documentation on data schema or some default data extractors. The programmer is encouraged to look at the content of data to know more about the data before coding. Moreover, the programmer could leverage domain knowledge to infer some data properties, e.g., a certain column is nullable.

Finding 1: Most of the failures (84.5%) are caused by defects in data processing rather than defects in code logic. The tremendous data volume and various dynamic data sources make data processing error-prone.
Implication: Documents on data and domain knowledge from data sources could help improve the code reliability. Programmers are encouraged to browse the content of data before coding, if possible.

The *table-level* failures are mainly caused by the programmers' mistakes and frequent changes of data schema. Since the *table-level* failure could typically be triggered by every row in the table, running the job against a small portion of the real input data could effectively detect these failures.

Finding 2: 22.5% failures are *table-level*; the major reasons for *table-level* failures are programmers' mistakes and frequent changes of data schema.
Implication: Local testing with a small portion of real input data could effectively detect *table-level* failures.

Most of *row-level* failures are due to exceptional data. Since the exceptional data are unforeseeable, programmers could proactively write exceptional-data-handling code with domain knowledge to help reduce failures, sharing the similar philosophy with the defensive programming.

Finding 3: *Row-level* failures are prevalent (62.0%). Most of them are caused by exceptional data. Programmers cannot know all of exceptional data in advance.
Implication: Proactively writing exceptional-data-handling code with domain knowledge could help reduce *row-level* failures.

How to fix the failures is closely related to the reasons that cause the failures. For most failure categories, there exist fix patterns. Fixes under these patterns are very small in terms of LOC. In Sample Set A, 95.0% fixes are within 10 LOC and 87.0% fixes are within 5 LOC; the average size of fixes is 3.5 LOC. In addition, fix patterns such as row filtering, nullity checking, and resources import, usually do not involve program semantics. Hence, based on these patterns, it is possible to automatically generate some fix suggestions to help programmers fix corresponding defects.

Finding 4: There exist some fix patterns for many failures. Fixes are typically very small in size, and most of them (93.0%) do not change data processing logic.
Implication: It is possible to automatically generate fix suggestions to programmers.

V. Debugging in SCOPE

Debugging in distributed systems is challenging. Since the input data are huge, it is prohibitively expensive to re-execute the whole job for step-through diagnosis. To balance the cost of data storage and shifting, SCOPE enables programmers to debug the failure-exposing stage of the job locally. When a failure happens, the SCOPE system locates the commodity machine where the failure happens, and persistently stores the input data (a partition of the whole stage input) on that machine. Later the programmers could download the input data along with executables from the failed machine, and start live diagnosis in their local simulation environment. The downloaded input would guarantee to reproduce the failure because executions on each distributed machine are independent so that the local environment is the same with that on the failed machine.

To investigate the effectiveness of current debugging practice, we studied Sample Set B, consisting of 50 failed jobs that were debugged with the local debugging tool. An important indicator for effectiveness of the debugging tool was whether programmers came up with correct fixes

```
  ...
  Statistics = SELECT StartTime, EndTime,
-                 ~~(EndTime-StartTime)/60~~
+                 (EndTime−StartTime)/60.0
                  AS Duration, ...
                  FROM ResultSet;
  ...
  Out = SELECT TaskID, Name, ... ,
        WaitingTime/Duration AS Percentage
        FROM Info;
  ...
```

Fig. 7. A real-world example that illustrates the root-cause problem. A divide-by-zero exception is thrown in the second select statement due to zero values in `Duration` while the root cause is in the first select statement from a different computation stage.

by using this tool. We found that all the 50 failures in Sample Set B were correctly fixed, which, to some extent, demonstrated the effectiveness of the debugging tool, although the debugging tool might not be the only helper to find a fix. Moreover, from the programmer's perspective, we measured how long it took to initiate the debugger (i.e., time to download debugging-required data to the local machine). Our results showed that 35 out of 50 jobs in Sample Set B downloaded less than 1 Gigabytes data, and the average size of downloaded data for each job was 5.3 Gigabytes. With the high-speed internal network, the debugger could be initiated within few minutes in most cases.

Hence, the debugging tool was efficient in most cases, in terms of quickly reproducing partial failure execution locally. However, we found an interesting case in which the debugging tool may not work well. When a failure occurs, the root cause of the failure may not lie in the computation stage that exhibits the failure (failure-exposing stage). The program state may already turn bad long before the bad state is finally exposed. In this case, debugging the failure-exposing stage may not give sufficient information on how the program state turns bad. Hence, an interesting phenomenon happens that the programmer wants to debug earlier successful stages. We did find such request in the SCOPE internal mailing list. However, it is impractical to enable debugging all successful stages because it would require persistently storing all intermediate data between each stage for later downloading, and the cost is unaffordable. Even if we can afford temporarily storing such intermediate data, we are still unable to download all input data for a stage because they are too huge, sometimes even larger than the original input.

We found that there were 4 out of 50 failures with the root cause outside the failure-exposing stage. Figure 7 shows one example of them. A divide-by-zero exception is thrown in the second statement due to the zero value in `Duration`. The root cause for the zero value lies in the first select statement. Since `StartTime` and `EndTime` are integers, (`EndTime` - `StartTime`) / 60 is zero when the numerator is less than 60. However, the first statement is not in the failure-exposing stage and will never be debugged with this tool.

Essentially, this root-cause problem is due to the balance act: partial-program (failure-exposing stage) debugging.

```
#IF (LOCAL)
  #DECLARE input string = "input.tsv";
#ELSE
  #DECLARE input string = "/my/input.tsv";
#ENDIF
```

Fig. 8. Adaptive data selection by local versus remote execution.

To achieve low-cost whole-program debugging, we could try to automatically generate small program inputs to reproduce the entire failure execution by leveraging existing symbolic-execution engines [27], [9], [24]. This approach could be complementary to the current debugging approaches.

Finding 5: The current debugging practice is efficient in most cases in terms of fast failure reproduction. However, there are some cases (8.0%) where the debugging tool may not work well because the root cause of the failure is not inside the failure-exposing stage.
Implication: Automatically generating smaller program inputs to reproduce the entire failure execution could be complementary to current debugging approaches.

VI. Current and Future Work

In this section, we present a series of our current practices on failure reduction for SCOPE jobs, including language extension of SCOPE and compile-time analysis. Data synthesis and bug-fix suggestion remain as the future work to reduce debugging efforts.

SCOPE Extension. We extend SCOPE with the following embedded language supports.

1) *Nullable Type* is used to tolerate Null-Reference failures. "Nullable" data types defined in C# [19] can be assigned null to value types such as numeric and boolean types. It is particularly useful when dealing with databases containing elements that may not be assigned a value. We extend SCOPE by supporting C# nullable data types, and annotate them with "?" postfix, a shorthand for "Nullable<T>". If a nullable column is null, it returns the default value for the underlying type.
2) *Structured Stream* is designed to avoid failures occurring in the data-extraction phase. It provides schema metadata for unstructured streams so that SCOPE can directly read from and write to structured streams without extractor and outputter, and thus reduces data-extraction failures.
3) *Local Testing* greatly facilitates testing and diagnosis by enabling jobs to run entirely on a single machine with local test inputs. The local execution has nearly identical behaviors to its distributed execution in clusters. If the programmer specifies the *LOCAL* keyword together with the *#IF* directive in the script (see Figure 8), the SCOPE IDE generates a special job that has no data-shuffling stages and spawns very few instances. The programmers can test and debug the job step-by-step, as if the job is a local process.

Compile-Time Program Analysis. We have built the SCA (SCOPE Code Analysis) tool, which has been integrated into the SCOPE IDE, to report potential defects before job execution. SCA, built atop FxCop [18] and Phoenix [20] compiler, includes 11 SCOPE-related checking rules, e.g., the null reference and column assignment should accept correct types. A rule is a piece of C# code targeting at a specific failure category. Currently, SCA is capable of detecting the Undefined-Column, Wrong-Schema, Resource-Missing, and Null-Reference failures. However, there are non-trivial false alarms produced by SCA. Our ongoing efforts focus on reducing these false alarms by performing whole-program analysis across function calls, operators, and stages.

Data Synthesis. Motivated by Finding 5, it is useful to develop a small-scale-data synthesis tool for both testing and debugging purposes. First, to help failure diagnosis, we could leverage part of initial inputs and temporary results of the failed job to synthesize much smaller failure-triggering inputs for quick reproduction of the same failure. Second, we could generate special input to trigger the hidden defects to cause failures. It is feasible to implement these features by extending current symbolic-execution techniques.

Fix Suggestion. Motivated by Finding 4, it is promising to develop a tool to generate fix suggestions interactively with the programmers. A straightforward implementation is to first identify the failure pattern (type and reason) from error messages and call stacks, and then generate suggestions based on the corresponding fix patterns found in our study.

VII. Discussion

A. Generality of Our Study

Although our study is conducted exclusively on SCOPE jobs from Microsoft Bing, most of our results can still be generalized to other data-parallel systems, such as Pig Latin, Hive, and FlumeJava.

From the input-data perspective, the volume of datasets processed by SCOPE jobs ranges from a few gigabytes to tens of petabytes, representing the typical data volume of current big-data applications in industry. Moreover, the data processed by SCOPE jobs are from similar sources (e.g., websites, user logs) with those from web companies, such as Google, Facebook, and Yahoo!. Finally, the relational data model used in SCOPE is widely adopted by data-parallel platforms. From the programming-model perspective, SCOPE shares the same hybrid programming model with Pig Latin, Hive, FlumeJava. Such model is state-of-the-art for data-parallel programming.

Hence, our Findings 1, 2, and 3 about failure characteristics could be generalized, at least to those web companies who share similar data and programming models. For Finding 4, we believe that some of our fix patterns, such row filter and default value, also exist in other systems because they are quite intuitive and straightforward ways to handle exceptional data. While other patterns related to SCOPE language features, like adding `RESOURCE` keyword to import resources, are specific in SCOPE. Finding 5 can be generalized to systems that enable only partial-program debugging.

B. System Design for Failure Resilience

In our study, we found a job that executed for 13.6 hours and failed due to a null column value. The defect was easy to locate, and was fixed by just filtering out rows with null column values. Unfortunately, the patched job had to start all over again and execute for another long time.

Motivated by this observation, we propose a stop/resume mechanism for failure resilience. Instead of killing the job right upon a failure, SCOPE job manager could first suspend the job execution, and then notify the programmer of the failure, wait for her to fix the defect. After the programmer submits the patched job, it is re-compiled into a new execution plan, and the job manager resumes the execution by determining the stages needed to be re-computed based on the execution-plan changes. In this manner, we reuse the previously computed results so that resources for re-computation are saved and the job latency is also reduced, compared to re-executing the patched job from the beginning.

One key fact making this stop/resume mechanism promising is that in our study, 93.0% fixes do not change the code logic. This fact implies that a large portion of previous results could be reused. There may be some cases where our proposed mechanism would not work. Example cases are when the fix changes the program a lot or the programmer can not come up with a fix in short time. In such cases, the failure resilience could be turned off by programmers.

VIII. Related Work

Bug Characteristics Studies. A lot of work has been done to study bug characteristics in software systems. Most of these studies share the same goal of improving reliability of software systems. Earlier work mainly studied bugs in large open-source software systems such as operating systems, databases. Chou *et al.* [6] studied bugs in Linux and OpenBSD that were detected by their static-analysis tool. Gu *et al.* [10] studied Linux kernel behaviors under injected bugs. Chandra *et al.* [5] studied bugs in open-source software from a failure recovery perspective. Recently, there were some studies that focused on certain types of bugs, such as concurrency bugs [17], performance bugs [12].

Unfortunately, there was little work that studied the failures of distributed data-parallel programs. Kavulya *et al.* [14] studied failures in MapReduce programs. Although we shared some similar findings on failure workloads, there were essential differences between their work and ours. First, their subjects were jobs created by university research groups while ours came from production jobs. Second, the SCOPE jobs in our study were written with a hybrid programming model whereas their Hadoop jobs were based on the MapReduce model. Different programming models may lead to different failure characteristics. Third, their work mainly focused on studying the workloads of running jobs for achieving better system performance such as job scheduling whereas our work focused on studying source code and input data of failed jobs for the purpose of failure reduction and fixing. There was work [13] that studied fault tolerance in MapReduce. However, the failures in their study referred to failures in the underlying distributed systems including hardware failures, whereas our failures were caused by defects in data-parallel programs.

Fix Characteristics Studies. There were several studies on fixing patterns. Pan *et al.* [23] defined 27 bug-fixing patterns in Java software using syntax components involved in source-code changes. To fix bugs, Kim *et al.* [15] built *bug fix memories*, which were a project-specific knowledge base on bugs and the corresponding fixes. Some previous work [29], [25] also studied incorrect bug fixes. Again, our study about fixing patterns was specific to data-parallel programs, particularly for SCOPE jobs.

Debugging in Distributed Systems. Olston *et al.* [21] provided a framework *Inspector-Gadget* for monitoring and debugging Pig Latin [22] jobs. Their interviews of programmers, which served as the motivation of their work, shared some common conclusions with our study, such as the needs for detecting exceptional data that violated certain data properties and needs for step-through debugging. However, their debugging support was different from ours. Instead of debugging on local machine in SCOPE, they provided a remote debugger on the failed commodity machine. Although remote debugging did not require downloading data to the local machine, it occupied shared resources on the commodity machine until the end of debugging process. Moreover, their debugger also had the root-cause problem described in our Finding 5 because their debugger enabled only partial-program debugging as we did. Some other work employed program-analysis techniques to help debugging in distributed systems. Liu *et al.* [16] used runtime checkers of program states to reveal how the program states turned bad. Taint analysis for data tracing [8], [1] was used to locate data that triggered failures.

IX. Conclusion

This paper has presented the first comprehensive characteristic study on failures/fixes of production distributed data-parallel programs. We studied 250 failures of production SCOPE jobs, examining not only the failure types, failure sources, and fixes, but also current debugging practice. The major failure characteristic of data-parallel programs was that most of the failures (84.5%) were caused by defects in data processing rather than defects in code logic. The tremendous data volume and various dynamic data sources made data processing error-prone. In addition, there were limited major failure sources, with existing fix patterns for them, such as setting default values for null columns. We also revealed some interesting cases where the current SCOPE debugging tool did not work well and provided our suggestions for improvement. We believe that our findings and implications provide valuable guidelines for future development of data-parallel programs, and also serve as motivations for future research on failure reduction and fixing in large-scale data-processing systems.

Acknowledgements

This work is supported in part by a NIST grant and NSF grants CCF-0845272, CCF-0915400, and CNS-0958235. We thank the anonymous reviewers for their valuable comments. We are grateful to Lidong Zhou, and Jingren Zhou for their valuable feedback. We also thank the SCOPE team in Microsoft Bing for sharing their production jobs and data.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, pages 74–89, 2003.

[2] Apache. Hadoop. http://hadoop.apache.org/.

[3] R. Chaiken, B. Jenkins, P. ke Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[4] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI*, pages 363–375, 2010.

[5] S. Chandra and P. M. Chen. Whither generic recovery from application faults? a fault study using open-source software. In *DSN*, pages 97–106, 2000.

[6] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *SOSP*, pages 73–88, 2001.

[7] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[8] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: a pervasive network tracing framework. In *NSDI*, pages 20–20, 2007.

[9] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, pages 213–223, 2005.

[10] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of Linux kernel behavior under errors. In *DSN*, pages 459–468, 2003.

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[12] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *PLDI*, pages 77–88, 2012.

[13] H. Jin, K. Qiao, X.-H. Sun, and Y. Li. Performance under failures of MapReduce applications. In *CCGRID*, pages 608–609, 2011.

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

[15] S. Kim, K. Pan, and E. E. J. Whitehead, Jr. Memories of bug fixes. In *FSE*, pages 35–45, 2006.

[16] X. Liu, W. Lin, A. Pan, and Z. Zhang. Wids checker: combating bugs in distributed systems. In *NSDI*, pages 19–19, 2007.

[17] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339, 2008.

[18] Microsoft. FxCop. http://msdn.microsoft.com/en-us/library/bb429476(v=vs.80).aspx.

[19] Microsoft. Nullable Types. http://msdn.microsoft.com/en-us/library/1t3y8s4s(v=vs.80).aspx.

[20] Microsoft. Phoenix Compiler. http://research.microsoft.com/en-us/collaboration/focus/cs/phoenix.aspx.

[21] C. Olston and B. Reed. Inspector Gadget: A framework for custom monitoring and debugging of distributed dataflows. In *SIGMOD*, pages 1221–1224, 2012.

[22] 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.

[23] K. Pan, S. Kim, and E. J. Whitehead, Jr. Toward an understanding of bug fix patterns. *Empirical Softw. Engg.*, 14(3):286–315, 2009.

[24] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE*, pages 263–272, 2005.

[25] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR*, pages 1–5, 2005.

[26] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Li, and R. Murthy. Hive – a petabyte scale data warehouse using Hadoop. In *ICDE*, pages 996–1005, 2010.

[27] N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In *TAP*, pages 134–153, 2008.

[28] Yahoo! M45 supercomputing project. http://research.yahoo.com/node/1884.

[29] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *ESEC/FSE*, pages 26–36, 2011.