LLADD: An Extensible Transactional Storage Layer
(yaahd)

Russell Sears and Eric Brewer
UC Berkeley
{*sears,brewer*}*@cs.berkeley.edu, http://lladd.sourceforge.net*

Abstract

Although many systems provide transactionally consistent data management, existing implementations are generally monolithic and tied to a higher-level DBMS, limiting the scope of their usefulness to a single application or a specific type of problem. As a result, many systems are forced to "work around" the data models provided by a transactional storage layer. Manifestations of this problem include "impedance mismatch" in the database world and the limited number of data models provided by existing libraries such as Berkeley DB. In this paper, we describe a light-weight, easily extensible library, LLADD, that allows application developers to develop scalable and transactional application-specific data structures. We demonstrate that LLADD is simpler than prior systems, is very flexible and performs favorably in a number of micro-benchmarks. We also describe, in simple and concrete terms, the issues inherent in the design and implementation of robust, scalable transactional data structures. In addition to the source code, we have also made a comprehensive suite of unit-tests, API documentation, and debugging mechanisms publicly available.[1]

1 Introduction

Changes in data models, consistency requirements, system scalability, communication models and fault models require changes to the storage and recovery subsystems of modern applications.

For applications that are willing to store all of their data in a DBMS, and access it only via SQL, existing databases are just fine and LLADD has little to offer. However, for those applications that need more direct management of data, LLADD offers a layered architecture that enables simple but robust data management.[2] We also believe that LLADD is applicable in the context of new, special-purpose database systems such as XML databases, streaming databases, and extensible/semantic file systems [17, 6]. These form a fruitful area of current research, [20] but existing monolithic database systems tend to be a poor fit for these new areas.

The basic approach of LLADD, taken from ARIES [13], is to build *transactional pages*, which enables recovery on a page-by-page basis, despite support for high concurrency and the minimization of disk seeks during commit (by using a log). We show how to build a variety of useful data managers on top of this layer, including persistent hash tables, lightweight recoverable virtual memory (LRVM) [19], and simple databases. We also cover the details of crash recovery, application-level support for transaction abort and commit, and latching for multi-threaded applications. Finally, we discuss the shortcomings of a few common applications, and explain why LLADD provides an appropriate solution to these problems.

Many implementations of transactional pages exist in industry and in the literature. Unfortunately, these algorithms tend either to be straightforward and unsuitable for real-world deployment, or are robust and scalable, but achieve these properties by relying upon intricate sets of internal and often implicit interactions. The ARIES algorithm falls into the second category, and has been extremely successful as part of the IBM DB2 database system. It provides performance and reliability that is comparable to that of current commercial and open-source products. Unfortunately, while the algorithm is conceptually simple, many subtleties arise in its implementation. We chose ARIES as the basis of LLADD,

[1] http://lladd.sourceforge.net/

[2] A large class of such applications are deemed "navigational" in the database vocabulary, as they directly navigate data structures rather than perform set operations.

and have made a significant effort to document these interactions. Although a complete discussion of the ARIES algorithm is beyond the scope of this paper, we will provide a brief overview and explain the details that are relevant to developers that wish to extend LLADD.

By documenting the interface between ARIES and higher-level primitives such as data structures and by structuring LLADD to make this interface explicit in both the library and its extensions, we hope to make it easy to produce correct and efficient durable data structures. In existing systems (and indeed, in earlier versions of LLADD), the implementation of such structures is extremely complicated, and subject to the introduction of incredibly subtle errors that would only be evident during crash recovery or at other inconvenient times. Thus there is great value in reusing lower layers.

Finally, by approaching this problem by implementing a number of simple modules that "do one thing and do it well", we believe that LLADD can provide competitive performance while making future improvements to its core implementation significantly easier. In order to achieve this goal, LLADD has been split into a number of modules forming a *core library*, and a number of extensions called *operations* that build upon the core library. Since each of these modules exports a stable interface, they can be independently improved.

1.1 Prior Work

An extensive amount of prior work covers the algorithms presented in this paper. Most fundamentally, systems that provide transactional consistency to their users generally include a number of common modules. Figure 1 presents a high-level overview of a typical system.

Many systems make use of transactional storage that is designed for a specific application or set of applications. LLADD provides a flexible substrate that allows such systems to be developed easily. The complexity of existing systems varies widely, as do the applications for which these systems are designed.

On the database side of things, relational databases excel in areas where performance is important, but where the consistency and durability of the data are crucial. Often, databases significantly outlive the software that uses them, and must be able to cope with changes in business practices, system architectures, etc. [4]

Object-oriented databases are more focused on facilitating the development of complex applications that require reliable storage and may take advantage of less-flexible, more efficient data models, as they often only interact with a single application, or a handful of variants of that application. [10]

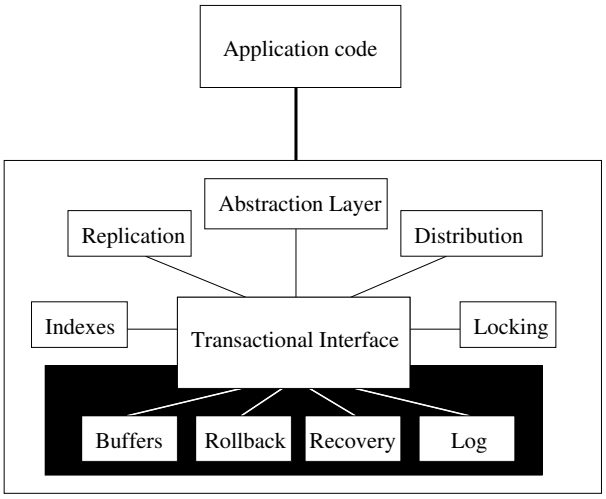

Figure 1: Conceptual view of a modern transactional application. Current systems include high-level functionality, such as indices and locking, but are not designed to allow developers to replace this functionality with application-specific modules.

Databases are designed for circumstances where development time may dominate cost, many users must share access to the same data, and where security, scalability, and a host of other concerns are important. In many, if not most circumstances these issues are less important, or even irrelevant. Therefore, applying a database in these situations is likely overkill, which may partially explain the popularity of MySQL [16], which allows some of these constraints to be relaxed at the discretion of a developer or end user.

Still, there are many applications where MySQL is too inflexible. In order to serve these applications, a host of software solutions have been devised. Some are extremely complex, such as semantic file systems, where the file system understands the contents of the files that it contains, and is able to provide services such as rapid search, or file-type specific operations such as thumb-nailing, automatic content updates, and so on. Others are simpler, such as Berkeley DB, [18, 2] which provides transactional storage of data in unindexed form, or in indexed form using a hash table or tree. LRVM is a version of malloc() that provides transactional memory, and is similar to an object-oriented database but is much lighter weight, and more flexible [19].

Finally, some applications require incredibly simple, but extremely scalable storage mechanisms.

Cluster hash tables are a good example of the type of system that serves these applications well, due to their relative simplicity, and extremely good scalability characteristics. Depending on the fault model on which a cluster hash table is implemented, it is quite plausible that key portions of the transactional mechanism, such as forcing log entries to disk, will be replaced with other durability schemes, such as in-memory replication across many nodes, or multiplexing log entries across multiple systems. This level of flexibility would be difficult to retrofit into existing transactional applications, but is often important in the environments in which these applications are deployed.

We have only provided a small sampling of the many applications that make use of transactional storage. Unfortunately, it is extremely difficult to implement a correct, efficient and scalable transactional data store, and we know of no library that provides low-level access to the primitives of such a durability algorithm. These algorithms have a reputation of being complex, with many intricate interactions, which prevent them from being implemented in a modular, easily understandable, and extensible way.

Because of this, many applications that would benefit from transactional storage, such as CVS and many implementations of IMAP, either ignore the problem, leaving the burden of recovery to system administrators or users, or implement ad-hoc solutions that employ complex, application-specific storage protocols in order to ensure the consistency of their data. This increases the complexity of such applications and often provides only a partial solution to the transactional storage problem, resulting in erratic and unpredictable application behavior.

In addition to describing a flexible implementation of ARIES, a well-tested "industrial strength" algorithm for transactional storage, this paper outlines the most important interactions in ARIES (that is, the ones that could not or should not be encapsulated within our implementation), and gives the reader a sense of how to use the primitives the library provides.

2 ARIES from an Operation's Perspective

Instead of providing a comprehensive discussion of ARIES, we will focus upon those features of the algorithm that are most relevant to a developer attempting to add a new set of operations. Correctly implementing such extensions is complicated by concerns regarding concurrency, recovery, and the possibility that any operation may be rolled back at runtime.

We first sketch the constraints placed upon operation implementations, and then describe the properties of our implementation that make these constraints necessary. Because comprehensive discussions of write ahead logging protocols and ARIES are available elsewhere, [9, 13] we only discuss those details relevant to the implementation of new operations in LLADD.

2.1 Properties of an Operation

A LLADD operation consists of some code that manipulates data that has been stored in transactional pages. These operations implement high-level actions that are composed into transactions. They are implemented at a relatively low level, and have full access to the ARIES algorithm. Applications are implemented on top of the interfaces provided by an application-specific set of (potentially reusable) operations. This allows the the application, the operation, and LLADD itself to be independently improved.

Since transactions may be aborted, the effects of an operation must be reversible. Furthermore, aborting and committing transactions may be interleaved, and LLADD does not allow cascading aborts,[3] so in order to implement an operation, we must implement some sort of locking, or other concurrency mechanism that isolates transactions from each other. LLADD only provides physical consistency; due to the variety of locking systems available, and their interaction with application workload, [1] we leave it to the application to decide what sort of transaction isolation is appropriate.

For example, it is relatively easy to build a strict two-phase locking lock manager [8] on top of LLADD, as needed by a DBMS, or a simpler lock-per-folder approach that would suffice for an IMAP server. Thus, data dependencies among transactions are allowed, but we still must ensure the physical consistency of our data structures, such as operations on pages or locks.

Also, all actions performed by a transaction that committed must be restored in the case of a crash, and all actions performed by aborting transactions must be undone. In order for LLADD to arrange for this to happen at recovery, operations must produce

[3] That is, by aborting, one transaction may not cause other transactions to abort. To understand why operation implementors must worry about this, imagine that transaction A split a node in a tree, transaction B added some data to the node that A just created, and then A aborted. When A was undone, what would become of the data that B inserted?

log entries that contain all information necessary for undo and redo.

An important concept in ARIES is the "log sequence number" or LSN. An LSN is essentially a virtual timestamp that goes on every page; it marks the last log entry that is reflected on the page, and implies that all previous log entries are also reflected. Given the LSN, LLADD calculates where to start playing back the log to bring the page up to date. The LSN goes on the page so that it is always written to disk atomically with the data on the page.

ARIES (and thus LLADD) allows pages to be *stolen*, i.e. written back to disk while they still contain uncommitted data. It is tempting to disallow this, but to do so has serious consequences such as a increased need for buffer memory (to hold all dirty pages). Worse, as we allow multiple transactions to run concurrently on the same page (but not typically the same item), it may be that a given page *always* contains some uncommitted data and thus could never be written back to disk. To handle stolen pages, we log UNDO records that we can use to undo the uncommitted changes in case we crash. LLADD ensures that the UNDO record is durable in the log before the page is written back to disk and that the page LSN reflects this log entry.

Similarly, we do not force pages out to disk every time a transaction commits, as this limits performance. Instead, we log REDO records that we can use to redo the change in case the committed version never makes it to disk. LLADD ensures that the REDO entry is durable in the log before the transaction commits. REDO entries are physical changes to a single page ("page-oriented redo"), and thus must be redone in the exact order.

One unique aspect of LLADD, which is not true for ARIES, is that *normal* operations use the REDO function; i.e. there is no way to modify the page except via the REDO operation. This has the great property that the REDO code is known to work, since even the original update is a "redo". In general, the LLADD philosophy is that you define operations in terms of their REDO/UNDO behavior, and then build the actual update methods around those.

Eventually, the page makes it to disk, but the REDO entry is still useful: we can use it to roll forward a single page from an archived copy. Thus one of the nice properties of LLADD, which has been tested, is that we can handle media failures very gracefully: lost disk blocks or even whole files can be recovered given an old version and the log.

2.2 Normal Processing

Operation implementors follow the pattern in Figure 2, and need only implement a wrapper function ("Tset()" in the figure, and register a pair of redo and undo functions with LLADD. The Tupdate function, which is built into LLADD, handles most of the runtime complexity. LLADD uses the undo and redo functions during recovery in the same way that they are used during normal processing.

The complexity of the ARIES algorithm lies in determining exactly when the undo and redo operations should be applied. LLADD handles these details for the implementors of operations.

2.2.1 The buffer manager

LLADD manages memory on behalf of the application and prevents pages from being stolen prematurely. Although LLADD uses the STEAL policy and may write buffer pages to disk before transaction commit, it still must make sure that the UNDO log entries have been forced to disk before the page is written to disk. Therefore, operations must inform the buffer manager when they write to a page, and update the LSN of the page. This is handled automatically by the write methods that LLADD provides to operation implementors (such as writeRecord()). However, it is also possible to create your own low-level page manipulation routines, in which case these routines must follow the protocol.

2.2.2 Log entries and forward operation (the Tupdate() function)

In order to handle crashes correctly, and in order to undo the effects of aborted transactions, LLADD provides operation implementors with a mechanism to log undo and redo information for their actions. This takes the form of the log entry interface, which works as follows. Operations consist of a wrapper function that performs some pre-calculations and perhaps acquires latches. The wrapper function then passes a log entry to LLADD. LLADD passes this entry to the logger, *and then processes it as though it were redoing the action during recovery*, calling a function that the operation implementor registered with LLADD. When the function returns, control is passed back to the wrapper function, which performs any post processing (such as generating return values), and releases any latches that it acquired.

This way, the operation's behavior during recovery's redo phase (an uncommon case) will be identical to the behavior during normal processing, making it easier to spot bugs. Similarly, undo and redo

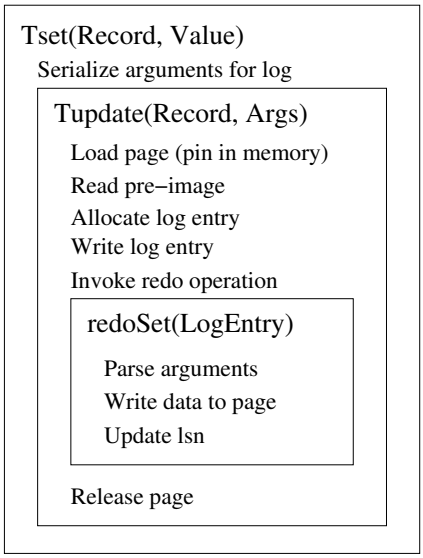

Figure 2: Runtime behavior of a simple operation. Tset() and redoSet() are extensions that implement a new operation, while Tupdate() is built in. New operations need not be aware of the complexities of LLADD.

operations take an identical set of parameters, and undo during recovery is the same as undo during normal processing. This makes recovery bugs more obvious and allows redo functions to be reused to implement undo.

Although any latches acquired by the wrapper function will not be reacquired during recovery, the redo phase of the recovery process is single threaded. Since latches acquired by the wrapper function are held while the log entry and page are updated, the ordering of the log entries and page updates associated with a particular latch will be consistent. Because undo occurs during normal operation, some care must be taken to ensure that undo operations obtain the proper latches.

2.3 Recovery

In this section, we present the details of crash recovery, user-defined logging, and atomic actions that commit even if their enclosing transaction aborts.

2.3.1 ANALYSIS / REDO / UNDO

Recovery in ARIES consists of three stages, analysis, redo and undo. The first, analysis, is implemented by LLADD, but will not be discussed in this paper. The second, redo, ensures that each redo entry in the log will have been applied to each page in the page file exactly once. The third phase, undo, rolls back any transactions that were active when the crash occurred, as though the application manually aborted them with the "abort" function call.

After the analysis phase, the on-disk version of the page file is in the same state it was in when LLADD crashed. This means that some subset of the page updates performed during normal operation have made it to disk, and that the log contains full redo and undo information for the version of each page present in the page file.[4] Because we make no further assumptions regarding the order in which pages were propagated to disk, redo must assume that any data structures, lookup tables, etc. that span more than a single page are in an inconsistent state. Therefore, as the redo phase re-applies the information in the log to the page file, it must address all pages directly.

This implies that the redo information for each operation in the log must contain the physical address (page number) of the information that it modifies, and the portion of the operation executed by a single redo log entry must only rely upon the contents of the page that the entry refers to. Since we assume that pages are propagated to disk atomically, the REDO phase may rely upon information contained within a single page.

Once redo completes, we have applied some prefix of the run-time log. Therefore, we know that the page file is in a physically consistent state, although it contains portions of the results of uncommitted transactions. The final stage of recovery is the undo phase, which simply aborts all uncommitted transactions. Since the page file is physically consistent, the transactions may be aborted exactly as they would be during normal operation.

2.3.2 Physical, Logical and Phisiological Logging.

The above discussion avoided the use of some common terminology that should be presented here. *Physical logging* is the practice of logging physical (byte-level) updates and the physical (page number) addresses to which they are applied.

Physiological logging is what LLADD recommends for its redo records. The physical address (page number) is stored, but the byte offset and the actual difference are stored implicitly in the parameters of the redo or undo function. These pa-

[4] Although this discussion assumes that the entire log is present, the ARIES algorithm supports log truncation, which allows us to discard old portions of the log, bounding its size on disk.

rameters allow the function to update the page in a way that preserves application semantics. One common use for this is *slotted pages*, which use an on-page level of indirection to allow records to be re-arranged within the page; instead of using the page offset, redo operations use a logical offset to locate the data. This allows data within a single page to be re-arranged at runtime to produce contiguous regions of free space. LLADD generalizes this model; for example, the parameters passed to the function may utilize application specific properties in order to be significantly smaller than the physical change made to the page. [7]

Logical logging can only be used for undo entries in LLADD, and is identical to physiological logging, except that it stores a logical address (the key of a hash table, for instance) instead of a physical address. This allows the location of data in the page file to change, even if outstanding transactions may have to roll back changes made to that data. Clearly, for LLADD to be able to apply logical log entries, the page file must be physically consistent, ruling out use of logical logging for redo operations.

LLADD supports all three types of logging, and allows developers to register new operations, which is the key to its extensibility. After discussing LLADD's architecture, we will revisit this topic with a concrete example.

2.4 Concurrency and Aborted Transactions

Section 2.1 states that LLADD does not allow cascading aborts, implying that operation implementors must protect transactions from any structural changes made to data structures by uncommitted transactions, but LLADD does not provide any mechanisms designed for long-term locking. However, one of LLADD's goals is to make it easy to implement custom data structures for use within safe, multi-threaded transactions. Clearly, an additional mechanism is needed.

The solution is to allow portions of an operation to "commit" before the operation returns.[5] An operation's wrapper is just a normal function, and therefore may generate multiple log entries. First, it writes an undo-only entry to the log. This entry will cause the *logical* inverse of the current operation to be performed at recovery or abort, must be idempotent, and must fail gracefully if applied to a version of the database that does not contain the results of the current operation. Also, it must behave correctly even if an arbitrary number of intervening operations are performed on the data structure.

Next, the operation writes one or more redo-only log entries that may perform structural modifications to the data structure. These redo entries have the constraint that any prefix of them must leave the database in a consistent state, since only a prefix might execute before a crash. This is not as hard as it sounds, and in fact the B^{LINK} tree [11] is an example of a B-Tree implementation that behaves in this way, while the linear hash table implementation discussed in Section 4.1 is a scalable hash table that meets these constraints.

2.5 Summary

This section presented a relatively simple set of rules and patterns that a developer must follow in order to implement a durable, transactional and highly-concurrent data structure using LLADD:

- Pages should only be updated inside of a redo or undo function.
- An update to a page should update the LSN.
- If the data read by the wrapper function must match the state of the page that the redo function sees, then the wrapper should latch the relevant data.
- Redo operations should address pages by their physical offset, while Undo operations should use a more permanent address (such as index key) if the data may move between pages over time.
- An undo operation must correctly update a data structure if any prefix of its corresponding redo operations are applied to the structure, and if any number of intervening operations are applied to the structure.

Because undo and redo operations during normal operation and recovery are similar, most bugs will be found with conventional testing strategies. It is difficult to verify the final property, although a number of tools could be written to simulate various crash scenarios, and check the behavior of operations under these scenarios. Of course, such a tool could easily be applied to existing LLADD operations.

Note that the ARIES algorithm is extremely complex, and we have left out most of the details needed

[5] We considered the use of nested top actions, which LLADD could easily support. However, we currently use the slightly simpler (and lighter-weight) mechanism described here. If the need arises, we will add support for nested top actions.

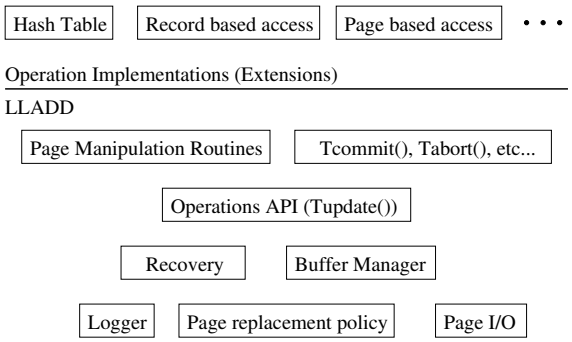

Figure 3: Simplified LLADD Architecture: The core of the library places as few restrictions on the application's data layout as possible. Custom "operations" implement the client's desired data layout. The separation of these two sets of modules makes it easy to improve and customize LLADD.

to understand how ARIES works, or to implement it correctly. Yet, we believe we have covered everything that a programmer needs to know in order to implement new data structures using the functionality that ARIES provides. This was possible due to the encapsulation of the ARIES algorithm inside of LLADD, which is the feature that most strongly differentiates LLADD from other, similar libraries. We hope that this will increase the availability of transactional data primitives to application developers.

3 LLADD Architecture

LLADD is a toolkit for building transaction managers. It provides user-defined redo and undo behavior, and has an extendible logging system with 19 types of log entries so far (not counting those internal to LLADD, such as "begin", "abort", and "clr"). Most of these extensions deal with data layout or modification, but some deal with other aspects of LLADD, such as extensions to recovery semantics (Section 4.2). LLADD comes with some default page layout schemes, but allows its users to redefine this layout as is appropriate. Currently LLADD imposes two requirements on page layouts. The first 32 bits must contain an LSN for recovery purposes, and the second 32 bits must contain the page type (since we allow multiple page formats).

Although it ships with basic operations that support variable-length records, hash tables and other common data types, our goal is to decouple all decisions regarding data format from the implementation of the logging and recovery systems. Therefore, the preceding section is essentially documentation for users of the library, while the purpose of the performance numbers in our evaluation section are not to validate our hash table, but to show that the underlying architecture is able to efficiently support interesting data structures.

Despite the complexity of the interactions among its modules, the basic ARIES algorithm itself is quite simple. Therefore, in order to keep LLADD simple, we started with a set of modules, and iteratively refined the boundaries among these modules. Figure 3 presents the resulting architecture. The core of the LLADD library is quite small at 2218 lines of code, 2155 lines of implementations of operations and other extensions, and 408 lines of installable header files.[6] The code has been documented extensively, and we hope that we have exposed most of the subtle interactions among internal modules in the online documentation.

As LLADD has evolved, many of its sub-systems have been incrementally improved, and we believe that the current set of modules is amenable to the addition of new functionality. For instance, the logging module interface encapsulates all of the details regarding its on disk format, which allows for some of the exotic logging and replication techniques mentioned above. Similarly, the interface encodes the dependencies between the logger and other subsystems.[7]

The buffer manager is another potential area for extension. Because the interface between the buffer manager and LLADD is simple, we would like to support transactional access to resources beyond simple page files. Some examples include transactional updates of multiple files on disk, transactional groups of program executions or network requests, or even leveraging some of the advances being made in the Linux and other modern OS kernels. For example, ReiserFS recently added support for atomic file-system operations. This could be used to provide variable-sized pages to LLADD. We revisit these ideas when we discuss existing systems such as CVS and IMAP, although they are applicable in many other circumstances.

From the testing point of view, the advantage of LLADD's division into subsystems with simple interfaces is obvious. We are able to use standard unit-testing techniques to test each of LLADD's subsystems independently, and have documented both external and internal interfaces, making it easy to add new tests and debug old ones. Furthermore,

[6] These counts were generated using David A. Wheeler's `SLOCCount`.

[7] For example, the buffer manager must ensure that the logger has forced the appropriate log entries to disk before writing a dirty page to disk. Otherwise, it would be impossible to undo the changes that had been made to the page.

by adding a "simulate crash" operation to a few of the key components, we can simulate application level crashes by clearing LLADD's internal state, reinitializing the library and verifying that recovery was successful. These tests currently cover approximately 90%[8] of the code. We have not yet developed a mechanism that models hardware failures, but plan to develop a test harness that verifies operation behavior in exceptional circumstances.

LLADD's performance requirements vary wildly depending on the workload with which it is presented. Its performance on a large number of small, sequential transactions will always be limited by the amount of time required to flush a page to disk. To some extent, compact logical and physiological log entries improve this situation. On the other hand, long running transactions only rarely force-write to disk and become CPU bound. Standard profiling techniques of the overall library's performance and micro-benchmarks of crucial modules handle such situations nicely.

Each module of LLADD is reentrant, and a C preprocessor directive allows the entire library to be instrumented in order to profile latching behavior, which aids in performance tuning and debugging. A thread that is not involved in an I/O request never needs to wait for a latch held by a thread that is waiting for I/O.[9]

There are a number of performance optimizations that are specific to multi-threaded operations that we do not perform. The most glaring omission is log bundling; if multiple transactions commit at once, LLADD must still force the log to disk once per transaction. This problem is not fundamental, but simply has not been addressed by current code base. Similarly, as page eviction requires a force-write if the full ARIES recovery algorithm is in use, we could implement a thread that asynchronously maintained a set of free buffer pages. We plan to implement such optimizations in the future.

4 Sample Operations

In order to validate LLADD's architecture, and to show that it simplifies the creation of efficient data structures, we have have implemented a number of simple extensions. In this section, we describe their design, and provide some concrete examples of our experiences extending LLADD. We would like to emphasize that this discussion reflects a "worst case" scenario; if LLADD extensions appropriate for an application already exist, the process detailed in this section is unnecessary. If an application does not require concurrent, multi-threaded applications, then physical logging can be used, allowing for the extremely simple implementation of new operations.

4.1 Linear Hash Table

Linear hash tables are hash tables that are able to extend their bucket list incrementally at runtime. They work as follows. Imagine that we want to double the size of a hash table of size 2^n, and that the hash table has been constructed with some hash function $h_n(x) = h(x) \, mod \, 2^n$. Choose $h_{n+1}(x) = h(x) \, mod \, 2^{n+1}$ as the hash function for the new table. Conceptually we are simply prepending a random bit to the old value of the hash function, so all lower order bits remain the same. At this point, we could simply block all concurrent access and iterate over the entire hash table, reinserting values according to the new hash function.

However, because of the way we chose $h_{n+1}(x)$, we know that the contents of each bucket, m, will be split between bucket m and bucket $m + 2^n$. Therefore, if we keep track of the last bucket that was split, we can split a few buckets at a time, resizing the hash table without introducing long pauses while we reorganize the hash table [12]. We can handle overflow using standard techniques; LLADD's linear hash table uses linked lists of overflow buckets.

The bucket list must be addressable as though it was an expandable array. We have implemented this functionality as a separate module reusable by applications, but will not discuss it here.

For the purposes of comparison, we provide two linear hash implementations. The first is straightforward, and is layered on top of LLADD's standard record setting operation, Tset(), and therefore performs physical undo. This implementation provided a stepping stone to the more sophisticated version which employs logical undo, and uses an identical on-disk layout. As we discussed earlier, logical undo provides more opportunities for concurrency, while decreasing the size of log entries. In fact, the physical-undo implementation of the linear hash table cannot support concurrent transactions, while threads utilizing the logical-undo implementation never hold locks on more than two buckets.[10]

[8] generated using "gcov", and "lcov,"

[9] Strictly speaking, this statement is only true for LLADD's core. However, there are variants of most popular data structures that allow us to preserve these invariants.

[10] However, only one thread may expand the hash-table at once. In order to amortize the overhead of initiating an expansion, and to allow concurrent insertions, the hash table is expanded in increments of a few thousand buckets.

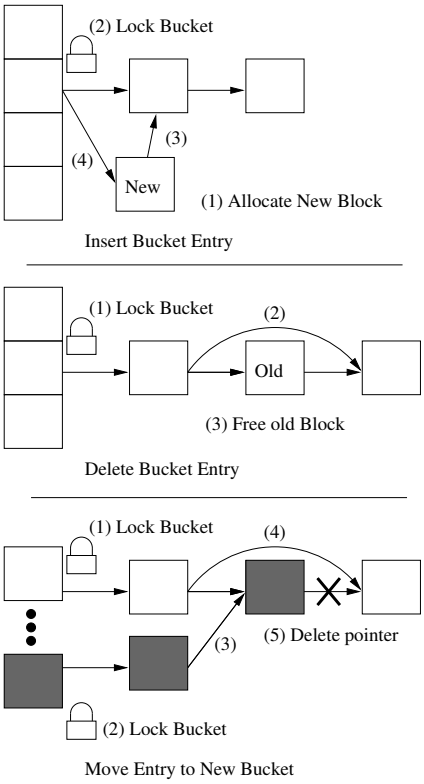

Figure 4: Linear Hash Table Bucket operations.

We see some performance improvement due to logical logging in Section 5.

From our point of view, the linked list management portion of the hash table algorithm is particularly interesting. It is straightforward in the physical case, but must be performed in a specific order in the logical case. See Figure 4 for a sequence of steps that safely implement the necessary linked list operations. Note that in the first two cases, the portion of the linked list that is visible from LLADD's point of view is always logically consistent. This is important for crash recovery; it is possible that LLADD will crash before the entire sequence of operations has been completed. The logging protocol guarantees that some prefix of the log will be available. Therefore, because the run-time version of the hash table is always consistent, we know that the version of the hash table produced by the REDO phase of recovery will also be consistent. Note that we have to worry about ordering because the buffer manager only provides atomic updates of single pages, but our linked list may span pages.

The third case, where buckets are split as the bucket list is expanded, is a bit more complicated. We must maintain consistency between two linked lists, and a page at the beginning of the hash table that contains the last bucket that we successfully split. Here, we use the undo entry to ensure proper crash recovery, not by undoing the split, but by actually redoing it; this is a perfectly valid "undo" strategy for some operations. Our bucket split algorithm is idempotent, so it may be applied an arbitrary number of times to a given bucket with no ill-effects. Also note that in this case there is not a good reason to undo a bucket split, so we can safely apply the split whether or not the current transaction commits.

First, we write an "undo" record that checks the hash table's meta-data and redoes the split if necessary (this record has no effect unless we crash during this bucket split). Second, we write (and execute) a series of redo-only records to the log. These encode the bucket split, and follow the linked list protocols listed above. Finally, we write a redo-only entry that updates the hash table's meta-data.[11]

We allow pointer aliasing at this step so that a given key can be present for a short period of time in both buckets. If we crash before the undo entry is written, no harm is done. If we crash after the entire update makes it to log, the redo stage will set the hash's meta-data appropriately, and the undo record becomes a no-op. If we crash in the middle of the bucket split, we know that the current transaction did not commit, and that recovery will execute the undo record. It will see that the bucket split is still pending and finish splitting the bucket. Therefore, the hash table is correctly restored.

Note that there is a point during the undo phase where the bucket is in an inconsistent physical state. Normally the redo phase brings the page file to a fully consistent physical state. We handle this by obtaining a lock on the bucket during normal operation. This blocks any attempt to write log entries that alter a bucket while it is being split. Therefore, the log cannot contain any entries that will accidentally attempt to access an inconsistent bucket.

Since the second implementation of the linear hash table uses logical undo, we are able to allow concurrent updates to different portions of the table. This is not true in the case of the implementation that uses pure physical logging, as physical undo cannot generally tolerate concurrent structural modifications to data structures.

[11] Had we been using nested top actions, we would not need the special undo entry, but we would need to store *physical* undo information for each of the modifications made to the bucket, since any subset of the pages may have been stolen.

4.2 Two-Phase Commit

The two-phase commit protocol is used in clustering applications where multiple, well maintained, well connected computers must agree upon a set of successful transactions. Some of the systems could crash, or the network could fail during operation, but we assume that such failures are temporary. Two-phase commit designates a single computer as the coordinator of a given transaction. This computer contacts the other systems participating in the transaction, and asks them to prepare to commit. If a subordinate system sees that an error has occurred, or the transaction should be aborted for some other reason, then it informs the coordinator. Otherwise, it enters the *prepared* state, and tells the coordinator that it is ready to commit. At some point in the future the coordinator will reply, telling the subordinate to commit or abort. From LLADD's point of view, the interesting portion of this algorithm is the *prepared* state, since it must be able to commit a prepared transaction if it crashes before the coordinator responds, but cannot commit before hearing the response, since it may be asked to abort the transaction.

Implementing the prepare state on top of the ARIES algorithm consists of writing a special log entry that informs the undo portion of the recovery phase that it should stop rolling back the current transaction and instead add it to the list of active transactions.[12] Due to LLADD's extendible logging system, and the simplicity of its recovery code, it took an afternoon for a programmer to become familiar with LLADD's architecture and add the prepare operation. This implementation of prepare allows LLADD to support applications that require two-phase commit. A preliminary implementation of a cluster hash table that employs two-phase commit is included in LLADD's CVS repository.

4.3 Other Applications

Previously, we mentioned a few systems that we think would benefit from LLADD. Here we sketch the process of implementing such applications.

LRVM implements a transactional version of malloc() [19]. It employs the operating system's virtual memory system to generate page faults if the application accesses a portion of memory that has not been swapped in. These page faults are intercepted and processed by a transactional storage layer which loads the corresponding pages from disk. A few simple functions such as abort() and commit() are provided to the application, and allow it to control the duration of its transactions. LLADD provides such a layer and the necessary calls, reducing the LRVM implementation to an implementation of the page fault handling code. The performance of the transactional storage system is crucial for this sort of application, and the variable length, keyed access, and higher levels of abstraction provided by existing libraries impose a severe performance penalty. LLADD could easily be extended so that it employs an appropriate on-disk structure that provides efficient, offset based access to aligned, fixed length blocks of data. A, LRVM requires a set_range() operation that efficiently updates a portion of a record, saving logging overhead. This feature could also be implemented as an operation, providing a simple, fast version of LRVM that would benefit from the infrastructure surrounding LLADD.

CVS provides version control over large sets of files. Multiple users may concurrently update the repository of files, and CVS attempts to merge conflicts and maintain the consistency of the file tree. By adding the ability to perform file system manipulations to LLADD, we could easily support applications with requirements similar to those of CVS. Furthermore, we could combine the file-system manipulation with record-oriented storage to store application-level logs, and other important metadata. This would allow a single mechanism to support applications such as CVS, simplifying fault tolerance, and improving the scalability of such applications.

IMAP is similar to CVS, but benefits further since it uses a simple, folder-based locking protocol, which would be extremely easy to implement using LLADD.

These last two examples highlight some of the potential advantages of extending LLADD to manipulate the file system, although it is possible that LLADD's page file would provide improved performance over the file system, at the expense of the transparency of file-system based storage mechanisms.

Another area of interest is in transactional serialization mechanisms for programming languages. Existing solutions are often complex, or are layered on top of a relational database or other system that uses a data format that is different than the representation the programming language uses. J2EE implementations and the wide variety of other persistence mechanisms available for Java provide a nice survey of the potential design choices and tradeoffs.

[12] Also, any locks that the transaction obtained should be restored, which is outside of the scope of LLADD, although a LLADD operation could easily implement this functionality on behalf of an external lock manager.

Since LLADD can easily be adapted to an application's desired data format, we believe that it is a good match for such persistence mechanisms.

5 Performance

We hope that the preceding sections have given the reader an idea of the usefulness and extensibility of the LLADD library. In this section we focus on performance evaluation.

In order to evaluate the physical and logical hash-table implementations, we first ran a test that inserts some tuples into the database. For this test, we chose fixed-length (key, value) pairs of integers. For simplicity, our hash-table implementations currently only support fixed-length keys and values, so this this test puts us at a significant advantage. It also provides an example of the type of workload that LLADD handles well; LLADD is designed to support application specific transactional data structures. For comparison, we also ran "Record Number" trials, named after the Berkeley DB access method. In this case, data is essentially stored in a large on-disk array. This test provides a measurement of the speed of the lowest level primitive supported by Berkeley DB, and the corresponding LLADD extension.

The times included in Figure 5 include page file and log creation, insertion of the tuples as a single transaction, and a clean program shutdown. We used the "transapp.cs" program from the Berkeley DB 4.2 tutorial to run the Berkeley DB tests, and hard-coded it to use integers instead of strings. We used the Berkeley DB "DB_HASH" index type for the hash-table implementation, and "DB_RECNO" in order to run the "Record Number" test.

Since LLADD addresses records as {Page, Slot, Size} triples, which is a lower level interface than Berkeley DB exports, we used the expandable array that supports the hash-table implementation to run the "LLADD Record Number" test.

One should not look at Figure 5, and conclude "LLADD is almost five times faster than Berkeley DB," since we chose a hash table implementation that is tuned for fixed-length data. Instead, the conclusions we draw from this test are that, first, LLADD's primitive operations are on par, performance wise, with Berkeley DB's, which we find very encouraging. Second, even a highly tuned implementation of a "simple" general-purpose data structure is not without overhead, and for applications where performance is important a special purpose structure may be appropriate.

The logical logging version of LLADD's hash-table outperformed the physical logging version for two reasons. First, since it writes fewer undo records, it generates a smaller log file. Second, in order to emphasize the performance benefits of our extension mechanism, we use lower level primitives for the logical logging version. The logical logging version implements locking at the bucket level, so many mutexes that are acquired by LLADD's default mechanisms are redundant. The physical logging version of the hash-table serves as a rough proxy for an implementation on top of a non-extendible system. Therefore, it uses LLADD's default mechanisms, which include the redundant acquisition of locks.

As a final note on our first performance graph, we would like to address the fact that LLADD's hash-table curve is non-linear. LLADD currently uses a fixed-size in-memory hash-table implementation in many areas, and it is possible that we exceeded the fixed-size of this hash-table on the larger test sets. Also, LLADD's buffer manager is currently fixed size. Regardless of the cause of this non-linearity, we do not believe that it is fundamental to our design.

The multi-threaded test run in the first figure shows that the library is capable of handling a large number of threads. The performance degradation associated with running 200 concurrent threads was negligible. Figure 6 expands upon this point by plotting the time taken for various numbers of threads to perform a total of 500,000 read operations. The performance of LLADD in this figure is essentially flat, showing only a negligible slowdown up to 250 threads. (Our test system prevented us from spawning more than 250 simultaneous threads, and we suspect that LLADD would easily scale to more than 250 threads. This test was performed on a uniprocessor machine, so we did not expect to see a significant speedup when we moved from a single thread to multiple threads.

Unfortunately, when ran this test on a multiprocessor machine, we saw a degradation in performance instead of the expected speed up. The problem seems to be the additional overhead incurred by multi-threaded applications running on SMP machines under Linux 2.6, as the single thread test spent a small amount of time in the Linux kernel, while even the two-thread version of the test spent a significant time in kernel code. We suspect that the large number of briefly-held latches that LLADD acquires caused this problem. We plan to investigate this problem further, adopting LLADD to a more advanced threading package with user-level latches [3], or providing an "SMP Mode" compile-time option that decreases the number of latches that LLADD

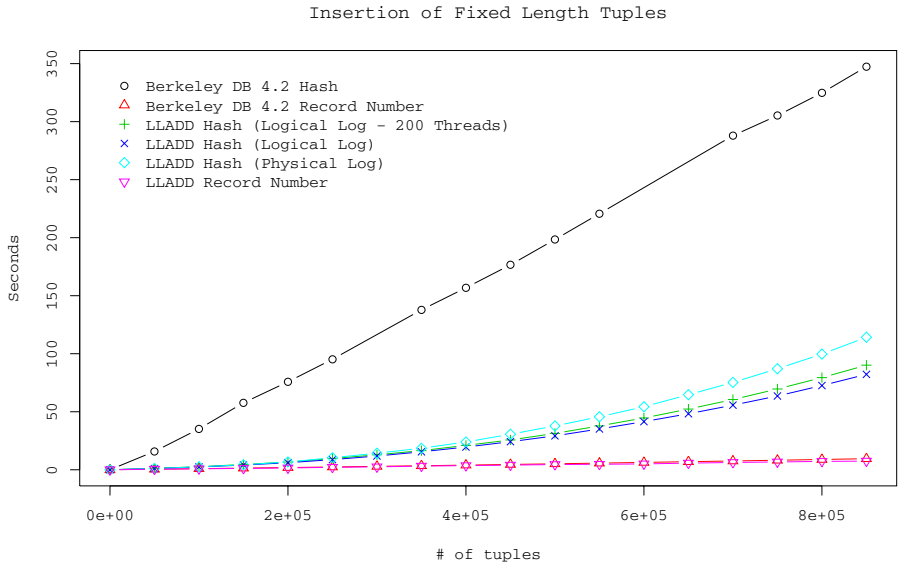

Figure 5: The final data points for LLADD's and Berkeley DB's record number based storage are 7.4 and 9.5 seconds, respectively. LLADD's hash table is significantly faster than Berkeley DB in this test, but provides less functionality than the Berkeley DB hash. Finally, the logical logging version of LLADD's hash table is faster than the physical version, and handles the multi-threaded test well. The threaded test spawned 200 threads and split its workload into 200 separate transactions.

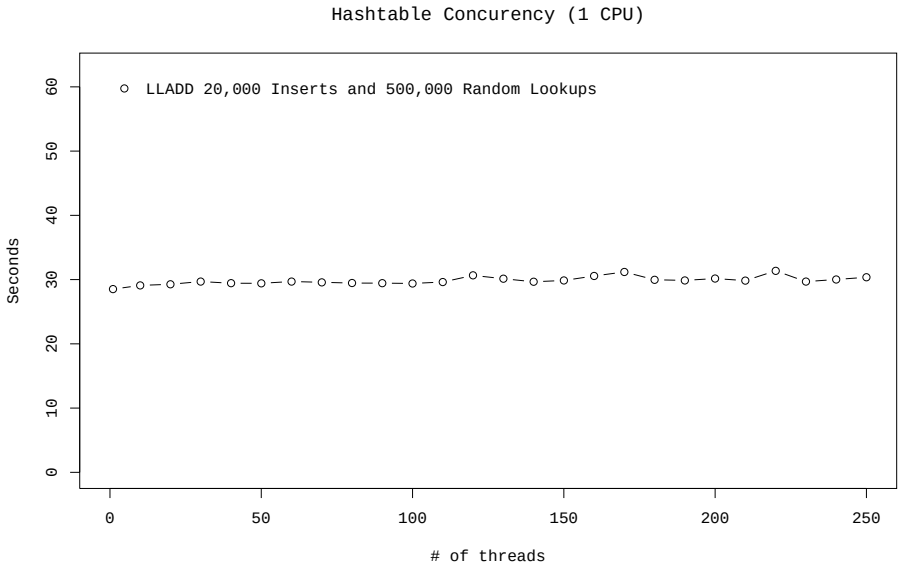

Figure 6: The time required to perform a fixed amount of processing, split across various numbers of threads. This test was run against the highly concurrent Logical Logging version of the linear hash table. No significant performance degradation was seen within the range measured. The inserts were done in serial, and the lookups were performed in parallel.

acquires at the expense of opportunities for concurrency.

6 Future Work

LLADD is an extendible implementation of the ARIES algorithm. This allows application developers to incorporate transactional recovery into a wide range of systems. We have a few ideas along these lines, and also have some ideas for extensions to LLADD itself.

LLADD currently relies upon its buffer manager for page-oriented storage. Although we did not have space to discuss it in this paper, we have a blob implementation that stores large data outside of the page file. This concept could be extended to arbitrary primitives, such as transactional updates to file system directory trees, or integration of networking or other operations directly into LLADD transactions. Doing this would allow LLADD to act as a sort of "glue code" among various systems, ensuring data integrity and adding database-style functionality, such as continuous backup, to systems that currently do not provide such mechanisms. We believe that there is quite a bit of room for the development of new software systems in the space between the high-level but sometimes inappropriate interfaces exported by existing transactional storage systems, and the unsafe, low-level primitives provided by current file systems.

Currently, although we have implemented a two-phase commit algorithm, LLADD really is not very network aware. If we provided a clean abstraction that allowed LLADD extensions and operations to cross network boundaries, then we could provide a wider range of network consistency algorithms and cleanly support the implementation of operations that perform well in networked and in local environments.

Although LLADD is re-entrant, its latching mechanisms only provide physical consistency. Traditionally lock managers, which provide higher levels of consistency, have been tightly coupled with transactional page implementations. Generally, the semantics of undo and redo operations provided by the transactional page layer and its associated data structures determine the level of concurrency that is possible. Since prior systems provide a monolithic set of primitives to their users, these systems typically had complex interactions among the lock manager, on-disk formats and the transactional page layer. Due to the clean interfaces that LLADD provides between on-disk formats and its transactional page layer, and because of its extensible log entries, the implementation of general purpose, modular lock managers on top of LLADD seems to be straightforward. We plan to investigate this in the future, as it would provide significant opportunities for code reuse, and for the implementation of extremely flexible transactional systems.

As a final note, we believe that a large fraction of the "application specific" extensions made to LLADD will be reusable in their own right. Over time, we hope to provide a set of specialized, but reusable components that will be able to support an unprecedented range of applications. We have focused upon LLADD's extensibility in this paper, but we also intend for the library to be useful to relatively casual users that are simply interested in obtaining a transactional data structure that is appropriate to their application.

7 Conclusion

We have outlined the design and implementation of a library for the development of transactional storage systems. By decoupling the on-disk format from the transactional storage system, we provide applications with customizable, high-performance, transactional storage. By summarizing and documenting the interactions between these customizations and the storage system, we make it easy to implement such customizations.

Current applications generally must choose between high-level, general-purpose libraries that impose severe performance penalties and ad-hoc "from scratch" atomicity and durability mechanisms. We aim to bridge this gap, allowing applications to make use of high-level, efficient and special-purpose transactional storage. Today, many applications have to choose between efficiency, reliable storage and ease of development. As a result, applications that handle important or complex information are often difficult to develop and maintain, or fail to meet their users requirements.

Because of the stable interface between operation implementations and the underlying implementation of the ARIES algorithm, we allow operation implementations and the implementation of the library to evolve independently, allowing applications to make use of advanced replication and storage techniques as the circumstances in which they are deployed changes and as LLADD evolves

By releasing LLADD to the community, we hope that we will be able to provide a toolkit that aids in the development of real-world applications, and is flexible enough for use as a research platform.

8 Acknowledgements

We would like to thank Jason Bayer, Jim Blomo and Jimmy Kittiyachavalit for their implementation work and contributions to earlier versions of LLADD. Joe Hellerstein and Mike Franklin provided us with invaluable advice. Rob von Behren provided us with some last minute assistance during the benchmarking process.

9 Availability

LLADD is free software, available at:

`http://www.sourceforge.net/projects/lladd`

References

[1] Agrawal, et al. *Concurrency Control Performance Modeling: Alternatives and Implications*. TODS 12(4): (1987) 609-654

[2] Berkeley DB, `http://www.sleepycat.com/`

[3] R. von Behren, J Condit, F. Zhou, G. Necula, and E. Brewer. *Capriccio: Scalable Threads for Internet Services* SOSP 19 (2003).

[4] E. F. Codd, *A Relational Model of Data for Large Shared Data Banks.* CACM 13(6) p. 377-387 (1970)

[5] Envangelos P. Markatos. *On Caching Search Engine Results*. Institute of Computer Science, Foundation for Research & Technology - Hellas (FORTH) Technical Report 241 (1999)

[6] David K. Gifford, P. Jouvelot, Mark A. Sheldon, and Jr. James W. O'Toole. *Semantic file systems*. Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles, (1991) p. 16-25.

[7] Gray, J. and Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann (1993) San Mateo, CA

[8] Jim Gray, Raymond A. Lorie, and Gianfranco R. Putzulo. *Granularity of locks and degrees of consistency in a shared database*. In 1st International Conference on VLDB, pages 428–431, September 1975. Reprinted in Readings in Database Systems, 3rd edition.

[9] Haerder & Reuter *"Principles of Transaction-Oriented Database Recovery."* Computing Surveys 15(4) p 287-317 (1983)

[10] Lamb, et al., *The ObjectStore System.* CACM 34(10) (1991) p. 50-63

[11] Lehman & Yao, *Efficient Locking for Concurrent Operations in B-trees.* TODS 6(4) (1981) p. 650-670

[12] Litwin, W., *Linear Hashing: A New Tool for File and Table Addressing.* Proc. 6th VLDB, Montreal, Canada, (Oct. 1980) p. 212-223

[13] Mohan, et al., *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging.* TODS 17(1) (1992) p. 94-162

[14] Mohan, Lindsay & Obermarck, *Transaction Management in the R* Distributed Database Management System* TODS 11(4) (1986) p. 378-396

[15] Mohan, Levine. *ARIES/IM: an efficient and high concurrency index management method using write-ahead logging* International Converence on Management of Data, SIGMOD (1992) p. 371-380

[16] *MySQL*, `http://www.mysql.com/`

[17] Reiser, Hans T. *ReiserFS 4* `http://www.namesys.com/` (2004)

[18] M. Seltzer, M. Olsen. *LIBTP: Portable, Modular Transactions for UNIX*. Proceedings of the 1992 Winter Usenix (1992)

[19] Satyanarayanan, M., Mashburn, H. H., Kumar, P., Steere, D. C., AND Kistler, J. J. *Lightweight Recoverable Virtual Memory.* ACM Transactions on Computer Systems 12, 1 (Februrary 1994) p. 33-57. Corrigendum: May 1994, Vol. 12, No. 2, pp. 165-172.

[20] Stonebraker. *Inclusion of New Types in Relational Data Base* ICDE (1986) p. 262-269