Caustic

Ashwin Madavan

5 September 2017

1 Introduction

Caustic provides **relational semantics for arbitrary transactional key-value stores**. It consists of two main components: the runtime which executes transactions and the syntax which defines a high-level programming language for expressing transactions.

2 Runtime

The Caustic runtime is responsible for executing transactions on arbitrary databases. Transactions are represented as abstract-syntax trees, because they are easy to manipulate, serialize, and optimize. The runtime exposes a Thrift interface for cross-language transaction execution, and is capable of running on any database that supports the following primitive operations:

1. **get**: Returns the values of a set of keys.
2. **cput**: Conditionally apply changes iff dependent keys remain unchanged.

2.1 Optimizations

Because transactions are represented as abstract-syntax trees, the runtime is able to perform some interesting optimizations to improve execution performance. Most importantly, the runtime performs read and write batching. In other words, the runtime fetches *as many reads as possible* and applies *all writes together*. Clearly, the runtime performs optimally few roundtrips to and from the underlying key-value store, which is typically the performance bottleneck in database applications.

2.2 Tunable Consistency

Transactions are executed using MVCC, and derive their consistency and isolation guarantees from the underlying key-value store. If the underlying key-value has a repeatable isolation level, for example, then so will transactions in Caustic. If the underlying key-value is not transactional, then neither are transactions in Caustic. Therefore, applications may substitute a key-value store with weaker transactional guarantees (or no transactions at all!) to improve performance.

2.3 Benchmarks

Compare the Caustic runtime (running on RocksDB) to comparable systems like Calvin and Tango. Rely on quantitative metrics like end-to-end latency and transaction throughput on standard database benchmarks like YCSB and TPC-C.

2.3.1 Effect of Transaction Size

Performance scales linearly with transaction size. This is enabled by the tail recursive design of the runtime and can be verified empirically from benchmarks.

2.3.2 Effect of Read or Write Skew

Transactions with more reads than writes will have a different performance profiles than transactions with more writes than reads. However, this difference in performance is an artifact of the choice of underlying key-value store and not due to a read or write bias in the runtime. In other words, a read optimized key-value store will fare better with read-heavy transactions and a write optimized key-value store with write-heavy transactions in Caustic.

2.3.3 Effect of Contention

What happens to performance under contention? The number of retries required to execute a transaction may be modeled as a negative binomial distribution in which the probability of success p is the contention probability. Therefore, $A = 1 + NB(1, p)$ is the distribution of attempts. Known results about the negative binomial distribution, may be used to make predictions about the mean and variance of transaction execution latency under contention. For example, the mean number of attempts $\bar{A} = 1 + \frac{p}{1-p} = \frac{1}{1-p}$. The contention probability is determined by a number of factors including: the number of keys that are read and written, the number of database reads and writes, and the latency of database reads and write.

3 Syntax

The Caustic syntax is a new high-level programming language that compiles into transactions on the Caustic runtime. Regardless of the features they might have (objects, functions, types, etc.), all programs eventually compile down into load, stores, and operations on memory - and memory is a key-value store. The purpose of Caustic syntax is to provide a rich language for expressing *transactional* loads, stores, and operations on *key-value stores*.

3.1 Concurrency

Because all programs written in Caustic compile into transactions, all programs are automatically thread-safe. No fancy synchronization mechanisms are required to build parallelizable programs in Caustic, everything is concurrent by default.

3.2 Separation of Code and Data

Every database has its own language for writing stored procedures. Even SQL databases typically support very different subsets of the SQL specification, that are not interoperable for anything but the most trivial of programs. Because the syntax varies so wildly between different databases, stored procedures are very tightly coupled with the choice of underlying database. This makes it tremendously expensive to change databases once development is under way, and impossible to test stored procedures in isolation. Caustic provides a uniform interface over arbitrary databases. In other words, Caustic programs written for ione database will be identical to the same program written for any another. This allows programmers to select whichever database is most appropriate for their use-case (eg. in-memory for tests, MySQL for production).

3.3 Interoperability

Because of the simplicity of the Caustic grammar and runtime, the compiler is able to generate executable binaries that serve Caustic programs over a variety of mediums over any underlying database. Most OLTP applications can be divided into three parts: a database connector (eg. SQLAlchemy), a set of stored procedures (eg. Flask), and an external interface (eg. REST, Thrift). By specifying their stored procedures in Caustic, programmers can now rely on the Caustic compiler to generate the first and third parts for them.

3.4 Benchmarks

Compare the Caustic syntax qualitatively to SQL, Tango, and Java. Measure differences in code complexity (measured in Cyclomatic or Halstead Complexity), size of executables, and lines of code for a variety of sample use cases (eg. implementing ZooKeeper or BookKeeper).