Real-Time Quantification and Classification of Consistency Anomalies in Multi-tier Architectures

Kamal Zellag #1, Bettina Kemme #2

School of Computer Science, McGill University
Montreal, Canada
[1] zkamal@cs.mcgill.ca
[2] kemme@cs.mcgill.ca

***Abstract*—While online transaction processing applications heavily rely on the transactional properties provided by the underlying infrastructure, they often choose to not use the highest isolation level, i.e., serializability, because of the potential performance implications of costly strict two-phase locking concurrency control. Instead, 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 the anomalies that are possible under a certain level of isolation, it is much more difficult to *quantify* the amount of anomalies that occur during run-time of a given application. In this paper, we address this issue and present a new approach to detect, in real-time, consistency anomalies for arbitrary multi-tier applications. As the application is running, our tool detect anomalies online indicating exactly the transactions and data items involved. Furthermore, we classify the detected anomalies into patterns showing the business methods involved as well as their occurrence frequency. We use the RUBiS benchmark to show how the introduction of a new transaction type can have a dramatic effect on the number of anomalies for certain isolation levels, and how our tool can quickly detect such problem transactions. Therefore, our system 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.**

I. 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 3-tier architecture, where clients (e.g., web browser) first send requests to a web-server, which processes the presentation logic. Then, the request is passed to an application server, to which we also refer as middle-tier, 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 has become popular which offers more isolation than read committed but less than serializability. The choice between these isolation levels depends on performance and consistency needs. Selecting an isolation level with strict consistency can result in performance penalty, while lower levels of isolation provide more concurrency at the cost of potential inconsistencies.

Several studies have analyzed the potential anomalies associated with the existing isolation levels [1], [2], [3], [4], [5]. However, it is less clear whether a given application might actually experience such anomalies and if yes, to what degree. For example, *lost-updates* are allowed under *Read-Committed*, but the actual occurrence can vary widely depending on the application. Mechanisms presented in [6], [7] detect and/or avoid potential anomalies for applications that run under snapshot isolation. 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 that occur during run-time is very desirable as it can help to decide which isolation level to use for the given application. Moreover, detecting such anomalies in real-time is extremely useful for early diagnosis and immediate intervention.

In this work, we propose a new approach for detecting in real-time consistency anomalies that may occur during the run-time of an arbitrary multi-tier application. Our approach does not require any knowledge about the studied application and supports any isolation level that is higher than or equal to read-committed. It is implemented in form of a consistency anomaly detector tool (ConsAD) consisting of two agents COLAGENT and DETAGENT. COLAGENT observes the execution and keeps track of the data items accessed by transactions and their execution order. COLAGENT is integrated into the middle-tier layer and is completely independent of the database system. COLAGENT currently supports any multi-tier platform that is conform to the JavaEE (Java Enterprise Edition) standard, both for single-server as well as clustered configurations.

COLAGENT'S observations are forwarded to DETAGENT which performs the anomaly detection. DETAGENT is completely independent from the multi-tier architecture and can run on the same or a different machine. DETAGENT uses COLAGENT'S observations to dynamically build serialization graphs, and detects anomalies by finding cycles in the graph.

It takes advantage of particular properties of the different isolation levels in order to help speed up graph creation. For each detected cycle, DETAGENT generates detailed information about the transactions and the affected entities. It also classifies the detected anomalies into *patterns of anomalies*, where each pattern has a different set of business methods that are involved.

We have integrated COLAGENT into Hibernate[1], a default persistence layer for the open-source application server JBoss[2]. The system uses PostgreSQL[3] as underlying database system. We have tested ConsAD on an extension of the RUBiS benchmark [8], an auction based application, under three isolation levels: *Read Committed (RC)*, *Snapshot Isolation (SI)*, and *JOCC*, an optimistic concurrency control mechanism common in many middle-tier servers. We have tested the system for a single application server as well as clustered configurations. Our analysis shows that COLAGENT's overhead during runtime is very low and cycle detection under DETAGENT works efficiently even if the serialization graph contains tens of thousands of transactions. Furthermore, DETAGENT enables a detailed analysis of the number of anomalies that occur for RUBiS under the various isolation levels. In fact, our analysis shows that the injection of a single new transaction type into an existing application can completely change the number of anomalies that occur under the different isolation levels. Thus, the tool is extremely helpful during the development and maintenance of an application in order to see which isolation level provides an acceptable amount of anomalies and which transaction types produce problem cases.

In summary the paper makes the following contributions:

- We present a new approach to detect, in real-time, consistency anomalies of arbitrary multi-tier applications running under any isolation level that is higher than or equal to read-committed.
- We classify the detected anomalies into patterns of anomalies that show exactly which business methods are involved in each type of anomaly as well as their occurrence frequency.
- We provide a complete consistency anomaly detector tool (ConsAD) which supports multi-tier platforms. The approach is completely database independent. It conforms to the JavaEE standard, and can be extended to cover additional platforms by adjusting a single component, namely the COLAGENT.

II. BACKGROUND AND PRELIMINARIES

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

A. Transactions and Isolation Levels

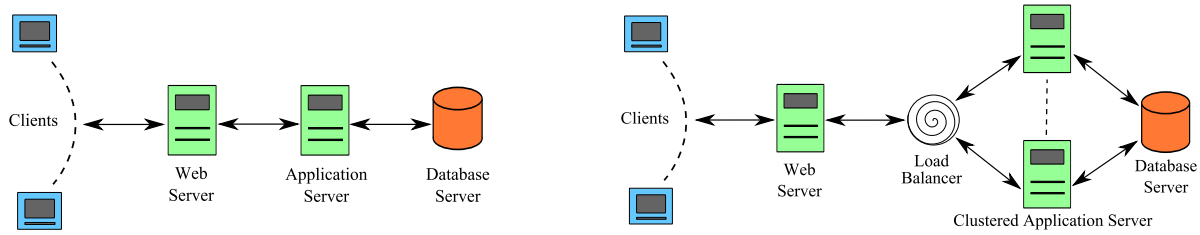
A *transaction* T_i is a logical unit of read and write operations on data objects. T_i runs atomically, i.e., either all operations succeed and T_i commits (c_i), or none of its operations succed and it aborts (a_i). Two transactions are concurrent if neither terminates (commit/abort) before the other starts. Following [9], we assume that each write operation $w_i(x)$ of transaction T_i on data item x generates a new version x_i of x. We say that x_i is *installed* when T_i commits. The commit order of transactions defines a total order on all object 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. A *schedule* for a set of transactions $\mathcal{T}$ describes the order in which the operations of the transactions in $\mathcal{T}$ are executed.

Concurrent transactions must be isolated from each other. The main correctness criteria in the research community is serializability. It requires a schedule S with interleaved execution of transactions to be conflict-equivalent to a serial schedule over the same set of transactions, meaning that the order of any pair of read/write or write/write operations on the same data item (called *conflicting* operations) in S must be the same as in a serial schedule where transactions execute serially one after the other.

A serialization graph is a directed graph where nodes are the committed transactions of a schedule 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*. A schedule is serializable if and only if its serialization graph is acyclic [10].

Commercial database systems do not use this formal definition of serializability but rely on the ANSI SQL isolation levels. They define anomalies such as *dirty-read* (a transaction reads uncommitted updates), *unrepeatable-read* (two consecutive reads of the same transaction return different versions), or *lost update* (a transaction performs first a read and then later a write on the same entity, while another transaction updates this entity between the read and the write operation). A concurrency control mechanism is said to belong to a certain isolation level if it avoids a specific set of these anomalies. We indicate an isolation level that only reads committed versions of entities, and thus, avoids dirty-read, as a RC isolation level, and by NOLOSTUPD any isolation level that is conform to RC and avoids lost-updates. At the time isolation levels were defined, all existing concurrency control mechanisms that avoided all anomalies provided serializability. However, since then, several concurrency control mechanisms have been developed that avoid the ANSI anomalies but are not serializable. We discuss one of them later.

[1]http://www.hibernate.org/
[2]http://www.jboss.org/
[3]http://www.postgresql.org/



(a) stand-alone middle-tier server (b) clustered middle-tier

Fig. 1. Multi-tier Architecture

B. Multi-tier Architectures

Many web-based information systems follow a multi-tier architecture as depicted in Figure 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. Fig. 1(a) shows a stand-alone middle-tier configuration with one application server while in Fig. 1(b) 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 Platform Enterprise Edition (Java EE) standard as many of the major industrial application servers and several popular open-source products, such as Sun GlassFish[4] and JBoss[5], 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 needs to be persisted in a database system.

C. Persistence Layer

In JavaEE, persistence is managed by the *Java Persistence API* (JPA)[11]. JBoss Hibernate[6] and Oracle TopLink[7] 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.

D. 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 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. Given that each transaction caches its own local copies of entities, the middle-tier requires some form of concurrency control to provide the appropriate isolation level. In this paper, we look at three isolation levels. Read-Committed (RC) and Snapshot Isolation (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. All three middle-tier algorithms rely on the fact that the database system sets long exclusive locks for updates, that is, the database records in the database can never be updated concurrently by different transactions.

Read-Committed (RC). This isolation level is widely used in database systems. Thus, it is not surprising that it is also offered at the middle-tier. A read operation $r(x)$ accesses the last committed version of x as of the start time of the operation. Generally, two consecutive read operations of the same transaction on x may see different versions if another transaction writes x and commits in between. Thus, unrepeatable reads, lost updates and also other anomalies can occur [12]. One possible implementation of RC at the middle-tier simply runs the database transaction T_{DB} under the RC level guaranteeing that whenever an entity x is loaded into the middle-tier, the last committed version of x is returned and cached as loc_x. Updates are always done first on this local copy loc_x. At the end of T_{MT}, if x was updated, the new version stored in loc_x is sent to the database. Once all writes have been successfully executed at the database, both T_{MT} and T_{DB} commit. Interesting to note is that whenever the middle-tier transaction T_{MT} reads an entity several times, it always accesses the cached version loc_x. Thus, in contrast to traditional RC, most reads are repeatable. However, unrepeatable read 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.

JPA OCC (JOCC). As described in [11], JPA implementations provide an Optimistic Concurrency Control which we denote as JOCC (JPA OCC). It is different to traditional textbook optimistic concurrency control [13] in that it does not provide serializability. The reason is that it only detects

[4]http://glassfish.dev.java.net/
[5]http://www.jboss.org/
[6]http://www.hibernate.org/
[7]http://www.oracle.com/technology/products/ias/toplink/

conflicts between write operations. JOCC assumes that the database transaction T_{DB} runs under the RC isolation level. In order to detect conflicts between writers, it enforces a transaction T_{MT} (and its corresponding database transaction T_{DB}) to abort if any entity x that T_{MT} changed was updated since T_{MT} read it into its local copy loc_x. JOCC implements this conflict detection 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_{MT}, its version is increased and at commit time of T_{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_{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_{MT} is aborted. To get a closer look on how this works at the SQL level under Hibernate (a JPA implementation), here is a concrete example. Assume that a transaction T_{MT} loads a *Product* from the database with a *productId* equal to 100 and *objVersion* equal to 1. It then modifies its price to the value 75.0. At the commit time of T_{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 another 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_{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, the same problem with complex reads as with RC can occur, as well as other non-serializable schedules.

Snapshot Isolation (SI). Snapshot Isolation has been a popular isolation level within the database system 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 is 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. Although SI avoids all ANSI anomalies, is allows for some non-serializable schedules, as discussed in [12] and [14]. SI is simple to achieve at the middle-tier. The only difference is that the DB transaction must run under the SI isolation level instead of RC. Of course, this assume that the database provides SI. 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.

Remark. The middle-tier often provides an optional internal cache known as a *second level cache*. It is shared by all middle-tier transactions. When such a cache is in place, extra concurrency control mechanisms are needed at the middle-tier. Current implementations of JavaEE can handle the RC and JOCC properties. We are not aware of any commercial system offering SI, but such SI caching was developed in [15]. We have designed and implemented a variation of that mechanism which can be plugged into any JavaEE compliant middle-tier. However, due to space limitations, we present the discussion in this paper without any further consideration of caching.

III. Detection of Anomalies

The goal of our system is to be able to detect anomalies that occur when arbitrary applications run with any of these concurrency control algorithms. The challenge is to detect these anomalies at the middle-tier layer. An anomaly occurs when the execution is unserializable, that is when the serialization graph has a cycle. Thus, in the remainder of this paper, we use the words *anomaly* and *cycle* interchangeably.

ConsAD system consists of two agents. The collector agent COLAGENT collects information about transactions and their accessed entities while the detector agent DELAGENT determines all *read-dependencies*, *write-dependencies* and *anti-dependencies* and detects cycles.

A. The Collector Agent

The collector agent is integrated within the middle-tier application server and intercepts any read or write access to any entity by any middle-tier transaction. One instance of COLAGENT is installed on each middle-tier server. COLAGENT must extract enough information so that DETAGENT can determine all dependencies.

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 track entity versions by adding an extra $txnInfo$ attribute to each entity class. This is similar in spirit to the version attribute used by JOCC and completely transparent to the application developer. Before an entity x is written to the database, COLAGENT sets $x.txnInfo$ to the identifier i of the writing transaction T_i. 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. This means that if T_j loads a version x_i, then a wr-dependency from T_i to T_j can be established.

Write-dependencies. ww-edges are built between transactions that install consecutive versions of an entity. For that, COLAGENT must know the commit order for any two transactions that update the same entity. There exist many different approaches, e.g., [16], [17], [18], that discuss how to extract or enforce the commit order of transactions. However, these approaches require access to the database log, enforce a commit order at the database, or enforce the commit order at the middle-tier. In contrast, we are able to conclude in some cases the version order at the middle-tier by exploiting some properties of the system. First, as we discussed in Section II-C, under JPA a transaction T_j must first load an entity x before it can update it (i.e., there are no blind writes). Second, we can determine the transaction T_i that created the version of x loaded by T_j through the $txnInfo$ attribute. Third, under NOLOSTUPD isolation levels such as SI and JOCC, 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. This leads to the following proposition.

Proposition 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.

Proof. Since NOLOSTUPD avoids lost updates, it is guaranteed that no other transaction T_k has created a version x_k and committed between $r_j(x_i)$ and commit of T_j. At commit time of T_j, x_j is installed, and since no version has been installed between x_i and x_j, x_j is the immediate successor of x_i.

However, if the isolation level allows lost-updates, such as possible with RC, Proposition 1 is not valid. In this case, COLAGENT enforces a commit order for update transaction in order to be able to deduce ww-edges. We offer two implementations that provide each update transaction T with a unique and sequential commit order denotes as $T.co$. The first solution is a memory based counter. In the first solution, at commit time of an update transaction T, COLAGENT get an exclusive lock on a memory-based counter, increments it and assigns the new value to $T.co$. The lock is only released after T committed at the database. However, this solution works only when a stand-alone middle-tier is used as shown in Fig. 1(a). The second solution is a database counter, that is accessed in an exclusive way using the SELECT FOR UPDATE statement and works for both stand-alone and clustered middle-tier configurations as shown in Fig 1.a and Fig 1.b. More details about those two solutions will be provided in Section V. Alternatively, we could have implemented any of the solutions that have been proposed in the past [16], [18].

Therefore, under the RC isolation level ww-edges can be derived using the following property.

Proposition 2 Assuming isolation level RC and 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) $T_i.co < T_j.co$, and (iii) not $\exists T_k$ such that T_k created version x_k and $T_i.co < T_k.co < T_j.co$. Then x_j is immediate successor of x_i.

In summary, if transactions are running under a NOLOSTUPD isolation level, the ww-edges can be detected by using Proposition 1. However, if they are running under RC, Proposition 2 can be used.

Anti-dependencies. There is a rw-edge from T_k to T_j if there is an entity x for which T_k has read the version x_i (created by another transaction T_i) and later T_j creates the immediate successor x_j of x_i. Note that a rw-edge exists between T_k and T_j only if there is a wr-edge from T_i to T_k via x and ww-edge from T_i to T_j via the same entity x. Thus, once the wr and ww edges are present, the conclusion of the rw-edge is immediate.

Summary. COLAGENT keeps track of all entities read and all entities updated by T_i. At commit time of each transaction T, COLAGENT overwrites $txnInfo$ of each entity updated by T_i with T_i's identifier. Additionally, in case of RC, it guarantees a commit order and generates $T_i.co$ for each update transaction. After T_i has successfully committed COLAGENT sends the following information to DETAGENT. For each entity x read, it provides 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. It also indicates the business method which initiated the call to the entity. Note that JPA provides annotations that help to 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 order to know the structure of the primary key.

B. The Detector Agent

The detector agent DETAGENT takes the information collected by COLAGENT and dynamically builds the graph and detects cycles. As discussed earlier wr-edges can be detected by using the field $txnInfo$. Each rw-edge requires first the existence of one wr-edge and one ww-edge. However, the detection of a ww-edge depends on the isolation level used in place. Under NOLOSTUPD we use Proposition 1 to derive ww-edges between transactions, while under RC we use Proposition 2 to extract such edges. Thus, while DETAGENT is nearly the same for all isolation levels, the steps to determine ww-dependencies differ for NOLOSTUPD and RC.

DETAGENT builds the serialization graph as it receives transaction information from COLAGENT. For each transaction T, whether read-only or update transaction, DETAGENT calls the method PROCESSTX as shown in Algorithm 1. However, depending on the isolation level, the method might be called at different time-points. PROCESSTX is called for all transactions immediately after being received from COLAGENT, except for update transactions under RC. They have to be queued until all previously committed update transactions are received and processed. This means, under RC, DETAGENT must process update transactions in commit order in order to correctly derive the ww-dependencies by the means of Proposition 2. In all other cases (read-only transactions or any transaction type for NOLOSTUPD) the arrival order does not matter. For any dependency edge $T_i \rightarrow T_j$, DETAGENT builds the edge when the second of the two transactions arrives. In the case of RC and update transactions, T_j will always be the second, otherwise it might be T_i or T_j.

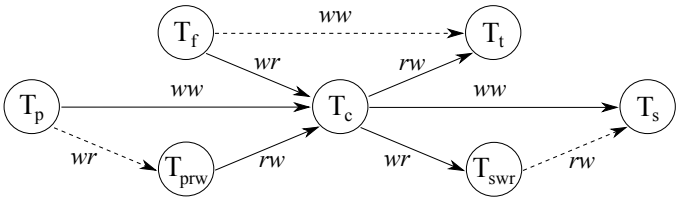


Fig. 2. Incoming and outgoing edges for transaction T_c

In case of a single middle-tier server, a simple FIFO channel between COLAGENT and DETAGENT is enough to send update transactions in commit order. However, in clustered middle-tier servers out-of-order delivery might occur. Nevertheless, in our experiments, transactions had to be queued for only negligible time.

PROCESSTX as shown in Algorithm 1 takes the information of a transaction T_c as input, builds additional edges where T_c is involved, and then searches for possible cycles starting from T_c. 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. Each additional edge extracted during the calls to PROCESSREADENTITY and PROCESSUPDATEDENTITY is added to the global serialization graph which contains all previously processed transactions. After that, PROCESSTX calls DETECTCYCLES which starts its search of cycles from 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 2 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 an outgoing $T_c - ww \rightarrow T_s$ edge to successor transaction T_s that creates the successor version x_s of x_p (unless T_c deletes the entity 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).

PROCESSREADENTITY. This algorithm aims in building the incoming wr-edge and the outgoing rw-edge for T_c in regard to an entity x. If first extracts the identifier of T_f from the field $x.txnInfo$ (line 1). T_f might have arrived at 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 arrives, T_c is added to a list that contains all readers of version x_f (lines 4-5).

In order to create the $T_c - rw \rightarrow T_t$, 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 15). Again, T_t might have arrived at 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. Again, for this to happen, we have to keep track that T_c read version x_f (lines 9-10).

PROCESSUPDATEDENTITY. This algorithm is called for each update on entity x performed by update transaction T_c. Under RC, it is called only if all update transactions with commit order lower than $T_c.co$ have been processed. In contrast, such 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

Algorithm 1 PROCESSTX (T_c)

Require: a transaction T_c
Ensure: $builds$ additional edges for T_c, then $detects$ possible cycles starting from T_c.
1: **for** x **in** $T_c.Entities$ **do**
2: PROCESSREADENTITY(x, T_c)
3: **for** x **in** $T_c.Entities$ where $x.flag = TRUE$ **do**
4: PROCESSUPDATEDENTITY(x, T_c)
5: DETECTCYCLES(T_c)

Algorithm 2 PROCESSREADENTITY (x , T_c)

Require: an entity x read by a transaction T_c
Ensure: $builds$ additional wr- and rw-edges for T_c
1: $T_f\_id = x.txnInfo$
2: **if** T_f was processed **then**
3: $create$ an wr-edge : $T_f - wr \rightarrow T_c$
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, see line 19 in Algorithm 3
6: $T_t\_id = getSuccessorOf(x.key, T_f\_id)$
7: **if** T_t was processed **then**
8: $create$ a new rw-edge : $T_c - rw \rightarrow T_t$
9: **else**
10: add T_c to $readersOf(x.key, x.txnInfo)$ // edge will be created once T_t is processed as an update-transaction, see line 17 in Algorithm 3

Algorithm 3 PROCESSUPDATEDENTITY (x , T_c)

Require: an entity x created by a transaction T_c
Ensure: $builds$ additional ww-, wr- and rw-edges for T_c

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 was processed // always true with RC **then**
7: $create$ ww-edge : $T_p - ww \rightarrow T_c$
8: **else**
9: do nothing, it will be created once T_p is processed, see line 12
10: $T_s\_id = getSuccessorOf(x.key, T_c\_id)$
11: **if** T_s was processed // never occurs with RC **then**
12: $create$ ww-edge : $T_c - ww \rightarrow T_s$
13: **else**
14: do nothing, it will be created once T_p is processed, see line 7
15: add $[(x.key, T_p\_id), T_c\_id]$ **to** $successors$
16: **for each** $T_{prw} \in readersOf(x.key, T_p\_id)$ **do**
17: $create$ rw-edge : $T_{prw} - rw \rightarrow T_c$
18: **for each** $T_{swr} \in readersOf(x.key, T_c\_id)$ **do**
19: $create$ wr-edge : $T_c - wr \rightarrow T_{swr}$

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 1 guarantees that there is no other version in between. In case of RC, the system keeps track for each entity x of the last processed transaction that updates x. As update transactions are processed in commit order and Proposition 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 for RC), an $T_p - ww \rightarrow T_c$ edge is immediately built. If T_p has not yet been processed no action is taken. This edge will be created once PROCESSUPDATEDENTITY is called for x and T_p. After that, this algorithm checks if the successor T_s of T_c was already processed before T_c (lines 10-14). This is only possible for NOLOSTUPD. If this is the case, then an $T_c - ww \rightarrow T_s$ is created. Also, the successor structure is updated as it is needed if any transaction T with dependency $T \rightarrow T_c$ (ww or rw-dependency) has not yet arrived (line 15).

From line 16 to line 19, missing wr- and rw-edges are built. These are edges where the reading transaction arrived before the writing transaction, and thus, could not be built during PROCESSREADENTITY.

DETECTCYCLES. After the edges for transaction T_c are added, PROCESSTX calls the cycle detection algorithm with T_c as input. It implements an extension $extDLS$ of the Depth-Limited-Search (DLS) algorithm [19], which does a depth-first search and stops at a certain depth limit. Our choice for DLS derives from the experience that cycles in serialization graphs do usually contain only a limited number of transactions, thus specifying a reasonably small DLS depth works well enough for our purposes. For this, we have defined a configurable parameter in DETAGENT that limits the depth of search to a length d. The algorithm $extDLS$ starts from T and explores all its outgoing edges. At the end, it returns all possible paths starting from T with a size smaller than d. Note that duplicated nodes are not allowed by $extDLS$. If the last transaction (node) T_L, in a certain extracted path, has any outgoing edge to T, then a cycle is detected.

C. Correctness

Detection of edges. 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 serialization graph at the time the second of the two transactions is processed. Assume without loss of generality that $T_i \rightarrow T_j$ due to entity x. We consider several cases.

1. T_i is delivered 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 delivered 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 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.

Detection of cycles. It is easy to see that each cycle C is detected once the last transaction in the cycle has been processed. Assume a cycle $C = T_1 \rightarrow T_2 ..... T_n \rightarrow T_1$ involving n transactions and n edges in the serialization 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. Thus, once all transactions are processed, all edges in the cycle have been constructed. Without 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 PROCESSTX for T_n calls DETECTCYCLES, the cycle will be detected as long as the length d of $extDLS$ is at least n.

D. Complexity

Each edge is added exactly once. All data structures maintained by PROCESSREADENTITY and PROCESSUPDATEEN-

TITY can be implemented with simple hash-functions. Thus, building the graph is linear with the number of operations.

For the cycle detection, given a node as starting point, DLS has a complexity of b^d, where b is the branching factor (outgoing edges) and d is the depth. In order to determine b, we have to know the number of edges for each transaction. This depends on the number of operations. For simplicity we assume no inserts and no deletes and each entity is updated on a regular basis. Then, for each write operation, a transaction T has an incoming ww-edge and an outgoing ww-edge. Thus, if we assume an average of wt write operations per transaction we have wt incoming and wt outgoing ww-edges per transaction. For each read operation, a transaction T has an incoming wr-edge, and one outgoing rw-edge. Thus, if we assume an average of rt read operations per transaction we have rt incoming wr-edges and rt outgoing rw-edges per transaction. Additionally, a transaction might have incoming rw-edges and outgoing wr-edges caused by read operations of other transactions and its own writes. On average, there will be rt incoming rw-edges and rt outgoing wr-edges per transaction. Thus, a transaction has on average $wt + 2rt$ incoming, and $wt + 2rt$ outgoing edges. Therefore, for each transaction T, $extDLS$ explores at most $wt + 2rt$ outgoing edges (as not all outgoing edges might have been created when $extDLS$ is called for T). Then, recursively, from each of the reached successor nodes, it can explore at most another $wt + 2rt$ outgoing edges, etc. Thus, the complexity is $(wt + 2rt)^d$ per T,

In summary, the worst case complexity is exponential in the depth of the cycles. However, our experiments have shown that the depth can be kept low, as cycle lengths are typically short. Furthermore note that $extDLS$ is searching for paths by following outgoing edges of transactions. We have observed that in many cases (but of course not all cases) transactions are processed in dependency order. That is, whenever, there is a $T_i - ww/wr/rw \rightarrow T_j$ dependency, then DETAGENT often processes T_i before T_j. Therefore, when $extDLS$ is started for T_i, many outgoing edges do not yet exist (because they are only created when T_j is processed), and $extDLS$ terminates very quickly.

E. Discussion

Inserted and Deleted Entities. Insertions and deletions are handles 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 disappeares 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.

Applications. Note that we haven't specified any condition on the studied application. Instead, our system can work with arbitrary applications. All what we have added is an extra field to each entity class at the middle-tier which is not visible to the application designer. We have used a similar mechanism as is applied by the JOCC mechanism to create an extra version field. We used the ALTER command for each entity class for this purpose. Note that this field represent a 4 bytes number which makes it light on the database size and on entities in memory at the middle-tier.

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.

Mixture of Applications. Often, several applications can run over the same database. That is, 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 next section can tell exactly if a certain cycle has transactions triggered by methods of app_1 and/or app_2.

IV. PATTERNS CLASSIFICATION

So far, the information that DETAGENT provides for each cycle C contains the identifiers of the transaction involved, the edges between the transactions and their types, and the accessed entities. For each edge, DELAGENT can also indicate the business methods that were involved as COLAGENT provides this information.

However, if there are many different cycles, it is difficult to check what in the application actually caused these cycles. For that purpose, we provide a pattern classification module, that classifies detected cycles into patterns of cycles. In other words, for each detected cycle C, we extract the business methods that have created it and present the pattern of calls that lead to the cycle. With this, each cycle can be assigned to a cycle pattern. We classify the detected cycles in two ways. The first is $ordered$ while the second is $unordered$.

Ordered classification. In this classification, the order of business methods in each cycle in taken in consideration. Each cycle C composed by transactions is mapped to another cycle $ordC_{bus}$ composed by business methods that created the dependencies. For example, assume a $C = T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow T_1$. We replace now each transaction T_i with the business method bm_i called by T_i that caused the dependency, deriving $ordC_{bus} = bm_1 \rightarrow bm_2 \rightarrow ... \rightarrow bm_n \rightarrow bm_1$. Any other cycle of transactions that also maps to $ordC_{bus}$ belongs to the same cycle pattern.

Note that cycles $ordC_{bus1} = bm_1 \rightarrow bm_2 \rightarrow bm_3 \rightarrow bm_1$ and $ordC_{bus2} = bm_1 \rightarrow bm_3 \rightarrow bm_2 \rightarrow bm_1$ are different, since the order of methods are different. In fact, bm_1 has bm_3 as predecessor in $ordC_{bus1}$ while in $ordC_{bus2}$ the predecessor of bm_1 is bm_2.

Unordered classification. In this classification, the order of business methods is not important. Moreover, if a business

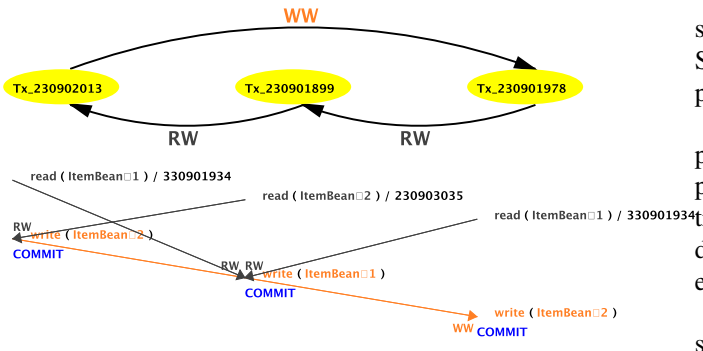


Fig. 3. A Cycle of size 3 under RC

method is involved many times in the same cycle, it is counted only as one node. With this, any cycle of transactions that maps to either C_1 or C_2 belongs to the same unordered pattern $unordPattern = \{bm_1, bm_2, bm_3\}$. Similarly, a cycle of transactions that maps to an ordered cycle pattern $ordC_{bus3} = bm_1 \rightarrow bm_3 \rightarrow bm_2 \rightarrow bm_1 \rightarrow bm_2$ has the same unordered pattern since duplications are not taken in consideration.

Pattern classification is also done online within the DETAGENT. After a transaction is processed and a new cycle of transactions is found, the cycle is mapped to an ordered cycle pattern and an unordered cycle pattern. If the pattern does not yet exist, it is first created and its counter is set to 1, otherwise, the counter for this pattern is incremented. Thus, DETAGENT keeps track of patterns, and how often each of them occurs.

The use of cycle patterns makes it easier to not only extract the business methods that cause cycles, but also the cycle frequency. As an example of using such patterns detection, in one of our experiments, DETAGENT detected 43 transaction cycles, that belonged to 14 different ordered patterns and to four different unordered patterns. Thus, in order to better understand what in the application causes the anomalies, the system administrator can first start checking the 4 unordered patterns, and if needed, continue checking ordered patterns or concrete transaction cycles.

V. IMPLEMENTATION

Agents. Both COLAGENT and DETAGENT are written in standard Java. While DETAGENT is independent of the middle-tier technology, 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 Both agents together build the Consistency Anomalies Detector Tool (ConsAD) which can be used with any multi-tier platform following the JavaEE standard. To support additional non-JavaEE platforms in ConsAD, only COLAGENT has to be changed, while DETAGENT remains the same.

In a stand-alone configuration (Fig. 1(a)), only one instance of COLAGENT is needed, while in a clustered configuration (Fig. 1(b)), each application server requires an instance of COLAGENT. Each COLAGENT instance sends its collected information to DETAGENT using TCP/IP sockets. Recall that COLAGENT stamps each updated entity with a unique transaction identifier. In order to guarantee uniqueness across several application server instances, we concatenate the server identifier to the locally generated transaction identifier. Server identifiers are added for each server as a configuration parameter.

Note that we do not change how the application server processes entities, how it handles the caches and how it performs concurrency control of cache coherency if multiple transactions or several application servers access the same data items. The only thing we do is capture what entities are executed.

By default, COLAGENT is disabled. It can be enabled by a special request sent from the DETAGENT console. COLAGENT does not store any information in memory, while DETAGENT maintains information about the serialization graph, transactions and their entities in LRU (Least Recently Used) structures. Such LRU structures have a maximum size, and if any new space is needed, the least recently used object is removed automatically. For example, if space is required in the memory assigned for the serialization graph, the edges that weren't traversed for the longest time will be deleted. While in theory it might be possible that we remove edges that later might be included in cycles, this is highly unlikely if memory sizes are large enough.

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. These sub-histories can be visualized graphically as shown in Figure 3. This figure shows a cycle between three transactions under RC. It shows as well the type of edges. The $txnField$ is shown at the end of each read. DETAGENT also continuously produces statistics such as the total number of cycles and their distribution grouped by size as well as their patterns.

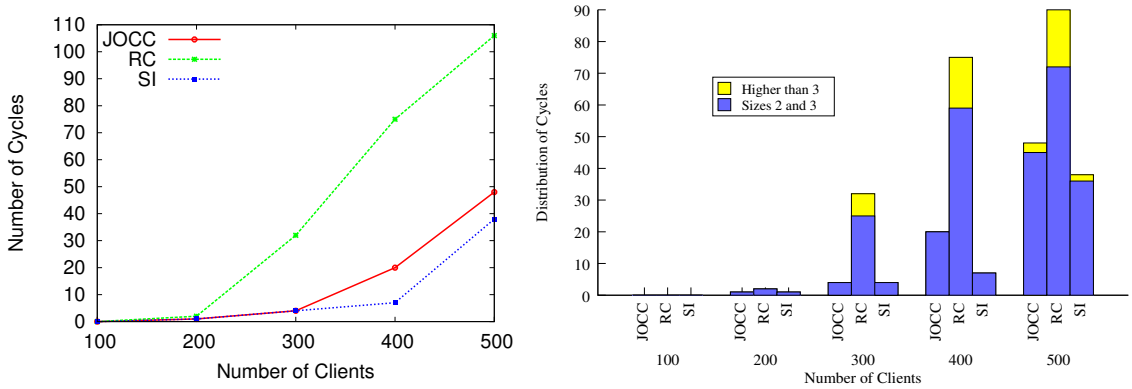
Commit Order. In order to provide transactions running under RC with a unique and sequential commit order, we have implemented two solutions. The first is memory-based, while the second is database-based. Note that we only serialize the commit requests, otherwise transactions run concurrently.

Our first solution uses a memory-based $COUNTER$ that is accessed in an exclusive write-lock by concurrent transactions. Before the commit request of each transaction T, it acquires an exclusive write-lock on COUNTER, assigns its value as commit order for T, then increases it. It only releases the COUNTER after the transaction has committed in the database. However, this solution works only in a stand-alone application server. In the second solution, we have created a database table with one record, which represents the COUNTER. To get an exclusive lock on COUNTER, we have used the SELECT FOR UPDATE query. Such query releases the lock on COUNTER only when the calling transaction has committed or aborted. This solution also works with a clustered middle-tier as in Fig. 1(b).

VI. EXPERIMENTS

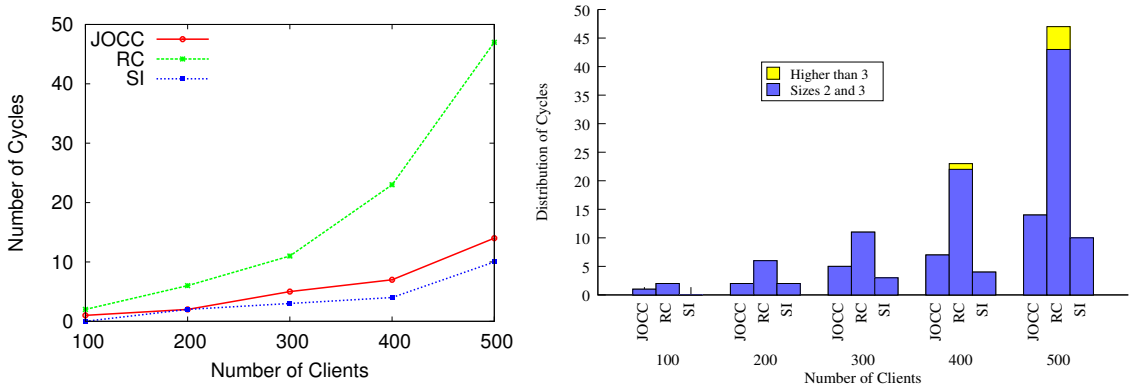
Application Under Test. To see how ConsAD works, we have used RUBiS [8], a popular benchmark that is similar to eBay[8].

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



(a) Detected Cycles (b) Distribution of Cycles

Fig. 4. Results for Stand-alone Configuration



(a) Detected Cycles (b) Distribution of Cycles

Fig. 5. Results for Clustered Configuration

RUBiS provides two workloads: *browsing-mix* and *bidding-mix*. While the first has only read-only transactions, the second provides read and write 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. Such behaviour can lead to non-serializable executions under all three isolation levels that we looked at.

Hardware and Software Configurations. We have conducted our tests under two multi-tier configurations: a stand-alone and clustered confugation, denoted as *stand-conf* and *clust-conf*, respectively. For the *stand-conf*, we have used one machine for the database server, one for the application server and one for RUBiS clients. For the *clust-conf*, we have used one machine for the database server, three machines for application servers one machine for RUBiS clients, and an additional machine for the load balancer that distributes RUBiS requests equally on the three application servers. All machines have the same hardware configuration and operating system. Each machine has 1GB of RAM and has an Intel Pentium D 2.8 as processor, and operates under the Ubuntu 10.04 operating system. We have used PostgreSQL8.4 as database server, JBoss6.0.0 as an application server with its persistence layer Hibernate3.5.0. Finally, we have used IPVS (IP Virtual Server) as a load balancer.

Results: We have performed several tests under two configurations. The first denoted as $config_1$ is without DailyDeal, which has 80% browsing-mix and 20% bidding-mix. In the second, denoted as $config_2$, we have used 50% browsing-mix, 20% bidding-mix and 30% DailyDeal. For the DailyDeal, we have selected 4 items. The tests were conducted under both *stand-conf* and *clust-conf*, by varying the number of remote clients (achieving up to 2500 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 minutes after a warm-up period of 10 minutes.

For $config_1$ we were unable to see any cycle under any isolation level for all evaluated throughput levels, thus we are not showing any figures related to tests under $config_1$. 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, as even lower isolation levels do not produce any unserializable executions.

In contrast, $config_2$ leads to cycles under all three isolation levels. Thus, we have a closer look at this configuration. We

have observed that 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 both settings, there is actually no performance difference between the different isolation levels. The reason might be that we use PostgreSQL which does not set shared locks, even for RC. Instead, it is a multi-version system. When run under RC, it reads versions as of start of operation; for SI, it reads version as of start of transaction. Thus, performance is similar. When the cache is enabled, SI performs slightly better than RCC and JOCC. Again, the reason could be that the SI cache provides multiple versions while in RCC and JOCC, short shared locks need to be set to control the access to the single object copy. Of course, given different hardware, 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.

Let's now have a look at the consistency analysis. Figure 4(a) shows the number of cycles with increasing number of clients for each isolation level under *stand-conf*, Figure 5(a) shows the same information for *clust-conf*. Figure 4(b) shows the cycle sizes with increasing number of clients for each isolation level under *stand-conf*. Figure 5(b) shows the same information for *clust-conf*. 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. The figures present the absolute number of cycles. To put this into perspective, during the runtime of the experience, between 10000 transactions (for 100 clients) and 55000 transactions (for 500 clients) are executed in total. The amount of aborted transactions under *stand-conf* 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. These numbers were decreased to about 40% to 45% under *clust-conf*.

We can observe that RC has generally a much higher number of cycles than JOCC and SI for both *stand-conf* and *clust-conf*. It quickly experiences a large number of cycles when it reaches the saturation point. It can be expected that RC has more unserializable executions since it is a lower level of isolation than JOCC and SI. A high proportion of cycles is either of size 2 or size 3 and this under both *stand-conf* or *clust-conf*. This confirms the assumption that consistency anomalies (cycles) involve generally a few number of transactions.

Note that we have less cycles under *clust-conf* for all tests and isolation levels. The reason is that response times are shorter under *clust-conf*. Thus, given a certain load, each transaction is concurrent with fewer other transactions reducing the chance of conflicts.

Although the number of cycles appears to be small before saturation, it affects approximately 2-3% of transactions which is significant. This is due to the fact that our daily deal only runs on 4 items. Thus, for our system under test, 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 anomalies, which is SI in our case.

For all our experiments, the number of ordered patterns was ranging from 1 to 17, while for unordered patterns, they were ranging from 1 to 5. In total, there were only five methods involved in all cycles detected in our experiments, and each pattern contained the DailyDeal business methods. The pattern visualization shows quickly how dependencies are generated between the business methods leading to cycles. In particular, two DailyDeal methods alone can already lead to a cycle, if one reads entities x and y and updates x, while another reads x and y and updates y. However, DailyDeal also triggers cycles with other methods that otherwise interacted cycle-free before. By looking at these cycles and their frequencies, the application designers can decide whether they can accept the inconsistencies, whether a higher level of isolation should be chosen, or whether the methods should be rewritten to avoid the anomalies.

Overhead: In order to check the overhead of ConsAD on the performance of applications, we have run several of the tests above with COLAGENT interception enabled and then disabled. The difference in response time was less than 3% for all selected tests, even for the RC tests where we had to execute commits serially. This confirms that the impact of interception is very low.

Cycle detection times are very short and for the experiments above it took never more than 0.1 second to detect cycles in which a transaction T is involved, even for graphs with more than 50.000 transactions. This is due to the fact that the cycle detection algorithm is based on the depth, and it rarely goes beyond depth five since it stops finding outgoing edges.

VII. RELATED WORK

Most related to our work is [20]. The authors provide quantitative information about the violations of integrity constraints when a database system runs under snapshot isolation or read committed. The analysis is performed over a microbenchmark consisting of two related tables. The authors were able to show that in some special cases SI leads to more anomalies than read committed. ConsAD differs from this work in several aspects. First, it is designed to work in a typical 3-tier architecture where execution is spread across middle- and database-tiers. Second, ConsAD works for any kind of application. It is an add-on tool for standard middle-tier servers and independent of the application. Third, it is not only able to quantify the anomalies that occur under read committed and SI, but it supports any isolation level higher than or equal to read-committed and classifies the detected anomalies into patterns. Moreover, the detection and classification in ConsAD are real-time.

Fekete et al. [6] present an approach that characterizes nonserializable 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. in [7] suggest a tool based on [6] 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 [6] and [7] 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 contrast, ConsAD supports multiple isolation levels, is application independent, real-time, completely automated, and provides quantitative information about anomalies as well as their patterns.

In [21], 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 [22], [23]. Our system 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.

We are not aware of older work quantifying the anomalies during the execution of a transactional workload. If it exists, it will very likely be limited to the database system and not consider execution across several tiers. We are also not aware of any recent research work or commercial products that build serialization graphs and perform cycle detection online.

VIII. Conclusion

This paper presented a new approach for quantifying and classifying isolation anomalies at the middle-tier layer of a multi-tier architecture. This approach is ensured in real-time by two agents integrated into one tool $ConsAD$ which supports the most common isolation levels found in current multi-tier architectures: snapshot isolation, a variant of optimistic concurrency control (JOCC) and read committed. It supports as well any other isolation level that is higher than read-committed. $ConsAD$ injects light-weight interception of transactional information during run-time of an application via its collector agent. Such information is sent in real-time to a remote detector agent which uses it to extract and classify anomalies. The worst case complexity for the cycle detection algorithm is exponential in the depth of the cycles and not related to the size of the serialization graph. For most of our tests, such depth was not exceeding the value 5.

We integrated $ConsAD$ into an open-source application server and used it to analyze RUBiS, a multi-tier benchmark. Our evaluation shows that the detector agent has very low overhead, The results show that for RUBiS most cycles are with 2 or 3 transactions only, and that SI experiences less anomalies than read committed and JOCC.

In our future work, we are planning to extend ConsAD 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 them.

References

[1] A. Adya, B. Liskov, and P. E. O'Neil, "Generalized isolation level definitions," in *ICDE*, 2000, pp. 67–78.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," in *ACM SIGMOD Conf.*, 1995.

[3] A. Bernstein, P. Lewis, and S. Lu, "Semantic conditions for correctness at different isolation levels," in *In Proceedings of IEEE International Conference on Data Engineering*. IEEE, 2000, pp. 57–66.

[4] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Commun. ACM*, vol. 19, no. 11, pp. 624–633, 1976.

[5] A. Fekete, "Serialisability and snapshot isolation," in *In proceedings of the Australian Database Conference*, 1999, pp. 201–210.

[6] A. Fekete, D. Liarokapis, E. J. O'Neil, P. E. O'Neil, and D. Shasha, "Making snapshot isolation serializable," *ACM Trans. Database Syst.*, vol. 30, no. 2, pp. 492–528, 2005.

[7] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan, "Automating the detection of snapshot isolation anomalies," in *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 1263–1274.

[8] C. Amza, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, K. Rajamani, and W. Zwaenepoel, "Specification and implementation of dynamic web site benchmarks," in *In 5th IEEE Workshop on Workload Characterization*, 2002, pp. 3–13.

[9] A. Adya, "Weak consistency: A generalized theory and optimistic implementations for distributed transactions," Ph.D. dissertation, MIT, Cambridge, 1999.

[10] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[11] Sun Microsystems, "The Java Persistence API, JSR 220," http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html.

[12] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil, "A critique of ansi sql isolation levels," in *Int. SIGMOD Conf.*, 1995.

[13] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, 1981.

[14] A. Fekete, E. O'Neil, and P. O'Neil, "A read-only transaction anomaly under snapshot isolation," *SIGMOD Rec.*, vol. 33, no. 3, pp. 12–14, 2004.

[15] F. Perez-Sorrosal, M. Patiño-Martínez, R. Jiménez-Peris, and B. Kemme, "Consistent and scalable cache replication for multi-tier J2EE applications," in *Int. Middleware Conf.*, 2007.

[16] D. Georgakopoulos, M. Rusinkiewicz, and A. P. Sheth, "On serializability of multidatabase transactions through forced local conflicts," in *ICDE*. Washington, DC, USA: IEEE Computer Society, 1991, pp. 314–323.

[17] K. Daudjee and K. Salem, "Inferring a serialization order for distributed transactions," in *IEEE International Conference on Data Engineering (ICDE'06)*, Apr. 2006.

[18] S. Elnikety, S. G. Dropsho, and F. Pedone, "Tashkent: uniting durability with transaction ordering for high-performance scalable database replication," in *EuroSys*, 2006, pp. 117–130.

[19] R. Akerkar, *Introduction to Artificial Intelligence*, illustrated ed. PHI Learning Pvt. Ltd., 2005. [Online]. Available: http://books.google.com/books?isbn=8120328647

[20] A. Fekete, S. Goldrei, and J. P. Asenjo, "Quantifying isolation anomalies," *PVLDB*, vol. 2, no. 1, pp. 467–478, 2009.

[21] P. A. Bernstein, A. Fekete, H. Guo, R. Ramakrishnan, and P. Tamma, "Relaxed-currency serializability for middle-tier caching and replication," in *ACM SIGMOD Conf.*, 2006.

[22] U. Röhm, K. Böhm, H.-J. Schek, and H. Schuldt, "Fas - a freshness-sensitive coordination middleware for a cluster of olap components," in *VLDB*, 2002, pp. 754–765.

[23] U. Röhm and S. Schmidt, "Freshness-aware caching in a cluster of j2ee application servers," in *WISE*, 2007, pp. 74–86.