Atomic transaction chains for reliably updating partitioned databases

Abstract

To scale to millions of page requests per day, many popular web services such as Facebook, eBay, and Amazon partition application data, storing each partition on a separate database instance running on its own machine. Since these user-facing web services have strict latency and availability requirements, it costs too much to run distributed commit protocols to support ACID transactions across data partitions stored on different machines. Programmers can decompose an application operation into several smaller transactions each modifying a single data partition; however, programmers lack tools to help automatically cope with failures that occur between multiple transactions. This paper introduces the abstraction of an Atomic Transaction Chain (ATC), which allows programmers to group a series of related transactions that modify different data partitions under all-or-nothing atomic execution. We also present the design and implementation of Parti-SQL, which provides a SQL extension to simplify operations on a partitioned database. In particular, Parti-SQL allows programmers to perform primary key-based lookups/updates across partitioned tables, specify custom ATCs, and create secondary indices that are automatically maintained by the underlying storage system through its use of ATC.

We have ported a third-party application RuBIS (an online auction website) to run on top of Parti-SQL. Evaluations on a local cluster testbed show that Parti-SQL achieves good scaling performance across many machines.

1 Introduction

Most web services today rely on a database system as the storage backend. In order to scale these web services to handle growing user demand, programmers commonly resort to a partitioned design that divides data into a large number of partitions. Since each partition can be stored in a separate database instance, one can instantiate many independent database instances running on a large collection of machines to scale the storage backend’s processing capacity. Many large-scale web sites including Facebook [31], Ebay [30], Amazon [24] and Flickr [28] store partitioned data in multiple database instances.

Once data is partitioned across many nodes, one faces the daunting design question of whether or not to support distributed transactions for accessing data in different partitions. Although distributed transactions allow one to maintain the familiar ACID interface to applications, distributed transactions require a complex implementation that is often based on a two-phase commit protocol. With two-phase commit, nodes storing different partitions become tightly coupled, resulting in decreased availability and increased transaction latency. Consequently, practitioners building latency-sensitive web services advocate avoiding distributed transactions [24, 30] and restricting applications to use ACID transactions within a single data partition.

Unfortunately, restricting transactions to within a single data partition increases programming complexity. In order to modify data in multiple data partitions, programmers must break down a logical operation into a group of several smaller steps, each using a transaction to access data in a single partition. However, since there is no all-or-nothing atomicity guarantee on a group of steps, the failure of an application server might leave an operation “partially” finished, resulting in permanently inconsistent application state. For example, in an auction website, the operation to place a bid may be broken down into two steps: one to update the highest bid value on the corresponding item, and the other to record the bidding activity of the bidder. If an application server fails after executing the first step, the bid would be associated with the item, but is “lost” with respect to the bidder.

Existing systems provide the abstraction of a persistent message queue to help applications achieve fault-tolerance. Programmers explicitly enqueue application messages as part of the current step to record the next step to be performed and dequeue messages to perform the corresponding steps. These message queues are either provided as a separate service (e.g. Amazon Simple Queue Service [2], eBay’s message queue [30]) or natively supported by the storage system (e.g. Google’s Megastore [13]). While useful, the asynchronous messaging abstraction is too low-level and cumbersome to program with: first, application code has to be restructured to explicitly enqueue/dequeue and process messages. Second, for correctness, programmers must take extra care to atomically consume a queued message and execute the corresponding action. Third, to optimize performance, programmers must schedule the processing of a message near the database instance that stores the required data partition.

In this paper, we propose a new abstraction, called Atomic Transaction Chain (ATC), which enables programmers to bundle a series of related ACID transactions to guarantee the eventual execution of the entire group of transactions. In the traditional ACID parlance [21],
an Atomic Transaction Group provides “A”, meaning the grouped transactions achieve all-or-nothing atomic execution with respect to failures, but not “I”, meaning the entire group of transaction is not isolated from other concurrently executing transactions. All-or-nothing atomicity is essential for applications to modify multiple data partitions in a fault-tolerant manner. More importantly, this property can be achieved efficiently, without tightly coupling machines together in a distributed commit protocol like two-phase commit. By contrast, programmers are better at coping with the lack of isolation for the grouped transactions [24, 30].

To realize ATC in a concrete system, we build Parti-SQL, a partitioned database capable of scaling across a large collection of machines. In Parti-SQL, each participating node stores one or multiple table partitions in a local SQL database instance (e.g., PostgreSQL). To simplify programming efforts on partitioned data, Parti-SQL extends the SQL language to provide explicit support for users to (a) define partitioned database table schemas, (b) annotate a transaction to optimize its locality of execution, (d) specify ATCs to update data atomically in multiple table partitions, and (d) create alternative indices on a partitioned database table. Parti-SQL automatically keeps alternative indices update-to-date with the main table through its internal use of ATCs.

Parti-SQL expresses an ATC as a series of stored procedures each accessing a single table partition. An ATC is forwarded from one server to the next to execute each step along the chain. Parti-SQL ensures that each ATC is executed in its entirety and exactly once. Each server records the ATC to be executed and its current state of execution in a local persistent message queue. To execute a step, a server uses a local ACID transaction to atomically dequeue the ATC, perform the corresponding stored procedure and modify the ATC’s execution state.

Our implementation of Parti-SQL consists of the client-side library for parsing Parti-SQL extensions and the Parti-SQL server for forwarding and executing ATCs. Our server implementation uses PostgreSQL as the local database. We have ported a PHP-based web applications, RuBIS [1, 12] (an eBay-like auction service) to run on top of Parti-SQL. Our experience with porting this application suggests that Parti-SQL simplifies the task of operating on partitioned data. The hardest part lies on finding the appropriate partitioned table schema and the right set of alternative indices to avoid distributed joins. It is relatively straightforward to write ATCs to update partitions in multiple tables. Our performance evaluation on a local cluster is encouraging. XXX: Numbers, numbers, numbers.

Figure 1: A simple auction service example consisting of two relational tables, user-table, item-table.

2 Motivation and Approach

In this section, we motivate the ATC abstraction as well as the design of Parti-SQL. We use an online auction service as a concrete example to illustrate the challenges in partitioning application data across a collection of database servers.

In its simplest form, the application state of an auction service consists of two tables (see Figure 1). The user-table maintains per-user information such as the user identifier, her alias, physical location, her recent bid etc. The item-table maintains per item information such as a description of the item, the current highest bid on the item and the corresponding high bidder. In order to scale the service to handle hundreds of millions of users and items, we need to partition both the user-table and item-table across many machines. Specifically, we can split both tables into multiple pieces based on their respective primary keys, i.e. the user_id for user-table and the item_id for item-table. By storing different partitions on different database servers, we can scale the overall auction service using a large number of machines.

When each table is split and spread across multiple database servers, local ACID transactions are available only when accessing data within a single table partition. There exist designs for distributed ACID transactions based on two phase commit [26, 27], but the resulting performance penalty is not tolerated well by user-facing web applications (e.g. the auction service) which desire very low latency and highly available data accesses.

To avoid distributed transactions, a user needs to decompose a logical application operation into multiple steps, each of which can be accomplished by a local ACID transaction. To alleviate application programmers from handling any intervening failures, we introduce the notion of ATC which guarantees that the entire chain of steps is executed in its entirety and exactly once.

To simplify discussion, our example table schema only tracks one bid for each user.
For example, in the auction service, when a user places a bid on an item, the application server can handle this request in two serial steps: 1) add the bid into the corresponding row in `item-table` 2) update the highest bid (if necessary) on the corresponding row in `item-table`. Without ATC, programmers would have to explicitly cope with any failure that might occur in between step 1 and 2. With ATC, failure recovery is transparent to the programmers. ATC does not provide isolation of the entire chain. Therefore, a concurrent access might observe the bid in the `user-table`, but the bid is not yet reflected in the `item-table`. Two independent ATCs might also interleave their steps of execution in any arbitrary order. To guarantee correctness, programmers must ensure that those steps modifying the same table partition in concurrent ATCs commute with each other [22]. In the previous example of placing a bid, concurrent ATCs from different users commute with each other when modifying both `user-table` and `item-table`.

The ATC abstraction merely provides a means for avoiding distributed transactions. There are greater systems challenges in providing the right mechanisms to help application programmers access partitioned data in an efficient and user-friendly manner. Below, we discuss these challenges and explain Parti-SQL’s approach towards addressing them.

### Perform lookup and updates based on primary keys.

Each application server in the auction service frequently looks up a user or item by its identifier. Since table entries are partitioned across different machines, it is cumbersome for programmers figure out which machine to send a lookup or update request to. We should let application programmers explicitly specify the schema for a partitioned table at the high-level and have the Parti-SQL run-time automatically direct an application server’s requests to the appropriate database node based on the partitioned table definition.

### Update multiple tables or different table rows.

Requests like placing a bid, it is the responsibility of application programmers to manually decompose the operation into a series of steps as an ATC. Therefore, we need to support arbitrary user-defined ATCs. Parti-SQL’s runtime must guarantee that the chain of transactions is executed serially to completion so that failures do not leave an operation in a partially executed state.

### Perform lookups based on secondary keys.

A common operation in the auction service is to list all the items currently being sold by a given user. Such a lookup can be efficiently executed if there exists a secondary index based on the seller’s user identifier for the `item-table`. The secondary index is partitioned differently from the main table according to the secondary key. Therefore, it is challenging to keep these alternative indices up to date as changes to the main table happen.

The traditional method to maintain a secondary index is use ACID transactions to update the index while processing modifications on the primary table. As we aim to avoid distributed transactions, Parti-SQL uses ATC internally to update all secondary indices after the main table has been modified. Since ATC does not provide isolation, an alternative index is not always in sync with the main table. Parti-SQL alleviates this problem by updating all secondary indices of a table in a single fixed order. Parti-SQL exposes each secondary index under a distinct table alias so that a programmer can be aware that she is performing lookups using a secondary index that might be slightly out-of-sync with the primary table.

### Avoid distributed joins.

With the full power of SQL, one can write arbitrarily complex relational queries. For example, in the auction service, one might want to query the database to find all item listings for “Nikon camera” whose sellers reside in California. This query can be accomplished via a single SQL statement which “selects” based on the location attribute in the `user-table` and the item description attribute in `item-table` and “joins” the results together using `user.id`. This type of join query is expensive even for a centralized database. Supporting such queries in a distributed setting requires generating and executing general distributed dataflow graphs [14, 19, 20], which is both resource intensive and time-consuming. In practice, performance-sensitive web applications shun away from using such queries. They do so by de-normalizing certain tables. Additionally, whenever the search functionality is needed, web applications almost always employ a separate search engine to handle search queries at real time [34].

Avoiding complex relational queries in databases used by latency-sensitive web applications exemplifies the recent argument that there is no high-performance one-size-fits-all database [33, 35]. Parti-SQL is designed to help web application scale. As a result, we only aim to support a subset of SQL involving index-backed lookups and updates.

### 3 Programming Interface

Parti-SQL’s programming interface is exposed to users as a small set of SQL extensions. These extensions serve three broad functions: (a) help users manage and access partitioned tables automatically (§ 3.1), (b) allow users to specify ATCs that modify multiple table partitions as an atomic-unit (§ 3.2), and (c) help users create and use
In this example, the user-table is partitioned by explicitly specifying three partitioning attributes: (1) the column used for deciding which partition a row of the table belongs to, (2) the total number of partitions, and (3) the partition function mapping a column value to its corresponding partition number. Figure 2 shows the partitioned table schema for the example auction service in Section 2. In this example, the user-table is partitioned by the column user-id into 1000 total partitions and a hash function is used to determine how to map a row to its partition based on the corresponding user-id. To fully utilize all machines, one should specify more partitions than the total number of machines to be used for storing the table.

In addition to table definitions, users also need to write a table configuration file. For each given table, this configuration file specifies the set of machines to be used for storing the partitions of that table. Our design separates the mapping of table partitions to machines from the mapping of table rows to partitions. Doing so allows users to change the table configuration during runtime to add or remove a machine from an existing configuration. By contrast, Parti-SQL does not support changing the table partition definition on the fly; the entire system must be shut down in order to re-partition an existing table.

Access tables. In order to perform a lookup or update in partitioned tables, users annotate the corresponding SQL select or update statement to indicate which table partition is being accessed. Specifically, one needs to specify the corresponding table and the partition key, as shown by the annotated select and update statements in Figure 3. It is possible to automatically generate the desired annotations for simple SQL statements like those in Figure 3 based on the partitioned table definitions. However, for arbitrary general SQL statements, there can be ambiguities as to which table partition the access is intended for. Thus, we have decided to let users explicitly specify the table partition to be accessed by a given SQL statement.

3.1 Create and access partitioned tables

Create and configure tables. Users define a partitioned table by explicitly specifying three partitioning attributes: (1) the column used for deciding which partition a row of the table belongs to, (2) the total number of partitions, and (3) the partition function mapping a column value to its corresponding partition number. Figure 2 shows the partitioned table schema for the example auction service in Section 2. In this example, the user-table is partitioned by the column user-id into 1000 total partitions and a hash function is used to determine how to map a row to its partition based on the corresponding user-id. To fully utilize all machines, one should specify more partitions than the total number of machines to be used for storing the table.

In addition to table definitions, users also need to write a table configuration file. For each given table, this configuration file specifies the set of machines to be used for storing the partitions of that table. Our design separates the mapping of table partitions to machines from the mapping of table rows to partitions. Doing so allows users to change the table configuration during runtime to add or remove a machine from an existing configuration. By contrast, Parti-SQL does not support changing the table partition definition on the fly; the entire system must be shut down in order to re-partition an existing table.

Access tables. In order to perform a lookup or update in partitioned tables, users annotate the corresponding SQL select or update statement to indicate which table

CREATE TABLE user-table
    PARTITION_USER_ID user-id
    PARTITION_SIZE 1000
    PARTITION_METHOD HashPartitioner

{ user_id integer,
  alias varchar(16),
  location varchar(16),
  bid_item integer,
  bid_price integer,
  sell_item integer,
}

Figure 2: Create a partitioned user-table. Parti-SQL keywords are highlighted in boldface.

SELECT * FROM user-table WHERE user_id = 123,
PARTI_LOC = user-table:123
UPDATE user-table SET location = "california" \nPARTI_LOC = user-table:540

Figure 3: Annotate SQL statements to access partitioned user-table.

-- a transaction chain to record a user's bid
-- in both user-table and item-table
BEGIN_ATC placeBid(UserID integer,
ItemID integer,
BidVal integer)
BEGIN
FUNCTION storeUserBid() PARTI_LOC = user-table:UserID
BEGIN
  UPDATE user-table SET bid_price = BidVal, \n  bid_item = ItemID \n  WHERE user_id = UserID;
END
-- if a subsequent transaction fails, Parti-SQL will
-- invoke counterstep ATC_COMPFUNC storeUserBid()
FUNCTION updateHighBid() PARTI_LOC = item-table:ItemID
BEGIN
  SELECT * INTO x FROM item-table WHERE item_id = ItemID;
  IF x.high_bid < BidVal THEN
    UPDATE item-table SET high_bidder = UserID, \n    high_bid = BidVal \n    WHERE item_id = ItemID;
  END
END_ATC

Figure 4: Use a custom ATC to record a user's bid in two partitioned tables: user-table and item-table.

3.2 Define and use ATCs

Users specify an ATC consisting of transactions $T_0, T_1, \ldots$, as a collection of database stored procedures. $T_0, T_1, \ldots$ will be executed in serial. Parti-SQL guarantees that either all $T_0, T_1, \ldots$ finish execution successfully or if any of the transactions cannot execute successfully, appropriate countersteps are taken to revoke a partially executed chain.

Figure 4 gives an example user-specified ATC (placeBid) in Parti-SQL. When a user places a bid, the auction application needs to store the bid with the user in the user-table as well as update the highest bid with the corresponding item in the item-table. Since this operation involves two different partitions in two different tables, the ATC (placeBid) is used to ensure that both steps of the operation are performed reliably in an atomic unit.
CREATE TABLE item-table
 PARTI_BY item_id 
 PARTI_SIZE 1000 
 PARTI_METHOD HashPartitioner 
{
 item_id integer,
 description varchar(5000),
 high_bid integer,
 high_bidder integer,
 PARTI_INDEX_BY high_bidder DUPLICATE high_bid
}

Figure 5: (a) Specify a secondary indice when creating a partitioned table. (b) Users are only allowed to directly modify the main table. However, users need to explicitly specify an auxiliary index table for lookups.

As shown in Figure 4, an ATC is defined as a series of stored procedures (storeUserBid and updateHighBid), grouped together by Parti-SQL keywords BEGIN_ATC and END_ATC. In this example, neither storeUserBid nor updateHighBid could fail. Hence, the entire chain always executes successfully to completion and there is no need to specify any compensating transaction. In the scenario that a transaction other than the first one in the chain (i.e. \( T_i, i > 0 \)) might fail, one needs to include a stored procedure with keyword ATC_COMPFUNC as a counterstep for every transaction earlier in the chain. If \( T_i \) fails, Parti-SQL ensures that all countersteps for earlier transactions \( T_j, j < i \) will be executed. As the last statement in Figure 4 shows, when invoking an ATC, users can specify how many steps the execution should complete before returning control back to the application. By default, an ATC invocation returns as soon as the first transaction in the chain has finished.

### 3.3 Create and use multiple indices

#### Create secondary indices
To speed up various types of lookups, users may want to associate multiple secondary indices with a given table and to also duplicate certain columns of the original table in the secondary indices. For example, a common lookup query in the auction service is to list a given user’s current set of high bids on various items. To satisfy this query quickly, one needs to construct a secondary index on *item-table* using the column *high_bidder*. Figure 5(a) gives an example of how to specify such a secondary indice when defining the partitioned *item-table*. Using keyword PARTI_INDEX_BY, we specify a secondary index keyed by column *high_bidder* with the additional duplicated column *high_bid*. The indice is stored in an auxiliary table called *item-table#high_bidder*. Parti-SQL partitions table *item-table#high_bidder* by its index key *high_bidder*. By default, it uses the same partition function and generates the same number of partitions for the index table as its main table.

Users are not allowed to directly modify an auxiliary index table. Rather, Parti-SQL automatically updates each auxiliary index table in response to each modification on main table. Parti-SQL internally uses an ATC to group the modification to the main table as well as the corresponding updates to each of the auxiliary tables. All updates are performed in one specific order: the main table is modified first, followed by updates to the first auxiliary table, then updates to the second auxiliary table etc.

#### Lookup using secondary indices
Although users cannot directly update secondary indices, they are required to explicitly access a particular auxiliary index table to perform lookups based on secondary keys. For example, in order to lookup all items for which user with identifier 123 is the current high bidder, one issues the SQL statement in Figure(b) 5.

The reason for exposing secondary indices to users for lookups is that auxiliary index tables are not always in synchronization with the main table. As an auxiliary index table is updated in a non-ACID fashion using ATC, its content might slightly lag behind that of the main table. Users need to carefully structure an application so that the inconsistency observed by lookups between auxiliary and main tables does not cause correctness problems. By exposing secondary indices with distinct auxiliary table names, we essentially force users to be aware of the lag between the primary and secondary indices.

### 4 System Design

#### 4.1 Overview
A typical Parti-SQL deployment consists of one master, many server processes (one per machine) as well as a large collection of clients linked into application servers. Figure 6 illustrates the overall interaction between different entities in Parti-SQL.

Users supply table definitions and the table configuration file to the master. The master decides how different individual table partitions are assigned different servers and instructs each server to create its assigned table partitions. Each server runs a local database instance, e.g. PostgreSQL in our implementation, to store its assigned table partitions and additional server-side persistent state.
We rely on replication software that comes with existing databases [32] to ensure the availability of each table partition in the face of a server failure.

Each Parti-SQL client fetches the mapping of table partitions to servers from the master and caches the information locally. A client parses Parti-SQL extensions from application SQL statements and determines which server to forward the request to using its local mapping cache. In Parti-SQL, each server holds a lease from the master on its set of assigned table partitions and always checks that a received request is indeed intended for one of its partitions. Therefore, client-side cached mappings need not always be up-to-date to ensure correctness. Similarly, each server also caches the master’s partition mappings so that it can directly contact other servers to execute the next step in an ATC.

In the subsequent subsections, we explain how Parti-SQL executes an ATC.

4.2 Execute ATCs

ATC provides programmers with the guarantee that the entire chain of transactions will be executed eventually. Since each ATC consists of multiple transactions each accessing data partition on a different machine, the main challenge in executing ATC is to ensure that every transaction in a chain is executed in the right order on its intended server. Furthermore, each transaction must be executed exactly once in spite of intervening failures.

In order to move the execution of an ATC from one server to the next, Parti-SQL stores the corresponding ATC statement as well as its current state of execution (i.e. what steps it has finished so far) in local “message queues” maintained by each server along the chain. Message queues are stored as a table in a server’s local database, as seen in Figure 7. Doing so allows Parti-SQL to take advantage of existing database replication and the ACID transactions available to a local database.

Suppose the client is to invoke an ATC consisting of transactions \( T_0, T_1, T_2 \), as shown by the example in Figure 7. This chain needs to execute on servers \( s_0, s_1, s_2 \) in serial, where \( s_i \) is responsible for the table partition accessed by transaction \( T_i \). The client sends the ATC to the first server \( s_0 \) which assigns it unique identifier, applies the first transaction destined for itself in the chain and stores the remaining chain in its outgoing queue. The execution of \( T_0 \) and enqueuing of \( [T_1, T_2] \) are done in a single ACID transaction with the local database so that an intervening server failure does not cause the loss of the ATC. Once \( T_0 \) finishes, server \( s_0 \) notifies the client of the completion of the first step in the chain.

The network thread of \( s_0 \) periodically examines its outgoing queue and transfers the queued chain to the incoming queue of its next server \( s_1 \). This transfer needs to happen reliably and without duplication. To do so, \( s_0 \) associates a monotonically increasing message sequence number with each ATC and \( s_1 \) checks to ensure that its incoming queue does not contain duplicate messages from \( s_0 \). The worker thread of \( s_1 \) periodically examines \( s_1 \)’s incoming queue. In a single ACID transaction, the worker thread applies the corresponding step \( T_1 \), moves the remaining chain to \( s_1 \)’s outgoing queue and marks the chain as finished in the incoming queue. \( s_0 \) notifies \( s_1 \) of the largest sequence number that it knows \( s_1 \) has received so that \( s_1 \) can garbage collect the finished chains in its incoming queue.

4.3 Maintain multiple index tables

For each table definition specifying an alternative index, the master creates a separate auxiliary index table. The index table’s primary key corresponds to the desired secondary index key and its columns mirror the main table’s primary key and other additional fields as specified by the user. For the example table
definition in Figure 5(a), the master creates two tables, item-table and item-table#high_bidder. The auxiliary item-table#high_bidder table’s primary key is high_bidder and it contains two additional columns item_id and high_bid, mirrored from the main table item-table. The master partitions the item-table#high_bidder table based on high_bidder and assigns the resulting partitions to different servers.

Users are only allowed to directly perform updates on the main table. Parti-SQL relies on its use of ATCs internally to propagate any modification on the main table to its auxiliary index tables. For an example update in Figure 5, the client-side Parti-SQL library generates an ATC of length 2. In the first transactional step, the ATC updates the main table item-table according to the original SQL statement and fetches the corresponding modified table row. In the second step, the ATC updates the auxiliary table item-table#high_bidder using the corresponding fields in the modified row. The two steps will be executed by two different servers responsible for modified partitions in the main and auxiliary tables.

Parti-SQL generates ATCs that follow a fixed order of modifications, updating the main table (t#0) first, followed by the first auxiliary table (t#1), followed by the second auxiliary table (t#2) etc. Doing so helps a user better reason about the lag between the main and auxiliary tables; specifically, the state of t#2 lags behind t#1 which further lags behind the main table t#0.

It is not sufficient that all derived ATCs on a given table perform modifications in a fixed order. In particular, undesired anomalies can occur if two ATCs on the same set of tables interleave arbitrarily. For example, suppose there are two updates U1, U2 on the item-table: U1 sets the high bidder of item-555 to user-123 with price $100 and the subsequent U2 sets the high bidder of the same item to user-549 with price $125. Although ATCs of both updates modify item-table before item-table#high_bidder, it is possible that U1 modifies item-table before U2 (thus U2 comes with a higher bid price) and U2 modifies item-table#high_bidder table before U1. Such interleaving causes the same information about item-555 in item-table to permanently diverge from that in item-table#high_bidder: in item-table, the high bidder is user-549 with price 125 while in item-table#high_bidder, the high bidder on the same item is user-123 with price 100. Parti-SQL avoids these potentially anomalous interleaving by executing all ATCs on the same group of tables (t#0, t#1...) in the same order at every table in the group. When server s0 responsible for update in the main table t#0 first receives a derived ATC from the client, it assigns the ATC a monotonically increasing sequence number, t:seqno. For each server responsible for partitions in a derived table t#i, it aims to execute the corresponding derived ATC in serial without any gaps in the sequence number space. This ensures that any two ATCs on the same group of tables follow the same execution order at all servers.

5 Application Experience

5.1 Porting RUBiS

We converted an existing PHP implementation of the RUBiS benchmark. We used a deployment suitable for the PostgresSQL database. We used the Apache web server for servicing requests and connecting to both the database and our implementation of ATC. All together we made less than 10% changes to the original application source code.

When porting RUBiS we followed several steps which, we believe, constitute a general scheme for porting applications to use Parti-SQL:

• Decide which tables should be horizontally partitioned.
• Identify secondary indices lookups.
• From the above two steps, write the Parti-SQL schema.
• Identify transactional updates and break them down to smaller transactions by partition.
• Chain the original updates as ATCs.

While we list these steps as separate logical steps, a lot of the conversion work can be combined. We modified only 341 lines of PHP code out of the original 2643 lines of RUBiS code. We describe the steps as they were applied to the RUBiS application:

Horizontal partitioning. This general step isn’t directly inherent to Parti-SQL but is needed for achieving large scale. A large-scale deployment of RUBiS needs to support many users (both sellers and bidders). As the number of users grows, so does the number of items offered for sale, the number of bids in the system, and the number of comments posted. Hence we chose to horizontally partition the users, items, bids, and comments tables. We also partitioned the buy_now table which stores items that were bought by users who’s offers met the immediate sale price.
Handling secondary indices lookups and creating a schema. We scanned RUBiS looking for all lookups involving secondary indices. For example, in RUBiS, there are lookups on items by the seller id. Another example is lookups on bids by item. For each of these lookups we included a PARTI_INDEX_BY directive in our Parti-SQL schema, to generate secondary tables. For simplicity, we chose to duplicate all columns of the master table, even if not all are needed for satisfying current queries. At this point we also changed the secondary indices SELECT statements to access the correct tables by adding the required PARTI.LOC directives. Changing the original RUBiS schema involved only adding the Parti-SQL extensions to the original table definitions.

Changing transactional updates to use ATCs. We next looked for all transactional updates in RUBiS which include registering a user, making an item available for sale, placing a bid on an item, storing a comment, and buying an item. We replaced each transaction with ATC clauses. We created the ATC by splitting each of the above original 5 transactions into smaller transactions. Our longest ATC chain corresponded to the bidding operation and constituted an ATC of length 6. We coded our stored procedure using the native Postgres PL/pgSQL language. It is also possible to code the stored procedures in variants of Tcl, Perl, and Python.

The entire port plus debugging of RUBiS was done by two students in under three days.

6 Implementation

7 Evaluation

We evaluated ATC and Parti-SQL with the RUBiS benchmark to explore the benefits of ATC.

We demonstrate the following about ATC:

- Very string Point
- interesting point
- Strong point
- Best point

7.1 Testbed Setup

Our testbed consists of 12 machines: six have one quad-core Intel Xeon X3360 (2.83GHz) processors with 4GB of memory and the rest have two quad-core Intel Xeon E5520 (2.27GHz) processors with 8GB of memory. All of our testbed machines are connected by a gigabit ethernet switch.

We present three areas of evaluation: microbenchmarks of our queuing and messaging services, end-to-end ATC performance when partitioning data across multiple database machines, and end-to-end performance comparisons across three implementations of a web application. For the last experiment, we compared the RUBiS application across three implementations: a slightly modified default single database, two phase commit, and our ATC. For all of these experiments, we used the PostgreSQL database for our storage backend.
requests.

ATC’s latency vs Two Phase commit’s. We compare the latency and throughput of ATC and two phase commit under a highly conflict operation of Rubis: StoreBid. When a user places a bid on an item, Rubis executes StoreBid operation to update the number of bids, the current maximum bid value of the item, and insert the new bid into the bids tables. Many users bid on the same item will create conflict updates on the items record. The ATC implementation of the StoreBid operation consists of six transactions: three updates on the item record at three item tables (one primary and two), and three inserts of the new bid record to three bid tables (on primary and two secondary). A two phase commit execution of this operation requires contacting six tables which potentially are at different machines.

We construct scenarios under which many users compete in placing bids on the same item. Each user place 10 bids in total. A user places a new bid after an average thinking time of 5 second from the previous bid.

Our client emulates 100 (and 250) users compete in placing bids on the same item. The backend database consist of 6 machines. Table 1 shows the average, median and 99% percentile of latency of ATC and Two Phase Commit protocol.

Figure 9 shows the cdf of latency of ATC and Two Phase Commit in the 250 users bidding scenario.

7.3 Baseline

We now discuss the single database measurements that we performed to establish the limits of our PostgreSQL

Compared to the unpartitioned database scenario, we see that ATC increases the capacity of the application in operations per second.
7.4 Scaling

8 Related Work

Partitioned, distributed database. The late eighties and early nineties have seen a flux of partitioned database proposals, such as Gamma [19], Bubba [14], R* [27], Teradata and Tandem [20]. These distributed database systems were designed to exploit the aggregate compute/storage capacities of a small collection of networked workstations. All of them aim to provide the same transactional update and query interface that were well established in centralized database systems. Specifically, they provide distributed transactions through two phase commit [14, 19, 27] and employ a distributed dataflow graph to execute a complex relational query across many machines. Furthermore, most work in this era emphasize on the optimization of complex distributed query execution [14, 19, 20].

The modern day equivalent of Gamma, Bubba and Teradata are the assortment of distributed data warehousing or big data analytics engine, such as Greenplum [4], Vertica [8], Aster Data [3] and ParAccel [7]. These distributed database systems are mainly designed for executing batched read-only queries for analyzing large amounts of business data. In the systems community, such data analytics are typically done usingMapReduce [18], Dryad [25] or DryadLINQ [38].

Unlike previously mentioned systems, H-store [26, 36] and VoltDB [9] (a commercial variant of H-Store) are two recent distributed databases specifically designed for the online transaction process (OLTP) workload. H-store is a main-memory database which partitions and replicates data in the memory of a collection of servers. H-base supports distributed transactions using two-phase commit. These transactions can degrade the overall transaction processing throughput, as shown in [26].

In [22], Garcia-Molina has proposed dividing operations into a series of smaller steps and countersteps for execution in a distributed environment. The main goal of his work is to allow users to exploit their semantic knowledge in an organized fashion to enable more concurrency in the system. The notion of ATC is inspired by these ideas, however, our goal is to avoid distributed transactions in a partitioned, distributed database.

Non-relational, distributed storage systems. There have been lots of work on large-scale distributed storage systems providing various noSQL interfaces, e.g. BigTable [15], HBase [5], MongoDB [6], Megastore [13], SimpleDB [11], PNUTS [16]. These systems either support very limited transactions (e.g. BigTable and PNUTS only support single-row transactions) or no transactions at all. Sinfornia [10] is an object store that supports mini-transactions using two-phase commit and is targeted to be used by distributed systems infrastructures as opposed to user applications. Recently, Percolator [29] provide general distributed transactions on top of BigTable. However, unlike Parti-SQL, Percolator is designed to run Google’s search index pipeline instead of user-facing web applications. As a result, Percolator needs not worry about meeting strict latency requirement for its operations.

Workflow Management [37]. These systems are designed for application workflows such as travel planning or insurance claim processing that naturally consist of multiple activities, each of which is a transaction. Like ATC, workflow systems guarantee that the series of activities are executed completely and exactly once.

Workflow systems employ a centralized queue manager to maintain recoverable execution state of an ongoing workflow in a persistent message queue. With colocated with a centralized database, the queue message can perform enqueue and dequeue operations atomically with the execution of each activity in a local transaction. If workflow management spans across several database servers so the queue manager cannot be co-located with all servers, distributed transactions are used to ensure the atomicity of enqueue/dequeue operations and the actual execution of an activity in a database server.

Although the abstraction of ATC is inspired by workflow systems, our motivation is quite different. While workflow systems are meant to manage sophisticated application workflows, Parti-SQL uses ATC to allow programmers to decompose a single logical operation into multiple transactions each of which can execute exclusively on a partitioned database. Parti-SQL also internally uses ATC to update multiple indices for the same table without distributed transactions. Therefore, the key benefit of ATC in Parti-SQL is to avoid the need for distributed transactions in a partitioned database.

Spheres of Control In [23], Gray discusses the concept of Spheres of Control (SoC), originally proposed in [17]. SoC is concerned with computation on a hierarchy of abstract data types. To make the whole computation appear atomic to the outside world, processes can dynamically create a SoC to commit all actions accessing data or to encompass other existing SoCs. The overall system keeps track of process dependencies to redo or invalidate affected computations in case of a problem. The notion of SoC is very general and no system has been built to exploit it in full. By contrast, our notion of ATC is much simpler in that it only aims to execute a chain of transactions (or actions) to completion in a distributed environment.
9 Conclusion

In this paper, we propose a new abstraction, the Atomic Transaction Chain (ATC), which enables programmers to reason about the all-or-nothing atomicity of a series of transactions while relaxing transactional isolation guarantees. We have also introduced Parti-SQL, a partitioned database capable of scaling a database to a large collection of machines by leveraging ATCs.

References

Queue, 2005.


