RaimaDB Version 16 Architecture and Features

By David Nguyen, Director of Engineering & QA – February 2024

Abstract

RaimaDB 16.0 is an embeddable database management product which connects all types of devices from the edge of the cloud to enterprise databases completing the Internet of Things platform. Raima´s libraries, APIs and utilities can be employed in countless combinations, assuming the responsibility for data collection, storage, management and movement. Programmers using RaimaDB v16.0 can focus on their specialty rather than worry about managing their data. Given the variety of computing needs, a "one shape fits all" product will quickly be pushed beyond its limits. This paper discusses RaimaDB 16.0's different configurations and solutions built from those pieces into a number of configurations.

RaimaDB 16.0 can be used for meaningful solutions in multiple industries and deployment platforms, including Automotive, Industrial Automation, Aerospace and Defense, Oil and Gas, Telecommunications, and Medical. Deployment platforms include standard desktop computers, cloud, embedded computers and mobile devices. This paper is for those who want more than "what" RaimaDB can do, but also "how" it does it. As much as possible, this paper will just state the facts, and leave the hype out. A basic understanding of application software development is required to fully understand these facts. Also, knowledge of operating environments, such as Linux, Windows, Realtime/embedded operating systems, networking (both Local- and wide area), mobile environments, and computer architectures is assumed.

Prerequisites: A basic understanding of application software development.

Table of Content

The Bigger Picture

RaimaDB is a Software Development Kit (SDK) containing set of libraries and utilities that allow development of programs that perform ACID-compliant database management. RaimaDB becomes a part of the application program, being responsible for the database management requirements. RaimaDB is easily configurable, allowing it to serve a simple role within a single process in a single computer, or a much more sophisticated role in a wide-area network of cooperating computers and processes sharing distributed databases. Figure 1 shows RaimaDBs relationship to application code and its operating environment.

RaimaDB is fast and optimized. It contains an in-memory implementation designed specifically for quick data access and modification but also with the flexibility of being later stored on disk as needed. The on disk implementation utilizes an encoded database pack file format to also streamline insertions and updates. RaimaDB is easy for developers to use. It supports both proprietary and standard Application Programming Interfaces (APIs). Proprietary APIs are directly callable by C/C++ programs and support low-level database operations, object-oriented operations, or SQL language. Standard APIs support SQL language and SQL PL through ODBC, or JDBC. Nearly any development environment and language a software engineer is familiar with is supported allowing for quick development and deployment.   *Figure 1: RaimaDB*
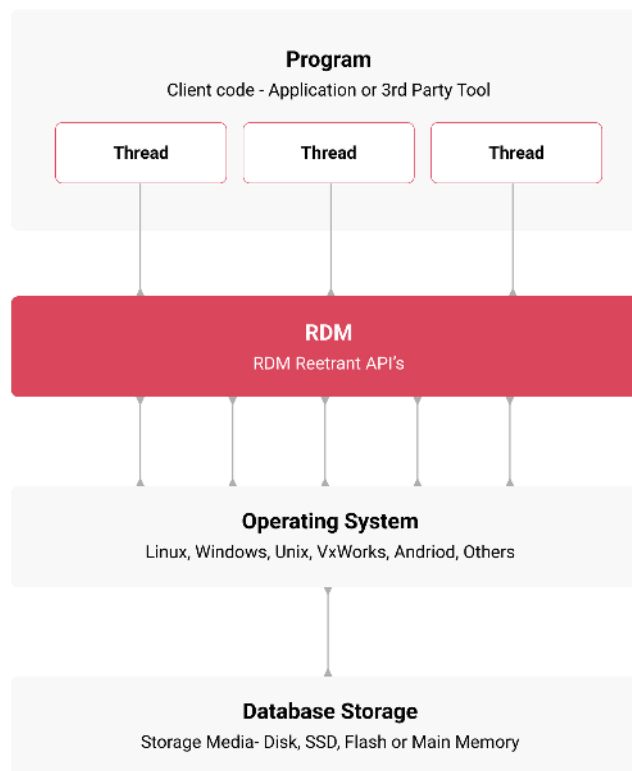
RaimaDB is compact and portable. It is written in the C language and can be built for almost any system that supports a C compiler. Thus, it runs on multiple operating systems, from larger Unix to small embedded or mobile systems. It runs on all common CPUs. It can be built for either 32- bit or 64-bit programs. Its memory and disk footprint are very small compared to other DBMSs with equivalent functionality, so it is available to solve problems on computers not previously considered powerful enough for non-trivial programming. The database files themselves are stored in a proprietary format, allowing the files to be directly copied onto any platform and utilized without data massaging or byte swapping. The RaimaDB APIs are reentrant. This means that programs running 2 or more threads can use RaimaDB concurrently. Multithreading also permits exploitation of multiple CPUs and multiple Cores, enhancing overall performance. The selection of multiple APIs means that you can use one you are familiar with (for example, ODBC or JDBC), or a low-level, high-performance API like the one we call the Core(C/C++). RaimaDB also supports mixing and matching of API's seamlessly and within the same application.

RaimaDB is system software, written in C to eliminate the overhead of higher-level languages. It performs data caching, disk I/O, and networking, implementing efficient database algorithms to manage data and database processing like SQL optimizations and query execution. The result is a fast and efficient system that leaves room for the application program to perform its own specialized functionality.

The remainder of this White Paper looks at components and features of RaimaDB at a high level with hands-on examples. This is not a Users Guide or Reference Manual, and is not exhaustive or rigorous in its descriptions.

Database Packaging and Functionality

RaimaDB comes in two different packages: RaimaDB Core and RaimaDB Enterprise.

RaimaDB Core contains only the core cursor C and C++ API interfaces. These are the underlying and most-optimized APIs designed for use with the C/C++ programming language. It includes all the utilities as well.

RaimaDB Enterprise contains both the core cursor API and the SQL interface in addition to all the remaining APIs. This package allows for the use of the JAVA JDBC interface and the ODBC interface. It also has full support for third-party connectivity and administrative tools, in addition to supporting the full legacy API from previous versions of the RaimaDB product line. It includes the latest REST interface and Administrative GUI as well.

Digging inside this system, we find most of the technology is within the RaimaDB core database engine with the other packages allowing for various interfaces into the database system.

RaimaDB Core Packaging

Storage Media

An RaimaDB is composed of computer files. These files are stored in an operating system's file system, which can be using disk drives, SD RAM, or SSD as the underlying media. RaimaDB uses standard file I/O functions to access the file system. Data stored in the files are portable, or "platform agnostic" so that they may be directly copied between any two computers.

RaimaDB contains has a new data storage engine optimized specifically for working with memory resident data sets. This new In-memory Database Engine (IMDB) allows for significant performance gains and a reduction in processing requirements compared to the On-disk-engine. The RaimaDB IMDB runs alongside the RaimaDB On-disk engine and databases can be opened with either one.

Database Functionality

As a basis for any database, you have a representation of your data, and operations on that data. The representation of the data, which can also be called the data model, is the way the database's user sees the data. Data is created, deleted and changed in this representation through operations on the database. Databases are normally shared and contain valuable information, so the operations on a database must follow carefully defined rules.

This database functionality is implemented in libraries of functions we call the "runtime libraries," referring to what is linked (or "embedded") into an application program. It is also called the Core, because it is available to application programmers and other higher-level database functionality like SQL or the C++ object oriented interface.

Data Modeling

Relational Model

The most commonly understood data model today is the relational model, where all data is defined in terms of tables and columns. We will not define the relational model here but will note that RaimaDB defines a database using SQL, (see below) the predominant relational database language. Relationships in a pure relational model are defined by associating common columns between two tables through what are called primary and foreign keys. Indexing is a common method used to optimize relational queries.

Network Model

Beneath the relational model in a RaimaDB is a network model, where all data is defined in terms of record types ("tables") and fields ("columns"). Fields may be indexed, and record types may have set relationships between them, which are defined as one-to-many, owner/member relationships. One record is an instance of a record type. In relational terminology, a row is a single line in a table. RaimaDB uses the set construct to implement primary and foreign key relationships, which will be shown in the DDL examples below.

Graphical View

The RaimaDB manual frequently uses a graphical representation of tables/columns ("record-types/fields") to show the structure of data in an easily understood form. The following figure shows a simple data model for students enrolled in classes:
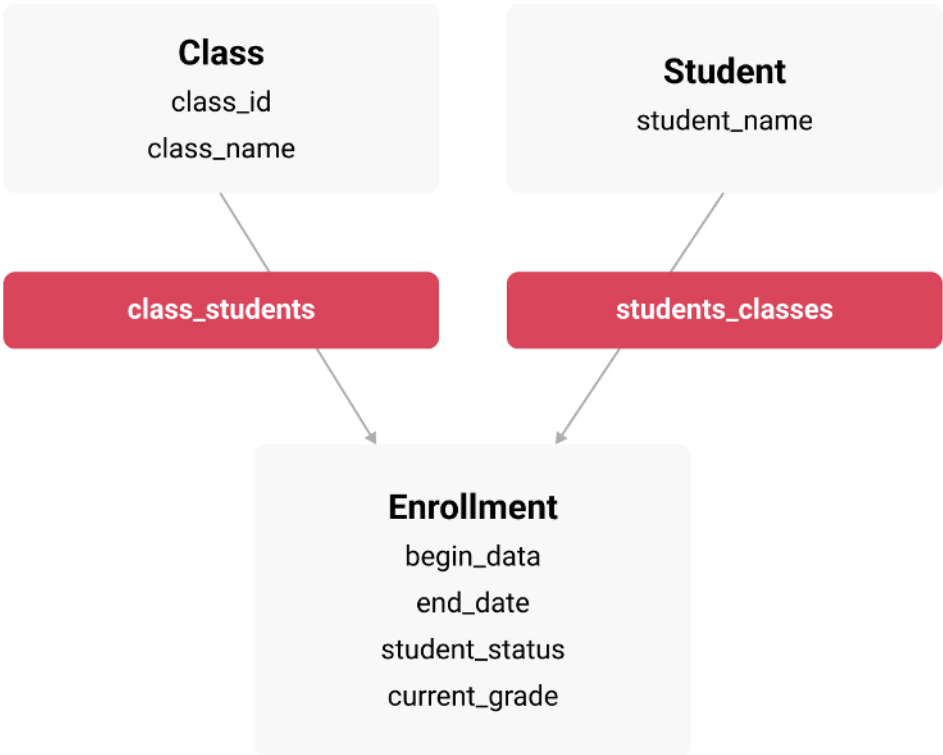

Figure 2 : Students and Classes

Figure 2 shows three table types: class, student and enrollment. There may be any number of classes or students, each represented by a row in the class or student table. If there is a relationship between a student and a class, it is represented by the existence of an enrollment row that is associated with a student and a class. The arrows in the diagram represent a one-to-many relationship, where one student may be associated with 0 or more enrollments, and one class may be associated with 0 or more enrollments. An enrollment doesn't make sense unless it is associated with both a class and a student, so it must always be connected (associated) to one of each.

A typical tabular view, or relational view showing table definitions would be as follows:

class	
class	class

student
student_name

enrollment					
begin_date	end_date	student_status	current_grade	class	student
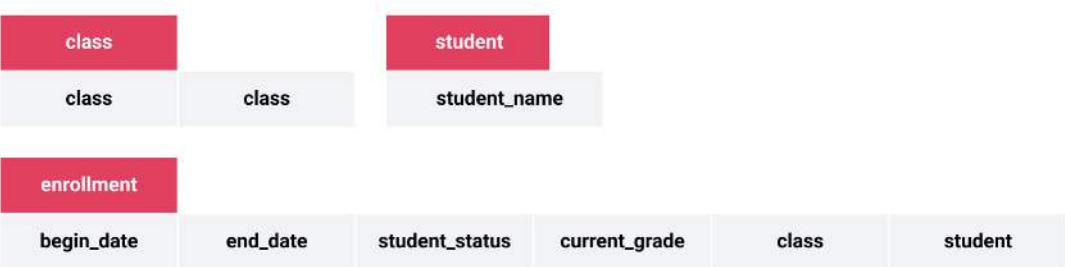
Figure 3: Relational Representation

The tabular view of the data structure makes more sense with an example, shown below with three students related to three classes, each represented by a row in a table:

class	
class_id	class_name
MATH01	Intro to Algebra
SFOT03	Advanced Java
HIST02	World History

student
student_name
Joe
Sarah
Henry

enrollment					
begin_date	end_date	student_status	current_grade	class	student
Sept 8	Dec 20	active	3.1	MATH01	Joe
Jan 4	Mar 31	enrolled		SOFT03	Joe
Sept 8	Dec 20	active	3.2	MATH01	Sarah
Apr 1	Jun 16	passed	3.5	HIST02	Sarah
Sept 8	Dec 20	active	2.6	SFOT03	Henry

Figure 4: Relational Data

The above example data shows Joe attending one class now, and enrolled in one next year; Sarah attending one class, having already passed one earlier; Henry attending one class.

The next section shows the SQL language used by RaimaDB to define this structure and allow the storage and viewing of data.

Data Definition Language (DDL)

Databases are always initially defined as a set of tables and columns. Attributes of the columns and relationships between tables are also defined in the DDL.

RaimaDB compiles DDL and maintains a "catalog" file that is easily ingested by the runtime library. The catalog is formatted as JSON. During the lifetime of a database, the catalog may be modified through database alteration commands.

RaimaDB defines its tables, column, indices and relationships using SQL. To model the student and class data shown above, SQL DDL is written:

```
create database students;
create table class (
    class_id char(6) primary key, class_name char(29)


);
create table student (
    student_name char(35) primary key


);
    create table enrollment ( begin_date integer, end_date integer, status char(9), current_grade integer,
    class_students char(6) references class,
student_classes char(35) references student );
```

This simple database will be used to illustrate some of the following examples.

The following types of data are supported in the database definition language:

Integer – 8-bit, 16-bit, 32-bit, 64-bit, signed or unsigned.

Character – single or string, fixed length, variable or large variable.

Boolean – True or False.

Float, Double - either 32-bit or 64-bit floating point numbers.

Date, Time, Timestamp - date, time or both together.

BCD - decimal numbers with variable precision and length.

GUID - globally unique identifier - 128-bit unique value. Binary – array or blob.

Indexes may be created on columns or combinations of columns. They are used to quickly locate a row given a key value, or to optimize queries of database contents. Four types of indexes are available: b-tree, hash, AVL, and r-tree. The b-tree index maintains key ordering according to commonly accepted collating sequences (e.g. alphabetic order), so index navigation is possible from one key to the previous or next key. A hash index is used to quickly find a key, if it exists. Once found, it is not possible to move to the next higher or lower key value. An AVL tree is very optimized for in-memory type applications. They work well with data that can fit entirely in-memory. R-tree indexes are designed to be used with geospatial data.

Geospatial support

With 14.1, Raima added support for the r-tree index type. This index is an optimized multi-dimensional index/array. It is designed for special access methods like geographical coordinates, rectangles or polygons which are commonly used for geospatial purposes. They can store locations or things like buildings very efficiently and when issued queries to provide information like "Return me every building within 1 mile", it is quickly retrieved and displayed. They can be utilized through both the SQL API and the standard CORE API.

For example, let us assume the following schema is used:

```
create table mytable( ...
 rectangle int32 array[4] key using r_tree not null;
 ... );
```

Then a query would look as follows:

```
select * from mytable where array[1,90,-45,+45] contains rectangle;
```

This would return the rows from mytable which are contained with the box bounded by [1, 90, -45, +45]. Additionally, Raima now provides built in support for data queries using the following shapes: point, line, circle, box and tilted boxes.

Dynamic DDL

SQL DDL allows tables to be created or dropped at any time.

The STUDENTS DDL above consists of three CREATE TABLE statements. Frequently these are defined in a text file and submitted to an SQL compiler all at once. However, SQL allows them to be defined at any time.

An index may be created at any time, although it is recommended that they be defined at the same time as the tables.

RaimaDB Dynamic DDL is fast. Operations that add or drop columns are instantaneous. New columns will have null or default values until otherwise set. Dropped columns remain in the stored rows but are ignored and will be eliminated if the row is modified.

As a simple example, we can add a column to the student table using the rdm-sql utility (discussed more later) as follows:

```
c:\RDM> rdm-sql
Enter ? for list of interface commands.
rdm-sql: alter database students;
rdm-sql: alter table student
rdm-sql> add column student_addr char(50);
rdm-sql: alter table student
rdm-sql> add column student_gpa float;
rdm-sql: commit; rdm-sql:
```

The new columns, student_addr and student_gpa, will appear to exist in subsequent queries immediately, but will have null values.

Note that the database schema is now different than the initial schema definition.

Time Series

With 15.0, Raima has added support for time series data. A C++ API can be generated specifically for any time series data. Time series is a data model for data that is collected periodically over some time. The data will typically be indexed, listed, analyzed, and graphed. Raima also supports Fast Fourier Transforms (FFT) to convert data between time and frequency domains. Time domains are data sampled on discrete timestamps whereas frequency domains are data sampled with respect to frequency over a time interval. Raima has a built in simple FFT algorithm but also contains support for custom or third party FFT algorithm libraries. The generated template class in RaimaDB allows for automatic processing of data prior to insertion into the database.

The Basics of Database Engine Implementation

ACID

RaimaDB is an ACID-compliant DBMS, meaning it maintains the following properties:

Atomicity	Multiple changes to a database are applied atomically, or all-or-nothing, when contained within the same transaction.
Consistency	Data relationships are made to follow rules so they always make sense.
Isolation	When multiple readers or writers are interacting with the database, none will see the partially done changes of another writer.
Durability	Changes that are committed in a transaction are safe. Even if something happens to the program or the computer's power, the updates made during the transaction will exist permanently in the database.

Maintaining the ACID properties is the "hard work" of a DBMS. Application programmers shouldn't handle these issues directly themselves. RaimaDB uses standard methods to implement them, as will be shown below. A key concept when viewing or updating a database is that of a transaction. Atomicity has to do with the grouping of a set of updates as one transaction. Consistency has to do with rules – for example the existence of a key in an index means that the row containing that key column value exists too. Isolation has to do with a community of users never seeing changes done by others
except as completed transactions. Durability has to do with writing to the database in a way that causes the entire group of updates to exist or not exist after a crash and recovery.

Support For Multiple API's

Raima supports the following API's:

- C/C++ with Cursor view
- JDBC 4.2 Type 4 – Raima´s Java API allows for database connection through Java.
- ODBC 3.9 – ODBC standard API database connection

Application Programmer Interfaces

Using the simple schema and data from above, this section will show how each API can be used to obtain the same results.

Please note that proper programming dictates that function return codes are checked after each call. The following examples omit this to allow focus on the actions of each function.

Raima´s core API (C/C++)

RaimaDB's primary API for C/C++ programmers uses the cursor view of data and is commonly called the Core API. It is fully aware of the set relationships between different types of rows but shows set members as the contents of a cursor, as shown in this example.

The header file "students_structs.h" is an artifact of "rdm-compile -c students.sdl", and contains structure definitions and constants used within the code.

```
#include "rdm.h"
#include "rdmapi.h" #include "tfsapi.h"
#include "students_structs.h"
RDM_RETCODE list_class_info(char *classId)
{

  RDM_TFS  tfs; RDM_DB  db;
  RDM_CURSOR cClass = NULL;
  RDM_CURSOR cEnrollment = NULL; RDM_RETCODE rc;
  ENROLLMENT enrollment_rec;
  rdm_tfsSetOptions(tfs, "TFSTYPE=embed;DOCROOT=.");
  rdm_rdmAllocTFS (&tfs); rdm_tfsAllocDatabase (tfs, &db);
  rc = rdm_dbOpen (db, "students", RDM_OPEN_EXCLUSIVE); if (rc != sOKAY) {
    printf("Error %d opening database\n", rc); return rc;

  }
  rdm_dbAllocCursor(db, &cClass); rdm_dbAllocCursor(db, &cEnrollment);
  rc = rdm_dbGetRowsByKeyAtKey(db, COL_CLASS_CLASS_ID, classId, 0, &cClass); if (rc != sOKAY)
    printf("Error %d in rdm_dbGetRowsByKeyAtKey\n", rc); if (rc == sOKAY) {
    rc = rdm_cursorGetMemberRows(cClass,REF_ENROLLMENT_MY_STUDENTS,&cEnrollment); if (rc != sOKAY)
      printf("Error %d in rdm_cursorGetMemberRows\n", rc);

  }
  if (rc == sOKAY) {
      while ((rc = rdm_cursorMoveToNext(cEnrollment)) == sOKAY) { rdm_cursorReadRow(cEnrollment, &enrollment_rec,
        sizeof(ENROLLMENT), NULL); printf("\t%s %s %d, %s\n", enrollment_rec.student_status,
          rdm_dateMonthAbr(enrollment_rec.begin_date),
          rdm_dateDayOfMonth(enrollment_rec.begin_date), enrollment_rec.my_classes);

    }
  }

  rdm_dbClose(db); rdm_dbFree(db); rdm_tfsFree(tfs)
```

Object Oriented C++ API

The C++ interface to RaimaDB is designed to augment the Core API by providing database specific sets of classes implementing higher-level abstractions. These generated interfaces are built to fit your specific database schema using an object orientated approach. The C++ API is designed for ease of use, but by giving full access to both RaimaDB's network and relational functionality, it is very powerful and can be used to create efficient database applications. There are two main interfaces that comprise the RaimaDB C++ API, the Db interface which encapsulates access to a particular database and the Cursor interface which encapsulates access to records within a database. These interfaces contain methods that are common to all databases and records and methods that are specific to a particular schema. By using these interfaces the C++ programmer is able to create applications that safely and efficiently query, insert, update, and delete data stored in a RaimaDB.

The header file "students_api.h" is an artifact of "rdm-compile -x students.sdl", and contains the customized class definitions used within the code.

```cpp
#include <iostream>
#include "students_api.h"

using namespace RDM_CPP; using namespace std;
RDM_RETCODE list_class_info(const char *classId)
{
  Db_students db; Cursor_class cClass; Transaction trans;
  ENROLLMENT enrollmentData; RDM_RETCODE rc;
   Cursor_enrollment cEnrollment; try {
    db = Db_students(TFS::Alloc("TFSTYPE=embed;DOCROOT=.").AllocDatabase());
    db.Open(RDM_OPEN_EXCLUSIVE);
     trans = db.StartRead();

    cClass = db.Get_class_RowsBy_class_id(classId);
    cEnrollment = cClass.Get_enrollment_my_students_MemberRows(); cEnrollment.MoveToFirst();
    while (cEnrollment.GetStatus() == CURSOR_AT_ROW)
    {

      cEnrollment.ReadRow(enrollmentData);     cout << "\t" << enrollmentData.student_status;
      cout << " " << rdm_dateMonthAbr(enrollmentData.begin_date); cout << " " << rdm_dateDayOfMonth(enrollmentData.begin_date); cout
       << " " << enrollmentData.my_classes << endl; cEnrollment.MoveToNext();

    }
    trans.End();
  }

  catch (const rdm_exception& e)
  {
    /* We display the error message in the controller class */ rc = e.GetErrorCode();
    cerr << "Error: " << rc << " " << endl; trans.EndRollback();
  }
return rc;
```

Cursor View

Raima´s cursor view of data is a cursor is opened by specifying a group of rows that is thereafter managed by a cursor handle. Once a cursor is opened, it will have a current position within the group and provide the ability to navigate from that position, or to read/write/delete/update data.

Cursors eliminate a shortcoming of the currency view. Currency allows only one current record for any type. For example, ifthe current owner and member of the set between class and enrollment (called class_students) is set to the class identified as "MATH01" and the first member, there can be no other current owners or members retained elsewhere. If another class_students set instance needs to be scanned, the existing current records must be saved (as database addresses) and then restored.

Cursors, however, will allow any number of cursor instances to be opened, each with its own position. So two cursors can be opened, one based on the owner and members of "MATH01" and another based on "SOFT03", allowing independent navigation within each cursor.

An SQL result set can also be viewed as a cursor. When SQL creates any number of statement handles, it can navigate them independently.

ODBC/SQL API

Access to SQL is done through the ODBC API. This API has two forms. The Microsoft ODBC Driver manager allows 3rd party tools to access the RaimaDB's through RaimaDB's ODBC API. But C/C++ programmers may write programs that call the ODBC API directly and not through the driver manager. This example shows a program calling ODBC functions directly.

```
#include "sqlext.h"
#define SQL_EMPSTR ((SQLCHAR *) "")
const char *mAbbr[] = {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
                                                            ", "Sep",
"Oct", "Nov", "Dec"};
SQLRETURN list_class_info(char *classID)
{


  SQLHENV hEnv; SQLHDBC hDbc; SQLHSTMT hStmt;
  SQLRETURN odbcCode;

  (void) SQLAllocHandle(SQL_HANDLE_ENV, NULL, &hEnv); (void) SQLAllocHandle(SQL_HANDLE_DBC, hEnv, &hDbc);
          (void) SQLConnect(hDbc, SQL_EMPSTR, SQL_NTS, SQL_EMPSTR, SQL_NTS, SQL_EMPSTR, SQL_NTS);
  (void) SQLAllocHandle(SQL_HANDLE_STMT, hDbc, &hStmt);

  odbcCode = SQLExecDirect(hStmt, (SQLCHAR *) "OPEN DATABASE students", SQL_NTS); if (SQL_SUCCEEDED(odbcCode)) {
    char strStmt[100];
    char student_status[10]; char my_classes[36]; DATE_STRUCT begin_date;
    sprintf(strStmt, "SELECT student_status, begin_date, my_classes "
      "FROM enrollment where my_students = '%s'", classID); (void) SQLExecDirect(hStmt, strStmt, SQL_NTS);

      while ((odbcCode = SQLFetch(hStmt)) == SQL_SUCCESS) { (void) SQLGetData(hStmt, 1, SQL_C_CHAR, student_status,
sizeof(student_status), NULL);
 (void) SQLGetData(hStmt, 2, SQL_C_TYPE_DATE, &begin_date,
TECHNICAL WHITEPAPER
 Page 13 of 32
 sizeof(begin_date), NULL);
odbcCode = SQLGetData(hStmt, 3, SQL_C_CHAR, my_classes,

sizeof(my_classes), NULL);
if (SQL_SUCCEEDED(odbcCode))
printf("\t%s %s %d, %s\n", student_status, mAbbr[begin_date.month-1],
begin_date.day, my_classes);
}
}
(void) SQLFreeHandle(SQL_HANDLE_STMT, hStmt);
(void) SQLDisconnect(hDbc);
(void) SQLFreeHandle(SQL_HANDLE_DBC, hDbc);
(void) SQLFreeHandle(SQL_HANDLE_ENV, hEnv);
return odbcCode; }
```

JDBC API

RaimaDB supports the JDBC standard SQL access to databases. Access to the RaimaDB functions from Java is facilitated in two ways. The single-process method uses JNI to call the local C procedures. The other method is pure Java, using a Type 4 driver to a separate server process identified in the DriverManager.getConnection() method. In the example below, the local, JNI form is used.

```
import java.sql.*;
import java.text.SimpleDateFormat;
...
  private static void list_class_info(String classID) throws SQLException { Connection conn = null;
  Statement stmt = null; ResultSet rs = null;
    SimpleDateFormat dateFormat = new SimpleDateFormat("MMM d"); try {
    /* Create the connection with a simple connection string */
    conn = DriverManager.getConnection("jdbc:raima:rdm://local"); conn.setAutoCommit(true);


    CreateDatabase(conn); LoadDatabase(conn);


    /* Open the connection to the database */ stmt = conn.createStatement();

              rs = stmt.executeQuery("SELECT student_status, begin_date, my_classes " + "FROM enrollment where my_students = '" +
                        classID + "'");
    while (rs.next() != false)
    {


        System.out.println("\t" + rs.getString("student_status") + " " + dateFormat.format(rs.getDate(2)) + ", " + rs.getString("my_classes"));


    } }
  catch (SQLException sqle) {    sqle.printStackTrace();
    } finally { rs.close();
    stmt.close();
    conn.close();

  }
}
```

# Raima's Transactional File Server Concept

The Transactional File Server (TFS) specializes in the serving and managing of files on a given medium. The TFS API is a set of functions called by the runtime library to manage the sharing of databases among one or more runtime library instances. In a normal multi-user configuration (see below for more about configurations), the TFS functions are wrapped into a server process called rdm-tfs that listens for runtime connections and serves the connections with remote procedure calls. This listener my be started in applications built with RaimaDB, but this is a more advanced topic. To connect to a particular rdm-tfs process, the runtime library needs to know the domain

name of the computer on which rdm-tfs is running, and the port on which it is listening, for example, "tfs.raima.com:21553". Standard TCP/IP can be used to make the connection, whether the runtime library and rdm-tfs are on the same computer or different computers (when on the same computer, optimizations are made, and a sharedmemory protocol is available by default).

Figure 5: Runtime Library/TFS Configuration

In Figure 5, it shows that one runtime library may have connections to multiple rdm-tfs processes, and one rdm-tfs may be used by multiple runtime libraries. To the applications using the runtime libraries, and the rdm-tfs processes, the locations of the other processes are invisible, so all processes may be on one computer, or all may be on different computers. This provides opportunities for true distributed processing. An rdm-tfs should be considered a "database
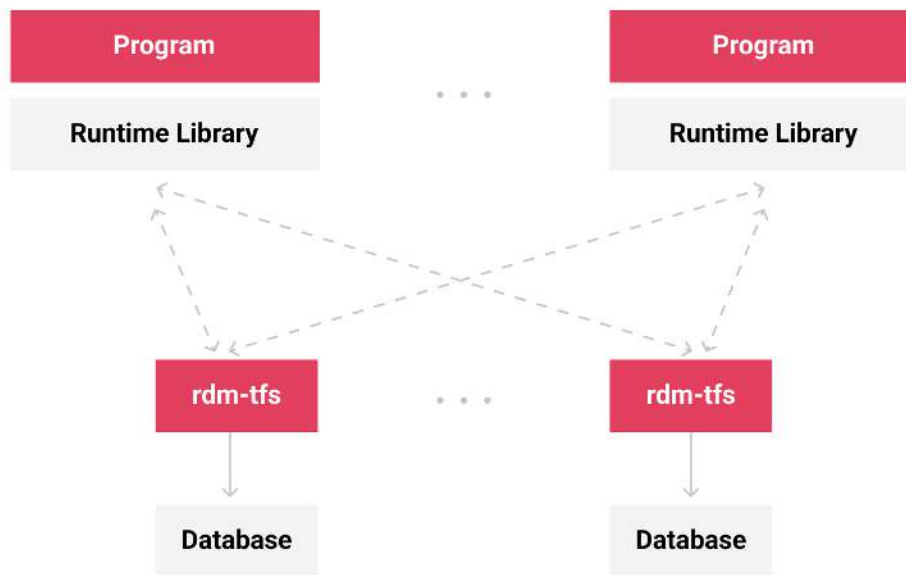
controller" in much the same way as a disk is managed by a disk controller. A TFS is initialized with a root directory (elsewhere called the "document root") in which are stored all files managed by the TFS. If one computer has multiple disk controllers, it recommended that one rdm-tfs is assigned to each controller. This facilitates parallelism on one computer, especially when multiple CPU cores are also present.

A complete system may have multiple rdm-tfs running on one computer, and multiple computers networked together. Each rdm-tfs will be able to run in parallel with the others, allowing the performance to scale accordingly.

TFS Configurations

There is a clear delineation between the runtime library and the TFS. The runtime library understands and manipulates databases, using catalogs that describe the database structure and keeping a local cache of database objects that have been read, created or modified at the request of the application. All of the row and column interpretation is done within the runtime library. The runtime library needs to have a handle to one or more TFSs to do any work.

The TFS is responsible for safely storing the retrieving objects, regardless of the contents of those objects. To the TFS, the objects are opaque sequences of bytes with a given length. When asked to store objects, it does so in a way that is transactionally safe and recoverable. When asked to retrieve objects, it returns the same opaque sequence of bytes. It is like a key/value store, but very fast and transactionally safe. The TFS owns and is co-located with the database files.

This delineation allows for a number of configurations between the runtime library and the TFS.

Physically, the TFS functions may be called directly from within the runtime library in order to operate on database files that are visible to the application process. The TFS functions may also be wrapped into a server process and called through a Remote Procedure Call (RPC) mechanism on the behalf of runtime libraries that are executing within other processes. This distinction between local and remote access to the TFS functions is called "embed" or "remote", and the application programmer may choose between them.

The "embed" configuration is used when an application process with one or more threads uses a database that is not shared with other processes. The "remote" configuration is used when the TFS functions are running within a server process.

By default, the "hybrid" configuration allows a runtime library to select an embedded or remote TFS at run time. It is possible to restrict an application to only embedded or only remote in order to save code space. The restriction is accomplished by using specific RaimaDB libraries when linking the application. However, the default library permits the location of the TFS to be identified when a database is opened.

The following code fragments demonstrate some of the options and assumes that the application executable has been linked with the default TFS library.

Open a database located from the current directory:

```
RDM_DB   db;

/* Defaults for TFS options: hybrid, document root is current directory */ rdm_rdmAllocTFS ("", &tfs);
rdm_tfsAllocDatabase (tfs, &db);

/* open the database named 'students', stored in a subdirectory of the
 * current directory, named 'students'
 */
rc = rdm_dbOpen (db, "students", RDM_OPEN_EXCLUSIVE);
```

Open a database located on a remote computer:

```
/* Defaults for TFS options: hybrid, but will open "remote" database */
rdm_rdmAllocTFS ("", &tfs); rdm_tfsAllocDatabase (tfs, &db);


/* open the database named 'students', managed by a TFS process running on a
 * remote computer with a visible domain name 'tfs.raima.com' on port 21553
 */
rc = rdm_dbOpen (db, "tfs-tcp://tfs.raima.com:21553/students", RDM_OPEN_SHARED);
```

Open a database in the directory "C:\RaimaDB\students" using document root "C:\RaimaDB":

```
/* hybrid TFS, document root is current directory */
rdm_tfsSetOptions(tfs, "DOCROOT= c:\\RaimaDB"); rdm_rdmAllocTFS (&tfs);
rdm_tfsAllocDatabase (tfs, &db);
/* open the database named 'students', found in a directory named
 * c:\RaimaDB\students'
 */
rc = rdm_dbOpen (db, "students", RDM_OPEN_EXCLUSIVE);
```

The following rdm_dbOpen() attempt will fail because the TFS type was specified as "embed" but the URI specifies a remote database:

```
/* hybrid TFS, document root is current directory */
rdm_tfsSetOptions(tfs, "TFSTYPE=embed;DOCROOT= c:\\RaimaDB"); rdm_rdmAllocTFS (&tfs);
rdm_tfsAllocDatabase (tfs, &db);

/* open the database named 'students', stored in a subdirectory of the
 * current directory, named 'students'
 */
rc = rdm_dbOpen (db, "students", RDM_OPEN_EXCLUSIVE);
```

Open two databases. One local, one remote. Both databases are named "students". The local database is found in the directory "C:\RaimaDB\students" using document root "C:\RaimaDB". The remote database is opened through the URI "tfstcp:// tfs.raima.com/students" where the TFS is running as a server (probably rdm-tfs) with a document root on that computer

(note that this application doesn't need to know the remote document root). Note also that the default port for a TFS is 21553. If the TFS is started up with a port other than 21553, it is necessary to include that port in the URI:

```
/* hybrid TFS, document root is current directory */
rdm_tfsSetOptions(tfs, "TFSTYPE=embed;DOCROOT= c:\\RaimaDB"); rdm_rdmAllocTFS (&tfs);
rdm_tfsAllocDatabase (tfs, &db);

/* open the database named 'students', stored in a subdirectory of the
 * current directory, named 'students'
 */
rc = rdm_dbOpen (db, "students", RDM_OPEN_EXCLUSIVE);
```

Data Storage Engine

Under a directory we call the "document root", one or more databases may be stored, each within a subdirectory named after the database with the suffix ".rdm". For example, a database named "students" will be a subdirectory of a document root, with the name "students.rdm". In this subdirectory will a file named p00000000.pack. The contents of this file, is the database catalog (in JSON format) and the database objects in a compressed "pack" file. Database objects can be rows, b-tree nodes, or other stored objects.

When an object stored in the pack file is modified, its location is changed. The old object will remain in place until it is no longer needed (as may be the case with MVCC operations), but will eventually be permanently abandoned. Then its space will be cleaned up through a vacuuming process. RaimaDB will, at any given time, contain a database which will have a certain percentage of empty space.

An important issue with durable storage like disk and SD RAM is that an operating system will almost always maintain a filesystem cache of the file contents for performance reasons. If file updates are written into the file system, they first exist in the cache only. If the computer stops functioning before writing the cache contents to the permanent media, not only can the updates be lost, but the files may be left in an inconsistent state.

To safeguard against this, RaimaDB asks the operating system to "sync" a file at key moments, ensuring that the data is safe no matter when a computer may fail. The "sync" operation (synchronize to disk) will not return control to the program until the file contents exist on the permanent media. Any database system that guarantees the safety of data must have sync points in its transaction handling.

Database Portability

RaimaDB's file format was specifically designed to be portable. Portable database files has two effects:

- Databases may be created in one environment and freely copied to other environments for use there. For example, a large database of static information may be created in a central office and distributed for use at several branch offices running different types of computers.
- Databases may be concurrently accessed by applications running on computers with different operating systems or CPU architectures.

To accomplish this, every piece of data is given a location and length in the files and is always reconstructed in memory for use by the local computer. This may mean that it is decompressed or decrypted. It may also mean that an integer value is translated into the local integer format. In fact, any integer stored in the database is translated into Raima's proprietary 'varint' (variable integer) format, using only as many bytes as are required to hold the value. In other words, a small value like 10 will be stored in 1 byte, but 10,158 will require 2 bytes. Integers with values requiring 17 bytes may be stored in the RaimaDB pack. C structures are never used to store or retrieve groups of values, because structure alignments vary by C compiler and CPU.

RaimaDB Enterprise Package

SQL Standard

In the enterprise package, Raima supports the full INCITS/ISO/IEC 9075-2:2011 [2012] SQL standard. Raima complies to the rules and definitions stated within that standard for the SQL support.

SQL Array

RaimaDB allows columns of a table to be defined as one-dimensional bounded arrays. Arrays of any built-in base type can be created but arrays of domains are not yet supported.

The ARRAY type allows other types to have multiple values within a single column of a table. For example, R-tree coordinates for a two-dimensional map would be stored in a column with an array dimension of 4 as shown below:

```
CREATE TABLE pointsofinterest
(
location DOUBLE ARRAY[4] KEY USING R_TREE NOT NULL,
city CHAR(29) DEFAULT "" KEY NOT NULL, state CHAR(2) DEFAULT "" NOT NULL,
zip INT32 DEFAULT 0 KEY NOT NULL,
tz INT32 DEFAULT 0 NOT NULL );
```

SQL Unions

RaimaDB supports the SQL UNION operator. This allows for the user to combine the result set of two or more SQL SELECT statements where each SELECT statement has the same number of columns of a similar data type and same ordering of columns. The syntax is as follows:

```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2
```

SQL Inner and Outer joins

Raima supports inner, left outer, right outer and full outer joins. Joins allow for the user to select rows that either have matching values in both tables or partial matches in either table. An inner join will only return rows that have matching values in both of the tables specified. A left outer join will return all rows in the first table along with matching rows in the second table. A right outer join will return all the rows in the second table along with matching rows in the first table. The result is a NULL column if there is no match. A full outer join will return all rows in both tables with NULL where there are no matches.

SQL Scrollable cursors

Raima now supports SQL scrollable cursors. A user can retrieve a data set with a SQL query and navigate in real time that data set forwards or backwards with the result set.

SQL Triggers

A SQL trigger is a procedure associated with a table that is executed whenever that table is modified by an insert, update or delete statement or alternatively by a call to the RaimaDB Core Cursor API function that inserts, updates or deletes a row from a database. Triggers are specified using SQL and conform to the SQL Standard.

The syntax for a create trigger statement is shown below:

```
create trigger students.validate_enrollment before insert on
enrollment referencing new row as new_enrollment for each row begin atomic
 if new_enrollment.current_grade not between 0.0 and 4.0 then
  signal sqlcode eCHECK set message_text = "Current Grade must be >= 0.0 and <= 4.0";
 else
set new_enrollment.class = ucase(new_enrollment.class); end if; end;
```

This trigger validates that the inserted student's grade is between a 0.0 and a 4.0. If the value is not within that range, then an error Check clause violation is returned. If a valid insert is made, then the class value is converted to uppercase in the case that it is not.

SQL PL

The RaimaDB SQL PL is a programming language for use in RaimaDB SQL stored routines (procedure or function), based on the computationally complete ANSI/ISO SQL Persistent Stored Modules specification. The language is block structured with the ability to declare variables that conform to the usual scoping rules with an assignment statement so that values can be assigned to them. Control flow constructs like if-elseif-else, case statements and a variety of loop control constructs including while, repeat-until, and for loop statements are available.

Also provided is the ability to declare cursors allowing rows from a select statement to be fetched into locally declared variables allowing the result column values to be checked and manipulated within the stored routine. Exception handling allows handlers to be coded for specific or classes of errors or statuses returned from execution of an SQL statement. In addition, it is also possible to define a user condition and exception handler and for the program to signal its own, special-purpose exceptions.

Stored procedures are executed using the SQL call statement.

A procedure named add_enrollment is shown below. This could be used to simplify the process of adding students, classes, and the class enrollment to the database. create procedure add_enrollment(cl_id

```
create procedure add_enrollment(cl_id char, cl_name char, st_name char)
modifies sql data begin atomic
    declare id, name char(35);  declare cl_cur cursor for select class_id from class where class_id = cl_id;
  declare st_cur cursor for
    select student_name from student where student_name = st_name; open cl_cur; fetch cl_cur into id;
if not found then
    insert into class values cl_id, cl_name; end if; close cl_cur;

  open st_cur;
  fetch st_cur into name; if not found then
    insert into student values st_name; end if; close st_cur;
  insert into enrollment(begin_date, student_status, my_students, my_classes)
values current_date(), "enrolled", cl_id, st_name; end;
```

This procedure uses two variables, named id and name, which are used to store results from two cursors, cl_cur and st_cur. The cursors select data from the existing tables to determine if the student or class already exists. For those that don't yet exist, a new row is inserted. A new enrollment row is created with default values for the begin_date (today) and student_status ("enrolled") which can be updated later.

The following sequence of SQL statements will populate the database to contain the rows shown in Figure 4 above.

```
call add_enrollment("MATH01", "Intro to Algebra", "Joe");
update enrollment
  set end_date="2015-12-20", student_status="active", current_grade=3.1 where my_students="MATH01" and my_classes="Joe"; call add_enrollment("SOFT03", "Advanced Java", "Joe");
update enrollment
  set begin_date="2016-01-04",  end_date="2015-03-31"  where my_students="SOFT03" and my_classes="Joe"; call add_enrollment("MATH01", "Intro to Algebra", "Sarah");
update enrollment
  set end_date="2015-12-20", student_status="active", current_grade=3.2 where my_students="MATH01" and my_classes="Sarah"; call add_enrollment("HIST02", "World History", "Sarah");
update enrollment
  set begin_date="2015-04-01", end_date="2015-06-16", student_status="passed", current_grade=3.5 where my_students="HIST02" and my_classes="Sarah"; call add_enrollment("SOFT03", "Advanced Java", "Henry"); update enrollment
  set end_date="2015-12-20", student_status="active", current_grade=2.6 where my_students="SOFT03" and my_classes="Henry";
```

Note that this procedure has chosen to use some default values (current date for begin_date, and "enrolled" for student_status). These values could also have been provided as procedure parameters. Instead, the SQL statements will update the non-default values following the procedure.

REST API

With RaimaDB 15.0, Raima introduced an easy to use REST API interface. With this interface, the developer can access any

Raima data through the use of HTTP Create, Retrieve, Update and Delete (CRUD) operations. Two types of interfaces are available to the users: an administrative interface and a database one.

Rdmadmin

The rdmadmin interface allows the user to retrieve the following information about the RaimaDB subsystem:

- Get a list of databases available to the TFS
- Get information about the TFS memory usage
- Get information about the TFS disk usage
- Get the TFS version information
- Get TFS configuration options
- Get a list of database users

It is accessible with the following URL syntax:

```
http://hostname:port/rdmadmin/?list=request[options]
```

where request is any of the following options:

- databases
- disk
- memory
- options
- user
- version

Database

The database interface will allow the user to Create, Retrieve, Update and Delete (CRUD) data in a specific RaimaDB in JSON format. The supported actions are:

- Retrieve a list of the tables in a database
- Retrieve a list of the columns for a table in a database
- Retrieve a list of the keys for a table in a database
- Retrieve a list of the reference for a table in a database
- Retrieve a list of rows in a table ordered by rowid
- Retrieve a list of rows in a table ordered by a specified key

- Insert a new row into a table
- Update a row in a table
- Delete a row in a table

This is done with the following URL syntax:

```
http://hostname:port/rdm/dbname/?list=request[options]
```

where request is any of the following options:

- tables
- info
- rows

Insertion of data is done through an HTTP POST command with the data in a JSON format. An example syntax is as follows:

```
curl -X "POST" "http://localhost:21553/rdm/bicycle_db/MANUFACTURER" -i -H 'Content-Type: application/json' -d $'{ "NAME":
"Trek Bicycle Corporation", "state": "WI"}'
```

Rest Admin GUI

RaimaDB includes a bundled webapp to allow for quick and easy administration of the database.

Figure 6: Administrative GUI

This webapp will allow the end user to quickly see engine status such as: memory usage, database size, users connected and engine configuration. It also includes a database browser for quick browsing of the contents of a database or easy manipulation/deletion of individual row data.

Raima Performance Features

Raima's Snapshot Mode

RaimaDB allow an application to perform repeatable read operations on a database without blocking other readers and writers through the use of database snapshots. The user would start a transaction with the core API call:

```
rdm_dbStartSnapshot();
```

This will generate a static view of the database at the time the snapshot was created. Any changes made to the database will not be visible until the snapshot is released but this will allow for writes to continue as a reader can read from the snapshot. Once the user is done with the snapshot, it can be released with a call to:

```
rdm_dbEnd(..);
```

In-memory Database Operation

Figure 7 shows a very simplified view of the in-memory database operation. Loading a database is optional, as is persisting the database.
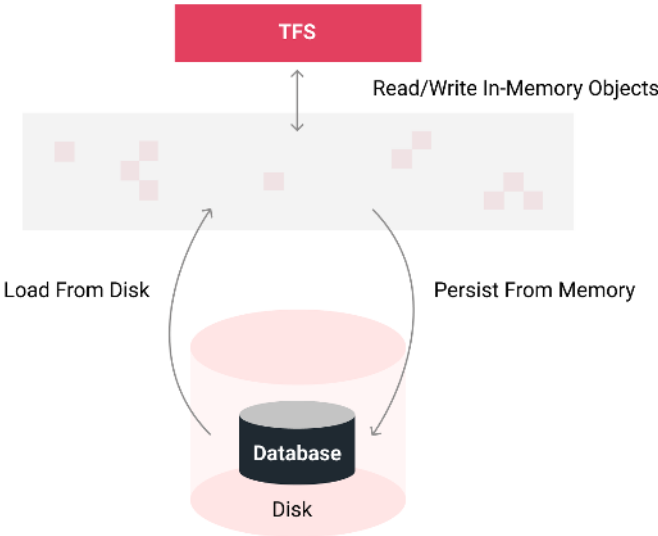

Figure 7: In-memory Database Persistence

Earlier versions or RaimaDB implemented in-memory databases by emulating a file system in memory. This allowed the runtime library functionality to remain agnostic of the storage media, but didn't take advantage of the efficiencies that can be gained by organizing data in such a way that facilitates fast in-memory lookups and modifications. When data is created or loaded into the RaimaDB in-memory format, it is in an "expanded" form, very different from the ondisk form that is flattened and compressed. The runtime library maintains a cache of data that it has created or read from the TFS. The in-memory format is very similar to the runtime cache, and the runtime functions recognize the structures from either their own cache or the in-memory database. For this reason, when an application links directly to the TFS functions (the "embed" configuration), the runtime library may directly refer to objects in the in memory database. Figure 8 shows an expansion of Figure 7 where the database is in-memory.
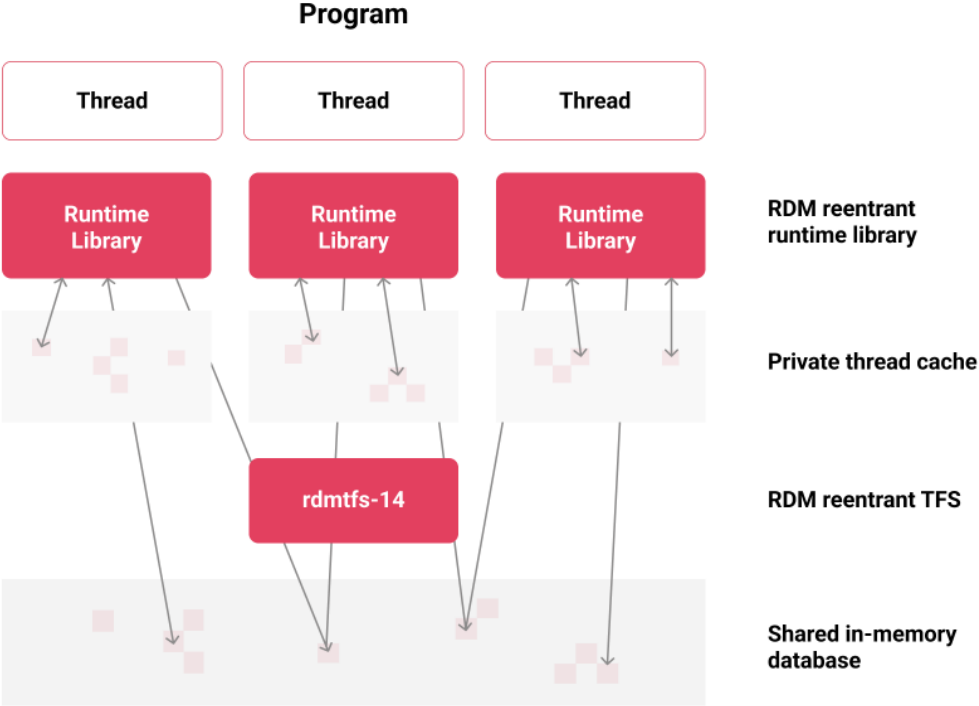

Figure 8: Single-Process, Multiple Threads Configuration

Figure 8 depicts multiple runtime libraries and a TFS modules, but in fact there is a single instance of all the reentrant code that is shared among all the threads. Each thread has its own private cache and a private thread cache of the database objects. However, in this configuration - Single Process, Multiple Thread - the in-memory contents are referenced directly from all of the threads. Direct reference to a database object is possible when the object is not modified. Modified objects are stored strictly in the private caches until they are committed.

This optimization means that threads performing queries will not need to read objects into their private caches. In a multi-user configuration ("remote"), this optimization is not possible because the memory storing the database is contained in another process, or even another computer. For this reason, Raima recommends this Single Process, Multiple Thread configuration for high-performance requirements

Shared Memory Transport

Memory is used in a different form for optimization of inter-process communication. When the "remote" configuration is used, and when the connecting applications are running on the same computer as rdm-tfs, the communications between the processes is dramatically accelerated by using shared memory rather than TCP/IP sessions. The shared memory transport is used by default when the two sides of the connection are running on the same computer. One rdm-tfs process may be connected to both local and remote runtimes, and it will use shared memory and TCP/IP concurrently in order to communicate most efficiently with its clients.

Utility Programs

The RaimaDB SDK comes with a number of command-line utilities that allow the programmer to define, configure and manipulate databases. This section briefly discusses the utilities that will be used most often. The easiest way to use any of these utilities is to put the RaimaDB SDK bin directory in your path.

rdm-sql

rdm-sql is an SQL utility with support for interactive and non-interactive use. When used interactively, rdm-sql allows you to execute SQL statements and view result sets. When used non-interactively (called "batch mode"), it processes the SQL statements as well as its interactive commands (described later) stored in an input file or a redirection of standard input.

All of the functionality of rdm-sql is available to the C/C++ programmer through the ODBC API.

To use it interactively, simply issue the command "rdm-sql" from a command prompt. It will have access to all databases stored in a "document root" which by default is the current directory.

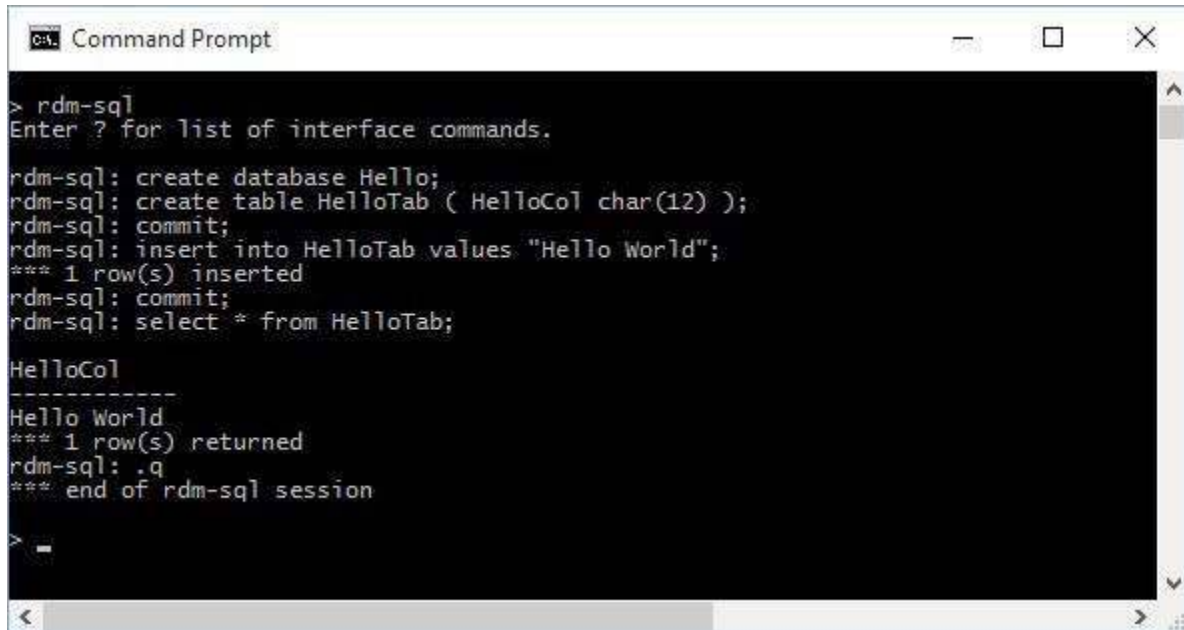To use it in batch mode, provide an input file on the command line:

```
rdm-sql input_file
```

Command-line help is provided with the -h option ("rdm-sql -h"). Once running the utility, interface commands help is available

with a '?'.

Here is a simple example:

```
Command Prompt                                    —   □   ✕

> rdm-sql
Enter ? for list of interface commands.

rdm-sql: create database Hello;
rdm-sql: create table HelloTab ( HelloCol char(12) );
rdm-sql: commit;
rdm-sql: insert into HelloTab values "Hello World";
*** 1 row(s) inserted
rdm-sql: commit;
rdm-sql: select * from HelloTab;

HelloCol
------------
Hello World
*** 1 row(s) returned
rdm-sql: .q
*** end of rdm-sql session

> _
```
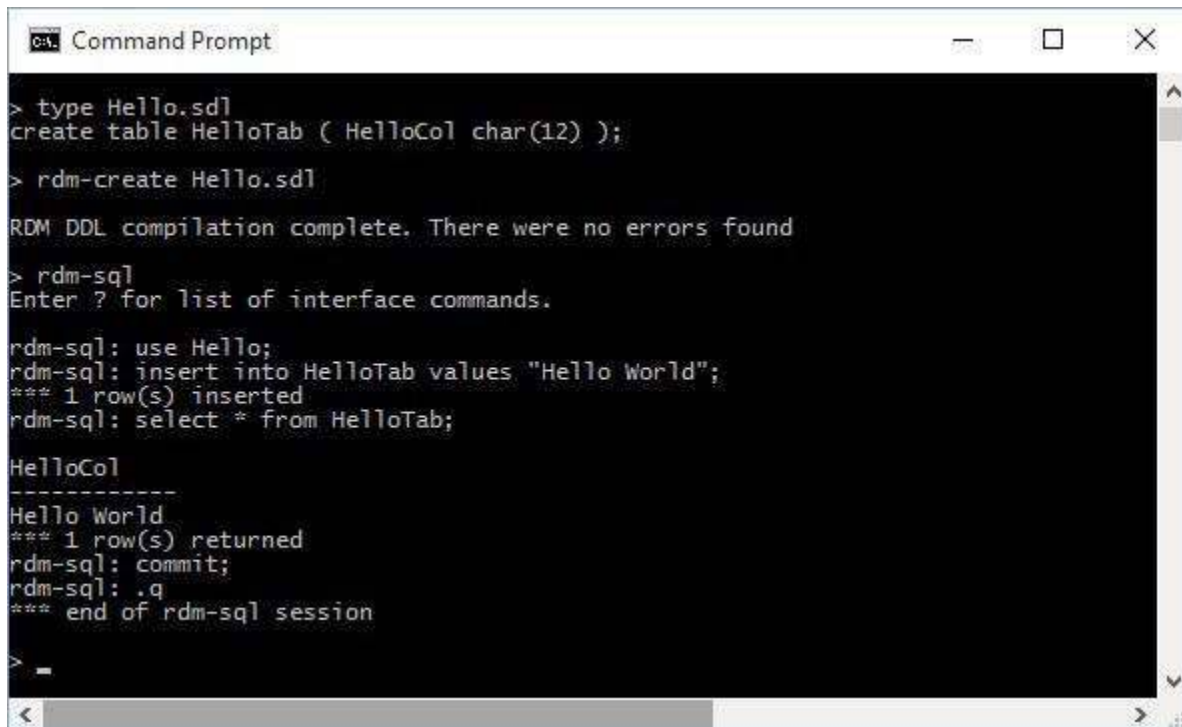
Commonly used interactive commands include ".r filename" to read a set of commands from a file, and ".q" to exit the utility. rdm-create rdm-create is a command-line utility that creates an RaimaDB database from a database definition file (.sdl).

The database will not be populated with data. The database will be created in the "document root" which is current directory by default.

The following example creates the same "Hello" database as above. Note that the database name is taken from the name of the .sdl file. No "create database" statement is issued or allowed.

```
Command Prompt                                    —   □   ×

> type Hello.sdl
create table HelloTab ( HelloCol char(12) );

> rdm-create Hello.sdl

RDM DDL compilation complete. There were no errors found

> rdm-sql
Enter ? for list of interface commands.

rdm-sql: use Hello;
rdm-sql: insert into HelloTab values "Hello World";
*** 1 row(s) inserted
rdm-sql: select * from HelloTab;

HelloCol
------------
Hello World
*** 1 row(s) returned
rdm-sql: commit;
rdm-sql: .q
*** end of rdm-sql session

> _
```

rdm-compile rdm-compile is a command-line utility that compiles a database definition into a platform-agnostic (JSON) catalog file. It can optionally generate C and C++ source and header files that can be included in an application.

Since the catalog file is created by other utilities such as rdm-sql or rdm-create, it is not normally necessary to use this one.

However with projects coded in C/C++, this utility creates source files not otherwise available from the SDK.

A common scenario when developing C/C++ applications is to define the DDL, compile it, then use the "dbname_structs.h" file in the C/C++ files for compilation. Note that this form of application programming will not adjust automatically for dynamic DDL changes, because the row structures defined in the "dbname_structs.h" file is fixed until the source file is recompiled.
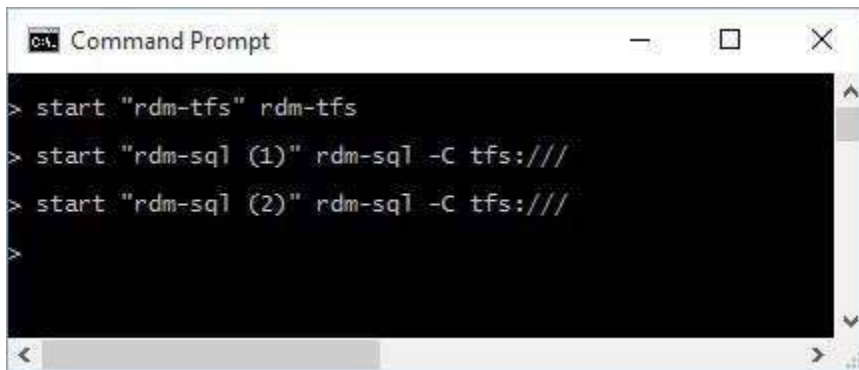
rdm-tfs

The default TFS configuration (see TFS Configurations above) is "hybrid", where databases may be opened locally and are accessed by TFS functions run within the same process, or they may be opened remotely and are accessed by remote TFS functions. This default configuration supports multiple threads in one process.

Remote TFS functions reside in the rdm-tfs process, and they are called through a RPC (remote procedure call) mechanism built into the RaimaDB runtime library.

It is possible to connect to an rdm-tfs on a computer other than the one executing the application. The rdm-tfs process is located by forming a URI containing the domain name of the computer it is running on. By default, the RaimaDB runtime locates a rdm-tfs process on the local machine. The following example shows local machine access.

Step one is to spawn the rdm-tfs process, then two rdm-sql processes. On Windows the 'start' command is used:
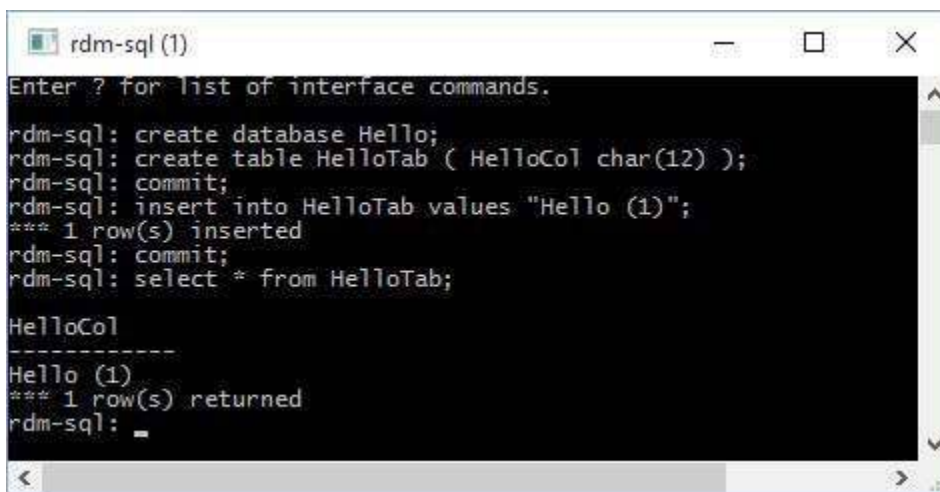
```
Command Prompt                           —    □    X

> start "rdm-tfs" rdm-tfs
> start "rdm-sql (1)" rdm-sql -C tfs:///
> start "rdm-sql (2)" rdm-sql -C tfs:///
>
```

Note that the -C argument (or --connect for better description), "tfs:///" assumes default values to identify the local rdm-tfs process. Fully spelled out, it would be "tfs://localhost:21553/.

But the "-C tfs:///" command line argument tells rdm-sql to use the separate rdm-tfs process instead of the embedded one.

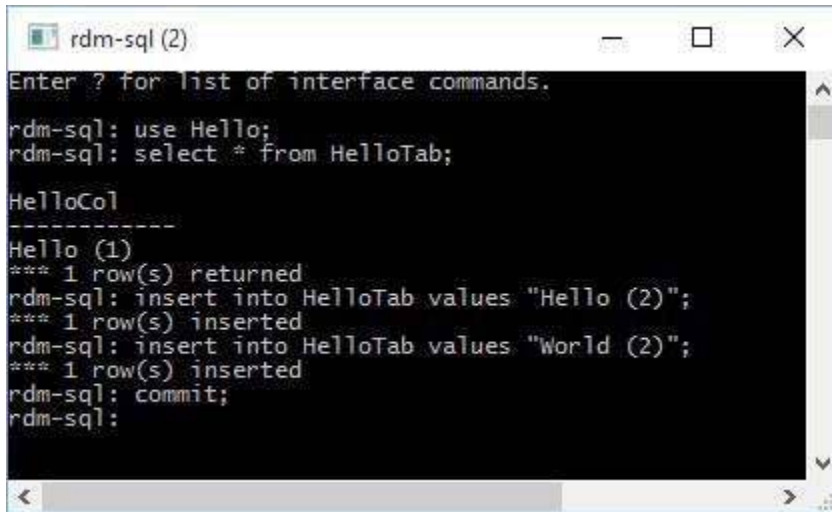Next we will define a simple database just as we did above:

```
rdm-sql (1)                              —    □    X

Enter ? for list of interface commands.

rdm-sql: create database Hello;
rdm-sql: create table HelloTab ( HelloCol char(12) );
rdm-sql: commit;
rdm-sql: insert into HelloTab values "Hello (1)";
*** 1 row(s) inserted
rdm-sql: commit;
rdm-sql: select * from HelloTab;

HelloCol
------------
Hello (1)
*** 1 row(s) returned
rdm-sql:
```
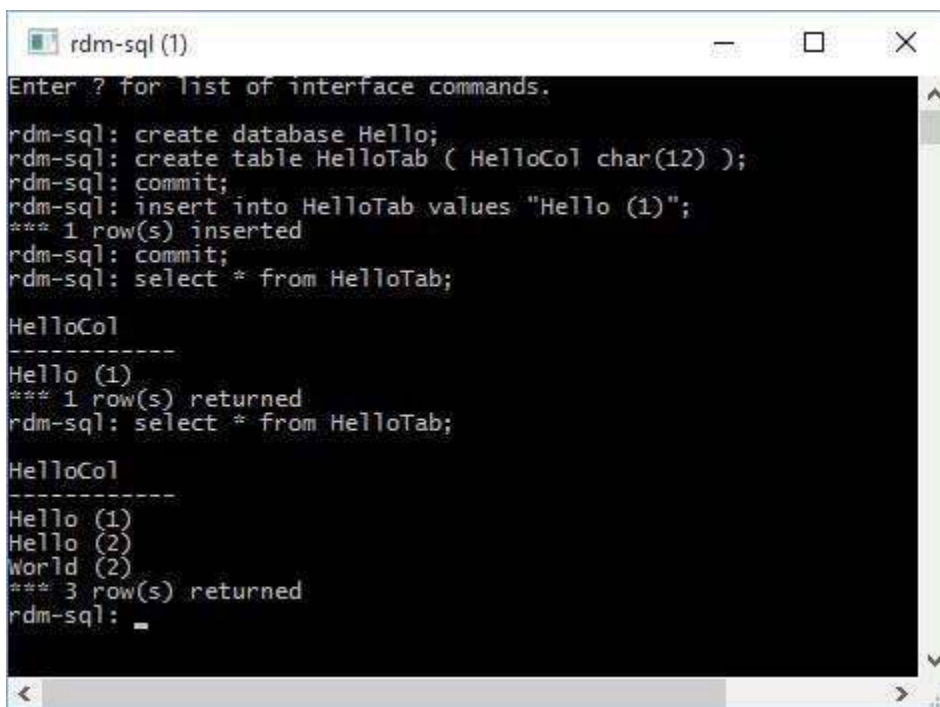
From the second rdm-sql, we can open the database and see the data. We can also add data.

```
rdm-sql (2)                              —   □   ✕
Enter ? for list of interface commands.

rdm-sql: use Hello;
rdm-sql: select * from HelloTab;

HelloCol
------------
Hello (1)
*** 1 row(s) returned
rdm-sql: insert into HelloTab values "Hello (2)";
*** 1 row(s) inserted
rdm-sql: insert into HelloTab values "World (2)";
*** 1 row(s) inserted
rdm-sql: commit;
rdm-sql:
```

Now the first rdm-sql is able to see the new data.

```
rdm-sql (1)                              —   □   ✕
Enter ? for list of interface commands.

rdm-sql: create database Hello;
rdm-sql: create table HelloTab ( HelloCol char(12) );
rdm-sql: commit;
rdm-sql: insert into HelloTab values "Hello (1)";
*** 1 row(s) inserted
rdm-sql: commit;
rdm-sql: select * from HelloTab;

HelloCol
------------
Hello (1)
*** 1 row(s) returned
rdm-sql: select * from HelloTab;

HelloCol
------------
Hello (1)
Hello (2)
World (2)
*** 3 row(s) returned
rdm-sql: _
```

From here, it is easy to experiment with multi-user behavior. For example, one rdm-sql can insert rows, but if the second rdmsql queries the database before the rows are committed, they will not be shown. Immediately after they are committed, they become visible.

Security Through Encryption

The RaimaDB objects are stored in the cache of the runtime library in an unencrypted form for use by the database functions. However, if the database contains sensitive information, RaimaDB allows the objects to be encrypted when they are written from the cache to the files. RaimaDB supports the Rijndael/AES algorithm with 128, 192 or 256 bit keys. A simple default encryption uses XOR operations.

When a runtime requests a database object, it will receive it in the encrypted form and must have the key to decrypt it when it places it into the local cache. During the commit of a transaction, objects sent over the network will be encrypted, and if the TFS is storing those objects, they remain encrypted on the disk file.

Third Party Replication

RaimaDB now supports replication both to and from a RaimaDB system by use of the third party replication system SymmetricDS (https://www.symmetricds.org/ and http://www.jumpmind.com/). Symmetric is a JAVA based asynchronous replication based system with support for multiple database management systems. The supported databases range from Oracle, Microsoft SQL Server, Amazon Web Services, Microsoft Azure, MySQL, SQLite, IBM DB2, etc and now RaimaDB. RaimaDB will support data filtering and transformation during the process as well.

Interoperability

Standard interfaces allow the outside world (that is, tools that can interface to a variety of data sources) to view and manipulate data in a RaimaDB. While most application systems based on RaimaDB are "closed," there are many advantages to using languages (Java, etc.) and tools (Excel, Crystal Reports, etc.) to access the data used by the system. Raima has chosen ODBC and JDBC as standard interfaces. ODBC is also implemented as a C API, meaning that C/C++ programmers can write programs that access the database through ODBC functions. This API may be used within any environment. On Windows, the ODBC driver has been provided for access from third party tools. JDBC permits connection to a RaimaDB using the standard methods.

Near Future

In the near future Raima will release RaimaDB with the added functionality of an optimized OCC locking system and further SQL improvements. Raima will also announce replication from the Raima to Raima directly from the edge.

Conclusion

The goal of this paper is to provide a technical description of the RaimaDB 16.0 product, sufficient for an evaluation and decision making process. Of course, there are many more details available for the evaluation process, but they may take hours or days longer to obtain. The best evaluation is through downloading and running the full product.

As a highly technical product with many shapes and sizes, many of Raima's customers benefit from a consultative analysis of their database design or coding process. This can result in significant optimizations and be instructive in the use of Raima's products.

Quick Start

This section assumes that you have downloaded and installed RaimaDB for your Windows, Linux or Unix system. While other operating system platforms are available for test, it is recommended that you become familiar with the product on one of these desktop systems.

This section will use Windows conventions in the examples, but the translation to Linux or Unix is straightforward. Another assumption is that you have installed RaimaDB in a home directory called C:\Raima\RDM\16.0. The following steps will allow to you create a simple database with SQL which will allow you to follow the remaining examples in this paper.

Environmental setup

- Open a command prompt.
- If the RaimaDB installation home directory is C:\Raima\RDM\16.0, include the \bin directory in your path:

```
C:\> PATH=C:\Raima\RDM\16.0\bin;%PATH%
```

- Create or use a directory that will contain RaimaDB's, for example:

```
C:\> mkdir \RaimaDB C:\> cd
\RaimaDB
```

- Make sure the rdm-sql utility is able to run. Check the path if this fails:

```
C:\RaimaDB> rdm-sql --help
Raima Database Manager Interactive SQL Utility (etc.)
```

Create the database

- Create an SQL Definition Language file containing the following contents. Name it students.sdl.

```
create table class (
class_id char(6) primary key,
class_name char(29)
);
create table student (
student_name char(35) primary key
);
create table enrollment ( begin_date date,
end_date date, student_status char(9),
current_grade float,
my_students char(6) references class, my_classes char(35)
references student );
```

- Run the rdm-sql utility from the example directory. This directory will be referred to as the document root. The ".r" command reads the file contents and submits them as if they had been typed interactively. You may optionally enter the SQL DDL by hand:

```
C:\RaimaDB> rdm-sql
Enter ? for list of interface command.
rdm-sql: create database students; rdm-sql: .r students.sdl
students.sdl(1): create table class (
students.sdl(2): class_id char(6) primary key, students.sdl(3):
class_name char(29) students.sdl(4): ); students.sdl(5): students.sdl(6):
create table student (
students.sdl(7): student_name char(35) primary key students.sdl(8): );
students.sdl(9): students.sdl(10): create table enrollment ( students.sdl(11):
begin_date date, students.sdl(12): end_date date, students.sdl(13):
student_status char(9), students.sdl(14): current_grade float,
students.sdl(15): my_students char(6) references class, students.sdl(16):
my_classes char(35) references student students.sdl(17): ); rdm-sql: commit; rdmsql:
.q
```

Note that a new subdirectory will exist in the document root named students.rdm. This directory contains the database files.

Populate the database

- Create an SQL file (say, populate.sql) containing the following contents:

```
insert into class values "MATH01", "Intro to Algegra"; insert into class values
"SOFT03", "Advanced Java"; insert into class values "HIST02", "World History";
insert into student values "Joe"; insert into student values
```

```
"Sarah"; insert into student values "Henry";
insert into enrollment values current_date(),
"2015-12-20", "active", 3.1, "MATH01", "Joe"; insert into enrollment values
"2016-01-04",
"2015-03-31", "enrolled", , "SOFT03", "Joe"; insert into enrollment values
current_date(),
"2015-12-20", "active", 3.2, "MATH01", "Sarah";
insert into enrollment values "2015-04-01",
"2015-06-16", "passed", 3.5, "HIST02", "Sarah";
insert into enrollment values current_date(),
"2015-12-20", "active", 2.6, "SOFT03", "Henry";
```

- With the rdm-sql utility, read the SQL statements into the database:

```
C:\RaimaDB> rdm-sql
Enter ? for list of interface commands.
rdm-sql: use students;
rdm-sql: .r populate.sql
populate.sql(1): insert into class values "MATH01", "Intro to Algegra";
*** 1 row(s) inserted
populate.sql(2): insert into class values "SOFT03", "Advanced Java";
*** 1 row(s) inserted
populate.sql(3): insert into class values "HIST02", "World History";
*** 1 row(s) inserted
populate.sql(4): insert into student values "Joe";
*** 1 row(s) inserted
populate.sql(5): insert into student values "Sarah";
*** 1 row(s) inserted
populate.sql(6): insert into student values "Henry";
*** 1 row(s) inserted
populate.sql(7): insert into enrollment values current_date(), populate.sql(8): "2015-12-20", "active", 3.1,
"MATH01", "Joe";
*** 1 row(s) inserted
populate.sql(9): insert into enrollment values "2016-01-04",
populate.sql(10): "2015-03-31", "enrolled", , "SOFT03", "Joe";
*** 1 row(s) inserted
populate.sql(11): insert into enrollment values current_date(), populate.sql(12): "2015-12-20", "active",
3.2, "MATH01", "Sarah";
*** 1 row(s) inserted
populate.sql(13): insert into enrollment values "2015-04-01",
populate.sql(14): "2015-06-16", "passed", 3.5, "HIST02", "Sarah";
*** 1 row(s) inserted
populate.sql(15): insert into enrollment values current_date(), populate.sql(16): "2015-12-20", "active",
2.6, "SOFT03", "Henry";
*** 1 row(s) inserted rdm-sql:
commit; rdm-sql: .q
```

Note that the ".r" (read file contents) command is not necessary because these statements may be entered interactively. Here we use a file because typing errors can be corrected and re-entered.

Query the database

- Again, with the rdm-sql utility, open and query the three tables:

```
C:\RaimaDB> rdm-sql
Enter ? for list of interface commands.
rdm-sql: use students; rdm-sql: select * from
class;
class_id| class_name
--------+-----------------------------
HIST02 | World History
MATH01 | Intro to Algegra
SOFT03 | Advanced Java *** 3 row(s)
returned
rdm-sql: select * from student;
student_name
----------------------------------
Henry
Joe
Sarah
*** 3 row(s) returned
rdm-sql: select * from enrollment;
begin_date| end_date | student_ | current_ | my_students| my_classes
| | status | grade |
----------+----------+----------+----------+-----------+-----------
2016-01-05| 2015-12-20| active | 3.1| MATH01 | Joe
2016-01-04| 2015-03-31| enrolled | *NULL*| SOFT03 | Joe
2016-01-05| 2015-12-20| active | 3.2| MATH01 | Sarah
2015- 04-01| 2015-06-16| passed | 3.5| HIST02 | Sarah
2016- 01-05| 2015-12-20| active | 2.6| SOFT03 | Henry
*** 5 row(s) returned rdm-sql: .q
```

- Experiment

RAIMA

Want to know more?

Please visit our website for the latest news, product downloads and documentation:

www.raima.com

Headquarters: 3214 W. McGraw St., Suite #212, Seattle, WA 98199, USA T: +1 206 748 5300

Europe: Forneburingen 33, Oslo, Norway : +47 97075600