Shifting Transaction Isolation on Graphs: From Systems to Data

Wenzhi Fu
University of Edinburgh
Edinburgh, UK
wenzhi.fu@ed.ac.uk

Yang Cao
University of Edinburgh
Edinburgh, UK
yang.cao@ed.ac.uk

ABSTRACT

Processing long-running read-write transactions on graphs is an open challenge, primarily due to the need for serializability to maintain basic structural consistency of graphs. We identify that a fundamental impediment to a solution arises from the homogeneous database-wide notion of transaction isolation developed for relations, which fails to capture the heterogeneous consistency semantics on graphs. We propose Ddi, a notion of fine-grained isolation for graph transactions that advocates per-operation isolation allocation. It extracts concurrency for graph transactions that traditional isolation cannot, by assigning one or multiple isolation levels to each traversal operation, while maintaining graph consistency as serializability does. We develop formal semantics for Ddi and prove the consistency guarantees of its transaction schedules. We also develop DD-OCC, an optimistic concurrency control protocol for Ddi isolation, and implement it on a state-of-the-art graph storage. Experiments over LDBC graphs confirm the effectiveness of Ddi.

PVLDB Reference Format:
Wenzhi Fu and Yang Cao. Shifting Transaction Isolation on Graphs: From Systems to Data. PVLDB, 18(11): 3784 - 3796, 2025.
doi:10.14778/3749646.3749654

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://git.ecdf.ed.ac.uk/dbg/ddi.git.

1 INTRODUCTION

Graph transactions are more ubiquitous than they may initially seem. To maintain even the most basic structural consistency of graphs, *e.g.,* no dangling edge [21, 30], an edge insertion is already a read-write transaction that involves reading vertices for consistency checking prior to updating edge records. As a result, running updates on graphs concurrently demands transaction isolation to prevent potential violation of consistency requirements of graphs.

As an example, removing isolated vertices while processing concurrent edge insertions would have already needed serializability as the lowest isolation level to eliminate the possibility of inserting edges attached to an isolated vertex that is removed concurrently, a classic "write skew" anomaly that yields dangling edges in graphs.

Previous work. In light of this, there has been growing interest in developing transactional graph systems [15, 20, 21, 28, 30, 49] for efficiently updating graphs while offering serializability to maintain the structural consistency of graphs. They develop highly engineered data structures that allow flexible trade-offs between graph scan performance and update efficiency. With these, they then adapt relational concurrency control protocols to the structures, naturally migrating isolation guarantees from relations to graphs.

Limitation. Due to the imperative of serializability for maintaining the most basic structural consistency of graphs, current transactional graph systems all choose to maintain serializability for their targeted workloads. This significantly increases the chance of conflicts between transactions on graphs, in particular for transactions that involve long-running graph traversals. Indeed, under serializability, any edge insertion or deletion in the traversed area of long-running transactions would cause a prohibited read-write conflict.

Example 1: Consider a graph G where each vertex v is associated with an importance score s_v determined by its neighbor vertices, *e.g.,* aggregated from scores of its k-hop neighbors via a GNN model [32] or personalized PageRank [35]. Consider transaction $T_L(u, v)$ that first updates s_u by aggregating importance scores of vertices in its neighborhood and then inserts an edge (u, v) if s_u is greater than a threshold. Then T_L is a long-running transaction as it involves graph traversal to compute the GNN score s_u. On the other hand, G may be concurrently updated by short transactions T_S that insert or delete an edge or a vertex. To prevent G from being corrupted by dangling or duplicated edges [21, 30], one needs serializability (Sr) to execute T_L and T_S. However, under Sr, when T_S operates on vertex or edge within k-hop of u, T_L may consistently block concurrent T_S for extended period of time under lock-based concurrency control, or be hardly able to commit under optimistic concurrency control due to excessive conflicts with T_S transactions. □

As a result, existing transaction graph systems often overlook long-running read-write transactions; when such transactions occur, they either give up on transactional semantics or degrade to a serial execution. Sortledton [21], a state-of-the-art transactional graph system, only supports read-only long-running transactions, by querying a separately maintained graph snapshot.

This is also observed by a recent case study [17], jointly reported by industry and academia peers, concluding that current systems simply "presume that long running read-write transactions are rare and choose to not support them due to the significant challenges in doing so efficiently". This is despite evidence showing that such transactions are becoming increasingly prevalent in real-world workloads [17, 34, 41], largely attributed to the flexibility of graph models and query languages, which are adept at expressing complex business logic through graph traversals, analytics and updates.

Present work. We aim to tackle the open challenge of supporting long running transactions on graphs. Unlike previous research that

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749654

predominantly concentrates on data structures that facilitate efficient short updates, we argue that a fundamental impediment to a complete solution for graph transactions arises from the direct application of transaction isolation for RDBMS to graphs.

Isolation is used to assure the consistency of databases in presence of concurrent transactions. In the relational world, a database is consistent if it satisfies *any possible* integrity constraints and application-dependent constraints [38], *e.g.,* database triggers [24]. Such worst-case interpretation of consistency is inherently homogeneous, assuming that every database value is equally important.

Hence, transaction isolation and associated concurrency control techniques are all built on the practice that isolation is a global, database-wide configuration that protects all data values in a database[1]. This however is an overkill for graphs due to the consistency heterogeneity on graphs. For instance, while graph structures need serializability to protect, property values, *e.g.*, the aggregate score s_u of T_L in Example 1, often need not. Even an edge insertion may have varying implications in different places of the graph.

Data-driven isolation (Ddi). In light of this, we propose *data-driven isolation* (Ddi) for transactions on graphs. It advocates a fine-grained isolation allocation that assigns different isolation levels to different transaction operations. For instance, it can run the traversal operation (GNN computation of s_u) of T_L in Example 1 with read committed or a mix of snapshot isolation and serializability, while executing the subsequent edge insertion (u, v) with serializability. By doing so, it extracts concurrency for cases where current practice cannot, while still maintaining the consistency of graphs.

To realize this idea, it however demands a fundamentally new understanding of transaction isolation. For example, the decades old foundation of transaction isolation is built on database-wide isolation allocation, as for instance we often refer to a system as serializable. The implication of mixing isolation of read-write transactions on consistency is unclear. For instance, while serializability is a global constraint over the execution trace of entire transaction workload, other lower isolation levels are local to individual transactions; moreover, a write transaction under read committed can overwrite already committed writes governed by snapshot isolation. This is made more challenging when different operations of the same transaction have different isolation levels.

To deal with the challenge, we develop Ddi with the following.

Ddi *isolation allocation*. We present a simple consistency model that concisely captures application-specifically consistency requirements on graphs. Given any consistency abstraction, we deduce an Ddi isolation specification that assigns isolation levels to transaction operations, such that concurrent transaction execution is guaranteed to comply with any application requirements (*e.g.,* no dangling or duplicated edges) modeled by the consistency abstraction.

Formal guarantees of Ddi. Critical to Ddi is its formal semantics that give us the exact definition of when a transaction execution is valid under a given Ddi isolation specification. Although the foundation of transaction isolation has been well understood in textbooks for several decades, it is built on database-wide global isolation practice, and thus offers little help for Ddi.

To this end, we develop the formal semantics of Ddi isolation based on an implementation-independent notion of transaction schedules for multi-version (MVCC) storage. As the main result, we formally prove that concurrent transaction execution permitted by a Ddi isolation specification strictly maintains consistency semantics that Ddi observes when deducing its isolation allocation. We also study fundamental properties of Ddi isolation under this semantics, including its expressiveness, consistency and concurrency.

Ddi *concurrency control protocol*. We show that existing concurrency control can be readily extended to capitalize on Ddi, by developing DD-OCC as an proof-of-concept optimistic concurrency control protocol for Ddi. We also prove its correctness: all DD-OCC transaction execution schedules conform to Ddi isolation specification.

System GDDI. Built on Sortledton [21], the state-of-the-art transactional graph storage, we develop system GDDI that implements DD-OCC and puts Ddi isolation into action. To make better use of the fine-grained isolation of Ddi, GDDI further implements *isolation partition*, a method that "partitions" a large traversal operation of long-running transactions into multiple smaller logical sub-operations, and assigns them with different isolation levels. In doing so, GDDI is able to run a long-running traversal operation with multiple isolation levels according to traversed vertices and edges.

Using LDBC benchmark and public graphs, we experimentally validated the benefits of Ddi isolation, by comparing system GDDI with current systems over varying workloads, including those with and without long-running transactions. Results verify that GDDI is able to deal with long-running transactions that existing systems struggle with, while being comparable or better for traditional short-transaction only workloads that existing systems are built for, thanks to unique properties of Ddi and low overhead of DD-OCC.

Organization. We present the Ddi isolation allocation scheme in Section 3.2. We then develop formal foundations of Ddi in Section 4, based on which we design DD-OCC protocol and prove its correctness in Section 5. We describe system GDDI in Section 6 and experimentally evaluate its performance in Section 7. We describe related work below, present preliminaries in Section 2 and conclude in Section 8.

Related Work. We categorize related work below.

Transactional graph storage. There has been a host of work on transactional graph storage [15, 20, 21, 28, 30, 49]. They focuses on efficient concurrent graph data structures that balance scan and update performance. They target concurrent single edge insertions and deletions under serializability [21, 30]. As remarked in [17, 21], long-running transactions are an open challenge. Hence they either restrict their transaction to read-only and lower their isolation guarantees to snapshot isolation [21, 30] when long-running transactions are present, or simply give up on isolation guarantees *e.g.,* Neo4j's Graph Data Science library [6] and MemGraph's MAGE library [5].

In contrast to prior attempts, we aim to support a complete mix of all types of graph transactions, including long-running read-write ones that are left open due to their the significant challenges [17]. In addition, our techniques are based on a fundamentally new theory of fine-grained isolation rather than data structure optimizations.

[1]Strictly speaking, an isolation level has connection-wide scope and is in effect for all transactions within a database connection, unless, rarely, a transaction statement contains an isolation hint. Normally, it is set as a global configuration for the database and we often refer to database system as a *e.g.,* serializable system.

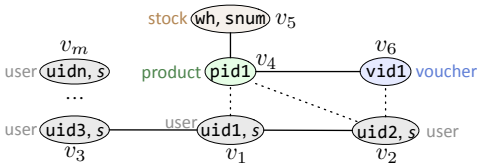


Figure 1: Graph G for running examples

Long-running transactions. Thanks to the rapid development of graph model and associated query languages [11, 23] and analytical libraries [5, 6], developers can now easily design transactions that span large parts of graphs for *e.g.,* real-time fraud detection via GNN classification [27, 42, 43], personalized recommendation [47] and community detection [31] over dynamic graphs. They demand in-place updates for real-time actions, which naturally demands transactional semantics, giving rise to long-running transactions similar to T_L of Example 1. Despite their increasing popularity, they remain underserved due to significant challenge in currency control [17, 21]. Moreover, as we will see in Section 7, even a very small fraction of such long-running transactions would significantly change the characteristics of existing transactional workloads on graphs.

While there has been little support by transactional graph system, long-running transactions have been recognized and studied since long time ago [39], often dubbed as long-lived transactions (LLTs) [22]. A typical solution, *e.g.,* Sagas [22], is to split the transaction into a sequence of smaller mini-transactions, which can then be executed separately, reducing resource occupation of LLTs [22, 37]. However, the mini-transactions are still executed with the same database-wide isolation as the original transaction does. In contrast, Ddi deals with the situation by seeking for fine-grained per-operation isolation allocation, to extract concurrency without comprising graph consistency. This said, we envisage that Sagas can potentially be incorporated in GDDI to help break down large traversals for isolation partition to work.

Mixing isolation. While we often refer to a database system as a *e.g.,* serializable system [13, 21, 38], and treat isolation levels as a property of databases, modern relational database systems in theory support per-transaction isolation allocation via *e.g.,* isolation hint in SQL statement [4]. However, the precise semantics of mixing isolation levels across transactions are rarely discussed. It has been a gray area in these systems without a universally agreed semantics.

As far as we know, there have been only two attempts to make sense of mixed isolation levels: mixing-correctness [9, 44] and serializability preserving [25]. However, they both focus on per-transaction level isolation, which is a special case of per-operation isolation that Ddi advocates. More importantly, we show in Section 4.3 that mixing-correctness is in fact ill-defined and is too weak to maintain graph consistency while serializability preserving is rather restrictive and admits much lower concurrency than Ddi does, even Ddi reduces to per-transaction isolation.

2 TRANSACTION ISOLATION ON GRAPHS

Graphs. We consider labeled undirected graphs with data values. Specifically, a graph G is a quadruple $(\mathcal{V}, \mathcal{E}, L, D)$, where (a) $\mathcal{V}$ and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ are multisets of vertices and edges, respectively, (b) L assigns a label $L(v)$ (resp. $L(e)$) for each $v \in \mathcal{V}$ (resp. $e \in \mathcal{E}$), and (c) D attaches a data value $D(v)$ (resp. $D(e)$) to each v (resp. edge e). We assume $L(\mathcal{E}) = L(\mathcal{V}) \times L(\mathcal{V})$ when $L(\mathcal{E})$ is not explicitly specified.

Table 1: Transaction operations over graph $G(\mathcal{V}, \mathcal{E}, L, D)$

op	Definition	RS(R) / WS(W)
Atomic Operations		
$\mathsf{R_V}(v)$	read and return $L(v)$ and $D(v)$ if $v \in \mathcal{V}$, or return nil otherwise	{v}
$\mathsf{R_E}(u, v)$	return True if $(u, v) \in \mathcal{E}$; or False otherwise	$\{(u, v)\}$
$\mathsf{R_\mathcal{N}}(v)$	return the set of direct neighbors of v	$\{v\} \cup \{u \mid (v, u) \in \mathcal{E}\}$ $\cup \{(v, u) \mid u \in \mathcal{V}\}$
$\mathsf{W_V}(v)$	add v to or remove v from $\mathcal{V}$, or update $D(v)$	$\{v\}$, or $\{D(v)\}$
$\mathsf{W_E}(u, v)$	add (u, v) to $\mathcal{E}$, remove (u, v) from $\mathcal{E}$ or update value $D(u, v)$	$\{(u, v)\}$
Compound (traversal) Operation		
$\mathsf{R_\mathcal{N}}(v, \ell)$	call $\mathsf{R_\mathcal{N}}(v)$ and return $\mathcal{N}(v, \ell)$: the set of direct neighbors of vertex v with label ℓ	the RS of $\mathsf{R_\mathcal{N}}(v)$
$\mathsf{R_\mathcal{N}}(S, \ell)$	given a set S of vertices and label ℓ as input, read and return $\cup_{v \in S} \mathcal{N}(v, \ell)$	$\cup_{v \in S}$RS$(\mathsf{R_\mathcal{N}}(v, \ell))$
$\mathsf{TR}_k(v)$	traverse G from v up to k hops via k-times recursive calls to $\mathsf{R_\mathcal{N}}(u)$, starting from v	union of RS of all calls to $\mathsf{R_\mathcal{N}}$
$\mathsf{TR}_k(v, \ell)$	traverse G from vertex v up to k hops via recursive calls to $\mathsf{R_\mathcal{N}}(u, \ell)$, starting from v	union of RS of all calls to $\mathsf{R_\mathcal{N}}(v, \ell)$
Update Operation (application-dependent)*		
insert(u, v) (resp. insert(v))	insert an edge (u, v) (resp. vertex v) in G	
delete(u, v) (resp. delete(v))	remove edge (u, v) (resp. vertex v) from G	

*Implementation varies depending on consistency guarantees the system upholds.

Each vertex is indexed by a vertex ID and an edge $e \in \mathcal{E}$ is indexed by the vertex IDs of u and v if $e = (u, v)$. We refer to the vertices and edges of G, *i.e.,* $\mathcal{V} \cup \mathcal{E}$, as the *data items* of G.

Example 2: Figure 1 (solid lines) depicts the graph G of Example 1. Vertices have types determined by their labels, *e.g.,* user. The value $D(v_1)$ of user vertex v_1 is its user ID uid1 and score (*e.g.,* PageRank) s; similarly $D(v_5)$ is warehouse wh and stock snum. □

Transaction model. We model the basics of graph transactions.

Atomic operations. A transaction T over graph $G(\mathcal{V}, \mathcal{E}, L, D)$ is composed of *atomic* operations $(\mathrm{op}_1, \ldots, \mathrm{op}_n)$, where each op_i is one of the 5 atomic operations in Table 1. Intuitively, each op_i either accesses set $\mathcal{V}$ or $\mathcal{E}$ (*i.e.,* read operations $\mathsf{R_V}$, $\mathsf{R_E}$ and $\mathsf{R_\mathcal{N}}$), or changes their elements (*i.e.,* write operations $\mathsf{W_V}$ and $\mathsf{W_E}$).

Update operations. We shall remark that $\mathsf{W_E}(u, v)$ is an atomic operation that adds (u, v) to set $\mathcal{E}$ or changes its value. However, $\mathsf{W_E}(u, v)$ does not necessarily express an edge insertion, which in most cases is not atomic and involves more than one atomic operations, depending on the consistency guarantees that the system upholds. When their implementation with atomic operations is irrelevant, we simply write insert(u, v) and insert(v) for edge insertion and vertex insertion, respectively; the same for deletion. To distinguish them from $\mathsf{W_E}$ and W_V, we refer to them as update operations in Table 1.

Access sets. The *access set*, also referred to as the read-set RS(R) of a read operation R or the write-set WS(W) of a write operation W, is the set of vertices and edges appeared in the operation (see Table 1 for details). Two operations are in conflict, or called conflicting, if their access sets overlap and one of them is a write, *i.e.,* $\mathsf{W_V}$ or $\mathsf{W_E}$.

Graph transactions. A graph transaction T over G is a pair $(C, \mathcal{T})$, where $C \subseteq \mathcal{V}$ and $\mathcal{T}$ is a sequence of atomic operations $\mathrm{op}_1, \ldots, \mathrm{op}_n$, such that for each $i \in [1, n]$, for each input v of op_i, either v is from C or there exists a read operation op_j in $\mathcal{T}$ with $j < i$ and $v \in \mathsf{RS}(\mathrm{op}_j)$. Intuitively, the input parameter of each operation in the transaction can only be specified by the input vertices in C or be retrieved by an earlier operation in the transaction.

The read-set of transaction T, denoted by $\mathtt{RS}(T)$, is the union of the read-sets of all reads in T; similarly for its write-set $\mathtt{WS}(T)$. Two transactions are in conflict if their read or write sets overlap.

Traversals and long-running transactions. Long-running graph transactions are often induced by compound operations that build atop atomic operations, particularly $\mathsf{R}_{\mathcal{N}}$. They follow a two-stage "traverse-and-aggregate" pattern, like T_L of Example 1: it first traverses G starting from a vertex v via *e.g.,* BFS (Breadth-First Search), up to k rounds; it then invokes an analytical algorithm, oftentimes an aggregation function, *e.g.,* PageRank or a GNN model, to aggregate over the data values of all vertices read by the traversal.

We simply refer to such compound operations as a *traversal operation,* denoted by $\mathsf{TR}_k(v)$, where v is the start vertex called origin; its output is the set of all vertices within k-hops from v (see Table 1).

Example 3: An instantiation of T_L of Example 1 over graph G in Fig. 1 (with dotted lines) is $T_L(v_1, v_4) = (\{v_1, v_4\}, \mathcal{T})$, where $\mathcal{T}$ is $(v_* := \mathsf{R}_{\mathcal{N}}(v_4, \mathtt{stock}), S := \mathsf{TR}_3(v_1, \mathtt{user}), \mathsf{W}_{\mathsf{V}}(v_1), \mathtt{insert}(v_1, v_4), \mathsf{W}_{\mathsf{V}}(v_*))$. Intuitively, $T_s(v_1, v_4)$ seeks to sell $\mathtt{pid1}$ (v_4) to user $\mathtt{uid1}$ (v_1). It first checks the in-stock number (snum) of $\mathtt{pid1}$ via $\mathsf{R}_{\mathcal{N}}(v_4, \mathtt{stock})$ to assure sufficient stock. It then traverses 3-hop neighbor users of v_1 via $\mathsf{TR}_3(v_1, \mathtt{user})$, computes the aggregate score s_u of v_1 over the traversed subgraph and updates it via $\mathsf{W}_{\mathsf{V}}(v_1)$, "sells" v_4 to v_1 via $\mathtt{insert}(v_1, v_4)$, and decreases snum of $\mathtt{pid1}$ by 1 via $\mathsf{W}_{\mathsf{V}}(v_*)$. Here $\mathtt{insert}(v_1, v_4)$ is a transaction that inserts edge (v_1, v_4); we will discuss its transactional implementation shortly. □

Isolation of graph transactions. To understand the choice of serializability for graph transactions, we need to start with consistency of graphs. A consistency constraint over graph G is a logic sentence over vertices and edges of G that we need to maintain invariant when we update G (via update operations in Table 1).

Example 4: We illustrate some common consistency constraints over graph G of Fig. 1. (a) A *functional dependency* over G is voucher → user, which states that a voucher can be connected to at most one user, *i.e.,* no double allocation. (b) A *duplicated edge* constraint over G is to require that no two $\mathtt{user}$-$\mathtt{product}$ edges could connect the same pair of $\mathtt{user}$ and $\mathtt{product}$ vertices, *i.e.,* no double charging. (c) A *dangling edge* constraint asserts that if for instance $(v_1, v_4) \in \mathcal{E}$ of G, then $v_1 \in \mathcal{V}$ and $v_4 \in \mathcal{V}$. (d) A *value constraint* over G is snum ≥ 0, which states that product stock cannot be negative. □

The transactional implementation of $\mathtt{insert}$ depends on the consistency constraints of concern. To avoid dangling or duplicated edges, edge insertion $\mathtt{insert}(v_1, v_4)$ of T_L in Example 3 would need already be a read-write transaction that checks if (v_1, v_4) is not in $\mathcal{E}$ and if both v_1 and v_4 are in $\mathcal{V}$, before adding (v_1, v_4) to $\mathcal{E}$, written as:

$$T^{(1)}_{\mathsf{ins}}(v_1, v_4) = (\{v_1, v_4\}, (\mathsf{R}_{\mathsf{V}}(v_1), \mathsf{R}_{\mathsf{V}}(v_4), \mathsf{R}_{\mathsf{E}}(v_1, v_4), \mathsf{W}_{\mathsf{E}}(v_1, v_4))).$$

If we further consider functional dependency $\mathtt{voucher} \rightarrow \mathtt{user}$, then $\mathtt{insert}(v_1, v_4)$ would need even more steps, as $T^{(2)}_{\mathsf{ins}}(v_1, v_4)$:

$$(\{v_1, v_4\}, (\mathsf{R}_{\mathsf{V}}(v_1), \mathsf{R}_{\mathsf{V}}(v_4), \mathsf{R}_{\mathsf{E}}(v_1, v_4), \mathsf{R}_{\mathcal{N}}(v_4, \mathtt{user}), \mathsf{W}_{\mathsf{E}}(v_1, v_4))),$$

where $\mathsf{R}_{\mathcal{N}}(v_4, \mathtt{user})$ checks if v_4 is linked to another $\mathtt{user}$ already.

Why serializability? Transactional graph systems thus would have already needed serializability for concurrent edge insertions to maintain consistency. Otherwise, a "write-skew" anomaly could allow the system to insert two edges (v_1, v_4) and (v_2, v_4) concurrently via $T^{(2)}_{\mathsf{ins}}$, breaking functional dependency $\mathtt{voucher} \rightarrow \mathtt{user}$.

3 RETHINKING ISOLATION FOR GRAPHS

In this section, we propose a method that alleviates the severe toll of serializability for long-running transactions on graphs. Our approach maintains essential isolation to protect against concurrency anomalies that could compromise consistency, but is able to extract more concurrency that is not possible with existing approaches.

The underpinning proposition is that we *do not need serializability everywhere* due to the heterogeneous nature of consistency on graphs; accordingly, the common practice of a *system-wide* isolation level is overkill for graphs. Instead, we advocate for a *fine-grained* isolation allocation scheme that assigns tailored isolation levels to different transaction operations based on the data they touch.

To elaborate on this, we present a consistency model that captures heterogeneous consistency semantics on graphs (Section 3.1) and outline our fine-grained isolation allocation scheme (Section 3.2).

3.1 Consistency: Vertices are Not Equal

We present a consistency model that captures the heterogeneity of consistency for graphs and graph transactions.

Consistency on graphs is about constraints over vertices and edges that must not violate in any case, to prevent graphs from being corrupted by concurrent updates. Formally, a consistency constraint ϕ over graph $G(\mathcal{V}, \mathcal{E}, L, D)$ is a first-order logic sentence whose variables represent vertices and edges in $\mathcal{V}$ and $\mathcal{E}$, with predicates for (a) equality and adjacency/association testing over vertices and edges [40], and (b) logical operators over labels $L(\mathcal{V})$ and values $D(\mathcal{V})$ [33]. Here *adjacency* test is a predicate $\mathcal{E}(u, v)$ that tells if two vertices u and v are adjacent in the graph, and *association* predicate decides if a vertex is an endpoint of an edge. Adjacency predicate can be expressed via association and vice versa [40].

Example 5: Functional dependency $\mathtt{voucher} \rightarrow \mathtt{user}$ of Example 4 can be expressed as $\phi(x_u, x_v, x_w) = \forall x_u, x_v, x_w \mathcal{V}(x_u) \wedge \mathcal{V}(x_v) \wedge \mathcal{V}(x_w) \wedge L(x_u) = \mathtt{voucher} \wedge L(x_v) = \mathtt{user} \wedge L(x_w) = \mathtt{user} \wedge (\mathcal{E}(x_u, x_v) \wedge \mathcal{E}(x_u, x_w) \rightarrow x_v = x_w)$, where $\mathcal{V}(x_u)$ is a predicate stating x_u is a vertex in $\mathcal{V}$; similarly for $\mathcal{E}(x_u, x_v)$. □

Rather than enumerating all consistency constraints via logic sentences like $\phi(x_u, x_v, x_w)$ in Example 5, we define *graph consistency rules*, a concise abstraction that abstracts away the semantics (predicates) of ϕ. Specifically, $\phi(x_1, \ldots, D(x_n), \ldots)$ is modeled by a consistency rule $\Phi(L_1, \ldots, D(L_m), \ldots)$ $(L_i \subseteq \mathcal{L}(\mathcal{V}) \cup \mathcal{L}(\mathcal{E}))$, where $D(L)$ denotes the value for all data items with label L, if there exist a injective mapping f that can map each of $x_i/D(x_i)$ to one of $L_i/D(L_i)$. Here ϕ is called an n-ary constraint and Φ is an m-ary.

Example 6: Recall the example consistency constraints of Example 4. From Example 5, we know that consistency (a) is modeled by rule $\Phi_a(\mathtt{voucher}, \mathtt{user}, \mathtt{user})$. Along the same lines, consistency (b) is modeled by rule $\Phi_b(\mathtt{product}, \mathtt{product}, \mathtt{user}, \mathtt{user})$. The no dangling edge of consistency (c) can be modeled by rule $\Phi_c(L(\mathcal{V}), L(\mathcal{E}))$. Finally, (d) is modeled by 1-ary rule $\Phi_d(D(\mathsf{snum}))$. □

3.2 Isolation: Conflicts are Not Equal

We next show how the consistency model helps decide isolation.

Ddi **isolation allocation scheme**. Given a set Γ of transactions and a set Γ of graph consistency rules over G, Ddi allocates isolation levels to the operations of transactions in Γ as follows.

Step (1): Isolating write operations. Ddi first assigns an isolation level to each atomic write operation op_W according to consistency rules. Specifically, an operation op matches L_i of $\Phi(L_1, \ldots, L_n)$ for some $i \in [1, n]$ if op operates on data item x with label from L_i. op on x can be matched to L_i or $D(L_i)$ but that on $D(x)$ can only map to $D(L_i)$. A transaction T matches Φ if each L_i of Φ is matched by an operation of T. We say that write operation op_W of transaction T is *covered* by Φ if (a) T matches Φ and (b) op_W matches some L_i of Φ.

If op_W is covered by a multi-ary (resp. 1-ary) consistency rule, then Ddi assigns op_W with serializability (resp. snapshot isolation). Otherwise, it is set to read committed. Note that op_W can be covered by multiple rules and thus has multiple isolation levels assigned. In such cases, Ddi picks the highest isolation level for op_W.

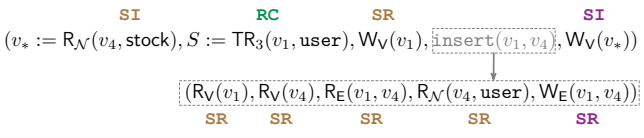
Step (2): Isolation propagation. Ddi then propagates the isolation of each write operation op_W to each read operation op_R that precedes them in the same transaction if op_W depends on op_R. Here op_1 depends on op_2 if (a) they are in the same transaction and (b) op_1 reads a vertex or edge returned or written by op_2, or op_1 reads a vertex or edge returned or written by op_3 and op_3 depends on op_2.

Similar to step (1), op_R can be assigned with multiple isolation levels propagated from different op_W and Ddi picks the highest.

Step (3). Ddi assigns the lowest isolation level (Rc by default) to the remaining operations as they cannot violate consistency.

Remark. Ddi isolation allocation is *local*. The isolation levels of operations in T do not depend on other transactions. This implies that Ddi isolation can be determined on-the-fly during execution.

Example 7: Consider T_L of Example 3. Assume that we consider all consistency semantics of Example 4. Then the implementation of $\texttt{insert}(v_1, v_6)$ is $T^{(2)}_{\text{ins}}$ in Section 2. According to Example 6, Ddi allocates isolation to operations of T_L as follows:



Here annotations in purple and brown colors are assigned by step (1) and step (2), respectively; green isolation is allocated by step (3). □

The power of Ddi isolation. Below we informally present the benefits of Ddi. We will formalize the statement and give a proof by developing a precise definition of Ddi isolation in Section 4.

Theorem 1 [Informal]: Ddi *maintains the consistency of graphs while enabling more concurrency than uniform isolation allocation.* □

Theorem 1 states that the per-operation isolation allocation of Ddi upholds all consistency constraints modeled by the consistency rules, while permitting more transaction schedules than any single isolation level that also complies with the consistency constraints.

4 FOUNDATIONS OF DDI

Below we give the formal semantics of Ddi isolation (Section 4.1). We then study its properties and prove Theorem 1 (Sections 4.2-4.3).

4.1 Formal Semantics of Ddi Isolation

We characterize transaction schedules permitted by Ddi isolation.

Ddi **specification**. A Ddi isolation specification of Γ is a function that maps operations of transactions in Γ to the set of isolation levels $\mathbb{I}$ = {Serializability (Sr), Snapshot Isolation (Si), Read Committed (Rc)}, according to the Ddi allocation scheme of Section 3.

Transaction schedules. A *schedule* of a set Γ of transactions is defined by a *time assignment function* τ that associates each operation op of each $T \in \Gamma$ with a time point which represents its execution time. Following [25, 29, 46], we focus on optimistic concurrency control (OCC) based systems given its wide presence. Under OCC, every transaction execution is carried out first at a local working copy, where values are retrieved from the global data graph upon request, and a transaction commits the changes of its operations in the local copy to the global graph at the end of the transaction execution.

Consequently, $\tau(\text{op})$ assigns each op of transaction T with time point at beginning of the execution of op if op is a read operation, or the end of T if op is a write operation. Note that this is a depart from the conventional case where a transaction runs under the same isolation level from start to end and thus time assignment for read operations under snapshot isolation is the start of the entire transaction.

Ddi **schedules**. A transaction schedule τ *conforms to* a Ddi isolation specification if it disallows certain transaction conflicts (defined below). A concurrency control protocol is Ddi-*compliant* if, for any transactions Γ and any Ddi specification of Γ, it always generates transaction schedules τ for Γ (*i.e.,* time assignment functions τ for Γ) that conform to the Ddi isolation specification of Γ.

We define the conformance of τ by adopting the notion of *Direct Serialization Graph* (DSG) [10], an implementation-independent notion for describing traditional isolation levels.

Data-driven DSG. The data-driven DSG (DD-DSG) of a schedule τ of transactions Γ *w.r.t.* a Ddi specification is a graph where vertices are transactions in Γ and edges correspond to pairs of transactions.

More specifically, for a transaction $T \in \Gamma$ and an isolation level $l \in \mathbb{I}$ = {Sr, Si, Rc}, let $\texttt{RS}_l(T)$ denote the subset of $\texttt{RS}(T)$ that consists of data items read by T via a read operation assigned with isolation level l in its Ddi specification. Similarly, we define $\texttt{WS}_l(T)$ as the set of data items of T that are written with isolation l.

Then there are three types of edges in DD-DSG. For any $l \in \mathbb{I}$:

(1) $T_i \xrightarrow{\texttt{rw}_l} T_j$ *(l-anti-depend edge).* If T_i reads some data item x and T_j is the next writer of x according to τ, and $x \in \texttt{RS}_l(T_i)$, then an l-anti-depend edge from T_i to T_j is in the DD-DSG.

(2) $T_i \xrightarrow{\texttt{ww}_l} T_j$ *(l-write-depend edge).* If T_i writes data item x and T_j is the next writer of x according to τ, and $x \in \texttt{WS}_l(T_j)$, then an l-write-depend edge from T_i to T_j is in the DD-DSG.

(3) $T_i \xrightarrow{\texttt{wr}_l} T_j$ *(l-read-depend edge).* If T_i is the last writer of some x before T_j reads x written by T_i, and $x \in \texttt{RS}_l(T_j)$, then is an l-read-depend edge from T_i to T_j is in the DD-DSG of the schedule of Γ.

Here T_i is called the source transaction and T_j is the destination.

Remarks. Observe the following about DD-DSG.

(1) Compared to the traditional DSG [10, 25], edges of DD-DSG are further annotated with an isolation level l from $\mathbb{I}$. As a result,

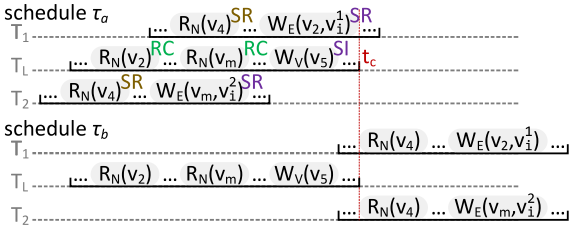


Figure 2: Schedules τ_a and τ_b for Example 8 and Example 9

there can be multiple edges from T_i to T_j of the same type (*e.g.*, read/write-depends) but annotated with different isolation levels, *e.g.*, both $T_i \xrightarrow{\mathtt{wr}_{\mathrm{SR}}} T_j$ and $T_i \xrightarrow{\mathtt{wr}_{\mathrm{SI}}} T_j$ can exist in DD-DSG.

(2) The isolation annotation of read/write-depends edge is determined by Ddi isolation specification of the destination transaction in the schedule, *i.e.*, T_j in $T_i \xrightarrow{\mathtt{wr}_l} T_j$ and $T_i \xrightarrow{\mathtt{ww}_l} T_j$; in contrast, the isolation annotation for anti-depends edge is determined by isolation of the source transaction, *i.e.*, T_i in $T_i \xrightarrow{\mathtt{rw}_l} T_j$.

Example 8: Figure 2 shows two schedules for three transactions. Along with T_L, there are also two concurrent edge insertion transactions. One is $T_1 = T_{\mathrm{ins}}^{(3)}(v_2, v_4)$, where $T_{\mathrm{ins}}^{(3)}(v_2, v_4) = (\{v_2, v_4\}, (v_i^1 := \mathsf{R}_{\mathcal{N}}(v_4), \mathtt{insert}(v_2, v_i^1)))$. It checks the stock number snum of product pid1, and allocates a voucher v_i^1 to user uid2 if stock is excessive. Similarly, $T_2 = T_{\mathrm{ins}}^{(3)}(v_m, v_4)$ performs the same for v_m.

The DD-DSG of τ_a is shown in Fig. 3(a), assuming that v_m is 3-hop away from v_1. Note that, in schedule τ_a, T_2 commits after $\mathsf{R}_{\mathcal{N}}(v_m)$ in T_L. Under OCC, all writes take effect at the commit time. For this reason, we have $T_L \xrightarrow{\mathtt{rw}_{\mathrm{RC}}} T_2$ due to v_m, and $T_2 \xrightarrow{\mathtt{rw}_{\mathrm{SR}}} T_L$ due to conflict on v_4. Similarly, conflicts between T_1 and T_s lead to $T_1 \xrightarrow{\mathtt{rw}_{\mathrm{SR}}} T_L$ and $T_L \xrightarrow{\mathtt{rw}_{\mathrm{RC}}} T_1$ in the DD-DSG of Fig. 3(a). □

Ddi **conformance**. A schedule τ of Γ conforms to a Ddi isolation specification if the corresponding DD-DSG satisfies both below:

(C1) There exists no directed cycle in DD-DSG that contains an Sr-anti-depend edge $T_i \xrightarrow{\mathtt{rw}_{\mathrm{SR}}} T_j$ and T_j commits before T_i.

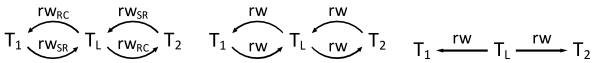
(C2) For any T_i and T_j of Γ that are concurrent in τ, if $T_i \xrightarrow{\mathtt{ww}_l} T_j$ or $T_i \xrightarrow{\mathtt{wr}_l} T_j$ is an edge in DD-DSG, then l must be Rc.

Example 9: The DD-DSG of τ_a, as shown in Fig. 3(a) contains two cycles: $T_L \xrightarrow{\mathtt{rw}_{\mathrm{RC}}} T_2 \xrightarrow{\mathtt{rw}_{\mathrm{SR}}} T_L$ and $T_L \xrightarrow{\mathtt{rw}_{\mathrm{RC}}} T_1 \xrightarrow{\mathtt{rw}_{\mathrm{SR}}} T_L$. However, it is a Ddi schedule and satisfies **C1**, although the cycle contains a $\mathtt{rw}_{\mathrm{SR}}$ edge. In contrast, the traditional DSG for τ_a, as shown in Fig. 3(b), cannot distinguish the two cycles and would disallow it for Sr. In fact, under Sr, much less concurrency is permitted as shown by Sr schedule τ_b in Fig. 2, of which the DSG is given in Fig. 3(c). □

This gives a precise definition of Ddi schedules, *i.e.*, the exact class of schedules permitted by a given Ddi isolation specification.

4.2 Correctness Guarantees of Ddi isolation

Consistency guarantee. With the formal definition of Ddi schedules in Section 4.1, we now prove Theorem 1. We start with the first part of Theorem 1: Ddi isolation allocation scheme in Section 3.2 always produces a Ddi isolation specification that prevents any violation of consistency constraints modeled by the consistency rules.



(a) DD-DSG for τ_a (b) DSG for τ_a (c) DSG for τ_b

Figure 3: DD-DSG and DSG for τ_a and τ_b

A set Γ of transactions is well-programmed if the serial execution of Γ in any order (*i.e.*, any schedule τ that schedules transactions one after another) will not break any consistency constraints modeled by data consistency rules of the consistency specification.

Let Γ be a set of well-programmed transactions and C be a consistency specification of Γ. Let Γ^C be the isolation specification generated by the Ddi isolation allocation scheme of Section 3.2 for Γ and C. The Theorem below validates the first part of Theorem 1.

Theorem 2: *For any graph G and schedule τ of Γ that conforms to Γ^C, τ does not violate any consistency constraint captured by C.* □

See the full version [8] for the proof of Theorem 2.

Fine-grained anomalies. Traditional concurrency anomalies are restricted to the database-wide uniform isolation assumption, and thus cannot capture anomalies that may arise for fine-grained per-operation isolation allocation. Below we present Dd-*anomalies*, which are concurrency anomalies in the context of fine-grained per-operation isolation allocation. Using **(C1)** and **(C2)**, we also show how they are prevented by Ddi-compliant concurrency control.

Dd-FracturedRead (Dd-FR). Given two data items x and y and two transactions T_i and T_j, a Dd-FR anomaly occurs if T_i writes both x and y, T_j reads its update on x but not that on y, **and** $x, y \in \mathtt{RS}_{\mathbf{SI}}(T_j) \cup \mathtt{RS}_{\mathbf{SR}}(T_j)$**.** Note that this is an fine-grained, more expressive variant of the Fractured Read (FR) anomaly [3] we have in database-wide isolation, which occurs if T_i writes both x and y and T_j reads its update on x but not on y. Indeed, if y is accessed with Rc by T_j, then T_j is allowed to commit without seeing the update on y from T_i; however, in contrast to Dd-FR, this will be captured by FR.

Prevention by Ddi. We show that any Ddi-compliant concurrency control protocol can prevent Dd-FR. Assume that Dd-FR occurs in a schedule τ. Consider the DD-DSG of τ. Since the write on x from T_i is read by T_j and $x \in \mathtt{RS}_{\mathrm{SI}}(T_j) \cup \mathtt{RS}_{\mathrm{SR}}(T_j)$, there must exist an edge $T_i \xrightarrow{\mathtt{wr}_{\mathrm{SR/SI}}} T_j$ in DD-DSG. As T_j does not see the write on y from T_i, T_i must commit after T_j starts, *i.e.*, T_i and T_j are concurrent:

T_j start $\longrightarrow$ read y in $T_j \longrightarrow T_i$ commit $\longrightarrow$ read x in $T_j \longrightarrow T_j$ commit.

This violates condition **(C2)** and will be disallowed and prevented.

Dd-Non-repeatableRead (Dd-NR). Given a data item x and a transaction T, a Dd-NR anomaly occurs if T reads x multiple times but return different values **and** $x \in \mathtt{RS}_{\mathbf{SR}}(T) \cup \mathtt{RS}_{\mathbf{SI}}(T)$. For instance, any $x \in \mathtt{RS}_{\mathrm{RC}}(T)$ should be allowed to be read multiple times and return different values since it is read with Rc in T. Dd-NR can permit this valid case while disallowing non-repeatable reads if T is protected with Sr or Si. The difference however cannot be observed by conventional non-repeatable read anomaly.

Prevention by Ddi. Given a transaction T_i and data item $x \in \mathtt{RS}_{\mathrm{SI}}(T_i) \cup \mathtt{RS}_{\mathrm{SR}}(T_i)$. If there are two reads on x by T_i that return different values, there must exist another transaction T_j that installs its write on x between these two reads; the second read on x by T_i thus loads the value written by T_j. Hence, T_i and T_j are concurrent and there exists edge $T_j \xrightarrow{\mathtt{wr}_l} T_i$ in the DD-DSG with l = Sr or Si. This violates **(C2)** and thus can be prevented by any Ddi-compliant protocol.

Dd-WriteSkew (DD-WS). For two transactions T_i and T_j, a Dd-WS anomaly occurs when T_i and T_j are concurrent and are both trying to commit their write simultaneously. Formally, this means $\texttt{WS}(T_i) \cap \texttt{WS}(T_j) = \emptyset$, $\texttt{WS}(T_i) \cap \texttt{RS}_{\text{Sr}}(T_j) \neq \emptyset$ and $\texttt{RS}_{\text{Sr}}(T_i) \cap \texttt{WS}(T_j) \neq \emptyset$.

To see how it differs the traditional write skew (WS) anomaly, consider the following example. Let T_i and T_j be two concurrent transactions and x, y are data items they operate on:

- T_i writes x and T_j writes y, *i.e.,* $\texttt{WS}(T_i) = \{x\}$ and $\texttt{WS}(T_j) = \{y\}$.
- T_i reads y with Si or Rc and T_j reads x with Si or Rc, *i.e.,* $y \in \texttt{RS}(T_i) \setminus \texttt{RS}_{\text{Sr}}(T_i)$ and $x \in \texttt{RS}(T_j) \setminus \texttt{RS}_{\text{Sr}}(T_j)$.

In this case, this is a WS anomaly but not a Dd-WS anomaly. However this should be a valid schedule that we would like to commit as reads on x and y are not instructed to use Si or Rc rather than Sr.

Prevention by Ddi. Let τ be a schedule in which Dd-WS occurs. Then there must exist data items x, y and concurrent T_i, T_j, such that:

- $x \in \texttt{WS}(T_i) \setminus \texttt{WS}(T_j)$ and $y \in \texttt{WS}(T_j) \setminus \texttt{WS}(T_i)$.
- $y \in \texttt{RS}_{\text{Sr}}(T_i)$ and $x \in \texttt{RS}_{\text{Sr}}(T_j)$.

As T_i does not see the update on y from T_j, it will then form a path from T_i toward T_j in the DD-DSG of τ, with one $\texttt{rw}_{\text{Sr}}$ edge and multiple (or 0) $\texttt{ww}_l$ edges, as $T_i \xrightarrow{\texttt{rw}_{\text{Sr}}} T_p \xrightarrow{\texttt{ww}_l} \ldots \xrightarrow{\texttt{ww}_l} T_j$. Meanwhile, as T_j also does not see the update on x from T_i, there will be a path of opposite direction from T_j to T_i, with one $\texttt{rw}_{\text{Sr}}$ edge and multiple (or 0) $\texttt{ww}_l$ edges, as: $T_j \xrightarrow{\texttt{rw}_{\text{Sr}}} T_q \xrightarrow{\texttt{ww}_l} \ldots \xrightarrow{\texttt{ww}_l} T_i$. Together, these two paths then form a cycle:

$$T_i \xrightarrow{\texttt{rw}_{\text{Sr}}} T_p \xrightarrow{\texttt{ww}_l} \ldots \xrightarrow{\texttt{ww}_l} T_j \xrightarrow{\texttt{rw}_{\text{Sr}}} T_q \xrightarrow{\texttt{ww}_l} \ldots \xrightarrow{\texttt{ww}_l} T_i$$

In OCC protocol, if there exists an edge $T_m \xrightarrow{\texttt{ww}_l} T_n$ in DD-DSG, then T_n must commit after T_m. In this case, as the only non-ww edges in the cycle are $T_i \xrightarrow{\texttt{rw}_{\text{Sr}}} T_p$ and $T_j \xrightarrow{\texttt{rw}_{\text{Sr}}} T_q$, then either T_p commits before T_i or T_q commits before T_j (otherwise, T_i would commit after T_i). Therefore, the cycle violates **(C1)** and τ will be disallowed.

4.3 Concurrency Benefits of Ddi Isolation

Concurrency. To complete the proof of Theorem 1, we further show that, for the same set Γ of transactions, Ddi isolation gives us higher concurrency than any traditional mono-isolation allocation could offer without compromising consistency.

Recall Γ, C and Γ^C from above. We say that a single isolation level $l \in \mathbb{I}$ upholds the consistency specification C of transactions Γ over G if running transactions in Γ concurrently under l maintains all consistency constraints captured by C.

Lemma 3: *The set of schedules conforming to* Ddi *isolation* Γ^C *for* Γ *and* C *is larger than that of any isolation* $l \in \mathbb{I}$ *that upholds* C. □

Mixing isolation. As a by-product, Ddi isolation gives us a precise definition of mixing *per-transaction* isolation levels, a special case of Ddi where operations of the same transaction are allocated with the same isolation. Moreover, below we show that it is provably better than *mixing-correctness* [9, 44] and *serializability preserving* [25], the only two attempts to define mixed isolation levels as far as we know.

Ddi *vs. mixing-correctness*. We first recast the notion of mixing-correctness [9, 44] using our notion of DD-DSG in Section 4.1. Given a per-transaction isolation specification of a set Γ of transactions, *i.e.,* each $T \in \Gamma$ is assigned uniformly with an isolation level $l \in \mathbb{I}$, a schedule of Γ is mixing-correct [9, 44] if its DD-DSG does not contain a cycle and satisfies **(C2)** after removing the following types of edges: (a) $T_i \xrightarrow{\texttt{wr}_{\text{Rc}}} T_j$; (b) $T_i \xrightarrow{\texttt{rw}_{\text{Rc}}} T_j$ and $T_i \xrightarrow{\texttt{rw}_{\text{Si}}} T_j$.

Unfortunately, we show that mixing-correctness is in fact ill-defined and too weak to maintain data consistency. Let Γ be set of transactions and $\Gamma^{\mathbb{I}}$ be a per-transaction isolation specification of Γ.

Theorem 4: *(1) For any* Γ *and per-transaction isolation specification* $\Gamma^{\mathbb{I}}$ *of* Γ*, any* Ddi *schedule* τ *that conforms to* $\Gamma^{\mathbb{I}}$ *is also mixing-correct. (2) There exists* Γ, $\Gamma^{\mathbb{I}}$ *of* Γ *and a* 1*-ary consistent constraint* ϕ *such that (a)* ϕ *is maintained by any* Ddi *schedule that conforms to* $\Gamma^{\mathbb{I}}$*, but (b) there is a mixing-correct schedule of* $\Gamma^{\mathbb{I}}$ *that violates* ϕ. □

Ddi *vs. serializability-preserving*. Serializability-preserving [25] is another attempt to mix per-transaction isolation, by tightening the condition of serializability. Recast in DD-DSG, a schedule for a per-transaction isolation specification is serializability-preserving if (a) it does not contain an Sr-anti-depend edge $T_i \xrightarrow{\texttt{rw}_{\text{Rc}}} T_j$ in which T_j commits earlier than T_i and (b) it satisfies condition **(C2)**. We show below that, unlike mixing-correctness, serializability-preserving schedules do not violate consistency; however, the notion is rather restrictive and admits fewer schedules than Ddi does.

Theorem 5: *For any* Γ *and per-transaction isolation specification* $\Gamma^{\mathbb{I}}$*, (1) any serializable-preserving schedule for* $\Gamma^{\mathbb{I}}$ *must also be a* Ddi *schedule that conforms to* $\Gamma^{\mathbb{I}}$*; and (2) there exist* Ddi *schedule that conforms to* $\Gamma^{\mathbb{I}}$ *but is not serializability-preserving.* □

See full version [8] for proofs of Lemma 3 and Theorems 4-5.

5 CONCURRENCY CONTROL FOR Ddi

In this section, we develop a concurrency control protocol for Ddi.

One can adapt OCC protocols [38] from RDBMS to support graph transactions, by treating vertices and edges as tuples. However, it is nontrivial to extend it and support Ddi isolation given its stark contrast to traditional isolation as shown in Section 4.

As a proof of concept, below we present DD-OCC, an OCC protocol dedicated for Ddi isolation. Its main property is summarized below.

Theorem 6: DD-OCC *is* Ddi*-compliant.* □

Below we present the design of DD-OCC. We defer the proof of Theorem 6 with DD-OCC in the full version [8]due to space limit.

OCC for Ddi. DD-OCC builds upon TicToc [46], a popular modern timestamp-based OCC protocol for relational transactions. TicToc differs from classic OCC protocols by lazily computing a timestamp for each transaction based on its local data items, rather than allocating it globally. DD-OCC inherits this design and extends it to support the per-operation isolation specification in $\Gamma^{\mathbb{I}}$.

Item set encoding. To support Ddi, we extend the data item encoding of TicToc to additional record data values and isolation specification. More specifically, each data item is encoded as ⟨*item*, *data*, *wts*, *rts*, *iso-level*⟩ in the local working space, where *item* is a pointer toward the data item in global data graph, which can be a vertex or an edge, *wts* and *rts* are the write and read timestamp of the item when it is loaded into the local working space from the global data graph,

ALGORITHM 1: Validation phase in DD-OCC

```
   Data: Read sets RS, write sets WS
   # Step 1 - Lock Write Set
1  for w in sorted(WS) do
       // lock on the write set, no matter whether it is vertex or edge
2      lock(w.item)
   # Step 2 - Compute the Commit Timestamp
3  set both commit_wts and commit_rts to 0
4  for e in WS ∪ RS do
5      if e in WS then
6          commit_wts ← max(commit_wts, e.item.rts +1)
7      commit_rts ← max(commit_rts, e.wts)
8  commit_ts ← max(commit_rts, commit_wts)
   # Step 3 - Validate read set
9  for r in WS ∪ RS do
10     item_commit_rts ← commit_rts
11     if r.isolation = SR then
12         item_commit_rts ← commit_ts
13     if r.rts < item_commit_rts then
           # Begin atomic section
14         if r.wts ≠ r.item.wts or (r.item.rts ≤ item_commit_rts and
             isLocked(r.item) and r.item not in WS ) then
15             abort()
16         r.item.rts ← max(item_commit_rts, r.item.rts)
           # End atomic section
```

data stores its value, *e.g.,* $L(v)$ and $D(v)$ of a vertex v. The new field, *iso-level* $\in \mathbb{I}$, stands for the isolation level required for read or write the item, which is determined in the reading phase below.

Read phase. In the read phase, DD-OCC fetches vertices and edges from the global data graph to the current transaction's local working space, just as TicToc and all OCC protocols do for relations. However, according to the DDI isolation specification $\Gamma^{\mathbb{I}}$, the same item can be loaded multiple times with different isolation levels. If a read operation loads a data item that already exists in the local working space, then its *iso-level* field will be updated to the higher one between the new read operation and its original *iso-level* field.

Validation phase. During validation, DD-OCC decides whether a transaction can commit its changes in the local working space to the global data graph while assuring that the schedule remains conformant to $\Gamma^{\mathbb{I}}$. It does so in three steps, as shown in Algorithm 1.

Step (1): Lock write set (lines 1-2). Following TicToc, write items in the global graph are protected by write locks throughout the validation phase to ensure that concurrent updates to the global database do not conflict. All transactions will lock these data items in the same order, to avoid dead-lock formed between them. Locks will be released after the commit or abort of the associated transactions.

Step (2): Compute commit timestamp (lines 3-8). Similar to TicToc, DD-OCC also avoids global transaction timestamp allocation, by local logical timestamps in the local working spaces. But different from TicToc, DD-OCC needs two timestamps for each transaction:

- *commit_wts* (line 6): the minimal timestamp that is later than the latest *rts* of all the *write* items in the global graph.
- *commit_rts* (line 7): the minimal timestamp later than or equal to the latest *wts* of all *read* items in the local working space.

Step 3: Validation (lines 9-16). After executing the transaction in its local working space, DD-OCC decides whether the local changes to WS can be installed in the global graph by validating each data item $r = \langle item, data, wts, rts, iso\text{-}level \rangle \in RS \cup WS$ according to its *iso-level*.

- *iso-level* = SR: DD-OCC verifies that if r has not been modified until *commit_ts* (lines 12). Hence, r passes the check if $r.rts \geq commit\_ts$. Otherwise, it checks whether its global *wts* ($r.item.wts$) matches the local one ($r.wts$), *i.e.,* whether $r.wts = r.item.wts$ (line 14). If not, this indicates an unseen updated version of r has been installed in the global graph after r was loaded into RS, requiring the transaction to be aborted and preventing the write in WS from being installed. Meanwhile, if $r.item$ is locked, the transaction will also be aborted as r is being modified by another transaction (line 14). If r passes the check, the global *rts* of r ($r.item.rts$) is also updated to reflect the read operation (line 16).
- *iso-level* = SI: the validation process is the same as SR, except that *commit_ts* is relaxed to *commit_rts* (line 10). This is because it only needs to guarantee that all the reads are from the same version, which allows more data items to pass the validation.
- *iso-level* = RC: r always passes the check.

6 PUTTING DDI ISOLATION INTO ACTION

We next develop system GDDI that implements DDI isolation.

6.1 GDDI: Implementing DDI on Sortledton

We implement DD-OCC over Sortledton [21], a transactional graph storage that targets frequent edge insertions and deletions.

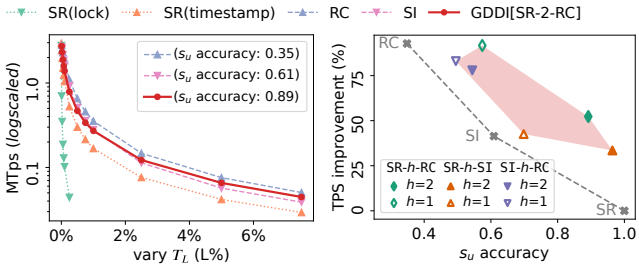
Sortledton overview. Sorledton uses an adjacency list based data structure that stores the adjacent edges of each vertex in a dedicated container. It adopts a lock-based method to provide hard-coded serializability (SR) by first locking containers that contain the vertices to be modified or whose neighboring edges are to be modified, then conducting the modification, and finally releasing locks upon commit. It does not support long-running read-write transactions.

Implementing DD-OCC on Sortledton. We next describe our extension to Sortledton to implement DD-OCC and support general graph transactions, including long-running read-write transactions.

On version storage. Since Sortledton uses lock-based protocol, to support DD-OCC we extend its version storage store *rts* and *wts* of each data item of the global graph as properties of the vertices and edges, where *rts* can be modified without introducing a new version.

To reduce the need for edge exploration in the validation phase, for each vertex, two additional timestamps, ne-rts and ne-wts, are stored to record the timestamps of the latest read and write to its neighbouring edges, respectively. Specifically, for a vertex u and an arbitrary timestamp ts_i, if ne-rts (resp. ne-wts) is ahead of ts_i, then it is guaranteed that rts (resp. wts) of all edges neighboring u are ahead of ts_i without the need to read them one by one.

On concurrency control. The transaction execution process of Sortledton is re-designed to support DD-OCC for general graph transactions. To fit with the three-phases OCC execution, a local working space is introduced for each transaction to store all the read data and temporarily hold the updates to be committed. To support



(a) Throughput: vary T_L (L%) (b) Isolation trade-offs in GDDI

Figure 4: Case study of T_s and `insert` over `citPatents`

`DD-OCC`, data items are loaded differently according to the isolation levels designated in the input Ddi isolation specification:

- *iso-level* = Sr or Si: chase through the version chain to get the corresponding version decided by the timestamps;
- *iso-level* = Rc: load the latest version at the head of the chain.

6.2 Breaking Down Large Operations

While Ddi offers per-operation isolation allocation, for long-running transactions that contain large traversals, it does not help much when the isolation allocated to the traversal operation is, *e.g.,* Sr, even if we have Rc for every other operation. In addition, for aggregate-over-traversal transactions, *e.g.,* T_L in Example 3, traversal operation would likely be allocated with the lowest isolation Rc in step (3) of Ddi isolation allocation scheme as shown in *e.g.,* Example 7. However, defaulting traversals to Rc would potentially give application (*e.g.,* aggregation atop traversal) an inaccurate view of the graph as the traversed subgraph is not protected by sufficient isolation, even though this does not violate any graph consistency rules.

Isolation partition. To deal with this, we again turn to the idea of fine-grained isolation that Ddi advocates, by further breaking down a traversal operation into multiple "partial" traversal operations. We then assign each partial traversal a dedicated isolation according to the view quality requirement by the applications.

Given a traversal operation $\mathsf{TR}_k(v)$, GDDI supports isolation partition scheme of the form ℓ-h-ℓ', where ℓ and ℓ' are isolation levels with ℓ stronger than ℓ', and $h \leq k$. GDDI protects read operations that traverses vertices and edges that are within h-hope from v by isolation ℓ and use ℓ' for vertices and edges further away from v.

The rationale is simple. The fine-grained isolation of Ddi can naturally take advantage of the "distance decaying effect" of graph aggregations, *e.g.,* GNN, personalized PageRank and random walk, which is often expressed as "all things are related, but near things are more related than far things" [19]. As a result, transaction conflicts further away from the origin v of traversal $\mathsf{TR}_k(v)$ has less impact than those closer to v. As such, isolation partition enables GDDI to further exploit fine-grained isolation for more flexible trade-off between transaction performance and traversal accuracy.

Case study: T_L and `insert`(u, v). Using the LDBC Graphalytics Benchmark, we implemented T_L and `insert`(u, v) of Example 1 over `citPatents` (with $k = 3$ in T_L), and evaluated the concurrent execution of T_L and $\in$ (u, v) with varying `L`% percentage of T_L transactions. The results are shown in Fig. 4. Specifically, SR, SI and RC are results of GDDI that allocates the traversal of T_L with Sr, Si and Rc, respectively; GDDI[SR-2-RC] uses an isolation partition; SR(lock) denotes Sortledton (original lock-based implementation), and SR(timestamp) is a variant of Sortledton that uses `TicToc` [46].

GDDI with isolation partition Sr-2-Rc achieves 22.7 times and 61.6% higher throughput than SR(lock) and SR(timestamp), respectively, with aggregate-over-traversal (PPR) accuracy as high as 89% of that running all transactions uniformly with serializability (Sr).

Figure 4(b) further demonstrates the flexible trade-off enabled by the isolation partition of Ddi, where it achieves throughput comparable to Si by partitioning Sr, while having (a) higher view quality of the traversals measured by the relative accuracy of aggregation s_u and (b) strictly enforced graph consistency as Sr does.

7 EXPERIMENTAL STUDY

Using benchmarks, we evaluated the benefits of Ddi isolation. Below we first specify the evaluation setup (Section 7.1). We then report the performance of system GDDI with varying workloads (Section 7.2) and examine the effectiveness of Ddi isolation (Section 7.3).

7.1 Experimental setting

Graphs. We used three data graphs provided in the LDBC Graphalytics Benchmark [14]: (a) `graph500`, a synthetic scale-free graph [26], (b) `LiveJournal` with 4.8M vertices and 34M edges, and (c) `citPatents` with 3.8M vertices and 16M edges. By default, we used `graph500` of configuration graph-500-22, with 2.3M vertices and 64M edges. To study the impact of different types of graphs, we also used configurations graph-500-21 with 1.2M vertices and 32M edges and graph-500-23 with 4.6M vertices and 129M edges. In addition to LDBC benchmark graphs, we additionally tested with `DBpedia` [7] as a representative knowledge graph.

Transactions. We consider both short and long transactions.

Topological transactions (`T-txn`) are short transactions that change the topology of the graphs by inserting or deleting an edge. These are exactly the type of transactions supported by prior systems and are the focus of previous evaluations [21, 30].

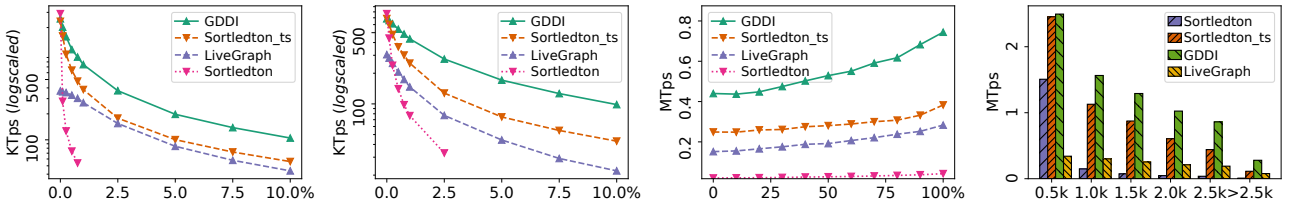
Update transactions (`U-txn`) are short transactions that read edges and update their properties, without change graph structures. Since `T-txn`'s are write-heavy, we consider `U-txn` transactions that are read-heavy: each reads 8 edges and update 2, following [18, 45].

Long-running transactions (`L-txn`). We consider `L-txn` transactions that are of pattern similar to T_L in Example 1: first traverse a subgraph G_u from a vertex u up to k-hops via $\mathsf{TR}_k(u)$, aggregates over G_u via aggregate function f_{agg}, and writes the aggregated score as u's data value. We employed two f_{agg} functions: (i) closeness centrality (CC), a measurement of the centrality of u in G_s [12]; and (ii) personalized PageRank (PPR), as a special case of random walk [16, 36] that computes PPR score of u in G_s. By default we set k to 2 for traversals $\mathsf{TR}_k(u)$ and used PPR as f_{agg}. We randomly generated vertex labels for both data graphs and traversal operations in `L-txn`s so that they could complete in reasonable time.

Workloads. We tested varying workloads with these transactions.

Short-transaction-only workloads. These are workloads targeted and used by prior system evaluations, and consist of short `T-txn` transactions only. They can be classified into 4 types, as follows:

- ins: construct graph via concurrent edge insertions only.



(a) vary L% of *write-int. mix* (T+L) (b) vary L% of *read-int. mix* (U+L) (c) vary T% of *complete mix* (T+U+L) (d) vary read-set size (*write-int. mix*)

Figure 5: Throughput of all systems under varying mixed workloads on `LiveJournal`

- del: remove all edges via concurrent edge deletions only.
- low-contention: a mix of random edge insertions and deletions.
- high-contention: the same as low-contention workload but with 30% of the transactions involving *hotspot* edges. By default, we picked 4 vertices with largest degrees and selected one connecting edge for each vertex as hotspots in each graph.

Mixed workloads. These are workloads that contain long-running transactions, which are not supported in prior studies. We consider:

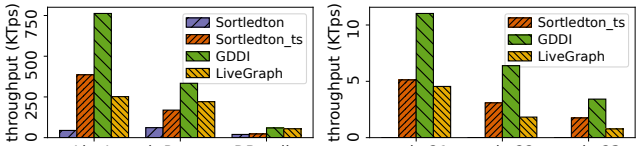
- *Read-intensive mix* (U+L): a mix of `U-txn` and `L-txn` transactions.
- *Write-intensive mix* (T+L): a mix of `T-txn` and `L-txn` transactions.
- *complete mix* (T+U+L): a mix of `U-txn`, `T-txn` and `L-txn`.

By default, transactions are randomly generated, *i.e.,* with vertices randomly sampled from the data graph. Similar to short-transaction-only workloads, we also varied their contention level by concentrating transactions to hotspots. Each type of mix workload can have different variants by varying the percentages U%, T% and L% of `U-txn`, `T-txn` and `L-txn` transactions. Unless stated otherwise, we adopt the *write-intensive mix* and set L% to 1%.

Systems. We extended the GFE (Graph Framework Evaluation) driver [2] to evaluate different graph transaction workloads. Specifically, we implemented and evaluated the following systems:

- Sortledton [21]: we used its original implementation, which employs a lock-based concurrency control and supports only edge insertions and deletions. We further optimized it in two aspects: (a) We incorporated NO_WAIT [45], which gives it the same performance improvement for high contention read-intensive workload as reported in [45]. (b) We extended it to support long-running `L-txn` transactions and all types of workloads.
- Sortledton_ts: a variant of Sortledton where we replaced the original lock-based protocol with `TicToc` [46] ported to graphs; `TicToc` is an optimistic timestamp ordering based protocol.
- Teseo [30]: we used its implementation from [1], which supports edge insertions and deletions only. (We tried to extend it for more transactions but observed the same problem reported also in [21]. Hence we only used it for short-transaction-only workloads.)
- LiveGraph [48]: similar to Sortledton and Teseo, it supports edge insertions and deletions only, with hard-coded Snapshot Isolation (SI) that equals Serializability (SR) for such workloads.
- GDDI: the GDDI system described in Section 6.

DDI *isolation specification*. In favor of competitors, which only support system-wide mono-isolation specification, by default for GDDI we set the isolation of all operations in short (`T-txn` and `U-txn`) transactions to Serializability (SR); for traversal operations in long-running transactions (`L-txn`), we used isolation partition of SR-*h*-RC, with $h = 1$ by default. Note that the conventional system-wide Serializability (SR) configuration is a special case of DDI isolation where all operations are allocated with SR; we denote such configuration of GDDI by GDDI[SR]. By Theorem 1 and 2, both GDDI and GDDI[SR] strictly prevent dangling edges and duplicated edges, the consistency model used by previous systems [21, 30, 48].



(a) Real-life graphs (b) different scales of `graph500`

Figure 6: Throughput for write-intensive mix across graphs

Configuration. We uniformly generated transactions of each workload across 24 working threads. Aborted transactions were resubmitted until they successfully committed or reached the maximum redo count, which is set to 3. We run our experiments on a dual-socket machine with 500 GB of memory. Each socket has two AMD EPYC 7302 16-Core CPUs, each with 256 MB of L3 cache and 16 hardware threads. We compiled the implementation with `GCC-10`.

7.2 Performance of System GDDI

Overall performance. We start with a complete evaluation of system GDDI with varying transaction workloads over all data graphs.

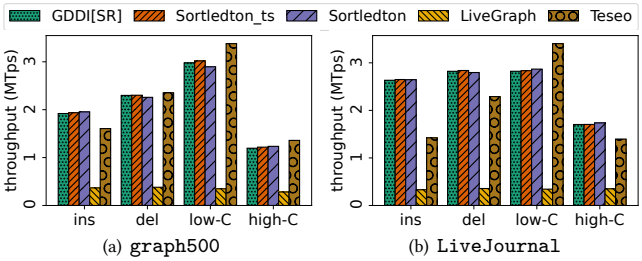
Varying mixed workloads. We first compared the throughput of all systems for varying mixes of different types of transactions.

(1) Read-intensive vs. write-intensive. Varying the percentage of `L-txn` transactions (L%), the results of GDDI, Sortledton, Sortledton_ts, and LiveGraph over `LiveJournal` for write-intensive and read-intensive mix are shown in Figures 5(a) and 5(b), respectively.

(a) GDDI consistently outperforms all other systems in the presence of `L-txn` transactions in the workload, and the gaps widen as L% of `L-txn` increases. For instance, GDDI improves the throughput of Sortledton_ts by 87.4% and 128.5% in write-intensive and read-intensive mix workloads, respectively, when L% = 10%. This improvement mainly comes from the reduction in compute and redo time for `L-txn` by GDDI (see more in breakdown analysis shortly).

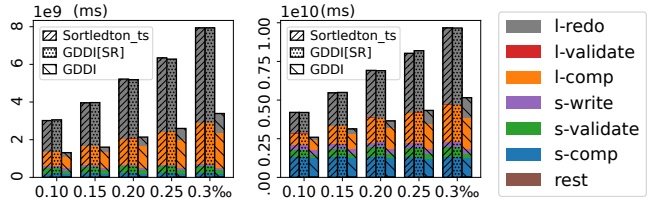
(b) Sortledton_ts consistently outperforms Sortledton. The gap increases when more `L-txn` transactions are present.

(2) Impact of structural updates (`T-txn`). We also examined the impact of structural updates, *i.e.,* topological transactions (`T-txn`).



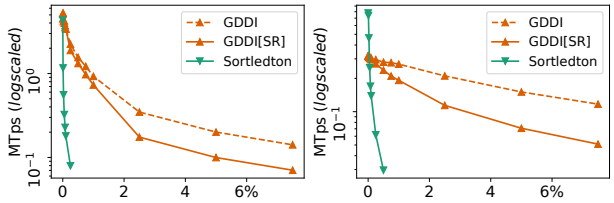
(a) `graph500` (b) `LiveJournal`

Figure 7: GDDI **overhead - throughput with short transactions**



(a) vary L‰ of *write-int. mix* (b) vary L‰ of *read-int. mix*

Figure 8: Runtime breakdown of 1.2×10^9 **txns** (`l-redo`: redo cost of `L-txns`; `s-validate`: validation of `S-txns`; similarly for others)



(a) vary L% of *write-int. mix* (T+L) (b) vary L% of *read-int. mix* (U+L)

Figure 9: Lock vs. **OCC under high contention (`LiveJournal`)**

To do this, we used the complete mix workload that contains all types of transactions, and evaluated the throughput of all systems with varying percentage of `T-txn` transactions (T%).

The results over `LiveJournal` are reported in Fig. 5(c). We find that both GDDI and Sortledton_ts benefit significantly when more structural updates are present, as their throughput increases with growing T%. As a result, systems (*e.g.,* GDDI and Sortledton_ts) that deal with `L-txn` better would benefit more from more `T-txns`.

(3) Impact of read-set size. We further studied the impact of read-set size of `L-txn` transactions. We generated write-intensive mix workloads with `L-txn` transactions of read-set size varying from less than 0.5k, 0.5-1k, 1-1.5k, 1.5-2k, 2-2.5k and greater than 2.5k vertices.

The results over `LiveJournal` are shown in Fig. 5(d). GDDI consistently has the highest throughput among all systems in all cases. Moreover, its gap over Sortledton_ts, the 2nd best system, increases when the read-set size of `L-txns` increases. The same applies to the improvement of Sortledton_ts over Sortledton. This is because `L-txns` with larger-read set are more costly to abort and thus have a greater impact on the overall system throughput.

Varying graphs. Using default transaction mix configurations, we calculated the average throughput of all systems over all graphs, including three real-life graphs and the three variants of `graph500`. The results are given in Fig. 6, and tell us the following.

(a) GDDI outperforms Sortledton_ts by 97.5%, 97.3% and 161.7% on LiveGraph, `citPatents` and `DBpedia`, respectively.

(b) GDDI also consistently outperforms all other systems on synthetic graphs `graph500` with varying scale factors.

Overhead of Ddi **isolation**. We also examined the overhead of the per-operation Ddi isolation. To do this, we compared the performance of GDDI with baseline systems for "traditional" workloads targeted by these baselines, *i.e.,* short-transaction-only workloads with short `T-txn` transactions only. We also analyzed the runtime breakdown of GDDI over write-intensive mix workloads.

Performance for short-transaction-only workloads. Figure 7 shows the throughput of all systems for existing short-transaction-only workloads, where GDDI uses a Ddi isolation that assigns Sr to all operations of all transactions. As such, the difference between GDDI (GDDI[Sr]) and Sortledton_ts exactly reflects the overhead of Ddi.

The results tell us that, for all 4 types of short-transaction-only workloads, *i.e.,* edge insertions, deletions, low-contention mix and high-contention mix of deletions and insertions, GDDI[Sr] performs as good as Sortledton_ts. This implies that the per-operation isolation configuration in GDDI, which builds atop Sortledton_ts, does not bring any perceptible overhead for concurrency control. It also confirms that the exceedingly well capability of GDDI in dealing with `L-txn` transactions as we have seen above does not sacrifice its performance for existing workloads with structural updates only.

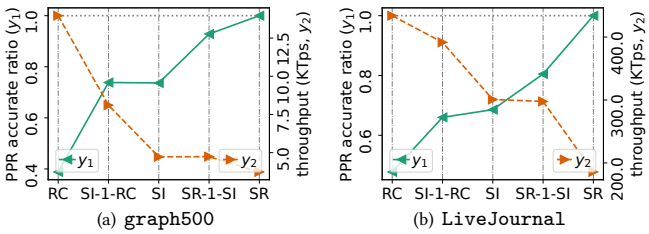
The unnoticeable overhead is due to the design of `DD-OCC` (Section 5). Compared to Sortledton_ts, which we derived from Sortledton by incorporating `TicToc` [46], the overhead of `DD-OCC` includes (a) extra storage of *iso-level* for each data item, which is implemented as `uint8_t` in C++ and accounts for only ~2% additional space, and (b) the per-operation validation to compute timestamps for read items (lines 10-12 of Algorithm 1). The latter has constant time complexity that runs at most |`RS`| times and is asymptotically subsumed by the validation cost of `TicToc` in the absence of Ddi.

We also find that GDDI[Sr], Sortledton_ts and Sortledton all perform best across workloads and graphs, except for low-contention mixes of edge deletions and insertions for which Teseo is better.

Execution time breakdown. To further consolidate our examination of the concurrency control overhead with Ddi isolation, we carried out a breakdown analysis of the total transaction execution time of mixed workloads with 1.2×10^9 transactions. Varying the percentage L% of `L-txn` transactions, the breakdown results of write-intensive mix and read-intensive mix are presented in Figures 8(a) and 8(b), respectively (bar patterns denote systems and color-codes on the right represent different stages of transaction execution).

(a) Compared to Sortledton_ts, GDDI has no visible overhead in all stages. The validation for `L-txn` consumes only ~0.1% of total time.

(b) Compared to Sortledton_ts, 85.1% and 6.37% of the reduction in execution time achieved by GDDI comes from redoing and computing `L-txns`, respectively, when L% = 0.01%. Surprisingly, there is also a 8.85% reduction in validation of `T-txns`, as the validation of Rc vertices/edges in `L-txns` does not require locks, making it easier for `T-txns` to acquire exclusive write locks during validation to install their updates. Moreover, with more `L-txns` (*i.e.,* larger L%), the savings from redoing and validating `L-txns` also increase.



(a) graph500 (b) LiveJournal

Figure 10: Impact of isolation partition for *write-int. mix*

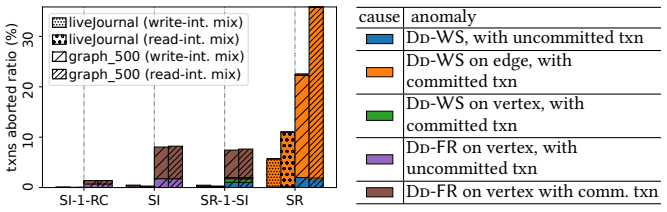


Figure 11: L-txn abort causes (anomalies): vary GDDI isolation

7.3 Effectiveness of Ddi Isolation

We next delve into the role of Ddi isolation in system GDDI.

Lock vs OCC for Ddi. We first justify the optimistic timestamp ordering based implementation of Ddi isolation in system GDDI, by comparing GDDI and GDDI[Sr] with the lock-based Sortledton over *high-contention* write-intensive and read-intensive mixed workloads. The results over LiveJournal are reported in Fig. 9.

It is common wisdom that lock-based protocols work better than OCC-based for high-contention workload, which is also confirmed by Fig. 7 where Sortledton (lock) outperforms Sortledton_ts (OCC) for high-contention short transaction. Interestingly, however, this no longer holds in presence of even quite few long-running transactions (L-txn). As shown in Fig. 9, Sortledton only performs better than GDDI[Sr] and GDDI in high-contention, read-intensive workloads where L% is below 0.025%. For high-contention write-intensive workloads, GDDI[Sr] outperforms lock-based Sortledton by 351% even with L% as low as 0.01%. Furthermore, as L% increases, GDDI[Sr] improves more over Sortledton in both workloads, and the gap between GDDI and GDDI[Sr] also widens.

The reason for this is that both types of protocols lock data items to be written similarly. Therefore, there is not much difference when processing write-intensive transactions, *e.g.,* edge insertions/deletions where all data items are to be written. This is also consistent with the observation in [45] regarding lock and OCC.

This indicates that even a small number of L-txn transactions necessitate a fundamental rethink of concurrency control for graph transactions. It also justifies the optimistic timestamp ordering design of DD-OCC to realize Ddi isolation in workloads with L-txn.

Impact of Ddi isolation. Varying the Ddi isolation partition of traversals in L-txns, we evaluated its impact on (a) the throughput of GDDI and (b) the accuracy of aggregation f_{agg} over traversals.

Here accuracy ratio of f_{agg} with a Ddi isolation partition is calculated as the percentage of transactions of which f_{agg} value does not change up to a threshold of 1% when we partition the isolation of their traversals. The results for PPR over write-intensive mix with varying Ddi isolation are shown in Figures 10(a) and 10(b).

We find that Ddi isolation partition has a small impact on the accuracy of traversal-aggregation in L-txns, while significantly boost the transaction throughput of GDDI. For instance, on graph500, when partition Si traversal into Si-1-Rc, over f_{agg} values of 99% of transactions remain the same, while this improves the throughput of GDDI by 71.5%, from 3.78 KTps to 6.48 KTps. In contrast, when we simply lower Si to Rc entirely, more than half of transactions have their f_{agg} significantly affected and varied over the threshold. This case shows that Si-1-Rc has the best of both on graph500: it has the throughput of Rc but retain the accuracy of f_{agg} protected by Si. Better still, such benefits do not compromise graph consistency modeled by the Ddi consistency rules according to Theorem 1.

Anomalies breakdown. Finally, we examined the frequency of different anomalies (recall Section 4.2) caused by L-txns that are prevented by GDDI with varying Ddi isolation, for both read-intensive and write-intensive mix workloads over all data graphs. The results over LiveJournal and graph500 are plotted in Fig. 11.

Over graph500 with the write-intensive mix (L+T) workload, (1) under Si all L-txn aborts are to prevent Dd-FR anomalies on vertices, as the version storage for edges naturally provides the Si guarantee for operations on edges. Among them, 48.8% the aborts are caused by conflicts with updates that have already been committed. (2) When the isolation level reaches Sr-1-Si, the causes for L-txn aborts become more diverse, 23.6% of them for preventing Dd-WS anomalies, while the rest are for Dd-FR anomalies. Preventing Dd-FR is the primary cause because Sr is only required within 1-hop (including the end vertices) from the source vertex of traversals, and Dd-WS can only arise from unseen updates within this region. (3) When the Ddi isolation for traversals further rises to Sr, the abort ratio also increases as expected. The primary cause for the aborts are to prevent Dd-WS caused by conflicts with committed updates on edges due to T-txns. Results on LiveJournal are similar.

The difference between the write-intensive mix and read-intensive mix is not obvious when the isolation level is below Sr. However, they differs significantly when traversals are allocated with Sr entirely in L-txns, *e.g.,* 22.5% for the write-intensive mix and 35.9% for read-intensive mix. We also observe that that the additional aborts of L-txns from read-intensive mix are most to prevent Dd-WS on edges when compared to traversals with lower level isolation.

8 DISCUSSION

Limitation. As far as our knowledge confirms, this is the first attempt that studies operation-level isolation allocation and explicitly connects isolation allocation with consistency guarantees. Due to its infancy, there are naturally limitations. (1) We target long-running transactions with large read set but small write set. As also observed in [17], there exist long-running transactions with large writes. (2) Another limitation is the assumption of distance decaying aggregation over traversals for isolation partition to take effect.

Vision. The overarching goal is to develop a notion of isolation for heterogeneous datasets where a uniform isolation is an impractical overkill. Towards this vision, we are currently exploring Ddi for fine-grained isolation for relations and other data models.

ACKNOWLEDGMENTS

This work is supported by RAEng RF\201920\19\319.

REFERENCES

[1] 2021. *https://github.com/cwida/teseo.*

[2] 2021. Graph Framework Evaluation Driver. https://github.com/PerFuchs/gfe_driver.

[3] 2023. *https://bugs.mysql.com/bug.php?id=112446.*

[4] 2024. Managing Concurrency with Isolation Levels. https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/sql/snapshot-isolation-in-sql-server.

[5] 2024. Memgraph Advanced Graph Extensions (MAGE) library. https://memgraph.com/docs/advanced-algorithms.

[6] 2024. Neo4j Graph Data Science. https://neo4j.com/product/graph-data-science/.

[7] 2025. *http://wiki.dbpedia.org.*

[8] 2025. Full version. https://homepages.inf.ed.ac.uk/ycao/DDIfull.pdf.

[9] Atul Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[10] A. Adya, B. Liskov, and P. O'Neil. 2000. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073).* 67–78. https://doi.org/10.1109/ICDE.2000.839388

[11] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, et al. 2023. PG-Schema: Schemas for property graphs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[12] Alex Bavelas. 1950. Communication Patterns in Task-Oriented Groups. *The Journal of the Acoustical Society of America* 22, 6 (11 1950), 725–730. https://doi.org/10.1121/1.1906679 arXiv:https://pubs.aip.org/asa/jasa/article-pdf/22/6/725/18729421/725_1_online.pdf

[13] Michael J Cahill, Uwe Röhm, and Alan D Fekete. 2009. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems (TODS)* 34, 4 (2009), 1–42.

[14] Mihai Capotă, Tim Hegeman, Alexandru Iosup, Arnau Prat-Pérez, Orri Erling, and Peter Boncz. 2015. Graphalytics: A Big Data Benchmark for Graph-Processing Platforms. In *Proceedings of the GRADES'15* (Melbourne, VIC, Australia) *(GRADES'15).* Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2764947.2764954

[15] Hongzhi Chen, Changji Li, Chenguang Zheng, Chenghuan Huang, Juncheng Fang, James Cheng, and Jie Zhang. 2022. G-Tran: A High Performance Distributed Graph Database with a Decentralized Architecture. *Proc. VLDB Endow.* 15, 11 (2022), 2545–2558.

[16] Yen-Yu Chen, Qingqing Gan, and Torsten Suel. 2004. Local methods for estimating pagerank values. In *Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management* (Washington, D.C., USA) *(CIKM '04).* Association for Computing Machinery, New York, NY, USA, 381–389. https://doi.org/10.1145/1031171.1031248

[17] Audrey Cheng, Jack Waudby, Hugo Firth, Natacha Crooks, and Ion Stoica. 2024. Mammoths are Slow: The Overlooked Transactions of Graph Data. *Proc. VLDB Endow.* 17, 4 (mar 2024), 904–911. https://doi.org/10.14778/3636218.3636241

[18] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10).* Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[19] Evgeny Dantsin and Alexander Wolpert. 2020. Similarity Between Points in Metric Measure Spaces. In *SISAP*, Vol. 12440. 177–184.

[20] David Ediger, Robert McColl, E. Jason Riedy, and David A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *HPEC.* 1–5.

[21] Per Fuchs, Domagoj Margan, and Jana Giceva. 2022. Sortledton: A Universal, Transactional Graph Data Structure. *Proc. VLDB Endow.* 15, 6 (feb 2022), 1173–1186. https://doi.org/10.14778/3514061.3514065

[22] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. *SIGMOD Rec.* 16, 3 (Dec. 1987), 249–259. https://doi.org/10.1145/38714.38742

[23] Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Martin Schuster, Petra Selmer, and Hannes Voigt. 2019. Updating graph databases with Cypher. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2242–2254. https://doi.org/10.14778/3352063.3352139

[24] Stephen J. Hegner. 2014. Guard Independence and Constraint-Preserving Snapshot Isolation. In *FoIKS*, Vol. 8367. 230–249.

[25] Stephen J Hegner. 2019. Transaction Isolation in Mixed-Level and Mixed-Scope Settings. In *Advances in Databases and Information Systems: 23rd European Conference, ADBIS 2019, Bled, Slovenia, September 8–11, 2019, Proceedings 23.* Springer, 390–406.

[26] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. 2016. LDBC graphalytics: a benchmark for large-scale graph analysis on parallel and distributed platforms. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1317–1328. https://doi.org/10.14778/3007263.3007270

[27] Nan Jiang, Fuxian Duan, Honglong Chen, Wei Huang, and Ximeng Liu. 2021. MAFI: GNN-based multiple aggregators and feature interactions network for fraud detection over heterogeneous graph. *IEEE Transactions on Big Data* 8, 4 (2021).

[28] Pradeep Kumar and H. Howie Huang. 2020. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. *ACM Trans. Storage* 15, 4 (2020), 29:1–29:40.

[29] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *Proc. VLDB Endow.* 5, 4 (dec 2011), 298–309. https://doi.org/10.14778/2095686.2095689

[30] Dean De Leo and Peter A. Boncz. 2021. Teseo and the Analysis of Structural Dynamic Graphs. *Proc. VLDB Endow.* 14, 6 (2021), 1053–1066.

[31] Xianghua Li, Xiyuan Zhen, Xin Qi, Huichun Han, Long Zhang, and Zhen Han. 2023. Dynamic community detection based on graph convolutional networks and contrastive learning. *Chaos, Solitons & Fractals* 176 (2023), 114157.

[32] Sunil Kumar Maurya, Xin Liu, and Tsuyoshi Murata. 2021. Graph Neural Networks for Fast Node Ranking Approximation. *ACM Trans. Knowl. Discov. Data* 15, 5 (2021), 78:1–78:32.

[33] Elliott Mendelson. 2009. *Introduction to mathematical logic.* Chapman and Hall/CRC.

[34] Mark Needham and Amy E Hodler. 2019. *Graph algorithms: practical examples in Apache Spark and Neo4j.* O'Reilly Media.

[35] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[36] Sungchan Park, Wonseok Lee, Byeongseo Choe, and Sang-Goo Lee. 2019. A Survey on Personalized PageRank Computation Algorithms. *IEEE Access* 7 (2019), 163049–163062. https://doi.org/10.1109/ACCESS.2019.2952653

[37] Calton Pu, Gail E. Kaiser, and Norman C. Hutchinson. 1988. Split-Transactions for Open-Ended Activities. In *Proceedings of the 14th International Conference on Very Large Data Bases (VLDB '88).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 26–37.

[38] Raghu Ramakrishnan and Johannes Gehrke. 2002. *Database management systems.* McGraw-Hill, Inc.

[39] Kenneth Salem, Hector Garcia-Molina, and Rafael Alonso. 1987. Altruistic Locking: A Strategy for Coping with Long Lived Transactions. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems.* Springer-Verlag, Berlin, Heidelberg, 175–199.

[40] Joel Spencer. 2013. *The strange logic of random graphs.* Vol. 22. Springer Science & Business Media.

[41] Bimal Viswanath, Muhammad Ahmad Bashir, Mark Crovella, Saikat Guha, Krishna P. Gummadi, Balachander Krishnamurthy, and Alan Mislove. 2014. Towards Detecting Anomalous User Behavior in Online Social Networks. In *USENIX Security.* 223–238.

[42] Bimal Viswanath, M. Ahmad Bashir, Mark Crovella, Saikat Guha, Krishna P. Gummadi, Balachander Krishnamurthy, and Alan Mislove. 2014. Towards Detecting Anomalous User Behavior in Online Social Networks. In *23rd USENIX Security Symposium (USENIX Security 14).*

[43] Yanling Wang, Jing Zhang, Shasha Guo, Hongzhi Yin, Cuiping Li, and Hong Chen. 2021. Decoupling representation learning and classification for gnn-based anomaly detection. In *SIGIR.*

[44] Jack Waudby, Paul D. Ezhilchelvan, and Jim Webber. 2022. Pick & Mix Isolation Levels: Mixed Serialization Graph Testing. In *TPCTC.*

[45] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (mar 2017), 781–792. https://doi.org/10.14778/3067421.3067427

[46] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *SIGMOD.* 1629–1642.

[47] Hongyang Zhang, Peter Lofgren, and Ashish Goel. 2016. Approximate personalized pagerank on dynamic graphs. In *SIGKDD.* 1315–1324.

[48] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2020. LiveGraph: a transactional graph storage system with purely sequential adjacency list scans. *Proceedings of the VLDB Endowment* 13, 7 (March 2020), 1020–1034. https://doi.org/10.14778/3384345.3384351

[49] Xiaowei Zhu, Marco Serafini, Xiaosong Ma, Ashraf Aboulnaga, Wenguang Chen, and Guanyu Feng. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proc. VLDB Endow.* 13, 7 (2020), 1020–1034.