Interactive Website for Visualizing CRDTs

Zane Reis, Department of Computer Science
Mentor: Dr. Lan Yang
Kellogg Honors College Capstone Project, RSCA 2023

Introduction

Conflict-free replicated data types are an important component of distributed computing. They allow independently running applications to update and maintain information without any form of coordination, all while ensuring strong eventual consistency of all replicas. While understanding them may sound daunting, CRDTs lend themselves to being easily understood through interacting with them. The hope of creating an interactive website that allows users to create, visualize, modify, and merge CRDTs is to lower the barrier of entry for understanding this important concept of distributed computing.

Features

The following actions are available for the user to perform:

- Initializing new instances of a CRDT.
- Performing operations to mutate the various CRDT replicas.
- Merging CRDT replicas together following the rules of that specific CRDT.
- Going backwards through the simulation's history to see how the CRDTs' states converge over time.

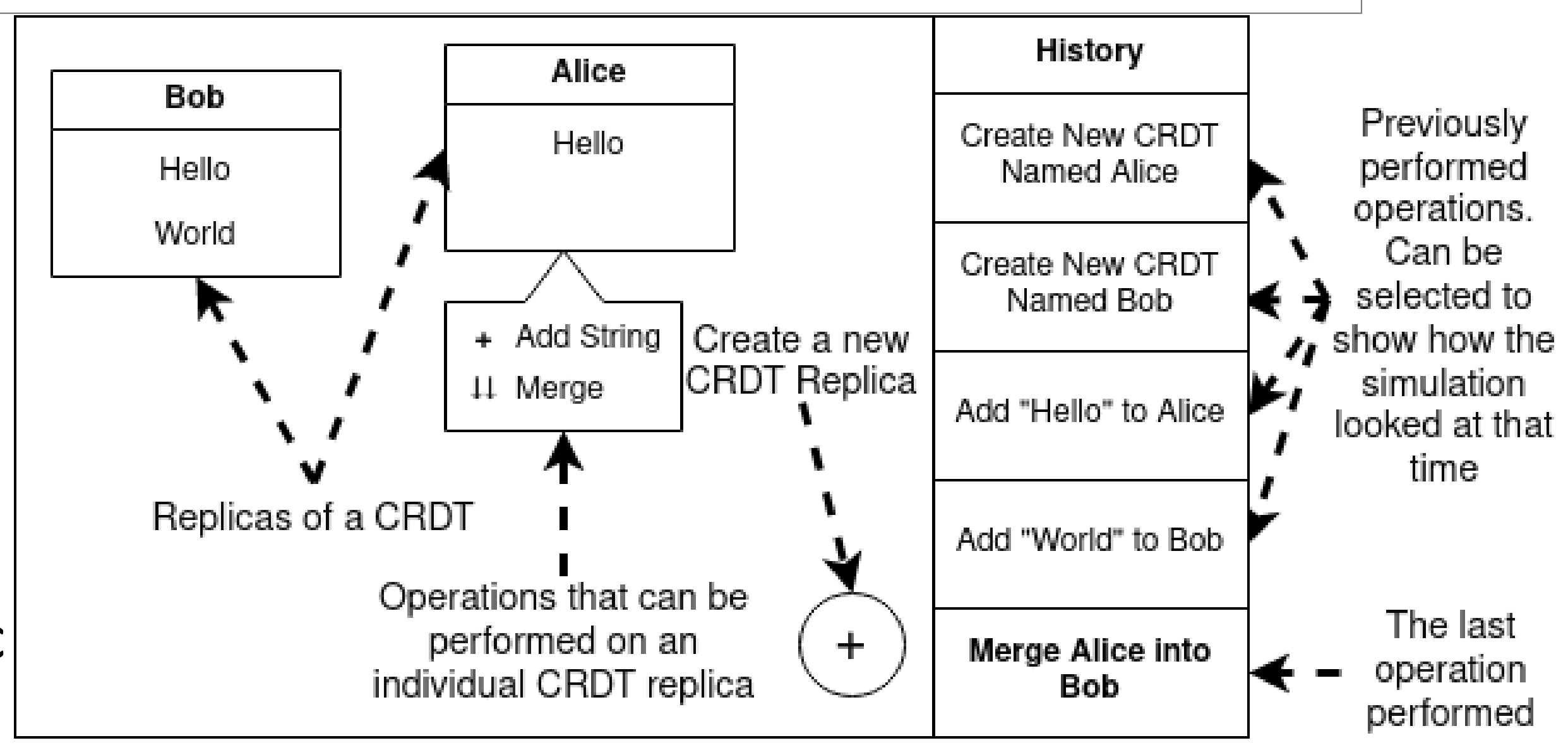

Bob
Hello
World
Alice
Hello
+ Add String
⇊ Merge
Create a new CRDT Replica
Replicas of a CRDT
Operations that can be performed on an individual CRDT replica
+
History
Create New CRDT Named Alice
Create New CRDT Named Bob
Add "Hello" to Alice
Add "World" to Bob
Merge Alice into Bob
Previously performed operations. Can be selected to show how the simulation looked at that time
The last operation performed

fig 1: Annotated mockup

Development

The main technology used to implement this project was Spring. Spring Boot, Spring Data JPA, and Hibernate are a very popular choice for implementing web services. In addition to these three components, Thymeleaf was used to generate the UI.

Web servers in general require some means of recording information between requests. This means of recording information is commonly referred to as persistence. For this application, not only does the current state of the simulation need to be persisted but, because of the requirement to maintain a history, all previous states need to be persisted as well. Additionally, while the operations that can be performed on CRDTs are similar, how they internally record their state is not. This lack of regularity further complicates the problem of persistence.

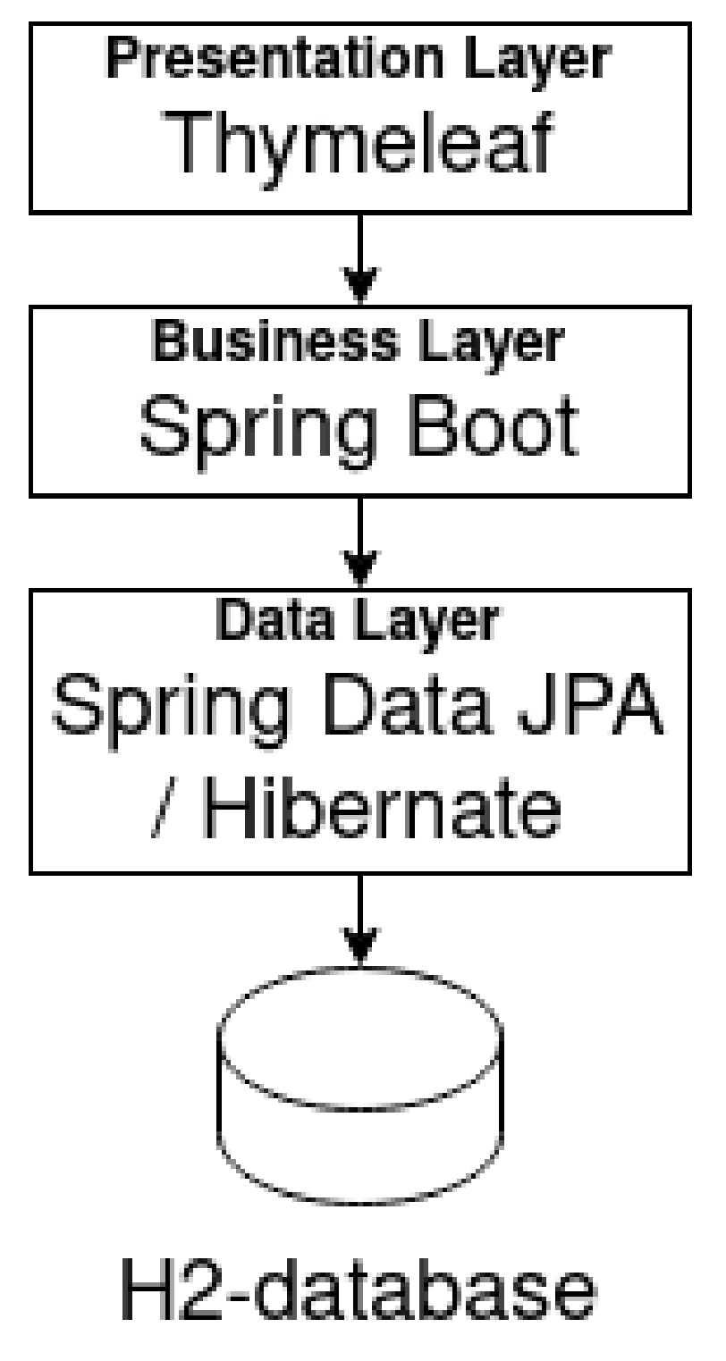
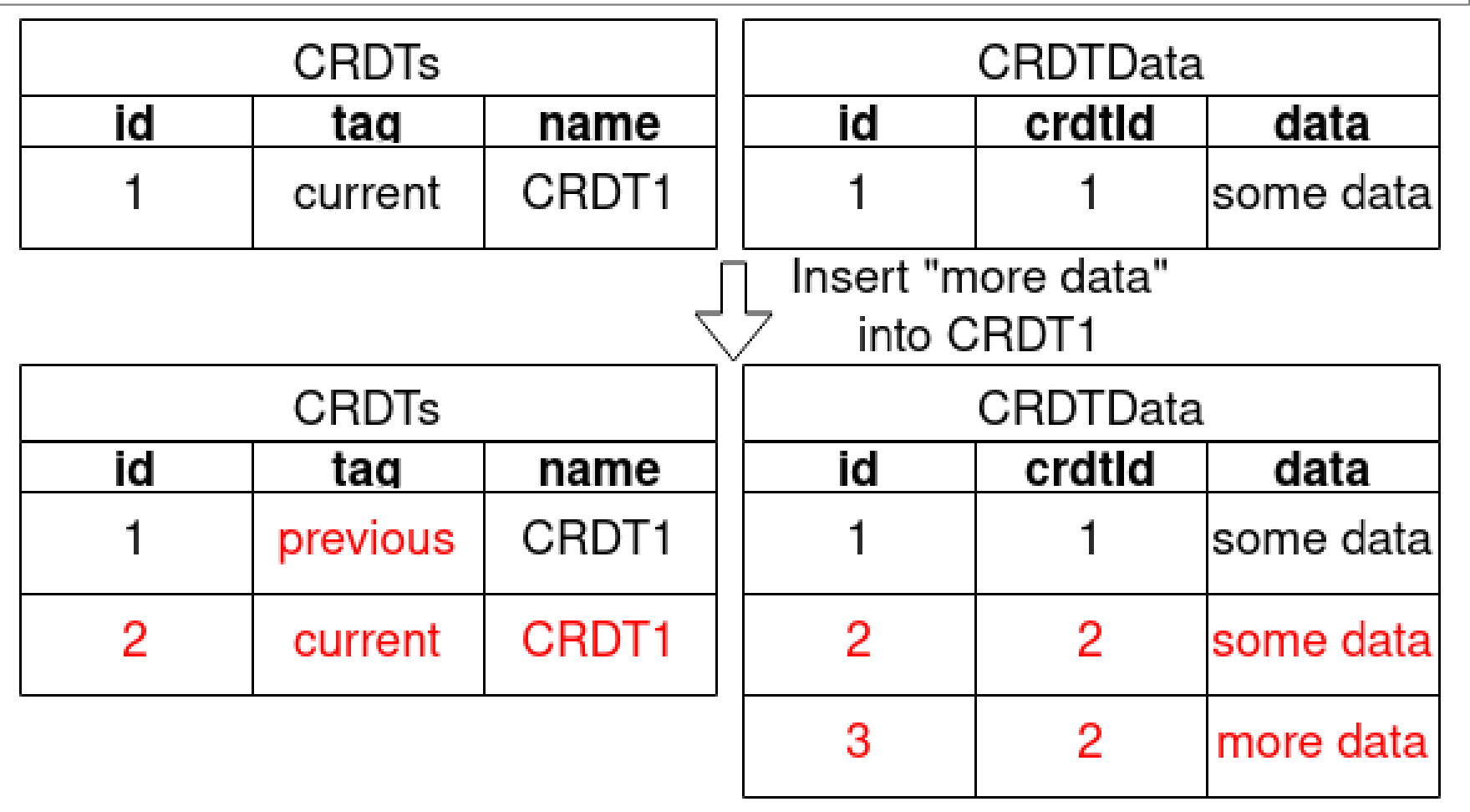


fig 2: The application stack

Techniques

The naive solution of this history problem is to record the simulation's history by copying the state of the simulation each time an operation is performed. The old copy is labeled with the point in history it is from, and the new copy becomes the new current state. Besides requiring a significant amount of row copying, an issue that worsens linearly as more data is inserted into the CRDT, the space complexity is $O(n^2)$. This also doesn't address the issue that CRDTs record their information differently depending on the type. The schema will need to be modified for each new type of CRDT introduced into the system.



CRDTs

id	tag	name
1	current	CRDT1

CRDTData

id	crdtId	data
1	1	some data

Insert "more data" into CRDT1

CRDTs

id	tag	name
1	previous	CRDT1
2	current	CRDT1

CRDTData

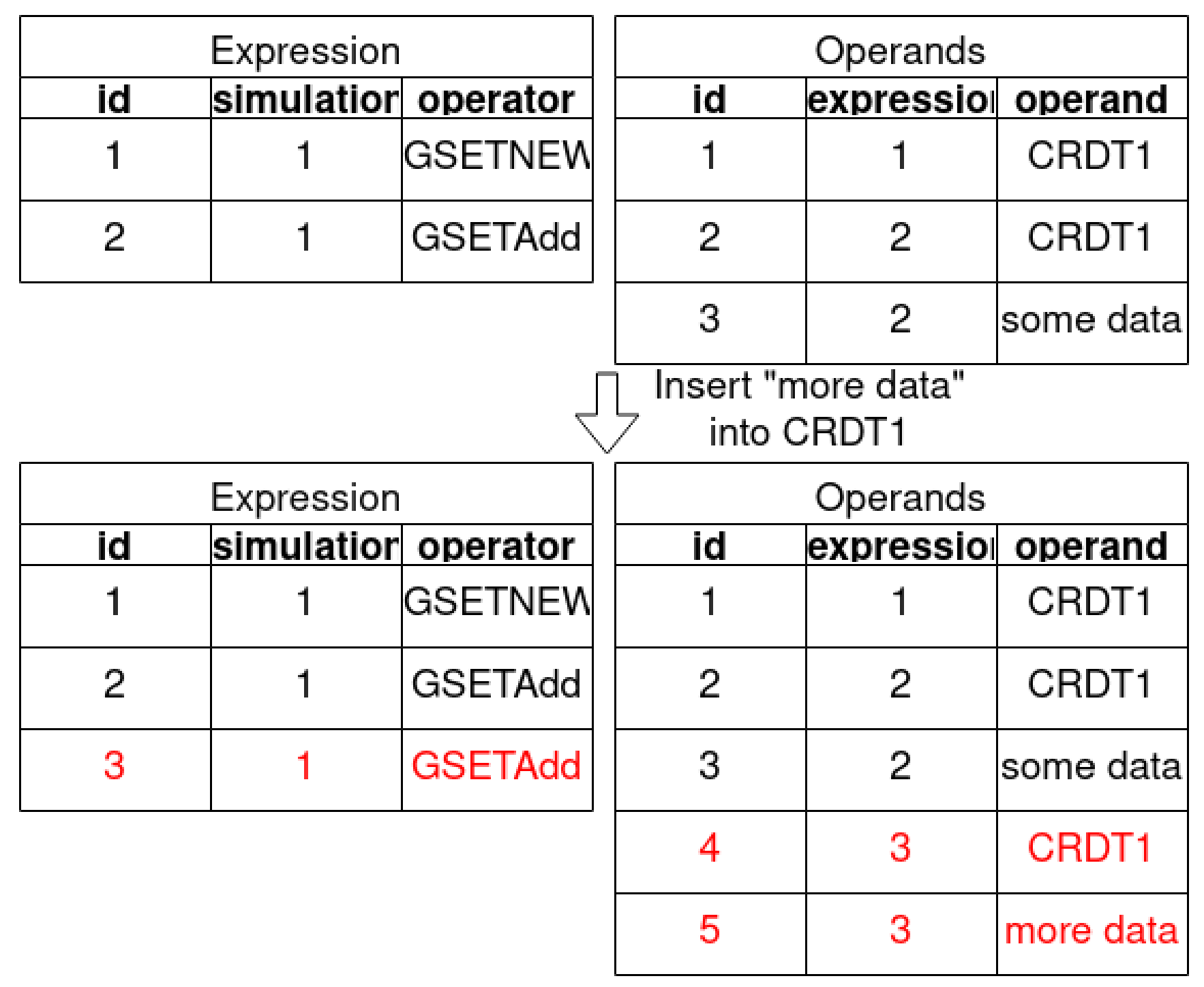
id	crdtId	data
1	1	some data
2	2	some data
3	2	more data

fig 3: "naïve" operation in database

This project did not implement the strategy outlined above. Instead, it opted to only store the operations performed on the simulation. The operations performed completely determine the simulation's current state. Evaluating each of these operations in order is all that is needed to calculate it. This solution is ideal performance-wise. The number of row insertions is constant per operation, the space complexity is $O(n)$, and the time complexity of evaluation is $O(n)$.

Interpreter Analogy

Evaluating operations one after another to calculate some result is similar to the role of an interpreter for an imperative programming language. Because of this very close analogy, interpreter and compiler design had significant impact on this project's implementation. To calculate what the simulation looks like at a specific point in history, the backend simply evaluates each operation in order up until that point in history. This is analogous to setting a breakpoint in a computer program before executing. This analogy also gave inspiration to representing each operation in the database as an operator with a set of operands. This format has the effect of being generic enough to encompass every type of simulation's CRDTs, simplifying persistence.



Expression

id	simulatior	operator
1	1	GSETNEW
2	1	GSETAdd

Operands

id	expressio	operand
1	1	CRDT1
2	2	CRDT1
3	2	some data

Insert "more data" into CRDT1

Expression

id	simulatior	operator
1	1	GSETNEW
2	1	GSETAdd
3	1	GSETAdd

Operands

id	expressio	operand
1	1	CRDT1
2	2	CRDT1
3	2	some data
4	3	CRDT1
5	3	more data

fig 4: as-implemented operation in database

Execution Safety

This project was able to borrow concepts and techniques usually reserved for ensuring correctness in programming languages. There is a simple linter which checks that statements can be executed without error before being inserted into the database. Additionally, prior to execution the backend checks the each of the operations are of the same type before executing, forming a rudimentary type checker. For example, if one operation is supposed to be performed on a GSET while the rest are for 2PSETs, the backend will not attempt to execute and will inform the user of the error. However, even this should not have a chance of occurring given the linter operates as intended.

Future Work

Many of the CRDTs used in Industry are composed of multiple smaller CRDTs. As it stands, each new CRDT needs to be implemented in the backend even if it is just composed of smaller CRDTs. The next step is to extend the type system and evaluation system to allow users to construct and interact with larger CRDTs.

Release

The source code is available at https://github.com/zanereis/CRDT