Measuring and Optimizing a System for Persistent Database Sessions

Roger S. Barga David B. Lomet
Microsoft Research, Microsoft Corporation
One Microsoft Way, Redmond, WA 98052

Abstract

High availability for both data and applications is rapidly becoming a business requirement. While database systems support recovery, providing high database availability, applications may still lose work because of server outages. When a server crashes, volatile state associated with the application's database session is lost and the application may require operator-assisted restart. This exposes server failures to end-users and always degrades application availability. Our Phoenix/ODBC system supports persistent database sessions that can survive a database crash without the application being aware of the outage, except for possible timing considerations. This improves application availability and eliminates application programming needed to cope with database crashes. Phoenix/ODBC requires no changes to database system, data access routines, or applications. Hence, it can be deployed in any application that uses ODBC to access a database. Further, our generic approach can be exploited for a variety of data access protocols. In this paper, we describe the design of Phoenix/ODBC and introduce an extension to optimize response time and reduce overhead for OLTP workloads. We present a performance evaluation using TPC-C and TPC-H benchmarks that demonstrate Phoenix/ODBC's extra overhead is modest.

1. Introduction

High availability for data and applications is increasingly a basic requirement for large enterprises and small businesses alike. The goal of the Phoenix project at Microsoft Research [1] is to increase the availability of applications, greatly reducing the programming and operational tasks of coping with system failures. This requires more than simply increasing database server reliability. Some server failures are inevitable. We focus on increasing the ability of the rest of the system to deal with database server failures. We have built a prototype system, called Phoenix/ODBC, which provides persistent client-server database sessions that can survive a server crash without the client application being aware of the outage. This masking of database server failures means the application (i) does not need to take special measures to deal with server failures, and (ii) does not experience an outage (it pauses, it does not fail) any larger than the time it takes to recover the database system, usually seconds or minutes. This is an enormous improvement over manual methods of recovering applications when important application state is lost due to a database system crash, which can take hours, perhaps days.

Database systems have long dealt with system crashes in a robust manner. Indeed, the technology that enables recovery to the last committed database transaction is very mature and cost effective. Unfortunately, things are not so simple when dealing with applications. It is very difficult for a programmer to design applications that cope with a system crash. Either the form of the application program must be very tightly controlled, e.g. keep the application stateless, or counter-measures involving rather elaborate exception handling are needed to provide an adequate response to system crashes. In either case, the application programmer needs to be very much aware that system crashes are possible, and write the application with great care. The all too frequent outcome is that applications, in fact, fail when the database server crashes. This shifts the burden elsewhere. The burden then is manifested as user frustration and operational headache as potentially ad hoc manual efforts are made to restore the application to an acceptable state.

Coping with system failures to minimize application outages and to avoid application program complexity thus presents a very large opportunity to make applications more robust. To succeed in this, we have built a system that extends database technology to the data access parts of the system and enables us to mask failures from client applications. Our goal is to both improve application availability and simplify application logic. We accomplish this by having the system take responsibility for application persistence across server failures. It is the system that acts to ensure the session state is recoverable and the application can continue execution after a failure. Should a server failure occur, the system automatically reconnects to the database, restores session state and continues execution, masking the effects of failure from the client application. This persistent database session component, called Phoenix/ODBC, is implemented without specialized database system support, and requires no changes to client applications or native ODBC drivers.

1.2 This Paper's Contribution

The role of Phoenix/ODBC, as described in [4,5], is to improve application availability by masking database server crashes from a client in as cost effective manner as possible. Our focus in this paper is on quantifying and reducing the impact on application response time and database server throughput. We report on experiments that measure this impact, and introduce an optimization for reducing this impact for OLTP style workloads.

Our experiments measured the performance impact to persist result sets on the server for queries with a high degree of complexity, such as those found in the TPC-H benchmark. This impact is modest. For the TPC-H *power test* the impact is approximately 1%, while for update functions it is less than 0.5%. Results from the TPC-H *throughput test* were encouraging, suggesting impact on database throughput is minor. Our experiments also demonstrate that a database session can be recovered in a fraction of the time required to recompute a single query and send its results to the client. These results confirm it is possible to transparently provide persistent database sessions, hence masking server crashes, without compromising performance.

High availability is particularly valuable in transaction processing. Transaction processing workloads are usually characterized by simple queries (and updates) generating small result sets. Phoenix/ODBC overheads are not large, particularly as a fraction of the cost of generating large results. Yet even small absolute costs make significant percentage differences for short queries. When we measured overheads for simple TPC-C queries that generate small results, response time ratios and server overheads were significant. This led us to specialize the handling of small result sets, which is the major overhead for Phoenix/ODBC. We expected the optimization, which we call client result caching, to essentially eliminate the overhead for OLTP style applications. Results from our experimental evaluation confirmed this.

2 Phoenix/ODBC Description

Two factors influenced our strategy for providing persistent database sessions. First, we wanted to leverage existing database system functionality to persist session state and, in the event of a failure, to automatically recover the database session. Second, we wished to preserve *transparency* by seamlessly integrating persistent database session mechanisms into normal ODBC data access interactions, without requiring changes to the database, application or native OBDC drivers.

Figure 1 illustrates the overall system architecture, from user to database server, and in particular, where Phoenix/ODBC fits in. The diagram illustrates a client application using ODBC (and, transparently, Phoenix as part of the data access library) to connect to a database. The client session is with Phoenix/ODBC – it creates the database session for the application.

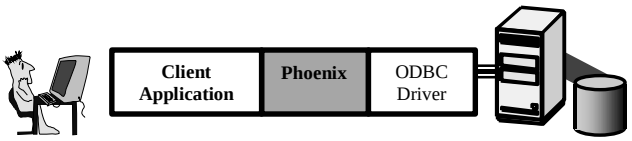

Figure 1: Phoenix-enhanced ODBC driver manager.

Phoenix/ODBC acts as a wrapper for a vendor supplied ODBC driver. In a Microsoft client, Phoenix/ODBC is integrated into the ODBC Driver Manager. The driver manager manages communications between a client application and the variety of commercial ODBC drivers. Application requests are first sent to the driver manager, which then routes the request to the native ODBC driver.

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. All ODBC calls and the database replies are intercepted by and massaged appropriately. Phoenix/ODBC performs three main functions in providing persistent database sessions. To make this paper self-contained, we present an overview of each function in the following subsections. Additional details can be found in an earlier paper describing the design of Phoenix/ODBC [5].

2.1 Persisting Volatile Session State

Actions that Phoenix/ODBC must take to analyze the application request and provide for the persistence of volatile session state are performed by Phoenix, prior to passing the request to the native driver. This approach to providing database sessions that survive system crashes is completely transparent to other system components. It requires no changes to the native ODBC drivers, client application programs, or SQL database systems.

We decompose session state into separate elements, which have different lifetimes and recovery requirements that we exploit. Session state includes:

- **Session Context:** Client specified attributes, including the connect request, user login information, and default database settings. Database specific information, such as user id, current database, temporary objects (tables, procedures), and messages sent by the server to the client.
- **Results of SQL Statement Execution:** A SQL statement will return one or more of following:
  - A result set for a SELECT statement;
  - Global cursor, referenced outside the SQL statement;
  - Return codes, which are always integer values;
  - Return messages for SQL updates.
  - Output parameters, either data or a cursor variable;
- **Database Procedures:** Stored at the database server, consisting of one or more precompiled SQL statements.
- **SQL Command Batch:** A group of two or more SQL statements, or single SQL statement that has the same effect.

While there are subtleties to each element of session state, handling SQL results is particularly relevant to this paper so we describe how they are treated in detail.

Result Sets

Phoenix/ODBC makes result sets persistent and ensures seamless delivery to a client application. A result is made persistent by being stored as a persistent table. Seamless

result delivery is ensured by re-accessing this table after a failure, repositioning to where delivery was interrupted. When Phoenix/ODBC intercepts an application request, it performs a one-pass parse to determine request type. If the request is a SQL statement that generates a result set, it takes the following steps to ensure the result set will be recoverable in case of server failure.

1. *Phoenix determines the structure of the result set, i.e. names of attributes, types, and order in which they appear.* Phoenix/ODBC acquires this metadata with a single request to the server by appending the clause "WHERE 0=1" to the original SQL statement. It sends this modified query to the server via the native ODBC driver. This "trick" guarantees the query will not be executed and no result data is returned. Only query compilation is performed on the server, and only the metadata is returned in the reply.

2. *A table is created at the server to hold the result generated by the SQL statement.* Phoenix/ODBC reads the metadata from the response and constructs a CREATE TABLE statement, sending the statement to the database to create an empty table at the server. This table is part of a special Phoenix database; it is not a temporary table.

3. *The result set is stored in the persistent table at the server.* What is materialized depends on both the SQL statement and how the application requests the result set from the server. With ODBC, the "how" is determined by statement options specified prior to executing a SELECT. Here we describe only *default result sets*.

The database server sends all rows in the default result immediately, and the client application must buffer rows until consumed (read). In contrast, Phoenix/ODBC sends the server a request to execute the original statement and store its result into the persistent table created at Step 2. It creates the following stored procedure to do this, using ANSI-standard SQL:

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

The stored procedure involves one round-trip message from client to server, and means the data is moved locally at the server, not sent first to the client. Once the server returns a response indicating the procedure is finished, the result set will persist across server failures.

4. *The result set is delivered "seamlessly" to the client application.* Phoenix issues the statement **`SELECT * FROM TName`** to open the table and returns control to the application for normal processing. This is illustrated in Figure 2. To ensure seamless delivery of the result set, Phoenix keeps track of the current location in the now persistent result set. Should a failure occur, 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 failure.

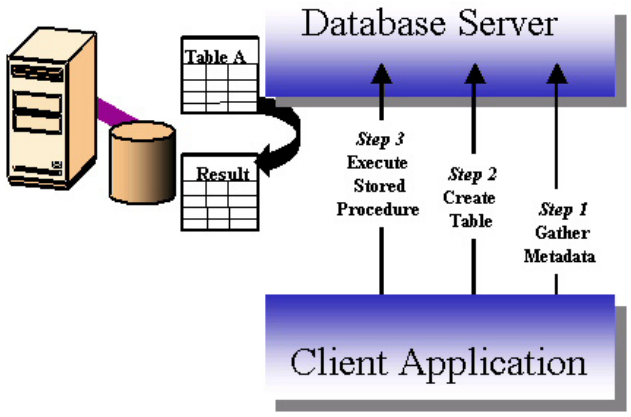

Figure 2: Steps to generate a persistent result set for a query.

SQL permits results to be returned in other ways, e.g. an application can control result delivery using *cursors*. In [5], we describe how Phoenix/ODBC persists cursors, and state associated with data modification statements, temp objects, and messages returned from the database.

2.2 Virtual ODBC Database Session

There are two notions of session that we need to distinguish. An application has an *ODBC session* with which it interacts. On behalf of the application, ODBC establishes connections (*database sessions*) with accessed database systems. 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 database sessions, we insulate the application from these underlying sessions. Instead, the client connects to a Phoenix/ODBC session only. This is possible because Phoenix/ODBC "wraps" native ODBC drivers, as illustrated in Figure 1. The Phoenix/ODBC session is mapped to a normal ODBC session. The ODBC session creates a database session via an ODBC connection. This connection is identified by a *connection handle*, which we map to a *virtual connection handle* before returning it to the application. Should a crash occur, Phoenix creates a new connection and re-maps the virtual connection handle.

A Phoenix/ODBC database session interrupted by a server crash will have two database sessions, the pre-crash session and the post-crash session. The pre-crash session information is volatile and hence 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 in persistent tables. After a crash, Phoenix/ODBC creates

the post-crash database session, again with trivial volatile state, and connects this post-crash session to the persistent tables built during the pre-crash session.

To mask activity required to (i) create persistent tables, (ii) ping for server status, and (iii) re-create session state, Phoenix/ODBC maintains a private database connection. When an application interrogates its virtual connection, it only sees it's own activity on the connection to which its virtual connection is mapped.

2.3 Automatic Recovery from Server Failures

Phoenix/ODBC detects database server failures (i) by intercepting errors raised by the ODBC driver or (ii) by timing out application requests. Once a potential problem is detected, it '*pings*' the server using its private database connection and periodically attempts to reconnect. After a period of time, if Phoenix is unable to connect, it gives up and reveals the failure to the application by passing along the original error raised by the ODBC driver.

If reconnection is successful, Phoenix determines if the database actually crashed or whether there was simply a communication failure or delay, and whether the database session still exists. There is no explicit test for this, so we test a proxy, i.e. whether a special temporary table created for the database session still exists. Temporary tables exist only within a session and are lost when the session terminates for any reason, including a crash.

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

No special treatment is required to handle a failure that occurs during recovery. Since session recovery consists of re-establishing a connection to the database server and re-establishing session state, recovery is idempotent, i.e., Phoenix can safely execute the recovery phase again.

Any active transactions on the database are aborted during the failure and must be restarted by the application. Transaction failure is considered a normal event that most applications already handle. Phoenix/ODBC permits the application to proceed as it would for other forms of transaction aborts.

3 Phoenix/ODBC Performance Evaluation

Our first set of experiments focused on workloads with high query complexity and large results sets, such as OLAP and decision support applications. Specific questions of interest to us were:

- What is the impact of persisting session state, and particularly result sets, on both response time to the application and database server throughput?
- How fast can Phoenix/ODBC recover and re-establish a database session after server failure?
- Exactly where does the overhead lie in persisting volatile database session state?

We conducted experiments using standard benchmarks and our own controlled experiments. Specifically, to evaluate performance on complex queries we selected TPC-H, a benchmark designed to test the performance of decision support queries in business environments. The TPC-H query suite ranges from a simple single-table query to a complex eight-way join query, while update functions concurrently perform `insert` and `delete` operations. TPC-H defines two benchmarking tests – the *power test* and *throughput test* – suitable for measuring application performance and server throughput respectively. Finally, we designed our own experiments to measure the time required to recover a database session, and to characterize overheads.

For these experiments we implemented an interactive application that connects to a named database server, with an option to select either Phoenix/ODBC or native ODBC for data access. The application then either submits TPC-H or TPC-C queries and updates to the server following the benchmark specification, or accepts ad hoc SQL queries as input and forwards the request to the server for processing. The database used was SQL Server 7.0.

We ran all our experiments on two 400Mhz Pentium II processors, each running Windows NT 4.0. Each machine had 256MB RAM and were connected by a 100Mbit/sec local area network. The server had three internal 9GB SCSI disk drives. We measured elapsed time on the client using the Pentium 64-bit cycle counter. This fine-grained counter allowed us to measure elapsed time for individual queries and measure each separate step in Phoenix/ODBC execution: e.g., intercepting and parsing application requests, materializing session state, creating a table on the database, loading the result into a table, and fetching result tuples. Since our purpose is not to report a benchmark result, but rather evaluate overheads, we do not compute metrics defined by the TPC specifications. Rather, we present our experimental results. Official reporting of TPC results requires careful configuration of processors, disk subsystems, and network. We simply used hardware we had access to in our lab to assess the performance impact.

3.2 TPC-H Power Test

TPC-H is designed to measure database systems for decision support [3]. The benchmark database has eight tables: REGION, NATION, SUPPLIER, PART, PARTSUPP, CUSTOMER, ORDERS, and LINEITEM. A *scaling factor* parameter determines the size of the tables. We set the scaling factor to 1.0 so the two largest tables, ORDERS and LINEITEM, had 1.5 million and 6 million tuples respectively. Total database size was 1 GB of raw data. TPC-H defines 22 queries and 2 update functions, which include a variety of operators and selection predicates.

The TPC-H power test executes all queries and update functions one at a time in a fixed order and their running time is measured individually. This measures "raw query execution power". For this experiment we executed the power test fifty times for both native ODBC and Phoenix. Table 1 presents the computed average of TPC-H power test runs. The standard deviation of these runs is generally less than 1% of the mean. The first column identifies the query and update function number and the second column provides the number of tuples either returned in the query result or modified by the update function. The third and fourth columns contain the running times using ODBC and Phoenix/ODBC. 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 (Q1-Q22), the total run time of the queries using Phoenix, which generates a persistent result, is approximately one percent greater than when using native ODBC, which generates a temporary (volatile) result set. As shown in Table 1, for compute-intensive queries that produce a small result, the overhead is small, averaging just over one second for each query. As we describe in the next section, much of this overhead is spent creating a persistent table to hold the result set, writing the result set into this table, then reopening the table once the transaction commits.

Refresh function RF1 `inserts` 1500 tuples into the ORDERS table and roughly 6000 tuples into LINEITEM to emulate the addition of new sales information, while refresh function RF2 `deletes` 1500 tuples from ORDERS and roughly 6000 tuples from LINEITEM to emulate the removal of obsolete information. We decomposed each refresh function into two transactions; each receives one-half of the key range that is to be modified. The tuples corresponding to new orders and new lineitems were loaded into the database, as were 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.

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

Query/ Update	**Result Set/ Updates**	**Native ODBC** in seconds	**Phoenix/ODBC** in seconds	**Difference** in seconds	**Ratio**
Q01	4	106.297	107.706	1.409	1.013
Q02	100	4.775	5.262	.486	1.101
Q03	10	67.580	68.885	1.304	1.019
Q04	5	106.979	108.275	1.296	1.012
Q05	5	109.984	111.145	1.161	1.010
Q06	1	15.932	17.768	1.836	1.115
Q07	4	62.120	66.752	4.632	1.074
Q08	2	114.111	115.489	1.377	1.012
Q09	175	168.701	167.769	-.931	.994
Q10	20	122.112	126.643	4.530	1.037
Q11	1048	16.372	17.967	1.594	1.097
Q12	2	121.391	123.076	1.684	1.013
Q13	42	40.337	41.871	1.533	1.038
Q14	1	3.771	5.057	1.286	1.341
Q15	1	6.464	8.207	1.742	1.269
Q16	18,000	18.868	18.034	-.834	.955
Q17	1	29.347	32.606	3.258	1.111
Q18	100	278.078	279.204	1.125	1.004
Q19	1	331.985	332.164	.178	1.001
Q20	204	81.371	91.896	10.525	1.129
Q21	100	463.432	448.040	-15.392	.966
Q22	8	7.549	9.052	1.502	1.199
RF1	7,500	56.700	57.540	.840	1.015
RF2	7,500	335.600	336.970	1.370	1.004
Total (Query)		2277.556	2302.868	25.312	1.011
Total (Updates)		393.300	394.510	1.210	1.003

Phoenix/ODBC *wraps* each `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 has successfully completed. Thus, the primary overhead for modification statements (`insert`, `delete`, `update`) is the transaction for the request and the single write to the status table to record statement completion. As we can see in Table 1, overhead for data modification statements is negligible.

3.3 TPC-H Throughput Measurements

In our second experiment, we assessed the performance impact on the database using the TPC-H *throughput test*. It is intended to measure the rate a database can process queries. Clients submit multiple concurrent query streams to the server and each stream executes the query suite one query after another, like the power test except that each stream is assigned a unique ordering for query execution. In addition, there is a single update stream that executes the refresh functions RF1 and RF2 to simulate a sequence of batched data updates executing against the database.

Results from the TPC-H throughput test are presented in Table 2, where elapsed time is the duration of the measurement interval measured in seconds. For our evaluation we ran two concurrent query streams, the minimum required for a 1GB database, and ran a separate refresh stream that executes refresh functions RF1 and RF2 twice (equal to the number of query streams). The measurement interval starts when the first query of the first stream is submitted, and ends when the last query of the second stream completes.

Table 2: Results from TPC-H throughput test on two streams.

Elapsed Time for Native ODBC	**5472.00**	**seconds**
Elapsed Time for Phoenix/ODBC	**5492.39**	**seconds**
Difference	**20.39**	**seconds**
Ratio	**1.003**	

Table 2 shows that Phoenix requires approximately 20 additional seconds to complete the throughput test – a .3% overhead over ODBC. If Phoenix were imposing a heavy cost on the server, we would expect to detect a noticeable drop in throughput even for two streams. This did not occur, indicating the cost of providing persistent database sessions is quite modest. We plan further tests using additional query streams to confirm this.

3.4 Recovering a Database Session

Next, we measured the time required to recover a database session. To conduct this experiment we submit TPC-H query Q11 to the server and begin fetching tuples until we near the end of the result set, leaving a few tuples unread. Then we "*crash*" the server by issuing the command "`shutdown with nowait`" from the Query Analyzer, terminating the process running the server. At this point the client application is left waiting for the server to respond to its fetch request. We restart the database server, then measure the time required for Phoenix/ODBC to recover the database session and respond to the outstanding fetch request.

Of interest in this experiment is: **i)** time required to recover the virtual database session, and **ii)** time required to recover SQL state for active application requests. Together, these represent the time required to recover a database session and continue application execution.

Once Phoenix/ODBC is able to contact the database system 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 this 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.

The results presented in Figure 3 are when Phoenix re-positions the result set by sequencing through it from the client. Because we crashed the server near the end of the result, it must advance through the result set to reach the appropriate tuple. Thus, these results represent the upper bound for recovering SQL state for the given result size.

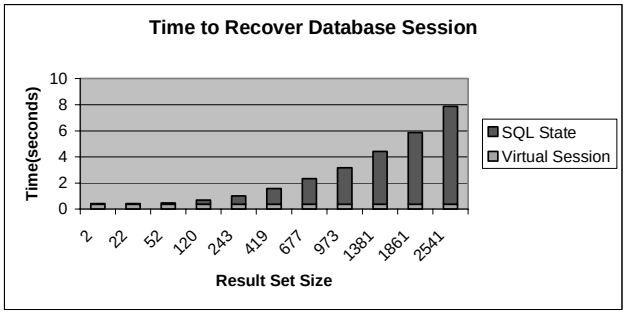

Figure 3: Elapsed time for session recovery, re-positioning at client.

It seemed to us that advancing through a result set during recovery without sending tuples across the client-server connection would speed up recovery by reducing unnecessary communication costs. So we implemented a stored procedure that advances to a specified tuple in a table, hence advancing through the result set on the server without passing tuples to the client. When the procedure completes, the table pointer is left in place and the next fetch request returns the desired tuple. The reduction in communication costs and, consequently, the time required to recover SQL session state, can be seen in Figure 4.

This is a dramatic 10 to one reduction in overhead for larger result sets. Comparing Figure 4 with the total query Q11 response time plus the 10 seconds or so to deliver the 2541 tuple results reveals that the recovery time for a Phoenix/ODBC database session is also a factor of 10 better. That is, Phoenix/ODBC can recover the complete ODBC database session in less than a tenth of the time required to simply recompute query Q11.

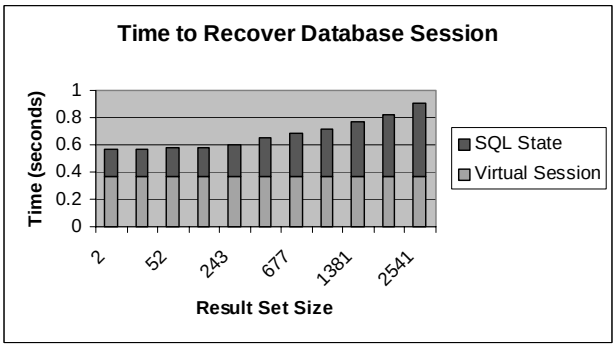

Figure 4: Elapsed time for session recovery, re-positioning at server.

3.5 Overheads for Persisting a Result Set

To gain further understanding of the overheads we performed two additional experiments. First, we selected the *Important Stock Identification Query* (Q11), which allows us to adjust query selectivity and vary result set sizes. Q11 identifies parts that represent a significant percentage of the total value of all available parts, by scanning available stock of suppliers in a given nation. It returns a part number and value of those parts in descending order of value, so each tuple returns approximately 24 bytes. The SQL Select statement for Q11 is shown in Figure 5.

```
SELECT ps_partkey,
   sum(ps_supplycost * ps_availqty) as value
FROM partsupp, supplier, nation
WHERE ps_suppkey = s_suppkey
      AND s_nationkey = n_nationkey
      AND n_name = 'GERMANY'
GROUP BY ps_partkey having
      sum(ps_supplycost * ps_availqty) >
  (SELECT
     SUM(ps_supplycost*ps_availqty)*[Fraction]
     FROM partsupp, supplier, nation
     WHERE ps_suppkey = s_suppkey
       AND s_nationkey = n_nationkey
       AND n_name = 'GERMANY')
     ORDER BY value desc;
```

Figure 5. The Important Stock Identification Query (Q11).

We measured response time and performance overheads for Q11 using both native ODBC and Phoenix for result sets of various sizes. By adjusting the Fraction parameter we varied the result set size from one to several thousand tuples, yielding the somewhat arbitrary looking result sizes reported on the horizontal axis of Figure 6.

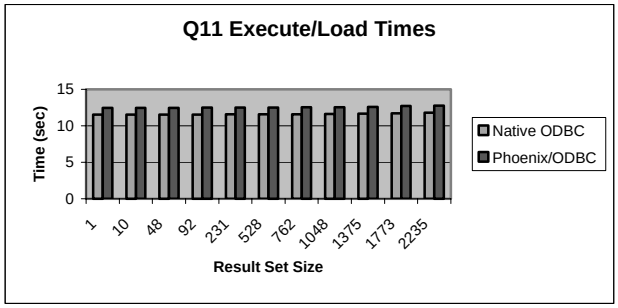

Figure 6: Response time for Q11 using OBDC and Phoenix/ODBC.

To identify where the additional time is spent, we measured elapsed time for each step taken to process a request. Time required to parse the query (.00023 sec), access metadata (.00062 sec), and create the persistent table (.321 sec) were negligible and constant for all executions. Figure 6 displays elapsed time to prepare and execute the stored procedure that evaluates the query and loads the result into the table. For comparison, it displays elapsed time for ODBC, which generates a volatile result.

The response time is dominated by the cost of query execution and writing the result to a persistent table. The primary ongoing overhead is the extra logging to store the result in a table. There is less than a 10% response time hit for producing a persistent result set for Q11.

Another concern is the time required to fetch tuples of the result after the query has executed. While there can be no doubt that reading data from a persistent table is more expensive than reading it from a volatile result set, nonetheless, the impact on the application is very small. The response time for ODBC tuple fetch averaged .00380 seconds, while for Phoenix a tuple fetch averaged .00397 seconds, less than a 5% overhead.

We conducted one final experiment to profile response time differences for an extremely simple query for which the cost to materialize the result dominates. We chose the query "`select TOP N * from LINEITEM`", doubling the tuples in the result from 1 to 256K by varying N. The LINEITEM table holds six million rows of about 150 bytes each. Table 3 presents query response time over a range of results sizes, along with the ratio of response times.

The ratios in query response time are quite large for small result sets. The differences are due not only to the run-time overheads of Phoenix/ODBC (request interception, scanning and parsing, transaction initiation), which are quite small, but also to table creation and logging. While the total of these costs contributes roughly nine-tenths of a second to response time, they have a large impact on the response time ratio between native ODBC and Phoenix/ODBC. As the result increases, however, the response time ratio decreases, and for result sets ranging from 256 to 4K tuples, the response time of Phoenix/ODBC is faster than native ODBC. We have no satisfactory explanation for this.

Table 3: Response time in seconds for ODBC and Phoenix/ODBC, w/ computed ratio between ODBC and Phoenix

Result Set Size	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	*continued…*
Native ODBC	0.001	0.003	0.008	0.008	0.009	0.214	0.432	0.871	1.516	
Phoenix/ODBC	0.93	1.044	1.050	1.060	1.103	1.072	1.121	1.147	1.156	
Ratio	930	348	131.25	132.50	122.5	5.009	2.594	1.316	0.762	

Result Set Size	**512**	**1 K**	**2 K**	**4 K**	**8 K**	**16 K**	**32 K**	**64 K**	**128 K**	**256 K**
Native ODBC	1.518	1.518	1.518	1.518	1.518	1.520	1.520	1.521	1.522	1.525
Phoenix/ODBC	1.160	1.192	1.279	1.467	1.715	2.290	3.458	5.876	11.139	18.739
Ratio	0.764	0.785	0.842	0.966	1.129	1.506	2.275	3.863	7.318	12.287

The response time for native ODBC did not increase after 512 tuples (75KB of data), even as the result grew to over 256K tuples. Using the SQL Server Profiler revealed there were no writes to the output buffer or reads from the database once the result reached 512 tuples. The reason for this is that our application does not consume results from the query. We are measuring query response time, not client transfer rate. Once the network buffer reaches capacity, the scan for data is suspended because no space is available to add rows. In contrast, when using Phoenix the scan for qualifying rows runs to completion in order to store the result in the persistent table. Once the result is materialized, the transfer of tuples also blocks when the buffer reaches capacity. In short, while the experiment gives an accurate measure of response time, it is not the whole story – once the application reads data to free up room in the output buffer, native ODBC must continue querying database tables for qualifying rows, while Phoenix/ODBC simply streams tuples from the table.

3.6 Discussion of Results

Our primary goal was to understand the performance of Phoenix/ODBC over a range of database applications. First, we sought to measure the overhead of persisting session state on application response time and database server throughput. When query complexity was high and results large, such as in TPC-H benchmark, we found on average the overhead is approximately 1%, while for update functions the overhead is less than 0.5%. Results from TPC-H throughput tests, measuring overhead on the database, were encouraging; Phoenix imposed less than a 0.3% overhead when running concurrent query streams, indicating impact on throughput is minor. Next, we asked how fast Phoenix/ODBC could recover and re-establish a session after a failure. We demonstrated that a database session could be recovered and reinstalled in a fraction of the time required to recompute a single query. Finally, we detailed the costs in materializing and persisting volatile results from SQL statements. Our tests revealed that for simple queries with small results, the overhead is dominated by table creation costs. While the time to create a persistent table on the server is small, the cost relative to query execution is significant, motivating the performance optimization we introduce next in Section 4.

4 Optimizing Phoenix/ODBC for OLTP

Early in the design of Phoenix/ODBC we realized the greatest overhead was the cost to create a table on the database to make result sets persistent. Performance measurements presented in Section 3.5 confirmed our intuition. While the cost to create a database table was small, its *relative overhead* for persisting small result sets was profound. This led us to an optimization specifically suited for simple queries with small result sets. The insight behind the optimization is that *caching results* on the client can mask server failures without materializing result sets on the server, hence improving both application *response time* and server *performance*.

4.1 Client Side Caching of Result Sets

We implemented a client side cache in the Phoenix-enabled driver manager, large enough to hold small result sets. The size of this client cache is a runtime parameter, set when a database connection is first created. When an application submits a result-generating SQL statement, Phoenix submits the original statement to the database. Once a response is returned from the server indicating the statement successfully executed, Phoenix reads the entire result into the cache. At this point Phoenix can guarantee the result will persist across failures. In fact, the client is isolated from the server until it services the next request. Phoenix returns control to the application, which can then `fetch` tuples until the end of result or the application issues a `close` for the set. Phoenix retrieves tuples by reading the cache, and when it is empty the application is notified that the result set has been consumed.

Thus, a client side cache is resistant to server failures. If the full result set does not arrive at the client, Phoenix performs its usual recovery procedure to reconnect to the database. If necessary, the query is re-executed, and the results again delivered to the client. Only when the entire result set is present at the client (and cached there) does Phoenix begin to deliver results to the application. At this point it can guarantee that a server crash will not affect its ability to deliver the result to the application.

In an implementation sense, the client cache provided by Phoenix differs little from a volatile result set managed by the server. There are, however, performance and

availability advantages for the client application and Phoenix. Many applications spend a significant amount of time bringing data across the network. Part of this time is spent actually transmitting the data and part of it is spent on network overhead, such as the call made by the driver to request a row of data. Phoenix reduces the latter cost by bringing the result set across the network into the client side cache using a single ODBC block cursor read. Since Phoenix does not need to create a persistent table on the server, the cost for providing a result set that will persist across database server failures is eliminated.

4.2 TPC-C Benchmark

We selected TPC-C [2], a standard benchmark for measuring performance for workloads representative of OLTP, to evaluate Phoenix/ODBC performance on simpler queries and assess the effectiveness of client side caching. The TPC-C workload is patterned after an order-entry system and tests a broad cross-section of database functionality using transaction types ranging from simple transactions, comparable to the credit-debit workload in TPC-A/B, to medium complexity transactions that perform several SQL calls to the server.

The database is composed of nine tables: warehouse, district, customer, stock, item, order, new-order, order-line, and history. The cardinality of the tables, except for the item table, grows with the number of warehouses specified. The order, order-line, and history tables grow indefinitely as orders are processed. TPC-C defines five transactions: 1) The new-order transaction places an order from a warehouse; it inserts the order and updates the corresponding stock level for each item; 2) The payment transaction processes a payment for a customer, updates the customer's balance, and reflects the payment in the district and warehouse sales statistics; 3) The order-status transaction returns the status of a customer order; 4) The delivery transaction processes orders corresponding to 10 pending orders, 1 for each district with 10 items per order. The corresponding entry in the new-order table is also deleted; 5) The stock-level transaction examines the quantity of stock for the items ordered by each of the last 20 orders in a district and determines the items that have a stock level below a specified threshold.

The TPC-C benchmark metric is the total number of new order transactions completed per minute. This metric measures "business throughput" rather than raw transaction execution and is expressed in transactions-per-minute C (TPM-C). The other four transactions are used as background load to provide a context for the new order transactions. The background transactions are defined to be at least 57 percent of the mix, so new orders are at most 43 percent of the work; for a complete description of the workload see [2]. This means a score of 100 (new order) transactions per minute actually represents over 230 transactions (of all types) per minute.

Again, we point out that our purpose is not to report a TPC-C benchmark number, but rather evaluate Phoenix/ODBC overheads. Thus, we simply ran the benchmark on hardware available in our lab. This was suitable for the purposes of our experiment, but does not constitute a true TPC-C benchmarking effort.

4.3 TPC-C Results

To carry out the benchmark we constructed a five-warehouse database (about 500 MB) on the server and created a backup of the database. In each experiment there were 32 emulated "users" on the client system submitting transactions to the server; this was the optimal number of users the client machine could support. Each of the simulated users operated with zero think time and submitted transactions to the server at random based on a predefined transaction mix, as defined by the TPC-C specification [2]. This ensured the transaction mix was well defined and a variety of transaction types were running concurrently. The database server checkpoint interval was set very large so there were no checkpoints taken during the measurements.

We conducted each experiment run for a minimum of 160 minutes. The measurement began approximately 30 minutes after the simulated users had begun executing transactions, to ensure the server attained a steady state, and lasted for 120 minutes. As each run progressed, new orders were generated and the cardinality of the history, orders, new-order, and order-line tables increased. After an experiment was finished we restored the TPC-C database from the backup and repeated the experiment to ensure the accuracy of our results.

Table 4 presents our TPC-C results, in which each row represents an experiment. The TPM-C column is the measured work rate in transactions-per-minute C, CPU UTIL is CPU utilization on the server, DISK UTIL is disk utilization on the server, and CPU RATIO represents the ratio of CPU cost per transaction required to support Phoenix/ODBC versus the cost using native ODBC.

Table 4: TPC-C using 1) ODBC, 2) Phoenix, 3) Phoenix w/ caching.

EXPERIMENT	TPM-C	CPU UTIL	DISK UTIL	CPU RATIO
1 **Native ODBC**	391	32%	100%	1
2 **Phoenix/ODBC**	327	34%	100%	1.27
3 **Phoenix/ODBC w/ client caching**	391	32%	100%	1

In experiment 1 we ran the benchmark using native ODBC, which generates temporary (volatile) result sets, to determine its TPM-C. Table 4 shows a rate of 391 TPM-C and disk utilization of 100% is evidence that our server is disk limited. Given that CPU utilization is 32%, we would expect a higher TPM-C rate if sufficient disks were available on the server. In experiment 2 we ran the

benchmark using Phoenix and the rate was reduced to 327 TPM-C, while CPU utilization increased to 34%. This characterizes the Phoenix overhead to create persistent result sets for transactions in the benchmark. The CPU RATIO indicates that it requires 27% more CPU per transaction, which is substantial.

Since the result sets of TPC-C transactions are small, typically less than 20 tuples, most of the overhead is the result of creating a persistent table on the server to store the result. This is exactly the scenario for which we introduced the client-side caching optimization. In experiment 3 we ran TPC-C using Phoenix with client caching enabled. Since results from select statements in the benchmark were small, no persistent tables were created on the server – entire result sets were cached in the client buffer. As a result of the reduced overhead, we achieved 391 TPMC, the same rate as native ODBC, and no additional CPU or disk resources were required. This should come as no surprise, as the work assigned to the server was identical in both cases.

To sum up, for the simple TPC-C transactions that are characteristic of an OLTP workload, e.g., that generate small result sets, the percentage overheads for persisting result sets on the server are significant. But our TPC-C experiments demonstrated the effectiveness of the client caching optimization for improving response time for small result sets, entirely eliminating this overhead.

5 Concluding Remarks

Phoenix/ODBC provides persistent database sessions that can survive database server failures, without the application itself having to take measures for its recoverability. Any application can use Phoenix/ODBC to enhance database session availability without having to modify the application program, ODBC driver, or database server. Methods to persist session state and to recover from failures are wired in, and do not require administrators or application programmers to remember and follow a procedure correctly. Indeed, the application need not be aware of the server crash. Masking server failures both improves application availability and simplifies the task of the application programmer.

Our performance evaluation, using industry-standard benchmarks, demonstrates that it is possible to provide persistent database sessions without compromising application performance or imposing undue load on the database server. Thus, while there is a modest system cost for database session persistence, Phoenix continues the trend of carefully expending system resources to conserve more expensive and error-prone human resources.

One way to view our work is to regard it as an extension of database recovery technology to the data access portion of the system. We leverage existing database recovery mechanisms by wrapping ODBC to capture application interactions with the database and log session changes as tables on the database. We exploit the notion of a virtual database session in which an application interacts with a Phoenix session, which in-turn is mapped to an actual ODBC session. Our procedures detect server failures and re-map the virtual session to a new session into which we install the pre-crash session state once the database has recovered. This integrates database recovery *and* transparent session recovery.

Bibliography

[1] Phoenix Project, Database Group, Microsoft Research, http://www.research.microsoft.com/db

[2] Transaction Processing Council (TPC). TPC Benchmark C. http://www.tpc.org, 1992.

[3] Transaction Processing Council (TPC). TPC Benchmark H. http://www.tpc.org, 1999.

[4] Barga, R. and Lomet, D. Phoenix: Making Applications Robust. (demonstration paper) Proceedings of ACM SIGMOD Conference, Philadelphia, PA (June, 1999).

[5] Barga, R., Lomet, D., Baby, T., and Agrawal, S. Persistent Client-Server Database Sessions. Proceedings of EDBT Conference, Lake Constance, Germany.

[6] Bernstein, P. Goodman, N. and Hadzilacos, V. Recovery Algorithms for Database Systems. IFIP World Computer Congress, (Sept. 1983).

[7] Gray, J. and Reuter, A. Transaction Processing: Concepts and Techniques. Morgan Kaufmann (1993) San Mateo, CA

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

[9] Lomet, D.B. and Weikum, G. Efficient Transparent Application Recovery in Client-Server Information Systems. ACM SIGMOD'98 Conference, Seattle, WA (June 1998).

[10] Lomet, D. Application recovery using generalized redo recovery. Int'l. Conference on Data Engineering , Orlando, FL (February, 1998).

[11] Lomet, D. and Tuttle, M. Redo recovery from system crashes. VLDB Conference, Zurich, Switzerland (Sept. 1995).

[12] Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference, MS Press, 1997.

[13] Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK, MS Press, 1998.

[14] White, S., Fisher, M., Cattell, R., Hamilton, G. and Hapner, M., JDBC(TM) API Tutorial and Reference, Second Edition, Addison-Wesley Publishing Co., 1999.

[15] Sceppa, D. Programming ADO, MS Press, 2000.