Checking Transaction Isolation Violations Using Graph Queries

Stefania Dumbrava, Zhao Jin, Burcu Kulahcioglu Ozkan, Jingxuan Qiu

Checking Transaction Isolation Violations using Graph Queries

Stefania Dumbrava[1], Zhao Jin[1], Burcu Kulahcioglu Ozkan[2], and Jingxuan Qiu[2]

[1] ENSIIE, France
[2] Delft University of Technology, Netherlands

Abstract. Distributed databases provide different transaction isolation levels for higher performance and fault tolerance. However, implementing isolation models is challenging, and database systems can produce executions that violate their isolation guarantees. In this work, we propose GRAIL, a new approach that uses graph databases and queries to detect isolation violations expressed as anti-patterns in transactional dependency graphs. We implement the approach on top of the popular ArangoDB and Neo4j graph databases and show its efficiency through an experimental analysis of real executions of ArangoDB as a system under test.

Keywords: Graph Queries · Distributed Databases · Transactions.

1 Introduction

Database isolation levels describe the degree to which the updates of a running transaction are isolated from other concurrent transactions. At the strongest level, the transactions are *serializable*, i.e., they produce an execution that is equivalent to running them serially in some order. While serializability provides strong guarantees, implementing it requires strong synchronization in databases with sharding and replication. To achieve higher performance, many databases support isolation levels weaker than serializability. However, it is difficult for them to ensure their claimed guarantees, as witnessed by the many violations discovered in popular distributed databases [24]. Moreover, checking the correctness of the executions for a given isolation level is challenging, e.g., checking serializability or snapshot isolation is NP-complete in general [28, 9].

Database isolation violations can be detected by inspecting the *dependency graph* of database executions [28, 7], which models read/write transaction dependencies between transactions. In this setting, violations can appear as anti-patterns indicating *isolation anomalies* prohibited in serializable databases. However, existing methods do not leverage their inherent graph structure.

Graph databases have been increasingly used to analyze interconnected data, due to their expressive graph models and custom query languages that allow to directly extract complex graph patterns [10, 30, 8]. As transaction dependencies and isolation anti-patterns are modeled by graphs, we investigate the feasibility

and performance of graph queries to capture isolation violations. Such a method is easily extensible to analyzing a wider set of execution patterns.

We introduce GRAIL [14, 13], a new GRAph-based Isolation Level checking approach that *is the first to use graph queries to detect database isolation violations*. We provide a proof-of-concept implementation of GRAIL through graph queries in AQL (using ArangoDB) and Cypher [16] (using Neo4j [27]). We used it to test ArangoDB [20] database executions, checking these for isolation anomalies. Our evaluation shows that the graph queries are effective and remain scalable with increasing execution lengths and transaction concurrency.

Related Work. Several methods and tools have been designed to check the strong consistency and isolation models, linearizability, and serializability of database transactions using enumerative exploration and/or SMT solvers [31, 23, 22, 34, 12, 2]. Recent works [9, 36, 34, 19] focus on checking snapshot isolation (SI), which is weaker than serializability but still provides strong guarantees on transaction conflicts. Different from the existing work, GRAIL uses graph queries for pattern-matching and can check *a spectrum of isolation levels*, including PSI [11], PL-2 [1], and PL-1[1].

2 Graph-Based Checking of Isolation Violations

Our approach for checking isolation violations using graph queries requires (i) collecting the execution histories of databases, (ii) constructing the dependency graphs from the execution histories, and (iii) checking the graphs for the existence of cycles with violation patterns. The main novelty of our approach lies in (iii), the detection of violations using graph database queries on dependency graphs. First, we generate and collect (i) the test executions using the Jepsen [23] tool. Then, we build the dependency graph (ii), as standard in the literature. A *dependency graph* $\mathcal{G} = \{\mathcal{H}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}\}$ is an execution history with a finite set of transactions $\mathcal{H} = \{T_1, \ldots, T_n\}$ and *read-dependency* (WR), *write-dependency* (WW), and *anti-dependency* (RW) relations (edges) between its transactions (nodes) [7]. We build the graph by creating a node for each transaction and placing the dependency edges between them, following the read/write operations in the execution history and the database write-ahead logs. Finally, (iii) we run graph database queries on the generated graphs, and we check for anti-patterns to detect violations.

2.1 Database Isolation Levels

Table 1 illustrates the allowed and disallowed serializability anomalies in the SER [28], SI [6], PSI [33], PL-2 [1], and PL-1 [1] transaction isolation levels.

Example 1. Consider an LDBC benchmark [15, 35] schema example representing *Persons* who communicate through *Forum* posts. The forums are managed by moderators who can update their titles. The schema supports *read* and *write*

Level	Write skew	Long fork	Lost update	Dirty read	Anti-Pattern
SER	✗	✗	✗	✗	any cycle
SI	✓	✗	✗	✗	cycle without two consecutive RW edges
PSI	✓	✓	✗	✗	cycle with less than two RW edges
PL-2	✓	✓	✓	✗	cycle without RW edges
PL-1	✓	✓	✓	✓	cycle with only WW edges

Table 1: Anomalies allowed/disallowed by the isolation levels

operations to view and update forum titles and the list of moderators. We assume that the initial titles of the two forums *f1* and *f2* are $A1$ and $B1$.

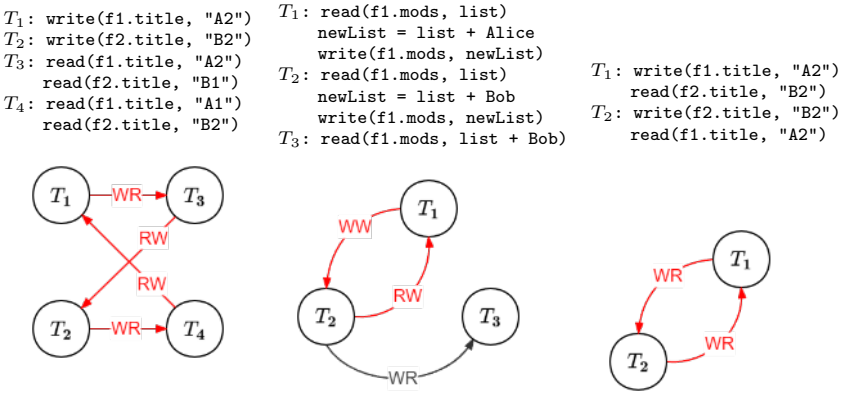
In Figure 1, we present the isolation level anti-patterns as follows. Figure 1a illustrates a *long fork*, as the concurrent transactions T_1 and T_2 write to different objects (the titles of the two forums *f1* and *f2*) and commit, while subsequent transactions T_3 and T_4 only see the effects of one transaction. The WR edges capture that T_1 writes $A2$ to *f1*'s title, read by T_3, and that T_2 writes a value to another object, read by T_4. The RW edges mark that T_3 and T_4 read values $A1$ and $B1$, overwritten by T_1 and T_2. Figure 1b represents a *lost update* anomaly. T_1 and T_2 concurrently write to the forum moderators, adding a new one to the list (we overload the + operation as append). T_1 writes to the variable read by T_2 (the RW edge), and T_2 overwrites the value written by T_1 (the WW edge). The subsequent transaction T_3 only sees the effect of T_2, T_1's update being lost.

We consider the *dirty read* anomaly without restriction to reading from uncommitted transactions, but including *aborted reads*, *intermediate reads*, and *circular information flow* [1]. An aborted read anomaly occurs when a transaction reads a value written by an aborted transaction. An intermediate read anomaly occurs when a transaction reads any intermediate, uncommitted value from another transaction. These two anomalies can be identified by checking whether the values read by the transactions are committed. Circular information flow occurs when transactions have a circular dependency on the values read. Figure 1c shows an example of a *circular information flow*. For concurrent transactions T_1 and T_2, T_2 reads the value $A2$ written by T_1 and, at the same time, T_1 reads the value $B2$ written by T_2 (the WR edges).

The existence of anomalies in transaction executions can be analyzed using their dependency graph. As in early database systems works for conflict serializability [28], isolation level violations can be detected by searching for anti-patterns in the dependency graph. Table 1 lists the anti-patterns that indicate violations to levels SER, SI, PSI, PL-2, and PL-1.

2.2 History Dependency Property Graph Model

Our methodology consists of relying on graph databases for isolation level checking. Graph databases are NoSQL data stores that provide custom support for processing interconnected data, leveraging expressive graph models. Among these, the most prominent one is the *property graph* model [3], i.e., a multi-labeled



(a) A long fork (b) A lost update (c) Cyclic info flow

Fig. 1: Isolation anomalies and their graph patterns

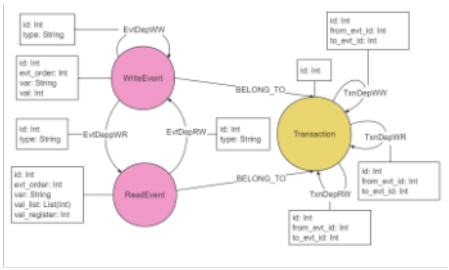
multi-graph, whose nodes and edges can be additionally enriched by sets of key-value properties. We model the history dependency graph as a property graph and depict its corresponding schema [4] in Figure 2a. Each node represents an event (`WriteEvent` or `ReadEvent`) or a `Transaction`. Event nodes are associated with the transactions that contain them through `BELONG_TO` edges and store their order within the transaction (`evt_order`) and their corresponding object (`var`). Event values are stored on `WriteEvent` nodes using the atomic `val` property, as these are written individually. For `ReadEvent` nodes, we store `val_list` and `val_register` properties, to account for list and register operations. `Event` nodes are interrelated by dependency edges (`EvtDepWW`, `EvtDepWR`, and `EvtDepRW`). We proceed analogously for `Transaction` nodes, explicitly storing, for ease of querying, the identifiers of the events they relate. Identifiers (`id`) for nodes and edges are automatically generated by the graph database.

As a standard graph query language has only just recently been published [18], graph databases have supported the extraction of expressive graph patterns through custom query languages. A leading one [32] is Neo4j's [27] Cypher [16], also used in numerous other databases, e.g., SAP HANA [29], Amazon Neptune [5], Memgraph [26], RedisGraph [25], and AgensGraph [21].

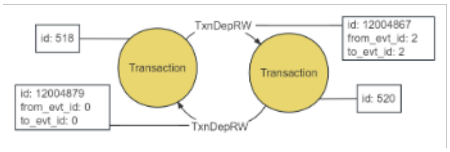
One of our checkers relies on Cypher queries to catch isolation level violations, e.g., the write skew anomaly in Figure 2b. The reported violation exposes relevant information to the user, e.g., transaction IDs and dependency edge labels, helping with both understandability and explainability.

2.3 Graph Query-Based Anti-Pattern Detection

We formulate multiple queries that will serve as checkers for isolation anti-pattern detection to investigate and compare their relative performances.



(a) Execution history



(b) Reported write skew anomaly

Fig. 2: Property graph schema of execution histories and example violation

Definition-based Checkers In ArangoDB, the `ArangoDB-Cycle` checker detects directed cycles using their definition: non-empty paths where only the first and the last vertices are equal. The corresponding AQL query performs a graph traversal on each transaction vertex. A cycle is detected when the traversal reaches the start vertex again, with a minimum depth of 2. We use ArangoDB's graph traversal, setting the minimum depth to 2 and the maximum one to 4 (see Listing 1.1). The maximum depth limits the number of vertices in a detected cycle and, therefore, becomes an important factor in the effectiveness of the checker. For example, setting the maximum depth to 4 means that the checker can only detect cycles with up to four transactions. The execution histories we collected do not contain any cycles with more than four transactions. As such, we set the maximum depth to 4, and it was sufficient for detecting the isolation anti-patterns (e.g., a cycle of length four in a *long fork* anomaly).

```
FOR start IN txn FOR vertex, edge, path IN 2..4 OUTBOUND start._id GRAPH txn_g
FILTER LAST(path.edges[*]._to) == start._id AND NOT REGEX_TEST(CONCAT_SEPARATOR(" ",path.
     edges[*].type),"(^rw.*rw$|rw rw)") LIMIT 1 RETURN path.edges
```

Listing 1.1: Checking SI: ArangoDB-Cycle Checker

Shortest-path-based Checkers In ArangoDB, the `ArangoDB-SP` checker (Listing 1.2) uses the shortest path algorithm to detect cycles. It iterates over dependency edges, using `K_SHORTEST_PATH` to find all the so-called *back paths*, from its end vertex to its starting one. It then detects cycles, by trying to connect each edge to its back paths. Cycles that match the anti-pattern can be filtered using ArangoDB functionalities. In Neo4j, the `Neo4j-APOC` checker runs a Cypher query to detect all the cycles on the set of vertices in the dependency graph. As `ArangoDB-SP`, its strategy is based on the shortest path algorithm, finding back paths and forming cycles. After the query returns all cycles, these are filtered to find anti-patterns for a certain isolation level (e.g., SI).

```
LET cycles = ( FOR edge IN dep
FOR p IN OUTBOUND K_SHORTEST_PATHS edge._to TO edge._from GRAPH txn_g
RETURN {edges: UNSHIFT(p.edges, edge),
        vertices: UNSHIFT(p.vertices, p.vertices[LENGTH(p.vertices) - 1])})

FOR cycle IN cycles
FILTER NOT REGEX_TEST(CONCAT_SEPARATOR(" ", cycle.edges[*].type), "(^rw.*rw$|rw rw)")
LIMIT 1 RETURN cycle
```

Listing 1.2: Checking SI: ArangoDB-SP

SCC-based Checkers In ArangoDB, the `ArangoDB-Pregel` checker uses the Pregel SCC algorithm to search for all strongly connected components (SCCs) within a dependency graph and filter those with at least two vertices. This ensures the existence of at least one cycle. In Neo4j, the `Neo4j-GDS-SCC` checker (Listing 1.3) runs a query to find all SCCs with Gabow's path-based SCC algorithm [17] and filters those with at least two vertices. For SER, PL-2, and PL-1, it first filters the graph to remove unwanted edges, as then any SCC will directly lead to an anti-pattern. For SI and PSI, it finds the SCCs and determines whether the subgraph with vertices only from an SCC can form a cycle.

```
CALL gds.alpha.scc.stream('g', {}) YIELD nodeId, componentId WITH componentId,
COLLECT(nodeId) AS ns, COUNT(nodeId) AS num WHERE num > 1 RETURN ns

CALL gds.graph.project.cypher('g','MATCH (n:txn) RETURN id(n) AS id',
'MATCH (n:txn)-->(n2:txn) RETURN id(n) AS source, id(n2) AS target')
```

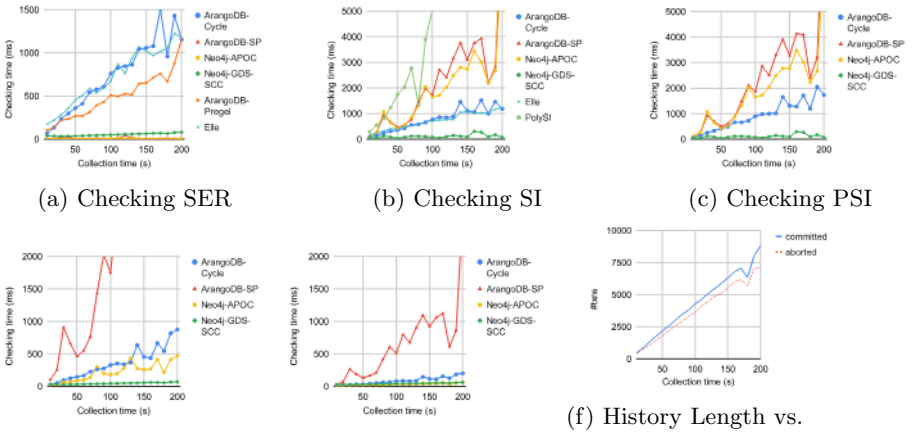
Listing 1.3: Neo4j GDS SCC Checker

Challenges. A major challenge of implementing the checkers in graph databases is that each provides different functionalities. For example, ArangoDB does not support local SCC algorithms, except for the Pregel-based SCC one, while Neo4j supports Gabow's path-based algorithm, which can be directly called from Cypher queries. While this restricts implementations within a graph database, as GRAIL is intended to be system-agnostic, users can choose their framework. Second, database query languages are often not fully-fledged programming languages and their built-in data structures are not always suitable to particular user needs. For example, ArangoDB does not support stacks or hash sets, which makes it difficult to further accelerate the graph queries. As such, we cannot record previous values in a hash set to ensure $O(1)$ access; arrays are the only option. Also, user optimizations are further limited by missing stack structures and by the difficulty of implementing linear-time SCC algorithms, such as Tarjan's. Finally, the query functions of the same algorithm may take different arguments across graph databases and, thus, exhibit performance changes. For example, ArangoDB's shortest path algorithm traverses the whole graph and does not support additional filtering. However, Neo4j can filter a subgraph based on labels provided by users and then execute the traversal on a reduced edge space.

3 Experimental Analysis

We compare the performance of the graph queries presented in Section 2.3 for checking anti-patterns and evaluate their performance, effectiveness, and scalability compared to the state-of-the-art isolation checkers. We use execution histories collected on the cluster setting of ArangoDB with an increasing collection time of execution histories (with/without network partitions) and an increasing rate of submitted client transactions. More information about the datasets and their dependency graph characteristics can be found in our repository [14].

Configuration. We ran all experiments on Linux Mint 21 with AMD Ryzen 7 5800H CPU, Radeon Graphics × 8 GPU, and 15.5GB RAM, using Neo4j Community Ed. 4.4.5, APOC 4.4.0.15, GDS 2.1.13, and ArangoDB 3.9.10.



(a) Checking SER (b) Checking SI (c) Checking PSI

(d) Checking PL-2 (e) Checking PL-1 (f) History Length vs. Collection time

Fig. 3: Checking anomalies with increasing collection time of list histories

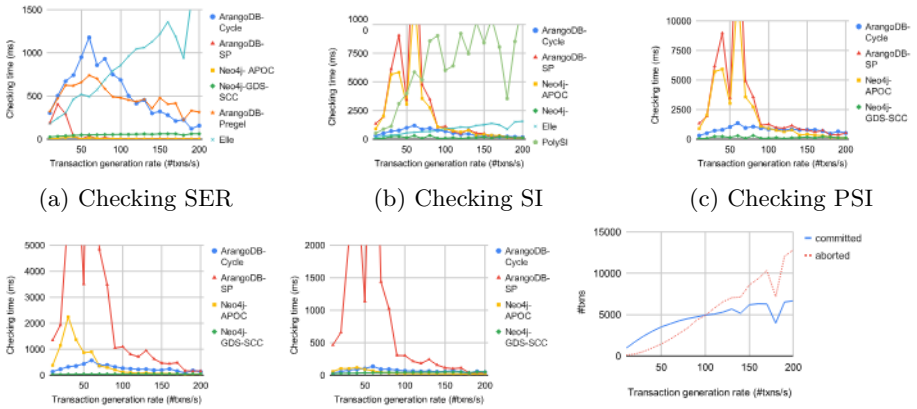
Experiments on histories with increasing collection time. Figure 3 shows the performance trends of different queries for checking SER, SI, PSI, PL-2, and PL-1 on increasing length of history collection. For `ArangoDB-Cycle`, we used the cycle-length bound $d = 4$, which suffices to detect all violating anti-patterns to all isolation levels in our experiments. Figures 3b-3e omit `ArangoDB-Pregel`, since it can only check for serializability.

Figure 3a shows the evolution of the checking time (**ms**) for SER with increasing collection time. `ArangoDB-Cycle` has the highest increase in analysis time because it traverses the graph to find all cycles with a polynomial complexity in the graph size. `ArangoDB-Pregel` exhibits similar polynomial behavior, as it runs BSP super step computations to search for SCCs with a linear cost for each step. However, `ArangoDB-SP`, `Neo4j-APOC` and `Neo4j-GDS-SCC`, have low analysis times with insignificant history length increases, as they search for the shortest paths and return as soon as they find a cycle.

For SI and PSI (Figures 3b and 3c), the trends of `ArangoDB-Cycle` and `Neo4j-GDS-SCC` are similar to those for SER. `ArangoDB-SP` and `Neo4j-APOC` take longer to check SI and PSI than SER, as these are more complex and require more graph traversals. For PL-1 and PL-2, `ArangoDB-SP` shows a significant performance degradation, but `Neo4j-APOC` remains efficient and scalable as it combines shortest-path detection with filtering.

We observed that network partitions in the test executions do not result in more anti-patterns. With network faults, history length is reduced without otherwise affecting the dependency graphs. This can be different for other databases. Since ArangoDB does not guarantee serializability in the cluster setting, its executions exhibit isolation anomalies without introducing faults.

Experiments on histories with increasing transaction rate. Figure 4 shows the checkers' scalability for SER, SI, PSI, PL-2, and PL-1 when increasing the transaction generation rate. Overall, the relative performances of the checkers are



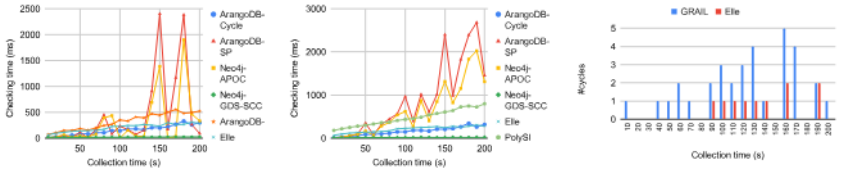
(a) Checking SER (b) Checking SI (c) Checking PSI

(d) Checking PL-2 (e) Checking PL-1 (f) History Length vs. Rate

Fig. 4: Runtime for checking anomalies in the increasing rate of transactions

similar to those of increasing history length. That is, when there are violations in the execution history, `ArangoDB-Cycle` requires more analysis time than the shortest-path and SCC-based checkers since it traverses the graph for each vertex. The performances of `ArangoDB-SP` and `Neo4j-APOC` degrade for execution histories without violations since they analyze the shortest paths between all vertices. `Neo4j-GDS-SCC` remains efficient for all isolation levels.

A significant difference from the scalability analysis in Figure 3 is that increasing the transaction generation rate to 80 **txns/s** or higher *decreases* the analysis time for the graph-based checkers. Our analysis shows that this is caused by the decreasing density of dependency graphs. As we generate a higher number of concurrent transactions, more transactions conflict and abort. The graph density of datasets can be found in the online documentation [14].



(a) Checking SER (b) Checking SI (c) Detected #cycles

Fig. 5: (a - b) Checking register histories (c) Cycle detection

Comparison to the State-of-the-art Checkers We compare the graph query-based checkers with state-of-the-art isolation checkers, i.e., Elle [2] for SER, and PolySI [19] for SI, which has been shown to outperform other SI checkers.

Figures 3a, 4a, 5a present the analysis time of checking SER with Elle and, Figures 3b, 4b 5b presents for checking SI with Elle and PolySI. In the figures, we report the analysis time of the checkers that detect violations. Figure 5 shows the performance of checking SER and SI on register variables with increasing

history length. We only present the plots for SER and SI for space reasons, the performance trends of the checkers for other isolation levels are similar to those in Figure 3. These trends are also similar to those in Figure 5, except for checking serializability with `ArangoDB-SP` and `Neo4j-APOC`. They need more time to check register histories without violations, since checking them requires a full exploration of the shortest paths. Graph query-based checkers are faster than Elle for histories with lists (Figure 3a), except for `ArangoDB-Cycle`, due to the cost of its cycle detection by graph traversal on each vertex. For the histories with registers (Figure 5a), Elle's performance stays more stable, while `ArangoDB-SP` and `Neo4j-APOC` have degraded performance, especially for executions without violations. Similarly, for SI (Figures 3b and 5b), Elle outperforms `ArangoDB-SP` and `Neo4j-APOC`, has comparable performance with `ArangoDB-Cycle`, and performs worse than `Neo4j-GDS-SCC`. When increasing the transaction generation rate (Figure 4a), Elle's performance significantly declines. Our results show that `Neo4j-GDS-SCC` significantly outperforms Elle in all cases. Figure 5c compares the number of cycles detected on the execution histories with registers. Elle can detect the isolation anomalies in the histories with list variables using their traceability and recoverability. We also use the database's write-ahead logs to infer dependencies on register variables, and thus can detect more violations.

Figures 3b, 5b, and 4b show that PolySI's performance degrades with increasing history length or transaction generation rate. PolySI introduces and uses the novel polygraph data structure, designed to characterize SI violations. PolySI generates polygraphs, recovering WR edges based on the *unique-writes assumption*, adds the RW edges based on the inferred version order, and then enumerates and prunes possible WW edges to recover the orders among transactions. This however largely expands the number of graphs to analyze.

4 Conclusion and Perspectives

We explore the feasibility and benefits of using graph databases to check isolation violations. As such, we introduce the GRAIL approach, which is *the first, to the best of our knowledge, to propose a generic methodology for isolation violation detection with graph queries*. We implement GRAIL through five checkers, using AQL and Cypher, and evaluate these against the state-of-the-art Elle and PolySI tools. Our results show that the graph query-based checkers provide comparative performance and remain scalable when increasing execution history length and transaction generation rates. The performance of the `ArangoDB-Cycle` definition-based checker highly depends on the characteristics of the analyzed execution, while SCC-based checkers exhibit less performance degradation than shortest-path-based ones, especially for executions without violations. Among these, `Neo4j-GDS-SCC` outperforms all other baselines.

Our empirical analysis also helps distill insights into the challenges of repurposing graph databases and queries as isolation checkers. Future work can analyze a more extensive set of patterns, including other consistency and isolation models. Other perspectives include extending our approach to runtime analyses of database executions and evolving graph processing systems.

Bibliography

[1] Adya, A., Liskov, B., O'Neil, P.E.: Generalized isolation level definitions. In: ICDE. pp. 67–78. IEEE Computer Society (2000)

[2] Alvaro, P., Kingsbury, K.: Elle: Inferring isolation anomalies from experimental observations. Proc. VLDB Endow. **14**(3), 268–280 (2020)

[3] Angles, R.: The property graph database model. In: AMW. CEUR Workshop Proceedings, vol. 2100. CEUR-WS.org (2018)

[4] Angles, R., Bonifati, A., Dumbrava, S., Fletcher, G., Green, A., Hidders, J., Li, B., Libkin, L., Marsault, V., Martens, W., Murlak, F., Plantikow, S., Savkovic, O., Schmidt, M., Sequeda, J., Staworko, S., Tomaszuk, D., Voigt, H., Vrgoc, D., Wu, M., Zivkovic, D.: PG-Schema: Schemas for Property Graphs. Proc. ACM Manag. Data **1**(2), 198:1–198:25 (2023)

[5] AWS: Amazon Neptune. `https://aws.amazon.com/neptune/` (2024)

[6] Berenson, H., Bernstein, P.A., Gray, J., Melton, J., O'Neil, E.J., O'Neil, P.E.: A critique of ANSI SQL isolation levels. In: SIGMOD Conference. pp. 1–10. ACM Press (1995)

[7] Bernstein, P.A., Goodman, N.: Concurrency control in distributed database systems. ACM Comput. Surv. **13**(2), 185–221 (1981)

[8] Besta, M., Gerstenberger, R., Peter, E., Fischer, M., Podstawski, M., Barthels, C., Alonso, G., Hoefler, T.: Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. ACM Comput. Surv. **56**(2), 31:1–31:40 (2024)

[9] Biswas, R., Enea, C.: On the complexity of checking transactional consistency. Proc. ACM Program. Lang. **3**(OOPSLA), 165:1–165:28 (2019)

[10] Bonifati, A., Dumbrava, S.: Graph queries: From theory to practice. SIGMOD Rec. **47**(4), 5–16 (2018)

[11] Cerone, A., Gotsman, A.: Analysing snapshot isolation. In: PODC. pp. 55–64. ACM (2016)

[12] Clark, J.: Verifying serializability protocols with version order recovery (2021). https://doi.org/10.3929/ethz-b-000507577

[13] Dumbrava, S., Jin, Z., Kulahcioglu Ozkan, B., Qiu, J.: GRAIL: Checking Transaction Isolation Violations with Graph Queries. In: ICSE Poster Track (2024), `https://hal.science/hal-04475697`

[14] Dumbrava, S., Jin, Z., Ozkan, B.K., Qiu, J.: GRAph-based Isolation Level Checker (GRAIL) Artifact (2023), `https://github.com/jasonqiu98/GRAIL-artifact`

[15] Erling, O., Averbuch, A., Larriba-Pey, J.L., Chafi, H., Gubichev, A., Prat-Pérez, A., Pham, M., Boncz, P.A.: The LDBC social network benchmark: Interactive workload. In: SIGMOD Conference. pp. 619–630. ACM (2015)

[16] Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., Taylor, A.: Cypher: An evolving query language for property graphs. In: SIGMOD Conference. pp. 1433–1445. ACM (2018)

[17] Gabow, H.N.: Path-based depth-first search for strong and biconnected components. Inf. Process. Lett. **74**(3-4), 107–114 (2000)
[18] GQL: GQL graph query language. https://www.gqlstandards.org/ (2024)
[19] Huang, K., Liu, S., Chen, Z., Wei, H., Basin, D.A., Li, H., Pan, A.: Efficient black-box checking of snapshot isolation in databases. Proc. VLDB Endow. **16**(6), 1264–1276 (2023)
[20] Inc., A.: ArangoDB. https://www.arangodb.com/ (2023)
[21] Inc., B.G.: AgensGraph. https://bitnine.net/agensgraph/ (2024)
[22] Kingsbury., K.: Gretchen: Offline serializability verification, in clojure. (2022), https://github.com/aphyr/gretchen
[23] Kingsbury., K.: Jepsen (2022), http://jepsen.io/
[24] Kingsbury., K.: Jepsen analyses (2022), https://jepsen.io/analyses
[25] Labs, R.: RedisGraph. https://oss.redislabs.com/redisgraph/ (2017)
[26] Memgraph: Memgraph. https://memgraph.com/ (2024)
[27] Neo4j: Neo4j. https://neo4j.com/ (2023)
[28] Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4), 631–653 (1979)
[29] Paradies, M.: Graph pattern matching in SAP HANA. https://tinyurl.com/ycxu54pr (2017)
[30] Sakr, S., Bonifati, A., Voigt, H., Iosup, A., Ammar, K., Angles, R., Aref, W.G., Arenas, M., Besta, M., Boncz, P.A., Daudjee, K., Valle, E.D., Dumbrava, S., Hartig, O., Haslhofer, B., Hegeman, T., Hidders, J., Hose, K., Iamnitchi, A., Kalavri, V., Kapp, H., Martens, W., Özsu, M.T., Peukert, E., Plantikow, S., Ragab, M., Ripeanu, M., Salihoglu, S., Schulz, C., Selmer, P., Sequeda, J.F., Shinavier, J., Szárnyas, G., Tommasini, R., Tumeo, A., Uta, A., Varbanescu, A.L., Wu, H., Yakovets, N., Yan, D., Yoneki, E.: The future is big graphs: a community view on graph processing systems. Commun. ACM **64**(9), 62–71 (2021)
[31] Sinha, A., Malik, S., Wang, C., Gupta, A.: Predicting Serializability Violations: SMT-Based Search vs. DPOR-Based Search. In: Haifa Verification Conference. Lecture Notes in Computer Science, vol. 7261, pp. 95–114. Springer (2011)
[32] solidIT: DB-Engines Ranking. https://db-engines.com/en/ranking_trend/graph+dbms (2024)
[33] Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: SOSP. pp. 385–400. ACM (2011)
[34] Tan, C., Zhao, C., Mu, S., Walfish, M.: Cobra: Making transactional key-value stores verifiably serializable. In: OSDI. pp. 63–80. USENIX Association (2020)
[35] Waudby, J., Steer, B.A., Karimov, K., Marton, J., Boncz, P.A., Szárnyas, G.: Towards testing ACID compliance in the LDBC social network benchmark. In: TPCTC. Lecture Notes in Computer Science, vol. 12752, pp. 1–17. Springer (2020)
[36] Zhang, J., Ji, Y., Mu, S., Tan, C.: Viper: A fast snapshot isolation checker. In: EuroSys. pp. 654–671. ACM (2023)