RDM 11 Architecture and Features

By Wayne Warren, CTO – March 2012

Abstract

RDM 11.0 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 11.0 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 RDM 11.0 as a whole, as pieces, and as solutions built from those pieces.

RDM 11.0 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 with 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 BIG PICTURE

RDM is packaged by distinct environment groups: Mobile, Embedded, Desktop & Server and Enterprise Lite. Figure 1 below illustrates the complete RDM offering:

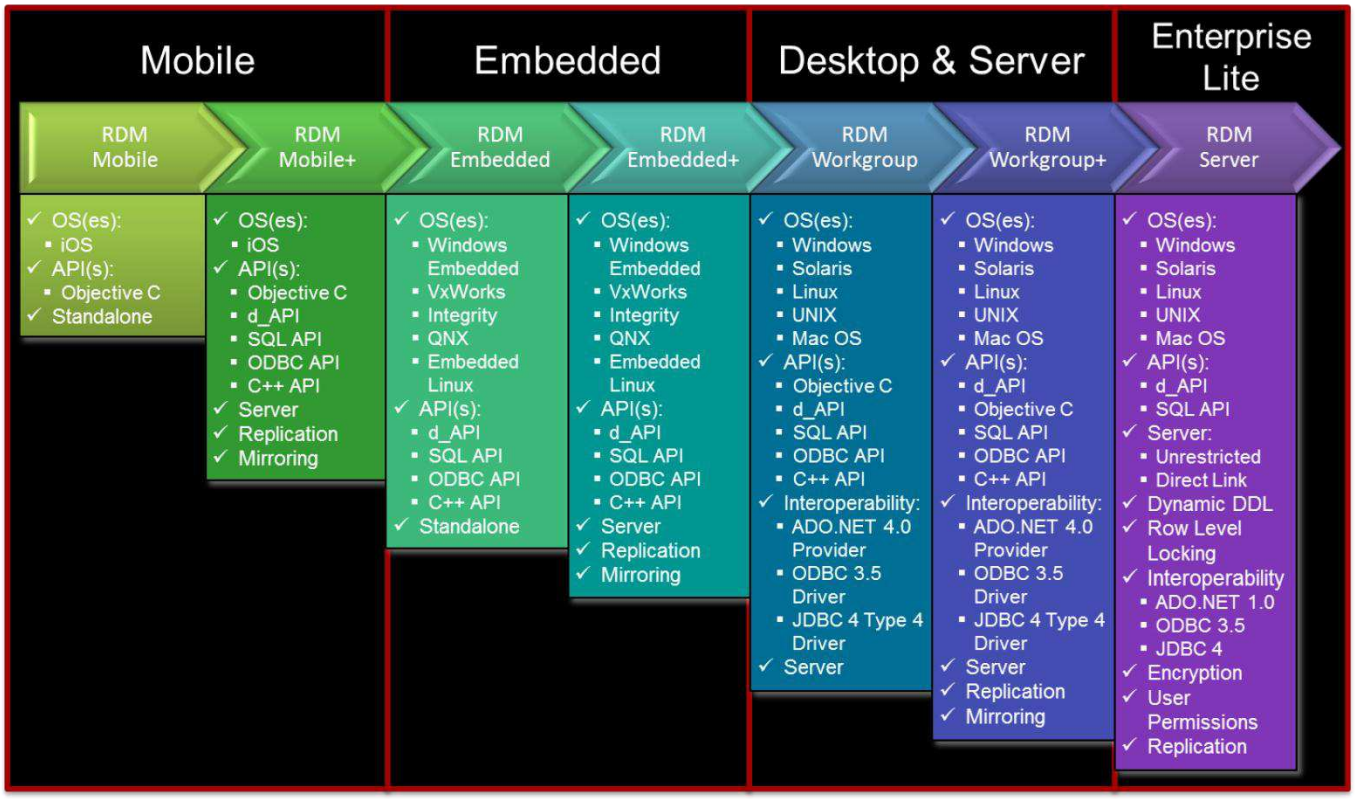

Figure 1: RDM Packages

Each environment can stand alone or interoperate with the others. A Mobile application can be developed to operate independently of the Desktop & Server environment, or it can be created as an extension of an application already running in the Desktop & Server environment.

Each environment group has a Standard and 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 Mobile and Embedded packages allow standalone development, meaning that databases are not shared between different computers. The Desktop & Server environment allows for basic distributed processing. With the Plus packages, distributed processing as well as mirroring and replication are available.

With the addition of the iOS environment, we have introduced an Objective-C interface, currently available only on Apple computers (phones, tablets, etc.). All other interfaces (C, C++, SQL, and ODBC) are available as C-callable APIs in all packages except standard Mobile. 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 Core Database Engine. Understanding its basic functionality is required before the optional 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 storage 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. 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 their transaction handling.

RDM also has its own internal in-memory file system that allocates RAM for storing file 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 do not maintain 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.

2.3 Core DDL

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.

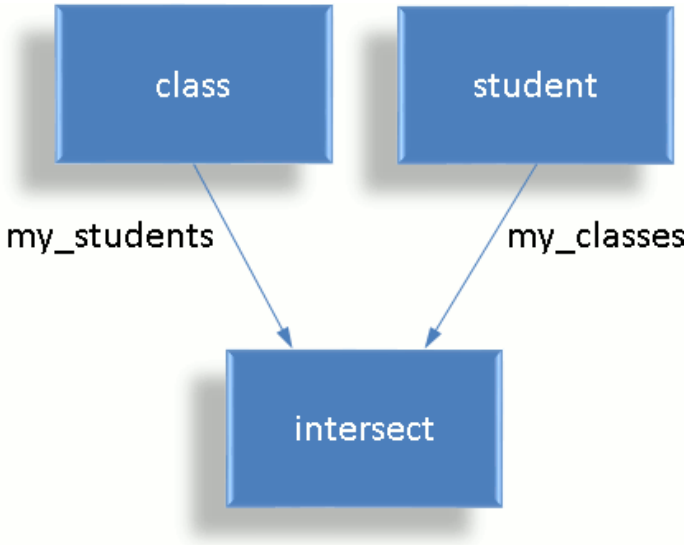

Figure 2: Student and Classes

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 {
        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.

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.

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 such as 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, it is 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.).

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.

2.7 TFS

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 2.8 Configurations 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 is 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).

In Figure 3 below, 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.

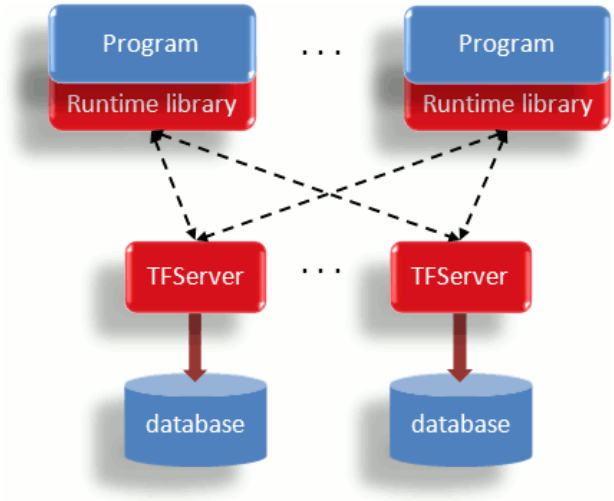

Figure 3: Runtime library, TFServer configuration

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.

2.8 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.

Currently, the runtime library is informed by the function called d_tfsinitEx. The default selection is TFSt, meaning that the TFS functions are called in-process and that a separate TFServer should not be started.

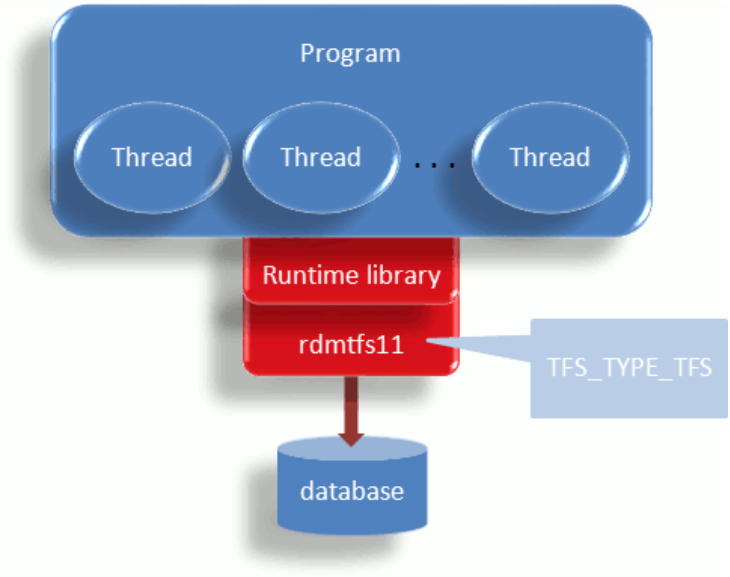

Figure 4: TFS_TYPE_TFS Functions called by Runtime

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

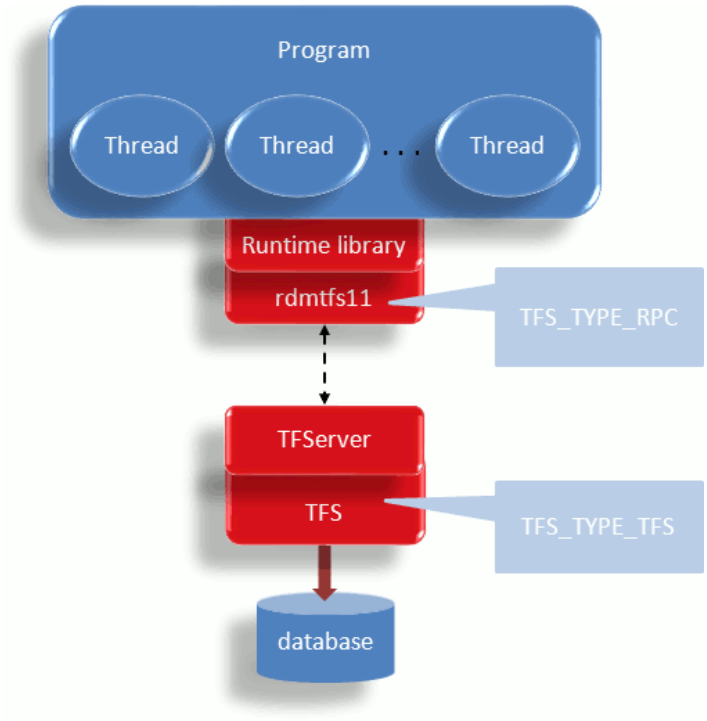

Figure 5: TFS_TYPE_RPC Functions called by Runtime

Here, the functions in rdmtfs11 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.

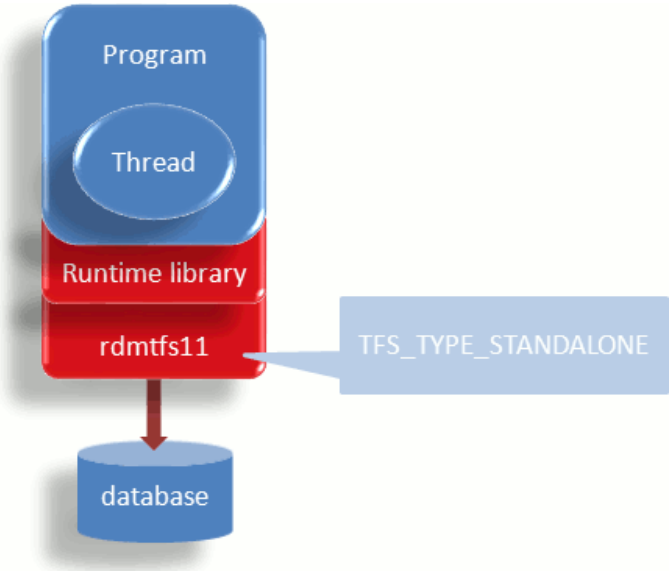

Figure 6: TFS_TYPE_STANDALONE Functions called by Runtime

The third configuration, TFSs, is shown in Figure 6. In the standalone configuration, the TFS functions in rdmtfs11 are deliberately stripped of their safety (double-writes to disk). This facilitates very fast batch (overnight, offline) processing.

2.9 In-Memory 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. The figure below shows the flow of a persistent in-memory database.

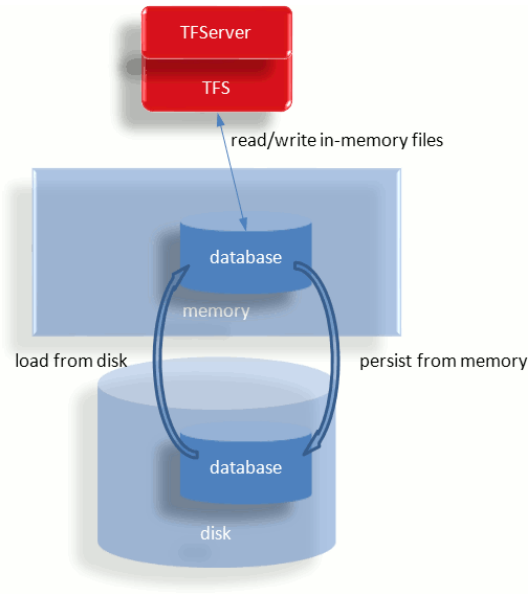

Figure 7: Loading & Saving a Persistent In-Memory Database

2.10 Language Support

Four programming APIs are available in RDM (five if you are using iOS). 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. 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.

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.

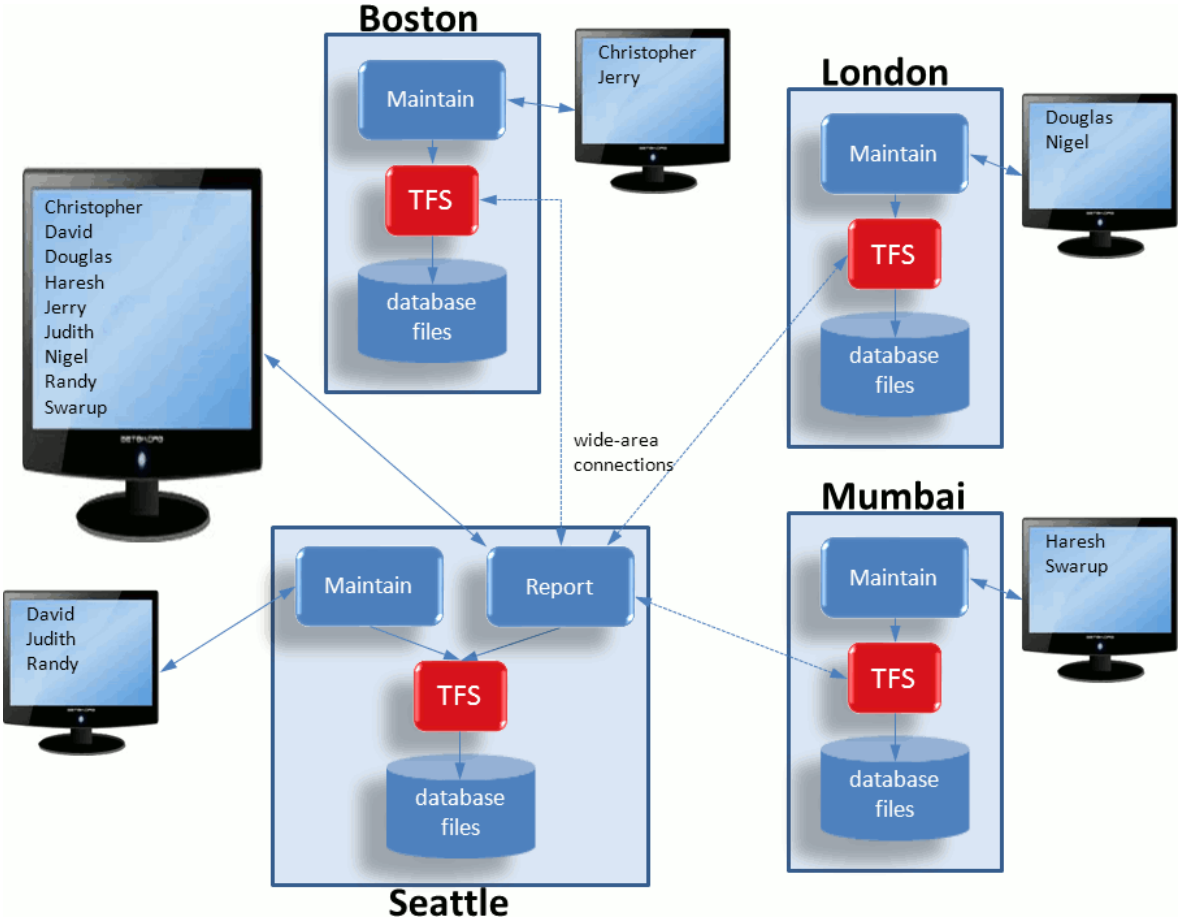

Figure 8: Wide-area Distributed Database

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.

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 already 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. 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 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.

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, 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. See the following figure.

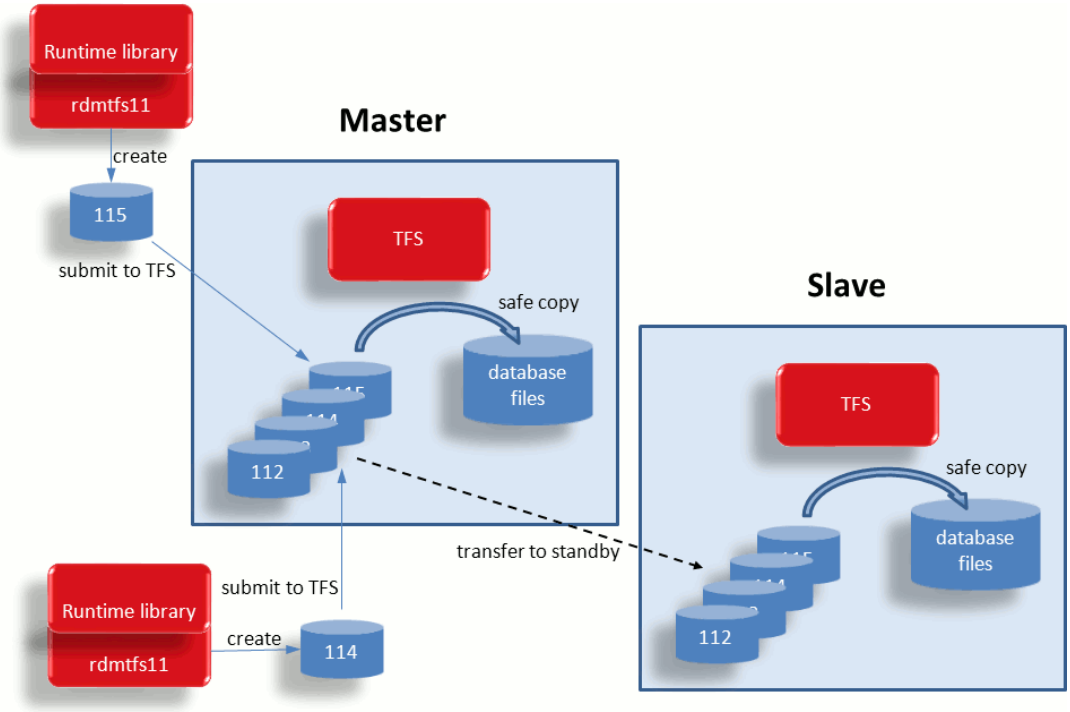

Figure 9: Mirroring by Copying Transaction Logs

5. 2 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, as discussed further below.

With action-for-action replication, changes in an RDM database are captured as a series of create, delete, update, connect, etc., operations. The runtime library creates this replication log when configured to do so. Each one contains a complete transaction's-worth of actions. 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, 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 below illustrates the replication process.

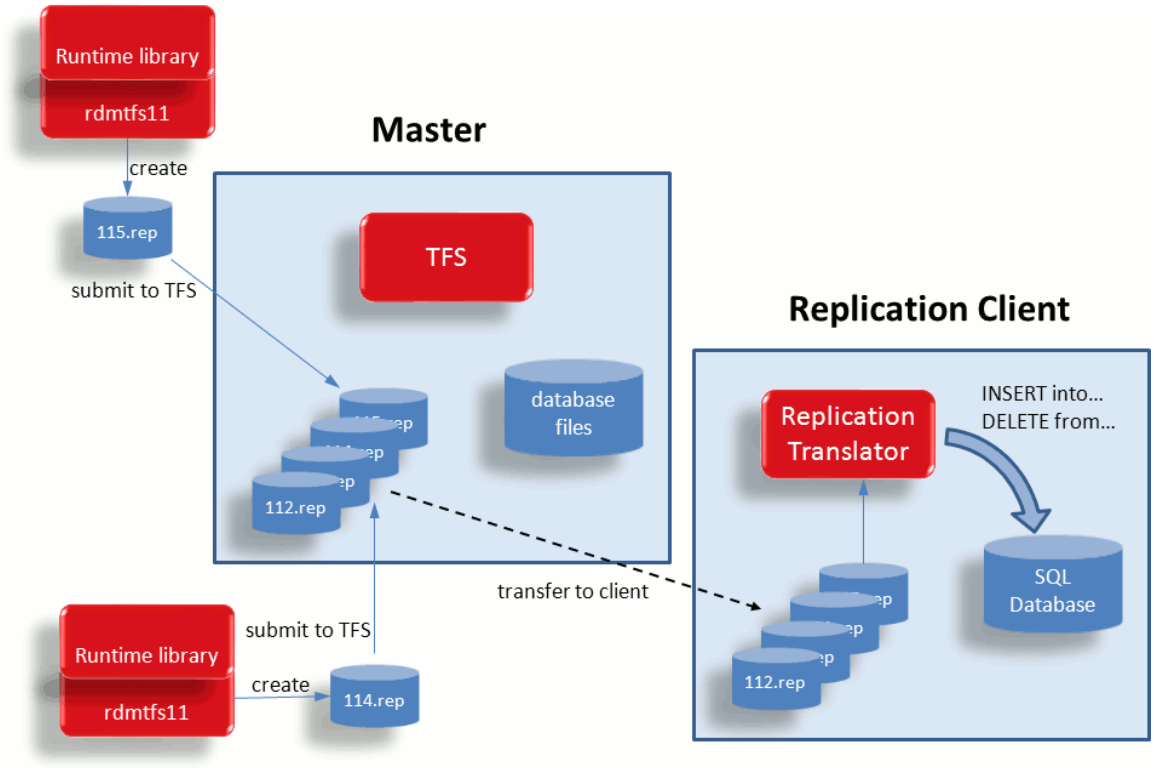

Figure 10: Replication using Action Logs

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 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.

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.

See the figure below for the general configuration of active/standby.

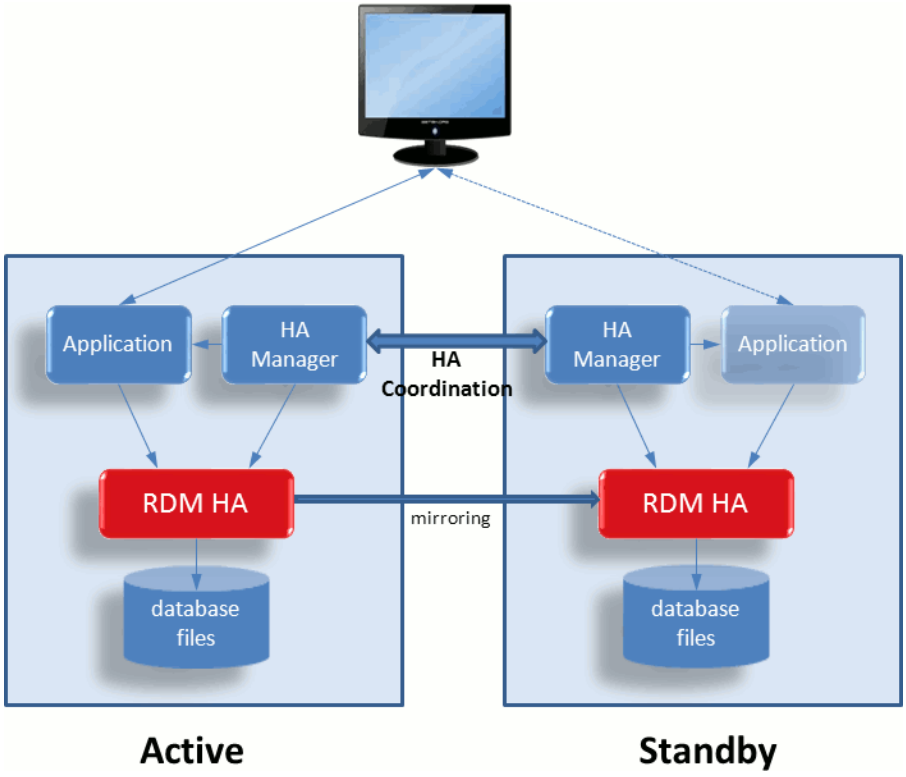

Figure 11: High Availability Configuration - Active/Standby

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 trick, 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. 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.

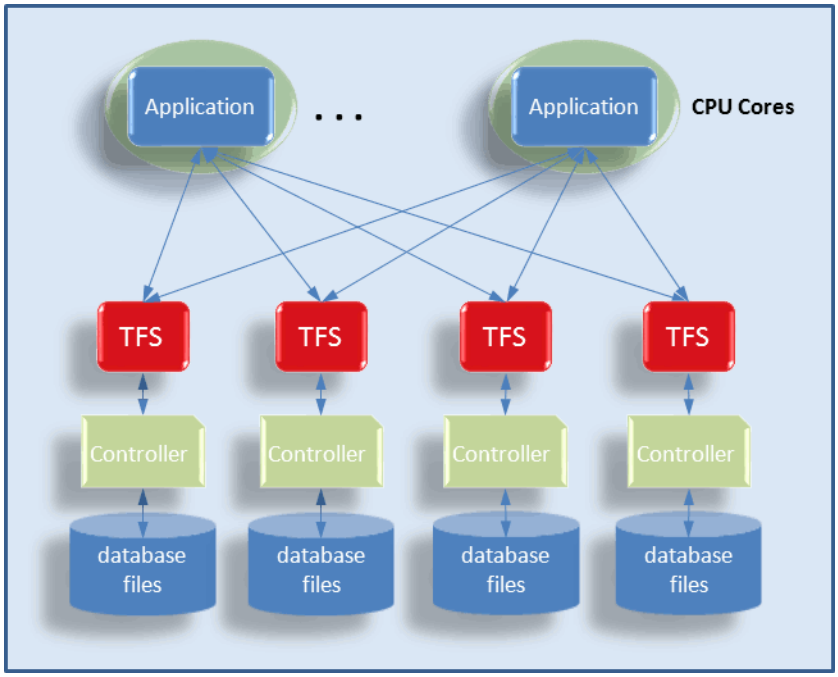

Figure 12: Scaling Up through Parallelism

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 Distributed Package's 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.

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.

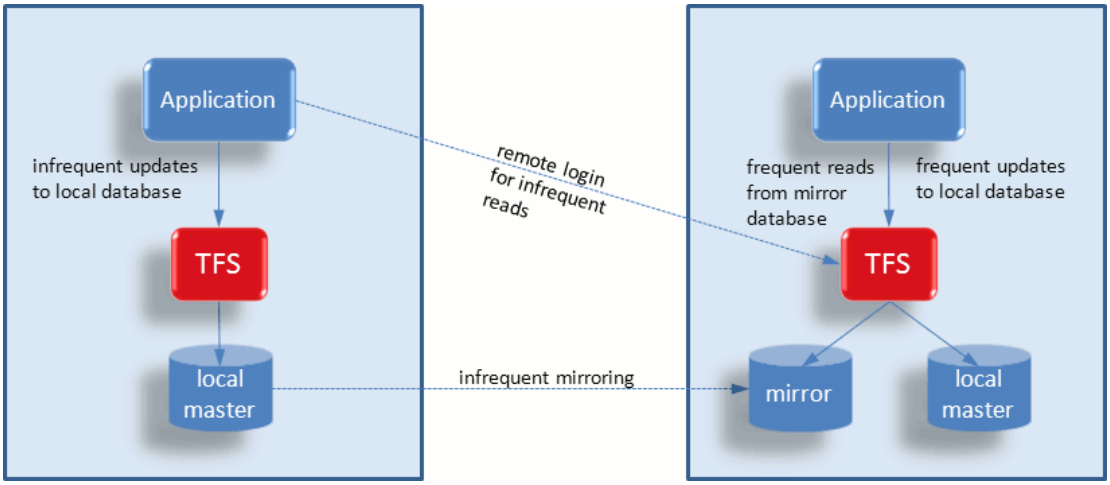

Figure 13: Mirroring or Remote Login

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 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. A 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:

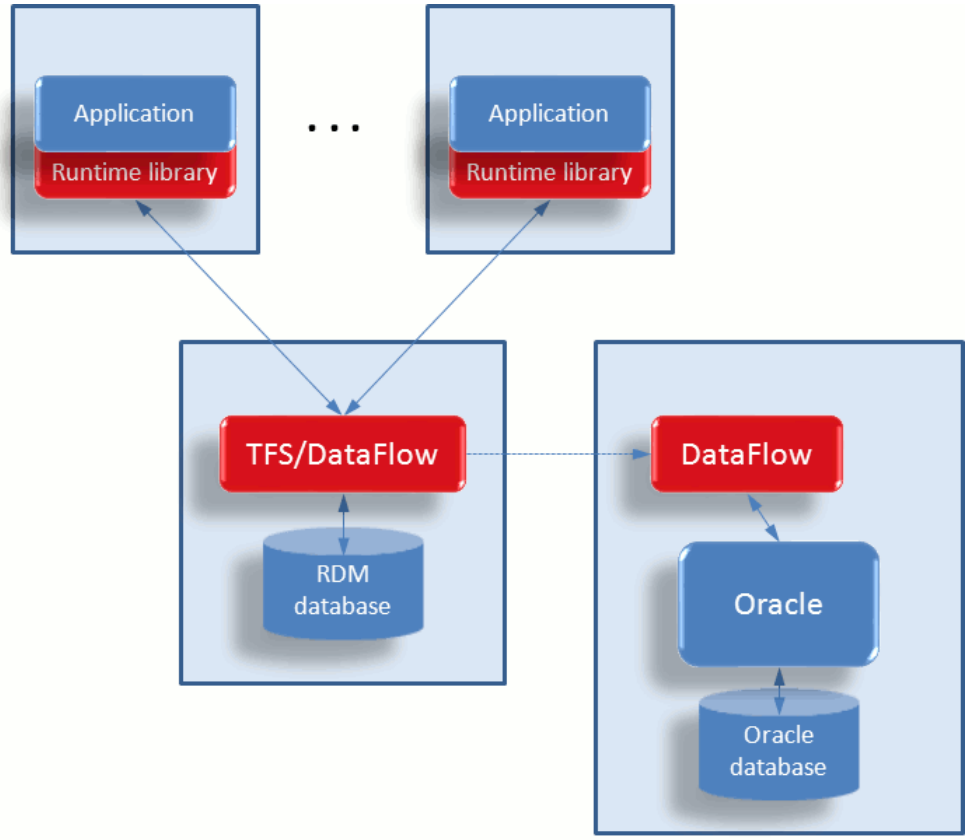

Figure 14: Data Entry Application System

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. 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.

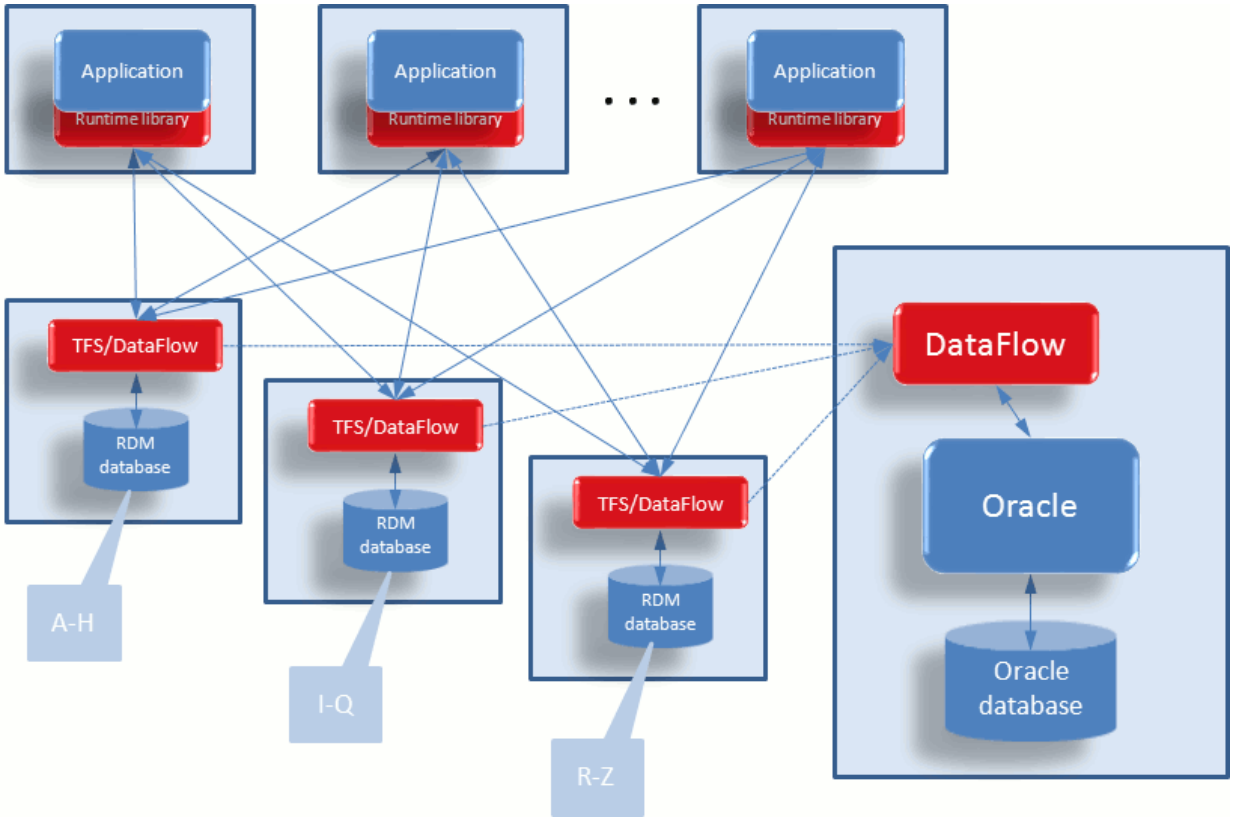

Figure 15: Partitioned Data Entry Application System

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 the powerful embedded computers requires order that can be realized with software. RDM can place pieces of database software at multiple levels so that information is shared in a timely way with parts of the system that need it.
Package:	RDM Plus

Consider a (imaginary) system installed in buildings where each elevator door counts people entering or exiting on every floor. Additional controllers measure power consumption on various circuits on each floor, and all are connected to a central computer in the building. The building is connected to a SCADA system at headquarters where several buildings are managed. If the power consumption controllers also had control over thermostats, HVAC systems and blowers, then there is enough information and control to administrate, optimize and report on energy use and costs.

An architecture for this system would include head count sensors at the elevator doors, embedded computers in place of all thermostats, and a central computer connected to all of these devices with a dashboard allowing a building operator to view building activity and issue come controls. The building computer would then be connected to headquarters through internet, where it passes information to/from the SCADA at headquarters.

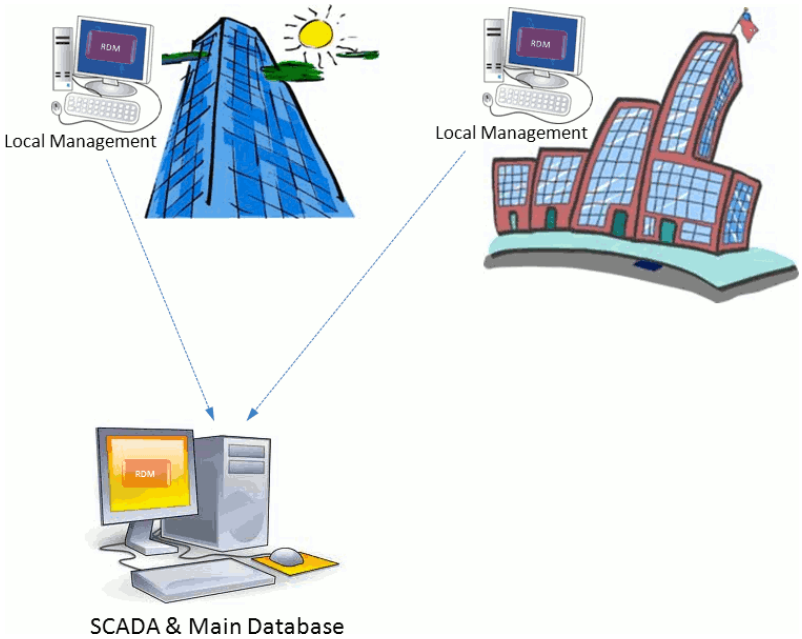

Figure 16: Data Flow and Building Power Management

RDM applications will be running within the embedded thermostat units, the building computers and the SCADA system. From the embedded computers to the building computer, data will be replicated. In the other direction, data will be mirrored.

The data mirrored to the embedded computer will contain operational parameters that are managed and set at the building level. Between a building computer and the SCADA, replication will be used to pass summarized building data to the SCADA, and control information from the SCADA will be passed to the building computer through mirroring.

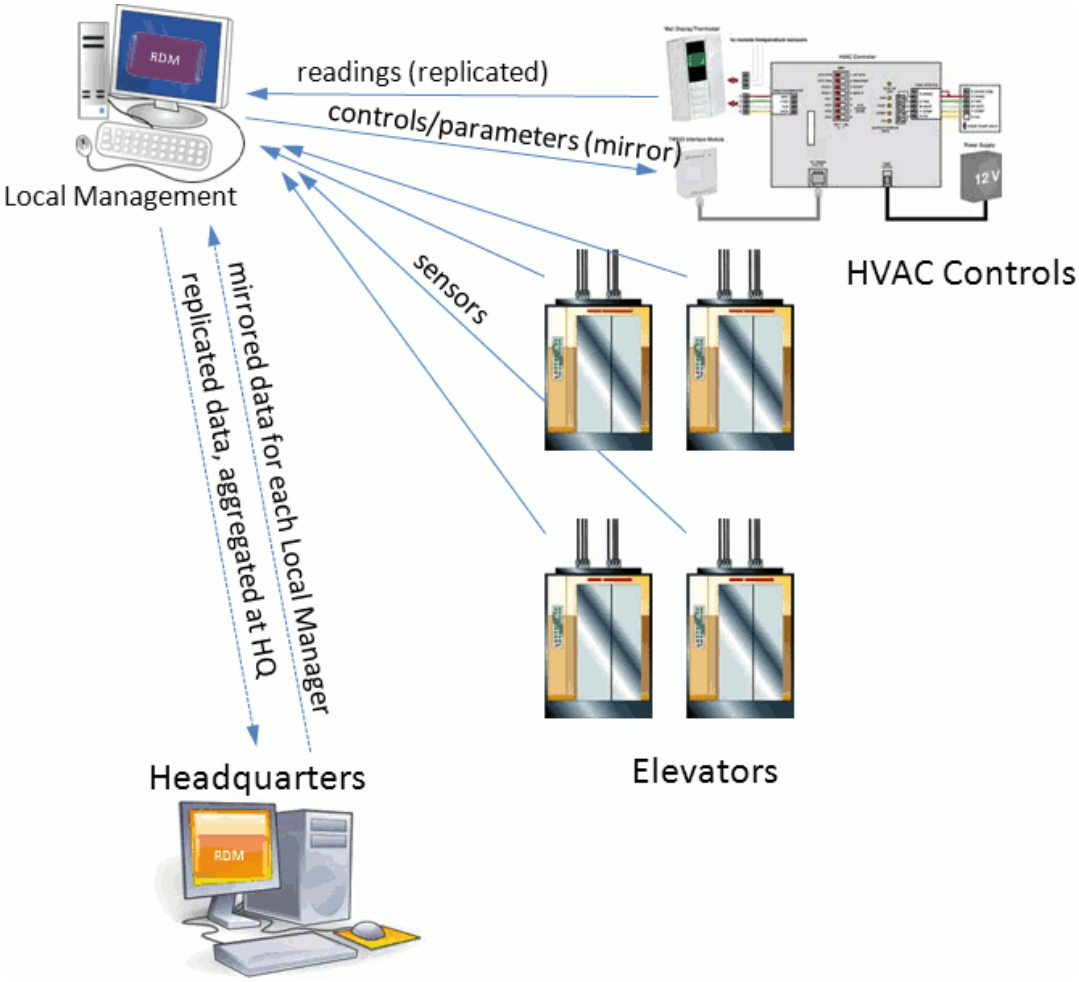

Figure 17: Details of Local Building Management

7. CONCLUSION

The goal of this paper was to provide a technical description of the RDM 11.0 product, sufficient for an evaluation and decision-make 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 product. 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 have also benefitted 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