ORM-Lock: Source-Linked Diagnosis and Repair Guidance for ORM-Backed Database Deadlocks

Zainab Altamimi
Software Engineering Department, College of Computer and Information Sciences
King Saud University
Riyadh, Saudi Arabia
[446200456@student.ksu.edu.sa]

ABSTRACT

Object-Relational Mapping (ORM) frameworks simplify database access but can hide transaction boundaries, generated SQL operations, accessed resources, and operation order. This limited visibility makes it difficult to diagnose how ORM-level behavior may lead to database deadlocks. This paper presents ORM-Lock, an abbreviation for *ORM-based Deadlock Detection and Repair Guidance*, as a PhD research direction for source-linked diagnosis and repair guidance. ORM-Lock extracts transaction-scoped database operations into a source-linked operation graph, enriches this graph with ORM and DBMS concurrency semantics to identify candidate wait-for cycles, and maps these cycles to ranked repair candidates. The goal is not complete deadlock proof or automatic repair, but actionable guidance for developers investigating ORM-backed transaction risks.

KEYWORDS

Object-Relational Mapping (ORM), database transactions, database deadlocks, static analysis, dependency graph, repair guidance

VLDB Workshop Reference Format:
Zainab Altamimi. ORM-Lock: Source-Linked Diagnosis and Repair Guidance for ORM-Backed Database Deadlocks. VLDB 2026 Workshop: VLDB Ph.D. Workshop.

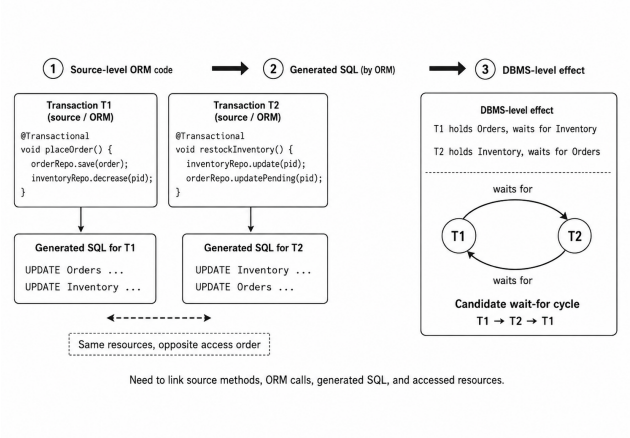

Figure 1: Illustrative ORM-to-SQL deadlock scenario.

1 INTRODUCTION

Modern web applications commonly rely on Object-Relational Mapping (ORM) frameworks to simplify database access. Instead of writing SQL statements directly, developers interact with databases through object-oriented APIs, while the ORM layer generates SQL statements and mediates database access. Although this abstraction improves productivity and maintainability, it can obscure transaction-scoped behavior such as generated SQL operations, accessed resources, and operation ordering [7, 9, 10]. This becomes especially problematic in web applications where transaction behavior may emerge from application-level logic and framework conventions [8].

Figure 1 illustrates this cross-layer challenge. Two transaction-scoped service methods may look like ordinary ORM operations, yet the generated SQL statements may access shared database resources in opposite orders. Such conflicting access orders may create cyclic waiting among transactions, which is the defining structure of a database deadlock [1, 5].

This work focuses on database transaction deadlocks, not circular method-call deadlocks in application control flow. Existing DBMSs can detect and resolve deadlocks at runtime, often by aborting one transaction [1, 5]. However, DBMS-level reports typically expose low-level evidence, such as locks, transactions, resources, and SQL statements. For ORM-backed applications, developers still need to map this evidence back to source methods, ORM calls, generated operations, accessed resources, and transaction interactions.

Existing work addresses parts of this problem. Source-level database analysis can recover SQL statements or identify ORM/database-access inefficiencies [7, 9, 10], while deadlock prediction and concurrency analysis techniques reason about locks, dependencies, and possible waiting patterns [2, 11]. WeSEER is closest to our setting because it targets database deadlock diagnosis for ORM-based web applications [3]. However, these lines of work are not fully connected for ORM-backed deadlock diagnosis with repository-level source links and ranked repair guidance.

To address this gap, we propose ORM-Lock, an abbreviation for *ORM-based Deadlock Detection and Repair Guidance*. ORM-Lock connects ORM-backed application code with DBMS-level concurrency behavior through three stages: extracting transaction-scoped operation graphs, enriching them with database concurrency semantics to diagnose candidate wait-for cycles, and mapping these cycles to ranked repair guidance. The goal is not exhaustive verification of all possible deadlocks or automatic rewriting of the

application, but to provide practical, source-linked guidance for understanding ORM-backed transaction risks. The main contributions of this research direction are:

- a cross-layer formulation of ORM-backed database deadlock diagnosis;
- a source-linked operation-graph representation for transaction-scoped database behavior;
- an ORM/DBMS-aware diagnosis approach for candidate wait-for cycles; and
- a repair-guidance stage that maps candidate cycles to ranked repair candidates.

2 PROBLEM FORMULATION

Given an ORM-backed application repository, ORM-Lock aims to recover transaction-scoped database behavior, reason about possible cross-transaction waiting dependencies, and report source-linked diagnosis and repair guidance. The input includes source code, ORM calls, transaction annotations or framework conventions, ORM mappings, configuration files, and schema metadata when available. The output is not a proof of deadlock freedom, but a developer-facing package containing candidate wait-for cycles, involved transactions, conflicting resources, code locations, explanations, and ranked repair candidates.

Table 1 summarizes the technical basis of each subproblem. ORM-Lock combines Java program analysis, wait-for graph reasoning, and repair-pattern ranking to connect ORM-backed source code with DBMS-level deadlock evidence.

Definition 1 (transaction-scoped operation sequence). A transaction-scoped operation sequence is an ordered sequence $S_T = \langle o_1, o_2, \ldots, o_n \rangle$ of database operations that may execute within a transaction scope T. Each operation corresponds to an ORM/database-access action observed in the application code.

Definition 2 (source-linked operation). A source-linked operation is represented as $o = \langle loc, orm, sql, r, e, pos \rangle$, where loc is the source location, orm is the ORM call, sql is the generated or inferred database operation, r is the accessed resource, $e \in read, write, read\text{-}write$ is the operation effect, and pos captures its position or may-before relation within the transaction scope.

Definition 3 (source-linked operation graph). The output of SP1 is an edge-labeled graph $G_O = (V, E, \lambda)$, where nodes represent transaction scopes, source-linked operations, and accessed resources. Edges represent containment, method calls, operation ordering, resource access, and read/write dependencies. The labeling function λ records the edge type and the supporting source or database evidence when available.

Definition 4 (candidate wait-for dependency). Given two transaction scopes T_i and T_j, a candidate wait-for dependency $T_i \rightarrow T_j$ exists when an operation in T_i may request a resource r after an operation in T_j may already hold an incompatible lock on r, under the assumed read/write effect, lock-compatibility model, and isolation level [1, 5].

Definition 5 (candidate wait-for cycle). Given G_O, SP2 constructs a candidate wait-for graph G_W. A candidate wait-for cycle is a cycle in G_W whose edges remain linked to the supporting source operations, ORM calls, database operations, and accessed resources. We use the term candidate because repository-level analysis may over-approximate feasible executions [3].

Definition 6 (source-linked repair guidance). Source-linked repair guidance is a ranked set of repair candidates for a candidate wait-for cycle. Each candidate contains a repair pattern, affected source locations, supporting evidence, applicability assumptions, and expected trade-offs. Examples include lock-order normalization, transaction-boundary restructuring, shorter lock holding, and localized retry or re-execution [4, 6, 8].

Based on this formulation, this work studies three research questions. **RQ1:** How can ORM-backed repositories be analyzed to construct G_O? **RQ2:** How can G_O be enriched to construct G_W and identify candidate cycles? **RQ3:** How can candidate cycles be mapped to ranked repair candidates that preserve source links and expose assumptions and trade-offs?

3 ORM-LOCK OVERVIEW

Figure 2 presents the high-level pipeline of ORM-Lock. The input is an ORM-backed application repository, including source code, ORM calls, mappings, configuration files, and schema metadata, together with analysis evidence such as transaction-boundary, ORM, DBMS, and index information when available. The output is a developer-facing package containing candidate wait-for cycles, conflicting resources, code locations, explanations, and ranked repair guidance.

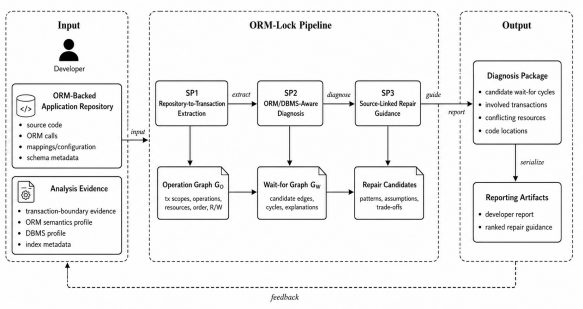

Figure 2: Overview of ORM-Lock from repository evidence to diagnosis and repair guidance.

The pipeline proceeds in three stages. SP1 applies Java program-analysis infrastructure and ORM-specific rules to recover transaction-scoped operations and construct the source-linked operation graph G_O [7, 9, 10]. SP2 enriches G_O with resource, read/write, ordering, lock, and isolation annotations to construct a candidate wait-for graph G_W and identify candidate cycles [1–3, 5, 11]. SP3 maps these cycles to repair patterns and ranks the resulting guidance by risk reduction, behavior-change risk, cost, and evidence strength [4, 6, 8, 9].

The feedback arrow represents iterative refinement: developer review or additional configuration and DBMS evidence can be fed back into ORM-Lock to refine later analysis runs. The next section expands the three subproblems with their concrete analysis steps and evaluation focus.

Table 1: Method mapping for ORM-Lock.

SP	Method direction	Output
SP1: Extraction	Soot/WALA-style interprocedural analysis; Spoon/JavaParser-style AST extraction; ORM-specific rules for transactions, repositories, mappings, and schema metadata [7, 9, 10].	Operation graph G_O.
SP2: Diagnosis	Operation annotation; lock-compatibility model; isolation assumptions; wait-for graph construction and cycle detection [1–3, 5, 11].	Wait-for graph G_W and candidate cycles.
SP3: Guidance	Repair-pattern taxonomy; applicability checks; ranking by risk reduction, behavior-change risk, cost, and evidence strength [4, 6, 8, 9].	Ranked repair candidates.

4 DETAILED SOLUTION DESIGN

This section expands the three subproblems with concrete examples. The running example follows Figure 1: one transaction updates `Orders` before `Inventory`, while another updates `Inventory` before `Orders`. ORM-Lock uses this type of source-to-database evidence to construct operation graphs, diagnose candidate wait-for cycles, and generate repair guidance.

4.1 SP1: Repository-to-Transaction Extraction

SP1 recovers transaction-scoped database behavior from ORM-backed repositories. It combines Soot/WALA-style interprocedural analysis, Spoon/JavaParser-style AST extraction, and ORM-specific rules to identify transaction boundaries, repository/JPA/Hibernate calls, entity-to-table mappings, and schema references [7, 9, 10]. This makes the transaction scope explicit: the transaction T is approximated from the annotated method body and the reachable ORM/database-access calls within its call chain. For example, if a service method annotated with `@Transactional` calls `orderRepo.save(order)` followed by `inventoryRepo.decrease(pid)`, SP1 treats both calls as part of the same transaction scope. Using ORM mappings, it summarizes them as operations such as `write(Orders)` and `write(Inventory)` and links each operation back to its source method and ORM call. The output is G_O, which records transaction scopes, operations, accessed resources, operation order, read/write effects, and source locations. Evaluation will compare extracted artifacts against manually inspected ORM-backed applications.

4.2 SP2: ORM/DBMS-Aware Diagnosis

SP2 converts G_O into candidate ORM-backed deadlock risks. It annotates operations with resources, read/write effects, order, transaction scope, lock mode, and isolation assumptions, then adds a wait-for edge $T_i \rightarrow T_j$ when T_i may request a resource held incompatibly by T_j. For example, if T_1 performs `write(Orders)` before `write(Inventory)`, while T_2 performs `write(Inventory)` before `write(Orders)`, SP2 can derive two candidate dependencies: T_1 may wait for T_2 on `Inventory`, and T_2 may wait for T_1 on `Orders`. Cycle detection over G_W then reports this as a candidate wait-for cycle with links to the supporting source methods, ORM calls, generated operations, and resources. This follows classical wait-for reasoning, dependency-based deadlock prediction, and ORM-based deadlock diagnosis work [1–3, 5, 11]. Evaluation will assess whether reported cycles and explanations match plausible database-level conflicts.

4.3 SP3: Source-Linked Repair Guidance

SP3 maps candidate cycles to repair candidates using a repair-pattern taxonomy and ranking model. In the running example, the cycle arises from opposite resource-access order. A natural repair candidate is therefore lock-order normalization, where both transaction paths are refactored or constrained to access `Orders` and `Inventory` in the same order. If the cycle is instead caused by a broad transaction scope, SP3 may suggest transaction-boundary restructuring; if locks are held longer than needed, it may suggest shorter lock holding or early release; and if the conflict is localized, it may suggest retry or fine-grained re-execution. Candidates are ranked by expected risk reduction, behavior-change risk, implementation cost, and evidence strength [4, 6, 8, 9]. Evaluation will assess applicability, source-location accuracy, and ranking usefulness.

5 RELATED WORK

Prior work provides partial foundations for ORM-Lock across source-level database analysis, deadlock diagnosis, and transaction-conflict mitigation. Source-level database analysis can recover SQL statements or identify ORM/database-access inefficiencies [7, 9, 10]. JSquash motivates the feasibility of extracting database behavior from application code, while Reformulator and empirical studies of web-application database inefficiencies show that ORM/database-access patterns can be analyzed and improved. However, these works mainly target SQL extraction, N+1 queries, redundant accesses, or performance issues rather than transaction-scoped wait-for risks.

Deadlock prediction and concurrency diagnosis reason about locks, traces, dependency relations, and possible waiting behavior [2, 11]. WeSEER is closest to our setting because it diagnoses database deadlocks in ORM-based web applications using concolic execution and trace analysis [3]. These works motivate dependency-based diagnosis, but ORM-Lock focuses on preserving repository-level source links across operation graphs, candidate wait-for cycles, and developer-facing explanations.

Transaction-conflict mitigation techniques provide mechanisms such as early lock release, fine-grained re-execution, and evidence that application-level transaction coordination can be error-prone [4, 6, 8]. These works inspire the repair-pattern taxonomy used by ORM-Lock. In contrast to DBMS-, protocol-, or runtime-level mitigation, ORM-Lock targets the application layer by translating diagnosed wait-for cycles into ranked repair candidates that developers can inspect before applying changes.

6 CONCLUSION

This paper presented ORM-Lock, a research direction for source-linked diagnosis and repair guidance for ORM-backed database

deadlocks. ORM-Lock connects ORM-level evidence with DBMS concurrency behavior through a source-linked operation graph G_O, a candidate wait-for graph G_W, and ranked repair guidance. The goal is to help developers trace candidate deadlock risks back to source code and inspect possible mitigations, rather than to prove deadlock freedom or apply repairs automatically. Future work will implement and evaluate the three stages on ORM-backed applications and transaction-heavy case studies.

ACKNOWLEDGMENTS

This PhD research is supervised by Sadeem Alsudais.

REFERENCES

[1] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.

[2] Yan Cai, Ruijie Meng, and Jens Palsberg. 2020. Low-Overhead Deadlock Prediction. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, 1298–1309. https://doi.org/10.1145/3377811.3380367

[3] Zhiyuan Dong, Zhaoguo Wang, Chuanwei Yi, Xian Xu, Jinyuan Zhang, Jinyang Li, and Haibo Chen. 2023. Database Deadlock Diagnosis for Large-Scale ORM-Based Web Applications. In *2023 IEEE 39th International Conference on Data Engineering (ICDE '23)*. IEEE, 2864–2877. https://doi.org/10.1109/ICDE55515.2023.00219

[4] Zhiyuan Dong, Zhaoguo Wang, Xiaodong Zhang, Xian Xu, Changgeng Zhao, Haibo Chen, Aurojit Panda, and Jinyang Li. 2023. Fine-Grained Re-Execution for Efficient Batched Commit of Distributed Transactions. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1930–1943. https://doi.org/10.14778/3594512.3594523

[5] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann.

[6] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. 2021. Releasing Locks As Early As You Can: Reducing Contention of Hotspots by Violating Two-Phase Locking. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. Association for Computing Machinery, 658–670. https://doi.org/10.1145/3448016.3457294

[7] Dietmar Seipel, Andreas M. Boehm, and Markus Fr"ohlich. 2011. JSquash: Source Code Analysis of Embedded Database Applications for Determining SQL Statements. In *Applications of Declarative Programming and Knowledge Management*. Lecture Notes in Computer Science, Vol. 6547. Springer, 153–169. https://doi.org/10.1007/978-3-642-20589-7_10

[8] Chuzhe Tang, Zhaoguo Wang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. 2022. Ad Hoc Transactions in Web Applications: The Good, the Bad, and the Ugly. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, 4–17. https://doi.org/10.1145/3514221.3526120

[9] Alexi Turcotte, Mark W. Aldrich, and Frank Tip. 2022. Reformulator: Automated Refactoring of the N+1 Problem in Database-Backed Applications. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. Association for Computing Machinery. https://doi.org/10.1145/3551349.3556911

[10] Cong Yan, Junwen Yang, Alvin Cheung, and Shan Lu. 2017. Understanding Database Performance Inefficiencies in Real-World Web Applications. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management (CIKM '17)*. Association for Computing Machinery, 1299–1308. https://doi.org/10.1145/3132847.3132954

[11] Jinpeng Zhou, Hanmei Yang, John Lange, and Tongping Liu. 2022. Deadlock Prediction via Generalized Dependency. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22)*. Association for Computing Machinery, 312–323. https://doi.org/10.1145/3533767.3534377