Abstract

Concurrency is hard. A concurrent system is a collection of programs that simultaneously operate on shared data. Concurrency introduces ambiguity in execution order, because these programs may be run across threads, processes, or, in the case of distributed systems, networks. It is precisely this ambiguity that causes a certain class of failures known as race conditions. Race conditions manifest themselves in subtle ways in concurrent systems, but they can often have catastrophic consequences.

Many programming languages provide fundamental abstractions such as locks, semaphores, and monitors to protect against race conditions. Some, like Rust [10], are even able to statically detect race conditions between concurrent threads. But none are able to guarantee that race conditions will be exhaustively eliminated in a distributed system. For this reason, building correct distributed systems is extremely challenging.

In this thesis, we will explore transactional programming languages and their application to building race-free distributed systems. In Chapter 1, we will discuss a novel transactional programming language called Caustic and its associated runtime. In Chapter 2, we will discuss its underlying distributed, transactional storage system, Beaker.
Contents

1 Caustic: A Transactional Programming Language 3
   1.1 Race Conditions ............................................. 3
       1.1.1 Transactional Databases ................................. 4
       1.1.2 Pessimistic Concurrency ................................. 5
       1.1.3 Optimistic Concurrency ................................. 6
   1.2 Runtime ......................................................... 7
       1.2.1 Execution .................................................. 8
       1.2.2 Optimizations ............................................ 8
       1.2.3 Performance .............................................. 9
   1.3 Standard Library ............................................... 10
       1.3.1 Typing ..................................................... 10
       1.3.2 Records ................................................... 11
       1.3.3 Collections ............................................... 11
       1.3.4 Math ....................................................... 12
       1.3.5 Control Flow ............................................ 12
   1.4 Compiler ......................................................... 13
       1.4.1 Implementation ........................................... 13
       1.4.2 Programmability ......................................... 14
   1.5 Conclusion ...................................................... 18

2 Beaker: A Distributed, Transactional Database 21
   2.1 Consensus ....................................................... 21
       2.1.1 Terminology .............................................. 22
       2.1.2 Two-Phase Commit ....................................... 22
       2.1.3 Paxos ..................................................... 23
       2.1.4 Generalized Paxos ...................................... 24
       2.1.5 Egalitarian Paxos ...................................... 25
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2 Distributed Transactions</td>
<td>26</td>
</tr>
<tr>
<td>2.2.1 Terminology</td>
<td>26</td>
</tr>
<tr>
<td>2.2.2 Protocol</td>
<td>27</td>
</tr>
<tr>
<td>2.2.3 Correctness</td>
<td>28</td>
</tr>
<tr>
<td>2.2.4 Reconfiguration</td>
<td>30</td>
</tr>
<tr>
<td>2.3 Evaluation</td>
<td>31</td>
</tr>
<tr>
<td>2.3.1 YCSB</td>
<td>31</td>
</tr>
<tr>
<td>2.3.2 Contention</td>
<td>32</td>
</tr>
<tr>
<td>2.4 Conclusion</td>
<td>34</td>
</tr>
</tbody>
</table>
Chapter 1

Caustic: A Transactional Programming Language

Caustic is a transactional programming language for building race-free distributed systems. It allows programmers to write applications as if they were single-threaded, but distribute them arbitrarily without error.

In Section 1, we motivate the design of Caustic through a discussion of race conditions. In Section 2, we discuss the runtime and the various optimizations that allow it to efficiently and transactionally execute programs. In Section 3, we extend the functionality of the runtime in a robust, statically-typed Scala DSL called the standard library. In Section 4, we address the syntactic limitations of the standard library by introducing the Caustic programming language and its associated compiler.

1.1 Race Conditions

Race conditions occur whenever the order in which concurrent programs are executed affects their outcome. For example, suppose there exist two programs $A$ and $B$ that each increment a shared counter $x$. Formally, each program reads the current value of $x$ and then writes $x + 1$. If $B$ reads after $A$ writes, then $B$ reads $x + 1$ and writes $x + 2$. However, if $B$ reads before $A$ writes but after $A$ reads, then both $A$ and $B$ will read $x$ and write $x + 1$. 
This is an example of a race condition, because the value of $x$ after both $A$ and $B$ have completed depends on the order in which $A$ and $B$ are executed. Race conditions may seem relatively benign, but they can have catastrophic consequences in practical systems. Suppose the value of $x$ corresponded to your bank balance. What if your bank determined your balance differently depending on the order in which deposits are made?

Before introducing mechanisms to deal with race conditions, we must first define the necessary and sufficient criteria that any such mechanism must satisfy. Race conditions are a relatively well-studied problem in database literature, and we may draw upon known results from the field to define four properties of race-free programs. [8] We will hereafter refer to any system that satisfies these properties as transactional.

- **Atomic**: Programs never partially complete.
- **Consistent**: Programs see the effect of all completed programs.
- **Isolated**: Programs cannot see the effect of executing programs.
- **Durable**: The effects of completed programs are permanently visible.

### 1.1.1 Transactional Databases

We begin our discussion of transactional systems with the same databases that we used to formalize their definition. Over the past few decades, transactional databases have made incredible advances in performance and scalability. While they provide the necessary guarantees on which correct distributed systems can be built, their query languages are not suitable for general-purpose programming for the following reasons.

First, the absence of a standard query language tightly couples a database to the programs that are run on it. Relational databases, like MySQL and PostgreSQL, claim to implement the same SQL specification, but only support incompatible subsets of its functionality. Some NoSQL databases, like Cassandra and Aerospike, attempt to mimic SQL, but most implement their own bespoke interfaces. Stark differences between query languages makes it all but impossible to design general-purpose algorithms.
Second, these query languages are lacking in functionality and programmability. Query languages are usually designed to access and modify data, but not to perform intermediate computation. For this reason, fundamental abstractions, like objects and variables, that programmers have come to rely on are glaringly absent in most query languages. Christopher Date, one of the fathers of relational databases, criticized SQL for its ambiguous and often unintuitive syntax. [4] NoSQL databases are far worse, because they scale well only by shedding functionality.

1.1.2 Pessimistic Concurrency

Another way to deal with race conditions is pessimistic concurrency, or locking. Before a program accesses or modifies shared data, it first declares its intentions to other programs by acquiring a lock that it subsequently releases when it is finished with the data. Locks are trivially transactional because they require programs to have exclusive ownership before performing any operations on shared data. However, locking introduces new problems.

First, concurrent acquisition of multiple locks can cause a deadlock that prevents the system from making progress. For example, if program $A$ acquires lock $x$ and then attempts to acquire lock $y$ while program $B$ acquires $y$ and then attempts to acquire $x$, then neither $A$ nor $B$ can make progress because each is waiting for the other to release their lock. This problem is typically mitigated by imposing a total order on lock acquisition; for any two locks $x$ and $y$, $x$ will always be acquired before $y$ or $y$ will always be acquired before $x$. Programs must be carefully constructed so that all locks are acquired in the universal order.

Second, faulty programs may never release their locks. For example, if program $A$ acquires lock $x$ and subsequently fails, then no other program can ever acquire $x$. This problem is typically mitigated by using leases. [7] A lease is a lock that is automatically released after a certain amount of time. Programs must be carefully constructed so that they complete their operations on shared data within the lease duration or refresh the lease before it expires.

Third, locking is expensive. Locks must be acquired regardless of whether or not there actually are concurrent operations on shared data, because pro-
grams cannot know if there are or will be other programs that want or will want to simultaneously use the data. This significantly degrades performance in situations where contention between programs is low.

Fourth, locking cannot protect against programmer error. Programmers may omit or incorrectly use a lock and thereby introduce race conditions into their program. Locks cannot guarantee that they will be used correctly, and, therefore, cannot guarantee that race conditions will be exhaustively eliminated from a program.

1.1.3 Optimistic Concurrency

A final alternative is optimistic concurrency. Optimistic concurrency allows multiple programs to simultaneously access, but not modify, shared data without acquiring locks. Each program locally buffers any modifications that it makes and attempts to atomically apply them when it completes \textit{conditioned on the data that it accessed remaining unchanged}. If any data was modified, the program retries. This conditional update, known as a multi-word compare-and-swap, is known to be transactional and is widely used in a number of software transactional memory systems [18] including Sinfonia [2] and Caustic. We will hereafter refer to this operation as \texttt{cas} and its arguments as a \texttt{transaction}.

Optimistic concurrency requires a mechanism to detect that data has changed. The approach taken in Caustic and in similar systems is known as multi-version concurrency control. Data is uniquely identified by a \texttt{key} that is associated with a \texttt{revision}, or versioned value, whose version is incremented each time that its value is changed. We say that revisions $A$ and $B$ for a key conflict if the version of $A$ is less than $B$. Note that conflict is an asymmetric relation; if $A$ conflicts with $B$, then $B$ does not conflict with $A$.

Optimistic concurrency assumes that contention between programs will be low, because frequent retries can significantly degrade performance. It also assumes the existence of an underlying key-value store, hereafter referred to as a \texttt{volume}, that supports \texttt{get} and \texttt{cas} which non-transactionally accesses and transactionally modifies data in the manner described above. We will provide a distributed, fault-tolerant implementation of a volume in Beaker. For now, we will assume that such an implementation exists.
1.2 Runtime

A naïve implementation of optimistic concurrency would get each key whenever its value is needed by a program. This approach performs well when volumes are located in memory, but does not scale when they are on disk or across networks. Instead, a scalable implementation of optimistic concurrency would get each key whenever its value is needed or will be needed by a program. This is the approach taken in the runtime.

The runtime is a virtual machine that dynamically translates programs into transactions. A program is an abstract-syntax tree that is composed of literals and expressions. A literal is a scalar value of type flag, real, text, or null which correspond to bool, double, string, and null respectively in most C-style languages. An expression is a function that transforms literal arguments into a literal result. For example, Figure 1.1 is a program that increments a counter. Expressions may be chained together arbitrarily to form complex programs. Table 1.1 enumerates the various expressions that are natively supported by the runtime.
1.2.1 Execution

The runtime uses iterative partial evaluation to gradually reduce programs into a single literal result according to the following procedure.

1. **Fetch**: Get all keys that are read or written anywhere in the program that have not been fetched before, and add the returned revisions to a local **snapshot**.

2. **Evaluate**: Recursively replace all expressions with literal arguments with their corresponding literal result. For example,

   \[\text{add}(\text{real}(1), \text{sub}(\text{real}(0), \text{real}(2))) \rightarrow \text{real}(-1)\]

   The result of all writes is saved to a local **buffer** and the result of all reads is the latest value of the key in the local buffer or snapshot. This ensures that reads will see the effect of all previous writes within the program.

3. **Repeat**: Loop until the program is reduced to a single literal. Because all expressions with literal arguments return a literal result, all programs will eventually reduce to a single literal.

4. **Commit**: Cas all keys in the local buffer conditioned on all revisions in the local snapshot. The transactional guarantees of cas imply that program execution is **serializable**. Serializability means that concurrent execution has the same effect as some sequential execution, and, therefore, that program execution will be robust against race conditions.

1.2.2 Optimizations

First, execution is tail-recursive. This allows programs to be composed of arbitrarily many nested expressions without overflowing the stack frame during execution. It also allows the Scala compiler to aggressively optimize execution into a tight loop.

Second, the runtime batches I/O. Reads are performed simultaneously whenever possible and writes are buffered and simultaneously committed. By
batching I/O, the runtime performs a minimal number of operations on the database. This has significant performance implications, because I/O overhead is overwhelmingly the bottleneck by many orders of magnitude in most programs. [5]

Third, the runtime performs constant folding. Expressions with literal arguments are automatically simplified. This significantly reduces the size of programs and thereby improves the performance of the runtime.

1.2.3 Performance

In Figure 1.2, we examine execution latency and throughput. Benchmarks were conducted on a t2.2xlarge instance with 32GB of RAM and 8 CPUs. We first see that latency grows linearly with program size. In other words, each additional expression has a constant overhead. We then graph throughput under a variety of workloads. Each workload is composed of the specified number of threads simultaneously executing a series of programs that each read and write the specified percentages of the key-space uniformly at random. Note that the larger the percentage of writes and the greater the number of threads, the more likely that cas will fail and the more significant the reduction in throughput will be. We see that in read-only workloads, throughput scales linearly with the number of threads. As the percentage of writes and the number of threads increases, throughput falls considerably but still scales linearly with the number of threads.
1.3 Standard Library

The runtime provides native support for an extremely limited subset of the operations that programmers typically rely on to write applications. The standard library supplements the functionality of the runtime by exposing a robust Scala DSL complete with static types, records, collections, math, and control flow.

1.3.1 Typing

The runtime natively supports just four dynamic types: flag, real, text, and null. Dynamic versus static typing is a religious debate among programmers. Advocates of dynamic typing often mistakenly believe that type inference and coercive subtyping cannot be provided by a static type system. In fact, they can. Because static type systems are able to detect type inaccuracies at compile-time, they allow programmers to write more concise and correct code. [13] The standard library provides rich static types and features aggressive type inference and subtype polymorphism.

The standard library supports four primitive types. In ascending order of precedence, they are boolean, int, double, and string. A value represents a value of primitive type. There are two kinds of values. A constant is an immutable value, and a variable is a mutable value. Variables may be stored locally in memory or remotely in a volume.

```
1 // Creates an integer local variable named x.
2 val x = Variable.Local[Int]("x")
3 // Creates a floating point remote variable named y.
4 val y = Variable.Remote[Double]("y")
5 // Assigns y to the sum of x and y.
6 y := x + y
7 // Assigns x to the product of x and 4.
8 x *= 4
9 // Does not compile, because y is not an integer.
10 x := y
11 // Does compile, because floor(y) is an integer.
12 x := floor(y)
```
1.3.2 Records

In addition to these primitive types, the standard library also supports references to user-defined records. References use Shapeless to materialize compiler macros that permit the fields of a record to be statically manipulated and iterated. A current limitation is that records cannot be self-referential; a record cannot have a field of its own type.

```scala
1 // An example type declaration.
2 case class Bar(
3   a: String,
4 )
5
6 case class Foo(
7   b: Int,
8   c: Reference[Bar],
9   d: Bar
10)
11
12 // Constructs a remote reference to a Foo.
13 val x = Reference[Foo](Variable.Remote("x"))
14 // Returns the value of the field b.
15 x.get('b)
16 // Does not compile, because z is not a field of Foo.
17 x.get('z)
18 // Serializes x to a JSON string.
19 x.asJson
20 // Deletes all fields of x and all references.
21 x.delete(recursive = true)
22 // Constructs a local reference to a Foo.
23 val y = Reference[Foo](Variable.Local("y"))
24 // Copies x to y.
25 y := x
```

1.3.3 Collections

The runtime has no native support for collections of key-value pairs. The standard library provides implementations of three fundamental data structures: list, set, and map. These collections are mutable and statically-typed. Collections take care of the messy details of mapping structured data onto a flat namespace and feature prefetched iteration. A current limitation is that collections may only contain primitive types.

```scala
1 // Constructs a map from string to boolean.
2 val x = Map[String, Boolean](Variable.Remote("y"))
3 // Puts an entry in the map.
4 x += "foo" -> true
5 // Serializes x to a JSON string.
```
6 x.asJson
7 // Constructs a list of integers.
8 val x = List[Int](Variable.Local("x"))
9 // Increments each element in the list.
10 x foreach { case (i, v) => x.set(i, v + 1) }

1.3.4 Math

The runtime natively supports just nine mathematical operations: add, sub, mul, div, pow, log, floor, sin, and cos. However, these primitive operations are sufficient to derive the entire Scala math library using various mathematical identities and approximations. The div, log, sin, and cos functions can actually be implemented in terms of the other primitive operations; however, native support for them was included in the runtime to improve performance. The standard library provides implementations for all functions enumerated in Table 1.2.

1 // Returns the Taylor approximation of inverse sine.
2 asin(Pi / 2)
3 // Defines the sigmoid function.
4 def sigmoid(x: Value[Double]) = exp(x) / (exp(x) + 1)

1.3.5 Control Flow

The runtime natively supports control flow operations like branch, cons, and repeat. However, these operations are syntactically challenging to express. The standard library uses structural types to provide support for if, while, return, assert, and rollback.

1 // If statements.
2 If (x < 3) {
3   x **= 3
4 }
5
6 // If/Else statements
7 If (y < x) {
8   y **= 1
9 } Else {
10   x **= 1
11 }
12
13 // Ternary operator.
14 val z = If (y === x) { y - 1 } Else { y + 1 }

12
1.4 Compiler

The standard library provides additional functionality that is absent in the runtime to make it easier to construct programs. However, it does not address the syntactic challenges of expressing programs. At times the standard library is forced to use unintuitive operators like := and <> and verbose declarations like `Variable.Remote("x")`, because of the limitations of implementing a language within a language.

The compiler, Causticc, translates code written in the Caustic programming language into operations on the standard library. Caustic features aggressive type inference, static typing, and a concise grammar.

1.4.1 Implementation

We use ANTLR [16] to generate a predicated LL(*) parser from an ANTLR grammar and then walk the resulting parse-tree to generate code. Most statements in Caustic have direct equivalents in the standard library. For the most part, the compiler acts as a kind of intelligent find-and-replace. However, there are certain aspects of the compiler that are non-trivial.

First, the compiler performs type inference. It is able statically verify types and method signatures by maintaining a lexically-scoped universe of the various variables, records, and functions that have been defined. Because the type system is relatively simplistic, static types almost never need to be specified. The combination of a static type system and aggressive type inference allows programs to be both type-safe and concise.

Second, the compiler is directly integrated Pants. Pants is an open-source, cross-language build system. Integration into Pants means that Caustic programs are interoperable with a variety of existing languages and tooling.
Third, the compiler provides a TextMate bundle that implements syntax highlighting and code completion for most text editors and IDEs to make it easier for programmers to use the language.

### 1.4.2 Programmability

Programmability is impossible to measure, because there is no metric by which two languages can be objectively compared. We will present implementations of various algorithms in Caustic and in other languages, but, instead of comparing them with an imprecise measure of programmability like lines of code or cyclomatic complexity, we will reserve all judgement and leave it to the reader to verify for themselves that Caustic is indeed easier to use.

Consider the following example of a distributed counter written in Caustic. A similar implementation of a distributed, backend-agnostic counter is provided by Akka.

```plaintext
module caustic.example

/** *
 * A distributed counter.
 */

service Counter {

  /** *
   * Increments the total and returns its current value.
   *
   * @param x Reference to total.
   *
   * @return Current value.
   */
  def increment(x: Int&): Int = {
    if (x != null) x += 1 else x = 1
    x
  }
}
```

Consider the following example of a distributed file system in Caustic.

```plaintext
module caustic.example

/** *
 * A file.
 */

struct File {

  /** *
   * @param contents File contents.
   */
  struct contents File {
```

14
*/
}

module caustic.example

/**
 * A distributed message queue.
 */

service Queue {

/**
 * Adds the message to the end of the queue.
 */

/**
 * Deletes the specified path.
 */

/**
 * Updates the contents of the specified path.
 */

/**
 * Returns the contents of the specified path.
 */

/**
 * Returns whether or not the file exists.
 */

/**
 * Returns whether or not the file exists.
 */

}
* @param queue Queue.
* @param message Message.
*/
def push(queue: List[String]&, message: String): Unit = queue.set(queue.size, message)

/**
 * Returns the message at the front of the queue.
 *
 * @param queue Queue.
 * @return Head.
 */
def peek(queue: List[String]&): String = queue.get(0)

/**
 * Removes and returns the message at the front of the queue.
 *
 * @param queue Queue.
 * @return Head.
 */
def pop(queue: List[String]&): String = {
    var head = peek(queue)
    queue.set(0, null)
    head
}

/**
 * Returns the number of messages in the queue.
 *
 * @param queue Queue.
 * @return Length.
 */
def size(queue: List[String]&): Int = queue.size

Consider the following example of a distributed lock service written in Causitic. A similar implementation of a distributed read-write lock is provided by Curator and Sherlock.

module caustic.example

/**
 * A read-write lock.
 *
 * @param readers Number of readers.
 * @param writers Number of writers.
 */
struct Lock {
    readers: Int,
    writers: Int
}

/**
 * An access permit.
 */
struct Permit {
    lock: Lock &,
    forRead: Boolean,
    forWrite: Boolean
}

service LockService {
    /**
     * A distributed lock service.
     */
    def exclusive(lock: Lock&): Permit = {
        if (lock.writers > 0 || lock.readers > 0) {
            Permit(lock, false, false)
        } else {
            lock.writers += 1
            Permit(lock, false, true)
        }
    }
    /**
     * Attempts to acquire exclusive access to the lock.
     *
     * @param lock Lock.
     * @return Read-write permit.
     */
    def shared(lock: Lock&): Permit = {
        if (lock.writers > 0) {
            Permit(lock, false, false)
        } else {
            lock.readers += 1
            Permit(lock, true, false)
        }
    }
    /**
     * Revoke the permit’s access.
     *
     * @param permit Permit.
     */
    def release(permit: Permit): Unit = {
        if (permit.forRead) {
            permit.lock.readers -= 1
        } else {
            permit.lock.writers -= 1
        }
    }
}
1.5 Conclusion

Concurrency is difficult, but it does not need to be. Caustic allows programmers to build concurrent systems as if they were they were not. It provides a highly programmable alternative to traditional approaches for dealing with race conditions. We have shown that it is possible to build a robust, transactional programming language over any volume that supports just two operations. In Beaker, we will see that these operations can be efficiently implemented.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(x, y)</td>
<td>Sum of x and y.</td>
</tr>
<tr>
<td>both(x, y)</td>
<td>Bitwise AND of x and y.</td>
</tr>
<tr>
<td>branch(c, p, f)</td>
<td>Executes p if c is true, or f otherwise.</td>
</tr>
<tr>
<td>cons(a, b)</td>
<td>Executes a and then b.</td>
</tr>
<tr>
<td>contains(x, y)</td>
<td>Returns whether or not x contains y.</td>
</tr>
<tr>
<td>cos(x)</td>
<td>Cosine of x.</td>
</tr>
<tr>
<td>div(x, y)</td>
<td>Quotient of x and y.</td>
</tr>
<tr>
<td>either(x, y)</td>
<td>Bitwise OR of x and y.</td>
</tr>
<tr>
<td>equal(x, y)</td>
<td>Returns whether x and y are equal.</td>
</tr>
<tr>
<td>floor(x)</td>
<td>Floor of x.</td>
</tr>
<tr>
<td>indexOf(x, y)</td>
<td>Returns the index of the first occurrence of y in x.</td>
</tr>
<tr>
<td>length(x)</td>
<td>Returns the number of characters in x.</td>
</tr>
<tr>
<td>less(x, y)</td>
<td>Returns whether x is strictly less than y.</td>
</tr>
<tr>
<td>load(n)</td>
<td>Loads the value of the variable n.</td>
</tr>
<tr>
<td>log(x)</td>
<td>Natural log of x.</td>
</tr>
<tr>
<td>matches(x, y)</td>
<td>Returns whether or not x matches the regex pattern y.</td>
</tr>
<tr>
<td>mod(x, y)</td>
<td>Remainder of x divided by y.</td>
</tr>
<tr>
<td>mul(x, y)</td>
<td>Product of x and y.</td>
</tr>
<tr>
<td>negate(x)</td>
<td>Bitwise negation of x.</td>
</tr>
<tr>
<td>pow(x, y)</td>
<td>Returns x raised to the power y.</td>
</tr>
<tr>
<td>prefetch(k, s)</td>
<td>Reads keys at k/i for i in [0, s).</td>
</tr>
<tr>
<td>read(k)</td>
<td>Reads the value of the key k.</td>
</tr>
<tr>
<td>repeat(c, b)</td>
<td>Repeatedly executes b while c is true.</td>
</tr>
<tr>
<td>rollback(r)</td>
<td>Discards all writes and returns r.</td>
</tr>
<tr>
<td>sin(x)</td>
<td>Sine of x.</td>
</tr>
<tr>
<td>slice(x, l, h)</td>
<td>Returns the substring of x between [l, h).</td>
</tr>
<tr>
<td>store(n, v)</td>
<td>Stores the value v for the variable n.</td>
</tr>
<tr>
<td>sub(x, y)</td>
<td>Difference of x and y.</td>
</tr>
<tr>
<td>write(k, v)</td>
<td>Writes the value v for the key k.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>abs(x)</td>
<td>Absolute value of x.</td>
</tr>
<tr>
<td>acos(x)</td>
<td>Cosine inverse of x.</td>
</tr>
<tr>
<td>acot(x)</td>
<td>Cotangent inverse of x.</td>
</tr>
<tr>
<td>acsc(x)</td>
<td>Consecant inverse of x.</td>
</tr>
<tr>
<td>asec(x)</td>
<td>Secant inverse of x.</td>
</tr>
<tr>
<td>asin(x)</td>
<td>Sine inverse of x.</td>
</tr>
<tr>
<td>atan(x)</td>
<td>Tangent inverse of x.</td>
</tr>
<tr>
<td>cbrt(x)</td>
<td>Cube root of x.</td>
</tr>
<tr>
<td>ceil(x)</td>
<td>Smallest integer greater than or equal to x.</td>
</tr>
<tr>
<td>cos(x)</td>
<td>Cosine of x.</td>
</tr>
<tr>
<td>cosh(x)</td>
<td>Hyperbolic cosine of x.</td>
</tr>
<tr>
<td>cot(x)</td>
<td>Cotangent of x.</td>
</tr>
<tr>
<td>coth(x)</td>
<td>Hyperbolic cotangent of x.</td>
</tr>
<tr>
<td>csc(x)</td>
<td>Cosecant of x.</td>
</tr>
<tr>
<td>csch(x)</td>
<td>Hyperbolic cosecant of x.</td>
</tr>
<tr>
<td>exp(x)</td>
<td>Exponential of x.</td>
</tr>
<tr>
<td>expm1(x)</td>
<td>Equivalent to $e^x - 1$.</td>
</tr>
<tr>
<td>floor(x)</td>
<td>Largest integer less than or equal to x.</td>
</tr>
<tr>
<td>hypot(x, y)</td>
<td>Hypotenuse of the triangle with base x and height y.</td>
</tr>
<tr>
<td>log(x)</td>
<td>Natural logarithm of x.</td>
</tr>
<tr>
<td>log10(x)</td>
<td>Log base 10 of x.</td>
</tr>
<tr>
<td>log1p(x)</td>
<td>Equivalent to $\log(x + 1)$.</td>
</tr>
<tr>
<td>pow(x, y)</td>
<td>Power of x to the y.</td>
</tr>
<tr>
<td>random()</td>
<td>Uniformly random number on $[0, 1)$.</td>
</tr>
<tr>
<td>rint(x)</td>
<td>Closest integer to x, rounding to the nearest even number.</td>
</tr>
<tr>
<td>round(x)</td>
<td>Closest integer to x, rounding up.</td>
</tr>
<tr>
<td>round(x, y)</td>
<td>Closest multiple of y to x.</td>
</tr>
<tr>
<td>sec(x)</td>
<td>Secant of x.</td>
</tr>
<tr>
<td>sech(x)</td>
<td>Hyperbolic secant of x.</td>
</tr>
<tr>
<td>signum(x)</td>
<td>Sign of x.</td>
</tr>
<tr>
<td>sin(x)</td>
<td>Sine of x.</td>
</tr>
<tr>
<td>sinh(x)</td>
<td>Hyperbolic sine of x.</td>
</tr>
<tr>
<td>sqrt(x)</td>
<td>Square root of x.</td>
</tr>
<tr>
<td>tan(x)</td>
<td>Tangent of x.</td>
</tr>
<tr>
<td>tanh(x)</td>
<td>Hyperbolic tangent of x.</td>
</tr>
</tbody>
</table>
Chapter 2

Beaker: A Distributed, Transactional Database

Beaker is a distributed, transactional database that serves as the underlying storage system for Caustic. Beaker is fault-tolerant, configurable, and performant.

In Section 1, we motivate the design of Beaker through a discussion of consensus. In Section 2, we discuss the algorithm that Beaker uses to commit distributed transactions and provide a rough proof of correctness and a description of its implementation. In Section 3, we evaluate the performance of Beaker under a variety of workloads.

2.1 Consensus

The only certainty in distributed systems is that machines will fail. The key challenge in building fault-tolerant distributed systems is constructing reliable systems from unreliable components. [15] Distributed databases replicate data across a cluster of machines. By storing copies in different places, distributed databases are tolerant of individual machine failures.

Replication solves the problem of fault-tolerance, but creates another; the various replicas must be kept consistent with each other. A naïve approach is to require all replicas to first agree to apply any modifications made to
the database. This would ensure that all replicas remained identical. However, this approach is not fault-tolerant. If any replica were to fail, then no modifications could ever be made to the database until it recovered. Instead, fault-tolerant distributed databases require only that a quorum, or simple majority, of replicas to reach agreement before modifications can be safely applied. This ensures that any quorum of replicas will contain at least one that has the latest copy of the database while remaining tolerant of a minority of individual machine failures. Reaching quorum agreement in a distributed system is known as consensus, and it is a relatively well-studied problem. In this section, we explore different consensus algorithms culminating with the approach taken in Beaker.

2.1.1 Terminology

We define a proposal as an abstract operation on a group of replicas and a proposer as the replica that initiates consensus on a proposal. Over the course of this discussion, we will gradually refine this definition of a proposal until we arrive at the one used in Beaker.

Replicas communicate by sending messages. We assume that delivered messages cannot be reordered; if message A was delivered before message B, then A was sent before B. In practice, this assumption is satisfied by most networking protocols including TCP.

2.1.2 Two-Phase Commit

In Two-Phase Commit, the proposer prepares a proposal by first acquiring locks on a quorum of replicas. If it successfully acquires all locks, the proposer informs all replicas to learn the proposal. When a proposal is learned, its operation is applied and all locks are subsequently released.

Two-Phase Commit is not fault-tolerant. If the proposer were to fail after it successfully prepared a proposal but before it requested that it be learned, the locks that it acquired would never be released and no new proposals could ever be learned. We will see in Paxos how we can modify the protocol to guarantee fault-tolerance.
2.1.3 Paxos

Paxos makes two modifications to Two-Phase Commit to address its fault-intolerance. First, it associates each proposal with a monotonically-increasing, globally-unique ballot number. Proposals are totally-ordered and uniquely-identified by their ballot. Each replica keeps track of the latest ballot that it has seen. Second, it introduces an intermediate accept phase to the protocol. We will see that this additional phase allows the system to recover from proposer failure. [12]

In Paxos, the proposer prepares a proposal by assigning it a ballot greater than any it has seen and sending its ballot to a quorum of replicas. If the ballot is greater than any it has seen, a replica promises not to accept any proposal less than it and returns any proposal that it has already accepted. Otherwise, the replica ignores the request. Intuitively, ballots function as a kind of implicit lock that the proposer holds until another proposer prepares a greater ballot. If a majority of replicas do not respond to its prepare request, the proposer retries with a greater ballot. Otherwise, the proposer selects a proposal to be accepted. If any replica returned an accepted proposal, then the proposer must select the latest accepted proposal and set its ballot to the one that it prepared. Otherwise, the proposer selects its own proposal. Intuitively, this allows the system to recover when a proposer fails after convinced a majority to accept its proposal but before it could be learned. A replica accepts a proposal if and only if it has not promised not to. When a replica accepts a proposal, it requests that all replicas learn it. A replica learns a proposal when a majority of replicas have requested that it be learned. When a replica is learned, its operation is applied and any accepted proposals are removed. Intuitively, this releases the implicit lock held by the proposer and consensus to begin on another proposal.

Paxos guarantees that all non-faulty replicas will learn proposals in the same order. Often, this guarantee is unnecessary because a large number of operations on a distributed system are commutative, so they may be performed in any order. For example, reads and writes to different keys in a database may be performed in any order without compromising consistency. We will see in Generalized Paxos that we can exploit commutativity to improve performance.

It is known that no deterministic fault-tolerant consensus protocol can guar-
antee progress in an asynchronous network. [6] Paxos is no exception. If a higher ballot is continuously prepared before any proposal can be accepted, no proposal will ever be learned. Implementations of Paxos typically elect a distinguished replica, called a leader, to which all other replicas forward their proposals to guarantee liveness. Whenever leaders fail, replicas run an instance of Paxos to acquire leadership of the cluster. The reliance on the existence of a single, stable leader is both an important simplifying assumption and a performance limitation. If there exists a leader, then prepare messages are superfluous. Intuitively, the leader implicitly holds a lock on all replicas because no other replica can initiate proposals. This allows proposals to be learned in just two message delays. However, the reliance on the leader to initiate all proposals is also a significant bottleneck at scale. The entire system moves at the rate of the leader. In fact, this is the fundamental limitation in implementations of Paxos like ZooKeeper [9] and Chubby [3]. We will see in Egalitarian Paxos that we can remove the dependence on a leader to improve performance.

2.1.4 Generalized Paxos

Generalized Paxos [11] addresses the scalability of Paxos by exploiting commutativity. An operation $A$ commutes with $B$ if performing $A$ after $B$ has the same effect as performing $B$ after $A$. For example, addition is commutative but division is not; $4 + 3 = 3 + 4$ but $\frac{4}{3} \neq \frac{3}{4}$. In fact, most operations on a distributed database are commutative. Reads commute with each other and reads and writes to different keys commute.

Generalized Paxos associates each proposal with a sequence of operations. We say that proposal $A$ is equivalent to proposal $B$ if all non-commutative operations in $A$ and $B$ are in the same order. All equivalent proposals have the same effect. Generalized Paxos permits replicas to learn different proposals as long as they are equivalent.

In Generalized Paxos, proposers do not forward their requests to leader. Instead, they immediately request that all replicas accept their proposed operation. A replica appends the operation to their currently accepted proposal and requests that all replicas learn it. A proposal is learned when a majority of replicas have requested that it or an equivalent proposal be learned. If no majority of replicas can agree on the ordering of non-commutative opera-
tions, it is the responsibility of the leader to select one and to run an instance of Paxos to convince the other replicas to accept its choice before resuming normal operation.

Like Paxos, Generalized Paxos relies on the existence of a single, stable leader to mediate ordering disagreements between replicas and guarantees that all commutative operations will be learned in two message delays. Unlike Paxos, it does not require all proposals to originate from the leader. If most operations are commutative, the leader will rarely be required to arbitrate. However, the existence of a leader can still be a scalability bottleneck. We will see in Egalitarian Paxos that we can remove the dependence on a leader to improve the performance of the system.

2.1.5 Egalitarian Paxos

Egalitarian Paxos [14] makes a subtle modification to Generalized Paxos to remove its dependence on a leader. Egalitarian Paxos associates with proposal with a directed acyclic graph of operations in which each edge corresponds to a dependency between two operations. The benefit of using a directed acyclic graph is that its various strongly connected components can be performed in parallel without impacting consistency. This has huge ramifications for performance, particularly in databases because reads and writes are relatively expensive operations.

In Egalitarian Paxos, an operation depends on all accepted proposals for which it does not commute. The proposer builds a dependency graph for a proposal from any proposals that it has already accepted and requests that all replicas accept it. A replica supplements the dependency graph of the proposal with any proposals that it has accepted and requests that the result be learned. If no majority of replicas can agree on the dependency graph of a proposal, it is the responsibility of the proposer to select one and to run an instance of Paxos to convince the other replicas to accept its choice before resuming normal operation.

Egalitarian Paxos implicitly assumes that operations are idempotent, because an operation may be in the dependency graph of multiple proposals. An operation $A$ is idempotent if repeated non-sequential applications of $A$ have the same effect as a single application of $A$. For example, multiplication by
one is idempotent but by two is not; \(4 \times 1 = 4 \times 1 \times 1\) but \(4 \times 2 \neq 4 \times 2 \times 2\).

This assumption will become important when we use Egalitarian Paxos to implement distributed transactions.

## 2.2 Distributed Transactions

We now use Egalitarian Paxos to implement distributed transactions. We begin by concretely defining the underlying system. We then describe the consensus protocol and conclude with a rough proof of correctness.

### 2.2.1 Terminology

A **database** is a key-value store that supports two operations: *read* and *write*. Databases guarantee durability; if a key is written, then subsequent reads will always return the updated value. Like Caustic, we will associate each key with a **revision**, or versioned value, with the same conflict relationship as before. Reads and writes on the database return or update the revisions of a set of keys. Operations on a database are non-transactional and can fail arbitrarily.

We define our abstract operation as a **transaction**. A transaction is composed of a set of dependent versions, called its *readset*, and a set of updates, called its *writeset*. We say that two transactions **conflict** if there exists a key in the readset or writeset of either transaction that is in the writeset of the other. A transaction may be **committed** on a database by applying the updates in its writeset if and only if it depends on the latest version of every key in its readset. We say that a transaction is **committable** if it can be committed. In order to guarantee idempotency of commit, any key in the writeset of a transaction must also be present in its readset.

A **cache** is a write-through database. Because dependencies are validated on commit, keys may be speculatively read from cache without sacrificing consistency. Beaker supports multi-level cache hierarchies over the underlying database. Revisions may be **fetched** or **updated** in cache. Cache coherency is maintained by updating the cache whenever transactions are committed on the underlying database.
An **archive** is a transactional database. Archives use an **executor** to linearize conflicting transactions on the underlying database. An executor schedules transactions in groups such that conflicting transactions are never simultaneously performed. This guarantees consistency and isolation; transactions see the effect of any previously committed conflicting transaction and conflicting transactions are never committed simultaneously.

We define our replica as a **beaker**. Beakers use a variation of Egalitarian Paxos to coordinate distributed transactions with several desirable properties. First, non-conflicting transactions may be simultaneously committed. Second, databases with stale revisions are automatically repaired. Third, transactions may be committed as long as at least a majority of beakers are operational.

### 2.2.2 Protocol

Beaker makes two key modifications to Egalitarian Paxos. First, it associates each proposal with a set of non-conflicting transactions, called its **commits**, and a set of revisions, called its **revisions**. We say that a proposal $A$ **conflicts** with a proposal $B$ if they have any conflicting commits. We say that a proposal $A$ is **equivalent** to proposal $B$ if their commits are the same. A proposal $A$ may be **merged** with a proposal $B$ by taking the maximum of their ballots, combining their commits choosing the transaction in the newer proposal in the case of conflicts, and combining their repairs choosing the highest revision in the case of duplicates. Second, it introduces an intermediate **get** phase to the protocol. We will see that this additional phase allows the system to repair databases with stale revisions.

In Beaker, the proposer prepares a proposal by sending it to all beakers. If a beaker has not made a promise to a newer proposal, it responds with a promise not to accept any proposal that conflicts with the proposal it returns. If it has already accepted any conflicting proposals, it merges them together and returns the result. Otherwise, it returns the prepared proposal with a zero ballot. If the proposer does not receive a majority of promises, it retries with a higher ballot. Otherwise, it merges the returned promises together into a single proposal. If the merged proposal is not equivalent to its prepared proposal, the proposer retries with a higher ballot. Otherwise, the proposer gets the revisions of all keys that are read by the proposal from a majority
of beakers. The proposer discards all transactions that cannot be committed
given the latest returned revisions and sets the repairs of the proposal to the
latest revisions of keys that are read - but not written - by the proposal for
which any two beakers have different revisions. The proposer then requests
all beakers to accept the proposal. A beaker accepts a proposal if it has not
promised not to. If a beaker accepts a proposal, it requests that all other
beakers learn the proposal. A beaker learns a proposal when a majority of
beakers have requested that it be learned. When a beaker learns a proposal,
it commits its transactions and repairs on its archive.

2.2.3 Correctness

We will make the assumption of connectivity; all learn messages are always
delivered. This ensures that the proposer will always learn that its proposal
was learned. This is a necessary requirement because it allows clients to
know with certainty whether or not their transaction was committed. In
practice, this assumption can be weakened by adding an additional decide
phase to the protocol. Whenever a proposal is learned by a beaker, it informs
all other beakers that the proposal was decided. As long as at least one
beaker receives a majority of learn messages and at least one decide message
is delivered to each beaker, any proposer is guaranteed to learn that its
proposal was learned. A proposal is learned by a beaker when it receives
either a majority of learn requests or a decide message. The decide phase
introduces additional partition tolerance and comes at no additional cost
if the network is perfectly reliable. It does, however, increase the number
of messages sent between beakers which could saturate the network. The
assumption of connectivity is reasonable if beakers are co-located in the same
data center, but is unreasonable if they are geographically separated.

We will also make the assumption of reliability; at least a majority of repli-
cas will atomically apply each learned proposal. This ensures that any ma-
ajority of replicas will contain at least one that has the latest revision of each
key-value pair. Our assumption of reliability is substantially weaker than
the one implicitly made by the previously presented Paxos algorithms; they
assume that a majority of replicas atomically apply every learned proposal.
We will see that consistency is guaranteed even if no replica has applied every
learned proposal.
We will also make use of the fact of quorum intersection; any two majorities of beakers must have at least one in common. Verification of this property is trivial and is left to the reader.

**Theorem 1** (Liveness). Any proposal $A$ that is accepted by a majority will eventually be learned by everyone.

*Proof.* By quorum intersection, at least one promise will contain $A$ until $A$ has been learned by sufficiently many beakers that it no longer has been accepted by a majority. But if $A$ was learned by any beaker, at least a majority of beakers must have requested that it be learned. By connectivity, if a majority of beakers requested that $A$ be learned then all beakers will learn $A$. Therefore, $A$ will eventually be learned by everyone. □

Note that we guarantee only that a proposal will eventually be learned once it has been accepted by a majority, but we do not guarantee that a proposal will ever accepted by a majority. It is possible for proposers to continuously prepare higher ballot proposals before any can be accepted by a majority. In practice, by retrying with exponentially jittered backoffs the likelihood of such an event asymptotically approaches zero.

**Theorem 2** (Serializability). If proposal $A$ is accepted by a majority, then any conflicting proposal $B$ that is subsequently prepared will always be learned after $A$.

*Proof.* Because $A$ was accepted by a majority before $B$, the majority that accepted $A$ before $B$ will request that $A$ be learned before $B$. Because messages are always delivered in order and by connectivity, $B$ will always be learned after $A$. □

**Theorem 3** (Commutativity of Repairs). Let $R$ denote the repairs for a proposal $A$ that has been accepted by a majority. Any proposal $B$ that is accepted by majority and conflicts with $A + R$ but not $A$ commutes with $A + R$.

*Proof.* Because $B$ conflicts with $A + R$ but not $A$, $B$ must read a key $k$ that is read by $A$. By reliability, $B$ must read the latest version of $k$. Suppose that $B$ is committed first. Because $B$ reads and does not write $k$, $A + R$ can
still be committed. Suppose that \( A + R \) is committed first. Because \( A + R \) writes the latest version of \( k \) and \( B \) reads the latest version, \( B \) can still be committed. Therefore, \( B \) commutes with \( A + R \).

**Theorem 4 (Consistency).** If a proposal \( A \) is accepted by a majority, its transactions are committable.

**Proof.** Suppose there exists a transaction that cannot be committed. Then, the transaction must read a key for which there exists a newer version. This implies that there exists a proposal \( B \) that was accepted by a majority after, but learned before \( A \) that changes a key \( k \) that is read by \( A \). By serializability, \( B \) cannot conflict with \( A \). Therefore, \( B \) must repair \( k \). By commutativity of repairs, \( A \) may still be committed.

### 2.2.4 Reconfiguration

In practical systems, beakers may join or leave the cluster arbitrarily as the cluster grows or shrinks in size. In this section, we describe how beakers are bootstrapped when they join an existing cluster. We say that a beaker is **fresh** when it initially joins a cluster. In order to guarantee correctness, fresh beakers must be immediately populated with the latest revision of every key-value pair. Otherwise, if \( N + 1 \) fresh beakers join a cluster of size \( N \) it is possible for a majority of beakers to consist entirely of fresh beakers. This violates the assumption of reliability.

A naive solution might be for the fresh beaker to propose a read-only transaction that depends on the initial revision of every key-value pair in the database and conflicts with every other proposal. Then, the fresh beaker would automatically repair itself in the process of committing this transaction. However, this is infeasible in practical systems because databases may contain arbitrarily many key-value pairs. This approach would inevitably saturate the network. Furthermore, it prevents any proposals from being accepted in the interim.

We can improve this solution by decoupling bootstrapping and consensus. A fresh beaker joins the cluster as a partial member, called a **learner**, that learns proposals, but is not involved in consensus. The fresh beaker **scans** a majority of full members, called **acceptors**, and repairs its own archive.
with the latest returned values. By assumption of reliability, it is guaranteed to have the latest value of every key. It then joins the cluster as an acceptor. This approach consumes less bandwidth and permits concurrent proposals.

Each beaker maintains its own view of the cluster configuration which it sends alongside any proposal that it proposes. If the view is outdated, beakers inform the proposer of the new configuration. If a beaker has already accepted a proposal from an outdated view, that proposal must be completed before the cluster can move to the new configuration. This ensures that reconfiguration is consistent across the cluster.

### 2.3 Evaluation

#### 2.3.1 YCSB

The [Yahoo Cloud Serving Benchmark](https://cs serv.berkeley.edu/ycsb/) (YCSB) is a common benchmark used to evaluate databases under various workloads. Each workload is composed of a different ratio of reads, inserts, and updates to benchmark performance under various kinds of load. Workload A is composed 50% reads and 50% updates, B of 95% reads and 5% updates, C of 100% reads, D of 95% reads and 5% inserts, and F of 50% reads and 50% read-modify-writes.

In Figure 2.1, we examine how throughput changes as the number of instances in the cluster increases. We see that throughput scales well in read-dominant workloads and falls considerably in write-dominant workloads. In Figure 2.2, we examine the 95% read latency as the number of instances in the cluster increases. We see that read latency remains relatively constant across all workloads, even as the number of instances increases. In Figure 2.3, we examine the 95% read latency as the number of instances in the cluster increases. We see that write latency increases as the number of instances increases, because more instances take longer to reach consensus.
2.3.2 Contention

In this section, we examine the performance of Beaker under contention both from a theoretical and a practical perspective. We begin with a mathematical treatment of consensus and conclude with an empirical verification of our analysis.

We may model the number of retries required to successfully commit a transactions as a negative binomial distribution. A negative binomial distribution $NB(p, r)$ measures the number of failed events each occurring with probability $p$ before $r$ successes are encountered. In this case, $p$ represents the contention probability, or the likelihood that any two concurrent transactions conflict. Therefore, the distribution of total attempts required to successfully commit any transaction as $A \sim 1 + NB(p, 1)$. We may then use known results about the negative binomial distribution to make predictions about the how the system will behave under contention.

We must first make concrete our definition of contention probability. Consider two sets $X$ and $Y$ of size $l$ selected uniformly at random from the set $[0, n)$. The contention probability is the likelihood that $X$ and $Y$ are not disjoint. Intuitively, the probability that $X$ and $Y$ are disjoint is equal to the likelihood that $Y$ is drawn from the set $[0, n) \setminus X$. Therefore, the probability
that $X$ and $Y$ are not disjoint is equal to $1 - \frac{\binom{n-1}{l}}{\binom{n}{l}}$ where $C(a, b)$ is the number of ways to choose $b$ items from a set of size $a$. We may extend this notion of uniformly random sets to uniformly random transactions. Suppose two transactions $T_1$ and $T_2$ each write $l$ keys chosen uniformly at random from a key space of size $n$. The probability that they will conflict is exactly equal to the probability that $X$ and $Y$ are not disjoint.

To verify that this theoretical definition of contention probability matches empirical results I conducted a Monte Carlo simulation in which two threads repeatedly generate and commit $M$ transactions that increment a uniformly random set of $l$ keys and monitor the total number of failures. If the hypothesis about the negative binomial distribution is correct, transactions should conflict on average $p \times M$ times. I found this to be empirically validated, but the reader is encouraged to verify this claim for themselves.

We assume for the sake of simplicity that there exist exactly two proposers that are simultaneously committing transactions. In practice, most systems will have many more active proposers. Unfortunately, I lack the mathematical ability to analytically derive contention probability in the general case. However, the contention probability may be numerically approximated by changing the number of threads in the Monte Carlo simulation.

We may use our empirically verified model of contention to make predictions
about the performance degradation of the system under varying degrees of contention. Given a key-space of size $n$ and an average transaction of size $l$, we would expect each transaction to require $E[A] = 1 + \frac{p}{1-p}$ attempts and for the average throughput of the system to decrease by the same amount.

2.4 Conclusion

We have presented the design and implementation of Beaker and shown that it is possible to build robust programming language from the minimal interface it provides.
Bibliography


