The VLDB Journal
DOI 10.1007/s00778-013-0318-x

REGULAR PAPER

# Consistency anomalies in multi-tier architectures: automatic *detection* and *prevention*

**Kamal Zellag · Bettina Kemme**

Received: 5 July 2012 / Revised: 25 April 2013 / Accepted: 27 April 2013


**Abstract** Modern transaction systems, consisting of an application server tier and a database tier, offer several levels of isolation providing a trade-off between performance and consistency. While it is fairly well known how to identify *qualitatively* the anomalies that are possible under a certain isolation level, it is much more difficult to *detect* and *quantify* such anomalies during run-time of a given application. In this paper, we present a new approach to detect and quantify consistency anomalies for arbitrary multi-tier application running under any isolation levels ensuring at least read committed. In fact, the application can run even under a mixture of isolation levels. Our detection approach can be online or off-line and for each detected anomaly, we identify exactly the transactions and data items involved. Furthermore, we classify the detected anomalies into patterns showing the business methods involved as well as analyzing the types of cycles that occur. Our approach can help designers to either choose an isolation level where the anomalies do not occur or to change the transaction design to avoid the anomalies. Furthermore, we provide an option in which the occurrence of anomalies can be automatically reduced during run-time. To test the effectiveness and efficiency of our approach, we have conducted a set of experiments using a wide range of benchmarks.

**Keywords** Consistency · Serializability · Transactions

K. Zellag (✉) · B. Kemme
School of Computer Science, McGill University, Montreal, Canada
e-mail: zkamal@cs.mcgill.ca

B. Kemme
e-mail: kemme@cs.mcgill.ca

Published online: 04 June 2013

## 1 Introduction

Information systems, such as online banking and shopping, have become ubiquitous and part of our daily lives. As more and more clients concurrently access a service, it becomes increasingly difficult to provide users with a consistent view of the data and to guarantee that the actions of different users do not interfere with each other. This problem is aggravated through the use of multi-tier architectures as execution is now distributed and data are spread across several components. Most common is a multi-tier architecture, where clients (e.g., web browser) first send requests to a Web server, which processes the presentation logic or handles the interchange with the external clients. Then, the requests are passed to a middle-tier, usually an application server, which takes care of the business logic (e.g., performing a purchase) and accesses the database backend tier to manage persistent data. As many requests can execute concurrently at both the middle-tier and the database, some concurrency control is needed. Modern middle-tier systems often implement their own concurrency control mechanism on top of the one provided by the database system. For instance, a variation of optimistic concurrency control defined in java persistence API (JPA) has become very popular which offers more isolation than read committed but less than serializability [18]. The choice between these isolation levels depends on performance and consistency needs. Selecting an isolation level with strict consistency can result in a performance penalty or might have high operational cost [19], while lower levels of isolation provide more concurrency or might be cheaper, at the cost of potential inconsistencies (e.g., overselling due to concurrent orders).

Several studies have analyzed the potential anomalies associated with the existing isolation levels [2,4,5,9,12,13]. However, it is less clear whether a given application might actually experience such anomalies when running, and if yes,

to what degree. For example, well-known anomalies such as *lost update* and *unrepeatable read* are generally allowed under the isolation level *read committed* but the actual occurrence can vary widely depending on the application. Mechanisms presented in [15,17] detect whether an application, if run under snapshot isolation (SI), can potentially have anomalies and/or avoid such potential anomalies. However, their approach requires a careful analysis of the application which might not always be possible. And even if one can determine the anomalies that are possible, it is not yet clear *how often* they will actually happen in practice. Quantifying the amount of anomalies as they occur in near real time is extremely useful and desirable for practitioners [21] as it allows for early diagnosis and for immediate intervention, such as adapting the level of isolation.

In this paper, we propose a new approach for *detecting* consistency anomalies and automatically reducing their occurrence. Our approach is completely independent of the underlying database system which we treat as a black box. Instead, our system is (partially) embedded into the middle-tier. Furthermore, our approach does not require any knowledge about the studied application, and we do not need to analyze or change the application code in any form. Finally, our system detects anomalies independently of the isolation level under which the application runs, as long as all transactions run under an isolation level that is equal or higher to read committed. In fact, the application can run individual business methods under different isolation levels. Once the system detects that certain anomalies occur frequently, it provides the option to automatically increase the isolation level for some transaction types in order to avoid inconsistencies in the further execution. More concretely, this paper makes the following contributions.

### 1.1 Anomaly detection

We detect anomalies by using traditional serializability theory [7], that is, we build the dependency graph of the execution and detect cycles in it as anomalies. While serializability theory has been developed decades ago, dependency graphs have barely been used in practice, as many challenges have to be overcome.

First, detecting dependencies between transactions is not trivial. In our case, this is even more complex since internal database information is generally not exposed to the application layer at which our approach operates. We obtain the necessary dependency information by transparently tagging data items with additional information. Furthermore, we take advantage of certain properties of the isolation levels to derive dependency information. Section 3 discusses information extraction and dependency detection in detail.

The second challenge is to efficiently build the graph (see Sect. 4) and perform cycle detection for systems that process hundreds and thousands of transactions per minute. Off-the-shelf cycle detection algorithms have exponential run-time cost and require huge memory space to hold the graph. To make cycle detection feasible, we performed a thorough analysis of the properties of cycles in dependency graphs depending on the isolation level used. We then take advantage of these properties to reduce the number of edges we have to consider when looking for cycles. In fact, our algorithms for cycle detection, discussed in Sect. 5, only need to follow a fraction of edges in the graph. Furthermore, we show in Sect. 6 that at any time, we only need a subset of the graph in memory to be able to detect all cycles. This means, we can load the graph incrementally and remove nodes dynamically when we are sure that we have detected all cycles in which they might be involved.

### 1.2 Anomaly classification and reduction

For each cycle that we detect, we generate detailed information about the individual transactions involved and the data items that are affected. But this is only the core information. We also categorize cycles according to the type of anomaly they cause, such as lost update or unrepeatable reads. Furthermore, we determine the business methods that are involved in the cycle which tell exactly which parts in the application are responsible for generating such cycles. These two classification mechanisms are presented in Sect. 7.

If certain types of cycles occur frequently, then one option is to run the related business methods with a higher level of isolation reducing the amount of cycles that can occur. Thus, in Sect. 8, we present an option to transparently and dynamically increase the isolation level of business methods that lead to common cycles.

### 1.3 Deployment, implementation, and evaluation

We have deployed to modes for cycle detection: *off-line* and *online*. In the off-line processing mode, we let the application run for a given time interval, collect all the information, and then perform consistency analysis on the committed set of transactions. Off-line processing can be used for a periodic analysis of the execution or in the testing phase of a new application. In the online mode, we stream dependency information of a running application as we collect it, build, and extend the graph as new transactions commit and detect cycles in near real time. Online processing is useful if we want to continuously observe an application in order to intervene quickly if an inconsistency is found.

We have implemented our approach within two components. A collector agent COLAGENT observes the execution and keeps track of the data items accessed by transactions and their execution order. COLAGENT is designed to work with java enterprise edition (JavaEE) compliant application servers and is completely independent of the underly-

ing database system. We have integrated it into Hibernate,[1] a default persistence layer for the open-source application server JBoss.[2] COLAGENT works both for single server as well as clustered configurations that consist of several application server instances. COLAGENT sends the logs it creates to DETAGENT, the agent that builds the dependency graph, and detects cycles. DETAGENT is completely independent from the multi-tier architecture and can run on any machine.

These implementation aspects are described in Sect. 9.

We have tested the feasibility of our approach in Sect. 10, by using three benchmarks, RUBiS [3], SPECjEnterprise 2010 [26], and TPC-C [28] using various mixes of isolation levels: read committed, SI, and an optimistic concurrency control mechanism common in many middle-tier servers. We provide a detailed analysis of the number of anomalies that occur under these benchmarks, their classification, and how our prevention works. Furthermore, our analysis shows that COLAGENT's overhead during run-time is very low and cycle detection is efficient. To stress-test DETAGENT, we have emulated large graphs with hundreds of thousands of transactions and many large cycles, and can show that graph creation and cycle detection are efficient even with such huge graphs.

This paper is based on [31] where we presented an approach for online anomaly detection and classification along business methods and showed results based on the RUBiS benchmark. We have extended this work to include efficient cycle detection in a complete graph, to provide offline detection, to support mixture of isolation levels, to handle limited memory, to classify along traditional anomaly types, and to offer anomaly prevention. Furthermore, we conducted experiments with the SPECjEnterprise and TPC-C benchmarks and evaluated the efficiency of our cycle detection.

## 2 Background and preliminaries

In this section, we present our transaction model and give an overview of the current middle-tier technology and the concurrency control mechanisms they implement.

### 2.1 Transactions and isolation levels

A *transaction* $T_i$ is a logical unit of read and write operations on data items. $T_i$ runs atomically, i.e., either all operations succeed and $T_i$ commits, or none of its operations succeed and it aborts. Two transactions are *concurrent* if neither terminates (commit/abort) before the other starts. We denote with $s_i$ the start time and with $c_i$ the commit time of transaction $T_i$. Following [1], we assume that each writes operation $w_i(x)$ of transaction $T_i$ on data item $x$ generates a new version $x_i$ of $x$ which is *installed* when $T_i$ commits. If both transactions $T_i$ and $T_j$ update the same data item $x$, we require the commits of $T_i$ and $T_j$ to be ordered and this commit order defines an order on the created versions. For each read operation $r_j(x_i)$ of transaction $T_j$ on data item $x$, we indicate the version $x_i$ that is read.

Concurrent transactions must be isolated from each other. The main correctness criterion in the research community is serializability. It requires an interleaved execution of transactions to be conflict equivalent to a serial execution over the same set of transactions, where transactions execute serially one after the other and read operations always read the latest committed version. Equivalence means that in both executions transactions that update the same data items commit in the same order (i.e, both executions create versions in the same order), and in both executions, the read operation on data item $x$ of a transaction $T_i$ reads the same version $x_j$ of $x$. In short, pairs of conflicting operations (i.e., operations on the same data item where at least one is a write operation) are ordered in the same way in both executions.

A *dependency graph* is a directed graph where nodes are the committed transactions of an execution and edges represent conflicts between them. There is a $wr$-edge $T_i - wr \rightarrow T_j$ from $T_i$ to $T_j$ if $T_i$ creates version $x_i$ of $x$ and $T_j$ reads this version. There is a $ww$-edge $T_i - ww \rightarrow T_j$ from $T_i$ to $T_j$ if both $T_i$ and $T_j$ write $x$, and $T_j$ is the first to commit after $T_i$ and write $x$, that is, $T_i$ and $T_j$ install consecutive versions of $x$. At last, there is a $rw$-edge $T_i - rw \rightarrow T_j$ from $T_i$ to $T_j$ if there is an item $x$ for which $T_i$ has read the version $x_p$ (created by another transaction $T_p$) and later $T_j$ creates the immediate successor $x_j$ of $x_p$. The edges $wr$, $ww$, and $rw$ are known, respectively, as *read dependencies*, *write dependencies*, and *anti-dependencies*. An execution is serializable if and only if its dependency graph is acyclic [7].

Commercial database systems do not use this formal definition of serializability but use concurrency control mechanisms that offer lower levels of isolation, each of them avoiding a set of specific *anomalies*. In [4], such anomalies are more formally defined as undesirable behavior during the concurrent execution of two transactions $T_1$ and $T_2$. Here, we discuss the most common ones.

With *dirty writes*, $T_1$ writes an entity $x$ and then $T_2$ writes $x$ before $T_1$ has committed. If dirty writes are allowed, unserializable schedules can occur where $T_1$ writes $x$, then $T_2$ writes $x$ and $y$, and finally, $T_1$ writes $y$, leading to cycle with two $ww$-dependency edges in the dependency graph. Basically, all concurrency control mechanisms implemented in commercial systems avoid dirty writes, and we assume this for this paper.

With *dirty reads*, transaction $T_2$ can perform a read $r_2(x_1)$ before $T_1$ has committed. This can lead to problems when $T_1$ actually aborts and certain cycles. Most isolation levels avoid dirty reads. In this paper, we only look at isolation levels that avoid dirty reads.

[1] http://www.hibernate.org/.

[2] http://www.jboss.org/.

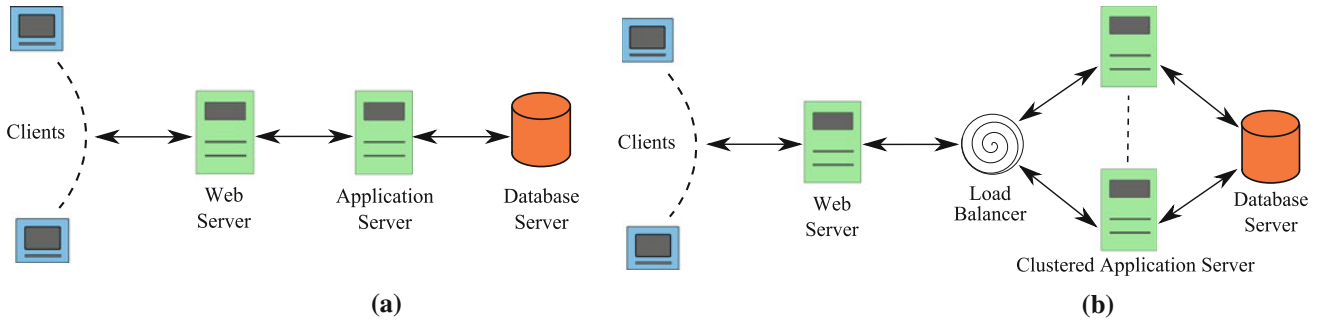


**Fig. 1** Multi-tier architecture. **a** Stand-alone middle-tier server. **b** Clustered middle-tier

With *read skew*, a transaction $T_1$ reads $x$ then a transaction $T_2$ writes $x$ and $y$, and then $T_1$ reads $y$. Thus, $T_1$ reads $x$'s version before $T_2$'s update, and $y$ after $T_2$'s update. If $x$ and $y$ are related, the reads are inconsistent. *Unrepeatable read* is a degenerate case of read skew where $x = y$ (Fig. 1).

In *lost update* (Fig. 2a), $T_1$ performs first a read and then later a write on the same entity $x$, while $T_2$ updates $x$ between $T_1$'s read and write operations.

In *write skew* (Fig. 2b), $T_1$ and $T_2$ read some entities $x$ and $y$, then $T_1$ writes entity $x$ and $T_2$ entity $y$, leading potentially to inconsistent data if $x$ and $y$ are related.

Note that there also exist phantom anomalies that can occur with predicate reads. Dependency graphs are not well suited to find such anomalies, and we do not further consider them in this paper.

## 2.2 Multi-tier architectures

Many Web-based information systems follow a multi-tier architecture as depicted in Fig. 1. The Web server (WS) tier handles the communication with external clients, provides static and dynamic web-pages, and calls the application server tier (in this paper referred to as middle-tier) for more complex requests. In turn, the middle-tier calls the database system when it has to access persistent data. Figure 1a shows a stand-alone middle-tier configuration with one application server while in Fig. 1b, the middle-tier is clustered which is often used when scalability is required. In this paper, we focus on application servers that are conform to the java enterprise edition (JavaEE) standard as many of the major industrial

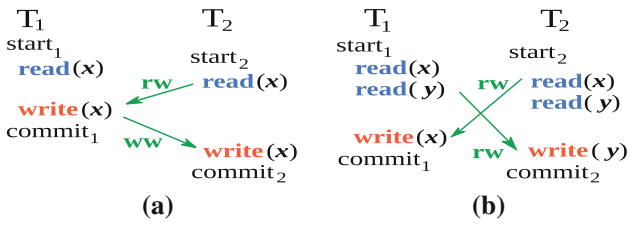


**Fig. 2** Some consistency anomalies. **a** Lost update. **b** Write skew

application servers and several popular open-source products, such as Sun GlassFish[3] and JBoss,[4] follow this standard. In JavaEE, the middle-tier consists of two layers. The first implements the business logic and takes care of starting and committing transactions. The second layer is the persistence layer which takes care of the mission-critical data that need to be persisted in a database system.

In JavaEE, persistence is managed by the JPA [18]. JBoss Hibernate[5] and Oracle TopLink[6] are two popular JPA implementations. This layer provides a high-level object-oriented abstraction of the database layer. Each record of a database table is represented as an entity in the persistence layer. An entity can be considered a cached version of the corresponding database record. Entities are accessed through a set of methods provided by an entity manager (EM). Entities can be read, inserted, and deleted through corresponding EM methods. To update an entity, it must be first read (loaded), then updated via setter methods, and then flushed back to the database. Thus, updates require more than one EM method call. In this paper, we will use the terms persistence layer and JPA interchangeably.

## 2.3 Middle-tier concurrency control

Transactions are started at the middle-tier which is also the client for the database tier. In principle, a transaction is split into a middle-tier transaction $T_{MT}$ which accesses entities and performs some computation, and a database transaction $T_{DB}$ executing at the database. A first call to an entity $x$ by $T_{MT}$ loads $x$ via $T_{DB}$ from the database into a local copy $loc_x$ visible only within $T_{MT}$. Subsequent calls of $T_{MT}$ to $x$ are served by $loc_x$ which reduces database interaction. Due to these local caches, the middle-tier requires some form of concurrency control to provide the appropriate isolation level. In this paper, we look at three isolation levels commonly imple-

[3] http://glassfish.dev.java.net/.

[4] http://www.jboss.org/.

[5] http://www.hibernate.org/.

[6] http://www.oracle.com/technology/products/ias/toplink/.

mented. Read committed (RC) and SI are two well-known isolation levels that are also frequently offered by database systems. Furthermore, we present JOCC, an isolation level provided by a special form of optimistic concurrency control offered by JPA implementations.

#### 2.3.1 Read committed (RC)

This isolation level is widely used in database systems so it is not surprising that it is also offered at the middle-tier. Most database systems implement it via locking: a read on $x$ requires a shared lock on $x$ which is released after the operation; write operations acquire exclusive locks that are only released at the end of the transaction. Thus, a read operation accesses the latest committed version. RC avoids dirty writes because exclusive locks are kept until the end of the transaction; it avoids dirty reads because it acquired shared locks albeit short ones. But generally, RC allows the other anomalies.

One possible implementation of RC at the middle-tier runs the database transaction $T_{\mathrm{DB}}$ under the RC level guaranteeing that whenever $T_{\mathrm{MT}}$ loads an entity $x$ into the middle-tier, the last committed version of $x$ as of start of the load is returned. Then, the middle-tier transaction $T_{\mathrm{MT}}$ cashes the value as $loc_x$. Updates are always done first on this local copy $loc_x$. At the end of $T_{\mathrm{MT}}$, if $x$ was updated, the new version stored in $loc_x$ is sent to the database in the form of an update operation. Once all writes have been successfully executed at the database, both $T_{\mathrm{MT}}$ and $T_{\mathrm{DB}}$ commit. Interesting to note that whenever the middle-tier transaction $T_{\mathrm{MT}}$ reads an entity several times, it always accesses the cached version $loc_x$. Thus, in most cases, a transaction does not perform multiple reads of the same entity at the database avoiding most cases of unrepeatable reads. However, they might still occur. Complex queries (such as SQL select statements with complex WHERE clauses) cannot be served by cached copies. As the database transaction runs under RC, submitting twice the same query can lead to different results if other transactions performed changes in between. Furthermore, general read skew caused by two different entities, lost update and write skew are possible.

#### 2.3.2 JPA OCC (JOCC)

As described in [18], JPA implementations provide an optimistic concurrency control which we denote as JPA OCC (JOCC). It is different to traditional textbook optimistic concurrency control [20] in that it does not provide serializability. The reason is that it only detects conflicts between write operations. JOCC assumes that the database transaction $T_{\mathrm{DB}}$ runs under the RC isolation level. In order to detect conflicts between writers, it enforces a transaction $T_{\mathrm{MT}}$ (and its corresponding database transaction $T_{\mathrm{DB}}$) to abort if any entity $x$ that $T_{\mathrm{MT}}$ changed was updated since $T_{\mathrm{MT}}$ read it into its local copy $loc_x$. This conflict detection is often implemented via automatic versioning. It adds a new attribute (version) to each entity class and also adds a new column for it to the database table representing this entity. This version is managed automatically and does not require any involvement of the developer. When an entity is modified by $T_{\mathrm{MT}}$, its version is increased and at commit time of $T_{\mathrm{MT}}$, JOCC checks the current version of $x$ at the database and compares it with the version when the entity was first read by $T_{\mathrm{MT}}$. If they are the same, the current value of $x$ is written to the database. If they are different, another transaction has changed the entity in between and $T_{\mathrm{MT}}$ is aborted. To get a closer look on how this works under Hibernate (a JPA implementation), here is a concrete example. Assume that a transaction $T_{\mathrm{MT}}$ loads a *Product* from the database with a *productId* equal to 100 and *objVersion* equal to 1. It then modifies the product's price to the value 75.0. At the commit time of $T_{\mathrm{MT}}$, Hibernate sends an UPDATE query to the database as follows:

```
UPDATE Product
      SET price = 75.0, objVersion = 2
WHERE productID=100 AND objVersion = 1
```

If a concurrent transaction updated and committed this entity, then the *objVersion* column no longer contains the value 1, the update statement does not update any record, and returns an error message upon which $T_{\mathrm{MT}}$ aborts. Like RC, JOCC only reads committed data and generally avoids unrepeatable reads as it always reads the cached version. It also avoids lost updates because the version checks abort transactions if the entities they write have been updated since the read operation. However, read and write skew can occur.

#### 2.3.3 Snapshot isolation (SI)

Snapshot isolation (SI) has been a popular isolation level within database systems for many years. The *Snapshot Read* property of SI requires that a transaction reads data from a snapshot that reflects the committed data as of its start time, that is, a transaction $T_i$ reads a data version $x_j$ created by a transaction $T_j$ which was the last to update $x$ and commit before $T_i$ started. The *Snapshot Write* property disallows concurrent transactions to update the same data item, that is, if there are two concurrent transactions and both update the same data item $x$, one of them must abort. As readers and writers do not interfere, SI provides good concurrency. SI avoids read skew and lost update but allows write skew. Recent implementations [22,23,29] extended SI to *serializable* SI where such types of anomalies are not allowed. SI is simple to achieve at the middle-tier if the underlying database system provides SI: the database transaction simply has to run under SI instead of RC. In contrast to JOCC, no versioning system is needed. Starting the DB transaction under

SI isolation level guarantees that all reads will read from a snapshot as of the first database access. This also holds for complex SQL queries that cannot be served from local copies. When changes are written back to the database at commit time, the database will guarantee that the Snapshot Write rule is enforced, i.e., it will abort transactions if there were concurrent writes.

#### *2.3.4 Serializability*

The serializability isolation level avoids all anomalies and no cycles occur in the dependency graph. This can be enforced, for example, by a strict 2-phase-locking protocol. In this case, our detection approach is not needed. Thus, we do not consider applications in which all transactions run under the serializability isolation level.

#### *2.3.5 Isolation level mixes*

We will see later that different isolation levels have certain properties that can be exploited when determining dependencies and detecting cycles. Thus, in the following, we distinguish applications in which transactions run under various types of isolation levels that are related to each other as depicted in Fig. 3.

**RC+**: An application runs under RC+ if all of its transactions run under an isolation level, or a mixture of isolation levels, that read(s) only committed versions of data items (and has no dirty writes). For instance, if all transactions run under RC, JOCC, or SI, or if the transactions run under any mixture of RC, JOCC, SI, and serializability, then the application is considered to run under RC+.

**NoLostUpd**: An application runs under NoLostUpd if all of its transactions run under an isolation level, or a mixture of isolation levels, that read(s) only committed versions and *avoid*(s) lost updates. For instance, if all transactions run under JOCC or SI, or if the transactions run under any mixture of JOCC, SI, and serializability, then the application is considered to run under NoLostUpd. Applications that run under NoLostUpd are a subset of the applications that run under RC+.

**LostUpd**: An application runs under LostUpd if all of its transactions run under isolation levels that read only committed versions and at least one transaction type within the application runs under an isolation level that *allows* lost updates. For instance, if all transactions run under RC, or some transactions run under RC while others run under JOCC, SI, or serializability, then the application is considered to run under LostUpd. Applications that run under LostUpd are a subset of the applications that run under RC+ and build the complement to the applications that run under NoLostUpd.

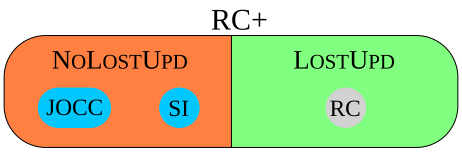


**Fig. 3** Isolation levels higher than or equal to RC

## 3 Detecting inter-dependencies

Detecting anomalies in transactional executions requires several steps. First, we have to detect when dependencies occur, that is, we must know when there are $wr$, $ww$, and $rw$ edges between transactions. For that, we have to make observations during the execution of transactions. Given these dependencies, we have to build the graph and then detect cycles. In this section, we discuss how we can detect dependencies at the middle-tier layer of the transactional system. We only focus on the more theoretical concepts and delay the description of the concrete implementation to Sect. 9.

### 3.1 Read dependencies

When a transaction $T_j$ reads an entity $x$, the middle-tier generally does not have any means to know the transaction $T_i$ that created the version of $x$ that $T_j$ reads. In order for us to extract this information, we tag an entity with the identifier of the transaction that updated the entity, respectively, created that particular entity version. Tags become part of the entity and thus are made persistent together with the entity, when the entity is written back to the database. In the following, we denote with $x.txnInfo$ the transaction identifier tag of an entity $x$. When a transaction $T_j$ loads an entity $x$ from the database, then $x.txnInfo$ reveals the transaction $T_i$ that has written this version of the entity. With this, we can establish a $wr$-dependency $T_i - wr \rightarrow T_j$ from $T_i$ to $T_j$. How exactly we implement tags transparently to the application within a JavaEE-server is discussed in Sect. 9.

### 3.2 Write dependencies

$ww$-edges are built between transactions that install consecutive versions of an entity. The question now is how to determine the exact order in which transactions write the entities.

Interestingly, we can easily determine $ww$-dependencies in case of applications running under NoLostUpd taking advantage of the fact that applications developed for JavaEE JPA do not perform blind writes, i.e., they always read an entity before they write it.[7]

[7] The property of no blind writes holds for many middle-tier based applications. It includes that we do not support blind-updates caused by triggers.

**Proposition 3.1** *Under* NoLostUpd*, if a transaction $T_j$ reads a version $x_i$ created by $T_i$, creates a new version $x_j$ and commits, then $x_j$ is the immediate successor of $x_i$ and there is a $ww$-dependency edge $T_i - ww \rightarrow T_j$ from $T_i$ to $T_j$.*

*Proof* By definition, under isolation levels that avoid lost updates, $T_j$'s update can only succeed and $T_j$ commit if no transaction $T_k$ updated $x$ and committed after $T_i$ and before $T_j$. Therefore, $x_j$ is the immediate successor of $x_i$. □

Thus, as we can easily determine the transaction $T_i$ that created the version of $x$ loaded by $T_j$ through the $txnInfo$ tag, determining $ww$-edges is simple.

However, if the isolation level allows lost updates, such as possible under LostUpd, Proposition 3.1 is not valid. Other transactions are allowed to write an entity between a read and a consecutive write. In this case, what is needed is to determine exactly in which order the database commits transactions (at least for those transactions that conflict). However, this is not possible at the middle-tier layer, at least not for commit operations that are submitted concurrently, unless we have access to database internal logs [10]. Therefore, our approach enforces a commit order for update transactions in order to be able to deduce $ww$-edges. Enforcing a commit order has been a common technique in the past [11,16], and in the implementation Sect. 9, we propose two solutions that work in stand-alone and clustered configurations. Our techniques only serialize the commits themselves while all read and write operations continue to execute concurrently. Our performance analysis shows that this serialization has very little impact on performance.

Given this commit serialization, we assume that under LostUpd, each update transaction $T$ has a unique commit time $c_i$ that we use to derive $ww$-edges by means of the following proposition.

**Proposition 3.2** *Given two transactions $T_i$ and $T_j$ such that (i) $T_i$ creates version $x_i$ and $T_j$ version $x_j$ of an entity $x$, (ii) $c_i < c_j$, and (iii) not $\exists T_k$ such that $T_k$ creates version $x_k$ and $c_i < c_k < c_j$. Then, $x_j$ is the immediate successor of $x_i$, and there is a $ww$-dependency edge $T_i - ww \rightarrow T_j$ from $T_i$ to $T_j$.*

In summary, we consider two cases for detecting the version order of entities:

1. If an application runs under NoLostUpd, then Proposition 3.1 is enough to detect the order of entities versions.
2. If an application does not run under NoLostUpd but is still conform to RC+, i.e., it runs under LostUpd, then we assume that for any update transaction $T$, we can determine the unique commit time $c_i$ and Proposition 3.2 is enough to detect the order of entities versions.

### 3.3 Anti-dependencies

There is a $rw$-edge $T_i - rw \rightarrow T_j$ from $T_i$ to $T_j$ if there is an entity $x$ for which $T_i$ has read the version $x_k$ (created by another transaction $T_k$) and later $T_j$ creates the immediate successor $x_j$ of $x_k$. This means there is a triangle of dependency edges: $T_k - ww \rightarrow T_j$, $T_k - wr \rightarrow T_i$ and $T_i - rw \rightarrow T_j$. Thus, once we have determined $T_k - ww \rightarrow T_j$ and $T_k - wr \rightarrow T_i$ through the mechanisms described above, we can immediately conclude $T_i - rw \rightarrow T_j$.

### 3.4 Collection

It is the task of the collector agent ColAgent, installed at the middle-tier layer, to collect all necessary information to determine dependencies among transactions. First, it creates transaction identifiers for all transactions. At commit time of a transaction $T_i$, ColAgent overwrites $txnInfo$ of each entity updated by $T_i$ with $T_i$'s identifier. In case of LostUpd, it also guarantees a commit order for update transactions and labels each transaction with the commit time stamp $c_i$. Finally, for each transaction $T$, ColAgent collects all necessary information to determine dependencies with other transactions. For each entity $x$ read by $T$, it logs its primary key $x.key$, the identifier of the transaction that created the version that was loaded ($x.txnInfo$) and a flag $x.flag$, indicating whether the entity was also updated (recall that the entities written is a subset of the entities read). Our implementation does not require any access to the source code of the studied application, thanks to several JPA features that allow us to access dynamically the primary key of any entity as well as its $txnInfo$ information. Our stress tests have shown that activities of ColAgent impose very little overhead at the middle-tier. ColAgent then sends the collected information to the detector agent DetAgent who determines the dependencies, builds the graph, and detects cycles. These tasks are described in the next two sections.

## 4 Building the dependency graph

DetAgent processes the transaction information it receives from ColAgent, one transaction at a time, adding the transaction to the dependency graph and determining the dependency edges this transaction has with transactions that were processed before. Processing a transaction is done by method ProcessTx as shown in Algorithm 1. ProcessTx is nearly the same for all isolation levels with some subtle differences, as for NoLostUpd, we use Proposition 3.1 to derive $ww$-edges while under LostUpd, we use Proposition 3.2 to extract such edges.

Furthermore, DetAgent ensures that under LostUpd, update transactions are processed in their ascending commit order in order to correctly derive the $ww$-dependencies

based on Proposition 3.2. It is able to correctly order transactions as it is provided with the commit time stamps $c_i$. The details of how to do so are discussed in Sect. 9. In all other cases (read-only transactions or any transaction type for NoLostUpd), no such requirement exists and transactions can be processed in any order. For any dependency edge $T_i \rightarrow T_j$, processTx builds the edge when the second of the two transactions arrives. In the case of $ww$-dependenices under LostUpd, $T_j$ will always be processed the second, otherwise it might be $T_i$ or $T_j$.

### 4.1 General data structures

All the algorithms that we show in this section assume some *global data structures*: (a) a dependency graph $G(V, E)$ where $V$ is the set of transactions that have already been processed and $E$ the set of dependency edges that have already been determined; (b) $readersOf$ keeps track for each entity $x$ and transaction $T$ that wrote $x$, the transactions that read the version created by $T$; (c) $successors$ keeps track for each entity $x$ and transaction $T$ that wrote $x$, the transaction that wrote the consecutive version of $x$; (d) $Last$ is a table that maintains for each entity $x$ the identifier of the last transaction that was added to the graph and updated $x$.

Furthermore, the transaction information $T$ detAgent uses to build the graph is as follows: $T\_id$ is the identifier and $T.entities$ is the set of entities accessed by $T$. Each $x \in T.entities$ records a key $x.key$, stores in $x.txnInfo$ the identifier of the transaction that wrote the version that $T$ read, and $x.flag$ is true if $T$ wrote $x$.

**processTx (Algorithm 1).**
processTx takes the information of a transaction $T_c$ as input and adds $T_c$ and its edges to the graph. First, processTx processes each entity read by $T_c$ by calling processReadEntity (see Algorithm 2). Then, it processes each entity updated by $T_c$ by calling processUpdatedEntity (see Algorithm 3). While processReadEntity looks for $T - wr \rightarrow T_c$ and $T_c - rw \rightarrow T$ edges, processUpdatedEntity checks for all types of edges involving $T_c$. In the following, we describe each of these steps in more detail.

detAgent includes an edge whenever the second transaction involved in an edge is processed. Figure 4 shows all possible edges for a transaction $T_c$ currently processed.

1. For any $x_c$ created by $T_c$
   - There is an incoming $T_p - ww \rightarrow T_c$ edge from predecessor transaction $T_p$ that created the predecessor version $x_p$ of $x$ (unless $T_c$ creates the entity).
   - There is an outgoing $T_c - ww \rightarrow T_s$ edge to successor transaction $T_s$ that creates the successor version $x_s$ of $x_c$ (unless $T_c$ deletes $x$ or there is no more update).
   - There are zero or more $T_{prw} - rw \rightarrow T_c$ edges from transaction $T_{prw}$ that reads the predecessor $x_p$ of $x$.
   - There are zero or more $T_c - wr \rightarrow T_{swr}$ to a transaction $T_{swr}$ that reads the version $x_c$ created by $T_c$.
2. For any entity version $x_f$ read by $T_c$
   - There is an incoming $T_f - wr \rightarrow T_c$ from transaction $T_f$ that created $x_f$.
   - There is an outgoing $T_c - rw \rightarrow T_t$ to transaction $T_t$ that creates the successor of $x_f$ (unless there is no further update on $x$).

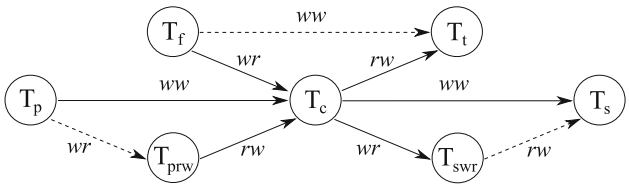


**Fig. 4** Incoming and outgoing edges for transaction $T_c$

**Algorithm 1** processTx ( $T_c$ )

**Input:** $T_c$ : a committed transaction.
**Output:** *builds* edges involving $T_c$

1: add $T_c$ to $G.V$
2: **for** $x$ **in** $T_c.Entities$ **do**
3: processReadEntity($x$, $T_c$)
4: **for** $x$ **in** $T_c.Entities$ where $x.flag = TRUE$ **do**
5: processUpdatedEntity($x$, $T_c$)

**processReadEntity (Algorithm 2).**
This algorithm aims in building the incoming $wr$-edge and the outgoing $rw$-edge for $T_c$ in regard to an entity $x$. It first extracts the identifier of $T_f$ from the field $x.txnInfo$ (line 1). $T_f$ might have been processed by detAgent before or after $T_c$. If $T_f$ was processed before $T_c$, then the $T_f - wr \rightarrow T_c$ edge can be immediately added to the graph (lines 2–3). If $T_f$ was not yet processed, then the edge will only be added once $T_f$ is processed. In order to detect this dependency when $T_f$ is processed, $T_c$ is added to a list that contains all readers of version $x_f$ (lines 4–5).

In order to create $T_c - rw \rightarrow T_t$, the algorithm calls $getSuccessorOf$ which tries to extract the identifier of the successor $T_t$ of $T_f$ in regard to $x$ (if there is any) by using the structure $successors$ which is populated after the creation of each new version of $x$ in processUpdatedEntity (see line 3). Again, $T_t$ might have been processed by detAgent before or after $T_c$. If it was processed before $T_c$, then the $T_c - rw \rightarrow T_t$ can be immediately added to the graph (lines 7–8). If $T_t$ was not yet processed, then the edge will only be added once $T_t$ is processed. For this to happen, we have to keep track that $T_c$ read version $x_f$ (lines 9–10).

**Algorithm 2** PROCESSREADENTITY $(x,\ T_c)$

**Input:**
$x$ : an entity.
$T_c$ : a transaction that has read the entity $x$.

**Output:** *builds* additional $wr$- and $rw$-edges for $T_c$

1: $T_f\_id = x.txnInfo$

2: **if** $T_f \in G.V$ **then**
3: *add* an $wr$-edge : $T_f - wr \rightarrow T_c$ to $G.E$
4: **else**
5: add $T_c$ to $readersOf(x.key, T_f\_id)$ // edge will be created once $T_f$ is processed as an update transaction, (line 3 in Algorithm 3)

6: $T_t\_id = getSuccessorOf(x.key, T_f\_id)$

7: **if** $T_t\_id$ valid identifier ($T_t$ was already added to $G.V$) **then**
8: *add* a new $rw$-edge : $T_c - rw \rightarrow T_t$ to $G.E$
9: **else**
10: add $T_c$ to $readersOf(x.key, T_f\_id)$ // edge will be created once $T_t$ is processed as an update transaction, (line 3 in Algorithm 3)

**Algorithm 3** PROCESSUPDATEDENTITY$(x,\ T_c)$

**Input:**
$x$ : an entity.
$T_c$ : a transaction that has created the entity $x$.

**Output:** *builds* additional $ww$-, $wr$- and $rw$-edges for $T_c$

// find the last transaction to update $x$
1: **if** isolation level NOLOSTUPD **then**
2: $T_p\_id = x.txnInfo$
3: **else**
4: $T_p\_id = getLast(x.key)$
5: $setLast(x.key, T_c\_id)$

6: **if** $T_p \in G.V$ (always true under RC+) **then**
7: *add* $ww$-edge : $T_p - ww \rightarrow T_c$ to $G.E$
8: **else**
9: do nothing, edge will be created once $T_p$ is processed (line 3)

10: $T_s\_id = getSuccessorOf(x.key, T_c\_id)$

11: **if** $T_s\_id$ valid identifier ($T_s$ was already added to $G.V$ which never occurs under RC+) **then**
12: *add* $ww$-edge : $T_c - ww \rightarrow T_s$ to $G.E$
13: **else**
14: do nothing, edge will be created once $T_p$ is processed (line 3)

15: add $[(x.key, T_p\_id), T_c\_id]$ **to** *successors*

// add missing $rw$-edges where $T_c$ is the writer and the reader was processed before $T_c$
16: **for each** $T_{prw} \in readersOf(x.key, T_p\_id)$ **do**
17: *add* $rw$-edge : $T_{prw} - rw \rightarrow T_c$ to $G.E$

// add missing $wr$-edges where $T_c$ is the writer and the reader was processed before $T_c$
18: **for each** $T_{swr} \in readersOf(x.key, T_c\_id)$ **do**
19: *add* $wr$-edge : $T_c - wr \rightarrow T_{swr}$

**PROCESSUPDATEDENTITY (ALGORITHM 3).**
This algorithm is called for each update on entity $x$ performed by update transaction $T_c$. Under LOSTUPD, it is called only if all update transactions with commit time smaller than $c_i$ have been processed. This condition does not apply under NOLOSTUPD. This algorithm aims to build one incoming $ww$-edge, one outgoing $ww$-edge, zero or more incoming $rw$-edges, and zero or more outgoing $wr$-edges for $T_c$.

First, it starts by extracting the identifier of transaction $T_p$ that was the last to update $x$ (lines 1–5). Under NOLOSTUPD, this information can be found in the $txnInfo$ of the entity $x$, as $T_c$ also reads every entity that it writes, and Proposition 3.1 guarantees that there is no other version in between. In case of LOSTUPD, this algorithm keeps track for each entity $x$ of the last processed transaction that updates $x$ (line 3). As update transactions are processed in commit order under LOSTUPD and Proposition 3.2 holds, this guarantees to capture the predecessor $T_p$ of $T_c$ in regard to $x$.

If $T_p$ had been processed before $T_c$ (which is always true under LOSTUPD), a $T_p - ww \rightarrow T_c$ edge is immediately built. If $T_p$ has not yet been processed, this edge will only be created once PROCESSUPDATEDENTITY is called for $x$ and $T_p$. After that, this algorithm checks whether the successor $T_s$ of $T_c$ was already processed before $T_c$ (lines 10–14). This is only possible for NOLOSTUPD. In this case, $T_c - ww \rightarrow T_s$ is created. Also, the successor structure is updated as it is needed if any transaction $T$ with $T - ww/rw \rightarrow T_c$ dependency has not yet been processed (line 15).

From line 3 to line 3, missing $wr$- and $rw$-edges are built. These are edges where the reading transaction was processed before the writing transaction and thus could not be built during PROCESSREADENTITY.

In "Appendix 13", we show the correctness of this approach, i.e., for any type of dependency between two transactions $T_i$ and $T_j$, PROCESSTX adds the corresponding dependency edge to the dependency graph exactly at the time the second of the two transactions is processed.

All data structures maintained by our approach can be implemented with simple hash functions. Thus, building the dependency graph is linear with the number of operations.

## 5 Detecting cycles

The most common mechanism to detect cycles in directed graphs is DFS [27] (Depth First Search). In its basic version, the algorithm simply indicates whether a given graph contains cycles or not. With this, execution can stop once the first cycle is detected. We, however, want to find all cycles that exist and also determine the nodes involved. A straightforward extension of DFS, denoted as EXTDFS, allows us to do so. For each node $T$ in the graph, EXTDFS starts a recursive routine EXTDFST, shown in Algorithm 4, with initial input $(T, T, \emptyset, outgoing/incoming, \emptyset)$ visiting all nodes reachable from $T$. For simplicity, we again assume that the graph $G$ is globally accessible. The search can either follow outgoing edges ($T$ is the source) or incoming edges ($T$ is the

**Algorithm 4** EXTDFST(...)

**Input:**
*currentT* : the start transaction of the search
*finalT* : the searched transaction
*visitedT* : a list of transactions already visited
*direction* : search along outgoing or incoming edges
*paths* : a list of paths, initially empty

**Output:** all found paths to *finalT* are added to *paths*

```
1: if (currentT == finalT and visitedT ≠ ∅ then
2:   add visitedT to paths
3: if (direction == OUTGOING) then
4:   ToFollow = {T|T → currentT ∈ G.E}
5: else
6:   ToFollow = {T|currentT → T ∈ G.E}
7: for t ∈ ToFollow do
8:   if (t ∉ visitedT) then
9:     newVisited = visitedT + t
10:      EXTDFST (t, finalT, newVisited, direction, paths)
```

sink). EXTDFST returns all paths of $G$ ending with $T$ and each returned path indicates the presence of cycles. A search starting from node $T$ only stops if there are no more edges to follow. The problem is that this approach is extremely costly, and with thousands of transactions in the graph, running times quickly become prohibitive.

Therefore, this section explores two approaches that are able to detect cycles much more efficiently by exploiting the specific structures and properties of dependency graphs and their cycles, depending on the isolation level. In our first approach, we take a set of transactions, build the dependency graph, and then perform cycle detection on the graph. By exploiting the fact that all cycles have a certain structure in terms of the types of edges and the commit order of transactions, we can dramatically reduce the number of edges that we have to traverse to find cycles to only a small fraction of the overall number of edges. In the second approach, we interweave graph construction and cycle detection. Whenever a transaction is added to the graph via PROCESSTX, we detect whether the transaction is part of any cycle. We guarantee that we detect a cycle when the last transaction involved in the cycle has been added to the graph. Again, by being careful in which order we add transactions to the graph and how we traverse the graph to detect cycles, we can dramatically reduce the number of edges to be followed.

The principle idea in both approaches is that while there are many edges in the graph, few are related to cycles. Ideally, we do not look at edges that are not involved in cycles but only traverse edges that are likely to be part of a cycle. In this section, we demonstrate how we can achieve this.

## 5.1 Detection in the overall graph

The first approach is the more natural to think of. We first build the entire graph as described in the previous section and then perform cycle detection on that graph by checking for each node in the graph, whether it is involved in a cycle.

### 5.1.1 Cycle properties

In our cycle detection, we take advantage of the fact that the edges in our graph are labeled, and that, depending on the isolation level used, the cycle can only contain certain types of edges. Fekete et al. [15] showed that all cycles that can be produced under the isolation level SI have a very specific property as indicated in Theorem 5.1.

**Theorem 5.1** ***(SI)*** *[15] Given an execution possible under the SI isolation level, any possible cycle C in its dependency graph has at least two consecutive rw-edges* $T_1 - rw \rightarrow T_2 - rw \rightarrow T_3$ *such that:*

- $T_3$ *is the first committing transaction in C*
- $T_2 \parallel T_1$ *and* $T_2 \parallel T_3$ *(*$\parallel$ *means concurrent)*
- $T_1$ *can be equal to* $T_3$

Motivated by this quite strong restriction on the type of cycles, we analyzed whether other isolation levels have similar restrictions. In fact, we determined that a very similar property, slightly less restrictive, holds for any execution under RC+.

**Theorem 5.2** ***(RC+)*** *Given an execution possible under* RC+*, any possible cycle C in its dependency graph has at least two consecutive edges* $T_1 - wr/ww/rw \rightarrow T_2 - rw \rightarrow T_3$ *such that the edge from* $T_2$ *to* $T_3$ *is a rw-edge while the edge from* $T_1$ *to* $T_2$ *can be of any type (*$wr$, $ww$ *or* $rw$*). Furthermore, the following holds:*

- $T_3$ *is the first committing transaction in C*
- $T_2 \parallel T_1$ *and* $T_2 \parallel T_3$
- $T_1$ *can be equal to* $T_3$

*Proof* see "Appendix 14.1" □

Theorem 5.2 is less specific than Theorem 5.1 as it allows any kind of edge from $T_1$ to $T_2$, that is, the cycles allowed under SI are a subset of the cycles allowed under isolation levels that are RC+. Figure 2a shows an execution possible under RC which produces a cyclic dependency graph with $T_1 = T_3$, $ww$-edge from $T_1$ to $T_2$, and an $rw$-edge from $T_2$ to $T_3/T_1$. Note that such an execution is not possible under SI as both transactions are concurrent and write the same entity. Thus, SI aborts one of them.

In summary, Theorems 5.1 and 5.2 show that the cycles possible under the most common isolation levels have a very specific structure (Fig. 5). And we exploit this structure to speed up the cycle detection process.

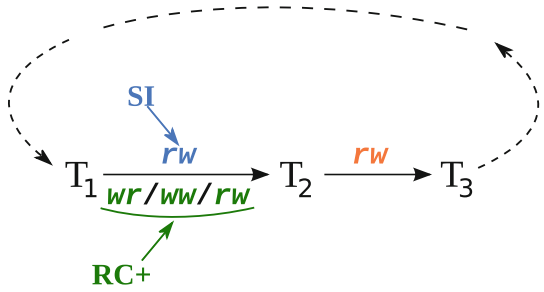


**Fig. 5** SI and RC+ cycles

**Algorithm 5** CONDEXTDFS( )

**Output:** *returns* cycles in the complete and globally accessible graph $G$

```
1: for T in G.V do
2:   T2 = T
3:   for T3 ∈ {T|(T2 − rw → T ∈ G.E} do
4:     if ( (  T2.start < T3.start < T2.commit
           or T3.start < T2.start < T3.commit )
         and T3.commit < T2.commit then
5:       for T1 ∈ {T|T → T2 ∈ G.E} do
6:         if ( T1 == T3 ) then
7:           cycle(T2, T3) detected
8:         else if ( (T2.start < T1.start < T2.commit
                 or T1.start < T2.start < T1.commit )
               and T3.commit < T1.commit )
9:               and T1 − rw → T2 ∈ V.E ) (only for SI) then
10:          segment = {T1, T2, T3}
11:          foundPaths = {}
12:          CONDEXTDFST (T3, T1, T2, {}, INCOMING ,
                          T3.commit, foundPaths)
13:          for path in foundPaths do
14:            add segment to path
15:            return foundPaths
```

### 5.1.2 Cycle detection process

Algorithm 5 describes the cycle detection process. Based on Theorem 5.2, each cycle $C$ has a section $T_1 - rw/ww/wr \rightarrow T_2 - rw \rightarrow T_3$, where $T_2$ is concurrent to both $T_1$ and $T_3$, and $T_3$ is the first transaction to commit in $C$. We take this property to scan through each transaction node $T$ and check whether $T$ could play the role of $T_2$ in a cycle $C$. Thus, for each candidate transaction $T$, considered as $T_2$, we check whether it has at least one outgoing $rw$-edge to concurrent $T_3$ that committed before $T$ and one incoming edge from concurrent transaction $T_1$ (in case of SI, a $rw$-edge) such that $T_3$ committed before $T_1$. We can determine whether two transactions $T_i$ and $T_j$ are concurrent if $T_i.start < T_j.start < T_i.commit$ or $T_j.start < T_i.start < T_j.commit$. Only if this is true, $T_1 \rightarrow T_2 \rightarrow T_3$ is a potential segment of a cycle. If $T_1$ is equal to $T_3$, then we have a cycle of size 2; otherwise, we run an algorithm **CONDEXTDFST** which is similar to EXTDFST with some additional conditions. More specifically, CONDEXTDFST starts its search from $T_3$ and targets $T_1$. It adds a new transaction $T$ to its search path, only if $T$ commits after $T_3$, as $T_3$ is the first committer in the cycles we are looking for. Given a *segment* $T_1(-rw) \rightarrow T_2 - rw \rightarrow T_3$, CONDEXTDFST returns all paths starting from $T_3$ and ending with $T_1$. Each of these paths is concatenated with *segment* to form a detected cycle.

**Algorithm 6** CONDEXTDFST(...)

**Input:**

*currentT* : the start transaction of the search
*finalT* : the searched transaction
*excludedT* : a transaction to exclude in the search
*visitedT* : a list of transactions already visited
*direction* : search follows outgoing or incoming edges
*smallestC* : first commit time in the cycle
*paths* : a list of paths, initially empty

**Output:** all found paths to *finalT* are added to *paths*

```
1: if (currentT == finalT) then
2:   add visitedT to paths
3: if (direction == OUTGOING) then
4:   ToFollow = {T|T → currentT ∈ G.E}
5: else
6:   ToFollow = {T|currentT → T ∈ G.E}
7: for t ∈ ToFollow do
8:   if (t ∉ visitedT and t != excludedT) then
9:     // First Committer Condition
10:    if (t.commit > smallestC) then
11:      newVisited = visitedT ∪ t
12:      CONDEXTDFS (G, t, finalT, excludedT,
           newVisited, direction, smallestC, paths)
```

### 5.1.3 Search direction

DFS can be performed both in the direction of outgoing edges, and reverse, along the direction of incoming edges. The question is which direction is more efficient? It turns out it is much more efficient to follow incoming edges, i.e., following the cycle in reverse order.

To better understand why this is the case, Fig. 6 shows a cycle with the transactions sorted by their commit order. $T_3$ must be the first committed, and we denote with $T_L$ the last committer. $T_3$ has one incoming $rw$-edge coming from $T_2$ which commits after $T_3$. In general, we group edges into two categories. An edge from $T_i$ to $T_j$ is a forward edge if $T_i$ commits before $T_j$. Forward edges are prefixed with an $f$ and shown below the commit line in the figure. An edge from $T_i$ to $T_j$ is called a backward edge if $T_j$ commits before $T_i$. They are prefixed with a $b$ and shown above the commit line in the figure. Lemma 5.1 provides us with an interesting property for backward and forward edges.

**Lemma 5.1** *If there is an edge from transaction $T_i$ to transaction $T_j$ in the dependency graph of an execution under*

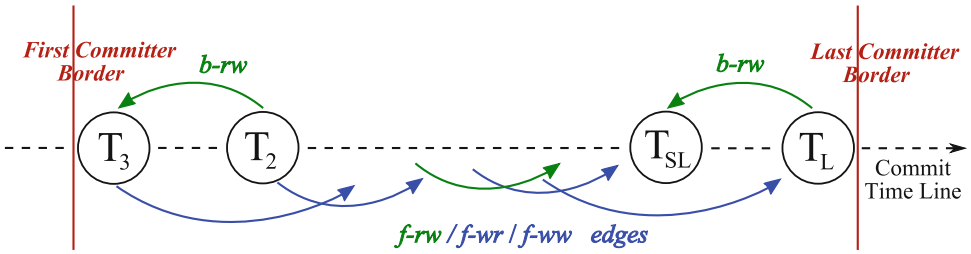


**Fig. 6** The dependency graph based on the commit time line

RC+ *and $T_i$ commits after $T_j$, then (1) this edge can only be of type $rw$ and (2) $T_i$ is concurrent to $T_j$.*

*Proof* see "Appendix 14.2" □

Basically, as only committed data are read and written, $ww$- and $wr$-edges are always forward edges, following the commit order of transactions. Only for $rw$-edges can the commit order be reversed. And if this is the case, the transactions must be concurrent. This limits the number of $b\text{-}rw$ edges comparatively to the number of forward edges in the graph. In fact, in all our experiments, $b\text{-}rw$ edges formed less than 4 % of all edges in the dependency graph.

We take advantage of this property and use CONDEXTDFST by following incoming edges starting from $T_1$ and trying to reach $T_3$. In this case, CONDEXTDFST is limited in its search by two criteria. First, incoming edges above the commit time line are limited by the existence of only few $b\text{-}rw$ edges in the graph. Second, if we follow incoming edges below the commit time line, we push CONDEXTDFST to explore nodes (transactions) in the direction of the $First\text{-}Committer\text{-}Border$ (as shown in Fig. 6) imposed by $T_3$ as the first committer. Once one of these incoming edges leads to a transaction that commits before $T_3$, our algorithm does not follow it.

In contrast, if CONDEXTDFST follows outgoing edges starting from $T_3$, then the $First\text{-}Committer\text{-}Border$ will not stop the search from exploring long paths following $f\text{-}rw/f\text{-}wr/f\text{-}ww$ edges that never will lead back to $T_1$. In Sect. 10.5, we analyze the performance of search both following outgoing and incoming edges in detail.

#### *5.1.4 Start and commit orders*

In Algorithm 5, we check in lines 4 and 8 whether two transactions are concurrent. To do so, we need both the start and commit time stamps of these transactions. More details on extraction of the start and commit time stamps of transactions will be provided in the implementation Sect. 9. Note that under some databases, it might not be possible to extract the exact start time stamp. In this case, the concurrency check in lines 4 and 8 can simply be skipped. However, it is important to know in which order transactions commit and we indicate how we can achieve this in Sect. 9.

### 5.2 Detection in the incremental graph

In this section, we propose an approach where we combine graph creation and cycle detection in a single process. While this might not appear obvious at first view, it has an attractive advantage. If we are able to do so in an efficient manner, then we can detect cycles, as we will later outline as our online approach, in near real time. Whenever new transactions commit, their information can be sent to DETAGENT. Then, DETAGENT adds one transaction at a time and immediately checks whether it is involved in a cycle. The problem is that at the time of cycle detection, the graph is not complete (as further transactions will commit and add new dependencies). Thus, it is not clear whether we can take advantage of the same cycle properties as in the previous section and whether we would be able to find all cycles at all.

In a first step, we analyze whether we can exploit the fact that each cycle contains a pair of edges $T_1 \rightarrow T_2 - rw \rightarrow T_3$ where $T_3$ is the first committer in the cycle. In our previous algorithm, we checked for each transaction $T$ whether it could take the role of $T_2$ by first checking whether $T$ has an outgoing $rw$-edge to a transaction that committed before $T$. If this is the case, we would start the search. Now assume, we do cycle detection incrementally and assume a transaction $T$ in fact plays the role of $T_2$ in a cycle. When we add $T$ to the graph, however, not all other transactions in the cycle might yet have been added to the graph. It is even possible that $T_3$ was not yet added to the graph. Thus, we would miss the cycle. Therefore, we cannot exploit the pruning options we had taken in the last section.

Nevertheless, if we apply the original EXTDFS, we can be sure that we find all cycles. More precisely, whenever a new transaction $T$ and all its dependency edges to already existing transactions are added to the graph in Algorithm 1, we look for cycles by calling EXTDFST($T$, $T$, $\emptyset$, $outgoing$, $\emptyset$). With a little bit of reasoning, it becomes clear that we will indeed find all cycles. Assume a cycle $C = T_1 \rightarrow T_2 \ldots T_n \rightarrow T_1$ involving $n$ transactions and $n$ edges in the dependency graph $G$. For any edge in the graph, the edge is created when the second of the two transactions involved in the edge is processed shown in "Appendix 13". Thus, once all $n$ transactions are processed, all edges in the cycle have been constructed. Without the loss of generality, assume that $T_n$ is the last transaction that is processed, adding the final edges

$T_{n-1} \rightarrow T_n$ and $T_n \rightarrow T_1$. When we now call EXTDFST on $T_n$, the cycle $C$ will be detected since all edges of $C$ are already present in the current graph.

The question now arises whether EXTDFST will be efficient enough. Quite surprisingly, it is, when we are smart enough about it. As a first step, DETAGENT will try to add transactions to the graph and process them as close to their commit order as possible. It has to do so anyways for update transactions in case of LOSTUPD. In the other cases, it does not need to be exactly the commit order but the closer the processing order is to the commit order, the more efficient the search will be. For simplicity, the following discussion will assume that transactions are added exactly in commit order. As a second step and opposite to before, our EXTDFST will follow outgoing edges and not incoming edges.

From there, we consider again Fig. 6 with $T_L$ being the last committer. At the time $T_L$ is added, the only outgoing edges that can be added to the graph are to transactions that committed before $T_L$. As indicated in Lemma 5.1, these can only be $rw$-edges to transactions that were concurrent to $T_L$. Very few such edges exist. Thus, we explore very few edges in the first step of the search. And if there is such an edge, and we continue to follow the path of outgoing edges, any $f\text{-}rw/f\text{-}wr/f\text{-}ww$ edge will quickly push the search again into the direction of the *Last-Committer-Border* where the search will stop as the edges beyond that border were not yet created. This will force the search to stop in a few steps and not follow long paths that are not related to cycles. If transactions are not processed in exactly the commit order, then the search might follow some extra $f\text{-}rw/f\text{-}wr/f\text{-}ww$ edges until the *Last-Committer-Border* is reached. Thus, the more the processing order follows the commit order, the less edges we need to investigate.

In contrast, if our search goes in the reverse direction along incoming edges, we may easily follow a long list of $f\text{-}rw/f\text{-}wr/f\text{-}ww$ edges (involving transactions that committed before $T$) since most of them exist at the time when $T$ is processed.

## 6 Memory management

In order to perform cycle detection, the data structures that represent the dependency graph need to be in main memory. If the graph is too large, memory restrictions can become a problem. The following theorem helps us to limit the parts of the graph that need to be kept in memory while at the same time guaranteeing that we do not miss any cycles.

**Theorem 6.1** *Let $C = T_1, \ldots, T_n$ $(n \geq 2)$ be a cycle generated by an execution running under* RC+. *Recall that $s_i$ $(c_i)$ denotes the time transaction $T_i$ starts (commits). Furthermore, let*

- $s_{min} = min(s_i), 1 \leq i \leq n$
- $c_{max} = max(c_i), 1 \leq i \leq n$
- $\Delta_{max} = max(c_i - s_i), 1 \leq i \leq n$

*We claim that:* $c_{max} - s_{min} \leq n * \Delta_{max}$

*Proof* see [30] □

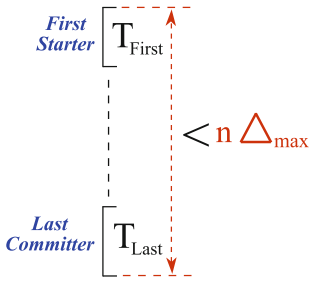


**Fig. 7** Cycle duration under RC+

This means that the duration of a cycle $C$ involving $n$ transactions, calculated as the time from the first start of any transaction in $C$ to the commit of the last committing transaction in $C$ is at most $n$ times as long as the longest running transaction in the cycle. This is depicted in Fig. 7.

We can use this property as follows to restrict the part of the graph that we have to keep in main memory. Our approach assumes an upper bound on the length of any cycle, denoted as $C_{\max}$, and an upper bound on the run-time of any transaction in the system, denoted as $\Delta_{\max}$. In order to determine all cycles in which a transaction $T_i$ is the last committer (with commit time $c_i$), DETAGENT has to have access to all transactions (and their dependencies) that started at time $c_i - C_{\max} * \Delta_{\max}$ or later and committed before $T_i$. All transactions that started earlier cannot be involved in a cycle with $T_i$. For example, if $C_{\max}$ is set to 20 and $\Delta_{\max}$ is equal to 30 s (both values chosen very conservatively), the transactions executing within a window of 10 min of $T_i$ must be hold in main memory in order to find all cycles. Even at a throughput of 100,000 transactions per minute, this is easily possible. Note that for this to work, we have to perform cycle detection close to the commit order of transactions, which we do.

## 7 Classification of cycles

So far, our cycle detection mechanism provides information about the exact transactions involved in a cycle, such as the transaction identifiers, the types of dependencies, and the affected entities. However, if there are many different cycles, it is difficult to check what in the application actually caused these cycles and whether these cycles are problematic from an application and data consistency point of view. Therefore, we also classify cycles according to the anomaly type they cause as well as the business methods involved.

In particular, we classify the detected cycles into two types of patterns. The first classification determines the kind of

anomaly that is caused by the cycle, such as lost update or read skew. In the second classification, we categorize cycles by their corresponding business methods linking cycles to the responsible parts in the applications.

### 7.1 Anomaly classification

While many practitioners are not familiar with serializability theory and dependency graphs, many know anomalies such as unrepeatable reads or lost updates. Therefore, we classify, whenever possible, the cycles according to the anomaly they cause. In Sect. 2.1, we have listed the most well known. For isolation levels RC+, these are read skew, unrepeatable read, lost update, and write skew. All of them are defined in terms of two conflicting transactions. It is quite straightforward which cycle of length represents which anomaly.

*Read skew* causes a cycle $T_1 - rw \rightarrow T_2 - wr \rightarrow T_1$ where the edges are due to different entities.
*Unrepeatable read* has the same cycle as read skew but both edges are caused by the same entity.
*Lost update* leads to a cycle $T_1 - rw \rightarrow T_2 - ww \rightarrow T_1$ and both edges are caused by the same entity.
*Write skew* results in a cycle $T_1 - rw \rightarrow T_2 - rw \rightarrow T_1$ where the edges are caused by different entities.

Thus, in our system, whenever a cycle of length two is detected, we can easily determine whether the cycle belongs to one of these anomalies.

The question arises whether cycles of larger size reflect conceptually similar anomalies. In fact, by analyzing the cycles that were produced in our test runs, we determined two frequently occurring sequences of dependencies in cycles of length three. They are shown in Figs. 8a, b. Both cycles (in green) have the form $T_1 - rw \rightarrow T_2 - rw \rightarrow T_3 - wr \rightarrow T_1$. However, in the first, the dependencies are caused by two different entities, while in the latter case by only one entity.

The cycle of the type shown in Fig. 8a is conceptually similar to a read skew with the difference that $T_1$ reads the version of $x$ before $T_2$ writes it and the version of $y$ after $T_3$ writes it. $T_2$ and $T_3$ are ordered due to a $rw$-dependency resulting overall in a cycle. Thus, we call this anomaly transitive read skew or ***t-read skew*** for short.

If a cycle of the type shown in Fig. 8b occurs, then there is also a related cycle (in dashed red) of length two $T_2 - rw \rightarrow T_3 - ww \rightarrow T_2$, indicating a lost update. A lost update in this case means conceptually that $T_3$'s update is lost as $T_2$'s read was based on a stale read and not on the latest version written by $T_3$. The read $r_1(x_3)$ of $T_1$ indicates that this lost update was visible to the outside before it was overwritten. Thus, we call it a visible lost update or ***v-lost update*** for short. Note that if $T_1$ later writes $x$, leading to another lost-update cycle $T_1 - rw \rightarrow T_2 - ww \rightarrow T_1$, the inconsistencies become highly intertwined.

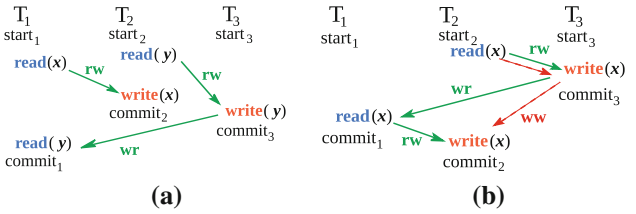


**Fig. 8** New consistency anomalies. **a** t-read skew. **b** v-lost update (color figure online)

Although we did not observe it in our test runs, transitive unrepeatable reads can just as easily be determined by finding cycles of the form $T_1 - rw \rightarrow T_2 - ww \rightarrow T_3 - wr \rightarrow T_1$, where all edges are caused by the same entity.

Note that we might categorize a cycle as, e.g., a read skew, while it might actually not be an anomaly for the application because the read entities are not related. However, this can only be analyzed by an expert, and our tool gives him/her all the information needed to make this analysis.

### 7.2 Business methods classification

In order to better link to the application, we extract for each detected cycle the business methods involved. More concrete, each cycle can be assigned to an $ordered$ and an $unor-dered$ pattern of business methods.

*Ordered Patterns.* Ordered patterns are a straightforward mapping of a cycle $C = T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$ of transactions in the dependency graph to an *ordered cycle pattern* $ordCP = bm_1 \rightarrow bm_2 \rightarrow \cdots \rightarrow bm_n \rightarrow bm_1$ of business methods where each transaction $T_i$ in $C$ is replaced by the business method $bm_i$ in which $T_i$ was embedded. Note that this is a many-to-one mapping, as many transactions map to the same business method. $ordC$ is called an ordered pattern, as the order in which the business methods appear in the cycle matters. For instance, the patterns

$ordCP_1 = bm_1 \rightarrow bm_2 \rightarrow bm_3 \rightarrow bm_1$ and
$ordCP_2 = bm_1 \rightarrow bm_3 \rightarrow bm_2 \rightarrow bm_1$

are different, since the order of methods is different: $bm_1$'s predecessor is $bm_3$ in $ordCP_1$, and $bm_2$ in $ordCP_2$.

#### 7.2.1 Unordered patterns

Ordered patterns give detailed information about the order in which business methods are interleaved but there might be many ordered patterns and they might be difficult to be parsed at first. Therefore, *unordered patterns* provide a more coarse-grade view on the cycles that occur, only indicating the set of business methods that cause the cycle but ignoring their order and how often each of them appears in the cycle. Continuing with the example above, the two different ordered patterns

$ordCP_1$ and $ordCP_2$ both map to the same unordered pattern

$unordCP = \{bm_1, bm_2, bm_3\}$.

Similarly, a transaction cycle that maps to the ordered pattern $ordCP_3 = bm_1 \rightarrow bm_3 \rightarrow bm_2 \rightarrow bm_2 \rightarrow bm_1$ has the same unordered pattern $unordCP = \{bm_1, bm_2, bm_3\}$ since duplications are not taken in consideration.

In summary, every cycle of transactions $C$ maps to a ordered list of business methods where each transaction is replaced by the business method it calls, and an unordered set of business methods that reflect the set of business methods involved in the cycle.

Furthermore, our system keeps track for each pattern how often it has actually occurred in the test run. Thus, in order to track down what problems incur in the system, the administrator might first start checking unordered business method patterns that occur frequently to find the culprit business methods, and if needed, continue checking ordered patterns as well as some concrete transaction cycles to find the exact dependencies that cause the cycles.

For example, the SEPECjEnterprise2010 benchmark has a method $completeWorkOrder$ that reads a work order entity, an assembly entity, and an inventory entity and then updates both the work order and inventory entities. Under RC, cycles of length two, where two transactions concurrently execute this same method, can occur leading to a lost update. Such a cycle is mapped to one ordered pattern $completeWorkOrder \rightarrow completeWorkOrder \rightarrow completeWorkOrder$ and one unordered pattern $completeWorkOrder$.

## 8 Cycle reduction

Our solution is mainly a diagnostic approach providing the system administrator or developer with information about the amount and the kind of inconsistencies that occur in a running application. How it is used depends on the circumstances. If no or few anomalies occur that are deemed to be acceptable, nothing needs to be done. For instance, if the cycles tagged as read skew involve unrelated entities, then there is no concern. However, if the number of cycles is high, countermeasures will likely need to be taken. There are mainly three options:

- The application itself remains unchanged but whenever inconsistencies occur that violate the integrity of the data, individual data entities can be manually changed to consistent values. As we provide information about the affected data entities, this is, in principle, possible. Some resolution mechanisms could be developed for this purpose. Such approach is highly application dependent.
- If a limited number of business methods cause a large number of cycles that actually reflect true anomalies for the application (e.g., a true write skew), it might be an option to rewrite these business methods, e.g., by reordering the way they read and write entities or by rewriting SELECT statements to SELECT FOR UPDATE statements (via JPA equivalent methods). Such a rewrite might lead to less anomalies. Again, this is very application dependent, requires a manual analysis of the code, and a semantically equivalent rewrite that avoids the problems might not always be possible.
- The isolation level of the application or some of its business methods is increased. Running in a higher isolation level will likely decrease the number of cycles observed during execution.

While the first two options require manual intervention and very good knowledge of the application domain to determine what are truly anomalies from an application point of view, the third option is much more general and requires much less understanding of the application. Furthermore, it can also be applied in an automatic and dynamic way by exploiting the pattern information that is collected. The idea is to assign stricter isolation levels to business methods that frequently cause cycles. Business methods that are not involved in cycles will continue to run with lower levels of isolation. For instance, whenever an unordered pattern passes a threshold of, e.g., 20 associated cycles, our system automatically increases the isolation levels of the business methods that occur in this pattern. Section 9.3 discusses how this is implemented by the DETAGENT and COLAGENT.

This mechanism is a generic preventive measure that automatically kicks in without the need of a detailed analysis of the cycles that occur. As an example of its usefulness, assume that a set of new methods is added to a running application and their dependencies are not well understood. If the new methods cause frequent cycles in a short amount of time after being added, quickly and dynamically increasing the isolation level will be the safest thing to do. If after an analysis of the cycles an expert determines that the cycles do not actually reflect true anomalies for the application, the isolation level can again be reduced. However, in the meantime, no severe harm is done to the database.

## 9 Deployment and implementation details

### 9.1 Off-line versus online deployment

As already hinted on, our approach can be deployed in two modes: *off-line* and *online*. In the off-line mode, the collector agent COLAGENT collects all necessary information during run-time, but the detector agent DETAGENT is only initiated periodically, e.g., when a certain time period has passed, a certain number of transaction has been executed or once every day such as after peak hour. Such off-line processing allows to delay the task of cycle detection to off-peak hours

or to a time where the system administrator or analyst has actually time to investigate in the cycles that occurred. It is also useful when a new/modified application is executed in a test phase in order to evaluate its performance and correctness before deployment. Then, consistency is only one issue to be analyzed which then can be done comfortably after the test run. In the offline mode, DETAGENT is assigned a complete set of transactions at start of the agent and can perform analysis on the entire graph as discussed in Sect. 5.1 or incrementally as outlined in Sect. 5.2.

In contrast, in the online mode, the COLAGENTs on the application servers and the DETAGENT (on any machine) are started at the same time. The COLAGENTs stream the collected information in near real time to the DETAGENT who uses the incremental graph creation and cycle detection strategy as outlined in Sect. 5.2. The online mode is useful if an application should be monitored in a continuous fashion, for example, in order to be able to intervene as soon as possible after inconsistencies have occurred.

### 9.2 Agents

Both COLAGENT and DETAGENT are written in standard Java. While DETAGENT is independent of the middle-tier technology, our current implementation of COLAGENT can work with any application server supporting the JavaEE standard. We have integrated it into the persistence layer Hibernate which follows the JPA standard. To support additional non-JavaEE platforms, only COLAGENT has to be changed, while DETAGENT remains the same.

COLAGENT. This agent has to capture dependency information and send it to DETAGENT. By default, COLAGENT is disabled. It can be enabled by a special request sent from the DETAGENT console.

In our JavaEE implementation, we track entity versions by adding an extra *txnInfo* attribute to each entity class (using the ALTER command for each entity class). This is similar in spirit to the version attribute used by JOCC and completely transparent to the application developer. The attribute has 4 bytes which makes it lightweight. Before an entity is written to the database, COLAGET sets $x.txnInfo$ to the identifier of the writing transaction. Note that $txnInfo$ is updated at the middle-tier like any other attribute of $x$; the persistence layer transparently maps its value to the corresponding table without the need for any additional update operations at the database. Thus, when a transaction $T_j$ loads an entity $x$, COLAGENT can extract the creator through the $x.txnInfo$ attribute and determine the $wr$-dependency. JPA provides annotations that help extract dynamically the primary key (simple or composed) from an entity class. Therefore, there is no need to check manually the source code of each entity class.

In the off-line mode, COLAGENT queues information on transactions in memory and flushes it asynchronously to the hard drive as XML files. Information is logged in ascending commit order. In the online mode, COLAGENT forwards information on each committed transaction $T$ to DETAGENT after $T$ commits. This is currently done via TCP/IP sockets.

DETAGENT. It receives the transaction information from COLAGENT, builds the graph, and detects cycles.

In the off-line mode, DETAGENT loads transaction information from the log files created by COLAGENT. If there are multiple COLAGENTs, DETAGENT merges the files to get a globally sorted list of committed transactions and processes them in commit order.

In the online mode, DETAGENT receives information on transactions in near real time immediately after their commit. Each COLAGENT sends information in ascending commit order using FIFO channels. If there is only a single COLAGENT, then DETAGENT can process information on any transaction as soon as it receives it. If there are many COLAGENTs, DETAGENT queues the received information based on ascending commit time and processes an update transaction $T_i$ only if $c_i$ is the smallest commit time stamp of all transactions it has not yet processed and it satisfies $c_i \leq \min(c_k)$ for all $k$, where $c_k$ is the commit time stamp of the last update transaction received from $\text{COLAGENT}_k$.

DETAGENT outputs cycles as they are detected. For each cycle, it generates a sub-history of the transactions involved, as well as the entities involved in the cycle. Each entity is presented by its class name and its primary key. Furthermore, the ordered and unordered patterns as well as the anomaly pattern to which the cycle belongs are determined. This information can be visualized graphically as shown in Fig. 9. The figure shows one of the cycles that occurred when running the SPECjEnterprise2010 benchmark under the RC isolation level. The cycle is of size 2, represents a lost update, and is between two transactions both calling the method $completeWorkOrder$, which belongs to the ordered pattern $Ord_2$ and the unordered pattern $Unord_2$. The $rw$ and $ww$ edges of this cycle involve an entity of type $Inventory$. The $txnInfo$ is shown at the end of each read.

DETAGENT also continuously produces statistics such as the total number of cycles, their distribution grouped by size as well as the number of occurrences for each business method pattern and anomaly type.

### 9.3 Anomaly reduction

As mentioned in Sect. 8, we offer a generic anomaly reduction mechanism by dynamically increasing the isolation level of business methods that are frequently involved in cycles. For that, a threshold variable has to be set in advance, and whenever a certain pattern (e.g., an unordered business method pattern) occurs more often than the threshold value, the isolation levels of the involved methods are increased. More specifically, DETAGENT determines when the thresh-

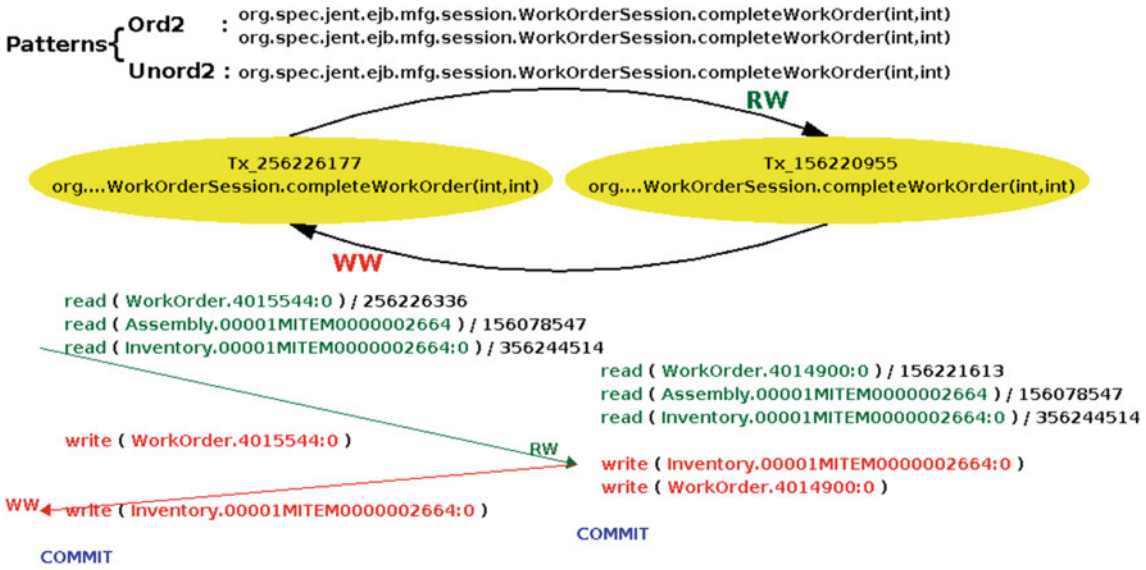


**Fig. 9** Detected cycle under read committed

old is reached and informs all COLAGENTs about the business methods. In their turn, each COLAGENT assigns dynamically a stricter isolation level to any transaction calling any of these methods. In our implementation, no restart of the application is needed. This approach allows anomalies to occur at the beginning of execution, but reduces them dynamically if too many of them occur.

## 9.4 Start and commit times

### *9.4.1 Commit times*

In order to provide update transactions running under LOSTUPD with a commit order, we have implemented two solutions. The first is memory-based, while the second is database-based. Note that we only serialize the commit requests of update transactions, otherwise transactions run concurrently.

Our first solution uses a COUNTER variable that resides in the main memory of the application server. Before the commit request of an update transaction $T_i$, an exclusive lock on COUNTER is requested and the current value is assigned as commit time $T_i.commit$ for $T_i$. Once the transaction has committed at the database, the COUNTER is increased and the lock released. However, this solution works only in a stand-alone application server. In a clustered configuration, COLAGENT acquires an exclusive lock on a database record using the SELECT FOR UPDATE query, then issues one SELECT query that extracts the current time at the database considered as the commit time $T_i.commit$. In both implementations, due to the exclusive lock and the monotony of the COUNTER, respectively, the current time at the database server, if $T_i.commit < T_j.commit$, then $T_i$ has committed before $T_j$ in the database.

### *9.4.2 Start times*

We need the start times of transactions in order to check whether two transactions are concurrent in Algorithm 5. In case of the use of the COUNTER variable, we can determine the start time $s_i$ of a transaction $T_i$ by reading the current COUNTER value. This does not require to set a lock as the current value is always guaranteed to be from a committed transaction.

In case of clustered configuration, we need to read the start time from the database server. Both PostgreSQL and Oracle provide SQL functions that return the start time of the current transaction. If this is not available, we can add a SELECT query at the begin of transaction that extracts the current time. No locks need to be set for that query. Note that the start and commit times we extract from the database are not the exact times. In particular, the actual commit time of a transaction is later than the time we extract through the SELECT statement. In order to compensate that when we compare for concurrent transactions, we add to the measured commit time a constant that ensures that the assigned commit time is at least as high as the true commit time at the database. Adding a too high value may lead to false comparisons of $T_1 \parallel T_2$ or $T_2 \parallel T_3$. However, this does not lead to the detection of incorrect cycles but might simply let CONDEXTDFS explore some edges in the graph unnecessarily.

## 9.5 Fault tolerance

For fault tolerance, we have to distinguish between the online and off-line mode. In the off-line mode, the COLAGENTS write the information to stable storage in any case. The interval in which the information is flushed from the main memory

buffer to disk determines the amount of information that might be lost in case a COLAGENT fails and can be a tunable parameter. Completely avoiding the loss of any information would require logging within the response time of the transaction—something we believe is too costly. In case the DETAGENT fails, it can be simply restarted.

In the online mode, COLAGENT sends transactional information shortly after commit time. If it fails, information about the most recently committed transactions might be lost. A failure of DETAGENT will lead to the loss of all transaction information so far sent to it. This can be avoided if either the COLAGENTS or DETAGENT log the information on stable storage. Then, only the information in transfer is lost.

### 9.6 Inserted and deleted entities

Insertions and deletions are handled just like normal updates. When an entity $x$ is created by transaction $T_i$, then COLAGENT keeps track of this fact and DETAGENT will not attempt to build an incoming $ww$-edge for $T_i$. If $T_i$ deletes $x$, then the entity disappears from the database and no subsequent transaction will load $x$ anymore. DETAGENT handles the delete as an update but will never create an outgoing $ww$- or $wr$-edge from $T_i$ to any other transaction.

### 9.7 Applications

Our approach works with arbitrary applications without putting any specific conditions on the studied application. Applications remain unaltered. Only an extra $txnInfo$ field is transparently added to each entity class. The approach also supports multiple applications running at the same time over the same database. In particular, it is possible to have two applications $app_1$ and $app_2$ where none of them experiences any cycle if it is running alone, but if they are running concurrently, some cycles may occur. Such cycles may appear if both $app_1$ and $app_2$ are accessing some shared entities in the database. This case is hardly detectable in real environments, since applications are generally developed and maintained by different teams. Our approach will detect cycles even if they involve transactions from different applications. Our pattern detection mechanism discussed in the previous section can tell exactly if a certain cycle has transactions triggered by methods of $app_1$ and/or $app_2$.

Note, however, that we require that all access to the data is through the middle-tier servers in which we have injected the COLAGENT. Otherwise, we would not be able to observe all transactions and dependencies.

## 10 Experiments

In this section, we present the experiments that we conducted to illustrate the various aspects of our detection approach: (a) a general analysis of the cycles that we were able to detect for three well-known benchmarks under the isolation levels RC, JOCC and SI, and for clustered and non-clustered configurations; (b) a detailed analysis of the anomaly types and business patterns that we were able to find for some of the applications; (c) an illustration of how our dynamic adjustment of isolation levels can help to reduce the number of cycles observed; (d) and finally, an evaluation of the overhead of the interception performed by COLAGENT and the efficiency of the two cycle detection mechanisms.

### 10.1 Benchmarks

Before we discuss the individual experiments, we shortly describe the different benchmarks that we have used.

**RUBiS.** RUBiS is a popular benchmark emulating an auction house similar to eBay.[8] RUBiS provides two workloads: *Browsing-mix* consists only of read-only transactions while *bidding-mix* has both read-only and update transactions. In order to provide more variation and create higher conflict rates, we have added a further workload type to RUBiS, called *DailyDeal*, which is an option that can also be found in real auction sites. In this option, there are a few items every day that are on special but only for this one day. Checking these deals and purchasing them generally generates many conflicts as there are few items. Our DailyDeal implementation has some transactions that read and write only one data item, while others read two data items and later write one of them. Thus, DailyDeal methods are prone to write skew which leads to unserializable executions under RC, JOCC, and SI. If the data items are related, which is often the case, this is a real write skew from the application point of view. Adding this transaction type allowed us to use RUBiS somewhat as a micro-benchmark as we could easily influence the conflict rate.

**SPECjEnterprise2010.** SPECjEnterprise2010 has been developed for JavaEE and models an automobile manufacturer. It has three domains: order, manufacturing, and supply. Within the ordering domain, a car dealer may browse (50 %) the catalog of cars, purchase cars (25 %), or manage (25 %) his inventory. Browse requests trigger read-only transactions, while purchase and manage requests trigger update transactions.

**TPC-C.** TPC-C is an industry standard benchmark for evaluating the performance of OLTP systems. It consists of nine tables and five procedures that simulate a warehouse-centric order processing application. Under the default settings, the transactions of TPC-C are write heavy and 96 % of them modify tables.

[8] http://www.ebay.com/.

## 10.2 Cycle analysis

In this section, we present basic results about the cycles we found when running the three benchmarks under different loads and configurations. For all experiments, we used PostgreSQL8.4 as database server and JBoss5.1 as application server with its persistence layer Hibernate3.5.0. The experiments were conducted at different times where we had access to different sets of machines; therefore, the machine configuration is described for each of the tests individually.[9]

### *10.2.1 RUBiS*

***Configuration.*** We deployed both a stand-alone configuration *stand-conf* and a clustered configuration *clust-conf*. In both, database server and RUBiS clients are on separate machines. In *stand-conf*, the application server is on a third machine, while *clust-conf* has three machines running application server instances and one extra machine for the load balancer (IP Virtual Server) that distributes RUBiS onto the three application servers. Each machine has 1 GB of RAM, an Intel Pentium D 2.8 processor, and runs Ubuntu 10.04. The tests were conducted by varying the number of remote clients (achieving up to 2,500 transactions per minute) and by using one of the middle-tier isolation levels RC, JOCC, or SI. For each test, we collected data during a steady state of 20 min after a warm-up period of 10 minutes.

***RUBiS without DailyDeal.*** In a first test, we ran RUBiS with 80% browsing-mix and 20% bidding-mix and without any calls to DailyDeal. With this setting, we were unable to see any cycle under any isolation level for all evaluated throughput levels; therefore, we are not showing any figures. Thus, at least for the tested configuration, none of the isolation levels leads to any anomalies and any of them can be used, that is, with this application type, the isolation level with the best performance can be chosen.

***RUBiS with DailyDeal.*** In a second setting, we ran RUBiS with 50 % browsing-mix, 20 % bidding-mix and 30 % DailyDeal. For the DailyDeal, we have selected 4 items. This configuration leads to cycles under all three isolation levels.

*Response Time and Aborts*. Not shown in any figure, the response time in *clust-conf* is almost five times faster than in *stand-conf*. This is due to the use of multiple application servers, so the load on each application server is lower, which leads to a quick processing time. However, for any of the two configurations, there is actually no response time differences between the different isolation levels. Of course, given a different database system, a different application, or any other differences in configuration parameters, the performance of the different isolation levels might vary much more. The amount of aborted transactions was ranging for RC from 1 to 90, for JOCC from 3 to 290, and for SI from 2 to 259 and this from 100 clients to 500 clients.

*Cycles*. Figure 10a shows the number of cycles with increasing number of clients for each isolation level To put this into perspective, during the run-time of the experience, between 10,000 transactions (for 100 clients) and 55,000 transactions (for 500 clients) are executed in total. For each isolation level, the column is divided into two parts. The bottom part reflects the number of cycles of size 2 and 3 while the top part reflects the remaining number of cycles.

We can observe that RC has generally a much higher number of cycles than JOCC and SI and quickly experiences a large number of cycles when it reaches the saturation point. This can be expected that RC has more unserializable executions since it is a lower level of isolation than JOCC and SI. Furthermore, a high proportion of cycles is either of size 2 or size 3. This confirms the assumption that cycles involve generally a few number of transactions.

Although the number of cycles given the total number of transactions executed is small, if they reflect true anomalies, inconsistencies can slowly creep into the database. Thus, given that the response times are almost the same for all isolation levels and the abort rates for JOCC and SI are comparable, the obvious choice is to use the isolation level with the lowest number of cycles, which is SI.

Figure 10b is for *clust-conf* and shows the same relative behavior as *stand-conf*. The main difference is that the total number of cycles is less for all tests and isolation levels. Furthermore (not shown in the figures), the number of aborts was also lower by about 40–45 %. The reason for this better behavior is that response times are shorter under *clust-conf* than under *stand-conf*. Thus, given a certain load, each transaction is concurrent with fewer other transactions reducing the chance of conflicts.

### *10.2.2 SPECjEnterprise2010 and TPC-C*

For the two other benchmarks, we show simplified results as they are conceptually similar to the results under RUBiS.

***SPECjEnterprise2010.*** We only show results for a clustered configuration with similar setup as under RUBiS just that each machine has an Intel Core2 2.66 GHz processor with 8 GB of memory and uses Fedora Core 13. Moreover, we have used the Apache Load Balancer since we require a load balancer that allows for sticky sessions.

SPECjEnterprise2010 controls the load submitted to the system via the injection rate (IR). For our experiments, we varied the IR from 15 to 35 and the number of generated transactions ranged, respectively, from around 45,000 to 100,000. Under JOCC and SI, we could not detect any cycle, while

[9] As we do not compare the actual performance nor compare the benchmarks with each other, we believe that having different machine configurations is not a problem.

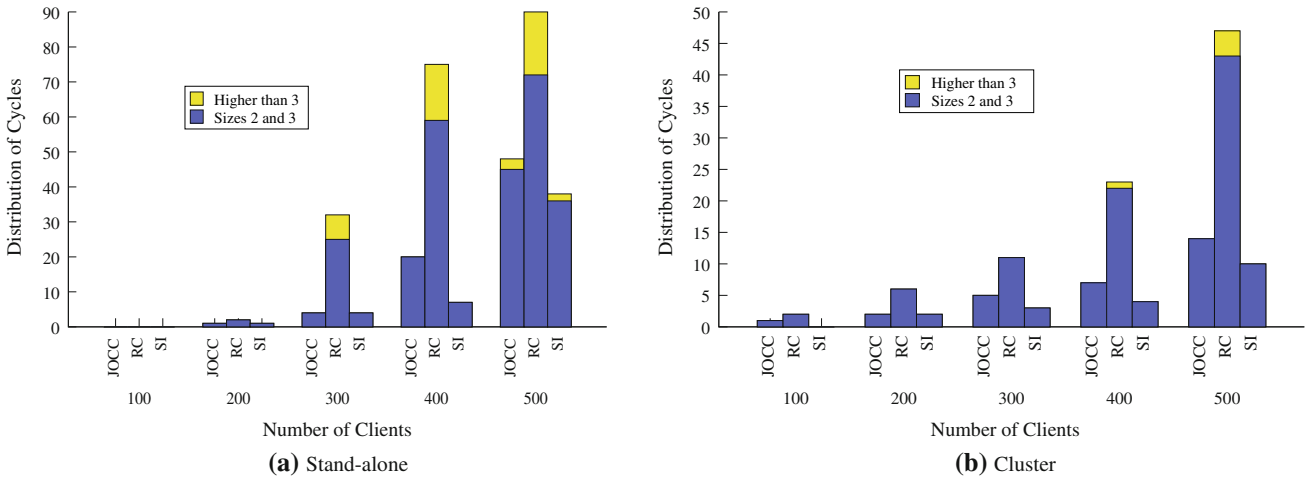


**Fig. 10** Distribution of cycles for RC, JOCC, and SI. **a** Stand-alone. **b** Cluster

**Table 1** Detected cycles for SPECjEnterprise2010

| Isolation level | Injection rate IR | | | | |
|---|---|---|---|---|---|
| | 25 | 30 | 35 | 40 | 45 |
| RC | 2 | 4 | 7 | 23 | 43 |

**Table 2** Detected cycles for TPC-C

| Isolation level | Number of terminals | | | | |
|---|---|---|---|---|---|
| | 25 | 30 | 35 | 40 | 45 |
| RC | 750 | 3,202 | 4,863 | 5,635 | 5,696 |

under RC we have detected a few cycles whose number becomes higher for higher values of IR (see Table 1).

***TPC-C.*** We only show results for a stand-alone configuration. In fact, all components are run on one powerful machine (Intel i7-2640 2.80 GHz dual-core processor, 8GB of memory running Ubuntu 12.04) for fast evaluation.

We chose TPC-C's write heavy workload where 96 % of transactions modify tables. We varied the number of terminals (clients) from 25 to 45 and this under RC, JOCC, and SI generating around 20,000 transactions. Under both JOCC and SI, we could not detect any cycles (for SI, [15] showed formally that TPC-C provides only serializable executions), while under RC we were able to detect many cycles that again increased with the load submitted to the system (see Table 2). The large number of cycles is due to the fact that TPC-C has many conflicting transactions that are not well isolated under RC.

## 10.3 Anomaly classification and patterns

In this section, we have a closer look at the anomaly types and the business method patterns that can be observed. We show results for TPC-C and RUBiS both run in a stand-alone configuration where all components are run on one powerful machine (Intel i7-2640 2.80 GHz dual-core processor, 8GB of memory running Ubuntu 12.04).

### *10.3.1 Anomaly classification*

**TPC-C.** In this part, we analyze the results of the experiments under TPC-C shown earlier in Table 2. Figure 11 shows the detected anomalies for TPC-C running under RC. The x-axis shows the number of terminals while the y-axis shows the number of detected anomalies. For each load (number of terminals), there are three histograms: number of cycles representing lost updates, number of cycles representing v-lost updates, and other cycle types with size higher than or equal to 4. As we see from the figure, most anomalies reflect some form of lost update. The number of v-lost updates is higher than the number of lost updates and this for all loads. By analyzing closely these cases, we found that each two transactions involved in a cycle of size 2 under lost update are also involved in a cycle of size 3 under v-lost update. As we discussed in Sect. 7, this shows how cycles and inconsistencies can be heavily intertwined. In particular, Table 3 shows the collocation of cycles among transactions. The first column shows the load (number of terminals), and the second column shows the total number of committed transactions for each experiments. The third column (size 0) shows the number of transactions not involved in any cycle, the fourth column shows the number of transactions involved in one cycle, and the fifth column shows the number of transactions involved in two or more cycles. The last three columns also indicate in parenthesis the percentage values over all transactions.

**RUBiS.** As RUBiS with DailyDeal shows cycles under all isolation levels, we focus on the differences between these isolation levels and show in Fig. 12 the number of detected

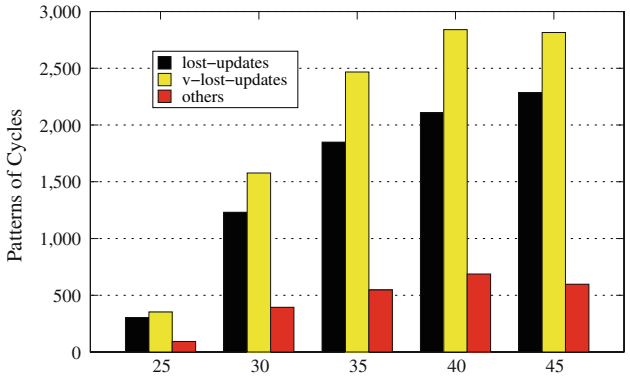


**Fig. 11** TPC-C: Anomalies under RC

**Table 3** TPC-C: collocated cycles under RC

| Load | Comm. | size 0 (%) | size 1 (%) | size 2+ (%) |
|---|---|---|---|---|
| 25 | 19,844 | 19,206 (96.7) | 173 (0.8) | 465 (2.3) |
| 30 | 19,339 | 16,480 (85.2) | 900 (4.7) | 1,959 (10.1) |
| 35 | 20,027 | 15,347 (76.7) | 1,687 (8.4) | 2,993 (14.9) |
| 40 | 19,446 | 14,077 (72.4) | 1,979 (10.2) | 3,390 (17.4) |
| 45 | 19,797 | 13,854 (70.0) | 2,458 (12.4) | 3,485 (17.6) |

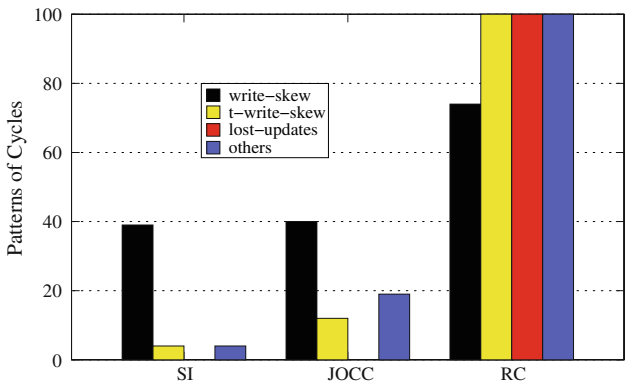


**Fig. 12** RUBiS: anomalies under different isolation levels

anomalies for these three levels for an execution with 800 clients. For these experiments, we have detected not only different types of lost update and v-lost update (for RC only) but also write skew and t-write skew under all isolation levels. For better visibility, for RC, we only show the true values for write skew and limit the values to 100 for the other anomalies (the true values are 777, 296, and 770). Generally, RC experiences many more anomalies than the other two isolation levels. Comparing SI and JOCC, write skew occurred nearly the same for both SI and JOCC and t-write skew values were higher under JOCC than SI.

Similarly to the TPC-C experiments, many transactions were also involved in more than one cycle. The Table 4 shows the collocation of cycles among transactions under the RUBiS experiments. The first column shows the isolation level, whereas the second column shows the total number of committed transactions for each experiments. The third column (size 0) shows the number of transactions not involved in any cycle, the fourth column shows the number of transactions involved in one cycle, and the fifth column shows the number of transactions involved in two or more cycles. Again, percentages are also provided.

**Table 4** RUBiS: collocated cycles under SI, JOCC, and RC

| IsL | Comm. | size 0 (%) | size 1 (%) | size 2+ (%) |
|---|---|---|---|---|
| SI | 69,794 | 69,711 (99.88) | 76 (0.11) | 7 (0.01) |
| JOCC | 69,919 | 69,824 (99.86) | 74 (0.11) | 21 (0.03) |
| RC | 69,780 | 69,146 (99.1) | 337 (0.48) | 297 (0.43) |

**Table 5** Business methods patterns in RUBiS

| Isolation level | Unordered business method patterns |
|---|---|
| SI | DailyDeal.createBuyNow |
| | DailyDeal.createBuyNow |
| | DailyDeal.getDailyDealsItems |
| JOCC | DailyDeal.createBuyNow |
| | DailyDeal.createBuyNow |
| | DailyDeal.getDailyDealsItems |
| | DailyDeal.createBuyNow |
| | DailyDeal.getItemDescription |
| RC | DailyDeal.createBuyNow |
| | DailyDeal.createBuyNow |
| | DailyDeal.getDailyDealsItems |
| | DailyDeal.createBuyNow |
| | DailyDeal.getItemDescription |
| | Bid.createBid |
| | DailyDeal.createBuyNow |
| | DailyDeal.getDailyDealsItems |
| | DailyDeal.getItemDescription |

### *10.3.2 Patterns*

To give the reader an idea on the business methods involved in the cycles observed under TPC-C and RUBiS, Table 5 shows the unordered business method patterns that caused all cycles in RUBiS with 800 clients under RC, JOCC, and SI, and Table 6 shows the unordered business method patterns causing cycles in TPC-C with 30 terminals under RC. It is easy to see that in both cases, few methods cause the anomalies and these are the ones that should be investigated. For instance, by analyzing the code of the method $createBu-yNow$, we found that it reads two items but updates only one of them. Both items are related via a constraint on their remaining quantities.

**Table 6** Business methods patterns in TPC-C (RC only)

| Unordered business method patterns |
|---|
| Payment.paymentTransaction |
| NewOrder.newOrderTransaction |
| Payment.paymentTransaction |
| NewOrder.newOrderTransaction |

## 10.4 Reduction of anomalies

We show the effectiveness of our anomaly reduction module using the SPECjEnterprise2010 with the same configuration discussed in Sect. 10.2. Figure 13 shows results for a test run that uses RC and a test run where we started RC as default isolation level, and increment it dynamically to SI for business methods involved in frequent anomalies. In the second run, COLAGENT sets the isolation level of a method from RC to SI once DETAGENT detects an unordered pattern with at least three cycles that contains this method. We denote the second type of test run as RC+SI.

The Fig. 13a shows the number of detected cycles under both RC and RC+SI for increasing IR. For low IR values, RC and RC+SI have the same results. Very few cycles are created; thus, no method is upgraded to use SI. However, for high IR values, RC generates many cycles, while RC+SI stabilizes the number of cycles to less than 10. These are the cycles that occur before the prevention option is triggered and elevates the isolation level for affected methods.

Figure 13b shows the response times for increasing IR. For high IR values, response times under RC+SI are slightly higher than under RC, although the penalty is nearly negligible. This is due to the fact that a subset of transactions are executed under SI, that is, while we have limited the number of generated cycles under RC+SI, we pay the price of higher response times. While not shown in the figure for better readability, the response times for both SI and JOCC were slightly higher than RC+SI. This indicates the benefit of only running those transactions with a stricter isolation level that would cause anomalies otherwise.

Figure 13c shows the absolute number of aborted transactions. Under RC+SI, there were more aborted transactions than under RC as it aborts concurrent transactions if their execution would lead to anomalies. The number of aborted transactions is negligible in comparison with the total number of committed transactions, but it is significant comparatively to the number of detected/prevented cycles. In fact, the number of transactions that RC+SI aborts more than RC is very similar in absolute terms to the number of cycles that RC+SI avoids compared to RC.

Figure 13d shows the percentage of transaction that run under SI using the RC+SI approach. The higher the workload, the higher the percentage of transactions that are upgraded to the higher isolation level.

## 10.5 Overhead and efficiency

### *10.5.1* COLAGENT

In order to check the overhead of COLAGENT on the performance of applications, we have run COLAGENT interception enabled and then disabled for RUBiS, SPECjEnterprise2010, and TPC-C. The difference in response time was less than 3 % for all selected tests, even for the RC (conform to RC+) tests where we had to execute commits serially. This confirms that the impact of interception is very low.

### *10.5.2* DETAGENT

Most of the cycles in RUBiS and SPECjEnterprise2010 were of size 2 or 3. These small size values did not allow us to compare how efficient our cycle detection performs. In order to get a clear understanding of the performance, we have generated a large dependency graph with 300,000 nodes and nearly 1,000,000 edges. The graph has more than 10,000 cycles with a maximum length of 15. Although this is quite extreme, it allows us to better compare our different cycle detection algorithms. In order to create such a large graph, we have developed an emulator that emulates a set of concurrent transactions accessing a limited set of entities under the RC isolation level.

We test two scenarios: cycle detection on the overall graph in the off-line mode and incremental cycle detection in the online mode. For the first, DETAGENT has access to the entire xml file created by COLAGENT, first builds the graph and detects cycles using CONDEXTDFS (Algorithm 5) that itself calls CONDEXTDFST (Algorithm 6). For the second, COLAGENT streams the transactions to DETAGENT, who builds the graph transaction by transaction and checks at each step for new cycles using EXTDFST (Algorithm 4).

Tables 7 and 8 show the results for off-line evaluation on the overall graph and for online evaluation with incremental graph creation, respectively. As both approaches can traverse the graph along outgoing edges or along incoming edges (i.e., in reverse order), we have two sets of results for both scenarios. The tables depict the time for building the graph, for the time spent in CONDEXTDFST, respectively EXTDFST, and for the actual number of edges that are explored in the graph traversal. The latter two values depend on whether the graph traversal follows incoming or outgoing edges. Furthermore, for the off-line approach, Table 7 shows the time spent in Algorithm 5 to check whether a transaction can play the role of $T_2$ in Theorems 5.1/5.2 (denoted as *overhead*), that is, the time needed to perform concurrency comparisons.

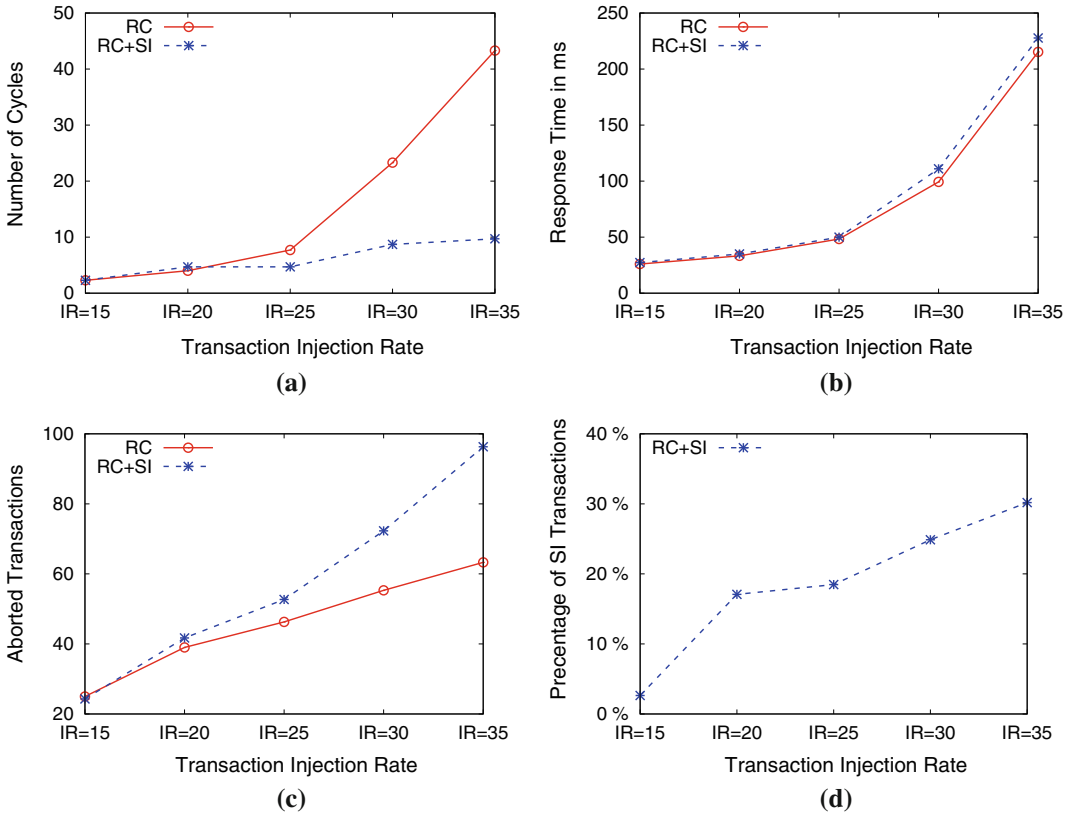


**Fig. 13** Results for SPECjEnterprise2010 under RC and RC+SI. **a** Detected cycles. **b** Response time. **c** Aborted transactions. **d** Percentage of SI transactions

**Table 7** Results for the overall graph/off-line

| | | |
|---|---|---|
| Building the graph | 7,037 ms | |
| Overhead | 2,923 ms | |
| | *outgoing* ⟶ | ⟵ *incoming* |
| Calls to CONDEXTDFST | ∞ | 163 ms |
| Number of explored edges | ∞ | 19,221 |

**Table 8** Results for the incremental graph / on-line

| | | |
|---|---|---|
| Building the graph | 7,213 ms | |
| | *outgoing* ⟶ | ⟵ *incoming* |
| Calls to EXTDFST | 861 ms | ∞ |
| Number of explored edges | 80,382 | ∞ |

The time to create the graph is very similar for both with the incremental approach taking slightly longer as it is created in many mini-steps. Overall, the time is only a few seconds for the 300,000 nodes and thus can easily be handled in real time.

For the cycle detection in the overall graph, we can see that finding cycles along incoming edges is extremely fast, as only few transactions qualify as being transaction $T_2$ in Theorems 5.1/5.2, and after that, only few dependency need to be followed as the *First-Committer-Border* shown in Fig. 6 eliminates many dependencies to be considered. In total, less than 20,000 of the 1,000,000 edges are explored. In contrast, following outgoing edges, the *First-Committer-Border* does not help, as the transactions that are explored along the outgoing edges likely commit all later and thus have to explored. We stopped the execution of the algorithm after a certain time (thus, the notation of ∞).

For cycle detection in the incremental graph, following outgoing edges was very fast, due to the *Last-Committer-Border* shown in Fig. 6. When cycles for a transaction are explored, the transaction is the last committed one in the graph and thus will have only few outgoing edges that have to be explored, as outgoing edges to transactions that committed earlier are rare. In contrast, when following incoming edges, we had to follow more and more edges as the graph grew, as we could not exploit a pruning mechanism, leading to unacceptable execution times (denoted as ∞).

In total, the cycle detection itself was shorter on the overall graph than for the incremental graph. This is due because the number of explored edges was smaller. Furthermore, CONDEXTDFST is called less than EXTDFST. It is only called when a transaction qualifies as transaction $T_2$ of Theorems 5.1/5.2, while EXTDFST is called for every transaction. However, CONDEXTDFST has additional overhead to compare

transactions which was not negligible in our case. Thus, in total, incremental cycle detection was slightly faster than cycle detection on the overall graph.

## 11 Related work

Most related to our work is [14]. The authors provide quantitative information about the violations of integrity constraints when a database system runs under SI or RC. The analysis is performed over a microbenchmark consisting of two related tables. The authors were able to show that in some situations, SI leads to more anomalies than read committed. Our approach differs from this work in several aspects. First, it is designed to work in a typical multi-tier architecture where execution is spread across middle- and database-tiers. Second, it is completely independent of any application additionally to treating the database tier as a black box. Third, it is not only able to detect the anomalies that occur under read committed and SI, but it supports any isolation level higher than or equal to RC. It also classifies the detected anomalies into patterns which can be used to dynamically prevent further anomalies.

Fekete et al. [15] present an approach that characterizes non-serializable executions for a given application when run under SI. It is based on the manual analysis of the application in order to find possible conflicting operations. Jorwekar et al. [17] suggest a tool based on [15] which automates the detection of possible conflicts between a set of operations under SI. These operations are extracted by manually analyzing a given application and by automatically extracting information from the SQL database logs. Both [15] and [17] are dedicated to SI and require to some degree manual analysis of the studied application. They also only give information about possible conflicts and do not quantify how often such conflicts actually occur during run-time.

In [8], Cahill et al. proposed an approach for preventing anomalies under SI at the database layer. They abort some transactions that may be part of cycles under snapshot isolation, but their approach is conservative in that it causes unnecessary aborts. Revilak et al. [23] extended this work by aborting only transactions that are certain to be part of real cycles. In their approach, they build the serialization graph at the database engine, and at commit time of each transaction $T$, they check whether $T$ is part of any cycle. If this is the case, then $T$ is aborted at the database layer. This approach stops any transaction $T$ from committing until it adds additional edges to the graph where $T$ is involved and checks if $T$ is part of any cycles.

In contrast, we build the dependency graph and detect cycles after commit time at detector agent which has no impact on any currently running transaction. Moreover, our detector agent is independent of any database. Both works in [8] and [23] are dedicated to SI only and implemented, respectively, under Berkeley DB or MySQL InnoDB. In [22], the authors presented an approach similar to [23] but implemented it under PostgreSQL.

In [6], the authors propose a concurrency control protocol for a middle-tier cache that allows update transactions to read out-of-date data items which satisfy some freshness constraints. A freshness constraint specifies how fresh a copy of a data item must be in order to be read. Similar ideas are presented in [24,25]. Our approach does not present a new concurrency control mechanism but measures the number of unserializable executions. In principle, it could be used to measure the number of unserializable executions produced by these novel concurrency control mechanisms, although the detector agent would require adjustments.

A preliminary version of the work presented in this paper appeared in [31] where we presented only the online detection approach that we have integrated later into the consistency anomaly detector tool ConsAD [32]. The present paper extends the approach to (1) cover as well the off-line mode, (2) support a mixture of isolation levels, (3) prevent dynamically consistency anomalies, and (4) manage efficiently the memory usage which allowed the detector agent to run for long periods under the online mode without any memory leaks. It allowed it as well to load and process, off-line, large histories of transactions.

In [33], we detect consistency anomalies for cloud applications. As not all dependencies are observable in such environment, we only provide an approximated graph which can lead to false positives during cycle detection. Isolation levels are not taken into account because many cloud applications do not even provide the concept of transactions.

## 12 Conclusions

This paper presented a novel approach for detecting and dynamically reducing consistency anomalies at the middle-tier layer of a multi-tier architecture. The approach collects enough information during normal transaction processing to be able to create a dependency graph and perform cycle detection. Our system can detect any anomalies for transactional workloads that run under any mix of isolation levels that is at least read committed. Our approach is completely independent of the application and treats the database as a black box. It provides quantitative information about anomalies as well as information about the business methods that cause them. Finally, our approach supports an option where the isolation level of some business methods is dynamically increased should they be involved in too many cycles.

We integrated our approach into an open-source application server and used it to analyze the types and number of cycles that can occur in the RUBiS, SPECjEnterprise2010,

and TPC-C benchmarks under various isolation levels. The performance evaluation shows that the overhead of extracting transactional information during run-time is very lightweight and cycle detection extremely efficient. Although cycle detection in general graphs is known to be exponential with the length of the cycle, we have developed pruning mechanisms that allow us to detect all cycles in seconds in graphs with millions of edges and thousands of cycles.

In our future work, we are planning to extend our approach to isolation levels that allow their transactions to read stale data items satisfying some freshness constraints. This requires to extend the collector agent as well as to analyze each isolation level provided in these environments in order to understand what anomalies can occur and how to detect and prevent them.

## 13 Correctness of algorithm 1

In this section, we show that for any type of dependency between two transactions $T_i$ and $T_j$, PROCESSTX adds the corresponding dependency edge to the dependency graph at the time the second of the two transactions is processed. Assume without loss of generality that there is an edge from $T_i$ to $T_j$ due to entity $x$. We consider several cases:

1 $T_i$ is processed before $T_j$
   a If $T_i - wr \rightarrow T_j$, then the edge is added when PROCESSREADENTITY, line 3, is executed for $T_j$.
   b If $T_i - rw \rightarrow T_j$, PROCESSREADENTITY for $T_i$ adds $T_i$ to the readers of the predecessor of $T_j$ (in regard to $x$) in line 10, and PROCESSUPDATEENTITY for $T_j$ adds the proper $rw$-edge in lines 16–17.
   c If $T_i - ww \rightarrow T_j$, PROCESSUPDATEENTITY for $T_j$ adds the edge in lines 6–7.

2 $T_j$ is processed before $T_i$
   a If $T_i - wr \rightarrow T_j$, PROCESSREADENTITY for $T_j$ adds $T_j$ to the readers of $T_i$ (in regard to $x$) in line 5, and PROCESSUPDATEENTITY for $T_i$ adds the proper $wr$-edge at lines 18–19.
   b If $T_i - rw \rightarrow T_j$, PROCESSUPDATEENTITY for $T_j$ adds $[(x.key, T_p\_id), T_j\_id)]$ to *successors* in line 15, and PROCESSREADENTITY for $T_i$ retrieves $T_j\_id$ in line 6 and adds the $rw$-edge in lines 7–8.
   c If $T_i - ww \rightarrow T_j$, PROCESSUPDATEENTITY for $T_j$ adds $[(x.key, T_i\_id), T_j\_id)]$ to *successors* in line 15, and PROCESSUPDATEENTITY for $T_i$ retrieves this information in line 10, and adds the $ww$-edge in lines 11–12.

Each edge is added exactly once. Therefore, all data structures maintained by PROCESSREADENTITY and PROCESSUPDATEENTITY can be implemented with simple hash functions. Thus, building the dependency graph is linear with the number of operations.

## 14 Proofs

For the following proofs, recall that $s_i < c_j$ if $T_i$ starts before $T_j$ commits, and $c_j \leq s_i$ if it starts after $T_j$ committed. Obviously, any read/write operation of $T_i$ occurs between its start and its commit, i.e., $s_i < r_i/w_i < c_i$.

### 14.1 Proof of THEOREM 5.2

**Lemma 14.1** *For any* RC+ *history, if there is a $wr$- or a $ww$-edge from $T_i$ to $T_j$, then $c_i < c_j$.*

*Proof* For $T_i - ww \rightarrow T_j$, this is true by definition of this edge type. For $T_i - wr \rightarrow T_j$, RC+ requires a read operation $r_j$ to only read a committed value $x_i$. Thus, $c_i < r_j$, and hence $c_i < c_j$ □

**Lemma 14.2** *For any* RC+ *history, if there is a $rw$-edge from $T_i$ to $T_j$ then $s_i < c_j$.*

*Proof* For $T_i - rw \rightarrow T_j$, then there is a read operation $r_i$ that reads a value written by committed transaction $T_k$ and $T_j$ is the next transaction to write $x$ and commit. As RC+ reads the last committed version as of start of operation, this means the read of $T_i$ must be after the commit of $T_k$ but before the commit of $T_j$, i.e., $c_k < r_i < c_j$. Therefore, $s_i < c_j$. □

**Lemma 14.3** *Under a* RC+ *history, if there is an edge (of any type) between $T_i$ and $T_j$ then $s_i < c_j$.*

*Proof* It is a straightforward conclusion from Lemma 14.1 and Lemma 14.2. □

*Proof of Theorem 5.2* We denote by $T_3$ the first committing transaction in a given cycle $C$, and $T_2$ the predecessor of $T_3$ and $T_1$ the predecessor of $T_2$.

We first show that $T_2 \parallel T_3$ and $T_2 \parallel T_1$. Assume that $T_2 \nparallel T_3$ ($\nparallel$ means not concurrent), then either (1) $c_2 < s_3$ or (2) $c_3 < s_2$. (1) is not possible, since $c_2 < s_3$ implies $c_2 < s_3 < c_3$, thus $c_2 < c_3$ which contradicts that $T_3$ is the first committer in $C$. Based on Lemma 14.3, since there is an edge from $T_2$ to $T_3$, we have $s_2 < c_3$ and thus (2) is also not possible. As neither (1) nor (2) is possible, $T_2$ and $T_3$ must be concurrent. Now, assume that $T_2 \nparallel T_1$, meaning either (3) $c_2 < s_1$ or (4) $c_1 < s_2$. (3) is impossible as there is an edge from $T_1$ to $T_2$ which requires $s_1 < c_2$ based on Lemma 14.3. The edge from $T_2$ to $T_3$ requires $s_2 < c_3$. If we assume (4) holds, then we have $c_1 < s_2 < c_3$ which contradicts the fact that $T_3$ is the first committer in $C$. As neither (3) nor (4) are possible, $T_1$ and $T_2$ must be concurrent. □

Finally, assume that the edge from $T_2$ to $T_3$ is not a $rw$-edge but a $wr$- or $ww$-edge. Based on Lemma 14.1, this requires $c_2 < c_3$. But this contradicts the fact that $T_3$ is the first committed in $C$.

### 14.2 Proof of LEMMA 5.1

Given an *edge* from a transaction $T_i$ to a transaction $T_j$ in a dependency graph generated under RC+, such as $c_j < c_i$ ($T_i$ commits after $T_j$). Let us assume that the edge from $T_i$ to $T_j$ is a $wr$ ow $ww$-edge. Based on Lemma 14.1, we will have $c_i < c_j$ which contradicts the fact that $T_i$ commits after $T_j$ ($c_j < c_i$). Therefore, the edge from $T_i$ to $T_j$ can only be of type $rw$. Based on Lemma 14.2, $T_i - rw \rightarrow T_j$ implies that $s_i < c_j$. And since $c_j < c_i$, we get $s_i < c_j < c_i$ which means $T_i$ is concurrent to $T_j$.

## References


1. Adya, A.: Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions. PhD thesis, MIT (1999)
2. Adya, A., Liskov, B., O'Neil, P.E.: Generalized isolation level definitions. In IEEE International Conference on Data Engineering (ICDE), pp. 67–78 (2000)
3. Amza, C., Chanda, A., Cox, A.L., Elnikety, S., Gil, R., Rajamani, K., Zwaenepoel, W.: Specification and implementation of dynamic web site benchmarks. In Workshop on Workload Characterization (WWC), pp. 3–13 (2002)
4. Berenson, H., Bernstein, P.A., Gray, J., Melton, J., O'Neil E.J., O'Neil, P.E.: A critique of ANSI SQL isolation levels. In ACM SIGMOD International Conference on Management of Data, pp. 1–10 (1995)
5. Bernstein, A., Lewis, P., Lu, S.: Semantic conditions for correctness at different isolation levels. In IEEE International Conference on Data Engineering (ICDE), pp. 57–66. IEEE (2000)
6. Bernstein, P.A., Fekete, A., Guo, H., Ramakrishnan, R., Tamma, P.: Relaxed-currency serializability for middle-tier caching and replication. In ACM SIGMOD International Conference on Management of Data, pp. 599–610 (2006)
7. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison Wesley, Reading (1987)
8. Cahill, M.J., Röhm, U., Fekete, A.D.: Serializable isolation for snapshot databases. ACM Trans. Database Syst. (TODS), **34**(4), 20:1–20:42 (2009)
9. Casanova, M.A.: The Concurrency Control Problem for Database Systems, Lecture Notes in Computer Science, vol. 116. Springer, Berlin (1981)
10. Daudjee, K., Salem K.: Inferring a serialization order for distributed transactions. In IEEE International Conference on Data Engineering (ICDE), p. 154 (2006)
11. Elnikety, S., Dropsho, S.G., Pedone, F. (2006) Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In EuroSys Conference, pp. 117–130
12. Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: The notions of consistency and predicate locks in a database system. Communications of the ACM **19**(11), 624–633 (1976)
13. Fekete, A.: Serialisability and snapshot isolation. In The Australian Database Conference (ADC), pp. 201–210 (1999)
14. Fekete, A., Goldrei, S., Asenjo, J.P.: Quantifying isolation anomalies. Int. Conf. Very Large Data Bases (VLDB) **2**(1), 467–478 (2009)
15. Fekete, A., Liarokapis, D., O'Neil, E.J., O'Neil, P.E., Shasha, D.: Making snapshot isolation serializable. ACM Trans. Database Syst. (TODS) **30**(2), 492–528 (2005)
16. Georgakopoulos, D., Rusinkiewicz, M., Sheth, A.P.: On serializability of multidatabase transactions through forced local conflicts. In IEEE International Conference on Data Engineering (ICDE), pp. 314–323 (1991)
17. Jorwekar, S., Fekete, A., Ramamritham, K., Sudarshan, S.: Automating the detection of snapshot isolation anomalies. In International Conference on Very Large Data Bases (VLDB), pp. 1263–1274 (2007)
18. JPA. The Java Persistence API, Sun Microsystems, JSR 220. http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html
19. Kraska, T., Hentschel, M., Alonso, G., Kossmann, D.: Consistency rationing in the cloud: Pay only when it matters. Int. Conf. Very Large Data Bases (VLDB) **2**(1), 253–264 (2009)
20. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM Trans. Database Syst. (TODS) **6**(2), 213–226 (1981)
21. Paulley, G.: The second deadly sin part un. PowerBuilder Developer's Journal, July 2012, 19(7), pp. 6–9, A similar version appeared as blog at http://iablog.sybase.com/paulley/2012/05/the-second-deadly-sin-part-un/
22. Ports, D.R.K., Grittner, K.: Serializable snapshot isolation in postgresql. In International Conference on Very Large Data Bases (VLDB) (2012)
23. Revilak, S., O'Neil, P.E., O'Neil, E.J.: Precisely serializable snapshot isolation (pssi). In IEEE International Conference on Data Engineering (ICDE), pp. 482–493 (2011)
24. Röhm, U., Böhm, K., Schek, H.-J., Schuldt, H.: Fas - a freshness-sensitive coordination middleware for a cluster of olap components. In International Conference on Very Large Data Bases (VLDB), pp. 754–765 (2002)
25. Röhm, U., Schmidt, S.: Freshness-aware caching in a cluster of J2EE application servers. In International Conference on Web Information Systems Engineering (WISE), pp. 74–86 (2007)
26. SPECjEnterprise2010. A performance benchmark for Java Enterprise Edition (JavaEE) 5 or later application servers. http://www.spec.org/jEnterprise2010/
27. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. (SICOMP) **1**(2), 146–160 (1972)
28. TPC-C.: An on-line transaction processing benchmark. http://www.tpc.org/tpcc/
29. Yabandeh, M., Ferro, D.G.: A critique of snapshot isolation. In EuroSys Conference, pp. 155–168 (2012)
30. Zellag, K.: On the Detection and Prevention of Consistency Anomalies in Multi-tier and Cloud Applications. PhD thesis, McGill University (2012)
31. Zellag, K., Kemme, B.: Real-time quantification and classification of consistency anomalies in multi-tier architectures. In IEEE International Conference on Data Engineering (ICDE), pp. 613–624 (2011)
32. Zellag, K., Kemme, B.: Consad: a real-time consistency anomalies detector. In ACM SIGMOD International Conference on Management of Data, pp. 641–644 (2012)
33. Zellag, K., Kemme, B.: How consistent is your cloud application? In ACM Symposium on Cloud Computing (SoCC), pp. 6:1–6:14 (2012)