Quantum Databases

Sudip Roy[1], Lucja Kot[1], Christoph Koch[2]

[1]Cornell University
Ithaca, NY 14853, USA
{sudip,lucja}@cs.cornell.edu

[2]EPFL
CH-1015 Lausanne, Switzerland
christoph.koch@epfl.ch

ABSTRACT

We introduce *quantum databases*, a new database abstraction that allows to defer the making of choices in transactions until an application or user forces the choices by observation. Conceptually, a transaction is in a quantum state – in one of many possible worlds, which one is unknown – until fixed by observation. Practically, our abstraction enables late binding of values read from the database. This allows more transactions to succeed in environments with high contention. This is particularly important for applications in which transactions compete for scarce physical resources represented by data items in the database, such as seats in airline reservation systems or meeting slots in calendaring systems. In such environments, deferral of the assignment of resources to consumers until all constraints are available to the system will lead to more successful transactions. Through entanglement of queries and transactions, a notion that we have explored in previous work, quantum databases can enable collaborative applications with a constraint satisfaction aspect directly within the database system.

1. INTRODUCTION

In the strict formulation of the law of causality
– if we know the present, we can calculate the future –
it is not the conclusion that is wrong but the premise.
— Werner Heisenberg

Database applications are often used to allocate some commodities or resources based on user requests. These commodities could be physical goods in retail applications, virtual goods in a game, the availability of shared-use physical or virtual resources such as a hotel room or an Amazon EC2 instance, time slots on a calendar, or even another human user, e.g., a carpool partner. There are two aspects that make resource allocation interesting. First, resource allocation comes with conditions from users, for example, travelers specify whether they want an aisle or window seat, employees scheduling meetings specify whether they want a large or small meeting room, and so on. However, there is often much flexibility on other features of the resource being requested. Second, typically there is some delay between the user requests and the actual "consumption" of the resource in question. For example, travelers reserve plane tickets some time in advance and there is time on the order of days – or months – between the time the booking transaction commits and the time the seat actually needs to be used.

In this paper, we observe that both flexibility and the time delay from resource request to usage can and should be exploited by the system for optimal resource allocation. The system can exploit user flexibility by explicitly keeping track of preferences and "don't cares"; it can also exploit the delay by deferring resource allocation as late as possible. Usually, requests for resources arrive over time from different users and the system has no knowledge of the future request sequence. Allocating a resource too early may prevent other requests from being fulfilled; for example, if a traveler – Mickey – who does not care about his seat is assigned the last available window seat on a plane, a subsequent user who only wants a window seat may be turned away, unless Mickey is willing to be reseated to an aisle seat. Reseating Mickey could be nontrivial and may require executing compensation logic; for example if he is traveling with his family and they all want to sit in the same row several people may need to be reseated.

Deferral of resource assignment can not only improve global utility, but the individual user's experience as well. For example, Mickey might have a preference for flying Delta, but if there are no such available flights, he is willing to fly any other airlines. It is possible that there are no seats available on Delta when Mickey submits his transaction; however, a seat may open up due to a cancellation in the future. If Mickey's flight plans have not been finalized yet, the system can automatically "reassign" Mickey's seat at this point. Moreover, users may specify more sophisticated requests that involve coordination with other users. Formalisms such as entangled and enmeshed queries [8, 3] make it possible for Mickey to specify requests such as "I want to sit next to my friend Goofy" (who will be booking his seat separately). If Goofy's seat request does not arrive in the system until much later, it is again desirable to defer seat assignment to maximize the chance that Mickey and Goofy can sit together.

Calendar management is another application domain where resource allocation is challenging. Users who schedule meetings often have flexibility in their respective calendars, but current software forces everyone to commit to a particular date and time, even if the meeting is months away. Anecdotal evidence suggests that such arbitrarily chosen time slots frequently end up conflicting with short-notice higher-priority meetings. For example, suppose Mickey schedules a work offsite with his team two months in advance for a Friday afternoon and everyone on the team makes sure to keep that time slot free. Now, suppose that on the Wednesday before the offsite, Mickey is notified that he needs to join a high-priority meeting with the company CEO on Friday afternoon. Mickey now needs to reschedule the offsite, taking into account the availability of all his team members; by now, it may be impossible to find a new slot in the immediate future where everyone is available at the same time. Rescheduling under such circumstances is time-consuming and often stressful; indeed, many companies employ full-time administrative assistants who devote a lot of time specifically to calendar management. On the other hand, suppose that Mickey and his team were all willing to delay finalizing their daily work schedules until the evening of the previous day; high-priority events could now be added at short notice with much less disruption. The uncertainty of not knowing your schedule until the day before may be worrisome to certain employees, but in highly dynamic environments where rescheduling is frequent, employees cannot be certain of their schedules anyway, so it is likely that they would accept the alternative we propose above.

Whether in travel planning, calendar management, or other application domains, most of today's resource allocation solutions do not fully utilize the opportunities provided by user flexibility and the request-to-usage time delay. If they do, they achieve this through custom ad-hoc code. There is currently no clean general solution to the entire class of resource allocation problems, and developing such a solution on top of an existing DBMS is not trivial. First, a user request for a resource that contains preferences does not directly translate into an SQL query that returns a single resource instance, unless one uses an ad-hoc solution like appending `LIMIT 1` to the query. Second, normal database transactions cannot commit without a concrete value being assigned, so deferred assignment is not possible. The typical solution used today is to allocate an ad-hoc placeholder value and change the assignment later as needed, with lots of error-prone logic at the middle tier. Third, databases do not come with functionality for keeping track of user conditions such as seat type requirements or preferences. Therefore, these must be serialized and saved for future use; again, this can be done in an ad-hoc fashion at the application layer but a better solution is desirable.

In this paper, we introduce a principled, end-to-end solution for performing resource allocation reasoning on top of standard relational databases in an OLTP setting. First, we present *resource transactions*, a formalism that extends SQL to allow the specification of transactions over resources that include user "don't cares" and preferences (Section 2).

Second, we introduce *quantum databases* as an abstraction for the associated state that the system maintains as it defers value assignment (Section 3). A quantum database allows resource transactions to commit without assigning concrete resource instances; it keeps track of all possible worlds corresponding to all possible concrete resource assignments that *could be made*. A quantum database is an intensional specification of these possible worlds using a set of constraints collected from committed resource transactions. In true quantum fashion, unless the state is observed (i.e., read) by someone, the database remains in all of these states simultaneously. This is similar to probabilistic and uncertain databases [17]; however, the key difference is that uncertainty is strictly internal to the quantum database and a read causes uncertainty to be eliminated, i.e., it forces an instantiation.

Third, we show how a quantum database is maintained as transactions execute in the system. We specify how the database is transformed by operations such as reads, writes, and updates; this involves in some cases reducing the uncertainty and in others introducing more uncertainty. We discuss algorithms and consistency issues associated with each of these operations, and we propose practical strategies which strike a sweet spot between computational tractability and optimizing resource allocation by keeping the number of possible worlds large (Section 3.2).

```
SELECT 'Mickey', F.fno AS @f, F.fdate AS @d,
       A1.sno1 AS @s
FROM Flights F, Available A1,
   OPTIONAL Available A2, OPTIONAL Adjacent J
WHERE
  OPTIONAL ('Goofy', F.fno, F.fdate, A2.sno2)
                                      IN Bookings
  AND Fdest='LA' AND ... -- join condition
CHOOSE 1
FOLLOWED BY (
  DELETE (@f, @d, @s) FROM Available;
  INSERT ('Mickey', @d, @f, @s) INTO Bookings;)
```

Figure 1: An example resource transaction

Fourth, we demonstrate the feasibility of quantum databases by implementing a prototype (Section 4) and evaluating it under realistic workloads of a real-world application (Section 5).

Lastly, we discuss the research challenges involved in making the quantum database abstraction a real-world tool for developers (Section 6). While these challenges are significant, we believe that they are possible to resolve and that working on them will yield new insights into multiple aspects of database systems, well beyond resource allocation applications.

For concreteness, most of our examples in the paper are drawn from the travel planning scenario above; however, we also discuss other application domains such as calendar management when relevant.

2. RESOURCE TRANSACTIONS

As explained previously, resource transactions are an extension of SQL allowing users to explicitly specify soft preferences and features that they do not care about in addition to traditional hard constraints. We first introduce the notion of resource transactions through an example and then discuss the salient features of their syntax and semantics.

An example resource transaction is shown in Figure 1. This is a transaction that Mickey might issue to specify that he would like one seat on any available flight to LA, and that he has a soft preference for sitting next to Goofy, if Goofy already has a booking on some flight to LA. The `SELECT` and `FROM` clauses are essentially standard SQL; the three new keywords we introduce are `OPTIONAL`, `CHOOSE` and `FOLLOWED BY`. `OPTIONAL` specifies that the specific conjunct in the `WHERE` clause is a soft preference rather than a hard constraint. `CHOOSE 1` specifies that only one seat (tuple) is desired as an answer to the query. Finally, the `FOLLOWED BY` block contains a specification of the database *writes* to be executed based on the result returned by the `SELECT-FROM-WHERE` query. In this case, the concrete seat chosen is to be deleted from the `Available` table and a suitable `Booking` is to be made.

A resource transaction has two components: first, a query with optional clauses and a `CHOOSE 1` clause, and second, a subsequent code block that involves a set of blind writes to the database. No reads are permitted within the `FOLLOWED BY` block. One might imagine using SQL queries with `OPTIONAL` and `CHOOSE` within more complex code blocks that include subsequent reads and other features. Such an extension to the basic resource transaction model is in principle possible although nontrivial; investigating the practical usefulness of this extension and developing it fully is ongoing work. Ordinary resource transactions as described above are more limited, but nevertheless provide a programming pattern suitable for the majority of our use cases. Requesting a resource and performing a set of blind writes to "reserve" the resource is by far the most common pattern used in resource allocation applications.

In the rest of the paper we use a Datalog-like notation for representing resource transactions, which is a straightforward equivalent of the SQL representation. Each transaction is denoted as follows:

$$U :-_1 B$$

U and B are conjunctions of relational atoms. We call B the body of the transaction, and U the update portion of the transaction. Any variables appearing in U must also appear in B (this is a *range-restriction* requirement). Each atom in U is either a delete (denoted with a leading $-$) or an insert (denoted with a leading $+$) of a single tuple into the database. Optional conditions are underlined and the CHOOSE 1 is denoted by $:-_1$. For example, the intermediate representation for the example in Figure 1 is as follows, with A, B, Adj, M and G abbreviating Available, Bookings, Adjacent, 'Mickey' and 'Goofy' respectively.

$$-\mathtt{A}(f_1, s_1), +\mathtt{B}(\mathtt{M}, f_1, s_1) :-_1 \begin{cases} \mathtt{A}(f_1, s_1) \wedge \\ \underline{\mathtt{B}(\mathtt{G}, f_1, s_2)} \wedge \underline{\mathtt{Adj}(s_1, s_2)} \end{cases}$$

The body of the transaction contains three atoms. The first specifies that the variable s_1 should be an available seat for Mickey. The two subsequent optional atoms specify that s_2 should be a seat adjacent to s_1 that is already reserved by Goofy. The update portion of the transaction specifies that once a suitable s_1 is found, appropriate changes should be made by deleting a tuple from Available and inserting a tuple into Bookings.

A detailed presentation of the semantics of resource transactions requires formalizing the deferred value assignment model mentioned in the Introduction. As described in detail in Section 3, the actual assignment of values to variables and "execution" of the FOLLOWED BY clause happens after the transaction has already committed. In this Section, we give a high-level intuitive overview of the semantics. First, we explain the semantics a resource transaction would have *without* the deferred assignment model, and then explain how deferred assignment both complicates certain things and presents certain unique opportunities.

A system that processes resource transactions without deferred assignment would proceed as follows. First, the body of a resource transaction is *grounded*, i.e., each variable is assigned one specific value from the database. For example, s_1 might be assigned the value 5A, f_1 might be assigned 123 (if 123 is a flight to Los Angeles), and s_2 might be assigned 5B (if this is the seat Goofy has already booked). We use the term *grounding* and (value) assignment interchangeably in the rest of the paper. Sometimes it may not be possible to find a value assignment that satisfies all the optional clauses; this is expected and permitted by the semantics, although if there is an assignment that satisfies the optional clauses it must be chosen in preference to one that does not. Once the grounding is finalized, the system makes appropriate changes to the database as specified in the update portion.

Quantum databases implement a variant of the above semantics in a setting where value assignment is not immediate, but rather deferred until after the commit of the transaction. As explained in the Introduction, instead of grounding and executing the update portion, the system maintains a guarantee that at least one suitable grounding for the committed transaction exists at all times. Once it is necessary or desirable to actually make the updates – for example, once Mickey is at the airport and checking in for his flight – then the system chooses a grounding and performs appropriate database writes. In terms of the programming API, the application is notified of the initial transaction commit only; because of the guarantee that the system subsequently maintains, the transaction will never need to be rolled back and the application is not notified again when the value assignments are actually made. (Such a second notification could in principle be issued if desired, although it is not clear whether this would ever be useful in practice.) The first notification – that the transaction has committed – represents a guarantee that the transaction will achieve its goal of booking a seat when value assignment actually happens.

The deferred grounding execution model has an impact on the "basic" semantics described above. It creates the need for two key design decisions with regard to the transaction semantics. The first relates to the treatment of optional constraints, and the second to choosing an appropriate notion of transaction serializability.

We begin with the issue of optional constraints. Suppose a committed transaction currently has a possible assignment that satisfies its optional constraints – for example, when Mickey's transaction commits, Goofy has seat 5B booked and seat 5A is open for Mickey. Now suppose another transaction arrives and makes a conflicting request. For example, Pluto specifically requests to book seat 5A, and his constraint is non-optional. Should the system allow Pluto in, or keep 5A for Mickey? Our design decision is to allow Pluto in, since Mickey's constraint was optional rather than hard. Somewhat more formally, the only invariant that the system maintains for a committed resource transaction is that there exists a satisfying assignment for its *non-optional* body atoms. In fact, optional conditions are checked only when the variables of the transaction are assigned values; if there is an assignment that satisfies optional as well as non-optional atoms, that assignment is chosen.

Second, the notion of serializability becomes interesting when we move to a deferred assignment model. The order in which resource transactions commit is not necessarily the order in which their variable assignments are fixed and in which their database updates are carried out. Suppose the system needs to fix a grounding for a committed resource transaction: in principle, it has two options. It can choose from the values available at the time the resource transaction was committed, or from those available at the time the value must be fixed. These sets may be different; in our running example, we may well expect that the seat availability has changed between the time that Mickey's transaction committed on Monday and the time that Mickey reads his seat number on Tuesday. Choosing Mickey's seat based on Tuesday's availability is a natural thing to do, but it violates classical transactional isolation. Mickey's transaction is now no longer serialized in commit order. However, the *intent* [7] of his transaction has definitely been preserved, and we have maintained *semantic serializability* as his transaction has achieved all its goals. Maintaining strict classical serializability is also possible; this requires the system to ensure that at least one seat *from those available on Monday* remains available for Mickey. This means the system must be more restrictive in terms of how many other transactions can commit. Quantum databases can implement transactions under either semantic serializability or the classical ACID-style *strong serializability*, although we expect the former to be more natural in most application scenarios.

3. QUANTUM DATABASES

We now present the details of our execution model for resource transactions which allows for deferred assignment of values to variables in committed transactions. We achieve this by maintaining the database in a partially uncertain state, called a *quantum state*, and updating it appropriately in response to various transactional operations. We call the resulting database a *quantum database.*

3.1 Defining a Quantum Database

Consider for a moment an execution model for resource transactions where value assignment is *not* deferred. Rather, at the time the transaction is executed, suppose the system finds all possible values that could be assigned – all possible flights and seats for Mickey, say – and *forks* the database state into several possible worlds. In the first possible world, Mickey gets assigned the first available seat, in the second he gets the second one, and so on. This yields a large, but finite set of possible worlds, each of which is a concrete database in which Mickey has a concrete seat on a concrete flight.

Suppose we now maintain all these possible worlds explicitly as our database state. Other resource transactions can run on each possible world as well and cause another "forking" of the state; for example, if Donald now submits a transaction, the system can create one possible world for each of Donald's possible seat assignments. If Donald only wanted a window seat, and there was only one available window seat, this would eliminate all worlds in which Mickey got that window seat. In other words, all worlds in which Donald's transaction cannot commit are eliminated. This is illustrated in Figure 2; we assume that there is only one available flight to LA, number 123, for simplicity. Now, suppose that Minnie submits a transaction of her own, requesting to sit next to Mickey. The new set of possible worlds is as shown in the final panel in Figure 2.

Suppose Mickey eventually needs to check-in and actually needs to know – i.e., read – his seat. Our goal is to make the existence of possible worlds invisible to the transaction. However, the read may have a different value depending on the possible world that it occurs in. Therefore, the system is forced to choose one possible value for the read and "collapse" the uncertainty as required so that this is the correct value read. All possible worlds consistent with the read are retained, and all others are discarded. It may so happen that some of his optional conditions cannot be satisfied in any world, in which case the world in which the maximum number of conditions are satisfied is preserved.

Of course, the number of possible worlds in the above setup would grow at an exponential rate as new transactions are processed. Therefore, an extensional representation of these worlds as described above is not practical. We can, however, represent the possible worlds *intensionally* in a concise way, and this is the idea behind our true quantum database abstraction.

At a very high level, a quantum database consists of an extensionally specified portion, which is the same in all possible worlds, and an ordered sequence of pending transactions – more precisely, committed transactions whose value assignments are still pending. To avoid cascading rollbacks and enforce serializability, it is necessary to guarantee two things – first, all the pending value assignments can be made successfully, and second, any reads on the database issued by transactions serialized after a pending transaction are handled correctly. Both of these requirements are challenging to enforce. The former requires ensuring the existence of a successful grounding for the body of each pending transaction, when these transactions are executed in a sequence. The latter requires defining what dependence between transactions really means in the case of deferred value assignments, and accurately identifying pending transactions that may affect the result of a read.

More formally, we define a quantum database as follows.

Definition 3.1 (Quantum Database). *Let D be a completely extensional initial database. Also, let $T_0, \ldots, T_N$ be a sequence of N resource transactions. A quantum database, denoted as $\widehat{D}$, represents the set of all possible database states obtained from D by applying the operations in U_0 through U_N under consistent groundings. A grounding for transaction U_i is consistent if it corresponds to a valid grounding of B_i on the database obtained by applying U_0 through U_{i-1} to D.*

The above definition does not assume the existence of consistent groundings for the bodies. In the absence of a consistent set of groundings, the quantum database $\widehat{D}$ corresponds to $\emptyset$; during normal execution, the goal is to avoid reaching such a state. This is done by disallowing changes to both the extensional portion D and the intensional portion (the set of pending transactions) as necessary. Most concretely, if adding a new transaction to the end of the sequence would cause the set of possible worlds to become empty, the transaction is disallowed.

The above intensional representation can be modified slightly to allow semantically serializable schedules as defined in Section 2. This essentially involves allowing changes to the ordering of the sequence of resource transactions as long as all constraints in the definition above can be maintained.

3.2 Maintaining a Quantum Database

We now describe how a quantum database is maintained and transformed in the presence of system operations, while satisfying the goal of retaining a nonempty set of possible worlds at all times. The operations that affect a quantum database state are reads, writes, the execution of new resource transactions, and explicit grounding (value assignment) that affects one or more pending transactions; this latter type of operation may be required for various reasons which will be explained below.

3.2.1 Composing resource transactions

As new resource transactions arrive and are processed, the system constructs a single logical formula whose satisfiability corresponds to the existence of a consistent set of groundings for all the pending transactions. Here, we explain at a high level how this formula is constructed. The intuition is that a sequence of resource transactions can be composed into a single resource transaction with a body that is more elaborate than the conjunction of the respective bodies. If the body of the new resource transaction is satisfiable on D, all the transactions are guaranteed successful execution (i.e., the existence of a successful value assignment). If a new resource transaction cannot be admitted in a way that preserves satisfiability, the transaction is rejected.

We begin with a key assumption about the database D. We assume that any relation R in D that appears in the `FOLLOWED BY` clause of a resource transaction has a key, i.e., satisfies set semantics. This property holds naturally in most cases; if it does not, it can be enforced either by normalization or by introducing dummy identifier columns.

We introduce a few definitions that are necessary for our presentation. Given a set of relational atoms containing variables and a database D, a *substitution* is mapping from variables to variables or data values from D. The most general unifier of two relational atoms b_1 and b_2 is defined as follows.

Definition 3.2 (Most General Unifier). *Let b_1, b_2 be two atoms. A unifier for b_1 and b_2 is a substitution θ such that $\theta(b_1) = \theta(b_2)$. A most general unifier (mgu) for b_1 and b_2 is a unifier θ for b_1, b_2 such that for each unifier ν of b_1 and b_2, there exists a substitution ν' for which $\nu = \nu' \circ \theta$.*

Based on the definition of most general unifier, we now define a unification predicate.

Definition 3.3 (Unification Predicate). *Let b_1, b_2 be two atoms. The unification predicate for b_1 and b_2, denoted φ, is a conjunction*

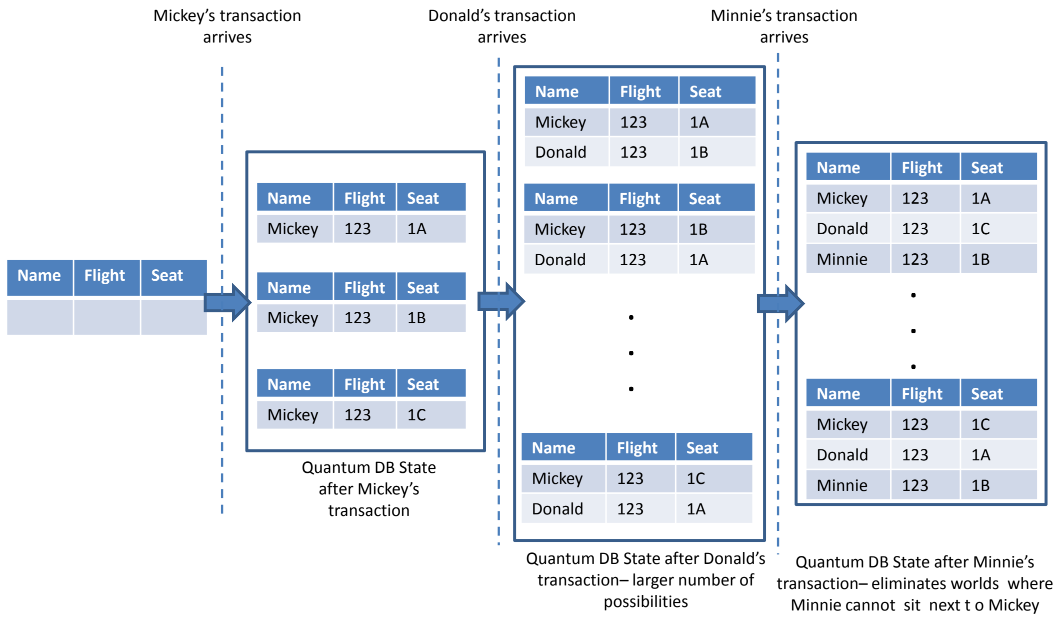
Mickey's transaction arrives
Donald's transaction arrives
Minnie's transaction arrives
Name | Flight | Seat
Mickey 123 1A
Mickey 123 1B
Mickey 123 1C
Donald 123 1B
Donald 123 1A
Mickey 123 1C
Donald 123 1A
Minnie 123 1B
Donald 123 1C
Quantum DB State after Mickey's transaction
Quantum DB State after Donald's transaction– larger number of possibilities
Quantum DB State after Minnie's transaction– eliminates worlds where Minnie cannot sit next t o Mickey

Figure 2: Evolution of an extensional quantum database

of equality constraints, where each equality constraint corresponds to a variable substitution in the most general unifier θ of b_1 and b_2.

For example, consider the atoms $\mathtt{R}(1, v_1, v_2)$ and $\mathtt{R}(v_3, 2, v_4)$. Their most general unifier is the substitution $\{\{v_1/2\}, \{v_2/v_4\}, \{v_3/1\}\}$. Correspondingly, $\varphi = (v_1 = 2) \wedge (v_2 = v_4) \wedge (v_3 = 1)$ is their unification predicate. In the absence of a most general unifier, the unification predicate is trivially *false*. If the most general unifier is empty (i.e., there are no variables in either atom), the predicate is trivially *true*.

We now explain how to compose a set of resource transactions into a single resource transaction. The Lemma below shows how this is done in the simple case of two resource transactions each containing a single body atom and a single atom in the update portion.

Lemma 3.4. *Let T_1 and T_2 be two resource transactions as given below, where each U_i and B_i is a single atom:*

$$(T_1)\;\; U_1 : -_1 B_1$$
$$(T_2)\;\; U_2 : -_1 B_2$$

A sequential execution of T_1 and T_2 on any database D is equivalent to the execution of the transaction T_{12}, given by

$$U_1, U_2 : -_1 B$$

$$\text{where } B = \begin{cases} B_1 \wedge (B_2 \vee \varphi(B_2, U_1)) & \text{if } U_1 \text{ is an insert} \\ B_1 \wedge B_2 \wedge \neg\varphi(B_2, U_1) & \text{if } U_1 \text{ is a delete} \end{cases}$$

Proof. The proof is given in Appendix A. □

Basically, the body of T_{12} reflects the fact that there needs to be a grounding for B_1 on the original database and a grounding for B_2 on the database modified by U_1; if this modification was an insert, this opens up the possibility that B_2 could ground on the inserted tuple. If U_1 was a delete, this means B_2 cannot ground on the tuple deleted by U_1. This idea extends to more general updates involving sets of inserts and deletes, and conjunctive queries with multiple body atoms.

Theorem 3.5 (Composition). *Let T_1 and T_2 be two resource transactions as given below:*

$$(T_1)\;\; U_1 : -_1 B_1$$
$$(T_2)\;\; U_2 : -_1 B_2$$

A sequential execution of T_1 and T_2 on any database D is equivalent to the execution of the transaction T_{12}, given by

$$(T_{12})\;\; U_1, U_2 : -_1 \;\; B$$

$$\text{where } B = B_1 \wedge \bigwedge_{i,j} (b_2^i \wedge \neg\varphi(b_2^i, d_1^j)) \wedge \bigwedge_{i,j} (b_2^i \vee \varphi(b_2^i, i_1^j))$$

and $b_2^i \in B_2$ and d_1^j, i_1^j are the deletes and inserts in U_1 respectively.

Proof. This can be shown with a generalization of the argument used to prove Lemma 3.4. □

An example of the composition of three resource transactions is shown in Figure 3. In the first transaction, Mickey cancels a reservation for a seat on flight number 1. In the second, Donald books a seat on a flight, where both the seat and the flight are unconstrained. In the third, Goofy books a seat on flight 2. Figure 3 (b) shows the composition of the first two transactions, and then the composition of all three. The satisfiability of the body of T_{123} guarantees that there is a possible value assignment corresponding to the ordered execution of the three original transactions.

3.2.2 Reads and Writes

We now explain how a quantum database manages the set of possible worlds it represents when handling reads and writes.

Reads: What should happen when a quantum database representing a set of possible worlds is read? For example, what happens if Mickey submits a query to find out his actual flight and seat number? There are several options. One is to expose the uncertainty to the user by returning all possible values for the read. The second is to pick a single value and return it, which can be done under

$$(T_1) \qquad -\mathtt{B}(\mathtt{M}, 1, s_1), +\mathtt{A}(1, s_1) :-_1 \ \mathtt{B}(\mathtt{M}, 1, s_1)$$
$$(T_2) \qquad -\mathtt{A}(f_2, s_2), +\mathtt{B}(\mathtt{D}, f_2, s_2) :-_1 \ \mathtt{A}(f_2, s_2)$$
$$(T_3) \qquad -\mathtt{A}(2, s_3), +\mathtt{B}(\mathtt{G}, 2, s_3) :-_1 \ \mathtt{A}(2, s_3)$$

(a)

$$(T_{12}) \ U_1, U_2 :-_1 \mathtt{B}(\mathtt{M}, 1, s_1) \wedge \{\mathtt{A}(f_2, s_2) \vee \{(f_2 = 1) \wedge (s_1 = s_2)\}\}$$

$$(T_{123}) \ U_1, U_2, U_3 :-_1 \begin{array}{l} \mathtt{B}(\mathtt{M}, 1, s_1) \wedge \\ \{\mathtt{A}(f_2, s_2) \vee \{(f_2 = 1) \wedge (s_1 = s_2)\}\} \wedge \\ \mathtt{A}(2, s_3) \wedge \neg\{(f_2 = 2) \wedge (s_3 = s_2)\} \end{array}$$

(b)

Figure 3: (a) Three resource transactions (b) The composition of the first two and all three transactions from (a) above; each U_i denotes the update portion of transaction i.

two different semantics. First, we can pick a single possible value and return it without nonetheless fixing it in the database, so that Mickey would see a particular seat number but have no guarantees that this number will remain fixed. Second, we can pick a single value at query answering time and fix it at that time, essentially "collapsing" part of the quantum state to a concrete, extensionally specified state. These three options represent different points in a trade-off between providing strong read consistency guarantees and maintaining a large set of possible worlds to allow for better future resource allocation.

The quantum databases we present in this paper use the third option mentioned above; this approach completely hides the uncertainty and allows the programmer to assume that he or she is working with a standard database that provides the expected read repeatability guarantees. In practice, we expect reads to be infrequent until the time of resource "consumption" (such as checking in for a flight), at which time of course the assignments must be fixed anyway. We note in particular that the programmer is notified by the system when his or her resource transaction commits; this notification is a guarantee that a suitable resource exists and will still exist when it is actually needed in the future. Therefore, there is no need for the programmer to issue an immediate read to "check" whether a suitable resource for the transaction exists in the system – the fact that the transaction committed already provides the desired confirmation.

In certain application-specific settings, handling reads in a way that exposes uncertainty and/or loosens consistency guarantees may be an acceptable solution. For example, consider the calendar management scenario we introduced in Section 1. We could use a quantum database to delay finalizing employee meeting schedules until absolutely necessary. However, it might be useful for employees to know *some* information about their schedules a day or two in advance: for example, they may want to know whom they are meeting so that they can prepare appropriately, without needing to know exactly when each meeting occurs. This would suggest a more complex read model where reading an employee's schedule for that day removes some of the uncertainty – regarding whom he or she is meeting – but retains the uncertainty regarding the specific time of each meeting. Developing quantum databases that use such alternate approaches for read handling is future work.

The approach we take in this paper, i.e., fixing a particular value assignment at read time, obviously reduces the opportunities available for future optimization. Two important challenges arise: what values must be fixed when handling a particular read, and how should they be fixed? Both of these are nontrivial, and it is desirable to resolve them intelligently.

Identifying the values that must be fixed to handle a read can be done at different levels of precision. At the highest level of generality, this question is related to the problem of computing information disclosure through views, which is Π_2^p-complete [14]. However, a simple practical solution is to use a conservative criterion based on unifiability. If a relational atom in our incoming read query unifies with a pending update U_i from a transaction T_i, the values involved in that transaction are fixed.

Certain reads necessitate more grounding than others. For example, a read requesting the full contents of the `Booking` table will cause many more groundings than a read asking only for Mickey's seat number. The programmer should be aware that such general reads have a potential negative impact on optimal resource allocation and should strive to avoid them. It is also possible to imagine solutions where the programmer is provided more explicit feedback before issuing a read on the potential "consequences" of that read on the possible worlds. This might be helpful to the programmer as he or she determines which reads to issue; on the other hand it violates the pure quantum database model where uncertainty is totally hidden from the programmer. Investigating the details and usefulness of such a solution is ongoing work.

Once the system decides which values need to be fixed due to a read, the system may still need to make a choice if more than one satisfying assignment is available for the transactions in question. Which specific seat among all those that are open should Mickey receive? Generally, it is desirable to fix values in such a way as to maximize the remaining number of possible worlds; more sophisticated application-specific heuristics may also be appropriate.

Writes: Writes are significant in a quantum database because they may cause the formula associated with the pending transactions to become unsatisfiable. Thus, all writes to the database which unify with the bodies of the pending transactions need to pass through a check and are rejected if the check fails. This is analogous to the check performed when processing a new resource transaction, with the difference that a blind write changes the database over which the formula must remain satisfiable rather than the formula itself.

3.2.3 Grounding

The uncertainty in the quantum database may need to be resolved at certain points, due to a read or due to application specific requirements. Fixing some concrete value assignments in the database may require actual execution of some of the committed resource transactions that were previously pending value assignment.

When a particular value assignment must be fixed and a particular pending transaction actually carried out, the naïve approach would be to also ground and apply all transactions that arrived in the system earlier. For example, consider a quantum database $\widehat{D}$ with pending transactions $T_0, \ldots, T_N$ over an extensional database D. Suppose that transaction T_i requires grounding. The naïve approach is to apply transactions $T_0, \ldots, T_i$ in turn on the database D to get a new database D', by grounding each of them and carrying out the appropriate update. The definition of a quantum database guarantees that suitable groundings exist for each of $T_0, \ldots, T_i$ as long as the set of possible worlds is nonempty. This procedure yields a quantum database with extensional state D' and the pending updates of $T_{i+1}, \ldots, T_N$. Such an approach provides strong serializability of the transactions in arrival order; however, it may overconstrain some assignments prematurely and reduce the window of opportunity for future optimization.

An alternative is to strive for *semantic serializability* (as introduced in Section 2); this approach avoids grounding transactions

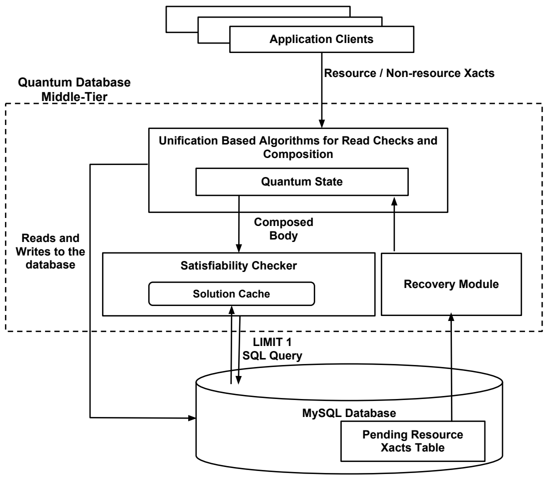
Application Clients
Resource / Non-resource Xacts
Quantum Database Middle-Tier
Unification Based Algorithms for Read Checks and Composition
Quantum State
Composed Body
Reads and Writes to the database
Satisfiability Checker
Solution Cache
Recovery Module
LIMIT 1 SQL Query
MySQL Database
Pending Resource Xacts Table

Figure 4: Quantum Database System Architecture

unless strictly necessary. The invariant associated with the quantum database $\widehat{D}$ is, in general, order dependent. While this invariant guarantees successful execution for the order in which the transactions arrived, there can be other orderings of the pending transactions which have the same desired effect. Of course, reordering the transactions affects the formula introduced in Section 3.2.1 whose satisfiability guarantees successful execution of the remaining pending transactions. This means we cannot reorder pending transactions arbitrarily, but must check for satisfiability first. That is, we must maintain the invariant that there exists some ordering of the remaining pending transactions under which the resulting formula is still satisfiable. Checking for all possible reorderings would be computationally intractable as there are exponentially many of them. A practical strategy is to check only the ordering where the transaction under consideration is moved to the front of the current ordering. In most cases, we expect the underlying satisfiability problem to be very under-constrained (i.e. many available resources and few pending transactions) so this strategy should yield good results.

4. QUANTUM DATABASE PROTOTYPE

Our quantum database prototype is implemented as a middle-tier service over a MySQL database. From the developer's perspective, the API is almost identical to the API provided by any standard database. It allows the developer to submit queries and updates to the database; the major new feature is support for resource transactions. Our current implementation does not accept and parse resource transactions in their SQL format, but only in the intermediate Datalog-like representation.

Figure 4 shows the architecture of our quantum database prototype. The constraint satisfiability checking required to maintain the quantum database invariant is performed using database `LIMIT 1` queries; alternate possible solutions are discussed in Section 6. We explain some of the important architectural features of our prototype below.

Quantum State: The prototype keeps an in-memory representation of the intensional portion of the quantum database state. This in-memory state is maintained as a set of composed transaction bodies, where each composed body is a single formula that looks like the one in Theorem 3.5. Some resource transactions are totally independent of each other, i.e., there is no unification possible between them – this is true for example of transactions that book seats on different and explicitly specified flights. The system partitions the resource transactions accordingly into independent sets and maintains a separate composed transaction body for each set. This partitioning obviously helps keep all the required computations efficient as the quantum state evolves. The partitioning is not fixed as new transactions arrive, however. For example, we may have a scenario involving two sets of transactions, one containing only requests for window seats and the other only for aisle seats. These sets are independent; however, if a new transaction arrives requesting either a window or an aisle seat, then this new transaction as well as the two original sets must all be merged and composed together.

Solution Cache: The prototype maintains an in-memory cache of possible solutions (i.e., value assignments) to the composed transaction bodies. Recall that a quantum database must maintain the invariant that there exists at least one grounding for each of the composed bodies. When a new resource transaction arrives in the system, we check whether an existing solution in the cache can be extended to accommodate the new transaction. If this is not possible, then we generate a `LIMIT 1` SQL query corresponding to the body of the new composed transaction and send it to the database. If this query has an answer, the solution cache is updated appropriately and the transaction commits. If not, then the new resource transaction is aborted. Maintaining a solution cache allows us to amortize the cost of checking satisfiability of composed bodies across a set of transactions. However, in the worst case, the search for a solution must be restarted from scratch each time. A strategy to avoid such recomputation is to increase the number of solutions maintained in the cache. Such additional solutions can be computed by a background process in order to keep the per-transaction latency low. Our current prototype does not implement this strategy, but instead maintains a single solution in the cache for every composed transaction.

Since the solution cache always contains at least one valid grounding for all the composed transactions, read queries which induce grounding of pending transactions can be answered without much overhead. If a read requires fixing some values in the database, the system can use appropriate values from the solution cache to apply the updates of the affected resource transaction(s). Once the required values are fixed in the database, the read query is processed normally.

Recovery: Since the execution of resource transactions is deferred post-commit, we need to maintain additional information about these transactions to ensure durability. We do this by utilizing the recovery mechanisms of the underlying database. Each pending resource transaction is serialized and inserted into a special database table called the *pending transactions table*. This insertion happens after the satisfiability check and before the transaction commits. During recovery, a quantum database module restores the in-memory quantum state to what it was before the crash based on the pending transactions table. When a pending resource transaction is grounded and executed, it is removed from the pending transaction table.

Our prototype is implemented as a Java application built over MySQL (version 5.5.28) using the InnoDB engine. The maximum number of relations that can be referenced in a single MySQL join operation is limited to 61; this means our system can only handle up to 61 atoms in each composed transaction body. This limitation is not fundamental to the problem itself, but arises out of the maximum number of joins supported by MySQL. The semantics of quantum databases allows the reduction of uncertainty through grounding at any time; therefore, we keep the size of the composed bodies small by forcibly grounding and executing some pending resource transactions as needed. Concretely, we ground transactions to keep the maximum number of pending transactions in each partition below a parameter *k*; when grounding, we start with the oldest transactions based on their arrival time in the system.

5. EXPERIMENTS

In this section, we show the results of an experimental evaluation of quantum databases in a realistic application setting. The aim of our evaluation is primarily twofold. One, to measure the overhead of quantum databases over traditional databases, and second, to quantify the improvement in allocation of resources due to deferred execution through quantum databases.

5.1 Application Scenario

Our experiments are set in the travel application scenario used throughout the paper, enhanced with the presence of user-defined coordination constraints that are expressed as entangled queries [8]. Entangled queries allow users to build powerful applications where the heart of the coordination – the choice of data values – is performed at the same declarative level as the data access. Entangled transactions [9] are transactions which include entangled queries. The algorithms presented in Section 3.2 can be modified easily to include functionality related to entanglement and turn quantum databases into a platform for executing entangled resource transactions.

For example, the transaction in Figure 1 can be read and executed as an entangled resource transaction, in the following way. If Mickey submits this transaction before Goofy has arrived in the system, the system can maintain Mickey's request to sit next to Goofy as a "forward constraint," to be satisfied if possible should Goofy arrive in the system later. If Goofy does arrive, the system will try to give him and Mickey adjacent seats. Of course, if Goofy never arrives, coordination is impossible, which is why Mickey's coordination constraint needs to remain `OPTIONAL`.

Entangled resource transactions are different from the (pure) entangled transactions [9] where coordination was required for successful execution. The execution model for entangled transactions does not allow an entangled transaction to commit until its partner(s) is also in the system; this means that entangled transactions must be executed in batches. Our new quantum database model allows entangled resource transactions to execute and commit individually. The benefit of the current approach is that while Mickey is no longer guaranteed to sit next to Goofy, he is now guaranteed to have a seat for himself regardless of when and whether Goofy might submit his transaction.

Since the primary motivation for deferring the execution of an entangled resource transaction is to allow coordination with a yet-to-arrive partner transaction, an entangled resource transaction waiting for its partner is finally executed as soon as its partner arrives and no longer remains in a quantum state. That is, when both coordinating users' transactions are in the system, their respective seat assignments are fixed. This situation is an example where the application logic decides how long a resource transaction should be kept in a quantum state.

5.2 Experimental Setup

We created a workload of simulated entangled resource transactions to model the output of the front-end social travel application as described above. Our workload simulates users desiring to coordinate with their friends on flights and to sit in adjacent seats. We compare this workload against a workload of non-entangled transactions issued by an "intelligent social" (`IS`) user. Such a user first issues a query to check whether his/her friend has an existing reservation. If so, he books the adjacent seat, and if not he books a seat with a free adjacent seat. The `IS` workload simulates the kind of coordination that is achievable without using a quantum database.

Order of Arrival	Characteristic	Max. Number of Pending Xacts
`Alternate`	T_i entangles with T_{i+1}	1
`Random`	T_i entangles with T_j for some i, j < N	$\lceil N/2 \rceil$
`In Order`	T_i entangles with $T_{i+N/2}$	$\lceil N/2 \rceil$
`Reverse Order`	T_i entangles with T_{N-i}	$\lceil N/2 \rceil$

Table 1: Four different transaction arrival orders and the maximum number of pending transactions in the quantum database assuming a transaction remains pending until its partner arrives.

The overhead of quantum databases depends on the complexity of checking the invariants maintained by the quantum database. Each invariant corresponds to the body of a composed transaction, and hence the complexity of checking the invariant depends on the number of pending transactions which are composed together. As described in Section 5.1, an entangled resource transaction is kept pending only until the arrival of its partner. Therefore, the order of arrival of the transactions w.r.t. their partners determines the complexity of the invariants. Table 1 shows four possible orders of arrival of transactions. In the `Alternate` arrival order, each user transaction is followed immediately by his or her partner's transaction. This only leaves a maximum of 1 pending transaction in the quantum database. The second possible order of arrival is `Random`, which orders transactions randomly and is expected to be the most realistic. While the maximum number of pending transactions for the `Random` order is $N/2$, it is expected to be lower on average. Finally, the orders of arrival which lead to over half the total number of transactions to be pending are denoted as `In Order` and `Reverse Order`. In `In Order` order of arrival, half the users submit their transactions followed by their respective partners in the same order. In `Reverse Order`, the second half of the users submit transactions in the reverse order, i.e., the first user sits entangles with the last user, the second user entangles with the second to last user and so on. While the maximum number of pending transactions for both `In Order` and `Reverse Order` are the same, for `Reverse Order` the period for which a transaction is kept pending varies from 1 to N, as opposed to a constant $N/2$ for `In Order`.

We artificially generate a database of flights over which the reservation requests are issued. Each flight in our database is represented as a set of seats arranged in rows of three. Each row has four possible adjacent pairs, only two of which can be booked simultaneously. The number of rows per flight and the number of flights in the database are changed across experiments. Appropriate indices are defined for each relation in the database.

In all our workloads, all coordination partners arrive in the system at some point so full coordination is theoretically achievable. A key metric for measuring the benefit of quantum databases is the percentage of maximum possible coordination which is actually achieved. For example, for a single flight in our database with ten rows (10×3 seats), a maximum of twenty coordination requests for adjacent seats can be accommodated, and therefore the benefit of using quantum database is measured as the fraction of users (of the maximum twenty) who actually are able to coordinate. The expectation is that quantum databases should allow us to significantly increase this percentage over what is possible with an intelligent

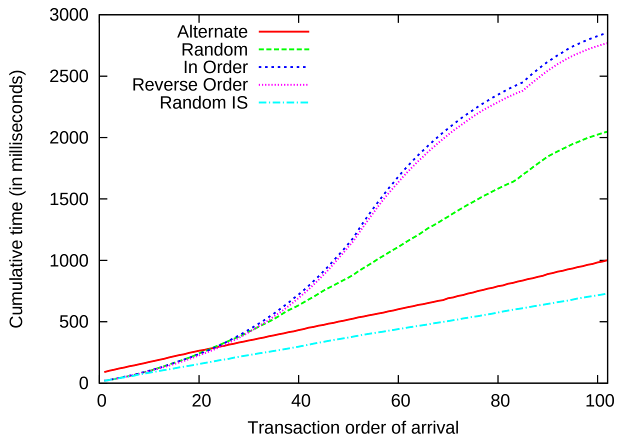

Figure 5: Cumulative time of transaction execution for different orders of arrival of transactions.

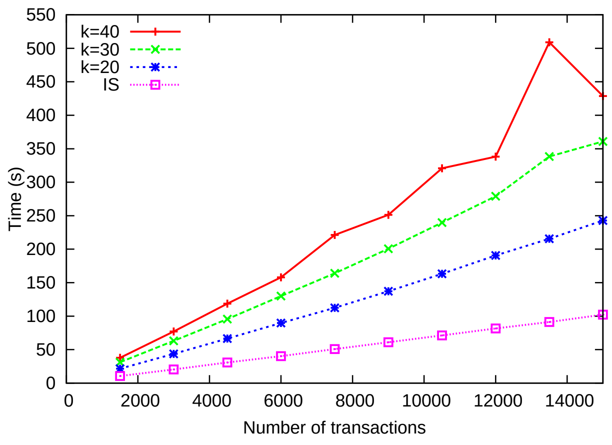

Figure 7: Scalability

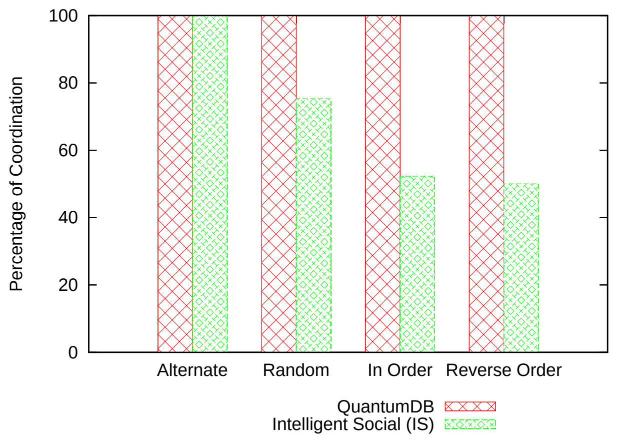

Figure 6: Percentage of coordination for different orders of arrival of transactions.

social strategy.

We ran all experiments on a 2.13GHz Intel(R) Xeon(R) E5606 with 48 GB of RAM. The MySQL query optimizer by default performs an exhaustive search over all possible query plans, and this number grows exponentially with the number of tables referenced in a join query [1]. Quantum database queries typically involve a high number of joins, and therefore for default values the database is observed to spend a disproportionate amount of time in query optimization as compared to query execution. For all our experiments, we set the value of the parameter `optimizer_search_depth` to 3 to reduce the amount of time spent in query optimization without significant change in query execution time. The reported values are averages over 5 runs. Executable `.jar` files and instructions for replicating our experiments can be found on our project website [2].

5.3 Results

Order of arrival: The first experiment measures the overhead of using a quantum database for the four different orders of arrival in Table 1. We set the parameter for k to its maximum value of 61 for this experiment. We start with a database containing a single flight with 102 seats (34 rows of 3 seats each), and issue a sequence of 102 transactions according to each order. Our choice of the value of k ensures that the partner for a transaction arrives within the window provided by k.

Our results are shown in Figure 5 and Figure 6. We ran the experiment with entangled and intelligent social workloads for each of the four transaction arrival orders described above. We found that the performance of the system on the intelligent social workload does not depend on arrival order, so we only show results for the four different arrival orders of the entangled workload and for the intelligent social workload in the `Random` arrival order. Figure 5 shows the cumulative execution time for each workload. Under the `Alternate` arrival order, the overhead of our system as compared to intelligent social is negligible. This is expected, as only a maximum of one transaction is kept pending by the quantum database. However, for both `In Order` and `Reverse Order` arrival orders, our system is substantially slower than the intelligent social approach. The steep slopes for both these arrival orders in Figure 5 are caused by the increasingly larger bodies of the composed transactions. As the partner queries start to arrive in the second half of the respective workloads, the slope reduces as the number of pending transactions decreases. The `Random` arrival order, which we expect to be by far the most realistic, shows a small overhead over intelligent social, on the order of a few milliseconds per transaction. We believe this is acceptable in practice, particularly given the significant increase in successful coordination that is gained through using a quantum database.

Figure 6 shows the percentage of the total possible coordination that the system actually achieves for each arrival order. The quantum database achieves the maximum possible coordination for all of the four workloads; for all arrival orders except `Alternate`, this is a substantial improvement over what is possible with the intelligent social approach.

Scalability: We test the scalability of our system as the number of flights in the database is increased from 10 to 100. Each flight in the database has 150 seats (50 rows of 3 seats each). We initialize the system in a state where all flights are fully available, and issue as many transactions as there are available seats in `Random` order. Upon completion of all transactions each user has a seat and all available seats are booked. Instead of maintaining a single large invariant for all flights, the system correctly identifies the independence of queries between different flights, and it maintains a relatively smaller set of invariants, one for each flight. Such partitioning allows the system to scale as the number of flights is in-

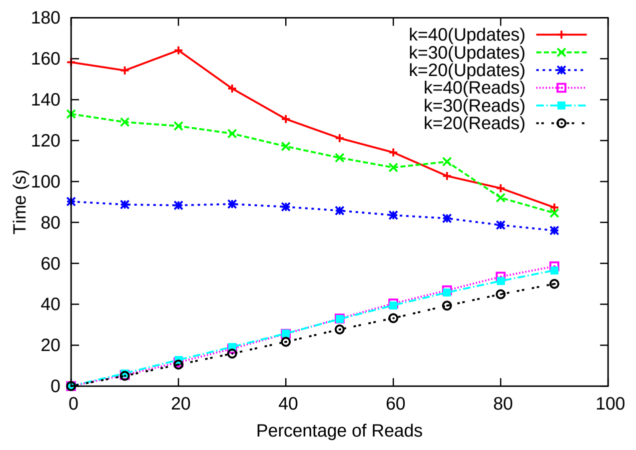

Figure 8: Performance under mixed workload.

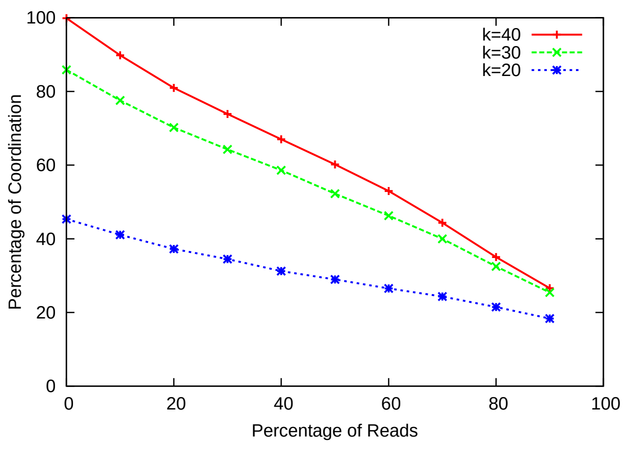

Figure 9: Percentage of coordination w.r.t. percentage of reads.

Quantum DB			Intelligent Social
k=20	k=30	k=40	
45.6	86.9	99.9	20.2

Table 2: Average percentage of successful coordinations

creased.

Figure 7 shows the total time taken for all transactions to complete and Table 2 shows the percentage of successful coordination among the transactions. The percentage of coordination is dependent on the maximum number of pending transactions per flight and as expected remains constant with respect to the number of transactions. The average values of the percentage of coordination for different values of k is shown in Table 2. From Figure 7 it can be seen that with smaller values of k, execution is faster since the body of the composed transaction we must maintain and check satisfiability on has a smaller number of joins. However, the percentage of successful coordination is lower as the system grounds transactions pre-emptively reducing the chances of successful coordination. Even for small values of k, we see a factor of 2 improvement in coordination percentage over the `IS` strategy.

We investigated the unexpected increase in Figure 7 for $k = 40$ at $13,500$ transactions. This increase occurs due to an unexpectedly large amount of time spent in finding the solution to the composed body of a few transactions. Digging deeper, we saw that this is an artifact of the MySQL query optimizer which chooses a bad query plan for several quantum database queries. We observed that such queries occurred rarely and only for certain random orderings. We modified the value of `optimizer_search_depth` to find an alternate plan, and with an alternate plan the problem queries could be answered about two orders of magnitude faster. We believe better query plans would eliminate this erratic behavior. An alternate ad-hoc approach would be to ground some pending transactions to keep the query complexity and the resulting query execution times within acceptable limits. A more fundamental approach to solving this problem is discussed in next section.

Our system scales linearly in terms of execution time and maintains a constant coordination percentage as the number of transactions increases. This linear increase is a consequence of the non-unification based partitioning strategy implemented by the quantum database.

Mixed Workload: Next, we study the behavior of our system under realistic workloads which are a mix of resource and non-resource transactions. The non-resource transactions are read queries by users who had earlier issued a resource transaction. Unlike in normal databases, a non-resource read transaction on a quantum database can induce updates to the database by forcing grounding of pending resource transactions. Such forced grounding, however, reduces the chances of successful coordination.

We run our experiment over a random ordering of 6000 transactions. The database contains 40 flights each with 150 seats (50 rows of 3 seats each). This ensures after all transactions are executed, every user has a seat. We increase the number of read transactions in steps of 10% (600 transactions) from 0% to 90% (5400 transactions), and use values of k ranging from 20 to 40. As shown Figure 9, the percentage of successful coordination decreases linearly with the percentage of reads. This is expected as under a high read workload, the system needs to pre-emptively fix many transactions and thereby prevents coordination. This is corroborated by Figure 8 which shows an increasing amount of time spent on the reads and a decreasing amount of time spent on executing the resource transactions. The decrease in the time spent on resource transactions is because of preemptive grounding of pending transactions on being read. This leads to fewer pending transactions in the system, and correspondingly simpler satisfiability checks.

The anomalous behavior in Figure 8 at 20% reads for $k = 40$ and 70% reads for $k = 30$ can be attributed to the choice of bad query plans by query optimizer as observed and explained in the previous experiment.

6. DISCUSSION

Making quantum databases a practical platform for developing applications involving resource allocation raises exciting research challenges. In this section, we highlight some of these challenges and discuss potential solutions.

Efficiency of evaluation: A core aspect of maintaining a quantum database is checking and maintaining the satisfiability of the composed transaction formula. Implementing the satisfiability check in the naïve way with relational queries is suboptimal in the general case, as the resulting query has a large number of joins and is not well-suited to the capabilities of a traditional relational optimizer. We discuss two potential solutions to overcome this limitation of quantum database.

First, this problem is an instance of the Satisfiability problem, which is known to have phase transitions [16]. Most real-world resource allocation problems are under-constrained at least in the initial phase, e.g., when seats on a given flight first go on sale. As resources are allocated the problem tends to reach a critical ratio of degrees of freedom (variables) vs. constraints at which the problem is hard; both comfortably under- and over-constrained problems tend to be easy to solve [16]. By identifying the difficulty of checking satisfiability, a quantum database can switch to a more aggressive fixing phase favoring faster response times over better assignments.

Second, we believe, we can leverage state-of-the-art Satisfiability Modulo Theory (SMT) solvers [4], which are essentially SAT solvers where the interpretation of some symbols is constrained by a background theory. Identifying the appropriate background theory for quantum databases and designing an algorithm which splits the task of satisfiability checking between the database and the solver to achieve maximum efficiency requires addressing many research problems, both theoretical and systems-related.

System design: Another key issue is the integration of quantum database functionality into the technology stack used by developers. Should quantum databases be implemented within or on top of a DBMS? The latter approach may be simpler and more portable, but the former should yield better performance as well as significant systems insights as we investigate how to integrate quantum database functionality with the internals of a DBMS.

The quantum database model for resource allocation has conceptual connections with existing concurrency control algorithms already implemented in the DBMS to maintain isolation and to perform recovery. Consider isolation first: in a crude sense, a transaction that has committed in a quantum database has a "logical lock" on an instance of the resource and the quantum database system is performing logical lock management. It is this logical locking which allows reordering of logically non-conflicting transactions as explained in Section 3.2.3. While we have presented a practically viable approach for detecting such logical lock conflicts using unification, it will be interesting to understand how the algorithms for maintaining a quantum database can be optimized by offloading parts of it to the underlying database. Conversely, the algorithms for maintaining a quantum database may provide insights for designing intention based concurrency control mechanisms. Regarding recovery, in many ways quantum databases avoid rollbacks by performing resource allocation late and dropping constraints if they cannot be satisfied. The question arises of whether this in any way relates to the work performed by the recovery manager. Even if not, it is possible that the recovery manager should be designed in a different way to better support the unique features of quantum databases.

7. RELATED WORK

Quantum databases are related to probabilistic databases [17] in that both represent sets of possible worlds, although the uncertainty in our case is strictly internal. More generally, the possible worlds semantics plays an important role in certain modal logics, notably logics of knowledge [6].

There is a large body of work on algorithms for resource allocation and more generally constraint processing [5]. Systems such as Tiresias [13] and Cologne [12] integrate constraint satisfaction formalisms and algorithms with database technology, although they address different problems than resource allocation. Also relevant is the work on constraint-based replica reconciliation in the IceCube system [10] and transactional intent [7]. Resource transactions are a clean and simple way of specifying transactional intent, and quantum databases perform similar reconciliation to that done in IceCube, although again in a different application setting.

Triggers or other active database constructs [18] might also be used to encode the check for satisfiability of pending updates. However, they are notoriously complex to design and debug, and they are not suitable for settings requiring transactional guarantees such as atomicity and durability.

Constraint databases [15, 11] have some conceptual similarities to quantum databases, particularly since both classes of systems work with intensional representations of state and/or relations. However, quantum databases do not share the constraint database goal of representing and reasoning about relations that hold over an infinite universe. Conversely, work on constraint databases does not focus on representing and maintaining a set of possible worlds which changes as the database is modified. Nonetheless, we may be able to exploit some techniques for query answering in constraint databases to improve our system; investigating these opportunities is future work.

8. CONCLUSIONS

In this paper, we presented quantum databases, a novel abstraction for declarative resource allocation. We introduced the idea of deferred execution of transactions to improve allocation of resources in a dynamic system. To make this idea practical, we proposed unification based algorithms for efficiently maintaining the database in a partially intensional representation in the presence of queries and other transactions. We evaluated the performance of quantum databases over our prototype implementation in a realistic application scenario, and demonstrated the improvement in allocation of resources due to deferred execution. We believe that research to address the quantum database-related research challenges which we identified will expand the power of and provide insights into multiple aspects of database engines.

9. ACKNOWLEDGEMENTS

Johannes Gehrke is a collaborator on this project but could not be credited as co-author of this paper because of CIDR's one-paper-per-author rule.

We would like to thank the anonymous CIDR reviewers for their insightful comments which helped us to improve this paper.

This research has been supported by the NSF under Grants IIS-1012593, IIS-0911036, by a Google Research Award, by ERC Grant 279804, and by the iAd Project funded by the Research Council of Norway. 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.

10. REFERENCES

[1] http://dev.mysql.com/.

[2] http://www.cs.cornell.edu/bigreddata/youtopia/.

[3] J. Chen, A. Machanavajjhala, and G. Varghese. Scalable social coordination with group constraints using enmeshed queries. In *CIDR*, 2013.

[4] L. De Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, Sept. 2011.

[5] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[6] R. Fagin, J. Y. Halpern, Y. Moses, and M. Vardi. *Reasoning about Knowledge*. MIT Press, 2004.

[7] S. Finkelstein, T. Heinzel, R. Brendle, I. Nassi, and H. Roggenkemper. Transactional intent. In *CIDR*, 2011.

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

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

[10] A.-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of divergent replicas. In *PODC*, Newport, RI, USA, 2001.

[11] G. M. Kuper, L. Libkin, and J. Paredaens, editors. *Constraint Databases*. Springer, 2000.

[12] C. Liu, L. Ren, B. T. Loo, Y. Mao, and P. Basu. Cologne: A declarative distributed constraint optimization platform. *PVLDB*, 5(8):752–763, 2012.

[13] A. Meliou and D. Suciu. Tiresias: the database oracle for how-to queries. In *SIGMOD*, pages 337–348, 2012.

[14] G. Miklau and D. Suciu. A formal analysis of information disclosure in data exchange. *J. Comput. Syst. Sci.*, 73(3):507–534, 2007.

[15] P. Revesz. *Introduction to constraint databases*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[16] B. Selman. Stochastic search and phase transitions: AI meets physics. In *IJCAI (1)*, 1995.

[17] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[18] J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1995.

APPENDIX

A. PROOF OF LEMMA 3.5

Proof. Let Var_1 and Var_2 denote the set of variables in U_1 and U_2, respectively. We assume T_1 and T_2 have no shared variables, i.e. $Var_1 \cap Var_2 = \emptyset$. Under the range restriction requirement for resource transactions, Var_1 and Var_2 are subsets of the variables in B_1 and B_2, respectively. In the proof below, we use the standard notion of valuation – a valuation is a map from variables to values in the database.

The proof is by cases depending on whether U_1 is an insert or a delete.

Case I : U_1 is a delete

Consider the database state transitions $D \xrightarrow{T_1} D' \xrightarrow{T_2} D''$, where the superscript above each arrow denotes the transaction operating on the database during the transition. Let ν_1 and ν_2 denote the valuations for Var_1 and Var_2 that lead to the database states D' and D'', respectively. We show that executing T_{12} on D under the same valuations for the respective variables leads to the same D''.

Using our assumption that each relation in D has a key, we claim that $\nu_1 U_1 \neq \nu_2 B_2$, that is, the relational atom deleted by U_1 is not the same as the relational atom that B_2 grounds on. Suppose this claim is false: then it must be that $\nu_2 B_2 \in D$ as the only difference between D and D' is the deletion of the tuple $\nu_1 U_1$. However, this means D contained two instances of the same tuple (equal to both $\nu_1 U_1$ and $\nu_2 B_2$), which is impossible by our assumption that each relation has a key.

Establishing that $\nu_1 U_1 \neq \nu_2 B_2$ tells us two things. First, by definition of φ, $\nu_1\nu_2\varphi(U_1, B_2) = false$. This implies that $B = B_1 \wedge B_2$. Second, we know that $\nu_2 B_2 \in D$. Therefore, $\nu_1 \cup \nu_2$ is a valid valuation for B, and under this valuation the execution of T_{12} on D leads to D''.

Now let us consider the other direction, i.e. suppose $D \xrightarrow{T_{12}} D''$ under valuation ν over $Var_1 \cup Var_2$. Let ν_1 and ν_2 be the projections of ν on Var_1 and Var_2. This implies that both $\nu_1 B_1$ and $\nu_2 B_2$ are in D. As $\neg\nu_1\nu_2\varphi(U_1, B_2)$ is true, $\nu_1\nu_2\varphi(U_1, B_2)$ must be false. This, by definition of $\varphi(U_1, B_2)$, implies $\nu_1 U_1 \neq \nu_2 B_2$. It follows that $\nu_1 B_1$ is a valid valuation for B_1 on D, and $\nu_2 B_2$ is a valid valuation for B_2 on D', where $D \xrightarrow{\nu_1 U_1} D'$ and $D' \xrightarrow{\nu_2 U_2} D''$.

Case II : U_1 is an insert

The reasoning here is very similar to the first case, so the proof is omitted. □