DoppelGanger++ in Action: A Database Replay System with Fast Dependency Graph Generation

Wonseok Lee
Jaehyun Ha
Wook-Shin Han*
wslee@dblab.postech.ac.kr
jhha@dblab.postech.ac.kr
wshan@dblab.postech.ac.kr
POSTECH
Korea

Changgyoo Park
Myunggon Park
Juhyeng Han
changgyoo.park@sap.com
myunggon.park@sap.com
juhyeng.han@sap.com
SAP Labs Korea
Korea

ABSTRACT

A Database Replay System (DRS) captures workloads from a production system and subsequently replays them in a testing environment to verify correctness and performance. Prior to the replay process, DRS initially generates a dependency graph from the workload to ensure output determinism and to maximize replay concurrency in the testing system. However, the state-of-the-art inefficiently generates unnecessarily larger dependency graphs, creating a major bottleneck in the end-to-end pipeline. DoppelGanger++ is a new DRS supporting fast dependency graph generation. This demonstration illustrates how it captures and replays workloads, with a focus on efficiently generating compact dependency graphs. Specifically, we showcase the end-to-end database replay workflow using the complete database replay workload, accompanied by a web tool developed for our demo which can animate the dependency graph generation process and visualize important internal data structures.

PVLDB Reference Format:
Wonseok Lee, Jaehyun Ha, Wook-Shin Han, Changgyoo Park, Myunggon Park, and Juhyeng Han. DoppelGanger++ in Action: A Database Replay System with Fast Dependency Graph Generation. PVLDB, 17(12): 4313 - 4316, 2024.
doi:10.14778/3685800.3685863

1 INTRODUCTION

Database Replay Systems (DRSs) test relational database systems within a testing environment. DRSs capture database workloads on a production system and then replay them in a testing environment. Here, a workload consists of user requests, each containing a SQL statement with session ID. With DRSs, DBAs can avoid risks such as (a) performance regression, (b) bugs, or (c) new resource contention prior to applying the system changes to production [3, 5].

DRSs generate a dependency graph on the captured workload before replaying it to provide output determinism [2]. Here, output determinism means that the replay of a captured workload produces the same output as the original run, enabling DRSs to ensure correctness in the test system by comparing the results of each query during capturing and replaying. Nodes in the dependency graph correspond to requests in the captured workload, while edges represent the relative ordering constraints between two dependent requests. By concurrently replaying the captured requests while preserving the order constraints in the dependency graph, DRSs can maximize replay concurrency while detecting correctness bugs.

The state-of-the-art [5] employs a generate-and-prune approach for creating a dependency graph. With this approach, the generation step generates the dependency graph using an algorithm called RBSS. RBSS generates the incoming edges of each node by finding the latest dependent nodes in each of the other sessions through backward scans. Then, the pruning step prunes all *redundant* edges using expensive transitive reductions [1]. Here, a direct edge (v, u) is redundant if removing it still allows v to be reached from u.

However, RBSS induces a major bottleneck in the capture-and-replay pipeline due to its inefficiency. First, it generates an unnecessarily large dependency graph containing many redundant edges, increasing the cost of the pruning step. Second, RBSS could show quadratic time complexity in relation to the number of requests, owing to repeated backward scans for each session. As a result, it can constitute over half of the total end-to-end time [4]. As customers continuously capture and replay evolving workloads to rapidly assess divergences in system changes, there is a substantial need to accelerate dependency graph generation.

We demonstrate DoppelGanger++ [4], a novel database replay system supporting a fast dependency graph generation. It efficiently removes two types of *dominant*, redundant edges in the dependency graph called object transitivity (OT) and inter-session transitivity (IT) (see Section 3.1 for those definitions), during dependency graph generation. For this, DoppelGanger++ employs a novel and efficient dependency graph generation algorithm called stateful single forward scan (SSFS), which avoids repetitive scans over requests using a novel memoization technique. Consequently, SSFS considerably boosts the dependency graph generation by up to two orders of magnitude compared to RBSS, reducing more than 50% of the end-to-end time in the capture-and-replay pipeline.

The main contribution of this paper is to analyze how DoppelGanger++ efficiently generates a compact dependency graph compared to RBSS. First, to help better understand DRS, we demonstrate the complete database replay pipeline using SAP HANA Cockpit,

*Corresponding author

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685863

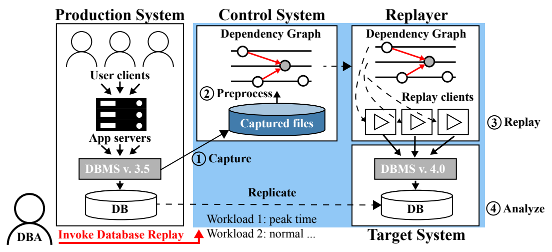


Figure 1: The architecture of DoppelGanger++.

which provides web interfaces to administrate the database, including capturing and replaying workloads with DoppelGanger++. By following the demonstration scenario in Section 4, the audience will see that DoppelGanger++ detects performance degradation due to software changes. Then, we showcase a web application that allows users to engage with the generation of dependency graphs in DoppelGanger++ through a live animation of the algorithm's execution, accompanied by visualizing memo structures.

2 ARCHITECTURE

Figure 1 shows the overall architecture of DoppelGanger++, including the four steps in its workflow. It consists of a production system, a control system, a replayer, and a target system. The production and target systems capture and replay a workload, respectively. The control system receives the workload from the production system and generates the dependency graph. The replayer loads the dependency graph and sends the requests associated with their dependency information to the target system. By separating the control and target system from the production one, the users can replay the workload without performance regression in their production system.

During the capturing step, DoppelGanger++ automatically captures the workload information, such as requests and execution contexts, from the production system with minimal overhead. It categorizes and archives captured information, including SQL and transaction data, into distinct files based on their respective types. It provides filters for the captured requests, allowing users to capture only workloads crucial for replay and to exclude sensitive information. Additionally, DoppelGanger++ creates a backup of the current snapshot, ensuring it is available for subsequent replay.

In the preprocessing step, a control system processes captured workload files and produces dependency graph files using SSFS for consistent workload replay. The input files hold data on requests made by each database session at capture time. The output files can be used for replay multiple times. The generated dependency graph file serializes requests, associating each with its dependent requests in other concurrent sessions. Note that if we use RBSS, the generation of the dependency graph would constitute over half of the overall end-to-end time.

In the replaying step, the captured workload is replayed using dependency graph files, maintaining the transactional order on the target system, which is initialized with the previously backed-up snapshot. In this process, loader threads load the dependency graph files and feed the requests on distinct request queues for each session. A request dispatcher manages each queue, determining when to execute the requests. These requests are then dispatched to execution threads. The requests not having incoming edges are executable without waiting. A non-commit request first acquires a snapshot and eliminates its outgoing edges, allowing long-read transactions to be completed successfully without obstructing write transactions. On the other hand, a commit request obtains a commit timestamp, and its execution eliminates the outgoing edges, facilitating successful commitment. After executing each (non-commit and commit) request, the execution threads return the result.

In the analysis step, the system compares the replayed outcomes with those captured and then visualizes the analysis reports. The comparison is performed regarding resource usage, performance, and consistency. To assess performance, it measures the execution times of respective requests as well as system-level throughput and resource consumption. For consistency check, it measures the overall database state and the execution result of respective requests.

3 DEPENDENCY GRAPH GENERATION

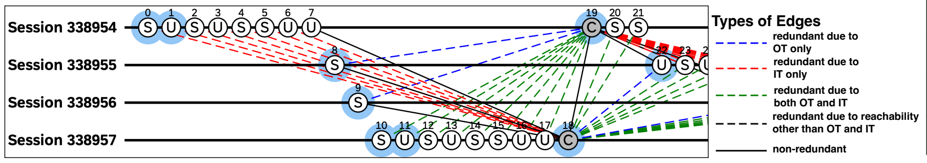
3.1 Workloads and Dependency Graphs

A workload $\mathcal{W}$ is represented as a directed graph $\mathcal{G}_{\text{ini}} = (\mathcal{V}_{\mathcal{R}}, \mathcal{E}_{\text{ses}})$ where each node corresponds to a request in $\mathcal{W}$, and consecutive requests are connected by edges in a session (e.g., the nodes in Session 338954 in Figure 2a). Here, each request r has attributes including a logical timestamp ($r.ts$), a session ID ($r.sid$), and a set of objects accessed by r ($r.objs$). In cases where r is a commit request, $r.objs$ denotes the set of all objects modified by the committed transaction. As in [5], 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. When a transaction does not modify any object, its commit request is not dependent on any request from the other sessions. Therefore, during the dependency graph generation step, it is simply disregarded, only to be replayed at the end of the transaction in the replay step. Requests within a session will be replayed in timestamp order, as imposed in $\mathcal{E}_{\text{ses}}$. However, $\mathcal{E}_{\text{ses}}$ does not need to be generated explicitly, as requests in a session are stored in a timestamp order [5]. Note that $\mathcal{G}_{\text{ini}} = (\mathcal{V}_{\mathcal{R}}, \mathcal{E}_{\text{ses}})$ is not a complete dependency graph since it only includes the ordering constraints within each session.

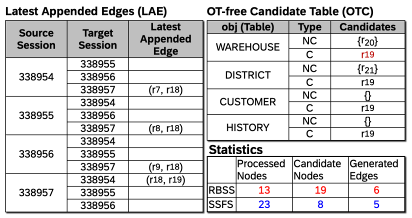
Given a workload, we need to generate the edges for every pair of dependent requests (r, r'). Here, we assume the isolation level is either statement-level or transaction-level snapshot isolation, which our underlying DBMS supports. Then, r and r' are dependent if the two requests access a common object, and at least one of them is a commit request modifying the object.

However, generating all the edges is unnecessary, as some edges are redundant. For example, in Figure 2a, all dotted edges are redundant, whereas only few solid edges are non-redundant. Generating redundant edges increases the cost of the transitive reduction since all redundant edges will be removed.

To formally address this problem, DoppelGanger++ avoids generating edges of OT and IT types. OT refers to redundancy due to a path where all requests on the path access a common object and every pair of consecutive requests in the path is dependent. IT refers to redundancy due to a path through requests in two sessions, regardless of whether the requests access a common object. IT and



(a) The dependency graph analysis page. The highlighted nodes access WAREHOUSE table.



Latest Appended Edges (LAE)

Source Session	Target Session	Latest Appended Edge
338954	338955	
	338956	
	338957	(r7, r18)
338955	338954	
	338956	
	338957	(r8, r18)
338956	338954	
	338955	
	338957	(r9, r18)
338957	338954	(r18, r19)
	338955	
	338956	

OT-free Candidate Table (OTC)

obj (Table)	Type	Candidates
WAREHOUSE	NC	{r20}
	C	r19
DISTRICT	NC	{r21}
	C	r19
CUSTOMER	NC	{}
	C	r19
HISTORY	NC	{}
	C	r19

Statistics

	Processed Nodes	Candidate Nodes	Generated Edges
RBSS	13	19	6
SSFS	23	8	5

(b) The algorithm analysis page ($r' = r_{22}$).

Figure 2: The screenshots of our web tool.

OT are dominant in the dependency graph. Note that RBSS avoids special cases of IT edges only [4].

For example, consider the dependency graph in Figure 2a. Then, (r_9, r_{19}) is an OT edge due to the path (r_9, r_{18}, r_{19}) where all requests access WAREHOUSE table (i.e., a common object) while every pair of consecutive requests is dependent. However, it is not an IT edge as the requests in the path are in three different sessions. The edges of OT only are highlighted in blue. On the other hand, (r_0, r_{18}) is an IT edge due to the path $(r_0, r_1, \cdots, r_7, r_{18})$ through requests in Sessions 338954 and 338957. However, it is not an OT edge as neither the requests in the path access a common object nor the pairs of consecutive requests in $r_0, r_1 \cdots, r_7$ are dependent (note they are all non-commit requests). The edges of IT only are highlighted in red. Finally, (r_{10}, r_{19}) is both OT and IT edge due to the path (r_{10}, r_{18}, r_{19}) which satisfies OT and IT connectivity. The redundant edges due to both OT and IT are highlighted in green.

3.2 Single Forward Session Scan (SSFS)

SSFS generates a compact dependency graph through a single scan over all requests using a memoization technique. While scanning each request r', SSFS generates the incoming edges of r' from the memo tables and updates them.

SSFS maintains the two succinct memo tables called the OT-free candidate table (OTC), and the latest appended edges between sessions (LAE). When generating the incoming edges of r', SSFS first enumerates the candidate source requests that do not have OT connectivity to r' using OTC, and then prunes the candidates that have IT connectivity to r' using LAE. Thus, we denote the resulting graph as $\mathcal{G}_{\text{IT[OT]}}$. Now, we explain each table and how SSFS generates the dependency graph using them.

OTC is a dictionary with keys represented by pairs of $(obj, type)$ and values represented by sets of requests. For key (obj, C), the value is the latest commit request r_C accessing obj. For key (obj, NC), the value is the set of non-commit requests accessing obj subsequent to r_C. For example, when processing r_{22} in Figure 2b, for key $(\text{WAREHOUSE}, C)$, OTC stores the latest commit request r_{19} accessing WAREHOUSE table (see Figure 2a for the requests accessing WAREHOUSE table). For key $(\text{WAREHOUSE}, NC)$, OTC stores $\{r_{20}\}$ as the set of non-commit requests accessing obj subsequent to r_{19}. Among the subsequent non-commit requests, OTC maintains only the latest one for each session as an optimization, since all requests in a session are connected due to $\mathcal{E}_{\text{ses}}$.

LAE is another dictionary, where each key is a pair of sessions, and the value is the latest appended edge between those sessions.

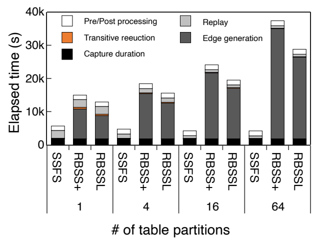


Figure 3: End-to-end time breakdown in TPC-C (256 clients).

For example, for a pair of sessions (338954, 338957) in Figure 2b, LAE stores (r_7, r_{18}) as the latest appended edge.

For the request r', SSFS first enumerates the candidate source requests without OT connectivity to r', by employing a two-step process. In the first step, SSFS iterates over every object $obj \in r'.objs$ to retrieve the candidate source requests from OTC. For brevity, we denote the retrieved request set from OTC using a key $(obj, type)$ as $\text{OTC}[obj, type]$. Given an obj, SSFS first retrieves $\text{OTC}[obj, C]$, that constitutes a singleton (i.e. a single request) having a dependency on r'. In addition, if r' is a commit request, SSFS additionally retrieves $\text{OTC}[obj, NC]$, whose requests have dependencies on r' only when r' is a commit request. Then, in the second step, SSFS can prune some of the identified candidate requests with OT connectivity to r' through the other retrieved requests. Note that a request r in the candidates may have OT connectivity to r' through other candidates retrieved using different obj or $type$ in keys. For the detailed algorithm, please refer to [4].

From the enumerated candidates using OTC, SSFS retains the latest request r for each session and discards the others, as the other requests in the same session have IT connectivity to r' through r. Then, for each r in the remaining candidates, SSFS retrieves the latest appended edge from LAE using key $(r.sid, r'.sid)$. If the edge constitutes the IT connectivity between r and r' (i.e., the source node of the edge succeeds r), SSFS discards r. Finally, it generates the incoming edges of r' from the remaining candidates.

Figure 3 shows end-to-end time breakdowns of SSFS and two optimized versions of RBSS. The dependency graph generation times of RBSSs take up to over 85% of the total end-to-end times, which are the bottlenecks, while those of SSFS are nearly negligible. Furthermore, in our extensive experiments using two synthetic benchmarks and a real-world customer workload varying capture duration, the number of clients, and the number of table partitions, SSFS shows robust performance considering both efficiency and compactness. Due to the space limitation, we omit them and encourage readers to refer to our paper for in-depth analysis [4].

4 DEMONSTRATION

The demonstration showcases the end-to-end database replay workflow finding a performance regression, and visualizes the dependency graph generation process to analyze its efficiency. A user can either capture new workloads or use pre-captured ones. We provide configurable test scripts for running a TPC-C workload with a new configuration as well as pre-captured ones with various configurations.

4.1 S1. DRS To Find Performance Degradation

In the first scenario, a user will have the experience of recognizing the risks of system changes in advance during the testing process through database replay. Consider that the user is a virtual service provider running a wholesaler management system which processes a TPC-C-like workload with an additional query accelerated by a secondary index. The user is planning a software update; however, the query optimizer of the new version of DBMS generates a suboptimal plan that does not leverage the secondary index for the additional query. Through database replay with DoppelGanger++, the user will detect performance regression for the additional query in the test system before updating the production system.

A user will follow the database replay workflow through the visualization tool in SAP HANA Cockpit, which is explained in Section 2. For the performance comparison in the following scenario, DoppelGanger++ uses both SSFS and RBSS. In order to simulate the changes in the query optimizer, we force the optimizer not to select the secondary index. Note that a user can apply other system changes, such as changing system configurations.

Finally, the web interface will provide the analysis report, which consists of four tabs: overview, load, performance comparison, and result verification. The overview tab contains the summary of the replay results. The performance comparison tab categorizes the replayed requests by whether they perform comparably, faster, or slower. Thus, a user will see the requests for the additional query slowed down in the test system due to not using the secondary index. The load tab shows the resource usage comparison, helping the administrators find a new resource contention. The last tab verifies each request produces the same result at capture and replay times, helping to find bugs.

4.2 S2. Dependency Graph Generation Analysis

In this scenario, our web tool visualizes the dependency graph generation phase to analyze the compactness of the dependency graph and the efficiency of the algorithm. The results for the captured workloads in the demonstration system, including those in the first scenario, are automatically imported.

The web tool provides three pages: 1) overview, 2) dependency graph analysis, and 3) algorithm analysis. When a user selects a workload, the overview page displays the summary of the captured workload information and visualizes the comparison of SSFS and RBSS in terms of algorithm efficiency and graph compactness. For algorithm efficiency, it visualizes the end-to-end times to execute capture-and-replay pipelines for both algorithms, with their breakdowns including dependency graph generation and transitive reduction times. For graph compactness, it visualizes the number of produced edges, where the portions for redundant edges are highlighted. As a result, a user will see how much DoppelGanger++ boosts database replay using SSFS.

Figure 2a shows the dependency graph analysis page, which visualizes dependency graphs to compare them and analyze the redundancies. Since the dependency graph is too huge and complex (i.e., containing more than millions of edges), one can adjust the window size to visualize tens or hundreds of requests at a time. This page shows three types of dependency graphs: 1) $\mathcal{G}_{\mathrm{IT[OT]}}$ (ours), 2) $\mathcal{G}_{\mathrm{RBSS}}$ (generated by RBSS), and 3) $\mathcal{G}_{\mathrm{col}}$ (containing all redundant edges). When the cursor hovers over each request, it displays its detailed information, including the SQL statement. The redundant edges are color-highlighted based on their types. Through this page, a user can visually compare the dependency graphs and inspect the redundant edges, figuring out that the two redundancy types are dominant and DoppelGanger++ generates a compact one.

The algorithm analysis page displays live animations for the dependency graph generation processes by SSFS and RBSS. The current node and the candidate nodes to be scanned in the graph are color-highlighted, and the statistics pane in Figure 2b shows the numbers of processed nodes, scanned candidate nodes, and generated edges. A user can adjust the playback speed or advance the generation process to a desired position. For SSFS, its memo tables are also displayed and updated according to the generation process, and the cells related to processing the current node are also highlighted. A user can see how each algorithm works, compare their efficiency, and understand how SSFS avoids repetitive scans and eliminates redundant edges.

5 CONCLUSION

DoppelGanger++ is a database replay system supporting fast dependency graph generation. The proposed demonstration showcases a use case of DoppelGanger++ to avoid risks from system changes and highlights its efficient dependency graph generation.

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). The presented demo is an academic prototype version which has been built by extending the existing commercial product version of SAP HANA Workload Capture and Replay. It does not necessarily imply that the presented feature will be incorporated into the product in the future.

REFERENCES

[1] Alfred V. Aho et al. 1972. The transitive reduction of a directed graph. *SIAM J. Comput.* 1, 2 (1972), 131–137.
[2] Gautam Altekar and Ion Stoica. 2009. ODR: Output-deterministic replay for multicore debugging. In *SOSP'09*. 193–206.
[3] Leonidas Galanis et al. 2008. Oracle database replay. In *SIGMOD'08*. 1159–1170.
[4] Wonseok Lee et al. 2024. DoppelGanger++: Towards Fast Dependency Graph Generation for Database Replay. *PACMMOD* 2, 1 (2024), 67:1–67:26.
[5] Konstantinos Morfonios et al. 2011. Consistent synchronization schemes for workload replay. *PVLDB* 4, 12 (2011), 1225–1236.