Why Property-Based Testing is Necessary for Data Intensive Scalable Computing

Yaoxuan Wu
UCLA
Los Angeles, CA, USA
thaddywu@cs.ucla.edu

Ingrid Lee
UCLA
Los Angeles, CA, USA
ingridlee@g.ucla.edu

Ahmad Humayun
Virginia Tech
Blacksburg, VA, USA
ahmad35@vt.edu

Muhammad Ali Gulzar
Virginia Tech
Blacksburg, VA, USA
gulzar@cs.vt.edu

Miryung Kim
UCLA
Los Angeles, CA, USA
miryung@cs.ucla.edu

Abstract

Data-intensive scalable computing (DISC) frameworks such as Apache Spark, Flink, Beam, and Dask underpin many modern analytics workloads by providing high-level programming models and scalable runtimes. Despite their widespread adoption, framework bugs remain common, and many manifest as *silent wrong results* rather than crashes. Property-based testing (PBT) has been successful across a range of domains by using semantic contracts as executable oracles when ground-truth outputs are difficult to obtain. We argue that DISC calls for a general and extensible PBT framework that provides reusable property templates and supports systematic instantiation across different DISC systems.

ACM Reference Format:
Yaoxuan Wu, Ingrid Lee, Ahmad Humayun, Muhammad Ali Gulzar, and Miryung Kim. 2026. Why Property-Based Testing is Necessary for Data Intensive Scalable Computing. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 05–09, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3803437.3806090

1 What and Why: Property-Based Testing for DISC

Property-based testing (PBT) checks whether a system satisfies *semantic contracts* over many automatically constructed inputs, rather than validating a small set of hand-written examples with fully specified expected outputs. A property serves as an *executable semantic oracle* when ground-truth outputs are difficult to construct. [1]

Data intensive scalable computing (DISC) frameworks provide high-level programming models and scalable runtimes for large-scale data processing, including Apache Spark, Flink, Beam, and Dask. Despite their popularity, we argue that property-based testing for DISC frameworks is necessary and remains an open area.

Silent logic bugs demand semantic oracles beyond crash signals. Many high-impact DISC failures are *silent logic bugs*: executions finish without crashing but produce incorrect results, a failure mode repeatedly discussed in issue trackers such as Spark JIRA. [2, 3] Existing DISC testing efforts [4–7] primarily stress *user programs*, making silent framework bugs harder to detect systematically.

Exercising optimizer and execution paths is desired. Many framework bugs arise in optimizer transformations and physical execution, such as plan rewrites, operator fusion, and code generation. PBT can leverage deliberate, semantics-preserving rewrites to increase the likelihood of exercising specific optimization or execution paths, and then check semantic consistency across the resulting executions. In comparison, mutation-based fuzzing often exposes shallow errors (e.g., analysis- and validation-stage failures), providing limited coverage of these late-stage behaviors.

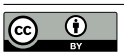
Property violations yield more actionable bug reports than crashing inputs. Crashing inputs often provide limited guidance about intended semantics. In contrast, a property violation directly identifies which semantic contract was broken, making bug reports more actionable for diagnosing late-stage optimization and execution behaviors. Property contracts can also serve as executable documentation for subtle framework semantics. [1]

Related Work. In DISC testing, SSCheck [8] and FlinkCheck [9] focus on temporal properties over streams, leaving optimizer-centric behaviors largely unexplored. Achilles' SPEar [10] adopts a TLP-style oracle [11], but its generated workloads cover only a narrow operator subset (filter/map/aggregation/windowed aggregation). DiffStream [12] provides differential testing with a stream-equivalence oracle, but it does not automatically generate equivalent program pairs beyond disabling parallelism.

In contrast, DBMS testing has shown the effectiveness of oracle-based testing. SQLancer integrates multiple property oracles and has uncovered many previously unknown DBMS logic bugs [13]. Representative examples include query partitioning, semantics-preserving query rewriting, and pivoted query synthesis. [11, 14, 15]

2 Why DBMS Testing Is Not Directly Transferable to DISC

Although DBMSs and DISC frameworks both offer declarative data processing, existing DBMS testing techniques do not transfer directly to DISC. The gap is semantic: data model and the unit of composition.



FSE Companion '26, Montreal, QC, Canada

ACM ISBN 979-8-4007-2636-1/2026/07
https://doi.org/10.1145/3803437.3806090

Table 1: Example property contracts for DISC frameworks, organized by shared templates.

What is tested
Per-group recomposition for aggregation operators
PerGroupRecompose(sum). Group by a key column, apply **sum**(y) per group, recompose by **sum**, and match the global **sum**(y).
PerGroupRecompose(max). Group by a key column, apply **max**(y) per group, recompose by **max**, and match the global **max**(y).
Cardinality relations for DataFrame operators
Cardinality(union). Apply **union** to two inputs and validate that output cardinality matches the intended union semantics.
Cardinality(filter). Apply **filter** with a predicate and validate that output cardinality does not exceed input cardinality.

Data model and operator gap. Most DBMS testing techniques assume flat relational schemas and scalar expressions. In contrast, real-world DISC workloads routinely operate on nested and semi-structured data (e.g., arrays, structs, JSON) and make heavy use of generators (e.g., `explode`) and higher-order functions over collections (e.g., element-wise `filter`/`transform`). These operators introduce element-level semantics and nontrivial cardinality changes, and they often trigger complex rewrites and operator fusion in DISC runtimes, creating correctness risks that fall outside the scope of existing DBMS testing. [16]

Composition gap: query blocks versus dataflow directed acyclic graphs (DAGs). DBMS testing commonly treats a single SQL query block as the primary unit for testing and validates query-local equivalences under optimizer rewrites. In DISC, programs are dataflow DAGs where intermediate datasets are explicitly reused and can be consumed by multiple downstream actions. Correctness can hinge on cross-action materialization and reuse, which is not captured by equivalences defined within a single query block. For example, DISC runtimes may reuse internal row objects and mutable buffers in physical execution (e.g., `UnsafeRow`, aggregation buffers). If an operator, code generation path, or serialization layer mistakenly retains references to such reused objects, later actions may observe corrupted data and produce incorrect results. Exposing these cross-action failures therefore requires DISC program generators that construct dataflow DAGs, whereas existing DBMS generators primarily target isolated query blocks.

3 Toward a General Framework for PBT in DISC

PBT for DISC calls for a general framework that makes generators and properties easier to develop and reuse. Rather than implementing each test as a one-off artifact, the framework should provide reusable program substrates together with abstract, adaptable property specifications.

A reusable program substrate is needed. Many DISC correctness bugs do not arise from isolated operators alone, but from how operators are composed, materialized, and reused within larger workflows. As discussed earlier, DISC programs are naturally represented as dataflow directed acyclic graphs (DAGs), rather than isolated query blocks. This makes a reusable program substrate important: the testing framework should support DAG-structured programs that can host different property instantiations, rather than rebuilding a separate generator for each property or framework. Such a substrate is especially important for exposing behaviors that depend on shared intermediates, multiple downstream actions, or cross-action materialization and reuse, which fall outside the scope of query-local testing alone.

Abstract and adaptable property specifications are needed. Useful DISC properties often reflect stable semantic intent, but their executable realization may vary across systems, versions, language bindings, and runtime settings. For example, even simple equality properties may require different treatment across execution environments, such as `NaN` semantics in Spark versus Python. [17] This suggests that PBT for DISC should not rely on isolated, hand-written checks tied to one system at one point in time. Instead, properties should be represented at a more abstract level, organized into reusable schemas or shared templates, and instantiated through system- and version-aware bindings. Table 1 illustrates this idea with example property contracts organized by shared templates.

Acknowledgement

This work was supported by NSF grants 2426162, 2106838, and 2106404, and in part by Amazon and Samsung. We thank the anonymous reviewers for their helpful feedback.

References

[1] N. Rinaudo, J. Hughes, K. Claessen *et al.*, "Property-based testing in practice," in *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, 2024.
[2] "SPARK-49000: Aggregation with distinct gives wrong results when dealing with literals," https://issues.apache.org/jira/browse/SPARK-49000.
[3] "SPARK-33726: Limit and offset ordering behaves differently between dataframe and sql," https://issues.apache.org/jira/browse/SPARK-33726.
[4] A. Humayun, M. Kim, and M. A. Gulzar, "Bigfuzz: Efficient fuzz testing of data analytics systems," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.
[5] M. A. Gulzar, M. Steiner, M.-W. Shih, and M. Kim, "White-box testing of big data analytics with complex user-defined functions," in *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019.
[6] A. Humayun, M. Kim, and M. A. Gulzar, "Co-dependence aware fuzzing for dataflow-based big data analytics," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023.
[7] Y. Wu, A. Humayun, M. A. Gulzar, and M. Kim, "Natural symbolic execution-based testing for big data analytics," in *Proceedings of the ACM International Conference on the Foundations of Software Engineering (FSE)*, 2024.
[8] A. Riesco and J. Rodríguez-Hortalá, "Property-based testing for spark streaming," *arXiv preprint arXiv:1812.11838*, 2018.
[9] C. V. Espinosa, E. Martin-Martin, A. Riesco, and J. Rodriguez-Hortala, "Flinkcheck: Property-based testing for apache flink," *IEEE Access*, vol. 7, pp. 150 369–150 382, 2019.
[10] M. E. Kroner, "Achilles' spear: Using metamorphic testing to find bugs in stream processing engines," in *BTW 2025: Datenbanksysteme für Business, Technologie und Web*, 2025, pp. 1031–1042.
[11] M. Rigger and Z. Su, "Finding bugs in database systems via query partitioning," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.
[12] K. Kallas, F. Niksic, C. Stanford, and R. Alur, "Diffstream: Differential output testing for stream processing programs," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, 2020.
[13] "Sqlancer," https://github.com/sqlancer/sqlancer.
[14] M. Rigger and Z. Su, "Detecting optimization bugs in database engines via non-optimizing reference engine construction," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1140–1152.
[15] ——, "Testing database engines via pivoted query synthesis," pp. 667–682, 2020.
[16] "Spark-39854: Catalyst 'columnpruning' optimizer does not play well with sql function 'explode'," https://issues.apache.org/jira/browse/SPARK-39854.
[17] "NaN Semantics," Spark 3.0.0 Documentation. [Online]. Available: https://downloads.apache.org/spark/docs/3.0.0-preview/sql-ref-nan-semantics.html