Design and Evaluation of a Benchmark for Main Memory Transaction Processing Systems

by

Elizabeth G. Reid

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2009

Author .... [illegible]
Department of Electrical Engineering and Computer Science
May 18, 2009

Certified by .... [illegible]
Samuel Madden
Associate Professor
Thesis Supervisor

Accepted by .... [illegible]
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Design and Evaluation of a Benchmark for Main Memory Transaction Processing Systems

by

Elizabeth G. Reid

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2009, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

We designed a diverse collection of benchmarks for Main Memory Database Systems (MMDBs) to validate and compare entries in a programming contest. Each entrant to the contest programmed an indexing system optimized for multicore multithread execution. The contest framework provided an API for the contestants, and benchmarked their submissions.

This thesis describes the test goals, the API, and the test environment. It documents the website used by the contestants, describes the general nature of the tests run on each submission, and summarizes the results for each submission that was able to complete the tests.

Thesis Supervisor: Samuel Madden
Title: Associate Professor

Acknowledgments

I would like to thank Sam Madden for his support, assistance and patience throughout this thesis. Not only did he help with issues that arose during my work, his help in moderating the contest was invaluable.

I would also like to thank Mike Stonebraker for his vision and feedback on the structure and content of the contest, Margo Seltzer for her patience while helping me debug my Berkeley DB implementation, my cousin Garrett Reid for loaning me the use of one of his computers for testing, and for being a guinea pig for many of my scripts and setups, and my father Brian Reid for helping me unravel many of the bugs that arose over the last eight months.

And finally, thank you to my family for helping edit my thesis, reading it when I no longer had the patience to go through it again myself, and my friends for keeping me entertained when I probably should have been working.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this document I discuss the motivation, framework and preliminary results of the First Annual SIGMOD (Special Interest Group on Management of Data) Programming Contest, which will culminate at the SIGMOD conference in June 2009 in Providence, Rhode Island. The competition is to create an indexing system for main memory data optimized for a multi-core machine running many threads over multiple indices. The contest was designed, implemented and run by myself, Samuel Madden and Mike Stonebraker beginning in October 2008. This thesis was written in May 2009 before the final stages of the competition were completed and the winner was determined.

1.1 Motivations For Main Memory Database Systems

Main memory database systems (MMDBs) store their data in main physical memory in order to provide much faster access speeds than traditional disk-resident database systems (DRDBs) can provide. Reading data from disk is often the slowest piece of computer systems, especially data-intensive applications like databases. By removing disks from a system's design, the efficiency and throughput of the system can be drastically improved.

Due to the historical costs of main memory, traditional DRDBs have been designed to work using a combination of disk-resident data and main memory caching for a subset of the data in an attempt to minimize the delays caused by waiting for data to be read from the disk. As technology has improved and chip density increased, main memory has become large enough to hold entire databases completely, opening the door for MMDBs to emerge as a different type of database system [4].

While DRDBs can be run entirely in main memory and be fully functional, their storage structures were specifically optimized to use the disk to store data. In order to take full advantage of the features added by living primarily in main memory, new systems must be designed. Not only is the access time significantly shorter for main memory systems, but they need not be block-oriented because there is no fixed-cost per access and sequential access cost is insignificant. Almost every aspect of a database system can be affected by these differences. This motivates research into the best design for a MMDB to maximize performance.

1.2 Contest Overview

The Call for Entries for the First Annual SIGMOD Programming Contest was posted to the conference's website in October 2008. It specified that student teams from degree granting institutions were welcome to participate for a cash prize of $5,000 by creating an indexing system for main memory data.

We released the official API, along with an example implementation and some basic unit tests, in mid-December and tweaked them slightly over the following months to fix bugs and clarify points of confusion for participants. We provided an initial version of the basic benchmark test to be used to compare the different implementations at the beginning of February 2009, although it went through more significant transformations over the following months than the rest of the released code. We also gave participants a basic harness that allowed them to run the benchmark on their own machines.

At the end of February, we launched a website that allowed participants to see the results when their implementations are run on the official testing machine. The site also contains a leaderboard to show the fastest results on a few basic benchmark tests to give participants an idea of the speeds being reached by other teams.

The contest closed at 11:59 pm EST on March 31, 2009, at which point participants must have turned in a working binary and all of their source code to be tested and inspected in order to determine the finalists. We ran a few basic benchmarks over all of the submissions, and the eight fastest were pulled out for closer inspection and further testing. From those eight, we selected five finalists to come to the SIGMOD conference in June to explain their design and for a final round of testing before a winner is chosen.

1.2.1 Transactions

The primary unit of work in a database is a transaction. A single transaction is made up of one or more smaller tasks, each reading or writing information to one or more indices in a database. These tasks occur in a specified order within the transaction, and once they have all completed, the transaction is committed and the changes it produced are made permanent. Alternately, if the client wishes to undo any of the smaller tasks performed in a transaction, all of the tasks performed in the transaction must also be undone. This property is called 'atomicity' and is sometimes referred to as 'all-or-nothing', as either all of the work in a transaction is done, or none of it is done.

The atomic property of a transaction is important so that any changes made to a database will not leave it in an inconsistent state. A database may have some internal logic dictating relationships between different pieces of data within it. Thus, if one piece of data is added, changed or removed, the associated data must be adjusted as well. By guaranteeing that transactions are atomic, clients can make all of the necessary updates within a single transaction and know that consistency will be maintained.

Multiple transactions may be running simultaneously on a database. However, each must behave as if it is the only transaction currently given access to the data it accesses. If one transaction reads a piece of information from an index and then a second transaction changes that value before the first transaction completes, the behavior of one or both of these transactions may be significantly affected. This 'isolation' property often requires some sort of locking mechanism to give the implementation some concurrency control, although how the locks behave depends on the implementation. The most straightforward implementation would be to remove the parallelization between any of the threads to run one transaction at a time, but this would deliver very poor performance.

The final property of transactions that database systems must guarantee is 'durability'. Once a transaction has completed, all of the changes made during that transaction must exist in the database even if the system crashes. However, for the purposes of this contest, crash-recoverability was not required of the implementations. Previous research [5, 8] has shown that, while crash-recoverability is important, there are different tools that can be used to support it given different hardware performance requirements (duplication, logging, etc.). By removing any form of long-term durability code from these implementations, the code remains a useful tool for all of these different systems, rather than those using whatever durability scheme we might have required.

1.3 Previous Main Memory Data Structures

As discussed briefly in Section 1.1, the optimal layout of data that lives in main memory should be different from the layout of data that lives on disk. The structure of this data can have a significant effect on the performance of a system that interacts with it, and as such a number of data structures have been proposed over the years to solve the problem.

B-trees were invented and optimized for systems that read and write large chunks of data to disk [1], which is especially common in databases and file systems. They are organized in block-sized chunks and occasionally reorganize themselves to support searches, inserts and deletions in O(log n) amortized time.

T-trees were proposed in 1986 as an indexing structure that would work well for main memory database systems [6]. A T-tree is a binary tree with many elements in a node which gives it the good update and storage characteristics of a B-tree. However, later research [7] found that T-trees give the impression of being cache conscious by utilizing data in each node multiple times when searching a tree, but that the cache-locality of the data was not properly utilized. When they were first proposed, the gap between processor and main memory speeds was not very large and so the source of the speed-up was not properly understood. Thus, a T-tree's fundamental behavior is similar to a binary tree's until the bottom of the tree has been reached, at which time a more cache-conscious behavior is exhibited. For trees large enough to require many layers of nodes before the leaves are reached, that depth can significantly impede the performance of a T-tree.

The Cache Sensitive Search Tree (CSS-Tree) was proposed to surpass the performance of both B-trees and T-trees by paying closer attention to reference locality and cache behavior [7]. CSS-trees store a directory structure on top of a sorted array that represents a search tree. The nodes in the array are designed to match in size the cache-line size of the machine on which it is running.

Recent research has been done [2] into Cache-Oblivious String B-trees (COSB-trees), which are designed to improve the performance of B-trees on disk. Kuszmaul emphet al. propose that B-trees perform suboptimally when keys are long or variable length, when keys are compressed, for range queries and with respect to memory effects such as disk prefetching. They present the COSB-tree as a data structure that is more efficient when searching, inserting and deleting from a tree, performing range queries and using all levels of a memory hierarchy to make good use of disk locality, and also maintains an index whose size is proportional to the front-compressed size of the dictionary. While this data structure is not specifically designed for use in main memory, some of the optimizations it emphasizes, such as cache obliviousness, could be applicable to main-memory-specific data structures as well.

While the creators of CSS-trees did some analysis comparing the performance of different tree structures in main memory, there has not been a comparison done by an independent third party. By allowing participants to use any data structures they choose to create their database implementation, the relative merits of each system should be made more clear.

1.4 Contributions

The contributions provided by this thesis lie mainly in the infrastructure and design of the contest, as well as the analysis of the final submissions.

1.4.1 Benchmark Design

We designed the benchmarks for this contest to fairly measure the performance of the different implementations without introducing excessive overhead. The distribution and type of tests to be run over the indices was specified in the contest description. However, the design of the mechanism to generate data for the indices, as well as how to measure the performance of the implementations, went through several iterations before being finalized.

Performance Measurements

Initially, the proposed benchmark required each transaction to take only a small amount of time. The benchmark of an implementation was supposed to count each time a transaction took longer than that limit, with the final score being proportional to the number of 'violations'. However, this could lead to undesirable behavior if participants optimized their implementation too closely to this benchmark.

If a particular transaction takes significantly longer than the time limit, the penalty would be the same as finishing only milliseconds afterward. Thus, once the threshold has been crossed an implementation would be able to use the unpunished time to do any internal reorganization or cleanup that had not been properly completed during earlier transactions in an attempt to keep within the transactional boundaries.

The solution to this problem was to abandon the threshold entirely, and instead fall back on using a global timer that records how long the entire benchmark takes to complete. The amortized time for each transaction is thus the most important behavior in each implementation, rather than the individual transaction times.

Data Generation

We originally generated the data used to populate and provide queries for testing the indices using a random number generator (RNG) while the benchmark was running. This provided an even distribution of data and reproducible tests by seeding the RNG and carefully controlling how different threads accessed it.

However, it was eventually determined that the method used to produce a new number by the RNG introduced too much overhead relative to the time each implementation took to run a single transaction. This distorts performance measurements by making it difficult to distinguish overhead costs from the time required for the implementation to perform its required tasks.

In order to avoid this potential loophole, and to remove the random data generation from the performance time entirely, we pre-generated all of the random data used to test the indices at the beginning of the benchmark and stored it in large arrays in main memory. This method has the added benefit of restricting the number of random keys and payloads used during testing, thus increasing the potential for collisions. This forces implementations to handle them more often, and thus make sure that they are handled efficiently.

The exact behavior of the benchmark cannot be predicted before it begins, so we had to generate the worst-case amount of data required for the different tests every time they were run. This significantly increased the amount of space required to hold all of the generated data, which is stored in memory that would otherwise be available to the participants to use for their implementation's internal storage structures. However, we believe the benefit of a very small overhead for generating data during the test itself compared to the detriment of the wasted space to be preferable.

1.4.2 Testing Framework

We ran a multitude of tests over many different implementations a large number of times. In order to make this easier, we built a testing framework that organized the system at multiple levels.

We designed benchmarks themselves so that, given the same seed, the same tests are run in the same order with the same data, independent of any other factors. This allowed for side-by-side comparisons of different implementations that were given the same workload. It also allows a problematic test to be repeated for debugging purposes.

In order to automate the tests so that they could be run in bulk sequentially, we wrote two Python scripts, one to handle the organization of the tests being run, and one to run a specific test in isolation.

The outermost script queried the database of tasks to find any pending tasks that participants had issued, such as running the unit tests or one of the benchmark tests over their implementation. For each test, the script used the information provided to calculate parameters that are passed into the benchmark test. Parameters include: how many times to run the test, which test should be run, how many inserts are done per thread, how many tests are run per thread, how many threads are used in the test, and how many indices are created in the test. The script inserts this information into the database, and then starts the inner script. While the inner script is running, the outer script keeps track of how long it has been running, and if it exceeds a predetermined time threshold, it kills the test and moves onto the next one. If the inner script finishes testing within the time limit, the outer script parses the results of the test and enters them into the database. It also copies the output files from the test into a location that can be accessed by the participant via a web interface, so that the debug output can be analyzed.

The inner script provides isolated security for the test being run by building a jail directory into which all of the relevant test files are copied (further described in Section 3.1). It then runs the relevant benchmark test multiple times, compounding the results from each of them. Once the benchmarks have completed, the inner script runs various tests over the binary, such as the Phantom Test, and then returns the results to the outer script.

This modular framework can be easily adapted to run a variety of tests over a collection of binaries. This was done for the final testing of the submissions, thus exhibiting its usefulness.

1.4.3 Analysis of Submissions

The third significant contribution of this project is the analysis of the submissions to the contest. After winnowing down the submissions with an initial round of testing, we more thoroughly tested and hand inspected the code of the remaining submissions in order to determine their methodology and to identify some common aspects to their designs that are effective on the supplied benchmarks. The most important design decisions for any implementation were isolated and some of the more effective choices were identified.

In Chapter 2 I discuss the development of the API and what test and benchmarks we used to compare the submissions to the contest. In Chapter 3 I describe the website provided for the participants to test their implementations. In Chapter 4 I discuss the general design decisions each team made during their design. In Chapter 5 I discuss the results of the tests on the implementations and analyze the effectiveness of some of the design decisions and draw conclusions about the effectiveness of our benchmarks.

Chapter 2

API, Testing and Benchmarks

We first devised a standard application programming interface (API) for a database application to allow standardized testing and comparison of the submissions to the contest. The API is written in C but was also designed and tested to support C++ implementations. We also developed unit tests and a benchmark script to verify that the participants' code supported all behavior defined in the API and to determine which implementations ran the fastest under different loads.

2.1 API Overview

The API defines the function calls that the submissions must implement in order to be a valid solution and provides a small number of standardized data structures and constants. As a database application, the API supports a few basic features: the creation of indices in which data is stored in an organized manner, the ability to insert and remove data from an index, and the ability to retrieve data from an index or scan through the data in an index. Data is organized in each index in pairs of associated `Keys` and `payloads`, where the `Keys` have an absolute order relative to each other, but each `Key` can have multiple `payloads` associated with it.

```
ErrCode create(KeyType type, char *name);
ErrCode openIndex(const char *name, IdxState **idxState);
ErrCode closeIndex(IdxState *idxState);
ErrCode beginTransaction(TxnState **txn);
ErrCode abortTransaction(TxnState *txn);
ErrCode commitTransaction(TxnState *txn);
ErrCode get(IdxState *idxState, TxnState *txn, Record *record);
ErrCode getNext(IdxState *idxState, TxnState *txn, Record *record);
ErrCode insertRecord(IdxState *idxState, TxnState *txn, Key *k, const char* payload);
ErrCode deleteRecord(IdxState *idxState, TxnState *txn, Record *record);
```

Figure 2-1: Function calls defined in API.

2.1.1 Function Calls

The API contains ten function calls that a valid submission must implement. Those functions are used in the unit tests and benchmarks to verify and compare the different implementations. To lessen the performance impact of the testing infrastructure, the parameters passed to the functions are pointers to variables rather than the variables themselves. This minimizes the amount of data written to the stack for each call so that it takes less time to make the call.

Index Management

There are three functions that deal specifically with the creation and access to indices. The `create` function makes a new index with a `KeyType` (see Section 2.1.2) and `name` specified. It is called only once per index after which the index is available to any thread in the process. The `name` must be a unique string of characters used to identify the index when opening it via the `openIndex` function. If `create` is called with a `name` that was already used to make an index then the function returns an `ErrCode` (see Section 2.1.2) indicating the issue.

The `openIndex` function provides access to an index. The arguments specify the index by its unique `name` and include a handle to an `IdxState` pointer (see Section 2.1.2) pointing to whatever information the implementation will need to uniquely identify and support the thread's access to the specified index. If the `name` specified has not been used in a successful call to `openIndex` then the function returns an `ErrCode` indicating that the database does not exist.

Once a thread has finished using an index it can close its connection to it using the `closeIndex` function. The only parameter passed in is the `IdxState` pointer that was passed by the implementation in the `openIndex` call. Once the `closeIndex` call has completed this pointer is no longer a valid index identifier for the thread and its access to the index is closed until it calls `openIndex` again.

Transaction Management

There are three functions that manage transactions. Each thread can have only one outstanding transaction running at a time, but a single transaction can span multiple indices with a single thread. Other transactions should not see a transaction's updates until it has committed, but later operations within the a transaction must be able to see the previous effects of the transaction. Also, the transactions must be serialized so that the changes made to a database are equivalent to running the transactions in some serial order, one after another. The implementation may serialize concurrent transactions in any order.

The `beginTransaction` function signals the beginning of a transaction. Its only parameter is a pointer to a `TxnState` pointer (see Section 2.1.2) that the implementation uses to uniquely identify this transaction in the future. If the thread calling `beginTransaction` has a transaction that is currently open then the function returns an `ErrCode` indicating that a new transaction cannot be opened.

Once a thread has finished all of the operations involved in its transaction any changes it has made are committed using the `commitTransaction` function. The only parameter passed to the call is the `TxnState` variable that was used during `beginTransaction`, and after `commitTransaction` completes the variable is no longer valid. However, if the `TxnState` variable is not a valid identifier then the function returns an `ErrCode` indicating the problem. Alternatively, if the transaction was valid but could not commit due to deadlock with one or more other transactions then the function returns an `ErrCode` telling the client that the changes could not be committed at that time.

If a client wants to discard the changes made in a transaction, rather than calling `commitTransaction` to make the changes permanent, a transaction can be discarded using a `abortTransaction` call. Any changes made to the database over the course of the transaction will be rolled back as though the transaction had never happened. However, if the `TxnState` variable is not a valid transaction identifier or if rolling back the changes in the transaction caused a deadlock, then the function returns an `ErrCode` indicating the specific problem.

Database Calls

There are four functions used to access and manipulate the data stored in an index. Each function can only act upon one index at a time, even if multiple indices are open for a given thread. Each function call can be part of a larger, complex transaction by handing in a `TxnState` variable created during a `beginTransaction` call. If, instead `NULL` is passed in as the `TxnState` value then any changes made in the function call are immediately committed.

Information is inserted into an index using the `insertRecord` function. Beyond specifying the index and transaction, the other parameters for the function are the `Key` and `payload` for the entry. If that particular `Key/payload` pair already exists in that index only a single copy of the entry is kept and the function returns an `ErrCode` indicating what occurred. Alternatively, if inserting the record into the index causes a deadlock, the function returns an `ErrCode` indicating this.

Information is deleted from an index using the `deleteRecord` function. A `Record` variable is handed in as a parameter that specifies what information should be deleted from the index in question. The `Record` can either specify a specific `Key/payload` pair to be removed, or it can set the `payload` to `NULL`. If the latter is done then all `Records` in the index with the specified `Key` are deleted. If, in either case, the specified `Key` is not found in the index then the function returns an `ErrCode` indicating the problem. Alternately, if deleting the data from the index would cause a deadlock, the appropriate `ErrCode` is returned.

There are two functions that are used to retrieve a `Record` from an index. To retrieve a `Record` given a specific `Key` a client may use the `get` function. A `Record` is passed in as a parameter in which the `Key` is specified by the client and the `payload` is left blank. The implementation copies in the `payload` information associated with that `Key` into the available space. If there is more than one `payload` associated with that `Key` then the implementation can choose which to return. If the specified `Key` does not have any associated `payloads` in the index then the function returns an `ErrCode` indicating the issue.

The other `payloads` associated with a `Key` can be accessed using the `getNext` function. It is passed an empty `Record` object into which it copies both the `Key` and `payload` to be returned. If there are n `payloads` associated with a `Key` then a single call to `get` followed by *n-1* calls to `getNext` should return all n `Records` with the specified `Key`. However, `getNext` can scan an entire index of information, not just the values associated with a single `Key`. It will always return the `Record` following the previous `Record` retrieved by a call to either `get` or `getNext` in the same transaction. Also, if a call to `get` was made specifying a `Key` that did not exist in the index, `getNext` should return the first `Record` in the index that does exist after that `Key`. If there have not yet been any `get` or `getNext` calls then `getNext` returns the first `Record` in the index. When `getNext` is called after the last `Record` in an index has already been accessed then an `ErrCode` is returned indicating that the end of the index has been reached.

Unspecified Functions

The functions defined in the API were designed to allow the various implementations to be tested and compared using standardized tests. As such, some functions were not included that would be needed for a launch-worthy API. Most noteworthy of the missing functionality is the lack of a `deleteIndex` function which would completely delete an entire index from the database.

Memory Management

While the parameters handed into the function calls described above have space allocated for them in memory at the time of the function call, many of them are not

```
typedef enum KeyType {
        SHORT,
        INT,
        VARCHAR
} KeyType;

typedef struct {
        KeyType type;
        union {
            int32_t shortkey;
            int64_t intkey;
            char charkey[MAX_VARCHAR_LEN + 1];
        } keyval;
} Key;

typedef struct {
        Key key;
        char payload[MAX_PAYLOAD_LEN + 1];
} Record;
```

Figure 2-2: Data structures provided by API.

guaranteed to remain at that location after the call has completed. The `Records`, `Keys` and `payloads` are allocated by the client and the client is in charge of freeing that memory, which can be done any time after the call has completed. If the implementation needs to access the information stored in these parameters once the call has completed, it must copy it into some other location in memory that it has allocated. However, the implementation-specific data types (see Section 2.1.2) are allocated by the database implementations, which are also in charge of freeing them once they are no longer in use.

2.1.2 Data Structures

Record Information

The database records being passed in and out of the database are defined by a collection of C structs in the API. A `Record` is made up of a `Key` and a `payload`, where the `Key` can have three possible `KeyTypes`: 32-bit int, 64-bit long or a variable-length string of up to 128 bytes. These values are all stored in structs which have the smallest

```
typedef enum ErrCode {
        SUCCESS,
        DB_DNE,
        DB_EXISTS,
        DB_END,
        KEY_NOTFOUND,
        TXN_EXISTS,
        TXN_DNE,
        ENTRY_EXISTS,
        ENTRY_DNE,
        DEADLOCK,
        FAILURE
} ErrCode;
```

Figure 2-3: Error codes provided by API.

possible size while still allowing for all possible `Key` types and holding both a `payload` string and a `Key` string.

The structs passed into the database are kept as small as possible so that the amount of data passed in a function call can be minimized. In C, structs are of a constant size and of union type so that they have a known layout, but this means that if a `Record` contains a 32-bit `Key`, the `Record` will still be allocated large enough to hold a string-valued `Key`.

It would have required significantly less space to have the `payload` and varchar `Key` values in the struct be pointers to a string on the heap, cutting down the size of the `Key` struct by 120 bytes and the `payload` struct by 92 bytes. However, passing a pointer to a string on the heap leaves it ambiguous who is in charge of freeing the memory, as well as when it will occur. By including the entire string in the struct this ambiguity is removed.

Error Codes

The API also defines a set of error codes that are used to allow a database implementation to indicate the success or reason for failure of a function call. Not all of the error codes are applicable for every function call, so the specifications for each function define the circumstances under which a given error code is returned. Every function can return SUCCESS when the function completed successfully and FAILURE if some unknown error occurred.

Implementation-Specific Data Types

The API provides two unimplemented data types, IdxState and TxnState. They are used to pass implementation-specific information into function calls so that index- and transaction-specific information can be recorded and accessed by the implementation without requiring that the API specify what this structure contains.

The contents, structure and size of the data type are defined entirely by the implementation. It is also in charge of allocating and deallocating the memory that holds the structure, and simply hands the API a pointer to its location on the heap.

The IdxState structure is used to keep track of an index that has been opened by a specific thread. It is used when opening and closing connections to an index, as well as when a query is being run on an index. It is not needed for transaction-specific function calls. It is created by the implementation during the openIndex call and it is destroyed during the closeIndex call.

The TxnState structure is used to keep track of transaction-specific information that is also particular to a specific thread. It is the only argument needed for the functions that begin, commit or abort a transaction, and is included in all of the function calls in which a query is sent to an index. It is created by the implementation during the beginTransaction call and it is destroyed during either the commitTransaction or abortTransaction calls.

2.2 Unit Tests

Unit tests were developed with a dual purpose: to verify that the implementation behaves properly and to show the participants the basic idea behind the way their code will be used and what the expected edge-case behaviors are.

There are three threads that run during the unit tests, and the tests are run over three indices. The primary thread creates the primary index before branching off two other test threads to run other tests. Meanwhile, it inserts, deletes and accesses data in the primary index in a series of transactions designed to work the basic functionality of the API, including supplying multiple `payloads` for the same `Key`, scanning through the index using `getNext` and then properly indicating the end of the index, ensuring that duplicate `Key/payload` pairs cannot exist in the same index, aborting transactions and executing transactions that use multiple indices.

While the main tests are running in the primary thread, a second thread simultaneously runs some basic tests over a separate index to ensure that the two can exist simultaneously and their data is kept separate from one another.

A third thread runs a test to ensure that the transactional guarantees required from the index are respected by the implementation under test. It continuously queries the primary index, checking to make sure that it cannot see data that only exists mid-transaction during the primary tests. It loops the same test over and over until the primary test has finished all of its transactions.

Phantom Problem

One test that was not handed out to the contestants during the contest but was run over the each implementation is a test to ensure that an index properly handles the phantom problem. The phantom problem is a bug which arises when "phantom" data suddenly appears in an index mid-transaction. This generally occurs when a database uses a fine-grained locking scheme within an index, and so when a transaction is scanning through a section of data, only the data that is in the index at that time is locked, rather than the range as a whole. Thus, when a second transaction inserts a new piece of information within that range and commits while the first transaction is still running, the conflict is not detected and the first transaction sees this new information, which violates the 'repeatable reads' requirement of transactions.

The test for this situation uses two threads, one of which is a single transaction which loops continuously over an entire index using `getNext`. Meanwhile, a second thread attempts to add data into the index at the same time in a separate transaction. If at any time the first thread sees any of the new data appear, then the implementation does not properly handle the phantom problem. However, if the implementation either does not allow the second thread to insert new data while the first thread is looping through the index, or if the first thread does not see any new data, then the implementation properly handles the phantom problem.

2.3 Benchmarks

We developed a suite of benchmarks to compare the run-times of different implementations under the same work load. Different benchmarks stress different aspects of a system by making small adjustments to the workload, such as the number of threads running, how much data is used to populate each index, or the distribution of keys.

2.3.1 Initialization

We released the primary benchmark along with the API and unit tests so that the participants would be aware what behavior the contest emphasized and what exactly was being timed. Constant arguments to the benchmark control how many indices are created for the benchmark, how many threads are used to populate and run queries over the indices, and how many populate inserts and queries each thread runs. It is also passed a value used to seed the random number generator so that it performs the same inserts and queries in the same order when given the same seed, even though it is run across multiple threads.

When the test begins, it initializes all of the data structures and generates random values to be used for the tests. Pre-generating all of the values used in the test significantly decreases the overhead time required to run the benchmark so that the reported time is primarily spent in the implementation rather than the benchmark itself.

Each type of `Key` has its values stored in its own array, as well as the `payload` values. The threads share these arrays, which allows for repeated `Keys` and `payloads` to appear. Furthermore, each thread has its own array of random integers. Whenever a new random value is needed by a thread, it grabs the next integer in the array and uses it as an index into the relevant array, or simply as a random number. Because each thread has its own array, the rate at which one thread uses the random numbers does not affect what numbers the other threads see. Furthermore, because the numbers are generated using a seeded random number generator, they are consistent across multiple runs of the test.

2.3.2 Testing

Once the initialization stage is complete, the actual testing of the implementation begins. The time it takes for each part to run is recorded by the benchmark. First each of the indices being used in the test is `created`, each in a separate thread, using as many threads as there are indices. Therefore there will not be fewer threads than indices used overall for the benchmark.

Once each of the threads has finished the creation step, the threads populate the indices with randomly generated `Key/payload` pairs. Any thread can populate any of the indices; in each iteration a random index is selected and then a `Record` is inserted with a randomly generated `Key` and `payload` combination.

Once the threads have finished populating the indices, each thread begins running tests over them. There are three possible tests a thread can run on an index: `scan`, `get` and `update`. Each test runs in its own transaction, and if the implementation deadlocks during the test, the transaction is aborted and the same test is re-run. The benchmark keeps track of how many deadlocks the implementation reports, as well as how many transactions complete successfully and how many fail. These numbers are reported at the end of the benchmark, along with how long it took to run.

The `scan` test, chosen 10% of the time, picks a random index and a random `Key` that may be in that index. It then calls `get` on that `Key` and then scans forward through the index using `getNext` between 100 and 200 times. This test is meant to determine how well an implementation can go through an index in order.

The `get` test, chosen 30% of the time, picks a random index to test and then calls `get` between 20 and 30 times with a series of random `Key`s of the appropriate type.

This test is meant to determine how well an implementation can jump from one place in an index to another.

The `update` test, chosen 60% of the time, picks a random index to test and then generates a random `Key/payload` pair to insert into the index. It then generates a new `Key` and deletes all entries under that `Key` from the index. This is done between 5 and 10 times for that index before the transaction is complete. This test is meant to stress an implementation's ability to update its contents quickly. It should be noted that, while inserting a `Record` into an index will almost always succeed (it will only fail if the `Record` is already in the index), the call to delete all `Records` under a `Key` will not always remove any information from the index due to the fact that the `Key` is not guaranteed to have been inserted into that index. However, this discrepancy should be counterbalanced by the fact that the deletion will sometimes remove multiple `Records` from the index under the same `Key`.

Once all of the threads have finished testing the implementation, the time it took for all three stages to occur is recorded and reported back, along with the number of deadlocks, failed transactions and completed transactions accrued over the course of the benchmark.

2.3.3 Benchmark Variations

The primary benchmark can be run with a variety of behaviors by varying the constants controlling the number of indices, threads, insertions and tests run each time. However, in the system described above, the distribution of data used for `Keys` and `payloads` is drawn from a uniform distribution, and only one index is used at a time for each test. This is not necessarily the most stressful test to run over a database, and so two variations of the benchmark were developed.

One variant behaved similarly to the primary benchmark described above. However, instead of using all three possible `Key` types, only 64-bit integer `Keys` were used. Furthermore, only the 8 highest bits were varied between `Keys` in one instance of the test, and only the 8 lowest bits in the other. This causes an unfriendly `Key` distribution with many duplicates that exposes weak hash functions and non-general optimizations.

The second variation on the benchmark also compared the different implementations using the standard `Key` distribution, but used multiple indices in each test transaction. For this test, two indices are opened at the beginning of the transaction. Then, a random `Key` is generated and an associated `Record` is retrieved from the first index. The `payload` associated with this `Record` is then used as a `Key` to do a lookup in the second index. Because `Keys` must also be possible `payloads`, only variable-length string `Keys` can be used.

2.4 Example Implementation

We provided an example implementation of the API that used Berkeley DB. The point of the example implementation was twofold. It tested the API to ensure that it contained enough information for the required tasks to be completed appropriately. It also gave a baseline example for the participants to see what behavior was expected of their implementations that was not specifically covered in the API or unit tests.

Berkeley DB [3] is an open-source library that provides an embedded database capability. It originated at UC Berkeley in the early 90s and has since been acquired by the Oracle Corporation for further development and distribution. It is designed to provide all of the traditional database behavior, including full crash recoverability and scalability while eliminating the client/server communication and SQL processing delays by allowing users to embed the code directly into their software.

However, Berkeley DB's wide range of features makes it a much slower system than one that is aimed specifically for this competition. It is not a main memory system, and in order to provide crash recovery it makes many disk writes that could have been dropped for the purposes of our competition. Furthermore, it supports significantly larger databases than any of our benchmarks so has a large overhead to keep track of this potential information. However, because it was not designed with this competition in mind, Berkeley DB does not allow for these features to be turned off, and as such it has very poor performance with our benchmarks relative to the submissions to the competition.

Chapter 3

Website

A website was created to allow users to test their implementations on the machine used to test the submissions once the competition was closed. The machine is a dual-4 core 2.6 GHz Intel Xeon 5430 (64-bit) with 16 GB of RAM. It runs Red Hat Fedora Core 10.

In order to ensure that every test run on the machine had the same resources available for every user, we created a submission system to run the users' tests. Users submit their binary to be tested, at which point it is entered into a queue. One at a time, a test is taken from the queue and run in isolation. Once the test is complete, the results from the test are entered into a database and the next test is begun.

3.1 Security

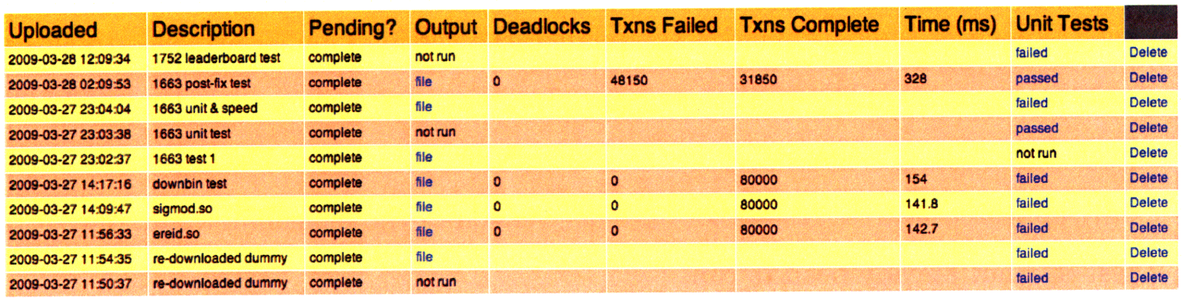
We took a few precautions in order to provide a modicum of security against the unknown and unchecked binaries being executed on the testing machine. We use a special user account on the machine whose sole purpose is to run tests for participants. When a test is run, the participant's binary and various testing binaries and supporting files are copied into a new folder by a system call. Then, the working directory is switched into that folder, and `chroot` is called for the process in order to isolate it from the rest of the system by changing the apparent disk root directory. At this point, the process that executes the tests on the participant's unknown binary believes that the root of the system is that folder, and cannot see or access any other files on the machine.

Creating a jail using chroot does provide some security, but care must be taken not to provide any hard links to files outside of the directory. This would negate the isolation and allow users access to the entire system. Even without creating any obvious ways out of the jail, this system does not ensure security. A binary might be able to exploit a buffer overflow in the provided C libraries to escape the limited scope of the system. However, this would require a lot of work by an attacker, and would have to be a deliberate attack. This system may not block all malicious users, but it does defend against accidents.

The script used to run each test and enter the results of the test into the database once it completes also keeps track of how long an individual test has been running. If a test runs longer than desired, the test is killed and no results are reported back to the user. There is also a system thread running in the background that checks for stray processes related to the tests and kills them if they have been running longer than is reasonable. Not only does this prevent a buggy implementation from never completing its benchmark test and thus hold the machine hostage against all other users' tests, but it also ensures that each user is given the same resources while his test is running.

3.2 Individual Results

The results of participants' tests are displayed on the web interface. Each user creates a unique login on his first visit to the website. Once he has logged in, any tests he requests to be run are associated with that user name. On his home page, a table of test results and pending tests for his account are available. For completed tests, the information available includes when the test was originally requested, the user-supplied description for the test, a link to a file of output produced by the test (containing the output from any printlines the user inserted into his code, as well as some general output of results from the benchmark itself), how many deadlocks



Uploaded	Description	Pending?	Output	Deadlocks	Txns Failed	Txns Complete	Time (ms)	Unit Tests	
2009-03-28 12:09:34	1752 leaderboard test	complete	not run					failed	Delete
2009-03-28 02:09:53	1663 post-fix test	complete	file	0	48150	31850	328	passed	Delete
2009-03-27 23:04:04	1663 unit & speed	complete	file					failed	Delete
2009-03-27 23:03:38	1663 unit test	complete	not run					passed	Delete
2009-03-27 23:02:37	1663 test 1	complete	file					not run	Delete
2009-03-27 14:17:16	downbin test	complete	file	0	0	80000	154	failed	Delete
2009-03-27 14:09:47	sigmod.so	complete	file	0	0	80000	141.8	failed	Delete
2009-03-27 11:56:33	ereid.so	complete	file	0	0	80000	142.7	failed	Delete
2009-03-27 11:54:35	re-downloaded dummy	complete	file					failed	Delete
2009-03-27 11:50:37	re-downloaded dummy	complete	not run					failed	Delete

Figure 3-1: Screen shot of example results for a participant.

occurred during the test, how many transactions failed during the test, how many transactions completed during the test, how long the test took to run, and the output from the unit tests, if the user chose to execute them as well.

When a test is requested and a binary is provided, the binary is saved to disk to be used when the test is run. Furthermore, once a test is run the output produced by the benchmark and the the output produced by the unit tests are each saved in a file that is also saved to disk. These results are made available to the user for inspection so they can see how their code behaved, including their own debugging or informative printlines.

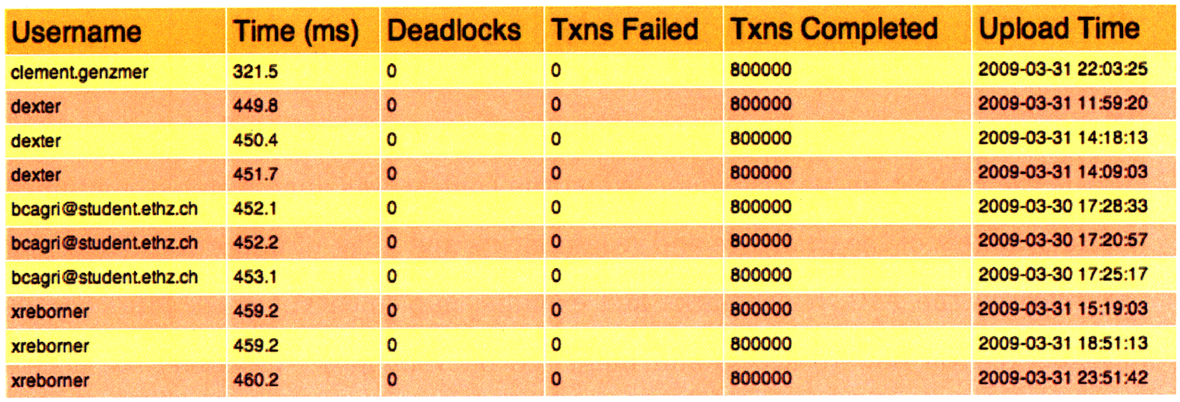
The web interface also allows users to delete their own results from the database that keeps track of all of the tests. When a result is deleted, the entry is completely removed from the database, although the supplementary files (the binary and the output files) remain on disk. This allows us (the contest managers) to look at the results from any old tests and re-run any old tests, even if the user can no longer see the results via the web interface. However, because the output files are kept in a location accessible via the Internet, there is no system in place to prevent anyone from looking at the output of other user's tests, including those that have been deleted from the database.

Over the month and a half that the website was available before the contest was closed, we revised the benchmark several times. The changes made to it varied from small bug fixes to changing how the random numbers were generated, and could affect the run-time of participant's code. In order to distinguish between the different benchmarks, a version number was included in the database information for any test run. Whenever the benchmark was updated, the version number attached to all new tests was incremented. After this occurred, a button was shown on the users' home page that allowed them to re-run the benchmark test on an old binary. This button only appeared next to results for an older version of the benchmark, and only if the test had not already been re-run.

3.3 Leaderboard

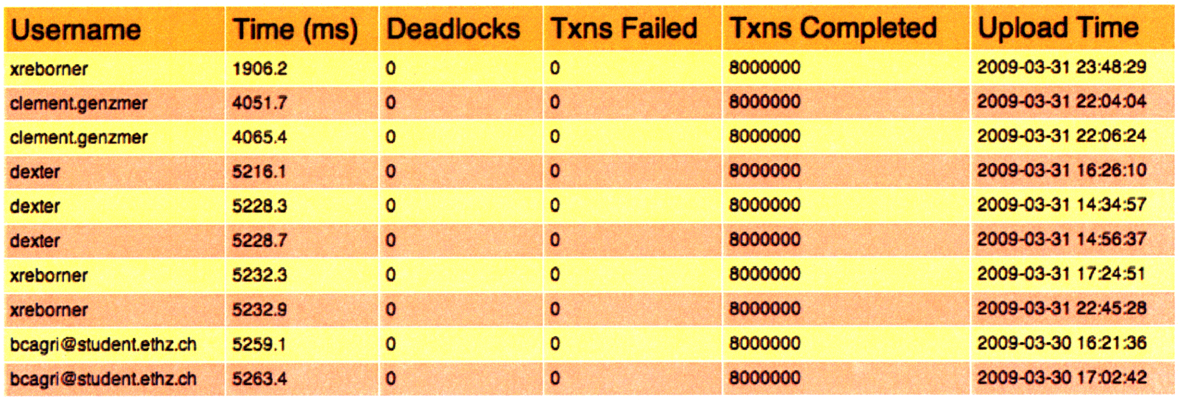
The website also provided a leaderboard of the 10 fastest results for two basic benchmark tests. Each user was allowed no more than three spots in the table, preventing the fastest user from re-running the same code multiple times to fill up the table with identical results. When submitting a binary to be tested on the site, participants were required to specifically state that it is being submitted for the leaderboard test. This automatically forces the system to run the unit tests over the binary, and if the binary does not pass all of the unit tests it is not eligible for the leaderboard. Furthermore, if a test does not complete all requested transactions (due to transaction failures or deadlocks), it is not eligible for the leaderboard. Finally, only tests run on the most recent version of the benchmark would appear on the leaderboard. Whenever a new version of the benchmark was released, all of the results on the leaderboard for the previous version of the benchmark would be re-run to immediately repopulate the leaderboard. This has the potential to show slow results on the leaderboard, but they quickly fall off the leaderboard when faster runs complete. Also, users have the ability to delete any of their own records from the system if they do not want them to appear on the leaderboard.

The two benchmark tests for which leaderboards were created were variants of the basic benchmark tests, with different loads. Each ran with a random number of indices and 50 threads, created indices of all three key types and ran all three types of benchmark tests (see section 2.3.2). One leaderboard test ran with 400 inserts per thread in the populate phase and 1600 tests in the testing phase. The final values of this leaderboard are shown in Figure 3-2. The other leaderboard ran 10 times as



Username	Time (ms)	Deadlocks	Txns Failed	Txns Completed	Upload Time
clement.genzmer	321.5	0	0	800000	2009-03-31 22:03:25
dexter	449.8	0	0	800000	2009-03-31 11:59:20
dexter	450.4	0	0	800000	2009-03-31 14:18:13
dexter	451.7	0	0	800000	2009-03-31 14:09:03
bcagri@student.ethz.ch	452.1	0	0	800000	2009-03-30 17:28:33
bcagri@student.ethz.ch	452.2	0	0	800000	2009-03-30 17:20:57
bcagri@student.ethz.ch	453.1	0	0	800000	2009-03-30 17:25:17
xreborner	459.2	0	0	800000	2009-03-31 15:19:03
xreborner	459.2	0	0	800000	2009-03-31 18:51:13
xreborner	460.2	0	0	800000	2009-03-31 23:51:42

Figure 3-2: Leaderboard displayed on the contest website.



Username	Time (ms)	Deadlocks	Txns Failed	Txns Completed	Upload Time
xreborner	1906.2	0	0	8000000	2009-03-31 23:48:29
clement.genzmer	4051.7	0	0	8000000	2009-03-31 22:04:04
clement.genzmer	4065.4	0	0	8000000	2009-03-31 22:06:24
dexter	5216.1	0	0	8000000	2009-03-31 16:26:10
dexter	5228.3	0	0	8000000	2009-03-31 14:34:57
dexter	5228.7	0	0	8000000	2009-03-31 14:56:37
xreborner	5232.3	0	0	8000000	2009-03-31 17:24:51
xreborner	5232.9	0	0	8000000	2009-03-31 22:45:28
bcagri@student.ethz.ch	5259.1	0	0	8000000	2009-03-30 16:21:36
bcagri@student.ethz.ch	5263.4	0	0	8000000	2009-03-30 17:02:42

Figure 3-3: Second leaderboard displayed on the contest website.

many inserts and tests per thread, which increased the overall size of the database and the number of tests run by a factor of 500 each. The final values of this leaderboard are shown in Figure 3-3.

By providing two different leaderboard tables, we acknowledged that different implementations could excel under different workloads. Some implementations could handle small databases extremely fast, but not scale well. Some implementations could scale nicely, but have a lot of overhead to support the scalability that would slow it down with smaller databases.

Participants were informed that the results displayed on the leaderboard were only used as reference, and did not indicate who would win the contest. Other factors than simply how fast the code ran would be involved in the decision.

3.4 Organizational Website

A separate website was also maintained to post announcements and basic information about the contest for participants. When the contest was announced, it contained a basic overview of the challenge and requirements and a list of important dates. Once we released the API, benchmarks and example implementation, the site contained instructions on how to access and build the relevant files. Whenever we updated benchmarks, API or unit tests, we also posted an announcement on the website explaining what changes had been made and any relevant additional information. All of the benchmarks, unit tests and phantom test are now posted on the website, which can be found here: `http://db.csail.mit.edu/sigmod09contest/`.

Chapter 4

Submissions

While there were over 90 different accounts created to access the testing site, by the deadline of 11:59 pm on March 31st, 2009 only 17 binaries had been submitted. Of these 17 submissions, one had not turned in a valid binary and there was no response to an email sent to the provided email account. A second submission could not run on our 64-bit Linux machine and so was also not a valid submission. The 15 remaining submissions were tested using the various benchmarks, unit tests and phantom test.

Submissions were received from teams associated with schools in countries all over the world, including Australia, China, Denmark, France, Germany, Poland, Switzerland and the United States. The number of people on a team varied from one person to up to three people listed as working on the code.

Five different versions of the basic benchmark test were run over all 15 submissions, along with the unit tests and the phantom test, in order to get a basic idea how each implementation performed. The five tests run were the two leaderboard tests used on the website (see Table 5.1), a test using 30 threads and indices, a test using 55 threads and indices, a test that inserts 1000 times as much data per thread during the populate phase in order to push the limits of the available memory and a test that only uses 5 indices (see Table 5.2). Each test was run multiple times with different random seeds for each implementation, and the average time was reported back.

From the results of these tests, eight submissions were chosen that successfully completed all of the benchmark tests in a short time. Some of these implementations did not successfully pass the phantom test, but this was considered a fixable issue, and the test had not been released to the participants.

The implementations not chosen for closer inspection were left behind for various reasons. Some of the implementations timed out before being able to complete the given benchmarks or failed the unit tests. Others completed all of the tests successfully, but were significantly slower than the eight that were chosen for closer inspection on all of the benchmarks, and so were not investigated further.

4.1 Data Structures

One important decision in every design is the data structure used to store the `Records` in memory. However the information is stored, `Records` must be able to be accessed either via a direct lookup using `get`, or while scanning the indices using `getNext`. Furthermore, the data cannot be static, as `Records` are occasionally added to or removed from the indices, even after the indices have been populated.

Most participants chose to use some form of tree to store the database information. Trees allow for roughly $O(\log n)$ lookup times, while maintaining a global order among the `Keys`, so the index can be scanned as well. There is a wide range of different tree-based data structures, however, so even implementations using a tree must decide what type of tree to use.

One implementation uses T-trees, which were invented with the intent to be better for main memory databases than B-trees. Rather than keeping all of the data stored in the structure itself, T-trees hold onto pointers to the data since all of the information is kept in memory and so can be accessed quickly without the random seek penalties of data stored on disk. By moving the core of the data out of the main tree structure, this also allows the tree to be restructured more quickly when it requires balancing, because less data is moved.

Another implementation uses CSS-(cache-sensitive search) B+-Trees. Cache-concious data structures treat the lines in the cache as the unit of memory of which they must be most aware, as opposed to blocks on the disk. This optimization is merged with a design feature of B+trees to keep the children of a node contiguous in memory so that only one pointer must be kept, thus minimizing the overhead of the data structure. The trees are also occasionally restructured once they have grown significantly in size. This keeps the trees balanced so that none of the data is particularly unfavorable to access in the future, but does take the system offline for a significant amount of time while each tree is being reorganized.

A few different implementations used variations on a prefix tree, or trie, as their primary data structure. Tries use the location of a node in the tree to indicate the key with which a value is associated. This helps minimize overhead in the trie by lessening the amount of key information stored in each node, while still keeping the keys in an absolute order. Further analysis of tries can be found in Section 5.3.1 in the discussion of *dexter*'s implementation.

A few implementations chose not to use a tree as the primary data structure. Instead, they chose a hash-based method of storage, like a multimap or a collection of simple hash tables. This allows for extremely fast access times, but forces the implementer to create a scheme to keep track of the global ordering of the keys. One implementation that does this, by *clement* is described in more detail in Section 5.3.1.

4.2 Concurrency Control

All of the official tests are run on a multi-core system and use multiple threads to run tests at the same time. This means that two different transactions may attempt to read or update an index at the same time. If this occurs, transactional guarantees are broken and the information stored in the database can be corrupted.

4.2.1 Locking

In order to prevent this problem, many implementations use some form of locking to prevent different threads from interfering with each others' behavior. Of the implementations that used locking, all of them used whole-index locking rather than locks on smaller pieces of an index. However, while some implementations used a more traditional locking system using both shared and exclusive locks, others chose to avoid handling two types of locks by holding exclusive locks on all open indices for each transaction.

Other implementations did not emphasize locks as much, but instead relied on either optimistic concurrency control in order to isolate transactions, or created a multi-version isolation system that takes a 'snapshot' of an index for each transaction.

Both of these designs work in a similar manner. When a transaction begins, some data from the index is copied into a local 'sandbox' for the transaction to update as needed. Once the transaction is ready to commit, an algorithm checks to see if the updates conflict with other transactions that have committed or are currently running in the system. If there are no conflicts, the changes are committed. If the updates do conflict, the transaction is aborted. This is particularly effective when most transactions do not conflict with each other, as is the case with the benchmarks used to judge this contest.

4.2.2 Deadlocks

If the locking system in the implementation allows situations in which one transaction cannot possibly complete due to another transaction's behavior, then one of the two transactions must be reported as being in a deadlock and aborted by the client code.

Deadlocks can be detected either by tracking the resources being used by different transactions and checking for conflicts, or simply by keeping track of how long a transaction has been waiting for a specific resource. If it has been longer than some predetermined time, the system declares it to be deadlocked and aborts it. While this is not always strictly true, it can vastly simplify the deadlock handling code in an implementation.

Some implementations do not require any deadlock-detecting code due to the locking system used. However, this often means that the locking system may be overly conservative, or that not all of the desired features have been fully realized or are buggy in the implementation in question. Some cannot properly handle opening an index partway through a transaction, whereas others cannot handle using multiple indices for a single transaction, which is common behavior in databases although was not part of our originally released benchmark.

4.3 Memory Management

Many participants chose to create custom memory allocators optimized for their implementation's needs. By reserving space from the operating system using a `malloc`-like call, they can then choose exactly how to subdivide the space so that fragmentation due to variable block sizes can be avoided. The preallocated space is held in a memory pool controlled by the implementation. Whenever new space is needed for the database, it is released from the memory pool rather than going all the way down to the kernel with the request. This allows for much tighter control over where different pieces of the system live relative to each other in memory. In Section 5.3.1 the custom memory management scheme used by *clement* is discussed in further detail.

4.4 Multicore Support

The machine used to test all of the implementations has a quad-core processor that can run multiple threads for a single process at the same time. The benchmark uses this fact to run transactions on many threads at the same time, which will ideally increase overall throughput. We implemented this using Pthreads, a C library that creates multiple threads within the same process. Once the threads have been created, Linux manages mappings of threads to cores.

This multi-threaded strategy can also be used by the implementations to speed up internal operations that do not depend directly upon each other. Another potential way to take advantage of the multi-core system is to assign specific tasks to a particular CPU and keep all of the relevant information for it stored in the local memory. This core affinity can decrease the penalties for non-uniform memory access between the different CPUs.

While these strategies were available to the participants to use in their implementations, none of them appear to have taken advantage of any of them.

Chapter 5

Evaluation

The eight implementations that were evaluated and investigated most thoroughly used a variety of methods to handle the different design issues. As a result, their behavior excelled in different ways, which will be evaluated below.

5.1 Tests

As discussed in Section 2.3, a variety of benchmarks were developed for performance testing. Each benchmark accepts five control variables that adjust how many times the test is run, how many inserts each thread runs during the populate phase, how many tests each thread runs during the testing phase, how many indices are created and tested and how many threads are used to do the creation and testing in the benchmark.

We made two instances of the primary benchmark available to participants to test their code while it was being developed. Table 5.1 shows the control variables used for these two instances of the primary benchmark. The number of indices used for these tests is a randomly generated number between 1 and 50, inclusive, that varied for each iteration of the benchmark. However, due to the seeded random number generator, we tested each implementation with the same random number of indices. All of the other control variables were constant values that remained consistent between iterations of the test.

Table 5.1: Control Variables For Preliminary Benchmark Instances

Instance	**Runs**	**Inserts/Thread**	**Tests/Thread**	**Indices**	**Threads**
Preliminary 1	10	400	1,600	[1,50]	50
Preliminary 2	10	4,000	16,000	[1,50]	50

Table 5.2: Control Variables For Additional Primary Benchmark Instances

Instance	**Runs**	**Inserts/Thread**	**Tests/Thread**	**Indices**	**Threads**
30 Threads	10	400	8,000	30	30
55 Threads	10	400	8,000	55	55
Fill Memory	3	400,000	1,600	50	50
5 Indices	10	4,000	16,000	5	50

Once the official submissions to the contest were in, we used four more instances of the primary benchmark to examine the 15 different implementations along with the two preliminary instances. These benchmarks were used to distinguish the broken and slow submissions from those that ran quickly enough to be of interest. The control variables for these iterations are listed in Table 5.2 and were designed to stress some basic edge cases within the standard framework. These six instances of the primary benchmark identified eight implementations of interest. The code for each of these implementations was hand inspected to determine their methodology and to ensure that their code behaved appropriately with respect to our tests.

As discussed in Section 2.3.3, two variations on the primary benchmark were developed to stress the implementations in ways that the primary benchmark was not designed to do. Table 5.3 shows the control variables used to run these variations on the benchmark.

5.2 Results

The results of the various benchmark tests are shown in Tables 5.4, 5.5 and 5.6. Table 5.4 shows the results of the preliminary benchmark that the participants could run via the contest website while developing their implementations. Table 5.5 shows the

Table 5.3: Control Variables For Variations On Primary Benchmark

Instance	**Runs**	**Inserts/Thread**	**Tests/Thread**	**Indices**	**Threads**
Vary High	10	4,000	16,000	50	50
Vary Low	10	4,000	16,000	50	50
Double Lookup	10	8,000	1,600	50	50

Table 5.4: Results of Preliminary Benchmarks

Team	**Preliminary 1**	**Preliminary 2**
bcagri	4,112	49,151
clement	2,946	37,540
dexter	3,949	46,914
frame	5,960	77,756
ji	8,272	100,545
kastauyra	4,881	65,968
qbolec	6,522	55,957
xreborner	4,487	49,096
Berkeley DB	9,809,710*	could not complete*
NULL Implementation	1,577	13,794
* These tests did not complete all iterations in the allotted time. The times were extrapolated from the completed iterations, if any completed		

Numbers represent time to complete the benchmark in milliseconds.

results of the additional instances of the primary benchmark that were used for extra insight into the implementations while determining the top eight implementations.

Table 5.6 shows the results from the additional benchmarks developed once the final code had been looked over. Because the behavior in these benchmarks is non-standard, some of the implementations could not finish all of the iterations in the allotted time, either because the implementation ran slowly under the given conditions, or due to an undetected deadlock. In these situations the time reported is extrapolated from all of the iterations, if any, that completed.

Table 5.5: Results of Additional Instances of Primary Benchmark

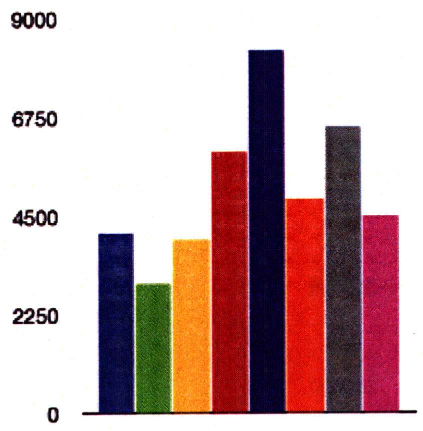
Team	30 Threads	55 Thread	Fill Memory	5 Indices
bcagri	13,112	24,101	105,369	9,235
clement	10,261	19,319	35,209	8,061
dexter	12,135	21,296	77,004	18,207
frame	21,480	34,701	157,435	13,780
ji	31,685	56,647	240,752	8,857
kastauyra	13,191	26,947	193,954	9,142
qbolec	18,503	32,406	84,217	12,554
xreborner	13,120	23,886	683,506	9,939
NULL Implementation	3,633	7,310	13,984	13,296

Numbers represent time to complete the benchmark in milliseconds.

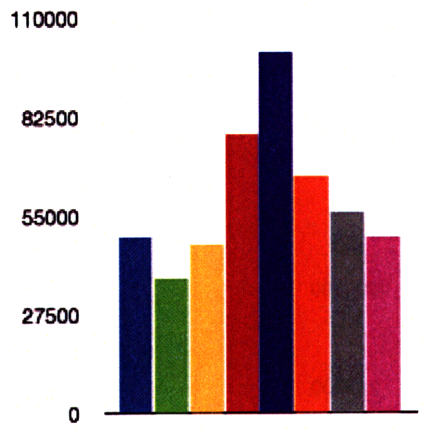
Table 5.6: Results of Benchmark Variations

Team	Vary High	Vary Low	Double Lookup
bcagri	34,177	35,660	3,109,263*
clement	41,973	34,794	stall*
dexter	29,241	32,714	61,703
frame	57,160	58,701	393,169
ji	2,049,315*	1,457,565*	822,015
kastauyra	34,370	35,600	365,249
qbolec	1,524,100*	927,516*	307,270
xreborner	28,396	30,825	258,227
NULL Implementation	11,593	11,809	99,995
* These tests did not complete all iterations in the allotted time. The times were extrapolated from the completed iterations, if any completed			

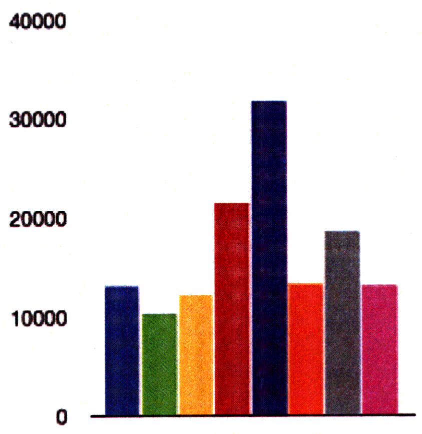
Numbers represent time to complete the benchmark in milliseconds.



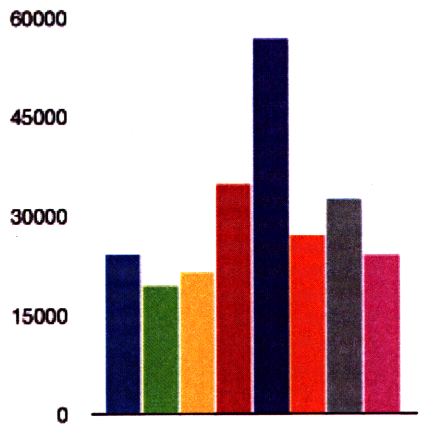
(a) Preliminary 1 Results.



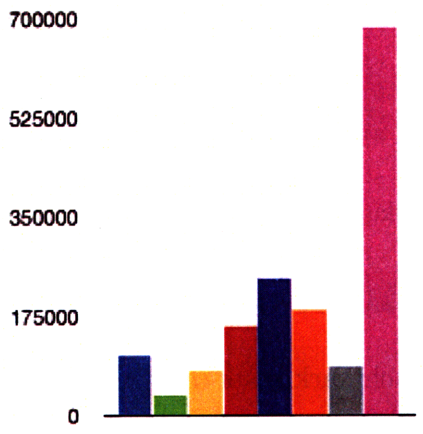
(b) Preliminary 2 Results.



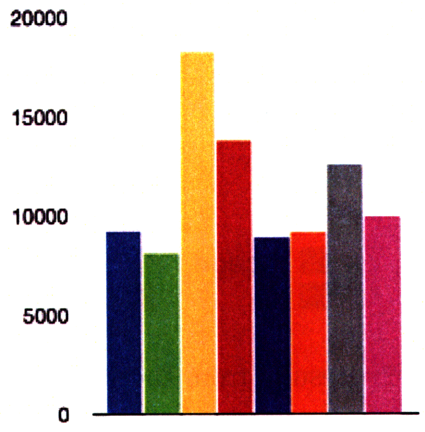
(c) 30 Threads Results.



(d) 55 Threads Results.



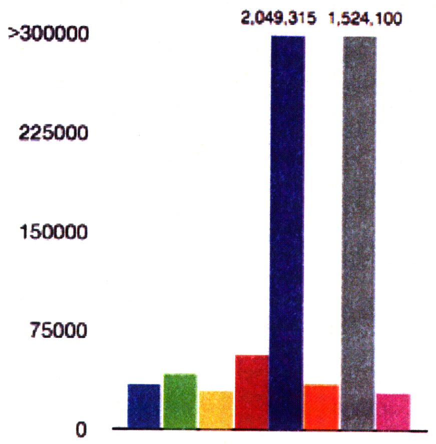
(e) Fill Memory Results.



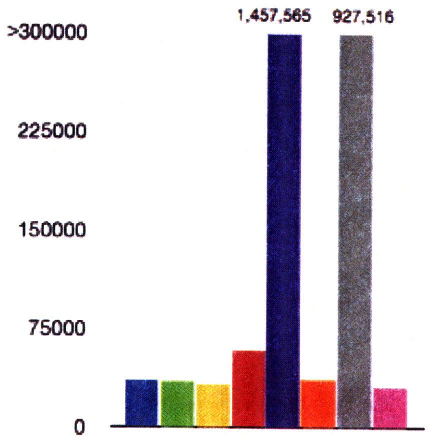
(f) 5 Indices Results.

🟦 bcagri 🟩 clement 🟨 dexter 🟥 frame
⬛ ji 🟥 kastauyra ⬛ qbolec 🟪 xreborner

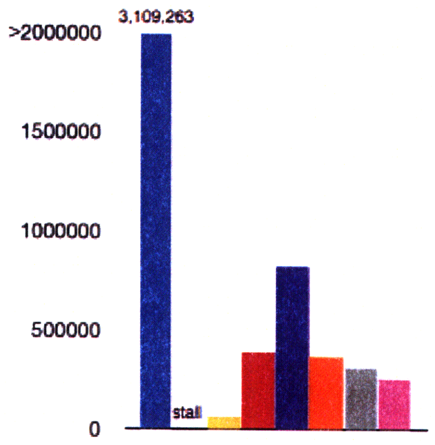
Figure 5-1: Results from the primary benchmark instances (all times in milliseconds).



(a) Vary High Results.



(b) Vary Low Results.



(c) Double Lookup Results.

🟦 bcagri 🟩 clement 🟨 dexter 🟥 frame
🟦 ji 🟥 kastauyra ⬜ qbolec 🟪 xreborner

Figure 5-2: Results from the variation benchmark tests (all times in milliseconds).

5.3 Discussion

In this section I discuss some of the implementations that exemplify the methods used. I explain their results relative to the other implementations and analyze the reasons for this behavior by discussing the corresponding code.

5.3.1 Individual Submissions

Clement

In all instances of the primary benchmark, team *clement* ran the fastest. However, for the benchmark variations *clement* was not the fastest implementation. In the Vary High and Vary Low benchmarks, *clement* was not the fastest implementation, but it was never more than 1.5 times slower than the implementation that did run the fastest. The one exception to this was the Double Lookup benchmark. Because *clement*'s locking scheme cannot handle opening two indices for the same transaction, it stalls out before a single iteration of the benchmark can complete and so no estimated time could be extrapolated.

Clement chose to use hash tables as the primary data structure for each index. By using an order-preserving hash function, which inserts keys into the hash table so that the global ordering of the keys is preserved, and merging the `Key` and `payload` together into a single value, the cost of a `get` is $O(1)$ and `getNext` is amortized $O(1)$ if the table is not too sparse. The 'hash function' uses the $\log_2(\text{size})$ most significant bits of the `Key/payload` 'key'.

This system works very well for keys that vary uniformly in the higher order bits, but other distributions of keys have a detrimental effect on performance, as shown with the Vary High and Vary Low benchmarks. In particular, one adversarial set of keys that could significantly affect the performance of this implementation would be a truncated uniform distribution in which the most significant bits of the key remain constant in all of the keys, while the less significant bits vary. While the keys themselves would be different, they would all hash to the same bucket in the array and the performance benefits from the hash table would be lost.

The benefit to using this as the hash function is that it preserves the global ordering on the `Key/payload` pairs inserted into the table. By preserving this order in the table, a range search is easily supported and does not require the implementation to search through the entirety of the index to determine the next value to return.

In order to allow the performance of this hash function to be uniform across all three provided `Keys`, *clement* created a system by which the first 10 characters of a string `Key` are compressed down to 64 bits. This is done by mapping the ASCII representations of the characters onto a numerical system designed specifically for this implementation. The non-printable characters are all mapped to the lowest possible value and are treated identically. The printable characters are each given their own unique numerical representations so that they may be distinguished in the hash. By throwing out the distinctions between the non-printable characters, it allows for the total number of bits required to represent each character to be lowered, which in turn allows for the strings to be compressed down to 64 bits.

This optimization is effective for the provided benchmarks because the only characters being used to generate `Keys` and `payloads` are the 52 alphabet characters in upper- and lower-case. In fact, the system provided by *clement*'s implementation handles more than the range of characters generated by the benchmarks. However, if the relative ordering of the non-printable characters is considered relevant, then this optimization would become less effective because there are not enough bits to sustain the distinction. This would prevent the compression of 10 characters into 64 bits, and fewer characters would need to be used for the hash function.

Clement created two types of custom memory allocators in order to optimize the use of and access to available space. The first allocator is for string `Keys` and `payloads`. A separate list is maintained for each possible string length, containing all the string of a specific length have have been deleted but whose space has not yet been reused. When a new string is being allocated, the relevant list is accessed and a location in memory of the appropriate size is determined and used. This prevents fragmentation and updates the pool of available memory when an insertion or deletion occurs in O(1), although it does create an overhead that can waste space if not many values are in the index.

The second allocator is a typical pool allocator for the structures that maintain the non-string `Keys`. It maintains a pointer to the `Key`, a hash of the `Key`, a pointer to the `payload`, and a pointer to the next element in the index if one exists.

Deadlocks are avoided by using whole-index locks. However, this design decision does not support multiple indices being used within the same transaction, which is why the Double Lookup benchmark cannot complete without deadlocking. The implementation assumes that there will never be any conflicts between the transactions due to the initial locks acquired, and so does not supply any further deadlock detection. However, the Double Lookup benchmark can create a deadlock because the order in which indices are opened for a given transaction is not guaranteed, and so two separate transactions can each hold a lock on the index that the other transaction is attempting to acquire.

By locking the whole index to avoid deadlocks, concurrency control does not require much extra support. *Clement* chose to insert the updates into the index immediately while logging the deletes to be completed when the transaction is committed. This allows the custom memory allocator to free the memory from the deleted elements all at once. If the transaction is aborted, the deletions are ignored and the inserts are rolled back, reclaiming the memory used for them, instead.

The logging system uses an array to keep track of the update operations performed over the life of a transaction. The array is not designed to dynamically resize relative to the requirements of a specific transaction, so the total number of update operations that can be recorded during

This implementation is highly tuned to the assumptions provided by the primary benchmark, including a uniform distribution of keys, only supporting the relative ordering of printable characters in the strings and only using one index per transaction. The issues that arise from these assumptions have been mentioned to *clement* and the code is being adjusted for the final competition at the SIGMOD conference in June.

Dexter

For most instances of the benchmark tests, both primary and the variations, team *dexter* took second place The two benchmark tests in which *dexter* did not take second place were the 5 Indices instance of the primary benchmark, where *dexter* was the slowest, and the Double Lookup variation, where *dexter* was the fastest. Furthermore, *dexter*'s performance on the Vary High and Vary Low benchmarks was better than *clement*'s. In fact, the non-uniform data did not have a significant effect on the implementation's performance at all.

Locking is per-index, but they are only held during the actual operation, and are read/write-specific. Transactions are isolated using optimistic concurrency control, maintaining a transaction table with time stamps to detect violations. Inserts and deletes are performed directly, and in the event of a rollback they are undone with deletes and inserts, respectively. This system prevents the system from stalling completely due to deadlock, but the per-index locking prevents it from scaling much better than other implementations with less advanced locking mechanisms.

Dexter chose to use a prefix tree with jump pointers to skip leading zeros as the primary data structure for each index. Multiple implementations used variations on a prefix tree as their primary data structure.

A trie, also known as a prefix tree, uses a node's location in the tree to show with what key it is associated, rather than storing the key in the node itself. Thus, all descendants from a given node share a common prefix represented by that node's location in the tree. While tries are commonly used to store character strings, the algorithm can easily be applied to the numerical `Keys` required by the API.

Due to the structure of a trie, given the `Keys` provided by the API, each tree will have a fixed height and relatively low complexity. Furthermore, a single node in the tree can be expanded to support multiple `payloads` associated with a single `Key`. Not only does this not expand the overall size of the tree, it also makes it much easier to delete all `payloads` associated with a single `Key`.

Given the randomly generated and uniformly distributed `Keys` used in most of the benchmark tests, tries are generally efficient in terms of overhead. However, given adversarial data specifically designed to create a sparse tree, a trie can have a very large overhead relative to the amount of data it contains. If the keys do not share many common prefixes, each key will create a series of new nodes that only exist to support a single key, rather than many.

Bcagri

Team *bcagri*'s implementation ran third or fourth fastest for most of the benchmarks. One exception to this was the Double Lookup test in which *bcagri*'s reported time was by far the slowest of the seven that completed. In fact, *bcagri*'s time had to be extrapolated to reflect an estimated time for the entire test, as only a portion of the required iterations managed to complete before the test timed out.

Locking is per-index, and uses traditional shared/exclusive locks on the indices as they are used. This implementation does not have any deadlock detection code, and thus deadlocks easily. Furthermore, because deadlocks are not detected explicitly and the locks are exclusive, the Double Lookup benchmark could potentially reach a point of deadlock that is never resolved, and the test stalls.

Bcagri's implementation has some bugs involving synchronization of indices, both when being created and when being opened. Furthermore, this implementation fails the Phantom Test, most likely due to some bug in the data structure implementation. However, these issues have been mentioned to *Bcagri* and should be resolved before the final competition in June.

Bcagri chose to use T-trees as the primary data structure for each index, with a few modifications to the original design in order to maximize the cache-friendliness of the structure. As introduced briefly in Section 1.3, T-trees are built on top of self-balancing binary trees, but store a range of elements in a node. When searching the tree, the two end keys of a node, containing the minimum and maximum values of the elements in the node, are compared against the queried value. Depending on the results of these comparisons, the search continues down the left branch of the tree, the right branch of the tree, or within the node that was tested. Within a node, the values are stored in order.

In spite of T-trees' attempt at cache-consciousness, their behavior is similar to that of a basic binary tree. Their performance is improved by grouping multiple values together in a node, thus potentially allowing fewer comparisons by virtue of a shallower tree. However, each node is only used for two comparisons before the search continues down the tree, and the two values are at opposite ends of the node, so they are often kept on separate cache lines. Due to this behavior, most of the data stored in a cache line is not utilized until the final node is searched.

T-trees do not obviously support multiple `payloads` being associated with a single `Key`. If each `Key/payload` combination is stored as a separate value within a node, then it is possible for a single `Key` to require up multiple nodes to store all of the `payloads` associated with it. Not only is this a potentially large drain of space within the tree, it could also require additional metadata to be stored and compared during a query.

Bcagri made some modifications to the basic T-tree design to offset some of these issues. Some of these changes include storing the maximum and minimum keys in each node together on the same cache line and storing the data within the nodes themselves in a cache-conscious manner. Duplicate `Keys` are stored in an auxiliary binary search tree rather than storing duplicates within the T-tree itself. This also allows for easy removal of all `payloads` associated with a single `Key`.

Frame

Team *frame* generally completed the benchmarks as one of the slower implementations, although never the slowest. However, in the Fill Memory instance of the primary benchmark, *frame*'s performance indicated that the implementation has a relatively low overhead and scaled well for large data sets.

This implementation uses a variation of a cache-conscious CSS-B+-Tree, although stays closer to the original concepts of CSS-trees in order to make full use of its cache usage and query performance. The tree occasionally rebuilds itself while still responding to requests to update the tree by keeping a list of pending inserts and deletes. While this keeps the tree more balanced, it is likely also the cause of its increased slowness for the All Memory instance of the primary benchmark.

Locking is per-index, although leaf-level locking was also implemented and can be turned on by changing a single variable. Each granularity of locks use a standard optimistic concurrency control system with logging, and performed similarly for all of the benchmarks.

Deadlocks are detected using timeouts, which means that it will not perform well under contention, such as in the Double Lookup benchmark. Also, because it does not strip off common prefixes before inserting `Keys` into the CSS-tree, it does not perform well for the vary-high and vary-low tests because it must reorder the tree more often in an attempt to keep it balanced.

5.3.2 Optimization Analysis

All of the implementations submitted to the contest were optimized in part or in large for the benchmarks used to compare and rate them. Some of these optimizations are more effective than others, as seen by the varied results in Section 5.2.

One effective optimization is the use of a custom memory manager with one or many memory pools, allowing for careful control over where and how the data is organized. Not only does this help minimize fragmentation, it also allows some implementations to lay out their data structures in a cache-conscious manner.

Another useful tactic is implementing optimistic locking. Many of the tests run in the benchmark do not change the data stored in the indices, so multiple threads should be able to access the same index without incident. While this does add some bookkeeping overhead, the overall effect is still positive.

The choice of data structure has an effect on an implementation's performance that has the potential to be much more drastic than many other design decisions. As discussed in Section 1.3, there have been a number of different tree-like structures proposed to be most effective for main-memory data storage. According to the results of our benchmark, the most efficient design appears to be *clement*'s hash table-based system. However, that design is highly optimized for our specific benchmarking tests. Of the more general solutions, *dexter*'s prefix tree appear to run the fastest over our benchmarks, although *bcagri*'s T-trees performed admirably when not dealing with multiple indices in the same transaction.

5.3.3 Conclusion

If a contest winner had to be chosen from this preliminary analysis, it appears that *clement*'s implementation would be the best choice. While it cannot properly handle multi-index transactions, this was not explicitly mentioned as one of the original requirements of the contest. It uses an order-preserving hash function to distribute the data in each of the indices so that each `Record` can be accessed quickly, but range searches are still easily supported. This handles all of the behavior required in the original contest description quickly and efficiently, and without too much wasted space.

Clement's code is highly optimized for the provided primary benchmark, which is why we created the variations on the benchmark to stress some of the cases that were not handled well. This will force *clement*'s implementation (as well as other teams') to be made more flexible before the final decision is made.

In any competition participants must optimize their implementations for the supplied benchmarks. Creating benchmarks that will properly stress all of the different aspects of a design is difficult, and given enough time and effort applied to a collection of varied benchmarks, participants will still find a way to take advantage of their structure and assumptions.

In the future, the benchmarks provided during the development stage of the contest should cover a wider range of behavior so that participants cannot hone their implementations too finely for one set of requirements. While the overall performance may be decreased, the increased flexibility of the implementations is more important so that the code produced can be used in other contexts.

Bibliography

[1] Rudolf Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Mathematical and Information Sciences Report*, 20:all, 1970.

[2] Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. Cache-oblivious string b-trees. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–242, New York, NY, USA, 2006. ACM.

[3] Oracle Corporation. Berkeley db, oracle embedded database. `http://www.oracle.com/database/berkeley-db/db/index.html`, March 2009.

[4] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowl. and Data Eng.*, 4(6):509–516, 1992.

[5] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[6] Tobin J. Lehman and Michael J. Carey. A study of index structures for main memory database management systems. In *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*, pages 294–303, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[7] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 78–89, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[8] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.