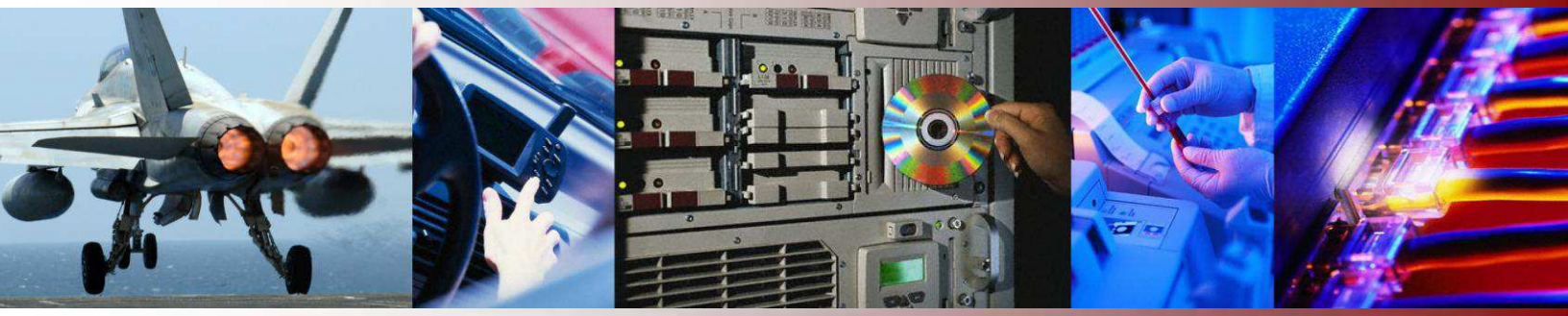

IS USING SQL IN AN EMBEDDED COMPUTER APPLICATION LIKE TRYING TO SQUEEZE AN ELEPHANT INTO A MINI?

A Raima Inc. Technical Whitepaper

Published: April, 2010
Author: Randy Merilatt
Distinguished Engineer

Abstract

SQL, like it or not, has become the industry's standard database access language. This being the case many companies that are involved in the development of embedded computer applications with database management requirements would like to be able to use SQL to access and manipulate that database information. This article takes a look at ways developers of embedded applications can efficiently make use of SQL in their applications.

This article is relative to the following versions of RDM:

- ✓ RDM Embedded 9.x, 10.0

Contents

Introduction

According to Wikipedia's entry entitled "Elephant joke", there's an old one that goes like this:

> Q. How many elephants will fit into a Mini?
> A. Four: two in the front, two in the back.
> Q. How many giraffes will fit into a Mini?
> A. None. It's full of elephants.

Of course, if it is possible to get four elephants into a Mini then it must be pretty easy to get one in. In which case, there must also be no problem using SQL in an embedded computer application! But, even if one does succeed in getting the elephant into the car, the added weight will certainly have a significant negative impact on its speed. Such is the opinion of many—including me—on the advisability of using SQL in an embedded database application. The most recent edition of Volume 2 of the ANSI/ISO SQL standard is over 1300 pages long. That's about twice the size of the 1992 standard which itself was considerably larger than the original 1989 standard. A fully-compliant implementation of SQL (which I do not know actually exists) is indeed a monster. For any SQL DBMS implementer, just the effort involved to understand the standard in order to construct a commercially-viable, fully-compliant implementation is immense.

Nevertheless, SQL has become the industry standard database access language. As such, there are many software developers who know how to use SQL. Because of this vast availability of SQL database skills, many companies that are involved in the development of embedded computer applications with database management requirements would like to be able to use SQL to access and manipulate that database information.

The DBMS capabilities that are needed in embedded computing applications are not nearly as broad as those needed in enterprise systems. The purpose of this paper is to identify the features of SQL that are not useful in embedded applications and to show some of the ways that Raima is adapting an abridged version of SQL that allows it to be tightly integrated with the application.

An Abridged SQL for Embedded Applications

What should an[1] SQL DBMS for embedded applications look like? First of all it needs to look more like a greyhound (dog, not bus) than an elephant. Embedded computing environments often have resource limitations and execution timing requirements that cannot tolerate applications that consume a large amount of memory and time. Hence, only those DBMS capabilities that are really needed should be provided in order to minimize the memory consumption of the DBMS.

The following features of standard SQL are not features that are typically needed in embedded computing environments. Of course, there may be legitimate situations where this assumption is not valid but we maintain that for the vast majority of embedded applications these features are not required. The primary goal is to provide basic SQL capabilities in as "lean and mean" an implementation as possible.

[1] There is often uncertainty about the indefinite article ("a" or "an") to use with "SQL." It really all depends on how one is supposed to pronounce "SQL." Is it "sequel" or is it "ess kue ell?" According to Wikipedia (http://en.wikipedia.org/wiki/SQL), it is pronounced according to the latter.

Security

The SQL **GRANT** and **REVOKE** statements are provided to specify which parts of a database a particular user or class of users can access as well as the kinds of access they can perform. Embedded applications typically execute in environments where the number of users (not necessarily human) is restricted to one or just a few. This is not to imply that security is not an important issue in embedded applications. It is just that the kind of needed security is quite different than that which is provided through SQL. The device on which the embedded application runs does not typically have a lot of different users accessing the data. Also, SQL security is designed to provide the ability to isolate different segments of the data to different users. This is simply not how data on an embedded device is typically used.

Views

Views are typically used in conjunction with SQL security to define which segments of the data can be accessed by particular users. Thus, the same argument against SQL security applies to views as well.

Check Integrity Constraints

The **CHECK** clause of the **CREATE TABLE** statement is used to define constraints on the data that can be stored in the table. Now embedded applications do indeed have important integrity constraints that need to be enforced. It is just that doing so in an interpreted expression evaluation method embedded within the SQL system is probably not the most efficient way to implement them given the timing and memory constraints in a typical embedded application.

Triggers

Triggers are stored procedures that are automatically "fired" by the SQL system when the conditions on which the trigger is defined are met. These conditions involve updates to specified columns in the table. Trigger actions usually involve making changes to other tables which can themselves have triggers. Because it is very difficult to control the timing behavior associated with triggers they are often not really all that suitable for embedded applications. This is not to imply that the trigger concept is incompatible with embedded applications. On the contrary, as an event-driven mechanism it is conceptually at least very useful. The issue, however, lies with which part of the system maintains control. Our contention is that it is best for the application code itself to maintain control rather than within a 3rd party SQL DBMS implementation of triggers.

Static versus Dynamic DDL

Dynamic DDL allows the structure of a database (as defined by SQL DDL statements) to be modified "on the fly." This provides for great flexibility and ease of making system upgrades that involve changes to existing database structures. This flexibility does not come without a cost in the amount of additional complexity needed to accommodate this structural flexibility. Alternatively, while surely less flexible, static DDL is much less complex and processing is usually much more efficient—a situation that is preferable for embedded applications.

Integrating SQL with Embedded Applications

Raima is incorporating several special features into its new RDM Embedded SQL system that allow it to be tightly integrated with an embedded computing application. While these features are not particularly elegant, they are compact, efficient, easy to use and very practical. Three of the key features which will be described in more detail in the remainder of this article are:

- Compiled Database Catalog Modules,
- Static execution of pre-compiled SQL statements,
- User-defined, C-based functions, procedures, and virtual tables.

Compiled Database Catalog Modules

SQL DBMS implementations store the meta-data associated with a particular database in what is usually called the "system catalog." The meta-data includes the database's definitions of tables, columns, foreign and primary keys, constraints, security information, views, and stored procedures among other things. This data is often stored in its own database. In fact, RDM Server SQL does just that. Access to the catalog information is needed to compile and execute SQL statements. It is particularly important in an embedded application that the cost to access catalog information be as small as possible. To that end, RDM Embedded (RDMe) SQL provides the option where the catalog information for a database can be encapsulated in a C program module that can compiled in directly with the application program.

For example, the following gives the RDMe SQL DDL specification for a very simple personal library database.

```
create database mylib;

create table author
(
    id char(5) primary key,
    name char(48) not null
);
create table genre
(
    class char(3),
    code char(3),
    descr char(32),
    primary key g_key(class, code)
);
create table book
(
    id integer primary key,
    title char(72),
    publ_info char(50),
    g_class char(3) not null,
    g_code char(3) not null,
    auth_id char(5) not null references author,
    foreign key genre_id(g_class, g_code) references genre(class, code)
);
```

RDMe SQL will compile these statements and store the information in the following C file called `mylib_cat.c`.

```
/* RDMe-Created SQL catalog initialization file: February 10, 2010 */

#include "mylib_dbd.h"
#include "sqlcat.h"

SYSDOMAIN * mylib_domain = NULL;
SYSTYPE * mylib_type = NULL;
SYSTABLE mylib_table[] = {
    {"author",10000,74,0,2,0,1,0,0,0,0,0,0,'n',55,1,NULL},
    {"genre",10001,60,0,3,2,1,1,0,0,0,0,0,'n',41,1,NULL},
    {"book",10002,159,0,6,5,1,2,2,0,0,0,0,'n',128,2,NULL}
};
int16_t mylib_colkeys0[] = {0};
int16_t mylib_colkeys2[] = {1};
int16_t mylib_colkeys3[] = {1};
int16_t mylib_colkeys5[] = {2};
SYSCOLUMN mylib_column[] = {
    {"id",1,6,0,-1,-1,0,0,0,10000,0,0,1,1,mylib_colkeys0},
    {"name",1,49,6,-1,-1,0,1,1,10000,0,0,0,0,NULL},
    {"class",1,4,0,-1,-1,1,0,1000,10001,0,0,1,1,mylib_colkeys2},
    {"code",1,4,4,-1,-1,1,1,1001,10001,0,0,1,1,mylib_colkeys3},
    {"descr",1,33,8,-1,-1,1,2,1002,10001,0,0,1,0,NULL},
    {"id",10,4,0,-1,-1,2,0,2000,10002,0,0,1,1,mylib_colkeys5},
    {"title",1,73,4,-1,-1,2,1,2001,10002,0,0,1,0,NULL},
    {"publ_info",1,51,77,-1,-1,2,2,2002,10002,0,0,1,0,NULL},
    {"g_class",1,4,0,-1,-1,2,3,-1,10002,20001,2,0,0,NULL},
    {"g_code",1,4,0,-1,-1,2,4,-1,10002,20001,3,0,0,NULL},
    {"auth_id",1,6,0,-1,-1,2,5,-1,10002,20000,0,0,0,NULL}
};
int16_t mylib_keycols0[] = {0};
int16_t mylib_keycols1[] = {2,3};
int16_t mylib_keycols2[] = {5};
SYSKEY mylib_key[] = {
    {"id_SYSK01",3,3,0,0,1,mylib_keycols0,0,0,0,'n'},
    {"g_key",3,1004,1,0,2,mylib_keycols1,0,0,0,'n'},
    {"id_SYSK03",3,2004,2,0,1,mylib_keycols2,0,0,0,'n'}
};
int16_t mylib_refcols0[] = {10};
int16_t mylib_refcols1[] = {8,9};
SYSREF mylib_ref[] = {
    {"auth_id",20000,0,0,2,'r','r',1,mylib_refcols0},
    {"genre_id",20001,1,1,2,'r','r',2,mylib_refcols1}
};
SYSDB mylib_cat = {
    "mylib",'n',0,0,11,0,0,3,11,3,2,
    (void *)&mylib_type,(void *)&mylib_domain,(void *)&mylib_table,
    (void *)&mylib_column,(void *)&mylib_key,(void *)&mylib_ref,
    (void *)&mylib_dbd,sizeof(mylib_dbd),0
};
```

A file named `mylib_cat.h` is also created containing an external declaration to the `mylib_cat` global variable. By the way, RDMe SQL maps the SQL DDL into RDMe core-level DDL and then invokes the core-level DDL processor (DDLP) to compile the core DDL. The "database dictionary" is created by the core DDLP and it stores it in a C file called `mylib_dbd.c`. An external reference to the core dictionary global variable is contained in `mylib_dbd.h` which, as you can see in the above example is included in the `mylib_cat.c` module.

RDMe SQL provides the user with the ability to specify the address of the `mylib_cat` struct through a call to **SQLSETCONNECTATTR** so that the SQL system knows where to find the catalog information when the database is

first opened when **SQLCONNECT** is called. Thus, the system catalog data becomes an integral part of the application code and does not need to be stored in a file. However, the catalog (and dictionary) data is also written to standard, binary files. Hence, the database meta-information can be compiled in with the application or maintained separately in an outside file.

Static Execution of Pre-Compiled SQL Statements

SQL is designed to provide the ability to dynamically compile and execute SQL statements. This allows users to issue a wide variety of queries on the database information. Figure 1 gives a diagram that shows a typical RDMe SQL application that includes the ability to compile and execute SQL statements. (The RDMe TFS refers to the "Transaction File Server" which is a separate program that manages all database I/O, transactions, and locking).

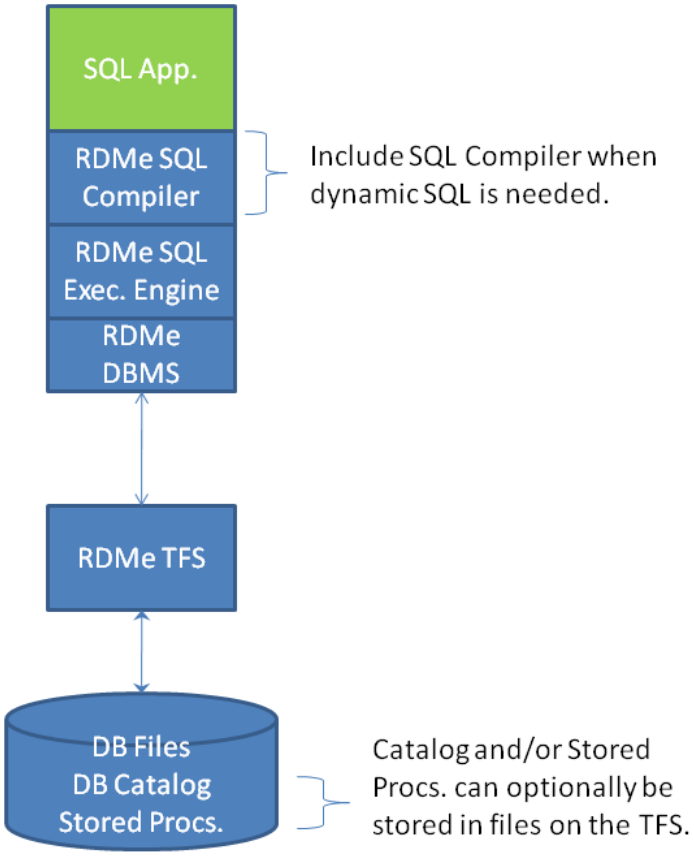

Figure 1 - Dynamic RDMe SQL Application

Embedded applications, however, typically have well-defined data access and manipulation requirements and so they just simply do not need to have the ability to support ad hoc query processing. As much as 25-30% of an SQL implementation goes to the support of dynamic compilation. Thus, if this can be eliminated from the embedded application code, a not insignificant amount of memory can be saved.

In order to do this, RDMe SQL provides the ability to define a basic stored procedure that can contain either one or more **SELECT** statements or one or more **INSERT**, **UPDATE**, or **DELETE** statements. These statements would be compiled on a host development computer system. The compiled form of the stored procedure is stored in both a C file and a binary file. Like the catalog modules described earlier in this article, the stored procedure C file can be compiled with the application.

For example, the following SQL stored procedure retrieves the row of table book with id equal to the specified argument.

```
create procedure getbook( bid smallint ) as
    select * from book where id = bid
end proc;
```

SQL stores the compiled procedure in a binary file called `getbook.ssp` and in a C file called `getbook_ssp.c`. A portion of the contents of the C file is shown below.

```
/* RDMe-Created SQL Stored Procedure File: March 23, 2010 */

#include "rsqltypes.h"

RSQL_VALUE getbook_args[1] = {
    {9}
};
STMT_DESCR getbook_descrs[1] = {
    {3, 360, 0}
};
uint8_t getbook_stmts[360] = {
    0x03,  0x00,  0x00,  0x00,  0x68,  0x01,  0x00,  0x00,  0x00,  0x00,  0x05,  0x00,
    . . .
    0x00,  0x00,  0x75,  0x17,  0x00,  0x00,  0x00,  0x32,  0x90,  0x00,  0x00,  0x00
};
PROC_EXEC getbook_defn =
{"getbook", 0, 1, 1, 0, 360, getbook_args, getbook_descrs, getbook_stmts, NULL};
…
```

The application program calls an RDMe SQL API function passing in the address of the `getbook_defn` struct to directly execute the stored procedure.

Given that all necessary database operations have been encapsulated in pre-compiled stored procedures the embedded application executable code does not need to include the SQL compilation module. Figure 2 depicts a static RDMe SQL application in which the catalog and stored procedures have been compiled in with the application program and since no dynamic SQL is required, the SQL compilation library is not linked in with the application code. (Note that this diagram also shows that it is possible for the application to access databases that are managed by more than one TFS).

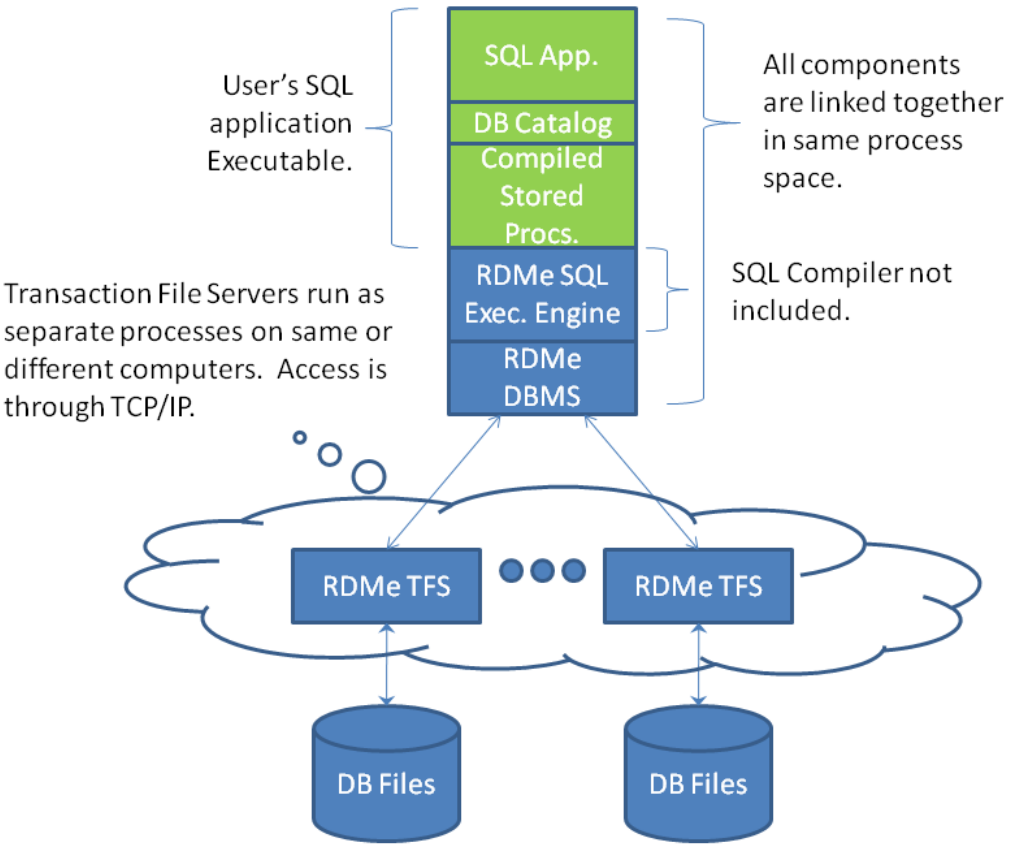

Figure 2 - Static RDMe SQL Application

User-Defined, C-Based Functions, Procedures, and Virtual Tables

RDMe SQL provides three different but related methods by which special purpose processing and data requirements can be incorporated into SQL. All three methods are implemented by the embedded application developer in C/C++. A *User-Defined Function* (UDF) is a scalar or aggregate function that can be used in any SQL expression. A *User-Defined Procedure* (UDP) is a C-based module that looks to SQL just like a stored procedure. An *External Table* provides the ability to present any kind of data to SQL as a table. As these methods are similarly implemented, for the sake of brevity, only the details of the external table implementation will be presented here.

An RDMe SQL external table is defined through a combination of a special DDL **CREATE TABLE** statement and a set of user developed C functions that conform to a particular interface specification. A pointer to a pre-defined structure array that contains an entry for each external table with the addresses of each of the external table interface functions is passed into SQL before the database is opened. These functions are then called by SQL at the appropriate times during the execution of any SQL statement that references the external table. This interaction is depicted in Figure 3 which shows SQL calling the function in the application's external table function module to fetch a row of weather data from a wireless sensor network (WSN). Note that in this example by storing the data retrieved from the external table in a standard table, RDMe can then replicate that data to an outside host DBMS (e.g., RDM Server, or, if you absolutely have to, some other well-known SQL DBMS).

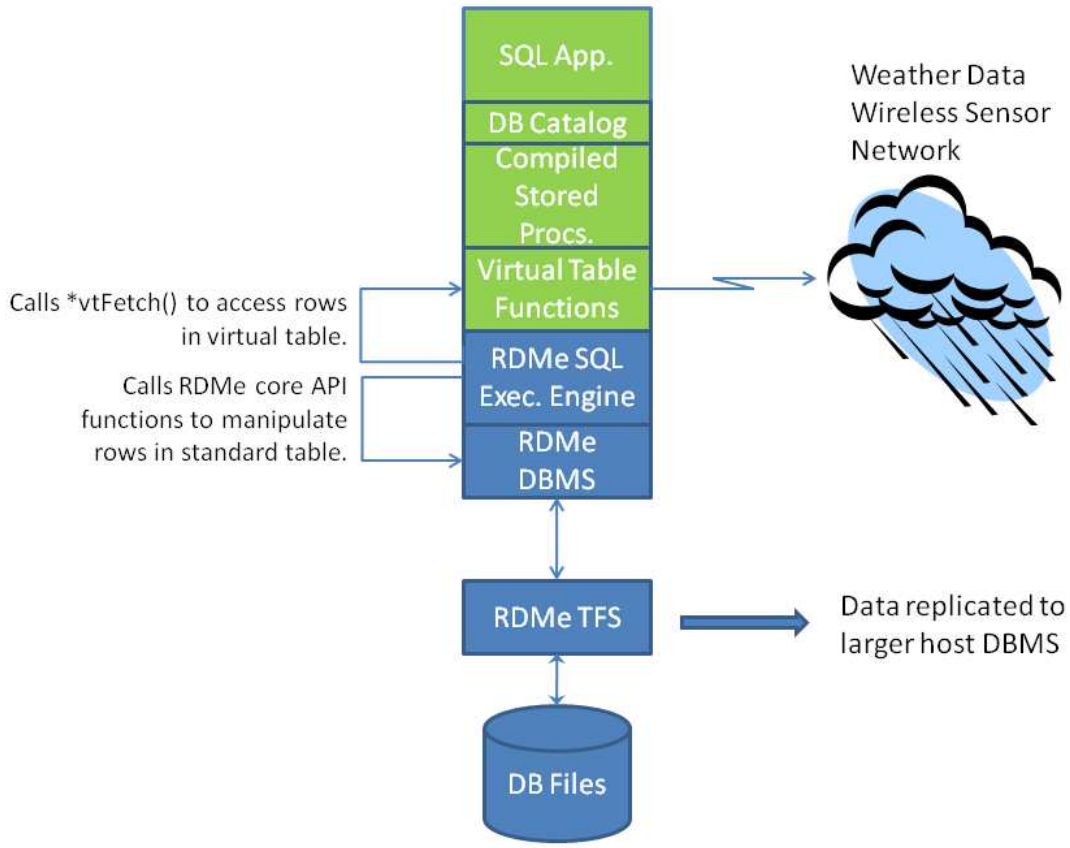

Figure 3 - RDMe SQL Virtual Tables

In a DDL specification, all external table declarations must come after all standard tables have been declared. For example, the DDL specification for our example weather data WSN might look like the following:

```
create database weather_db;
create table location(  /* location of weather sensor */
    longitude integer,
    latitude integer,
    sensor_id bigint,
    descr char(48),
    county char(24),
    state char(2),
    primary key loc_id(longitude, latitude)
);
create table weather_summary(
    longitude integer,
    latitude integer,
    rdg_date date,
    hour_of_day smallint,
    avg_temp smallint,
    avg_ press smallint,
    avg_hum smallint,
    avg_lumens smallint,
    foreign key (longitude, latitude) references location
);
```

Continued…

```
create external readonly table weather_data(
    sensor_id bigint primary key,
    loc_long integer,
    loc_lat integer,
    rdg_time timestamp,
    temperature smallint,
    pressure smallint,
    humidity smallint,
    light smallint,
    power integer
);
```

Note that in this example, the external table **weather_reading** is a read only table. Hence, it can only be referenced in a **SELECT** statement. It is also possible to have an external table be updateable so that it can be referenced in an **INSERT**, **UPDATE** or **DELETE** statement.

The function entry points are declared in the external table function load table as shown in the C code excerpt below.

```
XTFLOADTABLE wdFcnTable[] = {
    {"weather_data", wdInit, wdRowCount, wdExecute, wdFetch, wdClose, wdTerm}
};
```

The XTFLOADTABLE type definition is as follows.

```
struct xtfloadtable {
    char         xtName[NAMELEN]; /* name of the external table */
    PXTINIT      xtInit;          /* ptr to initialization function */
    PXTROWCOUNT  xtRowCount;      /* ptr to function that returns the current # rows */
    PXTEXECUTE   xtExecute;       /* ptr to execution function */
    PXTFETCH     xtFetch;         /* ptr to fetch next row function */
    PXTCLOSE     xtClose;         /* ptr to close out select processing */
    PXTTERM      xtTerm;          /* ptr to termination function */
} XTFLOADTABLE;
```

An excerpt from the `wdExecute` and `wdFetch` functions is given below.

```
/* ============================================================================
    External table execution function
*/
static RSQL_ERRCODE EXTERNAL_FCN wdExecute(  /* xtExecute() */
    STMT_TYPE    stype,    /* type of statement (sqlSELECT, sqlINSERT, etc) */
    int16_t      nocols,   /* no. of ref'd columns */
    XCOL_INFO   *colsvals, /* array of ref'd column value containers */
    RSQL_VALUE  *pkeyval,  /* ptr to primary key's value */
    WD_CTX      *pCtx)     /* context pointer */
{
    . . . /* save the nocols & colsvals in the pCtx structure */

    if ( pkeyval ) {
        . . . /* set up to locate the specified weather sensor */
    }
```

Continued…

```
return errSUCCESS;
}
. . .
/* ============================================================================
    External table fetch function
*/
static RSQL_ERRCODE EXTERNAL_FCN wdFetch(  /* xtFetch() */
    WD_CTX    *pCtx)         /* context pointer */
{
   if ( no more sensors )
      return errNOMOREDATA;

   . . . /* access the next (or specified) sensor */
   . . . /* save the readings in the appropriate column value containers */

   return errSUCCESS;
}
```

The `wdExecute` function is called by SQL when the **SELECT** statement that references table `weather_data` is executed. The `STMT_TYPE, XCOL_INFO, RSQL_VALUE` data types are declared in a standard RDMe SQL header file while the `WD_CTX` type is declared by the application developer in this C module. The `pCtx` pointer is allocated by the `wdInit` function which is called by SQL when the database is first opened. The details of these types are not important for this discussion except to note that the `XCOL_INFO` type contains information about the table column and a pointer to where that column's data value is to be copied when `wdFetch` is called. Hence the need to store these argument values in the context structure pointed to by `pCtx`.

Function `wdFetch` is called by SQL during **SELECT** statement processing to retrieve the next row of data from the weather sensor network. In this example, a given **SELECT** will make one reading from either a single sensor (specified by the `pkeyval` argument to `wdExecute`) or all available sensors. After the last sensor has been read, status code `errNOMOREDATA` is returned (these status codes are also defined in a standard RDMe SQL header file).

Only a little imagination is needed to see that data from sources such as a wireless sensor network has no natural end. As long as the sensors continue to operate, data will always be available. This presents a particularly difficult problem when the data needs to be summarized over some aggregate collection. In the above example, a standard table is declared in the database that contains the averages of the readings from each sensor as collected over each hour of the day. In order to compute these aggregated values, SQL needs to sort the fetched rows by `sensor_id` and `rdg_time` (timestamp when the sensor data was read). But any sort needs to have a fixed number of rows. How is this done when there are an unlimited number of rows?

To address this problem, the RDMe SQL **SELECT** statement includes a clause that can limit the number of rows that are returned as shown in the following syntax.

```
select_stmt:
   select … from external_table where … limit( num limit_unit)
limit_unit:
   rows | hours | mins | secs | msecs
```

The **LIMIT** clause limits either the number of rows that are returned or the amount of time the **SELECT** statement is allowed to run. Thus, the following example shows a **SELECT** statement that stores the averages per hour from each weather sensor in the `weather_summary` table.

```
insert into weather_summary
   select loc_long, loc_lat, convert(rdg_time,date), hour(rdg_time),
      avg(temperature), avg(pressure), avg(humidity), avg(light) from weather_data
      group by 1,2,4 limit(4 hours);
```

Each row is fetched and sorted over each four hour span of time. At the end of that time, the sorted data is scanned and the aggregate calculations performed and the resulting rows are then stored in the `weather_summary` table. The time limit can be shorter but, in this case, not any less than an hour as that is the smallest unit over which the aggregation is made (of course, this assumes that the **SELECT** is synchronized to execute at the start of an hour).

Hopefully, from this discussion, the usefulness and power of virtual tables in RDMe SQL for embedded computing applications is clear.

Other Significant RDM Embedded Features

In addition to the SQL features described above, RDM Embedded provides a variety of capabilities that are also important in embedded applications:

- Replication: the ability to replicate all or parts of a database between the RDM Embedded database and other databases (RDM Embedded, RDM Server, or other 3rd party SQL DBMSs).
- Mirroring: the ability to efficiently mirror a database on one or more other RDM Embedded transaction file servers. This can be used to maintain backups or copies on other computers that are not subject to the operational constraints of the embedded computing device.
- Read-only Transactions: allow transaction-consistent reading of database content that does not lock out other update transactions.
- In-memory tables and/or indexes: some or all of a database can be kept entirely in-memory while still maintaining full ACID database properties.

Conclusion

By shrinking the elephant down to the size of, say, a greyhound, it can now easily fit into the Mini. Whether or not this satisfies the need to use "standard" SQL in embedded applications will ultimately be decided by the embedded application developers themselves. Nevertheless, we at Raima are convinced that the SQL we're developing will provide just the right choice of features that balance the resource and performance requirements of many embedded applications with the ease of use of SQL.

Contact Information

Raima® Inc.

Website: http://www.raima.com

NORTH AMERICA

Raima Inc.
720 Third Avenue
Suite 1100
Seattle, WA 98104
Telephone: +1 206 748 5300
Fax: +1 206 748 5200
E-mail: sales@raima.com

EUROPE

Raima Inc.
Stubbings House
Henley Road
Maidenhead SL6 6QL UK
Telephone: +44 7786 176 375
E-mail: sales@raima.com