Building Practical Big Data Services for Knowledge Driven Software Verification

Generating Optimal Test Sets from Complete Customer Data
for Compliance Applications Based on Logic Coverage

Cem Ünsal Anu Sreepathy
Saikat Mukherjee Dave Hanekamp Gang Wang
Intuit, Inc., 2700 Marine Way, Mountain View CA, 94043 USA
cem_unsal@intuit.com

ABSTRACT

This short manuscript introduces a cloud-based solution to provide significant reduction in software application re-platforming and regression testing efforts. By taking advantage of existing customer data, internal tools, off-the-shelve cloud services, we designed and implemented a big data service that can respond to financial data queries at great speed and scale. By pre-processing large scale data using our customized software as well as readily available cloud services, we were able to speed up QA efforts while guaranteed optimal coverage of application logic in software code.

CCS CONCEPTS

Software and its engineering → Software creation and management → Software verification and validation → Empirical software validation; Software defect analysis → Software testing and debugging

KEYWORDS

Financial software development, map-reduce, automated test case generation, code coverage

1 INTRODUCTION

Quality assurance in software testing is one of the most important processes in compliance domain. Any bugs, albeit small, may lead to significant errors in a document that is returned to specific agency for processing. In financial software development, the errors in code may lead to large deviations in calculated amounts which are very costly to fix after production release, and especially after submission of financial data to the agency in charge.

Having an absolute guarantee on the correctness of the software is paramount when adapting to changes over the months/years and/or making significant changes to the application platform.

This guarantee increases customer confidence and provides a lasting brand recognition for the company providing the application.

In this manuscript, we document one of the internal services we implemented to enable our personal tax software services and applications to move seamlessly from one software stack to another and from one year's financial rules to the next year. Both, stack/platform/model changes in our code base and year over year difference in tax documents introduce a constant need for software development efforts for our company. Most of these efforts are supported by automation and intelligent systems; however, there is always a need to provide a faster and more robust process to adapt to changes in our code base. An incremental updates to our automated processes directly affect our software's quality and our company's baseline.

One of the most important issues in compliance software testing is the need to guarantee that all edge cases have been covered. In other words, we want to make sure that when the new software platform, new data model or new version of our code is introduced, we do not inject any error or miss any scenario that is feasible in our financial software package.

It would be clearly unacceptable for a company that tries to provide the best tools for its customers to prosper financially, to fail calculating their taxes, monthly cash flow, or anything pertinent to their personal or business finances accurately. Hence, the extreme desire for us to make sure that our software 'does not miss anything.' However, defining a test set that covers all possible scenarios is not an easy task. Our method solves this problem.

Moving a financial software product that is being actively used by millions of customers from an existing trusted platform/model to a new relatively untested platform/model is a big adventure. Code coverage and similar methods are used for critical software packages (e.g., in aviation, health, and financial domains) where a single bug may create drastically hazardous consequences. The idea is to make sure that there is not a single line of code left unchecked; we want every program statement to be observed in action [1]. Fixing errors after release is costly; after submission/processing by government agency, it becomes much costlier and decreases our products' net promoter score (NPS).

We will give examples from a specific application to Tax Software, namely our DIY tax products, TurboTax Desktop and TurboTax Online, and highlight our approach to ensure calculation accuracy. However, the method described here is applicable to other applications and software platforms where customer data is readily available.

The method described in this document is based on a few assumptions, which we will describe in detail later in subsequent sections. These are:

1. We are able to instrument our calculation engine, i.e., we must be able to observe what our code is executing while processing user data and record it for further processing
2. We are able to create simpler representations of this data by mapping the output of the instrumented engine to unique representations
3. We can correlate and query these unique representations of the 'behavior' of user data in our code base at scale.

Based on these assumptions, we hope to scale down the size of the test data significantly, as described in subsequent sections. We see our approach to finding minimal test sets as a 'code coverage' method, since we based the data sets on logic behavior of our user data in our code. In the following sections, we will discuss how we implemented the capabilities listed above, as well as our data flow model and software stack. We will include few examples illustrating the advantages of this approach, and conclude with discussions.

Note that the user data we process has been stripped of any personally identifiable information, including dates, which may or may not affect the calculations in some code sections. Copies of financial data files are processed and destroyed as soon as they are decrypted on server instances.

2 INSTRUMENTING OUR ENGINE

The first step in implementing our 'code coverage' method is the instrumentation of our existing code base – our core tax engine in this case), i.e., adding logging capability to specific areas in code, such as decision points, method entry/exits, function returns, etc.

Borrowing ideas from static program analysis [2] (and currently lacking the capability to do a fully-detailed static code analysis), we aim to run our core engine using previously recorded user data. For example, for a tax software product, we can take previous year's return data, re-process all the input lines with our 'instrumented engine' which will print out every line that we visit in code for every user file we input. We call these resulting output files 'trace logs.'

Note that the data we use to re-run out core engine has been successfully submitted earlier, i.e., we know that the data in the file is a *correct representation of a single user scenario*. This is a big advantage over a data file that is created synthetically.

Trace logs created with this instrumented version of our core tax engine enables us to identify:

- Loops
- Code branches due to conditional statements
- The exact condition that causes a specific branch
- Different assignment cases for variables
- Different outputs of specific functions

We aim is to instrument/log as many decision points as possible. A list of potential candidates for logging/instrumentation are as follows. The examples in Table 1are based on decision and branch points in the software we are currently re-platforming.

Table 1. List of Instrumented Functions

Type	Description
ABS_xxx_[P|N]	Absolute value
ASSIGN_xxx_[P|N|B|Z|T|F|V]	Value assignment
CND_JMP_xxx_Y_yyy	Conditional jump to new line
CND_JMP_xxx_N_0	Conditional jump to next line
CSE_JMP_xxx_OTHER_yyy	Case jump
CSE_JMP_xxx_v_yyy	Case jump where v in [0, +]
EQU_xxx_[T|F]	Equals
GTE_[EQ|GT]_xxx_[T|F]	Greater Than or Equal
GT_xxx_[T|F]	Greater Than
IN_xxx_v	Value in list, matching v; v in [0, +]
LGL_CND_xxx_[T|F]	Logical condition
LTE_[EQ|LT]_xxx_[T|F]	Less than or equal
LT_xxx_[T|F]	Less than
MATH_ABS_xxx_[B|N|P]_B	Absolute value
MATH_[ADD|SUB]_xxx_[B|P|N]_[B|P|N|Z]	Addition/subtraction of various value pairs
MATH_DG_xxx_[P|N_[P|N]	Get digit
MATH_DIV_xxx_[B|N|P]_[B|NP]	Division for various value pairs
MATH_[MIN|MAX]_xxx_[Z|N|P|B]_[Z|N|P|B]	Minimum/maximum of two values
MATH_MUL_xxx_[B|N|P|Z]_[B|N|P|Z]	Multiplication of two values
MATH_NEG_xxx_[P|N|B]_[P|N|B]	Negation with input/output pair
MATH_[RD|RU|RO]_xxx_[P|N|B|Z]_B	Round down/up function
[MIN|MAX]_xxx_v	Min/max with selection (v in [1,2])
NCND_JMP_xxx_zzz	Non conditional jump from xxx to zzz
NCND_JMPB_xxx_yyy	Non conditional jump back (yyy < xxx)
NEQ_xxx_[T|F]	Not equals
xxx: instruction line, yyy, zzz: line to jump Y | N: yes | no, P | N | Z: positive | negative | zero, T | F: true | false, B: blank, V: value (string)	

We normally add the assembly code line number where code is encountered and the line number of the assembly where the jump is occurring. Our trace logs also include information about the specific tax form to which the code section belongs, a count of repeated entries into the same code section, and number of times we visit a specific line. The last value enables us to identify loops with different counts, which we later separate into three main groups: no visit (0), single visit (1), and multiple visits (>=2). The format of the data rows is as follows:

```
userid, form, section, copynum, line, count
24175863214, FCONFIG, QUALIFY_DISCLOSURE, 0, EQU_42_F, 1
```

Figure 1. Raw Data Format

With these specifications, we create fairly large files (5MB to 14MB in size) listing all the lines of all the sections of active code base for a single customer data. (Note that the all PII information on these files have been removed, and only the input line data relevant to our calculations are kept.)

We repeat this process of re-running our core tax software for all the financial data files we collect in the previous year(s).

At the end of this process, we possess a large set of files listing each and every code line visited by our customer base and detailing the behavior of each customer data set in our code base.

Assuming, there are no sections that are 'dead' in our code base and our previous year data includes enough diversity, we can assert that the files we obtained has at least one representative example of each and every branch of our original code base. On other words, based on these two assumptions, we will have full 'code coverage'.

The process of re-calculating the tax return data for every customer we had in previous year(s) requires a large number of resources. Re-running a return through our instrumented (and scaled down) core engine takes usually between 10 to 25 seconds, with a 15secs median. Almost all returns are completed under 35secs.

Given the large number of personal tax returns Intuit processes (~ 40M), Total processing time is approximate 15sec*40M = 600M secs = 6944 days.

In our earlier runs, by using 30 AWS EC2 instances with 36 virtual cores, running 28 to 32 instrumented processors, total processing time for generating instrumentation logs was reduced to 7 to 8 days, when the load is distributed fairly evenly between servers.

3 IDENTIFYING 'FOOTPRINTS" FOR EACH USER DATA SET

Once we obtain the trace logs for 'behavior' of each user's financial data file, we would need to scale this data down to a size that can be efficiently queried. Without any additional processing, it is possible

to query these logs and obtain a list of lines per filer/customer and code section. However, the total data size is close to 80TB (for all filers with federal and state forms) and even a single code section query returns a very large number of lines to be processed.
As mentioned earlier, we are not directly interested in what the exact lines for each filer is; we are rather interested in 'behavior' of each filer's data in our code base.
For example, if we want to know if user #123 and user #124 have the same behavior (i.e., visiting the same code decision points/branches and exiting the same way) for a specific code section, all we have to do is check whether the trace logs for both filings for users #123 and #124 have the logs with the exact same lines in the exact same order[1].
This similarity will enable us to say that both of these user data files are causing the same business logic to fire at the same code points.

Proposition 3.1: Assuming:

1. the order the lines of trace logs are kept (or all lines are sorted in the same fashion) for all files
2. log lines are concatenated in the same fashion for all data files
3. hashing function used for the concatenated log strings does not lead to collisions

Then, two data files that have the same hash value are said to be behaving exactly the same in our code base, in other words, they visit the exact same lines of code and exit them the exact same way.
Proof:

- Conditions 1 and 2 are easy to guarantee; we just use the same code during processing of all files and make sure that sorting function in AWS services (such as Athena) are behaving the expected way (we have seen that sorting combined aggregate functions over large data sets is not deterministic).
- The number of possible branch combinations, even for the largest code section we have (3000+ lines) is still significantly smaller than the simplest standard hashing function, so collisions are not observed.

Since we only care about exact similarities in code execution, the same check between different filers can be done with much shorter data: by using a hash function. If we define a simple hash function for a string that is a concatenated representation of all trace lines for a section in the order they are logged, the resulting hash would suffice to identify similarity between two filers/data files. Then, by comparing hash values of two filers for the same set of sections, we can detect if they belong to the same group (of similarly behaving user data files/sets) or not. This grouping will enable us to filter a very large number of filers from our test samples while guaranteeing 'coverage'.
Figure 2 shows a matrix visualization where each trace log file is represented as a column and each code line (in every section) is represented as a row. This matrix will have the number of visits from the raw data file in each cell. For every column of this matrix, we can create a unique hash for the values in cells corresponding to a specific section. This value will be the only data we save for (user id, code section) pairs for querying. We call this hash value the 'footprint' of the customer for a specific section. We can then visualize a large matrix where the columns are the users and the rows are the code sections, and the cells contain a single hash value.

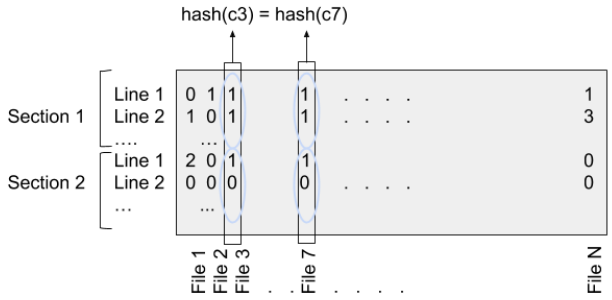

Figure 2. Matrix representation of (user file, section) to hash mapping.

This matrix would be very large (3000+ columns and ~40M rows) but, it is significantly smaller than keeping the full list of trace lines for each section.
Our earlier data size for this version was ~13TB with MD5 hashes. Our trace logs usually include sections with 10 log lines to 2500 log lines; hence the reduction. Note that we do keep the 'raw' data for the trace logs in our permanent storage, in order to enable our analyst to better understand errors they may encounter during testing.

3.1. COMPLETENESS FOR CODE COVERAGE

Without a full static code analysis, it is impossible to claim that we have 'full code coverage'. However, with this method, we can claim the following:

- Assuming we have enough diversity in our customer base, we can say that we are very close to complete code coverage
- If there is a code section that is not been visited in the last *n* years by any of our customers, then that code section is probably obsolete and can be discarded.
- It is always possible, using our internal tools, to synthetically create user data that will force visits on the sections of the code that haven't been visited by organic data.

3.2 PRE-PROCESSING

Given that we have approximately 40M user data files for personal tax returns, more than 2000 forms represented in our code base, and a fairly large number of instrumented functions (see Table 1), the total number of lines generated by instrumented financial engine is fairly large. Once the 'raw' data (Figure 1) for each user is saved into a single file, the next phase of the off-line process is to transform these data rows into a single line representing the behavior of a single user (data) in a single code section, across all ~40M log files. Note that every tax form translates to 10 to 200+ different sections in our code base.

[1] Note that we do not care about the actual values of user data; we only care about the lines in code that were executed because of these specific values.

The process starts with many rows of the data shown in Figure 1 and generates single row data representing multiple of these rows in the following format:

```
userid, code section, hash value
24175863214, FCONFIG:QUALIFY_DISCLOSURE:0, 34FB56A021E329CCA376
```

Figure 3. Final Data Ready for Query

It needs to be parallelized due to large number of data files/rows. In order to transform multiple lines per section into a single hash, we use AWS EMR with Spark to create this mapping:

(userid, code section) → hash representation

We employ Spark cluster of 30 to 60 servers to process 250K-400K user ids at a time. This chunking provides small enough number of files to be stored in AWS S3 in CSV format that can be quickly uploaded into AWS Athena service.

When a set of user id files are processed, they are 'partitioned' into sections to enable fast SQL queries. Python code that enables this in EMR/Spark is given below:

```
inputFileDF = spark.read.format('com.databricks.spark.csv'). \
                options(header='false',delimiter=',').load(inputPathList)
# replace counts
newCountCalc = F.when(F.col("countPasses") > 2 , 2). \
                  otherwise(F.col("countPasses"))
inputDFWith01M = inputFileDF.withColumn("count01M", newCountCalc)
# copy number
dfcopynum = inputDFWith01M.groupBy("fileid", "formid", "sectionid"). \
                agg(F.min("copynum").alias("copynum"))

df_reg = inputDFWith01M.select("fileid", "formid", "sectionid", "copynum",
                    F.concat_ws(":", "lineId", "count01M").alias('prehash1'))
df_reg_hashed = df_reg.groupBy("fileid", "formid", "sectionid", "copynum"). \
          agg(F.hash(F.concat_ws(",",F.sort_array(F.collect_list("prehash1")))). \
          alias("hashed"))
df_joined_reg = df_reg_hashed. \
    join(dfcopynum, ["fileid","formid","sectionid","copynum"]). \
    select("fileid","formid","sectionid","hashed")
df_joined_reg.repartition("formid", "sectionid").write. \
      partitionBy("formid", "sectionid").format("csv"). save(outputFilePath)
```

The code snippet here shows the three main steps of our Spark process. We load a set of files of suitable size for processing with 30-60 server instances, cap the number of line visits at 2 (which corresponds to 'multiple entry'). We then concatenate all lines in a section in to a single string before creating a single hash value per (userid, section pair). Final step in the MR process is slice the rows per (form and) section and save it in comma separated format.

Note that the input data files contain information about each user; however, after processing, output data files are grouped/partitioned by (form and) section, which requires us to carefully track which user data file has bene processed. In case of an error, re-processing a specific user id is not a simple task, once the results for a specific user has been processed by EMR and uploaded to S3 bucket in files representation section partitions. Our data flow employs a DynamoDB table to make sure that we track each and every data file in the flow properly.

In EMR/Spark, Each data set (of 250-400K user ids) are processed in 10-20 minutes depending on the number of rows.

3.3. FOOTPRINT RESULUTION: "TIERS"

Once, we process every user's data file with our core engine and upload the raw data, i.e., trace logs (per user) into AWS S3, the EMR/Spark process can be run separately. Saving the this raw data, enables us to re-process it to generate a different footprint (i.e. hash value per user per section) if there is a need.

We actually take an advantage of this two-step processing model to create different 'tiers', which are different ways of selecting log lines in the raw data to construct the hash function input.

When we take all the lines that we traced into account while generating the hash values, we create a tier (called 'Reg01M') where all branches, math functions, decisions are represented. This tier gives us significant reduction over our earlier method of statistical sampling (based on distribution of values for each line of the form, not described in the manuscript). For an average tax form, as seen in Figure 6, we have multiple orders of reduction in test set size using this tier.

Although 'Reg01M' tier is very good for automated regression tests, and guarantees that we track every method/branch/decision we care about, the size of the test set is not always useful for quick debugging/testing, especially during the re-platforming processing. Our analysts need smaller set to quickly identify potential errors. For this, we let our teams to tell us what they are interested in (branches only, or only assignment lines, or only a specific function), and filter the trace logs accordingly before generating the representative hash for (user, section) pairs. By limiting the lines we include in the hash, the test set number can be reduced significantly, in many cases multiple orders of magnitude.

We currently generate 4 such tiers, in addition to 'Reg01M', where we look at only assignment lines ('AO') or only the branch conditions ('NoAssign', in Figure 6).

It is possible to visualize these different tiers as the interest to track different instructions or phases of the code base. For example, if our analysts are looking at 'AO' tier, they are not interested in understanding why a specific 'if-else' branch has been taken, rather want to know if a value is assigned as positive/negative/zero, etc.

Different tiers with significantly smaller test sets, *while still guaranteeing full coverage* for the instructions of interest, enables quick checks for sanity and fast debugging. Our analysts then can expand to more complete tiers. Note that data for all tiers that we have listed are returned by our service at the same time.

4. QUERIES INTO DATABASE

Once the trace log data is processed by EMR/Spark as described in Section 3.2, it resides in an AWS S3 bucket, distributed into in multiple folders partitioned by code sections. The data files are comma-separate rows that include the following data:

userid, code section, hash value

The number of files depends on the data chunks that enters the EMR Spark processing at every step. We have 100 to 300 files with number of rows ranging from 10K to 450K for each code section. We surface this data as a SQL table by using AWS Athena [3], which enables us to create meta data definitions by pointing it to the S3 bucket containing all the data for a single fiscal year.

As seen in Figure 4, incoming queries that are trying to find a useful sample of data files with optimal coverage does not have the sections of code defined. Our system takes the queries which include the list of (form, line) pairs of interest, and transforms then by mapping (form, line) pair into a single code section. An earlier simple analysis of our code base provides us with this mapping which we keep in memory.

Once we get the list of sections we need to query, we use the Athena table to ask for a list of all user ids that are associated with these sections (the second column of our table). The list of user ids for a list of sections could be extremely long. For example, if there is a section in the code that checks the SSN of the filer, this section will be visited by all our customers and returns largest set of ids we can possibly have.

This list of user ids for each section is then grouped by their hash values. Since each hash value indicate a unique behavior (or set of branches, or business logic) in our code, we only need to provide one id from each group. For multiple section, we create hashes of hashes to find unique groups of behavior combinations.

This greatly reduces the number of files that needs to be tested or re-run, automatically or manually. The reduction in the number of test files directly affects the development time for the re-platforming effort.

The SQL query that is sent to AWS Athena service is given below. Note that we generate a new hash value for the given set of sections we are interested in, then combine on the server side:

```
fieldDataSplit = [k.split(":") for k in fieldData]
qbeg = """SELECT fileid,
        TO_HEX(
          MD5(
            CAST(
              array_join(
                array_sort(
                  array_agg(
                    concat( formid, ':', sectionid, ':', CAST(hvalue AS VARCHAR))
                  )
                ),
              ',')
            AS VARBINARY)
          )
        ) AS tothash
      FROM %s.%s
      WHERE """ % (self.athena_db, mode.lower())
qend = " GROUP BY fileid ORDER BY fileid"
qwhere = " OR ".join([ "( formid = '%s' AND sectionid = '%s' )" % (str(k[0]), str(k[1]))
for k in fieldDataSplit ])
return qbeg + qwhere + qend
```

The queries into Athena service usually return under 2 minutes. The longest query recorded is 1 min 50 secs for 37 code sections.

Creation of test set groups based on hashes returned is a simple step carried out on the server side.

5. DATA FLOW

Figure 4 below shows the flow and transformations of data in our system.

During the off-line processing phase, our data goes through the following transformations, all carried out in parallel:

- Download, unzip, decryption of XML data files
- Minor edit for re-processing
- Upload of data file, upload of required financial definition file for federal and state forms into instrumented engine, calculations run and generation of trace logs
- Clean up, debug and upload of trace logs to S3
- Hash generation for final data, partitioned by year, fed/state, form, section (for multiple tiers).

During online processing, the following steps are taken:

- Check submitted payload and record in task database
- In background, map payload to section query and send to Athena
- Athena loads required (partitioned section) files into memory and runs the query
- Receive resulting data from S3, generate groups (and create additional tiers, such as 'Reg01M_NoComb')
- Save results in S3 and serve client

Subsequent queries that carry the same payload will be served from cache/memory.

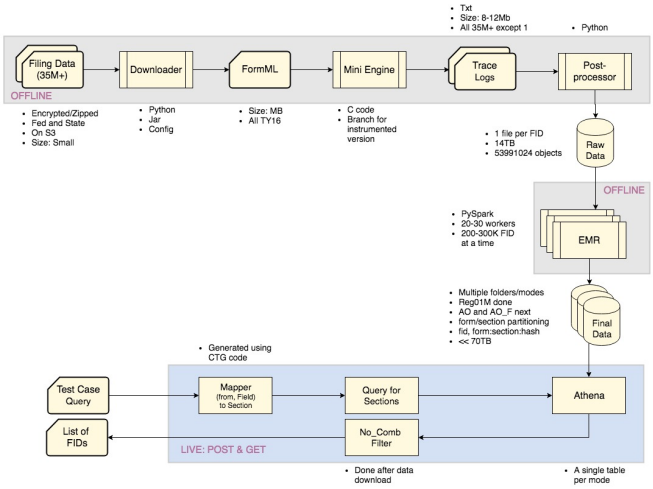

Figure 4. Data Flow in offline and live processes

6. SOFTWARE STACK

The largest task in this effort to bring large scale data into a form that can be queried fast was putting together a variety of code bases and platforms to enable offline and online data flow show in previous section:

- The initial data files we need to start our offline processing resides in AWS S3 service, encrypted. The file format is XML. We incorporated a Scala project and associated configuration files to enable on-the-fly data download. (We do not keep user data files in encrypted or decrypted format in local disk; every file is deleted as soon as it is processed, this code belongs to Data group at Intuit.)
- Instrumented core tax engine requires user data to be in a specific format, we edit the files minimally to make them work. Engine also requires code sets that corresponds to each user file. All required files are downloaded from Tax Group's code repository as needed. Instrumented engine is in C and provided by a special branch build by the Tax Group. We pre-load processing instances with required libraries via Chef scripts.
- All custom code and data file locations are part of our configuration files that bootstrap each processing instance. AWS S3 is where we keep all temporary data.
- Off-line processes user AWS DynamoDB to track status of the processes. We also have an MySQL server to track pre-calculated information on data location, etc. (Now running on AWS Aurora)
- Raw data to final data transformation is carried out in AWS EMR/Spark instances, with S3 as input /output points.
- Final data is queried via AWS Athena, where section-based partitioning and parallel processing enables fast queries over TB-scale data.

- Our ‘glue code’ is based on bash and Python; all instances on AWS EC2 are customized RedHat Linux instances.

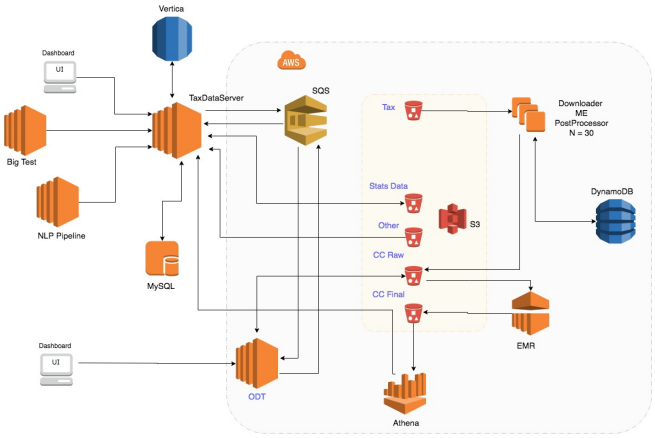

Figure 5. Software Architecture

7. RESULTS AND DISCUSSION

We have been running multiple instances of the service described here using user data from 2017 tax season for a few months, and the number of distinct sets we have seen for different tiers described earlier support our initial expectations about significant reduction in data set.

As seen in Figure 6, for a variety of tax forms we see 10x to 1000x reduction over statistical methods (results of which are also provided by our service via Vertica distributed database).

Note that the total number of users for these forms are in the 1-10M range. The actual reduction is much greater. For example, for form 2441, we simplify the test set ‘Reg10M’, that includes every trace log for every customer, from 3M+ to 63. Also note that, with this set of only 63 user data files, we guarantee that every active line of code in our code base is covered.

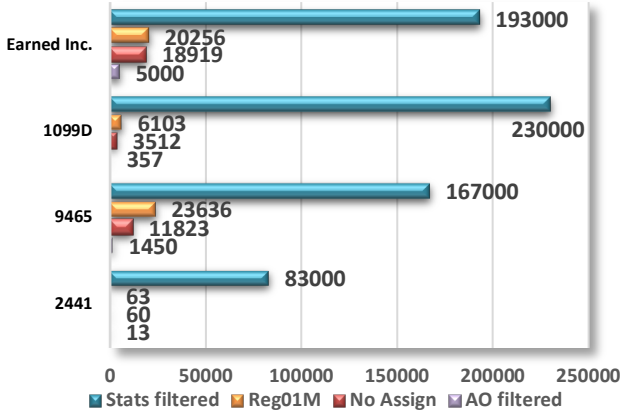

Figure 6. Resulting groups for different tax forms using different tiers.

Besides the large reduction in data size (which directly translates to reduction in time and labor costs), our method has the advantage of covering all possible scenarios. Statistical methods based on sampling are bound to miss edge cases. For example, when we look at the number of different groups (i.e. scenarios) and the number of user files in each group for a specific federal form, we observe the distribution in Figure 7.

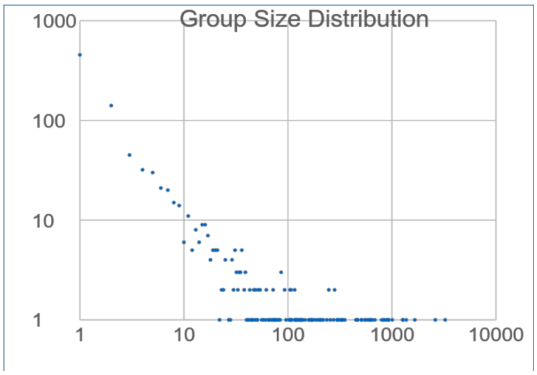

Figure 7. Group size for various number of groups for Form 3903.

As seen in this log-log plot, there are a very large number of groups where the number of users (data file) is simply 1. In our customer data, we do have a significant number of users where their specific financial situation corresponds to a unique flow in our code base. Our method encapsulates all such cases.

9. CONCLUSIONS

We presented a method merges off-the-shelve cloud services, existing internal tools and some custom code to create an efficient cloud service for large scale financial data, resulting in significant speed-up in software quality assurance efforts.

The method presented in this document to generate minimal test sets for application re-platforming and software regression tests has been implemented and released to two separate internal groups as an always-on service at Intuit [4].

We were able to transform large amount of data into a concise format that can be queried fast at scale. Based on approximately 2000+ various queries the service received, we are estimating 10 to 300 times reduction in test data size.

Our efforts to process most recent data for this year is continuing in a streaming fashion, and will be completed in October 2018, which is the deadline for this tax year. We expect to improve the off-line processing time by readjusting parallelization in the first phase and changes to map reduce code in second phase.

ACKNOWLEDGMENTS

Authors would like to acknowledge following individuals for their help in designing and instrumenting this system, and for providing valuable feedback on our service: Roger Meike, Steven Atkins Saneesh Joseph, Jeff Clyne, Daniel Bremiller, Mike Artamonov, and Regis Gimenis.

REFERENCES

[1] Code Coverage, Wikipedia, Inc., May 2018, https://en.wikipedia.org/wiki/Code_coverage.

[2] Static program analysis, Wikipedia, Inc., May 2018, https://en.wikipedia.org/wiki/Static_program_analysis.

[3] AWS Athena, https://aws.amazon.com/athena/, Amazon Inc, May 2018.

[4] Intuit, Inc., https://www.intuit.com/