Information Systems 58 (2016) 14–23

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Determining serialization order for serializable snapshot isolation

Elizabeth J. O'Neil*, Patrick E. O'Neil

Department of Computer Science, University of Massachusetts Boston, 100 Morrissey Blvd., Boston, MA 02125, USA

ARTICLE INFO

Article history:
Received 8 January 2015
Received in revised form
4 January 2016
Accepted 4 February 2016
Available online 13 February 2016

Keywords:
Database management
Snapshot isolation
Serializability
Concurrency control
Time travel query
Timestamp

ABSTRACT

Snapshot Isolation (SI) is in wide use by database systems but does not guarantee serializability. Serializable snapshot isolation (SSI) was defined in 2008 by Cahill, Röhm, and Fekete in Ref. [1] and an enhanced version that we call ESSI was defined in 2009 by the same authors [2]. Both guarantee serializable execution by aborting transactions that might be involved in anomalies, but occasional transactions are aborted unnecessarily. The resulting commit order is often different from the serialization order and this can confuse bank auditors, for example. In this paper we show how to determine the proper serialization order of these transactions and store this information for later access. With this known serialization order the database system can perform time-travel queries to capture snapshots of consistent database states in the past. Our algorithm assigns to each SSI or ESSI transaction at its commit time an appropriate serialization order timestamp that falls within the transaction lifetime, i.e., between its start-time and commit-time. The algorithm uses only information already gathered by the algorithm implementing ESSI in [2].

1. Introduction

Snapshot isolation (SI) was originally defined in [3], where its lack of serializability was discussed, as well as its significant advantages that readers do not block writers and writers do not block readers. It is now in use in Oracle and Microsoft SQL Server, and Postgres (before version 9.1) among others. Serializable snapshot isolation (SSI) is defined in [1] and Enhanced SSI (here abbreviated ESSI) in [2]. These both provide serializable isolation, and are the main subject of our results. The open-source database Postgres starting with version 9.1 implements ESSI for its serializable isolation level [4], with an additional optimization for read-only transactions. Another closely related form of serializable SI is Precisely Serializable Snapshot Isolation (PSSI) defined in [5]. In SSI, ESSI, and PSSI, reads are handled as in SI, that is, a transaction reads the latest committed version of a row that existed at its own start time. Writes are also handled as in SI, that is, transactions are aborted to ensure that no two concurrent transactions write the same data. In what follows we will refer to "The SI Family" with "members" from the set {SI, PSSI, ESSI, SSI}, listed here in order of increasing number of aborts to ensure serializability, starting from zero for SI.

Any serializable scheduler guarantees the existence of a serialization order for committed transactions, that is, the order of transactions in some conflict-equivalent serial execution. In strict 2PL, the serialization order is simply the commit order. In SSI, ESSI, and PSSI the serialization order is not always the commit order, as shown by Example 1 below.

Example 1. Serialization order can differ from commit order in SSI, ESSI, and PSSI

Consider the execution shown in the schedule and temporal diagram of Fig. 1 below. An antidependency is

* Corresponding author. Tel.: +1 617 354 6460.
E-mail addresses: eoneil@cs.umb.edu (E.J. O'Neil), poneil@cs.umb.edu (P.E. O'Neil).

http://dx.doi.org/10.1016/j.is.2016.02.001
0306-4379/© 2016 Elsevier Ltd. All rights reserved.

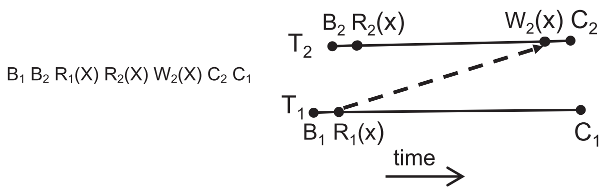

Fig. 1. Example 1, showing Serialization Order T_1 T_2 opposite to Commit order T_2 T_1.

depicted with a dashed arrow, following [1] and [2]. The transaction starts with the begin (B) operation. The serialization order is T_1 T_2 because of the R–W conflict from T_1 to T_2 ($R_1(X)$ and $W_2(X)$), where T_2 overwrites the X value accessed by T_1. We call this R–W conflict an "item antidependency" (Definition 2).

The serialization order is not always obvious in SSI, ESSI and PSSI, so the natural question is how we can efficiently determine that order. The main result of this paper is an algorithm for determining the serialization order position of each transaction running under ESSI or SSI (but not PSSI) at its commit time. It is practical in the sense that it only uses information that ESSI already uses to implement its read, write and commit operations. We can also do this determination with SSI, but must then use some of ESSI's capabilities. See Section 5. In Section 4.1, we show how PSSI's allowed behavior eludes our treatment.

The organization of this paper is as follows. In Section 2 we define the needed concepts of snapshot isolation, characterize the three serializable SI methods, and show elementary results about serialization orders. In Section 3 we will prove the main result of the paper, an algorithm to assign a Serialization Timestamp to each ESSI or SSI transaction at its commit time. Serialization Timestamps provide by their numerical order a serialization ordering for the transactions. In Section 4 we will provide further related results, including what can happen in non-ESSI histories such as PSSI histories. Section 5 briefly discusses implementation of Serialization Timestamps. Section 6 discusses related work.

2. Background

We assume that database timestamps (e.g., for transaction start and commit times) are provided by a central facility to provide a system-wide time. In most systems, this facility provides unique times on each access to it. We will assume that all start times and commit times are unique for ease of presentation. In fact, it is possible to extend this analysis to include the cases where the start time of one transaction can equal the commit time of another transaction, and two commit times can be equal, but in this extended case more transactions are aborted. See Section 3.3 for details. Since start and commit times are unique, concurrent transactions cannot overlap at just one point in time, but rather have non-trivial overlaps in time.

Definition 1. *SI Reads and Writes.* In all members of the SI Family, a read by T_j finds the last committed version of a selected data item prior to the start time of T_j. The set of versions of data, each of which is the last committed version before T_j's start time, is called the *snapshot* from which T_j is reading. In write actions, the SI Family uses a rule that ensures that writes by two concurrent transactions never update the same data item. One such rule is *first-committer-wins*, which specifies that if multiple concurrent transactions update the same data item, the first one to commit succeeds, and other transactions abort. Alternatively, some systems (such as Oracle) use *first-updater-wins*, a rule that if multiple concurrent transactions update the same data item, the first update is successful, and other transactions abort, after waiting for the first updater to commit.

Definition 1 implies that a transaction never reads a version written by a concurrent transaction. Data dependencies between transactions can be *direct dependencies*, meaning W–R or W–W dependencies, or *antidependencies*, which are the R–W dependencies, as detailed in Definition 2. Of these, the antidependencies are the important ones to track in SSI and ESSI and for our work on serialization ordering for these isolation methods.

Definition 2. An *antidependency* is a R–W dependency between two transactions, either an item or predicate antidependency, defined as follows for all members of the SI family. An *item antidependency* consists of a read of some data item x by transaction T_i and a write of x by a transaction T_k that creates a version of x after the version read by T_i. (See Example 1 above and Example 2 below.) A *predicate antidependency* occurs when there is a predicate read by T_i and a write by T_k of some data item x, where the version of x written by T_k was not accessed by T_i *but changes the result of the predicate read by* T_i. (See Example 3 below.)

As in [6], we symbolize an antidependency by $\dashrightarrow$, an arrow with dashes in its shaft. Note that although it is conventional to call an antidependency a "*R*–*W*" dependency, in SI family executions the write operation may precede the read operation in time, as shown by Example 2. Since the operation $R_1(X)$ reads the snapshot as of transaction-start, it is effectively moved back in time to before the write operation.

Example 2. Fig. 2. shows an item antidependency with the write executing first.

In Fig. 2, R_1(X) reads from the snapshot that corresponds to the transaction start at B_1, that is, the database state *before* $W_2(X)$ occurs. Thus even though $R_1(X)$ follows $W_2(X)$ in real time, there is an antidependency from T_1 to T_2 here, which creates an ordering requirement (T_1 before T_2).