International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:8, No:7, 2014

Ensuring Consistency under the Snapshot Isolation

Carlos Roberto Valêncio, Fábio Renato de Almeida, Thatiane Kawabata,
Leandro Alves Neves, Julio Cesar Momente, Mario Luiz Tronco,
Angelo Cesar Colombini

Abstract—By running transactions under the SNAPSHOT isolation we can achieve a good level of concurrency, specially in databases with high-intensive read workloads. However, SNAPSHOT is not immune to all the problems that arise from competing transactions and therefore no serialization warranty exists. We propose in this paper a technique to obtain data consistency with SNAPSHOT by using some special triggers that we named DAEMON TRIGGERS. Besides keeping the benefits of the SNAPSHOT isolation, the technique is specially useful for those database systems that do not have an isolation level that ensures serializability, like Firebird and Oracle. We describe all the anomalies that might arise when using the SNAPSHOT isolation and show how to preclude them with DAEMON TRIGGERS. Based on the methodology presented here, it is also proposed the creation of a new isolation level: DAEMON SNAPSHOT.

Keywords—Data consistency, serialization, snapshot.

I. INTRODUCTION

SNAPSHOT is an isolation level not originally described by the ANSI specifications [1] and requires a multiversion concurrency control system in order to maintain multiple versions of the data items in a database [2], [3]. Fundamentally, the SNAPSHOT isolation just adds the detection of concurrent *write-write* conflicts to these systems.

In systems where the SNAPSHOT isolation is available, the transactions receive unique identifiers at the time they start and commit, known as *start-timestamp* (t_S) and *commit-timestamp* (t_C). In general these identifiers are assigned by a centralized service to ensure the order of start and end of each transaction and the uniqueness of each moment [2].

A transaction T_i under the SNAPSHOT isolation sees only the consistent state of the database at the time of its beginning, in other words, only those data with *commit-timestamp* δ less than the *start-timestamp* of the transaction T_i ($\delta < t_{S_i}$). This snapshot of the database is totally immune to operations performed by concurrent transactions and behaves as if it were under the action of a single transaction. For the SNAPSHOT transaction, the only accessible data are those committed before its beginning or generated by itself [2]–[4].

Two transactions T_i and T_j are in conflict if they have spatial and temporal overlap. A spatial overlap occurs when both transactions write the same data item, giving rise to a *write-write* conflict. In turn, a temporal overlap occurs when T_i and T_j run concurrently, that is $t_{S_i} < t_{C_j}$ and $t_{S_j} < t_{C_i}$ [2]. Thus, a transaction T_i under the SNAPSHOT isolation is allowed to commit only if no other transaction with *commit-timestamp* δ in the running time of T_i ($t_{S_i} < \delta < t_{C_i}$) wrote a data item also written by T_i [4].

Implementations of the SNAPSHOT isolation have the advantage that writes of a transaction do not block the reads of others. Moreover, because the protocol is concerned only with *write-write* conflicts, there is no transaction management overhead on read data items [2]. But even working on a consistent and isolated view of the database, and also preventing concurrent *write-write* conflicts, transactions running under the SNAPSHOT isolation are not free from inconsistencies in data [2]. In this paper we describe a technique that keeps the benefits of SNAPSHOT use and, at the same time, ensures only serializable histories. Although initially designed as an engineering solution, the technique can be generalized to a new isolation level that considers business rules among data items.

II. SNAPSHOT ANOMALIES AND HOW TO AVOID THEM

Before delving into the details of the SNAPSHOT anomalies, the following history notation must be considered [5]:

- $w_i(x_i)$ — Transaction T_i writes data item x.
- $w_i(x_i,v)$ — The same as $w_i(x_i)$, but the value v written is described.
- $r_i(x_j)$ — Transaction T_i reads data item x, which has been previously written by transaction T_j. When $j = 0$, it means data item x has been previously loaded into the database.
- $r_i(x_j,v)$ — The same as $r_i(x_j)$, but the value v read is described.
- c_i — Transaction T_i commits.
- a_i — Transaction T_i aborts.

A. Case 1: Write Skew

Consider the relational schema depicted in Fig. 1, where two tables are defined: People and Accounts. Each row in the People table will represent a person in real life and each row in the Accounts table will represent one bank account. According to the schema, a person is not required to have a bank account, but such account, when exists, must be assigned to only one person. Whereas the attribute Customer keeps the holder of an account, the attribute Spouse links a person to his or her spouse, who must be another person.

C. R. Valêncio, F. R. Almeida, T. Kawabata, L. A. Neves, and J. C. Momente are with the Department of Computer Science and Statistics, São Paulo State University – UNESP, São José do Rio Preto, SP, Brazil (e-mails: valencio@ibilce.unesp.br, fabiorenato.al@gmail.com, thatianekawabata@gmail.com, leandro@ibilce.unesp.br, juliocesar.momente@gmail.com).

M. L. Tronco is with the Department of Mechanical Engineering, University of São Paulo – USP, São Carlos, SP, Brazil (e-mail: mltronco@sc.usp.br).

A. C. Colombini is with the Department of Computer Science, Federal University of São Carlos – UFSCAR, São Carlos, SP, Brazil (e-mail: accolombini@dc.ufscar.br).

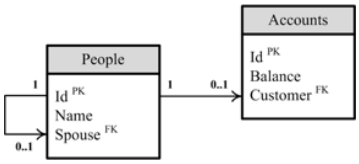

Fig. 1. Relational schema representing the People and Accounts tables

In Fig. 2 we can see a possible instance of a relational database designed according to the schema depicted in Fig. 1. The rows in the People table with Id values 1 and 2 represent a couple and therefore are linked together through the attribute Spouse. For those people who have a bank account (customers), the attribute Customer keeps the Id value of the person holding it.

People		
Id	Name	Spouse
1	Sheldon	2
2	Amy	1
3	Leonard	
4	Penny	

Accounts		
Id	Balance	Customer
1	50.00	1
2	50.00	2
3	100.00	4

Fig. 2. Content of the People and Accounts tables

The following business rule exists as well: a withdrawal can be made from a customer's account, even in case of no sufficient funds, considering that the account balance of the customer's spouse can cover the remaining debt.

Essentially the rule dictates that an account balance can become negative as long as the total balance of the couple cannot. Consider now two transactions T_1 and T_2 with the following characteristics:

– T_1 attempts to withdraw \$60 from the account with Id = 1.
– T_2 attempts to withdraw \$60 from the account with Id = 2.

In series just one of the transactions would be completed successfully and thus the database would remain in a consistent state. However, consider the following history H_1 as the result of an interleaved execution of the transactions, where x represents the account with Id = 1 and y represents the one with Id = 2.

H_1: $r_1(x_0, 50)$ $r_2(y_0, 50)$ $r_1(y_0, 50)$ $w_1(x_1, -10)$ c_1
$r_2(x_0, 50)$ $w_2(y_2, -10)$ c_2

Initially transaction T_1 checks the balance of account x and transaction T_2 does the same with account y. T_1 notices that account x has no sufficient funds to cover the withdrawal and then starts checking the balance of account y. Transaction T_1 then confirms the possibility of withdrawing and subtracts \$60 from the balance of account x, committing right after. By the time T_1 ends, the database is still in a consistent state. Following that, T_2 notices that account y has no sufficient funds to cover the withdrawal and then starts checking the balance of account x. We should note at this point that even though T_1 has already committed, T_2 sees only a previous version of the value written in x by T_1. Transaction T_2 wrongly concludes the possibility of withdrawing and then subtracts \$60 from the balance of account y, committing right after. By the time T_2 ends, the balance of both accounts are taken to $\$-10$, thus violating the business rule and leading the database to an inconsistent state. History H_1 is, therefore, not serializable but possible to happen under the SNAPSHOT isolation.

In order to avoid the anomaly we can define a special trigger for the Accounts table. Basically, the trigger must enforce the following constraint to the database management system (DBMS): every time an account balance is updated by a transaction running under the SNAPSHOT isolation, 1) the holder of the account must be identified, 2) the spouse of the holder must also be identified (if he or she exists), and 3) the account of the spouse (if it exists) must be written.

As a result of the trigger insertion, the history H_1 would be executed as follows, where the notations $r_i\{\Delta_j\}$ and $w_i\{\Delta_i\}$ denote, respectively, read and write operations made by transaction T_i on data item Δ because of the trigger execution.

H_1': $r_1(x_0, 50)$ $r_2(y_0, 50)$ $r_1(y_0, 50)$
$w_1(x_1, -10)$ $r_1\{\alpha_0\}$ $r_1\{\beta_0\}$ $w_1\{y_1\}$ c_1 $r_2(x_0, 50)$
$w_2(y_2, -10)$ $r_2\{\beta_0\}$ $r_2\{\alpha_0\}$ $w_2\{x_2\}$ a_2

This time though there is a trigger defined for the rows in the Accounts table and because the attribute Balance was updated by transaction T_1, the trigger's body is executed. As the trigger runs, the data items α, β, and y are also accessed, which represent respectively the holder of the account x, the spouse of the holder, and the spouse's account. Three more actions are processed by the trigger when T_2 updates the balance of account y, this time in the holder of account y (β), the spouse of the holder (α), and the spouse's account x. By the time T_2 tries to commit, the SNAPSHOT isolation detects a spatial and temporal overlap due to the writings in data items x and y made by both transactions T_1 and T_2. So, transaction T_2 aborts and the database remains in a consistent state.

B. Case 2: Phantom

Consider the relational schema depicted in Fig. 3, where three tables are defined: Employees, Projects, and Tasks. Each row in the Employees table will represent an employee in real life and each row in the Projects table will represent a project currently being developed by a company. A project may be assigned to one or more employees, and an employee can work on many projects as long as the total daily hours does not exceed eight. The assignment of projects is done by inserting records in the Tasks table, where the attribute Hours specifies the number of daily hours that the employee must devote to the project.

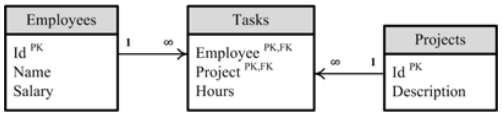

Fig. 3. Relational schema representing the Employees, Projects, and Tasks tables

In Fig. 4 we can see a possible instance of a relational database designed according to the schema depicted in Fig. 3. As we can see in the Tasks table, the employee with Id = 1 works in three projects, totaling six daily hours.

Employees		
Id	Name	Salary
1	Howard	5,000.00
2	Rajesh	9,000.00

Projects	
Id	Description
1	Hit the moon
2	Sheldon's secret
3	Top model house
4	Project A
5	Project B

Tasks		
Employee	Project	Hours
1	1	4
1	2	1
1	3	1
2	1	2
2	2	2
2	3	2
2	4	1
2	5	1

Fig. 4. Content of the Employees, Projects, and Tasks tables

Consider now two transactions T_1 and T_2 with the following characteristics:
– T_1 attempts to allocate one daily hour of the employee with Id = 1 for the project with Id = 4.
– T_2 attempts to allocate two daily hours of the employee with Id = 1 for the project with Id = 5.

In series just one of the transactions would be completed successfully and thus the database would remain in a consistent state. However, consider the following history H_2 as the result of an interleaved execution of the transactions, where a, b, and c represent, respectively, the first three already assigned tasks to the employee in the Tasks table, and x and y represent the possible tasks to be inserted by transactions T_1 and T_2, respectively.

H_2: $r_1(a_0,4)$ $r_1(b_0,1)$ $r_1(c_0,1)$ $r_2(a_0,4)$ $r_2(b_0,1)$ $r_2(c_0,1)$ $w_1(x_1,1)$ c_1 $w_2(y_2,2)$ c_2

Transaction T_1 checks the assigned hours for tasks a, b, and c and concludes that the employee still has two daily hours available. In the same way, transaction T_2 also concludes that the employee has two daily hours available. T_1 then assigns a new one-hour task x to the employee, committing right after. By the time T_1 ends, the database is still in a consistent state. Following that, T_2 assigns a new two-hour task y to the employee, committing right after. By the time T_2 ends, the database is led to an inconsistent state because the employee receives a nine-hour working day. History H_2 is, therefore, not serializable but possible to happen under the SNAPSHOT isolation.

In order to avoid the anomaly we can define a special trigger for the Tasks table. Basically, the trigger must enforce the following constraint to the DBMS: every time a task is assigned to an employee by a transaction running under the SNAPSHOT isolation, 1) the employee itself must be written.

As a result of the trigger insertion, the history H_2 would be executed as follows.

H_2': $r_1(a_0,4)$ $r_1(b_0,1)$ $r_1(c_0,1)$ $r_2(a_0,4)$ $r_2(b_0,1)$ $r_2(c_0,1)$ $w_1(x_1,1)$ $w_1\{\alpha_1\}$ c_1 $w_2(y_2,2)$ $w_2\{\alpha_2\}$ a_2

This time there is a trigger defined for the rows in the Tasks table and because the attribute Hours was updated by transaction T_1, the trigger's body is executed and the employee α is also written. When T_2 assigns a new two-hour task y to the employee, one more write action is performed on the employee α, this time by transaction T_2. By the time T_2 tries to commit, the SNAPSHOT isolation detects a spatial and temporal overlap due to the writings in data item α made by both transactions T_1 and T_2. So, transaction T_2 aborts and the database remains in a consistent state.

C. Case 3: Read-Only Transaction Anomaly

Consider the relational schema depicted in Fig. 5, where three tables are defined: Customers, SavingsAccounts, and CheckingAccounts. Each row in the Customers table will represent a bank customer in real life and each row in the SavingsAccounts and CheckingAccounts tables will represent, respectively, the customer's savings and checking accounts. According to the schema, every bank customer must have both types of accounts, which in turn must be assigned to only one customer. The attribute Customer in both account tables keeps the Id of the customer.

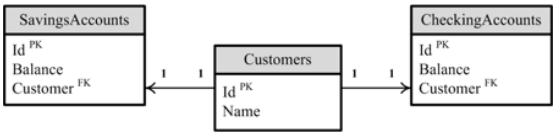

Fig. 5. Relational schema representing the Customers, SavingsAccounts, and CheckingAccounts tables

In Fig. 6 we have a possible instance of a relational database designed according to the schema depicted in Fig. 5. As we can see, the row in the Customers table with Id = 1 represents a bank customer that has a savings account with Id = 10 and a checking account with Id = 101.

Customers	
Id	Name
1	Sheldon
2	Leonard
3	Howard
4	Rajesh

SavingsAccounts		
Id	Balance	Customer
10	0.00	1
20	80.00	2
30	100.00	3
40	190.00	4

CheckingAccounts		
Id	Balance	Customer
101	0.00	1
102	18.00	2
103	10.00	3
104	999.00	4

Fig. 6. Content of the Customers, SavingsAccounts, and CheckingAccounts tables

The following business rule exists as well: a withdrawal can be made from any of a customer's account, even in case of no sufficient funds, considering that the balance of the other account can cover the remaining debt; the withdrawal is also granted if the total balance cannot cover the debit, but a penalty charge of $1 is added to the account being operated.

Consider now three transactions T_1, T_2, and T_3 with the following characteristics:
– T_1 deposits $20 into the savings account with Id = 10.
– T_2 makes a withdrawal of $10 from the checking account with Id = 101.
– T_3 is a read-only transaction that checks the balance of both accounts.

If T_1 runs before T_2, in other words, if the deposit is done before the withdrawal, then the savings account balance will

be \$20 and the checking account balance will be $\$-10$. In case T_2 runs before T_1, that is to say, the withdrawal is done before the deposit, then the checking account balance will be $\$-11$ and the savings account balance will be \$20. In theory, transaction T_3 should always see a consistent state of the database, especially because it is a read-only SNAPSHOT transaction. However, consider the following history H_3 as the result of an interleaved execution of the transactions, where x represents the checking account and y represents the savings account.

H_3: $r_2(x_0, 0)$ $r_2(y_0, 0)$ $r_1(y_0, 0)$ $w_1(y_1, 20)$ c_1
$r_3(x_0, 0)$ $r_3(y_1, 20)$ c_3 $w_2(x_2, -11)$ c_2

At the end we have a checking account balance of $\$-11$ and a savings account balance of \$20, but transaction T_3 returns a checking account balance of \$0 and a savings account balance of \$20. According to [6], T_3 result is inconsistent considering the final state of the database: T_3 confirms that the deposit was done before the withdrawal and, in this case, no penalty charge should have been added and the final result should be a checking account balance of $\$-10$ instead of $\$-11$. History H_3 is, therefore, not serializable but possible to happen under the SNAPSHOT isolation.

In order to avoid the anomaly we can define special triggers for both accounts tables. Basically, the definition of the triggers must enforce the following constraint to the DBMS: every time the balance of one of the accounts of a customer is updated by a transaction running under the SNAPSHOT isolation, 1) the other account of the customer must also be written.

As a result of the definition of the two triggers, the history H_3 would be executed as follows.

H_3': $r_2(x_0, 0)$ $r_2(y_0, 0)$ $r_1(y_0, 0)$ $w_1(y_1, 20)$ $w_1\{x_1\}$ c_1
$r_3(x_0, 0)$ $r_3(y_1, 20)$ c_3 $w_2(x_2, -11)$ $w_2\{y_2\}$ a_2

This time, when transaction T_1 deposits \$20 into the customer's savings account y, the trigger for the SavingsAccounts table is executed and the customer's checking account x is written. As soon as transaction T_2 makes the withdrawal from the customer's checking account x, the trigger for the CheckingAccounts table is executed and the customer's savings account y is also written. By the time T_2 tries to commit, the SNAPSHOT isolation detects a spatial and temporal overlap due to the writings in data items x and y made by both transactions T_1 and T_2. So, transaction T_2 aborts and the database remains in a consistent state. We should also note that transaction T_3, which was previously reported as inconsistent considering the final state of the database, now gets consistent results.

III. DAEMON TRIGGERS THEORY

We have named the special triggers defined so far as DAEMON TRIGGERS because they primarily work as background entities that only ensure data consistency by performing some identity writes, which in fact do not change the values in a row but are sufficient for the detection of *write-write* conflicts by the SNAPSHOT protocol. Essentially, a DAEMON TRIGGER must be defined considering a business rule and a dependency path among the data items that are under a constraint.

For Case 1, there is a business rule enforcing that the possibility of a withdrawal from a customer's account is conditioned to the balance of his or her spouse's account, so when one of the balances is updated, the DAEMON TRIGGER must perform an identity write on the other balance. Due to the normalization of the database, some tuples must be accessed in order to be able to reach one account from the other, establishing a dependency path between the two accounts.

In the same way, there is a business rule in Case 2 enforcing that an employee daily hours must not exceed eight, so when a new task is assigned to an employee, the DAEMON TRIGGER must perform an identity write on the employee itself. In this case the employee and task tuples are considered to be under a constraint.

Finally, in Case 3 there is a business rule enforcing that the application or not of a penalty charge during a withdrawal operation depends on the balance of both accounts of a customer, so when one of the balances is updated, the DAEMON TRIGGER must perform an identity write on the other balance.

IV. TESTS AND RESULTS

Benchmark systems such as TPC-C do not trigger the occurrence of the Write Skew phenomenon [4], [7] or any of the other SNAPSHOT anomalies and hence are not suitable for measuring the efficiency of an algorithm based on SNAPSHOT and adapted to provide serialization warranty. Because this work is concerned with this assurance, the tests must consider workloads that are not serializable under the SNAPSHOT isolation so as to measure the cost associated with obtaining only serializable histories. Therefore, the adopted system employs features of the benchmark SmallBank [8] and the test platform used in [7].

Originally the benchmark system SmallBank was designed for the relational model aiming the occurrence of the problem "Read-only transaction anomaly," identified by [6]. A similar system, named SmallBank++, was built in this work and also aims the occurrence of anomalies when transactions run under the SNAPSHOT isolation. As in SmallBank, the benchmark SmallBank++ provides a set of transactional tasks that reflect the functionalities of a simple banking system. The relational schema used by SmallBank++ is depicted in Fig. 7.

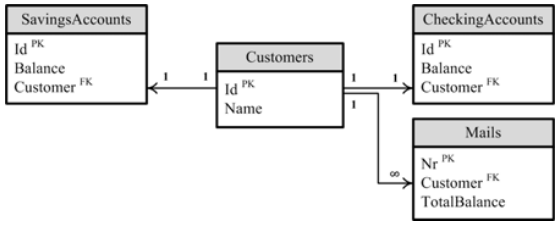

Fig. 7. Relational schema used by SmallBank++

SmallBank++ uses the same relational schema described in Section II Case 3, but with one additional table: Mails. Each row in the Mails table will represent a notification sent to a customer even by the traditional posting service or by e-mail reporting his or her current total balance. In SmallBank++ the

following business rule must be enforced: a withdrawal can be made from any of a customer's account, even in case of no sufficient funds, considering that the balance of the other account can cover the remaining debt; the total balance of the two accounts should never be negative and a customer mail notification should never be processed twice for the same total balance.

SmallBank++ consists of six transactional tasks:

1) Check balances — Checks the balance of the two accounts of a customer.
2) Change savings account — Performs a deposit or withdrawal from a customer's savings account.
3) Change checking account — Performs a deposit or withdrawal from a customer's checking account.
4) Transfer of funds — Transfers the total balance available in both accounts of a customer to the checking account of another customer.
5) Cheque processing — Performs the clearing of a cheque emitted by a customer.
6) Mail notification — Checks the total balance of a customer and verifies if he or she has been notified about it by looking for a corresponding entry in the Mails table. A notification is sent to the customer only if the entry does not exist.

Initially the benchmark system loads the database with one thousand customers and their respective accounts. During this process a random value between $0 and $10,000 is generated for each customer, with half of this amount being allocated to the savings account and the other half being assigned to the checking account. The workload to be processed is then divided into 10 steps (S_1 to S_{10}), where 100 transactional tasks are executed in S_1, 200 tasks are executed in S_2, and so on until S_{10} in which 1000 tasks are executed. In the end a total of 5500 tasks are executed. Before the start of each step, 10% of the customers are randomly chosen so that 90% of the tasks are executed on them (*hotspots*). About 50% of the workload is set up by Task 1 (Check balances) and the remainder is distributed with equal probability among the other tasks, thus characterizing a 50% read to 50% read-update ratio. The benchmark system then registers a concurrency error for each unsuccessful attempt to commit. At the end of each step, SmallBank++ verifies the balances of all accounts and looks for unwanted sent mails in order to identify whether the database has been taken to an inconsistent state.

The workload is processed first considering a single running transaction. Soon after the database is cleaned and the process of customers generation and their respective accounts is executed again before a further workload is dispatched, this time considering 8, 16, 32, 64, and 128 concurrent transactions. In all cases the actions of each task are triggered one at a time and in interleaved fashion according to the number of currently running transactions.

Our first test ran on an instance of SQL Server 2012 Express Edition using READ COMMITTED, SNAPSHOT, and SERIALIZABLE isolation levels. In the second test we ran SmallBank++ on an instance of Firebird 2.5.2 using READ COMMITTED, SNAPSHOT, and SNAPSHOT TABLE STABILITY isolation levels. The SNAPSHOT TABLE STABILITY isolation is different from the SNAPSHOT isolation in a way that only one transaction can write into a table at a time. In order to verify the efficiency of our technique, we evaluated the SNAPSHOT isolation twice, with and without DAEMON TRIGGERS. The serialization and inconsistency graphs obtained from both tests are depicted in Fig. 8.

As we can see, there are no serialization errors or concurrency anomalies (inconsistencies) when only one transaction is running at a time, no matter what is the isolation level in use. As the number of simultaneous running transactions increases, chances are the ratios of serialization errors and inconsistencies also increase. The only native isolation level that ensures serializability on SQL Server is SERIALIZABLE, which aborted about 28% of the 5500 tasks when 128 transactions ran concurrently. An implementation of DAEMON TRIGGERS on SQL Server brought the same level of consistency but aborting less than 13% of the tasks. On the other hand, in a concurrent environment on Firebird, about 50% of the tasks were aborted under its strongest SNAPSHOT TABLE STABILITY isolation level, and yet the database was taken to an inconsistent state. With 128 concurrent transactions, the SNAPSHOT isolation aborted around 9% of the tasks and some inconsistencies were also generated. By using DAEMON TRIGGERS, this number increased to about 12% but the database remained consistent all the time. So, an implementation of DAEMON TRIGGERS in SQL Server can brought the same level of consistency as SERIALIZABLE but providing a higher degree of concurrency. In turn, DAEMON TRIGGERS in Firebird can even beat its strongest level of isolation twice, first by aborting much less transactions and second by ensuring only serializable histories.

The number of inconsistencies in the database when we use the READ COMMITTED isolation is smaller in SQL Server than it is in Firebird, but at the cost of a larger number of serialization errors in SQL Server. This is because SQL Server uses the Two-Phase Lock (2PL) protocol whereas Firebird does not lock the data items a transaction reads. On the other side, at least with regard to the SmallBank++ workload, the implementation of the SNAPSHOT protocol in SQL Server resulted in a smaller number of serialization errors and inconsistencies in the database.

V. RELATED WORK

In [4] a set of theoretical tools is presented in order to ensure serialization under the SNAPSHOT isolation. The technique is to initially make a static analysis of the transactions in an application and identify every possible interleaved execution that is not serializable. Fundamentally, the analysis is based on the possibility of cycle occurrences in a direct serialization graph [9] resulting from a potential history involving all transactional tasks. Although the technique does not require any change in the DBMS kernel, the possibility of occurrence of a dangerous structure must lead to a change in the application code and sometimes the database. Thus, consistency is achieved by moving part of the responsibility in dealing with competition issues from the DBMS to the application. Our technique requires the definition of DAEMON

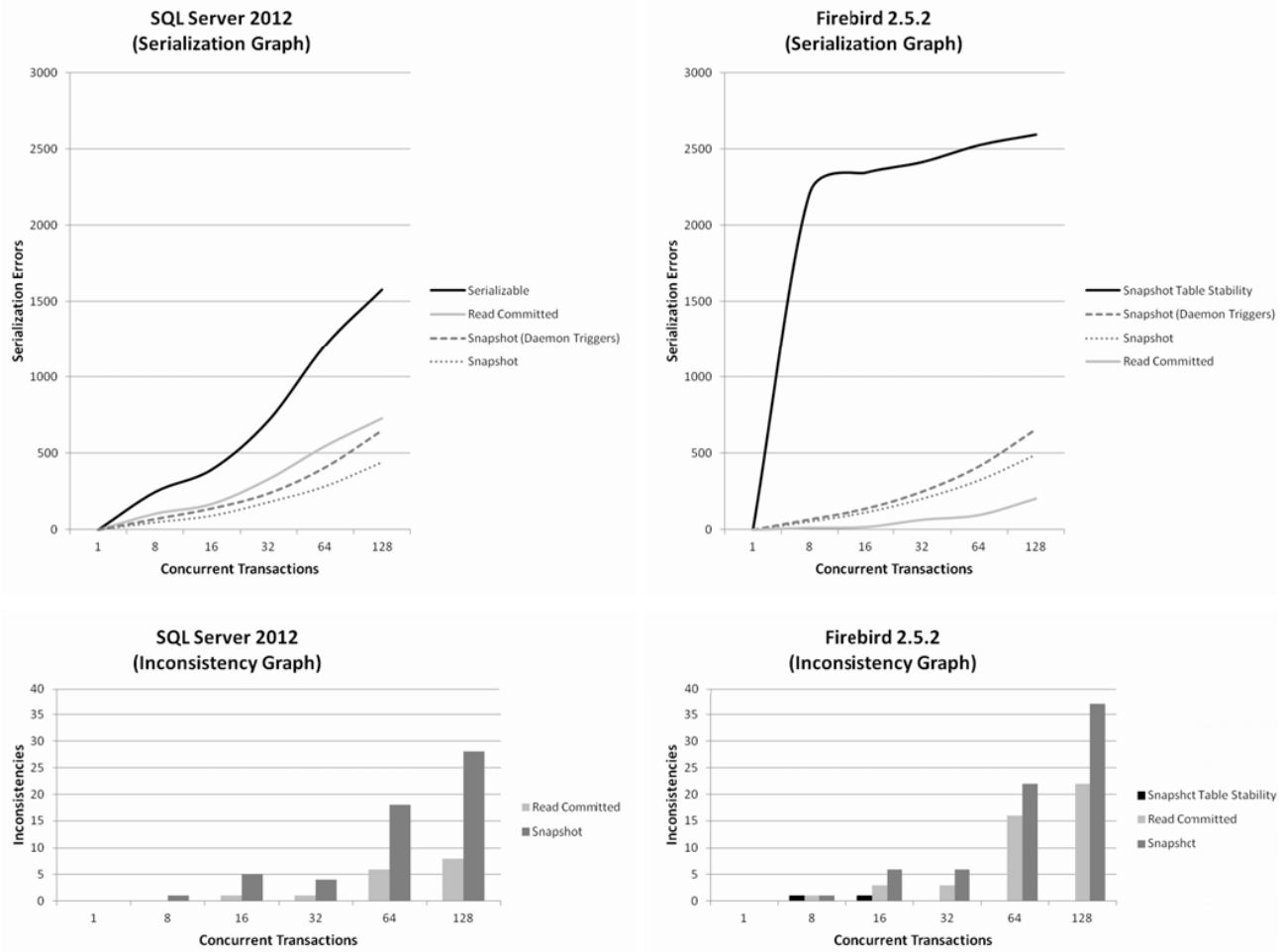

Fig. 8. Serialization and inconsistency graphs for SQL Server 2012 and Firebird 2.5.2

TRIGGERS based on business rules but leaves only the DBMS as responsible for concurrency control.

The technique proposed in [10] and [11] to ensure serialization is based on a lock mechanism outside a DBMS transactional system. This additional software component is named External Lock Manager and must also work together with the application. DAEMON TRIGGERS keep the benefits of the optimistic concurrency control system in SNAPSHOT by not using locks. Moreover, the level of concurrency is better considering that the writes of a transaction do not block the reads of others.

SERIALIZABLE SNAPSHOT is an isolation mechanism able to ensure serialization without any changes in application code or database [12]–[14]. It is based on the SNAPSHOT rules and the theory presented in [9] and [4]. Basically, the algorithm looks for successive anti-dependencies related to concurrent transactions, and when such pattern happens, one of the transactions is aborted. The drawback of the algorithm is that it needs to trace the data items a transaction reads and writes. With DAEMON TRIGGERS we can achieve serialization warranty the same way, but the management overhead occurs only on written data.

The algorithm proposed in [7] and [15], PRECISELY SERIALIZABLE SNAPSHOT, is also based in the theory presented in [4], but the approach is aimed at the detection of cycles in a dependency graph, where a transaction is aborted during a commit phase in case a cycle is introduced. In this case, the algorithm also needs to trace the data items that a transaction reads and writes.

According to [2], the *write-write* conflict detection mechanism used by SNAPSHOT, besides allowing some non-serializable histories, unnecessarily reduces concurrency precluding some serializable ones. Thus, the authors question the effectiveness of the approach behind SNAPSHOT highlighting that it is not sufficient, nor necessary, and proposing an approach based on a *read-write* conflict detection. Unlike the *write-write* approach, where only a transaction reading phase is immune to concurrency, the *read-write* approach also allows for the isolation of a transaction writing phase, and for this reason the technique is named WRITE-SNAPSHOT. But contrary to our approach, the WRITE-SNAPSHOT algorithm imposes an overhead on both read and written data items.

VI. Conclusion

Considering the simplicity of the presented technique, it is possible to build a new isolation level on top of the already existent SNAPSHOT isolation: DAEMON SNAPSHOT. Together with the new level, we need a statement to make non-structural changes in a database schema by defining special entities that establish relationships between those data items that are under some constraint. These entities, named *daemons*, would play the role of DAEMON TRIGGERS. Fundamentally, a transaction running under the DAEMON SNAPSHOT isolation level must behave like a transaction running under the SNAPSHOT isolation, except that *daemons* would be raised only for transactions under the DAEMON SNAPSHOT isolation. The syntax proposed for the creation of *daemons* follows:

```
CREATE DAEMON daemon
  ON table [(column[,...])]
  KEY (column[,...]) REFERENCES table
      (column[,...]) [WRITE]
  KEY ... [;]
```

When creating a *daemon* we must specify its name and the table it guards. The *daemon* will be raised for every row in the table that has one of the columns specified in the ON clause updated. In case no column is specified the *daemon* will be raised for every written row in the table. The first KEY clause defines the first data item to be reached, the possible second KEY clause defines the second data item to be reached, and so on. Ideally, the row updated by the application and the last row reached by the *daemon* in the chain of KEY clauses would be part of a business rule, so the identity write operation must be specified only for the last reached row in the chain through the WRITE clause. For the cases considered in this paper, we would have the following *daemons*:

Case 1

```
CREATE DAEMON Accounts_d
  ON Accounts (Balance)
  KEY (Customer) REFERENCES People (Id)
  KEY (Spouse) REFERENCES Accounts
      (Customer) WRITE;
```

Case 2

```
CREATE DAEMON Tasks_d
  ON Tasks (Hours)
  KEY (Employee) REFERENCES Employees
      (Id) WRITE;
```

Case 3 (Two *daemons*)

```
CREATE DAEMON SavingsAccounts_d
  ON SavingsAccounts (Balance)
  KEY (Customer) REFERENCES
      CheckingAccounts (Customer) WRITE;
CREATE DAEMON CheckingAccounts_d
  ON CheckingAccounts (Balance)
  KEY (Customer) REFERENCES
      SavingsAccounts (Customer) WRITE;
```

In this work, DAEMON TRIGGERS were applied to the management of conventional data but the concept of a DAEMON SNAPSHOT isolation can also be used in complex data management systems, like object-oriented databases or even cloud storage systems. Unlike the other isolation levels proposed in the literature on top of SNAPSHOT, the data management overhead imposed by our technique occurs only on the data items a transaction writes. Due to this, DAEMON SNAPSHOT becomes a potential candidate to be applied in distributed systems where certain transactional tasks require data consistency assurance.

Acknowledgment

The authors thank CAPES–PROPG for the support.

References

[1] Ansi, "Ansi x3.135-1992, american national standard for information systems — database language — sql," American National Standards Institute, Tech. Rep., 1992.

[2] M. Yabandeh and D. Gómez Ferro, "A critique of snapshot isolation," in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 155–168.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ansi sql isolation levels," *ACM SIGMOD Record*, vol. 24, no. 2, pp. 1–10, 1995.

[4] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha, "Making snapshot isolation serializable," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 2, pp. 492–528, 2005.

[5] A. Adya, B. Liskov, and P. O'Neil, "Generalized isolation level definitions," in *Data Engineering, 2000. Proceedings. 16th International Conference on*. IEEE, 2000, pp. 67–78.

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

[7] S. A. Revilak, "Precisely serializable snapshot isolation," Ph.D. dissertation, Office of Graduate Studies, University of Massachusetts Boston, 2011.

[8] M. Alomari, M. Cahill, A. Fekete, and U. Rohm, "The cost of serializability on platforms that use snapshot isolation," in *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*. IEEE, 2008, pp. 576–585.

[9] A. Adya, "Weak consistency: a generalized theory and optimistic implementations for distributed transactions," Ph.D. dissertation, Massachusetts Institute of Technology, 1999.

[10] M. Alomari, "Ensuring serializable executions with snapshot isolation dbms," Ph.D. dissertation, University of Sydney, 2008.

[11] M. Alomari, A. Fekete, and U. Rohm, "A robust technique to ensure serializable executions with snapshot isolation dbms," in *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*. IEEE, 2009, pp. 341–352.

[12] M. J. Cahill, U. Röhm, and A. D. Fekete, "Serializable isolation for snapshot databases," *ACM Trans. Database Syst.*, vol. 34, no. 4, pp. 20:1–20:42, Dec. 2009. [Online]. Available: http://doi.acm.org/10.1145/1620585.1620587

[13] M. J. Cahill, "Serializable isolation for snapshot databases," Ph.D. dissertation, The University of Sydney, 2009.

[14] M. J. Cahill, U. Röhm, and A. D. Fekete, "Serializable isolation for snapshot databases," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 729–738. [Online]. Available: http://doi.acm.org/10.1145/1376616.1376690

[15] S. Revilak, P. O'Neil, and E. O'Neil, "Precisely serializable snapshot isolation (pssi)," in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, 2011, pp. 482–493.