LazyDB: Using Semantic Reordering for Improved Commit Rate

Nuno Preguiça[1], Marc Shapiro[2], J.Legatheaux Martins[1]

[1]CITI/DI, FCT, Univ. Nova de Lisboa, Portugal [2]INRIA, France

Abstract

Recent work on reconciliation in optimistic replication systems has shown that it is possible to minimize the number of aborted operations by reordering them. In this paper, we present LazyDB, a system that leverages this idea in a non-replicated architecture. In our system, when a user submits an update operation, the system immediately informs the user if the operation will be executed with success or not. However, the system reserves the right to reorder operations until a deadline specified by the user. This allows, in some situations, to commit additional operations that would otherwise be aborted, by reordering operation execution.

1 Introduction

Optimistic replication is used to improve availability by allowing execution of concurrent updates at different replicas. System that use this approach usually include some mechanism to merge these concurrent updates, repairing data divergence – see [12] for a recent survey. However, if concurrent updates conflict irreparably some must be aborted. Bayou [15] shows that it is possible to decrease such aborts if semantic-rich conflict detection and resolution rules are used. IceCube [7] improves on this result by reordering update execution.

In this paper we propose a novel approach which extends these ideas and applies them in a non-replicated environment. In our data management system, LazyDB, when a user submits an operation, the system immediately returns a *high-level* result that indicates only if it will be possible to execute the operation (*promised commit*) or not (*abort*). If the result is *promised commit*, the system guarantees that it will be able to execute the update operation with success before the deadline specified by the user (when he submits the operation). This allows the system to reorder operations to improve the chance of success for operations submitted later.

Although it is clear that this approach cannot be used in all situations, there are situations where it could be used with benefit. In fact, a similar approach is already used by some ad-hoc implementations. For example, consider the organizers of HotOS reserving the meeting room for the next conference: they will state only the requested capacity of the meeting room. If the hotel has a room with such capacity, it can confirm the reservation without immediately saying which specific room will be used. For HotOS organizers, this *high-level* result is sufficient. It is not until close to the meeting (the operation *deadline*) that they need the specific room information. Without immediately deciding which room will be used, the hotel has more flexibility to accommodate future requests (e.g. for a specific room, for a smaller capacity, etc).

The remainder of this paper describes how to build the LazyDB data management system. Section 2 briefly presents related work. Section 3 discusses the LazyDB principles and section 4 presents the system design. Section 5 outline the generic scheduler that reorders operations. Section 6 presents an application example and section 7 concludes the paper with some final remarks.

2 Background

The study of algorithms to control replicated data has been an important topic of research for years. The devised algorithms can be broadly divided into eager and lazy replication [4]. Eager replication algorithms provide single-copy serializability and maintain all (or a quorum of) replicas exactly synchronized [1, 6]. Lazy replication algorithms allow replicas to diverge and usually include some mechanism to merge concurrent updates – see [12] for a recent survey. Reconciliation solutions can be divided into two groups: state-based (e.g. [13, 5]) and operation-based (e.g. [15, 8, 9, 7, 2, 14]) protocols.

Our approach to reorder operations is operation-based. However, unlike previous system, we apply it in a system that is not based on optimistic replication. Moreover, unlike the common approach in systems that use optimistic replication, that try to merge concurrent operations and commit the reconciliation result as soon as possible, we postpone the final execution of operations until the deadline specified by the users. This allows the system to explore the reordering of operations to improve the chance of executing additional operations with success. Finally, unlike the common approach in mechanisms that use semantic information, our system auto-

matically extracts the needed information, thus imposing no overhead on programmers to use LazyDB.

3 System principles

In this section we present the main ideas of LazyDB without tying the description to the specific details of the prototype we are building.

LazyDB is a client/server data management system. The server stores shared information, which the users modify by submitting update operations to be executed on the server. Update operations are small programs – in our prototype small PL/SQL program. An operation ends in failure if it executes the rollback statement; otherwise it terminates in success.

The main idea of LazyDB is to allow the system to postpone the final execution of a submitted update operations until a deadline specified by the user, but still provide immediate confirmation that the operation will be executed with success (or not). Thus, when a client submits an update operation, the possible results are:

Promised commit The system guarantees that it will be able to execute the update operation with success before the given deadline.

Abort The system cannot execute the update operation with success - by default, the operation is immediately discarded.

To guarantee the *promised commit* result for some operation, the system must verify that it can create an execution order where the operation and all other previous *promised commit* operations can execute with success. If this is not possible, the result is *abort.*

We believe that this model often closely matches the requirements of users and it is used in real-life situations, as described in section 1, because *promised commit* is closely related to the users' intentions instead of being coupled with the specific implementation details of the application that handles the operations. For achieving the same goals, it is common to have a two-step procedure in other situations, such as the reservation of flight seats, hotel rooms, etc. For example, in a flight reservation application, the availability of some seat is guaranteed before the exact seat is assigned to the client. LazyDB simplifies and automates the creation of such two-step procedures.

In a more general way, LazyDB can be useful for all applications that can delay the final execution of operations thus allowing the system to explore operation reordering for improving the chance of committing new operations. For example, consider an online shopping company with more than one warehouse for dispatching requests. If more than one warehouse has available stock to fulfil a request, it is not necessary to decide which one will process it until dispatch time. Suppose that a request, r_2, is submitted that can only be satisfied in a specific warehouse if some previous request, r_1, is not processed in the same warehouse (due to the number of available items). In this case, LazyDB can solve this problems by reordering operations – by ordering r_2 before r_1, r_2 will be processed in the only warehouse with enough stock for processing it, while r_1 will be processed by some other warehouse.

4 System Design

LazyDB is a data management system that manages data structured according to the relational data model. It is based on a client/server middleware architecture. Applications run on client machines and access the system through the LazyDB API, that allows to submit read and update operations to the server. A read operation is a SQL query. An update operation is a small PL/SQL program that is executed in the server as transaction.

The server stores data in an unmodified relational database system. The server maintains a log of *promised commit* operations (*promised operations* for short) and two (logical) data versions: the committed and the tentative. The committed data version results from the execution of all operations that have been definitely executed. The tentative data version results from the additional execution of the log of *promised operations.* In our prototype, the database safely stores only the log of *promised operations* and the committed data version. The tentative data version is maintained only in soft-state by executing the *promised operations* in a running (non-committed) transaction.

When the server receives a query operation, it immediately executes the query in the tentative version, i.e., in the context of the transaction with promised operations.

Each update operation is handled sequentially as it is received in the server. To determine its result, the system starts by trying to execute the operation program in the the current tentative state. If it succeeds, the system returns the result *promised commit* and appends the operation in the log of updates. If it fails, the system tries to efficiently create a new execution order (called schedule) that can execute the operation and all previously promised operations with success. If it is possible to find such schedule, the system returns the result *promised commit* and replaces the old log of promised operations by the new schedule. If it is not possible to find such schedule in a small period of time[1], the system returns the result *abort* and discards the operation. Thus,

[1]The period of time used to search for a new schedule is config-

the system guarantees that the log of operations always contains all previously promised operations. In the next subsection we explain how LazyDB efficiently produces new optimal schedules.

The previous description represents the standard procedure for handling updates. However, two extensions have been considered. First, the client may specify that if an operation cannot be *promised committed*, the system should nonetheless try to commit the operation until its deadline. In this case, the result is *probably abort*. The server stores the operation and tries to include it in the new schedules. Second, when the deadline of an operation is *now*, the system starts by trying to create new schedules that include the operation. The rationale for this approach is that, when the deadline of an operation is reached, the operation and all operations that precede it in the schedule must be committed and cannot be reordered again. Thus, the system tries to find a schedule that minimizes the number of operations that must be committed. If such a schedule is found, the result of the operation is *definite commit* (instead of *promised commit*). Otherwise, the result is *abort*, as before.

When the deadline of a *promised operation* is reached, the operation and all preceding operations in the log of updates become committed and will not be reordered again. The new *committed* data version is updated by executing the new committed operations, which are removed from the log of updates.

It is possible to configure the system to create new schedules during idle periods. These schedules try to include *probably aborted* operations and reorder operations according to their deadlines.

5 Generic scheduler engine

LazyDB includes a scheduler engine which aims to create schedules that maximize the number of successful operations. We treat this as an optimization problem, trying to find the schedule that allows more operations to succeed. Semantic information collected from the operations directs the search using an heuristic approach. The developed solution builds on our previous work in the SQLIceCube reconciliation engine [11, 10], by adapting and extending the proposed approach to this new setting.

5.1 Semantic information

The system uses static and dynamic semantic information. The static information is based on relations among operations that can be obtained by the static analysis of the operation code. The dynamic semantic information can only be obtained by analysis of the system state.

urable in the system and can also be modified for each transaction by the clients.

5.1.1 Static semantic information

Data relations encode the semantics of the operations, independently of the database state. The following static data relations are used:

Commute Two operations commute if the result of executing both is independent of the execution order.

Helps The operation o_1 *helps* the operation o_2 if the execution of o_1 may change the database state in a favorable way for the success of o_2 - e.g. increasing the stock of some product improves the chance of accepting a new order. The degree of *help* is specified as an integer value.

Prejudices The operation o_1 *prejudices* the operation o_2 if the execution of o_1 may change the database state in a prejudicial way for the success of o_2 - e.g. decreasing the stock of some product reduces the chance of accepting a new order. The degree of *prejudice* is specified as an integer value.

Makes possible The operation o_1 *makes possible* the operation o_2 if the execution of o_1 changes the database state in a way that makes possible to execute o_2 with success - e.g. removing all reservations for a given room makes possible a new reservation.

Makes impossible The operation o_1 *makes impossible* the operation o_2 if the execution of op_1 changes the database state in a way that makes impossible to execute o_2 with success - e.g. reserving a room makes impossible to reserve the same room.

The relations helps, prejudices, makes possible and and makes impossible encode the influence of one operation over some other operation.

Log relations are independent of the semantics of each specific operation and express additional relations that are usually specified by the users to combine several operations. The following log relations are defined:

Explicit alternatives Specify that a single operation must be committed from a set of alternative operations.

Predecessor-successor The successor operation can only be executed after and if the predecessor operation is executed.

Parcel Defines an all-or-nothing group of operations.

5.1.2 Dynamic semantic information

Dynamic semantic information includes information that may depends on the current database state. For keeping the system efficient, as this information may vary with the execution of new operations, we have only considered the following information:

Self-alternatives Specify the number of alternatives internally defined in the code of the operation.

5.2 Extraction of semantic information

A possible approach to obtain the semantic information would be to request programmers to write rules to compute the needed information given pairs of operations. This is the common approach for obtaining sematic information in semantics-based reconciliation (e.g. [7, 8]), but it 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 system, like LazyDB, where clients may submit new operations, these drawbacks make it impossible to use a similar approach.

LazyDB automatically extracts the needed semantic information. To this end, it adapts and extends the solution we have proposed in the SqlIceCube reconciler [11]. This solution has two steps that we briefly outline.

In the first step, the code of each operation is statically analyzed checking all execution paths. For each path leading to a successful execution, the following information is obtained: the read set (from select statements), the write set (from insert, update and delete statements) and the set of preconditions (from conditions in conditional control-flow instructions). These sets only contain semantic descriptions [3] of the data items (with table and column names and conditions used to refer the data).

In the second step, the static data relations between each pair of operations are inferred comparing the information extracted from each operation. For example, operation o_1 helps operation o_2 if o_1 writes change the database in a favorable way for o_2 preconditions to be true. The weight of the help relation is inferred from the o_1 writes – e.g. if o_1 cancels the reservation of two rooms that o_2 is trying to reserve, the weight of help is 2.

The dynamic semantic information is inferred by accessing the current database state. Thus, the number of self-alternatives for each operation is computed by checking the data items that satisfy the pre-conditions of the operation – e.g. if o_1 is trying to reserve one room, and there are currently four rooms that satisfy the specified conditions, the number of self-alternatives is 3.

5.3 Scheduler algorithm

The scheduler algorithm tries to find a schedule that maximize the number of operations that can be executed with success. It works in two steps. First, the operations are partitioned in subsets of inter-related operations – an operation is related with other if there is some log relation between the two or if they do not commute. This divides the scheduling problem into smaller sub-problems that can be solved independently by any arbitrary order.

Second, for each subset and as it may still be impossible to check all possible (partial) schedules, the scheduler probes the space of possible solutions heuristically. To this end, the scheduler creates and executes a sequence of schedules until an an optimal schedule is found (or the specified search time is exhausted). The scheduler creates each schedule incrementally, selecting at each step one operation based on its merit. The merit of an operation is estimated by an oracle based on the semantic information associated with each operation.

The implemented algorithm uses an approach similar to the algorithm of the SqlIceCube reconciler [11]. However, some difference are worth mentioning. First, some modifications have been introduced to the basic scheduler algorithm to cope with differences in the available semantic information, notably the new *self-alternatives* and the weights in the *helps* and *prejudices*. Second, whenever possible, the scheduler implements an incremental strategy relying on previously computed partial schedules. Unlike the case of reconciliation, this is often possible in LazyDB. For example, when a new operation needs to be integrated, the system already has a computed schedule that includes all previously promised operations – only the (partial) schedule for the subset that includes the new operation needs to be recomputed.

6 Application example

In this section we exemplify how the system works in the context of a simple application for reserving rooms. In this example, two basic operations can be executed: reserve a room or cancel an existing reservation. In the first case, the conditions that the meeting room has to satisfy may vary depending on the users' requirements. Figure 1 shows an operation, o_1, that reserves a room with capacity larger than or equal to 60. Figure 2 shows an operation, o_2, that reserves the meeting room *Ballroom A*. If o_1 is submitted first, and if *Ballroom A* has capacity larger than 60, this room might get reserved making it impossible to execute the o_2 without reordering the operations.

When the scheduler is invoked, it starts by extracting the semantic information associated with the operations. In this case, the two operations do not commute as, if executed with success, each one modifies data items read by the other. Moreover, each operation prejudices the other, as their writes may reduce the set of data items that satisfy the precondition of the other (in this case, there might be one less room available). Finally, the number of self-alternatives for the o_1 depends on the number of available rooms. o_2 has no self-alternative, as the user has specified the room to reserve.

Based on this information, the scheduler will favor schedules where the o_2 precedes o_1, because the number

```
--- RESERVES ROOM WITH CAPACITY >= 60 FOR DAY '7-MAY-2007'
BEGIN
  id = newid;
  SELECT count(*) INTO n FROM rooms WHERE capacity >= 60 AND
        name NOT IN (SELECT name FROM reservations
                                 WHERE day='7-MAY-2007');
  IF (n > 0) THEN
    SELECT name INTO name FROM rooms WHERE capacity >= 60 AND
        name NOT IN (SELECT name FROM reservations
                                 WHERE day='7-MAY-2007');
    INSERT INTO reservations
          VALUES( id, '7-MAY-2007', name, 'HotOS');
    COMMIT;
  ENDIF;
  ROLLBACK;
END;
```

Figure 1: Operation to reserve a meeting room with capacity for 60 persons (declaration of variables is omitted; when the result of a select into statement includes more than one row, one row is returned, instead of raising an exception).

```
--- RESERVES 'Ballroom A' FOR DAY '7-MAY-2007'
BEGIN
  id = newid;
  SELECT count(*) INTO n FROM reservations
                WHERE day='7-MAY-2007' AND room='Ballroom A';
  IF (n = 0) THEN
    INSERT INTO reservations
          VALUES( id, '7-MAY-2007', 'Ballroom A', 'HotOS');
    COMMIT;
  ENDIF;
  ROLLBACK;
END;
```

Figure 2: Operation to reserve the meeting room 'Ballroom A' (declaration of variables is omitted).

of self-alternatives is smaller and all other information is equal. With such schedules, if there is another meeting room that satisfies the condition o_1, it will be possible to execute both operations. If not, when the o_2 is submitted, the scheduler will not find a schedule where both operation can be executed with success. Thus, the result of o_2 will be *abort*, thus honoring the *promised commit* result previously returned for o_1.

7 Final remarks

In this paper we have presented LazyDB, a data management system that uses reconciliation techniques to improve the chance of operation success in a non-replicated environment. When an operation is submitted, the system immediately informs the user if the operation will be executed with success or not. However, the system reserves the right to reorder *promised committed* operations until a deadline specified by the user. This allows, in some situations, to commit operations that would otherwise had to be aborted, as discussed in the paper.

A large number of data management systems that include reconciliation mechanisms have been built. However, to our knowledge, the LazyDB approach is novel in a number of aspects. First, it uses reconciliation techniques in a non-replicated system. Second, it proposes the notion of *promised commit*, that closely matches users' requirements in several situations. This notion allows to postpone the final execution of operations. Third, it reorders *promised committed* operations to improve the chance of success for new operations. Finally, it includes an automatic mechanism to extract the semantic information needed by the scheduler, by static analysis of the operation code and by dynamic analysis of the system state. Thus, programmers can use LazyDB with no overhead.

Although it is clear that the LazyDB approach is not useful in all situations, there are several scenarios where it can be used with benefit, as discussed in section 3.

References

[1] Philip Bernstein and Eric Newcomer. *Principles of transaction processing: for the systems professional.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[2] Ugur Cetintemel, Peter Keleher, Bobby Bhattacharjee, and Michael Franklin. Deno: A decentralized, peer-to-peer object-replication system for weakly connected environments. *IEEE Trans. Comput.*, 52(7):943–959, 2003.

[3] Shaul Dar, Michael J. Franklin, Björn Jónsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *Proc. VLDB'96*, pages 330–341. Morgan Kaufmann, September 1996.

[4] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182. ACM Press, 1996.

[5] Michael B. Greenwald, Sanjeev Khanna, Keshav Kunal, Benjamin C. Pierce, and Alan Schmitt. Agreeing to agree: Conflict resolution for optimistically replicated data. In Shlomi Dolev, editor, *International Symposium on Distributed Computing (DISC)*, 2006.

[6] Ricardo Jimenez-Peris, M. Patino-Martinez, Gustavo Alonso, and Bettina Kemme. Are quorums an alternative for data replication? *ACM Trans. Database Syst.*, 28(3):257–294, 2003.

[7] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The icecube approach to the reconciliation of divergent replicas. In *PODC '01: Proceedings of the 20th ACM symposium on Principles of distributed computing*, pages 210–218. ACM Press, 2001.

[8] Jonathan P. Munson and Prasun Dewan. Sync: A java framework for mobile collaborative applications. *IEEE Computer*, 30(6):59–66, June 1997.

[9] Shirish Phatak and B. R. Badrinath. Multiversion reconciliation for mobile databases. In *Proceedings of the 15th International Conference on Data Engineering (ICDE)*, pages 582–589. IEEE Computer Society, March 1999.

[10] Nuno Preguiça, Marc Shapiro, and J. Legatheaux Martins. Automating semantics-based reconciliation for mobile databases. In *Proceedings of 3ème Conférence Française sur les Systèmes d'Exploitation (CFSE'03)*, October 2003.

[11] Nuno Preguiça, Marc Shapiro, and J. Legatheaux Martins. SqlIceCube: Automatic semantics-based reconciliation for mobile databases. Technical Report TR-02-2003 DI-FCT-UNL, Dep. Informática, FCT, Universidade Nova de Lisboa, 2003.

[12] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[13] M. Satyanarayanan. The evolution of coda. *ACM Trans. Comput. Syst.*, 20(2):85–124, 2002.

[14] David Sun and Chengzheng Sun. Operation context and context-based operational transformation. In *CSCW '06: Proceedings of the 2006 Conference on Computer supported cooperative work*, pages 279–288. ACM Press, 2006.

[15] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP '95: Proceedings of the 15th ACM symposium on Operating systems principles*, pages 172–182. ACM Press, 1995.