RAIMA

TECHNICAL WHITEPAPER

Raima Database Manager 12.0 Architecture and Features

By Wayne Warren, CTO – August 2013

Abstract

Raima Database Manager (RDM) v12 is a product developed by programmers for programmers. Its packages, APIs and utilities can be employed in countless combinations, assuming the responsibility for data collection, storage, management and movement. Programmers using RDM v12 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 RDM v12 as a whole, as pieces, and as solutions built from those pieces.

RDM v12 can be used for meaningful solutions in multiple industries, including Industrial Automation, Aerospace and Defense, Telecommunications, and Medical. This paper is for those who want more than "what" RDM can do, but also "how" it does it. As much as possible, this paper will just state the facts, and leave the hype to the reader. A basic understanding of application software development is required to fully understand these facts. Also, knowledge of operating environments, such as Linux, Windows, Real-time/embedded operating systems, networking (both Local- and wide-area), mobile environments, such as iOS, and computer architectures is assumed.

Prerequisites: A basic understanding of application software development.

CONTENTS

1. THE BIGGER PICTURE 3
2. STANDARD FUNCTIONALITY 3
2.1 Core Database Engine 3
2.1.1 Storage Media 3
2.1.2 Database Definition Language (DDL) 4
2.1.3 Database Functionality 4
2.2 Data Modeling 4
2.2.1 Relational Model 4
2.2.2 Network Model 4
2.3 Core DDL 5
2.4 SQL DDL 6
2.5 ACID 6
2.6 Runtime Library 7
2.7 Security through RDM Encryption 8
2.8 Raima's Transactional File Server concept 8
2.9 TFS Configurations 9
2.10 RDM In-Memory Optimizations 11
2.10.1 In-Memory Database Operation 11
2.10.2 Shared Memory Transport 11
2.11 Bulk Inserts 11
2.12 Raima Supports Five API's 11
2.12.1 Currency View 12
2.12.1 Cursor View 12
3. Database Unions 12
4. Interoperability 13
5. RDM Plus Functionality 13
5.1 RDM Mirroring 13
5.2 RDM Replication 14
5.2.1 Replication Client API 15
5.2.2 Notifications 15
6. Putting the Pieces Together 16
6.1 High Availability 16
6.2 High Throughput 17
6.3 Networking 18
6.4 In the Office 19
6.5 In the Field 20
7. Conclusion 24

Copyright © 2013 Raima, Inc. All rights reserved.

1. THE BIGGER PICTURE

Raima Database Manager (RDM) is packaged by distinct environment groups: RDM Mobile, RDM Embedded and RDM Workgroup. Figure 1 below illustrates the complete RDM offering:

Each RDM environment can stand alone or interoperate with the others. A Mobile application (on a smartphone or tablet with Android or iOS) can be developed to operate independently of the RDM Workgroup (desktop, laptop, or server computer running Windows, Mac OS X, Linux or Unix) environment, or it can be created as an extension of an application already running in an RDM Workgroup environment.

Each environment group has an RDM Standard and RDM Plus package. The Standard Packages will satisfy most application development needs in that environment, while the Plus Packages add the more sophisticated data movement options.

The standard RDM Mobile and RDM Embedded packages allow standalone development, meaning that databases are not shared between different computers. The RDM Workgroup environment allows for basic distributed processing. With the RDM Plus packages, distributed processing as well as RDM mirroring and RDM replication are available.

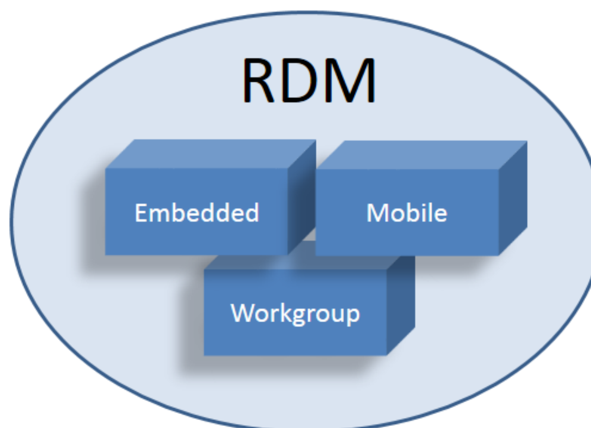

Figure 1: RDM Packages

Only on the Apple environments (iOS and Mac OS X), we have provided an Objective-C interface. All other interfaces (C, C++, SQL, and ODBC) are available as C-callable APIs in all packages. For Windows environments, ADO.NET, JDBC, ODBC drivers are also available.

The following discussion about RDM functionality is applicable, with minor exceptions, for all environments.

2. STANDARD FUNCTIONALITY

Digging inside each package, we find most of the technology is within the RDM Core Database Engine. Understanding its basic functionality is required before the Plus packages can be understood.

2.1 Core Database Engine

The Core Database Engine was first released in 1984 under the name db_VISTA. In the years since then, the basic functionality of this engine has remained intact while many additional features have been added.

2.1.1 Storage Media

An RDM database is composed of one or more computer files. Frequently, these files are stored in an operating system's file system, which can be using disk drives, SD RAM, SSD or main memory as the underlying media. RDM uses standard file I/O functions to access the file system.

An important issue with durable storage like disk and SD RAM is that an operating system will almost always maintain a 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.

Copyright © 2013 Raima, Inc. All rights reserved.

To safeguard against this, RDM 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.

RDM also has its own internal in-memory storage system that allocates RAM for storing database contents. This is much faster than permanent media storage, but is vulnerable to loss if there is a program or computer error.

2.1.2 Database Definition Language (DDL)

Database files are named in the Database Definition Language (DDL) written by the programmer (see 2.2 Data Modeling below). RDM defines 6 different file types for database storage:

1. Database Dictionary—also called DBD file, and ends with a “.dbd” suffix. This contains a definition of the sizes and locations of all data stored in the other database files.
2. Data File—a data file stores records. It is typical to name a data file after the record type it stores, using a “.dat” suffix, or to name the database and use a sequential “.dNN” suffix. Each record is stored in a *slot*, which is stored in a *page* in a data file. For any given data file, the slot and page sizes are fixed. Normally, one type of record is stored in a data file, but multiple record types may be stored in the same file. A *page* is a unit of I/O, where everything in the page is either read from or written to the file as a unit.
3. Key File—a key file uses a b-tree indexing structure to maintain a sorted, direct-access list of keys. Like data files, their suffixes are usually “.key” or “.kNN”. Key files also use fixed-length pages and slots.
4. Hash File—hashing is a different method used to store and look up keys. Hashing allows quicker lookups than b-trees, but does not allow key ordering.
5. Vardata File—variable-length strings are managed by storing a reference to the string in the data file, and storing the string, probably in fixed-length pieces, in a vardata file.

2.1.3 Database Functionality

At the root of 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. As discussed below, databases are normally shared and contain valuable information, so the operations on a database must follow carefully defined rules.

2.2 Data Modeling

2.2.1 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 RDM allows a database to be defined using SQL, (see below) the predominant relational database language. Relationships in a pure relational model are defined by comparing column values in one table to column values in another. Indexing is a common method to optimize the comparisons.

2.2.2 Network Model

Beneath the relational model in an RDM database is a *network model*, where all data is defined in terms of record types and fields. Fields may be indexed, and record types may have *set* relationships between them, which are defined as one-to-many, owner/member relationships.

Note that *set* relationships occupy space in the records, stored in the data files. The owner record will contain pointers to member records. Member records will contain pointers to the owner record, plus the next and previous members. This allows for quick navigation among the members of a set.

RDM uses the set construct to represent relational equi-joins, which will be shown in the DDL examples below. Data in RDM is modeled by creating DDL (Database Definition Language). When DDL is compiled, a database dictionary file (DBD, as defined above) is created.

Copyright © 2013 Raima, Inc. All rights reserved.

2.3 Core DDL

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.
Wide characters – single or string.
Double, float
Date, Time, Timestamp
BCD
GUID
Binary – array or blob.

Core DDL defines records and indices, and identifies the files containing them. As a simple example, the following figure represents three record types and two sets that model the relationships between students and classes.

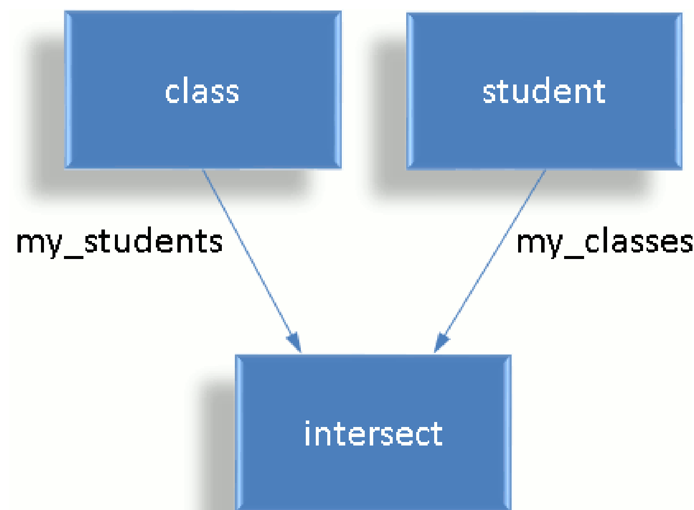

Figure 2: Student and Classes

To code the above data model in DDL, a textual language is used, which is shown below.

```
database students {
    data file "class.dat" contains class;
    data file "student.dat" contains student;

    key file "class_id.key" contains class_id;
    key file "name.key" contains name;

    record class {
        unique key char class_id[6];
        char class_name[30];
    }
    record student {
        key char name[36];
    }
    record intersect {
        int32_t begin_date;
        int32_t end_date;
        char status[10];
        int32_t current_grade;
    }
    set my_students {
```

Copyright © 2013 Raima, Inc. All rights reserved.

```
        order last;
        owner class;
        member intersect;
    }
    set my_classes {
        order last;
        owner student;
        member intersect;
    }
}
```

Note that *set* relationships occupy space in the records, stored in the data files. The owner record, for example, *class*, will contain pointers to member records. Member records will contain pointers to the owner record, plus the next and previous members. This allows for quick navigation among the members of a set.

The "`data file`" and "`key file`" statements are no longer required, as they were from version 1 through 11. They will be assigned default names, with one file per record type, key type or other type of file.

2.4 SQL DDL

If SQL is the chosen interface from the program to the database, you would start with SQL DDL. In RDM, the SQL DDL gets translated into Core DDL, and the extra information needed only by SQL is stored in a separate catalog file that is used together with the DBD file. To model the student and class data shown above, SQL DDL is written:

```
create database students;

create table class (
    class_id char(5) primary key,
    class_name char(29)
);

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

create table intersect (
    begin_date integer,
    end_date integer,
    status char(9),
    current_grade integer,
    my_students char(5) references class,
    my_classes char(35) references student
);
```

Note that the *primary key* in the class record, and the *references* in the intersect cause RDM to create a set relationship between the class and intersect record types at the Core DDL level, representing an equi-join.

2.5 ACID

RDM 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.

Copyright © 2013 Raima, Inc. All rights reserved.

Maintaining the ACID properties is the “hard work” of a DBMS. Application programmers shouldn’t solve these problems again. RDM 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 record containing that key field exists too. Isolation has to do with a community of users never seeing changes done by others except as complete 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.

The isolation property was enhanced starting with version 10.0 of RDM, when read-only-transactions were introduced. Read-only-transactions are a form of Multi-Version-Concurrency-Control (MVCC), allowing readers to view what appears to be a snapshot of a database at a moment in time, even though the database is being actively updated. This is implemented by keeping track of the version of each page in a database. Whenever a page is changed by a transaction, RDM keeps a copy of the version of the page needed by the reader. When the reader asks for that page, they are presented with the correct one. This means that readers using read-only-transactions do not need to issue locks, which would prevent updates from occurring until the reading is completed and the locks are freed. So the isolation property is maintained without the need for locks, which significantly increases the performance of read operations (reporting, etc.).

Yet another isolation-related feature is called “lockless reading” or “dirty reading.” Dirty reading allows an application to use data that is already in its cache, provided it is not too old (where “too old” can be decided by the application as a time-to-live parameter). Data read this way may be out of date. When this is understood and acceptable, it results in improved performance.

2.6 Runtime Library

The runtime library is linked into applications and performs database operations through the Core API, defined as a set of C functions. It keeps a cache of database file pages in its memory. Some of those pages may have been read from the database, others may have been created as new pages by the runtime library. The functions read or update the contents of the pages in the cache.

Functions in the runtime library can be grouped into the following general categories:

Database Control	Create or destroy databases. Open or close databases.
Transaction Control	Begin, commit or abort transactions.
Locking Functions	Lock records for shared reading or exclusive writing.
Record/Set Create/Delete	Create or delete records, connect and disconnect records from sets.
Navigation	Key lookup and scanning. Set scanning. Sequential scanning.
Read/Write Data	Read or write entire record contents or individual field contents.

If an application is only reading a database, its cache will be populated with pages from data and key files. To read pages, the application must either have the database exclusively (no other users) open, have locks on the records, or use read-only-transactions.

If an application is updating a database, it must begin a transaction, obtain locks on the records, make the updates, and then commit the transaction. All updates made by the application are kept in the cache until commit time. The application’s view of the database will include the updates, although no other applications will see any of the updates. To commit a transaction, all changed or new pages are written to a *transaction log* file, which is then applied in a controlled and recoverable manner to the database files. A separate component is responsible for performing the actual reads from the database files and safely committing the transaction log files, so that the runtime library doesn’t actually read or write the database files directly. This is the job of the specialized Transactional File Server, discussed next.

Copyright © 2013 Raima, Inc. All rights reserved.

2.7 Security through RDM Encryption

The RDM database pages are stored in the cache of the runtime library in a clear form for use by the database functions. However, if the database contains sensitive information, RDM allows the pages to be encrypted when they are written from the cache to the files. RDM supports the Rijndael/AES algorithm with 128, 192 or 256 bit keys.

Since the procedure to write cache pages to database files involves transaction log files, the encryption occurs when the transaction log file is created. Whether that file is committed to database files on the same computer or another one, the contents of those pages cannot be interpreted without first being decrypted. This means that data sent over the network will be in the form of encrypted pages, and if the Transactional File Server (TFS) is storing those pages, they are stored encrypted.

When a runtime requests a page, it will receive it in the encrypted form and must have the key to decrypt it when it places it into the local cache.

2.8 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 is a set of functions called by the runtime library to manage the sharing of database files 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 TFServer. To connect to a particular TFServer process, the runtime library needs to know the domain name of the computer on which TFServer 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 TFServer are on the same computer or different computers (when on the same computer, optimizations are made, and a shared-memory protocol is available by default).

In Figure 3, it shows that one runtime library may have connections to multiple TFServers, and one TFServer may be used by multiple runtime libraries. To the applications using the runtime libraries, and the TFServers, 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.

A TFServer 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* in which are stored all files managed by the TFS. If one computer has multiple disk controllers, it is recommended that one TFServer is assigned to each controller. This facilitates parallelism on one computer, especially when multiple CPU cores are also present.

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

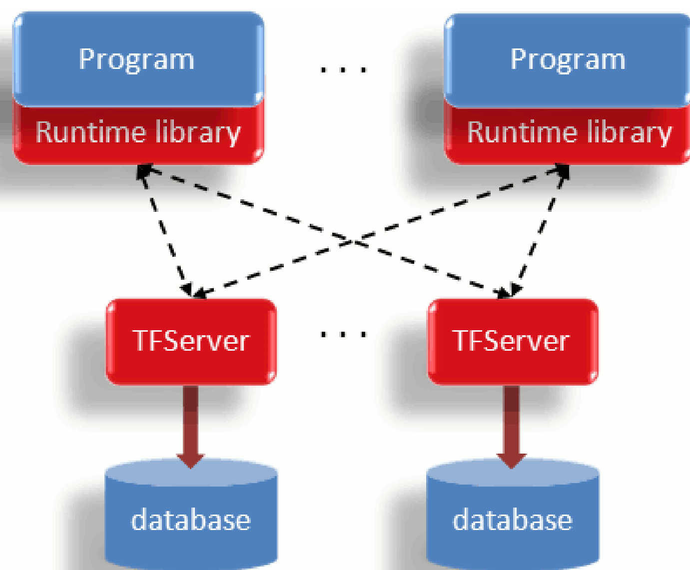

Figure 3: Runtime library, TFServer Configuration

Copyright © 2013 Raima, Inc. All rights reserved.

2.9 TFS Configurations

This may be one of the most powerful, yet confusing aspects of RDM. The TFS functions are used by the runtime library, so the programmer has no visibility of the calls made to them. These functions are made available to the runtime library in three forms. For descriptive reasons, we call them TFSr, TFSt and TFSs:

TFSt	The actual, full-featured TFS functions, called directly by the runtime library. Supports multiple threads in a single application.
TFSr	The RPC (Remote Procedure Call) library. When called by the runtime library, these functions connect to one or more TFServer processes and call the TFS functions within them. A client/server configuration.
TFSs	"Standalone" TFS functions called directly by the runtime library, but intended only for single-process use (if multiple threads are used, each must be accessing a different database only). To be used for high-throughput batch operations while the database(s) are otherwise offline. Unsafe (but fast) updates are allowed, meaning that database(s) should be backed up before making updates in this configuration.

The application calls the runtime library function d_tfsinitEx to select the TFS configuration. The default selection is TFSt, meaning that the TFS functions are called in-process and that a separate TFServer should not be started. Each of the command-line utilities has an option to select the TFS configuration.

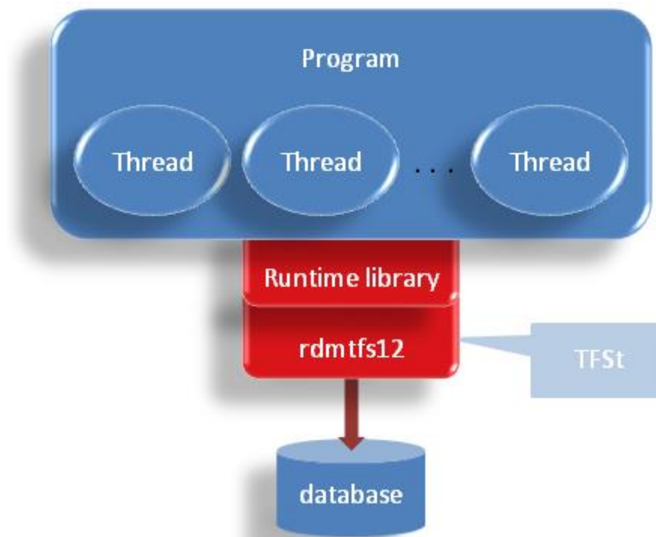

Figure 4: TFSt Functions called by Runtime

The program may be multi-threaded, and the TFS functions are the full-featured, ACID-compliant functions.

Copyright © 2013 Raima, Inc. All rights reserved.

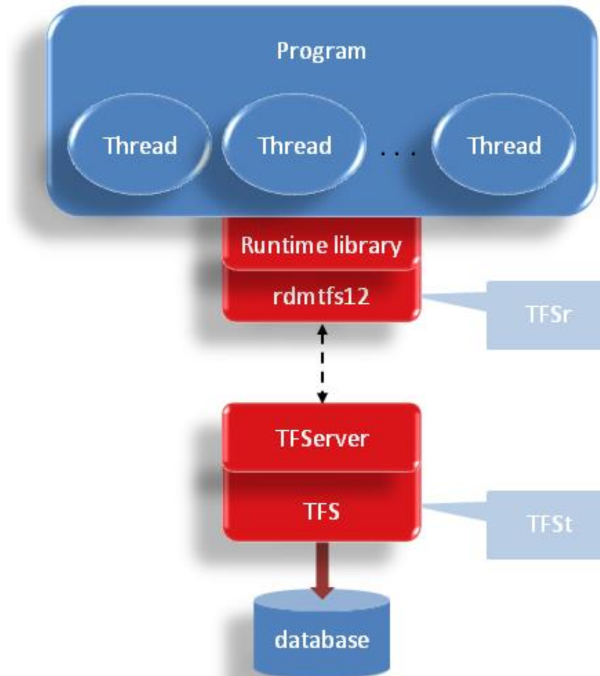

Figure 5: TFSr Functions called by Runtime

Here, the functions in rdmtfs12 are RPC stub functions that marshal the parameters into a packet and send the packet to TFServer, which demarshals the parameters, calls the actual TFS function, and sends the results back. The runtime library sees the same behavior from the RPC functions as it does from the TFS functions when they are linked directly (as in Figure 4). Like the TFSt functions, the TFSr functions are threadsafe.

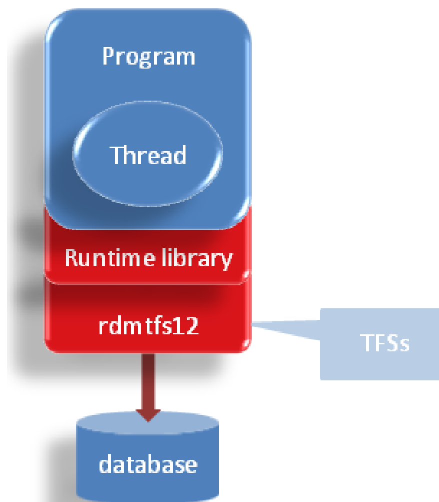

Figure 6: TFSs Functions called by Runtime

The third configuration, TFSs, is shown in Figure 6. In the standalone configuration, there are no true transactions. When space is needed in the cache, changes are flushed to disk safely. They are also flushed to disk when the database is closed. This facilitates very fast batch (overnight, offline) processing.

Copyright © 2013 Raima, Inc. All rights reserved.

2.10 RDM In-Memory Optimizations

2.10.1 In-Memory Database Operation

Databases may be declared as “in memory,” meaning that the entire database will be maintained in RAM. Files within databases may be declared as “in memory,” meaning that those files are maintained in RAM while the other files are stored in a file system (this may be referred to as hybrid storage).

A TFS, running within TFServer, can be told to be diskless. In diskless mode, it cannot accept database definitions that are not all in-memory. It will also keep all log files in memory. Log files are not stored in memory, even for in-memory databases, unless the TFServer is operating in diskless mode.

An in-memory database may be either *persistent* or *volatile*. Volatile databases are temporary, and are expected to be built and discarded in the course of an application’s operation. Persistent databases will have non-volatile backup. Figure 7 shows the flow of a persistent in-memory database.

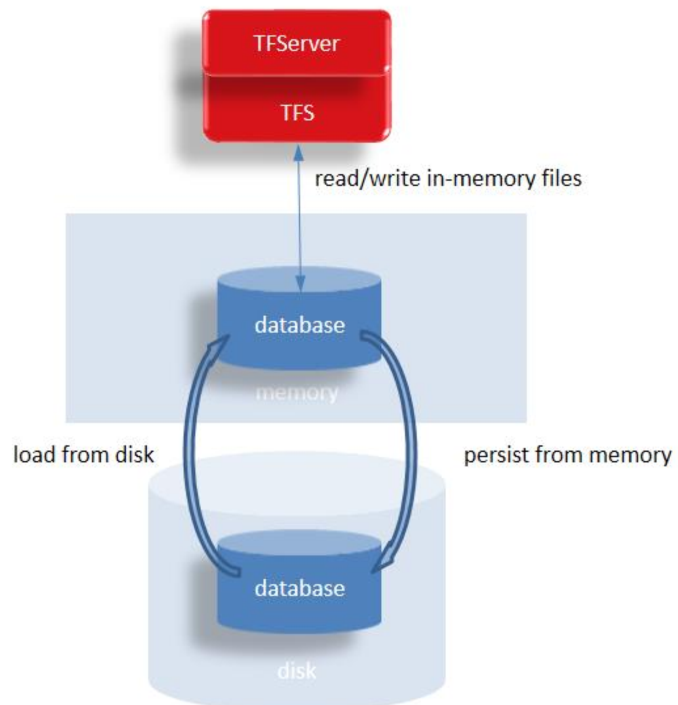

Figure 7: Loading & Saving a Persistent In-Memory Database

2.10.2 Shared Memory Transport

The RDM shared memory transport is an alternative to TCP/IP communication when both the server and the client communicating with it reside on the same machine. Instead of creating a socket for communications, a simple shared memory buffer is used.

When TCP/IP is used, the location of a TFS is usually transparent, whether it is on the same computer or not. However, the performance advantages of using shared memory segments are quite high, so Raima recommends the use of this transport protocol when one computer is being used for runtime(s) and TFServer(s). The transport is selected through a URI notation that is specified in the manual.

2.11 Bulk Inserts

Streaming data can create heavy loads on a DBMS. The least efficient way to capture a stream of data is to create one record at a time and commit it in its own transaction. Sometimes this is necessary, but it is not efficient. Transaction overhead can be controlled by “batching” updates within a single transaction. The overhead of each individual insert is very little compared to the overhead of the transaction, so if 100 inserts can occur in one transaction, that overhead is reduced substantially.

If the operation of the application allows the batching of inserts in a transaction, then it is also possible to store the record contents into an array in memory and submit them in one API call in the runtime library. The combination of the batch transaction and the insert of an array of records allows the maximum possible performance in storing streaming data.

2.12 Raima Supports Five API’s

Five programming APIs are available in RDM. The Core API is intended for use with the C language. The other APIs go through it. The C++ API uses C++ methods that have been created from the DDL. The RSQL API is Raima’s Embedded SQL API that is compact and simple so that it works well in small-footprint applications.

Copyright © 2013 Raima, Inc. All rights reserved.

The Objective C API is available if you are on an Apple (iOS or Mac OS X) platform. Finally, ODBC is the standard API for accessing SQL databases. This one is built on top of RSQL, and is available for programmers who are already familiar with the standard.

The Core API operates directly on the network model structure of the database. SQL benefits from it, but the network model sets are hidden from the SQL user. The Core API contains two distinct views of the network model database, which we define as the *currency* view and the *cursor* view.

2.12.1 Currency View

This is the classic view, where the word *currency* refers to the current record, current owner of a set, current member of a set, and current position in a table. Almost all navigation operations result in a new current record. For example, a key search will locate the record containing a certain key. If the current record should be the starting point for finding all members of a set, then the current owner of a set instance will be assigned from the current record. Then from the current set owner, movement from one member to the next (or previous) will change the current member.

The strength of the currency view is that all of these positions can be maintained within the runtime library, and application needs to only be aware of where it stands in its own algorithm.

The weakness of this view is that there can only be one *current* record, owner, member etc. What if three or four positions within the same set are all important? Currency cannot help solve that in an elegant way.

2.12.1 Cursor View

The cursor view allows a range of records of a specific type to be defined and assigned a current position within the range, representing one record. Navigation within a cursor involves moving to the first, last, next, previous, or keyed position. Cursors are defined in various ways, such as “all members of a set instance” or “all records included in an index.” Multiple cursors may be defined on the same range, overcoming this shortcoming of the currency view.

Navigation also involves relationships between cursors. For example, the current record in one cursor might be the owner of a set of member records. A new cursor can be defined to be all of the members of that record. This new cursor, derived from the other one, can be used for its own independent navigation.

3. Database Unions

The database union feature provides a unified view of multiple identically-structured databases. Since RDM allows highly-distributed data storage and processing, this feature provides a mechanism for unifying the distributed data, giving it the appearance of a single, large database. As a simple illustration, consider a widely distributed database for an organization that has its headquarters in Seattle, and branch offices in Boston, London and Mumbai. Each office owns and maintains employee records locally, but the headquarters also performs reporting on the entire organization. The database at each location has a structure identical to the others, and although it is a fully contained database at each location, it is also considered a partition of the larger global database. In this case, the partitioning is based on geographical location.

The mechanism for querying a distributed database is simple for the programmer. When the database is opened, all partitions are referenced together, with OR symbols (“|”) between the individual partition names.

Partitioning and unified queries are also used for scaling the performance. Consider a database where each operation begins with a lookup of a record’s primary key. If the “database” is composed of four partitions, each stored on the same multi-core computer, but on different disks controlled by different disk controllers then the only requirement is a scheme that divides the primary key among the four partitions.

If that scheme is a modulo of the primary key, then the application quickly determines which partition to store a record into or read the record from. Since there are multiple CPU cores to run the multiple processes (both the applications and the TFSs), and the four partitions are accessible in parallel (the four controllers permit this), the processing capacity is four times bigger than with a single-core, single-disk, single-partition configuration.

Copyright © 2013 Raima, Inc. All rights reserved.

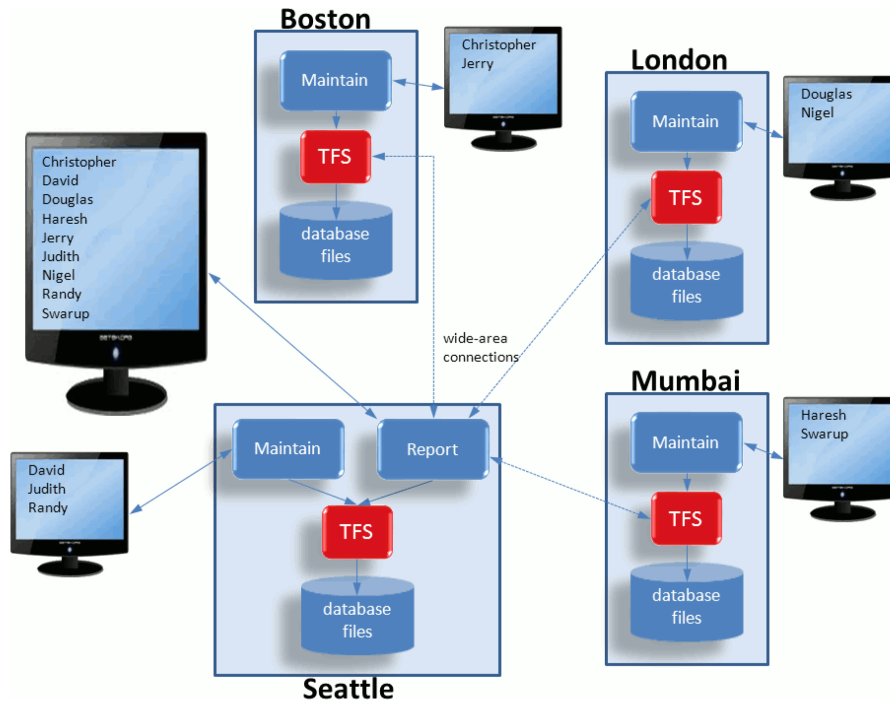

Figure 8: Wide-area Distributed Database

4. 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 an RDM database. While most application systems based on RDM are “closed,” there are many advantages to using languages (Java, C#, etc.) and tools (Excel, Crystal Reports, etc.) to access the data used by the system. Raima has chosen ODBC, JDBC and ADO.NET 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 and ADO.NET permit connection to an RDM database using the standard methods.

5. RDM PLUS FUNCTIONALITY

The Plus functionality adds the ability to move data (or operate in a distributed environment) through mirroring or replication (each of which will be defined below).

5.1 RDM Mirroring

Raima defines a database *mirror* as a byte-for-byte image of an original (master) database. Hence, mirroring creates one or more additional copies of an RDM database. And rather than just copying files, the process involves copying transaction logs so that mirror copies are updated *incrementally* and *synchronously*.

As discussed above, runtime libraries create transaction log files and submit them to the TFS. The TFS then assigns the transaction log a number and places it into a file identified by the transaction number. In due time, the log will be safely written to the database files, together with other transaction logs.

Copyright © 2013 Raima, Inc. All rights reserved.

Ordinarily, the transaction log files may be deleted after they are written to the database files. But when mirroring is active, the log files are retained. Then, upon request (from “subscribers”), they are copied to one or more other computers, where they are written to the database files there, bringing those database files up to date with the originals. The same process for performing safe updates of database files is used on both master and slave computers. Besides the pieces required for safe transfer of the log files, no additional software (meaning no additional complexity) is required.

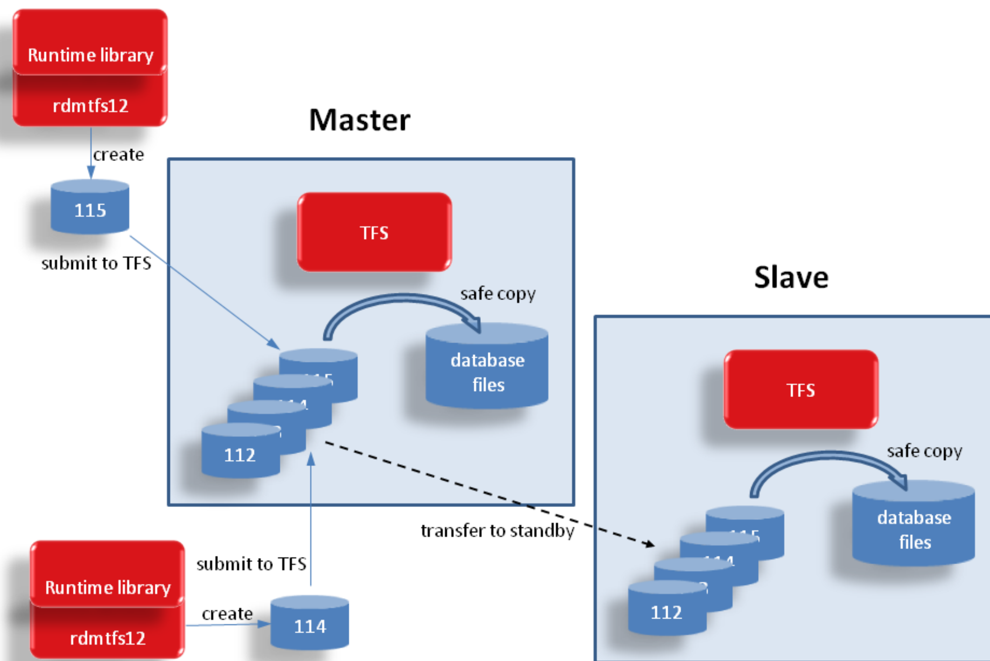

Figure 9: Mirroring by Copying Transaction Logs

5.2 RDM Replication

Raima’s replication functionality provides a different angle on moving data, as compared to mirroring. Raima defines *replication* as an action-for-action movement of data, rather than byte-for-byte, as it is with mirroring. Because of the replication of database actions, it is possible to *aggregate* data from multiple RDM master databases into a single database. It is also possible to represent the actions as commands to different types of database systems.

With action-for-action replication, changes in an RDM database are captured as a series of create, delete, update and connect operations, in replication logs. The runtime library creates these replication logs when configured to do so.

Each log contains the actions of one full transaction. The log is then transferred to the TFS which is in charge of administering all replication logs to replication clients. Replication clients receive the replication log files and convert them into the equivalent actions necessary for the local DBMS, which may be another Raima database (RDM Server or RDM), but more than likely is MySQL, SQL Server or Oracle. For the SQL DBMSs, the actions are converted into SQL.

The RDM processing of replication log files is the same as the processing of transaction log files. Figure 10 illustrates the replication process.

Copyright © 2013 Raima, Inc. All rights reserved.

Figure 10: Replication using Action Logs

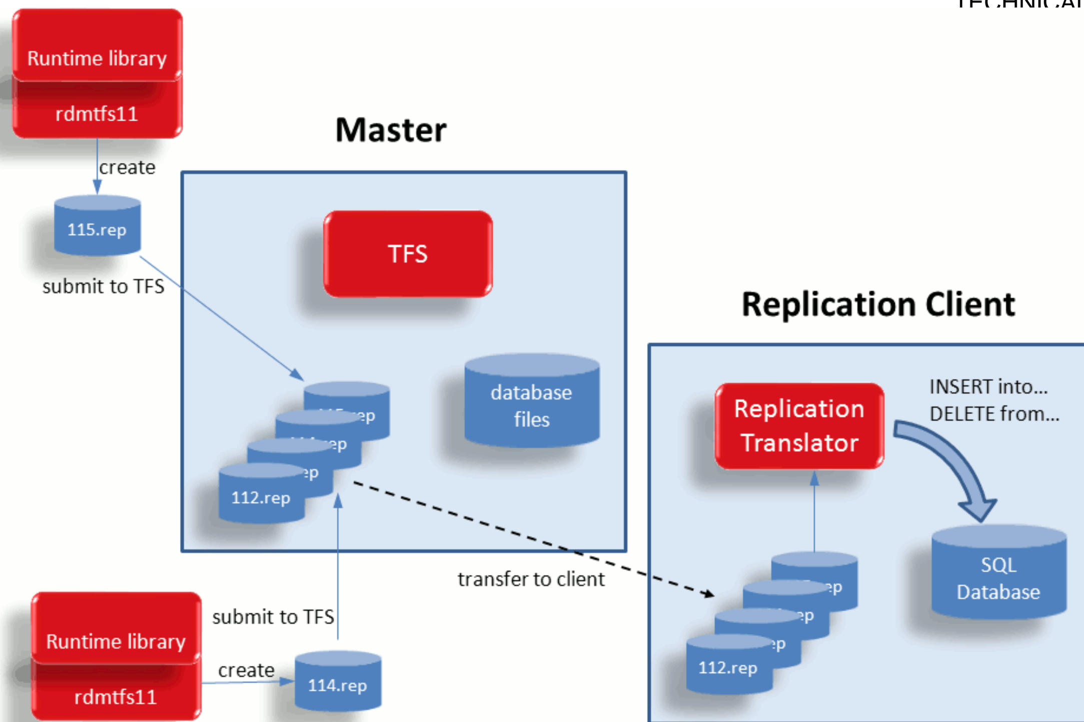

Figure 10: Replication using Action Logs

5.2.1 Replication Client API

RDM replication can be achieved through the use of the replication slave utility that comes in the RDM Plus package. But for programs that want to subscribe to replication logs for a certain database, an API will allow them to do that. The API allows a subscription to be started, and logs to be requested, opened and scanned, so that the configured changes (selected tables & columns, pre, post or no-image) can be processed in any desired manner.

As mentioned above, this allows a program that maintains a search database to know when to add or delete keywords. It also allows programs to know when and how certain records have been changed, enabling a notification system, described in the next section.

5.2.2 Notifications

Notifications are a special case feature that can be implemented through the replication system and Client API. If a process needs to know when certain rows in a database have changed, then the master database needs to be configured to selectively replicate those rows. Since replication logs are not completed and available to subscribers until their changes are committed to the master database, they can be delivered at the correct time to subscribers. Any data that subscribers attempt to read from the database will be found in the database.

When notification is required, a subscriber that receives a replication log file should open and scan it for the IDs of rows that are of interest. It then has the option to do something with pre or post images, or to read the row directly from the database using the row ID. The subscriber process may or may not be a database user, as access to the database is not required in order to request and process log files.

Copyright © 2013 Raima, Inc. All rights reserved.

6. PUTTING THE PIECES TOGETHER

This section will illustrate several configurations that solve different types of problems. The subsections will identify the problem, and show how the pieces are used together.

6.1 High Availability

The Problem:	The application(s) must keep running all of the time, this application's "state" is far more than a few kilobytes, and it changes frequently.
The Solution:	Build a redundant application with an active and a standby computer. Keep the application's "state" in a database, and synchronously mirror the database from the active to the standby computer.
Package:	RDM Workgroup Plus

Even more than ever, it is not acceptable for a computer system to discontinue its service even for a minute. There are many practical solutions to this requirement, and RDM has been designed to support an active/standby configuration. The active computer is performing the work, but the standby is prepared to take over should there be any problem with the active.

The solution involves an application that stores every important piece of information into a database, an HA monitor process that runs on both active and standby computers, and RDM for storing the data and mirroring it from the active to the standby computer.

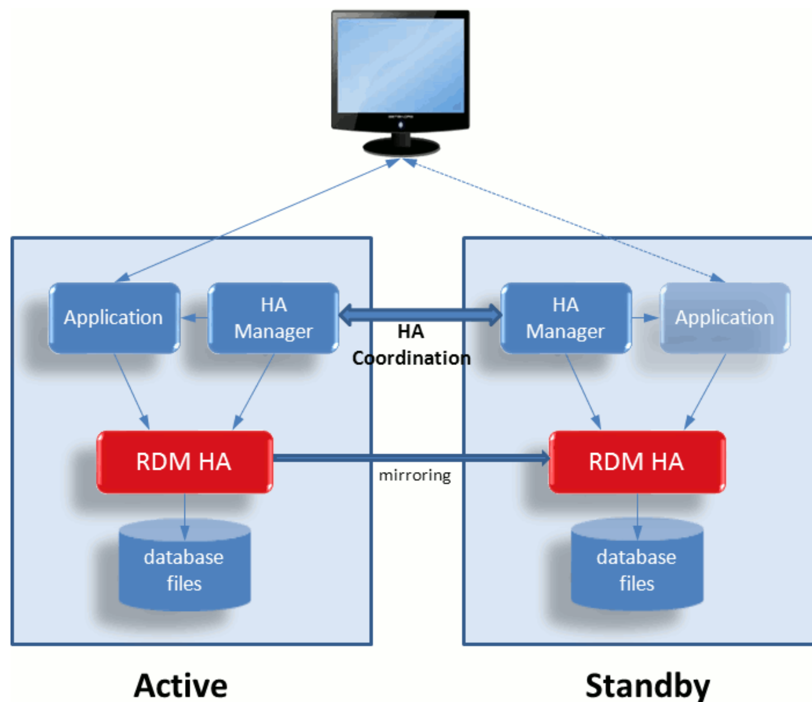

Figure 11: High Availability Configuration - Active/Standby

Should there be a problem with the active computer, where it is no longer able to respond or perform, the application's HA Manager on the standby computer is responsible for determining its need to take over. It will find the database fully up-to-date, and will be able to switch it from being a mirroring slave into a master database. Then the application can be restarted or activated on the standby computer and take over exactly where the active computer left off.

Copyright © 2013 Raima, Inc. All rights reserved.

6.2 High Throughput

The Problem:	System throughput is critical, and the load is expected to increase over time. You need to make sure the system keeps up with current demand and be able to scale up the performance to keep up with the expected growth.
The Solution:	Facilitate parallelism. This section will show one scalable configuration.
Package:	RDM Standard

It's important to note that scaling up performance of a system always involves adding computer hardware. The goal, especially with a shared resource like a database, is to add pieces (both hardware and software) that can run in parallel. If a system is divided up into pieces that end up blocking or interfering with each other, nothing is gained. Again, parallelism is the key, if parallel units do not impede the others.

The architecture recommended here requires a separate disk controller for every disk drive. Why? Because even with multiple CPU cores executing multiple independent processes on different disk files, a single disk controller will end up serializing the disk access, creating a bottleneck. So the computer is a multi-core computer with 2 cores for every disk controller/drive. For example, 8 cores with 4 controllers/drives. Given this hardware configuration, a software configuration needs to be designed for parallel operation. A necessary ingredient for parallel software operation is a database that is partitioned such that each partition can be updated independently from the other partitions.

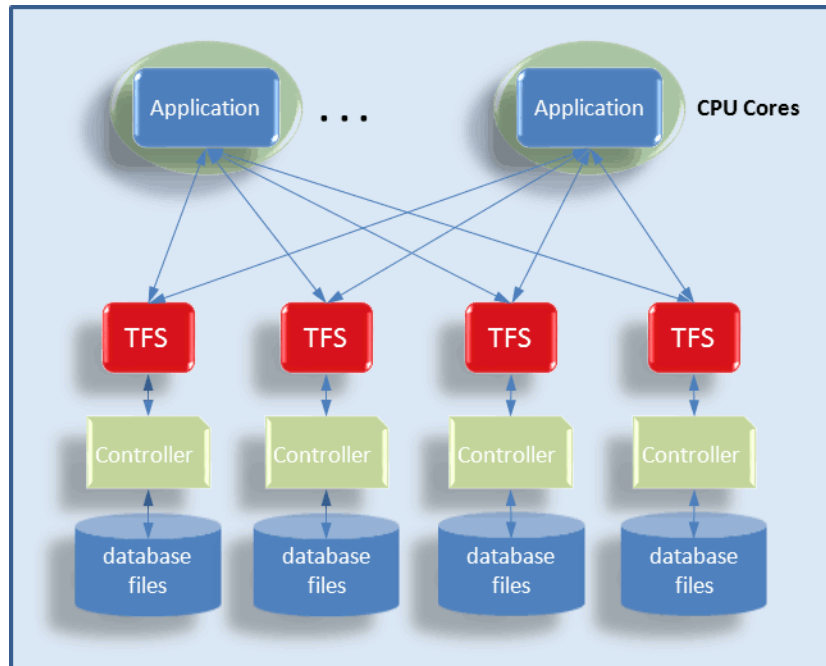

Figure 12: Scaling Up through Parallelism

The applications in the figure above will open 4 different databases within 4 different "task" structures, and then decide, based on a primary key, which database a record belongs in. It will either find it there or create it there. Reading is different. Within one "task" structure, all 4 databases should be opened in one call using the database union feature, and reading should be done without locks by using MVCC (Multi-Version Concurrency Control) read-only-transactions.

Note also that CPU cores are depicted as though they are assigned to application processes, but the reality is that they are normally operating as SMP, so they will be scheduled to execute the processes that are available. In this case, it will potentially be all 4 TFSs and up to 4 application processes.

Copyright © 2013 Raima, Inc. All rights reserved.

6.3 Networking

The Problem:	A wide-area application may be deployed worldwide, but may need to operate as a single suite of programs. Since processing may be widely distributed, it also makes sense to distribute the data. How do you do this without incurring performance problems?
The Solution:	The key is to minimize network communications and the latency that grows worse with distance. Use both mirroring and remote logins, depending on the particular interaction.
Package:	RDM Plus

The design heuristics are as follows:

- Databases should reside within the same computers as the processes that update them. Other processes that update the databases (through remote login) should be within a high-speed LAN.
- Databases that are frequently read by processes that are only accessible through WAN should be mirrored to the reading location. This will conserve network communications unless the database is updated frequently.
- Wide-area reading of databases that are frequently updated should be through remote login.

Since RDM allows both remote logins (accessing a TFS from a different computer) and mirroring (keeping a readable copy of a database that is mastered elsewhere), it is possible to optimize network performance by analyzing the volume of transactions and queries between different locations.

When a database is updated infrequently but read frequently from other locations, the overhead of sending changed pages to the mirror computers is much less than supporting remote logins from the remote computers. But very active databases can cause a flood of transaction logs to be transferred across a network, even if they are not going to be read before they are changed again. Remote logins are optimum when a remote process does infrequent reading, because the remote login will only send pages from the TFS to the runtime when they are needed by the reading process.

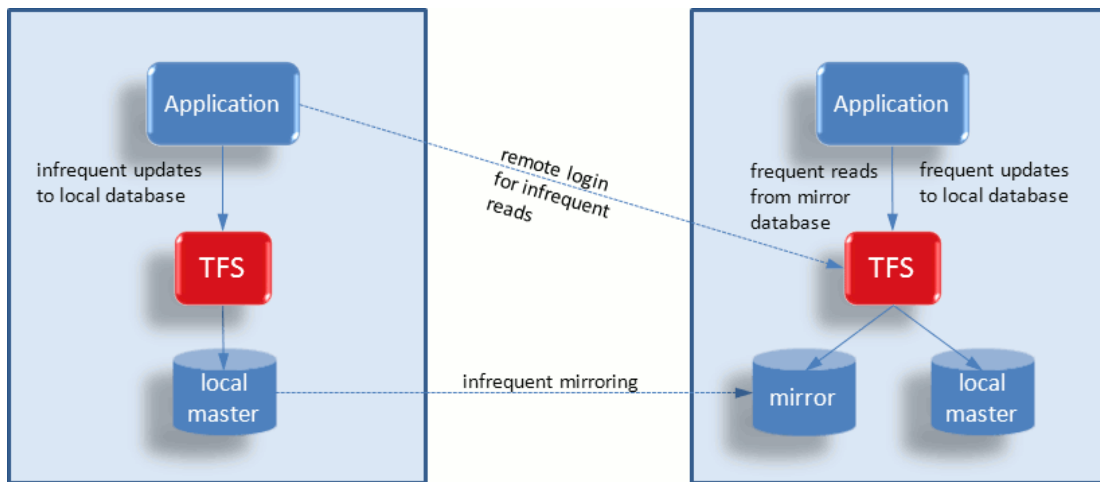

Figure 13: Mirroring or Remote Login

Copyright © 2013 Raima, Inc. All rights reserved.

6.4 In the Office

The Problem:	An application is developed to run very quickly in an office where several workstations are used for data entry. Each entry must be unique. The data must be replicated into an Oracle server after it is verified to be correct.
The Solution:	A single RDM database will be managed by a TFS running on one computer. All known data will be kept in this database so that existing entries can be updated or new ones can be added. All changes will be replicated to the Oracle database. Note that this solution is scalable through horizontal partitioning.
Package:	RDM Workgroup Plus

The generic concept of an office full of operators entering data into a database applies to a great many applications. For this example, consider it to be ticket orders, where operators receive calls from customers who may or may not have purchased tickets before. A record of all purchases will be maintained in the database. A completed order will be saved in the RDM database, and this order will also generate a replication log that is forwarded to the Oracle server, where the remainder of the ticket processing occurs.

A typical process cycle will have the application look up a name to find out if the person's record exists yet. An MVCC read-only-transaction does this without inhibiting performance. Then the person's record is created if necessary and the ticket order is processed. Once committed to the RDM database, the replication log will be forwarded to the Oracle computer where the RDM replication utility will enter the new or changed data into the Oracle database.

A single-partition solution is shown below:

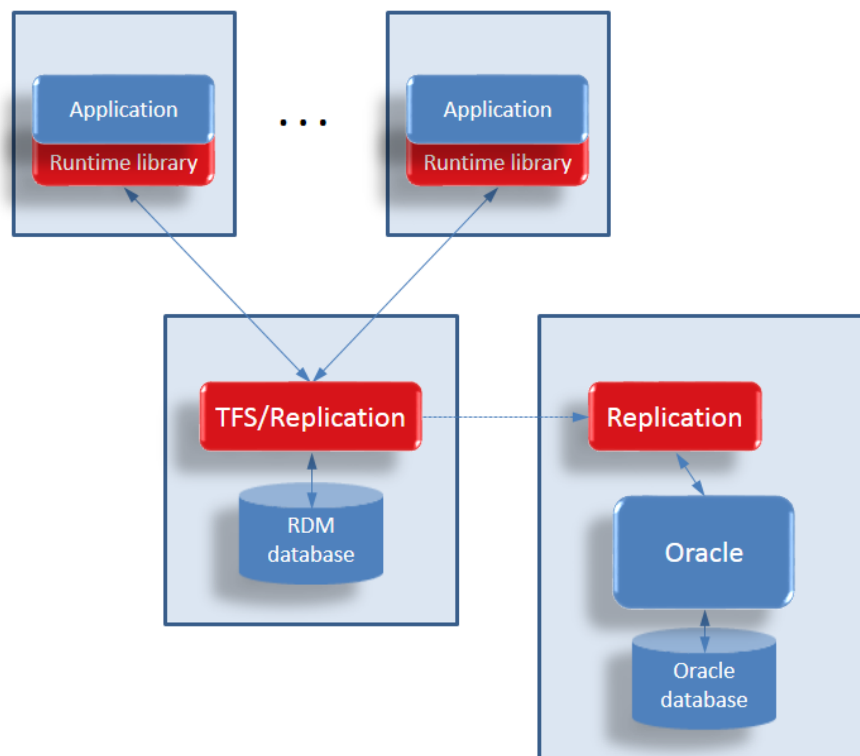

Figure 14: Data Entry Application System

Assuming that this is a highly successful call center, the system expands from 10 to 50 operators. The load created by the operators exceeds the capacity of one TFS, so a horizontally partitioned solution is deployed.

Copyright © 2013 Raima, Inc. All rights reserved.

This means that the primary key for a customer (probably last name, first name) is used by the application to determine which partition the customer record belongs in. If there are three partitions, each application will first determine which partition to use based on the name, and then perform exactly the same transaction as before.

The next figure shows the three-partition solution, where RDM is still used to replicate the orders to the Oracle server, which aggregates the entries from all sources. Note also that the Oracle server will receive updates identical to those that were submitted prior to the partitioning.

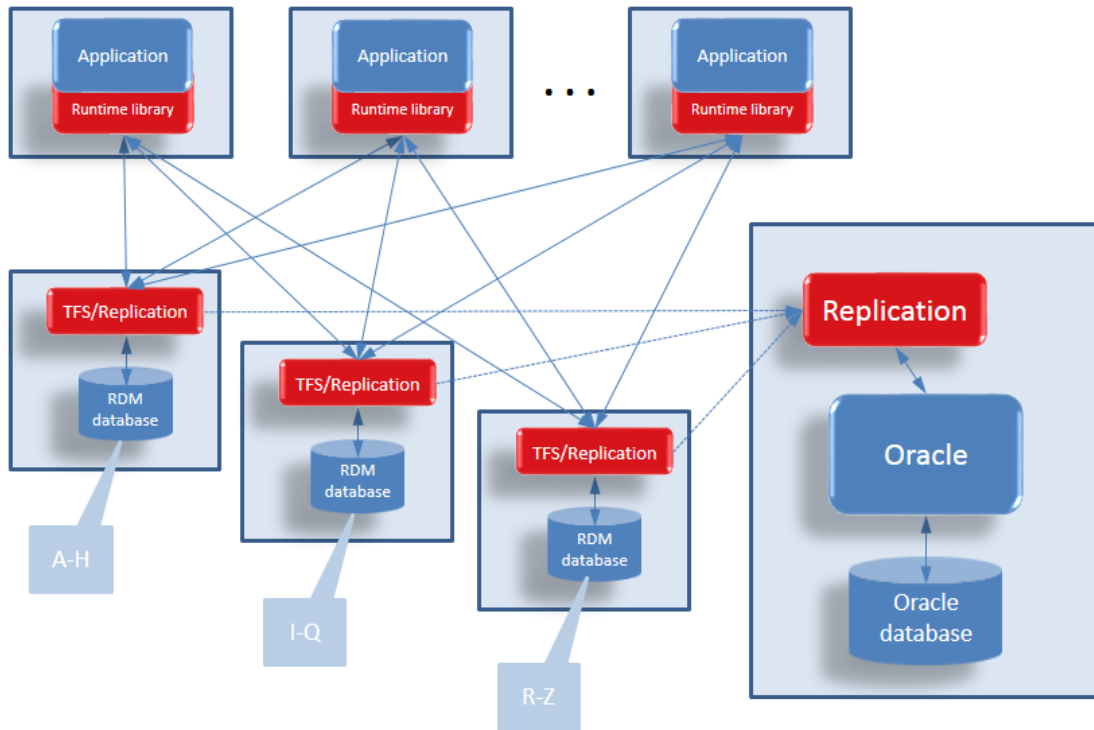

Figure 15: Partitioned Data Entry Application System

The Unified Query feature could be used with a partitioned database to perform queries on the entire RDM database (all three partitions). The queries could be written in the Core API when they are common or in SQL when ad-hoc queries are required.

6.5 In the Field

The Problem:	Embedded computers are now powerful enough to perform significant processing at the location of the relevant activity. Where devices formerly measured temperature, counts, pressure, etc., passing their readings on to another location that processed the inputs, now it is financially viable to replace these measurement devices with computers that can store parameters, read several inputs, filter and process the inputs, and pass the relevant data on to a higher level computer.
The Solution:	Taking advantage of powerful embedded computers requires order that can be realized with software. RDM can place manage fixed-length circular tables in multiple locations and replicate them into a single aggregated table where the information is used to make control decisions.
Package:	RDM Embedded and Workgroup Plus

Copyright © 2013 Raima, Inc. All rights reserved.

A wind farm consists primarily of a set of wind turbines that will generate power, but the reporting and control of such a farm is very data intensive.

Each turbine is a sophisticated machine that must react to its environment (wind speed and direction), monitor its condition, report its status, and respond to centralized controls. A SCADA system in a control room needs to gather the status from each turbine and produce visualizations for an operator, raise alarms for conditions that require attention, and allow controlling parameters to be sent to one or more turbines based on operator input.

Our solution for the wind farm is to place one embedded computer in the nacelle of each turbine, and one workgroup computer that runs or cooperates with a SCADA system.

The turbine's computer reads a number of sensed inputs which it records in a local database. The inputs include the state of the turbine (power output, rotor rpm's, pitch, yaw) and environmental conditions (inside and outside temperature, wind speed and direction). The local database table should be defined as "circular," meaning that it will wrap around on itself after a specified number of rows have been inserted. This keeps the table size bounded while offering control over the amount of time the rows are retained for local queries.

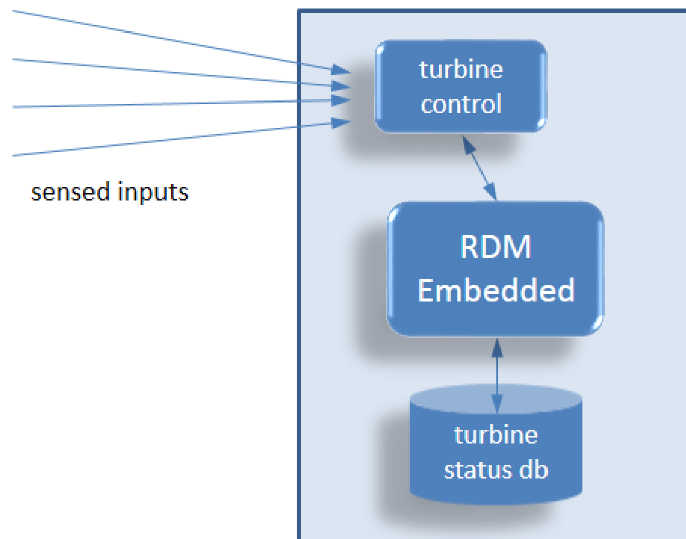

Figure 16: Turbine's Embedded Computer

Copyright © 2013 Raima, Inc. All rights reserved.

An important feature of the turbine's database is that it is available to the turbine's embedded computer for more intelligent decision making. Each row of the local database will be replicated to the SCADA which will have ultimate control, but some decisions must be made immediately. The ability to query its own historical status will be essential for this "reflexive" operation.

With local, fixed-length tables being produced at each turbine, is it possible to replicate each new row to a centralized aggregated database. This aggregated database can grow continuously so a complete history of the system is maintained. The following figure illustrates the flow of data:

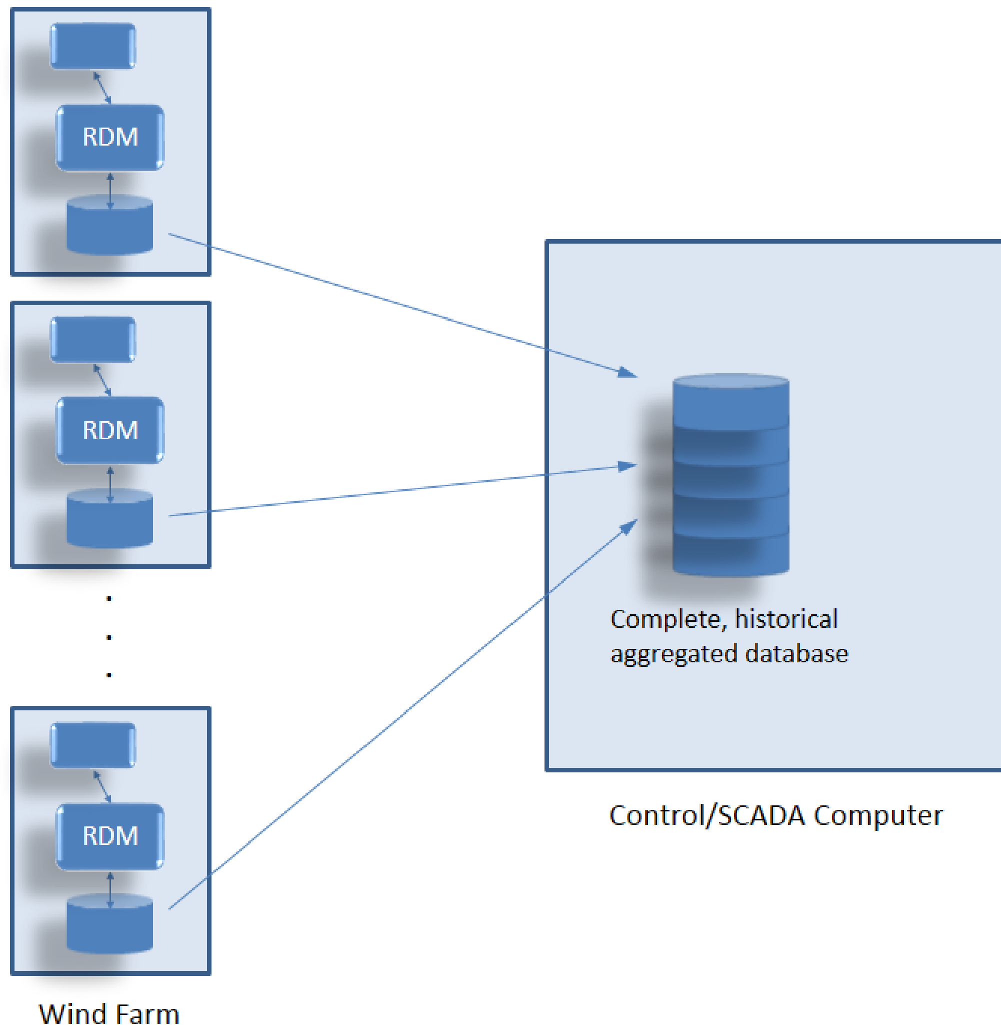

Figure 17: Aggregated Replication

Copyright © 2013 Raima, Inc. All rights reserved.

Operator/SCADA control can be implemented through the Notification Client API. This is implemented by defining a database that will reside on the Control computer containing control parameters that must be sent to one or more turbines (e.g. “park” or “run”). The rows will each identify the turbine(s) to receive the parameter values, and columns for each different parameter value. By storing a row into this master database, the Control computer triggers a broadcast of this row to all subscribers. Each turbine will subscribe to this master database, meaning that it will receive change records each time a transaction is committed in the master database. In this case, the master database is not replicated and stored in the slave (turbine) computers, but the notification of a new row is received and acted upon by the slave computers. The figure below depicts this type of data flow:

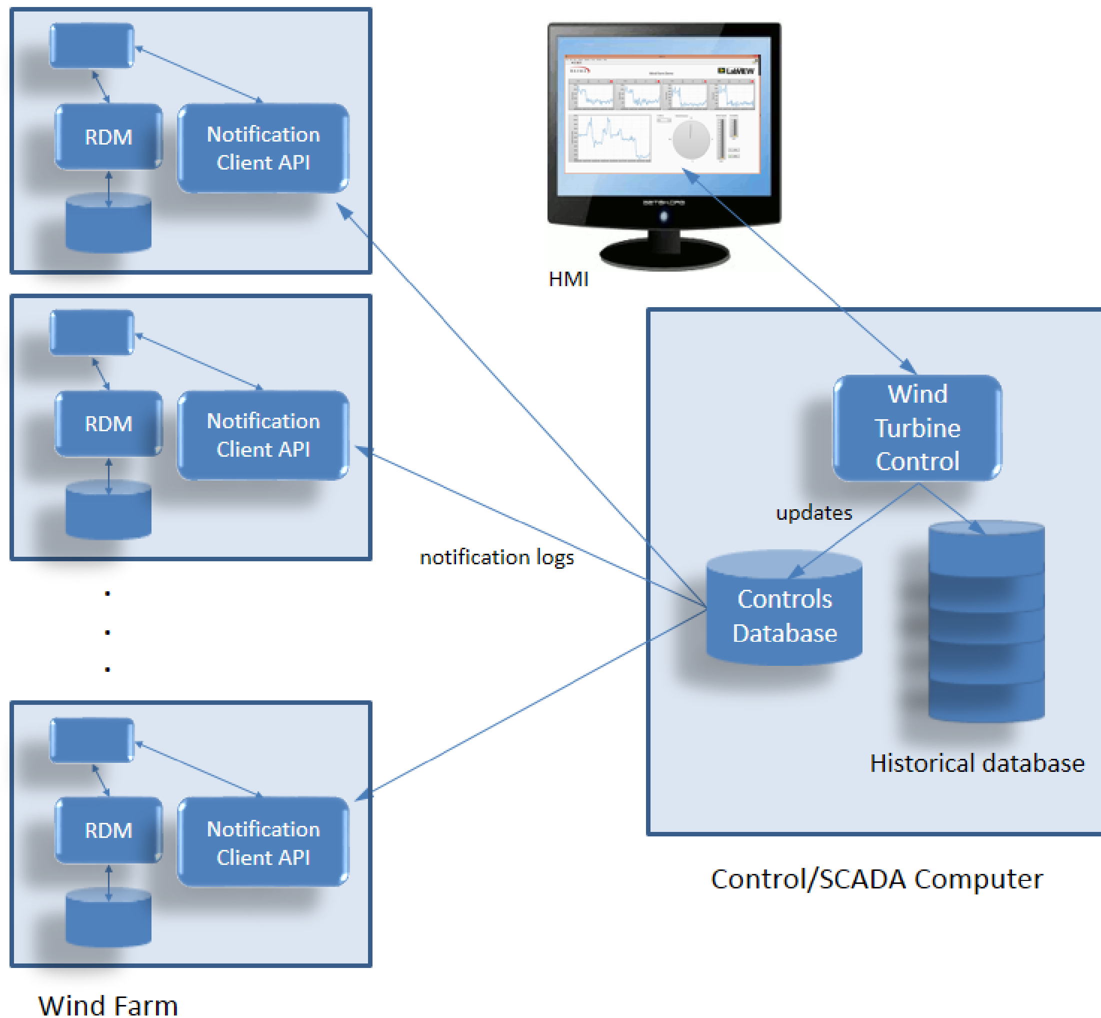

Figure 18: Notification Logs and Client API

Copyright © 2013 Raima, Inc. All rights reserved.

7. CONCLUSION

The goal of this paper is to provide a technical description of the RDM v12 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 both functionality and performance database samples, followed by downloading and running the full product. The database samples do not require any installation or setup, and allow you to see the RDM system in action within your own development environment. Together with that, the manual set contains extended examples and explanatory text.

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.

Want to know more?

Please call us to discuss your database needs or email us at info@raima.com. You may also visit our website for the latest news, product downloads and documentation:

www.raima.com

Headquarter: 720 Third Avenue Suite 1100, Seattle, WA 98104, USA T: +1 206 748 5300
Europe: Stubbings House, Henley Road, Maidenhead, UK SL6 6QLT: +44 1628 826 800

Copyright © 2013 Raima, Inc. All rights reserved.