software AG

Tamino

Transactions Guide

Version 10.1

April 2018

WEBMETHODS

This document applies to Tamino Version 10.1 and all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: INS-TRANSACT-101-20180413

Table of Contents

Preface

This document helps you to choose the correct transactional options for your Tamino applications in order to gain the required degree of concurrency. This just means finding the right balance between data consistency requirements and the performance of your application. It describes typical scenarios (sometimes including code examples in various programming languages), but no extensive documentation of implementation details will be given here. Refer to the reference documentation for the *API* of your choice and also to the *X-Machine Programming* documentation for such information.

This document is not a general introduction into transaction management, locking, isolation levels, etc. It assumes you are already familiar with the fundamental concepts of transactionality in databases and explains the way Tamino provides transactionality. It only provides a small introduction into the general terms used and additionally it presents some useful links to other (partly non-Software AG and non-Tamino) documentation on this subject.

This document addresses application designers and experienced application developers who intend to organize database access to Tamino by their applications in the most efficient manner.

The following topics are discussed in this document:

Introduction	The introduction briefly summarizes and clarifies the fundamentals of transaction processing in database management systems. It also introduces the most important terms.
Transaction Modes	This section explains the different transaction modes which can be applied in Tamino applications.
Isolation Levels and Locking	This section explains the different isolation levels which can be applied in Tamino applications.
Transaction Parameters	This section explains the different locking parameters which can be applied in Tamino applications.
Deadlocks and Deadlock Prevention	This section discusses the occurrence of deadlock situations and suitable precautions for avoiding them.
Examples and Scenarios	This section provides in-depth knowledge about consequences of choosing various combinations of transaction modes, isolation levels and locking modes. Typical scenarios are described in this section, partly with accompanying code examples.
Appendix 1 - Complete Java Code example of Scenario 1	This section provides the complete source code of scenario 1 in Java.

1 Introduction

Transactionality is a characteristic feature of nearly all DBMSs on the market. Tamino supports transactionality in its core and its programming interfaces. This chapter gives an introduction to the elementary concepts of transactionality that apply to Tamino as well as to most other databases.

It is divided in the following sections summarizing the fundamentals of local and global transactions:

General Transaction Concepts

Tamino applications often model real-life processes, in which business objects are accessed and modified. These modifications must be done in a consistent way, especially if another application stored the data or the data may be accessed concurrently by other applications. Consequently, dividing an application into a set of consistent parts (i.e. logical units of work) becomes an essential design requirement.

From a logical point of view, a *transaction* represents the smallest unit of work (as defined by the user) that must be performed in its entirety to ensure logical consistency of the information contained within the database. A transaction may comprise one or more Tamino commands that together perform the database operations required to complete a logical unit of work.

One criterion for categorizing transactions is whether they affect a single system only or multiple systems (in a network). In the first case they are classified as local transactions, otherwise they are classified as distributed (or global) transactions.

To introduce other aspects of transactions, let us just have a look at a simple example:

Imagine, for instance, the transfer of an amount of money (let us say, 50 $) from one bank account A to another bank account B. Then, the corresponding transaction would have to consist of two elementary operations:

- The account A must be reduced by 50 $
- and 50 $ must be added to the account B.

In this context, neither operation makes sense alone.

ACID transactions

In order to be logical units of work in the sense just defined, the transactions are required to have specific properties. These properties are generally known in database theory by the acronym “ACID” (denoting:

atomic
consistent
isolated
durable

which is defined as follows:

Atomicity
Either the transaction is fully completed or it is not executed at all;

Note: As a consequence of this fact, all actions performed by an atomic transaction will be undone in case of any failure or interruption.

Consistency
The transaction always has to provide reliable results:

Starting from a consistent state, a transaction transforms a database into another consistent state.

Isolation
The transaction is independent of any other process that may be run in parallel; other transactions may not be influenced by intermediate results of a transaction.

Durability
Once the transaction is completed, the results remain as permanent data. They are persistent so that they will never be lost. They should not even get lost in case of a catastrophe.

The main problem of concurrent database access is this:

Operations on documents temporarily create inconsistent database content. This applies both to the document itself and also to the index data the database creates internally. In order to achieve consistency, in an ideal database it would be required that applications or queries must never see these inconsistent states of documents and index data. As a consequence, query and update processing must be synchronized in such a database based on the following rules:

1. It is necessary to ensure that no document will be updated while it is used in query processing.
2. In many cases it is necessary to ensure that no document will be accessed from query processing while it is under update (but Tamino does not generally suppress such dirty reads).

A transaction T is considered as isolated from other transactions if the following conditions are met:

1. Data written by T's write operations are neither read nor written by other transactions until the end of transaction T.
2. T does not overwrite “dirty” (uncommitted) data.
3. T does not read “dirty” (uncommitted) data from other transactions.
4. Other transactions do not write data read by T before T completes.

Although in reality transactions run concurrently, i.e. in a parallel manner, a database management system should ideally behave as if the transactions would be processed sequentially. This goal can practically be achieved, but only by paying the price and massively applying locks leading to a reduced degree of concurrency.

Note: In Tamino, synchronization of transactions is implemented based on locking.

Therefore nearly all databases offer some possibilities to correctly adjust the balance between the requirements of isolation and consistency on one hand and a reduced degree of *locking* with its less massive impact on transaction throughput on the other hand depending on your needs.

Commit and Roll back

An active transaction either has to be completed successfully (“committed”) or terminated when unsuccessful (“rolled back”). In case the latter applies, all objects modified by the rolled back transaction are reset to their correct prior status. The main task of the application designer is therefore to first determine which logical units of work exist for the application. The actual composition of each logical unit of work depends on the application's design and is directly related to the business processes to be supported by the application.

A *commit* command must be issued at the end of each transaction to complete it. Successful execution of a commit command ensures that all the additions, updates, and/or deletes performed during the completed transaction are physically applied to the database.

Updates performed during transactions for which a commit command has not been successfully executed, are not yet permanent. Transactions which have not already been committed can be made ineffective with the “roll back” command.

Local Transactions

A local transaction is defined to be a transaction which affects only one database. A local transaction is managed directly by the database. A client starts and terminates a local transaction by using database specific commands.

- **Transaction start**
  If a previous transaction was terminated, a new transaction will be started implicitly.
- **Transaction termination**
  The transaction is terminated either
  - **explicitly**
    by one of the following:
    - via a `commit` command
    - via a `rollback` command
  - **or implicitly**
    by one of the following:
    - An implicit commit has been performed by auto-commit
    - An implicit commit has been forced by disconnect

- An implicit rollback has been caused by a transaction failure such as deadlock, time-out or something similar.

Global Transactions

This section discusses the difference between distributed transactions in a multi-server-environment and local transactions in a single server environment. It describes the following aspects of global transactions:

- Global vs. Local Transactions
- The Two-Phase-Commit Logic

Note: The terms *Global Transaction* and *Distributed Transaction* are used as synonyms in this document.

Global vs. Local Transactions

Where a local transaction involves only one database, a global transaction encompasses operations to two or more databases.

If more than one database or even more than one machine gets involved, the situation gets more complicated as you can no longer assume that the criteria for *ACID transactions* are fulfilled for the global transaction even if each single database system fulfills them.

Let us for instance come back to our bank account example from above and assume that in contrast to the given example the amount of 50 $ is not transferred between two accounts that are available on the same database system, but between two accounts on two different database systems (for instance, on a database system on another computer in another town). Even if we assume that both database systems fulfill the ACID criteria from a stand-alone point of view it is possible that the system on which account A is reduced by 50 $ commits this process and the other system on which account B is increased by 50 $ rolls back this process and that neither process is aware of the existence of the other. The result of this would of course be a violation of consistency. For a solution of this problem communication between the participating database systems is required ensuring the ACID properties.

Typically, this is achieved by introducing a two-phase-commit protocol for the communication between the participating databases.

The Two-Phase-Commit Logic

In a global transaction environment, each participating database is only capable of ensuring the ACID properties for the operations within the global transaction that are issued on their own data. None of the participating databases has the overview over the complete transaction; therefore none of them is capable of ensuring the consistency of the global transaction as a whole. What is even worse, is the fact that if each participating database would be allowed to decide on its own the situation could arise that one database decides to commit its sub-transaction whereas the other rolls its sub-transaction back as described in the above example.

In order to provide ACID properties for a global transaction two things are required:

- There needs to be communication amongst the participating databases.
- Each of the participating databases must relinquish the final control over the outcome of their respective sub-transactions

The *Two Phase Commit Protocol* (*2PC*) was created to provide a solution for this problem. In the 2PC, an additional role is introduced, namely the transaction coordinator. A transaction coordinator is responsible for the coordination of the participating resource managers. A resource manager is any type of transactional system, mostly in the form of a database system. The transaction coordinator communicates with the participating resource managers, and each resource manager also communicates with the coordinator.

Another important difference between running a local or a global transaction is the fact that the application program no longer communicates with a resource manager to start/commit/roll back a transaction, but needs to do this via the coordinator. The coordinator in turn tells the resource managers what to do. The 2PC gets its name from the fact that a commit of a global transaction takes place in 2 phases:

1. **Phase 1: prepare**
   The first step is initiated by the application requesting the coordinator to commit a global transaction previously started through that coordinator. In the first step, the coordinator asks all participating resource managers to prepare their sub-transactions and waits for their answers. During this preparation phase each resource manager must bring itself in a position so that it can either complete its sub-transaction successfully or roll it back even in the case of a system crash. This normally involves that the resource managers write essential information to a log file. Once the coordinator receives an answer from all resource managers, it will also write some essential information to its own log. Once this is done, the first phase is finished.
2. **Phase 2: completion**
   The second phase, then merely consists of the coordinator asking all the resource managers to commit their sub-transactions.

   By splitting the committing of a transaction in two phases, the coordinator can guarantee that in the case of a failure, the state of a global transaction can be determined and the consistency of the data is maintained. If for example the system crashes somewhere during phase 2 (therefore after the coordinator received the answers of all the resource managers), the coordinator can

inform the resource managers during a system restart to commit their sub-transactions after all, thus consistently completing the global transaction.

2 Transaction Modes

This section introduces and explains the most important terms of transactionality in the context of Tamino, compares the general transaction modes and gives general information about the *locking* concept which implements Tamino's *transaction* support. In detail, the following topics are discussed here:

Basic Definitions

Before the transaction modes of Tamino will be explained in detail, the main terms used in the description of the granularity of transaction execution need to be defined exactly.

Session

As with most of the other databases, local transactions within Tamino always take place within the context of a *session*. A session manages the connection from an application to the Tamino database. You can consider sessions as a channel between your application and the database that is opened at the beginning of the session and closed at the end of the session. In Tamino, a session is opened with a `_connect` command and closed with a `_disconnect` command. Within the context of the session, a series of single transactions can be performed until the termination of the session.

Note: For distributed transactions, there is a many-to-many relationship between sessions and transactions: Several sessions can belong to one single distributed transaction.

Transaction

As mentioned above in the description of the ACID criteria, a transaction represents the unit of work (as defined by the user) that must be performed in its entirety to ensure logical consistency of the information contained within the database. Within a session, there is at most one transaction at a time.

In Tamino, a local transaction always begins with the first command acquiring a lock following a `_connect`, `_commit`, or `_rollback` command (in the 2 last cases the transaction is started implicitly) and ends when a `_commit`,`_rollback` or `_disconnect` command is issued for the transaction.

Note: Every read, insert, update, delete, define or undefine operation opens a transaction because each of these commands requests locks.

Request

In general, a request can be either a single command or a group of related commands.

Currently all Tamino APIs only support requests consisting of one single command, the only way to bundle more than one command in a single request is to access Tamino directly via its *HTTP* protocol. The terms “request” and “command” can therefore be used synonymously in most cases. In the examples on the next pages the distinction between “request” and “command” is maintained only for reasons of completeness.

Command

A command is an elementary operation within the database. All Tamino commands are relevant in the context of transactionality. Also see the section *Transaction-Related Commands* of the *X-Machine Programming* manual.

An application can communicate with Tamino either directly on the protocol level via HTTP or through a programming language specific API.

Note: This document explains the general transactional concepts of Tamino independent of the chosen method of communication. Thus details of particular APIs are not discussed here (refer to the documentation of the particular API for those) and terms for the various operations will not always match precisely the term used in your API. Sometimes command names from *X-Machine Programming* or the APIs will be used as an example in this document. For all other APIs there is always a similar command available.

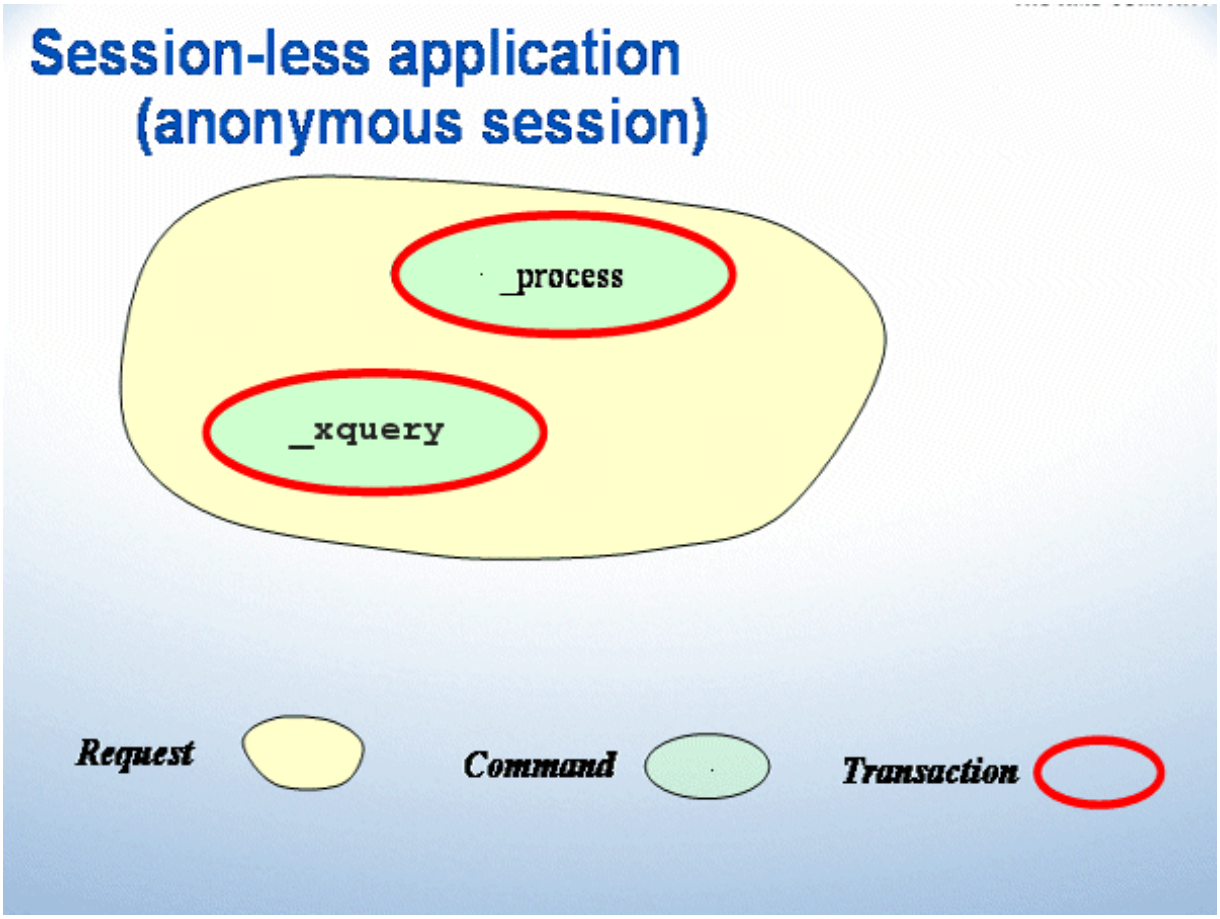
Commands are available for the following elementary operations:

- **Performing queries on data:**
  In *X-Machine Programming*: Use the `_xql` command for queries based on the query language Tamino *X-Query*. Use the `_xquery` command for queries based on the query language Tamino *XQuery* 4.
- **Processing data and storing it in the database:**
  In *X-Machine Programming*: Use the `_process` command to store (insert or update) data. You can also use the XQuery `update insert` command to insert data.
- **Updating data using XQuery**
  In *X-Machine Programming*: Use the XQuery `update replace` command to update data.
- **Deleting data:**
  In *X-Machine Programming*: Use the `_delete` command to delete data. You can also use the XQuery `update delete` command to delete data.
- **Creating new schema definitions:**
  In *X-Machine Programming*: Use the `_define` command to create new schema definitions.
- **Deleting schema definitions:**
  In *X-Machine Programming*: Use the `_undefine` command to delete schema definitions.

Summary of Transaction Modes

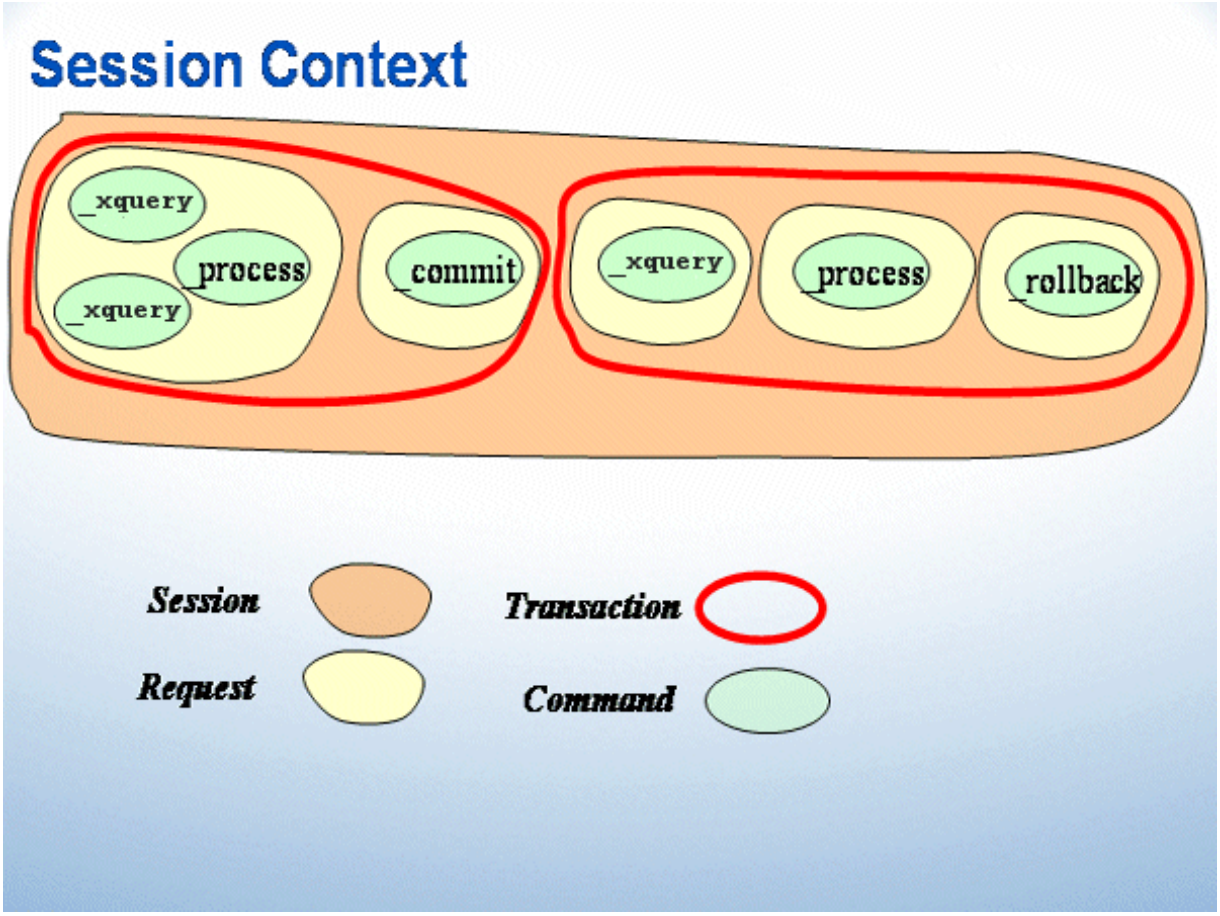
Tamino offers 2 transaction modes:

1. **session-less mode (auto-commit)**
   The session-less (or auto-commit) transaction mode is intended for executing local transactions with implicitly committed transactions in a session-less context.



   In a session-less context (also called an “anonymous session”), requests run without a user-defined session. Then every command in a request is executed in a separate (single-command) transaction and will be implicitly committed if the response code indicates successful execution or will be implicitly aborted (rolled back) otherwise.

2. **local transaction mode**
   The local transaction mode is intended for executing local transactions with explicitly committed transactions in a session context.



A session is established by a `connect` command. It can consist of one or more transactions containing one or more single commands that are performed during this session.

A transaction is implicitly started by the first command after a `connect` command or after the preceding transaction has completed.

A transaction is either explicitly closed by the application with a `rollback` or a `commit` command, or implicitly committed when the corresponding session is closed.

Note: Not every command starts a new transaction, only those commands that trigger a lock.

3 Isolation Levels and Locking

This section explains mechanisms used in Tamino for database access from concurrent *transactions*. It explains the terms used and discusses the available isolation levels and their effect on the various concurrency phenomena.

The topics described in this chapter are:

The Hierarchical Locking Concept of Tamino

Tamino's hierarchical locking concept is introduced here under the headings:

- Locking Granularities
- Lock Modes

Locking Granularities

Tamino makes use of a hierarchical locking system to implement isolation levels. A hierarchical locking system is characterized by the fact that locks may be set on different granularities. A lock set on a higher granularity implicitly sets the same lock on the lower granularities. The locking system in Tamino currently recognizes the following hierarchy of granularities:

1. database
2. collection
3. doctype
4. document

In practice, the highest level on which Tamino may set a lock is the level of a collection; a full database lock is never set.

Lock Modes

A locking system works with a set of different lock types, ranging from weaker to stronger locks. During its lifetime (normally within a transaction) a weaker lock set on an object may be escalated to a stronger lock.

In total, Tamino makes use of six lock types. They can be coarsely divided into two groups each consisting of three lock types. The first group contains lock types which affect objects directly; the second group contains lock types which represent an intention to lock further objects at a finer granularity.

The explicit lock types are:

1. **No lock**
   This mode represents the absence of any locks. In Tamino this mode is indicated by the term "unprotected".
2. **Share (S)**
   Provides the right to read the object at the granularity on which the lock is set or any object at a finer granularity and prevents concurrent transactions from holding IX, X or SIX locks. In Tamino this mode is indicated by the term "shared".
3. **eXclusive (X)**
   Provides the right to write the object at the granularity on which the lock is set as well as the right to set any lock on a finer granularity and prevents concurrent transactions from holding any locks on this or any finer granularity. In Tamino this mode is indicated by the term "protected".

The intention lock types are:

1. **Intention Share (IS)**
   Provides the right to set S or IS locks at a finer granularity and prevents concurrent transactions from holding X locks on this granularity.
2. **Intention eXclusive (IX).**
   Provides the right to set S, IS, X, IX or SIX locks at a finer granularity and prevents concurrent transactions from holding S, X or SIX locks on this granularity.
3. **Share and Intention eXclusive (SIX)**
   Provides the right to read the object at the granularity on which the lock is set or any object at a finer granularity and prevents concurrent transactions from holding S, IX, X or SIX locks. In addition it provides the right to set IX and X locks at a finer granularity. The SIX lock is essentially a union between lock types S and IX.

The following table summarizes the compatibilities of the above mentioned lock types.

	Already granted lock					
Requested lock	**No Lock**	**IS**	**IX**	**S**	**SIX**	**X**
IS	+	+	+	+	+	-
IX	+	+	+	-	-	-
S	+	+	-	+	-	-
SIX	+	+	-	-	-	-
X	+	-	-	-	-	-

Example

The following scenario gives an example how hierarchical locks are used.

Assume a transaction T1 in which all accounts that currently have a balance of more than 5000 $ need to be processed. This transaction is running in parallel to other transactions which add and subtract sums from accounts. The requirement for T1 is that it is guaranteed that all accounts with a balance of more than 5000 $ are found and also that no transfers are made that would possibly result in an account to drop below 5000 $ or increase to more than 5000 $ while T1 is executing. This can be achieved by T1 setting an "S" lock on the doctype in which all the account documents reside. All other parallel transactions that would like to make modifications to an account set an "IX" lock on the doctype (indicating the intent to set an "X" lock on one or more documents in the doctype). The "IX" and "S" locks on the doctype are incompatible. That means that T1 needs to wait until all existing "IX" locks have been relinquished, and once T1 has acquired the "S" lock, all other transactions need to wait for their "IX" lock until T1 has finished. In the following sections it will become clear that in Tamino this behavior is achieved by executing T1 in isolation level "serializable".

Concurrency Phenomena in DBMS

In databases (Tamino as well as relational databases), various possible scenarios can be considered that might occur in concurrent transaction processing. The chosen isolation level determines whether these phenomena may occur or not.

In the following, the most commonly known concurrency phenomena are described:

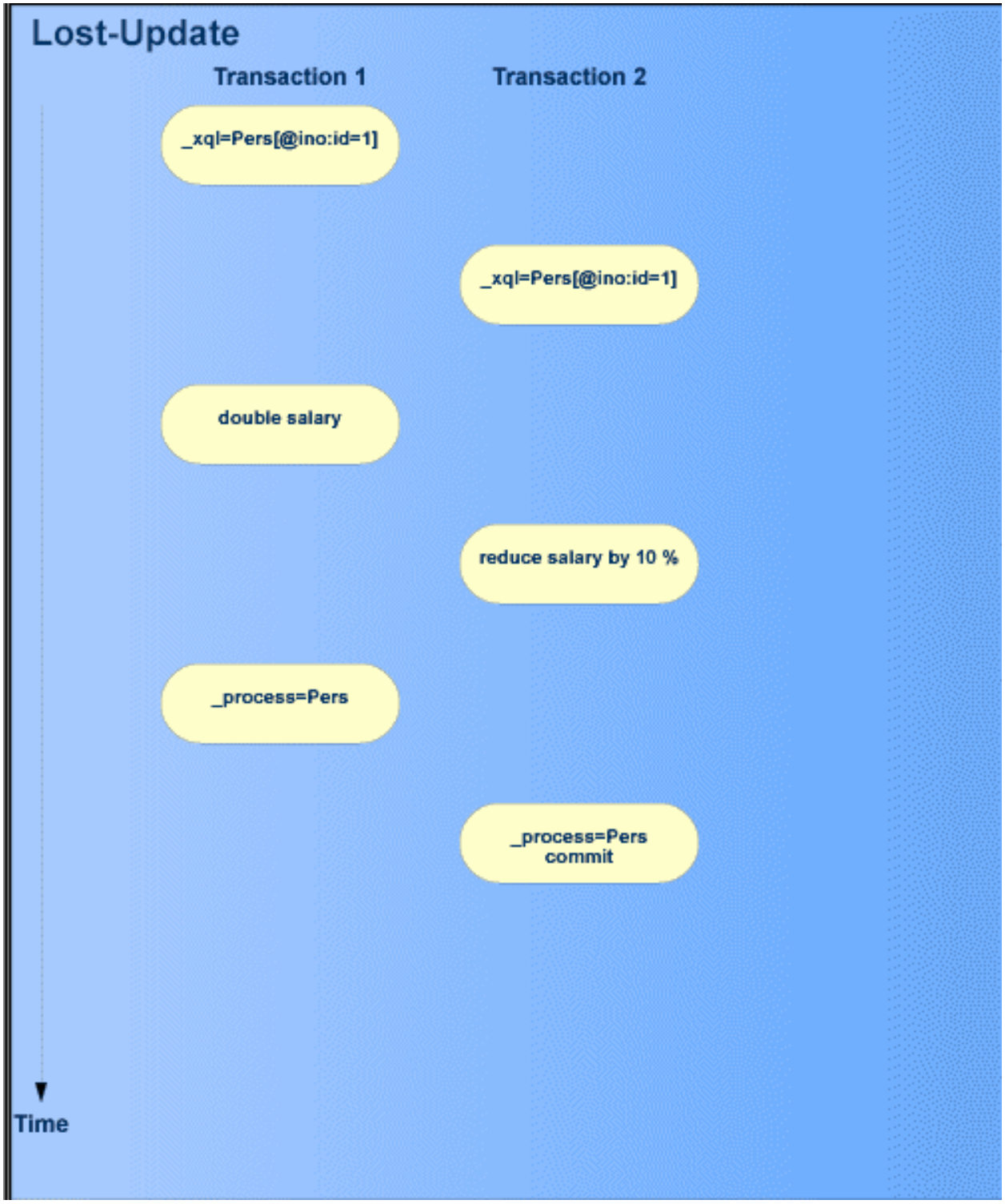
- Lost Updates
- Dirty Read Operation
- Non-repeatable Read Operation
- The Phantom Effect

Note: In the following examples mainly X-Query (`_xql`) syntax is used. In these cases, XQuery (`_xquery`) syntax can be used as well. The only reason for the usage of X-Query syntax is that the examples are shorter. The example for the phantom effect, however, is based on XQuery.

Lost Updates

The lost update phenomenon occurs when two or more concurrent transactions read and update the same data.

Lost updates represent a kind of write/write-dependency. The following situation will lead to the occurrence of lost updates:



Note: If you would try to perform a second update on one of the transactions, this will fail as all isolation levels provided by Tamino offer protection against lost updates.

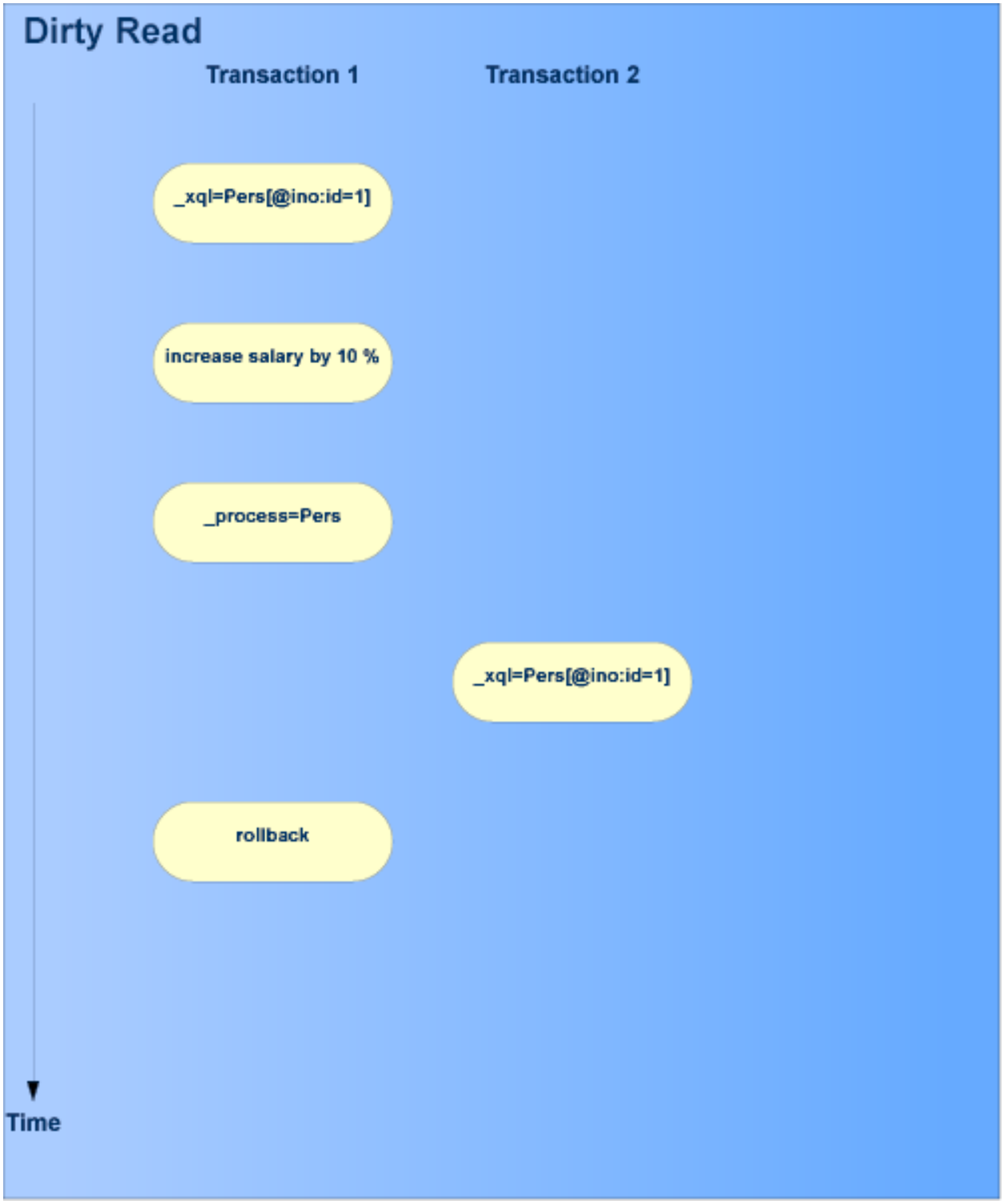
1. Transaction 1 reads a document.

2. Transaction 2 then reads exactly the same document.
3. Transaction 1 modifies the document.
4. Transaction 2 modifies the document.

Regardless of whether the commit is accepted or rejected, there would be one update lost in this situation!

Dirty Read Operation

The dirty read phenomenon may occur when a transaction modifies data and another transaction reads the same data before the first transaction is committed or rolled back.



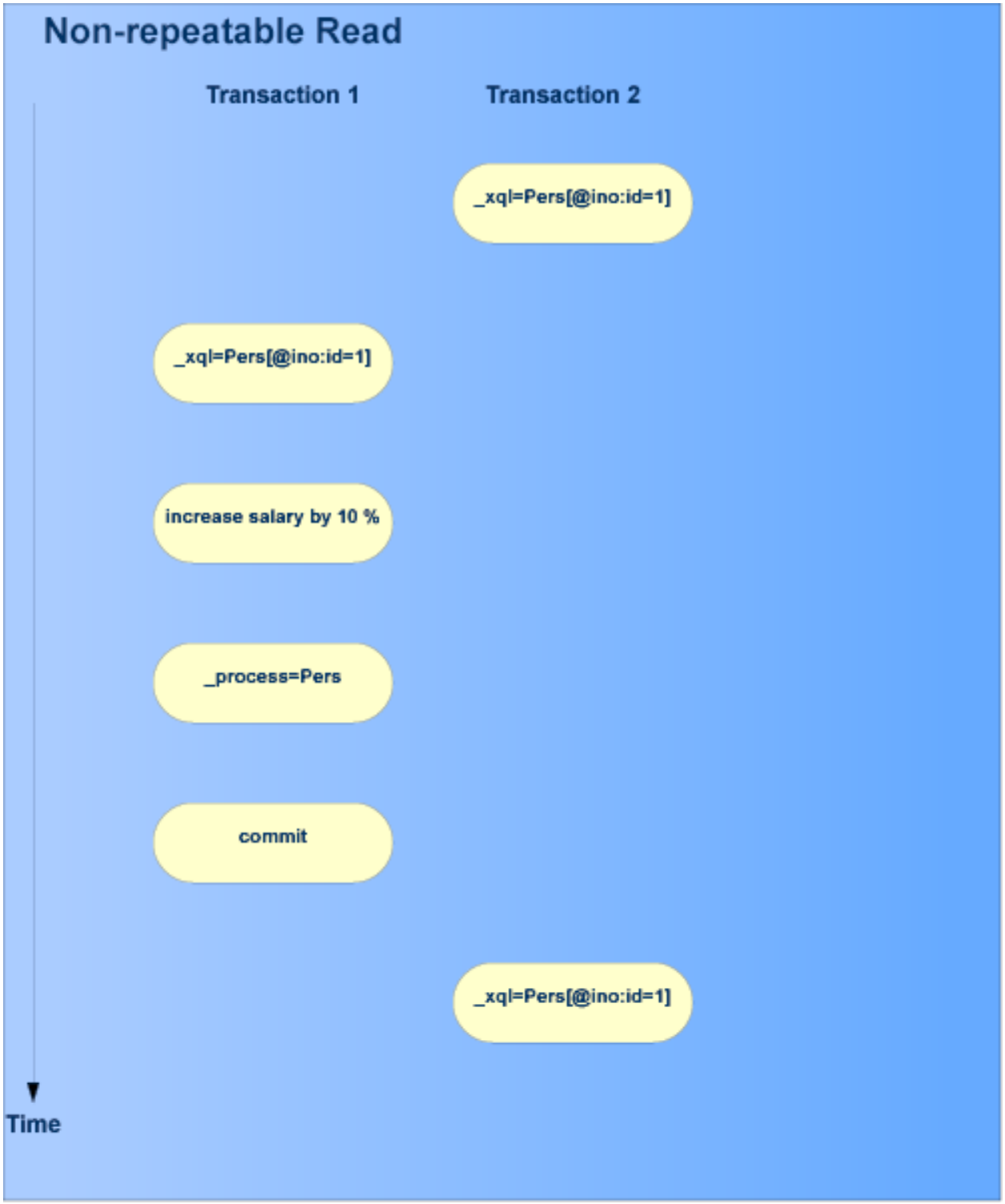
1. Transaction 1 reads a document.
2. Transaction 1 modifies the document.
3. Transaction 2 then reads exactly this modified document before a commit is performed on transaction 1.

4. Transaction 1 performs a roll back.

   The result of this scenario is that transaction 2 has read incorrect (“dirty”) data that has never been committed because transaction 1 has been rolled back afterwards.

Non-repeatable Read Operation

Non-repeatable read phenomena may occur when one transaction reads and modifies data and the same data is a read by a concurrent transaction multiple times:



The following situation will lead to the occurrence of such phenomena:

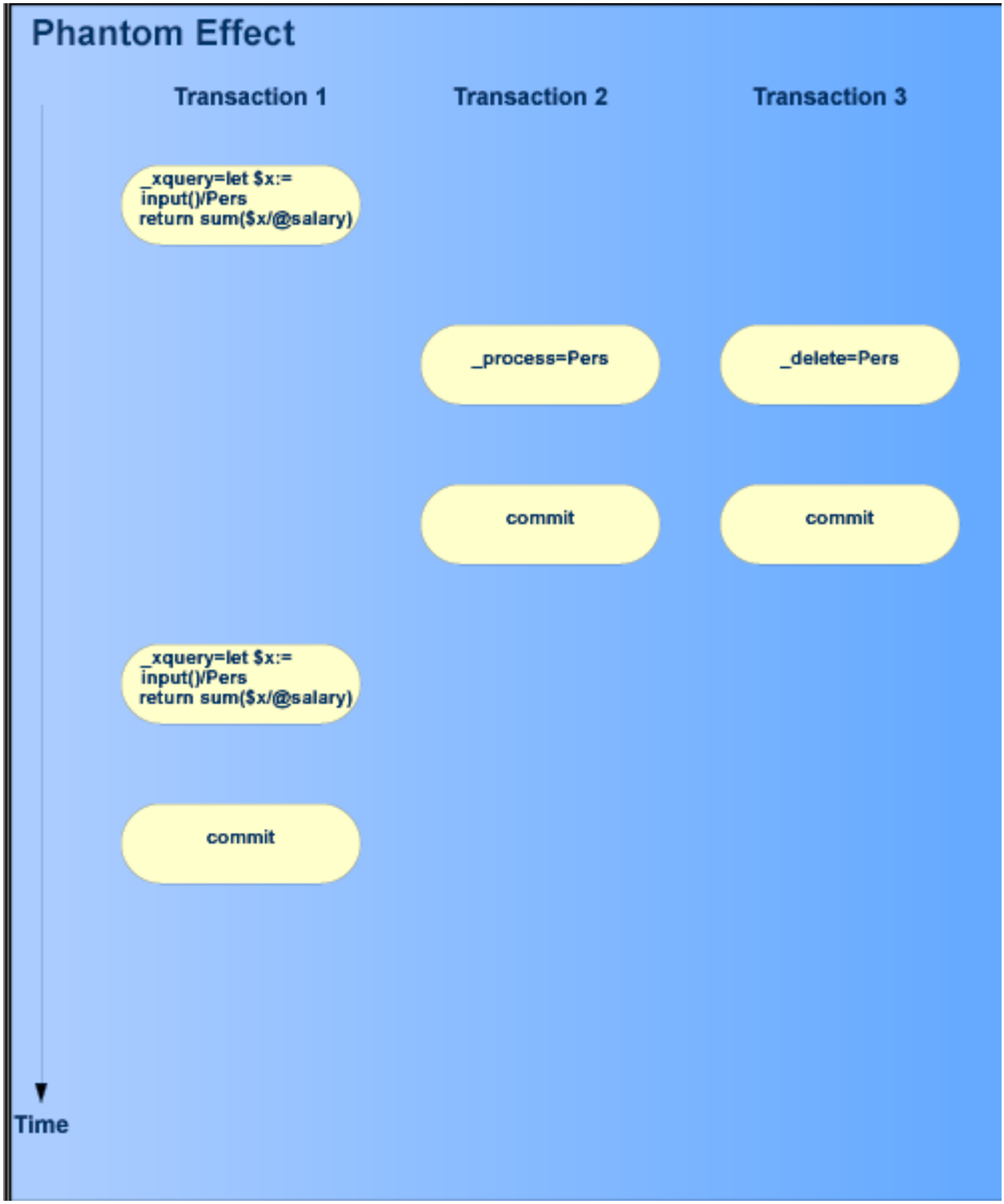
1. Transaction 2 reads a document.
2. Transaction 1 then reads exactly the same document.

3. Transaction 1 modifies the document.
4. Transaction 1 performs a commit.
5. Transaction 2 reads the same document again.

The result of the two identical read operations of transaction 2 will not be the same. This might lead to inconsistencies.

The Phantom Effect

The phantom phenomenon may occur when one transaction establishes a set of documents as the result of a query and other concurrent transactions insert or delete documents which would have been part of that set.



1. Transaction 1 reads some data that fulfills some search condition.
2. Transaction 2 inserts a document which would have been part of the result set calculated by transaction 1.
3. Transaction 3 deletes a document which was part of the result set calculated by transaction 1.

4. Both transaction 2 and transaction 3 commit.
5. Transaction 1 executes the same query. The results are different.

Isolation Levels in Tamino

As explained in the previous sections, a number of consistency phenomena may occur as soon as transactions can execute in parallel. In an ideal world, each transaction would be completely isolated from all other concurrently executed transactions and none of the above mentioned consistency phenomena would occur. It is clear however that this would require that to a large extent operations would have to be serialized. On the other hand not every application has the requirement that none of the consistency phenomena may occur. Some applications can live very well with a lower level of consistency.

The question is therefore always to find the right balance between the consistency requirements on the one hand and the maximization of parallelizing transactions on the other. For this purpose Tamino offers a number of isolation levels, each providing a different level of consistency. The isolation level of a transaction defines the degree of visibility of intermediate states of documents as produced by other concurrent transaction in the database. It is important to understand that the isolation level of a transaction does not determine to what degree intermediate states produced by this transaction are visible to other transactions.

Tamino supports the following isolation levels which are listed in order of ascending degree of isolation:

- **"uncommittedDocument"**
- **"committedCommand"**
- **"stableCursor"**
- **"stableDocument"**
- **"serializable"**

The default values are:

- In a session context, the default is "stableDocument".
- In a session-less context, the default is "uncommittedDocument", unless an *XQuery* `update` command is issued. In that case the isolation level is automatically upgraded to "committedCommand", the next stronger isolation level.

Important: The isolation level is a constant property of a transaction. It is determined at the start of each transaction and cannot be changed during the lifetime of that transaction.

In the following sections, each one of the isolation levels is described. The description of each isolation level includes a table which shows exactly which locks are set on the collection, doctype and document granularities for all relevant operations. The operations listed are:

- **query**
  These are either *X-Query* or XQuery queries which retrieve documents or parts thereof from the database or direct access via similar plain URL addressing.
- **process**
  These are `_process` commands which either insert a new document or update an existing one.
- **delete**
  These are `_delete` commands which delete existing documents from the database.
- **update(read)**
  These are the query parts of an XQuery update command.
- **update(write)**
  These are the update parts of an XQuery update command.
- **define**
  These are `_define` commands.
- **undefine**
  These are `_undefine` commands.

As the setting of the `lockMode` parameter also has an influence on the locking behaviour of Tamino, this is indicated in a separate column.

Note that the expression “sensitive XQuery cursor” is used to indicate a cursor over an XQuery query which is labeled to be “sensitive”, i.e. the parameter `_sensitive` has been set to the value "vague". This implies that the result of the query is not calculated at cursor open time but on the fly.

For more information see *The `_cursor` command* in *Requests using X-Machine Commands*.

Isolation Level uncommittedDocument

This value is the default value for the isolation level parameter if used in database operations in a session-less context.

In this isolation level the dirty-read situation can occur:

A command within a transaction with this isolation level can read a so-called “dirty” document at any time, which means that a concurrent transaction has changed the document but might abort later on.

Another possible situation leading to incorrect results:

The document content might be outdated in the sense that a concurrent transaction has changed the content after the current transaction has read it.

This isolation level only protects against lost updates and dirty writes, but not against any other of the above mentioned concurrency phenomena.

A command can also modify a document if no concurrent transaction is modifying the document, or no other transaction requires the document to be in a stable state (isolation level "stableCursor" and higher).

The characteristics of isolation level "uncommittedDocument" are:

- No lost updates.
- No dirty writes/updates.
- Dirty read is possible.
- Read any document at any time.
- Exclusive locks on documents persist until end of transaction.
- For queries using `_xql`: No shared locks.
- For queries using `_xquery`: No shared locks with insensitive cursors but shared lock with sensitive cursors.

The following table shows which kinds of locks are applied in isolation level "uncommittedDocument" depending on database operation and affected granularity:

Note: For queries the type of locks set depend on the chosen lock mode which is described later in this document. This lock mode does not have to be confused with the lock type.

	Lock Types for Isolation Level "uncommittedDocument"			
Action	**Lock Mode**	**Collection**	**Doctype**	**Document**
query	./.	IS	IS	*)
	unprotected	IS	IS	**)
	shared	IS	IS	S
	protected	IX	IX	X
process		IX	IX	X
delete		IX	IX	X
update (read)	***)	***)	***)	***)
update (write)	***)	***)	***)	***)
		X		
[up]define		X		
undefine				

*) S for sensitive XQuery cursors, no lock otherwise.

**) not allowed with sensitive XQuery cursors, no lock otherwise.
***) not applicable. The isolation level "uncommittedDocument" is not possible for XQuery `update` commands.

Isolation Level committedCommand

A command within a transaction in this isolation level can read documents that have been modified, inserted or updated by committed transactions but not documents that have been modified, inserted or updated by concurrent transactions. This means that the transaction only sees document states that have been committed but might have been changed by faster concurrent transactions.

The characteristics of isolation level "committedCommand" are:

- No lost updates.
- No dirty writes.
- Only read results of committed transactions (i.e. no dirty read is possible).
- Non-repeatable read is still possible.
- Exclusive locks on documents persist until end of transaction.
- Shared locks for query execution persisting for the duration of a command.

This isolation level protects against dirty reads, but not against non-repeatable reads.

The following table shows which kinds of locks are applied in isolation level "committedCommand" depending on database operation and affected granularity:

Note: For queries the type of locks set depend on the chosen lock mode which is described later in this document. This lock mode does not have to be confused with the lock type.

	Lock Types for Isolation Level "committedCommand"			
Action	**Lock Mode**	**Collection**	**Doctype**	**Document**
query	./.	IS	IS	S *)
	unprotected	IS	IS	**)
	shared	IS	IS	S
	protected	IX	IX	X
process		IX	IX	X
delete		IX	IX	X
update (read)	see query ***)	IX	IX	see query
update (write)	unchanged	unchanged	unchanged	X
[up]define		X		
undefine		X		

*) S for sensitive XQuery cursors. This S lock only exists temporarily. In all other cases no lock is held.
**) not allowed with sensitive XQuery cursors.
***) not allowed with lock mode "unprotected"

Isolation Level stableCursor

A transaction with this isolation level guarantees that a document in the cursor result set will not be changed by concurrent transactions (i.e. will still match the query predicate) in the following cases:

- In the case of a non-scrollable cursor: until the document has been returned to the requesting application and the document is no longer in the current fetch set of the cursor.
- In the case of a scrollable cursor: as long as the cursor exists.

The characteristics of isolation level "stableCursor" are:

- No lost updates.
- No dirty writes.
- Only read results of committed transactions (i.e. no dirty read is possible).
- Keep data in cursor stable until it is no more accessible.
- Exclusive locks on documents persist until end of transaction.
- Shared locks for query execution until data no more accessible in cursor.

The following table shows which kinds of locks are applied in isolation level "stableCursor" depending on database operation and affected granularity:

Note: For queries the type of locks set depend on the chosen lock mode which is described later in this document. This lock mode does not have to be confused with the lock type.

	Lock Types for Isolation Level "stableCursor"			
Action	**Lock Mode**	**Collection**	**Doctype**	**Document**
query	./.	IS	IS	S *)
	unprotected	IS	IS	**)
	shared	IS	IS	S
	protected	IX	IX	X
process		IX	IX	X
delete		IX	IX	X
update (read)	see query ***)	IX	IX	see query
update (write)	unchanged	unchanged	unchanged	X

	Lock Types for Isolation Level "stableCursor"			
Action	**Lock Mode**	**Collection**	**Doctype**	**Document**
[up]define		X		
undefine		X		

*) S for sensitive XQuery cursors

**) not allowed with sensitive XQuery cursors, no lock otherwise

***) not allowed with lock mode "unprotected"

Isolation Level stableDocument

This value is the default value for the isolation level parameter of transactions within a session context.

This isolation level guarantees that a document that has been read within the current transaction cannot be changed by a concurrent transaction until the end of the current transaction. Unqualified documents can be modified or deleted, new documents can be inserted.

The characteristics of isolation level "stableDocument" are:

- There are no lost updates.
- There are no dirty writes.
- Only read results of committed transactions can be obtained (i.e. no dirty read is possible).
- Keep documents stable until end of transaction (repeatable read).
- Exclusive locks on documents persist until end of transaction.
- Shared locks are used for query result until end of transaction.

This isolation level protects against non-repeatable reads, but not against phantom effects.

The following table shows which kinds of locks are applied in isolation level "stableDocument" depending on database operation and affected granularity:

Note: For queries the type of locks set depend on the chosen lock mode which is described later in this document. This lock mode does not have to be confused with the lock type.

	Lock Types for Isolation Level "stableDocument"			
Action	**Lock Mode**	**Collection**	**Doctype**	**Document**
query	./.	IS	IS	S
	unprotected	IS	IS	**)
	shared	IS	IS	S
	protected	IX	IX	X
process		IX	IX	X
delete		IX	IX	X
update (read)	see query ***)	IX	IX	see query
update (write)	unchanged	unchanged	unchanged	X
[up]define		X		
undefine		X		

**) not allowed with sensitive XQuery cursors, no lock otherwise

***) not allowed with lock mode "unprotected"

Isolation Level serializable

This isolation level guarantees the independence of the result set of a query from any influence by concurrent transactions. This means that the query result will always be the same.

Note: It is called "serializable" because the result is the same as if the execution of the transactions would have been serialized, meaning first transaction T1 would have been performed completely and afterwards transaction T2 would have been executed.

Isolation level "serializable" protects against all kinds of concurrency phenomena including phantom effects.

The characteristics of isolation level "serializable" are:

- There are no lost updates.
- There are no dirty writes.
- Repeatable read is guaranteed.
- Exclusive and shared locks on documents persist until end of transaction.
- Shared locks on a higher level apply not only on already existing documents, but also prevent input of new documents that influence query results.

The following table shows which kinds of locks are applied in isolation level "serializable" depending on database operation and affected granularity:

Note: For queries the type of locks set depend on the chosen lock mode which is described later in this document. This lock mode does not have to be confused with the lock type.

	Lock Types for Isolation Level "serializable"			
Action	**Lock Mode**	**Collection**	**Doctype**	**Document**
query	./.	IS	S *)	*)
	unprotected	IS	IS	**)
	shared	IS	S	
	protected	IX	SIX	X
process		IX	SIX	X
delete		IX	SIX	X
update (read)	see query ***)	IX	SIX	see query
update (write)	unchanged	unchanged	unchanged	X
[up]define		X		
undefine		X		
*) There is a special handling for `_xql=doctype[@ino:id=?]` : IS is applied on doctype level, S is applied on document level. The same applies for an equivalent XQuery.				
**) Not allowed with sensitive XQuery cursors, no lock otherwise				
***) Not allowed with lock mode "unprotected"				

Isolation Levels and Concurrency Phenomena

The Dependence between Concurrency Phenomena and Isolation Levels

The relationship between isolation levels and the types of concurrency phenomena which are allowed or forbidden for the individual isolation level can also be expressed in a table:

Isolation Levels and Concurrency Phenomena	Concurrency Phenomenon				
Isolation level	**Lost Update**	**Dirty Read**	**Dirty Write**	**Non-repeatable Read**	**Phantom Effect**
"uncommittedDocument"	No	Yes	No	Yes	Yes
"committedCommand"	No	No	No	Yes	Yes
"stableCursor"	No	No	No	Not in cursor	Yes
"stableDocument"	No	No	No	No	Yes
"serializable"	No	No	No	No	No

The table shows for each isolation level which concurrency phenomena are possible.

Lifetime of Locks

In order to be able to achieve a detailed understanding of locking it is essential to know how long a lock will live. In Tamino, the duration how long locks are held depends on:

- the type of the lock.
- and the chosen isolation level.

The following rules apply:

- **IS, IX and SIX locks**
  All intention locks (IS, IX and SIX) are kept for the duration of the transaction.
- **X locks**
  All X locks are also kept for the duration of a transaction.
- **S locks**
  In all cases where `_lockMode` is set to "shared", the S locks are kept until the end of the transaction.

  Otherwise, the amount of time that an S lock is kept depends on the isolation level:

Isolation Level	Lifetime of Locks
"uncommittedDocument"	no S locks are set. *)
"committedCommand"	S locks are kept for the duration of the command. *)
"stableCursor"	S locks are kept for the duration of the cursor.
"stableDocument"	S locks are kept for the duration of the transaction.
"serializable"	S locks are not explicitly set on the document granularity, S locks set on any coarser granularity are kept for the duration of the transaction.

Dependence of the Duration of Locks from Isolation Level

Note: *) except for sensitive XQuery cursors. In that case S locks are kept for the duration of the transaction. This may be subject of change, however.

General Rules for the Choice of the Isolation Level

The choice of the isolation level should be made on the basis of the following considerations:

1. The higher the isolation level, the lower the influences from other transactions. Setting the isolation level to the highest possible value **"serializable"** means in fact providing a virtual database to each transaction. You can consider a virtual database as a database where each transaction is totally isolated from effects of other transactions.
2. The higher the isolation level, the higher the potential impact on the performance. Performance may be deteriorated for instance by too many deadlock situations occurring between concurrent transactions or quasi-sequential execution of concurrent transactions caused by a too restrictive isolation level setting.
3. Keep in mind that isolation levels are technically based on internal locks which will affect more documents with increasing degree of isolation. For instance, increasing the isolation level can mean that not only one single document but its whole doctype is locked. Similarly, the time for which an object is locked may increase significantly by choosing a higher level of isolation.
4. It is important to understand that Tamino applies locks for query commands based on the index processing during the evaluation of the query. This means that depending on which indices are available and which of those are used to evaluate a query documents are locked. In other words it may well be the case that more documents are locked than those that are returned as the result of the query. Having appropriate indices defined is therefore not only important for the speed of queries but also minimizes the number of database objects that are affected by locks.

These considerations generally apply for all databases, they are not specific to Tamino.

The choice of the adequate isolation level means to find a reasonable compromise in such a way that you gain as much isolation, synchronization and performance as you need.

4 Transaction Parameters

The requirements for the level of data consistency (or transaction isolation) are determined by the application and may also differ for different types of transactions within the application. It is therefore the developer's responsibility to tell Tamino which kind of transactional behaviour is required in order to achieve the correct balance between the required level of concurrency and transaction isolation.

For this purpose Tamino provides the following parameters which may be manipulated by an application:

The isolationLevel Parameter

The `isolationLevel` parameter determines the level of isolation for a transaction and depending on the isolation level chosen it determines the types of locks that will be set on database objects for the duration of the transaction as described in ***Isolation Levels in Tamino***. In addition the locking behaviour of Tamino may by influenced by the `lockMode` parameter.

The default value of the `isolationLevel` parameter depends on the chosen session mode.

For more information see section *The _isolationLevel Parameter* of the *X-Machine Programming manual*.

The lockMode Parameter

The `lockMode` parameter overrides the default locking behaviour of Tamino as determined by the isolation level. This effect is also described in detail in ***Isolation Levels in Tamino***.

Overriding the default locking behaviour of Tamino influences the level of isolation achieved within a transaction and must therefore be used with care. In most cases, there is no need to explicitly set a different `lockMode`, the specified setting of the isolationLevel normally should provide the required functionality.

There are three possible options to choose:

1. "**shared**"
   In most isolation levels setting the `lockMode` parameter to the value "shared" might have only little or no effect.
2. "**unprotected**"
   Setting the `lockMode` parameter to "unprotected" weakens the locks set by the default behaviour determined by the chosen isolation level whereas setting the parameter to the value "protected" result in stronger locks.

3. "**protected**"
   Strengthening the locks by setting the `lockMode` parameter to "protected" can be used with all read commands. It raises the locks set on the documents retrieved to an X lock. This may be useful in situations where a document is read to be updated later in the transaction. By setting the X lock required for the update already at the time the document is read, the lock escalation from S to X is avoided and therefore the possibility of deadlock situation may be avoided.

Additionally, the following rules apply:

1. Weakening or strengthening locks for update commands has no effect in most cases.
2. There is no explicit default setting for the `lockMode` parameter, as Tamino's locking behaviour is determined by the chosen isolation level.

For more information, see section *The `_lockMode` Parameter* of the *X-Machine Programming manual*.

The querysearchmode Parameter

The optional parameter `querysearchmode` specifies whether a query will run in an atomic way, i.e. setting index locks until the request completes, or whether it is acceptable for concurrent requests to access and possibly modify the indexes while the query is running. Thus, the effect of this parameter is to specify to what extent the locking settings can be weakened during a dirty read (i.e. with _isolationLevel=uncommittedDocument/none or _lockMode=unprotected, but not _lockMode=shared or protected). The parameter can be used for a single query at a time, and cannot be set at a session-wide level. The parameter allows the following settings:

- **_querysearchmode=accurate**
  All indexes required by the query are locked. This is the default setting, and applies automatically if the parameter is not specified.
- **_querysearchmode=approximate**
  Indexes required by the query are not locked, but subsequent postprocessing will be performed if necessary.
- **_querysearchmode=nonserialized**
  Indexes required by the query are not locked, but additional postprocessing is omitted.

For more information, see sections *The _xql command* and *The _xquery command* of the *X-Machine Programming* manual.

The lockWait Parameter

With the lock wait mode parameter you can determine the strategy of how to proceed if a resource required for the *transaction* is locked. If the lock wait mode is set to "no", the transaction will be rolled back. Otherwise, if "yes" is chosen, the application will wait for the required resource to become available.

The default value of this option depends on the chosen session mode.

For more information see section *The* `_lockWait` *Parameter* of the *X-Machine Programming* manual.

Time Limits for Transactions

Tamino provides the possibility to specify a transaction time limit for applications or transaction users. It performs an automatic `rollback` operation on transactions that exceed the value of the specified limit. The following topics are discussed in this context:

The maximumTransactionDuration Parameter

The `maximumTransactionDuration` parameter sets the maximum duration of a transaction. If the time a transaction takes exceeds this limit, Tamino automatically rolls back the transaction, undoing all modifications done so far in that transaction and releasing all locks.

Notes:

1. The user session however remains opened even after the forced rollback.
2. This parameter was previously named `_transactionTimeout`.

The maximum duration of a transaction is important in case a transaction “hangs”. As long as a transaction is active, all locks it has acquired remain and may prevent concurrent transactions to proceed. The reason why a transaction does not terminate may be manifold.

One possible cause is that the application has been implemented in such a way that external resources are used within a transaction which may cause the transaction to wait. In general these situations are to be avoided. Another cause could be that a deadlock situation has arisen in which two or more transaction wait for each other to relinquish locks. Tamino detects deadlocks caused by locks inside of Tamino before a transaction timeout occurs. For deadlocks caused by resources outside of Tamino, a transaction timeout is the only way by which the deadlock can be resolved.

The first transaction that is rolled back due to a transaction timeout will relinquish its locks which may allow other transaction waiting for the release of these locks to continue.

It is recommended to set the `maximumTransactionDuration` parameter as low as possible so that Tamino is capable to release locks quickly in case a transaction does not terminate. Obviously, it is even more important to minimize the possibilities in which transactions do not terminate by avoiding using external resources which may cause delays within the boundaries of a transaction and by avoiding deadlock situations.

The default value for the `maximumTransactionDuration` parameter is set by the Tamino server property `maximum transaction duration`.

For more information see section *The _maximumTransactionDuration Parameter* of the *X-Machine Programming* manual.

The nonActivityTimeout Parameter

The `nonActivityTimeout` parameter sets the maximum time an active Tamino session can be idle. If an active Tamino session is idle for longer than this time, Tamino automatically rolls back any open transaction and closes the session. The effect is the same as if a `_rollback` and `_disconnect` is executed in that order. Any attempt by an application to use that session, will result in a Tamino response with an error code indicating that the session is no longer alive.

For more information see section *The _ nonActivityTimeout Parameter* of the *X-Machine Programming* manual.

Time Measurement

The time measurement for a transaction begins when the first database access (i.e. query, process, delete, create schema or delete schema commands) occurs after an initiating “connect” command or a previous `commit` or `rollback` command is executed, and ends when the program issues a `disconnect`, `commit`, or `rollback` command.

5 Deadlocks and Deadlock Prevention

This section discusses the following topics concerning deadlocks:

Deadlock Example

Deadlock situations can occur when concurrent transactions work on the same set of documents and attempt to set locks on these documents in a particular order. A typical deadlock situation occurs if two or more transactions are each waiting for a lock to be released held by one of the other transactions. If this occurs, Tamino usually detects this and chooses one transaction for rollback thus releasing all the locks it holds and resolving the deadlock. Finally, a transaction which is in a deadlock situation will receive a timeout on basis of the transaction timeout parameter, but normally a deadlock situation should already have been recognized by Tamino before the transaction is timed out. But also in a timeout situation, one transaction is rolled back releasing its locks and therefore resolving the deadlock situation.

Deadlock Example 1

1. Transaction 1 reads document 1 (setting an S lock)
2. Transaction 1 updates document 1 (setting an X lock)
3. Transaction 2 reads document 2 (setting an S lock)
4. Transaction 2 updates document 2 (setting an X lock)
5. Transaction 1 reads document 2 (attempting to set an S lock, resulting in waiting for the release of the X lock set by transaction 2)
6. Transaction 2 reads document 1 (attempting to set an S lock, resulting in waiting for the release of the X lock set by transaction 1)

Now both transactions wait for each other and a deadlock situation has arisen.

Deadlock Example 2

1. Transaction 1 reads document 1 (setting an S lock)
2. Transaction 2 reads document 1 (setting an S lock)
3. Transaction 1 updates document 1 (attempting to get an X lock)
4. Transaction 2 updates document 1 (attempting to get an X lock)

Now also a deadlock situation has arisen.

Deadlock Prevention

Although deadlock situations are detected by Tamino, and do normally not lead to excessive wait times, a deadlock still results in a rollback of a transaction. It is therefore better to avoid deadlock situations as much as possible. The most well known technique to avoid deadlocks is to determine a fixed order in which concurrent transactions acquire their locks.

6 Examples and Scenarios

This chapter covers the following topics:

Scenario 1: Read-only Scenario without Consistency Requirements

In this scenario documents are read from Tamino with the lowest possible consistency requirements. Documents may be read whose content may contain uncommitted changes of other concurrent *transactions*. This may lead to *dirty reads* and *non-repeatable reads* as described above. For an explanation of dirty reads and non-repeatable reads refer to section ***Concurrency Phenomena in DBMS*** of this document.

Cursor-free Approach

- If no cursoring is required, single read requests (direct reads or queries) may be issued in a session-less connection to Tamino without an explicit transaction context. Just choose the default isolation level for session-less context, namely **"uncommittedDocument"**. This avoids the cost of setting up and maintaining sessions, so this is the fastest alternative.

  This choice does not automatically suppress setting locks.

  An example *Java* application which only reads data efficiently in a session-less context without consistency requirements may for instance look like this example:

  - Open a Tamino connection in auto-commit mode by invoking the `newConnection` method of the class `TConnection` on an instance which had been created using `TConnectionFactory`'s method `getInstance`.

    ```
    TConnection connection = null;
    String taminoURI = "http://localhost/tamino/myDB";
    String collection = "encyclopedia";
    // Establish the Tamino connection (by default in auto-commit mode)
    connection = TConnectionFactory.getInstance().newConnection( taminoURI );
    ...
    ```

  - Create an appropriate accessor object by invoking the `newXMLObjectAccessor` method of `TConnection`.

    ```
    ...
    // Obtain a TXMLObjectAccessor with a DOM object model
    TXMLObjectAccessor accessor =
     connection.newXMLObjectAccessor(
      TAccessLocation.newInstance( collection ),
      TDOMObjectModel.getInstance() );
    ...
    ```

- You may additionally set the lock mode associated to this accessor object to the value "unprotected" in order to avoid any locks. In this manner the highest possible performance for read operations can be achieved.

```
...
// Set the lock mode to unprotected to avoid any locks
accessor.setLockMode(TLockMode.UNPROTECTED);
...
```

- Perform a read access using the class `TQuery` and display the result of the query.

```
...
// execute a query
TQuery query = ↩
TQuery.newInstance("/jazzMusician[./instrument=\"trumpet\"]/name/first");
TResponse response = accessor.query(query);
// display result
TXMLObject xmlObject = response.getFirstXMLObject();
System.out.println("result: "
+ ((Element)xmlObject.getElement()).getFirstChild().getNodeValue());
...
```

Or:

Cursor-based Approach

Alternatively, if cursoring is required you can implement this scenario also with explicit transactions. A *session* is therefore required. You need to explicitly specify the lowest possible isolation level "uncommittedDocument" to be used as it is not the default for session context.

Locks will still be set and cannot always be avoided:

- *X-Query* allows using the lockmode "unprotected".

Note: Dirty reads are possible for both XQuery and X-Query. However, for some join operations, XQuery (but not X-Query) might have to read a document more than once. If the document has been changed between the read operations (which is possible since the query is not using locks), the XQuery processing detects an inconsistent state and returns an error 6312.

For instance, an example Java application which only reads data efficiently in a session based context without consistency requirements could work in the following manner:

- It opens a Tamino connection similarly to the explanation given in the previous example (*Cursor-free Approach*).

- it switches this connection to session mode in order to work with explicit transactions.

```
...
// Establish a transaction context and therefore create a session
TLocalTransaction transaction = connection.useLocalTransactionMode();
...
```

  and creates a new session implicitly.

  These two steps are accomplished by invoking the `useLocalTransactionMode` method of the class `TConnection`.

- It sets the isolation level of this new session to the value "UNCOMMITTED_DOCUMENT" by invoking the `setIsolationDegree` method of the class `TConnection`.

```
...
// Set isolation level to UNCOMMITTED_DOCUMENT
connection.setIsolationDegree(TIsolationDegree.UNCOMMITTED_DOCUMENT);
...
```

- It creates an appropriate accessor object by invoking the `newXMLObjectAccessor` method of `TConnection`.
- It performs a read access using `TQuery` and displays the result.

```
...
// execute a query
TQuery query = ↩
TQuery.newInstance("/jazzMusician[./instrument=\"trumpet\"]/name/first");
TResponse response = accessor.query(query,5);
// display result
TXMLObjectIterator xmlObjectIterator = response.getXMLObjectIterator();
while (xmlObjectIterator.hasNext()) {
 TXMLObject xmlObject = xmlObjectIterator.next();
 System.out.println("result: "
+ ((Element)xmlObject.getElement()).getFirstChild().getNodeValue());
...
```

  The degree of *locking* is very low in all versions of this scenario. It is in fact very close to the minimum possible degree of locking at all. Processing with very high performance can therefore be expected.

This scenario is not adequate in the following situations:

- For data with high or medium consistency requirements.
- For data with frequent changes.

- If any changes to the data are required.

Important: This is the scenario with the lowest degree of consistency of all the scenarios discussed here.

Scenario 2: Read-only Scenario with Consistency Requirements

This scenario uses the isolation level **"committedCommand"** which protects against dirty reads, but not against non-repeatable reads.

The characteristics of this scenario are:

- No lost updates.
- No dirty writes.
- No dirty reads.
- Only results of committed transactions can be read.
- Non-repeatable read may still occur.
- Shared locks are used for query execution. They are released after command execution has finished.

Tip: This scenario is recommended for reading access to data with medium consistency requirements that does not change very frequently. The risk of non-repeatable read must not be high if this scenario shall be used.

This scenario is not recommended in the following situations:

- For data with high consistency requirements.
- For data with frequent changes.
- If any changes to the data are required.

Cursor-free Approach

- In case no cursoring is required, single read requests (direct reads or queries) can be issued in a session-less connection to Tamino without an explicit transaction context. Just choose the correct isolation level for session-less context, namely **"committedCommand"**. No sessions need to be maintained in this case.

  Transaction 1 only uses shared locks which are released after the read command is completely executed.

  For instance, a *Java* application which only reads already committed data in a session-less context with basic consistency requirements could look like this:

- First the *Java* application opens a Tamino connection either in auto-commit mode or in local mode depending on your specific requirements as already described in the previous scenario.
- It sets the isolation level of the connection object to the value "committedCommand" by invoking the `setIsolationDegree` method of the class `TConnection`.

```
...
// Set the isolation level to COMMITTED_COMMAND
connection.setIsolationDegree(TIsolationDegree.COMMITTED_COMMAND);
...
```

- It creates an appropriate accessor object by invoking the `newXMLObjectAccessor` method of `TConnection`.
- It performs a read access using the class `TQuery`.
- It displays the result of the query.

```
...
// display result
TXMLObject xmlObject = response.getFirstXMLObject();
System.out.println("result: " + ↩
((Element)xmlObject.getElement()).getFirstChild().getNodeValue());
...
```

Transactions belonging to this scenario will only read data which has been already committed by other transactions. In this scenario some locks are set that may lead to transactions having to wait for each other. But locks are only kept per command thus keeping the wait time low. Non-repeatable read situations may still occur but there will definitely be no dirty reads anymore.

Non-repeatable Read Operation

A non-repeatable read operation may happen in the following manner:

The following assumptions are made:

Transaction 1 is the transaction considered in this scenario using the isolation level "committed-Command". Transaction 2 can run on an arbitrary isolation level and use any possible locking mode.

1. Transaction 1 reads document x
2. Transaction 2 writes document x
3. Transaction 2 performs a `commit` operation.
4. Transaction 1 reads document x again.

Then transaction 1 performs a non-repeatable read operation.

The second read operation will be performed successfully, but the delivered result will be different from the result obtained in the first step. This is due to the fact that there was a `commit` operation performed between the two different read attempts, and the chosen isolation level is "committed-Command" which is equivalent to showing all committed results to other concurrent transactions. The locks set by transaction 1 are relinquished after the first read command completed.

Scenario 3: Single Insert/Blind Update Scenario

In this scenario either new documents are inserted or existing documents are updated regardless of the contents of the database. This means that all database access is done exclusively with the `_process` and no read access or updates via XQuery occur. Complete documents are replaced in the database without previously checking whether the document is already present or what its content is. This scenario represents the maximum of performance for writing access, but it is of course limited in its applicability.

Execute the `_process` commands in session-less context. Do not specify any isolation level or lock mode explicitly.

It is recommended to apply this scenario if you have to store large amounts of data independently of conditions.

This scenario cannot be applied if any contents of the database must be read prior to writing to the database.

For instance, a *Java* application which only reads already committed data in a session-less context with basic consistency requirements could look like this:

- First the *Java* application opens a Tamino connection in auto-commit mode as already described in the previous scenario.

```
TConnection connection = null;
String taminoURI = "http://localhost/tamino/myDB";
String collection = "encyclopedia";
// Establish the Tamino connection  in auto-commit mode
connection = TConnectionFactory.getInstance().newConnection( taminoURI );
...
```

- It does not set the isolation level or the lock mode explicitly.
- It creates an appropriate accessor object by invoking the `newXMLObjectAccessor` method of `TConnection`.
- It performs a sequence of `_process` commands in session-less context.

```
...
// process data
...
```

Scenario 4: Read-for-Update Scenario

In this scenario a document is first read and then changed afterwards. At the time the document is read it is clear (or at least likely) that an update of the document will follow. In order to prevent deadlock scenarios, the lock required for the update is already set at read time.

The occurrence of deadlock scenarios of the following type can be avoided if the lock mode is set to "protected" before reading the document:

T1		T2	
Read doc1	S on doc1		
		Read doc1	S on doc1
Update doc1	Wait for X on doc1		
		Update doc1	Wait for X on doc1
		Deadlock	

The following situation leads to the deadlock as both transactions wait for an X-lock being granted which will never occur. However, if the an X-lock is set already on reading document 1 (this is enforced by setting the lock mode to "protected" before reading the document) the following occurs:

T1		T2	
Read doc1	X on doc1		
		Read doc1	S on doc1 (will not be granted)
Update doc1	As X-lock is already on doc1		
Commit	Relinquish lock on doc1		

Thus the deadlock is resolved as transaction 1 can continue without waiting.

This scenario only makes sense if:

- Deadlock scenarios as described above exist.
- It is known or at least very likely that the document read will be updated in the same transaction at the time of the read.

If the conditions above are not met, this may lead to a large number of superfluous X locks. In that case it might be better to avoid possible deadlocks in another way.

Scenario 5: Stable Cursor Scenario

In this scenario the same document is read more than once in the same cursor. The content of the document is guaranteed to be identical.

In order to accomplish this, the isolation level must be set to **"stableCursor"**.

Stable Cursor Scenario - Cursor-based Approach

- Working with cursors requires a session context and therefore explicit transactions. The isolation level **"stableCursor"** needs to be specified explicitly as it is not the default for session context.

  Caution: In this scenario locks are kept for the duration of the cursor, thus keeping these locks for a potentially long period of time.

  - This choice can be considered to achieve a medium performance.

    For instance, an example Java application which only reads data in a session based context without consistency requirements could work in the following manner:

    - it opens a Tamino connection similarly to the explanation given in the previous examples.
    - it switches this connection to session mode in order to work with real transactions.

      ```
      ...
      // Establish a transaction context and therefore create a session
      TLocalTransaction transaction = connection.useLocalTransactionMode();
      ...
      ```

      and creates a new session implicitly.

      These two steps are accomplished by invoking the `useLocalTransactionMode` method of the class `TConnection`.

    - it sets the isolation level of this new session to the value "STABLE_CURSOR" by invoking the `setIsolationDegree` method of the class `TConnection`.

      ```
      ...
      // Set isolation level to STABLE_CURSOR
      connection.setIsolationDegree(TIsolationDegree.STABLE_CURSOR);
      ...
      ```

    - it creates an appropriate accessor object by invoking the `newXMLObjectAccessor` method of `TConnection`.

```
...
// Obtain a TXMLObjectAccessor with a DOM object model
TXMLObjectAccessor accessor =
 connection.newXMLObjectAccessor(
  TAccessLocation.newInstance( collection ),
  TDOMObjectModel.getInstance() );
...
```

- it finally accesses the data.

Scenario 6: Deadlock Prevention Scenario

Lock acquisition in a fixed document order can also be used as a means in order to prevent the occurrence of deadlocks.

A Appendix 1 - Complete Java Code example of Scenario 1

The following code represents a complete sample program written in Java and intended for use with the Tamino API for Java for scenario 1.

```
import java.io.*;

import com.softwareag.tamino.db.api.connection.*;
import com.softwareag.tamino.db.api.accessor.*;
import com.softwareag.tamino.db.api.objectModel.*;
import com.softwareag.tamino.db.api.response.*;
import com.softwareag.tamino.db.api.common.*;
import com.softwareag.tamino.db.api.objectModel.dom.TDOMObjectModel;
import org.w3c.dom.Element;

/**
 * Transaction scenario 01:
 * read only access without consistency requirements using a cursor
 */
public class TxScenario01b
{
 public static void main(String[] args) throws Exception
 {
  TConnection connection = null;
  String taminoURI = "http://localhost/tamino/myDB";
  String collection = "encyclopedia";
  try
  {
   // Establish the Tamino connection (by default in auto-commit mode)
   connection = TConnectionFactory.getInstance().newConnection( taminoURI );

   // Establish a transaction context and therefore create a session
   TLocalTransaction transaction = connection.useLocalTransactionMode();
   connection.setIsolationDegree(TIsolationDegree.UNCOMMITTED_DOCUMENT);

   // Obtain a TXMLObjectAccessor with a DOM object model
```

```
   TXMLObjectAccessor accessor =
    connection.newXMLObjectAccessor(
     TAccessLocation.newInstance( collection ),
     TDOMObjectModel.getInstance() );

   // Optionally set the lock mode to unprotected to avoid any locks
   accessor.setLockMode(TLockMode.UNPROTECTED);

   // execute a query
   TQuery query =
   TQuery.newInstance("/jazzMusician[./instrument=\"trumpet\"]/name/first");
   TResponse response = accessor.query(query,5);

   // display result
   TXMLObjectIterator xmlObjectIterator = response.getXMLObjectIterator();
   while (xmlObjectIterator.hasNext()) {
    TXMLObject xmlObject = xmlObjectIterator.next();
    System.out.println("result: "
+ ((Element)xmlObject.getElement()).getFirstChild().getNodeValue());
   }
  }
  catch (TException e)
  {
   System.err.println("Tamino Exception caught: " + e);
  }
  finally
  {
   if (connection != null) connection.close();
  }
 }
}
```

Index

U