Raima Database Manager In-memory Database Engine

By Jeffrey R. Parsons, Chief Engineer – September 2020

Abstract

Raima Database Manager (RDM) contains an all new data storage engine optimized specifically for working with memory resident data sets. This new In-memory Database Engine (IMDB) allows for significant performance gains and a reduction in processing requirements compared to the On-disk-engine. The RDM IMDB runs alongside the RDM On-disk engine and databases can be opened with either one. This document describes the use, strengths, and limitations of the RDM IMDB to allow a developer to know when it is appropriate for it to be used.

Table of Content

Using The RDM IMDB

The RDM IMDB is designed to be very simple to use. There is no need for proprietary keywords in the database schema to utilize the IMDB instead of the on-disk engine. Instead, the developer sets an option prior to opening the database to configure and use the IMDB. This allows the same database to be used with either the on-disk engine or the IMDB

MDB Configuration Options

When a database is opened there are several options available to configure the RDM storage engine. If no option value is given RDM defaults to the on-disk engine. The different configurations inform the engine what actions to perform when the database is first opened and when the database is closed by the last client to have it open. All clients that have the database open simultaneously must use the same storage configuration options. In addition, it is required that clients simultaneously opening a database use an identical schema (or no schema at all). The following storage configuration options are supported

On-disk

When the first client opens a database with no storage configuration option set, or the storage configuration option set to “ondisk” the on-disk engine will be used.

In-memory Volatile

When the first client opens a database with the storage configuration option set to “inmemory_volatile,” the newly created database is considered to be “throw away.” Database content will only be saved by programmatically calling the rdm_dbPersistInMemory API. When the last client closes the database any changes since the last call to rdm_dbPersistInMemory will be discarded. If no calls were made to rdm_dbPersistInMemory then the entire database is discarded. The RDM Transaction File Server (TFS) maintains an open count to determine when a database is first opened and when it is no longer in use by any clients. More information on the RDM TFS can be found here.

In-memory Load

When the first client opens a database with the storage configuration option set to “inmemory_load,” the RDM TFS will look for an on-disk idindex and pack file. If they are found the in-memory database will be loaded with the current contents of the on-disk files. When the last client closes the database any updates will not automatically be written to the on-disk files. Changes can be saved programmatically using the rdm_dbPersistInMemory API.

In-memory Keep

When the first client opens a database with the storage configuration option set to “inmemory_keep,” it will be opened without any rows in any tables. When the last client closes the database any rows that have been added will be written to on-disk files. The entire contents of any existing on-disk files will be replaced by the in-memory contents. This is also true if the rdm_dbPersistInMemory API is called.

In-memory Persist

When the first client opens a database with the storage configuration option set to “inmemory_persist.” the TFS will look for an on-disk-idindex and pack file. If they are found the in-memory database will be loaded with the current contents of the on-disk files. When the last client closes the database any rows that have been modified will be written to the on-disk files. At any time, changes can be persisted programmatically using the rdm_dbPersistInMemory API, in this case only changes made since the last time the database was persisted will need to be written to disk.

Opening a database

Configuration options are specified to RDM using key/value pairs. While the different RDM API sets each have a function to specify a configuration key/value pair the pairs used for configuration are consistent across the APIs. The key used for specifying the storage engine configuration when opening a database is called “storage”. If the database is not open on the hosting TFS, it will be opened with the specified storage configuration value. If the database is already opened on the hosting TFS then the storage configuration value specified must match the storage configuration currently in use. A client that requests to open a database using a storage configuration value other than the one currently in use will get an error code.

The follow storage values are supported

Value	Setting	Meaning
ondisk	“storage=ondisk”	Use the disk-based engine
Inmemory_volatile	“storage=inmemory_volatile”	Use the in-memory engine. The database is empty when the first client opens it and all contents are discarded when the last client closes it.
Inmemory_load	“storage=inmemory_load”	Use the in-memory engine. The database contents are loaded from disk when the first client opens it and contents are not automatically saved when the last client closes it.
Inmemory_keep	“storage=inmemory_save”	Use the in-memory engine. The database is empty when the first client opens it and any rows in the database will be written to disk when the last client closes it. Any existing disk-based files will be overwritten by the contents of the in-memory database.
Inmemory_persist	“storage=inmemory_persist”	Use the in-memory engine. The database contents are loaded from disk when the first client opens it and contents are automatically saved when the last client closes it.

Core Cursor API

The RDM Core Cursor API uses the rdm_dbSetOption API to set the storage engine options. This option must be set prior to opening the database.

```
RDM_RETCODE coreOpenDbInMemory(
RDM_TFS *pTfs,  RDM_DB *pDb)
{
    RDM_RETCODE rc;
    RDM_TFS tfs = NULL;
    RDM_DB db = NULL;

    rc = rdm_rdmAllocTFS("", &tfs);
    if (rc == sOKAY)
    {
      rc = rdm_tfsAllocDatabase(tfs, &db);
     if (rc == sOKAY)
    {
        rc = rdm_dbSetOptions(db, "storage=inmemory_volatile");
        if (rc == sOKAY)
        {
          rc = rdm_dbOpen(db, "testdb", RDM_OPEN_SHARED);
        }
    }
}
if (rc != sOKAY)
{
    rdm_dbFree(db); rdm_tfsFree(tfs);
}
 else
{
     *pTfs = tfs;
    *Db = db;
}

 return rc;
}
```


SQL API

The RDM SQL API uses the rdm_sqlSetOptions API to set the storage engine options. This option must be set prior to opening the database.

```
RDM_RETCODE sqlOpenDbInMemory(
RDM_TFS *pTfs, RDM_SQL *pSql)


{
RDM_RETCODE rc; RDM_TFS tfs = NULL; RDM_SQL sql = NULL;
 rc = rdm_rdmAllocTFS("", &tfs); if (rc == sOKAY)
{

    rc = rdm_tfsAllocSql(tfs, &sql); if (rc == sOKAY)
    {

        rc = rdm_sqlSetOptions(sql, "storage=inmemory_volatile"); if (rc == sOKAY)
        {

            rc = rdm_sqlOpenDatabase(sql, "testdb", RDM_OPEN_SHARED);
        }
    }
}
 if (rc != sOKAY)
{


    rdm_sqlFree(sql); rdm_tfsFree(tfs);
}

 else
{

    *pTfs = tfs;
    *pSql = sql;
}


 return rc;
}
```


ODBC

The ODBC API uses the SQLSetConnectAttr API to set connection handle attributes. The

SQL_ATTR_RDM_SQL_OPTIONS attribute can set the inmemory mode using the “inmemory=” option string.

```
SQLRETURN odbcOpenDbInMemory(
SQLHENV *phEnv, SQLHDBC *phCon)
{
SQLRETURN ret;

 SQLHENV hEnv = SQL_NULL_HENV;
 SQLHDBC hCon = SQL_NULL_HDBC;

    ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hEnv);
    if (ret == SQL_SUCCESS)
   {
       ret = SQLSetEnvAttr(hEnv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER)SQL_OV_ODBC3, 0);
       if (ret == SQL_SUCCESS)
       {
           ret = SQLAllocHandle(SQL_HANDLE_DBC, hEnv, &hCon);
           if (ret == SQL_SUCCESS)
           {
               ret = SQLSetConnectAttr(hCon, SQL_ATTR_CURRENT_CATALOG, “testdb”, SQL_NTS);
               if (ret == SQL_SUCCESS)
               {
                                   ret = SQLSetConnectAttr(hCon, SQL_ATTR_RDM_SQL_OPTIONS, “storage=inmemory_persistent”, SQL_NTS);

                  if (ret == SQL_SUCCESS)
                  {
                                           ret = SQLConnect(hCon, SQL_EMPSTR, SQL_NTS, SQL_EMPSTR, SQL_NTS, SQL_EMPSTR, SQL_NTS);

                                   }
               }
           }
       }
   }
 if (ret != SQL_SUCCESS)
{

     SQLFreeHandle(SQL_HANDLE_DBC, hCon); SQLFreeHandle(SQL_HANDLE_ENV, hEnv);
}
 else
{
     *phCon = hCon;
     *phEnc = hEnv;
}

 return ret;
}
```


On-Disk Storage Format VS. IMDB Storage Format

On-disk Storage

Physically, a disk-based RDM database consists of a set of files stored together in a subdirectory of the RDM

Transaction File Server (TFS) document root. The subdirectory is named after the database with the suffix ".rdm".

The types of files contained in the database directory are

- One catalog file (c00000000.cat) containing database meta information in JSON format
- One or more pack files (p00000000.pack) containing database objects
- One or more idindex files (i00000000.idindex) containing an index with the size and location of database objects in the pack file(s).
- One or more journal files (j00000000.journal) containing a journal of the updates to idindex
- One or more transient temporary files used for transaction processing

Logically, an RDM database is organized into a series of drawers. A drawer is a logical grouping of similar types of database objects. Each table will have a drawer containing all of the row objects for that table. There will be a separate drawer for each table defined in the database. A drawer will only have one type of object stored in it and each database object will be indexed by an object id value. The types of database objects include

- Rows
- B-tree Nodes
- Hash buckets
- References
- Variable data chunks (binary, char, or widechar)

A drawer has entries stored in the idindex file(s) to identify where drawer objects are located in the pack file(s). When a particular database object is requested, the engine will ask the TFS to lookup the object based on the drawer and object id. The TFS will search through the idindex entries for the specified drawer until it finds the id of the requested object. The idindex entry contains both the offset and size of the object within the pack file. The TFS will read the object's data out of the pack file and send it back to the runtime for decoding. When an object is updated. the data will first be written to the pack file in a contiguous location. The journal and index cache will be updated with the object's new location and size. The actual idindex entry in the idindex file will be updated later. If there is a crash, the idindex will be updated during recovery processing. The pack, idindex, journal concept is a very efficient method for storing database objects on disk. It has been designed and tuned specifically to allow large data sets to be quickly located and efficiently updated. When

performing journaling, the TFS does not need to write multiple copies of the data - it simply needs to journal the new

location of an object. Most new objects will be written to the end of a pack file, and the internal algorithms for re-using space prioritize putting multiple updates into the same block. However, when dealing with raw in-memory data sets, all of this design and implementation that make this approach efficient for disk-based systems are unnecessary overhead.

In-memory objects

Perhaps the most critical difference between in-memory databases and disk-based databases is that there is no need for objects to be read into a cache from disk when in-memory objects are accessed. All of the blocking, indexing, packing, and journaling required for persistent on-disk tables is unnecessary for the in-memory counterparts. In fact, while the logical view of the IMDB is similar to that of the traditional disk-based engine, the implementation only has a catalog and an idindex. The pack and journal are only needed for in-memory databases that will be persisted to disk. To access a database object, the runtime will ask the TFS to look it up based on the drawer and object id. The TFS will search the idindex of the specified drawer for the requested object. Once it locates the idindex entry for the row, it simply needs to send the information back to the runtime. If the runtime is in the same memory space as the TFS, it only needs to send a pointer to the object data. If the runtime is in a separate memory space, it will send a copy of the object data. Updating an object is more efficient using the in-memory engine. The in-memory engine does not need to keep track of blocks to reuse, but will update the existing memory buffer or reallocate the memory buffer if the size of the object has increased. As the in-memory engine utilizes the RDM memory manager to optimize the way memory is allocated and tracked, the reallocations typically will not require access to operating system resources.

Rows

Rows are organized, identified, and tied together through a virtual key on the object id. The object id for a row object is referred to as a rowid. The RDM Transaction File Server (TFS) maintains an index in the idindex file on the rowid for each table. This index allows the TFS to locate the row information and return it to the runtime engine for processing.

Packed Row Format

To reduce the amount of data read/written to disk, the standard disk-based RDM storage engine uses a “packed” row format. This format packs a row into a byte stream that is both compact and portable across different processor families. In order to convert the row to this pack format, the storage engine does a number of data transformations

- Integer data is converted into a variable integer format. The variable integer format is byte-order agnostic and is sized based on the value stored and not the column type. The

integer value 10 will be stored in the same number of bytes regardless of whether the column is an INT, SMALLINT, or BIGINT.
- Floating point types are converted to network byte-order
- Character strings are all stored as UTF-8 and the terminating NULL character (and all bytes after the NULL) are not stored. Instead, a character column will be stored with a variable integer encoded byte-length followed by the appropriate UTF-8 characters
- Binary types are stored with a variable integer encoded byte-length followed by the column data
- All row header data is stored using the variable integer format.

AVL Data	Reference Data	Column Data

Table 1: RDM Packed Row Format

AVL Data

The AVL data consists of

- The varuint (unsigned variable integer) encoded number of AVL entries in the row
- For each AVL entry in the row there is the following information
    - A varuint encoded Key ID identifying the AVL
    - A varuint encoded RowID identifying the left row of the AVL
    - A varuint encoded RowID identifying the right row of the AVL
    - A varuint encode flag value

More detailed information can be found in the section describing the RDM AVL

Reference Data

The Reference Data consists of

- The varuint encoded number of references in the row
- For each reference there is the following information
    - A varuint encoded Reference ID identifying the reference
    - A varuint encoded RowID identifying the referenced (primary) row

Column Data

The Column Data consists of a one or more column groups. Each group contains information about contiguous column IDs. The column groups are used so that we do not need to store information about columns that have NULL values. If a column is NULL, we will not put data into the row. In addition, if a column is dropped, we do not need to change the rows stored on disk. The dropped columns will simply be ignored when the column is unpacked.

- The varuint encoded number of columns in the group
- The varuint encode starting columns ID for the group
- For each column in the group
    - The encoded column value

The Packed row may also be compressed and encrypted.

Expanded Row Format

In many aspects, the Packed row format is not appropriate for an in-memory database. The encoding/decoding required to use rows that are in the packed format is significant for in-memory data sets. Unless the database is run in a memory constrained environment, or is shared between processors using different byte order, it is preferred to use the Expanded row format which matches the internal format the runtime uses.

Row Header	Row Data

Table 2: RDM Expanded Row Format

The Row Header consists of

- An unsigned 32-bit integer that specified the size of the Row Data

The Row Data consists of

- An array of RDM_AVL structures (One for each AVL index defined in the schema)
- An array of RDM_JOIN structures (one for each table the row references)
- An array of VAL_STATUS values (one for each active column in the row)
- A byte array of column data based on the table structure defined by the schema

The expanded row format is ideal of an in-memory database because it requires virtually no processing as the row is moved from the TFS to the runtime. In fact, in cases where the TFS and runtime are running in the same memory space, it is possible for the runtime to directly read the

memory from the TFS. The runtime will only need to make a copy of the row if it needs to do an update.

Design Considerations in RDM

While the reason for Raima to create an all in-memory storage engine was raw performance, there were several key design considerations used to drive the product. Those considerations were:

- Make it easy to use.
- Make it efficient

Ease of Use

There are many approaches to provide ease of use. To make the RDM IMDB easy to use, the decision on whether to use the disk or in-memory engine is made at runtime instead of during compilation or development. Other than setting a mode prior to opening a database, the same development interfaces and APIs are used regardless of whether the database is stored on disk or in-memory. Internal to the engine, however, things are very different as the storage format, logging requirements, data structures, and algorithms will all be tailored to an efficient in-memory implementation.

Efficiency

A disk based database assumes reading and writing data stored on a slow storage media (relative to main memory). A disk-based engine spends a lot of resources reducing the amount of data that needs to be written and optimizing the format of that data to be accessed efficiently by a disk. When the need to physically access the slower storage media is removed, the engine must be efficient enough to take advantage of this in order to provide better performance. The RDM IMDB gains this efficiency by

- Using a database object location implementation optimized for memory resident data
- Using a storage format specific for memory resident data sets
- Using a database object storage format that matches the format used internally by the engine
- Providing optional indexing algorithms optimized for in-memory data sets
- Providing alternative implementations for generic indexing algorithms optimized for in-memory data sets
- Journaling table changes only on demand

Designing Tables for Using The RDM IMDB

Indexing

B-Tree

The default indexing method for RDM is the b-tree. A b-tree is a self-balancing tree structure that keeps data ordered to allow for both searching and sequential access. Insertions, deletions, updates, and lookups can be done in a logarithmic scale. A b-tree is a good solution for on-disk databases because the width of the b-tree nodes can keep the depth of the node tree smaller than a self-balancing binary tree. This results in fewer disk access required for searching and updates to the tree.

The RDM b-tree implementation is a non-clustered external tree sorted on the column values specified in the index definition. There is an entry in a b-tree node for each row in the table the index is derived from. Nodes are identified by their object id, referred to as a node id for node objects. This allows the implementation to be efficient for databases using either the on-disk engine or the IMDB. In both engines references to a node id will be looked up via the id index for the drawer the b-tree is stored in. In the on-disk engine, the id-index will contain an offset and size of the node in the pack file. In the IMDB, the id index will contain a pointer to the node. The RDM b-tree implementation uses a node size of 32-items; however, only the items that are in-use will be stored on the TFS. RDM b-tree nodes are never updated; instead, a new node is created and the id index is updated to point to the location of the new node. The old node will be available for re-use in later transactions.

Hash

The hash implementation in RDM has been updated to use an extendible hashing algorithm. The extendible hashing algorithm does not require the developer to guess the cardinality of an index during design time. This is important for the in-memory storage engine as there is no need to reserve memory for a set of buckets that may never be used. Instead, as the number of buckets in the hash grow the directory size will increase. The current implementation only allows for unique keys and does order data based on the keyed columns (data is organized based on the hash of those columns). This means that a hash can be used for lookups, but cannot be used for sequential access or ranges.

AVL

RDM 14 has added an index algorithm specifically for use by the In-memory Storage Engine called an AVL tree. An AVL is a self-balancing binary tree that in RDM is implemented internal to a row instead of externally. There is no data duplication in the AVL as, unlike in a B-tree, external nodes containing copies of indexed columns are not maintained. An AVL is a binary tree, meaning the depth of the

tree will be much larger than that of a B-tree. For this reason, the AVL index is better suited for the In-memory Storage Engine using the expanded row format than the Disk-based Engine using the packed row format. The packed row format does contain an implementation for the AVL index, but this is included primarily for persisting an in-memory image to disk and not intended for general use in diskbased tables.

An AVL can be used for any operations that a b-tree index would be used for. The AVL supports lookups, ranges, scanning, and both duplicate or unique constraints. The SQL optimizer will utilize an AVL in the same manner as a Btree, but will use a slightly different weight based on the implementation differences between a B-tree and an AVL.

R-Tree

RDM 14 has also added an index algorithm designed specifically for geospatial data called an R-Tree. It is a balanced search tree that organizes its data into pages and is designed to group nearby objects and then represent that in the next level of the tree. This allows for quick retrieval of multi-dimensional data in a bounding box.

A Brief History of RDM Data Storage

This section gives a brief description of the history of the RDM storage engine.

Original Engine Design

RDM was originally released in 1984. At the time of the original design, processors were single core (and slow), disk drives were expensive (and slow), and main memory was even more expensive. The design of the RDM storage engine was dictated by those hardware limitations, and data stored in the engine would stay on disk until it was actively requested and only then would it be loaded into main memory. In most cases a record (row) would stay in main memory only a short while before something else was requested and the record was sent back to disk. Optimization techniques were designed around a very limited amount of main memory and a single core processor.

A Configurable Cache

As time went on and Moore's law continued to hold, the cost of memory began to drop, and the quantity available to the RDM engine began to rise. The first attempt to take advantage of all this memory was to allow the RDM cache to have a configurable size. A developer could make estimates on how large his database would be and request a cache large enough to hold most, if not all, of the data in the runtime cache. The data would be read from disk the first time it was requested and was not in danger of being swapped out to disk as other data was read. This had performance benefits, but just for a rather limited set of use cases (primarily single-user read intensive applications). As the underlying engine was still disk-based, all modifications required data to be written to disk, often

multiple times if logging was enabled. In addition, other clients could update the database and invalidate a portion (or all) of the now large cache requiring that it be retrieved, once again, from disk.

An Emulated Filesystem

The most expensive operations, when dealing with a disk-based database, is performing a flush where data is written to the physical media bypassing any filesystem or hardware cache. In order to have ACID compliant transactions, a database must periodically perform a flush on one (or more) of the database files. If a database does not require any flush operations. the performance will increase. In order to have a solution that takes advantage of a lot of available memory. Raima implemented a virtual file system for RDM. This file system was completely in-memory, so unlike the Configurable Cache, there were performance benefits for both reading and writing. In many cases, This sacrifice in a transactions durability could results in an improvement in throughput of an order of magnitude.

A Need for Something New

While the Emulated Filesystem implementation provided improved performance for a much larger subset of applications. it did not reach the level of performance possible with a storage engine designed from the ground up to support in-memory tables. For example, RDM also allows the developer to run disk-based but avoid all file flush operations. Several use cases could use this mode of operation and perform as well as, or even better than, a database that was using the virtual in-memory file system implementation. Something was needed that would be designed purely for memory-resident datasets. This new engine would need to store data differently and use data structures and algorithms allowing data to be processed much more efficiently than the disk-based engine.

Limitations

Dynamic DDL

Dynamically changing the schema is not supported for databases opened in-memory in RDM Version 14. It is planned to add this ability in a future update

Data Interoperability

Data in the in-memory engine is stored in a platform specific format. All clients accessing a database using the IMDB must use the same alignment, byte-order, and native wide-character format (if the database has wide-character columns).

Persistence

When using the RDM IMDB, transactions are not to be considered ACID compliant. In order to enhance throughput, the “D” (durability) component of the transaction is sacrificed. The other ACID

properties are maintained, but changes to a database running in-memory are not journaled. It is possible to persist the state of an in-memory database to ondisk files so that the contents can be loaded the next time a database is opened.

While individual transactions that utilize the RDM IMDB are not ACID compliant, the process of persisting an inmemory databased to disk is. When a database is persisted, the RDM IMDB goes through all database objects that have been inserted/deleted/modified since the last time the database was persisted and writes them to the pack file. The on-disk idindex file is updated with the objects location and size through the journaling process. Once the changes are in the journal, they are guaranteed to be visible the next time the database is opened.

32-bit support

While the RDM IMDB is supported on both 32-bit and 64-bit systems, the memory limitations of 32-bit systems result in a much smaller potential dataset

Summary

The RDM In-Memory Storage Engine provides the ability to work with databases that are optimized to hold the entire data set in-memory. The database organization and format are specifically architected to be efficient for in-memory use in order to provide performance beyond what is capable in a disk-based engine. The individual transactions in a database opened with the in-memory engine are not persistent, but the database as a whole (or just the deltas) can be persisted on demand or automatically at close. Using the RDM IMDB only requires setting an option prior to opening the database, once a database is opened in-memory all clients that use that database must use the same options.