Persistent Client-Server Database Sessions

Roger S. Barga, David B. Lomet, Thomas Baby, and Sanjay Agrawal

Microsoft Research, Microsoft Corporation
One Microsoft Way, Redmond, WA 98052
barga@microsoft.com

Abstract. Database systems support recovery, providing high database availability. However, database applications may lose work because of a server failure. In particular, if a database server crashes, volatile server state associated with a client application's session is lost and applications may require operator-assisted restart. This prevents masking server failures and degrades application availability. In this paper, we show how to provide persistent database sessions to client applications across server failures, without the application itself needing to take measures for its recoverability. This offers improved application availability and reduces the application programming task of coping with system errors. Our approach is based on (i) capturing client application's interactions with the database server and (ii) materializing database session state as persistent database tables that are logged on the database server. We exploit a virtual database session. Our procedures detect database server failure and re-map the virtual session to a new session into which we install the saved old session state once the server has recovered. This integrates database server recovery *and* transparent session recovery. The result is persistent client-server database sessions that survive a server crash without the client application being aware of the outage, except for possible timing considerations. We demonstrate the viability of this approach by describing the design and implementation of Phoenix/ODBC, a prototype system that provides persistent ODBC database sessions; and present early results from a performance evaluation on the costs to persist and recover ODBC database sessions.

1 Introduction

The Application Availability Problem Database systems support fault-tolerance and high availability by recovering quickly from system failures. However, recovery has been restricted to the database system and has ignored applications interacting with the database at the time of failure. Currently, those applications either fail, resulting in an application outage, or must cope with database failures for themselves, assuming they survive the database crash. The former compromises application availability and can increase operational complexity. The latter either severely restricts application flexibility or increases its complexity.

C. Zaniolo et al. (Eds.): EDBT 2000, LNCS 1777, pp. 462-477, 2000.

When an application fails because of a database system crash, organizations responsible for the application need to quickly bring the application back on line. In the enterprise-computing world, time is quite literally money and the high cost of downtime drives awareness. While database recovery ensures that the database state is consistent, an application retaining state across database transactions can have consistency requirements that are not captured at the database transaction boundary. Further, parts of the application state may be lost during a crash. Restoring and continuing application execution is all too frequently a very complex and time-consuming operational problem.

In some system configurations, an application can survive a database server crash. For example, when the application executes on a client machine while the database is on a separate server. This permits the application to include logic to deal with database crashes and hence avoid an application outage. However, handling errors is a very difficult part of getting applications right. Dealing with database failures at the application level is tedious and error-prone, even when the application stays alive.

Our Approach Coping with system failures to minimize application outages and to avoid application program complexity thus presents a very large problem for robust enterprise computing. Our goal is to simultaneously improve application availability and simplify application logic. We accomplish that by having the *system* take responsibility for application persistence across system failures. It is the *system* that acts to ensure that the application will be recoverable and be able to continue execution after a crash. The goal is to provide application persistence across system crashes without the application taking any measures to ensure its own survival. We call this *transparent* application recoverability.

The work we describe in this paper aims to provide application recovery at modest system implementation cost and at arm's-length from the database system and its recovery manager. This work focuses on providing transparent application recovery in a client/server context, but without introducing a substantial recovery manager or extensively modifying an existing database system's recovery manager. The advantage of this is that we can layer our middleware infrastructure on many SQL database systems, hence providing persistent applications for them all.

Phoenix/ODBC We have implemented the prototype system Phoenix/ODBC to explore the utility and practicality of persistent ODBC client-server database sessions. These sessions persist across database server failures. The application exploiting Phoenix/ODBC interacts with it as it would with a normal ODBC driver. And, except for issues of performance, which we discuss later, the application program does not detect a difference between Phoenix/ODBC and a database vendor supplied ODBC driver in the absence of a database system crash. Thus, Phoenix/ODBC is *transparent* during normal operation and in the absence of database system failures.

Should the database system server crash, i.e. the DBMS running on a separate server machine, the typical ODBC driver provides no graceful way for the application to cope with the server failure. Indeed, the SQL/CLI standard [ANSI SQL/CLI]

provides no defined semantics as to what should happen to an application should a SQL database system crash. At best, the application is notified of the server failure and must "pull itself together" to deal with the server failure. At worst, the application hangs waiting for ODBC to respond to its last (and interrupted) request. However, with Phoenix/ODBC, a SQL database system crash and recovery merely results in a delayed response to an ODBC request. Once the database system recovers, the application's request returns normally, and the application can continue its execution. Thus Phoenix/ODBC is also *transparent* during database system crashes in that the application is not even aware of the crash, and need take no special measures to deal with it.

Phoenix/ODBC performs three main functions to provide persistent database sessions:

1. It ensures that volatile session state will persist across server failures. Phoenix/ODBC materializes volatile session state on the database server as database tables. Phoenix/ODBC persists result sets, messages, and database updates that effect volatile session state. Phoenix/ODBC intercepts application requests going to the server and identifies session state that must be stored on stable storage, and then creates permanent tables on the underlying database to record this state. If the database system crashes, then the session state contained in these tables will automatically be recovered. Some state information is also saved on the client, but need not be persistent there because we are not protecting against client failures. This state permits the synchronization of recovered server state with the client state.

2. It provides a virtual database session to isolate client applications from database server failures. Clients connect to the Phoenix/ODBC virtual session, and Phoenix/ODBC maps that session to an underlying ODBC database session. Should a failure occur, the underlying ODBC database session is lost. However, once the database server has recovered, Phoenix/ODBC re-maps the virtual session to a new post-crash ODBC session. It associates the saved session state of the pre-crash ODBC session with this new post-crash session.

3. Fast recovery after a failure is critical for high application availability as short outages may go unnoticed by the human user. Phoenix/ODBC detects potential database server failures and, in the event the database crashed, transparently reinstalls database session state from which normal processing can resume. It re-syncs the recovered session state with the state of the client application, presenting the illusion that no failure has occurred.

The result is persistent client-server database sessions that can survive a server crash without the client application being aware of the outage, except for possible timing considerations. Equally important, from an implementation perspective, Phoenix/ODBC requires no changes to the database, native ODBC drivers, or client application. Phoenix/ODBC continues the trend of expending system resources to conserve more expensive and error-prone human resources.

This paper makes three primary contributions: 1) It describes a generic method for persisting database sessions. 2) It demonstrates that persistent database sessions are practical, based on the design and implementation of Phoenix/ODBC. 3) Finally, it presents initial performance results from our prototype implementation that suggests the response time cost of using Phoenix/ODBC is usually modest.

2 ODBC Background

ODBC (Open Data Base Connectivity) is a standard application programming interface (API) by which a client application accesses the data in the database. ODBC enables any application to access any database that supports this standard via SQL by using a standard collection of call points. Vendors support ODBC by providing a driver, a client stub, specifically designed for their particular database system. The ODBC driver's application interface adheres to the ODBC standard. All ODBC drivers support the same call points, regardless of what database engine is on the back end.

The ODBC driver interaction with the database system is customized to the specific database engine. This interaction requires the use of proprietary communication protocols, SQL language message formats, and result set representations. Using ODBC, applications don't have to be customized to or even aware of the proprietary aspects of accessing the data they are using. The driver makes any necessary translations in sending SQL statements to the DBMS and presents the results back to the application in a standard way, including standard return codes.

ODBC Components The ODBC infrastructure consists of two components that mediate between the client and database system storing the data, as follows:

- **Driver**:An ODBC driver is a dynamic link library that responds to calls an application makes to the ODBC API. If SQL statements from the application contain ANSI or ODBC SQL syntax not supported by the database, the driver translates the statements into database-specific SQL syntax and then passes the statement to the server. When the database responds with a result set, the driver reformats it in the reverse direction into a standard ODBC result set.

- **Driver Manager**: The ODBC driver manager is a Microsoft provided layer that manages the communication between the application and the appropriate ODBC driver. The driver manager selects the driver for a particular database when the application first connects to the server, then passes all ODBC function calls coming from the application to this ODBC driver. The driver manager also handles some ODBC function calls directly, and detects and handles some errors.

An Illustrative ODBC Client-Server Session To illustrate the use of ODBC interface and likely result were a database server failure to occur, we consider the following example, illustrated in Figure 1. Our database session is similar to those in

the TPC-H benchmark, and involves three tables: a master customer table, a detail orders table, and a summary invoice table. The task is to extract the appropriate records for a customer with the last name "Smith," find that customer's current orders, and then aggregate the order totals into the invoice summary table. This client application might be coded as follows:

1. Create an ODBC database session by opening a connection to a named database server, follow the standard protocol to log on the server, then issue a series of ODBC function calls to set application specific attributes on the database connection.
2. Submit an ODBC function containing a SQL statement to create a result set from the customer table (**A**) consisting of records with a last name of 'Smith'.
3. Issue fetch commands to retrieve records from result set until appropriate customer is found.
4. Submit an ODBC function containing a SQL statement to open a cursor on the orders table (**B**) for orders matching this customer's ID.
5. Issue fetch commands to retrieve all matching order detail records for this customer ID.
6. Calculate the aggregate of those order detail records.
7. Submit an ODBC function containing a SQL statement to update the invoices table (**C**) with the aggregate.
8. Issue an ODBC function call to close the connection to the database, terminating the ODBC session.

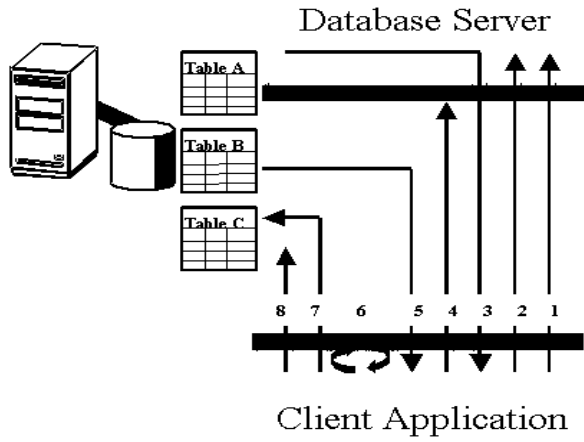

Fig.1. Client application using ODBC to access a customer order database.

Server Failure Behavior The ODBC standard leaves undefined what behavior the client application can expect should the database server to which it has a connection fail. Consider what the effects of a server failure might involve:

- **Detecting Server Failure:** ODBC functions may simply *"hang"* when the server fails. The user does not know whether the server is busy, the connection slow, or if a database failure has occurred. Once a database failure is verified the application must be terminated manually.

- **Application Availability**: Once a database server failure has been detected and the application terminated, the application must be restarted, a new ODBC session established, and new connections to the database opened. Partial execution can leave the application state confused, requiring long outages to reconstruct the application state. This too can require manual intervention.
- **Loss of Transient State.** If a failure occurs after the application has created volatile server state, for example result sets from SQL statements or temporary tables holding results or user input, this state is lost. Moreover, it may be impossible to reliably re-create this state because subsequent database updates have changed the subset of the data upon which the lost state depends.

Even should the client application be able to continue execution after a server failure, for an application to cope with the failure requires additional complex and subtle code to deal with these problems. This increases application complexity, delays application deployment, contributes to application bugs, and reduces overall application availability. This is where Phoenix/ODBC comes in.

3 Phoenix/ODBC

Overview Phoenix/ODBC provides persistent database sessions to client applications that interact with databases using ODBC. These sessions can survive a database server crash without the client application being aware of the outage, except for possible timing considerations. The design of Phoenix/ODBC is based on application message logging, in which all application requests going from the client to the database server are intercepted. Phoenix/ODBC records statements that alter session context, such as statement 1 in the example presented in the previous section. It rewrites selected SQL statements to create persistent database tables that capture the server's SQL session state, before passing the request on to the native ODBC driver (result sets of statements 2 and 4 will be made to persist). Phoenix/ODBC intercepts database server responses, variously caching, filtering, and reshaping result sets, and synchronizing with state materialized on the database server (partially delivered result sets in statements 3 and 5 are synchronized to provide seamless delivery). We exploit an ODBC virtual database session to isolate client applications from database server failures. When Phoenix/ODBC detects a database server failure it re-maps the virtual session to a newly created session into which, once the server has recovered, it installs the persistent state information of the pre-crash session. After the client application has successfully terminated, Phoenix/ODBC cleans up all persistent structures on the database server that were created to store database session state (statement 8), dropping all tables and stored procedures. Phoenix/ODBC integrates database server recovery *and* transparent session recovery. The result is persistent client-server ODBC database sessions that survive a server crash without the client application being aware of the outage, except for possible timing considerations.

Wrapping the ODBC API Phoenix/ODBC intercepts application requests and makes volatile session state persistent without requiring changes to the database

system, native ODBC driver, or any special application programming. Wrapping the ODBC API is the key to this end.

Our goal for Phoenix/ODBC was to provide persistent database sessions to client applications executing on Microsoft's Windows/NT, without the application itself needing to take measures for its recoverability. While the implementation specifics may differ, we believe our approach is also applicable to ODBC clients executing in other environments. We also wanted to avoid requiring modifications to either the database system or native ODBC drivers. Hence, the code for Phoenix/ODBC is completely contained within an enhanced Microsoft ODBC driver manager. As described in Section 2, the driver manager manages communications between an application and the ODBC drivers it uses. All application requests are first sent to the driver manager which then routes the requests to the native ODBC driver. Despite the variety of commercial database servers and available ODBC drivers, they are all managed on Microsoft systems by a common driver manager.

The *Phoenix-enhanced* driver manager wraps the call points of database vendor provided ODBC drivers in the same way as the original driver manager. It creates a surrogate for each function in the ODBC API that intercepts the client application ODBC request on its way to the ODBC driver. Actions that Phoenix/ODBC must take to analyze the request and provide for the persistence of volatile session state are performed in the surrogate, prior to passing the application request on to the native driver. This approach to providing server database sessions that survive system crashes is completely transparent to other system components. It requires no changes to native ODBC drivers, client application programs, or SQL database systems.

Decomposing and Persisting Application ODBC State The first step towards providing ODBC persistent database sessions is to decompose session state into separate elements, each of which can be managed as a distinct data object. These elements of session state have different lifetimes and recovery requirements that we exploit. Session state includes:

- **Session Context:** One element of this is the set of client specifiable attributes of the ODBC session, including the connect request, user login information, and default database settings. Database server specific information is also included, e.g., user identification, current database, user temporary objects, such as temporary tables and stored procedures, and unacknowledged messages sent by the server to the client.
- **Results of SQL Statement Execution:** SQL statements return one of the following: i) a result set for a SELECT statement; ii) a global cursor which can be referenced outside the SQL statement; iii) return codes, which are always integer values; iv) return messages for SQL updates; v) output parameters, which can be either data or a cursor variable;
- **Database Procedures:** Procedures can be stored at the server, consisting of one or more SQL statements that have been precompiled.

- **SQL Command Batch:** This is a group of two or more SQL statements, or a single SQL statement that has the same effect as a group of two or more SQL statements.

While there are subtleties to each element of an applications ODBC state, handling SQL results is particularly challenging. We describe how they are treated next in some detail.

Result Sets A result set is made persistent by being stored as a persistent SQL table. Seamless delivery of results is ensured by re-accessing this table after a failure, then re-establishing the place where pre-crash delivery was interrupted. When Phoenix/ODBC intercepts an application request, it performs a one-pass parse to determine request type. If the application request is a SQL statement that generates a result set, Phoenix/ODBC takes the following steps to ensure this result will be recoverable in case of server failure.

Step 1: *Phoenix/OBDC determines the structure of the SQL statement result set, i.e. the names of attributes, their types, and the order in which they appear in each returned tuple.* Since Phoenix/ODBC need only know the result format in order to create a persistent table for it, Phoenix/ODBC only requires the result set *metadata*. The metadata describes the number of columns in the result, the types of those columns, their names, precision, nullability, and so on. Because it is essential to minimize expensive network traffic between the server and client, we want to acquire this metadata with a single round trip to the server with minimum data transfer and with minimum server impact. To do this, Phoenix/ODBC appends the clause "WHERE 0=1" to the original SQL statement. It sends this modified query to the server via the native ODBC driver. ODBC delivers the result set metadata as a prefix to the delivery of the result data. This Phoenix/ODBC "trick" guarantees that the query will not be executed and that no result data will actually be returned, minimizing both server load and message size. Only query compilation is performed on the server, and only the metadata is returned in the reply message.

Step 2: *A table is created at the server to hold the result generated by the SQL statement.* Once a response is returned from the server indicating the rewritten SQL statement has successfully executed, Phoenix/ODBC reads the metadata and reformats it into a CREATE TABLE statement. It then sends the CREATE statement to the database server to create an empty persistent table at the server to hold the result set. This table is part of a special Phoenix database, it is not a temporary table.

Step 3: *The result set is stored in the just created persistent table at the server.* What is materialized depends on both the SQL statement and on how the application requests the results set from the server. With ODBC, the "how" is determined by the statement options specified prior to executing a SELECT.

Default Result Set When the ODBC default values are used as the options, the database server sends the result set in the most efficient way possible. The server assumes that the application will fetch all the rows from the result set promptly. Therefore, the database server sends all rows from the result set and the client

application must buffer any rows that are not used immediately but may be needed later. This is referred to as a default result set.

To materialize a default result set, Phoenix/ODBC sends the database server a request to execute the original SQL statement and stores its result in the persistent table created at Step 2 Phoenix/ODBC creates the following stored procedure to do this, using ANSI-standard SQL:

```
CREATE PROCEDURE P (@T string) AS
INSERT
  <original SQL statement>
INTO T
```

The advantage of using a stored procedure is that all data is moved locally at the server, not sent first to the client. It involves a single round-trip message from client to server; rather than having data moving across the network. The action is itself an atomic SQL statement, i.e. it executes in a separate transaction. Once the server has returned a response indicating the procedure was successfully executed, the result set is stable and will persist across server failures. Next, Phoenix/ODBC issues the SQL statement SELECT * FROM **T** to open the table and returns control to the application for normal processing.

Step 4: To ensure seamless delivery of the result set, Phoenix/ODBC keeps track of the current location in the now persistent result set. Should a failure occur, subsequent database recovery ensures the result set exists after the failure. Phoenix/ODBC resumes access to the result set at the remembered location of the last access before the database system failure.

Cursors The SQL language permits results to be returned in ways other than via default result sets. An application can control delivery of results at a finer granularity by exploiting *cursors*, two forms of which we discuss briefly: keyset cursors and dynamic cursors. An application's SQL query may return no rows, a few rows, or millions of rows. The user is unlikely to want to see millions of rows. Fetching and buffering millions of rows is usually a waste of time and resources. Server cursors, such as keyset and dynamic, allow an application to fetch results a block of rows at a time from an arbitrarily large result set. The application is permitted to "navigate" through the result set via a server cursor. A server cursor allows the application to fetch any other block of rows, including the next n rows, the previous n rows, or n rows starting at a certain row number. The server fills each block fetch request only as needed. Some server cursors also allow an application to update or delete fetched rows. If the row data changes between the time the SQL cursor definition statement is executed and the time the row is fetched by the application, the updated row data is returned. These features of server cursors present unique challenges for Phoenix/ODBC.

A *keyset cursor* captures the set of rows that satisfy a query at the time the cursor is opened and it permits those rows to be accessed and updated. If a keyset cursor is

requested, Phoenix/ODBC materializes only the keys of the result set rows in the persistent database table. When the application requests a row from the result set, Phoenix/ODBC reads the key from the table and SELECTs the record from the database using this key. If the row data has been changed or the record itself deleted, the updated row data is returned. Phoenix/ODBC transparently supports keyset cursors, but now the cursors persist across failures.

A *dynamic cursor* specifies a logical predicate (the *WHERE* clause) that defines the rows of interest. The set of rows fetched changes dynamically as rows are inserted and deleted. If a *dynamic cursor* is requested, Phoenix/ODBC again materializes the keys of the result set rows in a persistent database table, exactly as it did for keyset cursors. Now, however, when the next row is fetched, it is not necessarily the row with the next key as an insertion may have occurred. Thus, a fetch causes Phoenix/ODBC to use the last record key seen by the application and the next record key from the table to *SELECT* a *range* of rows from the database server. If records have been inserted into this range, Phoenix/ODBC fetches them and presents the appropriate row to the application. Again, Phoenix/ODBC transparently supports dynamic cursors, but now the cursor persists across failures.

Data Modification Statements "Results" A server failure can also occur while the application is attempting to modify data in tables using a data manipulation statement (`insert`, `update`, and `delete`). While there is no result set, there is state associated with a data manipulation statement, namely the number of tuples affected by the modification. In addition, for purposes of recovery it is necessary to determine whether the statement successfully executed or not; that is, we require *testable state*. Our approach is to intercept the application request and *wrap* a transaction around the data modification statement. Within this transaction we add an insert statement after the data manipulation statement to record the outcome (number of tuples affected) in a Phoenix-managed table. In the event of a server crash, Phoenix/ODBC can probe this table to determine the status of the application request; if the request completed Phoenix/ODBC simply returns the outcome logged in this table to the application, else Phoenix/ODBC resubmits the original application request.

Message "Results" When the database server commits a transaction and then fails before it can send a response to the client, that message will be lost to the client. Phoenix/ODBC prevents lost messages by including the transaction reply buffers in its persistent session context. The database server writes both the commit record and reply buffer to a Phoenix database table before committing the transaction and replying to the client. On database recovery, Phoenix/ODBC will deliver the reply buffer to the client, thus avoiding the lost message.

Temporary Objects SQL applications often declare temporary database tables to serve as program work spaces, and temporary stored procedures to execute the same task (sequence of SQL statements) several times in a session. In the event of a database server failure temporary objects would be lost. Our approach for persisting temporary objects is to intercept application requests to create temporary tables or temporary stored procedures, then rewrite the request to create a persistent table or

persistent stored procedure. Phoenix/ODBC records the name of all persistent tables and stored procedures created, and all subsequent references to the temporary object will be intercepted and redirected to the corresponding permanent object. Upon normal session termination, Phoenix/ODBC will explicitly remove these tables and stored procedures.

Virtual ODBC Sessions There are two notions of session within the ODBC framework. A client application has an ODBC session with which it interacts. On behalf of the application, ODBC establishes database sessions, called connections with database systems accessed during the ODBC session. When a server crashes, the database session does not survive the crash. The server's failure can also corrupt and hence lead to the termination of the client application's ODBC session as well.

To provide persistent sessions, Phoenix/ODBC insulates the client application from these underlying sessions. Instead, the client connects to a Phoenix/ODBC session. This is possible because Phoenix/ODBC is an extension of the Driver Manager and "wraps" native ODBC drivers. The Phoenix virtual session is mapped to an ODBC session. The ODBC session establishes a database session via an ODBC connection. This connection is identified by a *connection handle.* Phoenix/ODBC maps this connection handle to a virtual connection handle before returning it to the application. Should a crash occur, Phoenix/ODBC establishes a new connection with the database server and remaps the virtual connection handle to the handle for the new connection.

A Phoenix/ODBC session interrupted by a server crash will hence have two connections to the database at the crashed server, the pre-crash session and the post-crash session. The pre-crash session information is normally volatile and hence is lost when the server crashes. Phoenix/ODBC takes steps to materialize this volatile state as persistent tables at the server. Thus, the application sees a database session that is virtual in that the volatile state is trivial and the substantive state is mapped to persistent tables. After a crash, Phoenix/ODBC creates the post-crash database session, again with a trivial volatile state, and reconnects this post-crash session to the persistent tables built during the pre-crash session.

To mask Phoenix/ODBC activity required to (i) establish persistent tables on the server, (ii) to ping for server recovery, and (iii) re-create session state, Phoenix/ODBC establishes a private database connection. When an application interrogates its virtual connection, it only sees the activity on the connection to which its virtual connection is mapped, not the activity on the private connection. The mapped connection activity mimics the application's use of a normal ODBC connection.

Server and Session Crash Recovery Phoenix/ODBC detects server failures (i) by intercepting communications errors raised by the ODBC driver or (ii) by timing out application requests. Once a potential problem is detected, Phoenix/ODBC must re-contact the server. Phoenix/ODBC uses a private database connection to '*ping*' the server and periodically attempts to reconnect to the database. If after a period of time Phoenix/ODBC is unable to connect to the server, it assumes the database system crashed and passes the communication error on to the application.

If successful, Phoenix/ODBC must determine if the database system actually crashed or whether there is simply a communication failure or delay. We want to discover whether our database session, which will be erased in a crash, still exists. There is no explicit test for this. We test a proxy for this, i.e. we test whether a special temporary table created by Phoenix/ODBC for the session still exists. Temporary tables exist only within a session and are deleted when a session terminates for any reason.

Recovery of the ODBC virtual database session is separated into two phases. First, Phoenix/ODBC transparently reconnects the application to the database server and re-associates saved information with these new database connections. Phoenix/ODBC reinstalls each client connection to the database system using the original connection request and login information, then issues a series of calls to the database server in order to reinstall application specified ODBC connection options. Once complete, Phoenix/ODBC binds these new connections to the virtual database session. Second, Phoenix/ODBC reinstalls SQL state. Phoenix/ODBC first verifies that all application state materialized in tables on the server was recovered by the database recovery mechanisms. It then identifies the application's last completed request for each database connection and asks the server to re-send the result set if necessary; it can also resend any incomplete or interrupted SQL requests to the server. Once this step is complete, Phoenix/ODBC resumes normal processing of application requests. This creation of the new database session is masked from the application, giving the illusion of a single persistent database session.

4 Performance of Phoenix/ODBC

In this section we present early results from an ongoing performance evaluation of Phoenix/ODBC; complete results from this performance evaluation will be presented in a forthcoming paper. For this paper, our objectives were to (i) to measure the overhead of persisting session state on application performance, and (ii) measure the time required for Phoenix/ODBC to recover and reinstall a database session after server failure. For this evaluation we selected TPC-H, a current variant of the now obsolete TPC-D benchmark designed to test the performance of decision support queries in business environments. TPC-H defines queries and update functions that include a rich breadth of operators and selectivity constraints. The query suite, for example, ranges from a simple single-table query to a complex eight-way join query, while update functions carry out insert and delete. TPC-H also defines a benchmarking test, called the *power test*, suitable for measuring application performance.

TPC-H Power Test The TPC-H power test executes all queries and update functions defined in the benchmark one at a time in order and their running time is measured individually. This is intended to measure "raw query execution power". For this experiment we executed the power test fifty times for both native ODBC and Phoenix/ODBC and computed the average of these runs. The standard deviation of these runs is generally less than 1% of the mean.

Table 1 presents selected results from the TPC-H power test. Column one identifies the query and update function number, while the second column lists the number of tuples returned in the result or modified by an update. The third and fourth columns contain the running times using native ODBC and Phoenix/ODBC on the TPC-H database. For comparison purposes, the fifth column presents the difference between native ODBC and Phoenix/ODBC running times, while the final column displays the ratio of running times.

For query performance only (Total Query), the total run time of queries using Phoenix/ODBC is approximately one percent greater than when using native ODBC, which generates a temporary (volatile) result set. As we can see from measurements in Table 1, for these compute-intensive queries that produce a small result set this overhead is relatively small. We see an average overhead of just over one second for each query.

Table 1. Selected results from TPC-H Power Test using native ODBC and Phoenix/ODBC.

Query/ Update	Result Set/ Updates	Native ODBC seconds	Phoenix/ODBC seconds	Difference seconds	Ratio
Q01	4	106.297	107.706	1.4	1.013
Q02	100	4.775	5.262	.486	1.101
Q11	1048	16.372	17.967	1.594	1.097
Q16	18,000	18.868	18.034	-.834	.955
RF1	7,500	56.700	57.540	.84	1.015
RF2	7,500	335.600	336.970	1.37	1.004
Total Query		2277.556	2302.868	25.312	1.011
Total Updates		393.300	394.510	1.210	1.003

Refresh function RF1 inserts 1500 tuples into the ORDERS table and roughly 6000 tuples into the LINEITEM table to emulate the addition of new sales information, while refresh function RF2 deletes 1500 tuples from the ORDERS table and roughly 6000 tuples from the LINEITEM table to emulate the removal of stale or obsolete information. We decomposed each refresh function into two transactions, in which each receives one-half of the key range that is to be modified. The tuples corresponding to new orders and new lineitems were already loaded into the database, as were the keys corresponding to orders and lineitems to be deleted. Hence, the two transactions of refresh function RF1 submit a total of 4 insert requests to the server to insert tuples from these tables, while the two transactions of refresh function RF2 submit a total of 4 delete requests to the server to delete tuples that match these keys.

Phoenix/ODBC *wraps* insert and delete statement with a transaction, and within that transaction it records the number of tuples affected by the update in a Phoenix-managed table; this *status table* provides testable state for determining whether a statement completed. Thus, the primary overhead for data modification statements (insert, delete, update) is the creation of a transaction and a write to the status table to

record statement completion. As we can see in Table 1, the Phoenix/ODBC overhead for data modification statements is negligible.

Recovering a Database Session We also measured the time required for Phoenix/ODBC to recover a database session. To conduct this experiment we submit TPC-H query Q11 to the database and begin fetching tuples until we near the end of the result set, leaving a few tuples unread. Then we "*crash*" the server by terminating the database server. At this point the application is left waiting for the server to respond to its fetch request. We restart the server and measure the time required for Phoenix/ODBC to recover the session and respond to the outstanding fetch request.

Of interest in this experiment is: i) the time required to recover the virtual database session, and ii) the time required to recover the SQL state for active application requests. Together, these represent the time required for Phoenix/ODBC to recover an ODBC database session and continue application execution. Once Phoenix/ODBC is able to contact the server after a failure, it reconnects to the database, issues a series of ODBC function calls to reset connection options, and maps the new connections to the virtual database session. The time required to complete session recovery does not depend on the size of the result set, and in our client-server configuration this step required 0.37 seconds to complete for all experiments. Once the virtual session is reinstalled, Phoenix/ODBC then reinstalls SQL session state. It must first identify the client application's last outstanding request, in this case a fetch command, open the database table holding the result set and advance to the appropriate tuple.

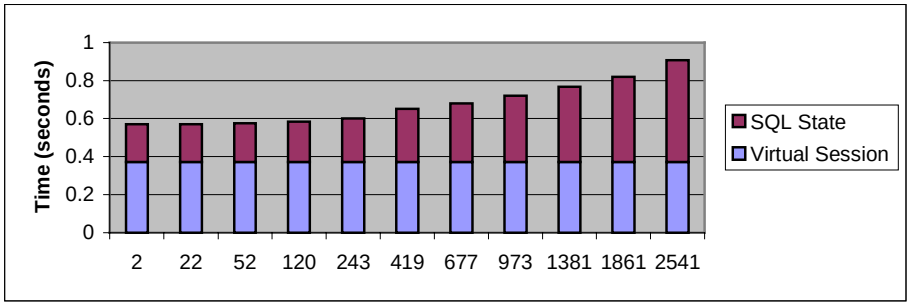

Figure 2. Elapsed time for session recovery over varying result sizes.

The results presented in Figure 2 are when Phoenix/ODBC re-positions the result set using a stored procedure that advances to a specified tuple, hence advancing through the result set on the server without passing tuples to the client. When the procedure completes, a table pointer is left in place and the next fetch request returns the desired tuple. Comparing Figure 2 with the total query Q11 response time plus the 10 seconds or so required to deliver the 2541 tuple results reveals that Phoenix/ODBC can recover an entire ODBC database session in less than a tenth of the time required to simply recompute query Q11.

5 Concluding Remarks

Status Phoenix/ODBC provides persistent database sessions to client applications across server failures, without the application itself needing to take measures for its recoverability. This offers improved application availability and reduces the application programming task of coping with system errors. We leverage existing database system recovery mechanisms by wrapping ODBC to capture application interactions with the database and logging application state changes as tables on the database server. We exploit the notion of a virtual database session in which an application interacts with a Phoenix/ODBC session that is in-turn mapped to ODBC and database sessions. Our procedures detect database server failure and re-map the virtual session to a new database session into which we install the pre-crash session state once the server has recovered. This integrates database server recovery *and* transparent session recovery. The result is persistent client-server database sessions that survive a server crash without the client application being aware of the outage, except for possible timing considerations. We implemented the Phoenix/ODBC prototype and have presented performance data on the costs to persist and recover ODBC database sessions, demonstrating the viability of the approach.

Initial results from our performance evaluation indicate the overhead to persist result sets for queries with a high degree of complexity, such as those found in the TPC-H benchmark, is modest. For the TPC-H power test the difference is approximately 1%, while for the update functions the difference is less than 0.5%. What is more, our evaluation demonstrates that an entire ODBC database session can be recovered in a fraction of the time required to recompute the original query and send its results.

Conclusion Using Phoenix/ODBC relieves the application developer from coping with the programming complexity of handling server failures, increases the availability of the application, and in many cases avoids the operational task of coping with an error. Any application can use Phoenix/ODBC to enhance database session availability without having to modify the application program, the original ODBC driver, or the database server. Indeed, a user of the application, end user or other software, may not even be aware that a database server crash has occurred, except for some delay. While there is an extra system cost for application persistence, Phoenix continues the trend of expending system resources to conserve more expensive and error-prone human resources.

References

[1] Bernstein, P., Goodman, N. and Hadzilacos, V. Recovery Algorithms for Database Systems. *IFIP World Computer Congress*, (September 1983) pp. 799-807.

[2] Kumar, V. and Hsu, M. (eds.) *Recovery Mechanisms in Database Systems*. Prentice Hall, NJ 1998

[3] Lomet, D.B. and Weikum, G. Efficient Transparent Application Recovery in Client-Server Information Systems. ACM SIGMOD 1998, Seattle, WA (June 1998) pp. 460-471.

[4] Lomet, D. Application Recovery Using Generalized Redo Recovery. *1998 Int'l. Conference on Data Engineering*, 154-163.

[5] Lomet, D. and Tuttle, M. Redo Recovery From System Crashes. 1995 *VLDB Conference*, Zurich, Switzerland, 457-468.
[6] Lomet, D. and Tuttle, M. A Formal Treatment of Redo Recovery with Pragmatic Implications. Available as Digital CRL Lab Technical Report.
[7] Transaction Processing Council. TPC Benchmark H. **http://www.tpc.org**, 1999.