Phoenix: Making Applications Robust

Roger Barga
David B. Lomet
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
{barga,lomet}@microsoft.com

1. Introduction

Dealing with errors or exceptions is a very large part of getting applications right. Failures are not only an application programming problem but an operational and an availability problem as well. The Phoenix goal is to increase the availability of an application and in many cases avoid the operational task of coping with an error. There are two aspects of this:

System Crashes: While database systems recover database state, the state of applications using the database, and their sessions, are "blown away" (erased). Our intent is to extend database recovery to session and application state. This will enable stateful applications to survive failures and continue execution.

Logical Errors: Transactions abort for logical errors as well as crashes. Aborted transactions roll back to transaction start. Our intent is to extend database recovery to support partial rollback for application errors, where the rollback resets not only database state (supported by savepoints) but also application state.

We are currently exploring technology that exploits database redo recovery to enable applications to persist across system failures. While forms of program persistence have been proposed, the costs have been high in logging and checkpointing. Our techniques [1,2,3] substantially reduce these execution costs and leverage the database's recovery mechanisms to accomplish this. Phoenix continues the trend of expending system resources to conserve more expensive and error-prone human resources.

1.1 Phoenix/ODBC – Persistent Sessions

In the Phoenix project, we have focused first on client application availability and persistence in the presence of database server failures. Our initial systems effort avoids the difficulty involved with making substantial changes to the internals of the database system by focusing on session availability. We have built a prototype Phoenix ODBC driver which provides persistent server sessions to ODBC-enabled clients, sessions that can survive a server crash without the client application being aware of the outage, except for possible timing considerations.

2. ODBC Background

ODBC (Open Database Connectivity) is a client application API to SQL database servers based on the X/Open and ISO/CAE SQL Call Level Interface standard. Applications can access data in any DBMS supporting ODBC for access to their databases. An application makes calls to ODBC using SQL statements written in either ODBC SQL or DBMS-specific SQL syntax. The usual components involved in going from an ODBC client application to a DBMS Server are outlined below and illustrated in Figure 1.

ODBC Driver: a DBMS vendor provided dynamic link library that responds to all client application calls to the ODBC API. The driver translates SQL statements into DBMS-specific SQL syntax and passes the statement to the server and reformats results returned from the DBMS into ODBC format.

ODBC Driver Manager: a platform component that manages communications between the application and vendor provided ODBC drivers for any database with which the application works. The driver manager loads the ODBC driver and passes all application requests to it.

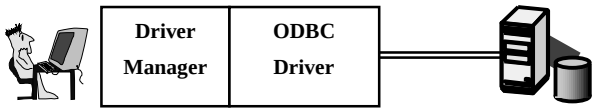

Figure 1: Illustration of the main components in the ODBC architecture.

2.1 An Example ODBC Database Session

To illustrate the use of ODBC and the likely result were a database failure to occur, we consider the following example. Our database session is centered on a data analysis query, similar to those in the TPC-D benchmark, and involves three tables: a master customer table, a detail orders table, and 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 aggregate the order totals into the invoice summary table. This client application might be coded as follows:

1. Create an ODBC session by opening a connection to the server, log on the database and set session specific attributes
2. Request the server create a result set from the customer table (**A**) consisting of records with a last name of 'Smith'
3. Fetch customer records from the result set, until the appropriate customer record is found.
4. Ask the server to open a cursor on the orders table (**B**) for orders matching this customer's ID
5. Fetch all matching order records for this customer ID.
6. Calculate the aggregate of those order detail records.
7. Send a command to update the invoices table (**C**) with the calculated aggregate.
8. Close connection to database, terminating the session.

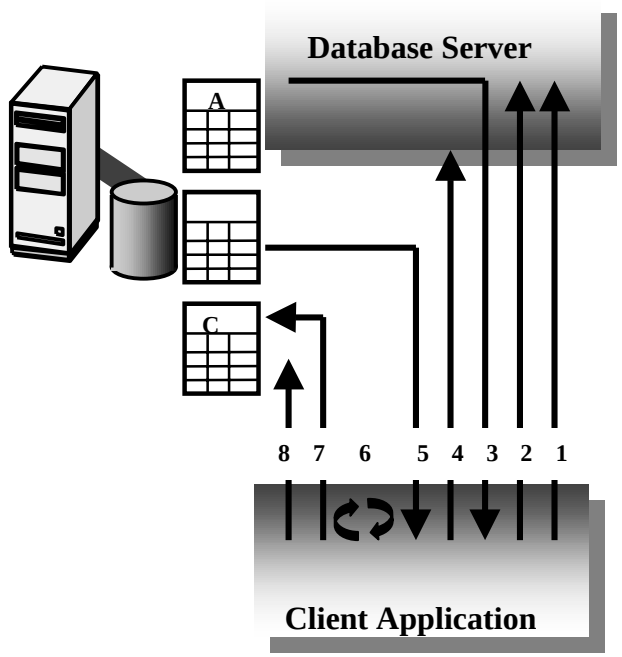

Figure 2: Client application using ODBC to access client database.

Consider what would happen were the database server to fail during this ODBC database session. The main problems are:
Server Failure: ODBC functions can have undefined behavior when the server is down and frequently require that the application be terminated.
Application Availability: After application termination, the application must be restarted, a new connection to the SQL Server must be created, and a new ODBC session must be established. Partial execution can leave the application state confused, requiring long outages in order to reconstruct it manually.
Loss of Transient State. If a failure occurs after the application has created volatile state, e.g. result sets from SQL statements, this state is lost.

For an application to cope with database server failure requires additional code to deal with these problems. This increases application complexity, delays deployment, contributes to bugs, and can further reduce overall application availability.

3. Overview of Phoenix/ODBC

3.1 The Phoenix/ODBC Architecture

To provide session availability we introduce Phoenix/ODBC, a Phoenix-enabled ODBC Driver Manager. Phoenix/ODBC wraps any native ODBC driver, intercepting application requests going to the database as well as responses returned from the server.

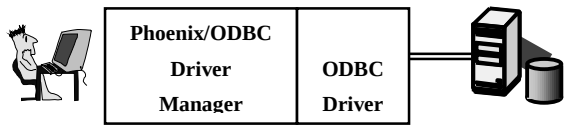

Figure 3: ODBC architecture with a Phoenix-enhanced ODBC Driver Manager.

3.2 Phoenix/ODBC Actions

Phoenix/ODBC creates a virtual ODBC database session for an application (statement 1 of our example) and maps it to one or more real ODBC database sessions. It detects server failures and recovery by timing out requests and pinging the server until it recovers. It then re-associates the virtual session to a new ODBC session by reconnecting to the server and re-associating saved information with this new session. Finally, when the session terminates it cleans up any persistent session state that was created (statement 8).

Phoenix/ODBC captures transient session state and makes it persistent. It logs statements that alter session context (statement 1). It rewrites SQL statements to force the creation of persistent database tables that capture application 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 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).

3.3 Decomposing Application State

Our first step towards providing ODBC persistent sessions was to decompose server session state into elements, each of which could be materialized as a distinct data object. These elements of session state have different lifetimes and recovery requirements that we exploit. These include:

ODBC Session Context – All client settable attributes of a session, including Database Connection – refers to connection request and user login information.

Environment, connection, and statement attributes – Context, not associated with attributes, includes user identification, current database, user temporary objects, and unacknowledged messages sent by the server to the client.

Result Generating SQL Statement – SQL statement that will return one of following:

- A result set returned from a SELECT statement.
- A global cursor that can be referenced outside the SQL statement.
- Return codes, which are always an integer value.
- Output parameters, which can return either data or a cursor variable.

Database Procedures – A procedure stored at the server, usually one or more SQL statements that have been precompiled.

SQL Command Batch – 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.

Transactional SQL Statements – State includes transaction manager state, uncommitted database changes, locks held, memory images of execution plans, buffers, and intermediate results such as sorts.

3.4 Delivery of SQL Statement Results

A challenging aspect of masking server failures is ensuring seamless delivery of results to the client. Phoenix/ODBC enables this by making the result of a SQL statement a persistent table and re-accessing the result set table after a failure.

Phoenix/ODBC intercepts each application SQL request and performs a one-pass parse to determine statement type. If the statement generates a result, Phoenix/ODBC takes the following steps to ensure it will be recoverable in case of server failure.

Step 1. Phoenix/OBDC accesses the metadata describing the columns in the result set. ODBC results include this metadata with the result data. Phoenix/ODBC acquires only the metadata by appending the clause "WHERE 0=1" to the original SQL statement, guaranteeing both that no result data will be returned and that the query will not actually access database data in generating the result. It then sends this request to the database server.

Step 2: Phoenix/ODBC reads the metadata and uses it to generate a CREATE TABLE statement specifying a persistent table to hold the result. It then sends the CREATE statement to the database server to create an empty table at the server to hold the result set.

Step 3: The SQL statement is executed so as to insert its result set into the just created persistent table at the server. What gets materialized depends on both the original SQL statement and on how the application requests the result set from the server. With ODBC, the "how" is determined by the statement options set prior to executing a SELECT.

Step 4: To ensure seamless result set delivery, Phoenix/ODBC keeps track of the current location in the now persistent result set as data is fetched. 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 fetch before the failure.

To make a default result set persistent, Phoenix/ODBC stores it in its entirety into the persistent table created at step 2 prior to returning rows to the client application. Phoenix/ODBC creates a stored procedures that encapsulates the original application SQL statement and takes only one parameter, the name of the empty database table. The advantage of using a stored procedure to execute the original SQL statement and load the persistent table is that all the data is moved locally at the server, not sent first to the client. This procedure, using only ANSI-standard SQL, is created as follows:

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

The procedure execution is itself an atomic SQL statement. Once the server has returned a response indicating the procedure was successfully executed, the result set is stable and will persist across server failures. Phoenix/ODBC then issues the SQL statement SELECT * FROM **T** to open the table and returns control to the application program for normal processing.

When Phoenix/ODBC detects that the database server has failed, it "pings" the server until it detects that the server has recovered. It then reconnects and reestablishes an ODBC session. Phoenix/ODBC then identifies the application's last completed request and asks the server to re-send the result set if necessary

4. Phoenix/ODBC Performance

Using queries from the TPC-D benchmark, we conducted an evaluation of Phoenix/ODBC to measure the costs of persisting and recovering ODBC database sessions[4]. We found the following:

Phoenix/ODBC overhead to persist result sets for queries with a high degree of complexity, such as those found in the TPC-D benchmark, is modest. For benchmark query Q5 the response time difference between Phoenix/ODBC and 'vanilla' ODBC is less than 4%, while for query Q11 there was less than a 30% difference in response time for result sets up to 100 tuples.

For computationally simple queries that generate small result sets the response time difference is small, but the response time ratio between Phoenix/ODBC and vanilla ODBC is more pronounced. The difference is due not only to run-time overheads of Phoenix/ODBC (request interception, scanning and parsing), which are quite small, but also due to table creation costs. Total Phoenix/ODBC costs contribute only about one third of a second to response time, but for simple queries this has a large impact on the response time ratio.

The time to fetch tuples from a Phoenix/ODBC persistent result set is within 5% of the fetch cost for an ODBC volatile result set. Response time measurements for ODBC fetch averaged .00380 seconds, while response time for Phoenix/ODBC fetch averaged .00397 seconds.

Once the database system recovers from server failure, Phoenix/ODBC can recover the entire database session and continue application execution in less than one second. Comparing this to the execution time for TPC-D query Q11, we note this is less than a tenth of the time required simply to recompute the query and send its results to the client.

5. Summary

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. 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 cost for application persistence, Phoenix continues the trend of expending system resources to conserve more expensive and error-prone human resources.

6. References

1. Lomet, D.B. and Weikum, G. Efficient Transparent Application Recovery in Client-Server Information Systems. ACM SIGMOD'98 Conference, Seattle, WA (June 1998) 460-471.

2. Lomet, D.B. Persistent Applications Using Generalized Redo Recovery. ICDE'98 Conference, Orlando, FL (Feb. 1998) 154-163.

3. Lomet, D.B. and Tuttle, M. Redo Recovery after System Crashes. VLDB'95 Conference, Zurich, Switzerland (Sept. 1995) 457-468.

4. Barga, R.S., and Lomet, D.B. Persistent Client-Server Database Sessions, Feb. 1999, Microsoft Technical Report (submitted).