Brief Announcement: On the Correctness of Transaction Processing with External Dependency

Masoomeh Javidi Kishi
Lehigh University, Bethlehem, PA, USA
maj717@lehigh.edu

Ahmed Hassan
Alexandria University, Alexandria, Egypt[1]
ahmed.hassan@alexu.edu.eg

Roberto Palmieri
Lehigh University, Bethlehem, PA, USA
palmieri@lehigh.edu

Abstract

We briefly introduce a unified model to characterize correctness levels stronger (or equal to) serializability in the presence of application invariant. We propose to classify relations among committed transactions into data-related and application semantic-related. Our model delivers a condition that can be used to verify the safety of transactional executions in the presence of application invariant.

2012 ACM Subject Classification Theory of computation → Concurrency

Keywords and phrases Transactions, Dependency Graph, Concurrency

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.46

Funding This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0367 and by the National Science Foundation under Grant No. CNS-1814974.

1 Introduction

When the concurrency control implementation of a transactional system is required to enforce an application-level invariant on shared data accesses (i.e., an expression that should be preserved upon every atomic update [4]), ad-hoc reasoning about its correctness is a tedious and error-prone process. Traditional (data-related) constraints (e.g., transaction conflicts) are well-formalized with established correctness levels, such as Serializability and Snapshot Isolation [1]. However, a unified model encompassing the various *external* (semantic-related) constraints that enforce application invariant has not been formalized yet.

In this brief announcement we make a step towards defining such a model. We introduce a theoretical framework that formalizes correctness levels stronger than (or equal to) serializability by defining their transaction ordering relations as a union of two sets of data and external dependency. This approach is opposed to the traditional way of defining these relations through an ad hoc analysis. This framework can be used to define an offline checker that verifies the safety of transactional executions. The intuition behind our formalization is simple. Assuming a serializable concurrency control [1], relations between transactions in an execution can be characterized as data dependency, if they are generated by data conflicts, or external dependency, if they affect the satisfaction of application invariant. This decomposition allows us to define a methodology to enrich the traditional transaction Direct

[1] Ahmed Hassan is currently affiliated with Lehigh University, USA.

Serialization Graph (DSG) [1] with such external ordering relations. We use the formalization to introduce a safety condition that verifies correctness of transactional executions (Theorem 3).

We motivate our model by showing an example of application with associated invariant. The example mimics a simple monetary application that imposes different requirements to clients interacting from different branch locations of the bank. The application mandates the following invariant: when a transaction is issued by a client in one branch, this transaction accesses the modifications performed by the latest transactions completed on the same branch prior its starting. At the same time, the application does not require special constraints on the order of monetary transactions issued from other branches. That is, transactions from a remote branch should execute atomically and in isolation, but they might access stale data.

Suppose clients C_1 and C_2 from branch α issue two subsequent non-concurrent transactions T_1 and T_2 accessing the same bank account Ac. The first deposits \$10 and the second checks the total amount of Ac and then withdraws the latest deposited amount (\$10). According to the application semantics, T_2 must observe the deposit by T_1. Consider another transaction T_3, issued by a client from branch β doing auditing on accounts, including Ac. Application semantics for T_3 does not enforce any requirement on the set of transactions whose outcome should be observed, including T_1 and T_2. A serializable concurrency control would "only" guarantee a transactions order of T_1, T_2 and T_3 equivalent to some serial order. This serial order does not consider the application invariant and might order T_2 before T_1. Such a mismatch is due to the lack of application invariant representation in the concurrency control.

One solution to overcome this problem in a serializable concurrency control is to provide session guarantee [3], meaning transactions from one branch belong to the same session. This guarantee imposes an additional constraint between T_1 and T_2 where T_2 must observe the output of T_1. Clearly, T_3 would belong to a different session. The other solution would be adopting a stronger correctness level (e.g., strict serializability [1]) among all transactions, irrespective of their originating branch. An even more conservative solution is to apply external consistency [2], which brings the clients perceived order among transactions into the concurrency control so that mismatches are prevented.

With our unified model, these three correctness levels can be modeled in the same way as a combination of data-related transaction dependency, to satisfy serializability constraints, and external transaction dependency, to satisfy application invariant. This way, despite the differences among these correctness levels, our model can assess the correctness of concurrency controls that satisfy each of them by relying on a single framework.

2 Formalization

A history [1] models the interleaved execution of a set of transactions T_1, T_2, ..., T_n, as an ordered sequence of their operations (such as *read*, *write*, *abort*, *commit*). The dependency graph for a history $\mathcal{H}$, denoted as $DSG(\mathcal{H})$, represents the data-related dependency among transactions in $\mathcal{H}$. Roughly, in this graph each node is a committed transaction in $\mathcal{H}$, and each directed edge between two nodes can be of the following categories:

- *read dependency:* ($T_i \xrightarrow{\text{WR}} T_j$) A transaction T_j read-depends on T_i if a read of T_j returns a value written by T_i.
- *write dependency:* ($T_i \xrightarrow{\text{WW}} T_j$) A transaction T_j write-depends on T_i if a write of T_j overwrites a value written by T_i.
- *anti-dependency:* ($T_i \xrightarrow{\text{RW}} T_j$) A transaction T_j anti-depends on T_i if a write of T_j overwrites a value previously read by T_i.

▶ **Definition 1.** *$DSG(\mathcal{H})$ contains a set of tuples and each tuple has the following form: $(T_i, T_j, type)$. This representation shows that a directed data-related (read/write/anti-) dependency edge exists from transaction T_i to transaction T_j. $DSG(\mathcal{H}) = \{(T_i, T_j, type) : i, j \in \{1, .., n\} \wedge type \in \{RW, WW, WR\}\}$.*

Since our model focuses on correctness levels stronger than, or equal to, serializability, we recall that a history $\mathcal{H}$ is serializable if its corresponding DSG does not contain any cycle [1]. Performing an offline analysis of the DSG graph is a convenient tool for reasoning about the correctness of data-related dependencies produced by a concurrency control. However, it does not help verifying correctness of application when invariant should be preserved in addition to serializability. Our model aims at filling this gap, as follows.

▶ **Definition 2.** *An External Dependency Graph (EDG) for a given history $\mathcal{H}$, denoted as $EDG(\mathcal{H})$, determines application-level constraints. In this graph, an edge from transaction T_i to transaction T_j means an application-level requirement forces an external dependency between T_i and T_j. We say T_j externally-depends on T_i ($T_i \xrightarrow{EXT} T_j$).*

Intuitively, application invariant expressed by EDG should neither violate data-related dependency produced by the concurrency control nor include any two contradicting constraints. This observation leads to the following theorem where, informally, we consider both DSG and EDG as a single graph made by the union of them. We can check if a history is serializable and does not violate application invariant by verifying that the aforementioned single graph does not contain any cycle.

First, given a history $\mathcal{H}$ of n transactions, we define DSG, EDG, and their union as follows:

- $DSG(\mathcal{H}) = \{(V, E1) : V = \{T_i : i \in \{1, .., n\}\} \wedge E1 = \{(T_i, T_j, type) : i, j \in \{1, .., n\} \wedge type \in \{WR, WW, RW\}\}$.
- $EDG(\mathcal{H}) = \{(V, E2) : V = \{T_i : i \in \{1, .., n\}\} \wedge E2 = \{(T_i, T_j, type) : i, j \in \{1, .., n\} \wedge type \in \{EXT\}\}$.
- $DSG(\mathcal{H}) \cup EDG(\mathcal{H}) = (V, E1 \cup E2)$.

We now define our new *External Serializability* consistency level. We call a history $\mathcal{H}$ Externally Serializable (or EC-SR) if: *1)* it is serializable, and *2)* external dependency defined by the edges of its EDG are not violated. To prove that, it is necessary and sufficient to show that the union of its DSG, built from the concurrency control implementation, with its EDG, built from application invariant, does not have any cycle. We formalize that in the following theorem (the proof is intuitive and omitted due to space limitations):

▶ **Theorem 3.** *A history $\mathcal{H}$ satisfies EC-SR iff $DSG(\mathcal{H}) \cup EDG(\mathcal{H})$ does not have any cycle. A concurrency control CC satisfies EC-SR iff all the histories produced by CC are EC-SR.*

References

1 Atul Adya. Weak consistency: a generalized theory and optimistic implementations for distributed transactions, 1999.

2 James C Corbett et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

3 Khuzaima Daudjee and Kenneth Salem. Lazy database replication with ordering guarantees. In *ICDE*, pages 424–435. IEEE, 2004.

4 Tim Harris and Simon Jones. Transactional memory with data invariants, 2006.