SCOPE Playback: Self-Validation in the Cloud

Ming-Chuan Wu, Jingren Zhou, Nicolas Bruno, Yu Zhang, Jon Fowler
Microsoft Corporation
Redmond, WA 98052-6399, USA
{mingchuw, jrzhou, nicolasb, yugzhang, jonfo}@microsoft.com

ABSTRACT

The last decade witnessed the emergence of various distributed storage and computation systems for cloud-scale data processing. SCOPE is the distributed computation platform targeted for a variety of data analysis and data mining applications, powering Bing and other online services at Microsoft. SCOPE combines benefits of both traditional parallel databases and MapReduce execution engines to allow easy programmability. It features a SQL-like declarative scripting language with .NET extensions, and delivers massive scalability and high performance through advanced optimization. SCOPE currently operates over tens of thousands of machines and processes over a million jobs per month.

Such massive data computation platform presents new challenges and opportunities for efficient and effective testing and validation. Traditional approaches for testing database systems are not always sufficient due to several factors. Model-based query generation typically fails to provide coverage of user-defined code, which is very common in SCOPE scripts. Additionally, rapid release cycles in the platform-as-a-service environment require tools to quickly identify potential regressions, predict the impact of breaking changes, and provide massive test coverage in a short amount of time. In this paper, we describe a test automation tool, denoted by SCOPE Playback, that addresses these new requirements. SCOPE Playback leverages the SCOPE system itself in two important ways. First, it exploits data about every job submitted to production clusters, which is automatically stored by the SCOPE system. Second, the testing process itself is implemented as a SCOPE script, automatically benefiting from transparent and massive computation parallelism. SCOPE Playback currently serves as one crucial validation technique and ensures product quality during SCOPE release cycles.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; H.2 [**Information Systems**]: Database Management

DBTest'12, May 21, 2012 Scottsdale, AZ, U.S.A.

Keywords

SCOPE, Playback, Testing, Validation, Distributed Computing

1. INTRODUCTION

Companies providing cloud-scale data services have increasing needs to store and analyze massive data sets, such as search logs, click streams, and web graph data. For cost and performance reasons, processing is typically done on large clusters of hundreds or thousands of commodity machines. SCOPE (*S*tructured *C*omputations *O*ptimized for *P*arallel *E*xecution) [1] is a distributed computation system built for cloud-scale data analysis over tens of thousands of machines at Microsoft, powering Bing and other online services. SCOPE relies on a distributed storage system, named Cosmos, for storing and analyzing massive data sets. The storage system is an append-only file system optimized for large sequential I/O. All writes are append-only and concurrent writers are serialized by the system. Data is distributed and replicated for fault tolerance and compressed to save storage and increase I/O throughput.

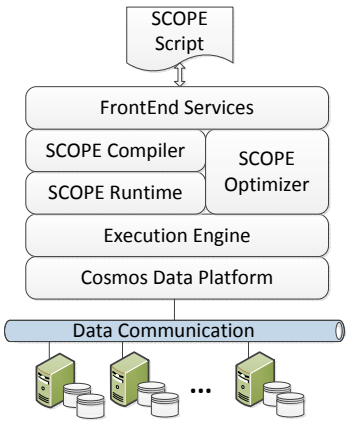

Figure 1: Architecture of the SCOPE /Cosmos Platform

Figure 1 shows the system architecture of SCOPE/Cosmos at a high level. The entire system runs as a service, serving tens of thousands of SCOPE jobs daily.

1.1 The SCOPE System

The SCOPE system combines benefits from both traditional parallel databases and MapReduce execution engines to allow easy programmability and deliver massive scala-

bility and high performance through advanced optimization. It consists of the SCOPE language and its compiler, the optimizer, the runtime and the execution engine. The SCOPE language is a declarative scripting language, resembling SQL, with integrated .NET extensions. The strong resemblance to SQL provides a single machine programming abstraction so that its users can focus on application logic rather than dealing with low-level details of distributed computations. At the same time, it enables transparent and systematic optimization. In addition to standard SQL constructs like joins, unions, and aggregates, users can easily define their own functions and implement their own versions of relational operators: *extractors* (parsing and constructing rows from a data source), *processors* (row-wise processing), *reducers* (group-wise processing), *combiners* (combining rows from two inputs), and *outputters* (formatting and outputting final results). This flexibility allows users to process both relational and non-relational datasets and solve problems that cannot be easily expressed in traditional SQL, while at the same time retaining the ability to perform sophisticated optimization of user scripts.

A SCOPE script goes through a series of transformations before it is executed on the cluster, as shown in Figure 2. Initially, the SCOPE compiler parses the input script, unfolds views and expands macro directives, performs syntax and type checking, and resolves names. The resulting annotated abstract syntax tree (AST) is passed to the SCOPE optimizer [4]. The optimizer returns an execution plan that specifies the steps to efficiently execute the script. A query plan is modeled as a dataflow graph: a directed acyclic graph (DAG) with vertices representing processes and edges representing data flow. Finally, code generation produces the final data-flow program, detailing the vertex execution and data dependencies among them, and assemblies containing user defined code. This package is then sent to the cluster, where the execution engine will take charge of scheduling, resource allocation/rebalancing, and monitoring the job progress.

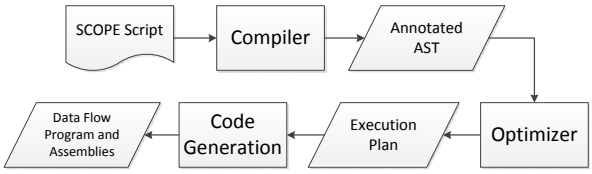

Figure 2: Compilation of a SCOPE Script

1.2 Test Requirements and Challenges

SCOPE /Cosmos is *the* big-data analysis platform for virtually all online services and various products at Microsoft. Many business critical applications and data processing pipelines rely on the functionality of the system. The number of daily jobs has doubled every six months for the past two years, and still continues to grow. Currently over a million scripts are executed monthly on SCOPE production clusters, processing several petabytes of data every day.

Analogously to many other online service providers, the SCOPE system is evolving rapidly with technical innovations and has frequent release cycles for customers to benefit from the latest development. Typically we release new versions of the system every 6 to 8 weeks. This strict requirement results in very short stabilization cycles, typically in the order of days rather than weeks or months like is the case for traditional software. It is critical for the business to make sure that the system functions and performs well from release to release. In fact, in such a live service environment, *stability* and *correctness* across releases are requirements rather than features.

The precise meaning of stability and correctness depends on each system component. In this paper, we focus on the compiler and optimizer components of SCOPE. In this context, correctness means that valid scripts can be compiled and optimized into a valid execution plan. Any code defect or regression may result in interruption of online service pipelines, immediately causing monetary loss.

In turn, stability refers to functional backward-compatibility, compilation latency, and overall plan quality. When software evolves, breaking changes are sometimes inevitable in order to fix legacy design or implementation problems. A goal of the testing infrastructure is to help identify individual scripts (and their corresponding owners) when breaking changes are introduced in a new release. This information enables us to implement an efficient engineering process for introducing breaking changes while minimizing any business impact.

We have originally –and exclusively– employed conventional test methods, such as hand-crafted unit-tests, model-based test-query generation [3], and *QueryBuilder* [2]. However, these tools do not provide sufficient test coverage on the .NET extensions and MapReduce-like extensions of the language. Such user-defined code is very common in SCOPE scripts. Thorough testing various user-defined code is important to prevent regressions on pipelines that are deployed into production.

1.3 SCOPE Playback

The goal of our testing infrastructure is to catch any correctness or stability issues early during release cycles. In this paper, we introduce a test automation tool, called SCOPE Playback, which was designed and implemented to meet the zero regression requirement for the compilation and optimization pipeline. As the system is running as a service, it observes and logs precise query workloads. SCOPE Playback leverages this information and validates the release under development by *playing back* query compilation and optimization with the previously submitted jobs. Additionally, SCOPE Playback takes full advantage of the SCOPE system itself to parallelize the test execution and automatically scale out test validation in response to increases of test load. All is done with minimum human intervention, and just hours of test turnaround time.

The rest of the paper is organized as follows. In Section 2, we present the design and implementation of the SCOPE Playback tool. In Section 3, we show some performance numbers and interesting statistics of our system. Finally, we will describe future work and conclude in Section 4.

2. DESIGN AND IMPLEMENTATION

The basic idea of Playback is to replay all compilation steps shown in Figure 2 on every previously submitted user script in a given time frame using a given version of the SCOPE compiler and optimizer. In the rest of this paper we denote the version of SCOPE that we are evaluating as the *test candidate*. As basic requirements, a Playback run needs

to cover at least a month of user scripts to provide sufficient test coverage, and the turnaround time needs to be in the order of hours to allow frequent validations at different stages of release cycles. There are over a million SCOPE scripts executed monthly. When counting both the scripts and associated resources such as pre-compiled user libraries, the data associated with such workload totals over 15 terabytes in size. SCOPE Playback therefore faces additional challenges in terms of time and space compared to traditional test infrastructures. To tackle these challenges, Playback leverages SCOPE itself to manage execution and achieve performance, reliability, and scalability with massive data parallelism.

2.1 Job Repository

One crucial property of the SCOPE system is that it observes and logs the entire user workload on production clusters as part of normal execution. Specifically, we maintain a *job repository* which stores information for every SCOPE job that was ever submitted to our production clusters. The job information is stored as a SCOPE table[1] with the following normalized schema:

```
JobRepository(date,jobGUID,resourceName,content)
```

Each SCOPE job is uniquely identified by a `jobGUID`, and consists of the following mandatory and optional resources: a SCOPE script (mandatory), referenced SCOPE views[2] (optional), referenced libraries (optional), and auxiliary runtime assemblies and data files (optional). In addition, in order to replay the optimization precisely, we also clone metadata and statistics of all the input datasets as additional job resources, and store them in the `JobRepository` table (this practice allows us to replay jobs whose input datasets are no longer available in the system). Each row in the `JobRepository` table represents one job resource of a given job identified by `jobGUID`. The table is partitioned and indexed by `date` and `jobGUID`.

The job repository maintains a complete history of the system workload, and can be used to understand how system utilization evolves over time. In addition to Playback, the job repository is also used for workload analysis tasks, such as data usage patterns, data dependencies between jobs, and full-text search on scripts, among others.

2.2 Playback using SCOPE

The main goal of Playback is to validate the compilation/optimization pipeline of the SCOPE system. To compile a given job from job repository, we need to reconstruct the original environment where the script was compiled, and then invoke the test candidate to compile the script. Conceptually, using the `JobRepository` table, we have to group the job data by `jobGUID`, and then pass each group to a *holistic* custom aggregate function which processes a group in its entirety. That aggregate function (i) reconstructs the scripts and resources, and (ii) invokes the test candidate to compile the script.

The `REDUCE` command in the SCOPE language provides a flexible way to implement custom grouping and aggregation and achieve the above goal. It takes as input a row-set that has been grouped by the columns specified in the `ON` clause, processes each group using the `Reduce` function of the reducer class specified in the `USING` clause, and output zero, one or multiple rows per group. The `Reduce` function in the reducer class is called once per group. It takes three parameters (lines 4 to 6 in Figure 4): the first one specifies the input row-set, the second one the output row object, and the optional third one the user specified argument in the script. Figure 4 shows pseudo code for the Playback reducer, named `PlaybackReducer`.

```
1. class PlaybackReducer : Reducer {
2.     ...
3.   public IEnumerable<Row> Reduce(
4.         RowSet input,
5.         Row outRow,
6.         string[] args)
     {
       // prepare a job folder for Playback
7.     string jobFolder = null;
8.     foreach (Row r in input.Rows) {
9.       if (!Directory.Exist(jobFolder)) {
10.        jobFolder=Directory.Create(r.jobGUID);
         }
         // save job resources to jobFolder
11.      WriteToDisk(r.content, jobFolder);
12.    }

       // unpack the private SCOPE package
       // invoke the private ScopeCompiler
       // in that package to playback
13.    Unpack(args[0]);
14.    Invoke(args[0], out outRow);

15.    yield return outRow;
     }
   }
```

Figure 4: Custom reducer for SCOPE Playback.

A sample SCOPE script that implements the Playback task using `PlaybackReducer` is shown below[3]:

```
RESOURCE "SCOPE.v1.bin";

Jobs = SELECT * FROM JobRepository
       WHERE date >= (DateTime.Today - 30) AND
             date < DateTime.Today;

Results = REDUCE Jobs ON jobGUID
          USING PlaybackReducer("SCOPE.v1.bin");

Failures = SELECT * FROM Results
           WHERE status != "succeeded";

OUTPUT Failures TO "/test/PlaybackReport.log";
```

[1]The job information is actually stored as a structured stream in the SCOPE system. A structured stream is similar to a table in traditional DBMSs and can be partitioned and indexed properly.
[2]Similar to a SQL view, a SCOPE view consists of stored SCOPE statements accessible as a virtual table.
[3]Reducers can also specify the schema of the output. For simplicity we omit such details in the description of `PlaybackReducer`, and assume that it produces additional columns (e.g., status) that provide information about each compilation and are filled out by the `Invoke` function in line 14 of `ScopePlayback`.

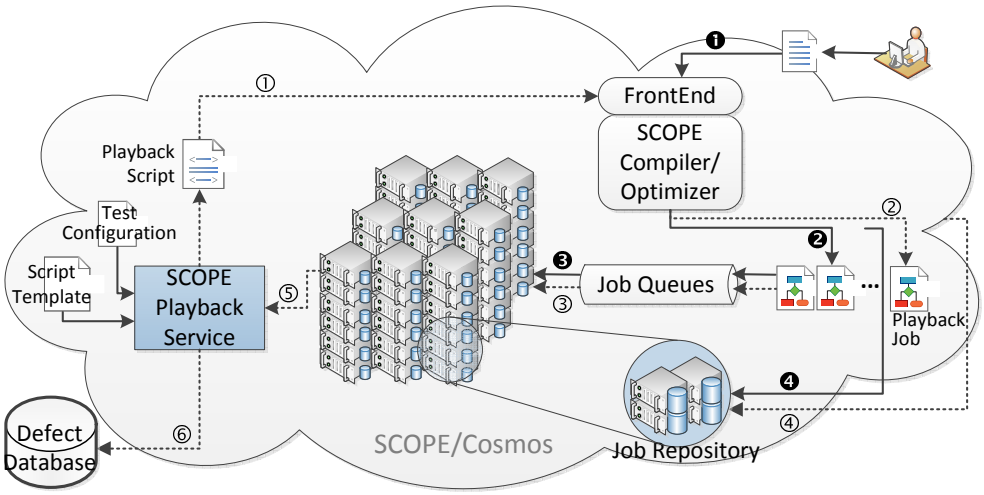

Figure 3: SCOPE **Playback Workflow Diagram**

The above script retrieves all the jobs from last thirty days. Then, it groups the data by `jobGUID`. Rows of the same group are passed to the `PlaybackReducer.Reduce` function in the parameter `input` (line 4 in Figure 4). It first reconstructs the job folder by saving all the resources into files in the previously created job folder (lines 7 to 12). Next, it unpacks the test candidate (line 13), specified in the argument to the `PlaybackReducer` (`"SCOPE.v1.bin"` in the above example). Finally, it invokes the specified test candidate to compile the script (line 14). The compilation status is passed to `outRow` and returned to its caller (line 15).

In this example, the subsequent `SELECT` statement in the script reports compilation failures only. In practice, all the playback results are outputted to a file. One can compose appropriate SCOPE scripts to meet different reporting needs.

2.2.1 *Playback Automation*

In order to keep the product free from regressions at any time during development, we also implement a Playback service to automate the process of submitting and monitoring Playback jobs. Figure 3 shows the high-level workflow of SCOPE job submissions, and running SCOPE Playback using SCOPE.

Before we describe the Playback service, we explain how user scripts are processed by the SCOPE system. When a user submits a SCOPE job, the script with all its resources are uploaded to the cluster (step ❶). Upon receipt, a front-end service will invoke SCOPE compiler and optimizer to compile the script. The final physical execution plan and the generated runtime package are sent to the job queue (step ❷). When computing resources become available, jobs will be dequeued for execution according to job priorities (step ❸). In the meantime, all the submitted jobs with all their associated resource files are collected into the job repository (step ❹).

The Playback service runs inside our production clusters. Based upon predefined intervals or events[4], it automatically kicks off Playback jobs. It reads a configuration file and the Playback script template to generate a custom Playback script which is then submitted to the cluster for execution, just like any other user jobs. It goes through the same steps to reach job queue and be dequeued for execution (steps ① to ③ in the figure). Similarly, the Playback job will be collected into the job repository (step ④), so that in any subsequent Playback run, we will also have test coverage on the Playback job itself. Once a Playback job is scheduled, the service will wait for the test results (step ⑤), and for any issues found, they will be entered into a defect tracking database by an automatic bug filing system (step ⑥).

2.3 Dual and Multi-Playback

The previous section describes the basic Playback implementation to validate functional backward-compatibility. However, there are advanced scenarios that require some extensions to the previous design to help evaluate new feature development. For instance, consider the task of finding out which scripts result in a different execution plan after a new optimization strategy is implemented. To achieve this goal we need to compare the output plans of the optimizer between two different versions of SCOPE. For another example, after introducing a new feature into the compiler, we would like to know its impact on compilation time. To achieve this, we need to compare compilation times between two SCOPE versions.

To address these scenarios we extend the Playback tool to take one or more test candidates. Specifically, the function `Reduce` of class `PlaybackReducer` in Figure 4 is slightly extended as shown in Figure 5.

The parameters in `args` contains the names of test candidates which are passed as parameters of `PlaybackReducer` in the script. The number of runs equals the number of test candidates (i.e., the size of the `args` array in line `13.a`). An additional array to hold the status of each compilation is defined at line `13.b`. The loop between lines `13.c` and `14.b` performs the following two steps for each test candidate defined in `args`. First, it unpacks the specified SCOPE binaries (line `13.d`). Then, it invokes the test candidate to compile the script (line `14.a`). Upon completion of the loop, the virtual function `ValidateResult` is called to validate the results (line `14.c`). Note that `PlaybackReducer` is now an

[4]One typical event that triggers the Playback job is whenever a SCOPE daily build is dropped. At Microsoft, a *build lab* is responsible for producing an executable binary for each product source tree on a daily basis.

```
 1. class PlaybackReducer : Reducer {
 2.     ...
 3.   public IEnumerable<Row> Reduce(
 4.         RowSet input,
 5.         Row outRow,
 6.         string[] args)
      {
        ... // line 7 ~ 12, same as previous

 13.a   int runs = args.Length;
 13.b   object[] status = new object[runs];

        // looping through all the private SCOPE
 13.c   do {
          // unpack the private SCOPE
 13.d     Unpack(args[runs]);

          // invoke the private ScopeCompiler
 14.a     Invoke(args[runs], out status);
 14.b   } while ((--runs) > 0);

        // custom logic to validate all results
 14.c   ValidateResult(status, out outRow);
 15.    yield return outRow;
      }
    }
```

Figure 5: Reducer Extensions for Dual Playback.

abstract class that can be parameterized by a specific implementation of the virtual function `ValidateResult`.

The delayed binding of the actual `ValidateResult` implementation allows different test intents to reuse most of the Playback infrastructure. For example, in order to understand optimizer plan changes, one may define a new class deriving from `PlaybackReducer` and override the function `ValidateResult` to perform plan comparisons. The following script snippet demonstrates how to identify optimizer plan changes between two test candidates:

```
Results = REDUCE Jobs ON jobGUID
          USING PlaybackOptimizerComparison(
                "scope.v1.bin",
                "scope.v2.bin");

#CS
public class PlaybackOptimizerComparison
                : PlaybackReducer {
   ...
   public ValidateResult(object[] results,
                         out Row outRow)
   {
   // plan comparison function
   }
}
#ENDCS
```

3. PLAYBACK IN ACTION

As described in the previous section, SCOPE Playback replays all compilation steps on previously submitted user scripts using one or more test candidates. In this section we show some results and experiences by using SCOPE Playback to validate our system.

In the past, we used a dedicated machine pool to run Playback as a stand-alone application (i.e., without leveraging the SCOPE system itself). The input data was partitioned and distributed to different machines manually. Then, we launched the Playback application on each of those machines. The total test turnaround time was about 14 days. Besides, it required heavy human intervention and monitoring to avoid unanticipated problems. As the test loads increased, we had to manually tune the partition buckets, increase the degree of parallelism, and add new machines into the test pool. As a result of such long turnaround time and heavy human intervention, it was very costly to do any private Playback runs before feature integration.

Today, SCOPE Playback runs daily on the official SCOPE build in all our production clusters. For each run, it covers over a million customer jobs, reading over 15 terabytes of compressed job data, and produces results in around 2.5 to 3 hours. During development, individual team members also schedule private Playback runs before integrating the features back into the main source tree. This validation step helps developers quickly identify regressions that are not caught by unit testing, and reduces stabilization time.

Our experiences show that SCOPE Playback not only results in a quick test turnaround time[5], but also exhibits very reliable test runs. The quick test turnaround time is a result of the high degree of data parallelism, which is automatically adjusted by the optimizer according to input characteristics. Robust test runs are a result of scalability and transparent fail-over provided by the SCOPE /Cosmos platform.

Figure 6 shows a fragment of the execution plan of one instance of the Playback jobs in SCOPE. First, data from `JobRepository` table is extracted and joined to an event table, `JobSubmittedEvent`[6], to obtain additional information, such as the job owner, the environment which the job was submitted to, and the job priority. Since event data is typically small, the optimizer broadcasts the small event table to `JobRepository` table to perform the join. As described in Section 2, data in `JobRepository` is sorted on `jobGUID`, a property leveraged by the optimizer to choose a merge-based join variant. The joined results are then partitioned by `jobGUID` so that all the rows with the same `jobGUID` are sent to the same processing node to perform the compilation step defined in `PlaybackReducer`. The results of `PlaybackReducer` is read by two consumers. The first consumer aggregates the results and produces summarized statistics about the all the jobs. The second consumer filters out the successful runs, and computes error messages and call stacks. Then, it repartitions the data by `stackHash` (i.e., a hash on the error message and call stack) so that a subsequent operator can remove duplicate instances of the same issue before it outputs the code defects for the automatic bug filing system.

At the time of this writing, and during the last 2-month release cycle, there were 165 Playback runs. Among those, 92 runs were private runs (*i.e.*, Playback jobs submitted by

[5]As a comparison, if we were to run Playback on a single machine, it would take approximately 240 days to complete a single run.

[6]The `JobSubmittedEvent` is a system table that records job submission events, including the job GUID, the job owner, the submission time, the job priority, *etc.*

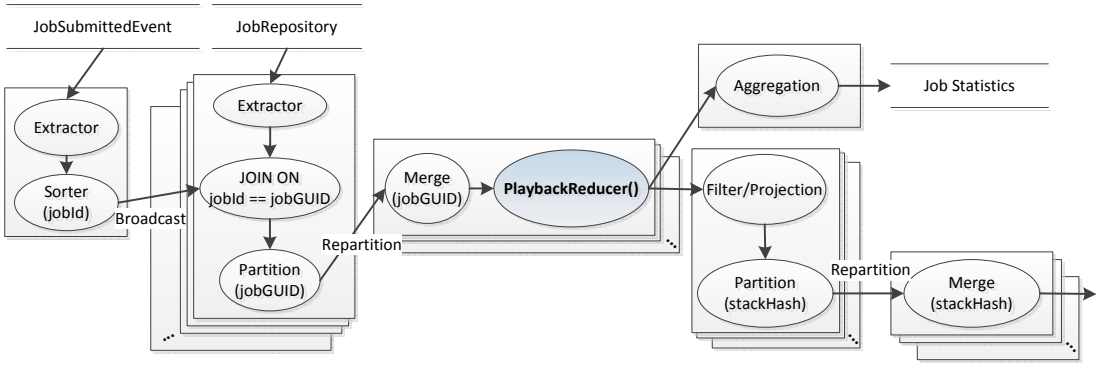

Figure 6: Playback execution plan in SCOPE

individuals). Table 1 summarizes the test coverage and total input data volume read by the Playback jobs. All the 165 runs cover over 100 million customer jobs and read over 1.5 petabytes of data.

	Daily Runs	**Private Runs**
# of Runs	73	92
Test Coverage	$\sim$ 75 million jobs	$\sim$ 30 million jobs
Data Volume	$\sim$ 1.1 PB	$\sim$ 415 TB

Table 1: Statistics of Playback Runs in a Two-month Release Cycle

We also observed that using SCOPE Playback to validate features before integration reduces the bug count in the stabilization period. The team is able to find out issues early in the release cycle, or even before integrating changes into the main source tree. Moreover, the regression in production environments in the compiler areas has dropped down to zero in the recent releases. The 100% test coverage on customer jobs does deliver the promise of zero regression on existing recurring jobs.

Another example of Playback usage is to manage breaking changes. As the SCOPE language evolves, whenever there was a breaking change, it also helped us identify the scripts and their owners. For example, when we had to align the `INNER JOIN` syntax with standard SQL, Playback helped us identify a single user using the non-standard syntax. We were able to contact the individual and correct the script proactively to avoid any impact to the customer's production pipeline.

4. CONCLUSIONS

Validating a cloud-scale distributed computation platform efficiently and effectively is quite challenging. In this paper, we present a test automation tool, called SCOPE Playback, for testing the SCOPE system over tens of thousands of machines at Microsoft. SCOPE Playback leverages the SCOPE system in unique ways. First, the system maintains a job repository which contains a full collection of customer jobs previously ran on production clusters. New features can be validated by playing back query compilation and optimization using such repository jobs. Second, SCOPE Playback itself is written as a SCOPE script and relies on the system to parallelize the test execution and automatically scale out with the increases of the test loads. Not only the tool benefits nicely from massive parallelism provided by the underlying SCOPE system and is able to process millions of job compilations in hours, but also the Playback jobs run without human intervention, as the underlying system provides transparent fail-over in face of failures. Together with an automatic bug filing system, SCOPE Playback serves as a fast and critical validation step in the SCOPE development.

One direction for future work is to extend the tool to cover runtime validation, which requires execution of parts of original jobs and comparing results. Given the proven flexibility and scalability of the SCOPE system, another direction for future work is to parallelize other time-consuming components, such as source code building, unit-testing, etc., using SCOPE . Finally, although the techniques in this paper are described in the context of SCOPE , the concept of using the cloud-scale service to validate its own development can be applied to many other systems as well.

5. ACKNOWLEDGMENTS

We would like to thank the following individuals for their contributions and support to the SCOPE Playback development: Sherry Li, Bill Ramsey, Brad Sarsfield, Sushil Chordia and Ed Triou. We would also like to thank Thomas Hargrove for his contributions to the job repository, and all the members of the SCOPE team for their support and collaboration.

6. REFERENCES

[1] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. In *Proceedings of VLDB Conference*, 2008.

[2] S. Chordia, E. Dettinger, and E. Triou. Different query verification approaches used to test entity sql. In *Proceedings of the first international workshop on testing database systems*, 2008.

[3] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. Patton, and B. Horowitz. Model-based testing in practice. In *Proceedings of 21st Annual Conference on Software Engineering*, 1999.

[4] J. Zhou, P.-Å. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *Proceedings of ICDE Conference*, 2010.