RealDB Master's Project Proposal

Jason Winnebeck

8th May 2008

Contents

1 Problem **1**
- 1.1 High-Level Requirements 2

2 Other Solutions **3**
- 2.1 MySQL 3
  - 2.1.1 MySQL MyISAM 4
  - 2.1.2 MySQL InnoDB 4
- 2.2 Apache Derby 5

3 Detailed Requirements and Functional Specification **5**
- 3.1 Operating Systems 5
- 3.2 Data Streams 6
  - 3.2.1 All Data Streams 6
  - 3.2.2 Continuous Data Streams 6
- 3.3 Data Gathering 7
  - 3.3.1 Sampled Data Gathering 7
  - 3.3.2 Compressed Data Gathering 8
  - 3.3.3 Native Data Gathering 8
- 3.4 Fault Tolerance 9
- 3.5 Time and Memory 9
- 3.6 Size Management 9

4 Design and Architecture **9**
- 4.1 Overall Design Decisions 9
- 4.2 Storage Format 9
- 4.3 Transactions and Buffering 10
- 4.4 Detailed Design 10
  - 4.4.1 Working Example 11
- 4.5 Reliability 13

5 Deliverables **15**

6 Schedule **16**

7 References **17**

Abstract

RealDB (RDB) will be a real-time embeddable database system for data streams. It is a non-relational database that is based on a changing value of a stream of sequential, timestamped data. To support this environment, the system will have a low overhead and deliver high performance; the trade-off is a narrower approach and less guarantees than a traditional relational database management system (RDBMS) would deliver.

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 high-speed sensor data from a sensor network consisting of sensors with little or no "intelligence." Because the platform is isolated, an embedded computer listens to the sensor network to collect and store the data. Also, to support advanced maintenance tasks or periodic data synchronization, the computer needs to support 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, and this is where RealDB comes in.

To achieve the goals set out for RealDB, we make assumptions about its environment and usage. Then, based on the problem and environment we develop a set of requirements for the solution, which define the trade-offs made to improve the suitablity over solutions involving tradtional relational database management systems (RDBMS). Some of the assumptions and requirements below reference ACID (Atomicity, Consistency, Isolation, Durability)[8] concepts that are commonly used as benchmarks in relational databases.

1. Assumptions about the environment that limit 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. By embedded, this means:
      i. No interactive interface to the user, or perhaps no end-user interface at all.
      ii. Low-powered processor, such as an ARM or Geode 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 system clock always moves forward or stands still for insignificant (compared to sample rate) amounts of time while the system is running. 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 streams, each operation stands alone, eliminating most concerns about atomicity.

(c) Losing recently written data is acceptable when it trades for fault-tolerance and fast 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.

RealDB does not need to be a complete data storage solution in every environment; RealDB's goal is in data recording. If extensive post-processing is required, data will 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[1]. 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 that system for archiving the data. Therefore, this work will not attempt to develop a processing framework or a specialized query language. While a query tool supporting rudimentary SQL-like syntax for offline queries will be provided as a deliverable as mentioned in section 5, the goal for this tool is mostly to support developers that would use the RealDB database.

There do exist SPEs and stream databases that have goals similar to RealDB, such as TelegraphCQ[3] and STREAM[2]. TelegraphCQ is based on the open source PostgreSQL RDMBS 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 Requirements

Based on these assumptions, we develop the following requirements and metrics. One note to take is that the number of streams and other database metadata defined by the user is fixed when the program begins, so complexities assume that memory use and time is constant with respect to the number of streams in the system.

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.

2. In other words, a power failure should only lose the last X amount of data but not corrupt the entire database or any indexes. Durability is guaranteed for all but the last X amount of data.

3. RealDB should be scalable to support data sets that are large on hardware with limited capabilities. Therefore, inserting new data should take constant O(1) time and memory with respect to the amount of data previously stored in the database. Retrieving ranges of data should take O(log n) time or better time to start returning records, and must stream data, so that memory use is O(1).

4. Since solid-state flash (NAND memory) is likely to be used in an embedded environment, RealDB should try to minimize 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.

5. Since the device will have limited storage capacity, the database should be compact, defined by the number of bytes to represent a particular set of stream data.

6. Since the device is an embedded system with a low-powered CPU, the database should use minimal processing power, defined by the ratio of CPU time the operating system must devote to the database management thread(s) to write and read a particlar set of stream data.

2 Other Solutions

This section compares other database solutions to the vision for RealDB, and summarizes major reasons why those solutions do not solve the problem, or do not solve it in the best way. RealDB's ability to archive data in the embedded environment will be evaluated against these traditional RDMBS.

2.1 MySQL

MySQL AB 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.[6]

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 solve most of 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++ and requires all code to be GPL, unless a license is purchased from MySQL AB.[7]

2.1.1 MySQL MyISAM

The MyISAM table format[5] is probably the most suited table format for the problem of the solutions presented in this section. This table format is based off of the ISAM table format and is highlighted as a close solution because of its simplicity. Each MyISAM table consists of three files: 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, desirable for a stream database.

The main drawback to this approach is that 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. By using a buffer, open, commit, close cycle we can dramatically decrease the processing overhead and corruption possibility, but it is possible to improve on this format for the particulars of the problem at hand. Also, the indexing required to store our data could generate an index that is larger than the data itself; in fact, the worst-case scenario for the index is $(keylength + 4)/0.67$ for records inserted in order.[4]

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. Effectively, 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 transactions in terms of speed, space, and wear on flash memory devices. Another issue is that when storing the same amount of data, we suspect based on anecdotal evidence that InnoDB will take considerably more space, and deleting records does not always predictably free space. The performance and suitability of InnoDB will be measured as part of this project.

2.2 Apache Derby

Apache Derby[9] is an embeddable Java database, and recently Sun Microsystems includes a rebranded version of it called "Java DB" into the latest Java runtimes[10]. 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. A significant amount of current official information on fault tolerance, recovery, and performance is not available, so these parameters will be measured as part of the project.

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

RealDB will be capable of running under at least the following operating systems:

- Linux 2.6
- Windows desktop (XP and Vista)

In order to be as cross-platform as possible and address these needs, RealDB will be implemented using Java. Running under Java 2 Standard Edition embedded or desktop editions should be an option. Compatibility with Linux is chosen because they are popular and embeddable. Windows desktop support is chosen due to its popularity on developer workstations. Native code should be kept to a minimum to enhance portability. An attempt will be made to run the software on the Windows CE platform; to make this feasible, compatibility with alternative Java virtual machines, GCJ (GNU Compiler for Java), and libraries such as GNU Classpath will be considered. GCJ support is useful as the application can be precompiled into native code, which could considerably reduce the overhead of the Java application, particularly in start up time, for embedded machines. GNU Classpath is an open-source implementation of the standard libraries used in GCJ and other products. Sun's implementation of Java is supplied only on a few platforms, so maintaining compatibility with alternative implementations is important to make ports for other embedded environments as easy as possible.

3.2 Data Streams

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

1. unknown
2. not yet recorded
3. specified value

There are two types of data streams supported by RealDB: continuous and sampled.

A continuous data stream is in the abstract sense an entity that always has some value at a given time. Because of the special values, "unknown" and "not yet recorded," a continuous data 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. For a given point in time, a data stream's state can only change from "not yet recorded" to another state. This limitation is acceptable because RealDB is a recording system and allows RealDB to be simpler to support the environment requirements.

Sampled data streams work in a similar way, except that the data stream has a value only at the time of recorded samples. For other times, the data stream is simply "unspecified".

3.2.1 All Data Streams

1. User-definable types, but records will be fixed size. The user defines the data type by combining the following primitive types to form a set of 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).
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 unsigned 16 bit integer.

3.2.2 Continuous Data Streams

1. 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 will come with some predefined algorithms:

(a) Last value (step).

   i. 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, return the latest point whose time is earlier than the requested time.

(b) Last value with range (step)

   i. Example: speed is 25 and deadband is 0.5. For this algorithm, the deadband is the amount a value must change by in order for the change to be significant (and subsequently stored). 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, return the latest point whose time is earlier than the requested time. The original value of the stream will be preserved within an error level as determined by the deadband.

(c) Linear

   i. Example: speed is 25 with slope of 1 per second and a deadband of 0.5. If a sample of 26 arrives one second later, do not store it, but if a sample of 25.4 arrives one second later, do store it. When interpolating, find the latest point whose time is earlier than the requested time, and extrapolate based on the time requested, the previous value and time, and the slope.

2. Users can provide custom compression and reconstruction algorithms. For example, it should be possible to implement a bezier reconstruction function.

There will always be a trade-off in the choice of algorithm. More complicated algorithms could store less points, but at the cost of more bytes per point and CPU utilization.

3.3 Data Gathering

RealDB will support the ability to gather data from a data stream. Data gathering is done on ranges of time. There are three main modes for data gathering: sampled, compressed and native. For all queries, RealDB will provide 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 Sampled Data Gathering

Sampled data gathering from a data stream is appropriate for applications that desire simplicity in their data. Sampled data will be able to be retrieved by specifying a start and end of a time range, and a sampling speed. RealDB will return data points that sample the reconstructed value of the data stream.

3.3.2 Compressed Data Gathering

Compressed data gathering returns smart objects that correspond to the native records, except that they represent 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)
- Integral (by element)
- Limit of X records to return; if specified, streams the chronologically first X records and stops.

The number of time ranges returned is a function of the compression and reconstruction algorithm.

3.3.3 Native Data Gathering

Native data gathering provides a method to obtain the direct records stored by the compression and reconstruction algorithm. This method is useful for database synchronization. Users ask RealDB for records between a given start and end time and only records within that time range are returned, without modification.

Parameters to native data gathering queries:

- Start Time (may be inclusive)
- End Time (always inclusive)
- Is start time inclusive or exclusive?
- Limit of X records to return; if specified, streams the chronologically first X records and stops.

3.4 Fault Tolerance

RealDB will attempt to be resilient to faults introduced by power loss. However, there can be limitations to the resilience based on the operating system chosen and the file system. RealDB will try to be as resilient as possible while still maintaining portability – not trying to rely on operating system and file system implementation details where possible and limiting the use of native code. If a transactional system is required, it should limit the amount of data written to the disk to limit the wear on flash memory, if used.

3.5 Time and Memory

The system will be able to add records to the database in O(1) time, and use O(1) memory when adding or retrieving records. The system will require O(n) memory for the number of data streams, but this amount of memory will be constant after startup. For reading records, O(log n) time will be required, based on the number of samples stored for the stream being read. This will allow the database to scale to arbitrarily large data sets.

3.6 Size Management

To fit onto a disk of finite size, RealDB will have configuration to keep size limits on a per-database basis. RealDB will delete the oldest records in a LIFO manner. The design and architecture section will describe the algorithms that control size management in more detail.

4 Design and Architecture

4.1 Overall Design Decisions

Due to the assumptions made on the environment, the resulting benefit is the ability to meet ideal O(1) time and memory to write records, when considering the number of records in the database. Data comes in sorted order, eliminating much of the time and space overhead to keep an effective index.

4.2 Storage Format

Because most file systems will not support RealDB's requirements through abilities such as truncating a file from the front or giving fine control over reliability, a storage allocation layer will be introduced into the design. Careful attention to the design of the allocation layer is required to make sure to keep the O(1) insert and delete complexity requirement (remembering that an insert may also require a delete if the database is at capacity), and to keep transactions to a minimum.

The practical effect to the end-user is that RealDB will be configured to point at a file with a preset size. In Linux this could even be a device file, such as a raw,

unformatted partition like `/dev/hda1`. Having a preallocated file is appropriate in an embedded environment as the space allocated to certain tasks is often fixed to make sure that resources are always available. Additionally, the hope is that a preallocated file will reduce or eliminate the possibility of additional corruption from the host file system as RealDB cannot make guarantees about how power failures or other faults will affect file structure.

4.3 Transactions and Buffering

The main goal of buffering and transactions in RealDB is to prevent old data from getting corrupted and marking the minimal amount of data to discard (if necessary) should a system failure occur. This is in contrast to the usage of transactions in many traditional relational database systems, which preserves referential integrity and user-level atomic modifications. RealDB is not a relational database and is limited to a single-process, which allows for room to optimize with respect to traditional systems. Transactions place a bigger toll on solid-state memory devices by wearing them faster through additional erase and write cycles, so transactions required by RealDB should cause the fewest writes required to work towards the ideal of 1 byte written for every byte of data.

4.4 Detailed Design

RealDB breaks the database's data file into two portions, a metadata portion, and a data portion. The metadata portion serves the configuration, indexing and transactional needs of the software, and the data portion stores actual record data for the streams. The metadata portion holds specifically the following items:

- Number of metadata blocks
- Number of data blocks (position is calculated from known metadata blocks)
- Number of redundant metadata blocks
- Number of data streams
- Size of a metadata block, in bytes
- Size of a data block, in bytes
- Data stream information:
  - Format specification
  - List of allocated blocks, in time order; this provides a mapping of stream-logical block to data file-logical block
- The data block number of the next free block

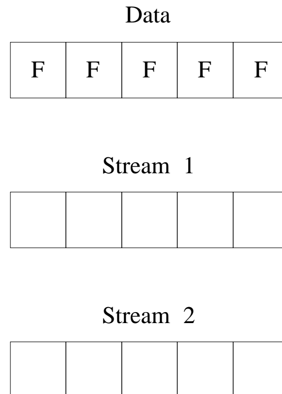

Figure 1: Initial system state

The data portion of the data file stores only the actual records for the data streams. The allocation map provides a mapping between the data file blocks and a linear address space for each data stream. This makes it possible, for example, to reference record 26 in data stream 2 with a O(1) time lookup. There is no defragmentation of records in a stream's address space, nor in the data portion of the data file; with this design the data is 100% packed when the database is at capacity for all blocks excluding the blocks currently being written to (as they are not yet full).

4.4.1 Working Example

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.

When data is written to the stream, first the current block in use for that stream is written until it is full. The physical writing only occurs when a buffer flush happens, which is performed in a sort of transactional style (discussed in detail later). However, in memory the data model changes are immediate, so the writing position advances in the block until it is full. When more data needs to be written, the database first checks to see if there are any blocks left in the free list, and if there are any, allocates the first available block. 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.

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

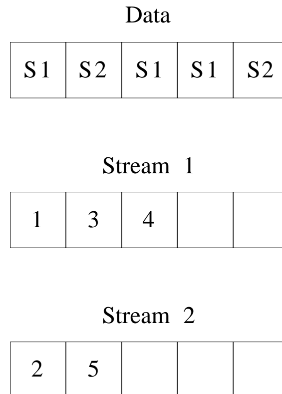

Figure 2: Database just filled to capacity

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 list 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. Stated more precisely, the database reclaims the block with the lowest maximum time stamp value. 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. It is expected that the number of streams will be small enough such that this information can be very easily cached in memory. This process still meets the RealDB goal that the time to write records does not increase as the number of data points grows.

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

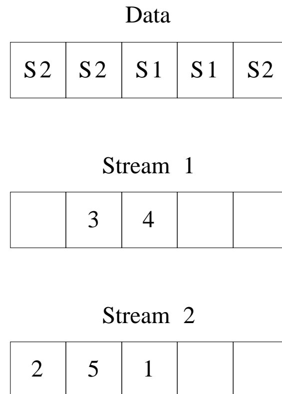

Figure 3: State of database after reclaiming a block from stream 1

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.

4.5 Reliability

The previous sections discussed the abstract theory behind managing the data; however, a key requirement for RealDB is to maintain its 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 will be the main challenge of the project. The key assumptions about the environment will be the key to performing this task, most specifically the aspect that it is acceptable to lose the newest "x" seconds (or "x" amount) of records when loss occurs, if this loss is necessary to ensure that there is no corruption or loss of previously committed data and a consistent startup time. The consistent startup time is key here – for example MyISAM database format is normally well suited for this problem, but when it gets corrupted the entire table needs to be scanned to rebuild the index. This requires a very long amount of time and requires 50% of the disk

to be reserved for the recovery operation as the tables are rewritten. RealDB aims to improve this situation by segmenting the database to limit recovery effort only to the most recently active segment (the open blocks). This puts a constant bound on the recovery time. MyISAM is not able to achieve this as it does not assume data to be written only in order.

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 mentioned in the problem, most specifically in this case delayed commits and ordered writes. RealDB will allow the administrator to configure the system to commit the database triggered by time, space, or both. For example the administrator might say to write the database every 30 seconds, or if 64K of records are pending. These options bound the memory requirements as well as the amount of data that can be lost.

In order to design something that is reliable within a preallocated file, we will try to determine the worst-case scenario from a hardware point of view, if power is lost. There are various 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. So we realistically hope that 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 anyway. While RealDB's expected environment has solid-state storage, scenario 3 I also hope and expect 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 we assume is not present on the system.

5 Deliverables

RealDB implementation The implementation will consist of open-source code written in the Java language, that implements the RealDB storage engine as meeting the requirements in this proposal and implementing the design described. It also contains support and query APIs to allow Java programs to interact with the engine and the data streams contained within.

RealDB maintenance tools A set of programs to perform administration and maintenance tasks on RealDB databases. At a minimum, this will include a tool or tools to create and configure a database.

RealDB query tool An interactive console-based program for querying a RealDB database that implements a minimal subset of SQL and SQL-like syntax for querying data streams.

RealDB performance tools A set of tools to support the experimental portion of this project. These tools will implement benchmarks of the functions in RealDB, and perform benchmarks of equivalent implementations/configurations of other products for comparison. Metrics will be measured by examining the program output and metric parameters (such as I/O activity and CPU times) in the operating system as the benchmarks run.

RealDB user manual Includes documentation on the RealDB API as generated by Javadoc, and instructions on how to use the maintenance, query, and performance tools.

Final report Covers all topics in this proposal based on the final state of development in the project, covers design and function in more detail, and provides results of how well RealDB and other solutions under study listed in section 2, with the exception of Echo Historian, meet each requirement and perform in each metric listed in section 1.1. Meeting requirements will be done by analysis of the solution, and by testing where possible.

The RealDB implementation, maintenance tools, query tool, and content for the user manual will be developed first, in parallel. Then, the performance tool will be written and run against RealDB itself and the other solutions under study. Finally, a final report will be written with the performance evaluation and the final copy of the user manual.

6 Schedule

Target	Planned	Completed
Preproposal	2006-10-17	2006-10-17
Preproposal Presentation	2006-10-17	2006-10-17
Proposal Approved	2008-04-30	
Implementation	2008-07-31	
Report	2008-08-15	
Defense	2008-09-12	

7 References

[1] 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.

[2] 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.

[3] 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.

[4] "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

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

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

[7] "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

[8] "Acid - wikipedia, the free encyclopedia." [Online]. Available: http://en.wikipedia.org/wiki/ACID

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

[10] "Java db from sun microsystems." [Online]. Available: http://developers.sun.com/javadb/