DoppelGanger++: Towards Fast Dependency Graph Generation for Database Replay

WONSEOK LEE*, POSTECH, Korea
JAEHYUN HA*, POSTECH, Korea
WOOK-SHIN HAN†, POSTECH, Korea
CHANGGYOO PARK, SAP Labs Korea, Korea
MYUNGGON PARK, SAP Labs Korea, Korea
JUHYENG HAN, SAP Labs Korea, Korea
JUCHANG LEE, SAP Labs Korea, Korea

A database replay system (DRS) captures workloads on a production system and then replays them in a test system to test various system changes, avoiding any risk before realizing them in production. The dependency graph generation in a DRS is crucial in preserving output determinism while maximizing concurrency. The state-of-the-art dependency graph generation algorithm deployed in a commercial DBMS uses a generate-and-prune strategy. It first generates a dependency graph by performing backward scans for each request in a workload. It then prunes all redundant edges using an expensive, transitive reduction algorithm. However, we notice that this generates a large dependency graph that contains many redundant edges and its worst-case time complexity is quadratic to the number of requests in a workload. In order to solve these challenging problems, we formally propose four classes of dependency graphs for DRSs. We then present a stateful single forward scan algorithm, SSFS, to generate any class of dependency graphs by performing a single scan over all requests while succinctly maintaining states. Here, states refer to information that is stored and maintained for efficient dependency graph generation. We also propose the parallel SSFS to utilize the computation power with multi-core CPUs while balancing the loads. We implemented our DRS in a leading commercial DBMS. Extensive experiments using the TPC-C, SD benchmarks, and a real-world customer workload show that our DRS significantly improves the dependency graph generation time by up to two orders of magnitude, compared to the state-of-the-art.

CCS Concepts: • **Information systems → Database utilities and tools**; *Database performance evaluation.*

Additional Key Words and Phrases: database replay

ACM Reference Format:
Wonseok Lee, Jaehyun Ha, Wook-Shin Han, Changgyoo Park, Myunggon Park, Juhyeng Han, and Juchang Lee. 2024. DoppelGanger++: Towards Fast Dependency Graph Generation for Database Replay. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 67 (February 2024), 26 pages. https://doi.org/10.1145/3639322

*Both authors contributed equally to this research.
†Corresponding author

Authors' addresses: Wonseok Lee, wslee@dblab.postech.ac.kr, POSTECH, Korea; Jaehyun Ha, jhha@dblab.postech.ac.kr, POSTECH, Korea; Wook-Shin Han, wshan@dblab.postech.ac.kr, POSTECH, Korea; Changgyoo Park, changgyoo.park@sap.com, SAP Labs Korea, Korea; Myunggon Park, myunggon.park@sap.com, SAP Labs Korea, Korea; Juhyeng Han, juhyeng.han@sap.com, SAP Labs Korea, Korea; Juchang Lee, juc.lee@sap.com, SAP Labs Korea, Korea.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 2836-6573/2024/2-ART67
https://doi.org/10.1145/3639322

1 INTRODUCTION

Database replay systems (DRSs) test relational database systems in a test system. DRSs capture workloads on a production system and then replay them in a test system to test various system changes (such as hardware/software upgrades) and to avoid any risk such as (a) performance regression, (b) bugs, or (c) new resource contention points before realizing them in production [20, 28]. Here, a workload consists of user requests each containing a SQL statement with session ID.

Output determinism [10] ensures that the replay of a captured workload produces the same output as the original run, even when physical plans for the workload change due to hardware or software updates. The relative ordering between two dependent requests in the original run must be preserved in the replay. Otherwise, the replay could not guarantee to produce the same output as the original run. For example, in Listing 1, Q1 is executed before Q2 in the original run. Assume that both queries are executed in auto-commit mode. However, during database replay, suppose that Q1 and Q2 are replayed in reverse order. Then, the output of Q2 differs from that in the original run, which violates output determinism.

Listing 1. Query examples

```
Q1: UPDATE emp SET salary=salary*1.1;
Q2: SELECT * FROM emp WHERE salary > 60000;
```

The state-of-the-art DRSs ensure output determinism by generating a dependency graph from a captured workload and replaying requests based on the dependency graph. Here, each vertex in the dependency graph corresponds to a request and an edge imposes a precedence constraint between two requests [3, 28]. Although one can ensure output determinism by sequentially replaying the requests in the same order in capture time, this naive approach fails to run a realistic replay, severely limiting concurrency [28]. Instead, by executing the requests in parallel while preserving the ordering in the dependency graph, DRSs can achieve output determinism while supporting realistic, concurrent replay.

DRSs have been supported by major database vendors [2, 3, 5, 38]. They support the following four phases in their database replay workflow: workload capture, dependency graph generation, workload replay, and report generation. In the first phase, a DRS records all requests in a workload, one capture file for each session. This step is done in a production system, while the other steps are typically done in a test system to avoid interfering with running applications in the production system. In the dependency generation phase, it generates a dependency graph that imposes precedence constraints among requests to ensure the output determinism. In the workload replay phase, it replays captured workloads using the minimal dependency graph with those capture files. The last step generates various reports about any divergence between system changes.

Note that the decision to migrate existing database instances to a target system cannot be made in one shot; instead, capture-and-replay is executed continuously to test the target system using various workloads. Thus, reducing the generation time is crucial, although dependency graph generation is an offline process.

To achieve output determinism while maximizing the concurrency, the DRS must generate the minimal dependency graph with the shortest critical path [39] and replay the workload based on it. Morfonios et al. [28] proposed RBSS, an efficient dependency graph generation algorithm using the generate-and-prune strategy. With this strategy, the generation step generates the incoming edges of each vertex by finding the latest dependent requests in other sessions using a backward scan, while the pruning step prunes redundant edges using a transitive reduction algorithm [9]. A

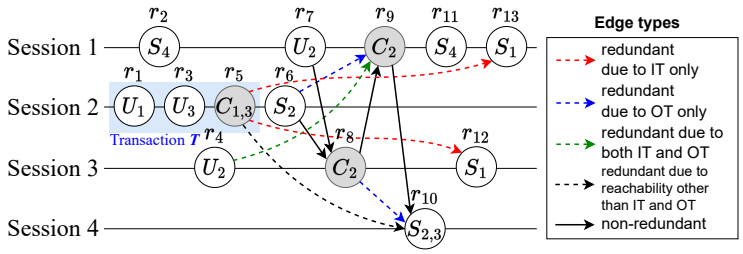


Fig. 1. An example of a dependency graph with edge types.

direct edge (u, v) is *redundant* if v is reachable from u after removing the edge. The technique has been employed in a commercial DBMS [2].

However, RBSS could still generate a large dependency graph containing many redundant edges, although they can be filtered in the generation step. For example, Table 1 shows that 99.6% of the edges in this dependency graph (labeled as $\mathcal{G}_{\text{RBSS}}$) are redundant in a representative ERP workload, SD-Benchmark [4]. Given a dependency graph $G(V, E)$, computing its transitive reduction $G^{tr}(V, E')$ could be expensive when $|E| >> |E'|$, since all redundant edges in E will be removed.

Secondly, the time complexity of RBSS could be $O(|V|^2)$ due to repetitive backward scans for every vertex. Specifically, to generate in-edges of each vertex in a session, RBSS needs to scan other sessions backward until reaching the start of the sessions in the worst case. Furthermore, backward scanning accesses many unnecessary sessions. This inefficiency in the dependency graph generation algorithm significantly increases the dependency graph generation time. Thus, we must develop an efficient algorithm to generate a dependency graph close to the transitive reduction's size.

RBSS incurs a major bottleneck in the capture-and-replay pipeline, constituting more than 50% of the total end-to-end time. Since our customers *continuously* capture and replay changing workloads over time to check any divergence between system changes quickly, the dependency graph generation needs to be significantly boosted.

To formally address this problem, we first define two types of *dominant*, redundant edges in the dependency graph (object transitivity (OT) and inter-session transitivity (IT)). OT refers to redundancy due to a path where all requests on the path access a common object. Consider a dependency graph in Figure 1. The label of a vertex represents an operation (Update, Select, or Commit) on an object. For example, the label (i.e., U_1) of a request r_1 represents an update statement on object 1. For example, (r_6, r_9) is redundant by OT since there is a path (r_6, r_8, r_9) where all requests access the common object 2. IT refers to redundancy due to a path through requests in two sessions, even if some of the requests access different object(s). Consider Sessions 2 and 3. Then, the edge (r_5, r_{12}) is redundant due to IT as a path (r_5, r_6, r_8, r_{12}) through these sessions exists. Note that r_5 and r_{12} access object 1, while the other requests access object 2. Although we can generalize the definition of IT for three or more sessions, pruning such redundancy will require expensive join operations, which is even more expensive than transitive closure. This motivates us to balance compactness and efficiency in dependency graph generation. We will elaborate on this issue in Section 3.2.

In order to find the most suitable dependency graph by systematically exploring the design space, i.e., taxonomy, of all combinations of IT and OT, we next propose four new dependency graphs: IT(k)-free, OT-free, OTIT-free, and IT[OT]-free graphs. We will formally define these graphs in Section 3. Our formalism can explain that 1) the IT(k)-free graph is a generalized one of [28], 2)

Table 1. Proportion of the redundant edges in each dependency graph in SD-benchmark with 16 server instances.

Dependency Graph	$\frac{\text{\# of redundant edges}}{\text{\# of edges}} \times 100\%$
$\mathcal{G}_{\text{RBSS}}$	99.6%
IT-free graph ($\mathcal{G}_{\text{IT}}$)	73.7%
OT-free graph ($\mathcal{G}_{\text{OT}}$)	62.9%
IT[OT]-free graph ($\mathcal{G}_{\text{IT[OT]}}$)	9.0%
OTIT-free graph ($\mathcal{G}_{\text{OTIT}}$)	5.3%

the OT-free graph is another new dependency graph to capture the transitive property of edges through common objects, and 3) the IT[OT]-free graph takes advantages from both. We discover that the IT[OT]-free graph is the most desirable one balancing both efficiency and size.

In order to avoid repetitive backward scans, we propose a novel, efficient dependency graph generation algorithm called *stateful single forward scan* (SSFS). SSFS performs a single forward scan over all requests while maintaining states. By analyzing the definitions of redundancies, we succinctly maintain the states depending on the dependency graph type. We also propose a parallel version of SSFS, where the requests are horizontally partitioned by time range. The parallel SSFS hierarchically merges these local dependency graphs to generate a global dependency graph, achieving linear speedup.

Our contributions are summarized as follows:

(1) We implement DoppelGanger++, a DRS with all proposed techniques in a leading, commercial DBMS. Thus, DoppelGanger++ can support real-world customer workloads.
(2) We formally define a taxonomy of the four types of dependency graphs by systematically exploring the design space of dominant redundant edges. We also provide a Venn diagram to illustrate the containment relationships among those dependency graphs.
(3) We propose a novel dependency generation algorithm SSFS to scan requests once, i.e., avoiding the quadratic complexity and achieving the linear complexity. Note that SSFS can generate any type of dependency graphs in the taxonomy.
(4) By analyzing all dependency graphs, we suggest SSFS with the IT[OT]-free graph, which achieves both the compactness of the dependency graph and the efficiency of the algorithm.
(5) We also propose a parallel version of SSFS, which reduces the dependency graph generation time further.
(6) Experiments using TPC-C, SD benchmarks, and a real-world customer workload show that SSFS outperforms RBSS by up to two orders of magnitude. We also show that the parallel SSFS achieves almost linear speedup.

The rest of this paper is organized as follows. Section 2 describes the overall architecture of DoppelGanger++. Section 3 proposes a taxonomy of dependency graphs. In Section 4, we review RBSS and its relationship to the dependency graphs defined in Section 3. Section 5 proposes our SSFS algorithm and shows how SSFS generates all types of dependency graphs. We also explain why the IT[OT]-free graph is a desirable dependency graph balancing both efficiency and size. Section 6 proposes a parallel version of SSFS. In order to substantiate our claims, Section 7 presents an extensive experimental evaluation of SSFS compared to RBSS using the two benchmarks and a real-world customer workload. Section 8 compares our contributions to related work, and Section 9 concludes the paper.

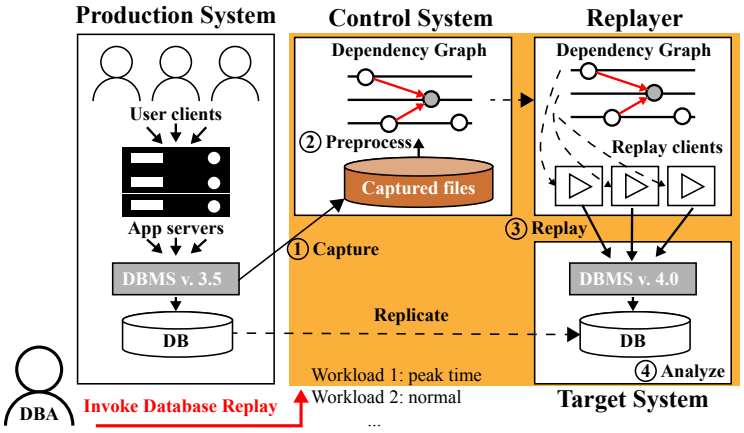


Fig. 2. The architecture of DoppelGanger++.

2 ARCHITECTURE OVERVIEW

DoppelGanger++ is a full-fledged DRS built on a state-of-the-art commercial DBMS. The system offers users a multi-purpose testing tool by allowing them to capture the workload running on a source system and replay the captured workload on a target system with detailed replay analysis reports. Users can easily manage the entire process through a visualizing tool. Figure 2 illustrates the overall architecture of DoppelGanger++, consisting of the following four steps.

The capturing step automatically captures the workload information including execution context information and requests running on the production system in a lightweight way. Captured information such as SQL data and transaction data is categorized and stored in different files according to its type. In this step, DoppelGanger++ backs up the current snapshot so that it can be used for replay.

The preprocessing step in a control system receives captured workload files as input and generates dependency graph files for consistent workload replay [28]. The input files contain information about requests issued from each database session at the time of capturing. The resulting files can be replayed multiple times. The dependency graph file generated is a serialization of requests where each request is associated with dependent requests in other concurrent sessions. Note that the dependency graph generation takes 50% of the total end-to-end time when we use the state-of-the-art [28], which is our motivation.

The replaying step replays the captured workload using dependency graph files while preserving transactional order on the target system which is initialized by the snapshot backed up. Specifically, loader threads read the dependency graph files and load them into session-specific request queues. Each queue is managed by a request dispatcher that controls the execution timing. The request dispatcher then sends the requests from the queue to execution threads. The requests without incoming edges can be executed. When a non-commit request starts, it obtains a snapshot and removes its outgoing edges. This enables successful completion of long-read transactions without blocking write transactions. A commit request, on the other hand, acquires a commit timestamp and removes the outgoing edges, allowing successful commitment. After running the requests, the execution threads return their results.

The analyzing step compares replayed results with captured results and generates visualized analysis reports. The comparison is performed in terms of performance and consistency. For performance comparison, system-level throughput, resource consumption, and the execution times

of individual requests are measured. The overall database state and the execution result of individual requests are measured for consistency check.

Although this is beyond our scope, DoppelGanger++ already supports database replay in a distributed environment. Our underlying DBMS maintains a single coordinator node with a global timestamp. Thus, the transactions executed in the distributed database nodes can be ordered by the same transaction timestamp generator, so it does not require changes to dependency generation. An edge between objects located in two database nodes is implemented as an inter-process communication call at the replay.

3 DEPENDENCY GRAPHS

We propose a formal taxonomy using the design space of all combinations of inter-session transitivity (IT) and object transitivity (OT): IT(k)-free, OT-free, OTIT-free, and IT[OT]-free graphs. Since the dependency graph generated from the workload is ultimately pruned by extensive transitive reduction, we need to find a compact dependency graph that can be efficiently generated by a sequential scan. Based on our formal taxonomy, we will show in Lemma 1 that the dependency graph generated by RBSS is a special case of IT(1)-free graph, since it fails to prune some edge (r, r') with more than one IT connectivity between r and r'.

3.1 Notation and Problem Definition

A workload $\mathcal{W}$ is modeled as a directed graph $\mathcal{G}_{\text{ini}} = (\mathcal{V}_{\mathcal{R}}, \mathcal{E}_{\text{ses}})$ where each vertex corresponds to a request in $\mathcal{W}$, and each edge is a pair of consecutive requests in a session (e.g., (r_2, r_7) in Figure 1). $\mathcal{G}_{\text{ini}}$ is called an *initial graph*. Here, each request r is associated with a unique timestamp ($r.ts$), a session ID ($r.sid$), and a set of objects accessed by r ($r.objs$). If r is a commit request, $r.objs$ denotes the set of all objects modified by the committed transaction. For example, in Figure 1, $r_5.objs$ is $\{1, 3\}$, since the transaction T modifies objects 1 and 3 in r_1 and r_3, respectively. S denotes a set of all sessions while O denotes a set of all objects in the workload. As in [28], an object can be a table or a table partition. Requests are classified into non-commit (NC) (i.e., SELECT or UPDATE) and commit (C) requests making their updates permanent. If a transaction does not update any object, its commit request has no dependencies to requests in the other sessions. Thus, it is ignored during the dependency graph generation phase but simply replayed at the end of the transaction during the replay phase. Requests within a session are constrained to be replayed in order of timestamp, as imposed in $\mathcal{E}_{\text{ses}}$. Since requests in a session are stored in a timestamp order [28], we do not need to store $\mathcal{E}_{\text{ses}}$ explicitly. For ease of explanation, we assume that there exists r_0 and r_∞ which correspond to the virtual initial and last requests. Note that $\mathcal{G}_{\text{ini}} = (\mathcal{V}_{\mathcal{R}}, \mathcal{E}_{\text{ses}})$ is not the dependency graph since it contains the ordering constraints within each session only. Table 2 shows the notations used throughout the paper.

We first define predicates to define other dependency (binary) relations [28]. Consider two requests $r, r' \in \mathcal{V}_{\mathcal{R}}$. The predicate $precedes(r, r')$ returns *true* if $r.ts < r'.ts$ or *false* otherwise; we define the set of precedence-dependent edges, $\mathcal{E}_{pre} = \{(r, r') \in {\mathcal{V}_{\mathcal{R}}}^2 \mid precedes(r, r')\}$. $commit(r)$ returns *true* if r is a commit request or *false* otherwise; we define the set of commit-dependent edges, $\mathcal{E}_{com} = \{(r, r') \in \mathcal{E}_{pre} \mid commit(r) \vee commit(r')\}$. $access\_common\_obj(r, r')$ returns *true* if r and r' access a common object (i.e., $r.objs \cap r'.objs \neq \emptyset$) or *false* otherwise; we define the set of object-dependent edges, $\mathcal{E}_{obj} = \{(r, r') \in \mathcal{E}_{pre} \mid access\_common\_obj(r, r')\}$. Finally, we define the set of collision-dependent edges, $\mathcal{E}_{\text{col}} = \mathcal{E}_{com} \cap \mathcal{E}_{obj}$. When $(r, r') \in \mathcal{E}_{\text{col}}$, we say that r is collision-dependent to r'. We use the terms "dependency relation" and "edge set" interchangeably.

Given an initial graph $\mathcal{G}_{\text{ini}} = (\mathcal{V}_{\mathcal{R}}, \mathcal{E}_{\text{ses}})$, a dependency graph generation algorithm generates a new edge set according to its specific relation. For example, a dependency graph algorithm can

Table 2. Notations mainly used in this paper.

Notation	Description
$\mathcal{W}$	a workload
$\mathcal{V}_\mathcal{R}$	a set of requests in $\mathcal{W}$
r	a request
$r.ts$	the timestamp of r
$r.objs$	a set of objects accessed by r
$r.sid$	the session ID of r
S	the set of sessions in $\mathcal{W}$
O	the set of accessed objects in $\mathcal{W}$
$\mathcal{G}^{tr}$	transitive reduction of a graph $\mathcal{G}$
$\mathcal{G}^{tc}$	transitive closure of a graph $\mathcal{G}$
$\mathcal{G}[P]$	the induced subgraph of $\mathcal{G} = (\mathcal{V}_\mathcal{R}, \mathcal{E})$ for a subset of vertices $P \subset \mathcal{V}_\mathcal{R}$
$V(\mathcal{G})$	the set of vertices of $\mathcal{G}$
$E(\mathcal{G})$	the set of edges of $\mathcal{G}$

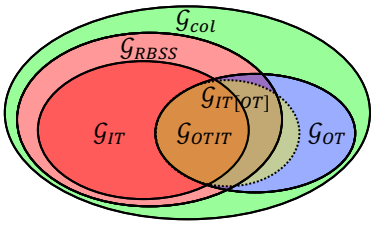


Fig. 3. Venn diagram for dependency graphs using the edge set containment relationships.

generate a dependency graph $\mathcal{G}_{\text{col}} = (\mathcal{V}_\mathcal{R}, \mathcal{E}_{\text{col}})$ by checking both the commit dependency and the object dependency.

One extreme dependency graph is a totally ordered dependency graph $\mathcal{G}_{\text{total}}$, constructed by connecting requests in increasing timestamp order. Using $\mathcal{G}_{\text{total}}$, the requests in $\mathcal{V}_\mathcal{R}$ are executed serially, leading to seriously limited concurrency [28], since the critical path of $\mathcal{G}_{\text{total}}$ is the longest among those of all possible dependency graphs. The other extreme graph is a minimal dependency graph $\mathcal{G}_{\text{min}} = (\mathcal{V}_\mathcal{R}, \mathcal{E}_{\text{min}})$, with which we can consistently replay $\mathcal{W}$ without any unnecessary synchronization overhead. Here, $\mathcal{G}_{\text{min}}$ can be constructed by removing all redundant edges in $\mathcal{E}_{\text{col}}$, i.e., by executing a transitive reduction algorithm on $\mathcal{G}_{\text{col}}$.

Now, we provide a formal definition of our problem:

Problem Definition 1. *Given a workload modeled as the initial graph $\mathcal{G}_{\text{ini}} = (\mathcal{V}_\mathcal{R}, \mathcal{E}_{\text{ses}})$, generate a compact dependency graph $\mathcal{G} = (\mathcal{V}_\mathcal{R}, \mathcal{E})$ efficiently such that $\mathcal{G}^{tr} = \mathcal{G}^{tr}_{col}$.*

We now provide the overview for the dependency graphs based on the types of redundant edges that each dependency graph prunes from $\mathcal{G}_{\text{col}}$. Figure 3 shows the containment relationships among the dependency graphs. Note that each dependency graph avoids specific types of redundant edges from $\mathcal{G}_{\text{col}}$. The IT-free graph $\mathcal{G}_{\text{IT}}$ and OT-free graph $\mathcal{G}_{\text{OT}}$ are those that respectively avoid redundant edges due to IT and OT, as explained in Section 1. $\mathcal{G}_{\text{RBSS}}$, generated by the generation step (i.e., before transitive reduction) in [28], is a special case of IT-free graph. The OTIT-free graph $\mathcal{G}_{\text{OTIT}}$ is the one that avoids redundant edges due to either IT or OT. The IT[OT]-free graph $\mathcal{G}_{\text{IT[OT]}}$ is the one that prunes edges redundant due to OT first and then prunes the edges redundant due to

IT. $E(\mathcal{G}_{\text{IT[OT]}}) \supseteq E(\mathcal{G}_{\text{OTIT}})$, as $\mathcal{G}_{\text{IT[OT]}}$ loses some IT connectivity after pruning redundant edges due to OT.

For example, in Figure 1, $\mathcal{G}_{\text{IT}}$ avoids all red and green dotted edges, whereas $\mathcal{G}_{\text{OT}}$ does all blue and green ones. $\mathcal{G}_{\text{OTIT}}$ avoids all dotted edges except for (r_5, r_{10}). Note that $(r_5, r_{13}) \notin E(\mathcal{G}_{\text{OTIT}})$ but $(r_5, r_{13}) \in E(\mathcal{G}_{\text{IT[OT]}})$. This is because, $\mathcal{G}_{\text{IT[OT]}}$ removes (r_6, r_9) first and thus the path $(r_5, r_6, r_9, r_{11}, r_{13})$ disappears in the resulting dependency graph. The edge (r_5, r_{10}) is redundant due to reachability other than IT and OT. However, this type of redundancy is relatively infrequent; only 5.3% of the edges in $\mathcal{G}_{\text{OTIT}}$ are redundant in Table 1. Pruning those edges requires computing transitive closure. Thus, we allow such edges in the generation step.

3.2 IT(k)-free Graph

We first define the concept of the k-forward path in two different sessions. We then define the IT(k)-free graph, using the k-forward path. Among the IT(k)-free graphs, we specifically discuss two of the special graphs IT(1)-free graph and IT(∞)-free graph. Finally, we explain why we consider two different sessions only.

In order to define redundant edges in $\mathcal{E}_{\text{col}}$ but not in the IT(k)-free graph, we first define the concept of the *k-forward path*. Given two vertices, r in session s and r' in session s' ($s \neq s'$), a path $(r, r_1^s, \cdots, r_i^s, r_1^{s'}, \cdots r_k^{s'}, r')(i \geq 0, k \geq 0, i + k \geq 1)$ is called k-forward-path from r to r' in $\mathcal{E}_{\text{col}} \cup \mathcal{E}_{\text{ses}}$ where all r^s's are in session s and $r^{s'}$'s are in session s'. Here, k is the length of the subpath consisting of vertices in session s'. Given an edge (r, r'), if there exists a k-forward-path from r to r', then (r, r') is redundant. This type of redundancy is referred to as inter-session transitivity (IT) because a k-forward path exists along requests in two different sessions (s and s'). For example, a path (r_5, r_6, r_8, r_{12}) in Figure 1 is a 1-forward path from r_5 to r_{12}. Therefore, (r_5, r_{12}) is redundant. Thus it is pruned in the IT(1)-free graph.

We then formally define a new redundancy-free dependency graph, the IT(k)-free graph using the k-forward-path. We first define a new notion of the redundancy-free dependency edge set. For generality, we define it over an arbitrary edge set $\mathcal{E}$: $IT_k(\mathcal{E}) = \{(r, r') \in \mathcal{E} \mid \nexists\ i\text{-forward path } p \text{ from } r \text{ to } r' \text{ in } \mathcal{E} \cup \mathcal{E}_{\text{ses}} \text{ where } 0 \leq i \leq k\}$. That is, every edge in $IT_k(\mathcal{E})$ lacks any forward path of length k or less. Definition 1 defines the graph where the edge set is $IT_k(\mathcal{E}_{\text{col}})$.

Definition 1. *An IT(k)-free graph $\mathcal{G}_{\text{IT(k)}} = (\mathcal{V}_{\mathcal{R}}, \mathcal{E}_{\text{IT(k)}})$ for $\mathcal{G}_{\text{ini}}$ is a dependency graph where $\mathcal{E}_{\text{IT(k)}} = IT_k(\mathcal{E}_{\text{col}})$.*

The dependency graph $\mathcal{G}_{\text{RBSS}} = (\mathcal{V}_{\mathcal{R}}, \mathcal{E}_{\text{RBSS}})$, generated by the generation step (i.e., before transitive reduction) of the dependency graph generation algorithm in [28], is in between $\mathcal{G}_{\text{IT(0)}}$ and $\mathcal{G}_{\text{IT(1)}}$ (i.e., $\mathcal{E}_{\text{IT(1)}} \subseteq \mathcal{E}_{\text{RBSS}} \subseteq \mathcal{E}_{\text{IT(0)}}$). We will discuss this in Section 4.

Although the $\mathcal{G}_{\text{RBSS}}$ provides higher replay concurrency than $\mathcal{G}_{\text{total}}$, it could have many redundant edges, resulting in severe overheads during replay. One can execute a transitive reduction algorithm on $\mathcal{G}_{\text{RBSS}}$. However, the overall performance of the dependency graph generation would be slow due to the high overhead of unnecessarily redundant edge generation and removal. For example, in Table 1, 99.6% of the edges in $\mathcal{G}_{\text{RBSS}}$ are redundant. This motivates us to define a much more compact dependency graph for efficient generation and transitive reduction.

The IT(∞)-free graph (simply denoted as the IT-free graph) is a general case of IT(k)-free graph where the edges do not have any forward paths of any length (i.e., $k = \infty$). The IT-free graph ensures the pruning of all redundant edges in two inter-sessions. For brevity, we denote $IT_\infty(\mathcal{E})$ as $IT(\mathcal{E})$ in the following sections.

Although we can generalize the definition of IT for three or more sessions, pruning such redundant edges will require expensive self-join operations over the edge set $IT(\mathcal{E})$ being constructed so far. For example, consider constructing $IT(\mathcal{E})$ for three sessions. Then, we need to perform

a three-way self-join over $IT(\mathcal{E})$ using quantifiers E_1, E_2 and E_3 where non-equi join conditions include "$E_1.src\_ts \leq E_2.src\_ts$ AND $E_1.dst\_ts \geq E_2.dst\_ts$ AND $E_2.src\_sid <> E_3.dst\_sid$." This would be even more expensive than transitive closure.

3.3 OT-free Graph

Although the IT-free graph removes some redundant edges, many redundant edges remain, since it considers inter-session redundancy due to two sessions only. For example, in Table 1, 73.7% of the edges in the IT-free graph are still redundant. We analyze that the most redundant edges in the IT-free graph are due to the *transitive* property of collision-dependent edges where the source and the target vertex of each edge access the same object. We call this redundancy *object transitivity (OT)*. That is, OT captures important and dominant redundancy among multiple sessions, which can be also detected efficiently, as we will see in Section 5.2. In the SD Benchmark for Table 1, 98% of the redundant edges in the IT-free graph have OT.

We first formally define another important type of redundancy-free dependency edge sets inspired by object transitivity-free (OT-free) edges. To define the OT-free edge set, we define a predicate. Consider two requests $r, r' \in \mathcal{V}_{\mathcal{R}}$. The predicate $access\_obj(r, o)$ returns $true$ if and only if $o \in r.objs$. We then define the OT-free graph in Definition 2. For example, in Figure 1, the OT-free graph does not have (r_6, r_9), since (r_6, r_8, r_9) exists due to OT.

- **Set of collision-dependent edges for object** o
  $C^o(\mathcal{E}_{\text{col}}) = \{(r, r') \in \mathcal{E}_{\text{col}} \,|access\_obj(r, o) \wedge access\_obj(r', o)\}$
- **Set of object transitivity-free edges for object** o
  $OT^o(\mathcal{E}_{\text{col}}) = \{(r, r') \in \mathcal{E}_{\text{col}} \,|\nexists$ a path p $= (r, r''_1, \cdots, r''_n, r')\ (n \geq 1)$ where all edges $(r, r''_1), \cdots, (r''_n, r') \in C^o(\mathcal{E}_{\text{col}})\}$. The path p is called *an object transitive path.*
- **Set of object transitivity-free edges**
  $OT(\mathcal{E}_{\text{col}}) = \bigcap_{o \in O} OT^o(\mathcal{E}_{\text{col}})$

Definition 2. *A OT-free graph* $\mathcal{G}_{\text{OT}} = (\mathcal{V}_{\mathcal{R}}, \mathcal{E}_{\text{OT}})$ *for* $\mathcal{G}_{\text{ini}}$ *is a dependency graph where* $\mathcal{E}_{\text{OT}} = OT(\mathcal{E}_{\text{col}})$.

3.4 OTIT-free Graph

A more concise dependency graph can be obtained by applying both $IT(\mathcal{E})$ and $OT(\mathcal{E})$ (i.e., removing both types of redundant edges). Among all possible compositions (i.e., $IT(OT(\mathcal{E}))$, $OT(IT(\mathcal{E}))$ and $OT(\mathcal{E}) \cap IT(\mathcal{E})$), the most concise dependency graph is one without any redundant edges, which are absent in either the OT-free or the IT-free graph. We call the graph OTIT-free graph.

The edge set of the OTIT-free graph is simply an intersection of the edge sets of OT-free and IT-free graphs, as defined in Definition 3. Note that the order of OT and IT is independent of the definition.

Definition 3. *A OTIT-free graph* $\mathcal{G}_{\text{OTIT}} = (\mathcal{V}_{\mathcal{R}}, \mathcal{E}_{\text{OTIT}})$ *for* $\mathcal{G}_{\text{ini}}$ *is a dependency graph where* $\mathcal{E}_{\text{OTIT}} = OT(\mathcal{E}_{\text{col}}) \cap IT(\mathcal{E}_{\text{col}})$.

3.5 IT[OT]-free Graph

The OTIT-free graph has disadvantages in graph generation efficiency as it is biased toward conciseness. We will discuss the specific time complexity in Section 5, but intuitively, the cost of determining which edge is in both $IT(\mathcal{E})$ and $OT(\mathcal{E})$ is expensive. We observe that IT[OT]-free graph, which can be obtained by applying $IT(\mathcal{E})$ after $OT(\mathcal{E})$, can be generated much more efficiently while adding only a few more redundant edges compared to the OTIT-free graph (in TPC-C, 1.3% more). In this section, we propose an IT[OT]-free graph, a compact graph that can

be efficiently generated, which will be empirically verified in Section 7. We formally define the concept of the IT[OT]-free graph in Definition 4.

Definition 4. *An IT[OT]-free graph* $\mathcal{G}_{\text{IT[OT]}} = (\mathcal{V}_{\mathcal{R}}, \mathcal{E}_{\text{IT[OT]}})$ *for* $\mathcal{G}_{\text{ini}}$ *is a dependency graph where* $\mathcal{E}_{\text{IT[OT]}} = IT(OT(\mathcal{E}_{\text{col}}))$.

By changing the order of OT and IT, we can obtain the OT[IT]-free graph. However, the OT[IT]-free graph is poor in efficiency compared to the IT[OT]-free graph since computing $OT^o(\mathcal{E}_{\text{IT}})$ for each object $o \in O$ is expensive. Therefore, we omit the explanation for the OT[IT]-free graph. We will discuss this in Section 5 before explaining how to generate each graph.

Although $\mathcal{G}_{\text{OTIT}}$ is the smallest, DoppelGanger++ opts for $\mathcal{G}_{\text{IT[OT]}}$ since it strikes a balance between size and efficiency. That is, in our extensive experiments, $\frac{|\mathcal{E}_{\text{IT[OT]}}|}{|\mathcal{E}_{\text{OTIT}}|}$ is at most 1.09, although the generation cost of $\mathcal{G}_{\text{OTIT}}$ is, on average, 3.2 times higher than that of $\mathcal{G}_{\text{IT[OT]}}$.

3.6 Output Determinism Guarantees

We first show that our capture and replay algorithms guarantee output determinism under (transaction-level) snapshot isolation. The snapshot isolation level prevents dirty read, non-repeatable read, and phantom read. To ensure output determinism, we must ensure that 1) each replayed request r returns the same output as during the capture time, and 2) r changes the database state to be the same as it was during the capture time. Here, we assume no random inconsistency exists as in [28]; a statement with the same input reading the same database state always results in the same output. In snapshot isolation, each transaction acquires its own snapshot with its start timestamp and reads (i.e., possibly by multiple statements) the snapshot of data committed before the timestamp. The transaction can be committed when no write conflict occurs. To ensure 1), each replayed transaction must see the same snapshot as during the capture time. For this, DRS records the snapshot timestamp of each transaction as the timestamps of its non-commit requests. In $\mathcal{G}_{\text{col}}$, the ordering between each non-commit request r and commit requests modifying some object in $r.objs$ is preserved. During replay, DRS needs to ensure that each non-commit request reads the snapshot of $r.objs$ committed before its timestamp. With $\mathcal{G}_{\text{col}}$, 2) is also guaranteed, since the commit requests with overlapping write sets are scheduled serially, avoiding any write conflict. Note that some DBMSs such as PostgreSQL claim to support repeatable read isolation, but the phantom read is not allowed in the level [6], while others such as SQL Server holds shared locks on all data read by each statement until the transaction completes. In both cases, the read stability condition (i.e., repeatable read) is satisfied if we use the same concurrency control under the same isolation level for both capture and replay.

Now we explain why our algorithms also guarantee output determinism for a weaker isolation level, statement-level snapshot isolation. In statement-level snapshot isolation, each statement acquires its own snapshot with its snapshot timestamp and reads the data committed before the timestamp, thereby allowing non-repeatable or phantom reads. Unlike transaction-level snapshot isolation, we also capture the timestamp of each non-commit request. During replay, we ensure that each non-commit request reads the snapshot of the objects committed before its timestamp.

Our correctness guarantees hold for all dependency graphs we propose. This is because removing redundant edges does not change the order of execution of requests. For example, in Figure 1, consider the redundant edge (r_4, r_9). If we remove the edge, r_9 still will be scheduled after r_4 due to the path (r_4, r_8, r_9).

Partition-level dependency ensures output determinism as table-level dependency does, avoiding the replay inconsistency problem that occurs in block-level or row-level dependency [28]. When we partition a table, we have knowledge of the partitioning information. Consequently, our system

can identify the appropriate partitions for a query using this information, whereas block-level or row-level dependency falls short.

Commercial DBMSs typically support either hash-based or range-based partitioning. Thus, once a query is given, our DRS identifies all related partitions accessed by the query using partition pruning techniques, such as [22]. If a query contains predicates on non-partitioning keys, the DRS identifies *all* partitions of table R even if some partitions may not have tuples relevant to the query. That is, our DRS takes a conservative approach to guarantee the correctness.

For example, consider a table $R(A, B, C)$ range partitioned on A: {[1-10], [11-20], [21-30], ..., [9990-10000]}. Assume that the two transactions in Listing 2 are serially executed at the capture time. Here, a transaction T1 uses a predicate on a non-partitioning key B, while a transaction T2 uses a predicate on the partitioning key A (i.e., $A = 1$). Then, the SELECT request of T1 is associated with all partitions of R, while both requests (i.e., UPDATE and COMMIT requests) of T2 are associated with the first partition only (note that the COMMIT request of T1 is ignored, since T1 does not update any object as explained in Section 3.1). Thus, the SELECT request of T1 and the COMMIT request of T2 are correctly ordered in the dependency graph.

Listing 2. Transaction examples

```
T1: SELECT * FROM R WHERE B<20; COMMIT;
T2: INSERT INTO R VALUES (1,5,20); COMMIT;
```

4 REPETITIVE BACKWARD SESSION SCAN (RBSS)

This section reviews how RBSS [28] generates a dependency graph $\mathcal{G}_{\text{RBSS}}$ from $\mathcal{G}_{\text{ini}}$. Note that RBSS refers to the generation algorithm, excluding the transitive reduction. We also provide the time complexity of RBSS and the relationship between $\mathcal{G}_{\text{RBSS}}$ and $\mathcal{G}_{\text{IT(k)}}$.

RBSS generates the incoming edges of each request $r' \in \mathcal{V}_{\mathcal{R}}$, by iterating over every session $s \in S$ except for $s' = r'.sid$, and performing a backward scan with a time interval $(r_{min}.ts, r_{max}.ts)$. It stops the scan whenever we find r collision-dependent to r'. Assume that r_{prev} is the previous request of r' in session s'. Then, r_{max} is the earliest request after r' in session s, while r_{min} is a request in session s which must wait for r_{prev}. Note that the edge (r_{min}, r_{prev}) needs to be generated before the generation of incoming edges of r'. For example, we explain how to compute the time interval of r_7 in Figure 4. Since the (r_2, r_4) is generated when we process r_4, r_{min} is set to r_2. r_{max} is set to r_8.

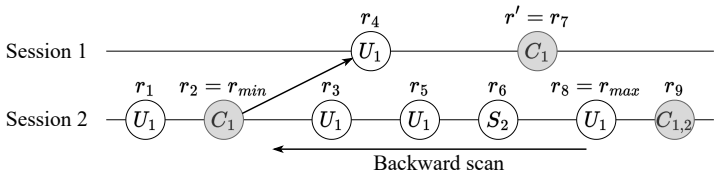


Fig. 4. How to determine the scan range $(r_{min}.ts, r_{max}.ts)$ in session 2 for r_7.

Time complexity. The time complexity of RBSS is $O(|\mathcal{V}_{\mathcal{R}}|^2)$. We assume that requests are evenly distributed among sessions. Also, we assume that finding r_{max} takes $O(log\frac{|\mathcal{V}_{\mathcal{R}}|}{|S|})$, and all the other subprocedures take constant time. In the worst case, RBSS must access all preceding requests of r' in every session except for $r'.sid$ (i.e., r_{min} can be r_0). Thus, the overall time complexity is $O(\sum_{i=1}^{|\mathcal{V}_{\mathcal{R}}|}(i + |S| \cdot log\frac{|\mathcal{V}_{\mathcal{R}}|}{|S|}) - |S| \cdot \sum_{j=1}^{|\mathcal{V}_{\mathcal{R}}|/|S|} j) = O(|\mathcal{V}_{\mathcal{R}}|^2)$.

In order to analyze experimental results using the time complexity analysis, we derive a tighter bound using a concept of the *average backward distance*, d_{avg}, an average number of requests to scan to find an incoming edge from a session for each request. Finally, we obtain $O(|\mathcal{V}_\mathcal{R}| \cdot (|S| - 1) \cdot (d_{avg} + log\frac{|\mathcal{V}_\mathcal{R}|}{|S|}))$ as the time complexity of RBSS.

We explain the relationship between the IT(k)-free graph and the dependency graph $\mathcal{G}_{\text{RBSS}} = (\mathcal{V}_\mathcal{R}, \mathcal{E}_{\text{RBSS}})$. Note that [28] did not define this graph; thus it is difficult to understand and analyze the properties of $\mathcal{G}_{\text{RBSS}}$. Lemma 1 establishes this relation. We briefly explain the intuition behind the lemma's proof. RBSS prunes every edge (r, r') with a 0-forward path $(r, \cdots, r_i^s, r')(i \geq 1)$ (i.e., $\mathcal{E}_{\text{RBSS}} \subseteq \mathcal{E}_{\text{IT}(0)}$). Suppose that there is a 0-forward path $(r, \cdots, r_i^s, r')(i \geq 1)$ for (r, r') in $\mathcal{G}_{\text{RBSS}}$. Then, during the backward scan in session s to generate the incoming edges of r', RBSS must find r_i^s and then stop. We now explain why RBSS fails to prune *some* edge $(r, r') \in \mathcal{E}_{\text{col}}$ with 1-forward path $(r, r_1^s, \cdots, r_i^s, r_{prev}, r')$ $(i \geq 0)$ (i.e., $\mathcal{E}_{\text{IT}(1)} \subseteq \mathcal{E}_{\text{RBSS}}$). As soon as (r_i^s, r_{prev}) is pruned from $\mathcal{G}_{\text{RBSS}}$, RBSS does not continue to prune the redundant edge (r, r') due to its 1-forward path. The following lemma shows the containment relationships among these dependency graphs. All formal lemmas and proofs are in the technical report [7]. We refer interesting readers to it for the details.

Lemma 1. $\mathcal{E}_{\text{IT}(1)} \subseteq \mathcal{E}_{RBSS} \subseteq \mathcal{E}_{\text{IT}(0)}$.

5 STATEFUL SINGLE FORWARD SCAN (SSFS)

This section describes our efficient dependency graph generation algorithm called SSFS using states. We first describe a common algorithm that generates the dependency graph using states. We then explain what states should be maintained and how to generate the dependency graph from the states for each type of dependency graph. SSFS can generate any dependency graph in Section 3.

We consider $\mathcal{V}_\mathcal{R}$ as a sequence of vertices according to their timestamp. We assume that the timestamp t is assigned by a monotonically increasing logical timer $(t \geq 1)$. A dependency graph generation algorithm takes $\mathcal{V}_\mathcal{R}$, processes the requests, and outputs a dependency relation $\mathcal{E} \in \{\mathcal{E}_{\text{OT}}, \mathcal{E}_{\text{IT}}, \mathcal{E}_{\text{OTIT}}, \mathcal{E}_{\text{IT[OT]}}\}$. We omit how to generate the OT[IT]-free graph since it requires computing $OT^o(\mathcal{E}_{\text{IT}})$ for each object $o \in O$, which is costly. Given an object o, computing $OT^o(\mathcal{E}_{\text{IT}})$ is equivalent to computing the transitive reduction of $(\mathcal{V}_\mathcal{R}, C^o(\mathcal{E}_{\text{IT}}))$, which takes $O(|C^o(\mathcal{E}_{\text{IT}})| + |S||\mathcal{V}_\mathcal{R}| + |S||OT^o(\mathcal{E}_{\text{col}})|)$ [32, 34]. On the other hand, we can compute $OT(\mathcal{E}_{\text{col}})$ efficiently from the observation Section 5.2. Our experiments also show that IT[OT]-free graph is efficient in both conciseness and efficiency in Section 7. We omit how to generate $\mathcal{E}_{\text{OTIT}}$ since it is obtained by the intersection of $\mathcal{E}_{\text{OT}}$ and $\mathcal{E}_{\text{IT}}$.

Algorithm 1 describes SSFS. SSFS first initializes a current edge set incrementally appended (Line 1) and the states (Line 2). SSFS iterates over all requests r' in $\mathcal{V}_\mathcal{R}$ in increasing timestamp order (Line 3). For each request r', SSFS generates a set of the incoming edges of r', $\mathcal{E}_{r'}$, using *states* (Line 4) and then updates *states* (Line 5). Then SSFS appends the generated edge set to the current edge set (Line 6). At the end of the iteration, SSFS finally returns the generated dependency graph (Line 7).

5.1 Generating $\mathcal{G}_{\text{IT}}$

Consider the current request r' of session s' at timestamp t. Since $\mathcal{E}_{\text{IT}(0)} \supseteq \mathcal{E}_{\text{IT(k)}}$ for any $k > 0$, we first find every edge (r, r') in $\mathcal{E}_{\text{IT}(0)}$ from each session s $(\neq s')$ (Case $k = 0$) and prune it if there exists a $k(\geq 1)$-forward path between r and r' (Case $k > 0$). That is, those not pruned must be in $\mathcal{E}_{\text{IT(k)}}$. For example, in Figure 5, we assume that the current request to process is r_7 $(= r')$, i.e., a commit request. Then, from session 1, we find $(r_6, r_7) \in \mathcal{E}_{\text{IT}(0)}$ and do not prune it since there is no

Algorithm 1: Stateful Single Forward Scan (SSFS)

Input: A workload $\mathcal{G}_{\text{ini}} = (\mathcal{V}_{\mathcal{R}}, \mathcal{E}_{\text{ses}})$, the dep. graph type T_G

```
1 E_curr ← ∅                                              // The current edge set
2 states ← ∅                                              // The states
3 for r' ∈ V_R in increasing timestamp order do
4 |  E_r' ← GenerateIncomingEdges (r', states, T_G)
5 |  states ← UpdateStates(r, states, T_G)
6 |  E_curr ← E_curr ∪ E_r'
7 return (V_R, E_curr)
```

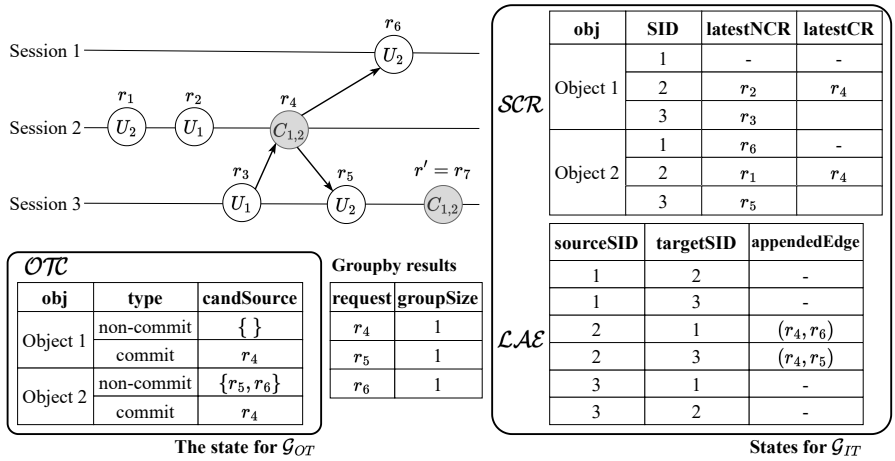


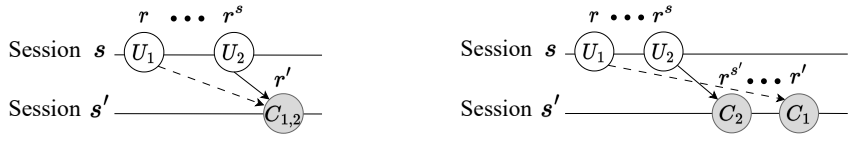
Fig. 5. The states for $\mathcal{G}_{\text{IT}}$ and $\mathcal{G}_{\text{OT}}$ when the current request $r' = r_7$. We show edges in $\mathcal{E}_{\text{OTIT}}$ only.

k-forward path from r_6 to r_7. On the other hand, from session 2, we first find $(r_4, r_7) \in \mathcal{E}_{\text{IT}(0)}$, and prune it since there is a 1-forward path (r_4, r_5, r_7).

For each session s ($\neq s'$), to ensure that $(r, r') \in \mathcal{E}_{\text{IT}(0)}$ (Case $k = 0$), r must be *the latest collision-dependent request* to r' in session s (see Figure 6a). Otherwise, there exists the latest collision-dependent request $r^s (\neq r)$ to r' in session s, and thus there exists a 0-forward path $(r, \cdots, r^s, r')$, contradicting the definition of $\mathcal{E}_{\text{IT}(0)}$.

For each edge $(r, r') \in \mathcal{E}_{\text{IT}(0)}$, to ensure that $(r, r') \in \mathcal{E}_{\text{IT}}$ (Case $k \geq 1$), no edge $(r^s, r^{s'}) \in \mathcal{E}_{\text{IT}(0)}$ such that $r^s.sid = s$, $r^s.ts \geq r.ts$, $r^{s'}.sid = s'$, and $r^{s'}.ts < t$ has been appended to the dependency graph being currently constructed (see Figure 6b). Otherwise, there exists a k-forward path $(r, \cdots, r^s, r^{s'}, \cdots, r')$, which is contradictory to the definition of $\mathcal{E}_{\text{IT(k)}}$.

States for $\mathcal{G}_{\text{IT}}$: To generate the incoming edges for $\mathcal{G}_{\text{IT}}$, we maintain two nested tables as states: the session-wise collision requests ($\mathcal{SCR}$) and the latest appended edges between sessions ($\mathcal{LAE}$). $\mathcal{SCR}$ stores the latest non-commit and commit requests for every pair of (obj, SID). Given an object o and a session s, we can retrieve and update the latest non-commit and commit requests accessing o in session s using $\mathcal{SCR}$. $\mathcal{SCR}$ is implemented in a two-dimensional array, enabling lookup and update operations to be performed in $O(1)$. For example, when we process $r_7 (= r)$ in Figure 5, $\mathcal{SCR}$ stores r_2 and r_4 as the latest non-commit and commit requests accessing object 1 in session 2, respectively. Likewise, $\mathcal{SCR}$ stores r_1 and r_4 as the latest non-commit and commit requests



(a) Case $\mathbf{k} = \mathbf{0}$ (b) Case $\mathbf{k} > \mathbf{0}$

Fig. 6. The two cases where there exists a k-forward path from r in session s to r' in session s'.

accessing object 2 in session 2, respectively. $\mathcal{LAE}$ stores the most recently appended edge in $\mathcal{E}_{\text{IT}(0)}$ for every pair of sessions, supporting lookup and update as well. $\mathcal{LAE}$ is also implemented in a two-dimensional array, enabling the operations to be performed in $O(1)$. For example, in the same figure, the fourth tuple in $\mathcal{LAE}$ represents the most recently appended edge (r_4, r_5) from session 2 to session 3.

We now describe a detailed algorithm for generating the incoming edges of r' for $\mathcal{G}_{\text{IT}}$. Given r', for each pair (s, o) such that $o \in r'.objs$, we retrieve the latest non-commit and commit requests from $\mathcal{SCR}$, that are collision-dependent to r'. Among all retrieved requests for r', if r' is a commit request, we retrieve the latest request r in session s in $O(|r'.objs|)$, otherwise we retrieve the latest commit request r in session s. Then, since $(r, r') \in \mathcal{E}_{\text{IT}(0)}$, we retrieve the latest appended edge $(r^s, r^{s'})$ from $\mathcal{LAE}$ using s and s' in $O(1)$. If not found, we generate (r, r') as a new edge. Otherwise, if $r.ts > r^s.ts$, we generate (r, r'). After (r, r') is generated, we update the tuple in $\mathcal{LAE}$ using s and s' in $O(1)$ by replacing the value of the *appendedEdge* column with (r, r'). For each request, updating $\mathcal{SCR}$ requires $O(|r'.objs|)$, while updating $\mathcal{LAE}$ requires $O(|S|)$, the upper bound of the number of incoming edges of r'.

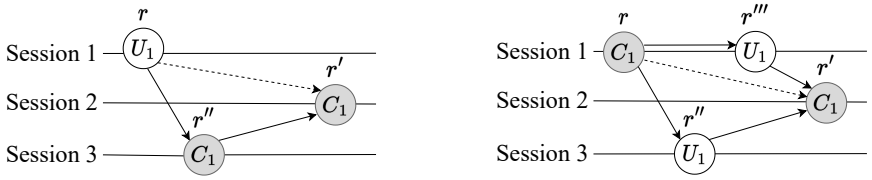
Now, we analyze the time and space complexity of edge generation and state maintenance for $\mathcal{G}_{\text{IT}}$. For each request r', we need to scan every tuple in $\mathcal{SCR}$ whose object ID $\in r'.objs$. Scanning each tuple in $\mathcal{SCR}$ takes $O(1)$ and thus, the time complexity of incoming edge generation for r' is $O(|r'.objs| \cdot |S|)$. The time complexity of maintaining $\mathcal{SCR}$ and $\mathcal{LAE}$ for each request is $O(max(|r'.objs|, |S|))$. If $|S|$ and $|r'.objs|$ are regarded as constants, $O(1)$ is guaranteed for each request. The space complexity for the states is $O(|S| \cdot |O| + |S|^2)$, since the sizes of $\mathcal{SCR}$ and $\mathcal{LAE}$ are $O(|S| \cdot |O|)$ and $O(|S|^2)$, respectively.

5.2 Generating $\mathcal{G}_{\text{OT}}$

Consider the current request r' accessing $r'.objs$ at timestamp t. To ensure that $(r, r') \in \mathcal{E}_{\text{OT}}$, for every object $o \in r.objs \cap r'.objs$, there exists no object transitive path from r to r'. Otherwise, $(r, r') \notin OT^o(C^o(\mathcal{E}_{\text{col}}))$ for some o (see the definitions of OT^o and C^o). Therefore, for each $o \in r'.objs$, we find the candidate source vertices of the incoming edges of r' for $OT^o(C^o(\mathcal{E}_{\text{col}}))$. We then group them by vertex ID to prune any candidate vertex r whose group size does not match $|r'.objs \cap r.objs|$. For example, in Figure 5, we assume that the current request is r_7 $(= r')$. For object 1, we find r_4 since $(r_4, r_7) \in OT^1(C^1(\mathcal{E}_{\text{col}}))$. Similarly, for object 2, we find r_5 and r_6 since $(r_5, r_7), (r_6, r_7) \in OT^2(C^2(\mathcal{E}_{\text{col}}))$. The results of the group by operation are also shown in Figure 5. Since the group size for r_4 (=1) $\neq |r_7.objs \cap r_4.objs|$ (=2), we discard r_4. However, since the group size for r_5 (=1) $= |r_7.objs \cap r_5.objs|$, we generate (r_5, r_7). Similar to (r_5, r_7), we generate (r_6, r_7).

To ensure that $(r, r') \in OT^o(C^o(\mathcal{E}_{\text{col}}))$ for $o \in r'.objs$, if r' is a non-commit request, r must be *the latest commit request accessing* o. Otherwise, there exists the latest commit request r'' $(\neq r)$ accessing o, and thus there exists an object transitive path (r, r'', r'), which is contradictory to the definition of $OT^o(C^o(\mathcal{E}_{\text{col}}))$. When r' is a commit request, r can be either a commit request or a non-commit request accessing o. If r is a commit request, r must be the latest request accessing o,

and there must be no non-commit request r'' accessing o after r. Otherwise, an object transitive path (r, r'', r') exists (see Figure 7a). If r is a non-commit request, there exists no commit request r'' accessing o after r. Otherwise, an object transitive path (r, r'', r') exists (see Figure 7b). We call a set of non-commit requests accessing o *after the latest commit request accessing* o *a latest non-commit request set for object* o. For example, in Figure 5, $\{r_5, r_6\}$ is the latest non-commit request set for object 2.



(a) r is a commit request. (b) r is a non-commit request.

Fig. 7. The two cases where there exists an object transitive path for object o from r to r'.

We store in a temporary table all retrieved candidate source vertices for $r'.objs$ in a temporary table. We then group them by the source vertex ID. If the group size of a source vertex r is equal to $|(r.objs \cap r'.objs)|$ (i.e., for every $o \in r.objs \cap r'.objs$, $(r, r') \in OT^o(C^o(\mathcal{E}_{\text{col}}))$), we generate the edge (r, r') as a new edge to $\mathcal{G}_{\text{OT}}$. Otherwise, we discard r.

The state for $\mathcal{G}_{\text{OT}}$: To generate the incoming edges for $\mathcal{G}_{\text{IT}}$, we maintain a nested table, the OT-free candidate table, $\mathcal{OTC}(obj, type, candSource)$ as a state. Each tuple in $\mathcal{OTC}$ corresponds to every pair of $(obj, type)$. If $type$ is COMMIT, $candSource$ stores the latest commit request accessing obj. Otherwise, $candSource$ stores the latest non-commit request set for obj. Given an object o and the request type $type$, we can retrieve and update the latest commit request accessing o or the latest non-commit request set for o depending on $type$, using $\mathcal{OTC}$. For example, when processing $r_7 (= r')$ in Figure 5, for $obj = 2$ and $type =$ NON-COMMIT, $\mathcal{OTC}$ stores $\{r_5, r_6\}$ as a latest non-commit request set. For $obj = 2$ and $type =$ COMMIT, $\mathcal{OTC}$ stores the latest commit request r_4.

We now describe a detailed algorithm for generating the incoming edges of r' for $\mathcal{G}_{\text{OT}}$. For each request r', SSFS iterates over every object $o \in r'.objs$. Given an object o, when r' is a non-commit request, we retrieve the latest commit request accessing o using (o, COMMIT) in O(1). If r' is a commit request, we first retrieve the latest non-commit request set using $(o, \text{NON-COMMIT})$. If such a non-commit request does not exist, then we retrieve the latest commit request accessing o using (o, COMMIT) in O(1). All retrieved requests are stored in a temporary table, and we group the candidate source vertices by vertex ID. Then we generate every edge (r, r') such that r's group size is the same as $|r.objs \cap r'.objs|$.

Now, we analyze the time and space complexity of edge generation and state maintenance for $\mathcal{G}_{\text{OT}}$. For each request r', we need to scan the candidate source vertices from $\mathcal{OTC}$, and store them in the temporary table. If r' is a non-commit request, it takes $O(|r'.objs|)$. Otherwise, it takes $O(\sum_{o \in r'.objs}$ # of non-commit requests in $\mathcal{OTC}$ for $o)$. If the temporary table is implemented as a hash table, the group by operation takes $\tilde{O}(max(|r'.objs|, \sum_{o \in r'.objs}$ # of non-commit requests in $\mathcal{OTC}$ for $o))$. The time complexity of maintaining $\mathcal{OTC}$ for each request is $O(|r'.objs|)$. The space complexity of $\mathcal{OTC}$ and the temporary table is $O(\sum_{r \in \mathcal{V}_{\mathcal{R}}}(|r.objs|))$.

5.3 Generating $\mathcal{G}_{\text{IT[OT]}}$

A naive implementation to generate $\mathcal{E}_{\text{IT[OT]}}$ is to 1) generate the incoming edges of r' in $\mathcal{E}_{\text{OT}}$ and 2) prune some of the edges not in $\mathcal{E}_{\text{IT[OT]}}$. The efficiency of this approach is severely limited, even inferior to the generation of $\mathcal{E}_{\text{OT}}$.

Instead of generating all incoming edges of r' in $\mathcal{E}_{\text{OT}}$, we discard some candidate requests from $\mathcal{OTC}$ to avoid generating the edges not in $\mathcal{E}_{\text{IT}(0)}$. Specifically, for each $(obj, type)$ pair, we only maintain the latest request for each session. As a result, the number of stored requests is significantly smaller than the original $\mathcal{OTC}$. Note that $\mathcal{SCR}$ is not needed for generating $\mathcal{G}_{\text{IT[OT]}}$.

We also iterate over sessions first and count the number of the *latest* candidate source vertex r for a given session. This way, we can check if the group size corresponding to r equals $|(r'.objs \cap r.objs)|$, which avoids the hash-based grouping. This optimization enables a significant speedup, compared to generating the edges for $\mathcal{E}_{\text{OT}}$ only. Thus, generating $\mathcal{E}_{\text{IT[OT]}}$ achieves both size and efficiency. Extensive experiments in Section 7 confirm our claim. After generating the incoming edges of r' for $\mathcal{E}_{\text{OT}} \cap \mathcal{E}_{\text{IT}(0)}$, we prune the edges using $\mathcal{LAE}$ (see case $k > 0$ in Section 5.1).

6 PARALLELIZING SSFS

In this section, we propose the parallel version of SSFS (PSSFS). We focus on the IT[OT]-graph since the other types of graphs can be obtained similarly.

It consists of three phases: 1) partitioning, 2) local dependency graph generation, and 3) hierarchical merging. In the partitioning phase, PSSFS divides the workload into p partitions by time range, $P_1, P_2, \cdots, P_p$, and generates their induced subgraphs, $\mathcal{G}_{\text{ini}}[P_1], \mathcal{G}_{\text{ini}}[P_2], \cdots, \mathcal{G}_{\text{ini}}[P_p]$. Here, a subgraph H is an induced subgraph of $\mathcal{G}$ if every edge in $E(\mathcal{G})$ whose endpoints are both in $V(H)$ [14]. Given a set of vertices $V(H)$, $\mathcal{G}[V(H)]$ denotes H. Then, PSSFS generates a local IT[OT]-free dependency graph $\mathcal{G}_{i(0)}$ for $\mathcal{G}_{\text{ini}}[P_i]$ by using the serial SSFS. Here, $\mathcal{G}_{i(m)}$ represents the i-th local dependency graph at the m-th level in the hierarchical merge. Note that we will explain what states need to be maintained additionally for efficient hierarchical merging. Finally, we hierarchically merge the local dependency graphs by generating missing inter-partition edges while maintaining the states. Except for the last level, the merged dependency graph is also regarded as a local dependency graph. For example, given local dependency graphs $\mathcal{G}_{i(m)}$ and $\mathcal{G}_{i+1(m)}$, the merged dependency graph of these two is also another local dependency graph for $V(\mathcal{G}_{i(m)}) \cup V(\mathcal{G}_{i+1(m)})$ for all $i < \lceil \log_2 p \rceil$. Please refer to [7] for the detailed algorithm.

While this may appear to be a typical parallel algorithm, there are three challenges. 1) What were the missing edges in local dependency graphs compared to the global dependency graph? 2) Does any edge in local dependency graphs need to be removed? 3) What states should be maintained additionally for efficient merging? Lemma 2 answers the first two questions. Lemma 2 states that 1) missing edges are inter-partition ones only, and 2) there is no risk of removing any edge from the local dependency graphs.

Lemma 2. *Given the global dependency graph $\mathcal{G}_{\text{IT[OT]}}$, every local dependency graph $\mathcal{G}_i$ is the induced subgraph of the global dependency graph using all vertices in $\mathcal{G}_i$. That is,*

$$\mathcal{G}_i = \mathcal{G}_{\text{IT[OT]}}[V(\mathcal{G}_i)].$$

Now, we answer the last question. As in SSFS, we first explain what states must be maintained to generate candidate inter-partition edges for the merged OT-free graph efficiently. We then explain how to prune redundant edges from them using the definition of the k-forward path. Thus, the resulting graph satisfies all conditions of the IT[OT]-free graph.

Consider the merge of two local dependency graphs $\mathcal{G}_i$ and $\mathcal{G}_{i+1}$. Then, for any inter-partition edge (r, r') between the two graphs, $r \in V(\mathcal{G}_i)$ and $r' \in V(\mathcal{G}_{i+1})$. Lemma 3 states a set of candidate destination vertices $\{r'\}$ for the inter-partition edges, which we need to maintain.

Lemma 3. *Consider merging two local dependency graphs $\mathcal{G}_i$ and $\mathcal{G}_{i+1}$. Then, for some object o, every vertex r' in the set of destination vertices in all inter-partition edges in the merged dependency graph is either the first commit request accessing o or a non-commit request accessing o before the first commit request accessing o in $\mathcal{G}_{i+1}$.*

Based on Lemma 3, we maintain a new state as a hash table where its key is an object $o \in O$, and its value is the pair of the first commit request and the first non-commit requests before the first commit request for o.

Here, we need not store all non-commit requests before the first commit request for each session for o, which is analogous to the optimization explained in Section 5.3. Then, for every $o \in O$, the set of candidate destination vertices are union of both sets in the value of the hash table for key o.

Now, we explain how to generate inter-partition edges of the merged OT-free graph with this additional state. Consider merging two local dependency graphs, $\mathcal{G}_i$ and $\mathcal{G}_{i+1}$. Then, we can obtain all candidate destination vertices for $\mathcal{G}_{i+1}$. Since we will generate the incoming edges of $\mathcal{G}_{\mathrm{IT[OT]}}$ as in SSFS, we sort these candidate destination vertices by timestamp order. Using the first commit request and the non-commit requests before the commit request for every $o \in O$ for $\mathcal{G}_i$, PSSFS generates the edges in $\mathcal{G}_{\mathrm{IT[OT]}}$ as in SSFS, discarding any edge whose both vertices are in $V(\mathcal{G}_{i+1})$.

To generate the inter-partition edges for the merged IT[OT]-free graph, we need to prune out any inter-partition edge (r, r') in $\mathcal{E}_{\mathrm{OT}}$ if there is a k-forward path from r to r'. As explained in Section 5.1, r must be the latest collision-dependent to r'. Therefore, the source vertex of the incoming edge for r' in the session s must be the latest candidate source vertex $r \in V(\mathcal{G}_i)$ in a different session. To prune out (r, r') such that a k-forward path from r to r' exists and all the edges in the path are in $OT(E(\mathcal{G}_{\mathrm{col}}[V(\mathcal{G}_i) \cup V(\mathcal{G}_{i+1})]))$ using the states, we provide the following lemma. It is straightforward to prove by the definition of the IT-free graph.

Lemma 4. *An edge $(r, r') \in OT(E(\mathcal{G}_{\mathrm{col}}[V(\mathcal{G}_i) \cup V(\mathcal{G}_{i+1})]))$ is pruned if and only if there exists another edge $(r^s, r^{s'}) \in OT(E(\mathcal{G}_{\mathrm{col}}\ [V(\mathcal{G}_i) \cup V(\mathcal{G}_{i+1})]))$ such that $r^s.sid = r.sid \wedge r^s.ts \geq r.ts$ and $r^{s'}.sid = r'.sid \wedge r^{s'}.ts \leq r'.ts$.*

According to Lemma 4, we need to find $(r^s, r^{s'})$ to check if the inter-partition edge (r, r') needs to be pruned. If any of the following cases is satisfied, it is pruned: 1) $r^s, r^{s'} \in V(\mathcal{G}_i)$ 2) $r^s \in V(\mathcal{G}_i)$ and $r^{s'} \in V(\mathcal{G}_{i+1})$, 3) $r, r' \in V(\mathcal{G}_{i+1})$. When the first case holds, a k-forward path $(r, r_1^s, ..., r_i^s, r_1^{s'}, ..., r_k^{s'}, r')(i \geq 0, k \geq 1)$ such that $r_i^s = r^s$ and $r_1^{s'} = r^{s'}$ exists. That is, such (r, r') must not be in $IT(OT(\mathcal{E}_{\mathrm{col}}))$. The other cases also hold according to the definition of the k-forward path.

The first case is checked by using the $\mathcal{LAE}$ of $\mathcal{G}_i$, since $(r^s, r^{s'})$ has been generated for $\mathcal{G}_i$. The second case is also checked by $\mathcal{LAE}$ of $\mathcal{G}_i$, since the candidate destination vertices are processed in increasing timestamp order. For the third case, we need to maintain an additional state, the first appended edges between sessions ($\mathcal{FAE}$). This case is checked by $\mathcal{FAE}$ for $\mathcal{G}_{i+1}$.

After generating inter-partition edges between $\mathcal{G}_i$ and $\mathcal{G}_{i+1}$, PSSFS merges all states explained. We omit how to merge in detail, since it is straightforward by definition.

Now we explain how to parallelize the transitive reduction. We can use any parallel transitive reduction algorithm which supports local transitive reduction and hierarchical merge. For computing transitive reduction ${\mathcal{G}_{\mathrm{IT[OT]}}}^{tr}$ of $\mathcal{G}_{\mathrm{IT[OT]}}$, we need to calculate the transitive closure ${\mathcal{G}_{\mathrm{IT[OT]}}}^{tc}$. Here, an edge $(r, r') \in E({\mathcal{G}_{\mathrm{IT[OT]}}}^{tc})$ if and only if there is a path from r to r' in $\mathcal{G}_{\mathrm{IT[OT]}}$.

For each vertex v, we need to maintain a set of vertices reachable to v as a state. During the merge phase, the transitive closure for the merged dependency graph needs to be updated since inter-partition edges are generated additionally. Instead of updating the state for all vertices in $\mathcal{V}_{\mathcal{R}}$, we need to update the state for 1) the requests in $\mathcal{SCR}$, 2) the first commit request for each obj, 3) non-commit requests accessing obj before the first commit request accessing obj for every object obj, and 4) the first and the last requests for each session. Updating the state for those vertices only is sufficient, since the additional edges are generated between those vertices during the merge phase. Experiments in Section 7 show that the overhead of our efficient merge strategy is negligible.

7 EXPERIMENTS

The goals of our experiments are as follows. SSFS($\mathcal{G}$) denotes SSFS with the dependency graph $\mathcal{G}$.

- SSFS consistently and significantly outperforms RBSS for varying workloads, including a real one (Sections 7.2 - 7.3).
- SSFS($\mathcal{G}_{\text{IT[OT]}}$) outperforms SSFS with the other dependency graphs in both in efficiency and in size (Sections 7.2 - 7.4).
- PSSFS achieves almost linear speedup (Section 7.5).

7.1 Experimental Setup

Workloads. We use two OLTP benchmarks, as our major customer workloads are OLTP ones: TPC-C [1] and SD benchmark [4]. These benchmarks are representative ones, simulating our customer workloads. The TPC-C benchmark is a standard OLTP benchmark that models a wholesale company with warehouses. We set the number of warehouses to 100. The database consists of nine tables. In Section 7.2, we partition the tables by warehouse ID to show the case when each table partition is an object. The SD benchmark simulates sales and distribution scenarios with six transactions. Each transaction involves several dialog steps [24]. We use a 3-tier architecture version of the SD benchmark. We vary the number of application server instances to vary the number of sessions. We use 5000 users and one sec of think time. Experiment results for SD benchmark show similar trends to those for TPC-C. Due to the space limitation, we omit them and encourage readers to refer to our technical paper for further information [7]. We also use the large-scale, real-world customer workload, which was captured from a real-world cloud-based business application system that is primarily for processing procurement operations of an enterprise. It was captured for 18 minutes, with 8.9 million requests from 7,393 sessions. The 2,812 tables are accessed, and 33 % of the transactions contain updates. Write transactions contain 9.3 non-commit requests on average, where INSERT, SELECT, UPDATE, DELETE, and MERGE_INTO statements account for 60%, 29%, 8.5%, 1.5%, 1%, respectively. The write transactions access 4.2 tables and update 2.7 tables on average. Read-only transactions contain 1.4 non-commit requests, reading 2.2 tables on average.

Running Environments. We conducted experiments on a Linux machine with four Intel(R) Xeon(R) CPU E7-8880 v4 CPUs, 1TB RAM, and one 745GB SSD. We used a single thread for all experiments except for Section 7.5.

Measure. We measured the elapsed time with a breakdown into the edge generation and transitive reduction times. We use the number of edges to report the dependency graph size. We also provide the size of the minimal dependency graph generated by transitive reduction labeled as TR. The number of captured transactions may vary with a given capture duration. Therefore, we normalized the measures by the number of captured transactions. For SSFS, we additionally measured the memory consumption for the states. Table 3 summarizes the experimental parameters with their ranges and default values marked in boldface. Here, we used 30 minutes as the default capture duration according to [28]. The timeout, labeled as TO, is set to 10 hours. To analyze the tradeoff in

logging granularity, we report the end-to-end time breakdowns, including capture time, dependency graph generation time, replay time, and pre/post-processing time for replay (i.e., importing statements and exporting results), for different numbers of table partitions.

Table 3. Experimental parameters with their ranges and default values in TPC-C.

Parameter	Range
capture duration	15, **30**, 45, 60
# of clients	32, **64**, 96, 128, 256[a]
# of thread	**1**, 4, 8, 12, 16, 20
# of table partitions	**1**, 4, 16, 64

[a]for end-to-end time measurement with high replay concurrency

Competitor We use enhanced versions of RBSS, RBSS$^+$ and RBSS^L, outperforming RBSS. RBSS$^+$ generates the IT-free graph instead of the $\mathcal{G}_{\text{RBSS}}$. To find r_{min} (in Section 4), RBSS$^+$ scans backward until it finds a request with an incoming edge. Unlike SSFS, RBSS$^+$ becomes significantly slower due to short sessions performing DB restore tasks such as table initialization in the captured workload; The cost of finding the incoming edge in such a session is significant for RBSS$^+$. For a fair comparison, we have allowed all algorithms to ignore these sessions. RBSS^L is a optimized version of RBSS$^+$ by memoizing the $\mathcal{LAE}$ table for RBSS, although this memoization is part of our technique. RBSS^L finds r_{min} in $O(1)$ using $\mathcal{LAE}$.

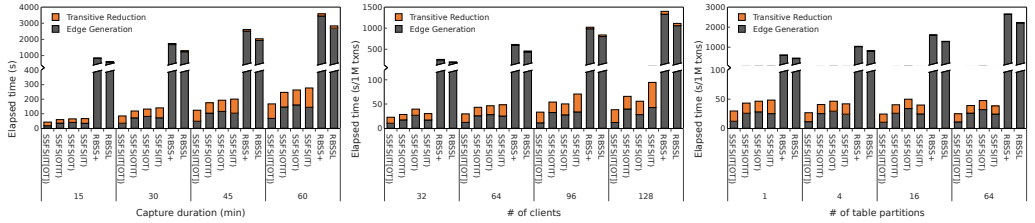
We parallelize RBSS$^+$ and RBSS^L (PRBSS$^+$ and PRBSS^L). Given a pair of sessions (s, s'), the edges from s to s' are sequentially generated by RBSS. The edges for different pairs of sessions are generated in parallel. For transitive reduction, we use horizontal partitioning as in PSSFS.

7.2 Experimental Results for TPC-C

Varying capture duration Figures 8a and 9a show the elapsed times and the edge set sizes for varying capture durations in TPC-C, respectively. Figure 9a empirically confirms the containment relationships in the Venn diagram in Section 3. The elapsed time increases linearly with the capture duration for all algorithms. In terms of elapsed time, SSFS($\mathcal{G}_{\text{IT[OT]}}$) outperforms SSFS($\mathcal{G}_{\text{IT}}$), SSFS($\mathcal{G}_{\text{OT}}$), SSFS($\mathcal{G}_{\text{OTIT}}$), RBSS$^+$ and RBSS^L by up to 1.6, 1.6, 1.4, 20.9 and 15.9. This is because the size of SSFS($\mathcal{G}_{\text{IT[OT]}}$) is only 5% larger than the minimal dependency graph and achieves both graph compactness and algorithm efficiency. In terms of edge generation time, SSFS($\mathcal{G}_{\text{OT}}$) is the slowest among SSFSs due to the overhead of 1) inserting candidate source vertices into a temporary table and 2) performing expensive group-by operations as analyzed in Section 5.3. The transitive reduction time is almost proportional to the edge set size.

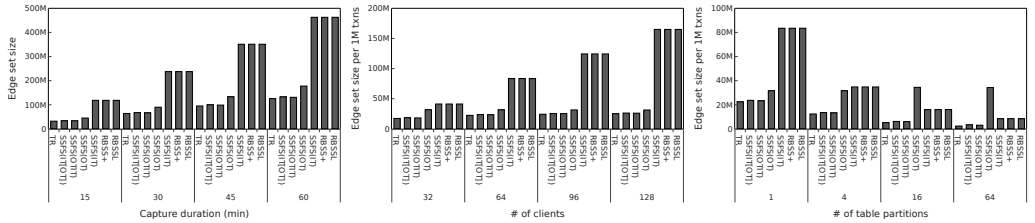
Varying # of clients Figures 8b and 9b show the elapsed times and the edge set sizes for varying the number of clients in TPC-C, respectively. As the number of clients increases, the number of sessions also increases. The elapsed times for RBSS$^+$ and RBSS^L super-linearly increase as the number of clients increases. The elapsed times of SSFS($\mathcal{G}_{\text{IT[OT]}}$) and SSFS($\mathcal{G}_{\text{OT}}$) slowly increase as the number of clients increases, since $|\mathcal{E}_{\text{OT}}|$ remains almost constant regardless of the number of clients as shown in Figure 9b. The elapsed times of SSFS($\mathcal{G}_{\text{OTIT}}$) and SSFS($\mathcal{G}_{\text{IT}}$) almost linearly increase, since the number of the incoming edges of each request in these algorithms also linearly increases. Their dependency graphs' sizes linearly increase as shown in Figure 9b.

Varying # of table partitions Figures 8c and 9c show the elapsed times and the edge set sizes by varying the number of table partitions, respectively. The elapsed times for RBSS$^+$ and RBSS^L increase as the number of table partitions increases, since d_{avg} in Section 4 increases due to the size reductions in $|\mathcal{E}_{\text{col}}|$ for varying the number of partitions. The elapsed times of SSFS($\mathcal{G}_{\text{IT[OT]}}$)



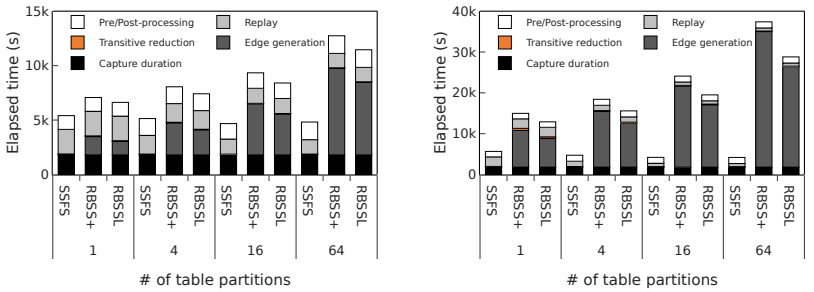
(a) Varying duration. (b) Varying # of clients. (c) Varying # of table partitions.

Fig. 8. Dependency graph generation time in TPC-C.



(a) Varying duration. (b) Varying # of clients. (c) Varying # of table partitions.

Fig. 9. Edge set size in TPC-C.



(a) 64 clients. (b) 256 clients.

Fig. 10. End-to-end time breakdown in TPC-C.

remain almost constant, since $|\mathcal{E}_{\text{OT}}|$ also remains almost constant. In contrast, the elapsed time of SSFS($\mathcal{G}_{\text{IT}}$) decreases, since $|\mathcal{E}_{\text{IT}}|$ also decreases. As the number of table partitions increases, $|\mathcal{E}_{\text{IT}}|$ decreases, since the size of $\mathcal{E}_{\text{col}}$ decreases. This explains why the elapsed time of SSFS($\mathcal{G}_{\text{IT}}$) slightly decreases as we increase the number of partitions. However, $|\mathcal{E}_{\text{OT}}|$ remains almost constant, since the probability of an edge being pruned by an object transitive path decreases when the number of objects increases. SSFS($\mathcal{G}_{\text{IT[OT]}}$) is the fastest, outperforming SSFS with other dependency graphs. This is because its edge set size is almost close to TR, and the cost to generate $\mathcal{G}_{\text{IT[OT]}}$ is the cheapest, as explained in Section 5.

End-to-end time breakdown Figure 10 shows the end-to-end time breakdowns of SSFS($\mathcal{G}_{\text{IT[OT]}}$), RBSS^{+}, and RBSS$^{\text{L}}$ for varying the number of table partitions in TPC-C. The replay time is affected by the number of table partitions and the number of clients. In Figure 10a, the replay time with no

partitioning is 25% longer than the capture time. This is because the replay can slow down due to the artificial dependencies in table-level logging. However, as the number of table partitions increases to 64, the replay time is 26% faster than the capture time, as the requests accessing different partitions can be replayed in parallel. In Figure 10b, the replay time is 58% faster than the capture time as replay concurrency increases with 256 clients. This phenomenon is explained as follows. During capture time, as the number of clients increases from 64 to 256, the capture throughput increases only 5% as write conflicts also increase. However, as the write conflicts do not occur during the replay as explained in Section 3.6, the replay throughput increases 67% as the number of clients increases from 64 to 256. The dependency graph generation times of RBSS^{+} and RBSS^{L} take up to 89% and 86% of the total end-to-end times, which are the bottlenecks.

The size of $\mathcal{E}_{\text{col}}$ Table 4 shows $|\mathcal{V}_{\mathcal{R}}|$, $|\mathcal{E}_{\text{col}}|$, and $|\mathcal{E}^{tr}|$ for varying the capture duration and the number of table partitions. We omit the results for varying the number of clients, since there is no change in trend. In all cases, the number of redundant edges in $\mathcal{E}_{\text{col}}$ (i.e., $|\mathcal{E}_{\text{col}} - \mathcal{E}^{tr}|$) is much larger than $|\mathcal{E}^{tr}|$ by at least five orders of magnitude. Thus, more than 99% of the redundant edges are pruned due to IT and OT, since $\mathcal{E}_{\text{OTIT}}$ has only 8% more edges than $\mathcal{E}^{tr}$ on average in Figure 9. This indicates the two types of redundancy are significantly prevalent. The other workloads also show similar trends.

Table 4. The size of $\mathcal{E}_{\text{col}}$.

(a) Varying capture duration.

min	$\lvert\mathcal{V}_{\mathcal{R}}\rvert$	$\lvert\mathcal{E}_{\text{col}}\rvert$	$\lvert\mathcal{E}^{tr}\rvert$
15	3.0×10^7	1.6×10^{13}	3.2×10^7
30	5.9×10^7	6.6×10^{13}	6.4×10^7
45	8.7×10^7	1.4×10^{14}	9.5×10^7
60	1.2×10^8	2.5×10^{14}	1.3×10^8

(b) Varying # of table partitions.

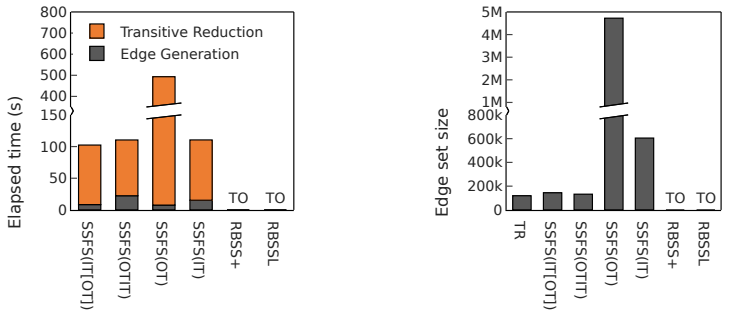
#	$\lvert\mathcal{V}_{\mathcal{R}}\rvert$	$\lvert\mathcal{E}_{\text{col}}\rvert$	$\lvert\mathcal{E}^{tr}\rvert$
1	5.9×10^7	6.6×10^{13}	6.4×10^7
4	5.9×10^7	1.6×10^{13}	3.6×10^7
16	6.1×10^7	4.2×10^{12}	1.5×10^7
64	6.2×10^7	1.3×10^{12}	7.1×10^6

7.3 Experimental Results for Real-World Customer Workload

Figure 11 shows the elapsed times and the edge set sizes in the large-scale, real-world customer workload. In Figure 11a, the trend of the edge generation time is similar to that of TPC-C with 128 clients. Note that the number of clients is a dominant factor in the elapsed time rather than the number of table partitions in TPC-C. The edge generation time of SSFS($\mathcal{G}_{\text{OT}}$) and SSFS($\mathcal{G}_{\text{OTIT}}$) is 2-3x longer than the others, as in the TPC-C with 128 clients. RBSS^{+} and RBSS^{L} reach timeout. The transitive reduction time is proportional to the edge set size. In Figure 11b, the trend of the edge set size is similar to that of TPC-C with 64 table partitions. $\mathcal{G}_{\text{OT}}$ is the largest, and thus transitive reduction for SSFS($\mathcal{G}_{\text{OT}}$) takes more than three times longer compared to that with the other graphs. Although $|\mathcal{E}_{\text{IT}}|$ is 7.8 times smaller than $|\mathcal{E}_{\text{OT}}|$, the edge generation time of SSFS($\mathcal{G}_{\text{IT}}$) is more than twice as slow as that of SSFS($\mathcal{G}_{\text{OT}}$). This is because SSFS($\mathcal{G}_{\text{IT}}$) first scans the candidates of the IT(0)-free graph from $\mathcal{SCR}$ and then prunes the edges with k-forward paths using $\mathcal{LAE}$. The number of scanned candidates from $\mathcal{SCR}$ in SSFS($\mathcal{G}_{\text{IT}}$) is four times larger than that from $\mathcal{OTC}$ in SSFS($\mathcal{G}_{\text{OT}}$). SSFS($\mathcal{G}_{\text{IT[OT]}}$) remains the best in efficiency and compactness.

7.4 Memory Usage

Table 5 shows the memory usages of the states in SSFS with the default configuration of TPC-C, TPC-C with 64 table partitions (denoted as TPC-C^{64}), and the large-scale, real-world customer workload, depending on the type of dependency graph. In TPC-C with default configuration, except for SSFS($\mathcal{G}_{\text{OT}}$), the states occupy less than 100 kB of memory. However, SSFS($\mathcal{G}_{\text{OT}}$) exhibits



(a) Elapsed time (s). (b) Edge set size.

Fig. 11. Experimental results in the large-scale, real-world customer workload.

significantly higher memory consumption. This is due to the unbounded growth of $\mathcal{OTC}$ in SSFS($\mathcal{G}_{\text{OT}}$) when there is an unmodified table (e.g., ITEM table in TPC-C). Unlike in SSFS($\mathcal{G}_{\text{OT}}$), $\mathcal{OTC}$s in SSFS($\mathcal{G}_{\text{IT[OT]}}$) and SSFS($\mathcal{G}_{\text{OTIT}}$) store at most one vertex for each session, and thus the memory usages for their $\mathcal{OTC}$ are bounded to $O(|S| \cdot |O|)$. In TPC-C^{64}, the memory usage for states increases by up to 32 times, as $|O|$ increases. The maximum space required for states is 269 MBytes only. In the large-scale, real-world customer workload with 7,393 sessions, the memory usages for the states increase by up to 1 GB. Note that the space complexity of $\mathcal{SCR}$ and $\mathcal{LAE}$ is $O(|S| \cdot |O|)$ and $O(|S|^2)$. Therefore, SSFS($\mathcal{G}_{\text{IT[OT]}}$) consume less memory than SSFS($\mathcal{G}_{\text{IT}}$) and SSFS($\mathcal{G}_{\text{OTIT}}$), since it does not maintain $\mathcal{SCR}$. For both workloads, SSFS($\mathcal{G}_{\text{IT[OT]}}$) requires a manageable size of states within modern server systems.

Table 5. The Memory Usages for the States in SSFS.

	SSFS($\mathcal{G}_{\text{IT[OT]}}$)	SSFS($\mathcal{G}_{\text{OTIT}}$)	SSFS($\mathcal{G}_{\text{OT}}$)	SSFS($\mathcal{G}_{\text{IT}}$)
TPC-C	50 kB	67 kB	134 MB	57 kB
TPC-C^{64}	785 kB	2.16 MB	269 MB	1.41 MB
Real.	630 MB	1.03 GB	299 MB	830 MB

7.5 Scalability of PSSFS

Figure 12 shows the speedups of the PSSFS($\mathcal{G}_{\text{IT[OT]}}$), PRBSS^{+} and PRBSS$^{\text{L}}$ compared to serial SSFS($\mathcal{G}_{\text{IT[OT]}}$) by varying the number of threads in TPC-C. The speedup is defined as the ratio of the elapsed time of the serial SSFS($\mathcal{G}_{\text{IT[OT]}}$) over that of a parallel algorithm. PSSFS achieves almost linear speedup with minimal overhead on the merge phase as the number of threads increases. Although PRBSS^{+} and PRBSS$^{\text{L}}$ achieve almost linear speedup, PSSFS($\mathcal{G}_{\text{IT[OT]}}$) outperforms them 17.2 and 12.4 times on average. We omit the results of the other workloads due to the similar trends.

8 RELATED WORK

The concept of database replay was first proposed in [20]. It provides a synchronization mechanism that is more restrictive than necessary, resulting in a low level of concurrency and poor performance. Morfonios et al. [28] enhanced this schema by proposing RBSS. We prove that RBSS generates the dependency graph between $\mathcal{G}_{\text{IT}(0)}$ and $\mathcal{G}_{\text{IT}(1)}$. Snowtrail [38] focuses on seamlessly executing production queries using different cloud instances without interfering with the customer's production

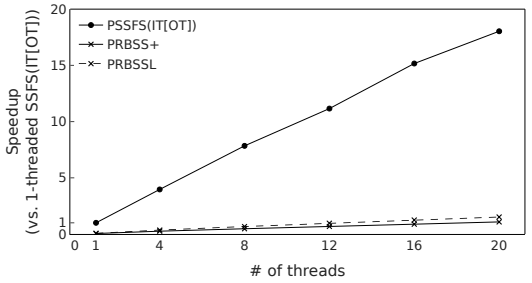


Fig. 12. Speedup rate by varying the number of threads.

workload in a cloud environment. The replay mechanism of Snowtrail is similar to Oracle Database Replay [20]. These techniques generate many redundant edges, which are subsequently pruned by the expensive transitive reduction.

Flex [11] is a prototype system for testing and tuning a production database instance. Flex mainly loads a specific snapshot s and executes an action on s, such as a user-defined executable, not supporting fine-grained workload replay. Doppler [12] is a recommendation engine for migrating on-premise data platforms to Platform-as-a-Service (PaaS) offerings. However, Doppler does not utilize workload information but instead utilizes rudimentary information for the recommendation: performance counters, all possible cloud target Stock Keeping Units (SKUs), and the real-time pricing for each SKU. It then relies on machine learning models to recommend the right-sized SKUs. Instead, DRSs focus on the fine-grained workload information to safely migrate to new database versions or hardware. Although these efforts are orthogonal to our approach, we believe our work complements them.

Diametrics [13] is a framework for end-to-end benchmarking and performance monitoring of query engines. It focuses on repeatable benchmarking for various query engines rather than diagnosing any performance issues faced by any software/hardware upgrade. TROD [26] is a framework for debugging web applications. It captures and sequentially re-runs the transactions to reproduce bugs.

Recording of runs of a general program and replaying with the log to debug non-deterministic failures has been extensively studied [10, 29, 30]. [10] utilizes consistency relaxation to debug data races in a multi-core environment. [29] proposes an approximate replay system that divides a sequence of events generated by a program into transactions and then logs conflicting operations at the transaction level. However, this scheme does not guarantee the correct replay, which is crucial in our target applications. [30] utilizes hardware-assisted virtualization extensions to implement a software-based deterministic replay system. However, all these approaches are for general programs incurring significant overhead in recording all conflicting operations on shared memory. However, a DRS is a tailored replay system designed for database workloads.

There are recovery systems utilizing dependency graphs to parallelize recovery [27]. Adaptive Logging [40] requires capturing all tuple IDs in read and write sets of each transaction, whose numbers can be large with scan operations. This can lead to significant overhead for edge generation, compared to ours. It is also challenging to maintain all read/write sets in our production system. Even if Adaptive Logging uses a coarser granularity, replaying the resulting dependency graph does not guarantee the output determinism in snapshot isolation, the most important desideratum in DRS. In [40], each node represents a transaction (i.e., a stored procedure call). Consider two transactions, T_1 and T_2, concurrently accessing a common table containing tuples r_1 and r_2 during

the capture phase. Suppose that T_1 reads r_1, and T_2 reads r_2. T_1 writes r_2, and T_2 writes r_1 using the values they read, respectively. Furthermore, T_1 commits before T_2 commits (i.e., write skew). Then, in the resulting dependency graph, there is an edge from T_1 to T_2. Thus, Adaptive Logging replays T_2 after T_1. However, this leads to a different database state, compared to the database state captured during the capture. This will be reported as a bug in our DRS. Note that this is not a problem since transactions are executed serially in H-store, the underlying DBMS of Adaptive Logging. Pacman [37] builds a dependency graph using the program analysis technique. However, Pacman assumes that all transactions are implemented as stored procedures and are known in advance, severely restricting the user applications and rendering it unusable for general DRSs.

There are four additional types of dependency graphs, where all of these dependency graph generation algorithms cannot be readily applied to our DRS: (G1) the dependency graph managed by a lock manager for deadlock detection [19, 36], (G2) the one for detecting possible anomalies in a given transaction workload under an applied (weak) isolation level [21, 25], (G3) the one for scheduling the transactions to reduce runtime conflicts [15–17, 35], and (G4) the one for making replicas consistent with the primary database [23]. The dependency graph is generated on the fly at the replica by comparing each transaction with all existing transactions in the graph. Here, a node corresponds to a transaction rather than a request. Even if the dependency graph is much coarser than ours, the system limits the insertion of vertices when there are many vertices. Note that G1 and G2 focus on finding cycles existing in the maintained dependency graphs. G3 focuses on reducing inter-transaction conflicts by reordering pending requests. G4 focuses on ordering among the replicated write transactions only, while our DRS should consider the ordering of both write and read requests.

Graph sparsification is the problem reducing the edge set size while approximately preserving the graph's properties, such as the cut sizes or the distances between vertices [8, 18, 31, 33]. Although some graph sparsifiers such as [33] reduce the graph in nearly-linear time, they do not guarantee to preserve all necessary dependent edges. However, as stated in our problem definition, all edges in $\mathcal{G}^{tr}_{col}$ must be preserved for DRSs to achieve output determinism. Thus, these methods cannot be readily applied to our problem.

9 CONCLUSION

In this paper, we presented a comprehensive and practical solution for fast dependency graph generation in a database replay system (DRS). In Section 3, we formally proposed a taxonomy of four types of dependency graphs for a DRS. In Section 4, we showed that the worst-case time complexity of the state-of-the-art technique is $O(|\mathcal{V}_{\mathcal{R}}|^2)$ due to repetitive backward scans for each request. We showed that its dependency graph is the one between IT(0)-free graph and IT(1)-free graph, potentially having many redundant edges. In order to solve this challenging problem, in Section 5, we proposed a novel dependency generation algorithm SSFS to scan requests once by succinctly maintaining the information required to generate the edges. We implemented our solution in a leading commercial DBMS. Experiments using our DRS showed that it dramatically improves the dependency graph generation time by up to two orders of magnitude, compared to the state-of-the-art.

ACKNOWLEDGMENTS

The authors would like to thank Jaehyun Lim for helping the experimental setting of the TPC-C benchmark. This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2021R1A2B5B03001551).

REFERENCES

[1] 2010. TPC Benchmark C. http://www.tpc.org/tpcc/. Accessed: 2022-06-23.

[2] 2020. *Oracle Database 19c: Real Application Testing Overview.* Technical Report.

[3] 2022. Capturing and Replaying Workloads. https://help.sap.com/docs/SAP_HANA_COCKPIT/afa922439b204e9caf22c78b6b69e4f2/4f3c88249d484b0faba0e6b27b82c2dd.html?locale=en-US

[4] 2022. SAP Standard Application Benchmarks. https://www.sap.com/about/benchmark.html.

[5] 2022. Sql server distributed replay. https://learn.microsoft.com/en-us/sql/tools/distributed-replay/sql-server-distributed-replay?view=sql-server-ver16

[6] 2023. The Internals of PostgreSQL. https://www.interdb.jp/. Accessed: 2023-10-20.

[7] 2023. Technical Report. https://sites.google.com/view/systemx-replay

[8] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. 2012. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*. ACM, Scottsdale Arizona USA, 5–14. https://doi.org/10.1145/2213556.2213560

[9] Alfred V. Aho, Michael R Garey, and Jeffrey D. Ullman. 1972. The transitive reduction of a directed graph. *SIAM J. Comput.* 1, 2 (1972), 131–137.

[10] Gautam Altekar and Ion Stoica. 2009. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 193–206.

[11] Nedyalko Borisov and Shivnath Babu. 2013. Rapid experimentation for testing and tuning a production database deployment. *Proceedings of the 16th International Conference on Extending Database Technology*, 125–136.

[12] Joyce Cahoon, Wenjing Wang, Yiwen Zhu, Katherine Lin, Sean Liu, Raymond Truong, Neetu Singh, Chengcheng Wan, Alexandra Ciortea, Sreraman Narasimhan, and Subru Krishnan. 2022. Doppler: automated SKU recommendation in migrating SQL workloads to the cloud. *Proceedings of the VLDB Endowment* 15, 12 (Aug. 2022), 3509–3521. https://doi.org/10.14778/3554821.3554840

[13] Shaleen Deep, Anja Gruenheid, Kruthi Nagaraj, Hiro Naito, Jeff Naughton, and Stratis Viglas. 2021. Diametrics: benchmarking query engines at scale. *ACM SIGMOD Record* 50 (2021), 24–31. Issue 1.

[14] Reinhard Diestel. 2005. Graph theory 3rd ed. *Graduate texts in mathematics* 173 (2005), 33.

[15] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving optimistic concurrency control through transaction batching and operation reordering. *Proceedings of the VLDB Endowment* 12, 2 (2018), 169–182.

[16] Jose M Faleiro and Daniel J Abadi. 2014. Rethinking serializable multiversion concurrency control. *arXiv preprint arXiv:1412.2324* (2014).

[17] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment* 10, 5 (2017).

[18] Wai Shing Fung, Ramesh Hariharan, Nicholas J.A. Harvey, and Debmalya Panigrahi. 2011. A general framework for graph sparsification. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*. ACM, San Jose California USA, 71–80. https://doi.org/10.1145/1993636.1993647

[19] Donald Fussell, Zvi M Kedem, and Abraham Silberschatz. 1981. Deadlock removal using partial rollback in database systems. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*. 65–73.

[20] Leonidas Galanis, Supiti Buranawatanachoke, Romain Colle, Benoît Dageville, Karl Dias, Jonathan Klein, Stratos Papadomanolakis, Leng Leng Tan, Venkateshwaran Venkataramani, Yujun Wang, et al. 2008. Oracle database replay. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 1159–1170.

[21] Yifan Gan, Xueyuan Ren, Drew Ripberger, Spyros Blanas, and Yang Wang. 2020. IsoDiff: debugging anomalies caused by weak isolation. *Proceedings of the VLDB Endowment* 13, 12 (2020).

[22] Herodotos Herodotou, Nedyalko Borisov, and Shivnath Babu. 2011. Query optimization techniques for partitioned tables. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, Athens Greece, 49–60. https://doi.org/10.1145/1989323.1989330

[23] Chuntao Hong, Dong Zhou, Mao Yang, Carbo Kuo, Lintao Zhang, and Lidong Zhou. 2013. KuaFu: Closing the parallelism gap in database replication. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 1186–1195.

[24] Bahman Javadi Isfahani. 2017. Evaluating a modern in-memory columnar data management system with a contemporary OLTP workload. (2017).

[25] Kyle Kingsbury and Peter Alvaro. 2020. Elle: inferring isolation anomalies from experimental observations. *Proceedings of the VLDB Endowment* 14, 3 (Nov. 2020), 268–280. https://doi.org/10.14778/3430915.3430918

[26] Qian Li, Peter Kraft, Michael Cafarella, Çağatay Demiralp, Goetz Graefe, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. 2023. Transactions Make Debugging Easy.. In *CIDR*.

[27] Arlino Magalhaes, Jose Maria Monteiro, and Angelo Brayner. 2022. Main Memory Database Recovery: A Survey. *Comput. Surveys* 54, 2 (March 2022), 1–36. https://doi.org/10.1145/3442197

[28] Konstantinos Morfonios, Romain Colle, Leonidas Galanis, Supiti Buranawatanachoke, Benoît Dageville, Karl Dias, and Yujun Wang. 2011. Consistent synchronization schemes for workload replay. *Proceedings of the VLDB Endowment* 4, 12 (2011), 1225–1236.

[29] Ernest Pobee, Xiupei Mei, and Wing Kwong Chan. 2019. Efficient transaction-based deterministic replay for multi-threaded programs. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. 760–771.

[30] Shiru Ren, Le Tan, Chunqi Li, Zhen Xiao, and Weijia Song. 2017. Leveraging hardware-assisted virtualization for deterministic replay on commodity multi-core processors. *IEEE Trans. Comput.* 67, 1 (2017), 45–58.

[31] Venu Satuluri, Srinivasan Parthasarathy, and Yiye Ruan. 2011. Local graph sparsification for scalable clustering. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, Athens Greece, 721–732. https://doi.org/10.1145/1989323.1989399

[32] Klaus Simon. 1988. An improved algorithm for transitive closure on acyclic digraphs. *Theoretical Computer Science* 58, 1-3 (1988), 325–346.

[33] Daniel A. Spielman and Shang-Hua Teng. 2004. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*. ACM, Chicago IL USA, 81–90. https://doi.org/10.1145/1007352.1007372

[34] Xian Tang, Junfeng Zhou, Yaxian Qiu, Xiang Liu, Yunyu Shi, and Jingwen Zhao. 2020. One Edge at a Time: A Novel Approach Towards Efficient Transitive Reduction Computation on DAGs. *IEEE Access* 8 (2020), 38010–38022.

[35] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*. 1–12.

[36] Gerhard Weikum and Gottfried Vossen. 2001. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier.

[37] Yingjun Wu, Wentian Guo, Chee-Yong Chan, and Kian-Lee Tan. 2017. Fast Failure Recovery for Main-Memory DBMSs on Multicores. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, Chicago Illinois USA, 267–281. https://doi.org/10.1145/3035918.3064011

[38] Jiaqi Yan, Qiuye Jin, Shrainik Jain, Stratis D Viglas, and Allison Lee. 2018. Snowtrail: Testing with production queries on a cloud database. *Proceedings of the Workshop on Testing Database Systems*, 1–6.

[39] C-Q Yang and Barton P Miller. 1988. Critical path analysis for the execution of parallel and distributed programs. In *The 8th International Conference on Distributed*. Computing Systems, 366–367.

[40] Chang Yao, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, and Sai Wu. 2016. Adaptive Logging: Optimizing Logging and Recovery Costs in Distributed In-memory Databases. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, San Francisco California USA, 1119–1134. https://doi.org/10.1145/2882903.2915208

Received July 2023; revised October 2023; accepted November 2023