RealDB: Low-Overhead Database for Time-Sequenced Data Streams in Embedded Systems

Project Report
Submitted to the Faculty of the
Rochester Institute of Technology,
Computer Science Department

In partial fulfillment of the requirements for the
Degree of Master of Science in Computer Science

By

____
Jason Winnebeck

Commitee Approvals:

____
Henry A. Etlinger, Advisor

____
Alan Kaminsky, Reader

____
T.J. Borrelli, Observer

September 19, 2010

Contents

Abstract

Embedded sensor monitoring systems deal with large amounts of live time-sequenced stream data. The embedded system requires a lower overhead data store that can work with limited resources and be able to run reliably and unattended even in the face of power faults.

Relational database management systems (RDBMS) are a well-understood and powerful solution capable of storing time-sequenced data; however, many have a high overhead, are not sufficiently reliable and maintenance free, or are unable to maintain a hard size limit without adding substantial complexity.

RealDB is a specialized solution that capitalizes on the unique attributes of the data stream storage problem to maintain maximum reliability in an unstable environment while significantly reducing overhead from indexing, space allocation, and inter-process communication when compared to a traditional RDBMS-based solution.

1 Problem

To describe the problem that RealDB will attempt to solve, an example use case is appropriate. An isolated platform (such as a truck or ship) is collecting time-sequenced sensor data from a sensor network consisting of sensors with little or no ability to process that data. Because the platform is isolated, an embedded computer listens to the sensor network to collect and store the data. To support periodic data synchronization, the computer supports a data query interface. The embedded computer needs to be small and low-power in this environment, so in order to address the needs of the data storage, a database engine that can work in this environment is required. RealDB aims to be a solution for the physical storage of the timestamped data that is significantly better than what can be achieved with a traditional relational database management system (RDBMS).

The design of the RealDB system revolves around the following assumptions about its environment and usage. Some of the assumptions and requirements below reference ACID (Atomicity, Consistency, Isolation, Durability)[14] concepts that are commonly used as benchmarks in relational databases.

1. Assumptions about the environment that affect the implementation of RealDB:
   (a) The database is storing incoming real-time sensor data; therefore, data is coming in order.
   (b) The system is running on an "embedded" platform, meaning:
      i. No interactive interface to the user, or perhaps no end-user interface at all.
      ii. Low-powered hardware, such as a system with 32-128MB of memory running at frequencies less than 1 Ghz.
   (c) The system has a limited storage capability (0.5 to 2GB, typically solid-state flash).
   (d) The power source for the system cannot be well-trusted, and power failures can occur.
   (e) The data is timestamped by a clock that always moves forward or stands still for insignificant (compared to sample rate) amounts of time while the system is running. During periods of collection, the clock moves forward at a rate very close to real time.
2. Assumptions about the user's accepted trade-offs:
   (a) While there may be multiple threads collecting data, only a single thread writes to or reads from the database, eliminating any concerns about isolation.
   (b) Because the database is not a traditional relational database and only deals with appending to streams, each operation stands alone, eliminating most concerns about atomicity.

   (c) Losing the most recent written data is acceptable when it trades for fast, unattended recovery.
   (d) The important part of the data is the value of the stream over time for most streams, and not the direct, original samples, giving the ability to store a compressed sample set that can be reconstructed.

3. Assumptions about the operating system regarding fault tolerance:
   (a) The operating system performs writes to the disk in the same order as RealDB performs them when operated in synchronous mode.
   (b) When the system has a power fault, blocks (bounded by some finite, known size) previously written are unmodified.
   (c) The data contained in the block being written to during a power fault is undefined (could be random, zeros, old content, new content, etc.).

RealDB does not aim to be a complete data storage and processing solution; RealDB's goal is in data recording. If extensive post-processing is required, data can be moved out of the embedded system to a more traditional database system. Additionally, some current research focuses on the data aspect of the system, such as Aurora[4]. Aurora is a stream processing engine (SPE) that provides a real-time, event-driven system for processing data stream data, but it does not include any mechanisms for storing the data. The goal for RealDB is not to develop an entire SPE, although it could potentially serve as one component in such a system for archiving the data. Therefore, RealDB does not provide a processing framework or a specialized streaming query language. Instead, it provides an application programming interface (API) as well as a basic command-line data tool for recording and exporting data.

There do exist SPEs and stream databases that have goals similar to RealDB, such as TelegraphCQ[6] and STREAM[3]. TelegraphCQ is based on the open source PostgreSQL RDBMS and has goals closest to RealDB, by providing a data store with a specialized query interface that can perform continuous queries and joins against other streams and tables. RealDB's goal is more restricted as the research focus will be on only the data storage portion and tailored for the embedded environment. Likewise, STREAM is a similar system, but the work focuses on the definition of Continuous Query Language (CQL), with little mention of the data storage.

In contrast to these previous works, RealDB will focus on the two main topics presented earlier in the section: data storage meant for embedding into a single purpose application (1, 2a-c), and defining methods for compressing sample sets and reconstructing stream state (2d) based on user-configurable parameters.

1.1 High-Level Features

RealDB provides the following features that address data recording in the embedded environment.

1. Unattended operation and availability are the most important goals; the database must be fault-tolerant and any startup recovery to restore database consistency runs in a short, bounded O(1) time, based on the (predefined) database configuration.

2. Durability: A power failure does not cause a loss of data written before a successful flush command. Data written after the last flush may be preserved, but for any lost data point, no data point following was preserved (streams can get truncated).

3. RealDB is scalable to arbitrarily large data sets. Inserting new data takes O(1) time and memory with respect to the amount of data previously stored in the database. Retrieving data takes O(log n) time for a single data point, and O(n) for a range of data points.

4. Since solid-state flash (NAND memory) is likely to be used in an embedded environment, RealDB's design focuses on minimizing write cycles to keep wear on the device to a minimum. NAND devices have a limited number of erase cycles, which are required to change the data in any memory cell. With future work, it may be possible to reduce write cycles even more.

5. The database file is compact relative to other solutions, defined by the number of bytes required to represent a particular data set.

6. Since the device is an embedded system with a low-powered CPU, the database should use minimal processing power, defined by amount of CPU time required to record a given data set.

1.2 Reliability

A key requirement for RealDB is to maintain a valid state even in the face of data corruption due to events such as power loss. The actual method of writing to the disk and performing the transactions that will prevent total loss of data but without largely sacrificing performance is the main challenge of the project. The assumptions about the environment will be the key to performing this task, most specifically the aspect that it is acceptable to truncate data streams up to the last flush point when loss occurs, if this loss is necessary to ensure that there is no corruption or loss of previously committed data and a consistent O(1) startup time (given a fixed maximum transaction log size and number of streams). The consistent startup time is key here – for example MySQL's MyISAM database format is very efficient, but when it gets corrupted the entire table needs to be scanned to rebuild the index. This requires at least O(n) time and disk space with respect to the number of rows as the entire data file is scanned and copied to a repaired file[11].

Other database formats besides MyISAM might use transactions, which have the practical effect of limiting or bounding the recovery time and space. However, performance degrades severely on embedded systems when using common transactional systems due to the overhead. RealDB will still need transactions, but intends to improve on this by minimizing transaction activity as allowed through the trade-offs taken such as delayed commits. Applications using the RealDB system can decide to trigger commits by time, space, or both. For example the application might commit/flush the database every 30 seconds, or every 1000 records. These options allow a trade-off between fault tolerance and throughput.

1.2.1 Storage Reliability Assumptions

In order to design something that is reliable within a preallocated file, the worst-case scenario from a hardware point of view when power is lost is considered. There several possibilities ranging from worst to best:

1. Entire storage corrupted: The entire disk contains random data, or the host file system, if any, is destroyed and the file no longer exists.
2. Some storage corrupted: Some portions of the database contain random data, but all others are intact.
3. One block corrupted: The physical block of data currently being written to becomes random.
4. Part of a block corrupted: The physical block of data currently being written to contains part of its old contents and part of its new contents (i.e. partially overwritten but each byte is either old or new state).
5. No data lost: The entire physical block currently being written to is either entirely in its old state or its new state.

Of course, option 1 is entirely unrecoverable, and option 2 is nearly impossible to deal with. The assumption used in RealDB is scenario 3 is the worst-case. The way that flash memory is programmed should only leave one physical block in jeopardy for corruption at a time. While RealDB's expected environment has solid-state storage, scenario 3 is also expected to be viable for modern magnetic drives that can park the head properly in power failure. Scenario 4 does not buy much extra room in design, and scenario 5 would likely only occur on transactional file systems, which is assumed not to be present on the system.

The CorruptionTest application (class org.gillius.realdb.experiments.CorruptionTest in the code) provided a reasonable validation of the assumption that scenario 3 is a worst-case for flash memory. The results of the test are described in more detail in the analysis section 7.1.

2 Other Solutions

RealDB's ability to archive data in the embedded environment is evaluated against the traditional RDBMS in this section.

2.1 MySQL

Oracle provides the following description of the MySQL database on their website:

> The MySQL® software delivers a very fast, multi-threaded, multi-user, and robust SQL (Structured Query Language) database server. MySQL Server is intended for mission-critical, heavy-load production systems as well as for embedding into mass-deployed software.[9]

MySQL is one of many popular relational database management systems (RDBMS). These databases store data in a series of tables that are related through relationships between their fields. The tables hold records (rows) that consist of a fixed number of defined columns (fields). In general, RDBMS can be adapted to functionally solve the time-series storage problem.

One reason why MySQL is not practical to use for solving this problem that is not specific to any type of table format: MySQL needs to run as a separate server daemon. Although it is quite efficient, there is no need for a separate process on an embedded system. There is an embeddable version of MySQL, but it is only usable from C/C++.[10]

2.1.1 MySQL MyISAM

The MyISAM table format[8] is probably the most suited table format for the problem of the solutions presented in this section. MyISAM is based on the ISAM table format, each table consisting of a data file, an index file, and a file describing the table structure. Assuming no deletes, data is stored sequentially into the data file. The sequential storage of the data elements allows for very fast inserts, which is desirable for a stream database.

However, it is easily marked as corrupted if tables are left open during a power loss or other failure that causes the server to terminate. The repair operation for MyISAM consists of reading the entire data file and copying it to a new location, and rebuilding the indexes from scratch. This takes a very long period of time for an embedded system and therefore is not feasible. Additionally, an index file is required that could be larger than the data itself; in fact, the worst-case scenario for the index is $(keylength + 4)/0.67$ for records inserted in order.[7] RealDB aims to reduce the indexing overhead by capitalizing on the fact that stream data is recorded in sequential order.

2.1.2 MySQL InnoDB

MySQL provides InnoDB as an alternative storage backend for its tables. Unlike MyISAM, InnoDB provides ACID (Atomicity, Consistency, Isolation, and Durability) guarantees in MySQL. The main difference is that InnoDB is a transactional database, which provides for greater reliability against faults as well as allowing atomic updates. The primary suspected issue with InnoDB is the overhead of its transactions in terms of speed, space, and wear on flash memory devices.

2.2 Apache Derby

Apache Derby[2] is an embeddable Java database, and recently Oracle includes a re-branded version of it called "Java DB" into the latest Java runtimes[12]. Compared to MySQL, Derby is meant to be used in a embedded software (into an application directly without the need of a server) context. There has been some consideration for running the system in an embedded (low-powered) hardware context, such as its small footprint around 2 megabytes and a Java Micro Edition compatible driver. Derby does support a server/client design through the use of an additional server, but only the embedded mode will be evaluated against RealDB as it most closely matches the requirements for the problem.

3 Detailed Requirements and Functional Specification

This section will describe the user-visible details of RealDB that will fit into the constraints of the stated problem as well as maintain some flexibility around potential user's current environments.

3.1 Operating Systems

In order to facilitate cross-platform operation, RealDB is written using Java, and is capable of running under at any system that supports the Java 1.6 runtime. Earlier versions of Java may work but are not tested in this project. However, the raw "partition" mode only works in certain operating systems such as Linux that expose low-level IO as files in the file system. The benchmarking tool also uses the proc filesystem in Linux to read metrics on process tree CPU usage and I/O device statistics. On other operating systems such as Windows only time and database size metrics are recorded.

3.2 Data Streams

Data streams are the main concept in RealDB. Data streams can have one of three states:

1. unknown (discontinuity)
2. specified value
3. not yet recorded

Data streams are "continuous" in RealDB, which means that a particular stream always has a state at a given time. Because of the special values, "unknown/discontinuity" and "not yet recorded," a stream has a value for any time point past or present. For the purposes of recording, time is always moving forward – retroactive changes to history are not allowed, and so each appended point must have a time greater than the previous. To mark recording sessions, applications can mark each stream with a discontinuity at the end of a session.

"Sampled" elements within a data stream are possible when using the SampledAlgorithm codec. An element being sampled is only meaningful to RealDB when accessing stream intervals, described later in this section.

3.2.1 Data Streams

1. Data streams have a user-definable, fixed, structure. The user defines the data type by combining the following primitive elements:
    (a) integers: signed 8, 16, 32, 64 bit / unsigned 8, 16, 32 bit
    (b) real numbers: IEEE-754 single and double precision floating point
    (c) Boolean values (true or false)
2. Each element in the record has an addressable name (as a string). Elements within a record are also ordered so they may also be addressed by position.
3. Each record contains an element "time," which is a signed 64 bit integer that is the time for that record in milliseconds since Jan 1, 1970 GMT.
4. Data streams are identified by a unique 32 bit integer and a name, which is not required to be, but is encouraged to be, unique.

3.2.2 Stream Element Codecs

A compression and reconstruction algorithm can be assigned to the data stream to reduce the number of data points stored and to be able to reconstruct the stream's value at any point in time. RealDB comes with predefined algorithms only for the zero-order cases:

1. StepAlgorithm
    (a) Example: speed is 0, and a sample of 0 (exactly the previous value) is given. We do not store the extra 0. If the value changes at all, store it. When interpolating, the value is flat over the entire interval.
2. DeadbandAlgorithm
    (a) For this algorithm, the deadband is the amount a value must change since the last recorded sample in order for the change to be significant (and subsequently stored). As an example, consider a speed of 25 with a deadband setting of 0.5. Therefore, if the next record with a value of 25.2 arrives, it will not be stored, but a record with value 32.0 will be stored as it is outside of the deadband. When interpolating, the value is flat over the entire interval. The original value of the stream will be preserved within an error level as determined by the deadband.

Users can provide custom compression and reconstruction algorithms. For example, it should be possible to implement a first-order (linear) or higher-order algorithm such as cubic. There will always be a trade-off in the choice of algorithm. More complicated algorithms could store fewer points, but at the cost of CPU utilization.

3.3 Data Gathering

RealDB supports the ability to gather data from a stream over specified time ranges. There are two methods for data gathering: records and intervals. For all queries, RealDB provides a streaming API (one that does not buffer and return the data), allowing for online algorithms that take O(n) linear time and O(1) constant memory.

3.3.1 Record Gathering

Record (sampled) data gathering from a data stream is appropriate when direct access to the stored data is required. Records are retrieved by specifying a start and end of a time range.

3.3.2 Interval Gathering

Interval gathering returns smart objects that represent reconstructed time-ranges of data. They are obtained by asking for data within a start and end range. The time-ranges can be queried for the following information:

- Start time
- End time
- Time span
- Average value (by element)
- Maximum value (by element)
- Minimum value (by element)
- Value at a specified time (by element)
- Integral (by element)

The behavior of the intervals is defined by the compression and reconstruction algorithm. The SampledAlgorithm will return the exact value over the whole time range, but the integral is undefined (NaN, or not-a-number as defined by IEEE-754). StepAlgorithm and DeadbandAlgorithm are zero-order and return values as if the function was flat over the time range. Higher-order custom algorithms would perform interpolation.

3.4 Fault Tolerance

RealDB is to be resilient to faults caused by power loss, given the operating system assumptions (ordered writes and data corruption only in the active block). However, there can be limitations to the resilience based on the file system when not operated on a raw device. RealDB attempts to reduce the possibility of file system corruption by preallocating the entire database file. RealDB is designed to be entirely resilient to userspace process crashes, where data and filesystem corruption would not occur (only truncated transactions, which can be completed or rolled back).

3.5 Size Management

The RealDB database has a fixed size defined when it is created. When the database is full, new data will replace the oldest data, regardless of which stream it belongs to.

4 Design and Architecture

4.1 Overview

A RealDB database file (RDB file) is a fixed size file consisting of a number of blocks with a user-defined size. Based on the reliability assumptions, to ensure maximum reliability the size of the blocks should be a multiple of the physical sector size; normally a size like 4096 is safe. When using a raw partition or device, the file "size" defines the amount of space to use in the beginning of the raw partition or device. The RealDB database is partitioned into several sections:

File Header Single block containing file size, block size, and database format version

Metadata Section Describes the data streams

Block Pool Tracks free blocks and manages transactions and block swapping (delete from one stream to add to another)

Data Index Tracks blocks allocated to each stream (not data points)

Data Section Contains the data blocks, which consist of a fixed number each of 1 or more file blocks

Databases are defined in a text file containing statements in the RealDB Definition Language (RDL), described in section 5.1. These files allow the user to set database parameters (such as the file's block size) and create streams. The creation of the database is performed in a single operation: reading the RDL, allocating and formatting the file, and populating the metadata section. Currently there is no ability to modify the streams after creation, although RDL allows the user to allocate extra space for additional indices to support adding streams in the future.

4.2 Metadata Section

The metadata section contains the following information:

- Data block size, in file blocks
- Maximum streams (data index section size is calculated from this)
- Stream information:
  - User ID (integer)
  - Name
  - Ordered list of record elements
    - Name
    - Codec algorithm used (such as SampledAlgorithm)
    - Data type
    - Whether or not the element is required in the record (nullable/optional)

4.3 Block Pool and Transactions

The job of the block pool in the database is to keep track of free data blocks, track allocations of free blocks, and track removals and additions of a block. When the database fails, it is the job of the block pool to rollback or complete started operations.

The initial design revolved around simply recording the active operations that needed to be recovered. However, it did not scale because it required the recovery process to scan the entire database to find out what happened to repair the situation. A better solution would operate with linear (O(n)) complexity on the size of the transaction log (which can be bounded to a maximum size giving an effectively O(1) recovery regardless of the amount of data in the database). Edward Sciore describes a design used by traditional databases [15], which is much easier and elegant than the initial design for RealDB. The design in Sciore creates a forward-only writing log containing the modifications (block ID, offset, old data bytes, new data bytes) to physical blocks on disk. The recovery manager can do its work using only the information in the transaction log. For rollbacks it writes the old data to the blocks, for commits it writes the new data. It might re-write something that was already done but it always ends up with the right result.

The final solution used in RealDB was not the same as that found in Sciore, but combined the ideas from both the initial design and the one described in Sciore. There were two main problems with the generic method given the current database strategy and design:

1. RealDB's architecture separates the logic from the storage format, so recording byte changes in blocks would involve the higher layer knowing how the lower layer serialized the data. There might have been a way to work around this, but that portion of the implementation was already done.

2. RealDB does not allow any arbitrary transaction – just "atomic moves" which cannot be rolled back, so there is no need for an "undo" log. The combined design leverages the more constrained situation, by reducing the space the transactions take on disk. Instead of writing the entire block, the transaction log record describes only the logical changes.

The key feature in from Sciore's design is a simple recovery process that does not need to examine the database. The final design in RealDB compromises by allowing the recovery process to access an exact piece of information identified by a transaction to confirm its state. In essence, each recovery of a transaction is a conditional modification (if X then perform Y). This is in contrast with Sciore's approach where rollback/playback is unconditional. The compromise allows for a significant reduction in log entry size by describing the logical changes at a high level. In the final RealDB design, there are four types of transaction log entries:

1. Change in the unallocated block pointer (used only before the database is 100% utilized). Fields = {next free block}

2. Removal (allocation) of an unallocated (free) block. Fields = {block number}

3. Remove a block from an existing stream index. Fields = {block number, index number, sequence number for index, next block start time}

4. Add a block to an existing stream index. Fields = {block number, index number, sequence number for index, last block}

Each transaction type has its own fields to provide the recovery process the information it needs. For the types dealing with block movements, the block number acts as a transaction ID. For example, with the remove block from index transaction, the recovery process performs the following steps:

1. Locate the data index noted in the transaction.

2. Check if the sequence number of the head equals the sequence number of the "original" block recorded in the transaction. If it matches:

   (a) Double check that the first block in the index equals the block marked in the transaction as being removed. Assuming the RealDB algorithms are correct and our assumptions on how database can be corrupted hold, this check should never fail. Since we have the block ID as the transaction ID anyway, we do this check.

Algorithm 1 Steps for transferring a block when database is full

Action	What happens if system crashes immediately after
Find the index whose first block is the oldest data block (source index)	Nothing; no disk modifications yet
Start a transaction with a Remove Block entry	Remove Block is replayed, block placed back on free list
Remove the first block from the source stream's index	Block added to (memory) free list
Write the data block	Block added to (memory) free list
End the transaction with a Add Block entry	Add block is replayed
Add the block to the destination stream's index	Nothing; transaction completed

(b) Remove the first block from the index.

Blocks can become free again if they were "in limbo" (removed but not added) when the database faulted. If there are any such blocks, they are allocated first before reclaiming old blocks in use by a stream.

The steps taken for transferring a block (happens when the database is full) are shown in algorithm 1.

Each of the actions are specified to occur in order by adding them to a in-memory transaction object. When any particular item gets flushed, it forces everything before it to occur first. When the transaction itself is committed, it causes all flush actions to occur in the order they were added (if they weren't done already). Ideally this is supposed to delay the operation as long as possible in the hopes of combining writes (for example by committing many transactions to disk at the same time). Unfortunately, despite much effort the RealDB system is unable to handle more than one outstanding "end" transaction at a time; due to problems with infinite recursion due to circular dependencies on flush ordering. Fixing this would be a top priority for future work on this project, as it could cut writes by half or more.

4.4 Data Index

The purpose of the data index is to provide a linear address space for each data stream. The main portion of the index consists of blocks with the following format:

1. Sequence number (used to determine the most recent copy in fault recovery)
2. Time of the first (earliest) record in the list of data blocks

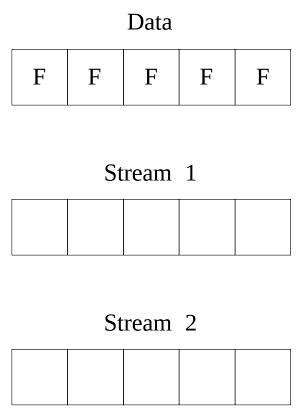


Figure 1: Initial system state

3. Time of the last (latest) record in the list of data blocks
4. Array of data block numbers (addresses)
5. Checksum, used to check for data integrity on load

The size of the index section is fixed. Each index is allocated enough space to be able to index a stream that takes up every block in the database to prevent the need for dynamic index allocation. A concern with pre-allocation is wasting space, but in the proof-of-concept database, the indexes only take up 0.28% of the database, which has 8 streams. The overhead is based on the number of data blocks (which is based on their size relative to a single file block) and the number of streams.

4.4.1 Index and Allocation Example

This section for the purposes of focusing on the data index and block allocation methods largely ignores the transaction logging that occurs to ensure reliability in a crash. Transactions are discussed in more detail in section 4.3.

As seen in figure 1, no data has been written to the database, so all blocks are free (F). The allocation map for each stream starts empty.

RealDB caches the state of the data block until it is ready to be committed (because it is full or due to an explicit flush command). When starting a new block, the database first checks to see if there are any free blocks left, and if there is one, allocates the next block by examining then advancing the free block pointer. Since blocks are taken in order, this operation takes constant O(1) time and requires only constant O(1) memory. Once all free blocks are used up, there will no longer ever be any free blocks, as the database will reclaim blocks starting with the ones containing the oldest data. It may be possible with future work to delete stream data ahead of time, but this was not implemented

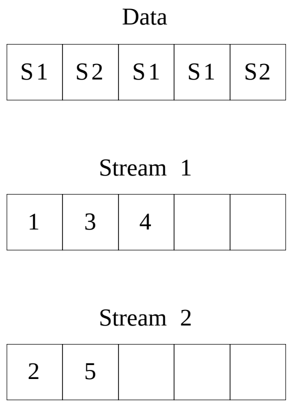


Figure 2: Database just filled to capacity

in the initial implementation since the database is a fixed size, there is no real benefit to explicitly deleting stream data.

In figure 2, all of the free blocks have been allocated. Because the first stream has been generating more data points than stream 2, it was allocated a larger portion of the database. The following sequence of events is one possibility of how the database got into this state:

1. The first records were written to stream 1, so it was allocated block 1, the first block on the free list.
2. Before the first block filled up, records were written to stream 2, so it was allocated block 2.
3. Before block 2 was filled up with stream 2's data, many more records were written to stream 1 such that it got the next two blocks.
4. Block 2 fills up, so stream 2 is allocated the last free block (5).

From this point on in the database, there will no longer ever be free blocks, so the free block pointer is no longer used. Whenever a stream needs a new block of data, a block must be reclaimed. The stream allocation map works as a FIFO queue, so blocks will only be reclaimed from the old end. The algorithm used in RealDB will select the block containing the oldest data. Due to the ordered inserts and the stream allocation maps, this process will take only linear time in the number of streams there are in the system, as the newest record in the oldest block for each stream needs to be examined. RealDB actually caches the oldest index block for each data stream under the assumption that the number of streams is low enough to be cached. Regardless of caching, this process ensures that the time to write records does not increase as the number of data points grows.

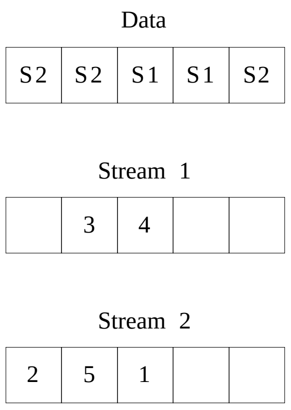


Figure 3: State of database after reclaiming a block from stream 1

It is quite possible that the oldest block may be the last block allocated for a stream, if that stream has not been written to for some time, with respect to other streams. Therefore, a potential configuration option for the database is to not consider removing the last block for a stream, if keeping the stream's last value is important. With or without this extra behavior, the time and space complexity and algorithm itself remain the same.

Following on the example, if stream 2 overflows its active block number 5, a block needs to be reclaimed. Since there are two streams, we only need to consider the newest record in blocks 1 and 2, the list of oldest blocks for each stream. The record for stream 1 is older than stream 2, so we reclaim block 1 and immediately give it to stream 2 for writing. The data that was in block 1 is now lost, if it was not preserved through some other external synchronization, archiving, or offloading process. Figure 3 shows the state of the database after this operation is complete.

The operation of the database continues similarly as long as data is written to the streams. The end result is that the database always contains the newest x bytes of records, excluding variations caused by deleting a chunk of records as a whole block at a time, and from partially filled blocks. The number and size of blocks should be configured to balance allocation performance with minimizing the two exclusions just mentioned to the ideal situation of always deleting purely the oldest record.

4.4.2 Index Lookup

Given a stream and a timestamp, RealDB locates the starting record for an iteration:

- The record corresponding to that time exactly (if it exists)
- The latest record before the given time (if it exists)

- The earliest record (if it exists)
- No record at all (if stream is empty)

To find the location of that record, RealDB drills down through the layers of indexing. The same criteria for record selection applies to index and data block selection:

1. Binary search over the index blocks for the block with the given time, the block immediately prior, or the first block. Because the index blocks are sorted physically on disk, this is possible.
2. Binary search over the sorted data blocks listed in the index block. Data blocks contain a header with the time range, so the same technique is used.
3. Binary search within the records of the chosen data block.

The first two steps involve disk I/O, but the data block is small enough to be cached in memory, so the final binary search is performed in memory. The addresses for each of the three items found are stored in a returned iterator object. The iterator object is capable of iterating over each level, loading each data block fully into memory at a time:

1. If there is another record in the current data block, return it, else:
2. If there is another data block in the current index block, load it and return to step 1, else:
3. If there is another index block in the stream index, load it and its first data block and return to step 1, else there are no more records.

4.4.3 Reliability

The main assumption about data corruption made in RealDB is that the block being written to during a system fault can change to a random state, but blocks previously written (in synchronous mode) to disk will be unmodified. The actual changes to the data index and its header are not logged as transactions, instead a two block approach is used for the header and modified index blocks . If only one block is corrupted on system fault, then as long as we do not overwrite the only copy of information, we can always recover some state. In the data index case, if the newest block is corrupted the state will rollback. Initially this technique was to be used to eliminate transactions entirely, but it is not sufficient – "atomic" operations affecting multiple streams still needed transactions.

The data index reserves 4 special blocks, two for the header, one as a backup for the tail, and another as a backup for the head. New data blocks are added to the tail, while blocks to be reclaimed are removed from the head. The blocks form a circular queue with a fixed size, so the head and tail points wrap around when they hit the end. The header block contains the following fields:

1. Sequence number, used to determine the most recent copy in fault recovery
2. Head pointer
3. Tail pointer
4. Head Advanced flag, set to true if the head pointer was just advanced, and the alternate head block should not be considered on reload
5. Checksum, used to check for data integrity on load

Loading an index and "recovery" from a fault is the same operation. Both the index blocks and the header blocks have a sequence number, which is incremented by one for every write. When loading the index, blocks with the higher sequence numbers are used.

A description of the algorithm used for flushing the tail, when the tail and head blocks are not the same:

1. If the tail pointer is being incremented, or if the last block written was the alternate tail, write to the block immediately after the tail.
2. If the tail pointer is not being incremented, write to the current tail location or the alternate tail block, whichever was not written last.
3. If the tail pointer was incremented, update the header. Each write of the header block alternates between the primary and secondary blocks, writing always to the oldest one.

The index block is written in a "safe" location, and before the header is updated. If there is a system fault in the above, one of the following happens upon restarting:

1. The block immediately after the tail was written, but nothing referenced it, so it had no effect.
2. The tail block version (alternate or primary) that did not contain the most recent data was being written to and was corrupted. This block will just be ignored (rollback operation).
3. The old header block was being updated to the new version but was corrupted. The original header will be used (rollback).

When loading the index:

1. Load both header blocks and choose the "best" block – the one with a proper checksum that has the greater sequence number, accounting for overflow ($a - b > 0$ means a is greater). Note the older or corrupted block as the next block to be written.
2. If the "head advanced" flag is true, load the head block, else load both the block pointed to by the head pointer plus the alternate head, picking the "best" and noting the next block to be written.

3. If the head and tail pointer is not the same, load both tail blocks, picking the "best" and noting the next to write.

A similar technique is used for updating the head block, except that when "deleting" the head block, the header is updated with the "head advanced" flag set true. In each failure state, either the block write is rolled back, or the header write was rolled back. When the index only has a single block (head and tail pointers are the same), the "alternate head" block is used for alternating writes of the index block.

4.5 Data Section

The data section simply consists of a set of data blocks, each of which comprise of a number of file blocks. The larger the data block, the fewer there will be in number, which reduces the size of the indexes and the number of transactions, leading to better performance. The main disadvantages to large data blocks in the current RealDB implementation are partial blocks and data loss – a DB flush will write a partial block, leaving the rest of the block blank until it is reclaimed later, and because there are fewer commits to disk, when a system failure occurs, more data will be lost. While RealDB does not guarantee any data past the last commit, in practice whenever a data block fills up it is written to disk and added to the index so that it is not lost. Smaller data blocks increase the probability and amount of data that will be recovered after the last commit before a system fault.

RealDB assumes that data blocks are small enough to fit in memory. RealDB caches the block pending to be written for each stream in memory, and when reading data, RealDB reads records an entire data block at a time. Therefore, the maximum data block size is limited based on the amount of memory that can be dedicated to the database.

4.6 Stream Codecs

In RealDB, a stream codec is responsible for reducing (or compressing) the original elements in records of a data stream when writing, and reconstructing the stream intervals when reading.

The codec algorithms are able to see a finite number (defined in their code) of points before (look-behind) and after (look-ahead) the sample point that is under consideration for dropping. The points behind were the previously written points and the points ahead are in "limbo" waiting to be considered. The code is required to handle the "start" and "end" of a compression/reconstruction period (denoted by discontinuities) where all points before and/or after are undefined (null). Therefore, the model of the codec algorithm is a streaming (single pass) transformation function with a delay equal to the look-ahead. The below table shows a theoretical transformation of a stream of 5 samples to a stream of 3 samples with a look-ahead value of 2:

Input	Output
2.0	null
3.0	null
4.0	2.0
4.0	null
4.5	4.0
null	null
null	4.5

Zero order interpolation methods (such as deadband, and always sample) are expected to need no look-ahead, first order needs one look-ahead, and higher order like cubic or Hermite would need two or more look-ahead points.

As it may be possible for “perfect” compressions based around cubic or Hermite methods to be NP-complete, a heuristic algorithm was developed to prove that it is possible to practically implement higher-order codecs within the current framework. One solution capable of considering groups of up to “m” points is the following:

1. Know the last 2 points output by the algorithm
2. Look at the m+2 most recent points in the stream (the finite sized m keeps this algorithm linear time with respect to the samples in the stream, complexity O(n*m))
3. Consider the last 2 output points and the 2 most recent input points. Form the interpolation based on these 4 points and check the error of the interpolation on the m (middle) points.
   (a) If the interpolation’s error is under the threshold, drop the 3rd most recent point.
   (b) If the error is out of threshold, output the 3rd and 2nd most recent points (and the most recent point becomes part of the "m" set for the next period).
4. If the size of the "m" buffer is full, then output the 2 most recent points.

There are some edge cases in the above omitted for simplicity, to express the point simply that such an algorithm is feasible and could be successful in eliminating a significant portion of points. The summary of this algorithm is that it theorizes that all of the points “m” can be dropped and continues to add points to “m” until that theory no longer holds. When it no longer holds it outputs the pair of points immediately following the “m” points where the theory held. In the worst case scenario it will output all points but in the best case scenario it could eliminate points that follow a curve.

Originally the intention was to implement more codecs and do more research in this area; however, due to the complexity of the project, the work remains focused on the underlying storage mechanism and therefore the benchmark deals only with zero order algorithms. The API provided to custom algorithms is sufficient to build virtually any algorithm that does not need to alter the element structure (i.e. change/add/remove element data types). The original concept proposed allowed for algorithms to modify the structure, but this grew very complicated very quickly. The discovery of Paul Bourke's interpolation methods, which all worked with the original sample points[5], led to the realization that the extra complexity would not be needed.

5 Implementation

5.1 RealDB Definition Language (RDL)

The RealDB Definition Language (RDL) allows the user to define the parameters and streams within a database. The lexer and parser for the language is defined using a grammar provided to the ANTLR parser generator[1] for Java. The small ANTLR runtime library used by the generated code is the only runtime dependency for the RealDB core database. The parser generates an object model for the file, which is used as an input to the DatabaseBuilder class. An example of an RDL file is shown in figure 4. The full grammar can be found in the source code file RDBDefinition.g.

```
SET blockSize     = 2048
SET fileSize      = 204800
SET maxStreams    = 3
SET dataBlockSize = 2

CREATE STREAM Test WITH ID 1 {
  value float NULL //will use SampledAlgorithm by default
}

CREATE STREAM CarSnapshots WITH ID 2 {
  rpm float WITH CODEC DeadbandAlgorithm PARAMS (deadband=50.0),
  speed float WITH CODEC DeadbandAlgorithm PARAMS (deadband=5),
  passengers uint8 WITH CODEC StepAlgorithm,
  driving boolean WITH CODEC StepAlgorithm
}
```

Figure 4: Example RDL File

5.1.1 Database Parameters

blockSize	Sets the low-level block size of the file, must be greater than or a multiple of the native block size to ensure complete reliability.
fileSize	Size of the file, in bytes, should be a multiple of the blockSize; a partial block at the end would be wasted.

maxStreams Amount of space to allocate in the file for data streams. This must be equal to or greater than the number of streams actually created. RealDB does not implement adding streams today, but this is a proposed future task.

dataBlockSize Multiple of blockSize for a raw stream data block.

5.1.2 Creating Streams

- CREATE STREAM <stream name> WITH ID <unique integer user ID>
- <element name> <element type> [NULL] [WITH CODEC <codec name>] [PARAMS(a=value, b=value, ...)]
    - If NULL is specified, the element is allowed to take on a null value, otherwise it must always have a defined value
- <codec name> takes one of the following options:
    - SampledAlgorithm (the default) - Every record is written
    - StepAlgorithm - Record is written if the value changed at all
    - DeadbandAlgorithm - works only on numeric types; value written if $|nextValue - previousValue| \geq deadband$; takes deadband as a parameter.
    - A fully-qualified class name of a Java class that implements ElementCodec (allows user-defined codecs).
- <element type>
    - sint8, sint16, sint32, sint64, uint8, uint16, uint32 = signed and unsigned integer types of 8, 16, 32, or 64 bits.
    - float, double = 32 bit and 64 bit types, supports IEEE-754 ranges and special states including denormals.
    - boolean = element can be true or false
- Parameters
    - Codecs can take parameters to customize their behavior. Out of the three provided codecs, only DeadbandAlgorithm takes parameters, a single parameter “deadband.”

5.2 Testing

Unit tests are used to validate the logical functionality of most of the components in the system. Integration tests tie those components together to test the reliability of the database. In the source code over 230 tests are provided. The most important tests are the integration tests that test the reliability of the database in system fault conditions. To make this testable, all code accesses the file through an interface called BlockFile that abstracts all access to the raw file. Three BlockFile implementations are key to the integration tests:

ByteArrayBlockFile Provides a memory implementation

ProfilingBlockFile Wraps another implementation to provide statistics on blocks accessed. This allows a test to determine how many operations were performed.

FailingBlockFile Wraps another implementation that causes a failure after X operations, where X can be set arbitrarily. If a write method fails, the write that was specified is replaced with a write that writes random data to the location.

There are several variations of the integration test, but they all follow the same pattern:

1. Create an in-memory database using ByteArrayBlockFile.
2. Run the full test with a ProfilingBlockFile and determine the maximum number of operations (N). Verify the contents of the database are as expected by scanning all data streams.
3. For i = 1 to N:
    (a) Run the test with FailingBlockFile to fail after i operations. Verify that the database can be re-opened without exceptions and that all constraints are valid, particularly:
        i. Data read for a particular data stream was the data actually written for that stream.
        ii. The only records lost are those after the last commit (flush).
        iii. The records that were lost represent a proper truncation of the stream; for example, if records 1-100 are written, then 1-90 would be a proper truncation, but 1-80 + 90-100 would not be, because record 81 was lost but records after it were kept.

The purpose of the integration test was to assert that no matter when the database failed, it could recover within the required bounds. A brute-force attempt (try the failure at every IO operation) ensures that all cases are covered. The different variations of the integration test use different numbers of streams with different data rates and flush patterns to try to invoke all code paths. A possible future work for the project is to run the tests under a code coverage analysis tool to investigate how good the coverage was. The current integration tests represent a best effort at covering the cases.

6 Deliverables

The website `http://www.gillius.org/realdb/` is the permanent home of the RealDB project and contains all of the deliverables of the project for public download. The following are the code (binary and source) deliverables for the project:

realdb-core The implementation consists of open-source code written in the Java language that implements the RealDB storage engine and provides the APIs to allow Java programs to interact with the engine and the data streams.

realdb-demo A demonstration and testing application that writes data to a device, handling faults when the device fails (such as removal of the flash media), and resuming writes where it last left off when the media is restored.

realdb-cli A command line interface allowing the user to create, describe and read databases, as well as the ability to bulk insert data from a CSV format.

realdb-browser A graphical tool for inspecting the structure and contents of a database, including the ability to generate graphs and tables of stream data. A screenshot of this application is shown in figure 5.

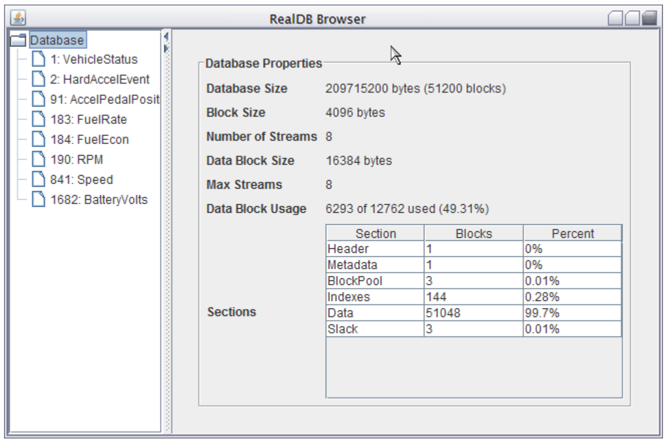
realdb-benchmark A benchmark to support the experimental portion of this project, discussed in detail in section 7.2.

realdb-concept Proof of concept application, described in section 6.1.

Documentation for the project includes Javadocs for at least the realdb-core component, the original proposal, and this final report. The only piece used in the project not delivered is the full bus maintenance dataset.

6.1 Bus Maintenance Proof-of-concept

The RealDB storage engine could be used in many situations involving high speed time series data collection. To highlight one such scenario, a proof-of-concept application was developed centered around a bus fleet maintenance situation. In this situation, the maintenance supervisor wishes to decrease the time it takes to diagnose and repair problems with buses that occur while they are out in the field. He or she installs an embedded device running RealDB and an application using it which reads the existing vehicle data bus to record data such as RPM and speed for the engine and transmission over the last few



Database Size	209715200 bytes (51200 blocks)
Block Size	4096 bytes
Number of Streams	8
Data Block Size	16384 bytes
Max Streams	8
Data Block Usage	6293 of 12762 used (49.31%)

Sections

Section	Blocks	Percent
Header	1	0%
Metadata	1	0%
BlockPool	3	0.01%
Indexes	144	0.28%
Data	51048	99.7%
Slack	3	0.01%

Figure 5: Screenshot of the RealDB Browser

weeks. When a driver complains of “poor shifting” while running a route, the maintenance team is able to connect a PC to the embedded system to download the RealDB database to analyze the data offline. Using their analysis software and the driver’s report, the maintainer can locate the anomalous time spans and see the exact behavior of the engine and transmission to the millisecond level. This analysis provides two benefits, one being that the maintainer receives exact information about the problem (beyond the driver saying, for example, “it shifts weird”), and the second being that the maintainer does not need to try to replicate the problem to diagnose it, since all of the information needed was collected by the monitoring system. Both of these benefits achieve the goal of reducing the time it takes to make a diagnosis. RealDB is a good fit for this scenario (when compared to the other solutions under study for this project):

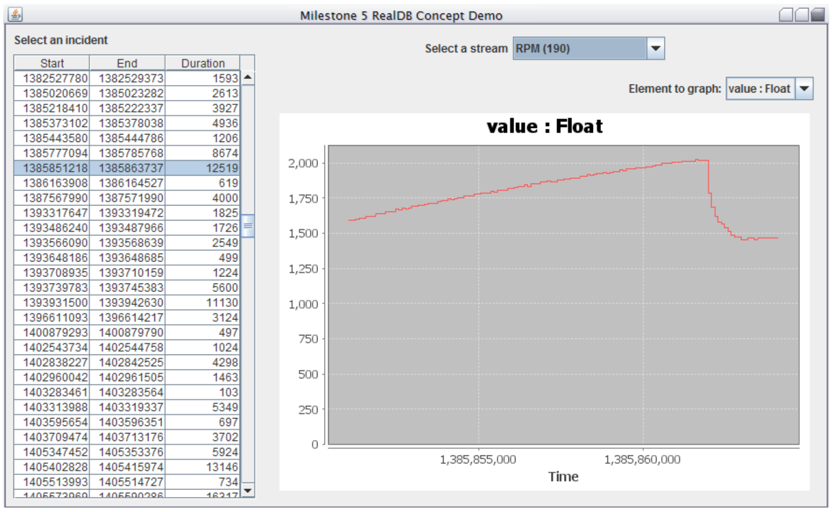
1. Its lower overhead and scalability allows the purchase of a cheaper, lower-powered system.

2. Its fault-tolerance allows for a zero maintenance system, even in the unstable power environment of the mobile vehicle.

3. Its ability for automatically managing size of the database allows it to store the most recent data.

4. The configurable compression and reconstruction capabilities allow the users to increase the amount of history they can store in the same space, based on their tolerance of error in the reconstruction. Or, RealDB can even record every sample if zero-error reconstruction is desired.

To demonstrate this scenario, the realdb-concept project simulates the software that feeds the storage engine with the vehicle bus data (class VehicleDataSimulation). The concept code includes a tool “AnalysisDemo,” which reuses components from realdb-browser to provide a possible targeted graphical interface for



Select an incident

Start	End	Duration
1382527780	1382529373	1593
1385020669	1385023282	2613
1385218410	1385222337	3927
1385373102	1385378038	4936
1385443580	1385444786	1206
1385777094	1385785768	8674
1385851218	1385863737	12519
1386163908	1386164527	619
1387567990	1387571990	4000
1393317647	1393319472	1825
1393486240	1393487966	1726
1393566090	1393568639	2549
1393648186	1393648685	499
1393708935	1393710159	1224
1393739783	1393745383	5600
1393931500	1393942630	11130
1396611093	1396614217	3124
1400879293	1400879790	497
1402543734	1402544758	1024
1402838227	1402842525	4298
1402960042	1402961505	1463
1403283461	1403283564	103
1403313988	1403319337	5349
1403595654	1403596351	697
1403709474	1403713176	3702
1405347452	1405353376	5924
1405402828	1405415974	13146
1405513993	1405514727	734

Figure 6: AnalysisDemo User Interface

the task, shown in figure 6. The AnalysisDemo tool displays the list of incidents on the left side of the screen. When the user selects an incident, the right side is populated with a graph of a particular vehicle parameter (stored as a data stream) over the time range of the incident. A drop-down box allows the user to select which stream and element of that stream to view.

6.1.1 Data

In the interest of using a realistic data set, the proof-of-concept and benchmark uses data collected from a city transit bus over a full month as part of a sponsored research project for the Office of Naval Research[13]. The data simulation adds a few derived data streams, giving a total data set size of just over 9.2 million data points. The RealDB project itself is not associated with, sponsored, or funded by the ONR research project. Unfortunately, due to restrictions on the data, the data points cannot be published.

7 Analysis and Conclusions

7.1 CorruptionTest

The CorruptionTest is an experiment that was performed before starting the RealDB implementation to ensure that the assumptions made about storage failures during power faults in section 1.2 are reasonable. This test allocates a fixed size file, clears it with all zeros, then starts writing blocks of random data with block numbers and checksums. The program was used to write to a CompactFlash card connected to a USB card reader. While the program was writing data, the card was physically pulled out of its slot. After reinserting the card, the analysis portion of the program checked which blocks were corrupted, and which were still zeros. The experiment, which was ran multiple times, showed that the block being written at the time of failure was the only block corrupted.

7.2 Benchmark Implementation

7.2.1 Metrics

The performance benchmark measures the following metrics:

- Database size (assuming no filesystem overhead). For RealDB this measures the utilized space, since the data files are a fixed size (50MiB and 100MiB).
- DB startup and creation:
  - time
  - disk sectors reads/writes
- DB load and shutdown:
  - time
  - disk sectors read/write
  - reads/writes disk millis
  - user and kernel mode jiffies (including those of child processes)

7.2.2 Dimensions

- Implementations: RealDB File, RealDB Partition, Derby, MySQL MyISAM, MySQL InnoDB
- Size Management: Y/N
- Number of records: every 1 million

7.2.3 Environment

- Ubuntu GNU/Linux 9.10 (Karmic), kernel 2.6.31-21-generic
- Intel Core 2 2.4 GHZ E6600 (CPU frequency scaling left on)
- 2GB RAM
- Java 1.6: OpenJDK 6b16-1.6.1-3ubuntu3
- Sandisk Ultra II CompactFlash "15 MB/sec" with 0.5GB unformatted partition (for RealDB Raw) and 3.5GB FAT32 partition (all others)

- Generic USB 2.0 memory card reader "ALLIN1"
    - I/O performance according to “dd”: 5.7MiB/s writes, 6.8MiB/s reads – The reader is likely the bottlebeck as other machines with different readers tested faster with the same card.
- MySQL 5.1.46
    - MySQL Connector/J JDBC Driver version 5.1.12
- Derby 10.5.3.0_1 with embedded JDBC Driver

7.2.4 Notes

The size management applies significantly only to the non-RealDB configurations, where the benchmark issues a DELETE after every batch of 1000 records to delete all records older than the first by some amount of time (right now 1000 seconds). For RealDB, the difference is a 50MiB database, which is too small to hold all of the data versus a 100MiB one, which can hold all of the data.

For SQL databases, the benchmark implements an equivalent step to the StepAlgorithm that RealDB uses to eliminate duplicate data points, which puts both systems on equal footing. MySQL and Derby are used as they come "out of the box", except that on MySQL the benchmark specifies the JDBC URL parameter rewriteBatchedStatements=true to turn JDBC batches into MySQL multi-inserts, which offers a huge speed up (about 10x but was not measured directly).

The benchmark software collects the information on disk reads/writes and CPU information using a Linux-specific method. The proc filesystem contains information on the number of CPU jiffies (slices of time) given to a particular process and its children, as well as read/write sector counters for each disk. The benchmark reads this information at the start and end of each test to find the deltas. Because the flash disk is dedicated to the benchmark, the counters are accurate because all reads and writes are due to the test.

7.2.5 Schema

The RDL used for the RealDB version of the benchmark is shown in figure 7. The SQL used for the MyISAM version is shown in figure 8. The InnoDB version is the same as MySQL except with “ENGINE = InnoDB”. The Derby version is similar except for changes due to the different dialect and data types. For the SQL databases, each type of stream shares the same table. An index is created on the time column because it is used to select rows for deletion.

7.2.6 Process

The benchmark software runs the entire test suite in a single execution, iterating over each of the dimensions: implementation, then size management, then by

```
SET blockSize     = 4096
SET fileSize      = 52428800 #or 104857600
SET maxStreams    = 8
SET dataBlockSize = 4
CREATE STREAM RPM WITH ID 190 {
  value float
}
CREATE STREAM FuelRate WITH ID 183 {
  value float
}
CREATE STREAM AccelPedalPosition WITH ID 91 {
  value float
}
CREATE STREAM BatteryVolts WITH ID 1682 {
  value float
}
CREATE STREAM FuelEcon WITH ID 184 {
  value float
}
CREATE STREAM Speed WITH ID 841 {
  value float
}
CREATE STREAM VehicleStatus WITH ID 1 {
  collecting boolean WITH CODEC StepAlgorithm
}
CREATE STREAM HardAccelEvent WITH ID 2 {
  active boolean WITH CODEC StepAlgorithm
}
```

Figure 7: Benchmark RDL

```
DROP DATABASE IF EXISTS ‘realdb_benchmark‘;
CREATE DATABASE ‘realdb_benchmark‘;
CREATE TABLE ‘realdb_benchmark‘.‘FloatData‘ (
  ‘streamId‘ INTEGER UNSIGNED NOT NULL,
  ‘time‘ BIGINT UNSIGNED NOT NULL,
  ‘value‘ FLOAT NOT NULL,
  ‘discontinuity‘ BIT NOT NULL,
  PRIMARY KEY (‘streamId‘, ‘time‘),
  INDEX ‘Index_Time‘(‘time‘)
)
ENGINE = MyISAM;
CREATE TABLE ‘realdb_benchmark‘.‘BooleanData‘ (
  ‘streamId‘ INTEGER UNSIGNED NOT NULL,
  ‘time‘ BIGINT UNSIGNED NOT NULL,
  ‘value‘ BIT NOT NULL,
  ‘discontinuity‘ BIT NOT NULL,
  PRIMARY KEY (‘streamId‘, ‘time‘),
  INDEX ‘Index_Time‘(‘time‘)
)
ENGINE = MyISAM;
```

Figure 8: MyISAM Benchmark SQL Create Script

every million records. To form a truncated dataset, the benchmark just uses the first N million records of the 9.2 million record dataset. Each implementation is set up to store its database files only on the flash partition within a specified directory, with some minor exceptions such as Derby's small "derby.log" administrative log file. Then the benchmark performs the following steps:

1. Delete all of the files in the data directory
2. Take a proc snapshot
3. Instruct the database implementation to create a new database, using an RDL or SQL script
4. Take another proc snapshot to gather the metrics for database creation
5. Load records
   (a) While loading records, if on a SQL-based database, after each batch issue a SQL delete for rows more than 1000 seconds older than the latest row.
6. Take a proc snapshot and gather metrics for database loading, and add up the file sizes of all files in the database directory to get database size (except for RealDB where the file size is already known and fixed, the benchmark queries RealDB for the number of allocated blocks in the file).

Because the benchmark software runs all iterations in the same execution process, the RealDB and Derby Java classes will be cached in memory. Due to the amount of work in the benchmark and the fact that the work is almost entirely I/O bound, classloading is likely not a significant factor in the performance.

7.2.7 Data

The dataset used is the same raw data used in the proof of concept plus the 2 generated Boolean streams. 9203285 records result from this. For more information on the data set, see section 6.1.1.

7.3 Benchmark Results and Analysis

7.3.1 Result Notes

While running the benchmark, the time taken to run InnoDB with the size management code, and Derby in any mode was extreme in comparison to the other benchmarks. Therefore the record set was truncated to 5 million for InnoDB with size management and Derby without size management, and Derby with size management was not even measured. For tables that contain data, the term "n/t" for "not tested" is used.

When graphing the data there are two types of charts. The first is a line chart showing each implementation's performance by the number of records, which shows the way that the data scales. The second type is a bar chart showing only 5 million (because Derby and InnoDB stop there) and 9.2 million records. The bar chart is easier to read for the "bottom line" performance.

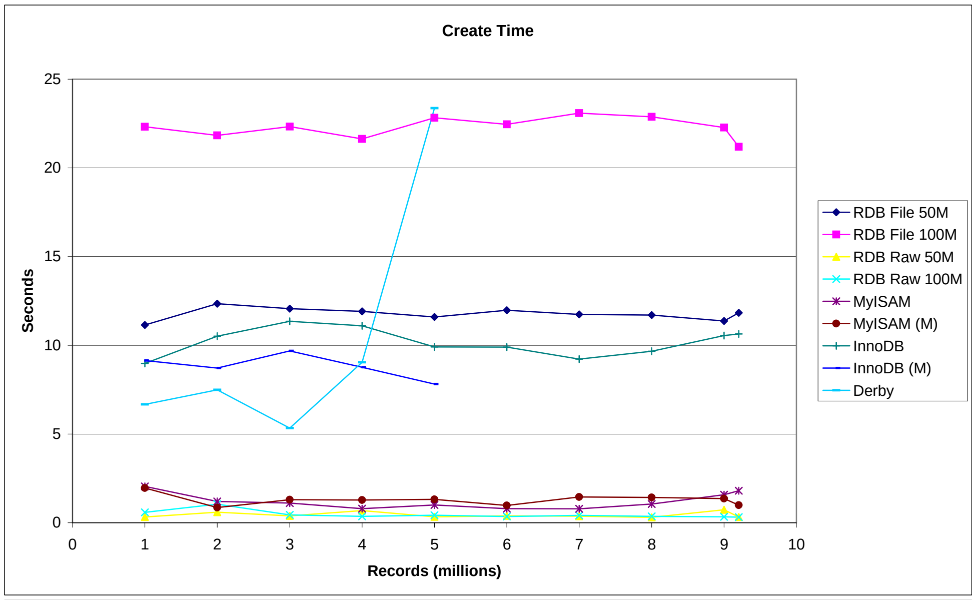
The full dataset for all data presented in this section as well as other collected data not presented in this paper are available with the other deliverables on the RealDB site in several formats.

7.3.2 Creation Time

The creation time for each implementation was constant time for each implementation regardless of the number of records, which was not surprising, because the action to create a database is independent from the number of records. The exception is the anomaly with Derby creation time; the reason is currently unknown. Only RealDB differed because of the file size between size-managed and not. The creation time is graphed in figure 9.

7.3.3 Load Time

As expected, in each implementation, the load time is linear. For RealDB we can see that the design meets the complexity requirements of O(n) time with respect to the number of data points. On the line graph Derby is not even present, because even at 1 million records without size management it takes 3963 seconds, which was 1.4x longer than the entire suite of tests took for MySQL combined, and far off of this graph. At 5 million records Derby is taking already



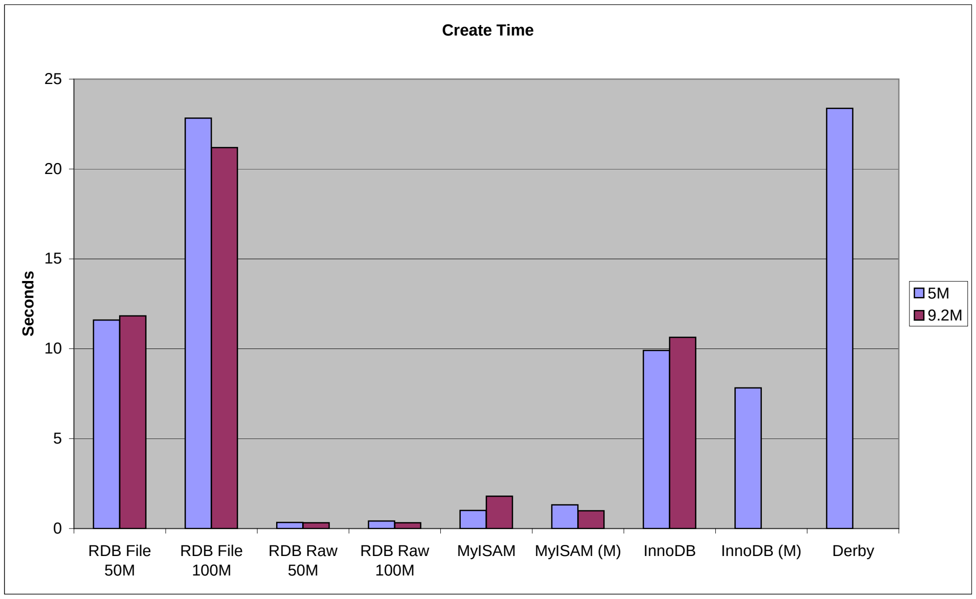


Figure 9: Creation Time

Implementation	5M Records	9.2M Records
InnoDB	858.618	1588.762
InnoDB (M)	3173.966	n/t
Derby	21970.746	n/t

Table 1: InnoDB and Derby Load Times

Implementation	5M File	9.2M File	5M Raw	9.2M Raw
MyISAM	-36.6%	-39.8%	-26.8%	-36.2%
MyISAM (M)	-12.9%	-15.6%	-6.8%	-9.2%
InnoDB	83.2%	84.1%	84.4%	84.5%
InnoDB (M)	95.5%	n/t	95.7%	n/t
Derby	99.3%	n/t	99.4%	n/t

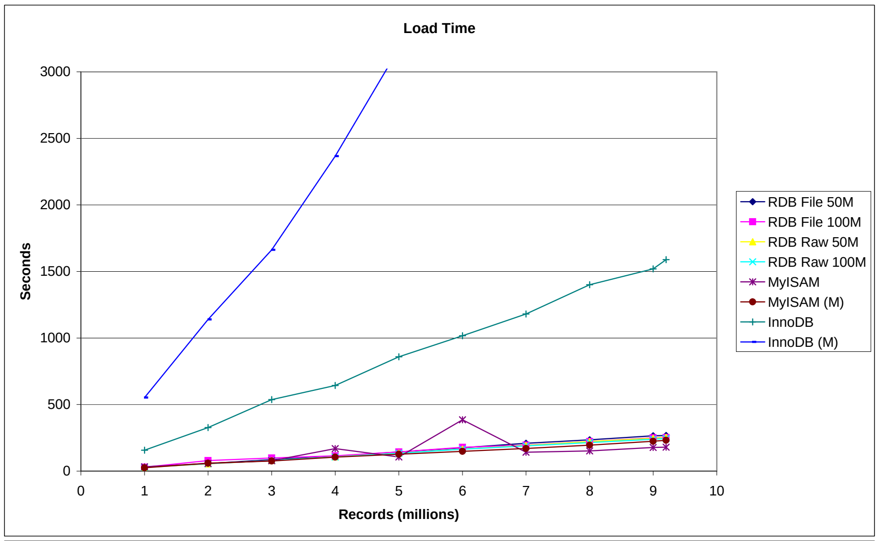
Table 2: Load Time Reduction with RealDB

6 hours to run the benchmark. InnoDB with size management quickly climbs off of the chart as well. On the bar graph, InnoDB and Derby extend off of the page. The graph bounds are modified to show the distinctions between the faster implementations. The data graphs are shown in figure 10. The values in seconds on the bar graph for InnoDB and Derby are shown in table 1, since they are not visible on the graph. In this test, MyISAM performs the best, with or without size management, but RealDB is much closer to MyISAM than InnoDB or Derby. RealDB takes 15.6% longer than MyISAM on a file system with size management enabled, as seen in table 2.

7.3.4 Database Size

The benchmark measures the size of the files on disk created by the database implementation at the end of each run. For RealDB, the file sizes are fixed at 50 or 100 MiB, so the size presented in this graph is the amount of space utilized (total space - free data blocks). This size represents roughly the minimum size file that could be used to store the given data. For the sized-managed versions of the test, we see RealDB's utilized space reaches the maximum and stays there, as expected. The charts are shown in figure 11.

Without issuing deletes, the versions without size management are an attempt to represent the minimum size of the dataset, without having to make explicit optimization calls. The size management versions of the test is not a perfect comparison to RealDB, because it is not possible to replicate the same environment. In the SQL version of the benchmark, after every batch, records more than 1000 seconds older than the most recent are deleted, whereas in RealDB, new blocks just overwrite the old. In practice, 1000 seconds is reached very quickly so deletes start happening even at the 1 million level. However, some comparisons are still possible. InnoDB and Derby's performance is particularly surprising because the size is increasing linearly with the number of



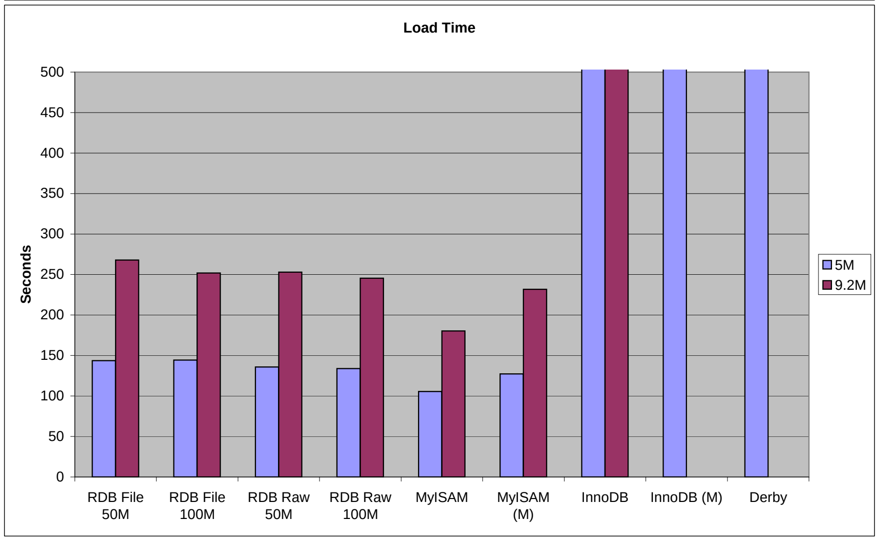


Figure 10: Load Time

records written, even though the number of records contained in the database is limited. This suggests that there may be an explicit compaction step required by the administrator to reduce database size. MyISAM behaves more like initially expected, staying near a size relative to the number of active rows in the table.

When only inserts are performed, the MyISAM data file contains a tightly packed sequence of binary structures representing each row, and taken alone would probably be smaller than the RealDB database. Therefore, the MyISAM index files on the primary key and time column likely represent the majority of the extra overhead. RealDB leverages its advantage of data being inserted (and stored) in order to eliminate the separate indexing structures.

The entire 9.2 million record dataset is able to fit in the 100MiB RealDB file, with just over a MiB to spare. Table 3 shows the data reductions when comparing RealDB to each implementation without size management.

7.3.5 CPU Utilization

The benchmark measures the total CPU usage of the benchmark process and its children (such as when it launches mysqld) over the load operation. For each implementation, the time used appears to be linear. In this test, RealDB appears at the very bottom of the graphs shown in figure 12, so it greatly outperforms the other solutions in terms of CPU used. The scale of the bar graph has been adjusted to make the RealDB bars visible in comparison to the MyISAM implementation. Table 4 shows the summaries of CPU reduction at 5 and 9.2M records against each implementation. The size-managed implementations are compared to the RealDB 50MiB database, and the others are compared against the 100MiB version.

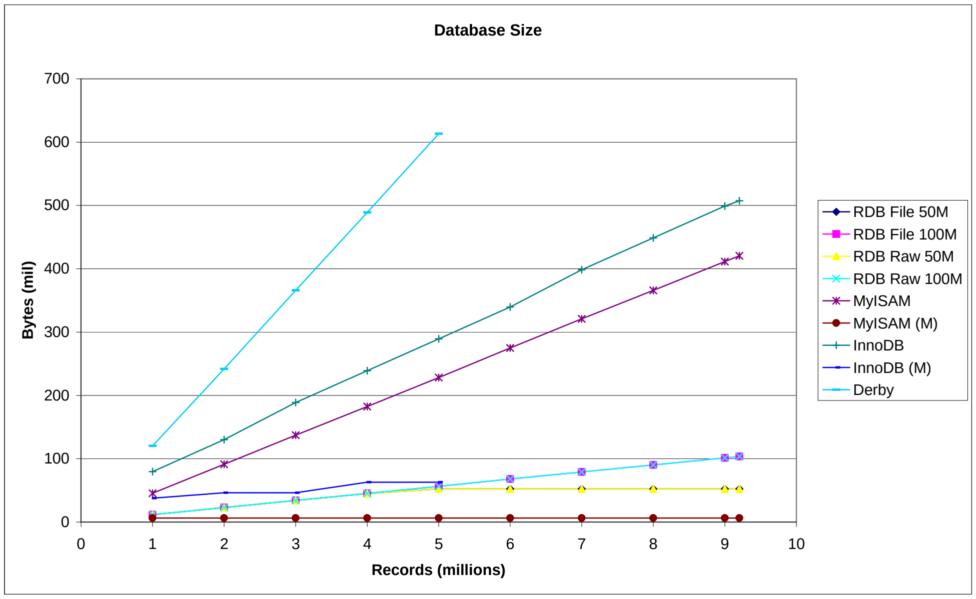
7.3.6 Disk Writes

The benchmark also measures the number of sectors written by the database during the load process. Minimizing the number of writes reduces wear on flash media. The results are shown in figure 13. The transactional RDBMS implementations have much more overhead in this area as compared to MyISAM and RealDB. The winner in this test is the MyISAM format with size-management, possibly because of the smaller overall database size. However, the size-managed version can be perfectly compared to RealDB because the number of active rows is different due to the delete method used with SQL. When comparing the MyISAM version without size management to RealDB, MyISAM takes more disk writes to store the same data. Similar to the conclusion with database size in section 7.3.4, the extra overhead likely comes from the indexing required in a generic relational database, which does not apply to the ordered data environment in RealDB.

The four RealDB variants perform approximately the same number of writes, regardless of size management. Since RealDB's size management overwrites the oldest block with new data, the number of writes is based almost entirely on

Implementation	Data Reduction at 5M	Data Reduction at 9.2M
MyISAM	75.3%	75.4%
InnoDB	80.5%	79.6%
Derby	90.8%	n/t

Table 3: Data Reduction with RealDB



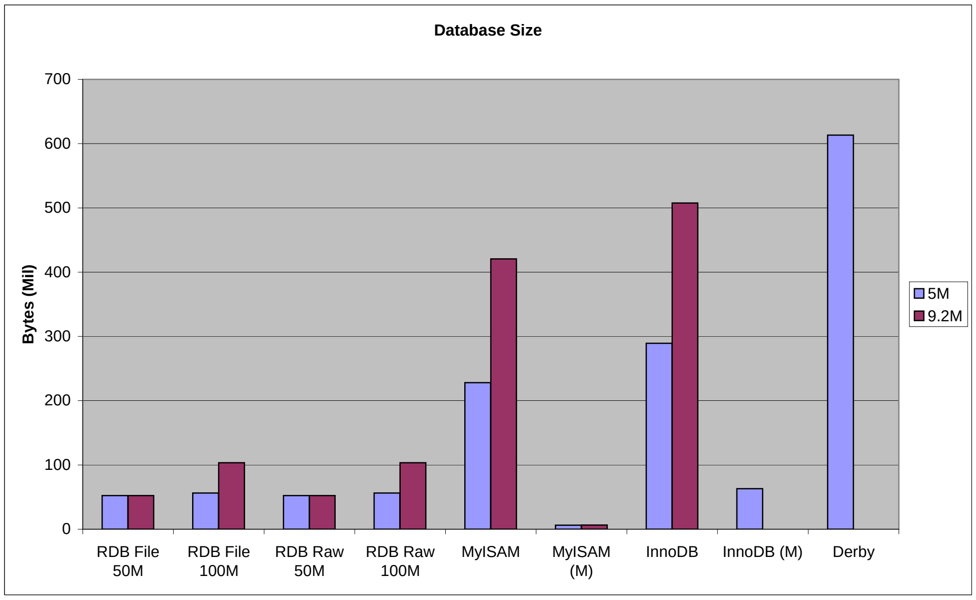
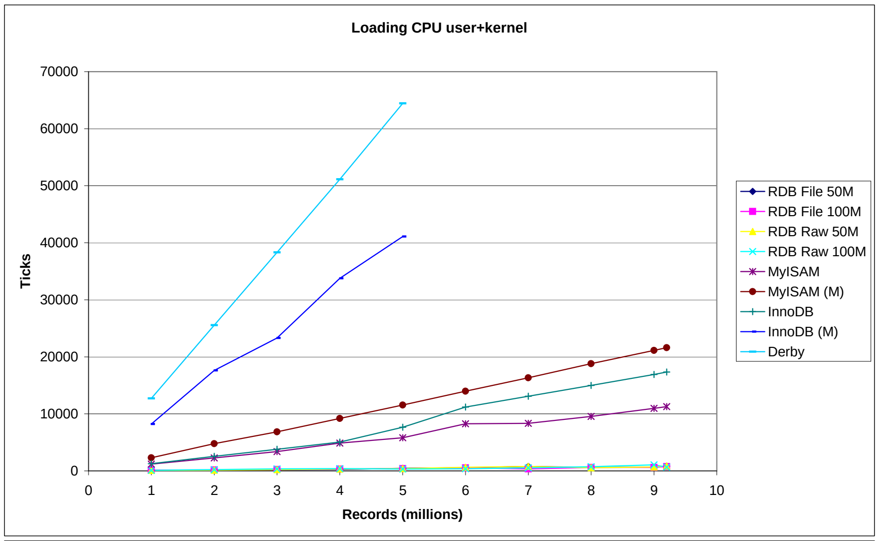


Figure 11: Database Size

Implementation	5M File	9.2M File	5M Raw	9.2M Raw
MyISAM	93.0%	93.3%	94.5%	94.2%
MyISAM (M)	96.1%	96.1%	96.4%	96.0%
InnoDB	94.7%	95.6%	95.8%	96.2%
InnoDB (M)	98.9%	n/t	99.0%	n/t
Derby	99.4%	n/t	99.5%	n/t

Table 4: CPU Utilization Reduction with RealDB



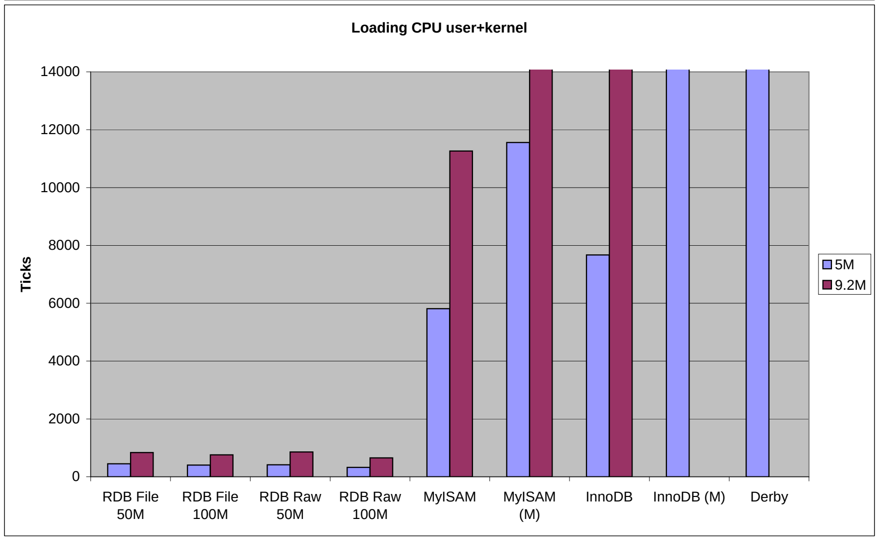
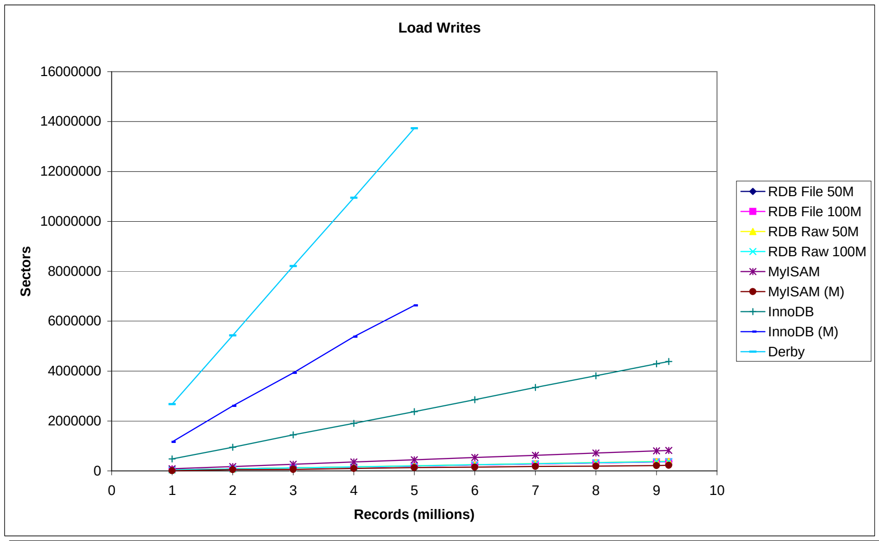


Figure 12: CPU Utilization

the number of records inserted, rather than the capacity or utilization of the database.

Implementation	5M File	9.2M File	5M Raw	9.2M Raw
MyISAM	55.3%	55.4%	53.7%	53.9%
MyISAM (M)	-49.9%	-67.5%	-55.0%	-74.1%
InnoDB	91.6%	91.7%	91.4%	91.4%
InnoDB (M)	97.0%	n/t	96.9%	n/t
Derby	98.6%	n/t	98.5%	n/t

Table 5: Load Writes Reduction with RealDB



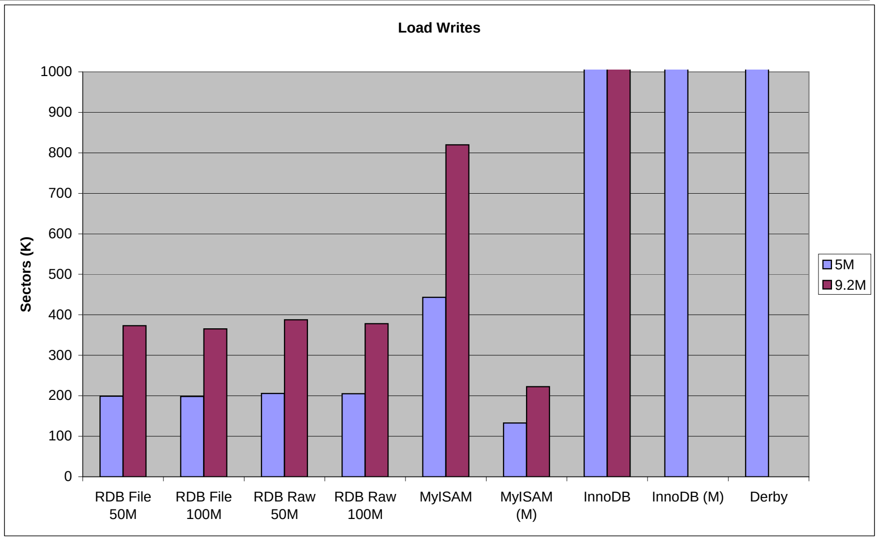


Figure 13: Disk Writes

7.4 Reliability Assumptions

Relaibility metrics were not explicitly measured in this project due to the difficulty of proving that something is not possible. For the conclusions section, reliability is defined as tolerance of faults described in section 1.2 without requiring a manual and/or expensive repair operation. Therefore, the following assumptions are used when comparing the implementations in the conclusions section:

- RealDB is reliable due to its design and testing as described in section 5.2.
- MySQL MyISAM format is not reliable because it is not transactional or advertised to be tolerant of system failures, and producing a corrupted file is very easy via power, software, or storage fault.
- MySQL InnoDB is reliable because it is transactional and advertised to be reliable.
- Derby is reliable because it is transactional, which implies reliability when used in a relational database context.

7.5 Conclusions

Based on the analysis of performance in section 7.3 and the features of each implementation, RealDB achieves its goal of significantly improved performance over the compared RDBMS implementations for the problem of data streams. The trade-off of a narrower focus provided the possibility of large improvements in certain areas, when compared to the top RDBMS implementations:

- Total load time is reduced by 95%-96% compared to the fastest reliable RDBMS implementation, InnoDB.
- Database size is reduced by 75% compared to the smallest RDBMS, MyISAM, and 81% compared to InnoDB, the smallest reliable RDBMS.
- CPU utilization is reduced by 93%-99% when compared to MyISAM and InnoDB.
- Sectors written while loading data is reduced by 54% versus MyISAM and 91% versus InnoDB when the database is large enough to hold all data. Sectors written when size management is required is inconclusive because it was not possible to replicate the same delete methods in SQL as was used in RealDB, but MyISAM may require about half as many writes. InnoDB requires more writes when size management is required.

Creation time differed greatly between the implementations. RealDB on a file system and InnoDB preallocate database files, which takes significant time depending on the file system. In this case FAT32 was used, which cannot quickly create large files of uninitialized content on Linux. RealDB requires linear time with respect to the maximum database size in this case, while InnoDB starts with a file of a configured minimum size. When running on a raw partition without a file system, RealDB can create a database in less than half the time as the next fastest, MyISAM. RealDB's long creation time could be an issue especially on the first startup of an embedded system. With a 4GiB database, initial startup would take almost 16 minutes if using the same file system and hardware as used in this test. Therefore, using RealDB on a file system that can create large, empty files in reasonable, constant time or operating RealDB on a raw partition without a filesystem may be required for a proper experience. An alternative mitigation would be to create the database on the device before first use in the deployed environment.

Another area to note with RealDB's performance as compared to MySQL is the fact that MyISAM without size management employed loaded faster despite more than double the CPU and disk usage of RealDB. This suggests that RealDB is not utilizing the resources as efficiently as MyISAM. Because RealDB is single threaded and uses exclusively synchronous writes, the CPU and disk cannot be utilized at the same time, leaving the disk idle for some periods. It is also possible that the disk write patterns used by RealDB lowers the effective I/O throughput of the flash device.

Outside of performance metrics, when comparing the feature set of the implementations, RealDB has some strong benefits:

- SQL does not provide a way to guarantee that the database stays under some size. In both reliable RDBMS implementations, Derby and InnoDB, even when deleting records, the database's size still increases linearly with inserts. For an embedded system with limited space, this is not acceptable.
- Ability to use a raw partition without a filesystem for storage, eliminating concerns about choosing (or having) a file system reliable against system faults.

However, RealDB has some major disadvantages as compared to RDBMS implementations in general, some due to the tightly focused design, and some due to the current implementation, that could be improved as future work:

- Its design requires the database to be recreated and all data to be reloaded if changes to the file's size or stream structure are required. This could significantly impact upgrades of a deployed embedded system.
- Explicit flushes leave partially written data blocks with empty space, which decreases the data density. The data reduction results of 75%-81% improvement is based on implicit flushing only. It should to be possible to augment the transactional capabilities to fill the remaining space of a data block on a subsequent flush.

A SQL database can store more than just stream data. An embedded system will likely need additional information storage outside of what RealDB can store, for example configuration, user preferences, and identification. However, if only storing a small number of records that rarely change, InnoDB and Derby again become viable choices, while leaving the possibility for a light-weight alternative such as SQLite or just simple flat files. Therefore, RealDB is a viable option for data collection system that delegates the majority of the database work to a highly specialized and scalable system (RealDB) and leaves the smaller remaining work, if any, to a lightweight, generic database implementation.

8 Project Post-Mortem

8.1 Schedule

The original milestone definition and schedule follows, along with actual completion dates:

Milestone 1 First design phase completed, creation of object model and some stub functionality and tests. Setup of environments and compilers, and prototypes for high-risk code.

Milestone 2 Creation of maintenance tools to create an empty database on disk, and simple storage implementation (single stream, no or incomplete space management)

Milestone 3 Completion of database metadata and functionality required for writing, including space management but excluding compressed data storage

Milestone 4 Completed research and implementation of compressed data storage and gathering, reconstruction algorithms, and read functionality including APIs and query tool

Milestone 5 Completion of proof-of-concept use for RealDB

Milestone 6 Design and implementation of RealDB version of performance tool and completed design for solving problem using other solutions

Milestone 7 Completion of performance tool versions for MySQL InnoDB, MySQL MyISAM, and Apache Derby

Target	Planned	Completed
Preproposal	2006-10-17	2006-10-17
Preproposal Presentation	2006-10-17	2006-10-17
Proposal Approved	2008-07-31	2008-08-11
Milestone 1	2008-08-31	2008-08-28
Milestone 2	2008-08-31	2008-09-29
Milestone 3	2008-09-08	2009-01-29
Milestone 4	2008-09-22	2010-02-28
Milestone 5	2008-10-20	2010-04-11
Milestone 6	2008-11-10	2010-07-27
Milestone 7	2008-11-24	2010-07-27
Report	2008-12-15	2010-09-19
Defense	2009-01-15	2010-10-05

8.2 Lessons Learned

- Trying to tackle both tasks of data storage and data post-processing to compress and later rebuild the data was too much. The entire concept of reducing data by downsampling could be a masters-level project of its own. As a result, the codec portion of the project was de-emphasized in favor of data storage. A future project could implement various codec for RealDB and evaluate their cost/benefit (in terms of CPU utilization versus space saved) and effects of the codecs on the ability to analyze the data after recording.

- The method of preserving database state against loss by never overwriting the latest copy of a critical block (alternating between backup blocks) was not sufficient in avoiding transactions entirely. Ultimately, the longest time on the project was spent on the transaction portion, and although it works within the proper complexity bounds, it is not anywhere near as efficient as hoped.

- The unexpected addition of complexity from transactions and reliability was the main technical reason for the delays from the original proposed schedule. The challenges of working life after college classes constitute the overall major delay in the project's completion.

- Version control comments, technical notes, code comments, and code documentation are critical when recalling the current status and next steps to work on in a project with long time periods between development sprints. However, even with strong organization, work in computer science still requires long periods of dedicated time to be most effective. Eight periods of 30 minutes over a couple weeks is much less productive than a single period of four hours, simply because of the time required to pick up where one left off.

- During work on a research project, new knowledge and ideas on different directions come up. Since we are always learning more through our work, the temptation to redesign or shift directions has to be limited to prevent indefinite delays of a project. At a certain point it is simply better to document the new knowledge or idea as a limitation or future work. The future work section notes some of the technical lessions learned during the project that could be solved with more effort or research.

8.3 Future Work

- Ability to modify the database after construction, in particular, add and remove streams.
- Ability for more than one transaction to be outstanding at a time (commit multiple transactions in a single write).
- Performing a flush will write partial data blocks, leaving unused space. With some work it may be possible to touch that data block again to fill it. This is not done now because when re-writing the block to add data, on failure it can get corrupted and lose the already committed data. By saving the original data to a backup in the transaction log we can preserve this space, at the expense of additional writes.
- Consider allowing deletion of data before the database fills up, if there is a possible reason to want this (the database is fixed size).
- Investigation to discover the attributes of RealDB's design that explains why MyISAM beats RealDB despite using more than twice the CPU and disk writes.
- Measurement of memory usage and read speed metrics.
- Improve the block collection algorithm by adding tuning parameters:
  - Select the data block whose latest record is the oldest (to prevent from deleting a very large time span from a stream with a low data density).
  - Don't reclaim a block if it is the only block for the stream, even if it is the oldest, preserving at least the most recent value.
  - Allow a user "plugin" into the process to disallow deletions of data; one example is to stop deletes of data that has not been archived externally.
- Improve integration tests via code coverage analysis.
- Comparison of RealDB to other solutions such as SQLite and non-RDBMS alternatives.

9 References

[1] "Antlr parser generator." [Online]. Available: http://www.antlr.org/

[2] "Apache derby." [Online]. Available: http://db.apache.org/derby/

[3] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom, "Stream: the stanford stream data manager (demonstration description)," in *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press, 2003, pp. 665–665.

[4] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik, "Retrospective on aurora," *The VLDB Journal*, vol. 13, no. 4, pp. 370–383, 2004.

[5] P. Bourke, "Interpolation methods." [Online]. Available: http://local.wasp.uwa.edu.au/~pbourke/miscellaneous/interpolation/

[6] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah, "Telegraphcq: continuous dataflow processing," in *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press, 2003, pp. 668–668.

[7] "Mysql ab :: Mysql 5.0 reference manual :: 12.1.2 space needed for keys." [Online]. Available: http://dev.mysql.com/doc/refman/5.0/en/key-space.html

[8] "Mysql internals myisam - mysqlforge wiki." [Online]. Available: http://forge.mysql.com/wiki/MySQL_Internals_MyISAM#The_.MYI_file

[9] "Mysql :: Mysql 5.1 reference manual :: 1 general information." [Online]. Available: http://dev.mysql.com/doc/refman/5.1/en/introduction.html

[10] "Mysql :: Mysql 5.1 reference manual :: 25.1 libmysqld, the embedded mysql server library." [Online]. Available: http://dev.mysql.com/doc/refman/5.1/en/libmysqld.html

[11] "Mysql :: Mysql 5.5 reference manual :: 4.6.3.6 myisamchk memory usage." [Online]. Available: http://dev.mysql.com/doc/refman/5.5/en/myisamchk-memory.html

[12] "Java db." [Online]. Available: http://www.oracle.com/technetwork/java/javadb/overview/index.html

[13] Office of Naval Research, "Defense systems modernization and sustainment," Rochester Institute of Technology/Nasr, Nabil, Unpublished Grant N00014-07-1-0823, 3/07 - 9/11.

[14] E. Sciore, *Database Design and Implementation.* John Wiley & Sons, Inc., 2009, pp. 377,379.

[15] E. Sciore, *Database Design and Implementation.* John Wiley & Sons, Inc., 2009, ch. 14.