1997-12-01

18th IEEE Real-Time Systems Symposium: Work in Progress Sessions

Jones, Michael
Boston University Computer Science Department

http://hdl.handle.net/2144/3748

Boston University
Exploring Consistency of Read-Only Transactions in Real-Time Systems

Kwok-Wa Lam¹, Sang H. Son² and Sheung-Lun Hung¹

Department of Computer Science¹
City University of Hong Kong,
Hong Kong.
kwlam@cs.cityu.edu.hk

Department of Computer Science²
University of Virginia,
U.S.A.
son@cs.virginia.edu

Abstract

In this paper, we describe our current work on exploring the consistency of read-only transactions (ROT) in real-time systems. A ROT is a transaction that only reads, but does not update any data items. Since there is a significant proportion of ROTs in several real-time systems, it is important to investigate how to process ROTs efficiently with separate algorithms. We identify three different consistency requirements for ROTs. Particularly, we define a weaker form of consistency, view consistency, which allows ROTs to perceive different serialization order of update transactions, thus permitting non-serializable execution of transactions. However, ROTs are still ensured to see consistent data. Based on view consistency, we present two algorithms which let ROTs read the most recent and consistent data without interfering with update transactions. The recency of data read by a ROT could be important in some real-time applications.

1. Introduction

A read-only transaction (ROT) is a transaction that only reads, but does not update any data items [1]. In real-time application domains, a ROT may be used to keep track of the movement of airplanes in a radar tracking system or to monitor the fluctuations of prices of stock shares in a stock market system. ROTs can be processed with general concurrency control protocols, which is the case in most previous studies on real-time concurrency control protocols [4, 5]. That is, the concurrency control protocols process ROTs as ordinary update transactions. In this case, ROTs may be required to hold locks on large amount of data items for long periods of time, thus causing update transactions to suffer long delays. However, it is also possible to use separate processing algorithms for ROTs in order to improve efficiency. With this approach, the separate algorithm can make use of the knowledge that ROTs will not update any data items.

Another important issue of processing ROTs in real-time applications is the requirement of recency of data. ROTs may be required to read the most recent versions of data which reflect the latest state of the system. Reading stale data means to reflect the past system state and gives wrong signals to initiate invalid actions to be taken. For example, a ROT may read the price values of a number of stock shares to calculate an index and return the result to an investor for stock trading decision. Suppose the prices of some of the shares have dropped significantly in the middle of processing the ROT. If we force the ROT to read the stale prices of the shares for data consistency, as required by serializability, it will deliver an incorrect result based on which the investor could probably make a wrong decision.

However, to our knowledge, no study has paid attention to ROTs in the context of real-time database systems. In fact, the semantics of ROTs has been exploited extensively to improve system efficiency in non-real-time database systems [2, 3]. For example, multiversion two-phase locking (MV2PL) [2] is an extension of two-phase locking (2PL) with versioning to avoid the data contention problem of 2PL. In these studies, multiversions of data items in the databases are maintained to allow ROTs to run against past transaction consistent database states. The existence of multiversions of data items allows ROTs to serialize before all concurrent update transactions. The obvious advantage of this is that ROTs and update transactions do not conflict. The cost paid for it is the increased storage requirement and the overheads of retrieving old versions of data items in processing of ROTs.

MV2PL may be helpful in reducing the degree of data contention in real-time database systems, thus increasing the probability of transactions meeting their deadlines. MV2PL reduces data contention by making ROTs read stale data. In this regard, MV2PL may not be suitable for some real-time applications where reactive actions should be based on the latest state of the system. In other words, MV2PL may not be adequate in ensuring the recency requirement for some real-time applications. Specifically, consider the following example.

Example 1: Suppose a ROT Q₁ reads data items x and y and an update transaction U₂ writes y. We assume that U₀ is the last transaction that has created x and y before Q₁ and U₂ start. Consider the following schedule:

\[ S₁ : w(x₀) \quad w(y₀) \quad r(x₀) \quad w(y₁) \quad c₂ \]

where \( r(x₀) \) denotes transaction Tᵢ reads a version of data item x written by Tᵢ and \( w(x₁) \) denotes that a new version of x created by Tᵢ, and \( c_2 \) denotes the commitment of Tᵢ. In S₁, Q₁ has read the version of x₀ and U₂ has just committed its new version of y, y₂. If Q₁ and U₂ are executed under MV2PL, Q₁ will read the stale version, y₀, instead of the new version, y₂. This example shows the problem that although Q₁ gives a
logically correct result, the result does not reflect the latest state of the system.

For certain real-time applications, the correctness based on serializability may not be necessary. To allow timely transaction executions and meeting deadlines, we may wish to relax serializability to increase the amount of concurrency of processing transactions. In fact, some research effort has proposed some protocols to allow non-serializable execution of transactions to gain performance advantages [7, 8]. Their objective is to reduce data contention, thus allowing more execution schedules to speed up the processing of transactions. The cost paid for their protocols is that ROTS will read inconsistent data or stale data. Moreover, they did not address the recency requirement.

Since there is a significant proportion of ROTS in several real-time applications, it is important to investigate how ROTS can be processed efficiently with separate algorithms. In this paper, we investigate how to ensure ROTS to read consistent data while non-serializable execution of transactions is allowed. The primary objective of relaxing consistency requirement of serializability is to trade for the recency requirement of ROTS.

2. Consistency of Read-Only Transactions

We assume that a database state is consistent if a set of static integrity constraints are satisfied. An integrity constraint is a predicate defined on the database which describes the relationships that must hold among the data items and their values. We also assume that each transaction must preserve consistency. That is, a transaction, when run alone and to completion without interference from other transactions, takes the database from one consistent state to another. We assume that it is possible a priori to distinguish ROTS from update transactions. Hence, transactions can be identified as read-only from their inception. Given these assumptions, we investigate possible consistency requirements for ROTS.

ROTS may be required to read the values of data items that are consistent. This requirement demands that the database state read by ROTS satisfies the set of integrity constraints. We call this requirement consistency. Obviously, it will incur considerable overheads to evaluate the set of integrity constraints whenever a ROT reads a data item. In addition to inefficiency, it might seem that this requirement is too weak to be useful.

Another consistency requirement for ROTS is the requirement of serializability which is the standard correctness of transaction processing. Serializability means that the concurrent execution of a set of transactions is equivalent to some serial execution of the same transaction set [1]. Both update transactions and ROTS are indiscriminately considered ordinary transactions and required to serialize with each other on a partial serialization order. Specifically, all ROTS are required to read the database states produced by the same serialization order of update transactions. We call this requirement strong consistency. While strong consistency could make it easy to understand the interactions among transactions, it might be unnecessarily strong for processing ROTS, causing inefficiency. More important, it could also make ROTS read stale data.

Hence, it may be beneficial to relax the requirement of strong consistency. A ROT could be required to read a database state which can be produced by the serializable execution of some subset of update transactions, but the ROT may not be required to serialize with other ROTS. That means, different ROTS can read the database states produced by different serializable executions of update transactions. We call this requirement view consistency. View consistency allows different ROTS to perceive different serialization order of update transactions. As serializable execution of update transactions leaves a database state consistent, it must be guaranteed that the state read by ROTS should satisfy the set of integrity constraints.

To illustrate the difference between the requirements of strong consistency and view consistency, consider the following example:

**Example 2**: Suppose we have the following two ROTS, Q1 and Q2, and two update transactions U3 and U4.

\[ Q_1 : r(x) r(y) \quad Q_2 : r(y) r(x) \]
\[ U_3 : r(x) w(x) \quad U_4 : r(y) w(y) \]

Consider the following schedule S2:

\[ S_2 : r(x_0) r(y_0) r(x_0) w(x_1) r(y_0) w(y_1) r(y_4) r(x_1) \]

In Figure 1, the serialization graph of S2 has a cycle among ROTS and update transactions, thus resulting in a non-serializable execution of transactions. Hence, strong consistency does not allow to process Q1 and Q2 in this execution. However, we can observe that the values read by Q1 are the result of a serial execution of the single transaction U4 while the values read by Q2 are the result of a serial execution of the single transaction U3. Q1 reads consistent data because it is serializable with respect to U3 and U4. The serialization order is U4 → Q1 → U3. Similarly, Q2 reads consistent data with the serialization order U3 → Q2 → U4. Thus, the execution is view consistency. We can easily see that Q1 and Q2 perceive a different serialization order of U3 and U4.

**View consistency** stronger requirement than consistency because not all consistent states can be reached by serializable execution of update transactions. On the other hand, view consistency is a weaker requirement than strong consistency because ROTS are not required to observe the same serialization order of update transactions. Although view consistency is a weaker requirement than strong consistency, it might be acceptable to some real-time applications.

3. Analysis of Consistency Requirement

We assume that transactions are not allowed to read a data version created by an uncommitted transaction. In order to observe a consistent database state, as required by view consistency, a ROT must not see the partial effects of any update transaction. For each update transaction, the ROT must see all or none of its effects either directly or indirectly. To avoid reading partial effects of an update transaction directly, a
ROT cannot read data items updated by the update transaction which has read some data item already read by the ROT.

To help describing the analysis of a ROT reading partial-effects of an update transaction indirectly, we define a read-from graph (RFG) which captures the relationships between a ROT and its associated update transactions. Each RFG is associated with a ROT. A RFG has a set of nodes N = Q ∪ U, where Q denotes the ROT in question and U denotes the set of update transactions that may affect the database state read by the ROT. A RFG has two types of edges: directed edges and undirected edge. An undirected edge represents the ROT reading some data from an update transaction in RFG. A directed edge represents an update transaction reading some data from another update transaction in RFG. We say that a transaction T₁ reads from another transaction T₂ if T₁ reads a data item which is written by T₂, i.e., RS(T₁) ∩ WS(T₂) ≠ ∅ where RS(T₁) and WS(T₂) denote the read and write sets of transaction T₁ (U or Q). Hence, the edges (directed or undirected) capture the read-from relationships among transactions in a RFG.

![Figure 2: RFG of Q₁](image)

**Example 3:** Consider the RFG in Figure 2. A ROT Q₁ reads data items which are updated by a set of transactions, U₂, ..., U₇. Note that although Q₁ does not read from U₅, U₆ is also included in the graph because U₅ may affect the database state read by Q₁. Before we go into the analysis, we need to define some additional concepts. If there is a cycle in the graph, starting from Q₁ along update transactions back to Q₁, we refer to it a RF-cycle, e.g., (Q₁ → U₁ → U₃ → U₅ → U₇ → Q₁). A RFG may have more than one RF-cycle. Figure 3 shows all the RF-cycles in the RFG of Q₁. We call the first update transaction in a RF-cycle primary-read (PR) transaction of Q₁, denoted by PR(Q₁), e.g., U₂, U₃ and U₆. Those update transactions other than the PR transaction in a RF-cycle are called dependency transactions of the RF-cycle, denoted by DT(Q₁), e.g., U₂, U₃, U₅, U₆. Note that a transaction which is a PR transaction in a RF-cycle may also be a dependency transaction of another RF-cycle such as U₆. We use PRED(U₆) to denote the set of immediate predecessors of dependency transactions in all RF-cycles involving U₆. For example, PRED(U₆) = {U₅}. Note that a dependency transaction may have different immediate predecessors in different RF-cycles.

We can see that a RF-cycle captures the read-from relationship, either directly or indirectly, between a PR transaction and one dependency transaction. can be stated that the dependency transaction may carry partial effects of the PR transaction. However, the PR transaction will not carry partial effects of other transactions which affects the consistency of data read by Q₁. A PR transaction may only carry its own partial effects. We can also observe that two update transactions in different RF-cycles, having no read-from relationship, will not carry the partial effect from one another.

![Figure 3: RF-cycles of RFG of Q₁](image)

Hence, a ROT may read the partial effects of a PR transaction indirectly via dependency transactions. In other words, a ROT may read an inconsistent data from dependency transactions. For example, Q₁ may see the partial effects of U₂ via U₁. Consider the following schedule:

S₃ : r₁(x₀) r₃(x₀) w₁(x₂) c₂ r₁(x₁) w₁(y₁) c₁ r₁(y₁) c₁

where the initial version of data item x is denoted by x₀. In this schedule, U₁ reads the version of x₂ from U₂ to derive y₁ which is, in turn, read by Q₁. It is obvious that Q₁ has read an inconsistent data because Q₁ sees the partial effects of U₂ via U₁ indirectly. This situation is characterized by having a cycle in the serialization graph. It can be easily seen that it is only U₂, the PR transaction, which may initiate a "partial-effect-chain" affecting the reads of Q₁ in the RF-cycle.

Based on the above analysis, a ROT will read inconsistent data if the resulting serialization graph has a cycle, which can be deduced from one of its RF-cycles. We can observe that for a ROT to read inconsistent data from the update transactions, the following three conditions must be true. First, the update transactions that the ROT reads are in a RF-cycle because only a RF-cycle may produce an execution schedule whose serialization graph has a cycle. Second, a ROT has read the prior versions of data items that one of its PR transactions will update and the PR transaction commits before the ROT because the "partial-effect-chain" may be initiated by the PR transaction. Third, the commit order of dependency transactions in a RF-cycle is consistent with the order of directed edge in the RF-cycle.

### 4. Processing Read-Only Transactions

In this section, we present two algorithms that try to let ROTs read the most recent and consistent data without interfering with update transactions. The two algorithms are based on the view consistency requirement of ROTs. The main difference between the two algorithms is the assumption of prior knowledge of read and write sets of transactions. Note that an underlying requirement of processing ROTs in the algorithms is that the execution of update transactions alone must be serializable, which is ensured by concurrency control protocols such as 2PL. Since these algorithms process ROTs separately from update transactions, they can be easily integrated with any concurrency control protocol.

With respect to a ROT, update transactions are divided into two subsets, the set of update transactions which should be placed before the ROT in the serialization order and the set of update transactions placed after the ROT. We call these two sets the `before set` and `after set` of the ROT respectively. Thus, a ROT always reads the most recent version of data items written by a transaction that belongs to its `before set`. Since
these two sets are complements of each other, it is not
necessary to maintain both sets for each ROT. Hence, we
only maintain the \textit{after set} of \textit{Q}, denoted by \textit{AS(Q)}.

Note that our primary objective is to let ROTs read the
most recent data items. We will therefore place an update
transaction in the \textit{after set} of \textit{ROT} only when the \textit{view}
consistency will be violated. A ROT always starts with an
empty \textit{after set}. Insertion of an update transaction into a \textit{ROT}
will only be carried out at the commit time of an update
transaction. For the recency requirement, if an algorithm is
able to place fewer update transactions into the \textit{after set} of
a ROT than another algorithm, it indicates that this algorithm is
more effective in allowing ROTs to read the recent version of
data.

Our first algorithm does not assume any knowledge of the
read and write sets of transactions. Let \textit{CRS(Q)} be the set of
data items that have been read by a ROT \textit{Q}.

\textbf{Algorithm 1 :} When an update transaction \textit{U} commits, it
will be placed into the \textit{after set} of a ROT \textit{Q}, if one of the following
conditions is true:

\begin{itemize}
  \item \textbf{C1 :} \textit{Q} has read a prior version of the data item that will be
    updated by \textit{U},
    i.e., \textit{WS(U) \cap CRS(Q) \neq \emptyset}.
  \item \textbf{C2 :} \textit{U} reads a data item that was created by another update
    transaction \textit{U'} and \textit{U'} is currently in the \textit{after set} of \textit{Q},
    i.e., \textit{\exists Q : RS(U) \cap WS(U') \neq \emptyset} and \textit{U' \in AS(Q)}.
\end{itemize}

Our second algorithm assumes that the read and write sets of
transactions are known in advance. With this knowledge, we
can determine the sets of the \textit{PR transactions} and of their
associated \textit{dependency transactions} and of immediate
predecessors of each \textit{dependency transaction} based on the
RFG analysis offline. Note that this algorithm does not need
to maintain RFGs in the processing of ROTs at runtime.
Hence, it is an efficient algorithm.

\textbf{Algorithm 2 :} When an update transaction \textit{U} commits, it
will be placed into the \textit{after set} of a ROT \textit{Q}, if one of the following
conditions is true:

\begin{itemize}
  \item \textbf{C1 :} \textit{U} is a \textit{PR transaction} of \textit{Q} and \textit{Q} has read a prior
    version of the data item that will be updated by \textit{U},
    i.e., \textit{U \in PR(Q)} and \textit{WS(U) \cap CRS(Q) \neq \emptyset}.
  \item \textbf{C2 :} \textit{U} has a predecessor that is currently in the \textit{after set}
    of \textit{Q},
    i.e., \textit{\exists Q : U' \in PRED(U) and U' \in AS(Q)}.
\end{itemize}

The purpose of \textbf{C1} in both algorithms is to detect the
possible occurrence of the “partial-effect-chain” initiated by the
committed update transactions. As we have discussed, it is
only the \textit{PR transaction} that will initiate this “partial-effect-
chain”. Since Algorithm 2 is based on the analysis of RFG of a
ROT, it can precisely detect the possible occurrence of the
“partial-effect-chain”. On the other hand, Algorithm 1 is more
conservative by putting more committed transactions into the
\textit{after set} of a ROT than necessary.

The purpose of \textbf{C2} in both algorithms is to avoid the
development of the “partial-effect-chain” between the
\textit{dependency transactions} along the directed edges in a RF-cycle
so that a ROT will not read inconsistent data from \textit{dependency
transactions}. Again, Algorithm 2 can precisely record the
exact development of the “partial-effect-chain” among the
\textit{dependency transactions} because it exploits the knowledge of
the order of read-from relationships between \textit{dependency
transactions} by using the sets of immediate predecessors of
each \textit{dependency transaction}. This exact recording of the
“partial-effect-chain” development cannot be done by
Algorithm 1 because it has no such information.

5. Conclusions

We have identified the various requirements for ROTs:
\textit{consistency, strong consistency, view consistency}. Particularly,
we have defined the weaker form of \textit{view consistency} for
processing ROTs, which allows non-serializable execution of
transactions. However, ROTs are still ensured to see consistent
data. We have also investigated the conditions under which a
ROT may read inconsistent data, based on the RFG analysis.
We have presented two algorithms to process ROTs separately
under the \textit{view consistency} requirement. The two algorithms
are based on different assumptions of prior knowledge of read
and write sets of transactions. The main objective of the two
algorithms is to address the recency requirement for ROTs,
which could be important in some real-time applications. We
are now developing another set of algorithms which process
ROTs requiring \textit{strong consistency}. These algorithms should
be more effective in allowing ROTs to read recent data than
MV2PL.

Transactions may have temporal consistency requirements
which may be absolute and relative temporal consistency [6].
Recency requirement is closely related to absolute temporal
consistency. In fact, our algorithms are able to provide more
flexibility in choosing which versions of data should be read,
thus helpful in satisfying temporal consistency requirement of
transactions. We will also pursue our work in this direction.

References

[1] Bernstein, P.A., V. Hadzilacos, and N. Goodman,
"Concurrency Control and Recovery in Database
Transactions”, \textit{IEEE Transactions on Software
Transactions in a Distributed Database”, \textit{ACM
Transactions on Database Systems}, 7(2), 1982.
Scheduling in Firm Real-time Database Systems,"
[5] Huang, J., J. A. Stankovic, K. Ramamritham, and D.
Towley, "Experimental Evaluation of Real-Time
Optimistic Concurrency Control Schemes," \textit{VLDB'91}.
Real-Time Database Systems”, \textit{IEEE RTAS’96}.
Concurrency Control of Real-Time Data Intensive
Applications”, \textit{IEEE RTSS’92}.
Synchronization Scheme for Distributed Real-Time