calmDB: A Distributed, Coordination-Free Key Value Store

QUINN VINLOVE, Lewis & Clark College, USA

Distributed, serverless computing is rapidly replacing traditional monolithic architectures, but most database systems rely on a few highly available machines. Following principals laid out by the CALM theroem and its proof, we designed a distributed database that is fast and easy to scale without complicated, difficult coordination protocols.

CCS Concepts: • **Theory of computation** → **Database theory**.

ACM Reference Format:
Quinn Vinlove. 2020. calmDB: A Distributed, Coordination-Free Key Value Store. In *ToC '20: ToC '20: Theroy of Computation Final Projects, December 16, 2020, Portland, OR.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Last year, half of all customers who used a popular cloud infrastructure monitoring tool had at least one active Amazon Web Services Lambda function running. Short-lived, easily scalable functions as a service, like Lambda, are becoming popular, yet because of their design, can't persist data. Wouldn't it be nice if storage could be replicated, scaled, and distributed with the click of a button, in the same way that Lambda or Docker containers can be elastically scales?

In this paper, we present calmDB, a unique key-key-value store that aims to make distributed databases simple to deploy, reliable, and fast. Using the CALM conjecture, we will show that some of the properties of our new database allow it to run quickly, and achieve consistency, without computationally expensive coordination.

2 INTRODUCING CALM

Joe Hellerstien, a professor of Computer Science at UC Berkley, first proposed the CALM conjecture in his keynote speech at the 2010 ACM PODS symposium (principals of database systems). CALM, or the coordination as logical monotonicity conjecture, was intended to draw a clear line between the kinds of problems that can be easily solved without coordination using a distributed system, and those that are impossible to solve with one. Coordination can result in serious slowdowns for distributed programs, so it's critical that we can prove that coordination isn't needed in order for our code to run very quickly and take advantage of the power parallelism can provide. Later, the CALM conjecture became a theorem, stating that a program has a coordination free distributed implementation iff it is monotonic [3]. A program P is logically monotonic if for any input set S and T, if $S \subseteq T$ then $\mathcal{P}(S) \subseteq \mathcal{P}(T)$.

3 PROVING CALM: A QUICK SKETCH

In order to prove that a program has a coordination free distributed implementation $\leftrightarrow$ it is monotonic, we need to prove that if a program has a coordination free distributed implementation, then it is monotonic, and if a program is monotonic, then it has a coordination free distributed implementation.

3.1 Relational Transducers

In a series of papers, Ameloot et al. sketched out a proof for CALM. In their proof, they use relational transducers, a common model in database theory. Each relational transducer is a networked database that can accept queries from a network, ingest them, and transmit them. The transducers themselves run a Prolog-based language for declarative networking, Datalog. Datalog was designed primarily for database queries, and is used to query several real-world databases like Apache's Hadoop. Monotonic Datalog, the subset of Datalog used in Ameloot et al.'s proof, is made monotonic with the exclusion of aggregate and negation functions, or in simpler terms, without the ability to reduce data (e.g. sum) or take the inverse of it [1].

Ameloot et al. rephrases the the CALM theorem into a different statement: a program has an eventually consistent, coordination-free execution strategy iff it is expressible in monotonic Datalog. The 'if' portion of the proof is simple, while the 'only if' portion of the proof is a little more difficult, because not all programs in monotonic Datalog have an eventually consistent, coordination-free execution strategy. In this paper, I'll use the first direction of the proof as a starting point, and more curious readers are welcome to explore their paper for a more in-depth approach.

3.2 Lemma 5

To start, Part 1 of Lemma 5 states that there is an inflationary transducer such that, on any network, splitting the transducer's input into a set of rows and a row I, after any fair run, a node has a local copy of I in it's memory, and an additional 'ready' flag that is set to true. This 'ready' flag doesn't become true until I is completely in the transducer's memory. An inflationary transducer can't delete any items, and a deletion query will return empty.

In proving this lemma, implement a multicast protocol. Every node v in the network sends out all the facts in its local input relations. Each fact is tagged with the id of the node that it comes from. Every node stores the facts it receives, and also forwards them through the network. For every fact that every node v receives, every node sends an acknowledge fact tagged with its own identifier. Every node keeps a record of which of its input facts it has received an acknowledgement for, and which node that acknowledgment came from. When a node has received an acknowledgment from every local input fact, it sends out a 'done' message with its id attached. When a node has received a 'done' from every node, it knows the query is complete. No deletions are needed.

3.3 Theorem 6

Next, part 1 of Theorem 6 states that every query can be computed by some abstract transducer in a distributed way. If the language L used to query is computationally complete, every partial computable query can be computed using distribution by an $\mathcal{L}$-transducer, which is a transducer whose queries can be expressed with L.

Proving Theorem 6, assume an abstract query Q. Run the transducer we described in Part 1 of Lemma 5 to obtain the entire input instance. Then, apply this input and output Q.

3.4 Proposition 11

Proposition 11, later in the paper, states that every network-topology independent, oblivious transducer is coordination free. Let be a network-topology independent, oblivious transducer, like the one we described above. Let Q be the query distributively computed by . On a one-node network, with a given input I, π reaches a state of inactivity and outputs $Q(I)$. With any other network, use any instance of I of the complete input, and also use a partition H that places I at every node. π is oblivious, so nodes can't tell if they're on a network with other nodes, unless they communicate. By only doing heartbeat transitions initially, every node will act the same as a one-node network, and will output $Q(I)$ as expected. It's important to note here that heartbeat transitions are simply an activity message, and are commonly implemented in situations where a system must be highly available, and where its operational state must be known.

3.5 Rest of Proof

So far, we've proved that one small part of one specific program that is eventually consistent and coordination free has a concrete definition, through proof by construction. Later, this definition can be written in Datalog The second half of the iff statement is proved to be monotone by Ameloot et al. First, before they try and use a set of corollaries to relate their work back to Datalog.

For the remainder of the proof, Ameloot et al. prove the theorem that every query that is distributively computed by a coordination-free transducer is monotone. Then, they prove two corollaries. Corollary 13 states that for any query Q, Q can be distributively computed by a transducer that is coordination free, Q can be distributively computed by a transducer that is oblivious, and Q is monotone. Corollary 14 states equivalency for three groups involving query Q: a computationally complete query language L that runs on some transducer that compute Q, Q can be distributively computed by a coordination free transducer, and a coordination free non recursive Datalog transducer that can also distributively compute Q. The last step of the third part is the most critical: it states that Q can be expressed in Datalog.

4 DATABASE DESIGN

At this point, once we've outlined a basic proof sketch of the CALM theorem. Now, we can go ahead and design a distributed database around its principals, and then guarantee that it doesn't need coordination.

calmDB only allows two operations: POST, which adds a single value to the database, and GET, which retrieves a value based on the value of one or more of its keys. As we proved above, adding a single item to a set is a monotonic operation, and requires no coordination. Each set is now larger, but set S is still contained as a subset of set T. Most SQL-like databases allow for an operation that updates an entry or particular set of entries. As discussed in the DynamoDB, updating under unknown network conditions is difficult to do consistently [2]. If database A receives an update that database B doesn't, then the user queries database B, then sends an update again, it's unknown which machine the update is intended for, and which record counts as the most 'recent.' Dynamo simply keeps track of what version of the record was last attributed to which machine with vector clocks, and returns all possible 'truths' for contested data fields. However, this is rather inelegant and can increase the size of an entry.

calmDB solves this problem in two ways. More obvious is the use of two keys for each value. By associating two identifiers with a piece of data, rather than one, less, a single piece of relevant data can be directly altered (more on this later) more specifically. Giving an analogy with Amazon's shopping cart example that is referred to in the Dynamo paper: we can now store the contents of all users carts as individual entries in the database rather than a JSON struct, which has to be constantly altered and updated to maintain the most current state.

The second approach leans on the features built into the CALM theorem by just eliminating UPDATE altogether with the use of tombstoning, a monotonic design pattern. Each data item in the database undergoes a transition from undefined, to defined as a particular value, to tombstoned, or no longer available. By preventing updates altogether, we can be sure that each node in this distributed database can be robust all on its own, with no conflicting updates to deal with in the first place. This also means that data values can't be deleted, only marked inaccessible. For each subsequent GET, tombstoned values are removed from the response, and cleaned from memory with a periodic cron job in order to free up space in the database.

5 FUTURE WORK

calmDB has potential to be fast, flexible, and distributed with little requirements. We intend to extend our implementation, which is single-node, to run and exist as one node in a large network.

REFERENCES

[1] Tom J. Ameloot, Frank Neven, and Jan Van den Bussche. 2011. Relational transducers for declarative networking. *Proceedings of the 30th symposium on Principles of database systems of data - PODS '11* (2011). https://doi.org/10.1145/1989284.1989321

[2] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *IN PROC. SOSP*. 205–220.

[3] Joseph M. Hellerstein and Peter Alvaro. 2019. Keeping CALM: When Distributed Consistency is Easy. arXiv:1901.01930 [cs.DC]

A ARTIFACTS

A.1 Source Code

A preliminary draft of calmDB, written in Golang, can be found at https://github.com/quin2/calmdb