Applying Combinatorial Test Data Generation to Big Data Applications

Nan Li
Research and Development
Medidata Solutions
New York, NY, USA
nli@mdsol.com

Yu Lei
Dept. of Computer Science
and Engineering
The University of Texas at
Arlington
Arlington, TX, USA
ylei@cse.uta.edu

Haider Riaz Khan
Research and Development
Medidata Solutions
New York, NY, USA
hriaz@mdsol.com

Jingshu Liu
Research and Development
Medidata Solutions
New York, NY, USA
jliu@mdsol.com

Yun Guo
Dept. of Computer Science
George Mason University
Fairfax, VA, USA
yguo7@gmu.edu

ABSTRACT

Big data applications (e.g., Extract, Transform, and Load (ETL) applications) are designed to handle great volumes of data. However, processing such great volumes of data is time-consuming. There is a need to construct small yet effective test data sets during agile development of big data applications.

In this paper, we apply a combinatorial test data generation approach to two real-world ETL applications at Medidata. In our approach, we first create Input Domain Models (IDMs) automatically by analyzing the original data source and incorporating constraints manually derived from requirements. Next, the IDMs are used to create test data sets that achieve t-way coverage, which has shown to be very effective in detecting software faults. The generated test data sets also satisfy all the constraints identified in the first step. To avoid creating IDMs from scratch when there is a change to the original data source or constraints, our approach extends the original IDMs with additional information. The new IDMs, which we refer to as Adaptive IDMs (AIDMs), are updated by comparing the changes against the additional information, and are then used to generate new test data sets. We implement our approach in a tool, called *comBinatorial bIg daTa Test dAta Generator (BIT-TAG)*.

Our experience shows that combinatorial testing can be effectively applied to big data applications. In particular, the test data sets created using our approach for the two ETL applications are only a small fraction of the original data source, but we were able to detect all the faults found with the original data source.

CCS Concepts

•Software and its engineering → Software testing and debugging;

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970325

Keywords

Big Data Testing, Combinatorial Testing, Input Domain Model, Adaptive Input Domain Model, Test Data Generation

1. INTRODUCTION AND CHALLENGES

One important characteristic of the *big data* concept is high volume [15]. The datasets used in industry are often measured in terabytes or higher orders of magnitude. For example, at Medidata, we deal with a large amount of data from various clinical trial sites, studies, and subjects. Also, as more clients are using *Internet of Things (IoT)*[1] such as Vital Connect and Actigraph apps, the amount of data to process is expected to grow quickly and tremendously in the future. Even with latest advances in computing technologies such as *Hadoop MapReduce*[2], processing large amounts of data can easily take days, weeks, or even months. Since processing high volumes of data can be time-consuming [22], small yet effective test data sets need to be constructed for testing during development of big data applications. This is particularly so for agile software development where testing is performed frequently.

In this paper, we deal with one common type of big data application called, *Extract, Transform, and Load (ETL)* application. ETL application developers write SQL, Hive or Pig[3] scripts for reporting and analytics purposes. In a typical ETL process, data is extracted from an original data source (e.g., a Microsoft SQLServer database), and is then transformed into a structured format (e.g., a flat file on Amazon Simple Storage Service (S3)) to support queries and analysis. Finally, the data is loaded into a final target (e.g., a PostgreSQL database) for customers to view. The original data source is called the source. The final target that loads data is called the target. For example, at Medidata, we compute, store, and analyze dozens of terabytes of clinical trial data through ETL processes using Amazon Web Services (AWS).

In industry, practitioners often manually generate small test data sets for testing ETL applications. Manual test data generation can be time-consuming, labor-intensive, and error-prone. On the one

[1]The IoT inter-connects embedded devices on the Internet.

[2]Apache Hadoop MapReduce processes large data sets over clusters of computers using *Hadoop Distributed File System (HDFS)*.

[3]Apache Hive and Pig scripts are transformed to MapReduce programs that run on top of HDFS.

hand, it may be difficult to ensure validity of test data when various business and/or structural constraints must be satisfied. On the other hand, the quality of test data heavily depends on human judgment and can vary significantly.

In this paper, we report our experience on applying a combinatorial testing approach to two real-world ETL applications. Our approach consists of two major steps, creating *Input Domain Models (IDMs)* and applying combinatorial testing to the IDMs. In the first step, we automatically create a set of IDMs, one for each database table, from the original data source [2]. We consider each column of a database table to be an input parameter. For each input parameter, important test values are derived from constraints that are either automatically extracted from database schema or manually specified by the user. For example, the user may specify a statistics-based constraint that considers distinct values with the most or least appearances in the original source to be important test values. In the second step, we apply combinatorial testing to the IDMs to generate test data. Combinatorial testing has been shown to be a very effective software testing strategy. The key observation is that most faults are caused by interactions involving only a few parameters [17]. A t-way combinatorial test set is designed to cover all the t-way interactions, i.e., interactions involving no more than t input parameters. When input parameters are properly modeled, a t-way test set is guaranteed to expose faults that are caused by interactions involving no more than t parameters [8].

One problem arises when there are changes in the original data source or constraints. Since the original data source can be large for big data applications, re-creating the IDMs from scratch whenever there is a change is very inefficient. To address this problem, we extend the original IDMs with additional information. The new IDMs, which we refer to as *Adaptive IDMs* (*AIDMs*), include the original IDMs and some analytical results of the original data source and constraints. For example, the analytical results may include the statistical distribution of the original data source. When the original data source or constraints change, our approach does not need to re-process the original data source and constraints. Instead, it quickly updates the AIDM by comparing the changes against the analytical results saved in the AIDM and then generates new test data sets to reflect the latest changes. Because we process the original source data only once, frequent test generation is made possible, which is necessary to support agile development. AIDMs will be discussed in Section 2.4 in detail.

We point out that our approach reuses the real data in the original source as much as possible, even though new data may also be generated. This helps practitioners to generate expected outcomes and also facilitates debugging because real data is more meaningful and thus easier to understand by practitioners, compared to using data that is entirely synthetic.

We have implemented our combinatorial testing approach in a tool, called *comBinatorial bIg daTa Test dAta Generator (BIT-TAG)*. We have used BIT-TAG to test two real-world ETL applications at Medidata. The applications contain nearly 50,000 lines of code, processing a total of over 1.5 TB data with 3.8 billion rows. The experimental results show that the test data sets created using our approach are only a small fraction of the original data source but still detect all the faults that are detected by the original data sources. In the rest of the paper, we use BIT-TAG to refer to our combinatorial approach and tool when there is no ambiguity. We highlight the following major contributions of this paper.

1. We present a systematic approach that applies combinatorial testing to big data applications.

2. We implemented our approach in an enterprise-strength tool, i.e., BIT-TAG, and evaluated the effectiveness and efficiency of BIT-TAG using two real-world industrial ETL products.

3. We report several compelling insights gained from our experience. These lessons could help practitioners better understand and improve the process of applying combinatorial testing to big data applications.

The paper is organized as follows. Sections 2 and 3 present the test data generation approach and implementation. Section 4 gives the experiment. Section 5 discusses the related work. Section 6 concludes this paper and discusses the future work.

2. THE APPROACH

This section presents in detail our combinatorial approach to generate test data. Section 2.1 provides a brief introduction to combinatorial testing. Section 2.2 discusses how we create IDMs with the original data source and constraints. Section 2.3 discusses how we generate test data from IDMs. Section 2.4 shows how BIT-TAG creates and updates AIDMs adaptively when there are changes.

2.1 Combinatorial Testing

Let M be a program with n parameters. Combinatorial t-way testing requires that, for any t (out of n) parameters of M, every combination of values of these t parameters be covered at least once. The value of t is referred to as the testing strength. Consider a program that has three parameters p1, p2, and p3, each parameter having two values, 0 and 1. A 2-way (or pairwise) tests set for the three parameters (p1, p2, p3) could be (0, 0, 0), (0, 1, 1), (1, 0, 1), and (1, 1, 0). An important property of this test set is that if we pick any two columns, i.e., columns p1 and p2, columns p1 and p3, or columns p2 and p3, they contain all four possible pairs of values of the corresponding parameters, i.e., 00, 01, 10, 11. An exhaustive test set for these parameters consist of $2^3 = 8$ tests.

A number of algorithms have been developed for combinatorial test generation [5]. In particular, Lei et al. reported on a combinatorial testing strategy called IPOG [19]. The IPOG strategy generates a t-way test set to cover the first t parameters and extends this test set to cover the first t + 1 parameters. This process is repeated until this test set covers all the parameters. Lei et al. [19] also reported a tool called ACTS that implements the IPOG strategy. In this paper, BIT-TAG uses ACTS to generate combinatorial test sets.

Combinatorial testing can significantly reduce the number of tests. For example, a pairwise test set for 10 Boolean parameters only needs as few as 13 tests, whereas an exhaustive test set consists of 1024 tests [8]. Despite this dramatic decrease in the number of tests, combinatorial testing has been shown to be very effective for general software testing [17].

2.2 Input Domain Model

To apply combinatorial testing, we need to create IDMs consisting of parameters and important test values. Unlike previous research where IDMs are often created manually [7], BIT-TAG automatically creates IDMs. BIT-TAG analyzes constrains derived from requirements and database schemas. Some constraints are automatically derived while others are manually specified. Next, BIT-TAG decides which test values are extracted from the original data, and then creates IDMs with the extracted test values.

Data in various formats (e.g., videos and graphs) could be saved in different kinds of databases (e.g., No-SQL database). In the context of this paper, we study how to generate IDMs from relational databases, which are widely used by many data analytical products nowadays. We will discuss how to create traditional IDMs in this section and extending IDMs to AIDMs in Section 2.4.

Creating an IDM consists of the following three steps. First, we identify all the parameters for which we need to generate test values in the IDM. Second, we select a characteristic for each parameter to create a partition over the domain of this parameter. The partition should be complete, i.e., it should cover the whole domain, and disjoint, i.e., the partitioned blocks must not overlap. For example, a characteristic for a parameter of the string type may be "if this parameter is null." The domain of this parameter is divided into two blocks: "null value" and "values that are not equal to null." Third, we can select null from the first block and a random value such as "test" from the second block since values in the same block are considered to be equivalent.

As mentioned in Section 1, we create one IDM for each table in a database. Each column is considered as a parameter of the IDM. To create partitions, we extract important values from the constraints (the types of constraints are shown below) that are applied to each parameter. All values derived from the constraints are used as boundaries to create blocks. For example, if we derive from constraints the minimum integer (min_int), 0, and the maximum integer (max_int) for an integer parameter, the domain is partitioned into three blocks: [min_int], (min_int, 0], (0, max_int]. Then we can select the max value from each block. Next, we will discuss how to collect the constraints and how to derive test values from the constraints to create partitions for each data type.

We collect constraints from three sources. First, constraints, e.g., foreign key constraints, are derived from database schema. Second, users may specify constraints that are derived from requirements. For example, users may specify some important test values to include in specific-value constraints because these values are required to be queried in the requirements. It is important for testers to get independent requirements and avoid interpreting source code (SQL, Hive or Pig scripts) developed by programmers. Third, BIT-TAG provides built-in edge case test values for different data types, included as specific-value constraints.

BIT-TAG supports ten general types of constraints. In this section, we discuss five constraints related to creating IDMs below, check, default, specific-value, logic, and statistics-based constraints. Check and default constraints are derived from database schema, and the other three are specified by users. The other five constraints used for test data generation will be discussed in Section 2.3.

- A check constraint is a logical expression specified for a column specified in database schema, e.g., $18 \leq \text{age} \leq 70$, and all the data values in this column must satisfy this expression. We can derive important boundary values from logical expressions such as 18 and 70.

- A default constraint provides a default value to insert if no values are explicitly specified. Default values are identified as important test values.

- Specific-value constraints add special values to test data sets. For instance, requirements may stipulate objects whose statuses are equal to 3, 4, and 5. Then for the status column, we can add 3, 4, 5 as specific values. BIT-TAG derives Unicode characters such as Chinese characters if no Unicode characters are included in the original data source. This is important for testing products that have worldwide customers.

- Statistics-based constraints are used to derive important test values by performing statistical analysis over the original data source. BIT-TAG counts the numbers of appearances (frequencies) of distinct test values for each input. By default, BIT-TAG selects the most frequent value, the least frequent value, and a value with a random frequency between the lowest and highest frequency values.

- Logic constraints are additional logical expressions derived from requirements from SQL *where* clauses. Similar to check constraints, BIT-TAG extracts values from logical expressions to satisfy the *where* clauses. For example, if all active projects are required to be selected in a script, and the *where* clause is "isActive = 1," then 1 is selected.

It is crucial to write constraints in a format that allows people to easily specify them and BIT-TAG to easily parse them. We use JSON[4] as the basic format to design the data structure for each constraint. JSON is a lightweight data format and has been widely used in industry. We specify the constraints for each table in a separate file. The detailed specification for each constraint will be presented in Section 3.2.

We consider database data types as three general categories, numeric, date-time, and string. Numeric types include Boolean, integer, float, double, etc. Date-time types include date, time, datetime, timestamp, etc. String types include char, varchar, text, etc. We create partitions for the general categories of data type as follows.

- If an input domain is numeric, the range from the minimum value to the smallest derived test value forms a block, and the range of the smallest derived test value to the second smallest derived test value forms another block, and so on.

- If an input domain is of a date and time data type, the partition is created similar to the process for a numeric data type. The only difference is that there could be a separate block that has a null value, which is not between any two dates. An empty string is automatically converted to a default date.

- If an input domain is of string data type, each distinct test value is a block and the rest of the domain is another block.

If a statistics-based constraint derives additional values 22 and 60 on top on the aforementioned blocks over an integer domain, [min_int], (min_int, 0], and (0, max_int], then (0, max_int] is further divided into (0, 22], (22, 60], and (60, max_int]. BIT-TAG does not derive more values if 22 and 60 are already selected before the statistics-based constraint is applied. However, this situation is rare for non-binary types in practice. For example, we may create a partition: [min_int], (min_int, 0), [0], and (0, max_int), and [max_int], given the min_int, 0, and the max_int. If the data type of a column (e.g., employee role) is string, and the derived test values are "manager", "engineer" and "director," then these three values are three blocks, and all other string values form another block.

After IDMs are constructed, we select test values from each block. If the data type of the domain is numeric, we select the max value of each block. If the data type of the domain is date and time, we select the max value of each block, as well as the null value if it exists. If the data type is string, we choose the sole value of each block since each derived test value is a block. For the remainder of the domain, we select the empty string from it if this value has not been selected. If the empty string has been selected, we select a random string. Note that other strategies could be used to select test values from each block.

2.3 Test Data Generation

In this section, we discuss foreign key, logic, density, unique key, combinatorial coverage, and test set size constraints used for test data generation. Section 2.3.1 shows the constraints. Section 2.3.2 describes an algorithm to generate test data with the constraints.

[4]http://www.json.org/

2.3.1 Test Data Generation Constraints

We use density constraints to control the size between table objects, foreign key and unique key constraints to ensure data validity, and use logic and combinatorial coverage constraints to achieve a well-defined test data coverage. The five constraints about test generation are explained as follows.

- Foreign key constraints define referential relationships between tables. BIT-TAG automatically extracts foreign key constraints from database schema. Sometimes foreign key constraints are purposely omitted due to performance concerns, then they can be manually specified in JSON.

- Density constraints apply when tables have foreign key constraints. In a table, a foreign key column can be referred to as a parent column and a primary key column can be referred to as a child column. For example, if the table "Studies" has a primary key column "StudyID" and a foreign key column "ProjectID," "ProjectID" is a parent column and "StudyID" is a child column. This indicates that a project could have one or more studies. If each project has only a few studies, the density is low. If each project has thousands of studies, the density is relatively high. If we have such constraints across multiple table objects at different hierarchies, then the constraints determine the size of the test data sets to be generated. For example, we can use this constraint to define how many projects are there, how many studies are in each project, how many subjects are in each study, and how many records are in each subject, etc.

- Usually we do not want to apply combinatorial coverage to all the columns in a table since doing so could generate too many rows. We usually specify a combinatorial coverage criterion such as *Pair-Wise (PW)* coverage to two or three critical columns that are used in join or filter conditions of SQL scripts. PW requires one value from each block for a characteristic to be combined with a value from each block for another characteristic.

- Logic constraints are additional logical expressions derived from requirements about SQL *where* clauses. If a logical coverage criterion such as *Predicate Coverage (PC)* and *Clause Coverage (CC)* [2] is specified in constraint configuration files, BIT-TAG analyzes the expressions to generate test values to satisfy the coverage. Note that PC requires a test set to evaluate the predicate to true and false. CC requires a test set to evaluate every clause to true and false.

- The test set size constraint specifies a number of rows to be generated in a test set of a table.

Users may or may not add density, combinatorial coverage, logic, or test set size constraints for a table. If they do not specify any of these constraints, BIT-TAG just uses the test values derived from IDMs. In this case, *Each Choice* coverage (EC) is applied. EC requires each value from each block for each characteristic of the IDMs to appear at least once in the test data. Since specific-value and statistics-based constraints may still be used, a column to which a specific-value constraint is applied may have more values than another column to which no constraints are applied. To apply EC, we may have to generate additional values for the columns that have few values from their IDMs. If a column has a unique key constraint, we generate new values randomly; otherwise, we re-use the existing test values from the IDM.

Table 1: The Studies Table

StudyID	Active (DC)	Active (no DC)	ProjectID
s1	0	0	$p1$
s2	1	0	$p1$
s3	0	1	$p2$
s4	1	1	$p2$

2.3.2 Test Data Generation Algorithm

When density, combinatorial coverage, logic, or test set size constraints are applied, we use the test data generation algorithm in Algorithm 1 to address the constraints.

First, we consider foreign key constraints. The row number for a table #rows is set to the maximum integer. We calculate an ordered list of tables using topological sorting of the foreign key constraints. We start generating test data for a table without foreign key constraints or dependencies, followed by the tables that have dependencies.

Second, we generate test values for density constraints before we consider combinatorial coverage and logic constraints. Otherwise, we have to re-process combinatorial coverage and logic constraints because density constraints could generate more test values, which may not satisfy the combinatorial coverage and logic constraints. We take values from the IDMs for the child and parent columns and generate test values for the parent columns first. If there is more than one parent column, BIT-TAG uses ACTS to generate a data set to satisfy all combination coverage. If users specify a specific number between 1 and the number of total combinations, the user specified number of instances are randomly selected from the complete set of combinations for the parent columns; otherwise, all the combinations are used. Then we generate values for the child columns, which are usually primary key columns. If the required IDs for the child columns are more than what we have in the IDMs, BIT-TAG randomly generates more IDs. #rows is updated with the number of child instances.

Third, we generate test values for columns to which combinatorial coverage is applied using ACTS. BIT-TAG understands the columns, test values for each column, and the coverage criterion and then passes them to ACTS to generate test values. If a density constraint exists, the combinatorial coverage is applied to all child objects of each parent instance. It is important that the number of the child objects for each parent instance is greater than the number of the values generated by the combinatorial coverage; otherwise, extra values could be discarded. #rows is updated with the number of the combinations.

Fourth, we generate test values to satisfy logic constraints. Similar to combinatorial coverage constraints, if a density constraint exists, the logic constraint is applied to all child objects for each parent instance. Otherwise, the logic constraint is applied to all existing rows of the table. In the current implementation, if a logic constraint specifies the predicate coverage, the test data for a table is expected to have one half of the rows to evaluate the predicate to true and the other half to evaluate the predicate to false.

Last, if the test set size constraint specifies a larger number than #rows, we generate additional rows. For each extra row, we randomly re-use test values from the AIDMs for each column. If unique key constraint is applied to a column, BIT-TAG generates new values randomly.

One important decision in Algorithm 1 is to handle combinatorial coverage and logic constraints when density constraints exist. We give an example to show how Algorithm 1 handles logic constraints with the presence of density constraints. The same idea is

applied to the handling of combinatorial coverage constraints. Assume we have a table that has a primary key column "StudyID," a foreign key column "ProjectID," and another column "Active" that indicates whether a study is active. In addition, we have a density constraint that requires each parent column "ProjectID" instance to have two children column "StudyID" objects. Then the project $p1$ has studies $s1$ and $s2$ and the project $p2$ has studies $s3$ and $s4$, as shown in Table 1. If we have a logic constraint that specifies PC over column "Active," BIT-TAG generates test values to satisfy PC for each parent instance, $p1$ and $p2$, as shown in the second column in Table 1 (DC means density constriants). However, if no density constraints are specified, BIT-TAG could generate test values in the third column. All the tuples in the table satisfy PC but the tuples for each project do not. This approach ensures PC is still satisfied no matter which project instance is filtered out in join operations.

Algorithm 1 Test Data Generation Algorithm

Require: A set of AIDMs, one for each table
Ensure: A set of test data TD for each table
1: calculate an ordered list of tables T based on topological order of the foreign key constraints between tables
2: **for** each table $t \in T$ **do**
3: $\#rows$ = INT_MAX
4: **if** a density constraint DC exists **then**
5: generate test values for parent and child columns
6: **end if**
7: **if** a combinatorial coverage constraint CCC exists **then**
8: **if** a density constraint exists **then**
9: **for** each parent instance $p \in DC$ **do**
10: generate test data so that all tuples for p satisfy CCC
11: **end for**
12: **else**
13: generate test data so that all tuples in t satisfy CCC
14: **end if**
15: update $\#rows$ with the current row number
16: **end if**
17: **if** a logic constraint LC exists **then**
18: **if** a density constraint exists **then**
19: **for** each parent instance $p \in DC$ **do**
20: generate test data so that all tuples for p satisfy the coverage in LC
21: **end for**
22: **else**
23: generate test data so that all tuples in t satisfy the coverage in LC
24: **end if**
25: update $\#rows$ with the current row number
26: **end if**
27: **if** a test set size constraint $TSSC$ exists **then**
28: **if** the row number of $TSSC > \#rows$ **then**
29: add extra rows from AIDMs
30: **end if**
31: **end if**
32: save the test data into TD
33: **end for**
34: **return** TD

2.4 Adaptive Input Domain Model

In this section, we discuss how BIT-TAG expands IDMs with additional information to create Adaptive Input Domain Models (AIDMs). AIDMs make it possible to update IDMs when a change arises without reprocessing the original data source. The statistics-based constraint is the only constraint that requires to process the original table data (excluding database schema). When applying this constraint, we select test values based on their frequencies. So BIT-TAG saves distinct test values with their frequencies for each column for each table and these are referred to as the analytical results in this paper. Later on when new data is added or the statistics-based constraint changes, BIT-TAG can re-use the saved analytical results to select the new values. In addition, all constraints including the schema-based constraints are saved in corresponding data structures. Thus, we do not need to re-process the original data source to get the database schema, either. Therefore, AIDMs include IDMs, the analytical results, and constraints. All the data are saved in corresponding data structures in JSON files.

If data is changed, we compare the new data with the saved analytical results. Medidata has an internal tool to automatically collect latest data changes and save them as special JSON data sets. The syntax of the JSON structure includes key information such as *type*, *changes*, *when*. *type* specifies the type of a change, update, create, or delete. *changes* specifies the actual changes on a column including the column name, the old value and the new value. *when* gives the time for the change. By analyzing these data sets, BIT-TAG knows what data is old and what data is new, without reprocessing the original data source. If a value is added and it does not exist in the analytical results, this new value with the frequency of one is added. If there is one change to a value that has existed in the analytical results, the frequency of the value is modified, or the value is deleted. Then BIT-TAG may derive different test values to create IDMs. For example, if the values of the most frequent and least frequent distinct test values are affected, the derived test values from statistics-based constraints may differ.

When constraints change, BIT-TAG finds out which columns are affected by the changes and only updates the constraint data for the affected columns in AIDMs. The IDMs for unaffected columns remain the same. BIT-TAG may derive different test values from the new constraints including check, default, specific value, logic, and statistics-based constraints. Note that when statistics-based constraints change, we derive new test values from the saved analytical results. Then BIT-TAG uses the newly derived test values to create IDMs. When the schema of a table is changed, BIT-TAG has to re-generate AIDMs for this table.

BIT-TAG takes the same test data generation approach, with the new foreign key, density, combinatorial coverage, and test set size constraints. Testers often need to add extra foreign key constraints when the database is missing some because the testers were unaware of those foreign key constraints at the start.

3. IMPLEMENTATION OF BIT-TAG

Section 3.1 describes the architecture of BIT-TAG. Section 3.2 presents the specifications of constraints. Section 3.3 discusses the most recent implementation and development of BIT-TAG.

3.1 Architecture

Figure 1 shows the overall architecture of BIT-TAG. BIT-TAG consists of two phases, the initial cycle and subsequent cycles. In the initial cycle, BIT-TAG collects and analyzes the original data source and initial constraints. BIT-TAG saves analytical results and derives important values to create IDMs for every input. Then BIT-TAG creates AIDMs by combining the IDMs with analytical results. An effective test set is generated from the AIDMs by satisfying combinatorial coverage and other constraints.

In the subsequent cycles, we deal with three types of change. First, new data coming from customers. Second, existing con-

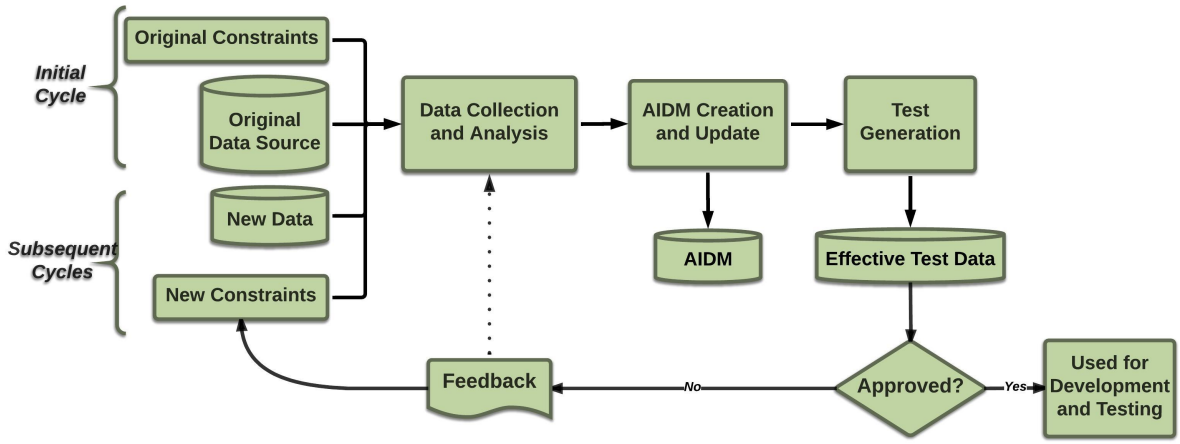

Figure 1: The Architecture of BIT-TAG

straints are modified or new constraints are added. Practitioners often need to update constraints in agile development as they understand more about requirements. Third, reviewers return feedback about the previously generated effective test data sets. Reviewers could include product analysts, architects, developers, and testers. Then the constraints are updated based on the feedback. Therefore, when the data or constraints change, BIT-TAG collects and analyzes the changes and updates the existing AIDMs. A new effective test data set is generated from the new AIDMs, independent of the prior test data set.

3.2 Specifications of Constraints

During agile development of ETL applications, we focus on analysis of requirements of data transformation. While analyzing the requirements, we need to understand how each column in the source is mapped to each column in the target. We also need to understand the detailed transformation logic, e.g., how to transform a column of the Integer type in the source to a column of the Boolean type in the target? In addition, when we perform a join operation between two or more tables, we need to understand the join and filter conditions, i.e., the conditions used in the *on* and *where* statements. Domain experts and architects usually construct the mapping information from the requirements. BIT-TAG users derive constraints from the requirements, along with the mapping information. Next, we explain each constraint in detail and discuss how to specify each constraint.

For statistics-based constraints, BIT-TAG saves distinct values with their frequencies for each column. If the original data source is very large (e.g., in Terabytes), it may take a long time to process the data and a lot of space to save the analytical results (when there are a lot of long distinct values). BIT-TAG could save a small subset (e.g., 10%) whose values are chosen randomly from the original data source. Then the saved partial data should have similar statistical distribution as the original data source. Note that we do not need to record data of some columns such as UUIDs. UUIDs in the original data source are not important for testing and we can generate new UUIDs in test data sets. In addition, saving them usually takes a lot of disk storage. By default, BIT-TAG derives the most frequent, least frequent test values and another value with a random frequency. Users can specify another number that is greater than three to derive more test values. The extra values besides the default selection are chosen randomly.

BIT-TAG automatically extracts check and defaults constraints from database schema. If they do not exist in database schema and users would like to specify them in JSON, check constraints can be specified as logic constraints and default constraints can be specified as specific-value constraints. Each table has a JSON file that specifies constraints. The types of constraints are explained in the next paragraph.

Statistics-based, specific-value, foreign key and unique constraints are specified in a *column* attribute, which includes *columnName*, *referredTableName*, *referredColumnName*, *specificValues*, *statisticsValues*, and *isUnique*. A density constraint includes *child*, *parent*, *maxChildren*, and *numParentInstances*. The *child* attribute is the child column ID (the primary key). The *parent* attribute shows the parent column IDs (with foreign keys). The *maxChildren* attribute gives the maximal number of child instances included by each parent instance. The *numParentInstances* attribute presents the number of parent instances. A combinatorial coverage constraint includes *columns* and *covType*. *covType* specifies a coverage criterion applied to *columns*. A logic constraint includes *expression* and *genType*, which specifies a logic expression and a logical coverage criterion to apply. A test set size constraints includes an integer to specify the number of rows of the test set to generated.

3.3 Implementation and Usage

We have implemented the combinatorial test data generation approach in a Java based software application, BIT-TAG. Some detailed information is listed in Table 2. Currently BIT-TAG is able to read database data and schema from Microsoft SQLServer and MySQL databases. To generate test values to satisfy combinatorial coverage, we integrate ACTS [18] into BIT-TAG. With the efficient algorithm, IPOG [19], ACTS can generate a test data set to satisfy t-way coverage pretty quickly. Potentially, satisfying t-way coverage when t is large ($t \geq 6$) and there are lots of test values for each input parameter could take a long time. However, in our usage, we usually apply t-way coverage to no more than four input parameters. Because we modestly and smartly select test values from constraints, test values for each input parameter are not many. Satisfying t-way coverage using BIT-TAG does not take a long time. Detailed information can be found in Section 4.

To improve the code quality, we have used JaCoCo [14] to measure the statement and branch coverage for BIT-TAG. To avoid potential faults and improve readability, we applied CheckStyle [6]

Table 2: BIT-TAG

Language	#Files	#Lines
Java	154	18,411
XML	21	1,381
Others	2	350
Total	**177**	**20,142**

and SonarQube [25] static analysis tools. In addition, we also created the Continuous Integration (CI) environment for BIT-TAG on Jenkins [16].

When a new test data set is generated, expected values in test oracles are likely to change. Testers have to manually write new expected values. To reduce the changes on the expected values, BIT-TAG uses the same seed for random methods. For security and privacy reasons, BIT-TAG sanitizes identifiable information of real customers such as emails when extracting data from original data sources. BIT-TAG can be used for big data applications that use databases and have constraints and business rules on data objects.

In the current implementation of BIT-TAG, most of the constraints (except foreign key constraints) are used to describe data relationships and combinations between no more than one table. Currently we are not able to specify special business rules that involve more than two tables. For example, consider tables A and B which are joinable and undergo the join operation. After the join of A and B, tuples that have the same value for a column in B must have the same value for a column in A. To handle such special business rules, we provide public APIs to users. The users can programmatically make BIT-TAG API calls to access the AIDMs and generated test data for each table object. Therefore, the users can implement the business rules by modifying the test data. The problem of how to manually derive constraints from requirements in a systematic manner and optimize data structures of constraints is an ongoing research topic. An in-depth discussion about this topic is out of the scope of this paper.

4. EXPERIMENTS

The objective of the experiments is to examine if we can use test data sets generated by BIT-TAG to replace data sets generated by existing approaches. Currently, there are three possible approaches, including 1) use the original data source, 2) select test data randomly from the original data source, and 3) generate test data sets manually.

We did not compare to manually generated data sets because the manual data generation is an ad-hoc approach. The manual data sets may differ to a great degree, as different developers could generate different test data sets, depending on their experience and domain knowledge. The development phase also has an effect in that the data set used in a later phase is supposed to be more effective than the data set used in an earlier phase since developers keep updating the data set as they have better understanding of requirements and find faults or deficiencies in the program.

We call the original data source and random test data sets traditional data sets. Specifically, we study the efficiency and effectiveness of BIT-TAG, compared to the traditional data sets. The efficiency is defined in terms of time of executing ETL programs, and the effectiveness is defined in terms of faults found.

4.1 Experiment Design

We show the subject systems first in Section 4.1.1, discuss three kinds of experimental test data and faults in Section 4.1.2, and present the experimental procedure in Section 4.1.3.

Table 3: The *Cloc* Report for the project A

Language	#Files	#Lines
SQL	219	33,516
XML	5	4,697
C#	2	334
Others	5	294
Total	**231**	**38,841**

Table 4: The *Cloc* Report for the project B

Language	#Files	#Lines
SQL	12	3,801
Java	42	2,466
Javascript	2	1,091
XML	14	557
Others	9	586
Total	**79**	**8,501**

4.1.1 Experimental Subjects

To evaluate the effectiveness and efficiency of data sets generated by different approaches, we used two real-world enterprise-level ETL projects from Medidata as our subjects. Due to non-disclosure agreements, we cannot show project names, customer names, or other detailed information about the products. Thus, we use *P_A* and *P_B* to refer to these two projects.

P_A extracts, transforms, and loads Microsoft SQLServer databases. For each customer, we have a separate database. In the experiments, we selected three databases from hundreds of customer databases to represent three categories in size: small, medium, and large. These three database are named *A-DB-1*, *A-DB-2*, and *A-DB-3*, and their sizes are 14 GB, 140GB, and 1.4TB, respectively. These three databases are often used in development sandbox environments. This project primarily uses Microsoft *SQL Server Integration Services (SSIS)* for data migration and transformation. The development of the SSIS packages are under the Microsoft Visual Studio environment.

The project B extracts and transforms data from a MySQL database and loads the transformed results into *Amazon Simple Storage Service (AWS S3)* in the CSV format. The original MySQL source used for *P_B*, referred to as *B-DB*, has 3 GB data. Unlike *P_A* that has separate databases for different customers, *P_B* has only one database for all customers. This project uses Pentaho [9] as the business intelligence and analytics solution for data transformation. Pentaho provides various widgets for developers to quickly perform tasks for ETL work-flow jobs such as copying files, validating XML files, truncating tables, writing logs, Sqoop exporting [11], and Pentaho MapReduce, etc. On top of the built-in widgets, developers can easily write SQL scripts and programs in another language such as Java to accomplish other general purpose operations.

Table 3 and Table 4 show the number of lines for *P_A* and *P_B*, measured by a line counter, *Cloc* version 1.6.2 [10]. The first three columns of Table 5 show the sizes and row counts of the core tables used for *P_A* and *P_B*. Out of 613 tables, *P_A* uses 19 core tables, which have a total of 386 columns. Out of 126 tables, *P_B* uses 22 core tables, which have a total of 257 columns.

4.1.2 Experimental Data Sets and Faults

We chose original data sources and randomly selected data (i.e., random data sets) as traditional data sets to compare with BIT-TAG

Table 5: Original and Test Databases

Databases	Size	Row Count	Test Database	Size	Row Count	%Size	%Row Count
A-DB-1	$14GB$	$47,284,837$	T-A-DB-1	$17MB$	$2,093$	0.12	0.004
A-DB-2	$140GB$	$126,350,671$	T-A-DB-2	$17MB$	$2,093$	0.012	0.002
A-DB-3	$1400GB$	$3,744,577,981$	T-A-DB-3	$17MB$	$2,093$	0.001	0.00006
B-DB	$3GB$	$4,107,768$	T-B-DB	$2MB$	$1,540$	0.07	0.04

data sets. We will analyze each data source and give the reason that we did not use data sets generated manually. Developers started with manually generated data sets (a common practise in industry as mentioned in the introduction) but soon they abandoned these data sets because it was difficult to manually update data sets with frequent constraint changes. They started using data sets generated by BIT-TAG with the help of BIT-TAG developers.

The original data sources used in the experiments are data in the sandbox environment used for development. These data sources are replicas of the production data at one time. Though the original data sources may not have the latest customer data, they are usually considered to be the most comprehensive in terms of constraints and relationships covered across different tables. So using the original data sources is supposed to find nearly all potential faults. The drawback is that the original data sources have high volumes of data and take a long time to process. Even though we will eventually deploy and run ETL programs in production against the original data sources, we do not want to run the original data source every time with frequent changes.

If we randomly select a small amount of data from a database, it is likely that no results are returned after join operations when the selected data do not satisfy foreign key constraints. Thus, the data set could not be very effective. If we randomly select many data from each table, the data could satisfy foreign key constraints but they may still take too long to process in agile development. We would like to see if a random data set has the same effectiveness as the BIT-TAG data set, with the same size. Currently, we do not have a tool that randomly selects data that satisfy foreign key constraints. So we just randomly selected data from the original data sources, with the same number of rows as those in the BIT-TAG data set for each table.

For the effectiveness, we identified faults based on the fault list of *P_A* and history changes of *P_B*. *P_A* has nine documented faults but *P_B* does not have faults recorded. Similar to many empirical studies that apply mutation analysis [2] where synthetic changes to source code are treated as potential faults, we treated the past changes in *P_B* as potential faults. The rationale is that we believe an effective test data set would cause developers to find faults in the existing code base and make changes to the source code. Ideally, when developers make a new change to the core transformation, running the test data would render a different result from a prior code commit. So we checked the changes to the 12 core transformation steps that have SQL scripts on the Github repository of *P_B*. Faults were counted by SQL scripts by commits. In each code commit, all the changes made to a SQL script are considered to be a fault. We examined every fault to make sure it is not semantically equivalent to the original program. We tested every faulty version to make sure executing it using the original data source rendered a different result from the original program. We counted 123 non-semantically-equivalent faults in the core transformation steps for *P_B*.

4.1.3 Experimental Procedure

We ran the experiments as follows.

1. We read and understood the requirements of *P_A* and *P_B*, with help from product analysts and developers.
2. We created the constraints for each table of the databases used in *P_A* and *P_B*.
3. We generated test data sets using BIT-TAG for *P_A* and *P_B* and recorded the sizes of the test data sets. We used *T-A-DB-1*, *T-A-DB-2*, *T-A-DB-3*, *T-B-DB* to refer to the test data sets generated by BIT-TAG from the corresponding original data sources.
4. We generated random test data sets from the original data sources of *P_A* and *P_B* and recorded the sizes of both the original data sources and random data sets.
5. We ran the ETL jobs in each project using the original data sources, saved the results as the expected values, and recorded the execution time.
6. We injected one fault into the corresponding project at one time. Ran the ETL job with the injected fault using the original data source, saved the results as actual values.
7. We compared the expected and actual values. If they differ, and if using the original data source detected this fault and recorded this fault.
8. We repeated the steps 5-7 using the BIT-TAG and random test data sets.

We conducted the experiments on a 64-bit Windows Server 2012 R2 platform, with 8 Intel i7-3770 (3.40GHz) processors, 32 GB RAM, and 5TB hard drive. We used Microsoft SQLServer 2012 and MySQL 5.6 for *P_A* and *P_B*, respectively.

During agile development of ETL programs, developers could make frequent changes to the core transformation jobs as their understanding about the requirements evolves. BIT-TAG plays a critical role in such the agile development since it allows a new test data set to be quickly generated when such changes take place.

The key to specification of constraints is to develop a good understanding about the requirements and the product databases. After such understanding is developed, actual specification of constraints does not take much time. This has been the case in our experiments. Note that business rules about a product database are relatively stable, while the actual data may be frequently changed in the database. In addition, database schema-related constraints are automatically extracted. These factors help to reduce the time and effort needed for constraint specification.

4.2 Experimental Results

Table 6 shows the constraints and business rules we manually specified and automatically derived from databases for *P_A* and *P_B*. We use "-" to represent that it is not applicable to specify constraints. The developers of BIT-TAG specified constraints and generated test data sets but did not know the faults. We added specifications for some of the foreign key constraints and unique key

constraints because they are missing in *A-DB-1*, *A-DB-2*, *A-DB-3*, and *B-DB*. For statistics-based constraints, BIT-TAG extracts three values by default. We did not give extra specifications. BIT-TAG applied combinatorial coverage criteria when they are specified in the combinatorial coverage constraints. For other columns where constraints are not specified, EC is applied by default. We also implemented some business rules using the APIs of BIT-TAG.

Table 6: Constraints and Business Rules for Each Project

Constraints	Manual		Automated	
	P_A	P_B	P_A	P_B
Foreign key	19	51	17	0
Check	-	-	2	0
Default	-	-	111	44
Specific value	21	40	-	-
Logic	10	2	-	-
Statistics-based	-	-	-	-
Density	9	6	-	-
Unique key	-	13	5	38
Combinatorial coverage	9	6	-	-
Test set size	-	2	-	-
Business rules	3	13	-	-

We did not measure the time for translating requirements into constraints because most of the time was spent on understanding requirements and product business rules. Once we understood the requirements and business rules, the translation did not take much time. In addition, it was difficult to separate the time for the translation from the time for understanding the requirements. Practitioners usually start coding while they are understanding requirements in agile development. Through a few agile development cycles, the practitioners fully understand the requirements. So it is hard to measure the time for learning the requirements.

Table 5 shows the sizes of the test data sets generated by BIT-TAG, compared with the original data sources. The last two columns show the percentages of the size of the test data set over the size of the original data source in terms of disk storage and row count. T-A-DB-1, T-A-DB-2, and T-A-DB-3 have the same size, even if their original data sources (i.e., A-DB-1, A-DB-2, and A-DB-3) vary from 14 GB to 1.4 TB. This is because we used the same constraints to generate the test data. A-DB-1, A-DB-2, and A-DB-3 have the same database schema. For the same product with different customer databases, we can use the same constraints for the test data generation. The test values may vary in each test data set because statistic-based constraints may extract different test values based on frequencies from each customer database. The test data sets are very small compared to the original data sources. T-A-DB-3 is only 0.001% and 0.00006% of A-DB-3 in terms of the disk storage size and row count, respectively. Since random test data have the same row counts as the BIT-TAG test data set, we do not show them.

Table 7 shows the generation time for the test data sets by BIT-TAG. "m" represents minutes. "s" represents seconds. "h" represents hours. "d" represents days. "w" represents weeks. "d" and "w" are used in Table 9. We measured the time for generating initial test data sets and subsequent test data sets using BIT-TAG. We measured the time five times and calculated the averages. Generating the initial test data sets could take a long time because this needs to process original data sources. Even if generating T-A-DB-3 took up to 18 hours from A-DB-3, this happened for only one time. Generating subsequent test data only took 1-2 minutes when we changed constraints. We used the built-in random method of SQL to generate random test data sets for each table from every original data sources. Generating random test data sets took dozens of minutes from A-DB-3 and took a few minutes from other databases.

Table 7: Time for Test Data Generation

Test Databases	Time (First)	Time (Second)
T-A-DB-1	$3m30s$	$2m$
T-A-DB-2	$40m$	$2m$
T-A-DB-3	$18h$	$2m$
T-B-DB	$2m$	$1m$

Table 8: Faults detected

Project	#Total Faults	#BIT-TAG	#Original	#Random
A	9	9	9	0
B	123	123	123	0
Total	**132**	**132**	**132**	**0**

Table 8 shows the faults detected by using the original data sources, BIT-TAG test data sets, and random test data sets. BIT-TAG test data sets found all the faults detected by the original data sources. We ran the ETL jobs of P_A and P_B with the random test data sets and they did not find any faults because the random data did not satisfy the foreign key constraints.

Table 9 presents the execution time for running ETL jobs of P_A and P_B using the original data sources, the BIT-TAG test data sets, and random test data sets. We can see running the ETL jobs of P_A and P_B using the original data sources took a long time, while using the BIT-TAG test data sets took very little time.

In summary, the BIT-TAG test data sets found all the faults detected by the original data sources. Running the ETL jobs in P_A and P_B using the BIT-TAG test data sets only took a very small fraction of the time of using the original data sources. BIT-TAG is more efficient than using the original data source. BIT-TAG has the same effectiveness as using the original data source and is more effective than the random test data generation. Since BIT-TAG is very effective in finding faults and efficient in generating and executing test sets for P_A and P_B, we believe BIT-TAG can be successfully used in agile development of big data applications. The developers and stakeholders have given very positive feedback about BIT-TAG. We are in a process of improving the usability of BIT-TAG and guiding the developers to use BIT-TAG. We will continue using BIT-TAG in future ETL projects.

4.3 Threats to Validity

As in most software engineering studies, one external threat to validity is that the subject programs may not be representative. We ameliorated this threat by selecting real-world subjects from industry with a variety of sizes and kinds of databases. Another threat in the experiments could be that we translated the requirements into constraints. Other people may specify different constraints. Once testers fully understand requirements and product business rules, they should come up with similar constraints even if they may dif-

Table 9: Execution Time

Original Databases	Time	Test Databases	Time
A-DB-1	$1d$	T-A-DB-1	$3m$
A-DB-2	$3d$	T-A-DB-2	$3m$
A-DB-3	$2w$	T-A-DB-3	$3m$
B-DB	$3h$	T-B-DB	$30s$

fer sightly. So this should not affect the experimental results much. Another external threat is that we used BIT-TAG and ACTS to generate test data sets. The faults in these two tools may affect the results. As described in Section 3, we have applied best practices of Java programming when developing BIT-TAG. ACTS has been used by many users for a long time. One construct validity threat is that we used real faults and historical changes as faults. Using synthetic faults may generate different results. However, we believe using real faults and actual changes should better reflect the effectiveness of our approach.

4.4 Lessons Learned

We summarize the following major lessons learned from our experience.

- Identification of constraints requires a good understanding about the requirements. Ideally all the important information should be clearly documented, which is however not the case in practice. In the course of this project, we held many discussion sessions with subject experts to clarify our understanding. We ran into situations in which an ETL script didn't return any results because some requirements were not converted into constraints. Many constraints were discovered and/or corrected during those discussion sessions as we obtained a better understanding about the requirements.

- Not all columns interact with other columns. Thus, it is often not necessary to achieve uniform t-way coverage among all the columns. In our experiments, we achieved pairwise coverage for columns that are involved in the same business rules, and all-combination coverage for columns that have many-to-many relationship. For columns that are not involved in any business rule, we covered them to achieve each-choice coverage. Doing so has significantly reduced the size of the generated dataset without compromising test effectiveness.

- When PC is specified for a logic constraint, we generated half of the rows in a table that evaluate to true and the other half to false. This seems redundant since PC only requires two rows, one evaluating to true and the other to false. However, when tables are joined over with filtering, i.e., with a *where* clause, this redundancy helps the remaining rows to still satisfy PC.

5. RELATED WORK

In this section, we discuss previous research on database-oriented test data generation and compare it to our work. Chay et al. [7] presented an approach that uses input space partition to generate test data for traditional database applications. They use a manual approach to create IDMs where synthetic data is used. In contrast, our approach generates IDMs automatically and derives test values from the original data source.

Olston et al. [24] presented an approach that generates example data sets for testing dataflow programs. In their approach, an example data set is first created by sampling the original database and then refined by propagation and pruning in multiple passes. Li et al. [21] improved the work in [24] to deal with user-defined functions. Specifically, they derive symbolic constraints from dataflow operators and use concolic execution [12] to solve those constraints. In both approaches, the user needs to provide equivalence classes for each dataflow operator. This requires a detailed understanding about the semantics of each operator, which may not be possible for general programs. Also, both approaches require access to the source code. In contrast, our work generates test data from the original data source and requirements, and it does not require access to the source code.

Li et al. [20] reported an approach that uses clustering to generate test data for data-centric applications. The main idea is to group similar records in a database together and compute a representative record for each group based on the notion of centroid object. In order to preserve test coverage, their approach performs static analysis to determine the weights of attributes in a database, which is needed to compute the distance between different records. As a result, this work also requires access to the source code, which is different from our work.

We note that many approaches generate large data sets to evaluate the performance of big data computing platforms [1, 3, 4, 13, 23, 26, 27]. These approaches are fundamentally different from test data generation approaches whose goal is to find faults that may exist in big data programs.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we report our experience of applying a combinatorial testing approach to two real-world ETL applications.This approach automatically creates models, i.e., AIDMs, from the original data source. AIDMs extend traditional input domain models (IDMs) to include analytical results that capture important characteristics of the original data source and constraints. Based on the AIDMs, our approach generates a data set to satisfy t-way combinatorial coverage and other constraints. When there is a change in the data source and constraints, our approach updates the AIDMs by comparing the saved analytical data with the change and generates a new test data set. Our results show that the test data set generated by our approach detected the same faults as the original data source, taking a fraction of the time executed by the original data sources. This suggests that combinatorial testing can be effectively applied to big data applications.

In the future, we plan to improve BIT-TAG in the following directions with our own interests. First, we plan to optimize the performance of existing constraint handling and also include support for new types of constraints to better capture requirements. Second, we will conduct experiments on more and even larger data sets, and compare BIT-TAG with other approaches described in Section 5. Third, we plan to use machine-learning techniques to identify important test values from the original data source. Machine-learning techniques could also be used to optimize AIDM creation and test data generation by learning from previously generated data sets. Fourth, we would like to refactor BIT-TAG to support more data types such as spatial types. Fifth, BIT-TAG could be improved to reuse existing data sets to generate a new data set when constraints change. Sixth, generating a minimal (in terms of row counts) test data set to satisfy all constraints is an interesting problem to study.

7. ACKNOWLEDGMENTS

We offer our sincerest gratitude to Isaac Wong, Anthony Escalona, Susan Finley, and Angela Sloan for supporting this research, and Neha Tyagi and Donald Lancaster, for helping us set up experiments. Lei's work is partly supported by a research grant (70NANB1 5H199) from National Institute of Standards and Technology.

8. REFERENCES

[1] A. Alexandrov, K. Tzoumas, and V. Markl. Myriad: Scalable and expressive data generation. *Proc. VLDB Endow.*, 5(12):1890–1893, Aug. 2012.

[2] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge, UK, 2008.

[3] A. Arasu, R. Kaushik, and J. Li. Data generation using declarative constraints. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 685–696, New York, NY, USA, 2011. ACM.

[4] N. Bruno and S. Chaudhuri. Flexible database generators. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 1097–1107. VLDB Endowment, 2005.

[5] R. C. Bryce, C. J. Colbourn, and M. B. Cohen. A framework of greedy methods for constructing interaction test suites. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 146–155, May 2005.

[6] O. Burn. Checkstyle. Online, 2001. http://checkstyle.sourceforge.net/, last access April 2016.

[7] D. Chays, S. Dan, P. Frankl, F. Vokolos, and E. Weyuker. A framework for testing database applications. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages Pages 147–157, Portland, Oregon, USA, 2000.

[8] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, July 1997.

[9] P. Corporation. Pentaho. Online, 2004. http://www.pentaho.com/, last access April 2016.

[10] A. Danial. CLOC. Online, 2006. https://github.com/AlDanial/cloc, last access April 2016.

[11] A. S. Foundation. Apache Sqoop. Online, 2011. http://sqoop.apache.org/, last access Sept 2014.

[12] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[13] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, pages 243–252, New York, NY, USA, 1994. ACM.

[14] M. Hoffmann, B. Janiczak, E. Mandrikov, and M. Friedenhagen. Jacoco code coverage tool. Online, 2009. http://eclemma.org/jacoco/, last access April 2016.

[15] H. Hu, Y. Wen, T.-S. Chua, and X. Li. Toward scalable systems for big data analytics: A technology tutorial. *Access, IEEE*, 2:652–687, June 2014.

[16] K. Kawaguchi. Jenkins. Online, 2011. https://jenkins-ci.org/, last access April 2016.

[17] D. R. Kuhn, D. R. Wallace, and J. A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, June 2004.

[18] R. Kuhn, R. Kacker, and Y. Lei. Automated combinatorial testing for software (ACTS). Online, 2009. http://csrc.nist.gov/groups/SNS/acts/, last access May 2015.

[19] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. Ipog/ipog-d: efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability*, 18(3):125–148, 2008.

[20] B. Li, M. Grechanik, and D. Poshyvanyk. Sanitizing and minimizing databases for software application test outsourcing. In *Proceedings of the 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, ICST '14, Cleveland, Ohio, USA, 2014.

[21] K. Li, C. Reichenbach, Y. Smaragdakis, Y. Diao, and C. Csallner. Sedge: Symbolic example data generation for dataflow programs. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 235–245, Palo Alto, CA, USA, Nov 2013.

[22] N. Li, A. Escalona, Y. Guo, and J. Offutt. A scalable big data test framework. In *IEEE 8th International Conference on Software Testing, Verification and Validation*, Graz, Austria, April 2015.

[23] Z. Ming, C. Luo, W. Gao, R. Han, Q. Yang, L. Wang, and J. Zhan. Bdgs: A scalable big data generator suite in big data benchmarking. In *Advancing Big Data Benchmarks*, volume 8585 of *Lecture Notes in Computer Science*, pages 138–154. Springer International Publishing, 2014.

[24] C. Olston, S. Chopra, and U. Srivastava. Generating example data for dataflow programs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD'09, Providence, Rhode Island, USA, 2009.

[25] SonarSource. Sonarqube. Online, 2001. http://www.sonarqube.org/, last access April 2016.

[26] E. Torlak. Scalable test data generation from multidimensional models. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 36:1–36:11, New York, NY, USA, 2012. ACM.

[27] P. Zhang, S. Elbaum, and M. B. Dwyer. Automatic generation of load tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 43–52, Washington, DC, USA, 2011. IEEE Computer Society.