Entangled transactions

Nitin Gupta[1], Milos Nikolic[2], Sudip Roy[1], Gabriel Bender[1],
Lucja Kot[1], Johannes Gehrke[1,3], Christoph Koch[2]

[1]Cornell University
Ithaca, NY 14853, USA

[2]EPFL
CH-1015 Lausanne,
Switzerland

[3]MPI-SWS
D-66123 Saarbruecken,
Germany

niting@cs.cornell.edu, milos.nikolic@epfl.ch,
{sudip, gbender, lucja, johannes}@cs.cornell.edu, christoph.koch@epfl.ch

ABSTRACT

As the world becomes more interdependent and computing grows more collaborative, there is a need for new abstractions and tools to help users work together. We recently introduced *entangled queries* – a mechanism for information exchange between database queries [6]. In this paper, we introduce *entangled transactions*, units of work similar to traditional transactions that however do not run in isolation, but communicate with each other via entangled queries.

Supporting entangled transactions brings about many new challenges, from an abstract model to an investigation of the unique systems issues that arise during their implementation. We first introduce a novel semantic model for entangled transactions that comes with analogues of the classical ACID properties. We then discuss execution models for entangled transactions and select a concrete design motivated by application scenarios. With a prototype system that implements this design, we show experimental results that demonstrate the viability of entangled transactions in real-world application settings.

1. INTRODUCTION

Empty-handed I entered the world
Barefoot I leave it.
My coming, my going –
Two simple happenings
That got entangled. — Kozan Ichikyo.

In twentieth century data processing practice, programs and processes were largely solitary entities. Each operated individually to achieve a given task. Physical systems needed to handle multiple simultaneous processes, so the research community developed protection mechanisms to prevent interference. In the database community, this work culminated in the concept of a transaction. Such a classical transaction represents a discrete unit of data processing work as reflected in the ACID properties of atomicity, consistency, isolation, and durability: it provides the conceptual properties of being executed completely or not at all, of preserving database consistency as it runs, of running without interference from other transactions, and if committing, making its changes persistent.

Proceedings of the VLDB Endowment, Vol. 4, No. 7

However, in recent years data processing programs have become interdependent by design. For example, web services frequently interact and coordinate to carry out tasks spanning enterprises [4]. Coordination is now needed in domains ranging from course enrollment [8] and travel planning [6] to online social games such as Farmville, where gameplay is fundamentally collaborative. The coordination strategies used in these games are similar to those found in more "serious" application domains such as managing charity donations with gift matching [3] and auctions [10]. With Farmville now attracting over fifteen million users each day, data-driven coordination has become big business, and it is here to stay.

We recently started to take first steps towards the problem of supporting data-driven coordination [7, 6]. We introduced *entangled queries*, a mechanism that admits a limited form of interaction between database queries by automatically coordinating — not on events or conditions, but on the choice of common values between the queries. However, most real-world data management applications that involve coordination require not just queries, but a transaction-like abstraction that covers larger units of work. As an example, assume that two friends, Mickey and Minnie, wish to travel to Los Angeles on the same flight and stay at the same hotel. Their arrival date is flexible, but their departure date is fixed. They start by jointly selecting a suitable flight. Once they know the flight number, and consequently their date of arrival in Los Angeles, they will try to make joint hotel reservations. With existing mechanisms, they can use entangled queries to coordinate on the choice of the flight and then on their choice of hotel. These queries, however, need to be embedded within a larger code unit that Mickey and Minnie separately execute and populate with their constraints such as the class of the hotel or airline restrictions. Once both their individual *entangled transactions* have been submitted, the system needs to match them up, execute the associated logic, and guarantee "transaction-like" semantics for this execution.

Research Challenges. What are these entangled transactions? How do they relate to entangled queries and to classical transactions? First, in order to define what we even mean by entangled transactions we need a clean semantic model which must capture 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. Furthermore, the input from other transactions is not arbitrary; it is restricted to what can be achieved with entangled queries. This means entangled trans-

actions have different semantics than nested transactions [9] and Sagas [5], where arbitrary communication is permitted between the components of a single unit of execution, or cooperative transaction groups [11], where such communication is regulated through complex custom policies. They are also different from split-transactions [12] as the components are defined statically and matched into a larger execution unit at runtime, and not the other way around.

The entangled transaction model must extend to transactions that contain more than one entangled query. Indeed, Mickey and Minnie's travel planning example requires entangled transactions with several entangled queries: the number of nights for the hotel reservation depends on the arrival date, which is not known until they have chosen a flight. This means Mickey and Minnie need to use separate entangled queries to coordinate on the flight and the hotel.

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. For entangled transactions, isolation is clearly relaxed, but we also do not want to throw out the baby with the bathwater – that is, completely give up on the advantages and convenience of isolation between transactions. Our need to relax isolation is motivated by the novel semantics of entangled transactions, not by performance considerations as with relaxations of classical isolation [2]. Therefore, it appears isolation should be relaxed only "as far as necessary" to permit controlled communication through entangled queries.

Formalizing the above intuition is an interesting problem in its own right, but it is not sufficient for a full treatment of entangled isolation. It is also necessary to deal with the fact that when entangled transactions run, they see more of the system's state than classical transactions do. A transaction that receives an answer to an entangled query becomes aware of the existence of another entangled transaction in the system. Since the ultimate goal of isolation is to ensure that each transaction sees a consistent system state during execution, entangled isolation requires a consistent view of both the database and the concurrent processes.

Defining consistency preservation for entangled transactions is nontrivial. Intuitively, Mickey and Minnie's transactions still appear to be coherent units of work; neither one of them should individually introduce inconsistencies in the database if implemented and executed correctly. However, neither can execute by itself, so formalizing this intuition is not straightforward.

Even once a semantic model of entangled transactions is in place and we understand how the ACID properties extend to them, the details of a full *execution* model are far from obvious. Returning to Mickey and Minnie, suppose Minnie's transaction aborts after the two friends have chosen and booked a flight; the corrective action to be taken is not immediately clear. Also, it is likely that the two transactions may not arrive in the system simultaneously; if one of them has to wait for the other, it is important to ensure usability of the system by other transactions in the interim. Designing an execution model to handle issues like the above in a principled way raises many research questions.

Last but not least, entangled queries are not useful until they are supported in a real system that can be deployed in practice. Designing the architecture of such a system and combining it with existing DBMS functionality presents deep systems challenges.

Our contributions. In this paper, we lead the reader into the new world of entanglement. First, we review our building block of entangled queries (Section 2). We then introduce a model of entangled transactions that comes with analogues of the classical ACID properties. Our model permits trading off isolation to achieve greater concurrency, albeit at the cost of some loss of consistency, resulting in the definition of isolation levels for entangled transactions (Section 3). Second, we discuss execution models for entangled transactions. We outline the major design issues involved and present a specific model that we found especially suitable for our motivating application scenarios (Section 4). Third, we outline the challenges that arise when implementing a system supporting entangled transactions. We present the architecture of our prototype implementation of entangled transactions within the Youtopia system (Section 5). The prototype is implemented at 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.

2. ENTANGLED QUERIES

Entangled queries are expressed in extended SQL as follows:

```
SELECT select_expr
INTO ANSWER tbl_name [, ANSWER tbl_name] ...
[WHERE where_answer_condition]
CHOOSE 1
```

The `WHERE` clause is a standard condition clause that may refer to both database and `ANSWER` relations. The `ANSWER` relations are not database tables; they serve only as names that are shared among queries and permit entanglement. As in our previous work [6], the `WHERE`-clause is restricted to contain only select-project-join queries.

To continue with our example from the introduction, suppose Mickey wants to travel to Los Angeles on the same flight as Minnie. He can express this with the entangled query below.

```
SELECT 'Mickey', fno, fdate INTO ANSWER Reservation
WHERE fno, fdate IN
    (SELECT fno, fdate FROM Flights
     WHERE dest='LA')
AND ('Minnie', fno, fdate) IN ANSWER Reservation
CHOOSE 1
```

The name `Reservation` refers to a conceptual relation which collects the answers to all the queries relating to flight bookings. The `SELECT` clause specifies Mickey's own expected answer, or, in other words, his contribution to the answer relation `Reservation`. This is a tuple containing the constant `Mickey`, the flight number and the date of the booking. The existence of Mickey's answer, however, is conditional on two requirements, which are given in the `WHERE` clause. First, the flight's destination must be Los Angeles. Second, the answer relation must also contain a tuple with the same flight number and date but `Minnie` as the passenger name. The `CHOOSE 1` at the end of the query specifies that the system should choose only one flight, even if more than one might be suitable.

Now suppose Minnie actually wants to fly with Mickey, but she wants to fly only on United. Her query is as follows:

```
SELECT 'Minnie', fno, fdate INTO ANSWER Reservation
WHERE fno, fdate IN
    (SELECT fno, fdate
     FROM Flights F, Airlines A WHERE
          F.dest='LA' and F.fno = A.fno
          AND A.airline = 'United' )
AND ('Mickey', fno, fdate) IN ANSWER Reservation
CHOOSE 1
```

When the system receives the two queries, it answers both of them simultaneously in a way that ensures a coordinated choice of flight. If the database is as shown in Figure 1 (a), the system

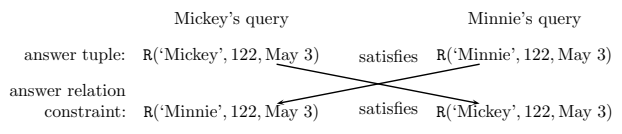
Flights

fno	fdate	dest
122	May 3	LA
123	May 4	LA
124	May 3	LA
235	May 5	Paris

Airlines

fno	airline
122	United
123	United
124	USAir
235	Delta

(a)



(b)

Figure 1: (a) Flight database (b) Mutual constraint satisfaction

nondeterministically chooses either flight 122 or 123 and returns appropriate answer tuples. Figure 1 (b) shows the mutual constraint satisfaction that takes place in answering for 122; the relation name `Reservation` is abbreviated as `R`. Neither Mickey nor Minnie sees the other's answer, but each of them is guaranteed that all answer constraints have been met.

After Mickey and Minnie receive answers to their queries, each of them can book a seat on the flight and date specified. The above queries are simplified; in practice, they would perform more work such as verification of seat availability.

Section A in the Appendix gives an overview of the semantics of entangled queries; for more details, see our previous work [6]. The semantics makes use of the notion of a *grounding* for each entangled query. To compute a grounding essentially means to evaluate of the portion of the `WHERE` clause which does not refer to an `ANSWER` relation. This identifies the set of acceptable answers for each individual query; for Minnie's query, for example, it would identify that only answers involving flights 122 or 123 are suitable. Answering a *set* of entangled queries involves choosing an acceptable answer for each individual query such that the corresponding individual answers all satisfy the appropriate constraints.

3. ENTANGLED TRANSACTIONS

In this section, we introduce our model for entangled transactions and discuss the new meaning of the ACID properties in the presence of entanglement.

3.1 Syntax and Semantics

Entangled transactions have the following syntax.

```
BEGIN TRANSACTION [WITH TIMEOUT duration]
[SQL standard syntax | entangled_query | ROLLBACK]*
entangled_query
[SQL standard syntax | entangled_query | ROLLBACK]*
COMMIT
```

Figure 2 shows a transaction that Mickey might run to coordinate with Minnie on a flight and a hotel in Los Angeles, as discussed in the introduction. The table `Hotels` contains information about hotels, including a hotel id (`hid`) and location attribute. `FlightRes` and `HotelRes` are the answer relations for flight and hotel booking coordination, respectively. `HotelRes` has attributes for customer name, hotel id, arrival date, and number of nights.

An entangled transaction is specified by the code enclosed within `BEGIN TRANSACTION` and `COMMIT`. In addition to the functionality offered by an ordinary transaction, an entangled transaction also contains one or more entangled queries. Calls to evaluate an entangled query are blocking: the transaction does not proceed until the entangled query receives an answer. The programmer may directly

```
BEGIN TRANSACTION WITH TIMEOUT 2 DAYS;

SELECT 'Mickey', fno, fdate AS @ArrivalDay
INTO ANSWER FlightRes
WHERE fno, date IN
    (SELECT fno, fdate FROM Flights
     WHERE dest='LA')
AND ('Minnie', fno, fdate) IN ANSWER FlightRes
CHOOSE 1;

-- (Code to perform flight booking omitted)

SET @StayLength = '2011-05-06' - @ArrivalDay;

SELECT 'Mickey', hid, @ArrivalDay, @StayLength
INTO ANSWER HotelRes
WHERE hid IN
    (SELECT hid FROM Hotels
     WHERE location='LA')
AND ('Minnie', hid, @ArrivalDay, @StayLength) IN
ANSWER HotelRes
CHOOSE 1;

-- (Code to perform hotel booking omitted)

COMMIT;
```

Figure 2: Example Entangled Transaction

bind the values returned by an entangled query to host variables by specifying `AS @varname` next to the appropriate value in the query; this can be seen in the example above with `@ArrivalDay`.

Because of the blocking calls to evaluate entangled queries, we associate a `timeout` parameter with each entangled transaction. This parameter limits the maximum time that this transaction may "wait" in the system for its entanglement partner(s). If a particular entangled query within the transaction is unable to succeed before the timeout expires, then the entire transaction is unable to complete. An error is thrown and must be handled by the application code. Entangled query failure is a relatively complex phenomenon that can happen for several reasons, not just the absence of a partner. Section B in the Appendix contains a more in-depth discussion of this issue and how it impacts transaction execution.

From a programmer's perspective, entangled queries have an additional advantage beyond allowing information exchange: They provide explicit synchronization points between transactions. This can be useful if the programmer knows the code of other transactions in the system. Once an entangled query is answered, a transaction can assume that all entanglement partners have executed the code preceding their corresponding entangled queries. For instance, if Minnie manages to coordinate with Mickey's transaction on a hotel, she knows that he has already booked his flight.

3.2 Consistency

We now present extensions of the ACID properties to entangled transactions. Consistency and isolation are particularly affected by entanglement, so we start by treating each of them in turn.

Classically, consistency is an abstract property of databases which transactions preserve by the following assumption:

ASSUMPTION 3.1 (CONSISTENCY). *Every transaction, if executed by itself on an initially consistent database, will produce another consistent database.*

The motivation behind this assumption is that an individual transaction is a logical and self-contained unit of work. A correct im-

plementation of such a transaction will never deliberately create data inconsistencies, except perhaps temporarily in the middle of its execution. The only time consistency of the final database is not guaranteed is if the initial database was inconsistent as well.

Assumption 3.1 is used to infer global consistency guarantees for the execution of a *set* of transactions. Suppose the permissible concurrent executions are constrained in such a way that every individual transaction sees (i.e. reads) a consistent version of the database as it runs. Then, the above assumption allows us to infer that any set of concurrent transactions, run on an initially consistent database, will produce another consistent database.

To formulate an analogous guarantee for entangled transactions, we need an equivalent of Assumption 3.1. The key is deciding what constitutes a logical and self-contained unit of work; this is non-trivial for an entangled transaction as it cannot execute by itself.

Three candidates for units of work are individual entangled transactions, *groups* of transactions that entangle during execution, and non-entangled *portions* of individual entangled transactions. With respect to the example shown in Figure 2, the first option would correspond to Mickey's transaction, the second to Mickey *and* Minnie's transactions, and the third to the two non-entangled code segments that are executed between entangled queries. The first option maintains the closest correspondence between entangled transactions and classical transactions; it is the one we use in this paper, and we leave the others as future work.

It is not obvious what it means for an individual entangled transaction to constitute a unit of work, given that a transaction like the one in Figure 2 is unable to run and complete by itself. However, intuitively, the only information this transaction needs "from the outside" is answers to the two entangled queries so that it knows which flight and which hotel to book. As long as Mickey's transaction is executed in the system alongside *some* process that provides this information, it will be able to complete correctly. This other process could be Minnie's transaction, but it could also in principle be a "query answering oracle" whose only functionality is to create Mickey's answer tuples.

We formalize the notion of an entangled query oracle as follows.

DEFINITION 3.2 (ENTANGLED QUERY ORACLE). *An entangled query oracle O is a process that executes alongside an entangled transaction t. Whenever t poses an entangled query, the oracle generates an answer and returns it to t. The oracle has no direct effect on the database's state, i.e. it performs no writes.*

The above definition deliberately does not constrain the kinds of answers that the oracle may supply to t. However, these answers should clearly simulate those received in true entanglement.

DEFINITION 3.3 (VALID ORACLE ANSWER). *Suppose a transaction t executes with an oracle O and poses an entangled query q at a time when the state of the database is D. An answer to q returned by the oracle is valid if it directly corresponds to a grounding of q on D.*

DEFINITION 3.4 (VALID ORACLE EXECUTION). *Suppose a transaction executes alongside an oracle O. If the oracle returns a valid answer to each entangled query, the entire execution is* valid.

This allows us to formulate the following consistency assumption for entangled transactions.

ASSUMPTION 3.5 (ORACLE CONSISTENCY). *Suppose an entangled transaction executes by itself on an initially consistent database, using an entangled query oracle to obtain answers to the entangled queries it poses, and suppose the execution is valid. Then the execution will produce another consistent database.*

This assumption is close in spirit to Assumption 3.1. It states that an entangled transaction will produce consistent "output" – i.e. a set of database writes that together do not violate consistency – as long as it is presented with consistent "input" – i.e. a consistent view of the data and valid answers to its entangled queries. This assumption holds for Mickey's transaction and is likely to hold for typical transactions in most realistic settings.

As with classical transactions, Assumption 3.5 can be used to infer a consistency guarantee for the execution of a set of entangled transactions. To this end, we again need to constrain the permissible concurrent executions so that each transaction is guaranteed to receive consistent "input". That is, we need to define isolation for entangled transactions, and it is to this issue that we turn next.

3.3 Isolation

Classical isolation is motivated by the need to provide each transaction with a consistent view of the database as it runs. As discussed, this together with Assumption 3.1 guarantees that the final database produced by a set of concurrent transactions is consistent.

An elegant way to define classical isolation is in terms of *serializability*, i.e. equivalence of an execution schedule to a serial execution schedule with the same transactions. In a serial schedule, Assumption 3.1 guarantees that each transaction does indeed see a consistent view of the database, so serial execution is a suitable gold standard for consistency. Equivalently, classical isolation can be defined as the avoidance of certain execution anomalies such as dirty reads and unrepeatable reads [2].

For entangled transactions, serializability is not directly applicable. However, we can use our entangled query oracles to define *oracle-serializability*, that is, equivalence to a schedule where the entangled transactions execute serially alongside an oracle. We can also formulate an anomaly based definition of isolation based on the classical anomalies and some new entanglement-related ones.

We have developed both an anomaly based and an oracle-serializability based definition of entangled isolation and proved a theorem that relates them to each other. Due to space constraints, we give only a high-level overview of each of these contributions here; the full formal treatment is found in the Appendix, Section C.

3.3.1 Anomaly-based definition

We first outline our anomaly-based definition of entangled isolation. Anomalies are pathological phenomena where a transaction observes an inconsistent view of the system state as it runs. If no such anomalies occur during execution, then by Assumption 3.5, we know that the final database produced after the execution of a set of entangled transactions is consistent. We begin by introducing the anomalies which are unique to entangled transactions. Classical anomalies such as dirty reads and unrepeatable reads can also happen in the entangled setting and must be avoided.

Widowed transactions To execute correctly and preserve consistency, entangled transactions need more than just a consistent view of the database. They need a consistent view of the entire system state, insofar as they have access to it. For classical transactions, the only accessible system state is the database. However, an entangled transaction t that has received an answer to an entangled query knows something else: it knows that there is another process running alongside it. If this other process subsequently aborts, t's view of the system may become inconsistent.

Consider for example the scenario in Figure 3 a), where Mickey runs the transaction in Figure 2 and Minnie runs a symmetric transaction containing the query from Section 2. The transactions successfully entangle on both the flight and hotel queries. However, while performing the hotel booking, Minnie's transaction aborts.

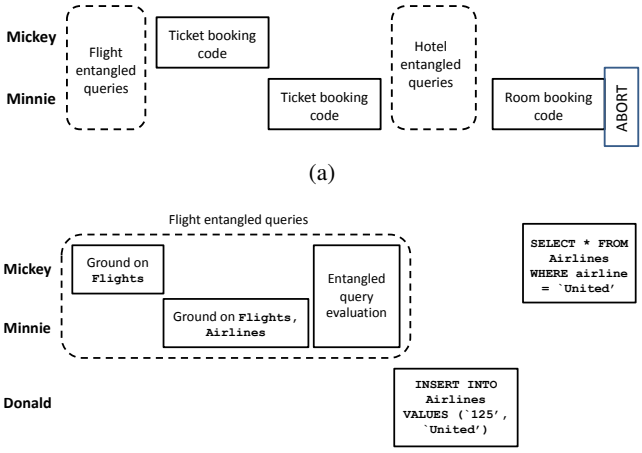


Figure 3: (a) Widowed transaction (b) Unrepeatable quasi-read

Mickey's transaction may still in principle be able to complete its hotel booking; however, Mickey would be booking a room based on the assumption that Minnie is traveling with him, and this may no longer be true. Mickey's transaction is *widowed* due to the abort of its entanglement partner. The **widowed transaction anomaly** is our first isolation anomaly which is unique to the entangled setting.

The additional system state visible to entangled transactions can be made explicit using the `ANSWER` relations. For simplicity, assume there is just one such relation. The transactions perform operations on this relation during entangled query answering. The answering process starts with a set of simultaneous *writes* by all transactions to the `ANSWER` relation – each transaction writes its corresponding answer tuple. Next, each transaction receives a guarantee that its partner's answer tuple is in the `ANSWER` relation – that is, the transaction performs an implicit *read* on the `ANSWER` relation. The reads are again performed simultaneously by all transactions.

The above discussion makes it clear why widowed transactions are a problem: two transactions that entangle perform a dirty read of each other's writes to the `ANSWER` relation. If one later aborts, the other's view of the `ANSWER` relation becomes inconsistent.

Unrepeatable quasi-reads The second new class of isolation anomaly is associated with the entangled query answering process itself, or more precisely with the computation of *groundings* for the queries. As explained in Section 2, the evaluation of a set of entangled queries conceptually begins by *grounding* each query. The actual evaluation algorithm does not need to operate in this way, but groundings are a useful tool for analysis. A grounding is a read on the database that corresponds to the portion of the `WHERE` clause of an entangled query that does not refer to any `ANSWER` relation. Two queries that entangle may ground on the same data; for example, Mickey and Minnie's flight entangled queries both ground on the `Flights` table by selecting all flights to Los Angeles.

If groundings are not handled carefully, anomalies can occur due to interference. To see an example of this, start with Mickey and Minnie's queries from Section 2 and consider the execution schedule shown in Figure 3 b). Minnie's query grounds on both `Flights` and `Airlines`, whereas Mickey's grounds only on `Flights`. Mickey receives flight number 122 as an answer. He decides to check which flights are operated by United, to gain a better understanding of the options from which his answer was chosen. However, between the time that Mickey gets the entangled query answer and the time he reads `Airlines` himself, Donald adds a new flight with number 125 operated by United. Clearly this creates a problem for Mickey. Mickey does not perform a classical unrepeatable read, because he only reads `Airlines` once. The key to understanding this anomaly is that Minnie has read the same table *prior* to entanglement; intuitively, there has been some information flow from the `Airlines` table to Mickey's transaction during the answering of his entangled query. His subsequent explicit read of `Airlines` therefore shows him information that is inconsistent with his prior knowledge of this relation.

This and other similar anomalies can be formalized using the notion of a *quasi-read*, which models the information flow that occurs through entangled query answering. In our example, we say that Minnie's grounding read on `Airlines` was also a *quasi-read* by Mickey's transaction on the same table. It is now clear that Mickey has indeed performed an unrepeatable read on `Airlines`: a quasi-read before Donald's write and a normal read afterwards. Consequently, we introduce **unrepeatable quasi-reads** as the second class of anomalies which is unique to entangled transactions. This includes all anomalies involving two reads on the same object by the same transaction, at least one of which is a quasi-read, and where the object changes value between the reads.

Entangled isolation Our anomaly-based definition of entangled isolation prohibits widowed transactions and unrepeatable quasi-reads, as well as all classical isolation anomalies; that is, an execution schedule is entangled-isolated if it exhibits none of these anomalies. The definition is presented formally in the Appendix, Section C.2; it uses the notion of a *conflict graph* which tracks operation conflicts between transactions to exclude both the classical anomalies and unrepeatable quasi-reads. As in the classical case, it is possible to relax this definition to admit lower isolation levels by permitting a specific subset of the above anomalies to occur.

3.3.2 Oracle-serializability based definition

The key idea behind oracle-serializability is to compare a given execution schedule to a schedule where the same entangled transactions are executed serially alongside a suitable query oracle. The oracle answers need to be consistent across transactions; if Mickey's transaction executes first and receives 122 as the answer to its flight query, Minnie's transaction should also receive 122 as an answer to the corresponding query when it executes later.

As explained in Section C.3 in the Appendix, the oracle is constructed in a custom way for each schedule σ. It essentially "stores" the entangled query answers that each transaction received in σ and returns them verbatim at the appropriate point during serial execution, whether or not these answers are valid (as per Definition 3.3).

We define a schedule σ to be oracle-serializable if there is some serial order of the transactions in σ for which execution with the above oracle is valid and yields the same final database as σ when run on the same starting database. This definition is the entangled equivalent of classical final-state serializability.

The following theorem is our main result and relates both of our definitions of isolation. It is proved in the Appendix, Section C.4.

THEOREM 3.6. *Any schedule that is entangled-isolated is also oracle-serializable.*

As expected, the serialization order for the oracle execution must be consistent with the conflict graph.

3.3.3 Enforcing Isolation

To enforce isolation for entangled transactions, a system must prevent widowed transactions and unrepeatable quasi-reads, in addition to the classical isolation anomalies. Widowed transactions can be avoided by enforcing **group commits**: if two transactions entangle, both must either commit or abort. This pairwise requirement induces a requirement on groups of transactions that have entangled with each other directly or transitively: all transactions in

such a group must either commit or abort. As for repeatability of quasi-reads, it can be enforced for example using an appropriate locking protocol. In a system that uses Strict 2PL, Donald's write in Figure 3 b) would not have been possible, as Minnie's transaction would have held a read lock on the `Airlines` table until commit.

3.4 Atomicity and Durability

Because we have identified individual entangled transactions as our basic unit of work in the system, we can define atomicity and durability based on their classical equivalents. For atomicity, each entangled transaction must execute to completion or be rolled back. For durability, if an entangled transaction commits, its database writes must be persistent despite any system failures.

The above are the minimal atomicity and durability requirements which the system must enforce at all times. Stronger guarantees are sometimes possible. For example, at isolation levels that disallow widowed transactions, all *groups* of transactions that entangle together are guaranteed to execute atomically. Moreover, once a transaction commits, both its changes and the changes of all its entanglement partners are durable.

4. EXECUTION MODEL

With the semantic model for entangled transactions in place, we turn to the challenges and performance tradeoffs associated with *execution* models. There is no single best execution model for entangled transactions; the final choice depends on the requirements of the application. In this section, we highlight the tradeoffs and propose a solution suited to realistic scenarios like travel planning.

Interactivity. One of the first key characteristics of an application is whether the transactions are *interactive* or *non-interactive*, or both. Interactive transactions are created by users online, statement by statement. Subsequent statements are constructed dynamically, based on the result of earlier operations. An interactive user may be willing to wait a few minutes for his or her entangled query to find partners and return results. If results are not forthcoming, then the user may decide to abort or issue another command. This interactive model is suited, for example, to social games.

However, in other scenarios such as travel planning, users who want to coordinate will most likely not issue their queries simultaneously and wait for answers at the computer. A non-interactive model is a better fit here: users can be expected to issue entire entangled transactions at once and specify an appropriate timeout. If no partner is found before the timeout expires, then the transaction aborts and is removed from the system. In this paper, we present an execution model for non-interactive transactions; exploring the unique issues associated with interactivity is future work.

Concurrency Control Protocol. As discussed in Section 3, several different isolation levels may be appropriate for entangled transactions. The choice of level is up to the system designer and depends on the application's consistency requirements. Whatever the isolation level(s) to be used, the execution model must include a suitable protocol to enforce them.

For scenarios such as collaborative travel planning, a high level of isolation is desirable to ensure consistency of the underlying database. Full isolation can be achieved by enforcing group commits and using a standard strict two-phase locking protocol. This protocol has the additional advantage of admitting isolation relaxations, if desired, by altering the length of time locks are held.

Scheduling. The system needs a policy for handling transactions that cannot currently be matched with entanglement partners. The best choice of policy depends on whether the transactions are interactive and on the isolation level desired. The discussion below makes the assumption that we are working with noninteractive transactions and that we desire full entangled isolation as defined in Section 3.3.

A naïve policy where each transaction blocks at each entangled query until it has found a partner is impractical. This blocked transaction may need to hold locks while it waits, unacceptably delaying the progress of other transactions in the system. One solution is to limit the time for which an entangled query blocks. If a partner does not arrive within a limited time frame, the transaction is aborted and restarted.

It is possible to take this idea further and organize the execution of transactions in discrete batches or *runs*. Each run is an execution of a set of transactions chosen by the scheduler. If an entangled transaction arrives in the system while a run is ongoing, it is suspended and added to a pool of dormant transactions. Designing an optimal scheduling policy is nontrivial. A simple policy is to schedule runs with a particular *frequency*, and include in a run all transactions present in the dormant pool. The frequency can be explicitly given as a time interval, or it can depend on the arrival rate of new transactions. For example, the system may schedule a new run once ten new transactions have arrived.

When a transaction is scheduled in a run, all classical queries and updates that precede the first entangled query are executed. At this point, the transaction blocks. Eventually, all transactions in the run either block, abort or reach the ready to commit state. Now, the system evaluates all pending entangled queries. If an entangled query receives an answer, the transaction is notified and resumes execution. The run terminates when each transaction has either aborted, reached the ready to commit state, or blocked on an entangled query and is unable to proceed. Transactions that are ready to commit and satisfy the group commit constraints (if applicable) are committed. Blocked transactions are aborted and returned to the dormant pool for execution in subsequent runs.

To illustrate run-based transaction scheduling, we walk through an example execution of three entangled transactions. The first is Mickey's transaction from Figure 2, and the second is a symmetric transaction by Minnie who wants to coordinate with Mickey. The third transaction follows the same structure, but it involves Donald who is interested in coordinating with Daffy.

Suppose Mickey's and Donald's transactions arrive in the system first. The scheduler creates the first run that includes these two transactions only. The first piece of code in each transaction is the respective flight booking entangled query. Neither transaction is able to progress; therefore, the system immediately aborts the run and returns both transactions to the dormant transaction pool.

Now, Minnie's transaction arrives in the system and is placed in the pool. The scheduler creates a second run containing all three transactions. The execution of this run is shown in Figure 4. Mickey and Minnie's transactions are able to execute their first entangled query; they proceed to their respective flight booking code. Donald's entangled query does not receive an answer, so his transaction blocks. Once Mickey and Minnie complete their flight booking, their transactions reach the hotel entangled queries. These are submitted for evaluation together with Donald's flight query, which has not received an answer yet. Again, Mickey and Minnie receive answers and are able to proceed, while Donald does not. Eventually, Mickey and Minnie both reach a state where they are ready to commit, pending their partner's commit. Donald's transaction, however, is still blocking on the flight query. The system recognizes that no-one can proceed further and takes action. Mickey and Minnie's transactions are allowed to commit, while Donald's is aborted again and returned to the dormant transaction pool.

Persistence and Recovery. Entangled transactions come with atomicity and durability requirements, as outlined in Section 3.4.

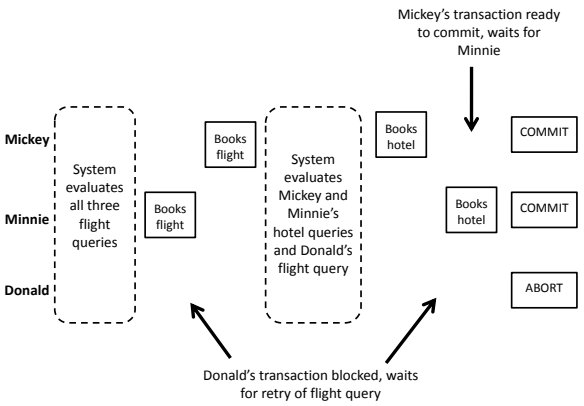


Figure 4: Example run of three transactions

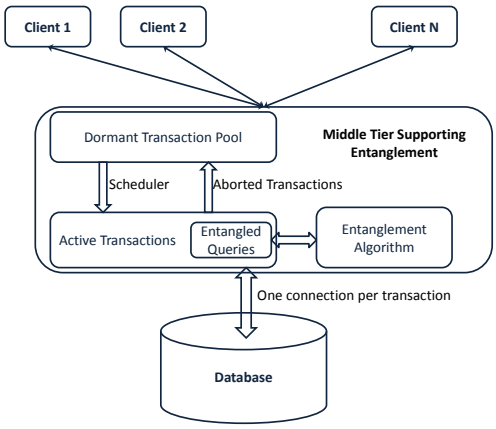


Figure 5: System architecture

It is therefore necessary to ensure correct crash recovery. Standard algorithms must be suitably modified to handle the additional entanglement-related recovery challenges.

In processing entangled transactions, the system maintains additional state to keep track of the transactions that are currently in the system and awaiting partners. It also may be keeping track of who has entangled with whom in order to enforce group commits. This state must be made persistent to ensure correct crash recovery. Further, the recovery algorithm must be entanglement-aware. For example, if two transactions entangle and only one manages to commit prior to a crash, both must be rolled back during recovery.

5. IMPLEMENTATION

In this section, we discuss our prototype implementation for the entangled transaction management component in the Youtopia system and present the results of our experimental evaluation.

5.1 Prototype

Our prototype implements entangled transaction support in the middle tier, as shown in Figure 5. This design makes it easy to port current applications without any significant change to the DBMS or the interface. Alternately, entangled transactions could be implemented within the DBMS itself, which has the potential to improve performance through deep optimizations of the entanglement logic; investigating this alternate architecture is future work.

The prototype is a component within our Youtopia system; it is implemented as a Java application over a MySQL database (version 5.5.0) using the InnoDB engine. It provides an API for clients to manage and query the database, with the added functionality of answering entangled queries and managing entangled transactions. Youtopia users submit transactions (entangled and classical) through a front-end interface. Youtopia executes classical transactions as-is and sends query results back to the client; entangled transactions are handled by our custom component.

The prototype supports the execution model discussed in Section 4. It handles non-interactive transactions and uses a run-based scheduling protocol for execution. During runs, entangled queries are evaluated using the algorithm described in [6].

Transactions can be executed at different isolation levels. If full isolation is desired for strong consistency, the system enforces group commits to prevent widowed transactions and uses Strict 2PL to prevent all other isolation anomalies. The locking protocol is implemented using the lock manager of the DBMS.

In our implementation, the middleware is stateless. All relevant system state is serialized and stored in the database to achieve persistency. This allows us to leverage the recovery algorithms implemented in the DBMS to ensure correct crash recovery.

5.2 Evaluation

In our experiments, we set out to measure the overhead associated with supporting entangled transactions relative to two other abstractions: classical transactions and non-transactional code containing entangled queries. We also wanted to investigate the tradeoffs associated with the design decisions described in Section 4.

5.2.1 Experimental Setup

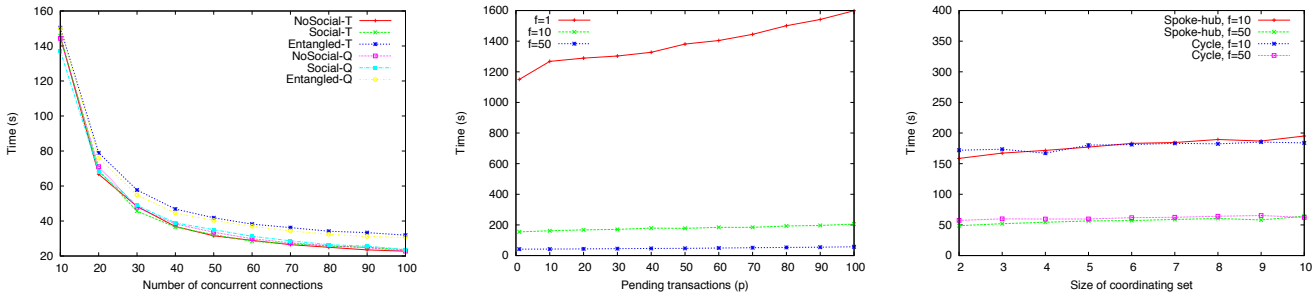
All our experiments were set in the travel scenario discussed throughout the paper and used a workload of simulated entangled transactions that modeled the output of a front-end social travel application. We created a set of users with friendship relations based on the *Slashdot* social network data [1]. Each transaction was generated by choosing a user and expressing his or her intent to coordinate on flight and/or hotel bookings with a set of friends; for examples, see Section D in the Appendix. Each transaction contained a single entangled query, except where indicated otherwise.

In MySQL, as in most commercial database systems, the amount of concurrency is restricted by the maximum permissible number of connections rather than the computational capacity of the system. This is because only a single transaction may run per connection. We worked within these constraints for the purpose of our experiments. By default, we used 100 concurrent connections, and we examined experimentally the impact of varying that number.

We ran all experiments on a 2.13Ghz Intel Core i7 CPU with 4GB of RAM; the reported values are averages over 3 runs. The standard deviation was less than 2% in each experiment. All experiments involved 10000 (ten thousand) transactions which were run to completion.

5.2.2 Results

Concurrency. In the first experiment, we varied the number of concurrent connections to MySQL from 10 to 100 and investigated the performance of six different workloads. As mentioned, we wanted to compare entangled transactions (`Entangled-T`) to non-entangled transactions in which users make travel bookings based on existing bookings by their friends (`Social-T`). Our third workload, `NoSocial-T`, contained individual travel booking transactions – that is, transactions that made no reference at all to the activity of the user's friends. In addition, for each of the above



(a) Concurrent transactions **(b) Number of pending transactions** **(c) Entangled queries per transaction**

Figure 6: Scalability of Youtopia

three workloads, we generated a corresponding non-transactional workload that used the same code without enclosing it within a transaction block. The non-transactional workloads are identified by the suffix `-Q` instead of `-T`. A further discussion of these workloads, together with examples, is found in the Appendix, Section D.

For simplicity, all workloads were generated to ensure that all transactions within a single run would be able to coordinate. That is, transactions were submitted in batches designed so that each transaction would find a coordination partner within the same batch.

Figure 6(a) shows the results. The time taken to execute any given set of transactions was observed to be inversely proportional to the number of concurrent connections for all three transactional workloads. Although the time taken by `Entangled-T` was always marginally higher compared to `NoSocial-T` (and `Social-T`), the difference was roughly equal to the difference is execution time between `Entangled-Q` and `NoSocial-Q` (and `Social-Q`). This shows that entangled transactions do not impose significant additional overhead relative to classical transactions, except for the extra time needed for the actual evaluation of entangled queries.

Pending Transactions. The first experiment was engineered so that all concurrently submitted entangled transactions would find coordination partners and commit. However, this may not be true in real life. We therefore ran a second experiment where the number of pending transactions remaining at the end of a run, p, was nonzero and varied from 10 to 100. This was achieved by submitting the transactions in carefully designed batches to ensure that each run contained p transactions without coordination partners.

We used three different run scheduling policies with different run frequencies f. We set f in terms of the arrival rate of new transactions in the system and varied it from 1 (start a new run after a single new transaction arrives) to $f = 50$ (start a new run after fifty new transactions arrive).

Figure 6(b) shows the results. As expected, using higher run frequencies had a negative impact on execution time. Moreover, increasing p caused a linear increase in the total execution time. However, this increase was much slower when the run frequency was lower. Clearly, the optimal run frequency for a given workload depends on the expected value of p.

Entanglement Complexity. Our last set of experiments investigated the impact of varying the complexity and structure of the entanglement between transactions. The intuition is that with more complex entanglement structure and more entangled queries per transaction, entanglement may be harder to achieve and transactions may abort more frequently before succeeding.

The specific parameters we varied were the number of entangled queries per transaction and the structure of the coordination. We considered two complex coordination structures. In the `Spoke-hub` structure, a single transaction with multiple entangled queries entangles with a different partner on each query. The `Cyclic` structure is even more complex and involves a cyclic set of entanglement dependencies between a set of entangled transactions.

On all the above workloads, we ran experiments with a run frequency f of 1 and 50. Figure 6(c) gives the results. Increasing the number of entangled queries per transaction increases the total execution time; however, the slope is very small. This suggests that increasing entanglement complexity does not have a significant negative performance impact.

6. ACKNOWLEDGMENTS

This research has been supported by the NSF under Grants IIS-0534404, IIS-0911036, by a Google Research Award, by NYSTAR under Agreement C050061, by the iAd Project funded by the Research Council of Norway, and by a Humboldt Research Award. Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsors.

7. REFERENCES

[1] http://snap.stanford.edu/data/soc-Slashdot0902.html.

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

[3] V. Conitzer and T. Sandholm. Expressive negotiation over donations to charities. In *ACM Conference on Electronic Commerce*, pages 51–60, 2004.

[4] S. Dalal, S. Temel, M. Little, M. Potts, and J. Webber. Coordinating business transactions on the web. *IEEE Internet Computing*, 7(1):30–39, 2003.

[5] H. Garcia-Molina and K. Salem. Sagas. *SIGMOD Rec.*, 16(3):249–259, 1987.

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

[7] 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.

[8] G. Koutrika, B. Bercovitz, R. Ikeda, F. Kaliszan, H. Liou, Z. M. Zadeh, and H. Garcia-Molina. Social systems: Can we do more than just poke friends? In *CIDR*, 2009.

[9] N. A. Lynch and M. Merritt. Introduction to the theory of nested transactions. *Theor. Comput. Sci.*, 62(1-2):123, 1988.

[10] D. Martin, J. Gehrke, and J. Halpern. Toward expressive and scalable sponsored search auctions. In *ICDE*, 2008.

[11] M. H. Nodine and S. B. Zdonik. Cooperative transaction hierarchies: A transaction model to support design applications. In *VLDB*, pages 83–94, 1990.

[12] C. Pu, G. E. Kaiser, and N. C. Hutchinson. Split-transactions for open-ended activities. In *VLDB*, pages 26–37, 1988.

APPENDIX

A. ENTANGLED QUERIES

This section briefly overviews the semantics of entangled queries. For more details, see [6].

The semantics of entangled queries is best explained using an intermediate representation that uses a Datalog-like syntax. In this representation, a query has the form

$$\{C\}\ H\ :\!-_1\ B$$

where C and H are conjunctions of relational atoms over answer relations and B a conjunction of relational atoms over database (non-answer) relations. B, H and C are the *body*, *head* and *postcondition* of the query, respectively. Each atom may contain constants and variables. Any variables that appear in H or C must also appear in B (a range-restriction requirement).

For an entangled query in extended SQL, H corresponds to the `SELECT INTO` clause in an obvious way. B and C correspond to information in the `WHERE` clause. C specifies all the conditions on tentative relations from the `WHERE` clause. B specifies the conditions on database relations from the `WHERE` clause, as well as serving to bind variables that are used in H and C.

As an illustration, Figure 7 (a) is the intermediate representation of Mickey and Minnie's queries from Section 2. The relation names are abbreviated in the same way as in the original SQL queries.

From the point of view of a single entangled query, evaluation is a process that returns an *answer*, i.e. a single row from the appropriate answer relation. From the point of view of the system, evaluation always involves a set of entangled queries, and the goal is to *populate* the answer relation in a way that respects all queries' coordination constraints. This process is described next; for correctness, it is necessary to ensure that the underlying database is not changed while it is being carried out.

Grounding the queries: Entangled query answering makes use of two concepts – *valuations* and *groundings*. If q is a query in the intermediate representation and the current database is D, a valuation is simply an assignment of a value from D to each variable of q. For example, on the database in Figure 1 (a), Mickey's query has three valuations: x can be mapped to either 122, 123 or 124. Every valuation of a query is associated with a *grounding*, which is q itself with the variables replaced by constants following the valuation.

Let $\mathcal{Q}$ be the set of queries to be evaluated in an entangled manner. We denote by $\mathcal{G}$ the set of all groundings of the queries on the database ($\mathcal{G}$ need not explicitly be materialized in evaluation). Figure 7 (b) shows the set $\mathcal{G}$ obtained by grounding Mickey and Minnie's queries on the database in Figure 1 (a). The bodies of the groundings are no longer needed, so we discard them.

Finding the answers: At a high level, the evaluation is a search for a subset $\mathcal{G}' \subseteq \mathcal{G}$ such that $\mathcal{G}'$ contains at most one grounding of each query and the groundings in $\mathcal{G}'$ can all mutually satisfy each other's postconditions. That is, if all the heads of the groundings in $\mathcal{G}'$ were combined into a set, this set would contain all the postconditions. Any set of groundings satisfying this property is called a *coordinating set*. Once such a $\mathcal{G}'$ is found, the evaluation produces an answer relation which consists of the union of all the head atoms in $\mathcal{G}'$ (the answer may consist of more than one relation – this will happen if the head atoms refer to more than one relation, i.e. more than one `ANSWER` relation is mentioned in the SQL queries).

In the example, the initial set $\mathcal{G}$ is as shown in Figure 7 (b). Groundings 1 and 4, as well as groundings 2 and 5, are suitable coordinating subsets $\mathcal{G}'$. Either of them may be used to generate the answer relation and return answers to the respective queries.

$$\{R(\texttt{Minnie}, x, y)\}\ R(\texttt{Mickey}, x, y)\ :\!-_1\ F(x, y, \texttt{LA})$$
$$\{R(\texttt{Mickey}, z, w)\}\ R(\texttt{Minnie}, z, w)\ :\!-_1\ F(z, w, \texttt{LA}) \wedge A(z, \texttt{United})$$

(a)

1:	$\{R(\texttt{Mickey}, 122, \text{May } 3)\}$	$R(\texttt{Minnie}, 122, \text{May } 3)$
2:	$\{R(\texttt{Mickey}, 123, \text{May } 4)\}$	$R(\texttt{Minnie}, 123, \text{May } 4)$
3:	$\{R(\texttt{Mickey}, 124, \text{May } 3)\}$	$R(\texttt{Minnie}, 124, \text{May } 3)$
4:	$\{R(\texttt{Minnie}, 122, \text{May } 3)\}$	$R(\texttt{Mickey}, 122, \text{May } 3)$
5:	$\{R(\texttt{Minnie}, 123, \text{May } 4)\}$	$R(\texttt{Mickey}, 123, \text{May } 4)$

(b)

Figure 7: (a) Intermediate representation of queries (b) Grounded queries

It is possible that for some queries q, the $\mathcal{G}'$ chosen will not contain any groundings of q. It is also possible that some queries in $\mathcal{Q}$ cannot be answered for other reasons; for instance, the algorithm in [6] requires all query sets to satisfy a property called *safety*, and queries that directly cause safety violations are not answered. Such failures in query answering are left for the programmer to handle in an application-appropriate manner.

B. ENTANGLED QUERY FAILURE

As mentioned in Section A, it is possible for the system to evaluate an entangled query but return an empty answer or determine that the query is unanswerable. The question arises of how this should be handled in an entangled transaction. In classical transactions, there is a meaningful distinction between the case of an empty query answer and a query that is unanswerable due to an error or exception. In the former case transactions can typically proceed, but in the latter they are usually aborted.

Such a distinction makes sense for entangled transactions as well. Intuitively, if a query found an entanglement partner but the two queries together could not be answered in a way that satisfied all constraints, this is different from the case where a query did not even find a partner. In the former case, the transaction should probably proceed, in the latter it should probably wait for the query to be retried. It remains to classify all possible cases of entangled query "failure" on either side of this dichotomy. We argue that any criterion used to define true query failure – i.e. a situation where a query cannot proceed – should be independent of the underlying database. That is, a given set of entangled queries should either fail on all databases or succeed on all databases. This is consistent with the philosophy behind the classical distinction discussed above – a query that receives an empty answer on one database may receive a nonempty one on another, and therefore an empty answer is not in itself a "pathological" phenomenon that would require an abort.

One concrete proposal for making this distinction for entangled queries, if using the evaluation algorithm from [6], is the following. If an entangled query q was used to formulate a combined query and this query was evaluated, but returned an empty result, this is considered query success and the transaction can proceed. Otherwise, if no combined query including q was formulated, q has failed and the transaction must wait.

C. FORMALIZING ISOLATION

In this section, we formalize the presentation of entangled isolation from Section 3.3. Our presentation does not handle predicate reads explicitly, nor does it deal with schedules produced in systems using explicit data versioning. The entire discussion that follows can be extended to handle predicate reads with suitable addi-

tional formalism [2]. We have chosen not to present this extension here as it requires significant additional notation and is orthogonal to our main focus, which is the unique meaning of isolation for entangled transactions and specifically the differences between entangled and classical isolation. Multiversion settings come with their own unique challenges – as in the case of classical transactions – and we leave their treatment as future work.

C.1 Transaction schedules

Operations A schedule for entangled transactions is very similar to a schedule for ordinary transactions and contains the familiar read, write, abort and commit operations, denoted R, W, A and C respectively. The only two differences pertain to entangled query processing: certain reads are distinguished as *grounding* reads and the schedules make use of an additional operation – *entanglement*.

Entangled query processing begins by grounding the queries. This can be done individually or through a combined query as in [6]. To remain implementation-independent, we model the more general case where each query grounds separately. Each grounding is a set of reads; we distinguish these as grounding reads and denote them as R^G rather than just R. Technically, the grounding reads are not performed by the transaction itself, but by the system *on behalf* of the transaction. Nonetheless, they clearly represent information flow from the database into the transaction; we therefore associate grounding reads with the transaction posing the entangled query, rather than with a special "system" process.

The next step in entangled query evaluation is to find a set of groundings that satisfy each other's postconditions, i.e. to construct the answer relation. We model this with an explicit *entanglement* operation, denoted E. We assign each entanglement operation a unique identifier, and associate each entanglement operation with the set of identifiers of the transactions that receive answers, denoted as $\mathcal{T}_k$, where k is the identifier of the entanglement operation. To introduce notation by example, if transactions 1 and 3 participate in entanglement operation 7, this is denoted as $E^7_{1,3}$

Validity constraints An entangled transaction execution schedule is a sequence of read, write, entangle, commit and abort operations. Obviously it is possible to construct sequences of such operations that do not correspond to any possible real transaction execution. This means that we need a notion of *valid* schedules – sequences of operations that are constrained to match the semantics of entangled transactions. The constraints involved are straightforward; however, for completeness we present them next.

First, for every transaction i, a valid schedule may contain at most one of $\{A_i, C_i\}$. Indeed, we find it helpful to require that it contain *exactly* one of these, thus ensuring we work with complete schedules (histories) only. Second, for every transaction i that aborts or commits, the abort or commit operation must be the last operation by i. Third, if a transaction i performs a grounding read $R^G_i(x)$ on some object x, then the schedule must contain either a subsequent entanglement operation involving i or a subsequent A_i. Fourth, consider the interval between a grounding read by transaction i and the next entanglement or abort operation by i that follows it (such an operation must exist by the previous requirement). During that interval, i may not perform any operations other than additional grounding reads. This is because entangled query evaluation calls are blocking: i cannot proceed with subsequent reads or writes until entanglement occurs and it receives answers.

Schedules We can now formally define (valid) schedules for entangled transactions.

Definition C.1 (Schedules). *A schedule is a sequence of the following operations: read, write, abort, commit, and entangle, where each operation is tagged with one or more transaction identifiers, and the sequence satisfies the validity constraints listed above.*

An example schedule is as follows:

$$R^G_1(x)R^G_2(y)R_3(z)E^1_{1,2}W_1(z)W_2(w)C_1C_2C_3$$

In this schedule, transaction 1 grounds on x, then 2 grounds on y. 3 performs a normal read on z, after which 1 and 2 and entangle together based on their grounding reads. Finally, 1 and 2 perform a write to z and w respectively. All three transactions commit.

When a schedule σ is executed on a database, the final database produced reflects exactly the writes of all the *committed* transactions in σ, in the order in which these writes occurred in σ. We assume that the entangled query evaluation algorithm is deterministic, i.e. always returns the same answers when processing the same set of queries on the same database. This implies that whenever σ runs on the same starting database, it produces the same final database. Lifting this assumption is possible but would make the presentation that follows more complex.

C.2 Anomaly-based entangled isolation

We now formalize our anomaly-based isolation definition.

C.2.1 Preliminaries

Quasi-reads Suppose two transactions i and j perform grounding reads on two different objects, say x and y respectively, and entangle immediately afterwards. Although i has not directly read y and j has not directly read x, there has been some information flow from each object to each transaction through entanglement. As discussed, we model this information flow through *quasi-reads*. Whenever a transaction performs a grounding read on an object, all of its partners in the subsequent entanglement operation are considered to perform a simultaneous quasi-read on the same object. We denote a quasi-read by R^Q. In the pathological case where a transaction performs a grounding read but there is no subsequent entanglement operation (i.e. the transaction aborts instead), no quasi-reads are associated with that grounding read.

Given an entangled transaction schedule, it is straightforward to identify which grounding reads are associated with quasi-reads by other transactions and to add in the quasi-reads explicitly. Concretely, our example schedule can be rewritten as follows:

$$\Big(R^G_1(x)R^Q_2(x)\Big)\Big(R^G_2(y)R^Q_1(y)\Big)R_3(z)E^1_{1,2}W_1(z)W_2(w)C_1C_2C_3$$

The brackets surrounding a set of operations denote that the operations occur simultaneously. Often they will not be necessary as the timing of the quasi-reads will be clear.

In the remainder of this section, we use the word *schedule* to refer to valid entangled transaction schedules in which the quasi-reads are made explicit. The unqualified term *read* refers to any read operation including a grounding read or quasi-read.

Conflicts Given a schedule σ, we can compute a *conflict graph* for the *committed* transactions in σ. This is a graph where the nodes correspond to transaction identifiers and edges are added based on conflicting operation pairs on the same object. A pair of operations on the same object by two different transactions i and j are *conflicting* if at least one is a write. If the operation by i occurs in the schedule first, we add an edge from i to j in the conflict graph.

It is important to realize that the graph is defined only for those transactions that *commit*; we only place restrictions on anomalies that affect committed transactions. This allows reasoning about schedules that exhibit correct isolation, but could not have been

generated by preventative (pessimistic) concurrency control implementations. For a further discussion of this issue, see [2].

C.2.2 Entangled isolation

The following three requirements on schedules σ can be used to rule out isolation anomalies.

Requirement C.2 (No cycles). *The conflict graph for σ must be acyclic.*

Requirement C.3 (No read-from-aborted). *If i is a transaction that aborts and j a transaction that commits, σ may not contain the sequence of operations $W_i(x) \dots R_j(x)$*

Requirement C.4 (No widowed transactions). *If σ contains an entanglement operation associated with transactions i and j, then it may not contain both A_i and C_j.*

Requirements C.2 and C.3 are sufficient to rule out classical isolation anomalies and unrepeatable quasi-reads, as the latter are made explicit in the schedule. Note also that when two transactions ground on the same object and entangle based on that grounding, Requirement C.2 guarantees that they see the same version of this object; as explained in Section A, this is necessary for correct entangled query answering. If the transactions were to ground on different versions of the same object and entangle, this would be an instance of an unrepeatable quasi-read.

We now formally define entangled isolation.

Definition C.5 (Entangled isolation). *A schedule is entangled-isolated if it satisfies Requirements C.2, C.3 and C.4.*

C.3 Oracle-serializability

Classical serializability compares a given execution schedule to a schedule where the same transactions execute serially. We formulate an entangled analogue of this where each transaction executes alongside an oracle. We can then reason about equivalence between an entangled schedule and its *oracle-serializations*.

C.3.1 Oracle construction

Suppose we are given a schedule σ; we explain how to construct an oracle O_σ that enables serial execution of the transactions in σ on a given starting database D. The oracle is customized to σ and to D, but not to any serialization order of the transactions in σ.

To build the oracle, identify all the entanglement operations in σ and create a procedure in the oracle that corresponds to each of these operations. In any serial schedule involving the transactions in σ, this entanglement operation will correspond to a number of oracle calls by the individual transactions, and the appropriate oracle procedure will be invoked each time. For example, suppose transactions i and j entangle in an operation E^k_{ij}, and the entangled queries involved were q_i and q_j. The oracle contains a procedure specific to E^k_{ij}. This procedure will be invoked twice in any serial execution – once when i executes and poses q_i, and once when j executes and poses q_j.

The procedure to handle an entanglement operation E^k is as follows. By observing σ's execution on D, we can keep track of the actual answers returned at each entanglement operation E^k. The answers can be recorded in a data structure Ans_k which is a map from $\mathcal{T}_k$ to the set of answers, so that $Ans_k(i)$ is the answer entanglement operation k returns to transaction i. The oracle makes use of the answer set Ans_k directly; when answering the query posed by transaction i, it simply returns $Ans_k(i)$. Therefore, by construction, it is guaranteed that the oracle returns consistent answers to all corresponding entangled queries, as the answers in Ans_k are assumed to be consistent. On the other hand, the oracle answers are not guaranteed to be valid according to Definition 3.3. This means that invalid executions (Definition 3.4) for some transactions may be possible with O_σ.

C.3.2 Oracle-serializations of schedules

We now define an oracle-serialization of a schedule σ.

Definition C.6 (Oracle-serialization). *Let σ be an entangled schedule run on a starting database D and O_σ the entangled oracle for σ and D as described above. An oracle-serialization of σ on D is a schedule generated when the* committed *transactions in σ are totally ordered in some way and each transaction executes individually alongside O_σ in this order. We use $os(\sigma)$ to denote an oracle-serialization of σ.*

Oracle-serializations include only the committed transactions in σ; this is consistent with our formalization of entangled isolation. Note also that oracle-serializations of σ will in general not contain the exact same operations as σ itself. Specifically, the entangled transactions no longer perform grounding reads or quasi-reads. For instance, consider our entangled schedule example from before:

$$R^G_1(x)R^Q_2(x)R^G_2(y)R^Q_1(y)R_3(z)E^1_{1,2}W_1(z)W_2(w)C_1C_2C_3$$

Suppose we serialize this schedule in the order $3, 1, 2$ on some database D. The serialization is as shown below; O^m_l denotes an oracle call by transaction l with the same entangled query that it posed in entanglement operation m in σ. We have not listed the specific answers returned by the oracle, so the below can more correctly be considered a *template* for an oracle-serialization of σ.

$$R_3(z)C_3O^1_1W_1(z)C_1O^1_2W_2(w)C_2$$

The grounding reads for the entangled queries posed by transactions 1 and 2 are no longer there. This is unsurprising, as the schedule represents reads and writes *to the database*, whereas now the entangled queries are answered without any reference to the database, solely based on the set of answers stored in the oracle.

C.3.3 Oracle-serializability

We now define oracle-serializability – the analogue of final-state serializability for entangled transactions.

Definition C.7 (Oracle-serializability). *An entangled schedule σ is oracle-serializable if there is some serialization order of the transactions in σ such that for all starting databases D, the oracle-serialization of σ in that order on D is a valid execution and yields the same final database as σ.*

Note that although the oracle required for serialization depends on the starting database the serialization order does *not*.

C.4 Entangled isolation guarantees

In this section, we give the proof of Theorem 3.6, that is, we argue that any schedule which is entangled-isolated is also oracle-serializable.

Proof. Start with any entangled-isolated schedule σ and compute its conflict graph. Choose any total ordering of the transactions in σ consistent with a topological sort on the graph; such an ordering must exist as the graph is acyclic by Requirement C.2. Choose an arbitrary D and let $os(\sigma)$ be the oracle-serialization of σ on D with respect to that order. We must show that:

(1) the execution of $os(\sigma)$ on D is valid, and

(2) the final database produced is the same as that produced by σ itself executing on D.

To prove (1), we need to show that at the time the oracle returns $Ans_k(i)$ to transaction i, the state of the database is such that $Ans_k(i)$ is valid. To do so, we introduce a technical device we call *validating reads*. Intuitively, suppose some process were to monitor the execution of $os(\sigma)$ and actually ground each entangled query before the oracle answers it, in order to perform a validity check. For the purpose of this proof, we explicitly introduce such validating reads into the schedule $os(\sigma)$ and associate them with the transaction that asked the original entangled query. Consider our example oracle serialization from above:

$$R_3(z)C_3O_1^1W_1(z)C_1O_2^1W_2(w)C_2$$

With validating reads (denoted as R^V) added, this becomes

$$R_3(z)C_3R_1^V(x)O_1^1W_1(z)C_1R_2^V(y)O_2^1W_2(w)C_2$$

For every validating read in $os(\sigma)$, there is a grounding read in σ by the same transaction on the same object, and vice versa. In fact, suppose we could show that every validating read in $os(\sigma)$ sees the same value as the corresponding grounding read in σ. Then we could guarantee that all oracle answers are valid and point (1) follows. This is because the oracle answers are exactly the answers in Ans_k, and these were computed based on the result of the actual grounding reads in σ.

Before we prove the above statement about validating and grounding reads, consider how we might prove point (2). Suppose we add a dummy transaction at the end of both σ and $os(\sigma)$ that reads every object mentioned in σ. It suffices to show that this transaction reads the same values in both schedules.

We show both (1) and (2) by proving the stronger statement that every read in $os(\sigma)$, including validating reads and reads by the dummy transaction, sees the same value as the corresponding read in σ. We prove this using an inductive argument similar to that used for classical conflict-serializability. We make (an entangled equivalent of) the standard determinism assumption – if a transaction sees the same values for its reads and entangled query answers, and if the process that provides the entangled query answers does not abort, then the transaction will produce the same writes.

The first transaction in the serialization order in $os(\sigma)$ sees the same values as it did in σ, because in σ its reads depended only on the original values and the results of its own writes; otherwise, it would have had an incoming edge in the conflict graph and could not have been serialized first. It also receives the same entangled query answers as it did in σ, by construction. Finally, in both schedules, it has an entanglement "partner" that does not abort – a real transaction in σ that commits due to requirement C.4, and the oracle in $os(\sigma)$. Consequently, by the determinism assumption, it produces the same writes as before. As our inductive step, consider the nth transaction in $os(\sigma)$, and suppose it reads object x. In σ, this transaction cannot have seen writes to x from any aborted transactions, since that would violate Requirement C.3. Consider all committed transactions in σ that wrote to x before the current transaction read it. All such transactions must have been serialized earlier in $os(\sigma)$, since the serialization ordering follows the conflict graph. The writes of each of these transactions are the same as in σ by the inductive hypothesis, and they were serialized in the same order as in σ because the conflict graph keeps track of write-write conflicts. It follows that the nth transaction in $os(\sigma)$ also reads the same values as it did in σ. Since it also receives the same entangled query answers, the determinism assumption can be applied to infer that it makes the same writes as it did in σ. □

D. EXAMPLES

In this section we present several examples that we used in our experiments. We created a set of users with friendship relations based on the Slashdot social network data [1].

The schema for our system is as follows:

```
Reserve(uid, fid)
Friends(uid1, uid2)
Flight(source, destination, fid)
User(uid, hometown)
```

The first workload (No-Social) simulates individual travel booking transactions. It queries the user table to get the source hometown, followed by a query to find flights from this source to the destination. Finally, it makes a reservation for the user.

```
BEGIN TRANSACTION;
SELECT @uid, @hometown FROM User WHERE uid=36513;
SELECT @fid FROM Flight WHERE source=@hometown
                          AND destination='FAT';
INSERT INTO Reserve (uid, fid)
       VALUES (@uid, @fid);
COMMIT;
```

The second workload (Social) also gives a list of friends who live in the same hometown and might be flying. This information is additional to the normal flight reservation.

```
BEGIN TRANSACTION;
SELECT @uid, @hometown FROM User WHERE uid=36513;
SELECT uid2 FROM Friends, User as u1, User as u2
WHERE Friends.uid1=@uid
  AND Friends.uid2=u2.uid
  AND u1.uid=@uid
  AND u1.hometown=u2.hometown
LIMIT 1;
SELECT @fid FROM Flight WHERE source=@hometown
                          AND destination='FAT';
INSERT INTO Reserve (uid, fid)
       VALUES (@uid, @fid);
COMMIT;
```

The third workload (Entangled) checks if a particular friend is also trying to coordinate with the user to make flight reservations.

```
BEGIN TRANSACTION WITH TIMEOUT 2 DAYS;
SELECT @hometown FROM user WHERE uid=45747;
SELECT 36513 AS @uid, 'CAT' AS @destination
INTO ANSWER Reserve
WHERE (36513, 45747) IN
      (SELECT uid1, uid2 FROM
              Friends, User as u1, User as u2
       WHERE Friends.uid1=36513
         AND Friends.uid2=45747
         AND u1.uid=36513
         AND u2.uid=45747
         AND u1.hometown=u2.hometown)
  AND (45747, 'PHF') IN ANSWER Reserve
CHOOSE 1;
SELECT @fid FROM Flight WHERE source=@hometown
                          AND destination=@destination;
INSERT INTO Reserve (uid, fid)
       VALUES (@uid, @fid);
COMMIT;
```