Testing Distributed Database Isolation through Anti-Pattern Detection

Jingxuan Qiu

Testing Distributed Database Isolation through Anti-Pattern Detection

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jingxuan Qiu
born in Heilongjiang, China

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

© 2023 Jingxuan Qiu.

Testing Distributed Database Isolation through Anti-Pattern Detection

Author: Jingxuan Qiu
Student id: 4785770
Email: `J.Qiu-2@student.tudelft.nl`

Abstract

Distributed databases often struggle to fulfill their transactional isolation guarantees due to sharding and replication. As a result, the problem of checking isolation levels is consistently receiving attention from academia and industries. Transactional dependency graphs form a useful abstraction to analyze the transactions' dependencies and check for isolation anomalies using graph-based anti-patterns. Meanwhile, graph databases, known for their efficiency and convenience in graph representations and analytics, become promising for implementing isolation level checkers. In this work, we present a novel isolation level checker in the distributed graph database, ArangoDB. We collect execution histories from ArangoDB, operating in both single-machine and cluster modes. Also, we transform the execution histories to a dependency graph in another ArangoDB server. We then utilize customized AQL queries to detect anti-patterns on the graph. Our evaluation demonstrates the effectiveness and scalability of our checker, as well as its efficiency compared to existing isolation checkers. Also, we have found three underlying factors that are significantly correlated with the runtime of the checker: *history length* (the number of committed transactions), *density* (the density of the dependency graph), and *contributing traversals* (the number of traversals spent on cycles). The thesis artifact is online at https://github.com/jasonqiu98/GRAIL-artifact/tree/thesis.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. Burcu Külahçıoğlu Özkan, Faculty EEMCS, TU Delft
External supervisor:	Dr. Stefania Dumbrava, ENSIIE
Committee Member:	Dr. Asterios Katsifodimos, Faculty EEMCS, TU Delft

Preface

During my three years of master's study, many events unfolded: public health crises, wars, systemic economic downturns, and inflation, among others. At the same time, I had the fortune of meeting my supervisors and research team, delving into knowledge, and completing the design and experiments for this thesis in the serene and picturesque city of Delft.

First and foremost, I would like to express my heartfelt gratitude to my two daily supervisors, Burcu Ozkan and Stefania Dumbrava, for their tireless guidance. They provided continuous assistance and feedback in shaping the conceptualization of my thesis, supplementing my knowledge, discussing and refining the details, and guiding me through the process of submission. Throughout this comprehensive training period, I have made significant progress in my academic pursuits, gaining the ability to comprehend and refine research plans. Additionally, I would like to thank a fellow master's student, Zhao Jin in ENSIIE, who provided assistance in conducting experiments and creating figures and plots.

Furthermore, I am grateful to my parents for their financial and emotional support. Due to the circumstances of the pandemic, we have been unable to meet face-to-face for three years. I am thankful for their patience, support, and understanding throughout this journey.

I would also like to express my gratitude to every friend who has been backing me up during this thesis period. Whether it was providing relief amidst the pressures, engaging in discussions and envisioning plans for the future, or offering feedback and assistance in specific issues, I sincerely appreciate their help.

As I approach the conclusion of this thesis, I would like to once again appreciate everyone who has provided me with assistance. I am grateful to the university for providing me with the opportunity to study at TU Delft. Here I made the past three years meaningful, and deepened my understanding of computer science, artificial intelligence, and software engineering. This also equipped me with advanced skills to embark on my future life as a software engineer.

Thank you all.

Jingxuan Qiu
Delft, the Netherlands
June 15, 2023

Contents

List of Figures

Chapter 1

Introduction

In this chapter, we provide the necessary terminologies that are commonly referred to throughout the thesis, and we present our motivation with an illustrative example with a schema from the LDBC SNB Benchmark Dataset. We also propose a way to check isolation levels with graph database queries. After that, we propose our research questions and give an overview of our research methodology to analyze the effectiveness, scalability, and efficiency of our novel checker. Finally, we list our contributions and present the outline of the remaining chapters in the thesis.

1.1 Terminologies

Isolation levels of (distributed) databases. Database transactions often group and encapsulate a series of events that retrieve or modify the variables stored in the database, and allow for preserving data integrity even in cases of system failure. Isolation, being part of the ACID properties of a transaction, describes the degree to which it executes independently of and is not interfered with by other transactions on the data or schema shared by concurrent transactions [8]. The strongest isolation level, *serializability*, expects every transaction to execute as if it is the only transaction running in the whole system. In this case, one order of transactions can be found so that the return values in the transactions remain the same even when the transactions execute serially (i.e., one after one). This also implies that the concurrency amongst the transactions does not seemingly exist. However, serializability requires more computational overhead in concurrency control [35]. Therefore, weaker isolation levels have been explored and applied to plenty of commercial database systems over the past decades.

A strong isolation level is more demanding in present-day distributed database systems with sharding and replication, as it requires synchronization and distributed consistency, which further entails longer latency and reduces efficiency [11]. Also, a weak isolation level is expected to satisfy the needs of the system in some cases. For instance, [58] introduces parallel snapshot isolation (PSI), which relaxes the requirements of a total order of the transactions from snapshot isolation (SI), and makes the level more applicable in a distributed setting.

Isolation level hierarchy – The preventative approach. The database system community has been proposing definitions of isolation level hierarchy since it was first introduced in [35]. Then, the ANSI/ISO SQL-92 specifications [8] incorporated this concept as a standard and categorized database isolation into four levels. For each level, it proposed an *anomaly* pattern, which is a sequence of events that should not be observed in (*disallowed by*) an execution. However, the standard was challenged by the community [19] for it highly relied on the locking mechanism of the database and failed to cover the counterpart where optimistic concurrency control (OCC) or multi-version concurrency control (MVCC) was present (e.g., in Gemstone [47]). In other words, the *preventative approach* adopted by the ANSI standard failed to achieve implementation-independence across all database systems.

Isolation level hierarchy – Graph-based specifications. To address the weaknesses of the locking mechanism, Adya et al. extended the ANSI standard by formulating isolation levels based on *dependency graphs* [13]. A dependency graph maintains the dependency edges between transaction vertices and also reshapes the preventative anomalies from sequential orderings to graph patterns, usually containing a certain type of cycle. As both construction and interpretation of dependency graphs are not relevant to the locking mechanism, Adya's formalism addressed the drawback of the preventative approach and presented isolation level hierarchy on a different path of *graph-based specifications*. Later, in a more formal way, Cerone et al. proposed an axiomatic framework to define isolation levels and anomalies on an abstract level [24] and proved its equivalence to Adya's definitions [25]. Overall, Cerone et al.'s formalism abstracted Adya's graph visualization to algebraic definitions, which are rigorous, uniform, and declarative.

Isolation level checking. Based on the presence of the isolation level hierarchy, many databases claim they achieve a certain isolation level by default or provide a range of isolation levels for users' choices. For example, MySQL 8.0 [10] claims its default isolation level as repeatable read while allowing other isolation levels. However, many violations have demonstrated that databases often fail to achieve the isolation levels as they claim [44]. Therefore, *isolation level checking* is important, which is the process of determining whether a database has achieved the claimed isolation level.

Graph databases. Highly interconnected data has emerged and now covered a variety of areas, ranging from network infrastructures, social networks, logistics, and life science repositories. The pervasive usage of graph data has led to an increasing need for efficient and convenient storage and processing. Graph databases, as a special type of NoSQL system, leverage native graph models to tackle this challenge. Among the types of native graph models, *property graph model* [14] is the most prominent and expressive one. It is a multi-labeled multi-graph, whose nodes and edges can be additionally enriched by sets of key-value properties. Currently, the ecosystem of commercial graph databases features dozens of systems, such as Neo4j [5], ArangoDB [2], TigerGraph [29], OrientDB [51], etc.

Graph queries. As a standard graph query language is still under development, many graph databases support query languages to extract patterns from graph database storage. Each graph database system has a different type of graph query language [22, 21], which may have a different level of expressiveness and coverage of first-order logic from other graph databases. A leading example is Neo4j's Cypher language, which has a broad coverage of use cases with built-in support and open-source contributions. Also, it is supported and extended in numerous other databases (e.g., Amazon Neptune [18] and Memgraph [4]).

Schema from LDBC SNB Benchmark Dataset. Furthermore, we introduce a database schema below, to illustrate the execution histories with their real-life scenarios throughout the thesis. Figure 1.1 shows part of the graph schema from the dataset of LDBC Social Network Benchmark (LDBC SNB, the full schema in Figure 2 of [59]). A many-to-many relation is represented by a thick edge. The details of vertex and edge collections are as follows.

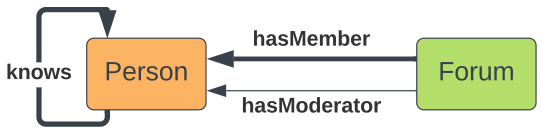


Figure 1.1: Partial schema from LDBC SNB dataset

- vertices
  1. `Person`: (`id` → int, `firstName` → string, `lastName` → string)
  2. `Forum`: (`id` → int, `title` → string)
- edges
  1. **`knows`**: (`id` → int, `fromPersonId` → int, `toPersonId` → int)
  2. **`hasMember`**: (`id` → int, `fromForumId` → int, `toPersonId` → int)
  3. `hasModerator`: (`id` → int, `fromForumId` → int, `toPersonId` → int)

In this partial schema, a group of persons communicate through forums. Each forum may have my persons as members, but only one person as the moderator. The moderator administers the forum and has permission to modify the title of the forum and add new members to the list. Persons may know each other. Furthermore, we list the operations on the following variables supported by this schema below. We use the partial schema to illustrate with examples in the rest of the thesis.

- title of a forum: view (`read`); update (`write`)
- members of a forum: view all members (`read`); add a new member (`append`)
- persons known by a person: view all persons (`read`); know a new person (`append`)

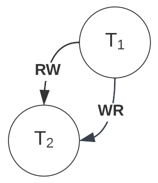
Anti-Patterns and Anomalies. Graph-based specifications translate each anomaly disallowed by an isolation level to subgraphs with specific cyclic structures. Therefore, isolation level detection is an example of *anti-pattern detection*. As such, the burden of isolation level checking is transferred to analyzing the underlying graph structure, which is manageable with a rich ecosystem of graph algorithms and infrastructures. This approach also saves the labor of handcrafting in the rule-based preventative approach.

From this section onward, the term *anti-pattern* will be used in place of interchangeable terms (e.g., *violation* or *phenomenon*) to highlight the task of anti-pattern detection in this thesis. We select five isolation levels with clear graph characteristics [24, 23, 13] and investigate the *anti-patterns* against them in the rest of the thesis. In addition, we also introduce a series of *anomalies*, where the term is used to denote the special cases when an anti-pattern happens against a certain isolation level.

1.2 Motivation

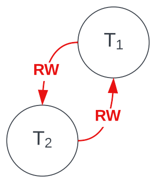
Graph-based specifications have presented an implementation-independent definition of isolation levels. We illustrate the usage of graph-based specifications and dependency graphs with the two transactional executions in Figure 1.2.

T_1:
```
  read(forum1.title, "A1")
  write(forum2.title, "B2")
```
T_2:
```
  read(forum2.title, "B2")
  write(forum1.title, "A2")
```



(a) A serializable execution

T_1:
```
  read(forum1.title, "A1")
  write(forum2.title, "B2")
```
T_2:
```
  read(forum2.title, "B1")
  write(forum1.title, "A2")
```



(b) Write skew

Figure 1.2: Example executions: serializability and write skew

Figure 1.2a shows a serializable execution that involves two transactions T_1 and T_2 on two forum instances `forum1` and `forum2`. We assume the initial titles are `"A1"` and `"B1"` for `forum1` and `forum2`, respectively. The interpretation of the execution is as follows: the moderator of `forum2` browses the title of `forum1` and finds `"A1"`; after that, the moderator updates the title of `forum2` to `"B2"`. Concurrently, the moderator of `forum1` views the updated title of `forum2` as `"B2"` and then decides to update the title of `forum1` as well. By observing the title of `forum1`, we draw an RW edge from T_1 to T_2 since T_2 writes a new title which replaces the old value in T_1. By observing the title of `forum2`, we draw a WR edge from T_1 to T_2 since T_2 reads the new title updated by T_1. In this case, no cycle is formed and no anti-pattern is found. In fact, the execution shows as if T_2 is committed after T_1 has taken effect on the database, which makes it serializable.

On the other hand, the example execution in Figure 1.2b is not serializable. The interpretation of the execution is similar to that in Figure 1.2a, except in T_2, where the moderator of `forum1` reads the title of `forum2` as the initial value `"B1"`. In this case, the RW edge formed based on `forum1` remained unchanged; however, an RW edge is formed based on `forum2`, since now T_2 has read an earlier version of `forum2`'s title which is further updated by `forum1`. This dependency graph contains a cycle, which is an anti-pattern that violates serializability. In fact, both transactions write to shared variables without observing each other's updates, which forms a *write-skew* anomaly.

In addition to the graph-based explanations, the traditional way of serializability checking is also effective in getting the correct results but suffers combinatorial explosions in its complexities. For example, in Figure 1.2b, by assuming T_1 takes effect before T_2, we can spot that T_2 reads the title of `forum2` as its initial value `"B1"`, which is, however, supposed to be `"B2"` following the update by T_1. Conversely, if we assume T_2 takes effect before T_1, then T_1 reads the title of `forum1` as its initial value `"A1"` instead of the value updated by T_2. In both cases, a serial order cannot be constructed between T_1 and T_2, which makes an anti-pattern against serializability. This approach still finds the anti-pattern but involves an enumeration of all possible orders in which transactions take effect. The enumeration further leads to a combinatorial explosion, mainly when the execution history contains a large number of transactions.

The two dependency graphs shown in Figure 1.2, instead, make the isolation level checking tractable. Assume we know that the numeric suffixes of the forum titles are always increasing. Then we can construct the dependency graph within polynomial time. Furthermore, anti-pattern detection is usually a cycle detection procedure that can also be finished within polynomial time. This example demonstrates that graph-based specifications have replaced the traditional combinatorial enumeration with a deterministic dependency graph, and reduced the complexity of isolation level checking.

Following the above example, graph-based specifications have converted isolation checking to an anti-pattern detection problem to check cycles within dependency graphs. This has enriched the isolation checking with a variety of independent implementations. For example, Elle [42] applies graph-based pattern matching to dependency graphs created from execution histories to check isolation levels. In addition, several serializability [57, 60, 20, 26] and snapshot isolation [20, 38, 69] checkers also exploit the functionality of polygraphs, which are sets of possible dependency graphs speculated based on the information contained in the execution histories. Both types of checkers utilize the graph properties to reduce the time complexity to polynomial [20] or linear [42, 38] level under certain conditions.

However, to the best of our knowledge, none of the previous work has applied graph databases to isolation level checking. Popular graph databases inherently support and optimize the toolkit for graph analytics. Also, databases can utilize both main and secondary memory, which allows for handling large-scale dependency graphs. Both of these factors make graph databases a feasible solution to checking isolation levels through graph-based anti-pattern detection.

For this reason, we select ArangoDB as a candidate graph database along with its query language (AQL) to perform the isolation checking through anti-pattern detection. In addition, we can collect histories from ArangoDB as a distributed database. In this way, our checker combines the usage and provides an in-database implementation with history collection and isolation detection, both in the same database ArangoDB.

1.3 Research Questions

To verify the feasibility and observe the performance of the novel isolation level checker developed on the graph database ArangoDB (*graph-based checker* hereafter), we address the following three research questions.

RQ1

Is the graph-based checker effective in detecting the isolation level of a distributed database?

RQ2

Does the graph-based checker scale in the increasing workload?

RQ3

How does the graph-based checker perform compared to the state-of-the-art isolation checkers?

1.4 Research Methodology

We select ArangoDB as both the distributed database under test and the graph database to construct the novel, graph-based checker. We run Jepsen to collect nine sets of histories from ArangoDB, five with sharding and replication, and the remaining four without.

To address RQ1, we compare our sound implementation to a state-of-the-art checker, Elle. We count the number of anomalous histories detected by both checkers and determine whether the results show a mismatch. In this way, we confirm the effectiveness of our graph-based checker.

To address RQ2, we explore a variety of factors that may affect the scalability of the graph-based checker, and use performance plots and statistical methods to establish the correlations. Through performance plots, we analyze the effect of two main factors, collection time and transaction generation rate, of our Jepsen history collector. Also, we find three underlying factors, *history length* (the number of committed transactions), *density* (the density of the dependency graph), and *contributing traversals* (the number of traversals spent on cycles). We also investigate these factors and their correlation with the execution time of our checker by linear regression. Finally, we consider two additional factors, the number of sessions and the maximum number of write operations per object, and observe their effect on the scalability through performance plots. We also present additional evaluation related to checker-specific settings and scenarios of fault injection. Our evaluation is also strengthened by exploratory analysis and statistical approaches.

To address RQ3, we investigate our checker's performance compared with two state-of-the-art checkers, Elle and PolySI. We use performance plots to illustrate the relative performance of these checkers. In addition to the state-of-the-art checkers, we also discuss the possibility to use an alternative graph database to construct the checker. We confirm its feasibility and potential for improvement through analysis with performance plots.

1.5 Contributions

The thesis has the following contributions.

- Designing and implementing a novel isolation level checker with a graph database;
- Demonstrating the effectiveness of using graph database queries to detect isolation levels of distributed databases;
- Providing an in-database implementation where the history collection and isolation detection happen on the same graph database;
- Performing extensive evaluation on the efficiency and scalability of our graph-based checker as well as comparison to state-of-the-art checkers;
- Proposing a smart visualization method to demonstrate the isolation level checking results in graph databases.

1.6 Thesis Outline

The outline of the rest of the thesis is given as follows.

- Chapter 2 provides necessary background of ArangoDB and isolation levels. It also discusses the feasibility of using graph databases to detect isolation levels.
- Chapter 3 presents the history collection and dependency graph construction in our graph-based checker. It contains history collection by Jepsen and dependency graph construction from events to transactions.
- Chapter 4 demonstrates the core part of the thesis. It explains how to use graph queries to check cycles for anti-pattern detection. In addition, we shed light on the way of visualizing the output. We also mention the isolation detection result on ArangoDB histories.
- Chapter 5 includes extensive evaluation of the effectiveness and scalability of our checker. We also give a comparison with the state-of-the-art checkers and alternative implementations with a different graph database, Neo4j.
- Chapter 6 discusses related work about isolation level histories, graph database trends, and state-of-the-art isolation level checkers.
- Chapter 7 concludes the thesis by providing the conclusions of results, the contributions, and directions to future work.

Chapter 2

Preliminaries

In this chapter, we start by introducing isolation levels with their formal definitions, based on the graph-based specifications proposed in Adya's and Cerone et al.'s work. Following that, we present varied examples to illustrate the usage of isolation levels, anti-patterns, and anomalies. We show that isolation checking is usually a cycle detection process. Next to the examples, we introduce our major tool of the novel checker, ArangoDB. ArangoDB is utilized as both a distributed database and graph database, which plays its part in history collection and cycle detection, and makes an in-database implementation. Then, we present a general overview of cycle detection algorithms. After that, we present the problem statement of isolation level checking and make relevant assumptions. Finally, we explore the feasibility of using graph databases and queries for the anti-pattern detection of the novel isolation checker, and summarize that ArangoDB is an ideal candidate for our research.

2.1 Isolation Levels: Formal Definitions

In this section, we explore the formal definitions of isolation levels and their anti-patterns in the context of dependency graphs. We first start this section with an introduction of the concept of relations based on [39] to simplify the notations that follow. Later, we explore the graph-based specifications of transactional executions and dependency graphs based on both *Adya's formalism* [13] and *Cerone et al.'s formalism* [24].

2.1.1 Relations

In this subsection, a short introduction to relations and their properties based on [39] will be given as a prerequisite for further concepts. A *binary relation* can be viewed as a collection of ordered pairs that associate elements of one set with elements of another set. The formal definitions regarding binary relations are listed as follows.

- An ***ordered pair*** (u,v) associates two elements u and v and satisfies the property $(u,v) = (x,y) \Leftrightarrow u = x \land v = y$.
- The ***Cartesian product*** of sets U,V is $U \times V = \{(u,v) \mid u \in U,\ v \in V\}$.

- A ***binary relation*** R from U to V is a subset of $U \times V$, i.e., $R \subseteq U \times V$. For an ordered pair (u,v), where $u \in U$ and $v \in V$, the notation $(u,v) \in R$ is interchangeable with $u \xrightarrow{R} v$.

It is a special case when the two sets associated by a binary relation R are the same set. Such a type of relation is called *homogeneous* binary relations. From now on, all *relations* are homogeneous unless specified otherwise.

Definition 1 *A* ***relation*** *R is a homogeneous binary relation* ***on*** *a set S, i.e., $R \subseteq S \times S$.*

In addition to the definition, a set of operations on relations are given below.

- The ***composition*** of two relations R_1 and R_2 on S is $R_1 \; ; \; R_2 = \{(u,v) \mid \exists x \in S.\ u \xrightarrow{R_1} x \xrightarrow{R_2} v.\}$.
- The ***converse*** of a relation R is $R^{-1} = \{(v,u) \mid (u,v) \in R\}$, i.e., $u \xrightarrow{R} v \Leftrightarrow v \xrightarrow{R^{-1}} u$.
- The ***identity relation*** on S is $I_S \triangleq \{(u,u) \mid u \in S\}$.
- The ***image set*** of a relation R for an element $u \in S$ is $R(u) = \{v \mid v \in S \wedge u \xrightarrow{R} v\}$. Similarly, for an element $v \in S$, $R^{-1}(v) = \{u \mid u \in S \wedge u \xrightarrow{R} v\}$.

Furthermore, define $R^0 = I_S$ and $R^n = R \, ; R^{n-1}$ for $n \geq 1$. Then, the following relations are defined.

- The ***transitive closure*** of R is $R^+ = \bigcup_{n=1}^{\infty} R^n$.
- The ***reflexive closure*** of R is $R? = I_S \cup R = \bigcup_{n=0}^{1} R^n$.
- The ***reflexive-transitive closure*** of R is $R^* = I_S \cup R^+ = \bigcup_{n=0}^{\infty} R^n$.

Several properties are further introduced for a relation on S.

- R is ***reflexive***: $\forall u \in S.\ u \xrightarrow{R} u$. Alternatively, $I_S \subseteq R$.
- R is ***irreflexive***: $\forall u \in S.\ \neg(u \xrightarrow{R} u)$. Alternatively, $I_S \cap R = \emptyset$.
- R is ***transitive***: $\forall u,v,w \in S.\ u \xrightarrow{R} v \wedge v \xrightarrow{R} w \Rightarrow u \xrightarrow{R} w$. Alternatively, $R \; ; \; R \subseteq R$.
- R is ***acyclic***: $\forall u \in S.\ \neg(u \xrightarrow{R} \ldots \xrightarrow{R} u)$. Alternatively, $I_S \cap R^+ = \emptyset$.

Additional definitions related to sets are also listed below.

- The ***power set*** of a set S is the set of all subsets of S, including the empty set and S itself. It is denoted by $2^S = \{X \mid X \subseteq S\}$.

2.1.2 Transactional Executions

In this subsection, terminologies relating to *transactional executions* are discussed and their formal definitions are listed accordingly. To start off, the concept of order is introduced.

- A relation R is a ***strict partial order*** on S iff R is irreflexive and transitive on S.
- A relation R is a ***total order*** on S iff the following conditions are satisfied.

1. R is a strict partial on S.
2. R associates any pair of distinct elements in S one way or another: $\forall a,b \in S.\ a \neq b \Rightarrow a \xrightarrow{R} b \lor b \xrightarrow{R} a$.

Furthermore, objects, events, transactions, and histories are defined as follows.

- An ***object*** is a key-value pair stored in a database.
  - The ***key*** of an object is an immutable string assigned by the system when the object is created. The set of object keys is Key.
  - The ***value*** of an object has a fixed data type (e.g., integer, string, array, set, etc.) and is mutable. The set of possible object values is Val.
- An ***event*** is the invocation of a read or write event on an object. The set of possible events is $\mathsf{Event} = \{\mathtt{read}(k,v),\ \mathtt{write}(k,v) \mid k \in \mathsf{Key},\ v \in \mathsf{Val}\}$. An event $\mathtt{f}(k,v)$ has three parts: $\mathtt{f}$ is the function, k is the key, and v is the value. Any of the three parts can be marked by an underscore _ if irrelevant or unknown.
- A ***transaction*** T is a pair (E, po), where $E \subseteq \mathsf{Event}$ is a finite, non-empty set of events and the ***program order*** $\mathsf{po} \subseteq E \times E$ is a total order on E.
  - $T \vdash \mathtt{write}(k,v)$ if T writes to k and the last value written is v, i.e., $\max_{\mathsf{po}}\{e \mid e = \mathtt{write}(k,\_)\} = \mathtt{write}(k,v)$.
  - $T \vdash \mathtt{read}(k,v)$ if T reads from k before writing to it and v is the value returned by the first such read, i.e., $\min_{\mathsf{po}}\{e \mid e = \_(k,\_)\} = \mathtt{read}(k,v)$.
  - $\mathsf{Writes}_k \triangleq \{T \mid T \vdash \mathtt{write}(k,\_)\}$ is the set of transactions that write to k.
  - $\mathsf{Reads}_k \triangleq \{T \mid T \vdash \mathtt{read}(k,\_)\}$ is the set of transactions that read from k.
- A ***history*** $\mathcal{H} \triangleq \mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ is a finite set of transactions with disjoint sets of events.

To simplify the notations of formal definitions for dependency graphs and isolation levels, the concept of execution is further introduced.

- An ***execution*** is a tuple $\mathcal{X} = (\mathcal{H}, \mathsf{VIS}, \mathsf{CO})$, where $\mathcal{H}$ is a history and the ***visibility*** and ***commit orders*** $\mathsf{VIS}, \mathsf{CO} \subseteq \mathcal{T} \times \mathcal{T}$ are such that $\mathsf{VIS} \subseteq \mathsf{CO}$ and CO is total.
  - visibility: $T \xrightarrow{\mathsf{VIS}} S$ means that the writes done by the transaction T have taken effect on the transaction S.
  - commit order: $T \xrightarrow{\mathsf{CO}} S$ means that T commits earlier than S.

In addition, two axioms related to executions are highlighted here.

- The ***internal consistency axiom*** INT ensures that within a transaction, a read event e on an object with the key x returns the same value as the last write to or a read from this object preceding the event e if such a last event exists. Formally, $\forall (E, \mathsf{po}) \in T.\ \forall e \in E.\ \forall k \in \mathsf{Key}.\ \forall v \in \mathsf{Val}.\ e = \mathsf{read}(k,v)\ \land\ \{p \mid p = \_(k,\_)\ \land\ p \xrightarrow{\mathsf{po}} e\} \neq \emptyset \Rightarrow \max_{\mathsf{po}}\{p \mid p = \_(k,\_)\ \land\ p \xrightarrow{\mathsf{po}} e\} = \_(k,v)$.

- If a read event e on an object with the key x is not preceded by another read or write event on this object within the same transaction T, then the internal consistency axiom does not apply. In this case, the ***external consistency axiom*** EXT ensures that a read event e on an object with the key x, in a transaction T, returns the same value as written by another transaction that includes the last committed write event on this object among the transactions visible by T. Formally, $\forall T \in \mathcal{T}.\ \forall k \in \mathsf{Key}.\ \forall v \in \mathsf{Val}.\ T \vdash \mathsf{read}(k,v) \Rightarrow \max_{\mathsf{CO}}(\mathsf{VIS}^{-1}(T) \cap \mathsf{Writes}_x) \vdash \mathsf{write}(k,v)$.

2.1.3 Graph-Based Anti-Patterns

A dependency graph is a graph with transactions as vertices, and dependencies between transactions as edges. The *transaction vertices* are connected by three types of *dependency edges*: *read dependency* (WR), *write dependency* (WW) and *anti-dependency* (RW). A finite sequence of dependency edges of the same or different types forms a *dependency path*. Both formal and informal definitions of these three relations are listed below.

- ***read dependency***: $T \xrightarrow{\mathsf{WR}(k)} S \Leftrightarrow S \vdash \mathsf{read}(k,\_) \wedge T = \max_{\mathsf{CO}}(\mathsf{VIS}^{-1}(S) \cap \mathsf{Writes}_k)$. Informally, S reads T's write to the object with the key k.
- ***write dependency***: $T \xrightarrow{\mathsf{WW}(k)} S \Leftrightarrow T \xrightarrow{\mathsf{CO}} S \wedge T, S \in \mathsf{Writes}_k$. Informally, S overwrites T's write to the object with the key k.
- ***anti-dependency***: $T \xrightarrow{\mathsf{RW}(k)} S \Leftrightarrow T \neq S \wedge \exists T' \in \mathcal{T}.\ T' \xrightarrow{\mathsf{WR}(k)} T \cap T' \xrightarrow{\mathsf{WW}(k)} S$. Informally, S overwrites the object with the key k read by T.

Based on all the previous notations and definitions, the formal definition of the dependency graph is given below.

Definition 2 *A* ***dependency graph*** *is a tuple $\mathcal{G} = (\mathcal{H}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$, where $\mathcal{H} = \mathcal{T}$ is a history and*

1. WR*:* $\mathsf{Key} \mapsto 2^{\mathcal{T}\times\mathcal{T}}$ *is such that:*
   - $\forall T, S \in \mathcal{T}.\ \forall k \in \mathsf{Key}.\ T \xrightarrow{\mathsf{WR}(k)} S \Rightarrow \exists v \in \mathsf{Val}.\ T \neq S \wedge T \vdash \mathsf{write}(k,v) \wedge S \vdash \mathsf{read}(k,v)$.
   - $\forall S \in \mathcal{T}.\ \forall k \in \mathsf{Key}.\ S \vdash \mathsf{read}(k,\_) \Rightarrow \exists T \in \mathcal{T}.\ T \xrightarrow{\mathsf{WR}(k)} S$.
   - $\forall T, T', S \in \mathcal{T}. \forall k \in \mathsf{Key}.(T \xrightarrow{\mathsf{WR}(k)} S \wedge T' \xrightarrow{\mathsf{WR}(k)} S) \Rightarrow T = T'$.
2. WW*:* $\mathsf{Key} \mapsto 2^{\mathcal{T}\times\mathcal{T}}$ *is such that for every* $k \in \mathsf{Key}$*,* $\mathsf{WW}(k)$ *is a total order on the set* Writes_k*.*
3. RW*:* $\mathsf{Key} \mapsto 2^{\mathcal{T}\times\mathcal{T}}$ *is such that* $\forall T, S \in \mathcal{T}.\ \forall k \in \mathsf{Key}.\ T \xrightarrow{\mathsf{RW}(k)} S \Leftrightarrow T \neq S \wedge \exists T' \in \mathcal{T}.\ T' \xrightarrow{\mathsf{WR}(k)} T \cap T' \xrightarrow{\mathsf{WW}(k)} S$*.*

Each isolation level is specified by a set of executions and, equivalently, by a specific pattern in a set of dependency graphs. This way of defining an isolation level is called *(dependency)* *graph-based specifications*. Following Cerone's work (Theorem 11 of [25]) [23], three isolation levels (serializability, snapshot isolation, and parallel snapshot isolation) are selected and presented in Theorem 1.

Theorem 1 *Assuming that the internal and external consistency axioms (*INT *and* EXT*) hold for an execution $\mathcal{X} = (\mathcal{H}, \mathsf{VIS}, \mathsf{CO})$ with a dependency graph $\mathcal{G} = (\mathcal{H}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$, where $\mathcal{H} = \mathcal{T}$ is a history, the dependency graph-based specifications of three isolation levels are given as follows.*

1. *An execution $\mathcal{X}$ is **serializable** (SER) iff its dependency graph $\mathcal{G}$ does not contain any cycle, that is, the relation $\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}$ is acyclic on $\mathcal{T}$.*
2. *An execution $\mathcal{X}$ is allowed by **snapshot isolation** (SI) iff its dependency graph $\mathcal{G}$ admits only cycles with at least two consecutive anti-dependency edges, that is, the relation $(\mathsf{WR} \cup \mathsf{WW})\ ;\ \mathsf{RW}?$ is acyclic on $\mathcal{T}$.*
3. *An execution $\mathcal{X}$ is allowed by **parallel snapshot isolation** (PSI) iff its dependency graph $\mathcal{G}$ contains only cycles with at least two anti-dependency edges, that is, the relation $(\mathsf{WR} \cup \mathsf{WW})^{+}\ ;\ \mathsf{RW}?$ is irreflexive on $\mathcal{T}$.*

Following Theorem 1, a corresponding set of anti-patterns can be easily reached by negation, which converts acyclic constraints to cyclic anti-patterns.

Proposition 1 *A **dependency graph-based anti-pattern** is a dependency path such that the transaction vertices of the path form a relation that does not satisfy the dependency graph-based specifications of an isolation level. Specifically, the anti-patterns of isolation levels specified in 1 are given below.*

1. *SER: $\mathcal{G}$ contains a cycle.*
2. *SI: $\mathcal{G}$ contains a cycle that has zero or one consecutive anti-dependency edge.*
3. *PSI: $\mathcal{G}$ contains a cycle that has zero or one anti-dependency edge.*

In comparison with Cerone et al.'s formalism, the earlier definitions by Adya also present a series of isolation levels. The major three of them are PL-3, PL-2, and PL-1, which are highlighted below in Theorem 2.

Theorem 2 *Assuming that the internal and external consistency axioms (*INT *and* EXT*) hold for an execution $\mathcal{X} = (\mathcal{H}, \mathsf{VIS}, \mathsf{CO})$ with a dependency graph $\mathcal{G} = (\mathcal{H}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$, where $\mathcal{H} = \mathcal{T}$ is a history, the dependency graph-based specifications of additional three isolation levels are given as follows.*

1. *An execution $\mathcal{X}$ is allowed by **PL-3** level iff its dependency graph $\mathcal{G}$ does not contain any cycle with or without anti-dependency edges, that is, the relation $\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}$ is acyclic on $\mathcal{T}$.*
2. *An execution $\mathcal{X}$ is allowed by **PL-2** level iff its dependency graph $\mathcal{G}$ does not contain any cycle without anti-dependency edges, that is, the relation $\mathsf{WR} \cup \mathsf{WW}$ is acyclic on $\mathcal{T}$.*

3. *An execution $\mathcal{X}$ is allowed by **PL-1** level iff its dependency graph $\mathcal{G}$ does not contain any cycle with only write dependency edges, that is, the relation* WW *is acyclic on $\mathcal{T}$. The definition remains valid when the external consistency axiom* EXT *does not hold.*

Adya's PL-3 level is in fact identical to SER because both of them are defined on the same acyclic condition. Therefore, only two additional levels PL-2 and PL-1 are included in Proposition 2 below.

Proposition 2 *The anti-patterns of isolation levels specified in 2 are given below.*

1. *PL-2: $\mathcal{G}$ contains a circular information flow, which is a cycle that has no anti-dependency edge.*
2. *PL-1: $\mathcal{G}$ contains a write cycle that has only write-dependency edges.*

2.2 Examples of Execution Histories and Anti-Patterns

Propositions 1 and 2 delineate five isolation levels by their graph-based anti-pattern correspondingly. We illustrate with eight example histories to further specify what may happen when these anti-patterns occur in practice. We attach a dependency graph to each history with the graph-based anti-pattern marked red. For all examples, we assume that $\forall\, i \in \mathbb{N}_+ \wedge i < N.\ T_i \xrightarrow{\mathsf{CO}} T_{i+1}$, where N is the total number of transactions within a history. In other words, we present all the transactions in their commit order, while the visibility order is unknown.

H1) Figure 1.2a shows two transactions that can be serialized into $T_1 \xrightarrow{\mathsf{VIS}} T_2$ and no cyclic anti-pattern is detected. Therefore, this history satisfies SER. In some cases, a serializable execution has some other serials that may induce an anti-pattern. However, serializability describes the existence of a series of transactions, and therefore only requires one of such sequences that do not contain any cycle.

H2) Figure 1.2b is a *write skew* anomaly. T_1 and T_2 concurrently read titles of both `forum1` and `forum2` and also write to part of the two objects concurrently. A write skew has two consecutive RW edges that form a cyclic anti-pattern which violates SER. This anomaly, however, is widely accepted in a system that supports SI by default. The snapshot of each object is well maintained and the two transactions do not commit updates on the same object.

H3) Figure 2.1a is a *long fork* anomaly. T_1 updates the title of `forum1` from `"A1"` to `"A2"`; similarly, T_2 updates the title of `forum2` from `"B1"` to `"B2"`. However, T_3 only observes T_1's update and ignores T_2's; on the other hand, T_4 only observes T_2's update and ignores T_1's. This anomaly causes a cyclic anti-pattern with two RW edges which are not consecutive. Therefore, a long fork anomaly violates both SI and SER but is accepted by PSI. In the long fork anti-pattern, we cannot find a total order of transactions to maintain the snapshots of all the objects. For example, to maintain the snapshot of `forum1.title`, we form a set of total orders including $T_4 \xrightarrow{\mathsf{VIS}} T_1 \xrightarrow{\mathsf{VIS}} T_3$, with T_2 taking place at any part of the order. However, this total order does not maintain the snapshot of `forum2.title` in any way as it requires an order that includes $T_3 \xrightarrow{\mathsf{VIS}} T_2 \xrightarrow{\mathsf{VIS}} T_4$, which imposes T_3 to be visible by T_4. However, it is acceptable by PSI, since SI restricts a total order while PSI does not.

H4) Figure 2.1b shows a transaction T_2 which repeats reading the same object `forum1` but retrieves different values, without any intermediate writes in between. This forms a *non-repeatable read* anomaly and causes an anti-pattern disallowed by SER, SI, and PSI.

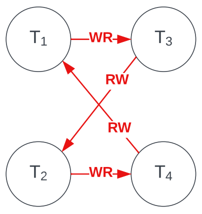
H5) Figure 2.2 shows a history that can be interpreted by two cases. In the first case, $T_1 \xrightarrow{\mathsf{VIS}} T_3$ and $T_2 \xrightarrow{\mathsf{VIS}} T_3$, with T_1 and T_2 concurrent with each other (as shown in Figure 2.2a). T_3 only reads T_2's update on forum members but does not manage to observe T_1's. In the second case, $T_2 \xrightarrow{\mathsf{VIS}} T_3$ with T_1 concurrent with T_3 (as shown in Figure 2.2b). T_3 is able to observe T_2's update, but T_1 fails. Both cases end up with a *lost update* anomaly: one read event cannot return the most recent value written by a successfully committed transaction that has also taken effect, but the one before, which means that the most recent update is lost. This anomaly causes a cycle with a single RW edge, which contributes to an anti-pattern against SER, SI, and PSI.

H6) Figure 2.3a shows an anomaly of a *cyclic information flow*. T_1 updates the title of `forum1`, and T_2 updates the title of `forum2` concurrently. However, both updates can be observed by the other transaction although the transaction has not been submitted. This causes an anti-pattern against PL-2 by a cycle without RW edges. In fact, the history shows that the two transactions are not independent of each other, which is a lower level of isolation compared with SER, SI, or PSI.

H7) Figure 2.3b shows an intermediate read anomaly. T_2 retrieves the intermediate value `"A2"` written by T_1. Although a dependency graph is provided, this anomaly actually violates the EXT axiom, since T_2 does not observe the last write of T_1. Therefore, we consider intermediate reads as a form of dirty reads, and do not construct any of such edges on the dependency graphs. Another form of dirty reads is *aborted reads*, which reads the values written by aborted transactions. We consider dirty reads by including the three types of anomalies: aborted reads, intermediate reads, and circular information flows [13].

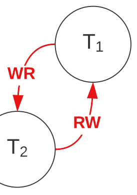
H8) Figure 2.4 shows a *write cycle* anomaly. In a social gathering, the couple Alice and Bob met another couple Cindy and David together. Both Alice and Bob first noticed and knew Cindy (T_1), then followed by David (T_2). However, the next day, when they recalled the order and how they got to know the two persons, Alice said they knew Cindy first, then David, but Bob said they knew David first, then Cindy (T_3). This cannot be explained unless one of them remembered the order wrong. In fact, Bob noticed David approaching and remembered him first, before David formally introduced himself. In Alice's view, $T_1 \xrightarrow{\mathsf{WW}} T_2$; while in Bob's view, $T_2 \xrightarrow{\mathsf{WW}} T_2$.This induces a write cycle between T_1 and T_2, and this history exhibits an anti-pattern with only WW edges, which is disallowed by PL-1. This example is naive and explains the case where the transactionality is not well established in the database. The order of the write events may not be ensured.

T_1:
```
write(forum1.title, "A2")
```
T_2:
```
write(forum2.title, "B2")
```
T_3:
```
read(forum1.title, "A2")
read(forum2.title, "B1")
```
T_4:
```
read(forum1.title, "A1")
read(forum2.title, "B2")
```



(a) Long fork

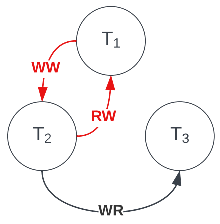
T_1:
```
read(forum1.title, "A1")
write(forum1.title, "A2")
```
T_2:
```
read(forum1.title, "A1")
read(forum1.title, "A2")
```



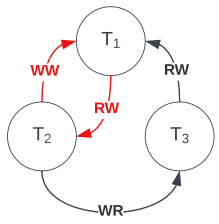
(b) Non-repeatable read

Figure 2.1: Anomalies: long fork and non-repeatable read

T_1:
```
read(forum1.members, list)
append(forum1.members, Alice)
```
T_2:
```
read(forum1.members, list)
append(forum1.members, Bob)
```
T_3:
```
read(forum1.members, list + Bob)
```



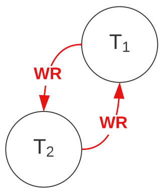
(a)



(b)

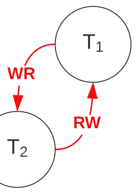
Figure 2.2: Anomaly: lost update

T_1:
```
write(forum1.title, "A2")
read(forum2.title, "B2")
```
T_2:
```
write(forum2.title, "B2")
read(forum1.title, "A2")
```



(a) Circular information flow

T_1:
```
read(forum1.title, "A1")
write(forum1.title, "A2")
write(forum1.title, "A3")
```
T_2:
```
read(forum1.title, "A2")
```



(b) Intermediate read

Figure 2.3: Anomalies: circular information flow and intermediate read

T_1:
```
append(Alice.known, Cindy)
append(Bob.known, Cindy)
```
T_2:
```
append(Alice.known, David)
append(Bob.known, David)
```
T_3:
```
read(Alice.known, [Cindy, David])
read(Bob.known, [David, Cindy])
```

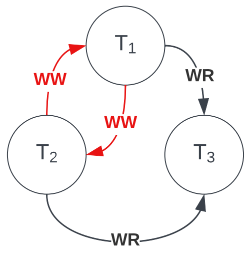


Figure 2.4: Anomaly: write cycle

2.2.1 Summary

Isolation Levels	Graph-Based Anti-Pattern	Write Skew	Long Fork	Non-Repeatable Read	Lost Update	Dirty Read
SER	any cycle	✗	✗	✗	✗	✗
SI	a cycle with zero or one consecutive RW edge	✓	✗	✗	✗	✗
PSI	a cycle with zero or one RW edge	✓	✓	✗	✗	✗
PL-2	a cycle without RW edges	✓	✓	✓	✓	✗
PL-1	a cycle with only WW edges (EXT being violated does **NOT** induce an anti-pattern)	✓	✓	✓	✓	✓

Table 2.1: Graph-Based Anti-Patterns with Anomalies

We illustrated a series of execution histories with anomalies (if present). To describe the anomalies shown in the histories, we borrowed the terminologies from the preventative approach. Furthermore, Table 2.1 summarizes the five isolation levels, their anti-patterns, and corresponding anomalies shown in the example histories.

Graph-based specifications are related to the preventative approach but they do not map to each other one-to-one. Furthermore, the definition from graph-based specifications is implementation-independent, and does not rely on the preventative approach. In fact, each anomaly case, as we borrowed from the preventative approach, is likely to be only a subset of graph-based anti-patterns, not all. For example, H4) and H5) are both disallowed by PSI, as both of them have a cycle with only one RW edge that matches the anti-pattern of PSI. Furthermore, graph-based anti-patterns are generalized to all database systems, while the traditional anomaly terms are often defined with the locking mechanism. Therefore, we should interpret the anomalies in Table 2.1 as only an explanation of our five isolation levels rather than any form of alternative definition.

In addition, the visibility of histories is essential, as it determines the version changes (called *version order*) of each object and further affects the dependencies between transactions. For example, the visibility between T_1 and T_2 is unclear in H5) and this leads to two interpretations of the version order of object x. This history is only disallowed by PSI if both cases are disallowed, or additional information can be obtained to ensure only one of the two cases can happen. Therefore, how to retrieve the visibility and the version order is one of the core problems that need to be handled by an isolation level checker.

It can also be observed in H7) that a write cycle is an anti-pattern that violates a much lower isolation level. The write cycle is interpreted at the event level rather than the transaction level, which makes it less expected in a well-established transactional database system and likely to be neglected by an isolation level checker.

2.3 ArangoDB: Distributed Graph Database

This section introduces the ArangoDB as a distributed graph database, including its isolation level guarantee. We have also provided the customized documentation of ArangoDB in Appendix A for reference.

2.3.1 Distributed Database

As a distributed database, ArangoDB supports four deployment modes, where the two deployment modes, the single-instance mode and the cluster deployment mode, will be explored and discussed in this thesis. The single-instance deployment involves only one stand-alone logical instance of ArangoDB and is the simplest way to start an ArangoDB system. On the other hand, the cluster deployment usually requires two or more logical instances that make a distributed system.

ArangoDB allows sharding and synchronous replication in its cluster mode. However, these two properties are not enabled by default; users need to explicitly activate these properties when starting the database servers. ArangoDB's distributed architecture allows it to scale horizontally across multiple servers while maintaining strong data consistency and fault tolerance. However, it may also trigger anti-patterns in the execution histories, which are to be verified in this thesis.

2.3.2 Graph Database

ArangoDB [1] is one of the most popular and efficient graph databases [32] [49] while having the capabilities to cover other data models, such as key-value pairs and JSON documents. ArangoDB has been designed as a native graph database with a focus on labeled property graph models. In a native graph database, nodes are linked directly in memory without further help of indexes. Such index-free adjacency ensures high-performance queries and rapid traversals. [52]

ArangoDB uses its own query language, ArangoDB Query Language (AQL), to perform operations on its data models. Listing 2.1 presents an example AQL query to read a person with the first name `"Alice"`. In ArangoDB, queries are formed in strings and passed to the relevant server by a driver. ArangoDB supports its official drivers in many advanced programming languages, such as Java, Go, and Python.

```
FOR p IN person
    FILTER p.firstName == "Alice"
    RETURN p
```

Listing 2.1: AQL: Example

In addition, ArangoDB provides its native support of graph traversals. Listing 2.2 presents an example AQL query to start a graph traversal from the person `alice`, with a minimum depth of 2 and a maximum depth of 4, on the graph `known_graph`. The graph traversal is DFS-based by default. BFS, and other relevant settings, are also supported by the graph traversal syntax.

```
FOR vertex, edge, path IN 2..4
    OUTBOUND "person/alice"
    GRAPH known_graph
    RETURN CONCAT_SEPARATOR("->", path.vertices[*].name)
```

Listing 2.2: AQL: Graph Traversal

2.3.3 The Isolation Level Guarantee of ArangoDB

ArangoDB claims its isolation level guarantee as LOCAL SNAPSHOT ISOLATION [1]. This guarantee can be decomposed into two scenarios: in the single-instance mode, ArangoDB achieves SNAPSHOT ISOLATION; in the cluster mode, the isolation level is unknown and no guarantee is provided (except for OneShard, where the isolation level guarantee is the same as in the single-instance mode).

ArangoDB uses the RocksDB storage engine [31] that provides the realization of snapshots. In the single-instance case (and the OneShard case), ArangoDB claims that the preventative definition of its isolation level is best described as "repeatable read" [2] but actually uses a statement of SNAPSHOT ISOLATION: "a transaction T does not see writes from other transactions, which have started after T was started, even if they commit before the read of T happens". In addition, ArangoDB states that dirty read is prevented, snapshots are applied together with a well-defined total order, and that phantom is not prevented. In a cluster, each DB-Server maintains its own snapshot locally but no guarantee is provided for the isolation level of the cluster as a whole.

2.4 Cycle Detection Algorithms

Propositions 1 and 2 indicate that an anti-pattern in the dependency graph of an execution is essentially a directed cycle that satisfies the edge conditions specified in Theorems 1 and 2, respectively. In this section, we list the feasible algorithms to provide the background of cycle detection.

2.4.1 Algorithms By Definition of Cycles

In a directed graph, a *directed cycle* is a non-empty path where only the first and last vertices are equal. From the perspective of edges instead, a non-empty path in a directed graph contains a directed cycle if and only if the path contains a *back edge* pointing from a vertex to one of the vertices that have already occurred in the path (including the vertex itself). (Lemma 20.11 of [27]) It is a natural idea to detect a directed cycle (*Starting Vertex Detection*), or a back edge by definition (*Back Edge Detection*) in combination with depth-first traversals (DFS).

[1]Limitations — Transactions (ArangoDB 3.9): https://www.arangodb.com/docs/3.9/transactions-limitations.html#isolation

[2]Transactional Isolation (ArangoDB 3.9): https://www.arangodb.com/docs/3.9/data-modeling-operational-factors.html#transactional-isolation

An alternative solution is through breadth-first traversals (BFS) based on the following proposition: a directed cycle is a non-empty path that consists of one edge and one directed path connecting the ending and starting vertices of that edge (called *back path*). This proposition converts a cycle detection problem to a single-source shortest path problem in an unweighted graph where BFS becomes an effective solution (*Shortest Path Detection*).

Table 2.2 lists the three algorithms and their complexities, where n is the total number of vertices and m is the total number of edges in the graph. The pseudo code is included in Appendix B.

	Time Complexity	Space Complexity	In Appendix B
Starting Vertex Detection	$O(n(n+m))$	$O(n^2)$	Algorithm 2
Back Edge Detection	$O(n^2)$	$O(n^2)$	Algorithm 3
Shortest Path Detection	$O(m(n+m))$	$O(mn)$	Algorithm 4

Table 2.2: Algorithms By Definition of Cycles

2.4.2 Strongly Connected Component Algorithms

A *strongly connected component* (SCC) of a directed graph G is a subgraph C in which the *strongly connected* property is satisfied that there is a directed path from each vertex to another vertex, and no additional edges or vertices from G can be included in C without breaking the strongly connected property.

Grounded on the extended definition of a directed cycle with a back edge, the problem of detecting the existence of a directed cycle can also be relaxed to detecting a strongly connected component in a directed graph (see Appendix C for proof). Several SCC detection algorithms are efficient through single-source DFS traversals, including Tarjan's SCC algorithm [61], Kosaraju-Sharir's algorithm [56], and path-based strong component algorithm [33] [34], etc. Compared with algorithms based on definitions, the SCC detection algorithms utilize additional data structures to maintain those vertices already visited in the graph, such that repeating visits on the same vertex can be avoided. Tarjan's algorithm stores two types of indices in two arrays to maintain the set of vertices that are already visited. Kosaraju-Sharir's algorithm initiates two DFS traversals on the transpose of the input graph and the graph itself. The path-based algorithm maintains two stacks to achieve a similar goal. Overall, the SCC detection algorithms manage to achieve linear complexities in both time and space. Table 2.3 lists the algorithms and the pseudocode is included in Appendix B.

	Time Complexity	Space Complexity	In Appendix B
Tarjan's	$O(n+m)$	$O(n)$	Algorithm 5
Kosaraju-Sharir's	$O(n+m)$	$O(n+m)$	Algorithm 6
Path-Based	$O(n+m)$	$O(n)$	Algorithm 8

Table 2.3: SCC Detection Algorithms

2.4.3 Bulk Synchronous Parallel Model

The Bulk Synchronous Parallel (BSP) model [63] is a parallel computing paradigm that simplifies the design of parallel algorithms. The BSP model regulates a program in a series of *supersteps*. Every superstep is divided into three phases: computation, communication, and synchronization. Corresponding to the three phases, a computer that supports the BSP model requires the three elements: a number of processors (called components in [63]), a router to facilitate the message passing among the processors, and facilities for synchronization. During the computation phase, each processor performs its computation locally and independently. In the communication phase, processors exchange data with each other. In the synchronization phase, each processor reaches the barrier and waits until all other processes have reached the same barrier. After finishing the three phases, the current superstep is concluded, and the next superstep will be started until the whole program reaches the termination. The cost can be estimated by summing up the complexities induced by the algorithm itself in the computation phase and additional overhead in the other two phases.

With the emergence of large-scale graph processing systems, the BSP model is receiving the attention of practitioners in the field of graph databases. In 2010, Google extended the BSP model to a distributed iterative graph computing framework called Pregel and presented the work in the paper [48]. Pregel enables users to develop graph algorithms in a high-level programming language, which can then be executed on multiple nodes in a distributed computing environment. Many graph processing systems and frameworks have adopted Pregel's principles to enhance their performance, such as Giraph [17], Spark GraphX [66], and Flink Gelly [9]. Various graph databases, such as Neo4j [3] and ArangoDB [4], also offer support for custom Pregel algorithms.

The BSP model and the Pregel framework allow cycle and SCC detection algorithms to be implemented in a non-recursive way in graph databases by relying on the distributed iterative workflow. For example, ArangoDB has developed a collection of Pregel-based algorithms to strengthen its capabilities in graph analytics, including an SCC detection algorithm [67]. Meanwhile, TigerGraph [29] utilizes the Rocha-Thatte algorithm [53], a BSP-based cycle detection algorithm. [5]

2.5 Isolation Level Checkers

2.5.1 Problem Statement: The Black-Box Isolation Level Checking

Isolation level checking is part of the system testing process to ensure the correctness of the database system. It proposes an isolation level (e.g., SI) and determines whether the database system under test achieves the level or not. In this process, black-box testing has been used by a rich set of checkers, such as Cobra [60], DBCop [20], and Elle [42]. These

[3]Pregel API (Neo4j Graph Data Science Library v2.3): https://neo4j.com/docs/graph-data-science/2.3/algorithms/pregel-api/

[4]Programmable Pregel Algorithms (ArangoDB v3.9): https://www.arangodb.com/docs/3.9/graph-analytics-custom-pregel.html

[5]Cycle Detection in TigerGraph: https://www.tigergraph.com/blog/cycle-detection-in-tigergraph/

black-box checkers have minimized the interaction with the databases by first collecting histories from the database and then running the checking progress on the histories. In this way, the checking is stand-alone with only history collection depending on the system, making the whole process black-box testing.

Therefore, in the context of black-box testing, the isolation level checking problem is reduced to the following.

Definition 3 *The **black-box isolation level checking** is the problem to determine whether an execution history $\mathcal{H}$ collected from a database satisfies or is allowed by a certain isolation level I, i.e., $\mathcal{H} \models I$.*

Also, we make an assumption that for each object, every write event creates a unique value for the given key. We also state the assumption in the following, as it universally holds throughout the context of (black-box) isolation level checking. [28]

Assumption 1 ***Assumption of Unique Writes**: In any execution history collected for (black-box) isolation level checking, every write event **installs** a unique **version** on an object.*

Based on this definition of black-box level checking, the development of the checker itself does not necessarily involve white-box database execution information. For example, a syntactic generator independent of any database can simulate an execution history and pass it to the checker [20]. This helps ensure that the checker is well-functional and ready to be applied in practice.

However, in order to verify the isolation level of a certain database, a black-box history collector should submit concurrent transactions to the database system under test, wait for these transactions to take effect on the system, and finally collect the histories in order. Such a collector largely utilizes the system I/O and does not inject into the database to achieve system-related or functional details, which makes it a black-box implementation.

2.5.2 Visibility and Version Order

Following the discussion in Section 2.2, the visibility and the version order are essential to constructing the dependency graph from a given history. However, histories do not usually include such information: a history consists of only a sequence of transactions, with each transaction further composed of read and write events. For example, in Figure 2.2, the history H5) does not reveal the version order of object x. A straightforward approach is to enumerate all possible visibility orders and then infer the version order, as illustrated in this example. However, this method leads to a combinatorial explosion in complexity as the number of transactions grows.

One method of reducing complexity to a tractable level is to construct BC-polygraphs (Begin and Commit polygraphs) and use SAT/SMT solvers [60] [69] [38]. Based on Assumption 1, each read value is installed by a single operation (or, a single transaction). This implies that all read dependencies are available at the beginning of graph construction. BC-polygraphs begin from the read dependencies, while retaining other possible write dependencies and anti-dependencies in a dense graph. Next, a set of logic constraints is formed to reduce edges by merging redundant ones and pruning anomalous ones. Finally, SAT/SMT solvers are utilized to identify any directed acyclic graph (DAG) that validates the search. If successful, this indicates that a dependency graph without any anti-pattern has been found, and the corresponding isolation level has been satisfied. This approach satisfies the black-box checking goal as it only requires the history itself. However, building multiple graphs and their accompanying pruning heuristics is expensive, and it may not scale. Additionally, the checker and its associated pruning heuristics are restricted to a predefined isolation level, limiting its adaptability to other isolation levels.

Another approach is to utilize specific data structures that can be fed into the database system, which would automatically reveal the version order. One example is the appendable list structure proposed by Elle [42], which has two key properties: recoverability and traceability.

- **Recoverability**: Read dependencies should be able to be recovered.
- **Traceability**: The order of all versions written before should be available.

Since we ensured that Assumption 1 holds in any case, recoverability has already been achieved and this does not depend on the list data structure. However, lists are beneficial for traceability because we can track how different values are added to the list that is currently being read. Now, a single dependency graph can be constructed by using the version order directly from the history, which reduces complexities compared with BC-polygraphs. After the dependency graph is constructed, Elle follows Adya's and Cerone et al.'s formalism to detect cyclic paths by Tarjan's algorithm. By the reduction to a single graph, Elle provides easy access to graph-related concepts, tools, and algorithms, which opens up more possibilities for combination with other graph ecosystems. Despite requiring an extra data structure, Elle follows the black-box principle by relying on only the system I/O without additional information. Moreover, Elle's usability is comparable to other black-box checkers, which implies that Elle is a de facto black-box checker.

Aside from using a specific data structure, the version order can also be obtained by utilizing change data capture (CDC) techniques or MVCC records to stream out the modifications made to each object [26]. It is preferable for these tools to be integrated into the database being tested for easier access to the version order. However, this approach changes the nature of testing from black-box to gray-box, as more internal database information is needed for verification.

Occasionally, non-graph checking methods are employed in graph-based checkers because certain isolation levels and related anomalies need to be examined before constructing the dependency graph. For instance, if a failed (aborted) read is detected in the dependency graph, the edge becomes ill-defined since the aborted transactions are excluded from the dependency graph, and that part of the history is non-recoverable. Nonetheless, checking such an anomaly is crucial to ensure that the checker is effective and general. Such a rule-based approach is commonly used in the Jepsen [44] project, which has successfully detected many real-world software bugs in different distributed database systems.

2.6 Anti-Pattern Detection by Graph Database Queries

To summarize this chapter, this section discusses the feasibility of checking isolation levels with the help of a graph database, for example, ArangoDB. Graph databases can utilize both main and secondary memory, which allows for handling large-scale dependency graphs. Also, the analytical tools, including query languages and cycle detection toolkits, are usually optimized by the development team of graph databases in various practical use cases. Both of these factors make graph databases a feasible solution to checking isolation levels by detecting graph-based anti-patterns.

It is also noteworthy that graph query languages are able to handle different types of cycles with a similar syntax, which reduces the learning and development costs associated with building a checker based on graph databases. Therefore, writing graph database queries will be an essential part of this thesis to demonstrate the power of queries to verify different isolation levels.

Therefore, we use ArangoDB as both a system under test and a graph database checker. We utilize ArangoDB as a graph database to form a graph-based checker. Also, we exploit the functionality of ArangoDB as a distributed database to generate histories that can be further processed by the graph-based checker. The nature of a distributed graph database has made ArangoDB an ideal candidate for our research.

Chapter 3

Graph-Based Checker: Part I - History Collection and Graph Construction

In the previous chapter, we have highlighted the strengths of ArangoDB as both a distributed database under test and a graph database that can perform isolation checking. We present the design and implementation of our graph-based checker in this chapter, which involves the three stages of History Collection, Graph Construction, and Cycle Detection. We introduce the first two stages in this chapter, which is the pre-processing part of the checker. We provide the last stage, Cycle Detection, in the next chapter.

3.1 The Workflow

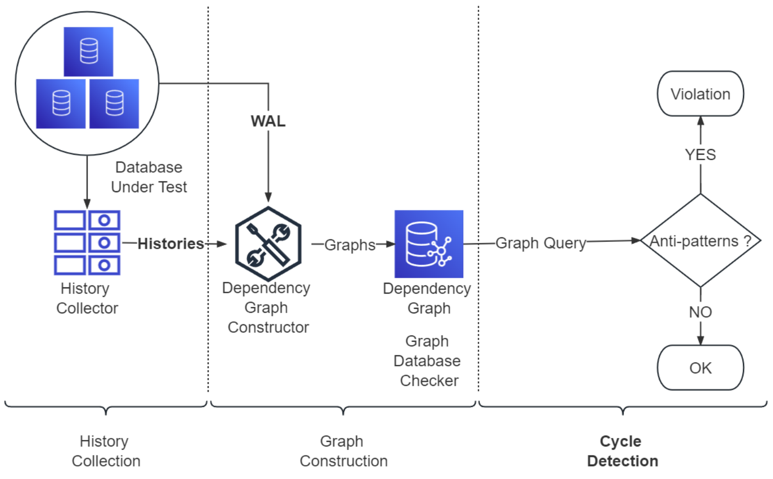


Figure 3.1: The workflow of the graph-based checker

Figure 3.1 demonstrates the general workflow of the graph-based checker. This implementation involves using graph queries to identify cycles in a dependency graph constructed from an execution history of a distributed database cluster. The checker can be decomposed into three stages: *History Collection*, *Graph Construction*, and *Cycle Detection*. As the major task is to verify the isolation level guarantee of ArangoDB, the History Collection stage initiates multiple transactions on a sequence of concurrent sessions. This is ensured by using the Jepsen injection framework [44] on an ArangoDB cluster. Next, the history collector retrieves these histories and passes the histories to the dependency graph constructor. In some cases, write-ahead logs (WAL) are also used to infer the version order, and they are also passed to the graph constructor. The Graph Construction stage reads the histories, performs the necessary conversions, and generates a new dependency graph for each history. Later, the heart of the checker is the Cycle Detection stage, which incorporates an anti-pattern detection process to identify at least one cycle from the dependency graph. In particular, AQL queries serve as a connection between the Graph Construction and Cycle Detection stages. These queries are used to send requests to an ArangoDB instance and retrieve the results from the graph database. The checker sends only one AQL query to the dependency graph for each isolation level checked.

3.2 History Collection

3.2.1 Jepsen: A Fault Injection Framework

We make use of Jepsen [44] to help us automatically generate random histories. Jepsen is a testing framework designed to verify the safety and consistency of distributed systems. It runs a test by executing a sequence of (transactional) operations on a distributed system and analyzing the resulting execution history to identify any inconsistencies or bugs. Meanwhile, it can additionally simulate a variety of faulty scenarios (called *nemeses*) and test the resilience of a system. The general workflow can be seen in Figure 3.2.

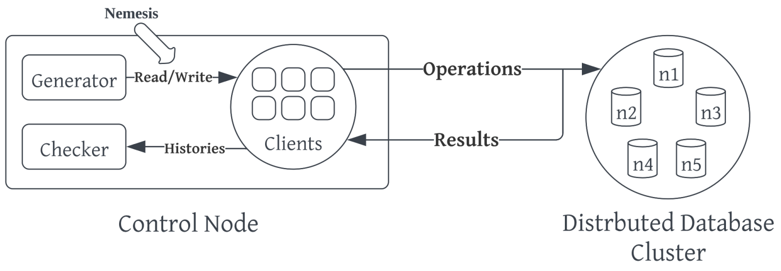


Figure 3.2: The Jepsen workflow

Jepsen uses a *control node* to coordinate the test and communicate with the distributed system under test. At the beginning of a test, the control node starts a cluster of five database nodes on virtual machines. Then Jepsen uses its control node to coordinate the transactions on the cluster and record the results in a history. When the test is finished, the history files are automatically generated in the output folder and this process is fully handled by Jepsen's injection mechanism.

As the central component of Jepsen, the control node orchestrates a set of client nodes that send out operations concurrently. The number of clients can be adjusted depending on the level of concurrency required. Additionally, each test includes a generator and a checker. The generator predetermines the flow of read and write operations, which are periodically passed on to the client nodes at random values. The checker analyzes the execution histories to determine consistency and visualizes the process and system-related metrics such as latency and rate. To simulate real-world conditions and potential failure scenarios, the generator can include Nemesis, which can inject faults such as node failure, network partition, and clock skew throughout the system under test. To prevent interference with the test results, Jepsen's control node is typically run on a separate machine from the cluster under test. This setup ensures that the results are accurate and reliable.

3.2.2 History Collection in the Graph-Based Checker

During the History Collection stage, the graph-based checker aims to execute a series of transactions and collect the corresponding histories. We have identified the Jepsen framework as a suitable tool for this task. The initial step involves setting up five ArangoDB nodes on five Vagrant virtual machines, with the SSH protocol used for communication between the control node and the virtual machines during the database nodes' installation.

Once the databases are installed successfully, the five ArangoDB instances on the virtual machines are connected as a cluster, and the workflow begins. The generator produces transactions. Each transaction consists of read and write operations, which are passed on to clients. Nemeses are also triggered in this step. After the transactions are generated, the clients execute these transactions concurrently. Each client contains a sequence of AQL queries sent via HTTP connections to the ArangoDB cluster. If any error is encountered, the client aborts; otherwise, the client will commit that transaction. The histories record whether a transaction is successfully committed or aborted. Successful histories include the information of read and write operations. In the final step, the internal checker designated by the Jepsen framework is used to run the verification process. The framework supports multiple Clojure-based checkers, such as Knossos, Elle, and a set of other checkers provided by the developer [1]. However, we have decided to ignore this step, as the new graph-based checker will replace the checker embedded in Jepsen by its algorithms in the checking stage.

3.2.3 Jepsen Histories and ArangoDB Write-Ahead Logs (WAL)

We collected two types of transactional histories using Jepsen, which have some similarities. Both types consist of only ***read*** and ***write*** operations within each transaction. In the Jepsen framework, we utilized AQL operations [2] and ArangoDB Stream Transactions [3] to control the execution of transactions. These histories adhere to the Assumption of Unique Writes (Assumption 1) and ensure recoverability.

[1]Available Checkers in Jepsen: https://github.com/jepsen-io/jepsen/tree/main/jepsen/src/jepsen/tests

[2]jasonqiu98/jepsen.arangodb: AQL operations

[3]HTTP Interface for Stream Transactions (ArangoDB v3.9): https://www.arangodb.com/docs/3.9/http/transaction-stream-transaction.html

However, there are differences between the two types of histories. The first type writes values as appendable lists, while the second type writes values as simple read-write registers for each object. We refer to them as *list histories* and *register histories*, respectively. List histories allow traceability, while register histories do not. Additionally, the write operations in list histories are commonly known as *append* operations, which accurately reflect their nature.

When collecting these histories, we can control their size and concurrency by providing relevant arguments to Jepsen.

- Collection time (in seconds): the collection time of the history in seconds; a metric linear to history length, referred to as `time-limit` by Jepsen
- Transaction generation rate (number of transactions per second): transaction generation rate, i.e., the number of transactions generated per second; a metric of system concurrency, referred to as `rate` by Jepsen
- Number of sessions: the number of concurrent sessions threads at the same time; a metric of parallelism but referred to as `concurrency` by Jepsen
- Maximum transaction length: the maximum number of events within any transaction; a metric that helps adjust the concurrency, referred to as `max-txn-length` by Jepsen
- Maximum writes per object: the maximum number of events for any object; a metric that helps adjust the concurrency, referred to as `max-writes-per-key` by Jepsen

Meanwhile, we fix the values of the following two arguments.

- `key-count`: the number of distinct objects at any time; a metric that affects system concurrency, fixed to 5
- `min-txn-length`: minimum transaction length, i.e., the minimum number of events within any transaction; a metric that helps adjust the concurrency, fixed to 2

In addition, the command line to run a Jepsen test has an optional argument for `nemesis`. Nemesis is a general term to describe various system faults covered by Jepsen. We used the nemesis argument to turn on/off the network partition in our history collection. We introduce a nemesis in a periodic pattern for some of our histories, which remains inactive for five seconds and then randomly partitions the network into two parts for five seconds, and so forth.

Two examples are listed below for a list history and a register history, respectively.

```
{:type :invoke, :f :txn, :value [[:r 4 nil] [:r 2 nil] [:append 3 1] [:append 4 1]
     [:r 3 nil] [:append 4 2]], :time 18417348107, :process 14, :index 0}
{:type :invoke, :f :txn, :value [[:append 4 3] [:append 5 1] [:append 5 2] [:r 3
    nil] [:r 5 nil] [:r 5 nil]], :time 18442734012, :process 19, :index 1}
{:type :fail, :f :txn, :value [[:r 4 nil] [:r 2 nil] [:append 3 1] [:append 4 1]
    [:r 3 nil] [:append 4 2]], :time 18460547580, :process 14, :error :
    ww-conflict, :index 2}
{:type :ok, :f :txn, :value [[:append 4 3] [:append 5 1] [:append 5 2] [:r 3 []]
    [:r 5 [1 2]] [:r 5 [1 2]]], :time 18468583939, :process 19, :index 3}
{:type :invoke, :f :txn, :value [[:append 5 3] [:r 1 nil] [:r 3 nil] [:r 2 nil] [:
    append 6 1] [:append 6 2] [:append 6 3]], :time 18486925084, :process 19, :
    index 4}
{:type :invoke, :f :txn, :value [[:append 0 1] [:r 7 nil] [:append 3 2] [:append
    2 1] [:r 3 nil]], :time 18495341516, :process 10, :index 5}
```

```
{:type :ok, :f :txn, :value [[:append 5 3] [:r 1 []] [:r 3 []] [:r 2 []] [:append
    6 1] [:append 6 2] [:append 6 3]], :time 18512103772, :process 19, :index 6}
{:type :ok, :f :txn, :value [[:append 0 1] [:r 7 []] [:append 3 2] [:append 2 1]
    [:r 3 [2]]], :time 18516867536, :process 10, :index 7}
{:type :invoke, :f :txn, :value [[:append 7 1] [:append 7 2] [:r 7 nil] [:append
    7 3] [:append 3 3]], :time 18519639722, :process 8, :index 8}
{:type :invoke, :f :txn, :value [[:r 7 nil] [:r 3 nil] [:r 2 nil] [:append 9 1]], :
    time 18547050309, :process 19, :index 9}
{:type :ok, :f :txn, :value [[:r 7 []] [:r 3 [2]] [:r 2 [1]] [:append 9 1]], :time
     18558278528, :process 19, :index 10}
{:type :ok, :f :txn, :value [[:append 7 1] [:append 7 2] [:r 7 [1 2]] [:append 7
    3] [:append 3 3]], :time 18563033073, :process 8, :index 11}
```

Listing 3.1: Jepsen list history

```
{:index 0, :time 16171464416, :type :invoke, :process 0, :f :txn, :value [[:r 2 nil
    ] [:w 2 1]]}
{:index 1, :time 16182764118, :type :invoke, :process 1, :f :txn, :value [[:w 2 2]
    [:w 2 3]]}
{:index 2, :time 16187072883, :type :ok, :process 0, :f :txn, :value [[:r 2 nil] [:
    w 2 1]]}
{:index 3, :time 16190092106, :type :fail, :process 1, :f :txn, :value [[:w 2 2] [:
    w 2 3]], :error :ww-conflict}
{:index 4, :time 16315834531, :type :invoke, :process 2, :f :txn, :value [[:r 0 nil
    ] [:w 2 4] [:r 0 nil] [:r 2 nil]]}
{:index 5, :time 16323925741, :type :ok, :process 2, :f :txn, :value [[:r 0 nil] [:
    w 2 4] [:r 0 nil] [:r 2 4]]}
{:index 6, :time 16334559998, :type :invoke, :process 3, :f :txn, :value [[:w 2 5]
    [:r 1 nil] [:r 2 nil] [:w 2 6]]}
{:index 7, :time 16343053422, :type :ok, :process 3, :f :txn, :value [[:w 2 5] [:r
    1 nil] [:r 2 5] [:w 2 6]]}
```

Listing 3.2: Jepsen register history

In general, a Jepsen history records each transaction twice: once for the invocation and once for the result. The invocation record is of type `:invoke` and lists the operations to be executed in the transaction. All read operations have the value `nil` as they have not yet been executed, while all write/append operations have the corresponding values ready in the invocation record. The object keys are generated as non-negative integers starting from 0, while the object values start from 1. The values of keys and values also depend on the arguments such as `key-count` and `max-writes-per-key`. For example, when the maximum number of writes is reached, the key will not be generated any further.

As for the result record, it usually has two types: `:ok` and `:fail`, indicating the success and failure of the transaction, respectively. Sometimes, there is another type called `:info`, which is present when Jepsen cannot determine the status of that transaction. For `:ok` transactions, each read operation records the value of a full list or an updated register, while each write operation only keeps the argument that has been written. In contrast, a `:fail` transaction retains the same items for both read and write operations and raises an error or error code provided by the database.

Unlike list histories which are traceable, register histories cannot directly determine the version orders of the objects. Therefore, additional strategies need to be used to reconstruct the possible records of values written to the objects, ensuring that the dependency graph can be successfully constructed. ArangoDB provides an HTTP API to tail the recent server operations from the Write-Ahead Logs (WAL) [4]. At the end of the history collection process, an HTTP request is sent from the control node to copy the currently available WAL from the database system to the local file system. Here is part of the WAL corresponding to the register history shown in Listing 3.2:

```
{"tick":"105","type":2200,"tid":"140","db":"rwRegister"}
{"tick":"105","type":2300,"db":"rwRegister","cuid":"h2E06B5FF24F1/136","tid
    ":"140","data":{"_key":"2","_id":"rwCol/2","_rev":"_f7LTaqa---","rwAttr
    ":1}}
{"tick":"106","type":2201,"tid":"140","db":"rwRegister"}
{"tick":"107","type":2200,"tid":"145","db":"rwRegister"}
{"tick":"108","type":2300,"db":"rwRegister","cuid":"h2E06B5FF24F1/136","tid
    ":"145","data":{"_key":"2","_id":"rwCol/2","_rev":"_f7LTay6---","rwAttr
    ":4}}
{"tick":"109","type":2201,"tid":"145","db":"rwRegister"}
{"tick":"110","type":2200,"tid":"150","db":"rwRegister"}
{"tick":"111","type":2300,"db":"rwRegister","cuid":"h2E06B5FF24F1/136","tid
    ":"150","data":{"_key":"2","_id":"rwCol/2","_rev":"_f7LTa0----","rwAttr
    ":5}}
{"tick":"114","type":2300,"db":"rwRegister","cuid":"h2E06B5FF24F1/136","tid
    ":"150","data":{"_key":"2","_id":"rwCol/2","_rev":"_f7LTa0K---","rwAttr
    ":6}}
{"tick":"115","type":2201,"tid":"150","db":"rwRegister"}
```

Listing 3.3: ArangoDB Write-Ahead Logs (WAL) of Jepsen register history

Each line of the WAL includes a recorded timestamp in the `tick` field. In addition, ArangoDB provides multiple operation types [5]. For instance, the code 2300 signifies all insertion or replacement operations of database documents. These operations enable the availability of the version order for constructing the dependency graph. On the other hand, this approach utilizes additional information from the database and is, therefore, considered gray-box. However, the retrieval of the Write-Ahead Log (WAL) is done through HTTP requests rather than analyzing the internals of the software execution, which still satisfies the black-box goal and does not make the entire process white-box.

Currently, the retrieval of the WAL is only supported in single-machine mode. However, through discussions with the developers of ArangoDB, we have learned that support for cluster mode is already planned and in progress. Therefore, different from list histories collected from a five-node cluster, the current demonstration is conducted under single-machine mode with concurrency to simulate the distributed setting for the register histories. Nonetheless, once the cluster mode becomes available, this approach remains reliable and promising.

The HTTP request to retrieve the WAL has a set of query parameters. A noteworthy one is the `chunkSize`. It states the maximum number of bytes allowed by the WAL. The

[4]Write-Ahead Log (ArangoDB v3.9): https://www.arangodb.com/docs/stable/3.9/replications-walaccess.html#tail-recent-server-operations

[5]HTTP interface for WAL access (ArangoDB v3.9): https://www.arangodb.com/docs/3.9/http/replications-walaccess.html

`chunkSize` is an optional parameter[6] with a default value of 2^{20} . This means that a WAL can hold 1MB of logs if the `chunkSize` is not specified. It is often necessary to set a higher value than the default to allow a larger space for the WAL. However, the maximum value allowed by `chunkSize` is 128×2^{20}, i.e., the maximum possible size of the WAL retrieved via the HTTP request is 128MB.

3.3 Graph Construction

After both list and register histories have been ready, we design two strategies to map both types of histories to dependency graphs with WR, WW, and RW edges. In this section, we introduce Algorithm 1 to construct a dependency graph based on an execution history. The algorithm starts by creating a vertex for each committed transaction in the history (`createVertices`). After that, we construct the dependency edges on events, since the dependencies are described within the scope of each object that is independent of each other, and events are the minimal units that contain an object in an execution history. For this reason, the algorithm consists of two major steps: construction of dependency edges on events (`getEvtDepEdges`) and projection from events to transactions (`getTxnDepEdges`). The complete version of the graph construction algorithm can be seen in Appendix D.

Algorithm 1 Dependency graph construction

```
Input: execution history $\mathcal{H} = \mathcal{T}$, write-ahead logs $L$ (for register histories only)
Output: dependency graph $\mathcal{G}$
 1: $\mathcal{T}' \leftarrow \{T \in \mathcal{T} \mid T$ is committed$\}$        ▷ filter committed transactions
 2: $\mathcal{G}$.vertices ← createVertices($\mathcal{T}'$)
 3: $E_r \leftarrow \{e \in T \mid T \in \mathcal{T}' \wedge$ isRead$(e)\}$
 4: $R \leftarrow$ queryReadEvts($E_r$)        ▷ read events
 5: if $\mathcal{H}$ is a list history then        ▷ append/write events
 6:     $E_a \leftarrow \{e \in T \mid T \in \mathcal{T}' \wedge$ isAppend$(e)\}$
 7:     $A \leftarrow$ queryAppendEvts($E_a$)
 8:     $\mathcal{E}_{event} \leftarrow$ getEvtDepEdges($R$, $A$)
 9: else
10:     $E_w \leftarrow \{e \in T \mid T \in \mathcal{T}' \wedge$ isWrite$(e)\}$
11:     $W \leftarrow$ queryWriteEvts($E_w$)
12:     $\mathcal{E}_{event} \leftarrow$ getEvtDepEdges($R$, $W$, $L$)
13: $\mathcal{G}$.edges ← getTxnDepEdges($\mathcal{E}_{event}$)        ▷ projection
14: return $\mathcal{G}$
```

The function `getEvtDepEdges` has different implementations for list and register histories. List histories support traceability, which allows us to infer the version order based on the list values that are returned by read operations. These values, read by the database, reveal information about the write operations already taking effect on the system. Also, the

[6]REST WAL Access Handler (ArangoDB v3.9)

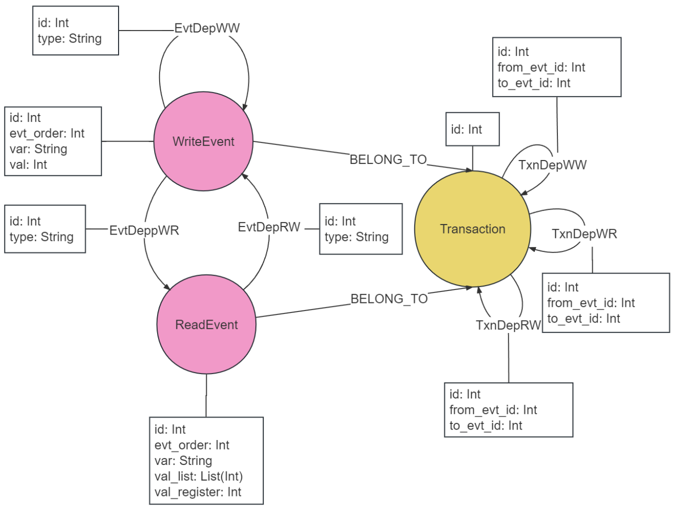


Figure 3.3: The dependency graph model in ArangoDB

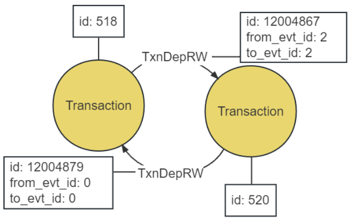


Figure 3.4: A write-skew anomaly detected in the graph model

values contained in a list maintain the order of write operations that update with these values. Therefore, we can iterate from the longest list to the shortest list per object to determine version order and reconstruct dependency edges.

We rely on WAL to restore the dependency edges for register histories. WAL records the timestamp and order for write operations to take effect. Therefore, we can retrieve the version order from WAL. We iterate over the version order per object, to determine the version changes and construct WW edges. We also recover WR and RW edges by comparing the values returned by read operations and the versions. Our approach for register histories requires a complete and sound WAL. Therefore, we should adjust the max size of WAL to ensure it is large enough to hold all logs of the system.

We also illustrate the dependency graph with Figures 3.3 and 3.4. Figure 3.3 presents the dependency graph on event and transaction levels. It has two event collections (`WriteEvent` and `ReadEvent`) and one transaction collection (`Transaction`). Also, it includes dependency edges constructed between events, and between transactions, respectively. After graph construction, we only consider the vertices in `Transaction` and the edges connecting two transactions, especially for the cycle detection stage discussed in the next chapter. Figure 3.4 illustrates an example of anti-pattern detection in our graph model.

Chapter 4

Graph-Based Checker: Part II - Cycle Detection in ArangoDB

In the previous chapter, we introduced how Jepsen and ArangoDB can help us construct the transactional dependency graph based on execution histories. This chapter discusses the core stage of our graph-based checker, Cycle Detection. We explore three cycle detection algorithms for identifying anti-patterns in dependency graphs in ArangoDB. To illustrate the query examples, we focus on the ***Snapshot Isolation (SI)*** level and utilize three versions of checkers: Cycle, SP, and Pregel. Some variants of the three major versions are also introduced. After discussing the algorithms for cycle detection, we also mention the ways of visualizing the output from the graph-based checker. Finally, we show our results that we did not find any anti-patterns by running our graph-based checker, and that ArangoDB does not violate its isolation level guarantee.

4.1 Checking SI: Definition-Based Checker

The Starting Vertex Detection algorithm detects directed cycles in $O(n(n+m))$ time complexity (See Section 2.4 and Appendix B.1), matching directed path where only the first and the last vertices are equal. As the ArangoDB graph traversal mechanism uses DFS in a default manner, we adapt the syntax to a cycle detection query as follows.

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

Listing 4.1: Checking SI: Cycle checker

We perform a graph traversal on each transaction vertex by a for loop `FOR start IN txn`. Here, the `start` is the starting vertex and a new graph traversal starting from this vertex is initiated whenever a new vertex is visited. A cycle is detected when the traversal reaches the starting vertex again. The traversal has a minimum depth of 2, because a cycle should contain at least two edges. However, it is important to select the maximum depth carefully to avoid exponential increases in time complexity, as discussed in Appendix B.1.

Since the Cycle checker requires an explicit input for the max depth, it may not be complete in detecting all cycles. This is because the maximum depth limits the number of vertices that can be involved in a detected cycle. For example, setting the maximum depth to 4 means the checker can only detect cycles with up to four transactions. However, in the context of isolation checking, we may assume that *cycles with more than four transactions do not exist in our histories*. The minimal max depth should be at least 4 to achieve full functionality, which corresponds to the long fork anomaly. In other histories collected from ArangoDB, it is rare to see a cycle involving more than four transactions. Also, we may increase the history size, or simply collect histories multiple times, to ensure that we can get a minimum depth of cycles of the same type. In practice, the user can still adjust the max depth according to specific requirements.

An anti-pattern for snapshot isolation is defined as a cycle with fewer than two consecutive RW edges. The query employs a negated condition, searching for at least two consecutive RW edges using a regular expression pattern `(^rw.*rw$|rw rw)` on the edge sequence. This regular expression matches the pattern where either one RW edge is at the start and another is at the end, or two consecutive RW edges appear in the sequence.

The Cycle checker filters all anti-patterns related to snapshot isolation. However, by using the `LIMIT 1` clause, only the first detected anti-pattern is outputted as the result. The `LIMIT` clause has minimal impact on performance, and the difference in I/O is negligible.

In addition to the standard version of the Cycle checker, we also propose three variants in the following.

Variant I: CycleFilter. Inspired by the idea of early stopping along a traversal, we have found two ways of implementing this in ArangoDB: using the `PRUNE` keyword and using the *filter-on-path* property. The `PRUNE` keyword specifies a pruning condition, which triggers early stopping when the condition is satisfied. However, it does not have the functionality of filtering. It implies that when the condition is not satisfied, the traversal will be complete, and no pruning seems to exist just by telling the results. Therefore, to achieve the correct set of results, we need a second round of filtering for each vertex, which increases the computational cost.

Another way is to use filtering on path [1]. Different from filtering on vertices or edges, if the path variables are explicitly used in the filtering condition for graph traversals, it will turn on the early stopping mechanism ensured by ArangoDB. This variant is promising when a cycle can be successfully found. However, when the cycle does not exist in a dependency graph, the path variable often causes more computational cost because of the processing from the path to edges or vertices that are easier for comparison. We present this variant in the query shown in Listing 4.2.

[1] Filtering on path vs. filtering on vertices or edges (ArangoDB v3.9): https://www.arangodb.com/docs/3.9/aql/graphs-traversals.html#filtering-on-the-path-vs-filtering-on-vertices-or-edges

Variant II: CycleRandom. Another variant is to randomize the iteration order to change the possibilities of detecting the cycle earlier. Randomization is often supported in more advanced programming languages and can be useful when combined with graph queries. Before the query is started, the sequence of starting vertices is randomly shuffled in the outside advanced programming language on which the driver is based. Then a set of queries will be executed in a loop, where each time a randomized starting vertex is passed to a query for cycle detection. Once an anti-pattern is found during the iteration, the program immediately breaks the loop so that the early stopping is triggered.

Based on randomization, it is possible to find a cycle earlier. However, there is also a possibility that the valid starting vertex that causes the cycle will appear toward the end of the iteration sequence. Therefore, randomization is considered only as a backup approach in this context.

```
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 4.2: Checking SI: CycleFilter variant

Variant III: CycleEdge (Disregarded). Back edge detection is another form of a definition-based algorithm. However, it is not suitable for this situation as it requires a hash set as an additional data structure. Unfortunately, the AQL query language does not currently support such a data structure that can change dynamically in value or size. As a result, back edge detection can only be supported in a much slower way. For instance, in the following query, each newly visited vertex is compared with the path that contains all the previous vertices to determine whether the new vertex has already been visited. The `POP` keyword is used to remove the last vertex from the path because it is exactly the new vertex. However, we have decided to disregard this variant due to its poor performance caused by comparisons between an element and an array, which is linear and adds one polynomial scale to the time complexity.

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

Listing 4.3: Checking SI: CycleEdge variant

4.2 Checking SI: SP Checker

The SP checker uses the Shortest Path Detection algorithm (See Section 2.4 and Appendix B.3) to detect cycles. It iterates over dependency edges and uses the `K_SHORTEST_PATH` query to find back paths that form cycles. Isolation checking identifies cycles that match a specific anti-pattern. We also notice that retrieving only the shortest cycles starting from each edge is insufficient. To address this, we employ the `K_SHORTEST_PATH` query to search for all cycles in the dependency graph. We then filter cycles that match the anti-pattern using regular expression matching. Neo4j, another graph database, also utilizes similar concepts and built-in shortest path algorithms for cycle detection.

```
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 4.4: Checking SI: SP checker

Additionally, the filtering step can be performed outside of the query without the final filtering step, using an advanced programming language. For instance, the provided code in Listing 4.5 demonstrates how the SI anti-pattern can be defined in the Go programming language. By adopting this approach, the query request primarily focuses on the intensive cycle search, while each cycle is individually evaluated outside of the query to determine if it matches the SI anti-pattern. Moving the filtering step out of the query does not affect the performance of SP Checker.

```go
func isAntiPatternSI(cycle []TxnDepEdge) bool {
    for i, edge := range cycle {
        if edge.Type == "rw" && cycle[(i+1)%len(cycle)].Type == "rw" {
            return false
        }
    }
    return true
}
```

Listing 4.5: Checking SI: anti-pattern definition of SI (in Go)

4.3 Checking SI: Pregel Checker

Combining Listings 4.6 and 4.7, we use the ArangoDB Pregel SCC algorithm to search all strongly connected components (SCC) within a dependency graph, and then filter valid SCCs with at least two vertices that represent cycles formed by SCCs (or subsets of SCCs). Listing 4.6 shows demonstrates how a Pregel job can be started on the dependency graph in ArangoDB. The resulting strongly connected component (SCC) IDs for each transaction vertex are stored in the `scc` field.

This checker, however, only applies to serializability checking. This is because the Pregel algorithm is based on Bulk Synchronous Processing (BSP) framework, which requires traversals along the whole graph in a distributive iterative manner. After the traversal is complete, the SCCs need to be retrieved, and the cycles need to be found within SCCs again. This process requires additional operations on subgraphs formed by SCCs. However, the subgraph-related functionalities are not well supported by ArangoDB, which causes problems with the post-filtering steps. Therefore, we only provide the way to apply the Pregel SCC algorithm to serializability checking with ArangoDB. In other graph databases (e.g., Neo4j), similar SCC algorithms can be combined with edge filtering, which makes it feasible to check isolation levels on other levels.

Note that in Community Edition, the correctness of results is ensured only in the single-machine mode, where the graph is stored within the same server node for cycle detection.

```
jobId, err := db.StartJob(context.Background(), driver.PregelJobOptions{
    Algorithm: driver.PregelAlgorithmStronglyConnectedComponents,
    GraphName: "txn_g",
    Params: map[string]interface{}{
        "resultField":       "scc",
        "shardKeyAttribute": "_from",
        "store":             true,
    },
})

if err != nil {
    log.Fatalf("Failed to start Pregel SCC algorithm: %v\n", err)
}
```

Listing 4.6: Checking SER: ArangoDB Pregel SCC Algorithm (Pregel)

```
FOR t IN txn
    COLLECT cycle = t.scc INTO cycles
    FILTER LENGTH(cycles) > 1
    RETURN cycle
```

Listing 4.7: Checking SER: retrieve cycles from SCCs

4.4 Output and Visualization

To present the results and indicate the existence of anti-patterns clearly, we provide both text and graph outputs that highlight any cycles detected in the dependency graph.

Text output. The text output represents the cycles using transaction IDs and retains the labels of the dependency edges, connecting them into a string. For example, an output like `T154 (rw) T155 (rw) T154` signifies a cycle with two read-write (RW) edges. However, the text output lacks detailed transaction information and event details. This limitation arises from the way dependency graphs are stored in ArangoDB. To enhance the output with more useful information and provide users with a visual representation of the cycle, we perform an additional read operation from the history and utilize the display format from Elle [42] along with the Graphviz visualization tool [3]. The updated graph output, as shown in Figure 4.3, provides a clearer illustration of a write skew anomaly. It indicates a cycle with one (RW edge from the event `[:r 66 []]` to `[:append 66 2]`, and another (RW edge from `[:r 62 []]` to `[:append 62 1]`. This cycle, consisting of two consecutive (RW edges, represents an anti-pattern of serializability (SER) but not of SI, PSI, or lower levels. Therefore, the example output suggests that the ArangoDB execution history achieves the SI level in a local mode.

ArangoDB visualization. In addition, ArangoDB offers a built-in visualization tool to visualize the graph model stored in the database. Figure 4.1 demonstrates an example of graph visualization using ArangoDB's Web Interface. It directly displays a cycle formed by four edges from the query's end result. However, the visualization based on the result only shows vertex information and lacks edge details, indicating room for improvement in visualizing labeled property graphs. This is because the edge information reveals the dependencies between transactions and is important in determining the type of cycles.

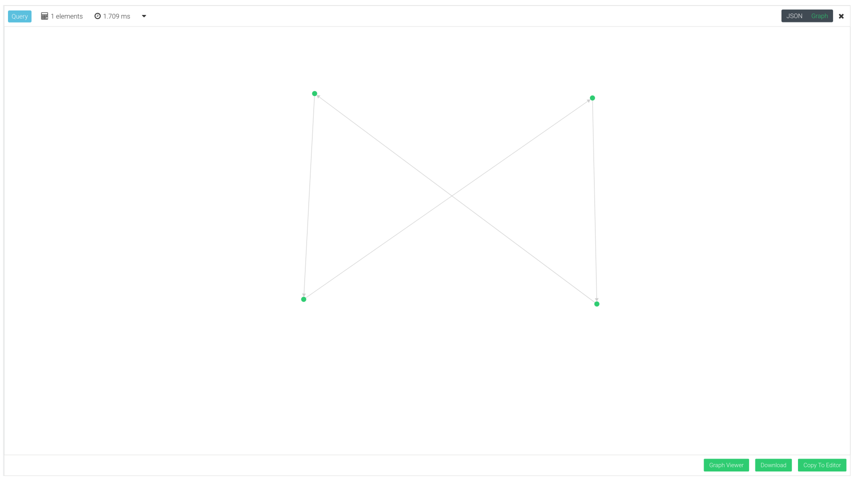
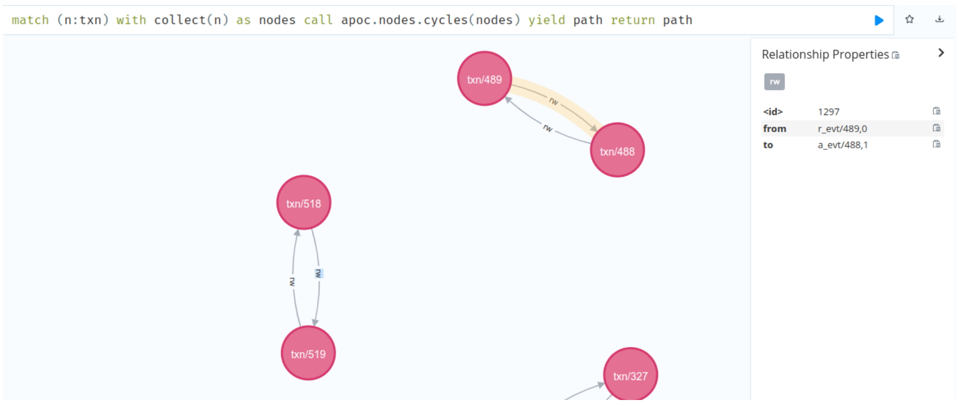


Figure 4.1: Cycle visualization by ArangoDB Web Interface

Neo4j visualization. Alternatively, other graph databases provide more robust visualization tools that cover both vertex and edge information. Figure 4.2 showcases a visualization result using Neo4j with the same graph schema. It includes well-maintained IDs and fields of both vertices and edges. In particular, Neo4j visualization well records the edge information, which contains the information of both end vertices and their IDs. This information, however, is missing in the ArangoDB visualization. This alternative way of smart visualization shows the feasibility of maintaining more useful information and providing insights. With the edge information, users can tell that all the cycles shown in the visualization are anti-patterns of SER, since all of them consist of two consecutive RW edges. Overall, Neo4j strengthens the result of smart visualization and also indicates that ArangoDB has the potential to improve its visualization capabilities for holding edge-related information.



```
match (n:txn) with collect(n) as nodes call apoc.nodes.cycles(nodes) yield path return path
```

Relationship Properties

rw

<id>	1297
from	r_evt/489,0
to	a_evt/488,1

Figure 4.2: Cycle visualization by Neo4j Browser User Interface

Visualization through business intelligence. Furthermore, an extensive ecosystem of business intelligence (BI) products customized in visualizing graph databases exists. Examples such as [6] and [7] provide solutions for visualizing ArangoDB graphs with advanced functionalities in data analytics and business intelligence. These examples illustrate the potential for connecting the graph-based checker to the broader graph database ecosystem, enabling further extension of the checker's capabilities.

4.5 Isolation Level of ArangoDB Cluster

ArangoDB achieves SNAPSHOT ISOLATION (SI) in its default, local mode, which is applicable when replication and sharding are not actively enabled. In this mode, each collection is stored entirely within a single server, ensuring that graph traversals only occur within the local server where all edges are stored. However, it is important to note that this local mode does not actively use distributed properties. We investigated this claim and our graph-based checker detected the anti-pattern of serializability (SER) with the specific anomaly known as Write Skew [2] (Figure 4.3).

When sharding and replication are enabled in an ArangoDB cluster, the locality of data storage no longer holds, allowing for the detection of other anti-patterns. By configuring a replication factor of 3 and a sharding factor of 2, we were able to find an anti-pattern of both SI (and PSI) (Figure 4.4) [3], in addition to the anti-pattern of SER. This particular anomaly,

[2]Anomaly: Write Skew
[3]Anomaly: Lost Update

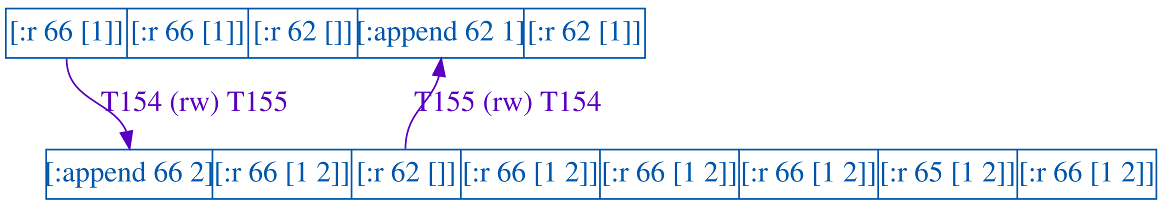


Figure 4.3: Cycle visualization: anti-pattern of SER

known as Lost Update, occurs when a committed update result for one object is perceived, while the update result for another object is lost. For example, T_{13379} updated two objects 2364 and 2365, and committed the transaction. However, the committed updated result of 2364 was perceived while that of 2365 was lost in T_{13381}. This violation involves a cycle with only one read-write (RW) edge, violating both SI and PSI. It should be noted that ArangoDB does not claim any isolation guarantee in a distributed setting. Based on our tests, the default isolation level in a distributed ArangoDB cluster can be classified as PL-2.

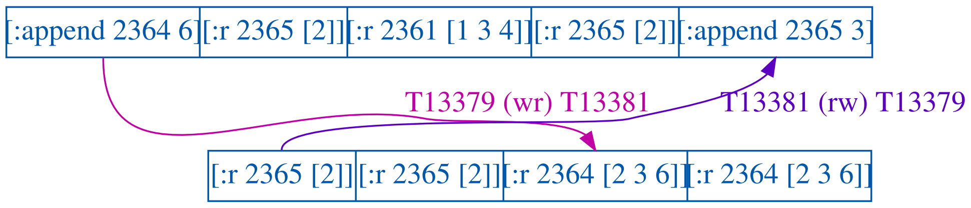


Figure 4.4: Cycle visualization: anti-pattern of SI & PSI

To conclude this chapter, we have developed a graph-based checker to assess the isolation level of a distributed database. The checker has proven to be informative and effective, and we have applied it to histories collected under different parametric settings in ArangoDB. Overall, we conclude that ArangoDB does not violate its isolation level guarantee of LOCAL SNAPSHOT ISOLATION (see Section 2.3.3).

Chapter 5

Evaluation

This chapter presents the evaluation of our graph-based checker in its effectiveness, scalability, and its efficiency compared to other checkers. We start by revisiting the three research questions proposed in Section 1.3 and decomposing each research question into detailed sub-questions. After that, we present the datasets and system configurations where we conduct the experiments. Following this, we perform exploratory data analysis, plot checker performance, and conduct statistical analysis to address the sub-questions of the research questions. At the end of this chapter, we give a summary of our results. We also extend the summary with a detailed discussion about our results, findings, and possibilities of improvements on the experiments.

5.1 Research Questions

5.1.1 RQ1: Effectiveness

RQ1 explores the effectiveness of the graph-based checker. To address RQ1, we apply the checker to five sets of list histories. Each set contains 20 histories, accumulating to 100 histories in total. We use these 100 histories to evaluate the research question in the following.

RQ1 Is the graph-based checker effective in detecting the anti-patterns of the 100 ArangoDB histories?

5.1.2 RQ2: Scalability

RQ2 asks for the scalability of the checker. We address RQ2 with a series of histories generated from ArangoDB, in both cluster and single-instance modes. In each set of histories selected to evaluate the scalability, we vary one certain factor without changing others. Furthermore, we explore some underlying factors that are not directly adjusted in the histories. We aim to explore the factors in the following.

- Two main factors used by Jepsen: *collection time* and *transaction generation rate*;

- Three underlying factors not directly used by Jepsen: *history length* (the number of committed transactions), *density* (the density of the dependency graph), and *contributing traversals* (the number of traversals spent on cycles);
- Two additional factors used by Jepsen: *number of sessions* and *maximum number of write events per object.*

Furthermore, we investigate the differences in scalability for the Cycle checker, when the maximum depth changes in the Cycle checker, or when the nemesis exists in the histories. Overall, we formulate the following sub-questions of RQ2.

RQ2.1 How does the graph-based checker scale with increasing collection times but a fixed transaction generation rate?
RQ2.2 How does the graph-based checker scale with increasing transaction generation rates but fixed collection time?
RQ2.3 How does the graph-based checker scale with the changes of the three underlying factors (history length, density, and the number of contributing traversals)?
RQ2.4 How does the graph-based checker scale when the number of sessions varies?
RQ2.5 How does the graph-based checker scale when the maximum number of write events per object varies?
RQ2.6 How does the Cycle checker scale with different maximum depths?
RQ2.7 How does the graph-based checker scale with increasing collection times but a fixed transaction generation rate, when a nemesis is active in the system?

5.1.3 RQ3: Comparison with State-of-the-art Checkers

RQ3 requires a response with the comparison between our graph-based checker and other state-of-the-art isolation checkers. We include two representative checkers, Elle and PolySI, into our experiments. Also, we attempt to change the graph database in use to illustrate the checker's performance with a different graph database, e.g., Neo4j.

RQ3.1 How does the graph-based checker perform compared to Elle?
RQ3.2 How does the graph-based checker perform compared to snapshot isolation checkers (e.g., PolySI)?
RQ3.3 How does the graph-based checker on ArangoDB perform compared to the graph-based checker in Neo4j?

5.2 Research Methodology

5.2.1 Datasets

To address the research questions and sub-questions, we conduct experiments on five sets of list histories and four sets of register histories. Each transaction has a range of 4-8 events.

List Histories. Table 5.1 includes the details of five list history sets List1-List5. The following two factors are fixed and not included in the table.

1. number of concurrent sessions to generate histories (#sessions): 10

	Collection time (s)	Rate (#txns/s)	Replication	Sharding	Nemesis
List1	10..200..10	80	3	2	✗
List2	10..200..10	80	3	2	✓
List3	100	10..200..10	3	2	✗
List4	100	10..200..10	3	2	✓
List5	30	80	5	3	✓

Table 5.1: Details of List Histories List1-List5

2. maximum number of writes per object: 8

We collect these histories through Jepsen, in a cluster setting with five nodes. We use shorthand notations `start..end..step`, to represent an increasing array of values from `start` to `end` with a step of `step`. For example, 10s..200s..10s means an increasing array 10s, 20s, 30s, ..., until and including 200s.

We adjust the two main factors, collection time and rate, on our list histories. List1 and List2 are with increasing collection times, but a fixed rate. On the other hand, List3 and List4 are with a fixed collection time, but increasing rates. In List5, both collection time and rate are fixed. We use List1-List3 for the evaluation of both effectiveness and scalability, with List4-List5 simply to verify the effectiveness of our checker.

We also introduce sharding and replication in this cluster. For example, List1-List4 histories have a replication factor of 3 and a sharding factor of 2. It means that each dataset is split into two shards. When new data comes into the database, consistent hashing is used to determine which shard to write to. Also, each shard is replicated three times to increase availability. In List5, we increase the replication and sharding factors to make data more scattered across the database servers.

We also introduce a nemesis in the generation of our histories. The nemesis is a network partition with a period of 10 seconds in the following manner. For every 10 seconds, there is no nemesis in the first five seconds, followed by a random partition that divides the network into two halves and lasts for five seconds. In our histories, List2 and List1 have the same configuration, except that List2 has a nemesis while List1 does not. It is also true of List3 and List4, with List4 having a nemesis.

Register Histories. Table 5.2 includes the details of four register history sets Reg1-Reg4.

Since register histories require the usage of WAL, which is supported only in the single-machine mode, we collect the register histories through Jepsen in a cluster with only one node. This setting disallows replication, sharding, or nemesis. In addition to collection time and rate, we also adjust the number of concurrent sessions (#sessions) and the maximum number of writes per object in our register histories (which are fixed in list histories). We use Reg1-Reg4 to explore the scalability of the checker with these factors.

	Collection time (s)	Rate (#txns/s)	#sessions	Max writes per object
Reg1	10..200..10	20	10	8
Reg2	30	10..200..10	10	8
Reg3	30	20	10..200..10	8
Reg4	30	20	10	1..20..1

Table 5.2: Details of Register Histories Reg1-Reg4

5.2.2 Dataset Characteristics

Appendix F lists the characteristics of all the datasets introduced above. For each dataset, the characteristics include the following.

- **#RW**: the number of RW edges in the dependency graph
- **#WW**: the number of WW edges in the dependency graph
- **#WR**: the number of WR edges in the dependency graph
- **#vertices**: the number of vertices in the dependency graph
- **#edges**: the number of edges in the dependency graph
- **#traversals**: the number of all the contributing traversals on paths that form cycles for each history
- **density**: the edge-vertex ratio to describe the graph density, which is the ratio of the number of edges to the number of vertices in the dependency graph
- **#committed**: the number of committed transactions in the history; the same as **#vertices**
- **#aborted**: the number of aborted transactions in the history

In the rest of this chapter, the descriptive results of these characteristics are reported in place of the raw data points.

5.2.3 System Configuration

We collect the list histories[1] and register histories[2] with Clojure 1.11.1 and Jepsen 0.2.7. To submit transactions from the Jepsen control node, we use ArangoDB Java Driver 6.16.0. ArangoDB 3.9.10 is used for history collection.

We conduct the experiments[3] on a Linux Mint 21 machine having AMD Ryzen 7 5800H processor with Radeon Graphics × 8 and 15.5 GiB of memory. The checker is developed with Go 1.19.4 and ArangoDB Go Driver 1.5.0. ArangoDB 3.9.10 is in use for the checker.

[1]jasonqiu98/jepsen.arangodb

[2]jasonqiu98/jepsen.arangodb.single

[3]jasonqiu98/GRAIL-artifact (thesis branch)

5.3 Exploratory Data Analysis

5.3.1 Correlation Coefficients of Key Factors

In this section, we conduct exploratory data analysis on the dataset characteristics. First, we report the correlation coefficients between each relevant factor and an array of 1..20, which is a common growth for all the datasets. Table 5.3 shows the results. For each dataset, we analyze the correlation coefficients below.

- **List1.** All the factors of **#vertices**, **#traversals**, **#aborted**, and **#edges**, are linearly correlated with regard to increasing collection time. The correlation of **density** is very low, meaning that the density is nearly unchanged across different collection times. It means that
- **List2.** The reported trends are nearly the same as **List1**, except the factor **#traversals**. The existence of a nemesis, with other factors keeping the same, will disturb the process of finding a cycle. It makes the trend of **#traversals** from strong linear correlation to nearly no correlation.
- **List3.** The signature trend for the histories with increasing rates is the strong negative correlation of **#density**. Compared with **List1**, the linear correlation of **#vertices** and **#edges** becomes weaker, which means the performance plot is not fully linear with these two factors.
- **List4.** We observe a similar trend to **List2**. The other factors are similar, except the **#traversals** which is largely disturbed.
- **List5.** This is a simple repetition of 20 times on the same configuration. Therefore, the data points, which show weak correlation, are consistent with our assumption.
- **Reg1.** For register histories with increasing collection times, we observe a similar trend to **List1**.
- **Reg2.** For register histories with increasing rates, we observe a similar trend to **List3**.
- **Reg3.** For register histories with an increasing number of sessions, all of the factors show a weak correlation. It implies that changes in the number of sessions do not take a strong effect on these key factors.
- **Reg4.** The main factor of max writes per object increases **#edges** and **density** and also decreases **#vertices** and **#traversals**. Also, **#aborted** increases.

5.3.2 Other Findings

We also briefly mention other findings in the exploratory data analysis. The first finding is that in list histories, the ratio of RW edges to the total number of edges has a strong correlation with the increasing rates; however, for register histories, the correlation is weak. Also, with a high rate, a list history has more RW edges than the other two types. Each history has nearly the same number of WW and WR edges. However, a register history does not have such a trend.

Another finding is that the number of aborted transactions is often strongly correlated with the main factor, but the slope may vary. For histories with a high rate, the growth of

	#vertices	#edges	#traversals	density	#aborted
List1	0.9927	0.9918	0.9807	-0.1158	0.9953
List2	0.9995	0.9992	0.0192	-0.136	0.9988
List3	0.885	0.663	0.9454	-0.9971	0.9743
List4	0.9475	0.6602	-0.1473	-0.9909	0.9946
List5	-0.2413	-0.2483	-0.1846	-0.1841	0.0011
Reg1	0.9998	0.9997	0.7278	0.0936	0.9656
Reg2	0.9967	0.9951	0.9725	-0.9029	0.9753
Reg3	-0.0672	0.0994	0.0994	0.1936	-0.0797
Reg4	-0.3501	0.767	-0.6714	0.767	0.5712

Table 5.3: Correlation coefficients of key factors

	SER	SI	PSI	PL-2	PL-1
Cycle (d=2)	100	73	73	0	0
Cycle (d=3)	100	75	75	0	0
Cycle (d=4)	100	75	75	0	0
SP	100	75	75	0	0
Pregel	100	/	/	/	/
Elle	100	75	/	0	0

Table 5.4: The number of histories that have anti-patterns in List1-List5

aborted transactions looks synchronous with that of committed transactions. With a low rate, the aborted transactions also grow with a low slope.

5.4 Effectiveness

We explore the effectiveness of our checker by comparing it with a reference checker. We select Elle as the reference checker for its stability and effectiveness. The comparison result addresses RQ1.

Table 5.4 presents the number of list histories where anti-patterns exist, across the datasets List1-List5. For the Cycle version, we also adjust the maximum depth to highlight its importance. This table suggests that our checker is capable of achieving the same level of effectiveness as Elle. The Pregel version can only be applied in the SER checking, but it can still fulfill the task. Cycles with a lower maximum depth may affect the effectiveness, e.g., a max depth of 2. This implies that part of our histories have minimal anti-patterns with a depth of 3, which invalidates the Cycle checker with a max depth of only 2. Therefore, we suggest the users set a max depth of at least 4 to ensure effectiveness.

Currently, there is no isolation checker with register histories that is complete on the checking of the five isolation levels selected by us. Elle highlights its use cases with list histories, but it lacks support to ensure the effectiveness of register histories. In **Reg1** histories, our checker detects 18 histories with anti-patterns of SER, while Elle only reports 3 of them. Therefore, Elle is not a reliable reference checker. This is because our checker utilizes WAL and retrieves more information than Elle, which only deduces based on intra-transaction information.

A limitation of our register histories is that since we rely on WAL, we cannot break through the restrictions of ArangoDB that WAL can only be collected in the single-machine mode. Therefore, we are not able to generate histories with replication and sharding, which are more likely to trigger anti-patterns in the execution histories. This causes our register histories only have anti-patterns of SER, and comparison with checkers (e.g., PolySI) of other levels becomes less meaningful.

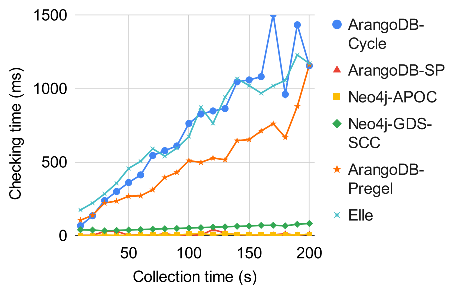
5.5 Scalability

RQ2.1: collection time Figure 5.1 presents the runtime of the graph-based checker for List1 histories with increasing collection time. We include the three versions based on ArangoDB to illustrate the growth trends, i.e., Cycle, SP, and Pregel. For the Cycle version, we set the maximum depth of cycles to 4, which ignores the paths over four depths. This maximum depth is sufficient in detecting a minimal long-form anomaly and also sufficient in detecting all anti-patterns in our datasets. Pregel is only present in Figure 5.1a, since it is only applicable in SER checking. As Figure 5.1f illustrates, the histories List1 also reflect the trends in increasing history lengths, which will be further discussed later in this section.

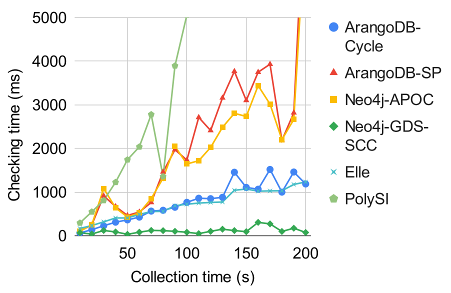
Both the Cycle checker and the Pregel checker are theoretically polynomial but achieve linear complexities in practice. The Cycle checker shows a linear trend across all five levels, with an R^2 value of 0.90 on average. The Pregel checker is also linear in the plot with an R^2 of 0.93. The Cycle checker executes complete traversals with the given depths when no early stopping mechanism is applied. The Pregel checker also traverses completely on the graph based on the BSP framework. Therefore, both checkers show a linear growth in their runtime.

However, the SP checker shows a constant level in 5.1a for SER checking, and a weaker correlation with other levels with many fluctuations. The R^2 values for SER and other levels on average are 0.01 and 0.66, respectively. The SP checker imposes an early stopping within the concept of the shortest path. Also, the `LIMIT 1` clause ensures that the checker can return as soon as one cycle is successfully found in the graph. Both reasons justify the constant level shown in SER checking. However, in our history, anti-patterns against other levels are fewer than those against SER. This makes the early stopping on other levels more difficult and explains why the runtime of the SP checker grows faster and its linear correlation with the horizontal axis is weak.

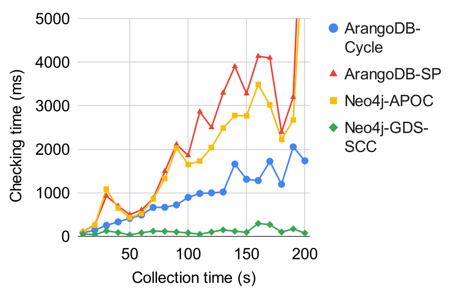
In general, the SP checker is superior in SER checking where anti-patterns are intensively present; however, the Cycle checker is better when only a few, or even no anti-patterns are present in the dependency graph. Pregel is stably linear and its performance lies between Cycle and SP. In Figure 5.1a, SP can return early, while Cycle and Pregel need complete traversals. In Figures 5.1b-5.1e, Cycle outperforms SP for the difficulty to find an anti-pattern and stop early. Also, SP traverses on edges while Cycle traverses on vertices. The



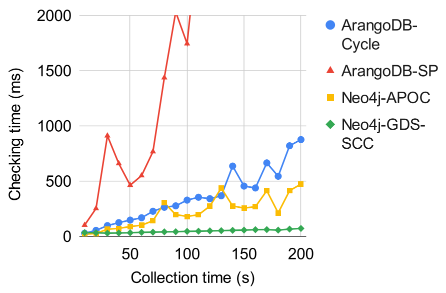
(a) Checking SER



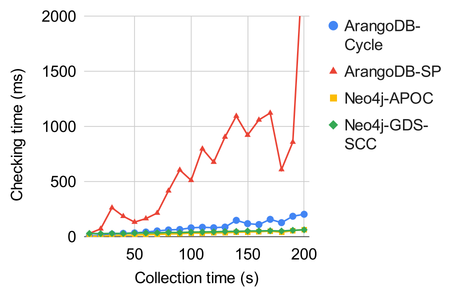
(b) Checking SI



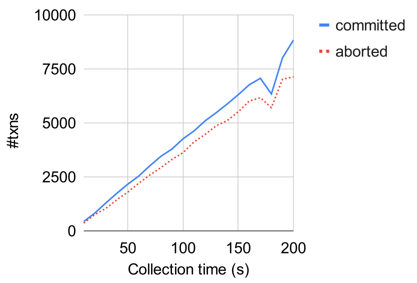
(c) Checking PSI



(d) Checking PL-2



(e) Checking PL-1



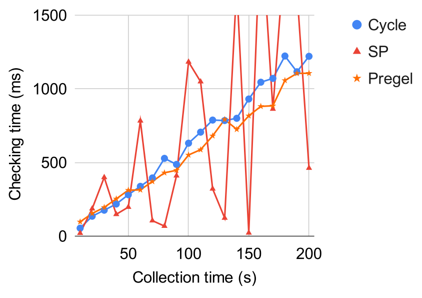
(f) History Length vs. Collection time

Figure 5.1: Runtime for checking anti-patterns in the `list-collection-time` histories

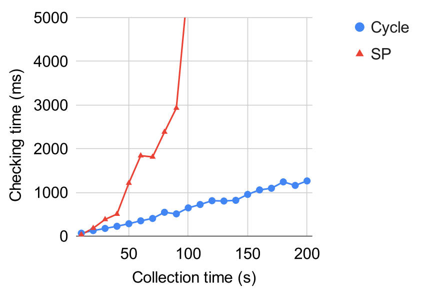
dataset characteristics show that the number of edges is more than the number of vertices (see Appendix F.1, which further explains the worse performance of the SP checker when it also needs to traverse completely.

Another finding is that the plots of SI and PSI checking are similar. SI and PSI checking both combine vertex searching and condition filtering. This similar mechanism causes similar performance for these two levels.

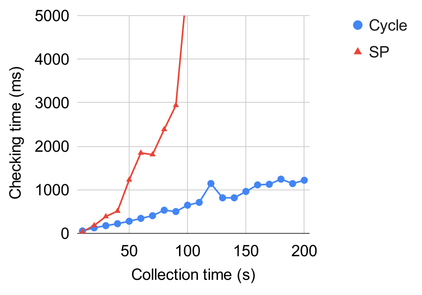
Figure 5.2 strengthens our arguments on the checker's performance with regard to increasing collection time for Reg1 histories, except that the SP checker shows more fluctuations instead of a constant trend in 5.2a. The major difference between Reg1 and List1 histories is that Reg1 is generated with a low rate, which lowers the concurrency of the dataset. This lower concurrency is also reflected in 5.2f, which shows that the dataset has only a small number of aborted transactions. With a low concurrency, there are fewer cycles



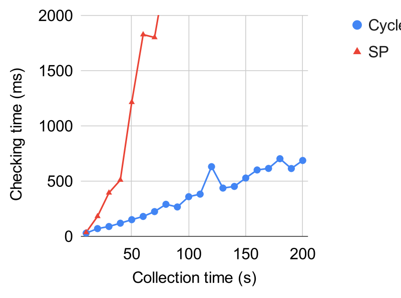
(a) Checking SER



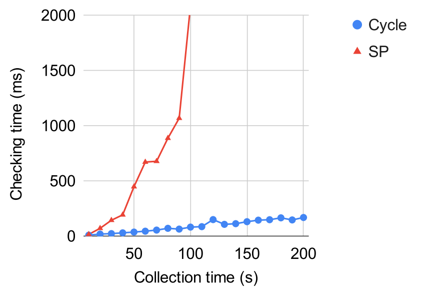
(b) Checking SI



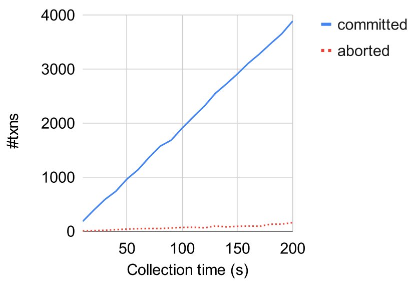
(c) Checking PSI



(d) Checking PL-2



(e) Checking PL-1



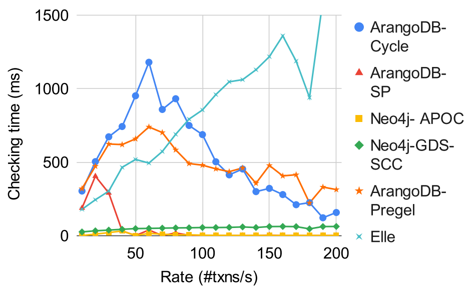
(f) History Length vs. Collection time

Figure 5.2: Runtime for checking anti-patterns in the `reg-collection-time` histories

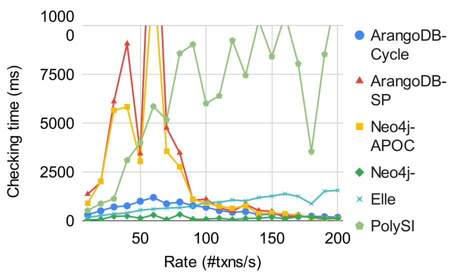
in the generated histories, which increases the difficulty to find a cycle and triggers more fluctuations in Figure 5.2a for the SP checker. On the other hand, the SP checker remains successful in achieving a low runtime for SER checking at some collection times (e.g., 70s, 80s, 130s, and 150s), which confirms the functionality of its early stopping mechanism.

RQ2.2: transaction generation rate Figure 5.3 presents the development trends of the checkers' performance for List3 histories, with an increasing array of rates. The rate is a representative metric of the concurrency in the system: a high rate causes more conflicts within the system, which further causes a higher ratio of aborted transactions. This trend can be seen in 5.3f.

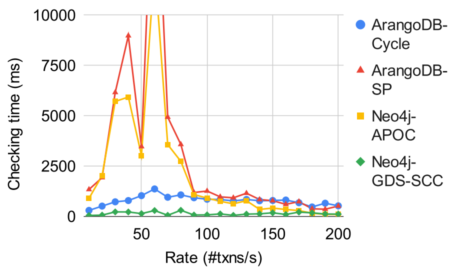
The increasing concurrency also changes the linear trends of the checkers' performance: the linear growth is largely broken for all checkers. Across all isolation levels, the runtime of



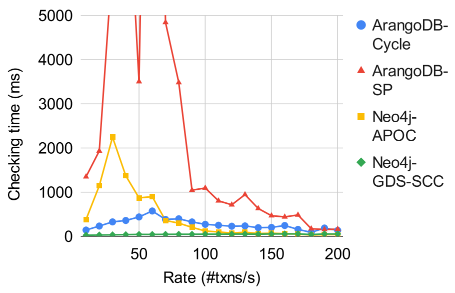
(a) Checking SER



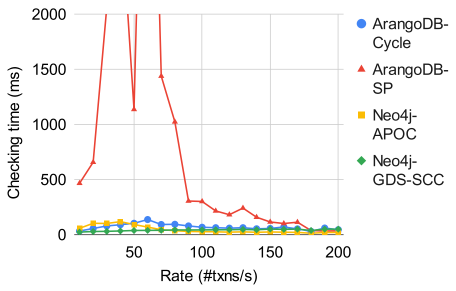
(b) Checking SI



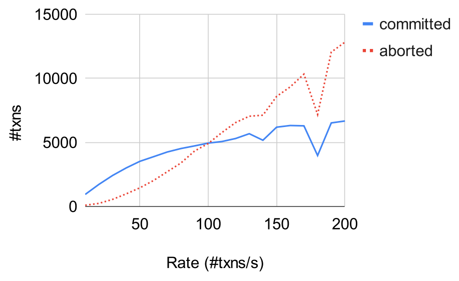
(c) Checking PSI



(d) Checking PL-2



(e) Checking PL-1



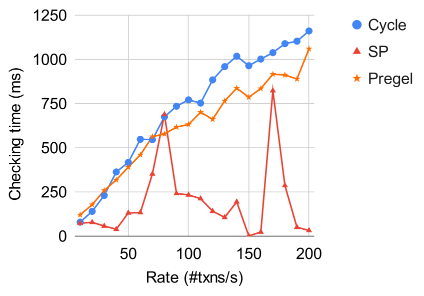
(f) History Length vs. Rate

Figure 5.3: Runtime for checking anti-patterns in `list-rate` histories

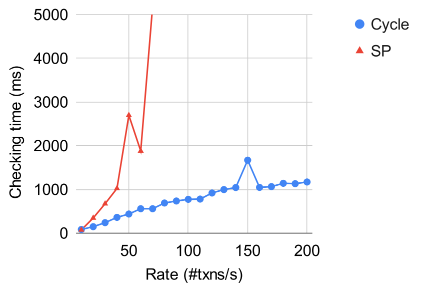
the Cycle checker shows a decrease-after-increase trend. The Pregel checker in SER checking also has a similar result. However, the SP checker still returns fast in SER checking (see Figure 5.3a) with effective early stopping.

The relative trends of the three versions are similar to the results for List1 histories. SP is superior in SER checking but returns more slowly for other levels. However, we also observe that the SP checker becomes faster for other levels when the rate becomes higher than 80. This is related to the decreasing trend of graph density (see Appendix F.3), which will be discussed later in this section.

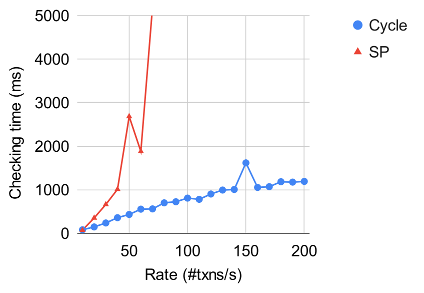
Figure 5.4 shows the development trend on Reg2, which increases rates on register variables. However, the trends show clearer linear trends and are less similar to those shown in Figure 5.3. Based on this result, we hypothesize that the development trends are less related to collection time or rate. The graph characteristics, instead, are more closely related



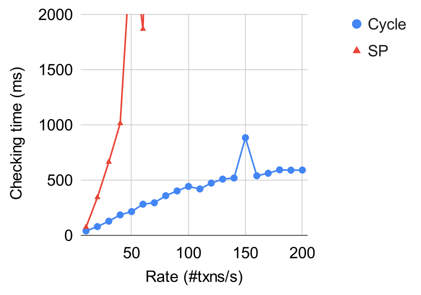
(a) Checking SER



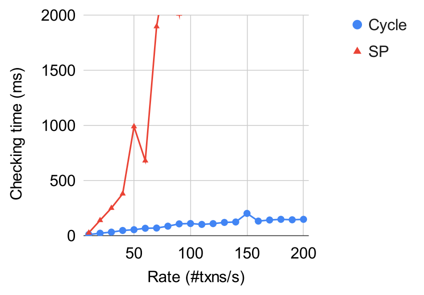
(b) Checking SI



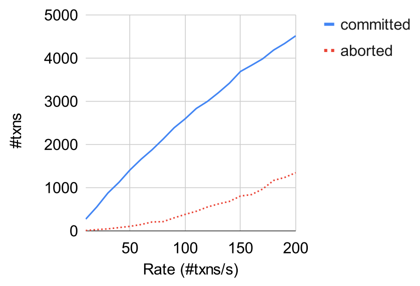
(c) Checking PSI



(d) Checking PL-2



(e) Checking PL-1



(f) History Length vs. Collection time

Figure 5.4: Runtime for checking anti-patterns in the `reg-rate` histories

to the runtime of the checkers. This will be detailed in the following paragraph.

RQ2.3: history length, density, and number of contributing traversals Based on the analysis on the history sets of List1, List3, Reg1, and Reg2, we select three underlying factors that affect the performance of the Cycle checker as follows.

- history length L: the number of committed transactions of the history
- density D: the edge-vertex ratio of the dependency graph
- number of contributing traversals N: the number of traversals spent on the paths that form cycles

We denote the runtime of the Cycle checker by T. With the three independent variables and one dependent variable, we establish a linear model.

$$T = \beta_0 + \beta_1 L + \beta_2 D + \beta_3 N$$

We use linear regression to explore the significance levels of the three underlying factors, with the runtime of SER and SI checkers on the four sets of histories List1, List3, Reg1, and Reg2. The runtime of SER is denoted by T_{ser} and that of SI by T_{si}. The cut-off for the p-value is 0.05. The results of linear regression are shown in Table 5.5. For both SER and SI checking, the coefficients of L and N are strongly significant with small p-values. The coefficient of D is also strongly significant with a cut-off p-value of 0.05 in SER checking. However, it is only marginally significant in SI checking. Both models have a good measure of fit based on their R^2 values. Based on these results, we confirm the effects of these three underlying factors. The only exception is that the density is less significant for isolation checking in levels other than SER.

To address this exception, we fit the model for List3 histories two more times and propose our hypothesis that density is an important underlying factor that affects the checker's performance. We start the linear regression procedures for two parts of the histories with rates less than 80, and at least 80, respectively. The results can be referred to in Table 5.6. The R^2 values of 0.95 and 0.88 ensure the validity of the model. Moreover, the coefficient density is strongly significant for histories with a rate of at least 80, which justifies our hypothesis. Therefore, we confirm that density is an important factor in List3 histories, and the modeling results are consistent with our observations.

T_{ser} (R^2=0.92)	β	p-value
Intercept	-588.42	0.078
L	0.27	3.50×10^{-41}
D	186.41	0.049
N	-0.64	2.62×10^{-16}

T_{si} (R^2=0.87)	β	p-value
Intercept	-603.59	0.178
L	0.29	5.71×10^{-34}
D	189.62	0.135
N	-0.67	7.38×10^{-12}

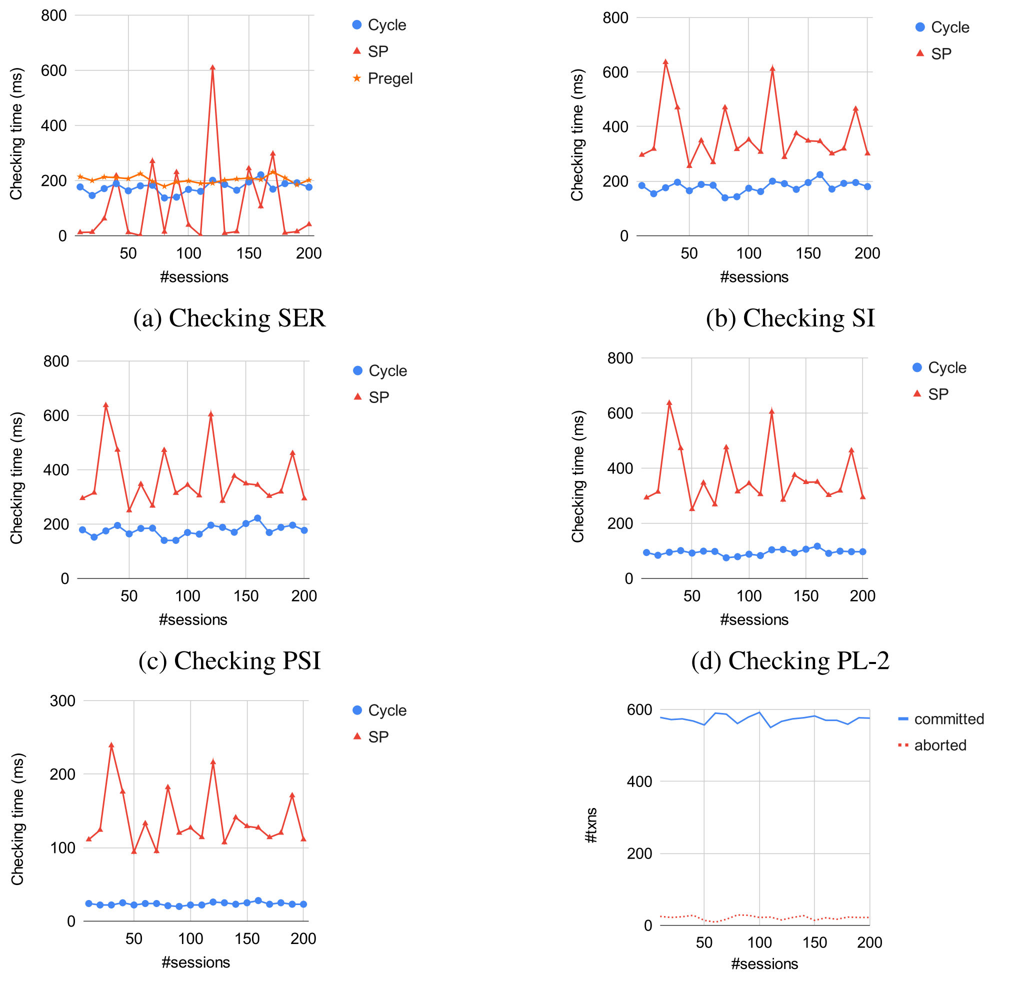
Table 5.5: Regression results for the runtime of SER and SI of Cycle checkers for List1, List3, Reg1, and Reg2 histories

T_{si} (R^2=0.95)	β	p-value
Intercept	-14406.06	0.070
L	0.64	0.026
D	3716.95	0.069
N	0.31	0.736

T_{si} (R^2=0.88)	β	p-value
Intercept	-1293.40	0.051
L	-0.05	0.636
D	701.97	0.011
N	0.02	0.903

Table 5.6: Regression results for runtime of Cycle checker in SI checking for List3 histories with rate less than 80, and at least 80

RQ2.4: number of sessions Figure 5.5 shows that the checkers have the same level of performance for histories with varying numbers of sessions. The number of sessions is



(a) Checking SER (b) Checking SI

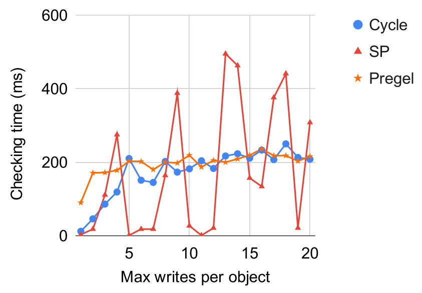
(c) Checking PSI (d) Checking PL-2

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

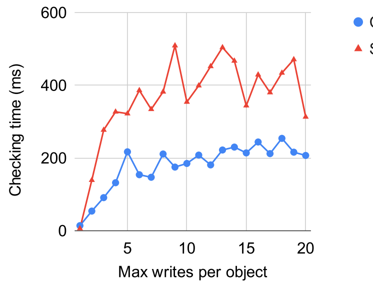
Figure 5.5: Runtime for checking anti-patterns in the `reg-session` histories

a metric to represent the parallelism of the system. With more sessions, the same load of data is more scattered across operating nodes, and therefore the concurrency is reduced. However, the underlying factors discussed in RQ2.3 do not change, which keeps the runtime at the same level.

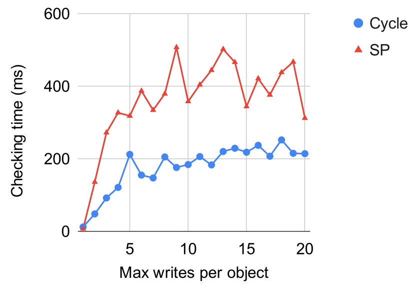
RQ2.5: maximum number of writes per object Figure 5.6 presents the effect of another additional factor, the maximum number of write events per object. We hypothesize a logarithm growth of the checking time with regard to max writes per object. The log-runtime has an R^2 of 0.84 on average of all levels for the Cycle checker. For the SP checker, the result of SER checking has a large extent of fluctuation and the R^2 is only 0.20. For the remaining four levels, the average R^2 of SP's log-runtime achieves 0.69, which is not good enough to determine a logarithm fit. Therefore, we conclude that the Cycle checker has



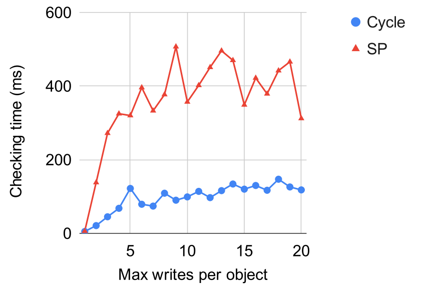
(a) Checking SER



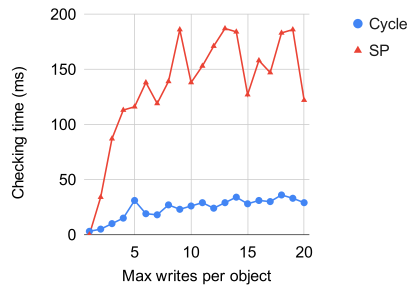
(b) Checking SI



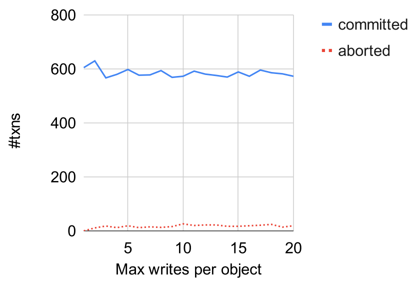
(c) Checking PSI



(d) Checking PL-2



(e) Checking PL-1



(f) History Length vs. Collection time

Figure 5.6: Runtime for checking anti-patterns in the `reg-max-write` histories

a logarithm increase in its checking time when the amount of the max writes per object increases. However, the SP checker deviates from a logarithm trend caused by fluctuation.

RQ2.6: max depth of the Cycle checker We also give an analysis of the effect of the max depth on the performance of the Cycle checker. Figure 5.7 presents the checking time for five histories selected from List1. The five histories correspond to those with collection times of 80s-120s. For all levels except PL-1, we observe an exponential growth with increasing depths for each level. The exponential trends in PL-1 are not clear enough. To verify our hypothesis, we construct the following statistical model with collection time t and max depth d as independent variables, and the runtime T as dependent variables.

$$T = \beta_0 + \beta_1 t + \beta_2 d + \beta_3 e^d + \beta_4 t e^d$$

We also add the interaction term to observe its effect on the performance. With the fit regression models, for simplicity, we list the p-values for the coefficients of each term to show their significance. A p-value below the cut-off of 0.05 is marked in boldface. It is shown that all the models corresponding to five isolation levels have high values of R^2, which indicates a good fitting level. In this case, the interaction term te^d is strongly significant for the checking of all levels. Also, the exponential term e^d is significant for the checking of all levels except PSI. Therefore, we conclude that our hypothesis is verified. The exponential terms are significant, and the depth has an exponential effect on the checking time. This is consistent with our analysis in Appendix 2.

	t	d	e^d	te^d
SER (R^2=0.98)	0.488	0.094	**3.67×10^{-4}**	**7.42×10^{-37}**
SI (R^2=0.97)	0.520	0.229	**8.08×10^{-3}**	**3.65×10^{-29}**
PSI (R^2=0.97)	0.920	0.386	0.343	**7.00×10^{-31}**
PL-2 (R^2=0.93)	**5.86×10^{-4}**	**0.001**	**0.019**	**1.74×10^{-19}**
PL-1 (R^2=0.89)	**7.11×10^{-17}**	**9.33×10^{-8}**	**4.33×10^{-5}**	**7.62×10^{-7}**

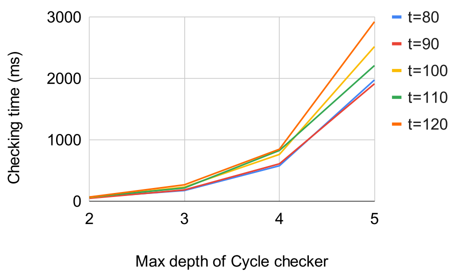
Table 5.7: A list of p-values for Cycle checkers with different max depths

RQ2.7: effect of nemesis We provide an additional experiment to explore the effect of nemesis on the development trends that we have observed in RQ2.1. In Figure 5.8, we present the trends of checking time for List2 histories, which illustrate the growth of the checking time with regard to increasing collection times. List2 has the same configurations as List1 except that List2 has a nemesis while List1 does not. From this figure, we have observed that the nemesis does not affect the general trends of the plots, and the plot in Figure 5.8 shows the same trends as Figure 5.1. In fact, the nemesis introduced by us, i.e., the periodic random partition, does not change the checkers' behaviors but affects only the size of the dependency graphs. It reduces the history generation time by half. Therefore, the development trends are consistent with the case without a nemesis.

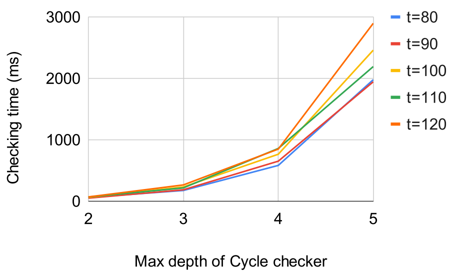
5.6 Comparison with Other Checkers

Following the discussion in the previous section, the three underlying factors are significant in affecting the checkers' runtime. In this section, we select two representative sets of histories, List1 (see Figure 5.1) and List3 (see Figure 5.3), to illustrate the comparison between our graph-based checker with other state-of-the-art checkers in SER and SI checking. We select two checkers for comparison: Elle and PolySI. Also, we explore the possibilities to create a new checker by replacing ArangoDB with Neo4j, and the new checker's performance.

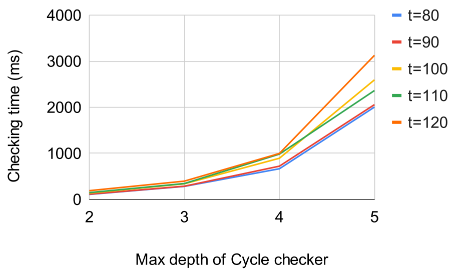
RQ3.1: Elle Elle is known for its linear-time checking on its benchmarks with list variables. Figure 5.1a demonstrates that Elle has a similar performance to Cycle but is outper-



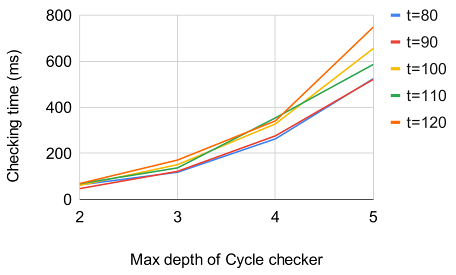
(a) Checking SER



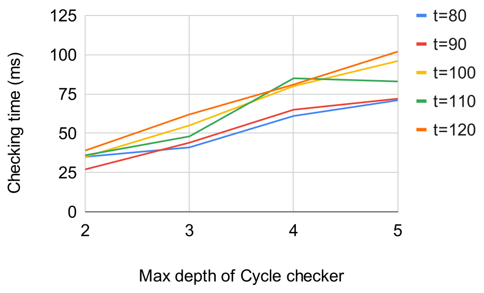
(b) Checking SI



(c) Checking PSI



(d) Checking PL-2



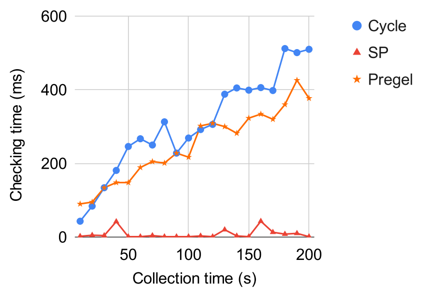
(e) Checking PL-1

Figure 5.7: Runtime for checking anti-patterns by Cycle checker with different max depths

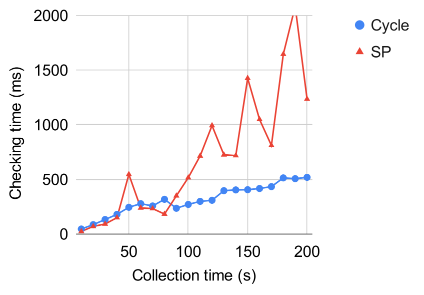
formed by our Pregel and SP checkers in SER checking for List1 histories. In SI checking (see Figure 5.1b), Elle's plot is still close to Cycle's, while SP checker performs worse because of its difficulty in finding a cycle.

We have also found that Elle has the same performance when checking different levels because of its mechanism: it first finds all SCCs by Tarjan's algorithm and then determines whether a certain level is achieved by rule. This way causes the same performance for all levels. Although we do not include Elle's plots in Figure 5.1d and 5.1e. Elle performs constantly worse than our PL-2 and PL-1 checkers, including Cycle and SP versions. Elle's PSI checker is ignored since its functionality is not ensured.

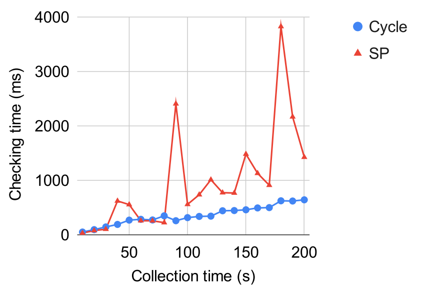
For List3 histories with regard to increasing rates, Elle shows a different trend from Cycle. Elle's mechanism requires a complete search on the whole graph, which causes its worse performance when the rate becomes higher (see Figures 5.3a-5.3b). When the rate is



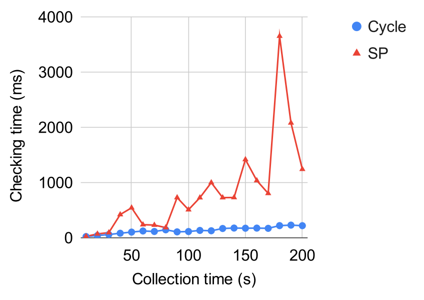
(a) Checking SER



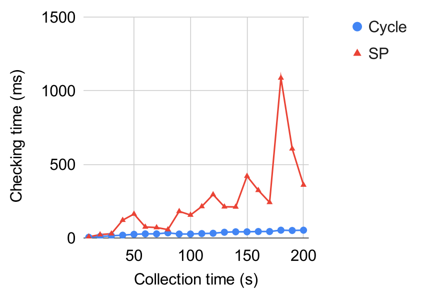
(b) Checking SI



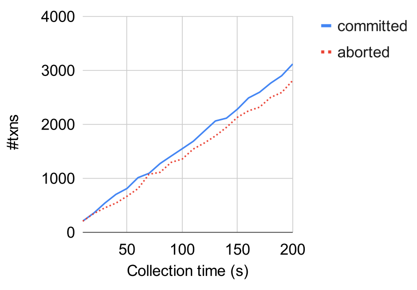
(c) Checking PSI



(d) Checking PL-2



(e) Checking PL-1



(f) History Length vs. Collection time

Figure 5.8: Runtime for checking anti-patterns in the `list-collection-time-nemesis` histories

low, Elle is generally better than Cycle because of the smaller size of dependency graphs. However, in Figure 5.3a, our SP checker still outperforms Elle in SER checking for low-rate cases, due to its early stopping mechanism.

RQ3.2: SI checker (PolySI) PolySI is a recent checker that detects violations against the SI level. As a representative solver-based checker, PolySI adopts the concept of BC-polygraphs for isolation checking. At the beginning of graph construction, PolySI recovers all the WR edges based on the assumption of unique writes (Assumption 1). Later, it enumerates version orders of objects, infers multiple dependency graphs by adding WW and RW edges that are possible to occur, and then designs pruning conditions to remove impossible cases. Following this, the MonoSAT solver is applied to retrieve any possible total order that is acyclic and does not violate the SI level.

PolySI suffers from the high complexities for creating BC-polygraphs, which inherently causes worse performance than our checker that constructs only a single dependency graph, as discussed in Section 2.5.2. This trend is seen in both Figures 5.8b and 5.3b. In Figure 5.8b, the runtime of PolySI quickly grows in a linear trend with a large slope, which is outperformed by our SP and Cycle checkers. In Figure 5.3b, PolySI fluctuates while maintaining a general linear trend, but our checkers still outperform PolySI, especially in high-rate cases. PolySI's mechanism does not essentially allow early stopping. It needs to construct all graphs to ensure a large search space for the solver to take as input. Also, the concept of an SMT solver has already included the idea of early stopping. Both two points explain the worse performance in Figure 5.3b and the difficulty in further improving the performance. In particular, PolySI claims it outperforms other state-of-the-art solver-based checkers in SI checking, which, in contrast, foregrounds the high cost to ensure a large search space by adhering to the black-box principle.

RQ3.3: alternative implementations with Neo4j As our proof-of-concept implementations can be further extended, Neo4j, as one of the most popular graph databases, can replace ArangoDB as the foundation of our graph-based checker. We identify two algorithms with ready implementations for cycle checking: the cycle checking algorithm in the APOC library (Neo4j-APOC) and the path-based SCC checking algorithm (see Appendix 8) in the GDS library (Neo4j-GDS-SCC). Figures 5.1 and 5.3 report the performance of these two Neo4j-based checkers.

In both figures, Neo4j-GDS-SCC performs consistently better than all the other checkers. This is because the Neo4j-GDS-SCC checker has a lower complexity for its path-based SCC algorithm is an improvement from Tarjan's algorithm. This ensures its checking time is linear with a very low slope, which makes it close to a constant level.

The other checker, Neo4j-APOC, is generally close to but occasionally outperforms ArangoDB-SP in checking time. Both checkers use shortest path algorithms based on BFS and share the easiness and difficulty to find cycles in Figures 5.1a-5.1c and 5.3a-5.3c. However, in Figures 5.1d-5.1e and 5.3d-5.3e, Neo4j-APOC outperforms ArangoDB-SP in PL-2 and PL-1 checking. This is because PL-2 and PL-1 only require a subset of the dependency edges. In this case, the APOC shortest path algorithm can be applied on subgraphs with part of edges reduced, while the ArangoDB SP algorithm has to complete its traversal on the whole graph.

Compared with ArangoDB SP, Neo4j APOC mainly reduces graph density, and further reduces the checking time. This example also gives an insight that multiple graph databases may have different implementations for the same algorithm. It implies that we may use another graph database to overcome difficulties found in the graph database currently being used.

5.7 Summary of Results

In this chapter, we have addressed the three research questions proposed in Chapter 1 regarding the effectiveness, scalability, and comparison of our graph-based checker.

Effectiveness. In general, our checker is effective in both list and register histories. It is effective in checking multiple types of anomalies and determining the correct isolation level for the execution histories.

Scalability. Also, the runtime of cycle detection is scalable with two main factors and three underlying factors in our checker. Behind the correlation with the two main factors used in the Jepsen history (collection time and rate), the checkers are linear with three underlying factors: history length, density, and contributing traversals. Both the Cycle and the Pregel checkers show significant linear trends while the SP checker has more fluctuations.

The three versions of checkers have some drawbacks. The selection of the max depth is important to the performance of the Cycle checker. The checker's runtime increases exponentially with increasing depths. The Pregel checker needs to do a complete traversal on the whole graph and requires some setup to execute the algorithm itself, caused by the BSP framework. The SP checker can quickly identify a cycle when the number of cycles is large; however, when only a few cycles exist in the history, the SP checker tends to spend a longer time traversing the graph and identifying one cycle.

As for additional factors, the nemesis and the number of sessions do not directly affect the growth trends, while the maximum number of write events per object causes a logarithmic increase in the checker's runtime.

Comparison. In comparison with other state-of-the-art checkers, our graph-based checker, built on ArangoDB, has better performance than Elle and PolySI. The Cycle checker has a similar performance to Elle in many cases, but in general, performs better than Elle when the search space is reduced or the rate becomes high. The SP checker also performs better than Elle in SER checking, where cycles are easier to find than at other levels. The PolySI checker compromises its performance while adhering to the black-box principle and inferring with a large search space.

Moreover, Neo4j can also be used to build a graph-based checker, which has a powerful Neo4j-GDS-SCC checker that performs better than all other versions. The Neo4j-APOC checker is close to the SP checker of ArangoDB in performance.

5.8 Discussion

Underlying factors. In fact, the three underlying factors describe the difficulty to find cycles and affect the checker's performance in different ways. Table 5.5 shows the trends related to the history length, density, and the number of contributing traversals. Overall, both history length and density are positively correlated with the checker's runtime, while the number of vertices is negatively correlated with runtime. When history length and density increase, the size of the dependency graph increases by having more vertices or edges. Therefore, in such cases, the runtime becomes longer. However, an increasing number of contributing traversals actually executed during cycle detection is a signifying factor for the occurrences of cycles. With a higher number of contributing traversals, it is easier to detect a cycle, and the traversals do not continue beyond the step where a cycle is found. Therefore, a higher number of contributing traversals actually reduces the difficulty to find cycles and improves the performance by reducing the search space. However, Table 5.6 displays slightly different trends in List3 histories. For example, when the rate is at least 80, the coefficient of the history length (L) becomes negative, marking a negative correlation between the runtime and the increasing rates. In fact, the coefficient of L is not dominant in this trend, with a high p-value that does not indicate significance. This implies that the general trends of the three underlying factors may vary in a special data load.

Max depths. We have also found that the max depth of the Cycle checker has an exponential effect on the performance. Also, the interaction term te^d is strongly significant. Based on this finding, the value of max depth can make changes on the slope of trendlines for collection time. It implies that with a larger depth, the runtime grows more quickly with collection time.

Nemesis. Another finding is that the nemesis does not affect the original trends between the checker's runtime and the collection time. However, the existence of nemesis introduces system faults and disturbs the difficulty to find cycles. We have found that cycles of longer depths often exist in the histories with a nemesis. For histories without any nemesis, shorter cycles are found more easily. In this thesis, we have made an assumption that the maximum depth of cycles is at most 4. However, in practice, when system faults often happen, users may adjust the max depth of the Cycle checker to ensure its functionality in different situations.

Variants of Cycle checker: CycleFilter and CycleRandom. We also consider the two variants proposed in Section 4.1, CycleFilter, and CycleRandom. CycleFilter and CycleRandom perform much better than Cycle in SER checking, where cycles are easy to find. For other levels, however, these two checkers have much worse performance than Cycle. This is because the additional mechanism for early stopping also introduces new costs. The filter-on-path checker requires comparison on the path, which involves operations on the array and increases time complexity. Also, although randomization may increase the chance of better performance, the CycleRandom does not make a breakthrough for the checker's runtime when cycles are difficult to find. Therefore, we stick to the original version of the Cycle checker and ignore the variants. We only included the performance of the original version in the performance plots of this chapter.

Limitations. The experiment setting has its own limitations, mostly caused by the systems in use. The Jepsen history collector requires a self-defined generator to generate transactions and submit them to the database under test. However, this generator needs to be implemented in Clojure, which is the language used by Jepsen. As an example of the functional programming paradigm, and also a part of the Jepsen framework, the generator needs to be handcrafted and the debugging is often difficult for its users. This happens especially when users need to implement new rules rather than directly use the examples provided by Jepsen. In this thesis, we borrow the generators already implemented by Jepsen to generate our list and register histories. However, a new generator will be required if a different format of histories is expected by users.

Also, our checker relies on WAL in collecting and checking register histories. However, in the current development progress, WAL is only supported in the single-machine mode. Developers claim that they will launch a new feature to implement the WAL collection in cluster mode in 2024. It causes the limitation of our checker that it cannot collect the WAL within a cluster at present. Therefore, we are not able to verify that our checker can successfully construct dependency graphs based on such histories. However, we have shown that the difference of histories in the single-machine mode and the cluster mode lies in the type of anomalies. This only affects the effectiveness rather than the performance of the checker. For example, List1 has anomalies of lost updates, which are collected in the cluster mode with sharding and replication. On the other hand, Reg1 does not have such anomalies. Then, when receiving these two sets of histories as input, the checker needs sufficient specificity to identify the correct isolation level. However, the procedures for checking anti-patterns remain the same. Different types of histories do not change the effects of the three underlying factors: the history length, density, and the number of contributing traversals actually executed to find cycles.

Furthermore, we did not manage to include more representative solver-based checkers. Most checkers have designed ways of generating new benchmarks tailored to their systems. However, it does not necessarily support our history format. For example, we have encountered difficulties in converting our Jepsen histories to the format that can be received by Cobra [60] or DBCop [20]. Viper [69] claims it handles Jepsen histories. However, Viper does not show sufficient effectiveness in checking our List1 histories. The reason behind the issue is not clear, either because of the format conversion or because of the internal mechanism of the checker. PolySI is the one with good support to handle Jepsen histories. However, it may still break the list histories due to its flawed history converter. Therefore, we designed our own history converter to transform the Jepsen format to PolySI's text format, in order to shape the correct trend for PolySI's execution performance.

Complexities. Another finding is that although the theoretical growth trend is polynomial for the Cycle checker with respect to the number of vertices, most plots in this chapter show linear trends. We do not reject the claim that our checker is possible to achieve a polynomial trend when the collection time or rate increases to even higher. However, since we have varied the parameters and the collected histories are already enough to detect isolation anomalies, which proves effectiveness, we still claim that the performance of our graph-based checkers is linear *on our histories*, especially with regard to the three underlying factors.

Summary. Overall, our experiments are successful in presenting the effectiveness and scalability of our graph-based checker. Our experiments can be further strengthened when more generators are designed, WAL is supported on clusters, or more format-friendly solver-based checkers are present.

Chapter 6

Related Work

In this chapter, we present related work about the development history of isolation level definitions, the current trend of graph databases, and a summary of state-of-the-art isolation level checkers.

6.1 Isolation Levels: A Brief History

The idea of isolation level hierarchy was first introduced under the name *Degrees of Consistency* with an accompanying locking protocol [35]. This work informally defined four degrees (Degrees 0-3). Each degree specified several conceptual *phenomena* that should not be seen by the transactions; also, it clearly stated how developers should set the database locks to achieve each degree. This hierarchy is preventative [13]: it defines each degree of consistency by disallowing a set of phenomena, especially when the degree goes weaker. This preventative approach is the foundation of the later progress, for example, the ANSI standard.

The ANSI/ISO SQL-92 specifications extended the previous degrees to the four isolation levels of SQL transactions: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE. This new standard aimed to be implementation-independent and more inclusive than just locking-based implementations. However, the standard proposed conceptual phenomena similar to the previous work, and a subsequent paper titled *A Critique of ANSI SQL Isolation Levels* [19] pointed out that the definitions of the four levels from the ANSI standard were essentially identical to the previous locking-based definitions. [19] further explored and found that it would be difficult for databases without locking mechanisms to implement such definitions, and therefore the ANSI standard failed to achieve its goal of implementation-independence. Meanwhile, the paper introduced some formalism, restating the previous conceptual phenomena into lower-level orderings of operations. It also proposed several adjustments against potential problems of the ANSI standard. For example, the paper highlighted the drawbacks of the REPEATABLE READ level for its broad range and ambiguous naming. Two additional levels, *Cursor Stability* and *Snapshot Isolation* were mentioned as examples to fill the gap between the two levels: READ COMMITTED and REPEATABLE READ. [19] proposed a rich set of phenomena and the major four of them can be expressed in the following order of events.

P0: $w_1[x]$ ... $w_2[x]$ ... (c_1 or a_1)	(Dirty Write)
P1: $w_1[x]$ ... $r_2[x]$ ... (c_1 or a_1)	(Dirty Read)
P2: $r_1[x]$ ... $w_2[x]$ ... (c_1 or a_1)	(Fuzzy Read / Non-Repeatable Read)
P3: $r_1[P]$ ... $w_2[y$ in $P]$ (c_1 or a_1)	(Phantom)

[19] also proposed other phenomena, examples of which included the following.

P4: $r_1[x]$ ... $w_2[x]$ ... $w_1[x]$ ... c_1	(Lost Update)
P4C: $rc_1[x]$ ... $w_2[x]$ ... $w_1[x]$ ... c_1	(Lost Update - Cursor Version)
A5A: $r_1[x]$ ... $w_2[x]$ ... $w_2[y]$ ... c_2 ... $r_1[y]$ ... (c_1 or a_1)	(Read Skew)
A5B: $r_1[x]$ ... $r_2[y]$ ... $w_1[y]$ ... $w_2[x]$ ... ($c1$ and $c2$ occur)	(Write Skew)

These shorthand notations of events shape the concurrency between two transactions T_1 and T_2. For example, $w_1[x]$ means an event of the transaction T_1 writes to object x; $r_2[x]$ means an event of T_2 reads from object x. Especially, c_1 or a_1 means T_1 is committed or aborted, respectively. Some notations are used only within a certain phenomenon, such as P (predicate) in Phantom and rc (cursor version of read) in Lost Update - Cursor Version. More details can be found in [19].

Consistency Level = Locking Isolation Level	Prevented Phenomena
Degree 0	none
Degree 1 = Locking READ UNCOMMITTED	**P0**
Degree 2 = Locking READ COMITTED	**P0**, **P1**
Cursor Stability	**P0**, **P1**, **P4C**
Locking REPEATABLE READ	**P0**, **P1**, **P2**, **P4**, **P4C**, **A5A**, **A5B**
Snapshot Isolation	**P0**, **P1**, **P2**, **P4**, **P4C**, **A5A**
Degree 3 = Locking SERIALIZABILITY	**P0**, **P1**, **P2**, **P3**, **P4**, **P4C**, **A5A**, **A5B**

Table 6.1: Isolation Levels and Prevented Anti-Patterns [19]

These phenomena served as an extension of the ANSI standard that covers only **P1**, **P2**, and **P3**. The preventative approach was gradually completed and more weaker isolation levels could be described by preventing a certain subset of phenomena (shown in 6.1). In addition, researchers began their exploration of optimistic and multi-version systems besides the pessimistic locking mechanism. [19] pointed out Snapshot Isolation as a form of multi-version concurrency control (MVCC): it is an optimistic way of control that allows multiple versions to be observed at the same time. For example, in H_1, multiple objects (x and y) have different versions, which is not the case only a single value and a single object are observed.

H_1: $r_1[x_0 = 5]$ $w_1[x_1 = 1]$ $r_2[x_0 = 5]$ $r_2[y_0 = 5]$ c_2 $r_1[y_0 = 5]$ $w_1[y_1 = 9]$ c_1

However, even with this extension of the ANSI standard, the research community did not substantially upgrade the definition framework, since the pessimistic locking mechanism remained to be an essential part of the preventative definitions, and therefore the framework was not implementation-independent. Furthermore, the preventative approach tends to be "overly restrictive", because any phenomenon that fell into the target category would be prevented in a *pessimistic* way, and it was ignored that certain phenomena disallowed by the system may still be valid in some use cases [13].

To address the weaknesses brought by the locking mechanism, Adya et al. extended the ANSI standard by formulating isolation levels based on the idea of dependency graphs [13]. A dependency graph maintains the relations between transactions with vertices and edges, and also reshapes the phenomena from sequential orderings to subgraphs, usually containing a certain type of cycles. Accordingly, new isolation levels were defined in [13] based on subgraphs and cycles. As the construction and interpretation of dependency graphs is not relevant of the locking mechanism, the Adya's formalism achieves implementation-independence, and the definitions can be generalized to both pessimistic and optimistic (and MVCC) cases.

In a more formal way, Cerone et al. proposed an axiomatic framework to define isolation levels and phenomena on an abstract level [24] and later proved its equivalence to Adya's definitions [25]. Cerone et al.'s formalism abstracts Adya's graph visualization to algebraic definitions which are rigorous, uniform, and declarative.

6.2 The Trend of Graph Databases

Graph databases have gained popularity and are increasingly used in a variety of applications. The article [54] presented by the academic communities of computer systems and data management points out the nature of graphs as "unifying abstractions that can leverage interconnectedness". This means that relationships between data records are just as important as the records themselves, with edges between vertices being highly valued in graph databases as "the first-class citizens". Additionally, compared with relation models, the difference is that graph databases have native graph models. They also allow for more flexibility in defining schemas, by supporting all or some of the schema-last, schema-flexible, or schema-less features [15]. It provides more flexibility to incorporate new features, especially as agile practices become more prevalent in industries [52]. By utilizing graphs for analysis, businesses can gain novel insights and statistics about edges, paths, and cycles, thereby improving their operations. Because of these advantages, graph databases are increasingly being used in industries that place a high value on the connections between data points, such as healthcare knowledge graphs [55], fraud analytics in finance [43] [36], and supply chain management in logistics [37], among others. Furthermore, Gartner, Inc. [12] predicts that by 2025, 80% of graph technologies will be utilized in the field of data analytics in enterprises and industries.

As academia and industries maintain their interest, graph databases are exploring new possibilities to expand their range of use cases, particularly in this era where the usage of machine learning and artificial intelligence (ML/AI) has proven to be effective in solving problems [46] [68]. The article [54] defines the future development possibilities in three dimensions: *abstractions* (data models, query formalisms, and graph algebra), *ecosystems* (standardization, scalability, and streaming), and *performance* (benchmarks, specialization vs. portability, and archiving).

Being part of the trends, the emergence and continuous growth of distributed graph databases have become an underlying assumption for big graphs with high data volume and velocity. Also, the community is expecting a standardized approach for big graph processing systems, which involves a distributed reference architecture. A distributed graph database extends the capabilities of a graph database by allowing data to be partitioned and distributed across multiple nodes, enabling horizontal scalability while maintaining strong internal consistency (or high availability) and fault tolerance. Data can be partitioned based on node properties or relationships, and distributed queries can be executed in parallel across the nodes to improve performance. Additionally, a distributed graph database can provide replication and failover capabilities to ensure data availability and reliability.

6.3 Isolation Level Checking

In Section 2.5, we have stated the black-box isolation checking problem. In this section, we divide isolation level checkers into two main categories: *Elle-style* checkers and *solver-based* checkers. Elle-style checkers use a single dependency graph to identify any anomaly that leads to the violation of a certain isolation level, while solver-based checkers combine BC-polygraphs (or occasionally other graph structures) and SAT/SMT solvers to filter a total order that does not cause anomalies. Also, we give an overview of various isolation checkers that are representative of these two categories.

6.3.1 Solver-Based Checkers: Linearizability, Serializability, and SI checking

The solver-based approach often involves combinatorial enumeration. Due to the intractability caused by NP-completeness, the traditional way of checking linearizability (e.g., Knossos [40] and Porcupine [16]) [65, 45] or serializability (e.g., Gretchen [41]) [50] followed the solver-based approach and caused a combinatorial explosion of the state space. Linearizability is equivalent to strict serializability if each event is viewed as an independent transaction. In this setting, linearizability requires a total order on the events shown in the history. Knossos [40] is a black-box isolation checker for linearizability, which adopts the checking method by [65] and the optimization by [45]. Knossos uses enumeration to search for one valid sequence of events from all possible sequences, using simple linear data structures or tree-based structures. However, both approaches require enumeration and are limited by NP-completeness. Another linearizability checker, Porcupine ([16]), also suffers the NP-complete nature of the linearizability checking problem. As a serializability checker, Gretchen [41] follows Cerone's specifications [24] and encodes the execution histories as constraints. After that, Gretchen uses Gecode [62] to explore a valid total order with the encoded constraints.

In addition to Gretchen [41], many other solver-based checkers are also applicable to serializability checking [57, 60, 20]. The checker [57] uses SMT-based and dynamic partial order reduction-based (DPOR-based) approaches to predict whether the history contains violations against serializability. Cobra [60] and DBCop [20] design different graph structures and combine the usage of the MonoSAT solver to check anomalies of serializability. Cobra uses the polygraph structure. It enumerates all possible dependency graphs, stacks the graphs together, and searches possibilities without anomalies, in order to generate a valid total order from the polygraph that satisfies serializability. DBCop, on the other hand, use undirected graphs and history transforming, to check serializability within polynomial time, with the assumption that the number of sessions is fixed.

In addition to serializability checking, with solver-based checkers, snapshot isolation (SI) is representative as it has a clear cyclic characteristic, and it is more complex than serializability checking. DBCop [20] proposes a way to create an auxiliary history to reduce the SI checking to serializability checking, which is still polynomial time with a fixed number of sessions. By extending Cobra, Viper [69] continues using polygraphs and SMT solvers with new constraints designed for SI checking. To the best of our knowledge, PolySI [38] has the best performance among solver-based SI checkers. It inherits Cobra's data structures and proposes refined pruning conditions to ensure a linear complexity for SI checking. Overall, the NP-completeness and its reduction are challenging for serializability and SI checking with solver-based checkers.

6.3.2 Elle-Style Checkers

Elle-style checkers (e.g., [42], [26]), on the other hand, directly infer version orders from known information and construct dependency edges within a single dependency graph. In this way, the combinatorial enumeration is reduced, and the checking is reduced from NP-completeness to polynomial complexity. This complexity applies to the checking of all levels because of the mechanism of Elle-style checkers. For example, for any level, Elle executes Tarjan's SCC algorithm to retrieve all the SCCs present in the dependency graph. Then, Elle determines the isolation level not violated by the execution histories based on the SCC results. This mechanism ensures that the execution procedures are the same for all isolation levels. It implies that the Elle-style checkers are stable across different levels, but on the other hand, do not allow level-specific optimizations.

6.3.3 Checkers of PSI and Other Levels

The checking of other isolation levels is less explored in the literature. Elle does not completely fulfill the goal of checking PSI and other levels specified by [24]. Some solver-based checkers claim that they cannot be directly transferred to isolation levels (e.g., PSI) [69]. In this thesis, we have filled in the gap for the checking of PSI, PL-2, and PL-1 levels.

Chapter 7

Conclusions and Future Work

In this chapter, we conclude our thesis with the conclusions. We successfully designed a novel isolation checker in ArangoDB. Also, we did not find any anti-patterns that violate ArangoDB's isolation level guarantee, LOCAL SNAPSHOT ISOLATION, by applying our checker to both single-instance and cluster modes of ArangoDB. Moreover, we state other contributions related to graph queries, in-database implementation, experimental evaluation, and smart visualization. Finally, we describe the potential future work of the thesis.

7.1 Conclusions

Through this thesis, *we designed and evaluated a novel isolation level checker with an implementation on top of the ArangoDB graph database*. We confirmed the effectiveness of the checker by comparing the checking results with Elle on list histories. Furthermore, we utilized a series of graph queries for efficient isolation checking, and our checker is comparable with other state-of-the-art checkers in terms of performance. Also, we proposed a way to collect benchmarking datasets along with WAL, which provides more version information than traditional black-box isolation checkers. We also proposed a novel way to visualize the detected graph-based anti-patterns directly within graph databases.

In general, ArangoDB achieves its isolation level guarantee of LOCAL SNAPSHOT ISOLATION. In the single-instance mode, ArangoDB achieves snapshot isolation (SI). In the cluster mode, it still achieves SI when sharding and replication are disabled; otherwise, ArangoDB achieves PL-2 level when sharding and replication are enabled.

7.2 Contributions

We extended the use cases of graph database queries to isolation level checking and confirmed its feasibility. In Chapter 4, the graph traversal and K shortest path queries are suitable for the implementation of anti-pattern detection across all levels. ArangoDB also provides a Pregel SCC algorithm, which is only applicable to serializability checking.

We provided an in-database implementation, with a history collection stage in ArangoDB as a distributed database, and a cycle detection stage in ArangoDB as a graph database.

The graph construction stage in between is also highly interactive with ArangoDB as a graph database. The whole workflow has the potential to be integrated as a fully in-database implementation.

We conducted experiments to extensively evaluate the efficiency and scalability of our graph-based checker. In Section 5.5, we found that the two main factors used in the Jepsen history collector, collection time and rate, were correlated with the execution time but the relationship was not consistent in all cases. We further analyzed three underlying factors (history length, graph density, and the number of contributing traversals) to establish a linear development trend between the execution time of the checker and the three factors. We also investigated two additional factors, the number of sessions and the maximum number of write operations per object. The number of sessions did not directly affect the runtime, while the maximum number of write operations had a logarithmic effect on the runtime based on our analysis. Furthermore, the nemesis did not change the development trend with increasing collection times. However, the maximum depth of the Cycle checker had an exponential effect on the runtime, which also affected the slope of the linearity between the checker runtime and history collection time.

We also proposed a novel way to smartly visualize the detected graph anti-patterns in graph databases. In Section 4.4, ArangoDB was able to present the vertices but missed edge information in its Web Interface. Neo4j Browser User Interface, on the other hand, provided sufficient information for us to analyze the anti-pattern caused by the history. Furthermore, graph databases bridge the data and information with a rich ecosystem of business intelligence and analytics software products. This further empowered the smart visualization process.

7.3 Future work

In this thesis, we have focused on two types of histories: list histories and register histories. Our graph-based checker can be further extended to other types of histories. For example, [64] proposes various workloads and read/write operation combinations in the context of graphs, which can be applied to our checker. However, these workloads usually have different formats in their execution histories. Some align with the format of our list or register histories in the thesis, for example, as shown in Listing 1.18 in [64]. However, if the execution histories have a different format, then we need to create new strategies of graph construction (especially `getEvtDepEdges`) to successfully convert the histories to the graph structure. We also require a self-defined, adaptable generator to create such workloads.

Furthermore, we have migrated our checker from ArangoDB to Neo4j. This process can be further extended to another candidate graph database. However, since each graph database supports a different set of operations when using a new graph database, the old queries in the previous database may not have their counterparts in the new database. Therefore, we should find and customize available queries to build the new checker.

As we have presented an in-database implementation, the graph-based checker can be fully migrated within distributed graph databases. The graph database community may design a new mode in ArangoDB for the purpose of internal verification. This new mode randomly generates concurrent transactions, maintains a dependency graph, and performs cycle detection all within ArangoDB. The database can also retrieve the WAL internally to reduce I/O costs. This mode can also start the verification process in multiple epochs and iterations. Different from the implementation presented in the thesis, the process of dependency graph construction can also be embedded in the database system, in the language that is used to build the system (e.g. C++ for ArangoDB). In this way, the verification becomes fully automated, and no external software is required.

As an isolation checker with a practical implementation, we can also extend the usage to other databases, and other isolation levels. The flexibility of graph queries allows us to cover more cyclic anti-patterns without effort. It also allows refinement of the isolation hierarchy, in order to better differentiate the levels with a relatively large gap (e.g., between PL-2 and PSI). Furthermore, we can explore non-graph anomalies as well as the way to use AQL queries or simple database operations to detect such anomalies.

Bibliography

[1] *ArangoDB Documentation v3.9.* . URL `https://www.arangodb.com/docs/3.9/`.

[2] *ArangoDB.* . URL `https://www.arangodb.com/`.

[3] *Graphviz.* URL `https://graphviz.org/`.

[4] *Memgraph.* URL `https://memgraph.com/`.

[5] *Neo4j.* URL `https://neo4j.com/`.

[6] *Using Cytoscape with ArangoDB.* . URL `https://www.arangodb.com/learn/graphs/using-arangodb-with-cytoscape/`.

[7] *Visualizing ArangoDB with KeyLines.* . URL `https://cambridge-intelligence.com/keylines/arangodb/`.

[8] *ANSI X3.135-1992, American National Standard for Information Systems - Database Language - SQL.* Nov 1992.

[9] *Introducing Gelly: Graph Processing with Apache Flink.* Aug 2015. URL `https://flink.apache.org/2015/08/24/introducing-gelly-graph-processing-with-apache-flink/`.

[10] *Transaction Isolation Levels.* Jun 2023. URL `https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html`.

[11] Daniel Abadi. *Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story*, volume 45, pages 37–42. 2012. doi: 10.1109/MC.2012.33.

[12] Merv Adrian and Afraz Jaffri. *Market guide for graph database management systems.* Aug 2022. URL `https://www.gartner.com/en/documents/4018220`.

[13] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. Generalized isolation level definitions. In David B. Lomet and Gerhard Weikum, editors, *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, pages 67–78. IEEE Computer Society, 2000. doi: 10.1109/ICDE.2000.839388. URL `https://doi.org/10.1109/ICDE.2000.839388`.

[14] Renzo Angles. The property graph database model. In *AMW*, volume 2100 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018.

[15] Renzo Angles, Angela Bonifati, and et al. *PG-Schema: Schemas for Property Graphs*, volume abs/2211.10962. 2022.

[16] Anish Athalye. *Porcupine*. 2017-2018. `https://github.com/anishathalye/porcupine`.

[17] Ching Avery. *Giraph: Large-scale graph processing infrastructure on hadoop*, volume 11, pages 5–9. 2011.

[18] Bradley R. Bebee, Daniel Choi, Ankit Gupta, Andi Gutmans, Ankesh Khandelwal, Yigit Kiran, Sainath Mallidi, Bruce McGaughy, Michael Personick, K. Jeric Rajan, Simone Rondelli, Alexander Ryazanov, Michael Schmidt, Kunal Sengupta, Bryan B. Thompson, Divij Vaidya, and Shawn Xiong Wang. Amazon neptune: Graph data management in the cloud. In *International Workshop on the Semantic Web*, 2018.

[19] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, page 1–10, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897917316. doi: 10.1145/223784.223785.

[20] Ranadeep Biswas and Constantin Enea. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.*, 3(OOPSLA):165:1–165:28, 2019. doi: 10.1145/3360591. URL `https://doi.org/10.1145/3360591`.

[21] Angela Bonifati and Stefania Dumbrava. *Graph Queries: From Theory to Practice*, volume 47. 12 2018. doi: 10.1145/3335409.3335411.

[22] Angela Bonifati, G.H.L. Fletcher, Hannes Voigt, and N. Yakovets. *Querying graphs*. Morgan Claypool Publishers, 2018. doi: 10.2200/S00873ED1V01Y201808DTM051.

[23] Andrea Cerone and Alexey Gotsman. Analysing snapshot isolation. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, page 55–64, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339643. doi: 10.1145/2933057.2933096.

[24] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. A framework for transactional consistency models with atomic visibility. In Luca Aceto and David de Frutos-Escrig, editors, *26th International Conference on Concurrency Theory, CONCUR*

2015, Madrid, Spain, September 1.4, 2015, volume 42 of *LIPIcs*, pages 58–71. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi: 10.4230/LIPIcs.CONCUR.2015.58. URL `https://doi.org/10.4230/LIPIcs.CONCUR.2015.58`.

[25] Andrea Cerone, Alexey Gotsman, and Hongseok Yang. *Algebraic Laws for Weak Consistency*. 2017.

[26] Jack Clark. *Verifying Serializability Protocols With Version Order Recovery*. ETH Zurich, Zurich, 2021. doi: 10.3929/ethz-b-000507577.

[27] Thomas H. Cormen, Charles Eric Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*, page 573–574. The MIT Press, 4th edition, 2022.

[28] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. Seeing is believing: A client-centric specification of database isolation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, page 73–82, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349925. doi: 10.1145/3087801.3087802. URL `https://doi.org/10.1145/3087801.3087802`.

[29] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. *TigerGraph: A Native MPP Graph Database*. 2019.

[30] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall series in automatic computation. Prentice-Hall, 1976. ISBN 0-13-215871-X.

[31] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. *RocksDB: Evolution of Development Priorities in a Key-Value Store Serving Large-Scale Applications*, volume 17. Association for Computing Machinery, New York, NY, USA, Oct 2021. doi: 10.1145/3483840.

[32] Diogo Fernandes and Jorge Bernardino. Graph databases comparison: Allegrograph, arangodb, infinitegraph, neo4j, and orientdb. In *Proceedings of the 7th International Conference on Data Science, Technology and Applications*, DATA 2018, page 373–380, Setubal, PRT, 2018. SCITEPRESS - Science and Technology Publications, Lda. ISBN 9789897583186. doi: 10.5220/0006910203730380.

[33] Hal Gabow. *History of Path-based DFS for Strong Components*. URL `https://home.cs.colorado.edu/~hal/Papers/DFS/pbDFShistory.html`.

[34] Harold N. Gabow. *Path-based depth-first search for strong and biconnected components*, volume 74, pages 107–114. 2000. doi: https://doi.org/10.1016/S0020-0190(00)00051-X.

[35] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. *Granularity of Locks and Degrees of Consistency in a Shared Data Base*. Elsevier North-Holland, Amsterdam, 1976.

[36] Amy Hodler. *Financial Fraud Detection with Graph Data Science - Neo4j*. Apr 2020. URL `https://go.neo4j.com/rs/710-RRC-335/images/Neo4j-Financial-Fraud-Detection-GDS-white-paper-EN-US.pdf`.

[37] Young-Chae Hong and Jing Chen. *Graph database to enhance supply chain resilience for industry 4.0*, volume 15, page 1–19. 2021. doi: 10.4018/ijisscm.2022010104.

[38] Kaile Huang, Si Liu, Zhenge Chen, Hengfeng Wei, David A. Basin, Haixiang Li, and Anqun Pan. Efficient black-box checking of snapshot isolation in databases. *Proc. VLDB Endow.*, 16(6):1264–1276, 2023. URL `https://www.vldb.org/pvldb/vol16/p1264-wei.pdf`.

[39] Peter Jipsen, Chris Brink, and Gunther Schmidt. *Background Material*, pages 1–21. Springer Vienna, Vienna, 1997. ISBN 978-3-7091-6510-2. doi: 10.1007/978-3-7091-6510-2_1.

[40] Kyle Kingsbury. *Knossos*. 2013-2023. `https://github.com/jepsen-io/knossos/`.

[41] Kyle Kingsbury. *Gretchen*. 2016. `https://github.com/aphyr/gretchen`.

[42] Kyle Kingsbury and Peter Alvaro. Elle: Inferring isolation anomalies from experimental observations. volume 14, page 268–280. VLDB Endowment, Nov 2020. doi: 10.14778/3430915.3430918. URL `https://doi.org/10.14778/3430915.3430918`.

[43] E. Kurshan, H. Shen, and H. Yu. *Financial Crime Fraud Detection Using Graph Computing: Application Considerations Outlook*. 2021.

[44] Kit Patella Kyle Kingsbury. *Jepsen*. 2013-2023. `http://jepsen.io/`.

[45] Gavin Lowe. Testing for linearizability. *Concurrency and Computation: Practice and Experience*, 29, 2017.

[46] Xiaoxiao Ma, Jia Wu, Shan Xue, Jian Yang, Chuan Zhou, Quan Z. Sheng, Hui Xiong, and Leman Akoglu. *A Comprehensive Survey on Graph Anomaly Detection with Deep Learning*, pages 1–1. Institute of Electrical and Electronics Engineers (IEEE), 2021. doi: 10.1109/tkde.2021.3118815.

[47] David Maier, Jacob Stein, Allen Otis, and Alan Purdy. Development of an object-oriented dbms. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '86, page 472–482, New York, NY, USA, 1986. Association for Computing Machinery. ISBN 0897912047. doi: 10.1145/28697.28746. URL `https://doi.org/10.1145/28697.28746`.

[48] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 135–146, New York, NY,

USA, 2010. Association for Computing Machinery. ISBN 9781450300322. doi: 10.1145/1807167.1807184.

[49] Konstanitnos Mavrogiorgos, Athanasios Kiourtis, Argyro Mavrogiorgou, and Dimosthenis Kyriazis. A comparative study of mongodb, arangodb and couchdb for big data storage. In *Proceedings of the 2021 5th International Conference on Cloud and Big Data Computing*, ICCBDC '21, page 8–14, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450390408. doi: 10.1145/3481646.3481648.

[50] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, oct 1979. ISSN 0004-5411. doi: 10.1145/322154.322158. URL `https://doi.org/10.1145/322154.322158`.

[51] Daniel Ritter, Luigi Dell'Aquila, Andrii Lomakin, and Emanuele Tagliaferri. Orientdb: A nosql, open source MMDMS. In *Proceedings of the The British International Conference on Databases 2021, London, United Kingdom, March 28, 2022*, volume 3163 of *CEUR Workshop Proceedings*, pages 10–19. CEUR-WS.org, 2021.

[52] Ian Robinson, Emil Eifrem, and James Webber. *Graph databases: New opportunities for connected data*. O'Reilly Media, 2015.

[53] Rodrigo Rocha and Bhalchandra Thatte. *Distributed cycle detection in large-scale sparse graphs*. Aug 2015. doi: 10.13140/RG.2.1.1233.8640.

[54] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, and the computer systems and data management communities. *The Future is Big Graphs: A Community View on Graph Processing Systems*, volume 64, page 62–71. Association for Computing Machinery, New York, NY, USA, Aug 2021. doi: 10.1145/3434642.

[55] Jero Schäfer, Ming Tang, Danny Luu, Anke Bergmann, and Lena Wiese. *Graph4Med: a web application and a graph database for visualizing and analyzing medical databases*, volume 23. Dec 2022. doi: 10.1186/s12859-022-05092-0.

[56] M. Sharir. *A strong-connectivity algorithm and its applications in data flow analysis*, volume 7, pages 67–72. 1981. doi: https://doi.org/10.1016/0898-1221(81)90008-0.

[57] Arnab Sinha, Sharad Malik, Chao Wang, and Aarti Gupta. Predicting serializability violations: Smt-based search vs. dpor-based search. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*, volume 7261 of *Lecture Notes in Computer Science*, pages 95–114. Springer, 2011. doi: 10.1007/978-3-642-34188-5_11. URL `https://doi.org/10.1007/978-3-642-34188-5_11`.

[58] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 385–400, New York, NY,

USA, 2011. Association for Computing Machinery. ISBN 9781450309776. doi: 10.1145/2043556.2043592.

[59] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. The ldbc social network benchmark: Business intelligence workload. *Proc. VLDB Endow.*, 16(4):877–890, dec 2022. ISSN 2150-8097. doi: 10.14778/3574245.3574270. URL `https://doi.org/10.14778/3574245.3574270`.

[60] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. Cobra: Making transactional key-value stores verifiably serializable. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 63–80. USENIX Association, 2020. URL `https://www.usenix.org/conference/osdi20/presentation/tan`.

[61] Robert Tarjan. *Depth-First Search and Linear Graph Algorithms*, volume 1, pages 146–160. 1972. doi: 10.1137/0201010.

[62] Gecode Team. *Gecode: Generic Constraint Development Environment*. 2005. `http://www.gecode.org`.

[63] Leslie G. Valiant. *A Bridging Model for Parallel Computation*, volume 33, page 103–111. Association for Computing Machinery, New York, NY, USA, Aug 1990. doi: 10.1145/79173.79181.

[64] Jack Waudby, Benjamin A. Steer, Karim Karimov, József Marton, Peter A. Boncz, and Gábor Szárnyas. Towards testing ACID compliance in the LDBC social network benchmark. In Raghunath Nambiar and Meikel Poess, editors, *Performance Evaluation and Benchmarking - 12th TPC Technology Conference, TPCTC 2020, Tokyo, Japan, August 31, 2020, Revised Selected Papers*, volume 12752 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2020. doi: 10.1007/978-3-030-84924-5_1. URL `https://doi.org/10.1007/978-3-030-84924-5_1`.

[65] J.M. Wing and C. Gong. Testing and verifying concurrent objects. *Journal of Parallel and Distributed Computing*, 17(1):164–182, 1993. ISSN 0743-7315. doi: https://doi.org/10.1006/jpdc.1993.1015. URL `https://www.sciencedirect.com/science/article/pii/S0743731583710154`.

[66] Reynold S. Xin, Daniel Crankshaw, Ankur Dave, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. *GraphX: Unifying Data-Parallel and Graph-Parallel Analytics*. 2014.

[67] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. Pregel algorithms for graph connectivity problems with performance guarantees. volume 7, page 1821–1832. VLDB Endowment, Oct 2014. doi: 10.14778/2733085.2733089.

[68] Hang Yin, Zitao Zhang, Zhurong Wang, Yilmazcan Ozyurt, Weiming Liang, Wenyu Dong, Yang Zhao, and Yinan Shan. *Behavioral graph fraud detection in E-commerce*. 2022.

[69] Jian Zhang, Ye Ji, Shuai Mu, and Cheng Tan. Viper: A fast snapshot isolation checker. In Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan, editors, *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, pages 654–671. ACM, 2023. doi: 10.1145/3552326.3567492. URL `https://doi.org/10.1145/3552326.3567492`.

Appendix A

An Overview of ArangoDB v3.9

In this thesis, we have selected ArangoDB, which is both a distributed database and graph database, to be both the database under test and the graph database used for cycle detection. ArangoDB is a prime example of an open-source, multi-model database that supports multiple data models, including key-value pairs, JSON documents, and graphs. It is a flexible, adaptable, and scalable database system that can handle complex and diverse use cases, especially those related to graphs. In this appendix, we present an overview of ArangoDB version 3.9 (Community Edition), including its concepts, architecture, and background.

A.1 Data Model

If ArangoDB is perceived as a general key-value storage, data is organized in a hierarchy consisting of databases, collections, and documents. A *database* in ArangoDB is the top-level container that stores zero or more collections. Each database has its own security settings, users, and permissions. There may exist one or more databases, with one default database *_system* that cannot be dropped.

The other two concepts lower in the hierarchy, collections and documents, are similar to tables and rows in the relational database terminologies, respectively. A *collection* is a group of zero or more documents and is uniquely identified by its name. Besides the document collection, edges can also be stored in a collection and such a collection is called an edge collection. Within a collection in ArangoDB, a *document* is a single record that is typically structured as a JSON-like object, which can include nested objects and arrays. Just like the rows in relational databases, a document has its *attribute* keys and values. However, ArangoDB differs from relational databases in that it is schema-less, meaning that it does not require the schema of a collection to be pre-defined and strictly adhered to; in contrast, ArangoDB permits documents with different attributes to be present in the same collection. A document has at least three attributes (*_id, _key* and *_rev*) that serve as identifiers, and zero or more other attributes that store the data. The value (*field*) of the *_key* attribute is called the *document key*, which is a string value specified by the user when the document is newly inserted into the collection. Every document is required to have a key. In some cases where the user decides not to assign a key for a document, an automatically generated key will be attached to that document. The *_id* field is the *document handle* in the form of "*collection_name/document_key*" that serves as a unique identifier of the document across the database. The *_rev* field is the *document revision* that specifies the versioning of a document and is maintained by ArangoDB automatically.

Based on the general data model hierarchy, the graph structure in ArangoDB is defined with a combination of document collections and edge collections, where the two types of collections contain the *vertices* and *edges* of the graph, respectively. In the edge collection, each edge is directed and identified by two vertices marking the start and the end of that edge. The document handles of the two vertices are stored in the *_from* and *_to* attributes of the edge, and the direction of the edge is $\_from \rightarrow \_to$. In some rare cases, two edges of one graph can be used as vertices to form a new edge of another "edge" graph.

A.2 Cluster Architecture

A.2.1 Default Setup

An ArangoDB Cluster can be created by ArangoDB Starter (*arangodb*) or set up manually with ArangoDB Server (*arangod*). Compared with the other way, ArangoDB Starter is a simplified setup tool that sticks to the default setting of a Cluster. The default setting will be detailed in the following paragraph.

An ArangoDB Cluster is a set of ArangoDB instances that are connected to each other and form a network. It is common, yet not the actual image of ArangoDB, that people relate a Cluster to a set of connected machines. In ArangoDB, however, a Cluster is structured by instance with one of the three roles: *Agents*, *Coordinators* and *DB-Servers*, and these instances are grouped in any form across the available machines. One can set up more Coordinators than DB-Servers, or the other way around, but a hidden rule is widely applied that "exactly one Coordinator and one DB-Server are run on each machine to achieve the classical master/master setup". In the default setting, a stricter requirement is that exactly one Agent runs on each machine, which is true in the practice of ArangoDB Starter (*arangodb*) though not compulsory according to their documentations.

Therefore, by default, one machine consists of exactly one Agent, one Coordinator and one DB-Server. Consider a Cluster of N machines. The N Agents of the machines form the *Agency* of the Cluster. The Agency controls the essential configuration and regulates the behaviors of the whole Cluster (for example, leader election and synchronization), so fault tolerance is a requirement of the Agency. To ensure that, *Raft Consensus Algorithm* is used among the Agents to keep the Agency alive throughout the life of the Cluster. The Coordinators are stateless and receive the requests (for example, queries) from the clients and transfer the requests to the DB-Servers. The DB-Servers store the data and handle the requests transferred by Coordinators. [1]

[1]Cluster Architecture (ArangoDB 3.9): https://www.arangodb.com/docs/3.9/architecture-deployment-modes-cluster-architecture.html

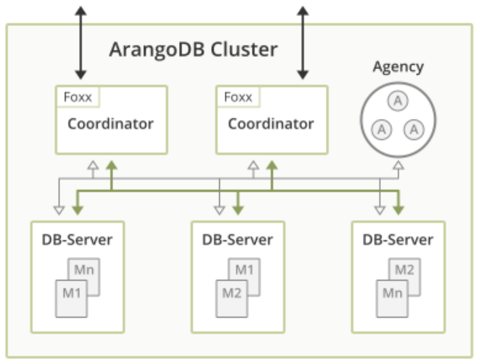


Figure A.1: Topology of an ArangoDB cluster [1]

A.2.2 Sharding

Sharding and replication are two important features of ArangoDB. [2] Sharding is a technique used for horizontal scaling in which the data is partitioned and distributed across multiple servers or nodes. Sharding is based on collections in ArangoDB; a data collection can be distributed among the nodes based on a predefined shard key with consistent hashing, which can be a specific attribute in the documents contained in that collection.

Sharding is not enabled by default. The user needs to configure two optional properties, *numberOfShards* and *shardKeys*. By default, the number of shards is 1; the shard keys only contain the attribute *_key*, which is the identifying key for each document contained in the collection. The default mode implies that there will be no sharding if the two properties are not specified, and the whole collection will be stored on the single server found by the hashed result of *_key*. The example below creates a collection named *col*, using 4 shards among the DB-Servers and the attribute *country* as one Shard Key. This would speed up queries reading data of the same country, as these documents are supposed to be placed within the same shard.

```
db._create("col", {"numberOfShards": 4, "shardKeys": ["country"]});
```

The sharding mechanism allows for higher data availability, increased storage capacity, higher data throughput. When some (but not all) shards fail to work in ArangoDB, the whole collection can still be accessible and functional overall if the user wants to read from those unaffected shards. In addition, sharding reduceds the size of the collection piece that each server needs to store, which allows a larger collection to be stored in the database and increases storage capacity. Meanwhile, the data flow is not stuck from Coordinator to one single server but spread to multiple DB-Servers, which increases data throughput. Shards can also be moved or balanced among DB-Servers.

However, sharding may reduce the performance of database queries as it requires more communication overhead, especially when the data required within the query is scattered around and located on different shards (and on different servers). This pattern of data storage will affect the scalability. If such a situation is not expected, the user can restrict collec-

[2]Cluster Administration (ArangoDB 3.9): https://www.arangodb.com/docs/3.9/administration-cluster.html

tions to one single shard by using an Enterprise feature *OneShard*. The *SmartGraph* is also an Enterprise feature for optimized sharding of graphs.

A.2.3 Replication

Replication is another important feature to achieve higher data availability, which increases the number of total copies of the data collections. Replication is often used together with sharding. In Figure A.2, the incoming collection is split into five shards and hashed into five different servers, where each "original" shards are called a *Leading Shard*, or *leader*. In addition to the Leading Shard, several *replicas* of Leading Shards from other DB-Servers also exist on each DB-Server, and these replicas are also called *followers*. In this way, replication increases data availability and becomes a key element to disaster recovery and failover.

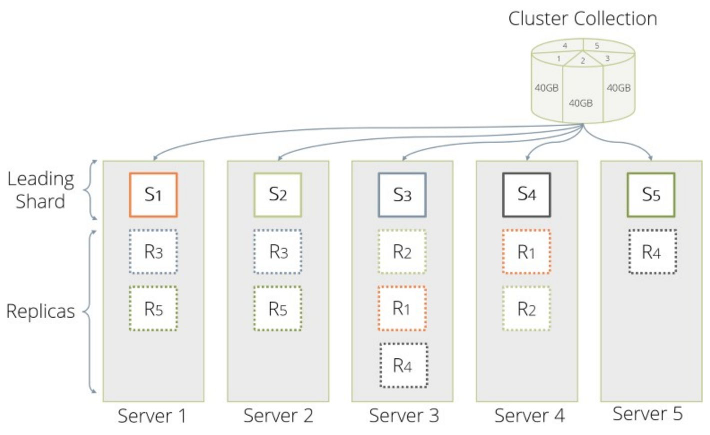


Figure A.2: Sharding and replication [1]

Replication is usually categorized into synchronous and asynchronous modes. Both modes are provided by ArangoDB. However, only the synchronous replication is used among the DB-Servers in the common cluster deployment, while the asynchronous replication is used in other deployment modes (Active Failover and Datacenter-to-Datacenter). Synchronous replication implies that when any write operation happens, all the replicas including the leader and followers will wait for the values to be written (or updated) before further operations can be processed. In ArangoDB v3.9, Merkle trees are used to ensure synchronous replication. It allows the system to quickly determine the difference between the leader and followers by using a tree of hash values.

In general, synchronous replication ensures strong data consistencies among different replicas. The replication is only enabled when the parameter *replicationFactor* is set to a value r ($r > 1$), and then a leader and $r - 1$ followers will be created for each shard accordingly; otherwise r is set to 1 and no replication exists. Also, the replication factor should normally not exceed the total number of DB-Servers. In Figure A.2, the number of shards is set to 5 and the replication factor is set to 3, meaning that every Leading Shard (of the five shards) has two followers, which add up to 15 shards in total.

A.3 Storage Engine and Transactions

Since v3.7 and above, ArangoDB has been using *RocksDB* [31] as its only storage engine. RocksDB is an open-source, embedded key-value store developed by Facebook that is optimized for fast, efficient storage and retrieval of data. It is built on top of LevelDB and provides additional features such as support for multiple column families, compaction filters and persistent cache. RocksDB uses a log-structured merge tree (LSM) data structure to store data. On the other hand, concurrent write conflicts and transaction size limit are two of the caveats that ArangoDB tries to deal with.

ArangoDB supports transactions that conform to the ACID principles. ACID stands for Atomicity, Consistency, Isolation, Durability, and it refers to a set of properties that guarantee that database transactions are processed reliably. ArangoDB supports multi-document transactions that ensure all changes made to the database within a transaction are either committed or rolled back as a unit. This ensures that the database remains in a consistent state throughout the transaction. Also, modifications made by other transactions will be hidden until the current transaction commits such that transactions are isolated from each other to prevent interference. Additionally, ArangoDB provides durability through the use of write-ahead logs (WALs, supported by RocksDB) that allow for recovery of data in the event of a crash or other system failure.

However, fully ACID cannot be achieved in certain cases. ArangoDB claims that fully ACID only applies for the following cases.

- single-document queries, in all deployment modes
- multi-document / multi-collection queries, only in the single-distance mode
- batch operations for multiple documents in the same collection, only in the single-instance mode

ArangoDB uses a three-level permission system (read, write, and exclusive) for collections within a transaction. These permissions work on top of RockDB's locking mechanism. Collections with the read permission can be concurrently read without restriction. However, the RocksDB engine will acquire a (shared) read lock for those collections with the write permission, which allows concurrent reads but prevents concurrent writes. This prevention is an optimistic way that if other concurrent writes are attempted, ArangoDB will abort them and raise an error (with code 1200). Collections with the exclusive permission take a more pessimistic approach, with RocksDB acquiring a write-lock to directly prevent concurrent writes while still allowing concurrent reads to execute successfully.

A.4 Database Operations

A.4.1 ArangoDB Query Language (AQL)

ArangoDB Query Language (AQL) is a declarative query language used for the retrieval and modification of data stored in ArangoDB. AQL's syntax and clarity are similar to SQL, and both languages support data manipulation operations such as inserting, reading, updating, deleting a document. Meanwhile, AQL supports additional operations such as upserting and replacing a document that are not supported in standard SQL. In addition, both AQL and SQL support subqueries and join operations to deal with complex use cases. However, unlike SQL, AQL does not contain the syntax of a data definition language (DDL) or a data control language (DCL). This means that clients cannot create or delete databases or collections using AQL queries and cannot grant specific permissions to a user purely through AQL queries. However, these operations do exist in the programming language-specific ArangoDB drivers that have been released by the official team of ArangoDB. Overall, it is necessary to utilize the capabilities of both the ArangoDB driver and AQL, especially when a client needs to create a database and execute queries on it to achieve a fully functional data workflow.

In general, the execution of a query involves the following two steps.

- A client application submits an AQL query to the ArangoDB server via an ArangoDB driver in a modern programming language (such as Java, Go, Python, etc.).
- ArangoDB will parse, execute and return the results of that query. If the execution is successful, then the client can retrieve the return results via an iterator available through the ArangoDB driver. If any error is raised by ArangoDB, the further process can be done by the exception handling mechanism supported by the programming language of the driver in use.

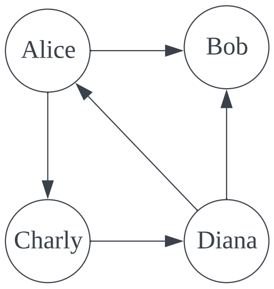


Figure A.3: Example: known graph

For example, Figure A.3 presents an example graph, called `known_graph`, to describe the knowing relations among four persons. Below is an example to execute an AQL query through ArangoDB Java Driver (Java 15 or higher) in this example graph. [3].

```
String query = """FOR p IN person
                      FILTER p.firstName == "Alice"
                      RETURN DOCUMENT(person, p._key)
                 """;
logger.info("Executing AQL Query: read documents with name \"Alice\"...");
try {
    ArangoCursor<BaseDocument> cursor = db.query(query, BaseDocument.class);
    cursor.forEach(doc -> logger.info("Key: " + doc.getKey()));
    logger.info("Query Success.");
} catch (Exception e) {
    logger.error("Query Failure: " + e.getMessage());
```

[3]Adapted from https://github.com/jasonqiu98/arangodb-docker

```
}
```

Listing A.1: AQL example: read documents with name “Alice”

This query tries to read all the documents in the collection *person*, filter those documents with first name `Alice` and return the results in an ArangoDB array. Line 1 stores the query in a plain Java String. Line 7 in the code submits the query to a *db* connection opened in advance. Then it handles the return results in an *ArangoCursor* data structure which is an iterator in essence by implementing the Java iterator interface. In addition, the Java try-catch flow control ensures the smooth running of the system in case of any error caused by the query itself.

A.4.2 Graph Traversals

The graph traversal syntax [4] is one of the key strengths of ArangoDB as a native graph database. Graph traversals can be executed on a named graph defined in advance, or simply a group of document and edge collections that can form an anonymous graph. In either way, the graph traversal query will start from a vertex provided in the query (*startVertex*), visit other vertices along the edges in a certain depth range (from *min* to *max*), and finally reach the end once the vertices are depleted or a certain pruning condition is satisfied.

Several optional arguments can be passed to the query to modify the execution of the graph traversal. By default, the graph traversal follows a depth-first search (DFS) on unique edges. This ensures that graph traversal can be later extended to cycle detection where non-unique vertices and unique edges are needed.

Below is an example query of graph traversals on the known graph in Figure A.3, starting from a vertex `Alice` in the document collection `person`. The depth range of the traversed paths is from 2 to 4 (inclusive). Along the traversal, three variables *vertex*, *edge* and *path* are useful to access the information related to the currently visited vertex in that iteration. The *vertex* is the one currently being visited. The *edge* is the one pointing to the *vertex*. The *path* stores the intermediate results of all the previously visited vertices, edges and paths until and including the *vertex*. This query will output all the possible 2-step to 4-step paths in string format, connecting the vertices on each path with the arrow sign $\rightarrow$.

```
FOR vertex, edge, path IN 2..4
    OUTBOUND "person/alice"
    GRAPH known_graph
    RETURN CONCAT_SEPARATOR("->", path.vertices[*].name)
```

Listing A.2: AQL: Graph Traversal

A.4.3 Scalability

The scalability issue of database operations varies depending on the data model used. Key/-value pairs are the easiest to scale since single key lookups and key/value pair insertions and updates can scale linearly. In contrast, document store and complex queries/joins can

[4]Graph Traversals in AQL (ArangoDB v3.9): https://www.arangodb.com/docs/3.9/aql/graphs-traversals.html

be challenging due to sharding configurations and communication requirements between nodes. AQL query language allows complex queries, but for complicated joins, there are limits as to what can be achieved.

Graph databases are particularly good at queries on graphs, but performance suffers if the vertices and edges are distributed across the cluster. Therefore, it is important to get the distribution of the graph data across the shards right. ArangoDB allows users to specify how their data is sharded to optimize performance. Most of the time, the application developers and users of ArangoDB know best how their graphs are structured. A useful first step is usually to make sure that the edges originating at a vertex reside on the same cluster node as the vertex.

A.5 Limitations

ArangoDB has limitations in its transactions and AQL query language. Long or large-size transactions are not optimized in its implementation, so it is recommended to break them down into smaller ones. The metadata of a transaction operation needs to fit into main memory, while the actual operation can exceed. Additionally, nested transactions are disallowed, and certain operations for creating and deleting system collections are reserved.

The setup of an ArangoDB cluster is limited by hardware resources such as CPU, main memory, secondary memory, and network quality. The Enterprise Edition imposes limits on the number of databases, collections, and shards. A few thousand databases are supported by an Enterprise Edition cluster, with each database supporting up to a thousand collections, and the total number of shards in the cluster being controlled under 50,000. On the other hand, the Community Edition lacks such guarantees and disallows certain operations that rely on Enterprise version-only features. For example, the OneShard feature is required for ensuring the correctness of Pregel Algorithms in an ArangoDB cluster, which is not available in the Community Edition.

AQL queries also have known limitations. A query can use no more than 1000 registers to store results, and it cannot use more than 2048 collections or shards. There are also system-related limitations, such as no more than 4000 “execution nodes” and 500 “nesting levels”, which must be considered while creating a query.

Some limitations in ArangoDB are in place for good design practices and ensuring database isolation. For example, concurrent writes are directly prohibited, and one write operation on an object cannot be followed by another read on the same object within a single AQL query.

Appendix B

Pseudocode of Cycle Detection Algorithms

This chapter lists the pseudo code of traversal-based cycle detection algorithms and discusses their time and space complexities. Unless specified otherwise, in all the following programs, $G = (V, E)$ is the input graph, n is the total number of vertices, and m is the total number of edges. The output of the cycle drops the ending vertex (as it is the same as the starting vertex).

B.1 Starting Vertex Detection

Algorithm 2 starts a DFS traversal from each vertex in G, and along each traversal finds whether the current path contains any vertex that is equal to the vertex where the traversal starts. This algorithm takes two additional input arguments: the minimum depth of DFS traversal *minDepth* and the maximum *maxDepth*. All the traversals will walk *maxDepth* steps at the maximum and explore the potential cycle only in the step range from *minDepth* to *maxDepth*. Once a starting vertex is detected, the algorithm will output the cycle.

The time complexity of this algorithm is $O(n(n+m'))$, where $m' = \min\{d^{maxDepth}, m\}$ is the maximum possible edges within each traversal, and d is the maximum degree of all vertices. When d and m are sufficiently small so that m' is constant compared with n, the time complexity is $O(n^2)$. If the effect of *maxDepth* is ignored, the time complexity has a loose upper bound of $O(n(n+m))$.

The space complexity of this algorithm is $O(n^2)$ as each traversal requires an auxiliary *visited* array taking $O(n)$ space and a *path* stack taking an additional $O(n)$ space. In practice, *visited* and *path* can be reused across traversals, and the space complexity may be reduced to $O(n)$.

B.2 Back Edge Detection

Algorithm 3 starts a DFS traversal from each vertex in G, and along each traversal finds whether the path contains a back edge to a previous vertex in this path. Each traversal

Algorithm 2 Starting vertex detection

```
Input: graph $G = (V, E)$, total number of vertices $n$, minimum and maximum depths of
    traversal $minDepth$, $maxDepth$
Output: a boolean value: whether a cycle with at least two vertices exists or not
 1: for $v$ in $V$ do
 2:     $visited \leftarrow$ [False] * $n$          ▷ init a $visited$ array for each starting vertex
 3:     if DFS($v$, $v$, [$v$], $minDepth$, $maxDepth$, $visited$) then
 4:         return true
 5: return false

 6: function DFS($v$, $start$, $path$, $minDepth$, $maxDepth$, $visited$)
 7:     $visited[v] \leftarrow$ true
 8:     if length($path$) - 1 $\geq maxDepth$ then
 9:         return false
10:     for $(v, w)$ in $E$ do                                  ▷ all edges out from $v$
11:         if $minDepth \leq$ length($path$) - 1 $\leq maxDepth$ and $w == start$ then
12:             print $path$                                      ▷ print the cycle
13:             return true
14:         if not $visited[w]$ then
15:             $path$.push($w$)
16:             if DFS($v$, $start$, $path$, $minDepth$, $maxDepth$, $visited$) then
17:                 return true
18:             $path$.pop()
19:     return false
```

maintains a hash set so that the process of checking whether an edge is a back edge can be reduced to $O(1)$ time. This allows each traversal to be finished within $O(n)$ time because at most $n+1$ edges will contribute a back edge. Therefore, the time complexity is $O(n^2)$. Similar to the analysis of Algorithm 2, the space complexity is $O(n^2)$ and has the possibilities to be reduced to $O(n)$.

B.3 Shortest Path Detection

Algorithm 4 iterates over all edges of G. Once each edge is visited, say $v \rightarrow w$, a BFS traversal will be started to find whether a shortest path from w to v can be found so that there exists a cycle with the edge $v \rightarrow w$ as a back edge. The time complexity is $O(m(n+m))$. The space complexity is $O(mn)$ as each traversal requires a *visited* array taking $O(n)$ space, a level-order queue taking $O(n)$ space, and a hash map to store the visited edges that takes $O(n)$ space. An additional array is used to store the result of cycle and takes an additional $O(n)$ space, but is not considered in the complexity analysis.

Algorithm 3 Back edge detection

```
Input: graph $G = (V, E)$, total number of vertices $n$
Output: a boolean value: whether a cycle with at least two vertices exists or not
 1: for $v$ in $V$ do
 2:     $visited$ ← [False] * $n$        ▷ init a $visited$ array for each starting vertex
 3:     if DFS($v$, [$v$], set{$v$}, $visited$) then
 4:         return true
 5: return false

 6: function DFS($v$, $path$, $vset$, $visited$)
 7:     $visited[v]$ ← true
 8:     for $(v, w)$ in $E$ do        ▷ all edges out from $v$
 9:         if $w$ in $vset$ then
10:             while $path[0]$ != $w$ do
11:                 $path$.popleft()
12:             print $path$
13:             return true
14:         if not $visited[w]$ then
15:             $path$.push($w$)
16:             $vset$.add($w$)
17:             if DFS($w$, $path$, $vset$, $visited$) then
18:                 return true
19:             $vset$.remove($w$)
20:             $path$.pop()
21:     return false
```

B.4 Tarjan's Algorithm

Algorithm 5 [61] finds a strongly connected component (SCC) through a DFS traversal on graph G, where each vertex and edge is visited once. It additionally maintains two arrays *indices* and *lowlinks*. The element $indices[v]$ maps to the index of the vertex v, which is the timestamp when v first shows up in the traversal; the element $lowlink[v]$ stores the smallest index of all vertices that v can reach, which is the earliest timestamp when a vertex of the same SCC shows up in the traversal. In this way, all the indices stored in *lowlink* within the same SCC are reduced to the earliest timestamp and then SCCs are successfully determined. The time complexity of Tarjan's algorithm is $O(n+m)$ and the space complexity is $O(n)$.

B.5 Kosaraju-Sharir's Algorithm

Algorithm 6 [56] finds an SCC through two DFS traversals, one on the transpose of the graph G and the other on the graph G itself. It can be decomposed to four steps.

1. Get the transpose G^T of graph G by reversing the direction of each edge in G.

Algorithm 4 Shortest path detection

```
Input: graph $G = (V, E)$, total number of vertices $n$
Output: a boolean value: whether a cycle with at least two vertices exists or not
1:  for $v$ in $V$ do
2:      for $(v, w)$ in $E$ do                                  ▷ all edges of $G$
3:          if BFS($w$, $v$) then                                ▷ find a back path
4:              return true
5:  return false

6:  function BFS($src$, $dest$)
7:      $visited$ ← [False] * $n$                               ▷ init a $visited$ array
8:      $visited[src]$ ← true
9:      $revEdgeMap$ ← empty map
10:     $q$ ← queue([$src$])
11:     while $q$ do
12:         $size$ ← length($q$)
13:         for $1..size$ do
14:             $v$ = $q$.poll()
15:             for $(v, w)$ in $E$ do                           ▷ all edges of $G$ out from $v$
16:                 if $w == dest$ then                          ▷ find the start
17:                     $s$ ← [$v$]                              ▷ append the end of the cycle
18:                     $ptr$ ← $v$
19:                     while $ptr$ != $src$ do
20:                         $ptr$ ← $revEdgeMap[ptr]$
21:                         $s$.append($ptr$)                    ▷ append along the reversed direction
22:                     $s$.append($w$)                          ▷ add back the start of the cycle
23:                     print reverse($s$)                       ▷ print the cycle in order
24:                     return true
25:                 if not $visited[w]$ then
26:                     $visited[w]$ ← true
27:                     $q$.offer($w$)
28:                     $revEdgeMap[w]$ ← $v$                    ▷ $v \rightarrow w$ but store map[$w$] = $v$
29:     return false
```

2. Get one topological order $topo$ of G^T.
3. Start a DFS traversal on graph G while replacing the iterative order of vertices with $topo$.
4. Every time a DFS recursion tree returns to the root node, an SCC is traversed and the algorithm outputs that SCC.

This algorithm is conceptually simpler than Tarjan's algorithm. However, it takes additional steps and more space to store the transpose of the input graph. The correctness of the algorithm is based on two properties of SCCs in a graph G and its transpose G^T. Firstly, G

Algorithm 5 Tarjan's algorithm

```
Input: graph $G = (V, E)$, total number of vertices $n$
Output: a boolean value: whether a cycle with at least two vertices exists or not
 1: $S \leftarrow$ empty stack, $index \leftarrow 0$
 2: $indices$, $lowlinks$ $\leftarrow$[-1] * $n$, $onStack \leftarrow$ [false] * $n$
 3: for $v$ in $V$ do
 4:     if $indices[v]$ == -1 and STRONGCONNECT($v$) then
 5:         return true                         ▷ early stop: when a cycle has been found
 6: return false                                ▷ no cycle has been found

 7: function STRONGCONNECT($v$)
 8:     $indices[v] \leftarrow index$
 9:     $lowlinks[v] \leftarrow index$
10:     $index \leftarrow index + 1$
11:     $S$.push($v$)
12:     $onStack[v] \leftarrow$ true

13:     for $(v, w)$ in $E$ do                  ▷ all edges out from $v$
14:         if $indices[w]$ == -1 then
15:             if STRONGCONNECT($w$) then      ▷ $w$ not visited; start recursion
16:                 return true
17:             $lowlinks[v] \leftarrow$ min($lowlinks[v]$, $lowlinks[w]$)   ▷ update to the start vertex
18:         else if $onStack[w]$ then
19:             $lowlinks[v] \leftarrow$ min($lowlinks[v]$, $indices[w]$)    ▷ found the start vertex
20:     if $lowlinks[v]$ == $indices[v]$ then   ▷ the start vertex of the SCC
21:         $scc \leftarrow$ []                 ▷ start a new SCC
22:         repeat
23:             $w \leftarrow S$.pop()
24:             $onStack[w] \leftarrow$ false
25:             $scc$.append($w$)
26:         until $w == v$                      ▷ pop all the vertices of the SCC
27:         if length($scc$) $> 1$ then
28:             print reverse($scc$)            ▷ print the vertices in order
29:             return true
30:     return false
```

and G^T has the same set of SCCs. Secondly, two distinct SCCs of graph G are not mutually reachable, meaning that the vertices of one of the two SCCs can reach all the vertices of the other SCC, but not vice versa. This can be proved by contradiction, because mutual reachability will reduce the two distinct SCCs to one single SCC. These two properties ensure that a topological order of G^T, as the iterative order of vertices, will restrict the each recursion tree of the DFS recursion forest within a single SCC instead of reaching nodes of other SCCs.

This algorithm has a time complexity of $O(n+m)$ and a space complexity of also $O(n+m)$, if the graph G is stored in an adjacency list. Transposing an adjacency list requires $O(n+m)$ time and space complexities. After that, a topological order of G^T can be achieved by exploring the reverse of its post-order traversal result through DFS (e.g. Algorithm 7), which requires $O(n+m)$ time complexity and $O(n)$ space complexity. The second DFS traversal on G requires an additional $O(n+m)$ time complexity and $O(n)$ space complexity. When the graph is stored in an adjacency matrix, the transposition costs $O(n^2)$ time and space complexities and will increase the complexities of total algorithm to $O(n^2+m)$.

Algorithm 6 Kosaraju-Sharir's algorithm

```
Input: graph $G = (V, E)$ in adjacency list, total number of vertices $n$
Output: a boolean value: whether a cycle with at least two vertices exists or not
$G^T = (V, E^T)$                          ▷ reverse the direction of every edge
$V' \leftarrow$ TOPOSORT($G^T$)
return STRONGCONNECT($G, V'$)

function STRONGCONNECT(graph, vertices)
    visited ← [False] * n
    for v in vertices do
        if not visited[v] then
            if DFS(v, graph, [v], visited) then
                return true
    return false

function DFS(v, graph, path, visited)
    visited[v] ← true
    counter ← 0                 ▷ counter equal to 0 marks an ending vertex
    for $(v, w)$ in E do
        if not visited[w] then
            counter ← counter + 1
            path.push(w)
            if DFS(w, graph, path, visited) then
                return true
            path.pop()
    if length(path) $\geq$ 2 and counter == 0 then
        print path                      ▷ an SCC with at least two vertices
        return true
    return false
```

Algorithm 7 Topological sort

```
Input: $graph = (vertices, edges)$, total number of vertices $n$
Output: a topological order by reversing the post-order traversal
 1: function TOPOSORT($graph$)
 2:     $visited$ ← [False] * $n$
 3:     $topo$ ← []
 4:     for $v$ in $vertices$ do
 5:         if not $visited[v]$ then
 6:             TOPODFS($v$, $graph$, $topo$, $visited$)
 7:     return reverse($topo$)

 8: function TOPODFS($v$, $graph$, $topo$, $visited$)
 9:     $visited[v]$ ← true
10:     for $(v, w)$ in $E$ do
11:         if not $visited[w]$ then
12:             TOPODFS($w$, $graph$, $topo$, $visited$)
13:     $topo$.append($v$)                                   ▷ post-order traversal
```

B.6 Path-based Strong Component Algorithm

Algorithm 8 [33] [34] is similar to Tarjan's algorithm but uses stack structures to maintain the visited vertices and their indices. The time complexity is $O(n+m)$ and the space complexity is $O(n)$.

Algorithm 8 Path-based strong component algorithm (Gabow's version)

Input: graph $G = (V, E)$, total number of vertices n
Output: a boolean value: whether a cycle with at least two vertices exists or not

```
 1: $S$, $B \leftarrow$ empty stack, $c \leftarrow n$, $I \leftarrow$[-1] * $n$
 2: for $v$ in $V$ do
 3:     if $I[v]$ == -1 then
 4:         if STRONGCONNECT($v$) then
 5:             return true
 6: return false

 7: function STRONGCONNECT($v$)
 8:     $S$.push($v$)
 9:     $I[v] \leftarrow$ length($S$) - 1
10:     $B$.push($I[v]$)
11:     for $(v, w)$ in $E$ do
12:         if $I[w]$ == -1 then
13:             if STRONGCONNECT($w$) then
14:                 return true
15:             else                                      ▷ contract if necessary
16:                 while $I[w] < B$[length($B$) - 1] do
17:                     $B$.pop()

18:     $path \leftarrow$ []
19:     if $I[v]$ == $B$[length($B$) - 1] then
20:         $B$.pop()
21:         $c \leftarrow c$ + 1
22:         $counter \leftarrow 0$
23:         while $S$ and $I[v] \leq$ length($S$) - 1 do      ▷ the SCC
24:             $counter \leftarrow counter$ + 1
25:             $vtop \leftarrow S$.pop()
26:             $path$.append($vtop$)
27:             $I[vtop] \leftarrow c$
28:         if $counter > 1$ then
29:             print reverse($path$)
30:             return true
31:     return false
```

Appendix C

Validity of the Relaxation from Cycle Detection to SCC Detection

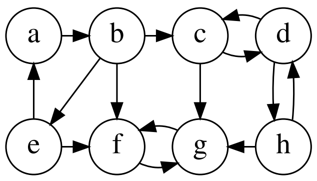


Figure C.1: Directed graph, cycles and SCCs

The relaxation is valid because the the relation between cycles and SCCs (of at least two vertices) is a surjection, i.e., a cycle must be within an SCC, and there always exists at least one cycle in an SCC. For example, the graph in Figure C.1 has three SCCs formed by vertex sets $\{a,b,e\}$, $\{c,d,h\}$ and $\{f,g\}$, and four cycles $a \rightarrow b \rightarrow e \rightarrow a$, $f \rightarrow g \rightarrow f$, $c \rightarrow d \rightarrow c$ and $d \rightarrow h \rightarrow d$. Every cycle can find one and only one accompanying SCC, while one SCC may contain one or more cycles. For instance, the SCC $\{c,d,h\}$ maps to two cycles $c \rightarrow d \rightarrow c$ and $d \rightarrow h \rightarrow d$. Therefore, successful detection of an SCC of at least two vertices ensures that the graph contains at least one cycle.

We show the proof with the following two steps.

1. Based on the definition, a cycle along with the vertices that form the cycle always satisfies the strongly connected property and therefore is a subset of an SCC. (Theorem 1 of Chapter 25 in [30])
2. In addition, an SCC of at least two vertices always contains a non-empty path that further contains a directed cycle. This is because for any directed edge e_0 pointing from vertex v to vertex w, in any SCC C in a graph G, a non-empty back path $\{e_1, e_2, \ldots, e_j\}$ can always be found in which the start and end vertices are w and v, respectively, based on the strongly connected property. The edge e_j is a back edge and then the existence of a cycle is proved.

Overall, a surjection from the set of cycles to the set of SCCs is established, which validates the reduction from cycle detection problem to SCC detection problem.

Appendix D

Graph Construction Stage (Complete Version)

This chapter uses the list history in Listing D.1 for illustration, which is adapted from the history in Listing 3.1 by removing invocation records, aborted transactions, and those keys irrelevant to construct dependency edges.

```
{:type :ok, :f :txn, :value [[:append 5 1] [:append 5 2] [:r 3 []] [:r 5 [1 2]]],
    :time 18468583939, :process 19, :index 3}
{:type :ok, :f :txn, :value [[:append 5 3] [:r 3 []] [:r 2 []]], :time
    18512103772, :process 19, :index 6}
{:type :ok, :f :txn, :value [[:r 7 []] [:append 3 2] [:append 2 1] [:r 3 [2]]], :
    time 18516867536, :process 10, :index 7}
{:type :ok, :f :txn, :value [[:r 7 []] [:r 3 [2]] [:r 2 [1]]], :time 18558278528, :
    process 19, :index 10}
{:type :ok, :f :txn, :value [[:append 7 1] [:append 7 2] [:r 7 [1 2]] [:append 7
    3] [:append 3 3]], :time 18563033073, :process 8, :index 11}
```

Listing D.1: Jepsen list history (filtered and adapted)

D.1 Vertices and Edges

To construct a dependency graph on transactions, we create a vertex collection `txn` to represent transactions as well as an edge collection `dep` to store dependency edges. Each vertex in the `txn` collection has only one field `_key`, which records the transaction id in the format `txn/i`. The `i` is the original `:index` in the execution history. Each edge in the `dep` collection has three main fields, `_from`, `_to`, and `type`. The first two fields mark the id's of the two end vertices of the edge, while the last field indicates the type of the dependency edge out of WR, WW, and RW. In addition, each dependency edge maintains the id's of two connected events in the fields `from_evt` and `to_evt`. The two collections `txn` and `dep` form the dependency graph `txn_g`.

In addition to the main graph, we also introduce an auxiliary dependency graph on events. This graph `evt_g` consists of three collections, `a_evt` for append events for list histories (or `w_evt` for read events for register histories), `r_evt` for read events, and `evt_dep` for event dependency edges. Each event vertex in `a_evt` has four fields: `_key`, `obj`, `arg`,

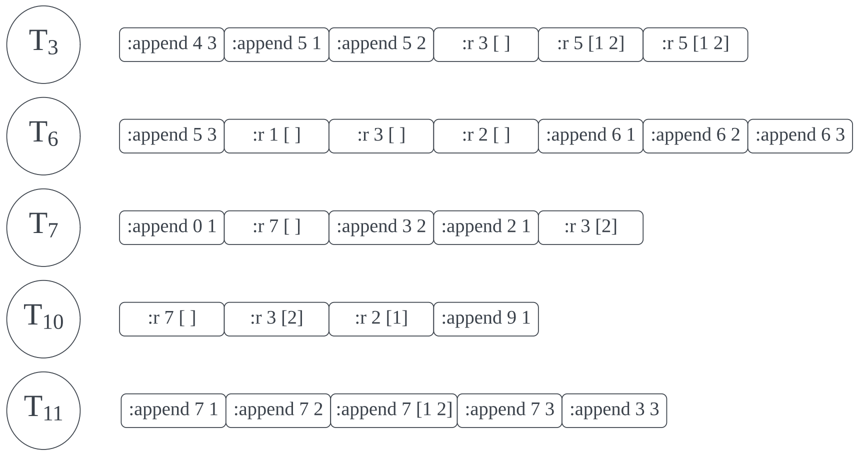


Figure D.1: Graph construction: empty dependency graph

and `index`. The field `_key` stores the event id in the format `evt/i,j`, where `i` is the `:index` of the transaction that contains this event in the history, and `j` is the zero-based index of the event within this transaction. The `obj` and `arg` are aliases of key and value of the object. In particular, the field `obj` is a string converted from a positive integer starting from 1; the field `arg` is a positive integer starting from 1. The last field, `index`, is reserved to check intermediate writes. It records the relative index of the append events on the same object within the transaction, but the `index` of the last such event is recorded as -1. For example, in Listing D.1, the append event `[:append 5 2]` is stored as `(evt/3,1, "5", 2, 1)`, `[:append 3 2]` as `(evt/7,1, "3", 2, -1)`.

Each read event in `r_evt` has three fields, `_key`, `obj`, and `v`, for the event id, object key, and value array, respectively. For register histories, the field `obj` is a single value instead of a value array. Furthermore, each event dependency edge in `evt_dep` has four fields, `_from`, `_to`, `obj`, and `type`, for the starting event id, the ending event id, the object key, and the dependency type, respectively.

The format of event id `evt/i,j` ensures that the transaction `txn/i` can be tracked without storing an additional field in event vertices. This is useful when we project the dependency graph from events to transactions.

D.2 Construction of Dependency Edges on Events

Under the Assumption of Unique Writes, each value can be written only once for each object. Also, a value cannot be read unless it has already been written by another previous event. This implies that values in read events can automatically locate the contributing write events on each object. For example, in Listing D.1, T_{10} (the transaction with `:index` of 10; the same hereafter) reads the value of the object with key 2 (object 2; the same hereafter) as [1], while T_7 appends 1 to object 2. Based on this information, we draw a WR edge from T_7 to T_{10} because T_7 appends the last value read by T_{10} on object 2 (see Figure D.2). Overall, an execution history automatically reveals all the read dependencies (i.e., WR edges) on

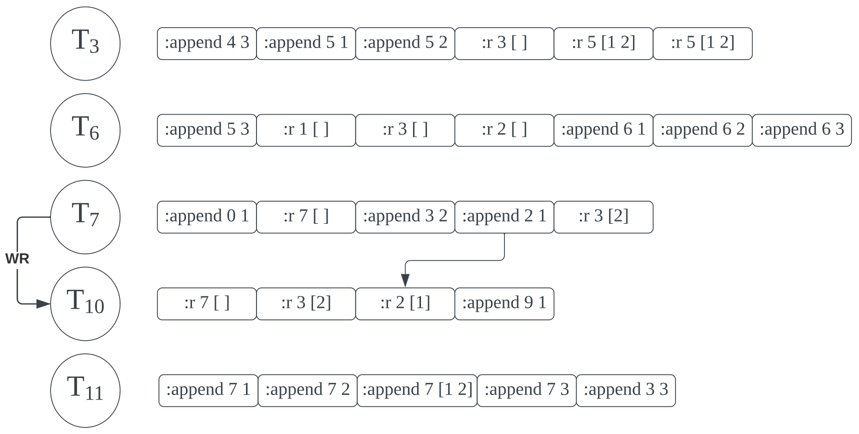


Figure D.2: Graph construction: WR edge under Assumption 1

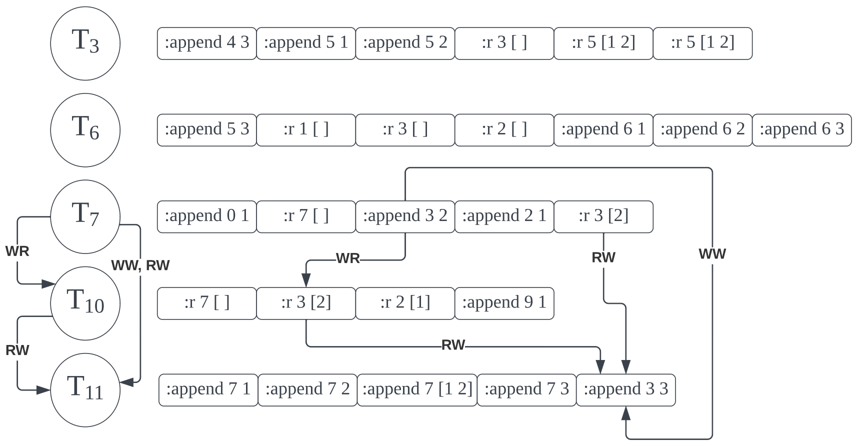


Figure D.3: Graph construction: the longest list

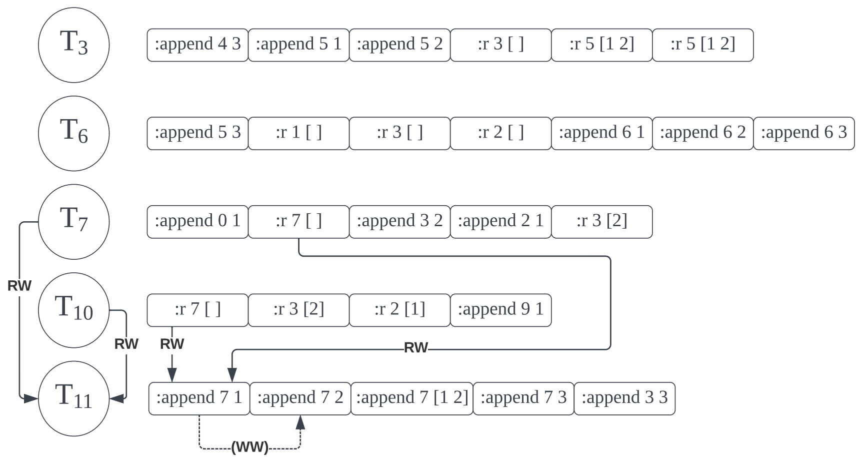


Figure D.4: Graph construction: the remaining list

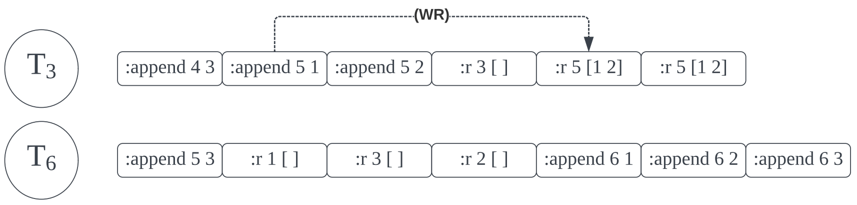


Figure D.5: Graph construction: the missing WW and WR edges

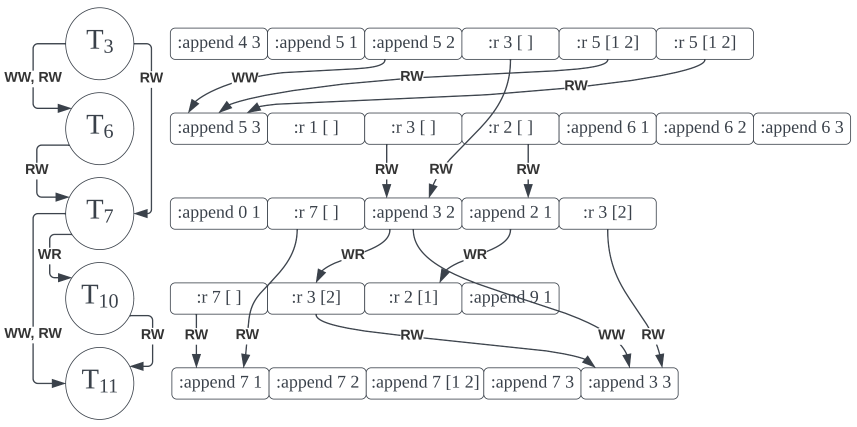


Figure D.6: Graph construction: complete dependency graph

each object.

The remaining two types, WW and RW edges, highly depend on the version orders recorded in the history. The process of recovering such version orders from the two types of histories also differs. The differences are discussed as follows.

D.2.1 List Histories

To restore the version orders from list histories, we use two functions to pre-process the events in the history $\mathcal{H}$ into the two data structures R and A, respectively. The two functions, queryReadEvts and queryAppendEvts, retrieve the relevant information of read and append events through AQL queries, listed in Listing D.2 and Listing D.3, respectively.

```
FOR e1 IN r_evt
    COLLECT obj = e1.obj INTO objs
    RETURN { obj, records: (
        FOR e2 in objs[*].e1
            COLLECT val = e2.v INTO vals
            SORT LENGTH(val) DESC
            RETURN { val, ids: vals[*].e2._id }) }
```

Listing D.2: Query read events from list histories

The query in Listing D.2 uses an aggregation function to group the information of all read events E_r by object. Each object maintains an array of **records**. Each record contains a list **value** and an array of **record ids**. The events with these ids read the value of this object. Furthermore, we refer to the last element of the list value as the **item** of the value. In Listing

D.1, both T_7 and T_{10} read [] while T_{11} reads [1 2] from object 7. We say object 7 has two records: one is with value [1 2] and ids [11]; the other is with value [] and ids [7, 10].

Since the records of an object reflects its version order, we sort the records based on the length of the list value contained in each record. In the records of object 7, the value [1 2] shows earlier than [].

After having the query, we directly query on E_r and store the result set into R.

Algorithm 9 The function `queryReadEvts` for list histories

```
Input: read events $E_r$
Output: array of records per object $R$
1: function QUERYREADEVTS($E_r$)
2:     $R \leftarrow$ runQuery($E_r$, query)                ▷ query in Listing D.2
3:     return $R$
```

```
FOR e1 IN a_evt
    COLLECT obj = e1.obj into objs
    RETURN { obj, evts: (
    FOR e2 in objs[*].e1
        COLLECT element = e2.arg INTO elements
        RETURN { element, ids: elements[*].e2._id, append_idx: elements[*].e2.index
    }) }
```

Listing D.3: Query append events from list histories

The query in Listing D.3 aggregates records of all append events in a similar way to read events. Differently, the values of append events are not arrays but single values. Also, `append_idx` is attached to the results as an auxiliary attribute to identify intermediate writes, which is a necessary part of identifying dirty writes. After the query is finished, we post-process the result into a hashmap based on the object keys. This converts the result into A, which is a lookup table of append events.

Algorithm 10 The function `queryAppendEvts` for list histories

```
Input: append events $E_a$
Output: lookup table of append events $A$
1: function QUERYAPPENDEVTS($E_a$)
2:     $I \leftarrow$ runQuery($E_a$, query)                ▷ query in Listing D.3
3:     $A \leftarrow$ empty hashmap
4:     for info $\in I$ do
5:         $A$[info.obj] $\leftarrow$ empty hashmap
6:         for $e \in$ info.evts do
7:             $A$[info.obj][$e$.arg] $\leftarrow$ $e$.ids[0]       ▷ add to the lookup table
8:     return $A$
```

D.2.2 Event-Record Relations: Installment, Visiting, and Event Version Order

To simplify the notations, we first introduce the following two relations in Definition 4 between an event and a record.

Definition 4 *The installment and visiting relations.*

1. *The* ***installment*** *relation is defined between an append (or write) event and a record of the same object in an execution history. An append (or write) event e installs a record r if and only if e appends the item of (or writes) r's value and there exists a unique event within a committed transaction that appends (or writes) so to the object, i.e., $e.arg == \texttt{LAST}(r.val)$. This is denoted by $e \vdash r$.*
2. *The* ***visiting*** *relation is defined between a read event and a record of the same object. A read event e_r visits a record r if and only if e_r reads r's value, i.e., $e_r.val == r.val$, or alternatively, $e_r.id \in r.ids$. This is denoted by $e_r \dashrightarrow r$.*

Furthermore, we denoted the event version order by $\ll$ in the following way.

Definition 5 *The* ***event version order*** *is defined between two records of the same object in an execution history. For two records r_1 and r_2, r_1 is prior to r_2 in event version order if and only if $r_1.val$ is a strictly shorter prefix of $r_2.val$. This is denoted by $r_1 \ll r_2$.*

In addition, we combine the event version order and installment, and propose the following.

Proposition 3 *The extension of event version order between an append (or write) event and a record.*

1. *For two records r_1 and r_2, $r_1.val \ll r_2.val$ is equivalent to $r_1 \ll r_2$.*
2. *For an append (or write) event e and a record r, $e \ll r$ is equivalent to $e.arg \ll r.val$.*

D.2.3 The Algorithm to Construct Event Dependency Edges

After both the array of records per object R and the lookup table of append events A are ready, and the concepts of installment ($\vdash$), visiting ($\dashrightarrow$), and event version order ($\ll$) are established, Algorithm 11 is designed to retrieve the set of dependency edges on events. Also, to simplify the notations, we use $\mathcal{E}_{events}$ to denote the set of event dependency edges, which consists of edges of the form $(e_{from}, e_{to}, type)$.

The algorithm emphasizes the traceability property in list histories with a function `getEvtDepEdges`. It iterates over the objects read in the history. Under each object, the loop body starts by checking the longest record, then the second longest record, until the shortest record. Furthermore, for each record, we form the dependency edges in the order of RW, WW, and WR edges.

For the longest record, we first check all possible later versions that are not present, and form RW edges between each read event that visits the longest record and each append event that installs a later version. After that, WW edges are formed between the unique append event that installs the longest record and each append event that installs a later

Algorithm 11 Construction of dependency edges on events for list histories

Input: an array of records per object grouped by each object R, a lookup table of append events A

Output: a set of event dependency edges $\mathcal{E}_{events}$

```
1: function getEvtDepEdges($R$, $A$)
2:     $\mathcal{E}_{events} \leftarrow \emptyset$
3:     for all $r \in R$ do
4:         $k \leftarrow r$.obj
5:         $Re \leftarrow r$.records                                        ▷ records of object $k$
6:         if $k \notin A$ then
7:             if $\{re.val \mid re \in Re\} == \{[]\}$ then continue          ▷ only initial reads
8:             else
9:                 Anomaly: aborted reads for each $r$          ▷ object not appended but read
10:        if $re[0]$.val == [] then                              ▷ the longest record is empty
11:            if $k \in A$ then                                          ▷ later versions exist
12:                $\mathcal{E}_{events} \leftarrow \mathcal{E}_{events} \cup \{(e_r, e_a, \mathsf{RW}) \mid e_r \rightarrow Re[0] \wedge Re[0] \ll e_a\}$
13:            continue
14:        if $\forall e_a \in A[k]. \neg(e_a \vdash Re[0])$ then        ▷ the longest record was not appended
15:            Anomaly: aborted reads for $Re[0]$

16:        for all $\{e_a \mid Re[0].\mathrm{val} \ll e_a.\mathrm{arg}\}$ do
17:            $\mathcal{E}_{events} \leftarrow \mathcal{E}_{events} \cup \{(e_r, e_a, \mathsf{RW}) \mid e_r \rightarrow Re[0] \wedge Re[0] \ll e_a\}$
18:            $\mathcal{E}_{events} \leftarrow \mathcal{E}_{events} \cup \{(e_a, e'_a, \mathsf{WW}) \mid e_a \vdash Re[0] \wedge Re[0] \ll e'_a\}$
19:        if $\{e_a \mid e_a \vdash Re[0]\} \neq \emptyset$ then
20:            $\mathcal{E}_{events} \leftarrow \mathcal{E}_{events} \cup \{(e_a, e_r, \mathsf{WR}) \mid e_a \vdash Re[0] \wedge e_r \rightarrow Re[0]\}$
21:        $re_{longer} \leftarrow re[0]$
22:        for all $re_{cur} \in Re \setminus Re[0]$ do
23:            if $re_{cur} \ll re_{longer}$ then
24:                $re_{next} \leftarrow re_{longer}[0{:}\mathrm{length}(re_{cur}){+}1]$                  ▷ a helper record
25:                $\mathcal{E}_{events} \leftarrow \mathcal{E}_{events} \cup \{(e_r, e_a, \mathsf{RW}) \mid e_r \rightarrow re_{cur} \wedge e_a \vdash re_{next}\}$
26:                if $re_{cur}.\mathrm{val} \neq []$ then
27:                    $\mathcal{E}_{events} \leftarrow \mathcal{E}_{events} \cup \{(e_a, e'_a, \mathsf{WW}) \mid e_a \vdash re_{longer}[0{:}l] \wedge e'_a \vdash$
       $re_{longer}[0{:}l{+}1] \wedge \mathrm{length}(re_{cur}) \leq l \leq \mathrm{length}(re_{longer}){-}1\}$
28:                    if $\{(e_a, e'_a) \mid e_a \vdash re_{cur} \wedge e'_a \vdash re_{next}\} \neq \emptyset$ then
29:                        $\mathcal{E}_{events} \leftarrow \mathcal{E}_{events} \cup \{(e_a, e_r, \mathsf{WR}) \mid e_a \vdash re_{cur} \wedge e_r \rightarrow re_{cur}\}$
30:                $re_{longer} \leftarrow re_{cur}$                                          ▷ update
31:            else
32:                Fatal: inconsistent records
33:        if $re_{longer} \neq []$ then
34:            $re_{cur} \leftarrow re_{longer}[0{:}1]$                         ▷ keep the first element only
35:            $\mathcal{E}_{events} \leftarrow \mathcal{E}_{events} \cup \{(e_a, e'_a, \mathsf{WW}) \mid e_a \vdash re_{longer}[0{:}l] \wedge e'_a \vdash re_{longer}[0{:}l{+}1] \wedge$
       $\mathrm{length}(re_{cur}) \leq l \leq \mathrm{length}(re_{longer}){-}1\}$
36:    return $\mathcal{E}_{events}$
```

version. Then, WR edges are formed between the unique append event that installs the longest record and each read event that visits the longest record.

For example, we can further filter the history in Listing D.1 for object 3 as follows (in Listing D.4).

```
{:value [[:r 3 []]], :index 3}
{:value [[:r 3 []]], :index 6}
{:value [[:append 3 2] [:r 3 [2]]], :index 7}
{:value [[:r 3 [2]]], :index 10}
{:value [[:append 3 3]], :index 11}
```

Listing D.4: Jepsen list history (filtered by object with key 3)

1. RW: the longest record shows in T_7 and T_{10}, which contains only an integer 2. However, T_{11} appends a later version with value 3. Therefore, two RW edges are first constructed between T_7 and T_{11}, and between T_{10} and T_{11}, respectively.
2. WW: After that, the append event that installs the longest record is located in T_7. With this append event, we draw one WW edge from T_7 to T_{11}.
3. WR: Finally, for the WR edges of the longest record, we construct one from T_7 to T_{10}, while drop the other one (as it is within transaction T_7 only).

While we iterate over the remaining records under an object, we always get a current, shorter record that is prior to the previous, longer record in event version order. Otherwise, the algorithm exits with a fatal error of inconsistent records, since the assumption of recoverability is violated. During the iterations, we keep the longer record for comparison. Also, we extract a helper record from the longer record such that the helper record is exactly one element longer than the current record, which we call the next record. After the three records are retrieved, the dependency edges are still constructed in the order of RW, WW, and WR. The RW edge is from the read event that visits the current record to the append event that installs the next record. Then, the WW edges are formed in an iterative way. This is because a value gap may exist between the shorter current record and the longer record. At first, the append event that installs the current record and the one that installs the next record are connected. After that, the pair moves forward along the longer record for one step: the pointer of the next record is passed to the current record, and the next record walks one step toward the longer direction. This traversal along the longer record ends until the next record is identical to the longer record (inclusive). Finally, a set of WR edges is constructed from the append event that installs the current record to each read event that visits the current record.

We still take the history in Listing D.1 but filter by object 7 as an example to illustrate the process (in Listing D.5 and Figure D.4).

```
{:value [[:r 7 []], :index 7}
{:value [[:r 7 []], :index 10}
{:value [[:append 7 1] [:append 7 2] [:r 7 [1 2]] [:append 7 3]], :index 11}
```

Listing D.5: Jepsen list history (filtered by object with key 7)

1. RW: the longest record is [1 2] while a shorter one is []. When the current record [] is observed, the longer record is [1 2], and the next record should be [1]. After the

three records are found, an RW edge is constructed from the event that reads [] to each event that appends 1: we can find two, T_7 to T_{11}, and T_{10} to T_{11}.

2. WW: Later, the pair of the current record and the next record walks along the longer record [1 2] in an iterative fashion. The pair starts at ([], [1]), but [] is the initial value and no append event exists. As the next step, the pair moves to ([1], [1 2]) by incorporating one more element from the longer record. The append events that install the two versions both lie in T_{11}, so this dependency edge is ignored. Since the next record is now identical to the longer record [1 2], the iterations end.
3. WR: Finally, the append events that install [] and the read events that visit [] form a set of WR edges. Since there is no such append event, we do not construct any new dependency edge.

As the final step to construct event dependency edges, after the shortest record is traversed, there may still exist missing WW dependencies revealed by it. We filter the object 5 from Listing D.1 (in Listing D.6 and Figure D.5). After the shortest record [1 2] is traversed, only the dependency edges in relation with the last value 2 are considered, while those with the earlier values are ignored. We reset the current record to a new array with only the starting value of the longer record, and then start the iterations to construct WW edges. In this case, the current record is reset to [1], and the longer record is [1 2]. The moving pair starts from ([1], [1 2]) and ends at the same place. However, since T_3 appends both 1 and 2 to object 5, this missing WW is evaluated and ignored again. By now, the construction of dependency graph on events has been completed (see Figure D.6).

```
{:value [[:append 5 1] [:append 5 2] [:r 5 [1 2]]], :index 3}
{:value [[:append 5 3]], :index 6}
```

Listing D.6: Jepsen list history (filtered by object with key 5)

In addition to edge construction, we evaluate two anti-patterns related to the external consistency axiom EXT in Theorem 1. These two anti-patterns are aborted reads and intermediate reads. Aborted reads are caught when a record is read while the append event that installs this record cannot be found. Intermediate reads are detected when a record is read while the append event that installs this record is not the final append on the same object within the transaction of the event. Both anti-patterns are checked during the construction of WR edges. However, the existence of aborted or intermediate reads does not stop the algorithm immediately. Instead, an additional result is returned to indicate whether either of the two are detected. This is because the checking of the lowest level in our checker (i.e., PL-1) does not require the EXT axiom, and these two anti-patterns are allowed to exist.

D.2.4 Register Histories

Compared to list histories, recovering version orders from register histories is more challenging. This is because read-write registers, which act as simple key-value stores, lack the traceability property. Inferring previous values from the current value becomes impossible.

```
{:index 2, :value [[:r 2 nil] [:w 2 1]]}
{:index 5, :value [[:w 2 4] [:r 2 4]]}
```

```
{:index 7, :value [[:w 2 5] [:r 2 5] [:w 2 6]]}
```

Listing D.7: Jepsen register history (filtered by object with key 2)

Consider the example history shown in Listing D.7, where we filter events related to object 2 (see Listing 3.2). In this history, the order of transactions reflects their commit order, not the version orders. Deducing the version orders requires comparing values. For instance, T_2 reads the initial value [] and then writes the value 1 to object 2, indicating that T_2 installs the first non-null value. However, without further read events, we cannot determine the version orders between T_5 and T_7. Here, the WAL log serves as a valuable resource for establishing version orders within the database.

By analyzing the WAL log, specifically the log entry with `"type"` 2300, we can identify the changes in versions for object 2. With this information, we conclude that T_5 is prior to T_7 in the version order, allowing us to create a WW edge.

```
{"tick":"105","type":2300,"db":"rwRegister","cuid":"h2E06B5FF24F1/136","tid
    ":"140","data":{"_key":"2","_id":"rwCol/2","_rev":"_f7LTaqa---","rwAttr
    ":1}}
{"tick":"108","type":2300,"db":"rwRegister","cuid":"h2E06B5FF24F1/136","tid
    ":"145","data":{"_key":"2","_id":"rwCol/2","_rev":"_f7LTay6---","rwAttr
    ":4}}
{"tick":"111","type":2300,"db":"rwRegister","cuid":"h2E06B5FF24F1/136","tid
    ":"150","data":{"_key":"2","_id":"rwCol/2","_rev":"_f7LTa0----","rwAttr
    ":5}}
{"tick":"114","type":2300,"db":"rwRegister","cuid":"h2E06B5FF24F1/136","tid
    ":"150","data":{"_key":"2","_id":"rwCol/2","_rev":"_f7LTa0K---","rwAttr
    ":6}}
```

Listing D.8: ArangoDB Write-Ahead Logs (WAL) of Jepsen register history (filtered by object with key 2)

Utilizing the WAL simplifies the construction of dependency graphs on edges, making it similar to list histories. Algorithm 12 and Algorithm 13 retrieve read and write events, respectively. They construct two lookup tables: one for records per object (R) and another for write events (W). In register histories, we continue using the same term **record**: a record represents a single value read from a register, rather than an array as in list histories. Since the records are values, they do not require sorting. Instead, they are grouped into a lookup table similar to write events.

```
FOR e1 IN r_evt
    COLLECT obj = e1.obj INTO objs
    RETURN { obj, records: (
        FOR e2 in objs[*].e1
            COLLECT val = e2.v INTO vals
            RETURN { val, ids: vals[*].e2._id }) }
```

Listing D.9: Query read events from register histories

```
FOR e1 IN w_evt
    COLLECT obj = e1.obj into objs
    RETURN { obj, evts: (
        FOR e2 in objs[*].e1
            COLLECT element = e2.arg INTO elements
        RETURN { element, ids: elements[*].e2._id, write_idx: elements[*].e2.
index }) }
```

Listing D.10: Query write events from register histories

Algorithm 12 The function `queryReadEvts` for register histories

```
Input: read events $E_r$
Output: lookup table of records per object $R$
1: function QUERYREADEVTS($E_r$)
2:     $I \leftarrow$ runQuery($E_r$, query)                ▷ query in Listing D.9
3:     $R \leftarrow$ empty hashmap
4:     for info $\in I$ do
5:         $R$[info.obj] $\leftarrow$ empty hashmap
6:         for $tr \in$ info.records do
7:             $R$[info.obj][$r$.val] $\leftarrow r$.ids        ▷ add to the lookup table
8:     return $R$
```

Algorithm 13 The function `queryWriteEvts` for register histories

```
Input: write events $E_w$
Output: lookup table of write events $W$
1: function QUERYWRITEEVTS($E_w$)
2:     $I \leftarrow$ runQuery($E_w$, query)                ▷ query in Listing D.10
3:     $W \leftarrow$ empty hashmap
4:     for info $\in I$ do
5:         $W$[info.obj] $\leftarrow$ empty hashmap
6:         for $e \in$ info.evts do
7:             $W$[info.obj][$e$.arg] $\leftarrow e$.ids[0]     ▷ add to the lookup table
8:     return $W$
```

In addition to these lookup tables, the algorithm relies on the WAL as an input for register histories. It iterates over the WAL logs of each object to establish dependency edges between versions. To simplify this process, the logs are organized into a WAL write map using Algorithm 14.

By utilizing lookup tables and a write-ahead log (WAL) as input variables, Algorithm 15 constructs event dependency edges with the necessary information. The algorithm follows these steps:

1. Initially, all objects mentioned in the records per object must exist in the WAL write map. If this condition is not met, the checker identifies a problem known as *aborted reads*.
2. Next, the algorithm loops through all objects recorded in the WAL. For each object:
    - The initial version is evaluated, and two types of edges are created: RW edges and an WR edge.

Algorithm 14 The function `getWALWriteMap` for register histories

```
Input: write-ahead logs $L$
Output: WAL write map $L'$
1: function GETWALWRITEMAP($L$)
2:     $L' \leftarrow$ empty hashmap
3:     for $row \in L$ do
4:         $k \leftarrow row$.data._key
5:         $v \leftarrow row$.data.rwAttr                    ▷ attribute name set by user
6:         if $k \notin L'$ then
7:             $L'[k] \leftarrow []$
8:         $L'[k]$.append($v$)                              ▷ append a new version
9:     return $L'$
```

- The RW edges connect read events that visit the initial zero value to the write event that installs the initial version.
- The WR edge connects the write event that installs the initial version to the read events that visit the initial version.

3. After analyzing the initial version, the algorithm proceeds to evaluate the remaining versions. For each current version, the previous version is also considered as a reference for constructing the edges. Three types of edges are constructed in a specific order: RW, WW, and WR.
   - RW edges connect read events that visit the previous version to the write event that installs the current version.
   - WW edges connect the write event that installs the previous version to the write event that installs the current version.
   - WR edges connect the write event that installs the current version to the read event that visits the current version.
4. This process continues until all remaining versions are processed.

Additionally, for each object, two extra checks are performed to ensure the consistency of the execution history with the WAL:

- Every non-zero version of the object should be included in the WAL. If a version is missing, it indicates that the history reads a value that hasn't been committed to the database, leading to the detection of aborted reads.
- Each version written to the object should also be present in the versions recorded by the WAL. If some writes exist in the history but are missing in the WAL, it implies an inconsistency between the history and the WAL.

By conducting these additional checks and constructing the appropriate edges, the algorithm ensures the coherence and integrity of the execution history in relation to the WAL.

When the size of the history increases, it is important to adjust the `chunkSize` parameter in the WAL log retrieval query (refer to Section 3.2.3). Failure to do so may result

Algorithm 15 Construction of dependency edges on events for register histories

```
Input: a lookup table of records per object $R$, a lookup table of write events $W$, write-ahead
    logs $L$
Output: a list of event dependency edges $\mathcal{E}_{events}$
 1: $\mathcal{E}_{events} \leftarrow \emptyset$
 2: $L' \leftarrow$ getWALWriteMap($L$)                                    ▷ WAL write map
 3: if $\exists object \in R.\ object \notin L'$ then
 4:     Anomaly: aborted reads                    ▷ require all objects are recorded in $map_L$
 5: for all $(object, versions) \in L'$.entries do
 6:     $map_R \leftarrow R[object]$
 7:     $map_W \leftarrow W[object]$
 8:     if $versions$ == [] then
 9:         Fatal: Broken WAL logs
10:     $v_{init} \leftarrow versions[0]$
11:     if $v_{init} \in map_W$ then                                          ▷ initial version
12:         $\mathcal{E}_{events} \leftarrow \mathcal{E}_{events} \cup \{(e_r, e_w, \mathsf{RW}) \mid e_r \rightarrow 0 \wedge e_w \vdash v_{init}\}$
13:         $\mathcal{E}_{events} \leftarrow \mathcal{E}_{events} \cup \{(e_w, e_r, \mathsf{WR}) \mid e_w \vdash v_{init} \wedge e_r \rightarrow v_{init}\}$
14:     else if $v_{init} \in map_R$
15:         Anomaly: aborted reads
16:     $v_{prev} \leftarrow v_{init}$                                  ▷ initialize previous version
17:     for all $v_{cur} \in versions \setminus \{v_{init}\}$ do
18:         if $v_{cur} \in map_W$ then
19:             $\mathcal{E}_{events} \leftarrow \mathcal{E}_{events} \cup \{(e_r, e_w, \mathsf{RW}) \mid e_r \rightarrow v_{prev} \wedge e_w \vdash v_{cur}\}$
20:             $\mathcal{E}_{events} \leftarrow \mathcal{E}_{events} \cup \{(e_w, e'_w, \mathsf{WW}) \mid e_w \vdash v_{prev} \wedge e'_w \vdash v_{cur}\}$
21:             $\mathcal{E}_{events} \leftarrow \mathcal{E}_{events} \cup \{(e_w, e_r, \mathsf{WR}) \mid e_w \vdash v_{cur} \wedge e_r \rightarrow v_{cur}\}$
22:             $v_{prev} \leftarrow v_{cur}$                              ▷ update previous version
23:         else if $v_{cur} \in map_R$
24:             Anomaly: aborted reads
25:     if $\{v \mid v \in map_R\} \setminus \{0, \text{nil}\} \nsubseteq versions$ then
26:         Anomaly: aborted reads
27:     if $\{v \mid v \in map_W\} \nsubseteq versions$ then
28:         Fatal: system faults
29: Return $\mathcal{E}_{events}$
```

in incomplete WAL entries, which means that values beyond the allocated space may be missing. If there is not enough space, the WAL may not contain the values at the end of the history that were successfully written and read by certain transactions. In our code, this situation leads to anomalies in aborted reads because we expect the versions shown in the history to be a subset of the versions recorded in the WAL. To prevent this issue, it is crucial to avoid the anomaly of aborted reads by adjusting the `chunkSize` in the HTTP query. This adjustment allows for lifting the size restrictions and ensuring the integrity of the WAL.

D.3 Projection from Events to Transactions

The query in Listing D.11 converts event dependency edges to transaction dependency edges. It does this by associating event IDs with their corresponding transaction IDs on the end vertices of the edges. The resulting transaction dependency edges are then provided in the desired format. To ensure the validity of each transaction dependency edge, certain rules are enforced. Firstly, the edge must not have the same transaction as both its start and end points. Additionally, for each set of end vertices, only one instance of each transaction type is kept. Once this process is complete, all the vertices and edges in the dependency graph are finalized, marking the completion of the dependency graph construction.

```
LET projs = (
    FOR d IN evt_dep
        LET from_txn = SPLIT(d._from, ["/", ","])[1]
        LET to_txn = SPLIT(d._to, ["/", ","])[1]
        FILTER from_txn != to_txn
        RETURN { _from: CONCAT("txn/", from_txn), _to: CONCAT("txn/", to_txn),
            from_evt: d._from, to_evt: d._to, type: d.type }
    )

FOR proj IN projs
    COLLECT from = proj._from, to = proj._to, type = proj.type INTO groups = {
        "from_evt": proj.from_evt,
        "to_evt": proj.to_evt
    }
    RETURN {
        "_from": from,
        "_to": to,
        "type": type,
        "from_evt": groups[0].from_evt,
        "to_evt": groups[0].to_evt
    }
```

Listing D.11: Projection from events to transactions

Appendix E

AQL Queries of the Graph-Based Checker

This Appendix lists the queries used in the graph-based checker [1].

E.1 Starting Vertex Detection (Cycle checker)

Starting Vertex Detection (Cycle) is used in all isolation levels. Option with randomization is also possible, where `@start` is an argument that accepts the id of a random starting vertex.

E.1.1 Serializability (SER)

```
FOR start IN txn
    FOR vertex, edge, path IN 2..4
        OUTBOUND start._id
        GRAPH txn_g
        FILTER edge._to == start._id
        LIMIT 1
        RETURN path.edges
```

Listing E.1: Checking SER: Starting Vertex Detection (Cycle)

```
FOR vertex, edge, path IN 2..4
    OUTBOUND @start
    GRAPH txn_g
    FILTER edge._to == @start
    LIMIT 1
    RETURN path.edges
```

Listing E.2: Checking SER: Starting Vertex Detection - Randomization (CycleRandom)

```
FOR start IN txn
    FOR vertex, edge, path IN 2..4
        OUTBOUND start._id
        GRAPH txn_g
```

[1]Thesis Artifact: https://github.com/jasonqiu98/GRAIL-artifact/tree/thesis

```
        FILTER LAST(path.edges[*]._to) == start._id
        LIMIT 1
        RETURN path.edges
```

Listing E.3: Checking SER: Starting Vertex Detection - Filtering on Path (CycleFilter)

E.1.2 Snapshot Isolation (SI)

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

Listing E.4: Checking SI: Starting Vertex Detection (Cycle)

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

Listing E.5: Checking SI: Starting Vertex Detection - Randomization (CycleRandom)

```
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 E.6: Checking SI: Starting Vertex Detection - Filtering on Path (CycleFilter)

E.1.3 Parallel Snapshot Isolation (PSI)

```
FOR start IN txn
    FOR vertex, edge, path IN 2..4
        OUTBOUND start._id
        GRAPH txn_g
        FILTER edge._to == start._id AND LENGTH(FOR e IN path.edges FILTER e.type
    == "rw" RETURN e) < 2
        LIMIT 1
        RETURN path.edges
```

Listing E.7: Checking PSI: Starting Vertex Detection (Cycle)

```
FOR vertex, edge, path IN 2..4
    OUTBOUND @start
    GRAPH txn_g
    FILTER edge._to == @start AND LENGTH(FOR e IN path.edges FILTER e.type == "rw
    " RETURN e) < 2
    LIMIT 1
    RETURN path.edges
```

Listing E.8: Checking PSI: Starting Vertex Detection - Randomization (CycleRandom)

```
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 LENGTH(FOR e IN path.edges
     FILTER e.type == "rw" RETURN e) < 2
        LIMIT 1
        RETURN path.edges
```

Listing E.9: Checking PSI: Starting Vertex Detection - Filtering on Path (CycleFilter)

E.1.4 Adya's PL-2

```
FOR start IN txn
    FOR vertex, edge, path IN 2..4
        OUTBOUND start._id
        GRAPH txn_g
        FILTER path.edges[*].type NONE == "rw" AND edge._to == start._id
        LIMIT 1
        RETURN path.edges
```

Listing E.10: Checking PL-2: Starting Vertex Detection (Cycle)

```
FOR vertex, edge, path IN 2..4
    OUTBOUND @start
    GRAPH txn_g
    FILTER path.edges[*].type NONE == "rw" AND edge._to == @start
    LIMIT 1
    RETURN path.edges
```

Listing E.11: Checking PL-2: Starting Vertex Detection - Randomization (CycleRandom)

```
FOR start IN txn
    FOR vertex, edge, path IN 2..4
        OUTBOUND start._id
        GRAPH txn_g
        FILTER path.edges[*].type NONE == "rw" AND LAST(path.edges[*]._to) == start
    ._id
        LIMIT 1
        RETURN path.edges
```

Listing E.12: Checking PL-2: Starting Vertex Detection - Filtering on Path (CycleFilter)

E.1.5 Adya's PL-1

```
FOR start IN txn
    FOR vertex, edge, path IN 2..4
        OUTBOUND start._id
        GRAPH txn_g
        FILTER path.edges[*].type ALL == "ww" AND edge._to == start._id
        LIMIT 1
        RETURN path.edges
```

Listing E.13: Checking PL-1: Starting Vertex Detection (Cycle)

```
FOR vertex, edge, path IN 2..4
    OUTBOUND @start
    GRAPH txn_g
    FILTER path.edges[*].type ALL == "ww" AND edge._to == @start
    LIMIT 1
    RETURN path.edges
```

Listing E.14: Checking PL-1: Starting Vertex Detection - Randomization (CycleRandom)

```
FOR start IN txn
    FOR vertex, edge, path IN 2..4
        OUTBOUND start._id
        GRAPH txn_g
        FILTER path.edges[*].type ALL == "ww" AND LAST(path.edges[*]._to) == start
    ._id
        LIMIT 1
        RETURN path.edges
```

Listing E.15: Checking PL-1: Starting Vertex Detection - Filtering on Path (CycleFilter)

E.2 Shortest Path Detection (SP)

Shortest Path Detection (SP) is used in all isolation levels.

E.2.1 Serializability (SER)

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

Listing E.16: Checking SER: Shortest Path Detection (SP)

E.2.2 Snapshot Isolation (SI)

```
LET cycles = (
    FOR edge IN txn_dep_edges
        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 E.17: Checking SI: Shortest Path Detection (SP)

E.2.3 Parallel Snapshot Isolation (PSI)

```
LET cycles = (
    FOR edge IN txn_dep_edges
        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 LENGTH(FOR e IN cycle.edges FILTER e.type == "rw" RETURN e) < 2
    LIMIT 1
    RETURN cycle
```

Listing E.18: Checking PSI: Shortest Path Detection (SP)

E.2.4 Adya's PL-2

```
LET cycles = (
    FOR edge IN txn_dep_edges
        FILTER edge != "rw"
        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 cycle.edges[*].type NONE == "rw"
    LIMIT 1
    RETURN cycle
```

Listing E.19: Checking PL-2: Shortest Path Detection (SP)

E.2.5 Adya's PL-1

```
LET cycles = (
    FOR edge IN txn_dep_edges
        FILTER edge.type == "ww"
        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 cycle.edges[*].type ALL == "ww"
    LIMIT 1
    RETURN cycle
```

Listing E.20: Checking PL-1: Shortest Path Detection (SP)

E.3 ArangoDB Pregel SCC Algorithm (Pregel)

The Pregel SCC Algorithm only applies to checking serializability.

```
jobId, err := db.StartJob(context.Background(), driver.PregelJobOptions{
    Algorithm: driver.PregelAlgorithmStronglyConnectedComponents,
    GraphName: dbConsts.TxnGraph,
    Params: map[string]interface{}{
        "resultField":       "scc",
        "shardKeyAttribute": "_from",
        "store":             true,
    },
})

if err != nil {
    log.Fatalf("Failed to start Pregel SCC algorithm: %v\n", err)
}
```

Listing E.21: Checking SER: ArangoDB Pregel SCC Algorithm (Arango-Pregel)

E.4 Neo4j-APOC Cycle Detection

As a supplement, an example of cycle detection in Neo4j is also attached. The following Cypher query detects all the cycles in a dependency graph.

```
MATCH (n:txn) with collect(n) as nodes
CALL apoc.nodes.cycles(nodes)
YIELD path
RETURN path
```

Listing E.22: Neo4j APOC Cycle Detection (Neo4j-APOC)

E.5 Neo4j-GDS SCC Algorithm

An example of SCC detection in Neo4j is also attached. The following Cypher query detects all the SCCs in a dependency graph.

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

Listing E.23: Neo4j APOC Cycle Detection (Neo4j-APOC)

Appendix F

Dataset Characteristics of List and Register Histories

This appendix lists the characterstics of each dataset of List1-List5, Reg1-Reg4. Each column is explained below.

	#RW	#WW	#WR	#vertices	#edges	#traversals	density	#committed	#aborted
10	532	405	433	434	1370	70	3.1567	434	360
20	1048	774	790	822	2612	87	3.1776	822	753
30	1580	1248	1264	1293	4092	174	3.1647	1293	1043
40	2152	1684	1722	1743	5558	274	3.1888	1743	1443
50	2761	2047	2177	2167	6985	345	3.2234	2167	1802
60	3148	2453	2497	2547	8098	381	3.1794	2547	2215
70	3822	2908	2953	3023	9683	685	3.2031	3023	2594
80	4257	3349	3483	3458	11089	540	3.2068	3458	2938
90	4562	3625	3808	3795	11995	593	3.1607	3795	3314
100	5277	4234	4234	4269	13745	665	3.2197	4269	3625
110	5699	4416	4617	4634	14732	825	3.1791	4634	4127
120	6329	4890	5066	5099	16285	787	3.1938	5099	4471
130	6675	5240	5350	5475	17265	851	3.1534	5475	4869
140	7190	5585	5753	5880	18528	919	3.151	5880	5110
150	7785	5999	6345	6309	20129	941	3.1905	6309	5518
160	8382	6408	6666	6771	21456	1029	3.1688	6771	6016
170	8553	6798	6960	7068	22311	1025	3.1566	7068	6168
180	7885	6016	6195	6342	20096	1001	3.1687	6342	5708
190	9698	7666	7945	8012	25309	1158	3.1589	8012	7023
200	10938	8631	8864	8839	28433	1253	3.2168	8839	7134

Table F.1: Dataset characteristics: `list-collection-time` histories (List1)

	#RW	#WW	#WR	#vertices	#edges	#traversals	density	#committed	#aborted
10	245	179	202	208	626	41	3.0096	208	207
20	410	345	355	357	1110	62	3.1092	357	348
30	674	539	524	542	1737	96	3.2048	542	454
40	872	699	710	703	2281	1871	3.2447	703	541
50	1052	799	839	812	2690	180	3.3128	812	668
60	1237	1008	1024	1013	3269	187	3.227	1013	810
70	1326	1034	1051	1094	3411	174	3.1179	1094	1077
80	1559	1239	1215	1275	4013	229	3.1475	1275	1112
90	1693	1337	1352	1413	4382	9550	3.1012	1413	1296
100	1920	1528	1515	1549	4963	324	3.204	1549	1358
110	2099	1628	1693	1687	5420	292	3.2128	1687	1543
120	2312	1778	1810	1876	5900	248	3.145	1876	1655
130	2511	1952	2090	2063	6553	369	3.1764	2063	1786
140	2605	1997	2121	2115	6723	305	3.1787	2115	1945
150	2773	2097	2204	2283	7074	403	3.0986	2283	2132
160	3023	2336	2422	2490	7781	487	3.1249	2490	2251
170	3193	2459	2545	2596	8197	563	3.1576	2596	2319
180	3415	2581	2732	2764	8728	1538	3.1577	2764	2500
190	3507	2653	2886	2901	9046	577	3.1182	2901	2591
200	3845	2865	3038	3119	9748	916	3.1254	3119	2814

Table F.2: Dataset characteristics: `list-collection-time-nemesis` histories (List2)

	#RW	#WW	#WR	#vertices	#edges	#traversals	density	#committed	#aborted
10	1228	1259	1082	940	3569	20	3.7968	940	94
20	2248	2224	1923	1722	6395	28	3.7137	1722	239
30	3054	3026	2705	2418	8785	90	3.6332	2418	552
40	3754	3662	3311	3009	10727	148	3.565	3009	983
50	4485	4121	3900	3534	12506	282	3.5388	3534	1457
60	5001	4389	4101	3885	13491	335	3.4726	3885	2026
70	5223	4535	4496	4251	14254	418	3.3531	4251	2711
80	5616	4562	4548	4521	14726	677	3.2572	4521	3392
90	5728	4479	4627	4728	14834	858	3.1375	4728	4328
100	6078	4411	4771	4946	15260	891	3.0853	4946	4917
110	6257	4242	4633	5069	15132	1258	2.9852	5069	5771
120	6411	4102	4751	5304	15264	1401	2.8778	5304	6547
130	6825	4443	5132	5679	16400	1324	2.8878	5679	7044
140	6141	3778	4437	5168	14356	1609	2.7779	5168	7124
150	7294	4361	5208	6189	16863	1692	2.7247	6189	8588
160	7555	4207	5190	6319	16952	2268	2.6827	6319	9339
170	7268	3786	4914	6293	15968	1973	2.5374	6293	10329
180	4488	2146	2876	3976	9510	1167	2.3919	3976	7177
190	7245	3430	4546	6522	15221	2580	2.3338	6522	12051
200	7479	3315	4625	6672	15419	2725	2.311	6672	12827

Table F.3: Dataset characteristics: `list-rate` histories (List3)

	#RW	#WW	#WR	#vertices	#edges	#traversals	density	#committed	#aborted
10	465	460	412	373	1337	32	3.5845	373	86
20	773	766	708	631	2247	2616	3.561	631	183
30	1054	1074	962	861	3090	1249	3.5889	861	277
40	1317	1221	1178	1071	3716	74	3.4697	1071	462
50	1555	1409	1318	1238	4282	99	3.4588	1238	691
60	1563	1401	1322	1322	4286	2394	3.2421	1322	928
70	1821	1528	1495	1474	4844	4063	3.2863	1474	1118
80	1982	1588	1636	1639	5206	686	3.1763	1639	1350
90	2117	1597	1632	1704	5346	10392	3.1373	1704	1611
100	2182	1509	1684	1767	5375	405	3.0419	1767	1885
110	2212	1456	1720	1814	5388	482	2.9702	1814	2196
120	2202	1365	1659	1865	5226	455	2.8021	1865	2489
130	2354	1372	1620	1960	5346	608	2.7276	1960	2758
140	2340	1275	1511	1957	5126	540	2.6193	1957	3001
150	2368	1218	1557	2052	5143	709	2.5063	2052	3452
160	2326	1172	1522	2082	5020	631	2.4111	2082	3741
170	2291	959	1332	2037	4582	658	2.2494	2037	3954
180	2359	966	1332	2116	4657	664	2.2009	2116	4401
190	2471	1014	1484	2241	4969	1126	2.2173	2241	4568
200	2436	933	1398	2269	4767	744	2.1009	2269	4860

Table F.4: Dataset characteristics: `list-rate-nemesis` histories (List4)

	#RW	#WW	#WR	#vertices	#edges	#traversals	density	#committed	#aborted
10	594	403	448	469	1445	112	3.081	469	493
20	621	426	472	482	1519	108	3.1515	482	491
30	569	375	448	473	1392	452	2.9429	473	516
40	590	417	443	475	1450	89	3.0526	475	494
50	585	408	432	490	1425	99	2.9082	490	525
60	577	396	438	469	1411	86	3.0085	469	515
70	568	416	417	470	1401	88	2.9809	470	466
80	616	421	466	490	1503	133	3.0673	490	488
90	630	401	487	488	1518	101	3.1107	488	508
100	591	427	431	479	1449	331	3.0251	479	478
110	613	443	443	490	1499	160	3.0592	490	454
120	588	425	497	499	1510	85	3.0261	499	481
130	568	416	432	472	1416	72	3	472	480
140	599	409	485	484	1493	361	3.0847	484	496
150	600	407	444	473	1451	99	3.0677	473	472
160	569	388	433	466	1390	114	2.9828	466	468
170	572	414	426	467	1412	66	3.0236	467	495
180	531	387	398	453	1316	104	2.9051	453	526
190	558	394	407	470	1359	89	2.8915	470	523
200	581	419	496	479	1496	98	3.1232	479	506

Table F.5: Dataset characteristics: `list-histories-30s` histories (List5)

	#RW	#WW	#WR	#vertices	#edges	#traversals	density	#committed	#aborted
10	211	245	218	184	674	2	3.663	184	8
20	448	526	451	392	1425	0	3.6352	392	12
30	689	779	654	587	2122	0	3.615	587	18
40	861	990	831	741	2682	10	3.6194	741	30
50	1103	1283	1138	967	3524	2	3.6443	967	41
60	1347	1517	1301	1138	4165	6	3.6599	1138	47
70	1547	1829	1555	1366	4931	26	3.6098	1366	51
80	1852	2166	1807	1573	5825	20	3.7031	1573	51
90	1944	2233	1930	1685	6107	2	3.6243	1685	61
100	2223	2546	2268	1910	7037	8	3.6843	1910	71
110	2557	2828	2496	2117	7881	10	3.7227	2117	75
120	2658	3140	2652	2314	8450	16	3.6517	2314	63
130	2937	3452	2983	2552	9372	18	3.6724	2552	98
140	3102	3631	3080	2726	9813	14	3.5998	2726	81
150	3381	3958	3283	2912	10622	25	3.6477	2912	92
160	3641	4221	3621	3111	11483	24	3.6911	3111	97
170	3797	4396	3728	3282	11921	20	3.6322	3282	93
180	4001	4713	4052	3471	12766	30	3.6779	3471	131
190	4202	4928	4211	3651	13341	20	3.6541	3651	132
200	4408	5200	4428	3889	14036	16	3.6092	3889	160

Table F.6: Dataset characteristics: `reg-collection-time` histories (Reg1)

	#RW	#WW	#WR	#vertices	#edges	#traversals	density	#committed	#aborted
10	326	344	325	273	995	0	3.6447	273	8
20	611	784	597	556	1992	6	3.5827	556	30
30	971	1188	990	876	3149	8	3.5947	876	46
40	1294	1499	1254	1124	4047	14	3.6005	1124	76
50	1669	1878	1597	1411	5144	26	3.6456	1411	105
60	1962	2179	1903	1658	6044	34	3.6454	1658	147
70	2172	2463	2157	1879	6792	34	3.6147	1879	206
80	2504	2746	2436	2126	7686	40	3.6152	2126	214
90	2678	3147	2643	2389	8468	50	3.5446	2389	302
100	2926	3390	2874	2597	9190	60	3.5387	2597	382
110	3240	3674	3123	2835	10037	84	3.5404	2835	454
120	3408	3797	3388	2997	10593	101	3.5345	2997	552
130	3658	4026	3603	3200	11287	122	3.5272	3200	624
140	3886	4335	3814	3421	12035	120	3.518	3421	683
150	4211	4617	4226	3692	13054	160	3.5358	3692	807
160	4297	4819	4247	3835	13363	168	3.4845	3835	841
170	4527	5037	4349	3984	13913	189	3.4922	3984	968
180	4748	5177	4577	4188	14502	163	3.4628	4188	1168
190	4958	5381	4842	4341	15181	209	3.4971	4341	1238
200	5099	5592	4930	4518	15621	246	3.4575	4518	1347

Table F.7: Dataset characteristics: `reg-rate` histories (Reg2)

	#RW	#WW	#WR	#vertices	#edges	#traversals	density	#committed	#aborted
10	660	791	668	578	2119	4	3.6661	578	25
20	616	765	616	572	1997	4	3.4913	572	22
30	666	773	660	574	2099	4	3.6568	574	24
40	670	775	652	568	2097	2	3.6919	568	28
50	649	759	645	557	2053	4	3.6858	557	14
60	657	808	685	590	2150	4	3.6441	590	9
70	679	786	658	587	2123	0	3.6167	587	17
80	608	759	597	561	1964	6	3.5009	561	29
90	610	763	656	579	2029	2	3.5043	579	28
100	705	798	665	592	2168	6	3.6622	592	22
110	643	745	611	550	1999	2	3.6345	550	23
120	683	761	678	567	2122	0	3.7425	567	15
130	647	768	667	574	2082	12	3.6272	574	22
140	646	766	661	577	2073	2	3.5927	577	27
150	656	768	677	582	2101	2	3.61	582	14
160	674	780	676	570	2130	4	3.7368	570	21
170	664	778	669	570	2111	0	3.7035	570	17
180	638	761	620	559	2019	4	3.6118	559	23
190	706	773	677	577	2156	11	3.7366	577	22
200	650	766	642	576	2058	2	3.5729	576	22

Table F.8: Dataset characteristics: `reg-session` histories (Reg3)

	#RW	#WW	#WR	#vertices	#edges	#traversals	density	#committed	#aborted
1	487	0	456	605	943	6	1.5587	605	0
2	598	395	562	630	1555	4	2.4683	630	11
3	578	532	595	567	1705	6	3.0071	567	18
4	606	644	658	580	1908	4	3.2897	580	12
5	622	716	651	598	1989	10	3.3261	598	19
6	673	716	652	577	2041	4	3.5373	577	12
7	672	711	637	578	2020	8	3.4948	578	15
8	692	789	664	594	2145	2	3.6111	594	13
9	647	765	643	569	2055	4	3.6116	569	16
10	679	794	649	573	2122	2	3.7033	573	26
11	691	850	697	592	2238	4	3.7804	592	20
12	668	819	676	581	2163	2	3.7229	581	22
13	698	841	681	576	2220	0	3.8542	576	22
14	647	839	679	570	2165	0	3.7982	570	17
15	688	864	707	589	2259	6	3.8353	589	17
16	694	851	699	573	2244	2	3.9162	573	19
17	691	887	681	596	2259	0	3.7903	596	21
18	715	883	707	586	2305	0	3.9334	586	24
19	667	887	674	582	2228	2	3.8282	582	14
20	661	861	671	573	2193	0	3.8272	573	19

Table F.9: Dataset characteristics: `reg-max-write` histories (Reg4)

Appendix G

Runtime of Cycle Checker with Different Max Depths

This appendix lists the runtime of the Cycle checker on List1 dataset, with the max depth varying from 5 to 2.

Collection time (s)	Cycle (d=5)	Cycle (d=4)	Cycle (d=3)	Cycle (d=2)
10	287	66	22	10
20	585	134	42	16
30	904	237	71	21
40	1410	299	97	30
50	1618	360	106	43
60	1198	412	120	42
70	1524	544	170	52
80	1974	577	174	60
90	1912	609	184	47
100	2517	762	230	60
110	2209	826	216	66
120	2922	847	267	68
130	2608	863	288	71
140	4289	1045	425	77
150	3309	1058	318	91
160	3314	1080	352	87
170	3790	1506	403	116
180	3908	960	432	116
190	4052	1433	473	105
200	3489	1155	378	158

Table G.1: Runtime of Cycle checker with different max depths in SER checking on `list-collection-time` dataset (List1)

Collection time (s)	Cycle (d=5)	Cycle (d=4)	Cycle (d=3)	Cycle (d=2)
10	286	63	21	8
20	580	138	45	14
30	913	228	72	21
40	1412	310	100	29
50	1617	363	107	43
60	1174	426	120	43
70	1512	564	167	54
80	1978	583	175	61
90	1944	652	186	50
100	2459	765	232	62
110	2192	859	217	68
120	2896	849	265	70
130	2611	880	291	74
140	4367	1455	422	80
150	3267	1107	323	91
160	3346	1065	358	87
170	3780	1519	408	114
180	3931	999	434	115
190	4054	1460	461	106
200	3551	1186	380	156

Table G.2: Runtime of Cycle checker with different max depths in SI checking on `list-collection-time` dataset (List1)

Collection time (s)	Cycle (d=5)	Cycle (d=4)	Cycle (d=3)	Cycle (d=2)
10	296	79	37	19
20	583	143	54	23
30	941	252	87	34
40	1440	332	127	53
50	1695	412	159	70
60	1229	489	171	84
70	1772	664	256	105
80	2005	666	283	109
90	2058	722	287	114
100	2596	894	339	138
110	2364	984	345	148
120	3128	998	397	190
130	2888	1018	424	152
140	4549	1660	573	182
150	3413	1310	476	191
160	3551	1281	539	185
170	4157	1721	573	235
180	4303	1195	611	221
190	6106	2054	727	246
200	4957	1734	641	308

Table G.3: Runtime of Cycle checker with different max depths in PSI checking on `list-collection-time` dataset (List1)

Collection time (s)	Cycle (d=5)	Cycle (d=4)	Cycle (d=3)	Cycle (d=2)
10	73	28	14	8
20	146	54	26	15
30	252	97	47	23
40	367	124	64	32
50	450	147	67	45
60	285	168	74	45
70	401	227	109	55
80	524	263	118	63
90	521	276	122	47
100	655	328	151	61
110	586	354	137	68
120	748	341	171	69
130	703	367	191	69
140	1160	636	285	75
150	837	455	202	90
160	839	438	225	85
170	1064	665	273	122
180	1051	544	283	122
190	1627	820	370	103
200	1224	876	278	168

Table G.4: Runtime of Cycle checker with different max depths in PL-2 checking on `list-collection-time` dataset (List1)

Collection time (s)	Cycle (d=5)	Cycle (d=4)	Cycle (d=3)	Cycle (d=2)
10	11	7	5	5
20	21	13	9	9
30	35	23	17	13
40	53	30	23	18
50	60	34	23	25
60	42	42	26	25
70	52	52	39	30
80	71	61	41	35
90	72	65	44	27
100	96	80	55	35
110	83	85	48	36
120	102	81	62	39
130	99	87	67	39
140	163	148	101	43
150	121	118	73	51
160	122	110	81	48
170	151	157	98	69
180	152	127	102	68
190	219	185	125	58
200	170	203	101	94

Table G.5: Runtime of Cycle checker with different max depths in PL-1 checking on `list-collection-time` dataset (List1)