SqlIceCube: Automatic Semantics-based Reconciliation for Mobile Databases

Nuno Preguiça*, Marc Shapiro#
J. Legatheaux Martins*

Technical report 2-2003 DI-FCT-UNL

Microsoft Research, Ltd.
Cambirdge, UK

* Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Quinta da Torre, 2829-516 Caparica, Portugal

SqlIceCube: Automatic Semantics-based Reconciliation for Mobile Databases*

Nuno Preguiça*, Marc Shapiro+, J.Legatheaux Martins*

*Dep. Informática, Univ. Nova de Lisboa, Portugal
{nmp,jalm}@di.fct.unl.pt

+Microsoft Research Ltd., Cambridge, UK
Marc.Shapiro@acm.org

Abstract

In a distributed system, optimistic replication enables users on different sites to query and update their local replicas of shared databases without a priori synchronization. Replicas may diverge, and updates must be reconciled; reconciliation is a difficult problem in the presence of conflicts, alternative execution paths, and dependencies between transactions. We present SqlIceCube, a general-purpose reconciliation system. SqlIceCube automatically extracts significant semantic relations from the program text of transactions. Examples of relations are a transaction that depends on another, helps another, hinders another, constitutes an alternative to another, etc. In turn these semantic relations feed into the SqlIceCube scheduler, which generates and executes a combination of transactions with the highest possible value. The approach is general and supports combinations of transactions across a variety of different applications. Application experience and benchmarks show the viability of the approach. SqlIceCube correctly extracts semantic relations from an interesting variety of application transactions, performs well, and scales well to large input sizes.

1 Introduction

Distributed systems replicate shared data to improve read availability and performance. Optimistic replication allows a user to *update* a local replica independently. This improves write availability in the presence of high network latencies, failures, voluntary disconnection, or parallel development, but allows replicas to diverge. This is especially useful for mobile computing environments.

Repairing divergence after the fact, called *reconciliation*, combines the isolated updates [28]. In operation-based (or log-based) approaches, update operations are recorded in a log; reconciliation *replays* the combined operations, from the initial state, according to some *schedule*.

*This work was partially supported by FCT/MCT. Nuno Preguiça was partially supported by a FSE schorlarship.

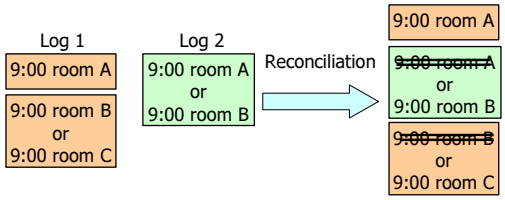

Figure 1: Syntactic scheduling spuriously fails on this example

Sometimes, the execution of some updates in a given database state would violate a *precondition* or an *invariant*. In this case, relying on conflict resolution rules (e.g. exploring alternative updates) and running updates in a different order may allow better reconciliation results (i.e., more updates can be executed). The updates that cannot be executed must be *dropped*.

Reconciliation must minimize the number of dropped updates, as dropping an update may have a high impact. Think of a calendar system dropping an important meeting, or a sales-support tool for a salesman that would drop an order.

Log-based reconciliation systems (e.g. Bayou [30] and Deno [15]) usually use a simple syntactic criterion to decide in which order updates should be run. This is inflexible and may cause spurious conflicts even in very small problems. Consider the example of figure 1, where two users make meeting requests to a calendar program. One user requests room A at 9:00, and either room B or C, also at 9:00. Meanwhile, other user requests either room A or B at 9:00. Combining the logs in some simple way does not work. For instance running Log 1 then Log 2 will reserve rooms A and B for the first user, and the second user's requests is dropped. Running Log 2 first has a similar problem. Satisfying all three requests requires reordering them. Thus, reconciliation systems must be prepared to reorder updates.

Some reconciliation systems reorder operation execution for improving concurrency [29] or reconciliation results [21]. In these systems, programmers must express the semantic information needed by the reconciliation engine. The intellectual complexity of expressing this information is usually non-trivial and programmers will not be willing to execute this extra work or will end up creating bad semantic rules that lead to poor reconciliation results. Moreover, this approach makes it difficult to introduce new operations, as all semantic rules must be extended to include the new operation. Although this approach may be acceptable in some settings (e.g. specific applications), it imposes unusual restrictions in generic database systems. In contrast, the system presented in this report automatically infers the semantic information needed for reconciliation.

1.1 Our approach

This report presents the SqlIceCube general-purpose reconciliation system for mobile database systems. This work builds on our previous experience on the devel-

```
---- ORDER PRODUCT: id = 80; quantity = 10; highest price = 100.00
DECLARE
  l_stock INTEGER;
  l_price FLOAT;
BEGIN
  SELECT stock,price INTO l_stock,l_price FROM products WHERE id=80;
                -- check price acceptability and stock availability
  IF l_price <= 100.0 AND l_stock >= 10 THEN
    UPDATE products SET stock = stock - 10 WHERE id = 80;
                -- newid returns the same unique identifier
                -- in the client and in the server
    INSERT INTO orders VALUES (newid,8785,80,10,l_price,'to process');
    COMMIT;      -- commit transaction (and return)
  ELSE
    ROLLBACK;    -- abort transaction (and return)
  ENDIF;
END;
```

Figure 2: Mobile transaction adding a new order in a mobile sales application.

opment of the IceCubegeneric reconciler [26].

In SqlIceCube, update operations are expressed as small programs written in a subset of PL/SQL [22]. These programs, dubbed mobile transactions (or simply transactions), are submitted by users' applications to modify the database. Figure 2 presents a simple mobile transaction to submit a new order in a mobile sales application. The code of the transaction verifies if the stock is sufficient and the price is acceptable before adding the new order to the database.

SqlIceCube is a generic reconciliation engine that combines concurrently executed mobile transactions into a single execution schedule. While disconnected, mobile transactions may be tentatively executed against local replicas. During reconciliation, the mobile transactions' programs are executed again, i.e., the transactions are replayed logically (as opposed to redoing physical writes).

The SqlIceCube generic reconciler can be integrated in different mobile database systems to reconcile transactions executed concurrently in disconnected devices. In section 7, we discuss the integration of SqlIceCube as the reconciliation system in several systems.

Reconciliation is executed in two steps: the semantic inference and the basic reconciliation.

The semantic inference module automatically infers the semantic information needed by the basic reconciler. Each transaction is statically analyzed to extract relevant semantic information (read/written data items and preconditions). Then, for each pair of transactions, a set of semantic static relations (e.g. overlap) is inferred using the extracted information.

The basic reconciler tries to find a schedule that allows more transactions to be executed. To this end, it performs an heuristic search within the space of solutions that respect the inferred semantic relations. Mobile transactions being reconciled may combine updates from different applications and even for different databases. As compared with similar approaches [26, 9], our reconciler presents a subtle but

important difference: reconciliation is seen as a planning problem instead of a constraint solving problem. As we discuss in section 2.2.2, this approach allows to overcome some problems with compensating transactions experienced in constraint solving-based approaches.

This work presents the following contributions. First, it identifies a set of static relations that can be used to expose semantic information relevant for reconciliation problems. This information can expose not only the semantics of the data types and operations but also the users' intents.

Second, it presents a generic semantics-based reconciliation engine that creates near-optimal reconciliation results. SqlIceCube reconciles consistently and seamlessly between transactions involving different applications and databases. Benchmarks presented in this report show that SqlIceCube reconciles in reasonable time and scales nicely to large logs.

Third, it includes a semantic inference module that simplifies the use of semantics-based reconciliation. Unlike previous systems, semantic information used during reconciliation is automatically extracted from operations, thus alleviating programmers from most of the work usually involved in semantics-based reconciliation. This approach also enables new operations and applications to be defined without need to modify the reconciliation system.

The remainder of this report is organized as follows. Section 2 presents our general-purpose semantics-based reconciler. Section 3 describes the techniques used in the basic semantic inference module and section 4 discusses some extensions to these techniques. Section 5 describes the status of our prototype. Section 6 presents an evaluation of SqlIceCube performance and solutions' quality. Section 7 discusses the deployment of SqlIceCube in different systems and architectures. Section 8 presents related work and section 9 concludes the report with some final remarks.

2 Generic semantics-based reconciler

In this section we present the generic SqlIceCube reconciler. This reconciler creates and executes near-optimal schedules that combine transactions submitted concurrently in multiple mobile clients. The schedules are created incrementally, selecting, at each step, the mobile transaction that may lead to a better reconciliation result. The selection algorithm is heuristic, based on the semantic static relations among transactions. The reconciler may create a set of alternative schedules, heuristically probing the space of all possible schedules.

We start by describing the semantic information used. Then, we present the basic heuristic reconciliation engine.

2.1 Dynamic constraints/pre-conditions

Dynamic constraints are restrictions to the execution of mobile transactions that depend on the database state. Therefore, they can only be verified in runtime. These restrictions are specified in the code of transactions by preconditions and commit and rollback statements. For example, the transaction of figure 2 can only commit if the stock and price for the specified product meet the given conditions.

Dynamic constraints are not used directly by the reconciliation algorithm, as evaluating the expressed conditions after executing each transaction would impose an excessive performance cost because it requires accessing the database. Instead, preconditions are used to infer suitable static relations, as explained next.

2.2 Static relations

Static relations are relations among mobile transactions that do not depend on the database state. Therefore, they are can be used during the reconciliation process without accessing to the database. Two types of static relations are used in our reconciler: log relations and data relations.

2.2.1 Static log relations

Log relations are relations that express users' intents. These relations are independent of the semantics of each specific basic transaction. Instead, they encode the semantics of a macro-operation composed of several basic transactions. Usually they are explicitly set by applications but some of them can also be automatically inferred as discussed later. The following log relations are defined:

Alternatives Specify that a single transaction must be committed from a set of alternatives. This allows the definition of basic conflict resolution rules. For example, when scheduling a meeting, a user can specify two or more alternative meeting rooms or dates.

Strong predecessor-successor The successor transaction can only be executed after (and if) the predecessor transaction commits. This allows to establish a causal order between transactions. For example, a transaction that submits requests related to a given meeting (e.g. food for the coffee break) should only be executed after committing the transaction that schedules the meeting.

Weak predecessor-successor If both transactions commit, the predecessor must be executed before the successor. This allows to establish a weak causal order between transactions. For example, a transaction that schedules a new appointment for some day can only be executed before the execution of a transaction that cancels all existent appointments for that day.

Parcel Defines an all-or-nothing group of transactions. Unlike the operations in a transaction, the execution of the basic transactions that compose a parcel may be interleaved with the execution of other transactions. The isolation property of transaction is not guaranteed by the execution of a parcel (although the execution of each basic transaction respects isolation). For example, a bank transfer can be defined as a parcel with two transactions, one withdrawing money from the source account and the other depositing money in the destination account.

The alternatives and weak predecessor-successor relations can be combined to create a set of ordered alternatives. The alternatives relation guarantees that only one transaction is executed. The weak-predecessor relation guarantees that the preferred transaction is executed, if possible (as detailed later, the reconciliation engine favors the executions of transactions that have no predecessor).

2.2.2 Static data relations

Data relations are relations between transactions that encode the semantics of the operations. These relations are defined independently of the database state. The following static data relations are used:

Commute Two transactions commute if the result of executing both is independent of the execution order, i.e., given two transactions t_1 and t_2, $\forall s \in \mathcal{S}, t_1(t_2(s)) = t_2(t_1(s))$, with $\mathcal{S}$ the set of all possible database states. For example, two transactions that modify unrelated data items commute.

Helps The transaction t_1 *helps* the transaction t_2 if the commitment of t_1 changes the database state in a favorable way for the success of t_2, i.e., $\exists s \in S : \neg valid(t_2, s) \wedge valid(t_2, t_1(s))$, with $valid(t, s)$ true if t can commit in database state s. For example, a transaction that increases the stock of some product improves the chance of accepting a new order request (with the available stock).

Prejudices The transaction t_1 *prejudices* the transaction t_2 if the commitment of t_1 changes the database state in a prejudicial way for the success of t_2, i.e., $\exists s \in S : valid(t_2, s) \wedge \neg valid(t_2, t_1(s))$. For example, a transaction that decreases the stock of some product hinders the chance of accepting a new order request.

Makes possible The transaction t_1 *makes possible* the transaction t_2 if the commitment of t_1 changes the database state in a way that makes possible to commit t_2, i.e., $\forall s \in S, valid(t_2, t_1(s))$. For example, removing all meetings in a given room makes possible to schedule a new meeting for that room.

Makes impossible The transaction t_1 *makes impossible* the transaction t_2 if the commitment of t_1 changes the database state in a way that makes impossible

to commit t_2, i.e., $\forall s \in S, \neg valid(t_2, t_1(s))$. For example, setting the price of some product to a value that violates the preconditions of an order request makes impossible to accept this requests.

The static relations *helps*, *prejudices*, *makes possible* and *makes impossible* encode the influence of one transaction over the dynamic constraints (preconditions) of the other transaction. Thus, they suggest a preferable execution order between pairs of transactions.

This approach represents a subtle but important difference when compared with the IceCube reconciler [26] and the Fages' constraint programming approach to reconciliation [9]. In these approaches, constraints between transactions (actions in their terminology) represent definite order and incompatibility relations that must enforced by the reconciler. In our approach, an incompatibility relation between two transactions can be cancelled by a third transactions, as we would expect.

For example, if two transactions are declared mutually exclusive they cannot be both scheduled, even if a third transaction makes possible the execution of both transactions. Consider two transactions that schedule the same meeting room for the same day. As these transaction would be considered mutually exclusive one of them would be dropped. However, if a third transaction cancels one of the scheduling requests, it is possible to execute all transactions. To prevent these problems, IceCube includes a log cleaning mechanism that removes pairs of transactions that compensate each other. In practice, it is hard to find pairs of perfectly compensating transactions, as they tend to leave a record in the system (e.g. when a meeting is cancelled, the systems often keeps a record with the information about the cancelled meeting).

2.3 Heuristic reconciliation engine

The goal of the reconciler is to create a schedule that maximizes the value[1] of transactions that can be executed with success. As, for non-trivial problems, it is impossible to check all possible schedules, we have implemented a reconciliation algorithm that probes the space of possible solutions heuristically. To this end, the reconciler creates and executes a sequence of schedules that combine the transactions submitted concurrently by multiple mobile clients. The reconciliation process ends when a schedule that meets a specified optimality condition is generated, or the specified search time is exhausted.

Merit of a transaction: The heuristic used in the SqlIceCube reconciliation engine is based on the merit of transactions. The merit heuristically computes the benefit of adding a given transaction to a given partial schedule. The reconciler creates a schedule incrementally by iteratively selecting a transaction to add to a partially created schedule.

[1] By default, every transaction has value 1, but applications can assign different values to a transaction depending on its importance.

An oracle estimates the merit of each transaction based on the static relations established between this transaction and all other transactions that can still be scheduled. The merit of transaction a is higher as:

1. It belongs to a parcel that includes a transaction already executed.
2. The value of (weak) predecessors of a, which would be aborted if a is selected, is lower.
3. The value of alternatives to a is lower, as it is more likely to be possible to commit one transaction with more alternatives.
4. The value of transactions a makes impossible is lower, as those transactions cannot be committed, at least temporarily; the value of transactions a makes possible is higher.
5. The value of (weak and strong) successors of a is higher. as successors are made possible by the execution of the predecessor.
6. The value of transactions a helps is higher; the number of transactions a prejudices is lower.

The above factors are listed in decreasing order of importance. Despite the previous rules, the merit of transaction a is zero if a has a strong predecessor that has not been executed yet or a has an alternative that has already been executed.

Create a single schedule: Figure 3 displays in pseudo-code the basic algorithm to create a single schedule. The algorithm maintains the set of candidate transactions that can be scheduled, `trxs`, and the set of transactions that are known to be temporarily impossible (i.e., the transactions that can not be executed in the current database state), `badTrxs`. Furthermore, it maintains a summary of relations established among transactions.

The algorithm creates each schedule incrementally as follows. At each step the algorithm selects (with randomization) one transaction among the candidates with highest merit (`selectTrxByMerit`). The selected transaction is executed against the current database state.

If the transaction executes successfully (i.e. it ends in a commit statement), it is added to the current partial schedule and the value of the schedule is updated. All transactions in the `badTrxs` set that are helped (or made possible) by this transaction are removed from `badTrxs` (as it might be possible to execute them again). All transactions that are temporarily made impossible by the execution of this transaction are added to `badTrxs`. All transactions that are definitely made impossible by this transaction (weak predecessors and alternatives) are removed from the set of transaction to schedule, `trxs`. The executed transaction is also removed from `trxs`.

```
scheduleOne (state, summary, goodTrxs) =   // pseudo-code
   schedule := []
   value := 0
   trxs := goodTrxs     // trxs to schedule
   badTrxs = {}         // trxs known to be impossible (temporarily)
   WHILE trxs <> badTrxs DO
                                // select next trx to execute
     nextTrx := selectTrxByMerit( trxs, badTrxs, schedule, summary)
     result = state.execTrx( nextTrx)
     IF result = COMMIT THEN     // trx has committed
       trxs = trxs \ {nextTrx}
                // trxs that are definitely made impossible
                // (alternatives, weak predecessors)
       toExclude = incompatible( nextTrx, trxs)
       trxs = trxs \ toExclude
                // trxs that are helped by the given trx
                // (helps, makesPossible)
       toInclude = helps( nextTrx, badTrxs)
       badTrxs = badTrxs \ toInclude
                // trxs that are temporarily known to be impossible
                // (makesImpossible)
       toTempExclude = makesImpossible( nextTrx, trxs)
       badTrxs = UNION( badTrxs, toTempExclude)
       summary.updateInfo( trxs, nextTrx, toExclude,
                           toInclude, toTempExclude)
       schedule := [schedule | nextTrx]
       value := value + nextTrx.value
     ELSE                        // trx has rollbacked
       badTrxs = UNION( badTrxs, {nextTrx})
       summary.updateInfoBadTrx( trxs, nextTrx)
     ENDIF
  ENDWHILE
  badParcels = incompleteParcels(schedule)
  IF NOT EMPTY( badParcels) THEN
    SIGNAL DynamicFailure( badParcels)
  ELSE
    RETURN { schedule, value }
```

Figure 3: Algorithm to create a single schedule.

If the transaction execution fails (i.e. it ends in a rollback statement), the transaction is added to the set of transaction that are temporarily known to be impossible.

In both cases, the summary of relations among transactions is updated.

This incremental process proceeds while there are transactions to schedule that are not known to be impossible. In the end, the algorithm verifies parcel integrity. If all parcel relations are respected, the created schedule is valid and is returned. Otherwise, the basic algorithm signals an error indicating the set of transactions that caused the problem.

The `scheduleOne` algorithm is called successively until it creates a schedule that meets a specified optimality condition (by default, until all transactions but alternatives of one executed transaction are executed) or the specified search time is exhausted. To this end, the reconciliation algorithm needs to create a checkpoint of the initial database state. Before calling the `scheduleOne` function, the initial

database state is restored using this checkpoint.

When the `scheduleOne` algorithm fails returns an error, no schedule is created. To prevent this situation from occurring every time `scheduleOne` is called, the reconciliation algorithm removes the transactions that caused the problem from the set of transactions to reconcile (parameter `goodTrxs`) until a good schedule is found.

Complexity: The cost of of `scheduleOne` is dominated by `selectTrxByMerit`. The overall complexity is $O(n^2)$, where n is the number of transactions to reconcile (size of `goodTrxs`). Intuitively, order n transactions are scheduled, and for each one `selectTrxByMerit` considers all the remaining transactions $(O(n))$[2].

The merit of a transaction evaluates in constant time thanks to a summary of static relations. This summary is computed at the beginning of the reconciliation process and its complexity is $O(n^2)$ because it compares every transaction with every other. During the execution of `scheduleOne`, this summary is updated in expected constant time because each transaction is expected to establish relations only with a small number of other transactions. In the worst case, when a transaction establishes relations with $O(n)$ transactions, the summary is updated in $O(n)$. In this case, the overall complexity of `scheduleOne` continues to be $O(n^2)$, but this cost is dominated equally by `selectTrxByMerit` and the update of summaries.

Assuming that only a small number of schedules is created, the overall time complexity of the reconciliation algorithm is $O(n^2)$. Furthermore, assuming that `scheduleOne` only tries to execute each transaction a small number of times, the reconciliation algorithm executes $O(n)$ transactions, i.e., $O(n)$ accesses to the database.

The space complexity of the reconciliation algorithm is $O(n^2)$ in the worst case because, for each transaction, it is kept the list of other transactions that establish relations with this transaction ($O(n)$ maximum). Additionally, it is necessary to keep a checkpoint of the database (or use nested transactions to commit or rollback a complete schedule).

Clustering: As the complexity of reconciling a set of n transactions is $O(n^2)$, the overall complexity of the reconciliation algorithm can be improved if transactions can be partitioned in independent subsets with a small number of transactions. To this end, our reconciler partitions transactions into disjoint subsets called clusters, thus dividing the reconciliation problem into smaller independent sub-problems. SqlIceCube reconciles each cluster independently in an arbitrary sequential order.

Two transactions are independent if they are not linked by any log relations and commute. This ensures that transactions from different clusters may be scheduled in arbitrary order, and that the decision to include or drop a transaction from one cluster does not affect any other cluster.

[2]The complexity of `scheduleOne` can be made $O(n \log n)$ by restricting the merit function and by using priority queues to maintain merits

Clustering executes in two stages. First, transactions are partitioned by domain using keys for each transaction (e.g. in a transaction that accesses a calendar, the days read and written can be the keys), with a complexity linear in the number of transactions. Then, a subset is re-partitioned according to the commute relation, for a complexity quadratic in subset size.

In the expected case, clustering executes in $O(n)$ time. The overall complexity of reconciliation is $O(n)$ if clustering partitions transactions in $O(n)$ clusters with a small number of transaction each. If there is one large cluster with $O(n)$ transactions, the overall complexity remains $O(n^2)$.

In the worst case, clustering executes in $O(n^2)$. This happens when the first stage of reconciliation creates one cluster with $O(n)$ transactions.

3 Automatic extraction of (data) relations

The semantic static relations established among transactions are used by the SqlIceCube reconciler to direct the search. The common approach in semantics-based reconciliation systems is to require the definition of methods (e.g. IceCube [26]) or tables (e.g. Sync [21]) to specify semantic information between operations. This approach has two drawbacks. First, even when it is simple to specify each rule, it tends to be a repetitive, verbose and error-prone work. Second, it makes difficult to introduce new operations, as it is necessary to extend the specified rules. In a relational database system where generic mobile transactions may be submitted, these drawbacks make it impossible to use a similar approach.

In SqlIceCube we have designed a mechanism to automatically infer the data relations between mobile transactions (log relations are naturally added by applications when they submit complex operations composed by several transactions). To this end, the code of mobile transactions is statically analyzed in order to extract the information used to infer the relations between transactions. In this section we detail the basic inference mechanism. In the next section we describe extensions to handle more complex mobile transactions.

3.1 Extract information

To extract information from a mobile transaction, the transaction program is statically analyzed checking all execution paths. For each path that ends in a commit statement, the following group of information is extracted:

Semantic read set The *semantic read set* contains the semantic description of each relevant *read data item* (if the read value is not used, it is not relevant). This information is obtained from select statements.

Semantic write set The *semantic write set* contains the semantic description of each *written data item*. This information is obtained from insert, update and delete statements.

```
1  BEGIN
2    SELECT stock,price INTO l_stock,l_price FROM products
3                       WHERE id = 80;
4          -- l_stock = read(products,id=80,stock)
5          -- l_price = read(products,id=80,price)
6    IF l_price <= 100 AND l_stock >= 10 THEN
7          -- as the contrary leads to a rollback
8          -- precond(read(products,id=80,stock)>=10 AND
9          --          read(products,id=80,price)<=100)
10     UPDATE products SET stock = stock - 10 WHERE id = 80;
11         -- update(products,id=80,stock,stock-10)
12     INSERT INTO orders
13            VALUES (newid,8785,80,10,l_price,'to process');
14         -- insert(orders,(id,client,product,qty,price,status),
15         --                (647,8785,80,10,l_price,'to process'))
16     COMMIT;
17   ELSE
18     ROLLBACK;
19   ENDIF;
20 END;
```

Figure 4: Information extracted from a mobile transaction that adds an order in a mobile sales application.

Precondition set The *precondition set* contains all conditions in the given execution path. This information is obtained from if instructions (loops are discussed later).

The semantic description of a data item includes the name of the table, the name of the column and the condition used to refer the data item.

Example: : Figure 4 shows, as comments (lines starting with --), the information that can be extracted from the simple mobile transaction that adds a new order in a mobile sales application. The select statement at line 2 associates the semantic description of the read data item with the given variables.

During static analysis, both values should be considered for conditions specified in if instructions. In this example, as considering the condition false leads to a rollback, there is no need to consider this possible execution path. Considering the condition true leads to a commit statements. Therefore, the given condition is a precondition to the success of the transaction: the correspondent semantic precondition is added to the precondition set.

The semantic descriptions of written data items are directly extracted from the update and insert statements at lines 10 and 12-13. These descriptions compose the semantic write set for the transaction. The semantic read set is composed by all semantic descriptions of read data items that are used in the preconditions or write statements. In this case, as data item read at line 2 are used in the precondition at line 6, the semantic read set is composed by the description of those read data items. From this mobile transaction, a single set of static information is extracted as there is only one execution path leading to a commit statement.

3.2 Infer relations

The relations between each pair of transactions are inferred *comparing* the semantic information extracted from each transaction. As this information only contains the semantic description of each data item, it is necessary to verify if the conditions expressed in the different semantic descriptions refer the same data items or not. This verification includes not only the data items read or written but also the data items used to select them (these data items are indirectly read). In database systems, similar analysis are executed in the context of query optimization [13] and semantic caching [6].

Given two transactions t_1 and t_2, each one with a single group of information, the following rules are used to infer each data relation.

Commute (default:false) t_1 does not commute with t_2 if t_1 reads a data item written by t_2 (or vice-versa), or t_1 writes a data item written by t_2 (or vice-versa) unless all t_1 writes commute with all t_2 writes (and vice-versa).

Helps (default:true) t_1 helps t_2 if t_1 writes changes the database in a favorable way for t_2 preconditions to be true.

Prejudices (default:true) t_1 prejudices t_2 if t_1 writes changes the database in a prejudicial way for t_2 preconditions to be true.

Makes possible (default:false) t_1 makes possible t_2 if t_1 writes makes t_2 preconditions true.

Makes impossible (default:false) t_1 makes impossible t_2 if t_1 writes makes t_2 preconditions false.

When there is more than one possible execution path in each program (and several groups of information are obtained from each transaction), it is necessary to analyze the different possible combinations. The default value is assumed if different results are obtained from different combinations.

Sometimes it is impossible to determine exactly if two semantic descriptions refer the same data item or not. For example, two select statements that use conditions over different columns to select the records to read in the same table cannot be precisely compared. In this case, if the comparison's result is important to evaluate if some relation is established between two transactions, the default value for the relation is assumed.

The default values, presented in parenthesis before, were chosen guarantee the safety of the reconciliation algorithm. To this end, the *helps* relation is set to true, thus guaranteeing that a transaction is removed from the set of bad transactions. The *prejudices* relation is also set to true to compensate the effect of a possibly incorrect *helps* relation in the merit of a transaction. The *makes impossible* relation is set to false to guarantee that no transaction is moved to the $badTrx$ set unless

```
BEGIN
  SELECT status INTO l_status FROM orders WHERE id = 3;
        -- l_status = read(orders,id=3,status)
  IF l_status = 'to process' THEN
        -- precond(read(orders,id=3,status)='to process')
    UPDATE orders SET status = 'cancelled' WHERE id = 3;
           -- update(orders,id=3,status,'cancelled')
    UPDATE products SET stock = stock + 30 WHERE id = 80;
           -- update((products,id=80),stock,stock+30)
    COMMIT;
  ELSE
    ROLLBACK;
  ENDIF;
END;
```

Figure 5: Information extracted from a mobile transaction that cancels an order in a mobile sales application.

it is known that the execution of some transaction makes the preconditions of the other transaction false.

The value of the *commute* relation is conservatively set to false to guarantee that transactions that may influence each other are clustered together. If the running time of reconciliation is critical, this value can be assumed true to improve the clustering results. In this case, it may be impossible to achieve the best reconciliation result as it is impossible to create some relevant schedules.

3.3 Examples

Mobile sales application: In the first example, we consider two types of mobile transactions submitted in the context of a mobile sales application. The first, presented in figure 4, adds a new order. The second, cancels an order if it has not been processed yet, as shown in figure 5.

The information that can be extracted from each mobile transaction is shown, as comments, in the figures. Let a be an add transaction and c be a cancel transaction acting on the same product. The system infers the following relations. a and c do not commute, as a reads the stock of product with $id = 80$ and c writes (updates) it. c helps a, as adding a positive value to the stock helps the stock to be bigger than some constant. No other relations exist. The inferred relations are the expected ones, leading the reconciler to execute the cancel transaction before executing the add transaction, thus improving the chances of committing both transactions.

As expected, two mobile transactions adding a new order for the same product do not commute and each one prejudices the other, as subtracting a positive value from the stock reduces the chances of the stock being larger than some constant.

Two mobile transactions cancelling the same order do not commute and each one makes impossible the other, as setting the order state to *cancelled* makes the precondition expressed in the transaction false. This is the expected behavior as an

```
--- RESERVES ROOM 'Ballroom A' FOR DAY '16-FEB-2002'
BEGIN
  SELECT count(*) INTO cnt FROM datebook
                  WHERE day='16-FEB-2002' AND room='Ballroom A';
        -- cnt = read(datebook,day='16-FEB-2002' AND
        --                     room='Ballroom A',count(*))
  IF (cnt = 0) THEN
        -- precond(read(datebook,day='16-FEB-2002' AND
        --                       room='Ballroom A',count(*))=0)
    INSERT INTO datebook
           VALUES( '16-FEB-2002', 'Ballroom A', 'Demo BLUE THING');
        -- insert(datebook,(day,room,info),
        --           '16-FEB-2002','Ballroom A','Demo BLUE THING'))
    COMMIT;
  ENDIF;
  ROLLBACK;
END;
```

Figure 6: Information extracted from a mobile transaction that inserts an appointment in a shared calendar.

```
--REMOVE RESERVATION
BEGIN
  SELECT count(*) INTO cnt FROM datebook WHERE day='16-FEB-2002' AND
                      room='Ballroom A' AND info='Demo BLUE THING';
      -- cnt = read(datebook,day='16-FEB-2002' AND
      --     room='Ballroom A' AND info='Demo BLUE THING',count(*))
  IF (cnt > 0) THEN
      -- precond(read(datebook,info='Demo BLUE THING' AND
      --     day='16-FEB-2002' AND room='Ballroom A',count(*)) > 0)
    DELETE FROM datebook WHERE day='16-FEB-2002' AND room='Ballroom A';
      -- delete(datebook,day='16-FEB-2002' AND room='Ballroom A')
    COMMIT;
  ENDIF;
  ROLLBACK;
END;
```

Figure 7: Information extracted from a mobile transaction that cancels an appointment in a shared calendar.

order can only be cancelled once.

Two mobile transactions acting on different products and orders commute because they do not access the same data items.

Shared calendar: In the second example, we consider typical transactions submitted in the context of a shared calendar used to maintain meeting room reservations. In the transaction presented in figure 6, a new reservation for a room is added if possible. In the transaction presented in figure 7, an existing reservation is removed.

The systems infers the following relations. Any two transactions that refer non-overlapping reservations commute as they access different data items. Transactions with overlapping transactions do not commute. For these transactions, the following additional relations are inferred.

For two transactions that insert a new reservation, each one makes impossible (and prejudices) the other, as the inserted record leads the precondition to be false.

For two transactions that cancel the same reservation, each one makes impossible (and prejudices) the other, as the delete statement guarantees that no record exists that satisfy the specified precondition.

For a transaction, *a*, that inserts a new reservation and a transaction, *c*, that cancels a reservation, two cases should be considered. If both refer to the same reservation (i.e. the value of the info field is equal), *a* makes possible (and helps) *c* because the inserted record guarantees that the reservation exists (and the precondition of *c* is true). Furthermore, *c* also makes possible (and helps) *a* because removing the reservation guarantees that it is possible to schedule a new reservation for that time and place. The reason for this lies in the fact that it might exist or not an identical reservation in the initial database. In this case, the static relations do not help the reconciler to decide which transaction should be executed first. Assume, without loss of generality, that *c* should be executed before *a* and the reconciler selects *a* first. In this case, *a* fails and it is moved to the set of bad transactions. However, after executing *c*, *a* is removed from the set of bad transactions and it will be selected in one of the following steps. Thus, both transactions are successfully executed.

If *a* and *c* refer to different reservations (i.e. the value of the info field is different), *c* makes possible (and helps) *a* as removing the reservation guarantees that it is possible to schedule a new reservation for that time and place. However, unlike before, *a* does not makes possible (or helps) *c* because the inserted record does not satisfy the condition of the select statement — the info field is different. Thus, as expected, the reconciler will cancel appointments before trying to schedule new appointments for the same time and place.

4 Extensions

In this section we further detail how mobile transactions are processed to infer static relations and discuss some extensions to the basic approach described so far.

4.1 Comparison of semantic descriptions

Sometimes it is impossible to compare precisely two semantic descriptions because they include conditions over different columns of the same table or indirections (i.e., they use values read in previous *select* statements). Consider the transaction that cancels an order, shown in figure 8. In this transaction, the update statement at line 5, depends on the values of the order being cancelled (read at line 2). Thus, it is impossible to directly verify if this transaction commutes with a transaction that inserts an order because the product involved in the cancelled order is not known. Furthermore, even if the product is known, it is impossible to know if this transaction helps or prejudices a transaction that inserts an order because it is not

```
1  BEGIN
2    SELECT status INTO l_status,l_prd,l_qty FROM orders WHERE id = 3;
3    IF l_status = 'to process' THEN
4      UPDATE orders SET status = 'cancelled' WHERE id = 3;
5      UPDATE products SET stock = stock + l_qty WHERE id = l_prd;
6      COMMIT;
7    ELSE
8      ROLLBACK;
9    ENDIF;
10 END;
```

Figure 8: Mobile transaction that cancels an order in a mobile sales application.

known if the value of `l_qty` if positive or negative.

As the details of an order are expected to be constant during reconciliation (i.e., none of the transactions that are under reconciliation will modify the order details), this problem can be solved reading the details of the order from the database and using them as constants during the reconciliation process. Thus, it is possible to obtain as much information from the transaction of figure 8 as from the transaction of figure 5. As statements that modify a record usually specify its key, it tends to be easy to verify if a record can be modified by any transaction.

An identical approach can be used to compare semantic descriptions that use conditions over different columns by reading the extra information needed. For example, in the shared calendar example presented previously, each appointment could be identified by an unique identifier. In this case, the transaction that cancels an appointment would use this identifier to verify if the appointment exists and to remove it (unlike the transaction presented in figure 7 that uses the appointment details). However, as the appointment details are constant, we can read these details and use them as constants to obtain as much information as in the transaction of figure 7. Thus, the systems can infer the same semantic relations.

4.2 Modifying multiple records

A single SQL statement can modify multiple records. The basic approach described so far addresses this problem, as each semantic description may refer more than one record.

A mobile transaction can also include a loop to read and modify a sequence of records. Two cases must be considered.

When the value of the condition that controls the loop is known, the analysis of the loop is trivial: it is just necessary to evaluate the code inside the loop as many times as needed. An example of this situation is a for instruction that only involves constant values (e.g. for i in 1..3 loop) or that include values read from the database that can be solved as described in the previous subsection (e.g. a cursor variable that iterates over a set of constant records).

When the value of the condition that controls the loop is not known, it may be very difficult or impossible to extract precise information. We are still investigating

```
BEGIN                          -- First alternative
  SELECT stock,price INTO l_stock,l_price FROM products
                     WHERE id = 80;
  IF l_price <= 100 AND l_stock >= 10 THEN
    UPDATE products SET stock = stock - 10 WHERE id = 80;
    INSERT INTO orders
    COMMIT;
  ENDIF;                       -- Second alternative
  SELECT stock,price INTO l_stock,l_price FROM products
                     WHERE id = 47;
  IF l_price <= 85 AND l_stock >= 10 THEN
    UPDATE products SET stock = stock - 10 WHERE id = 47;
    INSERT INTO orders
    COMMIT;
  ENDIF;
  ROLLBACK;
END;
```

Figure 9: Mobile transaction that specifies two alternative orders in a mobile sales application.

this problem. Currently, our prototype sets the default relations for transactions that include this type of loops.

4.3 Handling long transactions

Long transactions that access many data items may end up having contradictory relations with (many of) the other transactions. In this situation, the inferred relations are not useful for reconciliation, leaving the reconciler with no information for ordering the execution of the transactions.

To avoid this problem, the SqlIceCube API allows programmers to submit complex operations as a composition of smaller transactions linked by suitable log relations. Furthermore, the SqlIceCube system provides a preprocessing mechanism that can split each mobile transaction into smaller transactions linked by suitable log relations. This preprocessing step may be run at the client when a mobile transaction is submitted.

The following preprocessing rules are applied:

- A transaction composed of a set of alternative actions coded as a sequence of if instructions is split into an ordered set of alternative transactions (using the *alternatives* and *weak predecessor-successor* log relations), each one with a single alternative action. For example, the transaction in figure 9 is split into two transactions identical to the ones presented in figure 2.

- A long transaction composed of independent sequences of statements may be split into a parcel of smaller transactions. This rule is only applied if the application allows the system to do it, as parcel execution may not comply with the isolation property of transactions.

Other log relations can also be automatically inferred between transactions submitted in the same mobile client. For example, when a transaction reads a data item inserted by a previous transaction, the *strong predecessor-successor* log relation can be added between the older and the new transaction.

5 Status

We have implemented a prototype of the SqlIceCube reconciliation system in Java. Our prototype has five major components.

First, the database system. We are currently using the *hsqldb* Java Database Engine [1], but any SQL database engine could be used. We implement the checkpoint primitive needed by the SqlIceCube reconciler by making a copy of the database files.

Second, the PL/SQL interpreter that executes mobile transactions on top of a generic SQL database. Our prototype interprets an extended subset of PL/SQL; the parser is written using JavaCC.

Third, the module that extracts information from mobile transactions. This module analyzes the abstract parse trees built by the PL/SQL parser.

Fourth, the module that infers relations using the extracted information. This module uses the JSolver general-purpose constraint solver to verify the compatibility of semantic descriptions.

Finally, the generic semantics-based reconciler that creates and executes a schedule combining sets of mobile transactions. To this end, we have modified the IceCube reconciler [2].

Some issues of the prototype are still being worked on, namely the full interpretation of PL/SQL (in the PL/SQL interpreter) and SQL (in the semantic inference module).

6 Benchmarking SqlIceCube

In this section we present measurements to evaluate both the quality of reconciliation (by the size of the schedules), and its efficiency (by execution time).

This uses transactions submitted in the context of a shared calendar, as presented in section 3.3. The benchmark inputs are based on a trace from actual Outlook calendars, modified to control the rate of conflicts and to include alternatives.

The logs contain requests for new appointments, each one composed by one or more alternative elementary appointment requests. We varied the number of requests and the number and size of possible clusters. The average number of alternative elementary requests per request is two. Thus, results showing 100 requests represents the reconciliation of 200 elementary requests submitted as basic transactions identical to the one presented in figure 2. Note that execution times includes: partitioning the reconciliation problem; initialization for inferring semantic relations and computing summaries of relations; checkpointing, for creating and

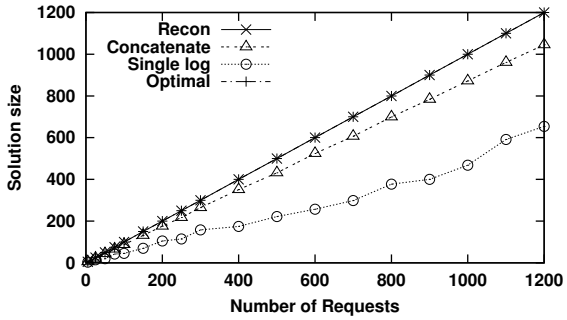

Figure 10: Size of reconciliation.

restoring database checkpoints; scheduling, for creating schedules; and schedule execution to run schedule against the database.

In each cluster, the number of different elementary requests is no larger than the number of requests. For instance, in the example of Figure 1, for the three requests, there are only three different basic transactions (scheduling the appointment for '9am room A', '9am room B' and '9am room C'). This situation represents a hard problem for reconciliation because the right elementary request needs to be selected in every request (selecting other alternative in any request may lead to the impossibility of committing any transaction for some requests).

In these experiments, all transactions have equal value, and longer schedules are better. A schedule is called a *max-solution* when no request is dropped (i.e. one elementary request from each request can be committed). A schedule is said *optimal* when the highest possible number of requests has been executed successfully. A max-solution is obviously optimal; however not all optimal solutions are max-solutions because of unresolvable conflicts. Since SqlIceCube uses heuristics, it might propose non-optimal schedules; we measure the quality of solutions compared to the optimum. (Analyzing a non-max-schedule to determine if it is optimal is an offline, *a posteriori* process.)

The experiments were run on a generic PC running Windows XP with 256 Mb of main memory and a 1.1 GHz Pentium III processor. SqlIceCube was run using the Sun's Java virtual machine version 1.4.0_01. The *hsqldb* database runs in *standalone mode*, with the database state being safely stored on disk after each transaction (i.e., we are not using the *hsqldb* mode that runs a database completely in memory). Results presented are the average of 10 runs using different sets of requests. Any comparisons present results obtained using exactly the same inputs.

6.1 Single cluster

We first evaluate the SqlIceCube heuristics without clustering. Our first set of inputs gives birth to a single cluster.

Figure 10 compares SqlIceCube with a syntactic scheduling algorithm, to jus-

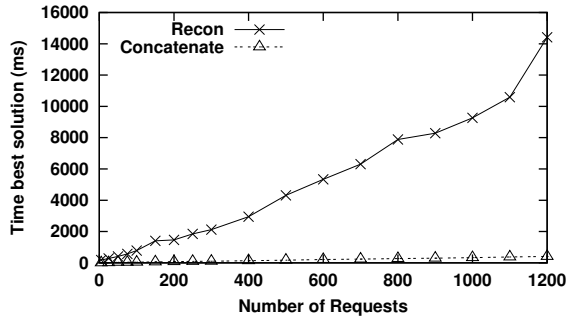

Figure 11: Execution time to solution (single cluster).

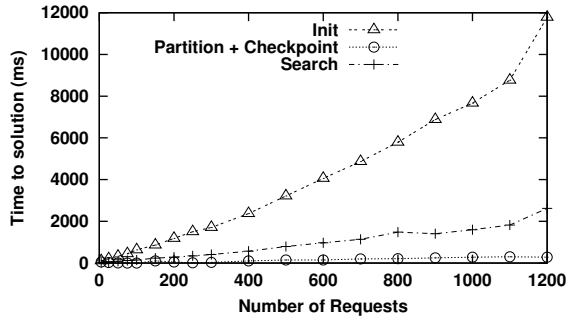

Figure 12: Decomposition of reconciliation time (single cluster).

tify our optimization approach. We choose the concatenation of the logs, but any syntactic algorithm will have similar worst-case performance. For instance, in Bayou [30], transactions are ordered as they are received in the primary server, which may lead to exactly the same order as concatenation order.

As expected, the results of semantic-directed search are better than syntactic ordering. In our tests, SqlIceCube could always find the best solution. With syntactic ordering, approximately 13% of the requests are dropped. The baseline marked "Single log" in the graph represents the simplest non-trivial scheduler that guarantees absence of conflicts (it selects all transactions from a single log and drops all transactions from the other).

The drop rate grows very slightly with size. Although the improvement may appear small, remember that dropping a single transaction may have a high cost — for instance missing an important meeting or violating the terms of a high-value contract.

Figure 11 shows the execution time of SqlIceCube reconciliation vs. a log-concatenation (hence suboptimal) scheduler. As expected, SqlIceCube is slower. This is in line with the expected complexities, $O(n^2)$ in SqlIceCube and $O(n)$ for concatenation.

Figure 12 decomposes the execution time into three parts. First, initialization

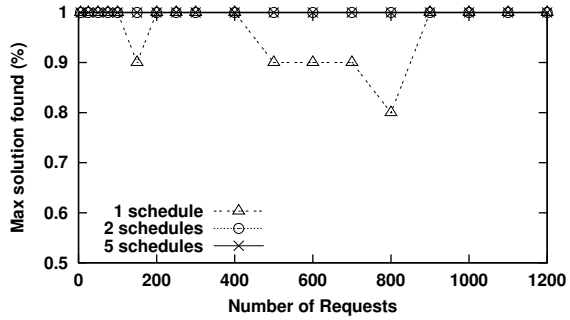

Figure 13: Percentage of cases where a max-solution was found in the first 1, 2 or 5 schedules created.

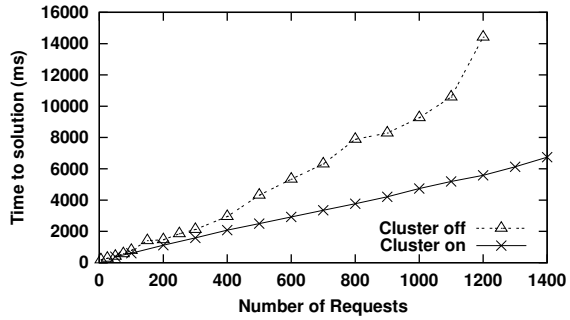

Figure 14: Execution time to solution (with and without clustering)

time, where relations are inferred and summaries of relations computed. This represent most of the execution time; as expected, it presents a non-linear growth ($O(n^2)$). Second, the time spent in partitioning and checkpointing; this component is negligible. Third, the search time, i.e., the time to create and execute the schedules. Unlike what we expected, the search time line does not show a clear quadratic increase. This is due to the small constants in search (when compared with larger constants for relation inference). These results show that relation inference dominates the reconciliation. Thus, if relations inference can be executed prior to reconciliation it is possible to improve reconciliation results (although complexity would remain $O(n^2)$).

In these experiments, the heuristic search ends when a max-solution is found, or after a given amount of time (two minutes). Figure 13 shows how quickly a max-solution is reached. The first schedule is a max-solution in over 80% of the cases. In all experiments, a max-solution was found in the first two iterations. This shows that our search heuristics work very well, at least for this series of tests. A related result is that in this experiment, even non-max-solutions were all within 1% of the max size.

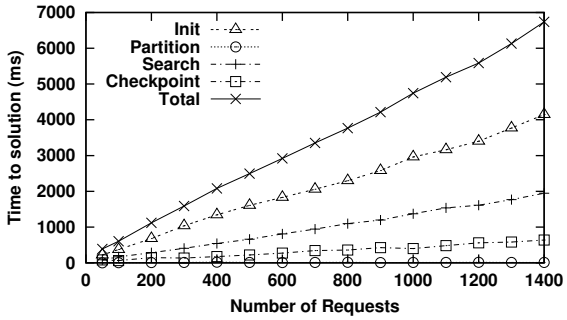

Figure 15: Decomposition of reconciliation time (multiple clusters).

6.2 Multiple clusters

We now show the results when it is possible to cluster the transactions. This is the expected real-life situation. For instance file-system traces obtained from real usage show that concurrent updates of the same file are rare [31, 18]. We expect that in a more collaborative environment, the number of concurrent updates may increase, but still remain small.

The logs used in these experiments contain a variable number of requests, and are constructed so that 25% of the elementary request can be clustered alone; 25% of the remaining elementary request are in clusters with two transactions; and so on. Thus, as problem size increases, the size of the largest cluster increases slightly, as one would expect in real life. For instance, when the logs contain 1000 requests, the largest cluster contains the elementary request from 12 requests. The number of clusters is approximately half of the number of transactions; this ratio decreases slightly with log size. The average number of alternatives per request is two.

SqlIceCube always finds the optimal solution, whether clustering is in use or not. In contrast, using log concatenation, about 8% of the requests were dropped. This value is smaller than in the previous subsection because a large fraction of the requests have zero or close to zero related transactions; e.g., 25% of the transactions commute with all other transactions, thus, they can be scheduled in any order.

Figure 14 shows the time to find a max-solution, with clustering turned on or off. As expected a solution is obtained more quickly when clustering is used. As the number of clusters grows almost linearly with the number of transactions and the size of the largest cluster grows very slowly, reconciliation time is expected to grow almost linearly. The results confirm this conjecture. Moreover, the decomposition of the reconciliation time of figure 15, shows that all components of the reconciliation time grow linearly, as expected.

7 Deployment

SqlIceCube is a generic semantics-based reconciliation engine that can be used to combine concurrently executed mobile transactions. In this section we present three scenario where SqlIceCube could be used easily to improve reconciliation results.

7.1 In the context of optimistic replication

Mobile database systems that use optimistic replication allow operations to be submitted in disconnected devices. These operations are considered tentative until they are propagated to a server where they are scheduled and committed by some form of global agreement. In general, a set of operations is submitted to be ordered simultaneously.

For example, in the Bayou system [30], a central server orders sets of transactions received from other servers during epidemic synchronization sessions. In the Deno system [15], successive election rounds order sets of operations simultaneously proposed in several sites.

Instead of defining an arbitrary order among transactions under consideration, SqlIceCube could be used in these systems to define an execution order that allows the commitment of a larger subset of transactions.

7.2 In the context of conflict-avoidance

Mobile database systems can guarantee the results of transactions in a disconnected mobile device by including a conflict-avoidance mechanism. This mechanism can guarantee the transactions executed by one or more users by restricting the transactions that can be executed by all other users. For example, locks [11] are a traditional mechanism of conflict avoidance: the user that obtains the lock can guarantee that her transactions can be executed without conflicts by forbidding the execution of transactions by other users.

Conflict-avoidance mechanisms suitable for mobile environments have been proposed to support mobile sales applications (escrow techniques) [17] and generic database applications (reservation in the Mobisnap system) [25].

These conflict-avoidance mechanisms can be combined with optimistic replication, thus guaranteeing results in mobile devices without forbidding the submission of tentative transactions that cannot be locally guaranteed [25]. These tentative transactions cannot be committed until the *locks* (or similar technique) used by the conflict-avoidance mechanism are released or expire. At that point in time, there might be a set of transactions awaiting execution. Using SqlIceCube to reconcile this set of transactions can improve the number of transactions that can be executed. For example, this approach could be used in Mobisnap [25] to execute transactions awaiting reservation expiration.

7.3 Batch-oriented systems

Batch-oriented systems, where sets of transactions are submitted for late execution, are a perfect scenario for using SqlIceCube. In these systems SqlIceCube can be used to define the execution order that allows more transactions to be committed. Furthermore, it is possible to, at least, partially infer relations before the time selected to start transaction execution.

8 Related work

8.1 Reconciliation

Several systems use optimistic replication and implement some form of reconciliation (see Saito and Shapiro [28] for a recent survey). The existing approaches, although suitable for the intended applications, tend to present several shortcomings when applied in a general-purpose database setting.

Application-specific reconciliation systems, e.g. CVS [5], are limited to a single application. Adapting the implemented reconciliation strategies to different applications tend to be very difficult or even impossible.

Systems that reconcile by comparing final tentative states, e.g. Lotus Notes [14] and Coda [18], or using the read and write set of client execution, e.g. Oracle Lite [23], cannot exploit the sematic information associated with the executed operation, thus leading to the detection of false conflicts. Although this approach may be the unique available in some settings (e.g. file systems), reconciliation systems benefit from exploring the semantic information associated with the executed operations.

Log-based systems, as SqlIceCube, use the logs of executed operations during reconciliation. Recent optimistically-replicated systems include Bayou [30], TACT [32] and Deno [15]. Balasubramaniam and Pierce [4] and Ramsey and Csirmaz [27] study file reconciliation from a semantics perspective.

In the two-tier replication model [10], mobile nodes may propose tentative update transactions. These transactions are propagated to a base node, where they are reapplied to the object master copy. An acceptance rule can be specified to verify the validity of transaction execution. Invalid transactions are aborted and diagnostic messages are returned to the mobile nodes. In Bayou [30], data is replicated over a group of servers that synchronize their state using epidemic techniques. Bayou updates include information to allow generic automatic conflict detection and resolution through dependency checks and merge procedures. A primary server is responsible for defining the commit order for updates — all servers execute the updates in the commit order. The SqlIceCube mobile transactions can be seen as an PL/SQL implementation of the updates proposed in these systems. However, unlike these systems, updates can be reordered to obtain a better reconciliation result (instead of relying on a pure syntactic order).

Lippe et al. [20] search for conflicts exhaustively comparing all possible schedules. Their system examines all schedules that are consistent with the original order of operations. A conflict (to be resolved manually) is declared when two schedules lead to different states. Examining all schedules is untractable for all but the smallest problems.

Phatak and Badrinath [24] propose a transaction management system for mobile databases. A disconnected client stores the read and write sets (and the values read and written) for each transaction. The application specifies a conflict resolution function and a cost function. The server serializes each transaction in the database history based on the cost and conflict resolution functions. As this system uses a brute-force algorithm to create the best ordering, it does not scale to a large number of transactions.

IceCube [16, 26] was the first to treat reconciliation as an optimization problem that needs to be addressed using heuristic techniques. The first version only supported a single semantic relation that indicates in which order two operation should be executed. A later version presents a rich set of relations, including the distinction between log and data constraints (object constraints in their terminology). However, IceCube reconciler treats reconciliation as a constraint-solving problem. As discussed in section 2.2.2, this approach does not handle pairs of compensating operations. Our planning-like approach seems more appropriate to handle reconciliation problems. We have also defined a different set of relations suitable to the new paradigm. Furthermore, by inferring semantic relations automatically, our system greatly simplifies the use of semantics-based reconciliation and makes possible the definition of new transaction at any time.

Other semantics-based reconciliation systems reorder the execution of operations. In Sync [21], programmers may define reconciliation rules for each pair of operations of each object. These rules can define the order of execution for operations (or define a new operation). Operational transformation algorithms [29] rewrite update parameters to enable order-independent execution of non-conflicting operations, even when they do not commute.

In these systems, programmers must express the semantic information needed by the reconciliation engine. However, the overhead of expressing the semantic information needed by the reconciliation engine is usually non-trivial. Therefore, some programmers will not be willing to execute this extra work or end up creating bad semantic rules that leads to poor reconciliation results. We believe that automating the inference of semantic information is the key to allow semantics-based reconciliation to be used in more applications.

8.2 Static analysis

Static analysis [8] of programs has been used before in several systems to verify the correctness [7, 12, 3] and equivalence of programs [19]. We have introduced any new static analysis technique . However, our semantic inference module presents the following uncommon characteristics: it is executed at reconciliation runtime,

the information extracted from each program is used to infer relations between two programs (other than equivalence) and it is combined with dynamic evaluation.

To our knowledge, this work is the first to use static analysis to infer semantic information for reconciliation.

9 Final Remarks

SqlIceCube is a general-purpose semantics-based reconciliation system for mobile databases. The system includes a semantic inference module that automatically extracts semantic information from the code of mobile transactions. This information, exposed as a set of static relations among transactions, is used by the basic reconciler to create near-optimal reconciliation results.

To our knowledge, SqlIceCube is the first to automatically infer the semantic information needed in reconciliation. This approach simplifies the use of semantics-based reconciliation by eliminating the overhead of writing long/complex rules to expose the semantics of operations. Thus, semantics-based reconciliation can be used without any extra effort by any programmer.

The SqlIceCube reconciler creates near-optimal schedules using an heuristic search approach. The approach is general and supports combinations of transactions across a variety of different applications. This approach, used previously in IceCube [26], has the potential to improve the reconciliation result, thus reducing the (possibly high) impact of dropping any transaction executed tentatively.

Unlike previous systems [26], SqlIceCube treats the optimization problem as a planning-like problem instead of a constraint-solving problem. Furthermore, SqlIceCube defines a new set of relations that can be used to expose semantic information in reconciliation problems. This set of relations is designed to be used with the new planning-like reconciliation approach and to be easily extracted from the code of mobile transactions automatically.

The examples presented in this papers show that SqlIceCube correctly infer semantic relations from different typical mobile applications. The results reported, obtained with a non-optimized prototype, suggest that the reconciler can find near-optimal solutions, that improve results obtained using simpler syntactic approaches, in reasonable time and scale to large logs. The deployment scenarios discussed in section 7 show that SqlIceCube can be used in different settings to improve the number of transactions that can be executed with success. To further validate our approach, we intend to continue studying the use of SqlIceCube to support different types of applications.

References

[1] hsqldb java database engine. `http://hsqldb.sourceforge.net/`.

[2] Icecube code. `http://research.microsoft.com/camdis/icecube.htm`.

[3] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proc. 29th ACM Symp. on Principles of programming languages*, 2002.

[4] S. Balasubramaniam and Benjamin C. Pierce. What is a file synchronizer? In *Int. Conf. on Mobile Comp. and Netw. (MobiCom '98)*. ACM/IEEE, October 1998.

[5] Per Cederqvist, Roland Pesch, et al. Version management with CVS, date unknown. `http://www.cvshome.org/docs/manual`.

[6] Shaul Dar, Michael J. Franklin, Björn Jónsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *Proc. VLDB'96*. Morgan Kaufmann, September 1996.

[7] Dawson Engler, David Yu Chen, and Andy Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proc.* 18^{th} *Symp. on Op. Sys. Principles*, pages 57–72, 2001.

[8] H.R. Nielson F. Nielson and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[9] François Fages. A constraint programming approach to log-based reconciliation problems for nomadic applications. In *Proc.* 6^{th} *Annual W. of the ERCIM Working Group on Constraints*, June 2001.

[10] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. Dangers of replication and a solution. In *Int. Conf. on Management of Data*, pages 173–182, Montréal, Canada, June 1996.

[11] Jim Gray and Andreas Reuter. *Transction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[12] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proc. 2002 Conf. on Programming language Design and Implementation*, pages 69–82. ACM Press, 2002.

[13] Yannis E. Ioannidis and Raghu Ramakrishnan. Containment of conjunctive queries: beyond relations as sets. *ACM Transactions on Database Systems (TODS)*, 20(3):288–324, 1995.

[14] Leonard Kawell Jr., Steven Beckhart, Timoty Halvorsen, Raymond Ozzie, and Irene Greif. Replicated document management in a group communication system. In *2nd. Conf. on Comp.-Supported Coop. Work*, Portland OR (USA), September 1988.

[15] Peter Keleher. Decentralized replicated-object protocols. In *Proc.* 18^{th} *Symp. on Princ. of Distr. Comp. (PODC)*, Atlanta, GA, USA, May 1999.

[16] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The IceCube approach to the reconciliation of diverging replicas. In *20th Symp. on Princ. of Distr. Comp. (PODC)*, Newport, RI, USA, August 2001.

[17] Narayanan Krishnakumar and Ravi Jain. Escrow techniques for mobile sales and inventory applications. *Wireless Networks*, 3(3):235–246, 1997.

[18] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *Proc. USENIX Winter Tech. Conf.*, New Orleans, LA, USA, January 1995.

[19] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proc.* 29^{th} *ACM Symp. on Principles of Programming Languages*, pages 283–294. ACM Press, 2002.

[20] E. Lippe and N. van Oosterom. Operation-based merging. *ACM SIGSOFT Software Engineering Notes*, 17(5):78–87, 1992.

[21] Jon Munson and Prasun Dewan. A flexible object merging framework. In *Proc. Conf. on Comp.-Supported Cooperative Work (CSCW)*, page 231, 1994.

[22] Oracle. Pl/sql user's guide and reference - release 8.0, June 1997.

[23] Oracle. Oracle8i lite replication guide - release 4.0, 1999.

[24] Sirish Phatak and B. R. Badrinath. Transaction-centric reconciliation in disconnected databases. In *ACM MONET*, July 2000.

[25] Nuno Preguiça, J. Legatheaux Martins, Miguel Cunha, and Henrique Domingos. Reservations for conflict-avoidance in a mobile database system. In *Proc. Mobisys 2003*, 2003.

[26] Nuno Preguiça, Marc Shapiro, and Caroline Matheson. Efficient semantics-aware reconciliation for optimistic write sharing. Technical Report MSR-TR-2002-52, Microsoft Research, Cambridge (UK), May 2002. `http://research.microsoft.com/scripts/pubs/view.asp?TR_ID=MSR-TR-2002-5%2`.

[27] Norman Ramsey and Előd Csirmaz. An algebraic approach to file synchronization. In *9th Int. Symp. on the Foundations of Softw. Eng.*, Austria, September 2001.

[28] Yasushi Saito and Marc Shapiro. Replication: Optimistic approaches. Technical Report HPL-2002-33, Hewlett-Packard Laboratories, March 2002.

[29] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proc. Conf. on Comp.-Supported Cooperative Work (CSCW)*, page 59, November 1998.

[30] Douglas Terry, Marvin Theimer, Karin Petersen, Alan Demers, Mike Spreitzer, and Carl Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. 15th Symp. on Op. Sys. Principles*, Copper Mountain CO (USA), December 1995. ACM SIGOPS.

[31] An-I Andy Wang, Peter Reiher, and Rajive Bagrodia. A simulation framework and evaluation for optimistically replicated filing environments. Technical Report CSD-010046, Computer Science Department, University of California, Los Angeles, Los Angeles CA (USA), 2001.

[32] Haifeng Yu and Amin Vahdat. Combining generality and practicality in a Conit-based continuous consistency model for wide-area replication. In *21st Int. Conf. on Dist. Comp. Sys. (ICDCS)*, April 2001.