PERVASIVE®

DATABASE

Leveraging Pervasive PSQL Relational Capabilities in COBOL Applications

A Pervasive Software White Paper

November 2006

Table of Contents

Introduction 3

Why Pervasive PSQL 4

SQL Access 4

Using Multiple Development Environments 4

Enabling SQL Access 5

Using the COBOL Schema Executor in Pervasive PSQL v9.5 5

Step 1: Evaluating Your Data 5

Step 2: Resolving Incompatibilities 7

Step 3: Describing the Data 9

Step 4: Deployment 9

Conclusions 10

Contact/Trademark Information 11

Introduction

This white paper is designed to help COBOL application developers who are using the ISAM side of Pervasive PSQL (the transactional interface, Btrieve) extend their applications to fully leverage the relational side of Pervasive PSQL.

Whether they have heard about the benefits of fully enabling Pervasive PSQL access in an application, or customers have asked for it, COBOL developers often have an existing body of code that represents years of work and customer feedback, and don't want to rewrite that code in order to take advantage of additional features. This paper explains how to take advantage of the full feature set of Pervasive PSQL while protecting the investment in your application and avoiding the expense and risk of rewriting the code.

It is assumed that the application's COBOL runtime uses Btrieve as the underlying data API. If the runtime is not currently using the Btrieve API (or Pervasive PSQL), and you want to take advantage of the performance and low cost of ownership of the Btrieve API, you should contact either your COBOL runtime developer or Pervasive Software. For an overview of moving your COBOL application to Pervasive PSQL, please read the whitepaper: *Database Migration: COBOL to Pervasive PSQL*.

ISAM and Btrieve

ISAM readily lends itself to the Btrieve API because of the similarities in their record storage and retrieval, which is why COBOL compilers often use Btrieve as the data access mechanism. Like ISAM, the Btrieve API has a one-to-one relationship between files and tables and Btrieve allows the application to define (and redefine) record layout, only needing to be made aware of key information.

Applications built on COBOL runtimes that use Btrieve are already Pervasive PSQL applications. They take advantage of the unparalleled performance, broad platform support, and low total cost of ownership for which Pervasive PSQL and Btrieve are recognized. However, there are some compelling features offered by Pervasive PSQL that are not immediately available to the Btrieve-only application. Pervasive PSQL also provides SQL query capability and access to the data from other languages, tools, and application servers using standard interfaces like ODBC, OLE DB, .NET and JDBC.

ISAM and SQL

SQL and ISAM have different data access requirements. The SQL Database Management System (DBMS) requires a clear definition for every field, and every piece of data must belong to exactly one field. Enabling an ISAM application for SQL access involves resolving the discrepancies between the requirements for ISAM access (which are currently being satisfied by the data) and those for SQL access. Depending on the data, such a migration may require nothing more than the creation of a database schema in Pervasive PSQL catalog tables, known as "DDF" files. In other cases, significant application changes may be required before the data is fully accessible by SQL; in these cases, it may still be useful to enable partial access and migrate to full access incrementally over time.

This paper provides both general and specific guidance for enabling your COBOL application with SQL access. Specifically, Pervasive PSQL v9.5 (Service Pack 2) includes some very useful utilities and examples to provide support for relational access to data using COBOL OCCURS constructs, partial REDEFINES, and variable record layouts. We have found that this will cover a significant percentage of SQL access needs for COBOL developers. For the remainder, this paper discusses general workarounds and examples for enabling more complete SQL access.

Why Pervasive PSQL

There are two major reasons why a COBOL developer would want to take advantage of Pervasive PSQL capabilities: SQL access and integration with a wider range of developer environments.

SQL Access

The feature that COBOL developers most often want to add to their application is SQL access. SQL's flexibility and standardization makes it ideally suited to meeting new and changing requirements in reporting. It's much easier to use SQL to query the database as a solution to ad hoc reporting requirements than it is to design, code, and test each new reporting feature needed by users.

In the traditional COBOL application support cycle, one of two things happens when a user requests a new report: either the IT staff adds the request to the queue and eventually creates a new report after several feedback cycles with the user, or the user goes without. With SQL, users can create ad-hoc reports either through direct or massaged SQL queries. This flexibility offers a competitive advantage over applications that don't have SQL access and cannot enable the user to create their own reports.

However, because of the difficulty in delivering comparable levels of application performance (relative to transactional access methods), most developers are reluctant to convert the entire application to SQL, even if it is supported by the compiler/runtime combination.

In some cases, it may make sense to add SQL support because of the increase of the application's value from extending the feature set. In other situations, the primary concern is meeting a user's immediate need, rather than with increasing the feature set of the application. Third-party tools that use SQL can be an invaluable resource in this case. There are a number of reporting tools available that can consume Pervasive PSQL through ODBC (a standard SQL access method). Once the database allows SQL access, these tools can be used independently of the application. ODBC and SQL can often provide a quick, reliable way to meet a user's requirements without forcing a major re-engineering of the application. Also, since some these tools are aimed at the end user, customers can use the tools to on their own to add the capability they need.

Using Multiple Development Environments

It is becoming more common to see application suites built from multiple development environments. For example, the core of an application may have been written in COBOL, while, the latest features have been added using Delphi or Visual Basic environments. The Btrieve API can be used within most common development environments. And for visual environments such as Delphi, C++ Builder, Visual Basic, and DevStudio (for ASP development), Pervasive PSQL provides friendlier alternatives.

In particular, the Pervasive OLE DB Provider offers navigational and SQL interfaces through ADO ideally suited for developing applications with Visual Basic or ASP-based web applications. Pervasive Delphi Access Components (PDAC) allow navigational and SQL access that is intuitive to the Delphi or C++ Builder developer.

For the Btrieve developer who is moving to one of these development environments, the Pervasive ActiveX Controls provide the ease of use of visual development and performance comparable to the Btrieve API. The Pervasive ActiveX Controls are drag-and drop components that can be used within any of the major visual development environments and provide access methods that will be familiar to any ISAM developer (Open, GetFirst, GetNext, etc.).

Each of these components is part of the Pervasive PSQL Software Developer Kit (SDK), which has excellent documentation and can be downloaded free from the Pervasive website. Using these components in an application requires data schema definitions similar to the information needed for SQL access so enabling a database for SQL access also enables it for use by these additional navigational access methods.

Enabling SQL Access

Fully enabling SQL access to the database may require no application changes or significant changes depending on the organization of the data. There are often cases where substantial benefits are derived from partial SQL access, so it may make sense to incrementally move the application toward full SQL access. Whether one chooses a fully enabled or incremental approach, the data structure must be clearly defined before the SQL engine can access the data.

The first step is to evaluate the database (the files that you use in your application) and determine which parts are compatible with SQL and which parts are going to have to be altered for SQL to access them. Once the data of interest is SQL-compatible, that data must be described to the SQL engine and persisted as metadata (information about the data, e.g., column names, tables names, etc). When these tasks are complete, SQL access is enabled and users may start enjoying the power and flexibility of run-time queries.

PSQL v9.5 provides tools for enabling SQL access for some specific types of data: using COBOL OCCURS constructs, partial REDEFINES, and variable record layouts. We'll first discuss how to use these tools, and then go into a more general discussion of enabling SQL access to other types of COBOL data.

Using the COBOL Schema Executor in Pervasive PSQL v9.5

PSQL v9.5 (Service Pack 2) includes a command line utility, called the COBOL Schema Executor, that populates the system tables used by the PSQL relational interface to interpret the ISAM data as normalized SQL tables. To use the COBOL Schema Executor, you must first describe your data to the SQL engine using XML files. PSQL v9.5 includes sample XML files for defining data in a simple table, data that contains OCCURS constructs, data that includes partial REDEFINES, and data in variable record layouts. The steps involved in using the Schema Executor are:

Step 1: Evaluating Your Data

There are two primary obstacles to full SQL access typically encountered in ISAM implementations: COBOL compiler data types that do not map to database types, and records with multiple definitions. Since a SQL DBMS needs to fully understand the data in order to evaluate queries, the data must be described to it in terms of types it can understand. In the ISAM model, data that is not part of a key is understood only by the application, not by the database itself. This means that non-key data is not necessarily of a type supported by the database. The first task is to determine the equivalent database type for each compiler data type used by your application.

Type Mapping

When enabling SQL access for a COBOL application, there are likely to be three type sets of interest:

1) COBOL data types, such as “9999 COMP”,

2) Btrieve data types, which can automatically map to SQL types, and

3) SQL data types

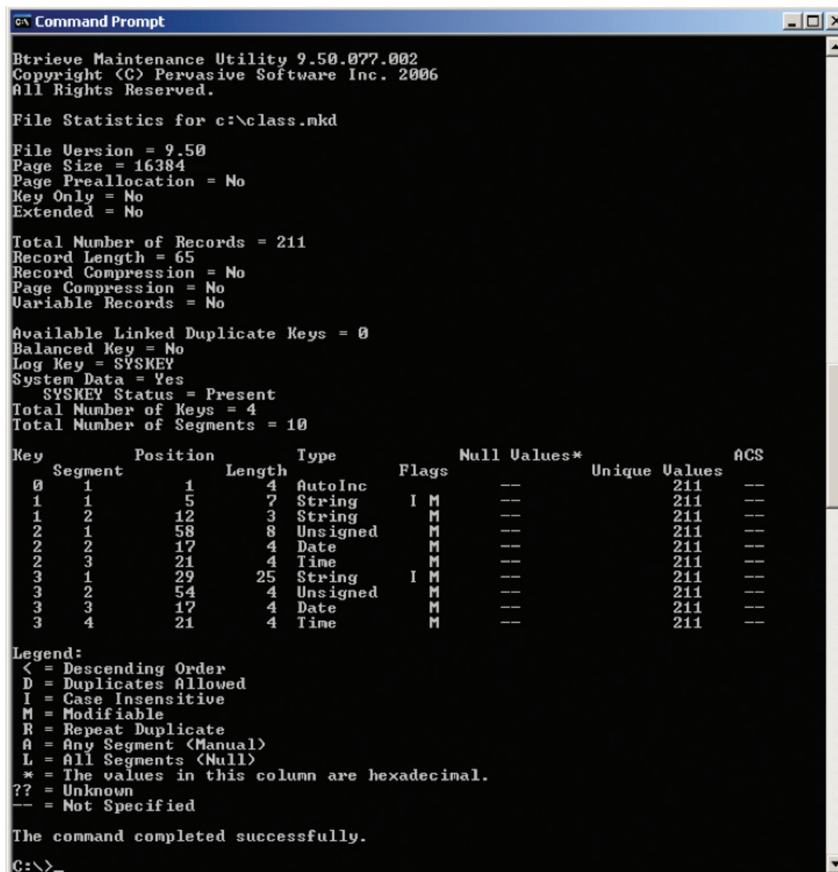
For a review of Btrieve to SQL data type mappings, please review "Pervasive PSQL Supported Data Types" in the SQL Engine Reference.

The best way to determine type mappings is to ask the COBOL compiler vendor. Since the compiler is responsible for persisting COBOL-typed data, it necessarily must be aware of the native data type it uses to represent this information. Although non-key data is application specific, key data must be understood by the DBMS in order to sort correctly. While it is possible for a COBOL vendor to use non-native types even for keys, if the data sorts correctly when using the Btrieve API, it's a good bet that the SQL API understands the data type. It is possible that the compiler vendor may support some types and not others.

If the compiler vendor is unable to supply the necessary type mappings, it is still possible to determine them using the key type information stored in the Pervasive PSQL file header. Since this information can be retrieved by PSQL utilities, type mappings can be determined by creating a key on each type used by the application and looking at the key information in the resulting file.

First, create a new file that includes all of the types used by the application (data length will usually not make a difference). Declare each of these fields as a key in the FD (File Description) and open the file for writing (which should create the file). After this application is run, the file will be in the file system. Then run the Pervasive PSQL utility "BUtil" on this file to determine the native type for each of the COBOL types. Simply go to the Pervasive PSQL bin directory (usually c:\pvsw\bin) and type "butil –stat <file name>", where <file name> is set to the full path of the newly created file. There will be one key entry for each key in the data file, showing the native data type declaration next to it. If a compiler does not map directly to a particular data type, there will be either the compiler's best approximation or the compiler's default type (which will likely be string or binary – a catch-all non-string type that will sort in binary order).

Figure 1: Btrieve Maintenance Utility



```
Command Prompt

Btrieve Maintenance Utility 9.50.077.002
Copyright (C) Pervasive Software Inc. 2006
All Rights Reserved.

File Statistics for c:\class.mkd

File Version = 9.50
Page Size = 16384
Page Preallocation = No
Key Only = No
Extended = No

Total Number of Records = 211
Record Length = 65
Record Compression = No
Page Compression = No
Variable Records = No

Available Linked Duplicate Keys = 0
Balanced Key = No
Log Key = SYSKEY
System Data = Yes
   SYSKEY Status = Present
Total Number of Keys = 4
Total Number of Segments = 10

Key         Position        Type            Null Values*              ACS
    Segment         Length          Flags               Unique Values
  0    1         1       4  AutoInc          --                 211    --
  1    1         5       7  String     I M   --                 211    --
  1    2        12       3  String       M   --                 211    --
  2    1        58       8  Unsigned     M   --                 211    --
  2    2        17       4  Date         M   --                 211    --
  2    3        21       4  Time         M   --                 211    --
  3    1        29      25  String     I M   --                 211    --
  3    2        54       4  Unsigned     M   --                 211    --
  3    3        17       4  Date         M   --                 211    --
  3    4        21       4  Time         M   --                 211    --

Legend:
 < = Descending Order
 D = Duplicates Allowed
 I = Case Insensitive
 M = Modifiable
 R = Repeat Duplicate
 A = Any Segment (Manual)
 L = All Segments (Null)
 * = The values in this column are hexadecimal.
?? = Unknown
-- = Not Specified

The command completed successfully.

C:\>_
```

Multiple Record Definitions

The second obstacle to full SQL access is record redefinition. The ISAM data model forces a single definition for key data but allows redefinitions of non-key data. The relational model (on which SQL is based) requires that every field of data represent exactly one kind of information (e.g. invoice date).

Step 2: Resolving Incompatibilities

After evaluating the data, the application may have a list of instances of type mapping and redefined records that need to be resolved before the data can be fully accessed by SQL. If not, then congratulations – you are ready to jump to Step 3: Describing the Data.

For each type of incompatibility, there are several possible solutions which vary in the amount of effort required and the impact on your application. The most complete solutions usually require the most effort. Resolving incompatibilities in a way that fully enables SQL access generally involves more rework and has a greater impact on the application than the workarounds. (There are times in pure SQL based applications where there really is an all-or-none situation. However, when coming from a navigational/ISAM approach, it's possible to allow partial SQL access and get the benefits of SQL without introducing integrity issues.)

Resolving Unsupported Types

If the compiler is creating keys for certain compiler types that do not sort correctly, then you have unsupported types. This means that in addition to sorting incorrectly, any SQL calculations (such as SUM) on this data type will also fail. If there are one or two instances of unsupported types and they are neither frequent nor particularly important, it's reasonable to stick with the compiler defaults for those types for the short term. The drawback of this approach is that SQL access to these fields will show garbage data.

Occasionally, the byte layout of a compiler type matches that of a database type, even though the compiler does not describe the type to Btrieve correctly when it creates keys (i.e., the compiler has made an inaccurate choice in data type mapping). In this case, it's possible to create the file outside of the application using the correct types. If the application creates the file, the end result will be incorrect types. This can be tricky to set up and maintain, so this solution is not ideal, but can be useful in some cases. To use this approach, verify that the compiler type and database type are identical at the binary level across a wide range of values.

If the types are incompatible and frequently used or very significant (or the goal is full SQL access with no workarounds), you may need to make application changes. Changing the types used in the FD to compatible types is not too difficult. More often than not, when exact representation is important, a particular piece of data is MOVE'd to a holding record (e.g., report layouts). In this case, the only requirement for the new type is that the compiler treats it qualitatively similar to the old type. For example, say that a FD contains a picture of 9(5) COMP and the compiler correctly maps 9(5), but not 9(5) COMP.

Changing the type to 9(5) will have few other code implications (although it will have deployment implications, since the physical record layout has now changed).

Resolving Redefined Records

In the event that the COBOL Schema Executor does not meet your needs for dealing with redefined records, try these three workarounds – Partial Record Definition, Multiple Definitions, and Offline Normalization.

Partial Record Definition – One of the workarounds for redefined records is to expose only one of the definitions of the record through SQL. For instance, the application may have a control file that has one record definition each for systems 1, 2, 3, and 4 (e.g., general ledger, payroll, accounts receivable, and accounts payable). If the only objective is to enable SQL access for system 2, then the best approach is to describe the control file to SQL as it is defined for use by system 2. Even when the decision is not as clear-cut as that, there are also times when the data of interest is isolated to an amalgamated definition. If SQL access to bytes 5-16 as record definition 1 is required for a G/L report and access to bytes 21-26 as record definition 2 is required for an A/P report, then one could describe a record definition to SQL that doesn't actually exist in the file, but that provides access to the largest amount of the data of interest. The down side of this is that bytes 17-20 might be meaningless (which would be shown as garbage data on a SQL query), but it would be possible to get to the fields you needed the most.

Multiple Definitions – Another common workaround for this problem is to have multiple definitions for a particular file. When describing the data to the SQL engine, it's possible to provide several table definitions referencing the same file, each with a unique table name. All of these descriptions should be set up to point to the same file. The benefit to this workaround is SQL access to all of the data with no application changes. The downside is that the proper filter on the record type column must always be applied, or queries to any of these tables will return garbage data for each record of another type. Another drawback is that SQL access would have to use different table identifiers than the COBOL application.

For example, let's say you have record types 1 and 2 in file A. You might describe two table definitions to Pervasive PSQL and name then A1 and A2. All table access would be performed through the names A1 or A2, but queries for all records in A1 would return all records and those that are intended for A2 would have garbage in the A1- specific fields. SQL WHERE clauses and filters could be used to build queries that only returned appropriate records, but unless SQL access was massaged by your application, all queries would have to include that business logic. Views can be used to help mitigate this problem (see the Pervasive PSQL documentation on SQL views).

Offline Normalization – Another approach that is probably only suitable for relatively static control records is occasional offline normalization. First, write an auxiliary application that writes the redefined records into their own separate files. Then, describe the created files to the SQL engine. The benefit of this workaround is that it requires no changes to existing code and only one very small application. The downside is that it is not "live" and that it requires process (someone has to remember to run the application whenever the control data changes). This could be chained from the control record modification program, which would require a modicum of code change, but would reduce the likelihood of reporting on invalid data.

The way to fully resolve this problem is to normalize all the redefined records across multiple files. This can require significant application changes (including changes to the logic). There may also be times when the business logic of an application depends on redefines -- for instance, record redefinition may be used to ease reporting tasks by lining up two types of related records in key order using a single cursor. This approach will not work in the SQL world, as each file description has an independent cursor. Reporting on this data would require two file links.

Step 3: Describing the Data

At this point in the process, a developer will have an understanding of the data to be described to the Pervasive engine to enable the SQL access required. Pervasive provides a tool to help describe tables called DDF Builder. This Java-based utility is used for creating, changing, and viewing the data dictionaries that describe the underlining metadata for the corresponding database files. DDF Builder can be downloaded as part of the SDK at http://www.pervasive.com/developerzone/sdk/ddfbuilder.asp.

Because DDF Builder allows you to change both metadata and the data files, it is important to make a backup copy of your data before starting.

It's important to understand the terms that DDF Builder uses. Remember that the relational model differs from ISAM in that it presupposes a well-known definition of the files that comprise the database. Also, files are referred to as tables in the relational model and fields are referred to as columns.

Start by creating a database and adding descriptions to an existing data file. DDF Builder allows you to describe data using SQL data types and descriptors (offset, length, etc). You will also be able to preview your configuration as you map the data to verify compatibility and correctness.

Fields are set as nullable by default, which allows accurate recording of the absence of data. Since this option changes the offsets of all fields after the null, accepting the default can drastically change record layout. It is very unlikely that your application's COBOL compiler supports true NULL's, so set all columns as non-nullable (uncheck the NULL checkbox). Finally, it will make deployment simpler by using relative paths in the data description. After describing a file in DDF Builder you can then use the Pervasive Control Center (PCC) to view the SQL layout and structure.

For those with SQL backgrounds, you may want to avoid describing key relationships. In the relational model, primary/foreign key relationships imply relational integrity (RI). RI is described extensively in the Pervasive PSQL documentation, but in a nutshell, the COBOL application probably assumes the DBMS does not support RI. Changing this may cause unexpected errors in your application – establishing a primary/foreign key relationship implicitly adds new database rules that affect when and how records may be deleted and changed.

Step 4: Deployment

Always make a backup of the production data before beginning deployment.

If there have been no changes to any record layouts or workarounds to "trick" the compiler/runtime, then there should be no changes to production data.

The next step, deploying the data definitions, consists of two parts: defining the database as an entity, and defining the data files that comprise the database. In most cases, particularly with only a few deployments, it is easier to use PCC to create the empty database, just as in Step 3. There are now at least three ways to deploy the data file definitions.

Option One is to use DDF Builder to describe the data and resolve any null issues on each deployment machine (usually the server). This is likely to be time-intensive if there are a large number of deployments, so this is usually not the best way to solve the problem.

Option Two, the quickest solution, is to copy the DDF's from your development machine (where you performed step 3) to the deployment machine. However, this will only work if either the DDF's contain relative paths or the paths on the two machines are identical.

Option Three, if you are familiar with SQL, is to use (and save) SQL scripts to create the descriptions on the deployment machine. This will only work with Pervasive.SQL 2000i SP4, Pervasive.SQL v8 and later, since it requires the USING clause to apply definitions to existing data files rather than creating new, pristine (and disappointingly empty) data files.

Once the data file definitions have been deployed, it will be possible to load PCC, Crystal Reports or any SQL tool on the server and access the data. This is pretty exciting, but there is still one more step: client DSN's. DSN's are required for ODBC access. It's possible to use ADO and some other methods without DSN's, but many of the third party tools use ODBC since it is the most widely used standard for SQL access. Client access can be made over the network using the DDF's set up on the server (and this is ideal for the additional navigational API's now available), but ODBC access will usually work much better if client DSN's are set up on the clients. Client DSN's are discussed in the Pervasive PSQL documentation and have slight variations from release to release.

If the record layout or key types have changed as a result of this process, then existing data will need to be converted. If data had to be mapped in a manner contrary to the compiler (i.e., the compiler/runtime always saves PIC 9(5)'s as strings, but they should be stored as a native type DECIMAL), then create the files outside of COBOL and populate them with a COBOL application. For this reason, this approach is not very robust and should be considered as one of the last resort solutions – although for identical data the compiler's perception of the data type should not affect inserts, updates, and reads, it will affect any file create requests.

Conclusion

If you've read this far, congratulations! You've just opened up a new world of functionality to your application while retaining the investment in your existing code base. Your existing applications should work unchanged (both in terms of function and performance), and you can use the power of SQL and the new increased platform support to provide your users with more capabilities than before.

Contact Information

Pervasive Software Inc.
12365-B Riata Trace Parkway
Austin, Texas 78727

United States
800.287.4383
512.231.6000
Fax: 512.231.6010
info@pervasive.com

EMEAA
+800.1212.3434
cic@pervasive.com

http://www.pervasive.com

© 2006 Pervasive Software Inc. All rights reserved. All Pervasive brand and product names are trademarks or registered trademarks of Pervasive Software Inc. in the United States and other countries. All other marks are the property of their respective owners.