

In Data Veritas — Data Driven Testing for Distributed Systems

Ramesh Subramonian
LinkedIn

Kishore Gopalakrishna
LinkedIn

Kapil Surlaker
LinkedIn

Bob Schulman
LinkedIn

Mihir Gandhi
LinkedIn

Sajid Topiwala
LinkedIn

David Zhang
LinkedIn

Zhen Zhang
LinkedIn

ABSTRACT

The increasing deployment of distributed systems to solve large data and computational problems has not seen a concomitant increase in tools and techniques to test these systems. In this paper, we propose a data driven approach to testing. We translate our intuitions and expectations about how the system should behave into invariants, the truth of which can be verified from data emitted by the system. Our particular implementation of the invariants uses Q, a high-performance analytical database, programmed with a vector language.

To show the practical value of this approach, we describe how it was used to test Helix, a distributed cluster manager deployed at LinkedIn. We make the case that looking at testing as an exercise in data analytics has the following benefits. It (a) increases the expressivity of the tests (b) decreases their fragility and (c) suggests additional, insightful ways to understand the system under test.

As the title of the paper suggests, there is truth in the data — we only need to look for it.

Categories and Subject Descriptors

H.3.4 [Systems and Software]: Software Testing, Distributed Systems

1. INTRODUCTION

The data driven testing approach that we advocate consists of three activities

1. **Instrumentation** Inserting probes that provide some characterization of the system state. Examples include (i) log files (ii) system monitors (iii) the use of HTTP proxies, (iv) tcpdump (v) journaling triggers to monitor changes to a database [6] (vi) Aspect Oriented

Programming [8] to record paths through the code, ... With apologies to physicists, we like to say that these probes should not fall prey to “Heisenberg’s Uncertainty Principle” i.e., the act of observation should not materially affect the system under observation.

It is often necessary to have some idea about how the data will be analyzed at the time it is collected. “Tracers” are one example of this. For example, many large systems are a collection of distributed services that communicate over a protocol such as HTTP. In such cases, the incoming request should be tagged with a URL parameter (with a unique value) that passes unmodified (but recorded in the Apache log file) as the initial request fans out to multiple servers before coalescing back to a single response. The value of the URL parameter is then used as a “link field” to “join” different tables (Section 5.1).

2. **Simulation** Making the system go through the full range of motion that we desire to test. Production usage is one particularly realistic form of stress. Random (or quasi-random) testing, of the “Chaos Monkey” variety [10], is useful to push the system into corner cases without explicitly programming transitions through the search space of possible system states.
3. **Analysis** Data collected as a result of the above is analyzed to determine whether the system is behaving as expected, in terms of functionality and/or performance. An explicit set of invariants, whose truth can be ascertained from just the data collected in the above phases, constitutes the heart of the testing.

There are many ways to perform this analysis. As a matter of practice, we have found that it is efficient to parse the data into a relational format and load it into a relational database that can be queried to determine whether the observed data matches expectation. The difficulty of expressing invariants in SQL (a perfectly valid option and our initial choice) led us to create Q (Section 3).

1.1 Weakness of conventional testing

Testing asynchronous, distributed systems is difficult when one adopts the conventional “action-reaction” style of tests. The action-reaction style of testing consists of (i) getting the system into a desired state, (ii) performing a single action, (iii) waiting for the system to quiesce, and (iv) verifying

that the new state of the system reflects the change made. Instead, we need to observe *every* transition made by *every* node in the system and convince ourselves that, *at no time*, was the system in an inconsistent state.

We single out “record and replay” for its seductive simplicity. Using it, one can quickly crank out tests in the form of scenarios. However, in our experience, the test engineer is quickly bogged down by (1) the difficulty of getting a distributed system to perform exactly the same sequence of actions under the same conditions. [9] have shown how to record the workload running on a one database server, and subsequently replay it on a different one. However, orchestrating a faithful replay on a distributed system is a much harder problem. (2) the maintenance of these scenarios, especially when the software under test is evolving rapidly.

To be fair, it is not our intention to proscribe record-and-replay tests — we do maintain a small battery of such tests ourselves (Point 1 of Section 1.2 shows how we devise these tests). The key point is that they are more expensive than they appear to be and they are best when complemented with less expensive (albeit less reproducible) approaches.

1.2 Strengths of data-driven testing

It is our contention that looking at testing as an exercise in data analytics makes it simple and straight-forward to write tests that capture the intuition of the system architect and the experience of the deployment engineer.

- Data driven testing is also useful for devising tests. When the analysis phase discovers an error, we trace back through the logs to attempt to determine the sequence of operations that caused the problem. When successful, these steps are converted into the conventional action-reaction class of tests.
- Data-driven testing can serve as an effective knowledge base. The term “knowledge management” is likely to have acquired the unfortunate connotation of being more fluff than substance. However, our claim is that tests written at the correct level of abstraction are a better knowledge management tool than a Wiki page, which often has a write-once, read-seldom existence. A good test is one that can be understood and debated by the architect who designed the system, the engineer who implemented it and the operations engineer tasked with keeping it alive.
- Rather than focusing on test cases and creating scaffolding to run the system through those cases (whether action-reaction or record-replay), we run the system “like production”, although with a wider range of workload to exercise corner cases more frequently than they’d come up normally. The only cost of testing this way is that of adding instrumentation, which is useful to monitor the system anyway and not a high cost. This allows us to test the system as it changes as long as one can run it, and one doesn’t have to worry about maintaining growing set of test cases, and the associated infrastructure associated with it. Of course, if the specification (not the implementation) of the system

changes, then the invariants we write need to change as well.

- Software development is fast-paced business and new versions, which take advantage of emerging hardware and software innovations, need to be developed and deployed smoothly, even as existing versions are in deployment. By divorcing the tests from the system itself, the functional specifications that carry over from one version to the next, can test the new version and make sure that it does not regress. We recommend not relying on the developer of the system under test for instrumentation. Especially in less mature systems, log files are more soliloquies than an explicit contract between developer and tester, that can be relied on to not change from one version to the next.

1.3 Weaknesses of data-driven testing

Truth in advertising requires us to note the weaknesses of our preferred approach with the same candor as we have disparaged the alternatives.

- It depends on acquisition of data with sufficient fidelity for the analysis in question. For example, let us assume that the way we correlate actions on two servers is by using the “time in microseconds” in their respective log files as a link field when performing a join. Despite synchronizing network clocks, there may be a small (but non-zero) difference between the times at which the actions were recorded. This makes linking them difficult.
- It is not always possible to get the “right” data in a non-intrusive manner. This requires cooperation from the developers. We are blurring the distinction between “white-box” and “black-box” testing.
- It does not dispense with the creativity needed to push the system to the breaking point in the Simulation phase.
- While the Analysis phase can help point out lacunae, it does not suggest how they can be rectified.
- It does not obviate the need for good software engineering practices such as code reviews and unit tests.

1.4 Related Work

We agree with [2] which states that “testing is (to a large extent) a database problem and that many testing activities can be addressed best using database technology”. Their focus on generating test data falls under “Simulation” in our terminology, not on analysis. While we agree with their claim that test code should be declarative, we disagree with their choice of SQL for this purpose. We found that expressing invariants in SQL to be possible but cumbersome, which motivated the development of Q

BloomUnit [1] is a testing framework for distributed programs written in the Bloom language. Like us, they advocate an “incremental process by which a programmer might provide and refine a set of queries and constraints until they define a rich set of correctness tests for a distributed system.” In contrast, they focus on specifications that describe

the input/output behavior, whereas the invariants we prefer must hold true regardless of the input.

1.5 Organization

The paper is organized as follows

1. Section 2 describes Helix, a general purpose cluster manager, which will be use as a case study through the rest of the paper. While Helix provides useful context for exposition, the technique of data driven testing is broadly applicable.
2. Section 3 describes Q, an analytical database used to implement the invariants. We recognize that the use of a special purpose language (as opposed to SQL) is controversial. Our (admittedly weak) defense is that SQL was our first choice, and it was only when it did not provide the expressivity and speed that we needed, did we invent something new. Regardless, the choice of invariant evaluation technology does not detract from the merits of the methodology.
3. Section 4 lists data collected by Instrumentation
4. Section 5 gets into the meat of the matter — analysis. We list some of the invariants designed and their implementation in Q. While we do not list Q code explicitly, the pseudo-code translates (almost) line for line to statements in Q.

2. THE HELIX USE CASE

Helix is a general purpose cluster manager, meant to simplify the development of distributed systems. By letting the system specify behavior in terms of a state machine and constraints, Helix handles the hard problems of partition management, fault tolerance and cluster expansion.

While Helix has been (and can be) used in a variety of different contexts, in this paper we describe its usage in managing a distributed database. For brevity, we must grossly oversimplify its functionality. Details can be found in [7].

Assume that we wish to distribute a database across a number of nodes in order to improve performance and gain fault-tolerance. The database is divided into partitions P_1, P_2, \dots , each of which is replicated e.g., P_{11}, P_{12} , would be the replicas of P_1 . These replicas are distributed over a set of nodes, M_1, M_2, \dots

With each replica is associated a state. It is Helix’s responsibility to manage the state of the replicas, subject to constraints placed by the user at configuration time. The “augmented state machine” managed by Helix allows the user to specify (i) the set of legal states, (ii) the set of legal state transitions, and (iii) the minimum and maximum number of replicas that should be in a given state.

For simplicity of exposition, and with no loss in generality, we assume that all inputs to, and observations of, the system are in the form of tables. In reality, these are in XML and/or JSON format. However, it is trivial to parse them into CSV files which are then loaded into tables in Q (Section 3).

We now describe the configuration provided by the user, which can be thought of as a contract that must be enforced by Helix on the distributed system.

2.1 State Machine

Helix allows one to specify, for each partition an augmented state machine as in Section 2.1.1 and 2.1.2.

2.1.1 States and Counts

Table T_S contains columns — (1) state, (2) n_R^{min} = minimum number of replicas in that state, (3) n_R^{max} and (4) priority (which specifies the relative importance of constraints). For example, Table 1 asserts that

1. there are 4 states in the system — M, S, O, D
2. we would like to have 1 replica in state master and 2 replicas in state slave
3. it is more important to have 1 replica in state master than it is to have 2 replicas in state slave

State	Min Count	Max Count	Priority
(M) Master	1	1	1
(S) Slave	1	2	0
(O) Offline	⊥	⊥	⊥
(D) Dropped	⊥	⊥	⊥

Table 1: Sample of State Specification

2.1.2 State Transitions

Table T_X contains 3 columns — from state, to state and priority. For example, if we have rows (O, S, 0), (M, S, 0) and (S, M, 1), it means (i) that (M, S), (S, M), (O, S) are legal state transitions and (ii) it is more important to execute the (S, M) transition than either the (M, S) or (O, S) transitions. By “important” we mean that if it were possible to either (i) an (S, M) for some partition or (ii) an (M, S) transition for some other partition but not both, then the more important transition should be executed.

3. Q — THE ANALYTICAL DATABASE

Q is a high-performance “almost relational” analytical column-store database.

By “analytical”, we mean that it is not meant for transactional usage — data is loaded infrequently. The relatively high cost of data change is amortized over the analysis workload. While Q can handle data changes, it is expensive and therefore inadvisable to use Q in situations where the data changes are frequent.

By “almost relational”, we mean that it would more correctly be called a “tabular model” [3]. As Codd states, “Tables are at a lower level of abstraction than relations, since they give the impression that positional (array-type) addressing is applicable (which is not true of n-ary relations), and they fail to show that the information content of a table is independent of row order. Nevertheless, even with these minor flaws, *tables are the most important conceptual representation of relations, because they are universally understood.*”

That said, the tabular model implemented by Q satisfies Codd’s requirements for a data model, which are

1. A collection of data structure types (the database building blocks)
2. A collection of operators or rules of inference, which can be applied to any valid instances of the data types listed in (1), to retrieve, derive, or modify data from any parts of those structures in any combination desired;
3. A collection of general integrity rules, which implicitly or explicitly define the set of consistent database states or changes of state or both — these rules are general in the sense that they apply to any database using this model

3.1 Operations in Q

The basic building block is a table which can be viewed as a collection of fields or columns, each of which has the same number of cells or values. This allows us to use notation of the form (i) $T[i].f$ which is the i^{th} row of column f of table T or (ii) $T[f = v]$ which is the subset of rows of table T where column f has value v (iii) the symbol for null is \perp

Every operation of Q consists of reading one or more columns from one or more tables and producing one or more columns in an existing table or a newly created one. This relatively simple statement makes for great simplicity, both in terms of implementation and programming (Section 3.2). There are a few exceptions to this rule e.g., computing an associative operation on a column, are the values of a column unique, meta-data based operations like table listing ...

3.2 Programming Q

Q is programmed with a vector language, not with traditional SQL. We list a few of the operations that can be performed in Q.

1. `sort T1 f1 f2 f1' f2' A_` would mean that we read columns `f1, f2` in table `T1` and create columns `f1', f2'`, such that `f1'` is a permutation of `f1` sorted in ascending order and field `f2'` is permuted, as a drag-along field.
2. `shift T1 f1 n f2` creates a new field `f2` in table `T1` as $T[i].f_2 \leftarrow T[i-n].f_1$ and $\forall j : 1 \leq j \leq n : T[j].f_2 \leftarrow \perp$
3. `w_is_if_x_then_y_else_z T f_w f_x f_y f_z` creates a new field `f_w` in table `T` as follows. `if T[i].f_x = true then T[i].f_w ← T[i].f_y else T[i].f_w ← T[i].f_z fi`

4. OBSERVATION OF SYSTEM BEHAVIOR

As mentioned before, a system emits several signals when it runs. In this paper, we focus on just one of the signals emitted by Helix — Zookeeper’s logs. Helix uses Zookeeper’s group membership and change notification to detect state changes. Zookeeper’s logs, when parsed, produce the tables listed in this section.

In general, prior to analysis, the data is massaged so that it is easy to reason about. Consider a specific example in

the Helix use case. A node may die and come back to life several times during the course of testing. During each of its “lives”, its IP address is the same but it has a different, unique session identifier. We convert the session into an “incarnation” number that starts at 1 and increments by 1 for each “life” that this node has.

4.1 Zookeeper Logs (Parsed)

4.1.1 LiveInstances

Every time an instance goes down or comes up, a row is inserted into T_{LI} table, with 4 columns — (1) time (2) instance (3) incarnation (4) op, whether added or deleted.

4.1.2 Messages

Every time a state transition occurs, a row is inserted into Table T_{MSG} , which contains 6 columns — (1) time (2) partition (3) instance (4) incarnation (5) from state (6) to state

4.1.3 CurrentState

Table T_{CS} tells us the state of an instance at different points in time. It contains the 5 columns — (1) time (2) partition (3) instance (4) incarnation (5) current state.

5. TESTING

5.1 Triangulation

In testing terms, triangulation is the observation of the same effect from two different viewpoints and making sure that they are consistent. While such data collection might appear redundant, our experience suggests that the benefit of the greater confidence far outweighs the collection (and analysis) cost.

5.2 Sanity Checks on Data

While this might sound like a serpent swallowing its tail, we recommend performing basic sanity checks on the data prior to using it for analysis. This process has often revealed errors in the data collection process or assumptions made in the post-processing parsing/loading routines. As an example, consider T_{LI} , Section 4.1.1. The checks made are as follows

INVARIANT 1. $(instance, incarnation, op)$ is unique

INVARIANT 2. $|T_{LI}[instance = x]| = n \Rightarrow \forall i : 1 \leq i \leq n, \exists j : T_{LI}[j].instance = x \wedge T_{LI}[j].incarnation = i$

5.3 Testing for Correctness

For every state transition of a replica, we need to check that it is legal.

- Add a Boolean column x to T_{MSG} as follows. $\exists j : T_X[j].from, to = T_{MSG}[i].from, to \Rightarrow T_{MSG}[i].x \leftarrow 0$; else, 1.
- $n \leftarrow \sum_i T_{MSG}[i]$. Error if $n > 0$

5.4 Testing as an ongoing dialogue

Testing, when performed as an exercise in data analytics, becomes a matter of finding the right mathematical abstraction to describe a complex system. In this light, it serves to improve the understanding of the system as much for the test engineer as for the architect. The following example illustrates the point. Consider the specification of Table 1. This is implemented as follows.

- Sort $T' = T_{MSG}$, with partition being the primary key and time being the secondary key.
- For each state, s , create a column δ_s as follows
 1. $to_state = s \Rightarrow \delta_s \leftarrow 1$
 2. $from_state = s \Rightarrow \delta_s \leftarrow -1$
 3. else, $\delta_s \leftarrow 0$
- Then, for each state, s , create a column n_s in T' as follows. $T'.partition[i] = T'.partition[i-1] \Rightarrow T'.n_s[i] = T'.n_s[i-1] + T'.\delta_s[i]$; else $T'.n_s[i] =$ number of machines in which partition $partition[i]$ is in state s at the start.
- Then, $T_S[s].n_R^{min} \leq T'.n_s[i] \leq T_S[s].n_R^{max}$

When we evaluated the above invariant, we found a partition where it had been violated, as shown in Row 4 of Table 2. Let (t, n) be a row of this table. This means that the number of replicas in state Slave at time t was n .

ID	Time	Number of Replicas
1	42632	0
2	43131	1
3	43275	2
4	43323	3
5	85795	2

Table 2: Number of Replicas in state Slave

On reflection, this is not surprising. Consider what happens when a master goes down. There will be a transient period of time when there is no master. Also, since (as configured in Section 2.1.2) a node cannot transition directly from state Offline to state Master (except through state Slave), there may be some time when the number of slaves is exceeded.

So, the question is not whether the constraint is violated, but for how long it is violated. As an example, during one of our test runs, we produced Table 3. Let s, n, p be a row of this table. This means that, averaged over all partitions, p is the percentage of time that the number of replicas in state s was n .

The above discovery led us to go back to the architects to force them to explicitly state a Service Level Agreement — how long is too long?

The discerning reader would have noticed an inadequacy in the formulation of Table 3. What we have computed is the *average* time a partition does not have a master. This needs

State	Number of Replicas	Percentage of Time
Slave	0	0.5
Slave	1	16.5
Slave	2	82.95
Slave	3	0.05
Master	0	7.2
Master	1	92.8

Table 3: Percentage of time for State Count

to be refined to say that the maximum time *any* partition does not have a master needs to be below a certain bound.

This is computed as follows.

1. Use *instance, incarnation* to create a single composite field p
2. Create a Boolean field x which is true when n_M , the number of replicas of that instance, is 0 or 1.
3. Let $T' \leftarrow T_{MSG}[x = true]$ be a subset of T_{MSG}
4. Sort T' with p being the primary key and time being the secondary key, both ascending
5. Create field δ where $delta_i \leftarrow t_i - t_{i-1}$
6. Create a Boolean field x in T' , $x[i] = true \Leftrightarrow n_M[i] = 1 \wedge n_M[i-1] = 0 \wedge p[i] = p[i-1]$
7. Let $T'' \leftarrow T'[x = true]$ be a subset of T''
8. Then, the maximum value of δ in T'' needs to be bounded appropriately.

The purpose of the above narrative was to convince the reader that a testing framework is not simply a means of efficiently recording tests enunciated from on high. It must support an ongoing dialogue between designer, tester, developer and deployer.

Most importantly, our approach recognizes that is difficult, in practice, to start with a fully specified system, where all possible test cases are known a priori. The fact that the data is available for analysis long after the simulation is done enables (and encourages) the test engineer to define and run more tests, as long as the instrumentation supports it.

5.5 Testing for Test Coverage

In the context of testing, the question “Who will guard the guardians”¹ becomes one of determining, from the data, whether the system was pushed to the potential points of failure. Testing whether every state transition was executed is done as follows

- Create a constant field $x = 1$ in T_{MSG}

¹attributed to the Roman poet Juvenal

- Create a field n in table T_X by using $(from, to)$ as a composite link field and summing the values of x using grouped aggregation
- Table 4 is an example of the output and we place conditions on the minimum number of times a transition must be executed for us to believe that it is well tested.

From	To	Count
Master	Slave	55
Offline	Dropped	0
Offline	Slave	298
Slave	Master	155
Slave	Offline	0

Table 4: Count of State Transitions

5.6 Testing for Knowledge Capture

It would be blatantly misleading to claim that the support provided by analytics obviates the need for creativity in designing tests. However, what our approach does provide is an easy and methodical way to capture past experiences and intuitions. As the saying goes, “Fool me once, shame on you. Fool me twice, shame on me.”

As an example, we discovered (by chance) that when a node went down and came back up in a very short period of time, it would throw the system out of whack. What this told us was that every subsequent test run should contain scenarios where machines are toggled on and off rapidly. Implementation in Q consists of the following steps.

1. Sort T_{LI} on node (primary) and time (secondary), both ascending
2. Let $T[i].\delta \leftarrow T_{LI}[i].time - T_{LI}[i+1].time$
3. Let $T' \subseteq T_{LI}$ contain only those rows such that
 - (a) $T_{LI}[i].node = T_{LI}[i+1].node$
 - (b) $T_{LI}[i].op = DELETE$
 - (c) $T_{LI}[i+1].op = ADD$
4. The distribution of $T'.\delta$ should be sufficiently diverse, small values of δ corresponding to rapid on-and-off cycling.

6. FUTURE WORK

The perceptive reader would have noted the following lacunae in our approach. They remain an area of active research and development.

- Given the impossibility of exploring the search space completely, how does one intelligently guide the exploration of this space so as to improve test coverage. In other words, could one have automatically created the “Amazon storm” scenario without explicitly programming it?
- Given that we wish to move the system from one state to another, how does one work backwards through the “Simulation” phase to perform the actions that effect that transition?

- Once the quasi-random simulation discovers a weak point in the system, how can one automatically convert that knowledge into a deterministic test?

7. CONCLUSION

If testing is difficult business (“program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence” [4]), testing distributed systems is even more so. Data driven testing provides a structured methodology to reason about the behavior of a system and to compare expectation with observation. It does not require repeatable tests, the maintenance of which is expensive, especially in an rapidly evolving software environment.

Instead, we focus on workload and analysis. We take the production system and put it through workloads designed to make it encounter corner cases and potentially troublesome, be they “thundering herds”, “flapping”, ... And we leave it to the data to tell the story as to whether those states did occur and whether the correct behavior happened.

It is hard to disagree with Dijkstra [5] that “The only effective way to raise the confidence level of a program significantly is to *give a convincing proof of its correctness.*” Nevertheless, data-driven testing is a practical step towards formal specification. Our difficulty is, as Dijkstra [5] pointed out, “It is psychologically hard in an environment that confuses between love of perfection and claim of perfection and, by blaming you for the first, accuses you of the latter.”

8. ADDITIONAL AUTHORS

9. REFERENCES

- [1] P. Alvaro, A. Hutchinson, N. Conway, W. Marczak, and J. Hellerstein. Bloomunit: Declarative testing for distributed programs. In *DBTest 2012*, May 21, 2012.
- [2] E. L. Carsten Binnig, Donald Kossmann. Towards automatic test database generation. *Data Engineering*, 31(1):28–35, March 2008.
- [3] E. F. Codd. Relational database; a practical foundation for productivity. *Communications of the ACM*, 25(2):109–117, February 1982.
- [4] E. W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.
- [5] E. W. Dijkstra. EWD 648. www.cs.utexas.edu/ewd/transcriptions/EWD06xx/EWD648.htm
- [6] P. Frankl. Somebody put a database in my software testing problem. In *DBTest 2010*, June 7, 2010.
- [7] K. Gopalakrishna, S. Lu, A. Silberstein, R. Subramonian, K. Surlaker, and Z. Zhang. Untangling cluster management with Helix. In *SOCC*, 2012.
- [8] G. Kiczales, J. Lamping, A. Mehdhekar, C. Maeda, V. Lopes, J. Loingtier, and J. Irwin. Aspect oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, Springer Verlag LNCS 1241, June 1997.
- [9] K. Morfonios et al. Consistent synchronization schemes for workload replay. In *SIGMOD*, 2011.
- [10] Netflix Engineering Team. Chaos Monkey. <https://github.com/Netflix/SimianArmy>.