Towards Testing ACID Compliance in the LDBC Social Network Benchmark

Jack Waudby[1], Benjamin A. Steer[2], Karim Karimov[3], József Marton[4], Peter Boncz[5], and Gábor Szárnyas[3,6]

[1] Newcastle University, School of Computing, j.waudby2@newcastle.ac.uk
[2] Queen Mary University of London, b.a.steer@qmul.ac.uk
[3] Budapest University of Technology and Economics
Department of Measurement and Information Systems
[4] Budapest University of Technology and Economics
Department of Telecommunications and Media Informatics
[5] CWI, Amsterdam, boncz@cwi.nl
[6] MTA-BME Lendület Cyber-Physical Systems Research Group
szarnyas@mit.bme.hu

Abstract. Verifying ACID compliance is an essential part of database benchmarking, because the integrity of performance results can be undermined as the performance benefits of operating with weaker safety guarantees (at the potential cost of correctness) are well known. Traditionally, benchmarks have specified a number of tests to validate ACID compliance. However, these tests have been formulated in the context of relational database systems and SQL, whereas our scope of benchmarking are systems for graph data, many of which are non-relational. This paper presents a set of data model-agnostic ACID compliance tests for the LDBC (Linked Data Benchmark Council) Social Network Benchmark suite's Interactive (SNB-I) workload, a transaction processing benchmark for graph databases. We test all ACID properties with a particular emphasis on isolation, covering 10 transaction anomalies in total. We present results from implementing the test suite on 5 database systems.

1 Introduction

Context. Organizations often complement their existing data processing pipelines with systems dedicated to analying graphs such as graph databases [7], graph analytical frameworks [6], and graph streaming engines [8]. The category of *graph databases* broadly refers to transactional systems that use the *property graph* data model, where nodes and edges can be annotated with key-value pairs of attributes. Such systems typically use a schema-free data model and provide operators with stronger expressive power than relational algebra, including transitive reachability, shortest path and regular path queries [2].

To stimulate competition between graph database vendors and allow fair comparison of their systems, several benchmarks have been proposed to capture realistic workloads, including those of the Linked Data Benchmark Council

(LDBC) [3]. In particular, the LDBC's Social Network Benchmark Interactive workload (SNB-I) was designed to target transactional graph databases [10]. To provide protection against violations of correctness arising from the concurrent execution of transactions and system failures, such transactional databases provide *Atomicity*, *Consistency*, *Isolation*, and *Durability* (ACID) guarantees.

Problem. Verifying ACID compliance is an important step in the benchmarking process for enabling fair comparison between systems. The performance benefits of operating with weaker safety guarantees are well established [13] but this can come at the cost of application correctness. To enable apples vs. apples performance comparisons between systems it is expected they uphold the ACID properties. Currently LDBC provides no mechanism for validating ACID compliance within the SNB-I workflow. A simple solution would be to outsource the responsibility of demonstrating ACID compliance to benchmark implementors. However, the safety properties claimed by a system often do not match observable behaviour [14]. To mitigate this problem, benchmarks such as TPC-C [20] include a number of ACID tests to be executed as part of the benchmarking auditing process. However, we found these tests cannot readily be applied to our context, as they assume lock-based concurrency control and an interactive query API that provides clients with explicit control over a transaction's lifecyle. Modern data systems often use optimistic concurrency control mechanisms [17] and offer a restricted query API, such as only executing transactions as stored procedures [19]. Further, tests that trigger and test row-level locking phenomena, for instance, do not readily map on graph database systems. Lastly, we found these tests are limited in the range of isolation anomalies they cover.

Contribution. This paper presents the design of an implementation agnostic ACID compliance test suite for LDBC SNB-I[7]. Our guiding design principle was to be agnostic of system-level implementation details, relying solely on client observations to determine the occurrence of non-transactional behavior. Thus all systems can be subjected to the same tests and fair comparisons between SNB-I performance results can be drawn. Tests are described in the context of a graph database employing the property graph data model [2]. Reference implementations are given in Cypher [12], the *de facto* standard graph query language. Particular emphasis is given to testing isolation, covering 10 known anomalies including recently discovered anomalies such as *Observed Transaction Vanishes* [4] and *Fractured Reads* [5]. The test suite has been implemented for 5 database systems.[8] A conscious decision was made to keep tests relatively lightweight, as to not add significant overhead to the benchmarking process.

Structure. The remainder of the paper is structured as follows: Section 2 provides an overview of the SNB-I workload. Sections 3 and 4, describe the Atomicity, and Isolation tests, respectively. In Section 5 we present results from running our tests

[7] We acknowledge verifying ACID-compliance with a finite set of tests is not possible. However, the goal is not an exhaustive quality assurance test of a system's safety properties but rather to demonstrate that ACID guarantees are supported.

[8] Available at `https://github.com/ldbc/ldbc_acid`.

on real-world systems. We discuss related work in Section 6 and briefly touch on consistency and durability test in Section 7 before concluding in Section 8.

2 SNB Interactive Workload

The goal behind LDBC's Social Network Benchmark Interactive workload was to motivate the maturing of transactional graph processing systems. SNB-I defines a schema to represent a network of *Person*s who communicate through *Post*s in *Forum*s. SNB-I consists of 14 complex read and 7 short read queries. There are 8 transactional update operations that insert vertices and edges to the graph. Whilst it is expected a system provides ACID transactions, from a transaction processing perspective, there is little contention in SNB-I's transactions. This makes the occurrence of non-transactional behavior rare and unfortunately makes the already defined update operations unsuitable for testing most of the ACID properties. To address this limitation, this paper presents a test suite of new transactions. These tests are defined on a small core of LDBC SNB schema (extended with properties for versioning) given in Figure 1.

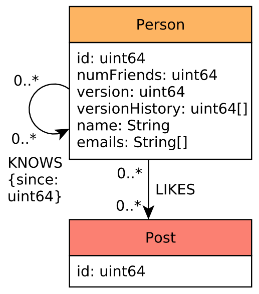

Fig. 1: Graph schema.

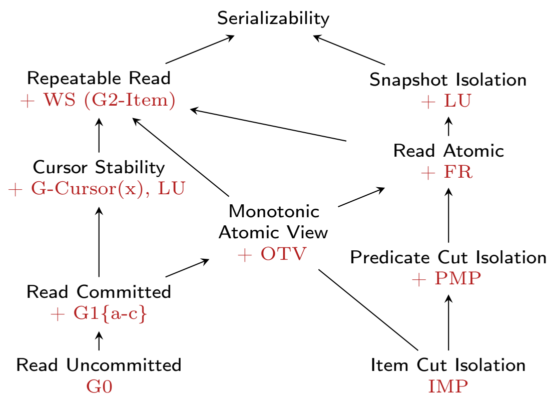

Fig. 2: Hierarchy of isolation levels as described in [5]. All anomalies are covered except G-Cursor(x).

3 Atomicity

Atomicity ensures that either all of a transaction's actions are performed, or none are. Two atomicity tests have been developed. **Atomicity-C** checks for every successful commit message a client receives that any data items inserted or modified are subsequently visible. **Atomicity-RB** checks for every aborted transaction that all its modifications are not visible. Tests are executed as follows: (i) load a graph of `Person` nodes (Listing 1.1) each with a unique `id` and a set of `emails`; (ii) a client executes a full graph scan counting the number of nodes, edges and emails (Listing 1.4) using the result to initialize a counter `committed`; (iii) N transaction instances (Listing 1.2, Listing 1.3) of the required test are then executed, `committed` is incremented for each successful commit; (iii) repeat the full graph scan, storing the result in the variable `finalState`; (iv) perform the anomaly check: `committed=finalState`.

The **Atomicity-C** transaction (Listing 1.2) randomly selects a `Person`, creates a new `Person`, inserts a `KNOWS` edge and appends an `email`. The **Atomicity-RB** transaction (Listing 1.3) randomly selects a `Person`, appends an `email` and attempts to insert a `Person` only if it does not exist. Note, for **Atomicity-RB** if the query API does not offer a `ROLLBACK` statement constraints such as node uniqueness can be utilized to trigger an abort.

```
CREATE (:Person {id: 1, name: 'Alice', emails: ['alice@aol.com']}),
       (:Person {id: 2, name: 'Bob', emails: ['bob@hotmail.com', 'bobby@yahoo.com']})
```

Listing 1.1: Cypher query for creating initial data for the Atomicity transactions.

```
«BEGIN»
MATCH (p1:Person {id: $person1Id})
CREATE (p1)-[k:KNOWS]->(p2:Person)
SET
  p1.emails = p1.emails + [$newEmail],
  p2.id = $person2Id,
  k.creationDate = $creationDate
«COMMIT»
```

Listing 1.2: Atomicity-C Tx.

```
«BEGIN»
MATCH (p1:Person {id: $person1Id})
SET p1.emails = p1.emails + [$newEmail]
«IF» MATCH (p2:Person {id: $person2Id}) exists
«THEN» «ABORT» «ELSE»
CREATE (p2:Person {id: $person2Id, emails: []})
«END»
«COMMIT»
```

Listing 1.3: Atomicity-RB Tx.

```
MATCH (p:Person)
RETURN count(p) AS numPersons, count(p.name) AS numNames, sum(size(p.emails)) AS numEmails
```

Listing 1.4: Atomicity-C/Atomicity-RB: counting entities in the graph.

4 Isolation

The gold standard isolation level is Serializability, which offers protection against all possible *anomalies* that can occur from the concurrent execution of transactions. Anomalies are occurrences of non-serializable behavior. Providing Serializability can be detrimental to performance [13]. Thus systems offer numerous weak isolation levels such as Read Committed and Snapshot Isolation that allow a higher degree of concurrency at the cost of potential non-serializable behavior. As such, isolation levels are defined in terms of the anomalies they prevent [13,4]. Figure 2 relates isolation levels to the anomalies they proscribe.

SNB-I does not require systems to provide Serializability [3]. However, to allow fair comparison systems must disclose the isolation level used during benchmark execution. The purpose of these isolation tests is to verify the claimed isolation level matches the expected behavior. For this tests have been developed for each anomaly presented in [5]. Formal definitions for each anomaly are reproduced from [1,5] using their system model which is described below. General design considerations are then discussed, before each test is described.

4.1 System Model

Transactions consist of an ordered sequence of read and write operations to an arbitrary set of data items, book-ended by a `BEGIN` operation and a `COMMIT` or an `ABORT` operation. In a graph database data items are nodes, edges and properties. The set of items a transaction reads from and writes to is termed its *item read set* and *item write set*. Each write creates a *version* of an item, which is assigned a unique timestamp taken from a totally ordered set (e.g. natural numbers) version i of item x is denoted x_i. All data items have an initial *unborn* version $\perp$ produced by an initial transaction $T_\perp$. The unborn version is located at the start of each item's version order. An execution of transactions on a database is represented by a *history*, H, consisting of (i) each transaction's read and write operations, (ii) data item versions read and written and (iii) commit or abort operations.

There are three types of dependencies between transactions, which capture the ways in which transactions can *directly* conflict. *Read dependencies* capture the scenario where a transaction reads another transaction's write. *Antidependencies* capture the scenario where a transaction overwrites the version another transaction reads. *Write dependencies* capture the scenario where a transaction overwrites the version another transaction writes. Their definitions are as follows:

Read-Depends Transaction T_j *directly read-depends* (wr) on T_i if T_i writes some version x_k and T_j reads x_k.

Anti-Depends Transaction T_j *directly anti-depends* (rw) on T_i if T_i reads some version x_k and T_j writes x's next version after x_k in the version order.

Write-Depends Transaction T_j *directly write-depends* (ww) on T_i if T_i writes some version x_k and T_j writes x's next version after x_k in the version order.

Using these definitions, from a history H a *direct serialization graph* $DSG(H)$ is constructed. Each node in the *DSG* corresponds to a committed transaction and edges correspond to the types of direct conflicts between transactions. Anomalies can then be defined by stating properties about the *DSG*.

The above *item-based* model can be extended to handle *predicate-based* operations [1]. Database operations are frequently performed on set of items provided a certain condition called the *predicate*, P holds. When a transaction executes a read or write based on a predicate P, the database selects a version for each item to which P applies, this is called the version set of the predicate-based denoted as $Vset(P)$. A transaction T_j changes the matches of a predicate-based read $r_i(P_i)$ if T_i overwrites a version in $Vset(P_i)$.

4.2 General Design

Isolation tests begin by loading a *test graph* into the database. Configurable numbers of *write clients* and *read clients* then execute a sequence of transactions on the database for some configurable time period. After execution, results from read clients are collected and an *anomaly check* is performed. In some tests an additional full graph scan is performed after the execution period in order to collect information required for the anomaly check.

The guiding principle behind test design was the preservation of data item's version history – the key ingredient needed in the system model formalization which is often not readily available to clients, if preserved at all. Several anomalies are closely related, tests therefore had to be constructed such that other anomalies could not interfere with or mask the detection of the targeted anomaly. Test descriptions provide (i) informal and formal anomaly definitions, (ii) the required test graph, (iii) description of transaction profiles write and read clients execute, and (iv) reasoning for why the test works.

4.3 Dirty Write

Informally, a *Dirty Write* (Adya's G0 [1]) occurs when updates by conflicting transactions are interleaved. For example, say T_i and T_j both modify items $\{x, y\}$. If version x_i precedes version x_j and y_j precedes version y_i a G0 anomaly has occurred. Preventing G0 is especially important in a graph database in to order to maintain *Reciprocal Consistency* [21].

Definition. A history H exhibits phenomenon G0 if $DSG(H)$ contains a directed cycle consisting entirely of write-dependency edges.

Test. Load a test graph containing pairs of `Person` nodes connected by a `KNOWS` edge. Assign each `Person` a unique `id` and each `Person` and `KNOWS` edge a `versionHistory` property of type list (initially empty). During the execution period, write clients execute a sequence of G0 T_{W} instances, Listing 1.5. This transaction appends its ID to the `versionHistory` property for each entity in the `Person` pair it matches. Note, transaction IDs are assumed to be globally unique. After execution, a read client issues a G0 T_{R} for each `Person` pair in the graph, Listing 1.6. Retrieving the `versionHistory` for each entity (2 `Persons` and 1 `KNOWS` edge) in a `Person` pair.

Anomaly check. For each `Person` pair in the test graph: (i) prune each `versionHistory` list to remove any version numbers that do not appear in all lists; needed to account for interference from *Lost Update* anomalies (Section 4.8), (ii) perform an element-wise comparison between `versionHistory` lists for each entity, (iii) if lists do not agree a G0 anomaly has occurred.

Why it works. Each G0 T_{W} effectively creates a new version of a `Person` pair. Appending the transaction ID preserves the version history of each entity in the `Person` pair. In a system that prevents G0, each entity of the `Person` pair should experience the *same* updates, in the *same* order. Hence, each position in the `versionHistory` lists should be equivalent. The additional pruning step is needed as *Lost Updates* overwrite a version, effectively erasing it from the history of a data item.

```
MATCH
  (p1:Person {id: $person1Id})
  -[k:KNOWS]->(p2:Person {id: $person2Id})
SET p1.versionHistory = p1.versionHistory + [$tId]
SET p2.versionHistory = p2.versionHistory + [$tId]
SET k.versionHistory  = k.versionHistory  + [$tId]
```

Listing 1.5: Dirty Write (G0) T_{W}.

```
MATCH (p1:Person {id: $person1Id})
-[k:KNOWS]->(p2:Person {id: $person2Id})
RETURN
  p1.versionHistory AS p1VersionHistory,
  k.versionHistory  AS kVersionHistory,
  p2.versionHistory AS p2VersionHistory
```

Listing 1.6: Dirty Write (G0) T_{R}.

4.4 Dirty Reads

Aborted Reads

Informally, an *Aborted Read* (G1a) anomaly occurs when a transaction reads the updates of a transaction that later aborts.
Definition. A history H exhibits phenomenon G1a if H contains an aborted transaction T_i and a committed transaction T_j such that T_j reads a version written by T_i.
Test. Load a test graph containing only `Person` nodes into the database. Assign each `Person` a unique `id` and `version` initialized to 1; any odd number will suffice. During execution, write clients execute a sequence of G1a T_{W} instances, Listing 1.7. Selecting a random `Person id` to populate each instance. This transaction attempts to set `version=2` (any even number will suffice) but always aborts. Concurrently, read clients execute a sequence of G1a T_{R} instances, Listing 1.8. This transaction retrieves the `version` property of a `Person`. Read clients store results, which are pooled after execution has finished.
Anomaly check. Each read should return `version=1` (or any odd number). Otherwise, an G1a anomaly has occurred.
Why it works. Each transaction that attempts to set `version` to an even number *always* aborts. Therefore, if a transaction reads `version` to be an even number, it must have read the write of an aborted transaction.

```
MATCH (p:Person {id: $personId})
SET p.version = 2
«SLEEP($sleepTime)»
«ABORT»
```

Listing 1.7: Aborted Read (G1a) T_{W}.

```
MATCH (p:Person {id: $personId})
SET p.version = $even
«SLEEP($sleepTime)»
SET p.version = $odd
```

Listing 1.9: Interm. Read (G1b) T_{W}.

```
MATCH (p:Person {id: $personId})
RETURN p.version
```

Listing 1.8: Aborted Read (G1a) T_{R}.

```
MATCH (p:Person {id: $personId})
RETURN p.version
```

Listing 1.10: Interm. Read (G1b) T_{R}.

Intermediate Reads

Informally, an *Intermediate Read* (Adya's G1b [1]) anomaly occurs when a transaction reads the intermediate modifications of other transactions.

Definition. A history H exhibits phenomenon G1b if H contains a committed transaction T_i that reads a version of an object x_m written by transaction T_j, and T_j also wrote a version x_n such that $m < n$ in x's version order.
Test. Load a test graph containing only `Person` nodes into the database. Assign each `Person` a unique `id` and `version` initialized to 1; any odd number will suffice. During execution, write clients execute a sequence of G1b T_{W} instances, Listing 1.9. This transaction sets `version` to an even number, then an odd number before committing. Concurrently read-clients execute a sequence of G1b T_{R} instances, Listing 1.10. Selecting a `Person` by `id` and retrieving its `version` property. Read clients store results which are collected after execution has finished.
Anomaly check. Each read of `version` should be an odd number. Otherwise, an G1b anomaly has occurred.
Why it works. The final version installed by an G1b T_{W} instance is *never* an even number. Therefore, if a transaction reads `version` to be an even number it must have read an intermediate version.

Circular Information Flow

Informally, a *Circular Information Flow* (Adya's G1c [1]) anomaly occurs when two transactions affect each other; i.e. both transactions write information the other reads. For example, transaction T_i reads a write by transaction T_j and transaction T_j reads a write by T_i.
Definition. A history H exhibits phenomenon G1c if $DSG(H)$ contains a directed cycle that consists entirely of read-dependency and write-dependency edges.
Test. Load a test graph containing only `Person` nodes into the database. Assign each `Person` a unique `id` and `version` initialized to 0. Read-write clients are required for this test, executing a sequence of G1c T_{RW}, Listing 1.11. This transaction selects two different `Person` nodes, setting the `version` of one `Person` to the transaction ID and retrieving the `version` from the other. Note, transaction IDs are assumed to be globally unique. Transaction results are stored in format `(txn.id, versionRead)` and collected after execution.

```
MATCH (p1:Person {id: $person1Id}) SET p1.version = $transactionId
MATCH (p2:Person {id: $person2Id}) RETURN p2.version
```

Listing 1.11: G1c T_{RW}.

Anomaly check. For each result, check the result of the transaction the `versionRead` corresponds to, did not read the transaction of that result. If so a G1c anomaly has occurred.
Why it works. Consider the result set: `{(`T_1`, `T_2`), (`T_2`, `T_3`), (`T_3`, `T_2`)}`. T_1 reads the version written by T_2 and T_2 reads the version written by T_3. Here information flow is unidirectional from T_1 to T_2. However, T_2 reads the version written by T_3 and T_2 reads the version written by T_3. Here information flow is circular from T_2 to T_3 and T_3 to T_2. Thus a G1c anomaly has been detected.

4.5 Cut Anomalies

Item-Many-Preceders

Informally, an *Item-Many-Preceders* (IMP) anomaly [4] occurs if a transaction observes multiple versions of the same item (e.g. transaction T_i reads versions x_1 and x_2). In a graph database this can be multiple reads of a node, edge, property or label. Local transactions (involving a single data item) occur frequently in graph databases, e.g. in *"Retrieve content of a message"* (SNB-I Short Read 4 [3]).

Definition. A history H exhibits IMP if $DSG(H)$ contains a transaction T_i such that T_i directly *item-read-depends* on x by more than one other transaction.

Test. Load a test graph containing `Person` nodes. Assign each `Person` a unique `id` and `version` initialized to 1. During execution write clients execute a sequence of IMP T_{W} instances, Listing 1.12. Selecting a random `id` and installing a new version of the `Person`. Concurrently read clients execute a sequence of IMP T_{R} instances, Listing 1.13. Performing multiple reads of the same `Person`; successive reads can be separated by some artificially injected wait time to make conditions more favorable for detecting an anomaly. Both reads within an IMP T_{R} transaction are returned, stored and collected after execution.

Anomaly check. Each IMP T_{R} result set `(firstRead, secondRead)` should contain the *same* `Person` version. Otherwise, an IMP anomaly has occurred.

Why it works. By performing successive reads within the same transaction this test checks that a system ensures consistent reads of the same data item. If the version changes then a concurrent transaction has modified the data item and the reading transaction is not protected from this change.

```
MATCH (p:Person {id: $personId})
SET p.version = p.version + 1
```

Listing 1.12: IMP T_{W}.

```
MATCH (pe:Person {id: $personId}), (po:Post {id: $postId)
CREATE (pe)-[:LIKES]->(po)
```

Listing 1.14: PMP T_{W}.

```
MATCH (p1:Person {id: $personId})
WITH p1.version AS firstRead
«SLEEP($sleepTime)»
MATCH (p2:Person {id: $personId})
RETURN firstRead,
  p2.version AS secondRead
```

Listing 1.13: IMP T_{R}.

```
MATCH (po1:Post {id: $postId})<-[:LIKES]-(pe1:Person)
WITH count(pe1) AS firstRead
«SLEEP($sleepTime)»
MATCH (po2:Post {id: $postId})<-[:LIKES]-(pe2:Person)
RETURN firstRead,
  count(pe2) AS secondRead
```

Listing 1.15: PMP T_{R}.

Predicate-Many-Preceders

Informally, a *Predicate-Many-Preceders* (PMP) anomaly [4] occurs if a transaction observes different versions resulting from the same predicate read (e.g. T_i reads $Vset(P_i) = \{x_1\}$ and $Vset(P_i) = \{x_1, y_2\}$). Pattern matching is a common

predicate read operation in a graph database, e.g. query *"Find friends and friends of friends that have been to given countries"* (SNB-I Complex Read 3 [3]).
Definition. A history H exhibits the phenomenon PMP if, for all predicate-based reads $r_i(P_i : Vset(P_i))$ and $r_j(P_j : Vset(P_j))$ in T_k such that the logical ranges of P_i and P_j overlap (call it P_o), the set of transactions that change the matches of P_o for r_i and r_j differ.
Test. Load a test graph containing `Person` and `Post` nodes. Within each node type assign unique `ids`. During execution write clients execute a sequence of PMP T_{W} instances, inserting a `LIKES` edge between a randomly selected `Person` and `Post`, shown in Listing 1.14. Concurrently read clients execute a sequence of PMP T_{R} instances, Listing 1.15. Performing multiple reads of the pattern `(po:Post)<-[:LIKES]-(p:Person)` and counting the number of `LIKES` edges; successive reads can be separated by some artificially injected wait time to make conditions more favorable for detecting an anomaly. Both predicate reads within a PMP T_{R} transaction are returned, stored and collected after test execution.
Anomaly check. For each PMP T_{R} transaction result set `(firstRead, secondRead)`, the `firstRead` should be equal to `secondRead`. Otherwise, a PMP anomaly has occurred.
Why it works. By performing successive predicate reads and counting the number of `LIKES` edges within the same transaction this test checks that a system ensures consistent reads of the same predicate. If the number of `LIKES` edges changes then a concurrent transaction has inserted a new `LIKES` edge and the reading transaction is not protected from this change.

4.6 Observed Transaction Vanishes

Informally, an *Observed Transaction Vanishes* (OTV) anomaly [4] occurs when a transaction observes part of another transaction's updates but not all of them (e.g. T_1 writes x_1 and y_1 and T_2 reads x_1 and $y_\perp$). Before formally defining OTV the *Unfolded Serialization Graph (USG)* must be introduced [1]. The *USG* is specified for an individual transaction, T_i and a history, H and is denoted by $USG(H, T_i)$. In a *USG* the T_i node is split into multiple nodes, one for each action read $r_i(\cdot)$ or write $w_i(\cdot)$ within the transaction. The dependency edges are now incident on the relevant event of T_i. Additionally, actions within T_i are connected by an *order edge* e.g. if T_i reads object y_j then immediately writes on object x an order edge exists from $w_i(x_i)$ to $r_i(y_j)$.
Definition. A history H exhibits phenomenon OTV if $USG(H, T_i)$ contains a directed cycle consisting of (i) exactly one read dependency edge induced by data item x from T_j to T_i and (ii) a set of edges induced by data item y containing at least one anti dependency edge from T_i to T_j. Additionally, T_i's read from y precedes its read from x.
Test. Load a test graph containing a set of cycles of length 4 of `Persons` with same `name` connected by `Knows` edges. Assign each `Person` an `id`, `name` and `version` property (initialized to 1). Note, `id` must be unique across nodes and `name` must be unique across cycles. During execution write clients select a `name`, `id` and executes a sequence of OTV T_{W} instances, Listing 1.16. This transaction

effectively creates a new version of a given cycle. Concurrently read-clients execute a sequence of OTV T_{R} instances, Listing 1.17. Matching a given cycle and performing multiple reads. Both reads within an OTV T_{R} are returned, stored and collected after execution.
Anomaly check. For each OTV T_{R} result set `(firstRead,secondRead)`, the maximum `version` in the `firstRead` should be less than or equal to the minimum `version` in the `secondRead`. Otherwise, an OTV anomaly has occurred.
Why it works. OTV T_{W} installs a new version of a cycle by updating the `version` property of each `Person`. Therefore when matching a cycle once a transaction has observed some `version` it should *at least* observe this version for every remaining entity in the cycle. Unfortunately, this cannot be deduced from a single read of the cycle as results from matching cycles often does not preserve the order in which graph entities were read. This is solved by making multiple reads of the cycle. The maximum `version` of the `firstRead` determines the minimum `version` of `secondRead`. If this condition is violated then a transaction has observed the effects of a transaction in the `firstRead` then subsequently failed to observe it in the `secondRead` – the observed transaction has vanished!

```
MATCH path =
  (n:Person {id: $personId})
  -[:KNOWS*..4]->(n)
UNWIND nodes(path)[0..4] AS p
SET p.version = p.version + 1
```

Listing 1.16: OTV/FR T_{W}.

```
MATCH p1=(n1:Person {id: $personId})-[:KNOWS*..4]->(n1)
RETURN extract(p IN nodes(p1) | p.version) AS firstRead
«SLEEP($sleepTime)»
MATCH p2=(n2:Person {id: $personId})-[:KNOWS*..4]->(n2)
RETURN extract(p IN nodes(p2) | p.version) AS secondRead
```

Listing 1.17: OTV/FR T_{R}.

4.7 Fractured Read

Informally, a *Fractured Read* (FR) anomaly [5] occurs when a transaction reads *across* transaction boundaries. For example, if T_1 writes x_1 and y_1 and T_3 writes x_3. If T_2 reads x_1 and y_1, then repeats its read of x and reads x_3 a fractured read has occurred.
Definition. A transaction T_j exhibits phenomenon FR if transaction T_i writes versions x_a and y_b (in any order, where x and y may or may not be distinct items), T_j reads version x_a and version y_c, and $c < b$.
Test. Same as the OTV test.
Anomaly check. For each FR T_{R} (Listing 1.17) result set `(firstRead, secondRead)`, all `versions` across both version sets should be equal. Otherwise an FR anomaly has occurred.
Why it works. FR T_{W} installs a new version of a cycle by updating the `version` properties on each `Person`. When FR T_{R} observes a `version` every subsequent read in that cycle should read the *same* `version` as FR T_{W} (Listing 1.16) installs the same `version` for all `Person` nodes in the cycle. Thus, if it observes a different `version` it has observed the effect of a different transaction and has read across transaction boundaries.

4.8 Lost Update

Informally, a *Lost Update* (LU) anomaly [5] occurs when two transactions concurrently attempt to make conditional modifications to the same data item(s).
Definition. A history H exhibits phenomenon LU if $DSG(H)$ contains a directed cycle having one or more antidependency edges and all edges are induced by the same data item x.
Test. Load a test graph containing `Person` nodes. Assign each `Person` a unique `id` and a property `numFriends` (initialized to 0). During execution write clients execute a sequence of LU T_{W} instances, Listing 1.18. Choosing a random `Person` and incrementing its `numFriends` property. Clients store local counters (`expNumFriends`) for each `Person`, which is incremented each time a `Person` is selected *and* the LU T_{W} instance successfully commits. After the execution period the `numFriends` is retrieved for each `Person` using LU T_{R} in Listing 1.19 and `expNumFriends` are pooled from write clients for each `Person`.
Anomaly check. For each `Person` its `numFriends` property should be equal to the (global) `expNumFriends` for that `Person`.
Why it works. Clients know how many successful LU T_{W} instances were issued for a given `Person`. The observable `numFriends` should reflect this ground truth, otherwise an LU anomaly must have occurred.

```
MATCH (p:Person {id: $personId})
SET p.numFriends = p.numFriends + 1
```

Listing 1.18: Lost Update T_{W}.

```
MATCH (p:Person {id: $personId})
RETURN p.numFriends AS numFriends
```

Listing 1.19: Lost Update T_{R}.

4.9 Write Skew

Informally, *Write Skew* (WS) occurs when two transactions simultaneously attempted to make *disjoint* conditional modifications to the same data item(s). It is referred to as G2-Item in [1,11].
Definition. A history H exhibits WS if $DSG(H)$ contains a directed cycle having one or more antidependency edges.
Test. Load a test graph containing n pairs of `Person` nodes `(p1, p2)` for $k = 0, \ldots, n-1$, where the kth pair gets ids `p1.id = 2*k+1` and `p2.id = 2*k+2`, and values `p1.value = 70` and `p2.value = 80`. There is a constraint: `p1.value + p2.value > 0`. During execution write clients execute a sequence of WS T_{W} instances, Listing 1.20. Selecting a random `Person` pair and decrementing the `value` property of one `Person` provided doing so would not violate the constraint. After execution the database is scanned using WS T_{R}, Listing 1.21.
Anomaly check. For each `Person` pair the constraint should hold true otherwise a WS anomaly has occurred.
Why it works. Under no Serializable execution of WS T_{W} instances would the constraint `p1.value + p2.value > 0` be violated. Therefore, if WS T_{R} returns a violation of this constraint it is clear a WS anomaly has occurred.

```
MATCH (p1:Person {id: $person1Id}),
      (p2:Person {id: $person2Id})
«IF (p1.value+p2.value < 100)» «THEN» «ABORT» «END»
«SLEEP($sleepTime)»
pId = «pick randomly between personId1, personId2»
MATCH (p:Person {id: $pId})
SET p.value = p.value - 100
«COMMIT»
```

Listing 1.20: WS T_{W}.

```
MATCH (p1:Person),
      (p2:Person {id: p1.id+1})
WHERE p1.value + p2.value <= 0
RETURN
  p1.id AS p1id,
  p1.value AS p1value,
  p2.id AS p2id,
  p2.value AS p2value
```

Listing 1.21: WS T_{R}.

5 Results

Experiment setup. The ACID-compliance test suite was implemented in a Java application as JUnit tests with all experiments executed on Ubuntu 18.04 running AdoptOpenJDK 11.0.4.hs. All tests were conducted on 4 graph database systems and 1 relational database, consisting of: Neo4j 3.5.20 and 4.1.1, Memgraph 1.0, Dgraph 20.03.3, JanusGraph 0.5.2 (BerkeleyDB 7.5.11 and Cassandra 3.11.0 backends) and PostgreSQL 9.6. For all systems, we used their declarative query languages and the officially recommended Java drivers. For Neo4j 3.5 and Memgraph, queries were defined in Cypher and the `neo4j-java-driver` package version 1.7.0 was used. For Neo4j 4.0, we used the same queries and v4.0.1 of the driver. For the rest of the systems: `dgraph4j` driver v20.03.0 with `GraphQL+-` queries, `janusgraph-driver` v0.5.2 with Gremlin queries, and the `postgresql` driver v42.2.14 with SQL queries, were used respectively.

Analysis. The results for all tests are shown in Table 1. As can be seen here, many of the systems under test met and appeared to exceed their claimed isolation levels. Neo4j promises Read Committed but in fact seems to provide the stronger isolation level Monotonic Atomic View due to proscribing OTV [4]. Interesting, however, Neo 4.0 fails the LU test, which could not be triggered in 3.5. Thus, suggesting a possible issue introduced within the new major version.[9]
Memgraph promises Snapshot Isolation and is successful in this reguard, only failing the WS test. Similarly, Dgraph passed all tests without issue, even though it only claims Snapshot Isolation. JanusGraph with the Cassandra backend, whilst offers no isolation due to the lack of arbitrary multi-object transactions, passed a number of test where this should have caused an issue. Unfortunately, upon investigation, this appears to be due to very stale reads. In a similar vein, the BerkeleyDB implementation was successful across its varying isolation levels, however, exibited heavy lock contention with as much as 95% of the transactions aborting. PostgreSQL was successful for all of the specified isolation levels, however, had notable issues with the LU tests, aborting 60% of transactions at Repeatable Read and timing out with Serializable isolation.

[9] Neo4j can guarantee Serializable isolation to avoid these anomalies, however this requires explicit locks.

Database	C	RB	Isolation Level	G0	G1a	G1b	G1c	OTV	FR	IMP	PMP	LU	WS
Neo4j 3.5	⊗	⊗	Read Committed	⊗	⊗	⊗	⊗	⊗	↯	↯	↯	⊗	↯
Neo4j 4.1	⊗	⊗	Read Committed	⊗	⊗	⊗	⊗	⊗	↯	↯	↯	↯	↯
Memgraph	⊗	⊗	Snapshot Isolation	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	↯
Dgraph	⊗	⊗	Snapshot Isolation	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗
JG/BerkeleyDB	⊗	⊗	Read Uncommitted	⊗	⊗	⊛	⊗	↯	↯	⊗	↯	↯	⊖
JG/BerkeleyDB	⊗	⊗	Read Committed	⊖	⊗	⊛	⊖	↯	⊖	⊗	⊖	↯	⊖
JG/BerkeleyDB	⊗	⊗	Repeatable Read	⊖	⊗	⊛	⊖	⊖	⊖	⊗	⊖	⊖	⊖
JG/BerkeleyDB	⊗	⊗	Serializable	⊖	⊗	⊛	⊖	⊖	⊖	⊗	⊖	⊖	⊖
JG/Cassandra	⊗	⊗	Read Uncommitted	⊗	⊗	⊛	⊗	↯	↯	⊛	↯	↯	↯
PostgreSQL	⊗	⊗	Read Committed	⊗	⊗	⊗	⊗	⊗	⊗	⊗	↯	⊗	⊗
PostgreSQL	⊗	⊗	Repeatable Read	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊘	⊗
PostgreSQL	⊗	⊗	Serializable	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊚	⊗

Table 1: Atomicity (Atomicity-C, Atomicity-RB) and Isolation Tests, ↯ indicates anomaly occurred and ⊗ indicates it did not occur. Notation – *JG:* JanusGraph. ⊛: G1b only passes due to JanusGraph's reads being very stale. ⊖: JanusGraph passed this test, but done so by aborting $> 95\%$ of transactions. ⊘: PostgreSQL passed this test but with 60% aborts. ⊚: Execution timed out after 5 minutes.

6 Related Work

The challenge of verifying ACID-compliance has been addressed before by transactional benchmarks. For example, TPC-C [20] provides a suite of ACID tests. However, the isolation tests are reliant on lock-based concurrency control, hence are not generalizable across systems. Also, the transactional anomaly test coverage is limited to only four anomalies. The authors of [9] augment the popular YCSB framework for benchmarking transactional NewSQL systems, including a *validation phase* that detects and quantifies consistency anomalies. They permit the definition of arbitrary integrity constraints, checking they hold before and after a benchmark run. Such an approach is not possible within SNB-I due to the restrictive nature of transactional updates and the distinct lack of application-level constraints.

The Hermitage project [16] with the goal of improving understanding of weak isolation, developed a range of hand-crafted isolation tests. This test suite has much higher anomaly coverage but suffers from a problem similar to TPC-C. Test execution is performed by hand, opening multiple terminals to step through the tests.[10] The Jepsen project [14] is not a benchmark rather it addresses correctness testing, traditionally focusing on distributed systems under various failure modes. Most of Jepsen's transactional tests adopt a similar approach to us, executing a suite of transactions with hand-proven invariants. However recently, the project has spawned Elle [15] a black-box transactional anomaly checker. Elle does not rely on hand-crafted tests and can detect every anomaly in [1] (except for predicate-based anomalies) from an arbitrary transaction history.

[10] We initially experimented with Hermitage but found it difficult to induce anomalies that relied on fast timings due to some graph databases offering limited client-side control over transactions, with all statements submitted in one batch.

7 Consistency and Durability Tests

While this paper mainly focused on *atomicity* and *isolation* from the ACID properties, we provide a short overview of the other two aspects. *Durability* tests are already part of the benchmark specification [3], while adding complex *consistency* checks is left for future work.

Durability is a hard requirement for SNB-I and checking it is part of the auditing process. The durability test requires an execution of the SNB-I workload and uses the LDBC workload driver. Note, the database and the driver must be configured in the same way as would be used in the performance run. First, the database is subject to a warm-up period. Then after 2 hours of simulation execution, the database processes will be terminated, possibly by disconnecting the entire machine or by a hard process kill. Note that turning the machine off may not be possible in cloud tests. The database system is then restarted and each client issues a read for the value of the last entity (node or edge) it received a successful commit message for, that should produce a correct response.

Consistency is defined in terms of constraints: the database remains consistent under updates; i.e. no constraint is violated. Relational database systems usually support primary- and foreign-key constraints, as well as domain constraints on column values and sometimes also support simple within-row constraints. Graph database systems have a diversity of interfaces and generally do not support constraints, beyond sometimes domain and primary key constraints (in case indexes are supported). As such, we leave them out of scope for LDBC SNB. However, we do note that we anticipate that graph database systems will evolve to support constraints in the future. Beyond equivalents of the relational ones, property graph systems might introduce graph-specific constraints, such as (partial) compliance to a schema formulated on top of property graphs, rules that guide the presence of labels or structural graph constraints such as connectedness of the graph, absence of cycles, or arbitrary well-formedness contraints [18].

8 Conclusion and Future Work

In this paper, we discussed the challenges of testing ACID properties on graph databases systems and compiled a test suite of 2 atomicity and 10 isolation tests. We have implemented the proposed tests on 5 database systems, consisting of 3400 lines of code in total.Our findings show that Neo4j, Memgraph, and Dgraph satisfy their claimed isolation levels, and, in some cases, they even seem to provide stronger guarantees. We found that JanusGraph is unfit to be used in transactional workloads. PostgreSQL satisfies the selected isolation levels but in the LU tests this came at a cost of aborting the majority of the transactions (Repeatable Read) or causing a timeout (Serializable).

Looking ahead, in the short term, we will include these tests in the LDBC SNB specification [3] and use them for auditing the ACID-compliance of SUTs. In the long term, we plan to extend the tests to incorporate complex consistency constraints and add tests specifically designed for distributed databases [21].

Acknowledgements

J. Waudby was supported by the the Engineering and Physical Sciences Research Council, Centre for Doctoral Training in Cloud Computing for Big Data [grant number EP/L015358/1]. B. Steer was supported by the Engineering and Physical Sciences Research Council and Alan Turing Institute [grant number EP/T001569/1]. P. Boncz was partially supported by the SQIREL-GRAPHS NWO project. G. Szárnyas was partially supported by the MTA-BME Lendület Cyber-Physical Systems Research Group.

References

1. A. Adya. *Weak consistency: A generalized theory and optimistic implementations for distributed transactions.* Ph.D. dissertation, MIT, 1999.
2. R. Angles et al. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5):68:1–68:40, 2017.
3. R. Angles et al. The LDBC Social Network Benchmark. *CoRR*, abs/2001.02299, 2020.
4. P. Bailis et al. Highly available transactions: Virtues and limitations. *VLDB*, 2013.
5. P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Scalable atomic visibility with RAMP transactions. *ACM Trans. Database Syst.*, 2016.
6. O. Batarfi et al. Large scale graph processing systems: Survey and an experimental evaluation. *Cluster Computing*, 18(3):1189–1213, 2015.
7. M. Besta et al. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *CoRR*, abs/1910.09017, 2019.
8. M. Besta et al. Practice of streaming and dynamic graphs: Concepts, models, systems, and parallelism. *CoRR*, abs/1912.12740, 2019.
9. A. Dey, A. Fekete, R. Nambiar, and U. Röhm. YCSB+T: Benchmarking web-scale transactional databases. In *ICDE*, pages 223–230. IEEE Computer Society, 2014.
10. O. Erling et al. The LDBC Social Network Benchmark: Interactive workload. In *SIGMOD*, pages 619–630. ACM, 2015.
11. A. Fekete, D. Liarokapis, E. J. O'Neil, P. E. O'Neil, and D. E. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.
12. N. Francis et al. Cypher: An evolving query language for property graphs. In *SIGMOD*, pages 1433–1445. ACM, 2018.
13. J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. pages 365–394, 1976.
14. K. Kingsbury. Jepsen analyses, 2020. `http://jepsen.io/analyses`.
15. K. Kingsbury and P. Alvaro. Elle: Inferring isolation anomalies from experimental observations. *CoRR*, abs/2003.10554, 2020.
16. M. Kleppmann. Hermitage: Testing transaction isolation levels, 2020. `https://github.com/ept/hermitage`.
17. A. Pavlo and M. Aslett. What's really new with NewSQL? *SIGMOD Rec.*, 2016.
18. O. Semeráth et al. Formal validation of domain-specific languages with derived features and well-formedness constraints. *Softw. Syst. Model.*, 16(2):357–392, 2017.
19. M. Stonebraker et al. The end of an architectural era (It's time for a complete rewrite). In *VLDB*, pages 1150–1160. ACM, 2007.
20. TPC. TPC Benchmark C, revision 5.11. Technical report, 2010. `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf`.
21. J. Waudby, P. Ezhilchelvan, J. Webber, and I. Mitrani. Preserving reciprocal consistency in distributed graph databases. In *PaPoC at EuroSys*. ACM, 2020.