In-Memory vs. RAM-Disk Database Systems: A Linux-based Benchmark

Abstract: It stands to reason that accessing data from memory will be faster than from physical media. A new type of database management system, the in-memory database system (IMDS), claims breakthrough performance and availability via memory-only processing. But doesn't database caching achieve the same result? And if complete elimination of disk access is the goal, why not deploy a traditional database on a RAM-disk, which creates a file system in memory?

This paper tests the *eXtreme*DB® In-Memory Database System against the db.linux embedded database in both traditional (disk-based) and RAM-disk modes, running on Red Hat Linux 6.2. Deployment in RAM boosts db.linux's performance by as much as 74 percent. But even then, the traditional database lags the IMDS. Fundamental architectural differences explain the disparity. Overhead hard-wired into disk-based databases includes data transfer and duplication, unneeded recovery functions and, ironically, caching logic intended to avoid disk access.

McObject LLC
33309 1st Way South
Suite A-208
Federal Way, WA 98003

Phone: 425-888-8505
E-mail: info@mcobject.com
www.mcobject.com

Introduction

It makes sense that maintaining data in memory, rather than retrieving it from disk, will improve application performance. After all, disk access is one of the few *mechanical* (as opposed to electronic) functions integral to processing, and suffers from the slowness of moving parts. On the software side, disk access also involves a "system call" that is relatively expensive in terms of performance. The desire to improve performance by avoiding disk access is the fundamental rationale for database management system (DBMS) caching and file system caching methods.

This concept has been extended recently with a new type of DBMS, designed to reside entirely in memory. Proponents of these in-memory database systems (IMDSs) point to groundbreaking improvements in database speed and availability, and claim the technology represents an important step forward in both data management and real-time systems.

But this begs a seemingly obvious question: since caching is available, why not extend it to cache entire databases to realize desired performance gains? In addition, RAM-drive utilities exist to create file systems in memory. Deploying a traditional database on such a RAM-disk eliminates physical disk access entirely. Shouldn't its performance equal that of an in-memory database?

This white paper tests the theory. Two nearly identical database structures and applications are developed to measure performance in reading and writing 30,000 records. The main difference is that one database, *eXtreme*DB, is an IMDS, and the other, db.linux, is designed for disk storage. The result: while RAM-drive deployment makes the disk-based database significantly faster, it cannot approach the in-memory database performance. The sections below present the comparison and explain how caching, data transfer and other overhead sources inherent in a disk-based database (even on a RAM-drive) cause the performance disparity.

The Emergence of In-Memory Database Systems

In-memory database systems are relative newcomers to database management. The technology first arose to enhance business application performance and to cache Web commerce sites for handling peak traffic. In keeping with this enterprise focus, the initial IMDSs were similar to conventional SQL/relational databases, stripped of certain functionality and stored entirely in main memory.

Another new focus for database technology is embedded systems development. Increasingly, developers of network switches and routers, set-top boxes, consumer electronics and other hardware devices turn to commercial databases to support new features. In-memory databases have emerged to serve this market segment, delivering the required real-time performance along with additional benefits such as exceptional frugality in RAM and CPU resource consumption, and tight integration with embedded

systems developers' preferred third-generation programming languages (C/C++ and Java).

The Comparison: *eXtreme*DB vs. db.linux

McObject's *eXtreme*DB is the first in-memory database created for the embedded systems market. This DBMS is similar to disk-based embedded databases, such as db.linux, BerkleyDB, Empress, C-tree and others, in that all are intended for use by application developers to provide database management functionality from within an application. They are "embedded" in the application, as opposed to being a separately administered server like Microsoft SQL Server, DB2 or Oracle. Each also has a relatively small footprint when compared to enterprise class databases, and offers a navigational API for precise control over database operations.

This paper compares *eXtreme*DB to db.linux, a disk-based embedded database. The open source db.linux DBMS was chosen because of its longevity (first released in 1986 under the name db_VISTA) and wide usage. *eXtreme*DB and db.linux also have similar database definition languages.

The tests were performed on a PC running Red Hat Linux 6.2, with a 400Mhz Intel Celeron processor and 128 megabytes of RAM.

Database Design

The following simple database schema was developed to compare the two databases' performance writing 30,000 similar objects to a database and reading them back via a key.

```
/*************************************************************
 *                                                           *
 * Copyright(c) 2001 McObject,LLC. All Right Reserved.       *
 *                                                           *
 *************************************************************/


#define int1       signed<1>
#define int2       signed<2>
#define int4       signed<4>
#define uint4      unsigned<4>
#define uint2      unsigned<2>
#define uint1      unsigned<1>


declare database mcs[1000000];

struct stuff {
   int2 a;
};
/*
 *
 */
class Measure {
   uint4 sensor_id;
   uint4 timestamp;
   string spectra;

   stuff thing;

   tree <sensor_id, timestamp> sensors;

};
```

Figure 1. *eXtreme*DB schema

```
/*********************************************************
 *                                                       *
 * Copyright(c) 2001 McObject,LLC. All Right Reserved.   *
 *                                                       *
 *********************************************************/


struct stuff {
   short a;
};

database mcs [8192]
{
   data file "mcs.dat" contains Measure;
   key  file "mcs.key" contains sensors;

   record Measure
   {
      long sensor_id;
      long m_timestamp;
      char spectra[1000];

      struct stuff thing;

      compound key sensors {
         sensor_id;
         m_timestamp;
      }
   }
}
```

Figure 2. db.linux schema

The only meaningful difference between the two schemas is the field 'spectra'. In the case of *eXtreme*DB it is defined as a 'string' type whereas with db.linux it is defined as char[1000]. The db.linux implementation will consume 1000 bytes for the spectra field, regardless of how many bytes are actually stored in the field. In *eXtreme*DB a string is a variable length field. db.linux does not have a direct corollary to the *eXtreme*DB string type, though there is a technique to use db.linux network model sets to emulate variable length fields with varying degrees of granularity (trading performance for space efficiency). Doing so, however, would have caused significant differences in the two sets of implementation code, making it more difficult to perform a side-by-side comparison. *eXtreme*DB has a fixed length character data type; however, the variable length field was used for purposes of comparison, because it is the data type explicitly designed for this task.

(An interesting exercise for the reader may be to alter the *eXtreme*DB implementation to use a char[1000] type for spectra, and to alter the db.linux implementation to employ the variable length field implementation. The pseudo-code for implementing this is shown in Appendix A).

Benchmark Application

The first half of the test application populates the database with 30,000 instances of the 'Measure' class/record.

The *eXtreme*DB implementation allocates memory for the database, allocates memory for randomized strings, opens the database, and establishes a connection to it.

```
void   *start_mem = malloc( DBSIZE );

if ( !start_mem ) {
   printf( "\nToo bad ..." );
   exit( 1 );
}

make_strings();

rc = mco_db_open( dbName, mcs_get_dictionary(), start_mem,
                  DBSIZE, (uint2) PAGESIZE );
if ( rc ) {
   printf( "\nerror creating database" );
   exit( 1 );
}

/* connect to the database, obtain a database handle */
mco_db_connect( dbName, &db );
```

Figure 3. *eXtreme*DB startup implementation

The db.linux implementation allocates memory for randomized strings, initializes a DB_TASK structure, and opens the database in the "s" shared mode that enables multi-threaded access and requires transactions for assured data integrity.

```
make_strings();

stat = d_opentask(&task);

if((stat = d_dbuserid("rdmtest", &task))) return;

if((stat = d_open("mcs", "s", &task))) return;
```

Figure 4. db.linux startup implementation

From this point, both implementations enter two loops: 100 iterations for the outer loop, 300 iterations for the inner loop (total 30,000).

To add a record to *eXtreme*DB, a write transaction is started and space is reserved for a new object in the database (Measure_new). Then the sensor_id and timestamp fields are

put to the object, a random string is taken from the pool created earlier and put to the object, and the transaction committed.

```
   for ( sensor_num = 0; sensor_num < SENSORS; sensor_num++ ) {
      for ( measure_num = 0; measure_num < MEASURES; measure_num++ ) {
         mco_trans_start(db, MCO_READ_WRITE, MCO_TRANS_FOREGROUND, &t);
         rc = Measure_new( t, &measure );
         if ( MCO_S_OK == rc ) {
            Measure_sensor_id_put(&measure, (uint4) sensor_num );
            Measure_timestamp_put(&measure, sensor_num + measure_num );
            get_random_string( str );
            Measure_spectra_put( &measure, str, (uint2) strlen(str) );
            rc = mco_trans_commit( t );
            if ( rc != 0 )
                              goto rep1;
         }
         else {
            mco_trans_rollback( t );
            printf( "\n\n\tOops, error allocating object: %d\n", rc );
            goto rep1;
         }
      }
      putchar( '.' );
   }
```

Figure 5. *eXtreme*DB 'write' implementation

In the db.linux implementation a transaction is started, requiring a write-lock. The code next assigns values to a local structure for sensor_id and timestamp, copies a random string from the pool of strings created earlier, writes the record to the database (d_fillnew), and commits the transaction.

```
for( sensor_num = 0; sensor_num < NSENSORS; sensor_num++ ) {
   for( measure_num = 0; measure_num < NMEASURES; measure_num++ ) {
      if((stat = d_trbegin( "tid", &task )))
         break;
      if((stat = d_reclock(MEASURE, "w", &task, CURR_DB)))
         break;
      mr.sensor_id = sensor_num;
      mr.m_timestamp = measure_num + sensor_num;
      get_random_string( &mr.spectra[0] );
      if((stat = d_fillnew( MEASURE, &mr, &task, CURR_DB )))
         break;
      if( stat == S_OKAY ) {
         if((stat = d_trend( &task )))
            break;
         } else if((stat = d_trabort( &task )))   {
               break;
         }
         putchar('.');
      }
   }
}
```

Figure 6. db.linux 'write' implementation

Because *eXtreme*DB is a multi-threaded database, all database operations, including read access, are carried out within the scope of a transaction, so there is no need to specify the open-mode when opening the database. In contrast, db.linux has distinct single-user (so called one-user) and multi-user modes. Transactions are optional in the db.linux one-user mode, but required with the multi-user mode in order to ensure multi-user cache consistency.

A second pair of nested loops is set up to conduct the performance evaluation of reading the 30,000 objects previously created.

```
for ( sensor_num = 0; sensor_num < SENSORS; sensor_num++ ) {
   uint2 len;
   for ( measure_num = 0; measure_num < MEASURES; measure_num++ ) {
      mco_trans_start(db, MCO_READ_ONLY, MCO_TRANS_FOREGROUND, &t );
      rc = Measure_sensors_index_cursor( t, &csr );
      rc = Measure_sensors_find( t, &csr, MCO_EQ, sensor_num,
                                 sensor_num + measure_num);
      if ( rc != 0 ) {
         rc = mco_trans_commit( t );
         goto rep2;
      }
      rc = Measure_from_cursor( t, &csr, &measure );
      /* read the spectra */
      rc = Measure_spectra_get( &measure, str, sizeof(str), &len );
      rc = Measure_sensor_id_get( &measure, &id );
      rc = Measure_timestamp_get( &measure, &ts );
      rc = mco_trans_commit( t );
   }
}
```

Figure 7. *eXtreme*DB 'read' implementation

The *eXtreme*DB implementation sets up the loops and, for each iteration, starts a read transaction, instantiates a cursor, and finds the Measure object by its key fields. Upon successfully finding the object, an object handle is initialized from the cursor and the object's fields are read from the object handle. Lastly, the transaction is completed.

```
for( sensor_num = 0; sensor_num < NSENSORS; sensor_num++ ) {
   for( measure_num = 0; measure_num < NMEASURES; measure_num++ ) {
      mr.sensor_id = sensor_num;
      mr.m_timestamp = measure_num + sensor_num;
      if((stat = d_reclock(MEASURE, "r", &task, CURR_DB)))
         break;
      if((stat = d_keyfind( SENSORS, &mr, &task, CURR_DB )))
         break;
      if((stat = d_recread( &mr, &task, CURR_DB )))
         break;
      if((stat = d_recfree(MEASURE, &task, CURR_DB)))
         break;
   }
}
```

Figure 8. db.linux 'read' implementation

For the db.linux implementation, the two loops are set up and on each iteration, the key search values are assigned to a structure's fields. db.linux does not use transactions for read-only access, but requires that the record-type be explicitly locked. Upon successfully acquiring the record lock, the structure holding the key lookup values is passed to the d_keyfind function. If the key values are found, the record is read into the same structure by d_recread and the record lock is released.

As alluded to above, the key implementation differences revolve around transactions and multi-user (multi-threaded) concurrent access (there is also a philosophical difference between the object-oriented approach to database access of *eXtreme*DB, but it is unrelated to in-memory versus disk-based databases, so we do not explore it here).

With *eXtreme*DB, all concurrency controls are implicit, only requiring that all database access occur within the scope of a read or write transaction. In contrast, db.linux requires the application to explicitly acquire read or write record type locks, as appropriate, prior to attempting to access the record type. Because db.linux requires explicit locking, it does not require a transaction for read-only access.

The following graph depicts the relative performance of *eXtreme*DB and db.linux in a multi-threaded, transaction-controlled environment, with db.linux maintaining the database files on disk, as it naturally does.

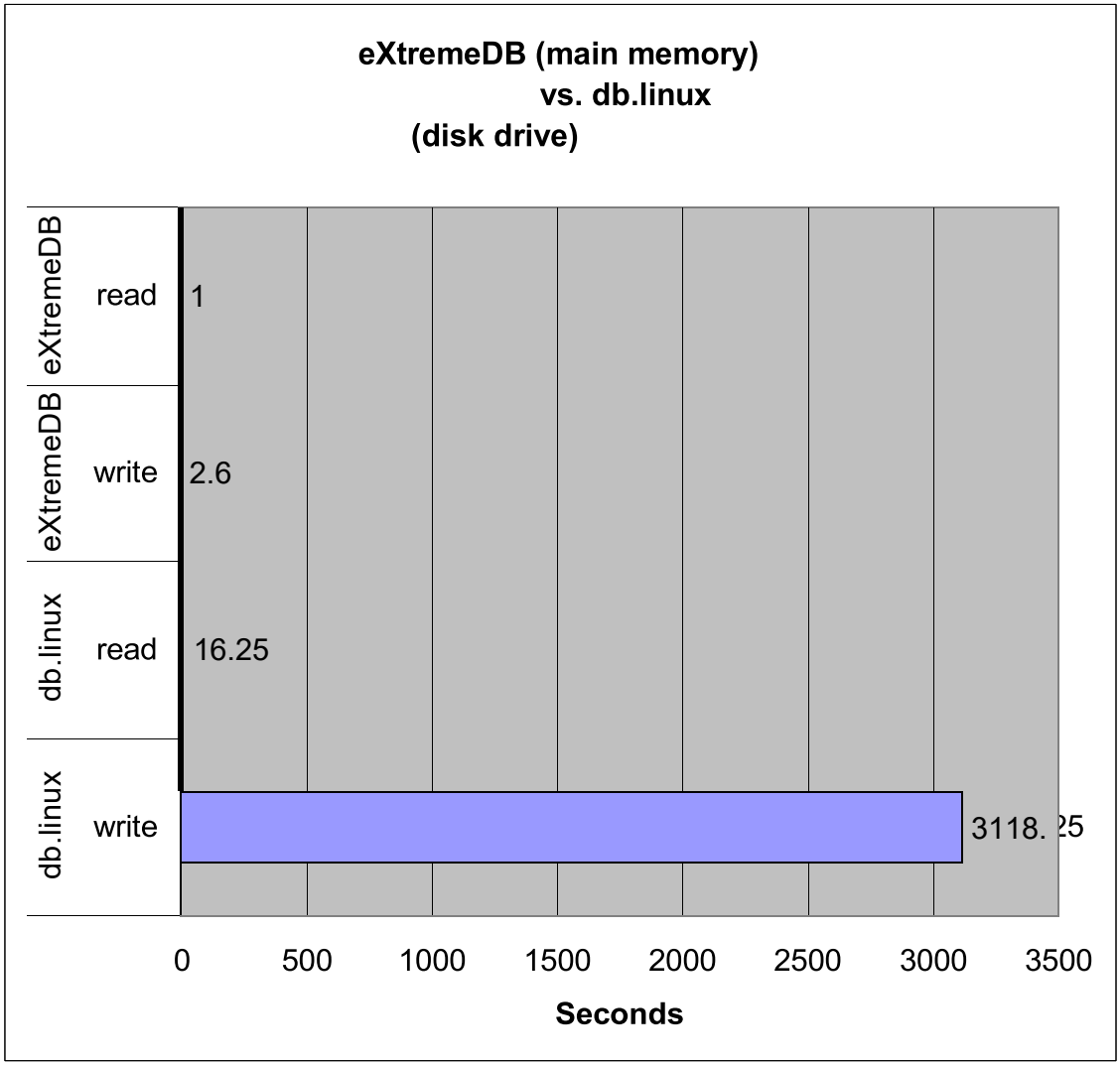

Figure 9. *eXtreme*DB and a disk-bound database

Clearly, processing in main memory led to dramatically better performance for *eXtreme*DB. By using a RAM-disk, will db.linux's performance equal or approximate that of an in-memory database?

Figure 10 shows the performance of the same *eXtreme*DB implementation used above, alongside db.linux with the database files on a RAM-disk, completely eliminating physical disk access (for details on the implementation of this RAM-disk on Red Hat Linux 6.2, see Appendix B).

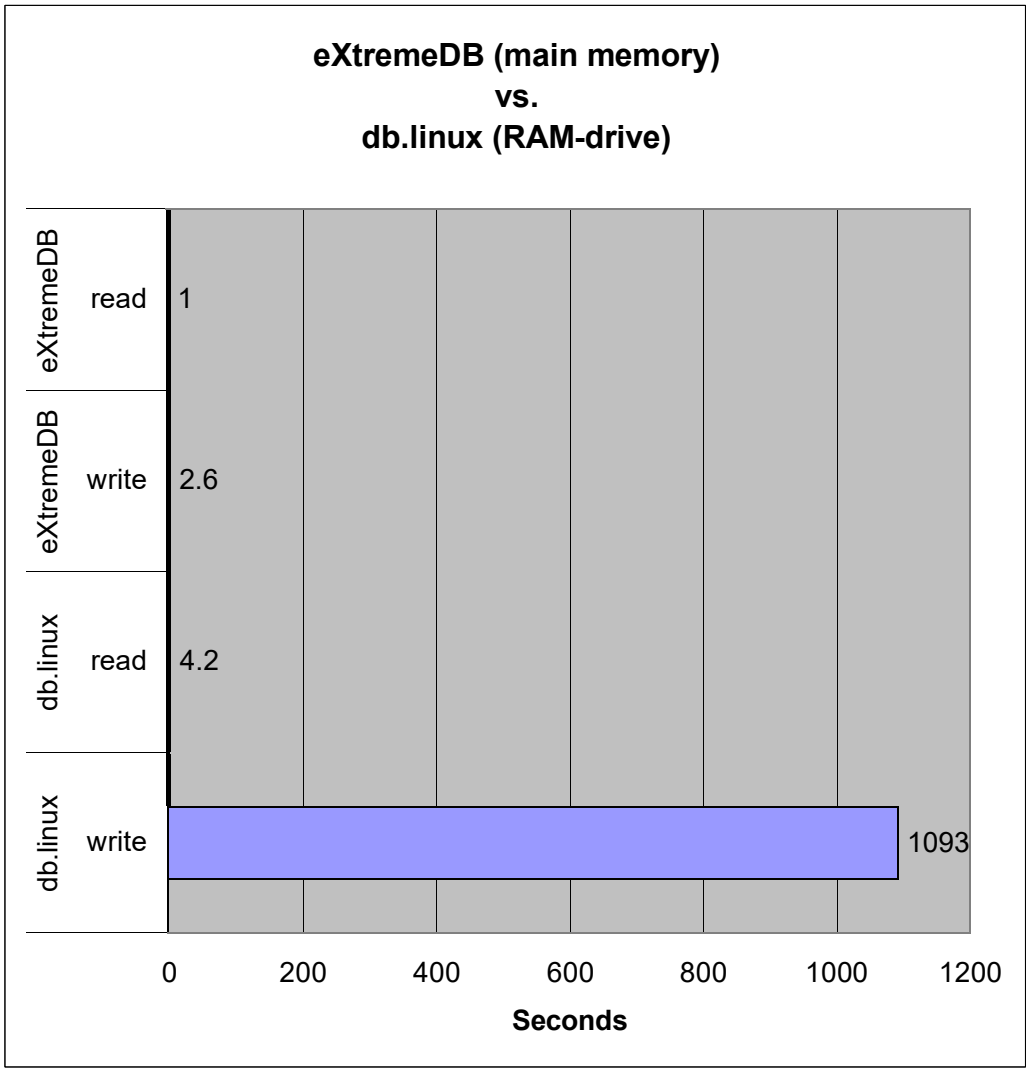

Figure 10. *eXtreme*DB and a RAM-disk database

Figure 10 demonstrates that RAM-drive deployment improves db.linux performance by almost 4X for read access and approximately 3X for writing the database. Clearly, moving a disk-based database's files to a RAM-drive can improve performance.

However, it is equally obvious that the database fundamentally designed for in-memory use delivers superior performance. The in-memory database system still outperforms the RAM-deployed, disk-based database system by 420X for database writing, and by more than 4X for database reads. The following section analyzes the reasons for this disparity.

Analysis – Where's the Overhead?

The RAM-drive approach eliminates physical disk access. So why does the disk-based database still lag the in-memory database in performance? The problem is that disk-based databases incorporate processes that are irrelevant for main memory processing, and the

RAM-drive deployment does not change such internal functioning. These processes "go through the motions" even when no longer needed, adding several distinct types of performance overhead.

Caching overhead

Due to the significant performance drain of physical disk access, virtually all disk-based databases incorporate sophisticated techniques to minimize the need to go to disk. Foremost among these is database caching, which strives to keep the most frequently used portions of the database in memory. Caching logic includes cache synchronization, which makes sure that an image of a database page in cache is consistent with the physical database page on disk, to prevent the application from reading invalid data.

Another process, cache lookup, determines if data requested by the application is in cache and, if not, retrieves the page and adds it to the cache for future reference. It also selects data to be removed from cache, to make room for incoming pages. If the outgoing page is "dirty" (holds one or more modified records), additional logic is invoked to protect other applications from seeing the modified data until the transaction is committed.

These caching functions present only minor overhead when considered individually, but present significant overhead in aggregate. Each process plays out every time the application makes a function call to read a record from disk (in the case of db.linux, examples are d_recfrst, d_recnext, d_findnm, d_keyfind, etc.). In the demonstration application above, this amounts to some 90,000 function calls: 30,000 d_fillnew, 30,000 d_keyfind and 30,000 d_recread. In contrast, all records in an in-memory database such as *eXtreme*DB are always in memory, and therefore require zero caching

Transaction Logging Overhead

Transaction logging logic is a major source of processing latency. In the event of a catastrophic failure such as loss of power, a disk-based database recovers by committing or rolling back complete or partial transactions from one or more log files when the system is restarted. Disk-based databases are hard-wired to keep transaction logs, and to flush transaction log files and cache to disk after the transactions are committed. A disk-based database doesn't know that it is running in a RAM-drive, and this complicated processing continues, even when the log file exists only in memory and cannot aid in recovery should system failure occur.

In-memory database systems must also provide transactional integrity, or so-called ACID compliant transactions. In plain English, an in-memory database application thread must be able to commit or abort a series of updates as a single unit. To do this, *eXtreme*DB maintains a before-image of the objects that are updated or deleted, and a list of database pages added during a transaction. When the application commits the transaction, the memory for before-images and page references returns to the memory pool (a very fast and efficient process). If an in-memory database must abort a transaction—for example,

if the in-bound data stream is interrupted— the before-images are returned to the database and the newly inserted pages are returned to the memory.

In the event of catastrophic failure, the in-memory database image is lost—which suits many of IMDSs' intended applications. If the system is turned off or some other event causes the in-memory image to expire, the database is simply re-provisioned upon restart. Examples of this include a program guide application in a set-top box that is continually downloaded from a satellite or cable head-end, a network switch that discovers network topology on startup, or a wireless access point that is provisioned by a server upstream.

This does not preclude the use of saved local data. The application can open a stream (a socket, pipe, or a file pointer) and instruct *eXtreme*DB to read or write a database image from, or to, the stream. This feature could be used to create and maintain boot-stage data, i.e. an initial starting point for the database. The other end of the stream can be a pipe to another process, or a file system pointer (any file system, whether it's magnetic, optical, or FLASH). However, *eXtreme*DB's transaction processing operates independently from these capabilities, limiting its scope to in-memory processing in order to provide maximum availability.

Transaction Logging can also be applied to an in-memory database, to enable recovery. The *eXtreme*DB Transaction Logging Edition does just that, as does Oracle's TimesTen in-memory database. In this case, the database is still entirely in-memory, and retains accompanying performance advantages for queries. But in addition to committing data to the in-memory database, transactions are appended to a log file and can be recovered in the event of a system failure. This impacts performance, but not as much as database writes penalize on-disk DBMS performance. The reason is that write operations for IMDS logging are always sequential, which minimizes disk head movement. In contrast, disk-based DBMS writes will cause the head to move whenever the cache has to be flushed, because random pages in the database (which are in random locations on the disk) must be updated.

So, an in-memory database with transaction logging will still outperform a conventional disk-based database by a substantial margin. Periodically, a checkpoint of the in-memory database is created so that the transaction log file(s) can be truncated and not grow without bounds. The checkpoint of the database is also a sequential write. Note that transaction logging is mandatory (i.e. not optional) with Oracle TimesTen.

Data Transfer Overhead

With a disk-based database, data is transferred and copied extensively. In fact, the application works with a copy of the data contained in a program variable that is several times removed from the database. Consider the "handoffs" required for an application to read a piece of data from the disk-based database, modify it, and write that piece of data back to the database.

1. The application requests the data item from the database runtime through some database API (e.g. db.linux's d_recread function).
2. The database runtime instructs the file system to retrieve the data from the physical media (or memory-based storage location, in the case of a RAM-disk).
3. The file system makes a copy of the data for its cache and passes another copy to the database.
4. The database keeps one copy in its cache and passes another copy to the application.
5. The application modifies its copy and passes it back to the database through some database API (e.g. db.linux's d_recwrite function).
6. The database runtime copies the modified data item back to database cache.
7. The copy in the database cache is eventually written to the file system, where it is updated in the file system cache.
8. Finally, the data is written back to the physical media (or RAM-disk).

In this scenario there are 4 copies of the data (application copy, database cache, file system cache, file system) and 6 transfers to move the data from the file system to the application and back to the file system. And this simplified scenario doesn't account for additional copies and transfers that are required for transaction logging!

In contrast, an in-memory database such as *eXtreme*DB requires little or no data transfer. The application *may* make copies of the data, in local program variables, for its own purposes or convenience, but is not required to by *eXtreme*DB. Instead, *eXtreme*DB will give the application a pointer that refers directly to the data item in the database, enabling the application to work with the data directly. The data is still protected because the pointer is only used through the *eXtreme*DB-provided API, which insures that it is used properly.

Operating System Dependency

A RAM-disk database still uses the underlying file system to access data within the database. Therefore, it still relies on the file system function lseek() to locate the data. Differing implementations of lseek() (for disk file systems as well as RAM disks) will exhibit better or worse performance based on the quality of the implementation, but the DBMS has no knowledge or control over this performance factor. In constrast, *eXtreme*DB has complete control over access methods and is highly optimized.

db.linux, in particular, is heavily dependent upon inter-process communication (IPC) for synchronization of concurrent access and transaction log recovery in the event of the failure of one or more clients, or the failure of the lock manager itself. The quality of the IPC implementation will impact the performance of db.linux but even the best implementation represents an area of significant processing overhead. Other embedded databases may or may not be dependent on inter-process communication.

Conclusion

This paper confirms two points:

- Deploying a disk-based database on a RAM-drive improves DBMS performance.
- This performance significantly lags that of an in-memory database, given an identical application task and processing environment.

The reason boils down to fundamental architectural differences between in-memory database systems and traditional database systems. Ironically, a major reason for disk-based databases lagging, even on RAM-disk, is logic that has been incorporated to *avoid* disk access, which continues to operate even though it is irrelevant in this setting. Other traditional database functions, such as sophisticated recovery from catastrophic failure, are similarly unnecessary in a memory-only environment, but cannot be "turned off" to achieve higher performance. IMDSs, while perhaps not suited for every application, offer a compelling alternative when high availability and performance are required.

While not this paper's primary focus, two other benefits of the in-memory database emerge from the experiment above. One is database footprint—the absence of caching functions and other unnecessary logic means that memory and storage demands are correspondingly low. In fact, the *eXtreme*DB database maintained a total RAM footprint of 108K in this test and 20.85MB when fully loaded with data (the raw data size is 16.7MB), compared to db.linux's footprint of 323K and 31.8MB with data (raw data is the same, 16.7MB). The second benefit is greater reliability stemming from a less complex database system architecture. It stands to reason that with fewer interacting processes, this streamlined database system should result in fewer negative surprises for end-users and developers.

Appendix A – db.linux variable length string emulation

To emulate a variable length string field with db.linux, alter the database schema as follows:

```
/*********************************************************
 *                                                       *
 * Copyright(c) 2001 McObject,LLC. All Right Reserved.   *
 *                                                       *
 *********************************************************/


struct stuff {
   short a;
};

database mcs
{
   data file "mcs.dat" contains Measure;
   key  file "mcs.key" contains sensors;

   record Measure
   {
      long sensor_id;
      long m_timestamp;

      struct stuff thing;

      compound key sensors {
         sensor_id;
         m_timestamp;
      }
   }
   record Text100 {
      char spectra100[100];
   }
   record Text200 {
      char spectra200[200];
   }
   record Text300 {
      char spectra300[300];
   }
   set Spectra {
      order last;
      owner Measure;
      member Text100;
      member Text200;
      member Text300;
   }
}
```

When populating the database, the following pseudo-code is used:

```
d_fillnew (MEASURE)
d_setor(SPECTRA)
char *p = spectra
do {
   if strlen(p) >= sizeof_spectra300
      strncpy( Text300.spectra300, p, sizeof_spectra300 )
      d_fillnew the Text300 record
      d_connect(SPECTRA)
      p += sizeof_spectra300
   else if strlen(p) >= sizeof_spectra200
      strncpy( Text200.spectra200, p, sizeof_spectra200 )
      d_fillnew the Text200 record
      d_connect(SPECTRA)
      p += sizeof_spectra200
   else if strlen(p) >= sizeof_spectra100
      strncpy( Text100.spectra100, p, sizeof_spectra100 )
      d_fillnew the Text100 record
      d_connect(SPECTRA)
   p = NULL
} while (p)
```

Note: the above pseudo-code is greatly simplified and does not cover all of the border conditions. The general idea is to break off the largest piece of the spectra string possible and store it in the appropriately sized TextNNN record and create a linked list of these records with db.linux's multi-member network model set, named SPECTRA in this example.

When retrieving the data, the linked list is traversed, concatenating the segmented spectra string back together into the whole:

```
d_keyfind (MEASURE)
d_recread (MEASURE)
d_setor(SPECTRA)
char spectra[1000];
for( (stat = d_findfm(SPECTRA)); stat != S_EOS; (stat =
d_findnm(SPECTRA)) {
   d_recread( &text300rec )
   strcat( spectra, text300rec.spectra300 )
}
```

The code to reassemble the string iterates over the set reading each set member record and concatenating the string segment to the whole. Again, the pseudo code is simplified to illustrate the primary logic of the variable length string technique.

Appendix B – RAM-Disk configuration

For the Red Hat Linux 6.2 operating system.

RAM disk setup procedures:

1. Add a line to /etc/lilo.config file:

```
ramdisk=38000
```

Here's an example of lilo.conf:

```
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
prompt
timeout=50
image=/boot/vmlinuz-2.2.5-15
        label=linux
        root=/dev/hda6
        read-only
        ramdisk=38000
```

2. Type /sbin/lilo and reboot

3. Create a mount point for the ram disk, for example:

mkdir /tmp/ramdisk0

Make sure to give appropriate access rights to this directory.

4. Create a file system on the block device:

/sbin/mke2fs /dev/ram0

Running df -k /dev/ram0 tells you how much can be used (the file system takes some space, too).

5. Mount the ramdisk

mount /dev/ram0 /tmp/ramdisk0

You are set to go.