ConsAD : A *Real-Time* Consistency Anomalies Detector

Kamal Zellag
School of Computer Science
McGill University
Montreal, Canada
zkamal@cs.mcgill.ca

Bettina Kemme
School of Computer Science
McGill University
Montreal, Canada
kemme@cs.mcgill.ca

ABSTRACT

In this demonstration, we present ConsAD, a tool that detects consistency anomalies for arbitrary multi-tier applications that use lower levels of isolation than serializability. As the application is running, ConsAD detects and quantifies anomalies indicating exactly the transactions and data items involved. Furthermore, it classifies the detected anomalies into patterns showing the business methods involved as well as their occurrence frequency. ConsAD can guide designers to either choose an isolation level for which their application shows few anomalies or change their transaction design to avoid the anomalies. Its graphical interface shows detailed information about detected anomalies as they occur and analyzes their patterns as well as their distribution.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Concurrency*

Keywords

Multi-tier Architectures, Consistency, Serializability

1. INTRODUCTION

Modern information systems are based on a multi-tier architecture, where the application server tier takes care of the business logic, and the database tier manages persistent data. The application server coordinates transactions across both tiers, and typically both application server and database system implement some form of concurrency control providing various levels of isolations. Stricter isolation levels provide stronger consistency guarantees while lower levels expose the applications to various types of consistency anomalies such as lost updates, unrepeatable reads, etc. As a trade-off, lower isolation levels typically have better response times and throughput.

Deciding on the right level of isolation is not trivial. As data is spread and replicated across tiers and even within a tier (application servers are often replicated), it is extremely difficult to predict the anomalies that might occur during runtime. While there exist standard measures to evaluate the performance of an application on a particular multi-tier platform, there do not exist any tools that provide a *quantitative measure* of the consistency anomalies that occur during the runtime of a given multi-tier application. Several studies provide a qualitative analysis of the potential anomalies associated with the existing isolation levels [1, 2, 3]. However, it is less clear whether a given application might actually experience such anomalies during its runtime [5] and if yes, to what degree. For example, *lost-updates* are allowed under the isolation level `Read Committed`, but the actual occurrences can vary widely depending on the application.

We address this issue and propose ConsAD, a *consistency anomalies detector* that detects and quantifies consistency anomalies for arbitrary multi-tier applications during runtime. We characterize an anomaly by the presence of a cycle in the serialization graph formed by committed transactions as nodes and dependencies between these transactions as edges. Thus, in the remainder of this demonstration, we use the words *anomaly* and *cycle* interchangeably. ConsAD requires limited adjustments in the application server tier but is completely independent from the database system.

As shown in Figure 1, ConsAD consists of three agents: the collector agent colAgent, the detector agent detAgent and the visual agent visAgent. The colAgent is integrated into the application server. It collects information about transactions and their accessed data items and forwards them to detAgent. detAgent implements a set of algorithms that extract all dependencies between transactions and detects cycles (anomalies). detAgent supports any isolation level higher than or equal to `Read Committed` which is the case for the most industrial isolation levels. Moreover, detAgent classifies the detected anomalies into *patterns* of anomalies that show exactly which business methods are involved in each type of anomaly. Any detected anomaly or pattern is visualized in real time in the visual agent visAgent. Moreover, visAgent shows graphically all detected patterns sorted by their occurrence frequency.

The use of ConsAD is manyfold. Developers can analyze ahead of deployment time whether their applications lead to anomalies if run under a certain isolation level. If anomalies occur frequently the pattern recognition helps designers to find the culprit methods and why their interleaving leads to inconsistencies. This can serve as guidance to rewrite some methods of the application and avoid such anomalies. De-

SIGMOD'12, May 20–24, 2012, Scottsdale, Arizona, USA.

signers might also consider that the number of anomalies that occur during runtime is acceptable. In this case, ConsAD can be used in the production system as a near real-time monitoring tool, allowing the system administrator to intervene immediately after anomalies occur and manually correct the affected data items should an anomaly occur.

In the following we only shortly describe COLAGENT and DETAGENT as they were presented in [6]. In contrast, VISAGENT will be presented for the first time in this demonstration. Adding VISAGENT to COLAGENT and DETAGENT, makes ConsAD a complete consistency anomalies detector tool with an easy-to-use dynamic graphical interface.

2. THE CONSAD SYSTEM

In order to detect anomalies (cycles), ConsAD has to detect all dependencies between transactions, build the serialization graph then detects all cycles in it. There is a wr-edge from T_i to T_j in the graph if T_i performs a write operation on a data item x, and T_j reads the version that T_i has written. There is a ww-edge from T_i to T_j if both T_i and T_j write a data item x and no other committed transaction writes x in between. Finally, there is a rw-edge from T_i to T_j if T_i reads a data item x and T_j is the next to write it. The task of COLAGENT is to collect enough information during runtime that DETAGENT can detect these dependencies and create the graph. COLAGENT and DETAGENT take advantage of the fact that the application server runs under a certain isolation level as it helps them to determine dependencies and speed up cycle detection. The minimum isolation level that is required by ConsAD is `Read Committed`, denoted in the following as RC. It requires a transaction to only read committed data. Furthermore, ConsAD can speed up execution if the isolation level avoids lost-updates. We denote such isolation levels as NOLOSTUPD.

2.1 The Collector Agent

This agent collects information about committed transactions and their accessed data items. In order to do so, it adds an extra field $txnInfo$ to each database table. At commit time of transaction T, COLAGENT stores the unique identifier of T into $x.txnInfo$ for each data item x updated by T. COLAGENT sends information to DETAGENT about each committed transaction T immediately after T commits using TCP/IP channels. If an application is running under NOLOSTUPD, information stored in the field $txnInfo$ is enough for DETAGENT to detect all $wr/ww/rw$ dependencies between transactions. However, if the isolation level RC is used, the field $txnInfo$ is not enough to detect ww-edges. In this case, COLAGENT enforces a sequential commit order for update transactions. In the case of a single application server instance, a simple memory-based counter serializes commit operations. If several application server instances exist, each transaction accesses a database counter in an exclusive way using the SELECT FOR UPDATE statement. With this, only the commit operations are sequentially, transaction execution itself can occur concurrently. As shown in [6], the overhead in response time created by COLAGENT, including the sequential commit order for RC, was less than 3% for all our experiments.

2.2 The Detector Agent

When DETAGENT receives information from COLAGENT about a newly committed transaction T, it adds it to its current graph and determines $wr/ww/rw$-edges involving T. Then it starts a cycle detection algorithm $extDLS$ that is based on Depth Limited Search. Starting from T it explores recursively all outgoing edges until it returns back to T or reaches a parametrized depth limit. Based on our experiments, the size of most cycles ranges from 2 to 4. We have proven in [6], that all $wr/ww/rw$ edges are detected and each cycle C smaller than the depth limit is detected when the last transaction in C is processed. Cycle detection is fast because at the time T is included in the graph, it typically has few outgoing edges. Thus, outgoing edges are typically only explored when there is indeed a cycle.

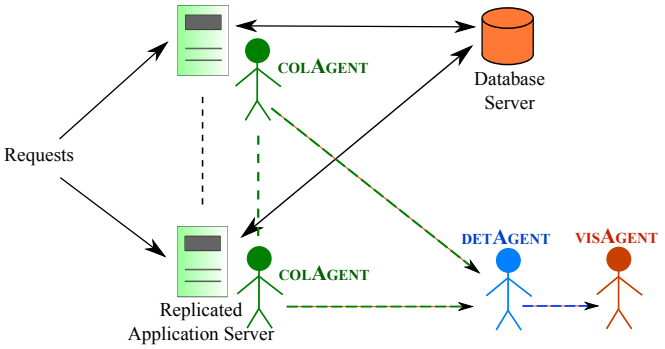

Figure 1: Agents of ConsAD

In typical applications each transaction is called within the boundaries of a business method. Thus, whenever DETAGENT detects a cycle, it maps each transaction in the cycle to the business method that created the transaction. The resulting cycle of business methods is called an *ordered pattern*. Many transaction cycles can belong to the same ordered pattern. Additionally, DETAGENT extracts the set of different business methods that are involved in the cycle, not considering how often or in which order they occur in the cycle. This set is called an *unordered pattern*. Many ordered patterns can belong to the same unordered pattern. For each ordered and unordered pattern DETAGENT keeps track of how many transaction cycles belong to this pattern. Looking at the collected pattern statistics, allows designers to locate sequences or sets of business methods that frequently create anomalies. Thus, they might be able to rewrite these business methods, possibly reordering the execution of some operations, in order to avoid many cycles. Concrete examples of both patterns will be presented in the next section.

2.3 The Visual Agent

After DETAGENT detects any cycle or pattern, it passes its information immediately to the visual agent VISAGENT. Figure 2 shows a snapshot of the graphical interface provided by VISAGENT. In this section, we will see how VISAGENT presents details about detected cycles and patterns.

2.3.1 Cycles

In the example of Figure 2, there are fifty six detected cycles. Each cycle, presented by its identifier, is inserted in a list sorted by cycle size. For example, cycle $C45$ with length 3 is added to this list as $C45/3$. Besides the list of cycles, there is a pie chart that shows the distribution of cycles. This chart is divided into three sectors, showing the percentage of cycles of size 2, 3 and 4+, respectively. By moving the mouse on one section, a tool tip shows the

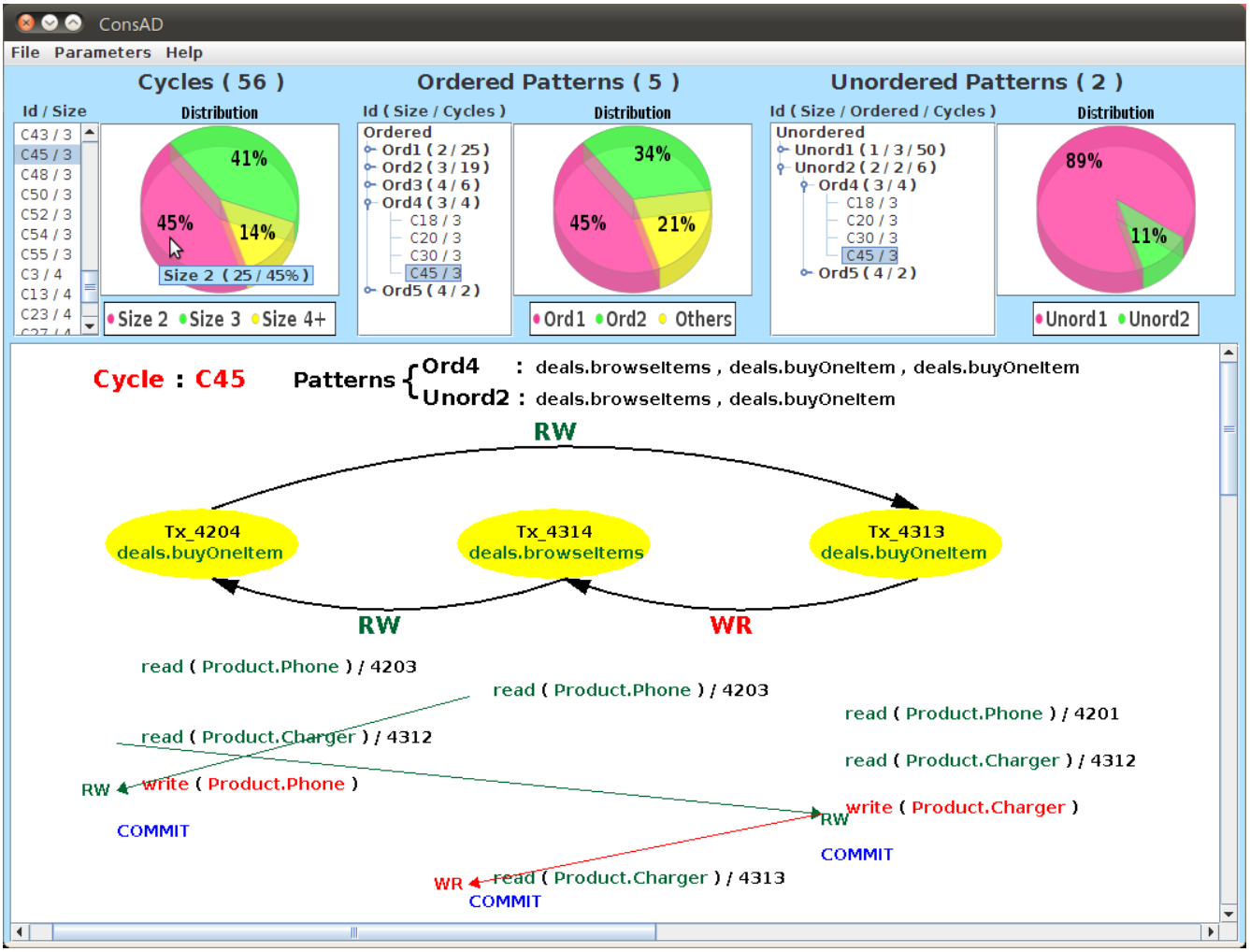

Figure 2: The Visual Agent Interface

number of cycles for this section (for example, 25 cycles with size 2 in the Figure).

Details on any selected cycle are shown in the bottom part of the graphical interface. Figure 2 shows details for cycle $C45$. First, it shows that $C45$ follows the ordered pattern $Ord4$ and the unordered pattern $Unord2$. The involved business methods are also depicted. Below, the serialization graph is shown. $C45$ involves three transactions Tx_4204, Tx_4314 and Tx_4313. Tx_4204 and Tx_4313 are update-transactions triggered by business method *deals.buyOneItem* while Tx_4314 is read-only, triggered by the business method *deals.browseItems*. The lower part of the interface shows a more detailed graph as it exactly presents the operations of the transactions. All methods access data items *Product.Phone* and *Product.Charger*. The value of field $txnInfo$, added by COLAGENT to each table (see Section 2.1), is shown at the end of each read operation. For example, $Charger.txnInfo$ is updated by Tx_4313 with the value 4313, and is extracted later by the read operation in Tx_4314. This detailed representation also depicts which *read/write*-operation pairs cause the *wr/ww/rw*-edges in the graph. For example, there is a *wr*-edge from Tx_4313 to Tx_4314 because Tx_4314 has read the *Charger* item with $txnInfo$ equal to 4313, a version that was previously created by Tx_4313.

Representing the detailed transactions and their execution order in a top-down time-line is not trivial as we do not know the exact execution times of the operations. To be able to provide a precedence order we exploit some properties between these operations. First, within the same transaction a read of a data item x (load from database) must precede the write of x. This generates the precedence $read(x) < write(x)$. Second, the *commit* operation of a transaction T is always after all other operations of T. Third, since we are dealing with isolation levels that provide at least `Read Committed`, we know that whenever there is a *wr*-edge from a transaction T_1 to a transaction T_2 involving a data item x, then we have $write_1(x) < commit_1 < read_2(x)$. Similar precedence orders are extracted for *ww*- and *rw*-edges. These precedence orders provide us with a partial order of operations. From there, we use the *topological-sort* algorithm [4], which generates a possible global logical order between all operations based on partial orders between some of them.

2.3.2 Ordered Patterns

The middle chart at the top of Figure 2 shows five detected ordered patterns: $Ord1$ to $Ord5$. Each pattern is presented by its identifier, the size of the cycle, and the number of cycles that belong to this pattern. Under each

ordered pattern, the cycles that follow this pattern are listed. For example, the four cycles $C18$, $C20$, $C30$ and $C45$ belong to the pattern $Ord4$, which involves the business methods *deals.browseItems*, *deals.buyOneItem* then *deals.buyOneItem*. Business methods can appear many times in such patterns, as ordered patterns depict the sequence in which methods are involved in the cycle. To avoid cycles following the pattern $Ord4$, the application designer can analyze the source code of the methods *deals.browseItems* and *deals.buyOneItem* and check why such combination of calls creates this type of cycles. If this is not enough, a closer analysis of the individual cycles can locate which items are involved in this pattern.

Ordered patterns are sorted in descending order based on the number of cycles following each pattern. In Figure 2, $Ord1$ has the highest number of cycles and it is recommended for designers to start analyzing methods involved in $Ord1$. The pie chart for ordered patterns is composed of three sections. While the first two sections show ordered patterns with the highest number of cycles, the third section shows the percentage of the rest of ordered patterns.

2.3.3 Unordered Patterns

The right chart at the top of Figure 2 provides information about unordered patterns which only indicate the set of business methods involved in cycles but not their order or occurrence frequency. $Unord1$ reflects cycles that have a single business method involved, three ordered patterns and a total of 50 cycles belong to this unordered pattern (depicted as 1/3/50). $Unord2$ reflects cycles that have two different business methods involved, two ordered patterns and six cycles belong to $Unord2$. For each unordered pattern we can retrieve information about the ordered patterns and cycles that follow this pattern. For instance, the two ordered patterns $Ord4$ and $Ord5$ belong to the unordered pattern $Unord2$, and under $Ord4$ there are four cycles ($C18$, $C20$, $C30$, $C45$).

Note that there are fifty cycles under $Unord1$ representing 89% of all cycles, while $Unord2$ has only six cycles representing 11%. In this case, we strongly recommend to the application designer to start by analyzing the single method involved in $Unord1$. If needed, further analysis could be done for ordered patterns under $Unord1$ ($Ord1$, $Ord2$ and $Ord3$), and deep analysis could be conducted for each cycle under these ordered patterns.

2.4 System Setup

In order to use ConsAD, one instance of colAgent must be allocated at each application server instance. In contrast, only one instance of detAgent and visAgent are required. They can run together on any machine supporting the Java virtual machine. Our current implementation of colAgent supports any application server following the Java Enterprise Edition (Java-EE) standard which is widely supported by many open-source and commercial application servers such as JBoss[1] and GlassFish[2]. Extending colAgent to support additional platforms does not require any changes to detAgent or visAgent. A set of configuration parameters can be set in ConsAD using the menu *Parameters* provided by visAgent or by using a configuration file.

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

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

3. DEMONSTRATION

In our demonstration, we propose two scenarios. While in *scenario*1 we run a complete multi-tier benchmark application, in *scenario*2 we use emulated transactions. We will use two laptops running under the Linux operating system Ubuntu, and supporting the Java virtual machine.

3.1 Scenario 1

Under this scenario, the first laptop will be hosting the database server PostgreSQL and the application server JBoss with one instance of colAgent, while the second laptop will be dedicated for both detAgent and visAgent. Most of our discussion with the audience will focus on the graphical interface showed by visAgent. The tests will be conducted using an extension of the RUBiS[3] benchmark that generates a reasonable number of cycles and patterns. After starting the database and application servers, we will start the RUBiS client with a large number of requests. Any detected cycle or pattern will be shown and explored on the graphical interface of visAgent. We are planning to run tests under two isolation levels provided under JBoss: `Read Committed` and a variant of *optimistic concurrency control* that does not provide serializability but disallows *lost updates*.

3.2 Scenario 2

As the real application of *scenario*1 only shows some options provided by ConsAD, we propose a second scenario under which colAgent collects information on emulated transactions. In contrast to *scenario*1, we can quickly restart all agents, change configuration and run many tests using several transactions types. We can also easily change the isolation level and execute a configured number of concurrent and conflicting transactions in a short amount of time. For this scenario, we use one laptop where several instances of colAgent collect information about emulated transactions, while a second laptop will be used for both detAgent and visAgent as in *scenario*1. The number and types of emulated transactions will be passed as parameters to the transaction emulator. The transaction mix contains read-only transactions and transactions that read and write several data items.

4. REFERENCES

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

[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 *Proceedings of IEEE International Conference on Data Engineering*, pages 57–66. IEEE, 2000.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[5] A. Fekete, D. Liarokapis, E. J. O'Neil, P. E. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[6] K. Zellag and B. Kemme. Real-time quantification and classification of consistency anomalies in multi-tier architectures. In *ICDE*, pages 613–624, 2011.

[3]http://rubis.ow2.org/