# How *Consistent* is your Cloud Application?

Kamal Zellag
McGill University
Montreal, Canada
zkamal@cs.mcgill.ca

Bettina Kemme
McGill University
Montreal, Canada
kemme@cs.mcgill.ca

## ABSTRACT

Current cloud datastores usually trade consistency for performance and availability. However, it is often not clear how an application is affected when it runs under a low level of consistency. In fact, current application designers have basically no tools that would help them to get a feeling of which and how many inconsistencies actually occur for their particular application. In this paper, we propose a generalized approach for detecting consistency anomalies for *arbitrary* cloud applications accessing various types of cloud datastores in transactional or non-transactional contexts. We do not require any knowledge on the business logic of the studied application nor on its selected consistency guarantees. We experimentally verify the effectiveness of our approach by using the Google App Engine and Cassandra datastores.

## Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications, Distributed databases*; H.2.4 [**Database Management**]: Systems—*Transaction processing*

## General Terms

Algorithms, Measurement, Theory

## Keywords

Cloud datastores, Consistency, Serializability

## 1. INTRODUCTION

Cloud services have become increasingly attractive for their elastic scalability, high availability and pay-per-use cost models. In particular, over the last years, a whole range of cloud datastores have been developed. They differ not only in the data model they offer (from NoSQL key-value stores to full-fledged relational models), but also in their transactional support. For instance, Amazon RDS[3] and Microsoft SQL Azure[26] are ACID-compliant. In contrast, Amazon SimpleDB[4] and Cassandra[10] do not provide transactions and limit consistency to the level of individual data items. In between, Microsoft Azure Table[26] and Google App Engine (GAE)[18] provide an optional use of transactions but with some limitations on the accessed entities. Apart of commercial systems, the database research community has also proposed many database solutions for the cloud [11, 23, 29, 34, 12, 28, 24] that support transactions with various consistency guarantees.

One can expect such variety in transactional support and consistency guarantees to remain. As stated by the CAP conjecture [9, 17], only two of the three properties consistency, availability and tolerance to network partitions can be achieved at the same time. Thus, developers have to choose a datastore according to their application requirements. Although network partitions are infrequent, they are possible, especially in the presence of a large number of nodes, which leaves the main trade-off between consistency and availability. Lowering the consistency level also has often the advantage of better performance and response times than strong consistency. However, choosing a lower level of consistency poses uncertainties for many applications. Thus, businesses might hesitate to migrate their application from a high consistency datastore to a system with weaker properties. As an example, when WebFilings[31], a financial reporting company that develops cloud-based solutions with high availability requirements, migrated their applications from GAE master-slave to GAE high-replication, they spent almost one month in writing tests and diagnosis until concluding that the weaker form of consistency did not have an impact on their application. Other applications might even be willing to accept certain inconsistencies, but would like to know how often and in which situations they occur.

Consistency anomalies in the context of transactional systems have been well studied qualitatively in the literature [1, 7, 8, 13, 14], but only few works have studied their occurrence during runtime of applications. Fekete et al. [15] provide quantitative information about the violations of integrity constraints when a centralized database system runs under various isolation levels. Wada et al. investigated in [30] consistency properties for some cloud datastores, but they limited their study to a subset of client-centric consistency definitions such as *monotonic-reads* and *read-your-writes*. In [32, 33], we detect consistency anomalies for multi-tier applications where multiple application servers access a centralized ACID database. Our approach assumed that (1) data items are accessed only in transactional contexts, (2) all transactions execute under the same isolation level and (3) the database is not replicated. In this paper, we propose a generalized approach for detecting consistency anomalies for arbitrary cloud applications that does not have any of these restrictions.


*SOCC'12,* October 14-17, 2012, San Jose, CA USA

We base our detection approach on the serializability theory and we characterize an anomaly by the presence of a cycle in the global dependency graph, that is, we detect unserializable executions. We provide applications not only with concrete numbers of how many anomalies occur in a certain time frame but also what kind of inconsistencies happened including the particular data items that were affected. Such analysis allows developers during testing to determine the true amount of inconsistencies that might occur for their particular application, providing an additional metric that provides an estimate of the trade-off between performance and consistency. Our detection mechanism can also be used during run-time reporting to system administrators whenever inconsistencies occur so that counter-measures can be taken.

The idea of our anomaly detection mechanism is to build the dependency graph during the execution of a cloud application and detect cycles in the graph. The objective is to do this at the application layer and independently of the underlying datastore. There are several reasons for that. First, we can use the same solution for a whole set of datastores making it easy to compare the levels of consistency that are provided by different datastores. Furthermore, performing the analysis at the datastore is often not even possible, as the internals of the datastores are generally hidden and not accessible. Thus, it is very hard to derive the exact consistency mechanism used in the datastore. The use of various replication strategies makes the task even more complex. Furthermore, many cloud databases do not even provide the concept of a transaction but work per operation, although the application layer actually would like to perform consistency analysis considering more than one operation as a logical unit of execution, such as a method call or a web request execution.

To support the latter, we introduce the concept of a unit-of-work, that can be defined by the application, and consider all operations executed within this unit-of-work as conceptually the same as a transaction representing a node in our dependency graph. From there, our approach relies on mechanisms that capture all necessary information to determine dependencies at the application level. More specific, we observe any operation submitted to the datastore, and determine the dependencies that occur between the units-of-work within the datastore. In some cases, we can be sure that we capture the real dependencies that occur in the datastore. Sometimes we do not have enough knowledge leading us to assume dependencies that might or might not occur in the datastore, so we call them potential dependencies. Thus, what we create at the application layer is an approximated dependency graph with real and potential dependency edges. As a result, if this approximated graph contains cycles of only real edges we can be sure to have found a true anomaly, while some cycles might only indicate potential anomalies. We propose first an off-line algorithm that observes an execution, and then builds the dependency graph and performs cycle detection. From there, we derive an on-line algorithm, that builds the dependency graph incrementally during the execution, and initiates cycle detection whenever a unit-of-work completes execution. Detected cycles can be reported to a visualization tool such as [33], that visualizes transactions involved in cycles and the entities they have accessed, groups cycles according to the request type combinations that caused them, and provides summary information about detected anomalies.

To see how our model can be applied to some cloud platforms, we conducted a set of experiments under two datastores: GAE high-replication and Cassandra.

In summary, our paper makes the following contributions:

1. We propose a novel model for approximating a dependency graph in order to detect consistency anomalies for arbitrary cloud applications running on cloud datastores with weak consistency properties.
2. We detail how we create the approximated graph and detect cycles that can be used to quantify and qualify consistency anomalies during run-time.
3. We show how our approach can be applied to some cloud platforms, more specifically GAE high-replication and Cassandra.

# 2. EXECUTION MODEL

As a first step, we have to understand the basics of how cloud datastores work, and what assumptions we can make about their behavior. We also have to understand what kind of inconsistencies we want to detect.

## 2.1 System Architecture

Cloud systems are, in principle, quite similar to two-tier architectures as depicted in Figure 1. The application is embedded in a web-service or other service interface, that can be called by some client. It is executed within the application tier. The application program itself performs the business logic and performs read and write operations on entities (data items) maintained by the cloud datastore. In our model, we assume that all write operations to entities explicitly specify the unique identifier of the entity. This is true for basically all key/value stores, but also for others as entities are typically first read and loaded into the application context, changed locally, and then written back to the datastore. Read operations could be predicate based. Nevertheless, we assume that all data items that are loaded are transformed into entities within the application context, and thus, are individually observable.

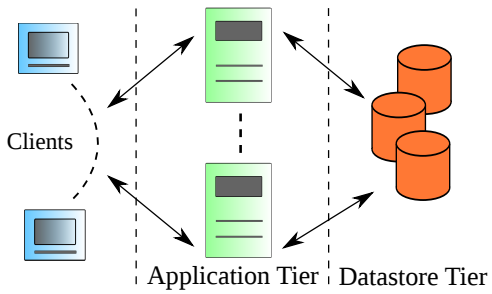



**Figure 1: Two-tiers Cloud Architecture**

## 2.2 Data Replication

Due to availability and performance needs, data is often replicated onto several nodes. Replica control approaches can be categorized according to where and when updates take place [19]. In a primary copy or single-master approach, all updates to an entity have to be submitted to the primary (master) copy of the entity, which propagates the changes to the secondary copies. In contrast, under update-anywhere or multi-master approach, all copies accept updates and coordinate among each other. From there, updates can be propagated eagerly (also called synchronous), through two-phase-commit, consensus or quorum protocols. However, this is costly and reduces the availability. Therefore, lazy replication

(also called asynchronous), first confirms to the user the successful execution of the operation/transaction and only then propagates changes.

Eager protocols guarantee that there is always only one "latest" visible version of an entity and all copies install versions in the same order. With lazy, primary copy, the execution order at the primary copy determines the order of updates, and thus, the order of versions. Secondary copies can easily follow the same order but they do miss the latest versions until the primary propagates them. In contrast, lazy, update-anywhere replication does not guarantee that all copies install versions in the same order. Different copies of the same entity might be updated concurrently by independent operations. Such divergence must be detected and reconciled to guarantee *eventual consistency*, assuring that all replicas converge to the same state within the eventual consistency window (assuming no further updates take place).

## 2.3 Serializability

Consistency can be seen from two points of view. The first is data centric, and concentrates on the state of the data before and after read or write operations. The second is client centric, and concentrates on how each client sees its own read and written entities. In this paper, we focus on data centric consistency.

Users typically group their read and write operations on the logical entities of the datastore into transactions. When transactions run concurrently, they must be isolated. The established correctness criterion is *serializability*, where the interleaved execution of transactions is equivalent to serial execution of these transactions. This is traditionally defined through dependencies of operations and reflected through a global dependency graph (GDG).

A transaction $T_i$ consists of a sequence of read and write operations on entities and terminates either with a commit $c_i$ which verifies that all write operations have succeeded, or an abort operation $a_i$ which indicates that none of the write operations succeeded. In this paper, we use a multi-version concept as it is more general. In the real system, no multiple versions might exist but we can consider the consecutive values of an entity as versions. In this model, each write operation on an entity $e$ creates a new version of $e$ and all versions of $e$ are totally ordered. We will discuss later how this version order is created. A *history* describes a partial order of the operations of a set of transactions which reflects the order in which read, write, and commit operations are executed. In particular, an order must be established between conflicting operations. Two operations conflict if they access the same entity, are from two different transactions, and at least one of them is a write operation.

From a history of a set of transaction we can derive the dependency graph $GDG$ (GDG for global dependency graph, given that execution on cloud environments can run over a distributed and possibly replicated system). In this graph, transactions are nodes, and there are edges between nodes if the corresponding transactions have conflicting operations. In particular, the graph contains three types of edges between transactions. There is a $wr$-dependency edge $T_i - wr \rightarrow T_j$ from a transaction $T_i$ to a transaction $T_j$, if $T_j$ reads the version of an entity $e$ that was written by $T_i$. There is a $ww$-dependency edge $T_i - ww \rightarrow T_j$ from $T_i$ to $T_j$ if there is an entity $e$, and $T_i$ and $T_j$ create consecutive versions of $e$. There is a $rw$-dependency edge $T_i - rw \rightarrow T_j$ from $T_i$ to $T_j$ if $T_i$ reads a version $e_k$ of an entity $e$, created by $T_k$, and $T_j$ creates the consecutive version of $e_k$. A history is *serializable*, that is, it is equivalent to a serial history where transactions execute one after the other, if the GDG is *acyclic*.

The basic idea in this paper is to observe the executions that occur in the datastore, create the corresponding GDG graph and test for cycles. However, this is far of being a trivial approach as it is difficult to see from the outside what is happening within the system. The main challenge is to determine $ww$-edges, that is, which is equivalent to determine the total version order for each entity. Determining $wr$-edges is quite simple. By tagging each version of $e$ with the transaction identifier of the transaction $T_i$ that created it (or if there are no multiple versions, tagging the single entity version with the transaction that was the last to update it), a reading operation of a transaction $T_j$ can immediately determine a $wr$-dependency edge $T_i - wr \rightarrow T_j$. Furthermore, if there is a $wr$-dependency edge $T_i - wr \rightarrow T_j$ due to an entity $e$, and a $ww$-dependency edge $T_i - ww \rightarrow T_k$, also due to $e$, then the $rw$-dependency edge $T_j - rw \rightarrow T_k$ can be immediately derived. However, no simple mechanism exists for determining $ww$-edges, because this depends strongly on the underlying system, and is not always observable from the outside. A considerable part of this paper is dedicated to handle such uncertainty.

## 2.4 Unit of Work in Cloud Datastores

In its most general form, a cloud datastore consists of a set of entities that can be read and written (we consider inserts and deletes as special writes). The first problem to solve is that many datastores do not even offer to the application programmer the primitives to bundle several read and write operations into a transaction. In this paper, we take a quite simple approach and introduce the concept of a *Unit of Work* which is conceptually similar to a transaction. If transaction primitives exist, then a transaction within the application program reflects a unit of work. Otherwise, we let the designer decide what they want to consider as unit of work. In our implementation, a method invocation (web request) consisting potentially of many read and write operations, can be the unit of work. The unit of work is for which we will detect anomalies. Note that the underlying datastore might use any concurrency control mechanism or none at all. If it offers transactions and implements strict 2-phase-locking, we will not detect any anomalies. For any system that uses a lower level of isolation or no transactions at all, anomalies are possible and our goal is to detect them when they occur.

Units of work are not exactly the same as transactions, as without transactional support from the datastore, atomicity is not provided. That is, if the datastore offers transactions, it usually offers rollback in case of aborts. Without transactions, a web request might only be executed partially, that is, some but not all of its updates might be executed in the datastore. In this case, we can assume that each update on an individual entity is successful (even if there are many copies and update propagation is done lazily). Generally, transactional atomicity is not a concern in this paper. Whatever part of a unit of work is executed, will be the scope of the unit.

## 2.5 Order of write operations

### *2.5.1 How do systems order writes?*

As mentioned before, some systems implement transactions, others only work on a per-operation basis. Transactional systems can use a variety of concurrency control mechanisms such as locking, being optimistic, or multi-version based. The system might or might not use replication. All these internal mechanisms influence how write operations, respectively entity versions, are ordered by the system.

In the following, we list a set of assumptions that we believe hold in most datastores and which help us to determine how the datastores order conflicting write operations. We first assume the system to not use replication, i.e., there exists only one copy of each entity (albeit it might have multiple versions).

**ASSUMPTION 1**

We assume that if a unit of work $U_j$ reads a version $e_i$ created by $U_i$ and then creates a newer version $e_j$, then internally $e_j$ is created and ordered after $e_i$.

This assumption derives naturally from the causality relationship.

• ***Transactional Datastores.***

If the system supports transactions, then we assume the write operations, and with it the versions these writes create, are ordered according to the commit order of their respective transactions.

**ASSUMPTION 2**

In a transactional datastore, if two units of work $U_i$ and $U_j$ both update the same entity $e$, and $U_i$ commits before $U_j$, then the version of $e$ created by $U_i$ is ordered before the version created by $U_j$.

In fact, this is true for basically all datastores that use one of the common concurrency control mechanisms. For instance, if the system uses strict 2-phase-locking for write operations, then this is obvious. If $U_i$ holds an exclusive lock on $e$ until it commits, then nobody can create a new version of $e$ until after $U_i$'s commit. But also concurrency control mechanisms such as *snapshot-isolation* guarantee that all dependencies are in the same order as the commit order. Therefore, in datastores supporting transactions, *we can determine $ww$-dependency edges if we can determine the commit order of transactions*.

• ***Non-transactional Datastores.***

If no transactions are supported, then order is determined by when independent operations are executed.

**ASSUMPTION 3**

In a non-transactional datastore, if two units of work $U_i$ and $U_j$ both update the same entity $e$, and the datastore executes $U_i$'s write before $U_j$'s write, then the version of $e$ created by $U_i$ is ordered before the version created by $U_j$.

Therefore, in datastores not supporting transactions, *we can determine $ww$-dependency edges if we can determine the order of execution of individual operations.*

• ***Replication.***

In case of replication, nothing changes if the system uses an eager approach, or a lazy primary copy approach. In an eager system using transactions, the commit operation returns only when all copies (or at least a majority of copies) are synchronized. In an eager system without transactions, the write operation only returns when a sufficient number of copies are synchronized. Thus, the commit order, respectively the write order, determines the $ww$-dependency edges. In case of a lazy primary copy approach, a transaction/operation returns when it has committed/executed on the primary copy, and the secondary copies apply the changes in the same order. Thus, the updates on the secondary copies lead to the same $ww$-edges as in the primary copy. This means that in these cases, if we are able to observe in which order transactions commit, respectively write operations return, we can determine the order of $ww$-dependency edges.

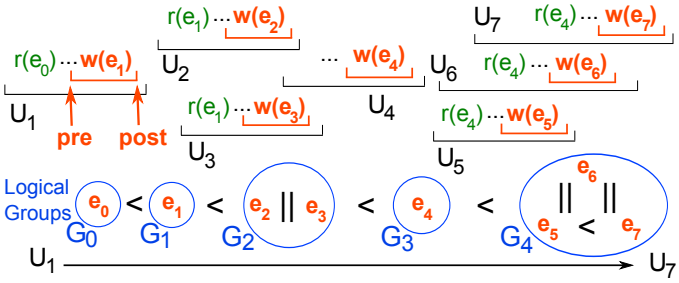



**Figure 2: Created versions of an entity $e$**

The problematic case are some lazy update anywhere approaches. Consider the scenario where transaction $T_1$ updates entity $e$ at replica $A$ and $T_2$ updates entity $e$ at replica $B$. $T_1$ commits before $T_2$. Later the updates are propagated, a conflict detected, and conflict resolution takes $T_1$'s version as final version. Conceptually, one can consider this as $T_2$ before $T_1$, being contrary to the commit order. However, there exist lazy, update anywhere systems that actually order write operations according to their execution order, respectively commit order, and we discuss them in Section 4. For those systems, either Assumption 2 or Assumption 3 are true, and our solution is applicable.

### 2.5.2 *Ordering entity versions*

The question is now how we can determine at the application tier the order of entities versions. As illustration, Figure 2 shows the execution (from left to right) of a set of non-transactional units of work $U_1 ... U_7$. Each unit reads a version of $e$ then creates a new version, except of unit $U_4$ that performs a blind write, i.e., it writes $e$ without first loading and reading it.

The simplest and most reliable way to determine precedence is to exploit Assumption 1 as we can easily observe the entity versions that a unit of work reads. Thus, we take advantage of it as much as possible. If there is little concurrency in accessing individual entities then this is all what we need to exactly determine the $ww$-dependencies. For instance, in Figure 2, $U_5$, $U_6$ and $U_7$ all read entity version $e_4$, thus, $e_5$, $e_6$ and $e_7$ are ordered after $e_4$.

However, if blind writes are possible, and if several units of work read the same entity version, Assumption 1 is not enough to determine all dependencies. For example, in Figure 2, Assumption 1 does not tell us how versions $e_5$, $e_6$ and $e_7$ are ordered with respect to each other.

For these cases, we take advantage of Assumptions 2 and 3 whenever possible. For that, we have to determine in which order transactions are committed, respectively operations executed, within the datastore engine. To illustrate this, Figure 2 shows for each write operation the time from its submission to the datastore to the time the write returns (red lines under the $w(e_i)$ symbols). For example, the write operation $w(e_3)$ returns before the write operation $w(e_4)$ is submitted. If we can observe this temporal precedence, we can be sure that the datastore executed $w(e_3)$ before $w(e_4)$. However, as applications can be multi-threaded, two requests could be submitted concurrently, as is the case for $w(e_2)$ and $w(e_3)$. In this case, the execution order within the datastore engine is not observable and any order is possible.

To capture such behavior, we define for each version $e_i$ of an entity $e$ two timestamps $pre$ and $post$ that encapsulate the *interval* in which $e_i$ has been created. If the unit of work $U_i$ is not transactional, then we capture at the application tier the before and after times of any individual write operation $w(e_i)$ within $U_i$ and they are assigned as $pre$ and $post$ values to $e_i$. If $U_i$ is transactional, then we capture the time when $U_i$'s commit operation is submitted and the time it returns. These times will be assigned as $pre$ and $post$ values to any version created by $U_i$. In summary, we assume that any entity version $e_i$ has been created between two times $e_i.pre$ and $e_i.post$. Thus, if for two entity versions $e_i.post < e_j.pre$, then $e_i$ is ordered before $e_j$.

Such temporal comparison might be imprecise, if there are several instances of the application tier (which is often the case for scalability). Capturing *pre* and *post* on different machines and comparing them might result in a wrong order if physical time is not properly synchronized. We discuss this issue in Section 4.2.1 where we show how $pre$ and $post$ values, captured at different application servers, can be adjusted taking advantage of an external NTP (Network Time Protocol) server. Although our experiments have shown that such solution is very reliable for most cluster configurations, we only resort to it if we cannot determine the order based on Assumption 1.

Thus, by using Assumption 1 whenever possible and resorting to Assumption 2/3 whenever necessary, we can create an entity order as follows:

**DEFINITION 2.1** ***Created Before***

A version $e_i$ is said to be *created-before* a version $e_j$, denoted as $e_i < e_j$, if:

**a**. there exists a unit of work that reads first $e_i$ then creates $e_j$

**or**

**b**. there exists a sequence of units $U_1...U_nU_j$ with:

1. $U_1$ reads $e_i$ then creates a version $e_1$
2. $U_k$ reads $e_{k-1}$ then creates a version $e_k$, for $k = 2..n$
3. $U_j$ reads $e_n$ then creates $e_j$

**or**

**c**. none of **a**. or **b**. is valid but $e_i.post < e_j.pre$.

In the case where none of the relations $e_i < e_j$ and $e_j < e_i$ hold, then $e_i$ and $e_j$ are considered *concurrently-created*.

**DEFINITION 2.2** ***Concurrently Created***

Given two versions $e_i$ and $e_j$ of an entity $e$, if $e_i$ is not *created-before* $e_j$ and $e_j$ is not *created-before* $e_i$ (both $e_i < e_j$ and $e_j < e_i$ are invalid), then we say that $e_i$ and $e_j$ are *concurrently-created*, and we denote them as $e_i \parallel e_j$.

Coming back to Figure 2, we exploit the condition *created-before*-**a** to determine $e_0 < e1$, $e_1 < e_2$, $e_1 < e_3$, $e_4 < e_5$, $e_4 < e_6$ and $e_4 < e_7$. Additionally, $e_2 < e_4$, $e_3 < e_4$ and $e_5 < e_7$ are due to the condition *created-before*-**c**. We have as well $e_2 \parallel e_3$, $e_5 \parallel e_6$ and $e_6 \parallel e_7$. For these concurrently created versions, we do not know the ordering within the datastore. Based on the previously defined relations created-before and concurrently-created, we can conclude that for each two versions $e_i$ and $e_j$, either $e_i < e_j$, $e_j < e_i$ or $e_i \parallel e_j$. However, we cannot tell the exact predecessor or successor of each version. For example, the predecessor of $e_4$ can be either $e_2$ or $e_3$ (depending how the datastore actually installed these versions), and the successor of $e_4$ can be either $e_5$ or $e_6$.

Another important concept to be used later in this paper is when two versions are not concurrent, but related to each others via concurrent versions.

**DEFINITION 2.3** ***Transitively Concurrent***

A version $e_i$ is said *transitively-concurrent* to a version $e_j$, denoted as $e_i \ |..| \ e_j$, if $e_i \parallel e_j$ or there is a sequence of versions $e_1...e_p$, such as:

1. $e_i \parallel e_1$
2. $e_k \parallel e_{k+1}$ , k=1..(p-1)
3. $e_p \parallel e_j$

For example, in Figure 2, $e_5$ is not concurrent to $e_7$ ($e_5 < e_7$), but $e_5$ is concurrent to $e_6$ which is in turn concurrent to $e_7$. Hence, we have $e_5 \ |..| \ e_7$.

The relations *created-before*, *concurrently-created* and *transitively concurrent* allow us to build *logical groups* of entities, where each group only contains transitively concurrent versions. For example, Figure 2 contains five consecutive logical groups where $G_0$, $G_1$ and $G_3$ are limited to one version, while groups $G_2$ and $G_4$ contain more than one version. Note that logical groups are disjoint and any entity version belongs only to one logical group.

# 3. DETECTION OF ANOMALIES

We characterize an anomaly by the presence of a cycle in the global dependency graph (GDG), as described in Section 2.3, where the nodes are the units of work. We detect anomalies by building the GDG and searching for cycles in it. We already mentioned in Section 2.3 that the main challenge is to determine $ww$-dependency edges, which basically means to determine the version order of entities. We have already seen before that for some versions it is not possible to observe from the outside in which order they were created. For example, in Figure 2, $e_5 \parallel e_6 \parallel e_7$, hence there can be several order possibilities between $e_5$ , $e_6$ and $e_7$, which has also an impact on both $e_4$ and $e_8$. In this case, we do not know the real successor of $e_4$ or the real predecessor of $e_8$ in the datastore. Therefore, the presence of concurrent versions makes some edges *undetectable* in the actual GDG.

To address this issue we approximate the exact GDG (the one that truly reflects the dependencies within the datastore) with a new graph, denoted as t-GDG (*transitive* GDG), that contains new types of edges where some of them are approximating the undetectable edges in GDG.

In the following, we assume that each entity version $e_i$ is tagged with the identifier of the unit of work $U_i$ that created it. Once an entity version is read, we assume a method *creatorOf* (Section 4.2.3) exists that provides us with the identifier of the unit of work that created the version. We also assume that for each entity $e$ we have partially ordered all versions and created the logical groups as described earlier.

## 3.1 Building the t-GDG

The t-GDG has the same nodes as the GDG, where each node presents a unit of work. While the GDG contains only $ww$-, $wr$- and $rw$-edges, the t-GDG contains additional types of edges that we define in this section.

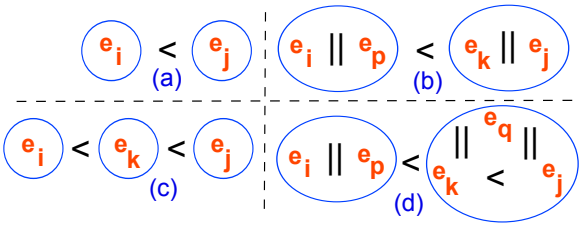



**Figure 3: Concurrent versions of an entity $e$**

### 3.1.1 *wr-edges*

$wr$-dependency edges are simple to build if entity versions are tagged. The reading unit simply has to extract the creator of the version read. Algorithm 1 shows, in lines 1 and 2, how we build the $wr$-edges.

**Algorithm 1** BUILDREADEDGES ( $e_c$ , $U$ )

**Require:** an entity $e_c$ read by a unit of work $U$

**Ensure:** *builds* $wr$, $rw$, $rw$-$t$-$ww$ and $rw$-$at$-$ww$ edges

1: $U_c = creatorOf(e_c)$
2: **add** $U_c - \boldsymbol{wr} \rightarrow U$
3: **for** *out* **in** $U_c.outgoingEdges$ $(U_c - out \rightarrow U_j)$ **do**
4: **if** $(out.type == ww)$ **then**
5: **add** $U - \boldsymbol{rw} \rightarrow U_j$
6: **else if** $(out.type == t\text{-}ww)$ **then**
7: **add** $U - \boldsymbol{rw\text{-}t\text{-}ww} \rightarrow U_j$
8: **else if** $(out.type == at\text{-}ww)$ **then**
9: **add** $a = U - \boldsymbol{rw\text{-}at\text{-}ww} \rightarrow U_j$
10: $b = alternate(out)$
11: **add**$(a)$ to $alternateSet(b)$
12: **add**$(b)$ to $alternateSet(a)$

### 3.1.2 *ww-edges*

We can determine a $ww$-edge from a unit $U_i$ ($e_i$'s creator) to a unit $U_j$ ($e_j$'s creator) if the following conditions are satisfied:

1. $e_i < e_j$
2. There is no concurrently created version to $e_i$ or to $e_j$, i.e., $e_i$ and $e_j$ are the only versions in their respective logical groups.
3. No other version $e_k$ is created between $e_i$ and $e_j$, i.e., $e_i$ and $e_j$ are in consecutive logical groups

If condition (1) is satisfied but one of the conditions (2) or (3) is not, we cannot say that there is a direct $ww$-edge from $U_i$ to $U_j$. The case (a) shown in Figure 3 satisfies all conditions, while cases (b), (c) and (d) do not satisfy the second or third condition.

Although cases (b), (c) and (d) in Figure 3 do not guarantee a direct $ww$-edge from $U_i$ to $U_j$, the exact GDG must have a path of $ww$-edges from $U_i$ to $U_j$:

**LEMMA 3.1**

If units $U_i$ has created version $e_i$ of $e$ and $U_j$ has created version $e_j$ of $e$, and $e_i < e_j$, then the exact GDG has a *path* of direct $ww$-edges from $U_i$ to $U_j$ that are all due to $e$.

***Proof.*** see appendix A.1

If we cannot observe the $ww$-edges in this sequence at the application tier, then we approximate it by using a single edge from $U_i$ to $U_j$ that we name a *transitive* $ww$-edge and define as follows.

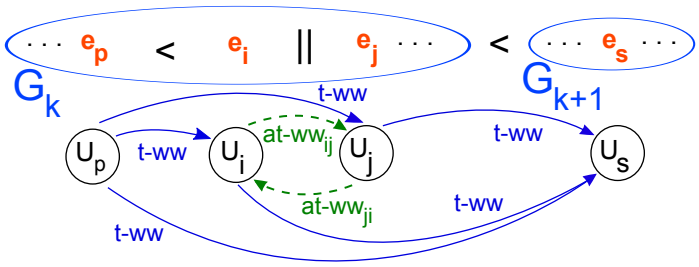



**Figure 4: write edges between consecutive groups**

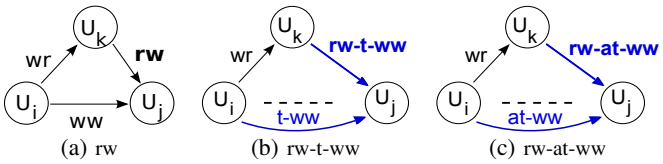



**Figure 5: rw, rw-t-ww and rw-at-ww edges**

**DEFINITION 3.1** ***Transitive ww-edge.***

There is a *transitive* $ww$-edge, denoted as ***t-ww***, from a unit $U_i$ to a unit $U_j$ if the following holds:

1. $U_i$ has created $e_i$, $U_j$ has created $e_j$, and $e_i < e_j$.
2. $e_i$ and $e_j$ belong to the same logical group or they belong to consecutive logical groups where at least one group has more than one version.

Figure 4 shows an example with two consecutive groups $G_k$ and $G_{k+1}$ and the $t$-$ww$ edges within group $G_k$ and from $G_k$ to $G_{k+1}$. Thus, whenever $e_i < e_j$, then we have a path of $ww$- and $t$-$ww$-edges from $U_i$ to $U_j$. However, if $e_i \parallel e_j$, then we do not know how the datastore created and ordered the two versions, it can be either one or the other way. To express this, we introduce the concept of an alternate transitive edge.

**DEFINITION 3.2** ***Alternate Transitive ww-edges.***

If $e_i \parallel e_j$, then we build two *alternate* $t$-$ww$-edges, denoted as ***at-ww***, between $U_i$ and $U_j$, $at\text{-}ww_{\boldsymbol{ij}}$ and $at\text{-}ww_{\boldsymbol{ji}}$, and we define an *alternate* function between them as $at\text{-}ww_{\boldsymbol{ij}} = alternate(at\text{-}ww_{\boldsymbol{ji}})$ and $at\text{-}ww_{\boldsymbol{ji}} = alternate(at\text{-}ww_{\boldsymbol{ij}})$.

Note that concurrent versions are always within a single logical group as shown in Figure 4. Note also that these can be transitive edges, i.e., the underlying GDG could have a direct edge or a path between the two units. For example, looking at group $G_2$ of Figure 2, the underlying system's GDG might have a direct $ww$-edge from $U_2$ to $U_3$ or a direct ww-edge from $U_3$ to $U_2$. Both edges are represented through $at\text{-}ww_{\boldsymbol{23}}$ and $at\text{-}ww_{\boldsymbol{32}}$ in t-GDG. That is, given a pair of alternate edges, one reflects the real flow in the underlying system, the other is in the wrong direction.

As we can see in Figure 4, a pair of $at$-$ww$-edges always builds a cycle in t-GDG. However, only one of the two edges reflects the exact direction in which dependencies flow in the exact GDG. Therefore, when we perform cycle detection we ignore cycles that contain at the same time both $at\text{-}ww_{ij}$ and $at\text{-}ww_{ji}$ (caused by the same entity).

Algorithm 2 shows how we build the edges $ww$, $t$-$ww$ and $at$-$ww$. It accepts a logical group $G_k$ and its successor $G_{k+1}$ as an input, builds $ww$ or $t$-$ww$ edges from $G_k$ to $G_{k+1}$, then builds $t$-$ww$ or $at$-$ww$ edges within $G_k$.

**Algorithm 2** BUILDWRITEEDGES ( $G_k$ , $G_{k+1}$ )

**Require:** a logical group $G_k$ and its successor $G_{k+1}$

**Ensure:** $builds\ ww$, $t\text{-}ww$ and $at\text{-}ww$ edges

```
1: for e_i in G_k do
2:   for e_s in G_{k+1} do
3:     U_i = creatorOf(e_i) , U_s = creatorOf(e_s)
4:     if ( sizeOf(G_k) == sizeOf(G_{k+1}) == 1 ) then
5:       add U_i − ww → U_s
6:     else
7:       add U_i − t-ww → U_s
8: for e_i , e_j in G_k do
9:   U_i = creatorOf(e_i) , U_j = creatorOf(e_j)
10:  if ( e_i < e_j ) then
11:    add U_i − t-ww → U_j
12:  else if ( e_i || e_j ) then
13:    add U_i − at-ww_ij → U_j
14:    add U_j − at-ww_ji → U_i
```

### 3.1.3 rw-edges

A $rw$-edge exists between $U_k$ and $U_j$ (Figure 5(a)) only if there is a $wr$-edge from $U_i$ to $U_k$ due to $e$ and $ww$-edge from $U_i$ to $U_j$ due to the same entity $e$. Thus, once the $wr$ and $ww$ edges are present, the conclusion of the $rw$-edge is immediate. However, and as seen earlier, some $ww$-edges are undetectable and are replaced with $t\text{-}ww$ or $at\text{-}ww$-edges. Therefore, some $rw$-edges are also undetectable. This leads to the definition of two new types of edges that replace undetectable $rw$-edges.

**DEFINITION 3.3** ***Transitive rw-edge.***

There is a *transitive* $rw$-edge, denoted as ***rw-t-ww***, from a unit $U_k$ to a unit $U_j$ (Figure 5(b)), if $U_k$ has read a version $e_i$ created by a unit $U_i$, and there is a $t\text{-}ww$-edge from $U_i$ to $U_j$ (due to $e$).

**DEFINITION 3.4** ***Alternate Transitive rw-edge.***

There is an *alternate transitive* $rw$-edge, denoted as ***rw-at-ww***, from a unit $U_k$ to a unit $U_j$ (Figure 5(c)), if $U_k$ has read a version $e_i$ created by a unit $U_i$, and there is an $at\text{-}ww$-edge from $U_i$ to $U_j$ due to $e$.

Algorithm 1 shows how we build the edges $rw$ in line 5, $rw\text{-}t\text{-}ww$ in line 7 and $rw\text{-}at\text{-}ww$ in line 9 where the unit $U$ reads version $e_c$. Note that the edge $a$ is built, in line 9, only if the outgoing edge $out$ is present, therefore $a$ should not be present in any cycle at the same time with the alternate edge of $out$ which is $b$.

## 3.2 Cycles in the t-GDG

The t-GDG is an approximation of the exact GDG, replacing undetectable edges (that are due to the presence of concurrently created versions) with transitive, and alternate transitive edges. The question is whether t-GDG has the same cycles as GDG. In the following, we claim that (1) any edge present in GDG from a unit $U_i$ to a unit $U_j$ due to an entity $e$, is either also present in t-GDG or is replaced with a new type of edge from $U_i$ to $U_j$, and (2) for any cycle $C$ involving units $U_1...U_n$ in GDG there is a cycle t-$C$ in t-GDG involving the same units in the same order. The latter property ensures that all cycles present in the exact GDG are detected in the t-GDG.

**LEMMA 3.2**

(i) For any edge $U_i - wr \rightarrow U_j$ in GDG, there is a corresponding edge in t-GDG.

(ii) For any edge $U_i - ww \rightarrow U_j$ in GDG there is either a $U_i - ww \rightarrow U_j$, a $U_i - t\text{-}ww \rightarrow U_j$, or a $U_i - at\text{-}ww \rightarrow U_j$ edge in t-GDG.

(iii) For any edge $U_i - rw \rightarrow U_j$ in GDG there exists either a $U_i - rw \rightarrow U_j$, a $U_i - rw\text{-}t\text{-}ww \rightarrow U_j$, or a $U_i - rw\text{-}at\text{-}ww \rightarrow U_j$ edge in t-GDG.

***Proof.*** see appendix A.2

**THEOREM 3.1**

For each cycle $C$ involving units $U_1...U_n$ in GDG, there is a cycle $t\text{-}C$ in t-GDG involving the same units in the same order.

***Proof.*** see appendix A.3

If all logical groups are of size 1, then all versions are in total precedence order, and all $ww$ and $rw$ edges are detectable without the need for any transitive or alternate edges. Otherwise, t-GDG contains more edges than GDG which may generate some cycles in t-GDG that are not present in GDG. In this case, some false-positive cycles are detected in t-GDG. Based on our experiments, they are very limited comparatively to the real cycles.

**THEOREM 3.2**

For any cycle $t\text{-}C$ in t-GDG involving units $U_1...U_n$, there are two cases:

1. $t\text{-}C$ does not contain any alternate edge. Then there is a cycle $C$ in GDG that contains the same units $U_1...U_n$ appearing in the same order as in $t\text{-}C$. $C$ may contain additional nodes. In this case we say that $t\text{-}C$ represents a *real-cycle*.
2. $t\text{-}C$ contains at least one alternate edge. In this case, there might not be a cycle $C$ in GDG that corresponds to $t\text{-}C$ as the exact ordering of some entity versions in the datastore might have been in reverse order. Therefore, we call $t\text{-}C$ a *potential-cycle*.

***Proof.*** see appendix A.4

## 3.3 Cycle Detection

Our detection of anomalies is based on two types of agents: a collector agent (COLAGENT) and a detector agent (DETAGENT). One COLAGENT is integrated within each replica of the application tier and collects all the information that is needed for building t-GDG. That is, each COLAGENT observes the execution of units of work, and their read and write sets. It tags entities versions with the identifiers of the units of work that create them. COLAGENT's implementation depends partially on the cloud platform and we will discuss some implementation details in Section 4.

COLAGENT's observations are forwarded to DETAGENT which performs the detection of anomalies. DETAGENT is completely independent from any cloud platform and can run on the cloud or on any standalone machine. For each detected anomaly, DETAGENT generates detailed information about the units of work involved and the affected entities. DETAGENT can process the collected information either *off-line* or *on-line*.

### 3.3.1 Off-line Detection

In the off-line approach, we assume that the execution of a set of units of work has completed and DETAGENT has all information about these units, their operations and entities. In this case, the tasks of DETAGENT are depicted in Algorithm 3. For each entity $e$ that was accessed during the execution, it first builds the logical groups and then calls BUILDWRITEEDGES (algorithm 2) on each of these groups. After all write-edges have been built, it calls for each entity version $e_r$ read by at least one unit $U$ BUILDREADEDGES (algorithm 1) to build all edges that contain a read. This completes the construction of t-GDG. DETAGENT now calls for each unit node $U$ in the graph the algorithm DETECTCYCLES which detects all cycles in which $U$ is involved.

**Algorithm 3** OFFLINEDETECTION($allUnits$,$allEntities$)

**Require:** all units and all entities

**Ensure:** detects all cycles

```
1: for each entity e ∈ allEntities do
2:    build all logical groups of e
3:    for each group G ∈ groupsOf(e) do
4:       G_succ = successorOf(G)
5:       BUILDWRITEEDGES( G , G_succ )
6: for each unit U ∈ allUnits do
7:    for e_r ∈ U.readSet do
8:       BUILDREADEDGES (e_r, U)
9: for each unit U ∈ allUnits do
10:   DETECTCYCLES ( U )
```

DETECTCYCLES implements an extension $extDLS$ of the Depth-Limited-Search (DLS) algorithm [2], which does a depth-first search and stops at a certain depth limit. Our choice for DLS derives from the experience that cycles in dependency graphs do usually contain only a limited number of nodes, 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 $U$ and explores all its outgoing edges. At the end, it returns all possible paths starting from $U$ with a size smaller than $d$. If the last unit (node) $U_L$, in a certain extracted path, has any outgoing edge to $U$, then a cycle is detected. Our implementation of $extDLS$ guarantees that any detected cycle does not contain a pair of alternate edges since it is impossible that the underlying GDG would have a cycle with both of these edges. We mark every detected cycle as a real cycle or a potential cycle, depending on whether it contains alternate edges. Starting the cycle detection from a unit $U$, DLS has a complexity of $b^d$, where b is the branching factor (outgoing edges) and d is the depth, thus, the overall complexity for cycle detection is in the order of $n * b^d$ where n is the number of units.

### 3.3.2 On-line Detection

The on-line detection of anomalies is in principle similar to the off-line detection, but DETAGENT does not wait to receive the information for a full set of units of work. Instead, immediately after receiving the information about a unit of work $U$, it extends its current t-GDG and then checks for cycles involving $U$. As shown in algorithm 4, DETAGENT starts by building the write edges ($ww$, $t$-$ww$ and $at$-$ww$) for each version $e_c$ created by $U$ by calling the algorithm REBUILDWRITEEDGES. This might lead to some existing edges be rerouted, therefore the algorithm is called **REBUILD**. Then, it builds the read edges for reads of $U$ and finally detects cycles involving $U$.

There are several cases for rebuilding the write edges. (1) If the newly created version $e_c$ is not transitively concurrent to any other version, then a new group $G$ is created and contains only $e_c$. (2) If $e_c$ is transitively concurrent to some versions and all belong to the same logical group $G$, then $e_c$ simply joins $G$. (3) However, if $e_c$ is transitively concurrent to versions belonging to more than one group $G_1...G_n$, then a new group $G$ is formed by *merging* all $G_i(i : 1..n)$ and $e_c$ joins $G$. In the three cases, some new edges are added into the t-GDG and some that were already built might have to be removed due to the restructuring of the logical groups. Note that after removing any edge, DETAGENT checks if this has an impact on any previously detected cycles. In some cases, a cycle might disappear and has to be removed. Based on our experiments, such cycles are removed very quickly after their detection. We noticed as well that under the on-line mode, $extDLS$ stops quickly its exploration of outgoing edges since the graph is built incrementally and most of the outgoing edges starting from a unit $U$ are not build yet.

**Algorithm 4** ONLINEDETECTION( $U$ )

**Require:** a unit of work $U$

**Ensure:** detects cycles where $U$ is involved

```
1: for e_c ∈ U.writeSet do
2:    REBUILDWRITEEDGES (e_c, U)
3: for e_r ∈ U.readSet do
4:    BUILDREADEDGES (e_r, U)
5: DETECTCYCLES ( U )
```

## 3.4 Approximation Metric

Since we are approximating the exact GDG with t-GDG, we would like to know how accurate this approximation is. For this purpose, we define a metric $errGDG$ that reflects how t-GDG is different from GDG. As described earlier, t-GDG has the same nodes as GDG, so their main difference is in edges. Based on Lemma 2 (section 3.2), any edge present in GDG is either present in t-GDG or replaced with another edge in t-GDG. In contrast, some alternate edges ($at$-$ww$ and $rw$-$at$-$ww$) present in t-GDG do not necessarily represent edges in GDG. Recall that for each two concurrently created versions, we create two alternate $at$-$ww$ edges between their creator units. Only one of the two edges is real and the other is not. The same applies for $rw$-$at$-$ww$ edges. Therefore, half of $at$-$ww$ and $rw$-$at$-$ww$ edges are not real edges in the GDG. By taking this in consideration, the metric $errGDG$ can be expressed as:

$$errGDG = \frac{\|at - ww\| + \|rw - at - ww\|}{2 * (Total\,Number\,of\,Edges)}$$

where $\|at - ww\|$ and $\|rw - at - ww\|$ represent the number of $at$-$ww$ and $rw$-$at$-$ww$ edges respectively.

The main challenge now is to calculate the total number of edges in GDG as we do not know the exact GDG. Let's first calculate the number of edges involving an entity $e$ with ordered versions $e_0, e_1, ..., e_n$. Regardless of the distribution of these versions on logical groups of $e$, we know that in GDG, there are exactly $n$ $ww$-edges, but some of them are replaced with $t$-$ww$ or $at$-$ww$ edges

in t-GDG. As described earlier, the GDG and the t-GDG have the same $wr$-edges which can be detected by tagging entities. Each $wr$-edge created via a version $e_i (0 \leq i < n)$ generates a $rw$-edge. Therefore the number of $rw$-edges is nearly always the same as the number of $wr$-edges. The only exception is if the last version $e_n$ is read by any other units, as the resulting $wr$-edges does not have a corresponding $rw$-edge to a consecutive version. However, we consider these extra edges negligible with high values of $n$. By denoting $\|e\|_v$ the number of versions of the entity $e$, and by $\|wr(e)\|$ the number of $wr$ edges created by reading versions of $e$, the error metric $errGDG$ becomes:

$$errGDG = \frac{\|at - ww\| + \|rw - at - ww\|}{2 * \sum_{e} (\|e\|_v + 2 * \|wr(e)\|)}$$

Note that if there are no concurrently created versions, all logical groups of all entities will be of size 1, and there will be no $at$-$ww$ or $rw$-$at$-$ww$ edges. Also, there will be no $t$-$ww$ edges, since between each two consecutive logical groups of size 1 there is an exact $ww$ edge. These properties lead to the following corollary.

**COROLLARY 3.1**

If there are no concurrently created versions, i.e. all logical groups of all entities are of size 1, then t-GDG is the same as GDG.

## 3.5 Summary

In this section we have seen that anomalies are characterized by the presence of cycles in the GDG, a graph that we cannot completely observe at the application level in the presence of concurrent updates. Instead, we propose a new graph t-GDG that approximates the GDG, containing all detectable edges present in GDG, and replaces undetectable edges with approximative edges. The cycles in t-GDG are a superset of the cycles in the exact GDG. However, the additional cycles always contain alternate edges, reflecting that we do not exactly know the order of execution within the datastore. We have also proposed off-line and on-line algorithms for detecting these cycles. Finally, we proposed a metric that reflects how t-GDG is different from GDG.

The detected cycles can be exploited in several ways. In a testing phase, where an application is tested on several datastores and/or on different consistency levels, the number of cycles in a test run together with the measured response time and achievable throughput can be taken as metrics to compare the real trade-off between consistency and performance. The number of inconsistencies for a certain consistency level might be too high to be acceptable, or even under very low consistency levels no inconsistencies might occur. Furthermore, the transaction types responsible for most of the cycles can be determined, and possibly rewritten to reduce the amount of inconsistencies. Second, the mechanism can be used during runtime to detect anomalies as they occur. These anomalies can be visualized by a tool such as [33]. This allows the administrator to correct the affected entities and/or adjust their values.

# 4. APPLICATIONS AND IMPLEMENTATION

To see how our model can be applied to existing cloud platforms, we have analyzed both Google App Engine (GAE) datastores (master-slave and high-replication) as well as the Cassandra datastore.

## 4.1 Google App Engine

Google provides two cloud datastores: GAE master-slave and GAE high-replication. They are differentiated by their availability and consistency guarantees.

Both solutions allow explicit use of transactions for accessing entities. In the absence of such explicit usage, an individual access to an entity $e$ is encapsulated internally within a transaction that is limited to $e$. Entities are written to the datastore in two phases: commit and apply. In the commit phase, the entities are recorded into certain logs. The apply phase occurs after the commit phase and consists of two parallel operations: (1) entities are written into the datastore and (2) the index rows for these entities are written. Under high-replication, the apply phase occurs after the commit operation returns while under master-slave the commit operation returns only when both phases (commit and apply) have completed.

Updates of committed transactions are applied in the same order of their commit phases, but the updated versions may not be immediately visible to all types of subsequent queries and this depends on how entities are accessed. Under high-replication, once the commit operation of a transaction returns, any of its updated entities $e$ is visible to subsequent queries, if $e$ is accessed via its primary key. However, if $e$ is accessed via a predicate query, then it is not always guaranteed that the latest value of $e$ is returned.

In summary, both master-slave and high-replication provide similar services and they differ in (1) their availability and (2) when the latest updated entities versions become visible to subsequent queries. Both solutions guarantee that created versions of entities are stored in the datastore following the same order of their commit phases, which is the same order as when their respective commit operations return. This property satisfies our Assumption 2 of Section 2.5.1, where we assumed that the order of storing entities is the same as the commit order of the corresponding transactions. The Assumption 1 is also satisfied when a unit of work $U_j$ reads a version $e_i$ (created by $U_i$) then creates a new version $e_j$. In fact, $e_i$ becomes only visible after the commit of $U_i$, and since $U_j$ will commit later after the read operation of $e_i$, then $e_j$ will only be installed after $e_i$. Since both Assumption 1 and Assumption 2 are satisfied, our model for detecting anomalies is applicable for any application running under both platforms.

## 4.2 Cassandra

Cassandra is an open source multi-master cloud datastore where updates can occur on some replicas then propagated to the rest of replicas. It does not support transactions and entities are accessed via individual read and write operations, and each operation executes under its own consistency level. Both read and write operations may execute under ONE, QUORUM or ALL consistency levels. A write operation running under ONE returns immediately after the updated entity has been written on at least one replica. Under QUORUM, the write operation returns only when this update was written on the majority of replicas, and under ALL, it does not return before applying the update on all replicas. A read operation running under ONE returns immediately after the first replica responds to the query. Under QUORUM, the read operation returns only when the majority of replicas respond, and under ALL, it returns after all replicas have responded. Cassandra guarantees that a read operation returns the latest version written of an entity $e$ if read and write operations for accessing $e$ execute under one of the following three modes:

1. writes execute under ALL and reads execute under any mode
2. reads execute under ALL and writes execute under any mode
3. both reads and writes execute at least under QUORUM.

By accessing Cassandra under one of these three modes, both Assumption 1 and Assumption 3 are satisfied. Therefore, our model can be applied to any application running in this environment. In fact, if a unit $U_j$ reads first $e_i$, it means that $e_i$ is the latest version of $e$ on the datastore at the time of the read (under the three modes) and $e_j$ will be installed on at least one replica. Any subsequent read of $e$ will ensure that $e_j$ and no more $e_i$ is returned under the three modes. This means, that $e_i$ is ordered before $e_j$. Therefore, Assumption 1 is satisfied. Furthermore, when two write operations $w_i(e_i)$ and $w_j(e_j)$ execute consecutively, then Cassandra will order $e_i$ before $e_j$ and make $e_j$ the latest version. Once $w_j(e_j)$ has returned, any consecutive read will return $e_j$ (until a new update takes place).

#### 4.2.1 NTP Time

NTP synchronization is common for cloud datastores. GAE is managed by Google's servers and their machines are already synchronized with the NTP time. For Cassandra, in the case machines are co-located in a local area network, the machines can be synchronized with an internal NTP server. If the machines are not synchronized with NTP time, we recommend the use of some API that extracts the NTP time from a public NTP server. This can be achieved by making one call at the startup of each application, then refresh it periodically. Actually, Cassandra's mechanism to order entities in the temporal order they are written and to always return the latest version written relies on the application servers to be synchronized using NTP or similar[1].

Despite NTP synchronization, there can be an approximation error $\varepsilon$. In order to overcome this error, we recommend that each captured $pre$ and $post$ value to be adjusted with this $\varepsilon$ as follows: $pre = currentTime - |\varepsilon|$ and $post = currentTime + |\varepsilon|$

#### 4.2.2 Information interception and processing

For capturing information on units of work, we have used a collector agent COLAGENT that is integrated with the application's code. Therefore, each application tier replica has one instance of COLAGENT. It is based on AspectJ[5], and intercepts calls to the Java API methods that access the underlying datastore. It mainly captures methods reading or writing entities as well as the $pre$ and $post$ values for each newly created entity. COLAGENT sends the collected information to a detector agent DETAGENT that creates the graph and detects cycles either off-line or on-line. In the on-line approach, COLAGENT forwards collected information immediately after the completion of each unit of work. In contrast, under the off-line approach, it stores such information asynchronously on a separate table, that will be accessed later by DETAGENT. Note that a table can be logically represented under GAE by an entity group, and under Cassandra by a column family.

[1]Cassandra requires the application to provide a timestamp for each write operation and orders entity versions according to this timestamp. Cassandra's APIs generate this timestamp automatically using local machine time – thus it uses the same NTP synchronization assumption as we do. In fact, we can exploit this application induced ordering of write operations as we can extract at the application layer this timestamp to determine the exact creation time for each entity version. In this case, there will be no alternate edges in the t-GDG and both t-GDG and GDG will be identical.

#### 4.2.3 Tagging entities

Both GAE high-replication and Cassandra are schema-less and attributes can be added on the fly without any changes at the application's code. To track the creator of each entity, we add an attribute $unitInfo$ on the fly to any updated or created entity, where we store a unique identifier of its creating unit of work. The method $creatorOf$ explained earlier in our algorithms extracts the value of $unitInfo$ from a given entity then returns it. If the database is not schema-less and enforces a predefined format of entities, $unitInfo$ can be added as an extra attribute to the structure of entities. A special annotation for this attribute, for example $@UnitInfo$, allows COLAGENT to access it dynamically.

## 5. EXPERIMENTS

To see how our model works, we have conducted a set of experiments under both Google App Engine and Cassandra.

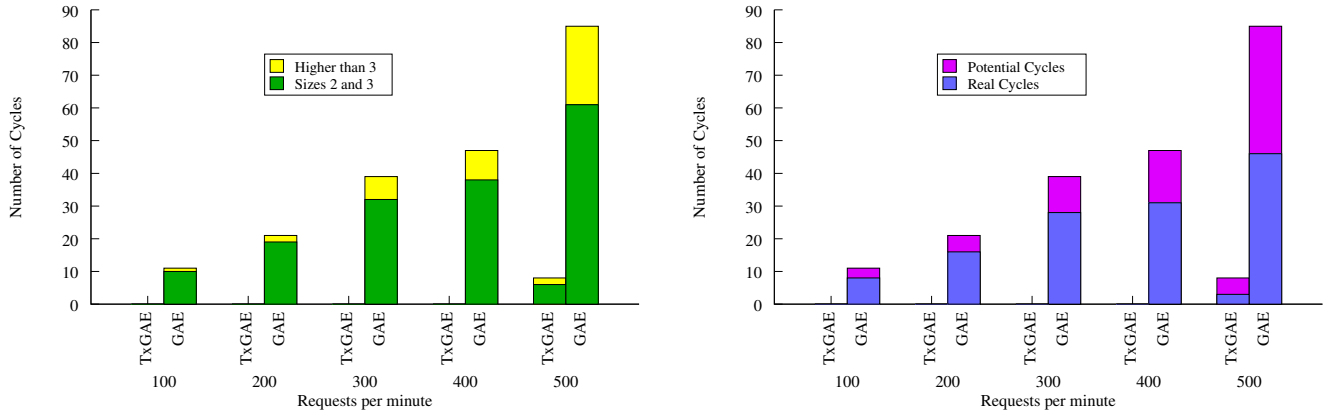
### 5.1 Google App Engine

**Application under Test.**

We have designed and implemented a microbenchmark that is inspired from Amazon *Today's deals* available under http://www.amazon.com. The microbenchmark uses three tables: *User*, *Item* and *Order*. A user browses items, adds some of them to its shopping cart, then places an order. A user can order one, two or three types of items with a different quantity for each type. Since the items available under *Today's deals* are limited, we have populated the *Item* table with 50 entities.

**Workload Configuration.**

We have used the JMeter[20] benchmark framework to emulate clients accessing our microbenchmark. We have configured it in such way to have 70% of read-only requests and 30% of read-write requests. The read-only requests are composed of 5% access to the *User* table, 5% to the *Order* table, and 60% to the *Item* table. The read-write requests access both the *Item* and *Order* tables and are divided as follows. 15% for buying one item, 10% for buying two items and 5% for buying 3 items. Regardless of the number of purchased items, only one entity is inserted into the *Order* table. We have performed several tests by varying the number of requests per minutes generated by JMeter, and this by running the microbenchmark under two configurations. First, we have run the microbenchmark under GAE while transactions are enabled. Under this configuration (TxGAE), any access to GAE datastore is within a transaction scope and each transaction is seen as a unit of work. The second configuration (GAE) is also under GAE, but without the use of transactions. Therefore units of work are represented by web requests. We have used GAE 1.5.0.1 Java SDK for accessing GAE datastore which is managed automatically on the cloud and the amount of allocated resources is transparent for us. For this configuration, we have used the off-line detection approach where each collector agent stores information on each unit of work as an additional record a separate table which will be accessed off-line by the detector agent. To minimize the overhead of this additional write operation on the current executing unit, we have used the concept of tasks under GAE, where the collector agent generates a quick task in memory, and the GAE engine automatically executes it shortly.




(a) Distribution of Sizes

(b) Real and Potential Cycles

**Figure 6: Detected Cycles and their Distribution under GAE**

**Results and Discussion.**

Figure 6(a) shows the number of detected cycles with increasing number of requests per minutes and this for the configurations TxGAE and GAE. For each configuration, 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 sizes of cycles. The figure presents the absolute number of cycles. The Figure 6(b) shows the same results but distinguishes between real and potential cycles. While the bottom part of each column represents real-cycles, the top part represents potential cycles. The numbers shown in both figures represent an average of a set of experiments for each throughput value.

The number of cycles increases with the load in the system. The number of potential cycles is generally lower than the number of real-cycles, but it becomes higher for high throughput values. A high proportion of cycles is either of size 2 or size 3 and this for all configurations. This confirms the assumption that consistency anomalies (cycles) involve generally a few number of units of work, and our cycle detection algorithm extDLS found most of cycles after following 2 or 3 edges from each node. Note that we started detecting cycles for TxGAE only at the throughput 500, and most of the detected cycles were potential. As the system is transactional the internal concurrency control mechanism avoids some form of cycles although it does not provide full serializability. However, for all our experiments for TxGAE, there were many aborted transactions, and in some cases the abort rate of read-write transactions was up to 20%. This explains the reduced number of detected cycles. For all our experiments under GAE, the overhead of the collector agent was less than 5% and the execution time of the detector agent was negligible.

## 5.2 Cassandra

**Application under Test.**

The Yahoo! YCSB benchmark is widely used to test the performance of cloud datastores. It consists of methods that represent typical units of work. We have added an additional type of unit of work that reads two entities and then updates them. This allowed us to stress-test the system by generating a workload where many units of work are interleaving and access the same entities.

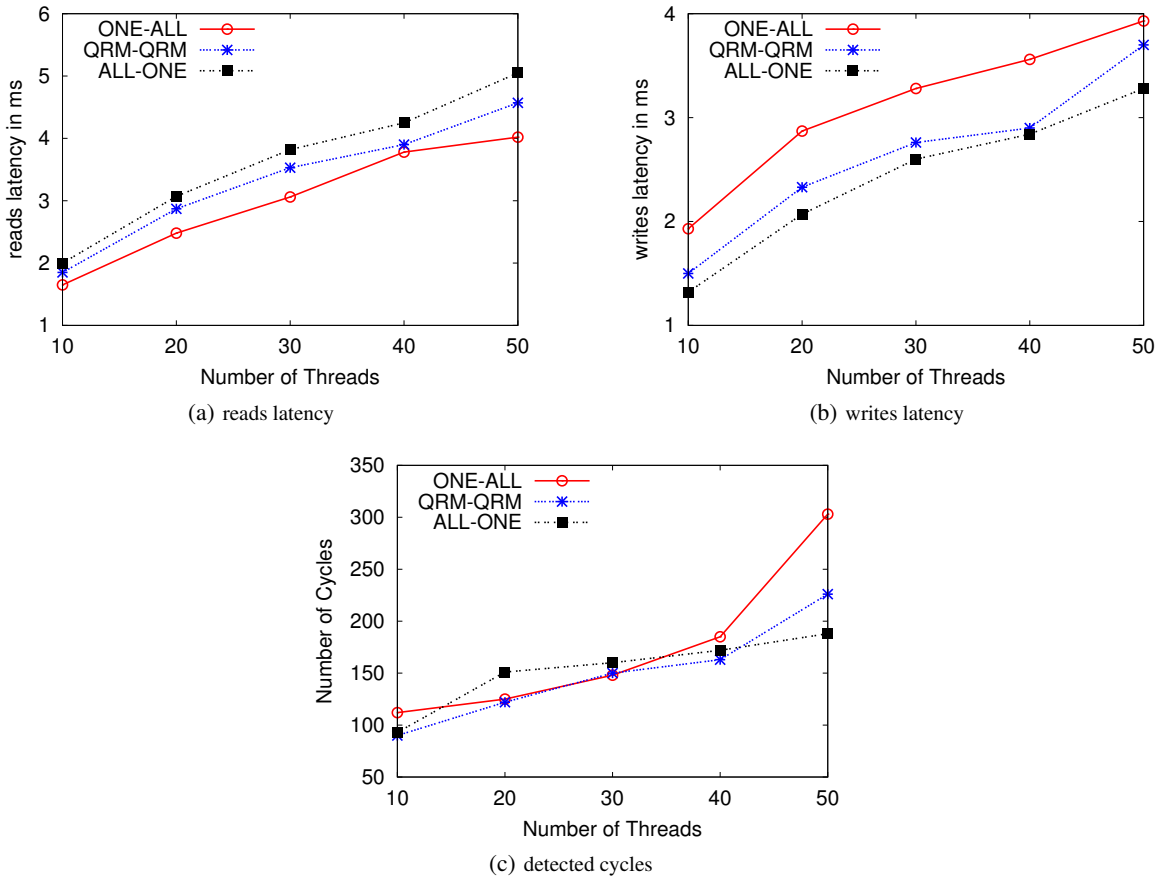
**Workload Configuration.**

We have configured YCSB to use 80% of read-only units, 10% of units reading one entity then updating it, and 10% for units reading two entities then updating them. All these units were accessing 10000 entities replicated on five Cassandra nodes (version 1.0.0) co-located in the same local area network. Each machine hosting an instance of Cassandra has an Intel Core2 2.66 GHz as processor with 8GB of memory and operating under Fedora Core 13. We have performed several tests by varying the number of concurrent threads under YCSB. We used three different modes. Under ONE-ALL, reads use consistency-level ONE, writes use ALL. Under QRM-QRM, all read and write operations execute under the consistency level QUORUM. At last, under ALL-ONE, all reads execute under ALL and all writes under ONE. For this configuration, we have used the on-line detection approach where the detector agent resides in the same network as the Cassandra nodes and it communicates with all collector agents via TCP/IP sockets. Similarly to GAE, the presented results are an average of a set of experiments for each number of threads values.

**Results and Discussion.**

The Figures 7(a) and 7(b) show respectively the latency for read and write operations. These figures show the expected costs of accessing one or more replicas. The question now is whether the choice of mode influences the conflict rate. Figure 7(c) shows the number of cycles detected under the three modes. In this particular case, we cannot see a significant difference in the number of detected cycles. Thus, the mode of replication can be chosen according to the response time requirements for read and write operations. For all our experiments under Cassandra, the overhead of the collector agent was less than 3%. As stated earlier in this paper, the relation *created-before* relied either Assumption 1 (read then write) or on Assumption 3 (write returns before next write starts) that we determine based on physical time. For our experiments, entity version order was determined by 95% exploiting Assumption 1. Thus, our reliance of accurate physical time is quite small.

The number of cycles presented in the experiments, can be used to make a better decision on the trade-off between consistency and response time requirements. For any cycle detected under both GAE and Cassandra, the ConsAD tool presented in [33] can be used to check in details which units of work, entities and business methods are involved in that cycle.




(a) reads latency

(b) writes latency

(c) detected cycles

**Figure 7: reads/writes latencies and cycles under Cassandra**

# 6. RELATED WORK

In [30], Wada et al. investigated stale reads and some client centric anomalies seen by clients for some cloud platforms. Their approach used a particular application that spans threads with writer or reader roles. For most of their experiments, threads with the writer role issue a write of the current time, and threads with reader role issue many reads for this stored time. In [25], Li et al. measure the "time to consistency" for some datastores. More specifically, they measure the time between when a data item is written to the datastore and when all reads for this data item return up to date results. Both approaches in [30] and [25] are specific to particular applications and concentrate on inconsistencies seen from the client side. In contrast, our model deals with data centric anomalies, and can be used for arbitrary cloud applications. Moreover, our model tells exactly which unit of work and which entities are involved in each anomaly.

In [32, 33], we proposed an on-line approach for quantifying consistency anomalies in multi-tier architectures, where multiple application servers access a centralized database. We supported any isolation level that is higher than or equal to read-committed and used its properties to detect data centric anomalies. We required that (1) access to entities is done only within transactions, (2) the studied application uses the same isolation level for all its transactions, (3) the database is not replicated while application servers can be replicated and (4) we used *internal* information from the application servers and the centralized database. In contrast in this paper, we use *external* information captured at the application level and the datastore can be replicated. Moreover, our new model supports applications running under any mixture of isolation levels and entities can be accessed in transactional and non-transactional contexts.

In [15], the authors quantify violations of integrity constraints when a database system runs under read-committed or snapshot-isolation. They analyze a microbenchmark that is composed of two related tables. Our approach differs from this work in several aspects. First, it is designed to work for arbitrary cloud applications. Second, it is not only able to detect anomalies that occur under snapshot-isolation or read-committed, but it supports any isolation guarantees. Moreover, the detection in our approach can be off-line or on-line.

Fekete et al. in [16] propose an approach where they characterize nonserializable executions for a given application when run under snapshot-isolation. They manually analyze the application in order to find possible conflicting operations. Based on this approach, Jorwekar et al. in [21] propose a tool which automates the detection of possible conflicts between a set of operations under snapshot-isolation. They extract these operations by manually analyzing the studied application and by automatically extracting information from the SQL database logs. Both [16] and [21] require some manual analysis of the studied application, and are dedicated to snapshot-isolation. They also only give information about possible conflicts and do not quantify how often such conflicts actually

occur during run-time. In contrast, our approach supports multiple consistency and isolation guarantees. It is by a large part independent of the datastore, completely automated, provides quantitative information about anomalies, and can be used off-line or on-line.

In this work, we focused on quantifying anomalies, not on the performance of the studied application. There have been several works that studied cloud applications from a perfomance point of view. For more details on these works, you can refer to [6, 22, 27].

# 7. CONCLUSION AND FUTURE WORK

This paper proposed an approach to detect consistency anomalies for arbitrary cloud applications. It can be used for a large variety of cloud datastores as it observes the execution at the application layer. If the behavior at the datastore cannot be completely observed we approximate it, leading us to distinguish between true and potential inconsistencies. We have conducted a set of experiments under both Google App Engine and Cassandra by using respectively the JMeter and YCSB benchmarks. As future work, we would like to analyze potential datastores for which conflict detection can be more precise at the application layer.

# 8. REFERENCES


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

[2] R. Akerkar. *Introduction to Artificial Intelligence*. PHI Learning Pvt. Ltd., illustrated edition, 2005.

[3] Amazon Relational Database Service. A distributed relational database service. http://aws.amazon.com/rds/.

[4] Amazon SimpleDB. A highly available and flexible non-relational data store. http://aws.amazon.com/simpledb/.

[5] AspectJ. The Java aspect-oriented extension. http://www.eclipse.org/aspectj/.

[6] C. Bennett, R. L. Grossman, D. Locke, J. Seidman, and S. Vejcik. Malstone: towards a benchmark for analytics on large data clouds. In *SIGKDD*, pages 145–152, 2010.

[7] 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.

[8] 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.

[9] E. A. Brewer. Towards robust distributed systems (abstract). In *PODC*, page 7, 2000.

[10] Cassandra. A highly scalable, distributed and structured key-value store. http://cassandra.apache.org/.

[11] S. Das, D. Agrawal, and A. E. Abbadi. Elastras: An elastic transactional data store in the cloud. In *USENIX HotCloud*, Boston,MA, 06/2009 2009. USENIX, USENIX.

[12] S. Das, D. Agrawal, and A. E. Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *SoCC*, pages 163–174, 2010.

[13] 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*, 19(11):624–633, 1976.

[14] A. Fekete. Serialisability and snapshot isolation. In *Proceedings of the Australian Database Conference*, pages 201–210, 1999.

[15] A. Fekete, S. Goldrei, and J. P. Asenjo. Quantifying isolation anomalies. *PVLDB*, 2(1):467–478, 2009.

[16] 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.

[17] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002.

[18] Google App Engine. A platform as a service cloud computing platform. http://code.google.com/appengine/.

[19] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. *SIGMOD Rec.*, 25:173–182, 1996.

[20] JMeter. A Java based performance measuring tool. http://jakarta.apache.org/jmeter/.

[21] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. In *VLDB*, pages 1263–1274. VLDB, 2007.

[22] D. Kossmann, T. Kraska, and S. Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 579–590, New York, NY, USA, 2010. ACM.

[23] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: Pay only when it matters. *PVLDB*, 2(1):253–264, 2009.

[24] J. J. Levandoski, D. B. Lomet, M. F. Mokbel, and K. Zhao. Deuteronomy: Transaction support for cloud data. In *CIDR*, pages 123–133, 2011.

[25] A. Li, X. Yang, S. Kandula, and M. Zhang. Cloudcmp: comparing public cloud providers. In *Internet Measurement Conference*, pages 1–14, 2010.

[26] Microsoft SQL Azure. A Microsoft cloud-based service offering data-storage capabilities. http://www.microsoft.com/windowsazure/.

[27] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, O. Fox, and D. Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0, 2008.

[28] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD Conference*, pages 1–12, 2012.

[29] H. T. Vo, C. Chen, and B. C. Ooi. Towards elastic transactional cloud storage with range query support. *PVLDB*, 3(1):506–517, 2010.

[30] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *CIDR*, pages 134–143, 2011.

[31] WebFilings. A cloud-based company hosting applications for financial and executive teams. http://www.webfilings.com/.

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

[33] K. Zellag and B. Kemme. Consad: a real-time consistency anomalies detector. In *SIGMOD Conference*, pages 641–644, 2012.

[34] W. Zhou, G. Pierre, and C.-H. Chi. Cloudtps: Scalable transactions for web applications in the cloud. *IEEE Transactions on Services Computing*, 99(PrePrints), 2011.

# APPENDIX

# A. PROOFS

## A.1 Proof of Lemma 3.1

$e_i < e_j$ means that $e_i$ is created before $e_j$, and there are two cases. Either (1) $e_j$ is created immediately after $e_i$, or (2) there are $p$ consecutive versions $e_{v1}...e_{vp}$ of $e$ that were created between $e_i$ and $e_j$. For case (1), there is a direct $ww$-edge from $U_i$ to $U_j$. And for case (2), there is:

1. $ww$-edge from $U_i$ to $U_{v1}$
2. $ww$-edge from $U_{vk}$ to $U_{v(k+1)}$ , $k = 1..(p-1)$
3. $ww$-edge from $U_{vp}$ to $U_j$

Therefore, in both cases, there is a path of $ww$-edges from $U_i$ to $U_j$.

## A.2 Proof of Lemma 3.2

An edge $a$ in GDG, from unit $U_i$ to a unit $U_j$ due to an entity $e$, can be of three types: $wr$, $ww$, or $rw$.

***case-1***: $a.type = wr$.

The $wr$-edge in GDG is the same as in t-GDG as we can detect this dependency by storing the identifier of the unit that created the version read.

***case-2***: $a.type = ww$.

Assume $U_i$ created $e_i$ and $U_j$ created $e_j$. The $ww$-edge means that $e_j$ is the immediate successor of $e_i$ in the datastore. By comparing the versions $e_i$ and $e_j$, there are three possibilities of observation: $e_i < e_j$, $e_i \parallel e_j$ or $e_i > e_j$.

*case 2-1* : if $(e_i < e_j)$ then there are three sub-cases:

*case 2-1-1* : $e_i$ and $e_j$ are in the same logical group

Based on Algorithm 2, $e_i$ and $e_j$ are in the same logical group and $e_i < e_j$, in line 11 we build a $t$-$ww$-edge, in t-GDG, from $U_i$ to $U_j$.

*case 2-1-2* : $e_i$ and $e_j$ belong to two consecutive logical groups $G_1$ and $G_2$, i.e., $e_i \in G_1$ , $e_j \in G_2$ and $G_2 = successorOf(G_1)$.

Based on Algorithm 2, if both $G_1$ and $G_2$ are of size 1 then we build (in line 5) a $ww$-edge from $U_i$ to $U_j$ in t-GDG. Otherwise, we build (in line 7) a $t$-$ww$-edge in t-GDG from $U_i$ to $U_j$.

*case 2-1-3* : $e_i$ and $e_j$ belong to two different non-consecutive groups $G_1$ and $G_2$, with $e_i \in G_1$ and $e_j \in G_2$.

$G_2$ is not the successor of $G_1$ means that there is at least another group $G_k$ such that $G_k$ is between $G_1$ and $G_2$. This means that there is at least one version $e_k \in G_k$ that has been created between $e_i$ and $e_j$, which contradicts the fact that $e_j$ is the immediate successor of $e_i$. Therefore the case that $e_i$ and $e_j$ belong to two different non-consecutive groups is not possible.

*case 2-2* : $e_i \parallel e_j$

As stated in Algorithm 2 (lines 13 and 14), if $e_i \parallel e_j$, then we build two alternate $at$-$ww$ edges, in t-GDG, between $U_i$ and $U_j$. The first is from $U_i$ to $U_j$ and its alternate from $U_j$ to $U_i$. Therefore $a$ is replaced with another edge ($at$-$ww_{ij}$), in t-GDG, from $U_i$ to $U_j$.

*case 2-3* : $e_i > e_j$

This means that one of the three conditions *created-before*-**a**, *created-before*-**b** or *created-before*-**c** is valid when comparing $e_j$ to $e_i$. In the three cases, $e_i$ should be created after $e_j$, and since there is a $ww$-edge from $e_i$ to $e_j$, this means that $e_i$ is created before $e_j$ which leads to a contradiction. Therefore the comparison $e_i > e_j$ is not possible.

***case-3***: $a.type = rw$.

The presence of a $rw$ edge requires the presence of one $wr$ edge and one $ww$ edge. Let's denote by $wr_a$ and $ww_a$ the edges required to conclude the edge $a$. As stated earlier in *case-1*, $wr_a$ is the same in both GDG and t-GDG, while $ww_a$ (based on *case-2*) can be represented by edges of type $ww$, $t$-$ww$ or $at$-$ww$ in t-GDG. If $ww_a$ has a corresponding $ww$-edge in t-GDG, then $a$ is a $rw$-edge in t-GDG. If $ww_a$ is represented by a $t$-$ww$-edge in t-GDG, then $a$ is a $rw$-$t$-$ww$-edge in t-GDG. And finally, if $ww_a$ is represented by a $at$-$ww$-edge in t-GDG, then $a$ is a $rw$-$at$-$ww$-edge in t-GDG. In all three cases, $a$ is represented by an edge from $U_i$ to $U_j$ in t-GDG.

## A.3 Proof of Theorem 3.1

By using lemma 3.2, each edge in $C$ from a unit $U_i$ to a unit $U_{i+1}$ is represented by an edge from $U_i$ to $U_{i+1}$ in t-GDG. Hence all edges of $C$ in GDG are represented by edges in t-GDG which leads to a cycle in t-GDG between $U_1...U_n$ in the same order.

## A.4 Proof of Theorem 3.2

If $t$-$C$ does not contain any alternate edge, then it will contain only edges of type $wr$, $ww$, $rw$, $t$-$ww$ and $rw$-$t$-$ww$. Since $wr$, $ww$ and $rw$ are the same in both GDG and t-GDG, what remains to discuss are the $t$-$ww$ and $rw$-$t$-$ww$ edges. Note that any $t$-$ww$ represents a path of $ww$-edges (one or more) while $rw$-$t$-$ww$ is composed of a $rw$ plus an optional path of $ww$-edges ($ww*$: 0 or more) as shown in Figure 5(b). Then any edge of type $t$-$ww$ from any unit $U_i$ to its successor $U_{i+1}$ in $t$-$C$ can be replaced with its real $ww$ sequence of edges in GDG, and the same applies to $rw$-$t$-$ww$ edges that are replaced with $rw + ww*$ edges which are the real edges in GDG. Therefore, any transitive $edge$ from $U_i$ to $U_j$ in t-GDG is replaced with a path from $U_i$ to $U_j$ in GDG. By concatenating these paths, a cycle C can be found in GDG, and the $U_1...U_n$ units appear in it in the same order at in the cycle $t$-$C$ in t-GDG.

In contrast, if $t$-$C$ contains an alternate edge, then it should be of type $at$-$ww$ or $rw$-$at$-$ww$. Since $rw$-$at$-$ww$ requires the presence of an $at$-$ww$-edge, we limit the reasoning to $at$-$ww$ edges. A pair of $at$-$ww$ edges is created between units $U_i$ and $U_j$ if they created concurrent versions. One of the two edges represents a real $t$-$ww$ edge from $U_i$ to $U_j$, the other is in the wrong direction. As we do not know which of the two represents the exact dependency path, we call the cycle a *potential* cycle.