Declarative Data-Driven Coordination

Miloš Nikolić

DATA, I&C, EPFL

***Abstract*—The vision of declarative-data driven coordination (D3C) [3] enables users to communicate and coordinate through declarative specifications. A declarative query mechanism for D3C, named entangled queries, has been presented in [2]. Raising this notion to the level of transactions introduces many new challenges, such as a formal characterization of additional anomalies [6] or a choice of a concurrency control protocol [9]. Our work [1] defines semantic and execution model for entangled transactions and demonstrate their viability in real-world application settings.**

***Index Terms*—data-driven coordination, entangled query, entangled transaction, portable isolation levels, serializable snapshot isolation**

I. Introduction

WITH the expansion of social applications more and more users organize and coordinate their daily activities online. In many cases, users want their activities to be interdependent and performed in collaboration with other users. For example, friends want to make joint travel plans, students want to enroll in courses together and players want to make joint strategies in online social games. Despite the simplicity of these activities, most of them is surprisingly difficult to perform using today's technology. Modern database systems are designed to execute transactions in isolation from each other and provide no mechanism for their coordination. Data-driven coordination requires communication between user programs which clashes with the ACID properties of transaction.

The authors of [3] introduced the concept of declarative data-driven coordination (D3C) as an abstraction that extend the classical notion of the transaction to allow some form of information flow. The idea behind D3C is to provide a way for users to express their individual preferences and constraints, rather than deal with the complexity of the actual coordination. To raise coordination to the level of primitive with formal semantics, a declarative query mechanism for data-driven coordination, named *entangled queries*, has been proposed in [2]. Entangled queries are expressed in a SQL-like manner, allowing the coordination constraints and the data involved in coordination to be specified on the same level of abstraction. At runtime, the system performs the coordination and produces a result that respects constraints of all coordinating partners.

Still, most real-world data management applications that involve coordination require not just queries, but a transaction-like abstraction that covers larger units of work. Addressing this challenge involves a fundamental reassessment of the classical notion of isolation. For *entangled transactions*, isolation is clearly relaxed since it requires communication between user programs. However, the need to relax isolation is motivated by the novel semantics of D3C, not by performance considerations as with relaxations of classical isolation. Formalizing this intuition might have important implications on the overall system architecture of DBMS.

In [1] we have introduced a semantic model of entangled transactions that comes with analogues of the classical ACID properties. The model reveals several isolation anomalies unique to entanglement. As our future work, we plan to utilize the graph-based approach presented in [6] to formalize these anomalies and define corresponding isolation levels. Furthermore, our current lock-based prototype implementation lacks the power of efficiently handling interactive entangled transactions. We plan to explore alternate concurrency control protocols, based on the research [9], to enforce the correct and more flexible execution model.

The rest of the paper is structured as follows. In Section II, we introduce entangled queries. In Section III we describe the implementation-independent generalization of the existing ANSI isolation levels. Section IV deals with the problem of serializable execution under snapshot isolation. Finally, in Section V we briefly present our current work and future plans.

Proposal submitted to committee: September 7th, 2011; Candidacy exam date: September 13th, 2011; Candidacy exam committee: Exam president, thesis director, co-examiner.

This research plan has been approved:

Date: _____

Doctoral candidate: _____
(name and signature)

Thesis director: _____
(name and signature)

Thesis co-director: _____
(if applicable) (name and signature)

Doct. prog. director: _____
(R. Urbanke) (signature)

II. Entangled Queries

The paper [2] makes giant steps towards enabling the vision of declarative data-driven coordination (D3C) [3] where users are provided with novel abstractions that enable them to communicate and coordinate through declarative specifications. It introduces *entangled queries*, a mechanism that admits a limited form of interaction between database queries by automatically coordinating on the choice of common values between the queries.

As an example, suppose that Donald wants to travel to Paris on the same flight as his friend Minnie. Rather than communicating out-of-band, he can simple specify his coordination request in a SQL-like manner:

```
SELECT 'Donald', fno INTO ANSWER Booking
WHERE fno IN (SELECT flightno
              FROM Flight
              WHERE dest="Paris")
AND ('Minnie', fno) IN ANSWER Booking
CHOOSE 1
```

One can think of this entangled query as a request to learn the number of a single flight to Paris on which Minnie also plans to travel. Assuming Minnie also wants to coordinate with Donald, she issues a symmetric query with the strings 'Donald' and 'Minnie' exchanged, possibly additionally constrained with her own preferences. The system receives these queries for evaluation, recognizes that Donald and Minnie want to coordinate, chooses a flight and returns the same flight number to both friends.

A. Coordinated Query Answering

Although entangled queries are specified in an extension of SQL, an *intermediate representation* which uses Datalog-like notation is more suitable for algorithmic processing and formal reasoning. In this representation an entangled query is expressed using three parts: the body (B), the head (H) and the postcondition (C). The atoms in B may only refer to database relations, while atoms in H and C only refer to answer relations. The above entangled query has the following intermediate form (the relations `Booking` and `Flight` are abbreviated as `B` and `F`):

$$\{\ \mathtt{B}(\mathtt{Minnie}, x)\ \}\ \ \mathtt{B}(\mathtt{Donald}, x)\ :-\ \mathtt{F}(x, \mathtt{Paris})$$

Entangled queries use special constraints on the answer relation as a means of coordination. The idea is that the answer to the query is returned through an answer relation that is shared among multiple entangled queries in the system. An individual entangled query can only be answered if the answer relation satisfies query's postconditions. If this is the case, the result of query evaluation is a single row from the answer table. A query whose constraint is not satisfied upon individual evaluation is not rejected, but waits for an opportunity to satisfy the constraint, which may happen through the evaluation of another query.

The evaluation process performed by the system involves a set of entangled queries Q executed over a database instance D. The goal of the evaluation is to populate the answer relation in a way that respects all queries' coordination constraints.

Conceptually, the answering process begins by grounding all the queries from Q on D. The resulting grounding set G contains all possible valuations of the queries on D. Thus, the problem of finding the answers can be represented as a problem of finding a subset $G' \subseteq G$ such that G' contains at most one grounding of each query and the groundings in G' can all mutually satisfy each other's postconditions.

In general, the complexity of entangled query evaluation is NP-complete which follows from the complexity results for constraint satisfaction problem (CSP). The main source of complexity lies in discovering the *coordination structure*, i.e. the way the queries match up together. The other source of complexity is due to the fact that each query in Q has a body that is a conjunctive query, and it is known [14] that the combined complexity of evaluating conjunctive queries is NP-complete. However, this type of complexity cannot be eliminated and one usually considers this acceptable because of the small size of queries.

B. Making Coordination Tractable

The main contribution of this paper is introduction of safety and uniqueness properties of entangled queries - syntactic conditions which eliminate the main source of coordination complexity and ensure efficient evaluation in many real-world settings.

The proposed query matching mechanism employs the concept of *logical unifiability* between various head and post-condition atoms of the queries in Q. Unification of relational atoms defined over the same relation is possible if there is a substitution of variables that makes these atoms equal and no attribute value is assigned two different constants.

Unification dependencies among the queries in Q are captured using a data structure called the *unifiability graph*. The unifiability graph is a multi-digraph that contains a distinct node for every query in the system. An edge is drawn from q_i to q_j for each head atom of q_i that unifies with a postcondition atom of q_j. Existence of a path from q_i to q_j means that groundings of q_j require groundings of q_i for satisfaction, directly or transitively.

The *safety* property of a set of queries Q guarantees that within Q each query has a unique coordination partner. Formally, Q is unsafe if it contains a query q with a postcondition atom that is unifiable with more than one head atoms found in Q. However, enforcing only the safety condition on a set of queries Q does not guarantee existence of the unique coordination structure. Depending on the database, there could be several possibilities for coordination between queries from different subsets of Q. To guarantee *uniqueness of the coordination structure* (UCS) each query from Q must belong to a strongly connected subcomponent of the unifiability graph defined for Q. On this way, for each such a set of queries Q' we guarantee that there are no proper subsets of Q' that may be able to coordinate "locally" even if the entire set cannot.

When applied together these two conditions allow efficient detection of the coordination structures. Safety property guarantees that within each structure there is a unique way in which queries match. Therefore, it is possible to combine the queries

from a coordination structure into a bigger query that expresses the desired joint outcome.

C. The Evaluation Algorithm

The goal of the evaluation algorithm is to compute the unifiability graph for Q, partition this graph into strongly connected components, and for each component generate a single SQL query that computes the answers between the queries in this component. This approach is based on the fact that queries from two different components do not require coordination and, thus, each component can be handled independently and in parallel.

During the graph construction, the algorithm iteratively removes unanswerable queries, i.e. those that have no chance to participate in a coordinating set. Each node in the graph stores unification information about currently known constraints on valuations of variables that must hold for this query to be answerable. As part of this step, the algorithm iteratively propagates unification constraints to other nodes using the structure of the unifiability graph until a fixed point is reached. For a successful coordination, constraints of all involved queries must hold.

If the coordination structure is found, the evaluation algorithm creates a postcondition-free combined query taking into account bodies of all the original queries and unification constraints. Such a combined query is sent to the database for evaluation; the answers to the combined query are used to generate answers for individual queries.

D. Results and Discussion

The whole approach has been implemented as a middleware layer on top of a standard DBMS and evaluated through experiments that shown namely the scalability of the coordination algorithm. The experiments were performed using realistic workloads in an increasingly complex set of scenarios. The results have also shown that the proposed algorithm introduces a negligible overhead due to coordination evaluation. Furthermore, the algorithm is efficient in removing unanswerable and unsafe queries.

In the presented approach the choice of coordination partners is given by the unifiability between query postconditions and heads. In principle this allows users to coordinate with unknown coordination partners, but the safety and uniqueness properties can limit this coordination in many case by placing too strict requirements that might prevent a query from being answerable. This may limit the applicability of the approach in certain scenarios (e.g. MMO games).

III. Generalized Isolation Levels

Isolation levels are provided by all commercial database systems to allow programmers to trade off consistency for a potential gain in performance. The current ANSI SQL standard [4] defines isolation levels in terms of phenomena that a transaction is forbidden to experience at each isolation level. These phenomena (“Dirty Read“, “Non-repeatable Read“ and “Phantom“) have not been stated in terms of any particular concurrency control scheme. An important goal of the ANSI SQL standard was to provide flexible definitions that permit a variety of concurrency control mechanisms. The standard defines four successively more restrictive isolation levels: Read Uncommitted, Read Committed, Repeatable Read and Serializable.

A. Problem Statement

A subsequent paper [5] showed that the definitions provided in [4] are ambiguous and incorrect. Even their broadest interpretations do not exclude some anomalous behaviors. To tackle these problems the authors of [5] proposed refined definitions and introduced an additional phenomenon (”Dirty Writes”) to properly characterize the standard locking implementation of the isolation levels.

However, the proposed new specifications have been overly restrictive since they disallow all histories that would not occur in a lock-based implementation. The problem is that the notion of a conflict that underlies the locking rules does not carry over directly to other concurrency control protocols, such as multi-versioning or optimism. To this end, some commercial systems have introduced additional isolation levels that assume a multiversion concurrency control algorithm and specify a controlled relaxation from multiversion serializability (e.g. Snapshot Isolation, Read Consistency...).

Thus, we have a serious problem: on one side, we have a standard intended to be implementation independent, but without a precise definition, and on the other side, commercial database systems that “conform“ the standard, sometimes even not providing serializability guarantees at the highest isolation level. All this makes it more important to find a standard that is correct and yet flexible enough to allow a variety of concurrency control techniques.

B. Implementation-Independent Isolation Levels

The paper [6] presents new levels for transactional isolation that capture the intent of the specifications in [4], [5]. For each ANSI SQL level, a corresponding portable isolation level is provided that is precise, correct (disallows all anomalous behaviors) and implementation-independent. These new levels are called PL-1, PL-2, PL-2.99 and PL-3, where PL-3 guarantees conflict-serializability [8].

The new definitions are based on the principle that transactions are not allowed to observe violations of multi-object constraints; these are invariants of the type $x + y = 10$, that involve multiple objects. These constraints are captured by taking into account all objects that are accessed by the committing transaction. In contrast to this, the approach suggested in [5] avoids this difficulty by disallowing conflicting operations to run concurrently on individual objects, i.e. the conditions are specified in terms of single-object histories. As mentioned before, such approach is overly restrictive since it proscribes many valid histories produced by non-locking schemes that allow conflicting operations to execute simultaneously and still correctly preserve multi-object constraints.

The goal of creating portable isolation definitions is achieved using a combination of constraints on transaction

histories and serialization graphs [8]. These graphs provide a simple way of capturing multi-object constraints. Portable isolation levels are specified in terms of the different types of cycles that must be disallowed in these graphs, e.g. serializability disallows all types of cycles whereas the lowest isolation level disallows only cycles involving updates (and not reads).

Database Model. The database model is represented as a multi-version model [8] extended with the semantics of predicate-based operations. An interleaved execution of transactions is captured by a history along with the version order that specifies a total order on versions of each object created by committed transactions.

Predicates. The important contribution of [6] is the definition of predicate-based operations in a correct and implementation-independent manner. These definitions provide a variety of guarantees for predicate-based operations at all isolation levels. Earlier definitions for these operations were either incomplete, ambiguous, lock-centric or specified in terms of a particular database language such as SQL.

Predicate-based operations are executed against a set of versions of relevant tuple chosen by the system. Predicate-reads are modeled as (quasi) readings of all selected versions, followed by the decision which tuples match the predicate. If the system reads the matched versions as part of the query, these reads show up as separate events in the history. A predicate-based modification is modeled as a predicate-read followed by write operations on the matching tuples. A slightly different model that provides stronger guarantees (than the model described above) to predicate-based modifications at lower isolation levels is presented in [7].

Dependencies. There are three kinds of direct dependencies that capture direct interactions among committed transactions, i.e. conflicts of two transactions on the same object version. Note that since we assume the existence of a version order for objects, it is always clear what transactions overwrote versions of what other transactions.

1) *Read dependencies* capture write-read conflicts; T_j read-depends on T_i if it reads a version produced by T_i.
2) *Anti-dependencies* capture read-write conflicts; T_j anti-depends on T_i if it overwrites a version that T_i has read.
3) *Write-dependencies* capture write-write conflicts; T_j write-depends on T_i if it overwrites a version produced by T_i.

Predicate-based read/anti dependencies are established between a transaction that reads a predicate P and every other transaction that writes the next version of some object causing it to change the set of matching tuples for the predicate P.

Serialization Graphs. The direct serialization graph (DSG) captures all types of conflicts arising from a history and a given version order. Each node in this graph corresponds to a committed transaction and directed edges corresponds to different types of direct conflicts. As an example, consider the following history:

$$H: \ \mathrm{w_1(z_1)\ w_1(x_1)\ w_1(y_1)\ w_3(x_3)\ c_1\ r_2(x_1)\ w_2(y_2)}$$
$$\mathrm{c_2\ r_3(y_2)\ w_3(z_3)\ c_3} \qquad [\mathrm{x_1} \ll \mathrm{x_3},\ \mathrm{y_1} \ll \mathrm{y_2},\ \mathrm{z_1} \ll \mathrm{z_3}]$$

Figure 1 Shows the DSG for this history. As we can see, there is no cycle in the graph and the history is serializable.

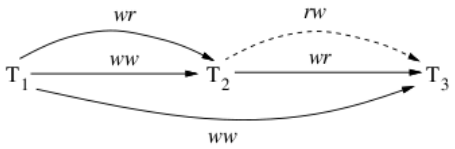

Fig. 1. DSG for history H

Portable Level PL-1. The weakest isolation level PL-1 guarantees that updates of conflicting transactions are not interleaved. Such a behavior is ensured by proscribing directed cycles from a DSG that consist entirely of write-dependency edges (*G0 phenomenon*). That prevents generation of unrecoverable schedules which are undesirable in some implementations. Another reason for proscribing this phenomenon is that it can violate database consistency as described in [5]. However, PL-1 provides weak guarantees for predicate-based modifications since they are modeled as queries followed by normal writes. Alternative approach with stronger guarantees is presented in [7].

Portable Level PL-2. This isolation level places constraints on read operations; it prevents circular information flow between transactions by proscribing directed cycles from a DSG that consist entirely of read/write (but not anti) dependency edges (*G1c phenomenon*). Furthermore, this level guarantees that a committed transaction has read only values that exists (or will exist) at some instant in the committed state. That does not mean that readings from uncommitted transactions are prohibited; PL-2 only prevents readings from aborted transactions (*G1a phenomenon*) and readings of non-final modifications of other transactions (*G1b phenomenon*). Such conditions cannot be defined in terms of serialization graphs; rather they are represented as constraints on transaction histories.

Portable Level PL-2.99. This isolation level guarantees full serializability with respect to data-item operations, but does not prevent the phantom reads. Such a behavior is achieved by proscribing directed cycles from a DSG that consist of read/write and item-anti dependency edges (*G2-item phenomenon*).

Portable Level PL-3. The strongest isolation level PL-3 prevents transactions from committing if they perform inconsistent reads or writes. In particular, it extends the guarantees of PL-2 by precluding all cycles that consist of one or more anti-dependency edges (*G2 phenomenon*).

C. Discussion

The definitions for PL-3 rule out all non-serializable histories and provide conflict-serializability even in the multi-version case. Furthermore, these new definitions are strictly less restrictive than the the lock-based interpretation of the existing ANSI standard, i.e. the new specification of isolation levels admits more good histories.

Another advantage of the presented graph-based approach is that it allows defining correctness conditions in a system in which different transactions may commit at different isolation levels. To that end, mixed serialization graphs are employed to capture only dependencies relevant to a transaction's level

or obligatory dependencies required by other transactions' modes.

An interesting property of the proposed isolation definitions is that they allow an application to request different isolation guarantees for committed and running transactions. This characteristic ensures that a wide range of concurrency control mechanisms are permitted by these isolation specifications. Correctness conditions for both types of transactions are presented in [7]. The approach in [5] allows only concurrency control schemes that provide the same guarantees for running and committed transactions (a lock-based implementation does indeed have this property).

In addition to supporting levels specified in [4], the proposed approach can also characterize commercial levels such as Cursor Stability, Snapshot Isolation and Oracle's Read Consistency, as well as additional levels that fall between PL-2 and PL-3 [7]. These levels are characterize by extending the graphs used for defining the ANSI levels; different types of nodes and edges are added to capture the constraints relevant to each level. These definitions demonstrate that the graph-based approach for specifying isolation levels is flexible.

IV. Serializable Snapshot Isolation

Snapshot isolation (SI) [5] is a concurrency control approach that uses multiple versions of data to provide non-blocking read operations. Under SI, a transaction T sees the database state as produced by all the transactions that committed before T starts, but no effects are seen from transactions that overlap with T. Reads are never delayed because of concurrent transactions' writes, nor do reads cause delays in a writing transaction. SI also enforces an additional restriction on execution in order to prevent Lost Update anomalies: it is not possible to have two transactions which both commit and both modify the same data item. This is called the "First-Committer-Wins" rule. In practice, implementations of SI usually prevent a transaction from modifying an item if a concurrent transaction has already modified it. Such approach is called "First-Updater-Wins".

A. Problem Statement

It has been known since SI was formalized in [5] that it allows non-serializable executions. In particular, it is possible for an SI-based concurrency control to interleave some transactions, where each transaction preserves an integrity constraint when run alone, but where the final state after the interleaved execution does not satisfy the constraint. Here is an execution that can occur under SI:

$$H: \quad \mathrm{r}_1(\mathrm{x}_0, 50)\ \mathrm{r}_1(\mathrm{y}_0, 50)\ \mathrm{r}_2(\mathrm{x}_0, 50)\ \mathrm{r}_2(\mathrm{y}_0, 50)$$
$$\mathrm{w}_1(\mathrm{x}_1, -20)\ \mathrm{w}_2(\mathrm{y}_2, -30)\ \mathrm{c}_1\ \mathrm{c}_2$$

This sequence of operations represents an interleaving of two transactions, T_1 and T_2, withdrawing money from bank accounts x and y, respectively. Each of the transactions begins when the accounts each contain \$50, and each transaction in isolation maintains the constraint that $x + y > 0$. However, the interleaving results in $x + y = -50$, so consistency has been violated. This type of anomaly is called a *write skew*. A predicate-based version of this anomaly can also generate non-serializable executions under SI as noted in [5]. Another type of anomaly that involves read-only transactions is presented in [9].

Despite these known anomalies, SI is supported in many commercial systems. In fact, it is the highest available level of consistency in widely used systems such as Oracle and PostgreSQL. Many organizations use these databases for running their applications, and so they are potentially at risk of corrupted data.

From the aspect of the DBA or application developer, the real concern is whether a particular database application, consisting of several interacting programs acting under SI, will produce only serializable executions. A theoretical foundation that allows application developers to tackle this problem has been established in [9]. The paper also provides some guidance how to modify the application to guarantee serializable execution.

B. SI Transaction Theory

The starting point for understanding how transactions can produce anomalies under SI is the theory of multiversion serializability, e.g. the theory based on directed serialization graphs (DSG) presented in Section III. Under snapshot isolation, the definitions of serialization graphs become much simpler, as versions of an item are ordered according to the temporal sequence of the transactions that created those versions. Figure 2(a) shows the DSG for the history with write skew, discussed above. As usual in transaction history, the absence of a cycle in DSG proves that the history is serializable. Thus it becomes important to understand what sorts of DSG can occur in histories of a system using SI for concurrency control.

Fig. 2. SDG and DSG for write skew example

It is important to note that read and write dependencies can not occur between two concurrent snapshot-isolated transactions. Thus a cycle in the DSG cannot occur in a history having only transactions with read and write dependencies. It was showed in [7] that any cycle produced by SI has two anti-dependency edges. This was extended by [9], which showed that any cycle must have two anti-dependency edges that occur consecutively, and further, each of these edges is between two concurrent transactions. This important theoretical result has formed a foundation for characterization of how SI serialization anomalies can arise and how they can be prevented.

C. Static Dependency Graph

In light of this result, the paper [9] proposes a graph-based approach for static analysis of the possible conflicts

between transactional programs. The goal of this approach is to determine whether or not a non-serializable execution of an application can occur under SI. To that end, a static dependency graph (SDG) is defined for a given collection of transactional programs. A static dependency edge is drawn from P_1 to P_2 if there is some execution of the system, in which T_1 is a transaction that arises from running program P_1, and T_2 arises from running P_2, and there is a dependency from T_1 to T_2. Of special importance are edges called vulnerable edges. An edge from P_1 to P_2 is vulnerable if there is some execution of the system, in which T_1 is a transaction that arises from running program P_1, and T_2 arises from running P_2, and there is a read-write dependency from T_1 to T_2, and T_1 and T_2 are concurrent. Figure 2 shows DSG(H) and SDG(A) for the history with an application A that can exhibit the write skew anomaly.

Within the SDG, certain patterns of edges are crucial in determining whether or not anomalies might occur. The paper defines that the graph SDG(A) has a dangerous structure if it contains nodes P, Q and R, which may not all be distinct, such that there is a vulnerable edge from R to P, there is a vulnerable edge from P to Q, and there is a path from Q to R (or else $Q = R$).

The main theorem of [9] shows that if a collection of programs A has SDG(A) without dangerous structure, then every execution of the programs in A under SI is serializable. This theory is applied in a manual static analysis to prove that the TPC-C [13] benchmark application has no serialization anomalies under SI.

D. Avoiding Anomalies

In cases when analysis of a SDG reveals presence of dangerous structures, the general approach is to modify one or both application programs, in ways that do not alter their business logic, but avoids dangerous structures. Programmers can explicitly introduce extra write-write conflicts in transactional programs, in order to prevent the transaction from running concurrently with the other transaction on the (formerly) vulnerable edge. Two different types of modifications are proposed by [9]:

1) **Materialization of the conflicts**. Typically, one introduces a new table, and both transactions are made to write the same row of this table. This method works particularly well for predicate conflicts.
2) **Promotion**. The program reading the data item in conflict is changed to also perform an identity write on the data item. We say that the read is promoted to a write.

E. Discussion and Related Work

Making SI serializable using static analysis has a number of limitations. Doing a design-time analysis of the application code requires highly skilled DBAs able to determine the possibility of dependencies between every pair of transaction programs, by looking at the program logic. In general, the program logic could be arbitrarily complicated, and this process might not be always feasible. In addition, the static analysis must be a continual activity as an application evolves. Every minor change in the application requires renewed analysis, and perhaps additional changes (even in programs that were not altered).

However, the theoretical foundation presented in [9] gave rise for future researches on how to ensure serializable executions when running under SI. The follow up paper [10] describes a system to automate the analysis of program conflicts. One important finding of that work is that snapshot isolation anomalies do exist in applications developed using tools and techniques that are common throughout the software industry. A more recent paper [11] provides Serializable Snapshot Isolation (SSI), which avoids such anomalies at runtime without any need to pre-examine the code. However, this runtime technique is conservative and relies on aborting transactions that give rise to dangerous structures causing unnecessary aborts. The most recent paper [12] demonstrates a new method for implementing serializable snapshot isolation based on precise criterion for cycle detection and abortion of transactions.

V. Current and Future Work

Designing a system that supports entangled transactions reveals many research challenges. The semantics of classical transactions is closely tied to the ACID properties; it is appropriate to understand what analogues of these can be expected to hold for entangled transactions. One is sure, isolation is clearly relaxed since it requires communication between user programs. Our need to relax isolation is motivated by the novel semantics of entangled transactions, not by performance considerations as with relaxations of classical isolation [6], [5]. Therefore, it appears that isolation should be relaxed only "as far as necessary" to permit controlled communication through entangled queries. We have detected several isolation anomalies unique to entanglement. Our future plan is to further formalize the entangled isolation and introduce new definitions of isolation levels using the approach defined in [6].

In [1] we have described a semantic model that captures both the fact that each entangled transaction represents a logical unit of work on its own, and that this work is dependent on input from other transactions in the system. We have also presented our prototype that implements entangled transaction support in the middle tier, and as such can be used with any existing DBMS. Experiments with our prototype show that the overheads associated with supporting entangled transactions are acceptable for real-world use.

However, our prototype supports only the non-interactive transactional model in which users can be expected to issue entire entangled transactions at once. Interactive model with users in the loop comes with its own challenges and cannot be achieved using our current lock-based implementation. Thus, our future plan is to further investigate the unique issues associated with interactivity and potentials of other concurrency control techniques, including snapshot isolation presented in this paper.

REFERENCES

[1] N. Gupta, M. Nikolic, S. Roy, G. Bender, L. Kot, J. Gehrke and C. Koch, *Entangled transactions*, VLDB, 2011.

[2] N. Gupta, L. Kot, S. Roy, G. Bender, J. Gehrke, and C. Koch, *Entangled queries: enabling declarative data-driven coordination*, In Proc. ACM SIGMOD Conf, 2011.

[3] L. Kot, N. Gupta, S. Roy, J. Gehrke and C. Koch, *Beyond Isolation: Research Opportunities in Declarative Data-Driven Coordination*, SIGMOD Record, 39(1):27–32, 2010.

[4] V. Atluri, E. Bertino and S. Jajodia, *ANSI X3. 135-1992, American Nalional Standard for Information Systems – Database Language – SQL*, George Mason University, 1992.

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

[6] A. Adya, B. Liskov and P. O'Neil, *Generalized Isolation Level Definitions*, In ICDE, 2000.

[7] A. Adya, *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*, PhD Thesis, 1999.

[8] G. Weikum and G. Vossen, *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*, Morgan Kaufmann Publishers Inc., 2001

[9] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil and D. Shasha, *Making snapshot isolation serializable*, ACM Trans. Database Syst., 2005

[10] S. Jorwekar, A. Fekete, K. Ramamritham and S. Sudarshan, *Automating the detection of snapshot isolation anomalies*, Proc. VLDB, 2007

[11] M. Cahill, U. Röhm and A. Fekete, *Serializable isolation for snapshot databases*, Proc. ACM SIGMOD, 2008

[12] S. Revilak, P. O'Neil and E. O'Neil, *Precisely Serializable Snapshot Isolation (PSSI)*, Data Engineering (ICDE), 2011

[13] TPC-C Benchmark Specification, available at http://www.tpc.org/tpcc/.

[14] S. Abiteboul, R. Hull and V. Vianu *Foundations of Databases*, Addison-Wesley, 2005